Ontology for OpenAPI REST Services Descriptions

Aikaterini Karavisileiou Nikolaos Mainas Euripides G.M. Petrakis
School of Electrical and Computer Engineering
Technical University of Crete (TUC)
Chania, Crete, Greece
akaravasileiou@isc.tuc.gr, {nmainas, petrakis}@intelligence.tuc.gr

Abstract—OpenAPI Specification (OAS) defines a description format for REST APIs. In order for a machine to understand the meaning of REST services, OpenAPI service descriptions must be unambiguous. In a previous work we analysed the reasons that cause ambiguities in OpenAPI and showed that, in order to eliminate ambiguities, OpenAPI properties must be semantically annotated and mapped to a semantic model. Leveraging latest results for hypermedia-based construction of Web APIs (i.e. Hydra), the present work forwards this approach and proposes a reference ontology for REST services along with a formal procedure for converting OpenAPI service descriptions to instances of this ontology.

Keywords—Web service, Ontology, OpenAPI, REST, Hydra

I. INTRODUCTION

Typically, Web services are described in plain text which the users have to browse and read, in order to determine whether a service meets their needs. However, text descriptions of Web services are not readable by machines and in some cases are inaccurate or vague. Web services need to be formally described in a way that is understandable by both, humans and machines. The last requirement would not only improve the accuracy of service descriptions but also, would allow for services to be discovered by other services and be orchestrated in composite services or applications.

The need for standardizing technologies for service publishing and discovery is of crucial importance for their adoption and market success. OpenAPI Specification (OAS) is a widely adopted standard for describing REST APIs which is supported by large industry users like Google, Microsoft, IBM, Oracle and many others. OAS v3.0 is the first major update of the specification since 2015. OAS v3.0 features a more elaborate (yet simple) structure and format than its predecessor OAS v2.0. The requirement for a single host server is relaxed (allowing a service to be installed on multiple servers). A request body is more flexible and allows consumption of different media types, such as JSON, XML, HTML, plain text and others. The descriptions for parameters have changed: FormData parameter was removed and, the cookie parameter type was introduced for documenting APIs that use cookies. The definition of Schema objects is enhanced with additional properties (e.g. anyOf, oneOf, not) allowing for the creation of more complex schemas with various data types. Regarding security definitions, OpenAPI v3.0 is enhanced with support for OpenID Connect. OpenAPI v3.0 now features a Components field where various reusable objects can be defined (i.e. responses, parameters, headers, links, callbacks, schemas and security schemes). Two new features are introduced 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 of OpenAPI v3.0 to incorporate HATEOAS functionality in the specification. Finally, 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.

OpenAPI format is based on JSON (or YAML) and a large set of properties are available for composing service descriptions. Despite its user friendliness, OpenAPI v3.0 service descriptions can be vague: the same property may appear with different names within the same OpenAPI document or, its meaning may not be defined at all. OpenAPI does not provide a mechanism for detecting or for dealing with ambiguities. In a recent (short) paper we analyzed the causes of ambiguity and we concluded that, in order to eliminate these ambiguities, OpenAPI properties must be semantically annotated and associated to entities of a semantic model (e.g. in www.schema.org shared vocabulary). Then, we showed that is plausible to represent OpenAPI descriptions using ontologies and we proposed a reference ontology for OpenAPI service descriptions. This is the first work in the literature that handles ambiguities in OpenAPI v3.0 descriptions.

The present work forwards this approach in certain ways: (a) presents the Beta (stable) version of the OpenAPI ontology and (b) proposes a formal methodology and an algorithm for converting OpenAPI v3.0 ser-
service descriptions to instances of the ontology. The new (Beta version) of the ontology takes full advantage of Hydra \(^2\) and SHACL\(^3\). Hydra is a promising technology towards understanding and constructing Web services that meet the HATEOAS requirement of REST architectural style. OpenAPI v3.0 ontology incorporates features of Hydra for modeling service operations along with models not foreseen in Hydra (e.g. REST security features, headers, constraints). Classes together with constraints on class properties are described using SHACL allowing service descriptions to be validated against the ontology. The OpenAPI v3.0 ontology combines the advantages of all and supports the efficient representation and dynamic discovery of hypermedia driven APIs.

Instantiating OpenAPI descriptions and services is a rather complicated process. The reasons can be any of all the following: the same property may appear many times in the same document (with the same or different meaning or, undefined meaning), with different scopes (local property declarations overshadow global ones) or, it can be nested inside other properties. The proposed algorithm handles all these issues. Instantiating REST services to the OpenAPI ontology would enable application of query languages (e.g. SPARQL) for service discovery, of reasoning tools (eg. Pellet) for detecting inconsistencies in service descriptions and, finally, it would enable service orchestration and synthesis of composite services. The last, is a long them and more ambitious goal of our approach.

The ontology should not be necessarily visible to the end-user or API developer. It acts in the background and its role is twofold: (a) informs the user for possible ambiguities and inconsistencies in OpenAPI descriptions and (b) works as a service repository for discovering similarities between Web APIs (e.g. in the spirit of Hamza et al. \(^3\) for discovering OpenAPI compliant REST APIs).

Related work on services description languages is presented in Sec. II. The OpenAPI v3.0 approach is discussed in Sec. II followed by issues for future work in Sec. V.

II. RELATED WORK

Syntactic description languages describe the requirements for establishing a connection with a service and the message formats to successfully communicate with it. WSDL\(^4\) for SOAP, WADI\(^5\) for REST, are popular mainly due to their simplicity and compatibility with common machine readable formats like XML and JSON. They are not specifically designed for hypermedia-driven APIs (such as REST) that call for the dynamic discovery of resources at run-time (referred to as HATEOAS). Their reputation has been overshadowed by OpenAPI Specification\(^6\), formerly known as Swagger.

Semantic approaches describe services by means of semantic models (i.e. ontologies) and are more capable of supporting automated service discovery and composition. WSMO\(^7\) defines a conceptual model and WSML language for the semantic description of Web services. OWL-S\(^8\) is an upper ontology for Web services but, similar to all other methods, do not support the dynamic discovery of resources at runtime.

Hydra \(^2\) simplifies the construction of hypermedia-driven APIs. The Hydra vocabulary defines concepts that a server can use to advertise valid state transitions to a client as a result of a sequence of service invocations. Server responses are provided as JSON-LD which a client can use at run-time to discover the available actions and resources, in order to formulate new HTTP requests and achieve a specific goal. Hydra is a promising technology towards understanding and constructing Web services that meet the HATEOAS requirement of REST architectural style.

Musyaffa et al. \(^4\) introduce annotations in Schema and Parameter objects for OpenAPI v2.0. They do not handle all causes of ambiguity nor do they handle OpenAPI v3.0 descriptions. The annotations appear within text properties and cannot be interpreted by a machine without preprocessing. Schwichtenberg, Gerth and Engels \(^5\) map OpenAPI v2.0 service descriptions to OWL-S ontology but do not deal with any causes of ambiguity in service descriptions, nor do they handle security properties. Their approach attempts to find a mapping of OAS v2.0 Schema objects to OWL-S using heuristics and name similarity matching techniques. The mapping is error prone and need to be adjusted manually. Most important, their choice of OWL-S model for representing REST services is controversial: OWL-S is good for SOAP services but not good for hypermedia-driven Web services like REST. Finally, they do not handle OAS v3.0 descriptions.

In a recent contribution, Hamza et al. \(^3\) propose an example-driven approach and a representation of REST APIs for discovering OpenAPI compliant REST Web APIs. This facilitates the API discovery favoring software reuse. The approach does not deal with ambiguity in OpenAPI and is complementary to our work, in the sense that, our proposed ontology representation is a far more powerful tool for supporting services discovery enhanced with reasoning for service synthesis and integration of existing APIs.

III. ANNOTATED OPENAPI V3.0 DESCRIPTIONS

OAS v3.0 service descriptions\(^1\) comprise of many parts (objects). Each object has a list of properties which can be objects as well (this way, objects can be linked.

\(^{1}\)https://www.w3.org/Submission/OWL-S/
\(^{2}\)https://www.openapis.org
\(^{3}\)https://www.w3.org/Submission/WSMO/
\(^{4}\)https://www.w3.org/Submission/wadl/
\(^{5}\)https://blog.readme.io/an-example-filled-guide-to-swagger-3-2/
\(^{6}\)https://raml.org
to each other). 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 on where the API’s servers are located. The *Servers* object specifies the base-path (the part of the URL that appears before the endpoint) of an API request. There are also variables that can be populated at run-time. Servers can be defined for different operations (i.e. a *Servers* object can be added as property in the *Path* object of an *Operation* object). If defined, these locally declared servers will override the base (i.e. global) servers.

The *Security* requirement object lists the security schemes that the service applies prior to executing an operation. Its name must correspond to a security scheme declared in *Security Schemes* under the *Components* object. The specification offers support for basic HTTP authentication, API keys, OAuth\(^2\), common flows or grants (i.e. ways of retrieving an access token) and OpenID Connect. The *Paths* object describes the available operations (i.e. HTTP methods) and contains the relative paths for the service endpoints (which are appended to a server URL in order to construct the full URL of an operation).

*Components* object holds a set of reusable objects which can be responses, parameters, schemas, request bodies and more. *Schemas* object define data structures that are used to describe the request and response messages. A *Schema* object can be a primitive (string, integer), an array or a model. A *Schema* object may also have properties of its own accord (i.e. *externalDocs*) and properties supporting model composition and polymorphism. The *discriminator* property in a schema informs about the existence of an alternative schema (based on a keyword such as *oneOf*, *anyOf*, *allOf*).

The *Responses* object describe the expected responses of an operation, by mapping them to a specific HTTP status code. A response object defines the message content, as well as HTTP headers that a response may contain. *Parameters* object describes parameters that operations may use. The specification, categorizes parameters into:

- **Path parameters** are used in cases where the parameter values are part of operation’s path.
- **Query parameters** are appended to the url when sending a request.
- **Header parameters** define additional custom headers that may be sent in a request.
- **Cookie parameters** are passed in the Cookie header.

In an OpenAPI document there are elements that share the same meaning that a human may understand but a machine cannot. In a recent (short) paper \(^1\) we introduced extra properties to annotate OpenAPI properties which are ambiguous. Table I summarizes the extension properties, their scope and their meaning. The *x-refersTo* extension property specifies the association between an OpenAPI element and a concept in a semantic model.

Listing \(\text{l}\) is an excerpt from the Swagger Petstore\(^3\) service description (the example is cited in all related literature). The original example has been modified to include additional properties showing polymorphism and extension properties. The full service is described in author’s thesis \(^6\). The *x-refersTo* extension property is used to semantically annotate a *Pet* model and its properties: it associates the model with *Pet* class in *Product ontology*.\(^14\) If the *Pet* model describes a specific group of pets (e.g. dogs), the *x-kindOf* extension property is used instead to denote that the model is a subclass of the referred semantic concept. The *x-mapsTo* is used to define *Schema* object elements within the same OpenAPI document that share the same semantics. In Listing \(\text{l}\) the *x-mapsTo* property is used in *Parameters* object to dictate that query parameter *name* refers to *Pet.name* in *Schemas* object. There is a semantic similarity between the *id* property of *Pet* model and the *petId* property in *http://www.schema.org/petId* which later appears in the document as *petId* (not shown in the example). Similarly, property *Pet* should refer to *http://www.productontology.org/doc/Pet* and *name* should refer to *http://purl.org/goodrelations/v1#name*.

Listing 1: OpenAPI v3.0 *Pet* model example.

```json
parameters:
  ...
  Query:
  name: name
  in: query
  required: true
  schema:
    type: string
    x-mapsTo: '#/components/schemas/Pet.name'
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Applies to</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x-refersTo</td>
<td>Schema &amp; Object</td>
<td>The concept in a semantic model that describes an OpenAPI element.</td>
</tr>
<tr>
<td>x-kindOf</td>
<td>Schema</td>
<td>A specialization between an OpenAPI element and a concept in a model.</td>
</tr>
<tr>
<td>x-mapsTo</td>
<td>Schema</td>
<td>An OpenAPI element which is semantically similar with another OpenAPI element.</td>
</tr>
<tr>
<td>x-collectionOn</td>
<td>Schema</td>
<td>A model describes a collection over a specific property.</td>
</tr>
<tr>
<td>x-onResource</td>
<td>Tag</td>
<td>The specific Tag refers to a resource described by a Schema.</td>
</tr>
<tr>
<td>x-operationType</td>
<td>Operation</td>
<td>Clarifies the type of operation.</td>
</tr>
</tbody>
</table>

---

\(1\)https://oauth.net/2/

\(2\)http://www.schema.org/petId

\(3\)http://petstore.swagger.io/

\(4\)http://www.productontology.org/
Pet: # A Pet model extended with properties
  type: object
  x-refersTo: http://www.productontology.org/doc/Pet
  properties:
    name:
      type: string
      x-refersTo: http://purl.org/goodrelations/v1#name
    photoUrls:
      type: string
      x-refersTo: http://schema.org/petPhoto
    id:
      type: integer
    ... 
  required:
    - name
    - photoUrls
  discriminator:
    propertyName: name

Dog: # A Dog model extending the Pet Model
  x-kindOf: http://www.productontology.org/doc/Pet
  allOf:
    - $ref: '#/components/schemas/Pet'
    - type: object
      properties:
        breed:
          type: string
        ... 
  required:
    - breed

Cat: # A Cat model extending the Pet Model
  ... 

The x-collectionOn extension property is used to indicate that a model in Schemas object is actually a collection. Typically, a collection (or a list) of resources in OpenAPI v3.0 is described using the array type. However, often the definition of collection, is encapsulated within an object type with additional properties. Then, x-collectionOn property is used to denote the data types of the objects of the collection.

The x-onResource extension property is used in Tag Objects to specify the resource that a tag refers to. In OpenAPI v3.0, tags are used to group operations either by resources or, by any other qualifier. If the tag is used to group operations by resources, a human may recognize that the referred resource is described by a Schema object in Schemas object but, a machine cannot. The x-onResource property is used to associate the tag with a Schema object that describes a specific resource.

Finally, x-operationType extension property is used to specify the type of an Operation object. A request is characterized by the HTTP method it applies. However, the semantics of the HTTP methods are too generic and may have a more specific meaning. For example, this property can be used to clarify that a get request can actually mean a search operation on Schema objects.

IV. OPENAPI V3.0 ONTOLOGY

The OpenAPI v3.0 ontology in Fig. 1 captures all information in a Semantic OpenAPI description. Properties of classes are mapped to classes as well. At the heart of the ontology is Hydra Core Vocabulary, enhanced with additional models in order to capture information about security, headers and constraints. Schemas, Operations, Resources and Properties are mapped to Hydra models.

Document class provides general information (Info class) about the service and specifies service paths, the entities and the security schemes that it supports. Path class represents (relative) service paths (pathName property). Operation class provides information for issuing HTTP requests. Request bodies are represented by Body class, while responses are declared in Response class specifying the status code and the data returned. The entire range of HTTP responses is represented. The MediaType class describes the format (e.g. JSON, XML) of a request or response body data. Class Operation refers to a security scheme in SecurityRequirement class.

Fig. 2 shows the security schemes supported by OpenAPI. Class Security has security schemes as sub-classes. Class OAuth2 has different flows (grants) as sub-classes. If the security scheme is of type OAuth2 or OpenID Connect, then scope names are defined as properties.

<table>
<thead>
<tr>
<th>TABLE II: Mapping Schema properties to SHACL.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema Object property</td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td>maximum</td>
</tr>
<tr>
<td>exclusiveMaximum</td>
</tr>
<tr>
<td>minimum</td>
</tr>
<tr>
<td>exclusiveMinimum</td>
</tr>
<tr>
<td>maxLength</td>
</tr>
<tr>
<td>minLength</td>
</tr>
<tr>
<td>pattern</td>
</tr>
<tr>
<td>maxItems</td>
</tr>
<tr>
<td>minItems</td>
</tr>
<tr>
<td>enum</td>
</tr>
<tr>
<td>allOf</td>
</tr>
<tr>
<td>oneOf</td>
</tr>
<tr>
<td>anyOf</td>
</tr>
<tr>
<td>not</td>
</tr>
<tr>
<td>default</td>
</tr>
</tbody>
</table>

SHACL objects are expressed as classes, object and data properties using SHACL. SHACL is an RDF vocabulary that can be used to define classes together with constraints on their properties. It provides built-in types of constraints (e.g. cardinality: minCount/maxCount) and allows expression of constraints on the type of properties and on the values they can take. Table III shows the mapping between Schema object and SHACL. Schema objects are mapped to Shape class. The NodeShape class represents operations related to a class (supportedOperation), which come from x-onResource extension property. Class PropertyShape represents the properties of a class, their
datatype and restrictions (e.g. a maximum value for a numeric property) and indicates whether the supported property is required or read-only.

Listing 2: Representation of Pet model.

```prolog
ex:PetShape
  a sh:NodeShape ;
  sh:targetClass po:Pet ;
  sh:property [
    sh:path purl:name ;
    sh:name "name" ;
    sh:datatype xsd:string ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] ;
  sh:property [
    sh:path schema:petId ;
    sh:name "id" ;
    sh:datatype xsd:integer ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] .
```

Listing 2 shows how the Pet model of Listing 1 is represented in the ontology. The model contains references to Schema.org vocabulary using x-refersTo. The SHACL class PetShape is defined according to Schema object definition of Pet with the addition of new data properties and constraints (e.g. each pet has exactly one name and an id). A Schema object defined using the combination of allOf property and a discriminator property, is represented as a subclass of the semantic model that is extended. A Schema object annotated with
the x-kindOf extension property is defined as a subclass of the referenced concept. Collections are represented by Collection class. Parameters are represented as separate classes for every parameter type. Header and Cookie parameters in HTTP requests and responses, become individuals of classes Header and Cookie, respectively. Class Parameter represents Path and Query parameters.

Details on the instantiation of OpenAPI service descriptions to the ontology can be found in the author’s thesis [6]. The method scans the OpenAPI document of a service and instantiates OpenAPI objects to classes of the ontology. The OpenAPI object (the root object of the OpenAPI document) is mapped to class Document. Property servers declares server information which applies across the description (global servers). This will be overwritten by server information defined in Path or Operations objects. Similarly, security property declares security schemes that are used across the API. Security schemes declared by an operation will also override global declaration of security schemes. Property tags contains the Tag objects for operations which are grouped. Through the x-onResource property Tag objects can associate operations with Schema objects. Path objects are converted to individuals of class Path. Property paths is replaced by supportedOperation which is described by Operation class. Then, individuals of class Operation are created. An operation individual captures all the information that Operation object contains. Property method defines the type of an operation (get, put, etc.). Parameter objects can be of type Path, Query Header or Cookie, which are instantiated to the corresponding classes (i.e. PathParameter, Query, Header Cookie). Property onPath associates an Operation individual with a Path individual.

A. Use case example

Service discovery on the Web might incorporate dedicated tools in the example of Hamza [3] or the Google Service Catalogue [4]. The catalogue provides an index to Google services with links to the description. The services in the catalogue are annotated and indexed using extra properties but the annotation scheme does not handle ambiguities. A more elaborate solution requires that services are first annotated and then instantiated to the ontology. Searching using SPARQL might also take advantage of reasoning for detecting inconsistencies and for discovering new inferred relations in service descriptions. In author’s thesis [6], we show how the PetStore and USPTO services [5] are annotated and instantiated to the OpenAI ontology. In this thesis, several SPARQL queries are discussed. The query of Listing 3 retrieves operations on pets. In this example, operations are grouped by their Tag value Pet which, in turn, is associated with Pet object using the x-on-resource property.

In addition, Cat and Dog are defined as specializations of Pet object using x-kindOf property. Due to inheritance (as of Listing 4), the query retrieves operations on pets and on all its sub-categories (i.e. cats and dogs) rather than operations on pets alone.

Listing 3: An example query on Petshop.

```sparql
PREFIX openapi: <http://www.aWebsite.xxx.xxx/ns/open-api>
PREFIX po: <http://www.productontology.org/doc/>

SELECT ?operationId WHERE {
?schemaind openapi:supportedOperation ?operationInd .
?operationInd openapi:operationId ?operationId .
}
```

V. CONCLUSIONS

The ontology is currently being extended to support LINKS and CALLBACKS properties. Instantiating the APIs of GURU [7] catalogue is an issue for future work. In a more ambitious scenario, we foresee a mechanism that is capable of understanding the intention of the service being created so that the service can be implemented as a synthesis of existing services.

REFERENCES


15https://www.googleapis.com/discovery/v1/apis/
16https://github.com/OAI/OpenAPI-Specification/tree/master/examples/v3.0