OBA: An Ontology-Based Framework for Creating REST APIs for Knowledge Graphs

Daniel Garijo(✉) and Maximiliano Osorio

Information Sciences Institute, University of Southern California, Los Angeles, USA
{dgarijo,mosorio}@isi.edu

Abstract. In recent years, Semantic Web technologies have been increasingly adopted by researchers, industry and public institutions to describe and link data on the Web, create web annotations and consume large knowledge graphs like Wikidata and DBpedia. However, there is still a knowledge gap between ontology engineers, who design, populate and create knowledge graphs; and web developers, who need to understand, access and query these knowledge graphs but are not familiar with ontologies, RDF or SPARQL. In this paper we describe the Ontology-Based APIs framework (OBA), our approach to automatically create REST APIs from ontologies while following RESTful API best practices. Given an ontology (or ontology network) OBA uses standard technologies familiar to web developers (OpenAPI Specification, JSON) and combines them with W3C standards (OWL, JSON-LD frames and SPARQL) to create maintainable APIs with documentation, units tests, automated validation of resources and clients (in Python, Javascript, etc.) for non Semantic Web experts to access the contents of a target knowledge graph. We showcase OBA with three examples that illustrate the capabilities of the framework for different ontologies.

Keywords: Ontology · API · REST · JSON · JSON-LD · Data accessibility · Knowledge graph · OpenAPI · OAS

Resource type: Software
License: Apache 2.0
DOI: https://doi.org/10.5281/zenodo.3686266
Repository: https://github.com/KnowledgeCaptureAndDiscovery/OBA/

1 Introduction

Knowledge graphs have become a popular technology for representing structured information on the Web. The Linked Open Data Cloud[1] contains more than 1200 linked knowledge graphs contributed by researchers and public institutions. Major companies like Google,[2] Microsoft,[3] or Amazon [16] use knowledge graphs

[1] https://lod-cloud.net/.
[2] https://developers.google.com/knowledge-graph.
[3] https://www.microsoft.com/en-us/research/project/microsoft-academic-graph/.

J. Z. Pan et al. (Eds.): ISWC 2020, LNCS 12507, pp. 48–64, 2020.
https://doi.org/10.1007/978-3-030-62466-8_4

to represent some of their information. Recently, crowdsourced knowledge graphs such as Wikidata [15] have surpassed Wikipedia in the number of contributions made by users.

In order to create and structure these knowledge graphs, ontology engineers develop vocabularies and ontologies defining the semantics of the classes, object properties and data properties represented in the data. These ontologies are then used in extraction-transform-load pipelines to populate knowledge graphs with data and make the result accessible on the Web to be queried by users (usually as an RDF dump or a SPARQL endpoint). However, consuming and contributing to knowledge graphs exposed in this manner is problematic for two main reasons. First, exploring and using the contents of a knowledge graph is a time consuming task, even for experienced ontology engineers (common problems include lack of usage examples that indicate how to retrieve resources, the ontologies used are not properly documented or without examples, the format in which the results are returned is hard to manipulate, etc.). Second, W3C standards such as SPARQL [12] and RDF [2] are still unknown to a major portion of the web developer community (used to JSON and REST APIs), making it difficult for them to use knowledge graphs even when documentation is available.

In this paper we address these problems by introducing OBA, an Ontology-Based API framework that given an ontology (or ontology network) as input, creates a JSON-based REST API server that is consistent with the classes and properties in the ontology; and can be configured to retrieve, edit, add or delete resources from a knowledge graph. OBA's contributions include:

- A method for automatically creating a **documented REST OpenAPI specification**[4] from an OWL ontology [10], together with the means to customize it as needed (e.g., filtering some of its classes). Using OBA, new changes made to an ontology can be automatically propagated to the corresponding API, making it easy to maintain.
- A framework to create a **server implementation** based on the API specification to handle requests automatically against a target knowledge graph. The implementation will validate posted resources to the API and will deliver the results in a JSON format as defined in the API specification.
- A method for **converting JSON-LD** returned by a SPARQL endpoint [6] into JSON according to the format defined in the API specification.
- A mechanism based on named graphs[5] for users to **contribute** to a knowledge graph through POST requests.
- Automatic generation of **tests** for API validation against a knowledge graph.

OBA uses standards widely used in Web development (JSON) for accepting requests and returning results, while using SPARQL and JSON-LD frames to query knowledge graphs and frame data in JSON-LD. We consider that OBA is a valuable resource for the community, as it helps bridging the gap between

[4] http://spec.openapis.org/oas/v3.0.3.
[5] https://www.w3.org/2004/03/trix/.

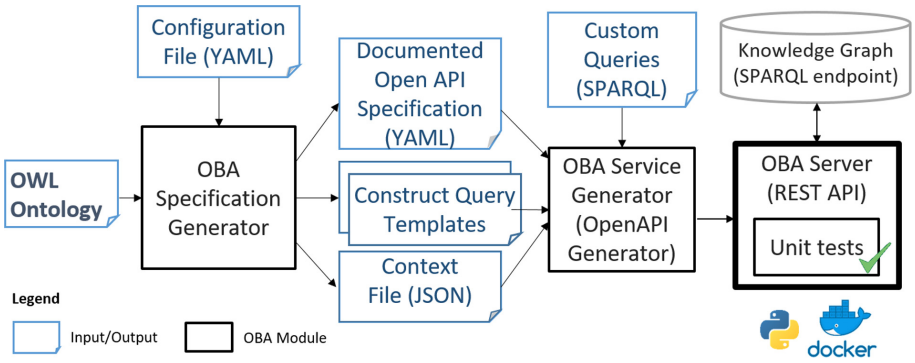

Fig. 1. Overview of the OBA Framework.

ontology engineers who design and populate knowledge graphs and application and service developers who can benefit from them.

The rest of the paper is structured as follows: Sect. 2 describes the architecture and rationale of the OBA framework, while Sect. 3 shows the different features of OBA through three different examples and a performance analysis of the tool. Section 4 discusses adoption, impact and current limitations of the tool, Sect. 5 compares OBA to similar efforts from the Semantic Web community to help developers access knowledge graphs, and Sect. 6 concludes the paper.

2 The Ontology-Based APIs Framework (OBA)

OBA is a framework designed to help ontology engineers create RESTful APIs from ontologies. Given an OWL ontology or ontology network and a knowledge graph (accessible through a SPARQL endpoint), OBA automatically generates a documented standard API specification and creates a REST API server that can validate requests from users, test all API calls and deliver JSON objects following the structure described in the ontology.

Figure 1 shows an overview of the workflow followed by the OBA framework, depicting the target input ontology on the left and the resultant REST API on the right. OBA consists of two main modules: the *OBA Specification Generator*, which creates an API specification template from an input ontology; and the *OBA Service Generator*, which produces a server with a REST API for a target SPARQL endpoint. In this section we describe the different features of OBA for each module, along with the main design decisions and assumptions adopted for configuring the server.

2.1 OBA Specification Generator

One of the drivers for the development of OBA was the need to use standards and file formats commonly used by web developers (who may not necessarily be familiar with Semantic Web technologies). Hence, we decided to use the OpenAPI specification[6] for representing REST APIs and JSON as the main interchange file format.

There are three reasons why we chose the OpenAPI specification (OAS): First, it "defines a standard, programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic".[7] Second, OAS is an open source initiative backed up by industry and widely used by the developer community, with more than 17.000 stars in GitHub and over 6.000 forks. Finally, by adopting OAS we gain access to a wide range of tools[8] that can be leveraged and extended (e.g., for generating a server) and are available in multiple programming languages.

2.1.1 Generating an OAS from OWL

OAS describes how to define *operations* (GET, POST, PUT, DELETE) and *paths* (i.e., the different API calls) to be supported by a REST API; together with the information about the *schemas* that define the structure of the objects to be returned by each call. OAS also describes how to provide examples, documentation and customization through parameters for each of the paths declared in an API.

Typically, an OAS would have two paths for each GET operation; and one for POST, PUT and DELETE operations. For instance, let us consider a simple REST API for registering and returning *regions* around the world. An OAS would have the paths '`/regions`' (for returning all available regions) and '`/regions/{id}`' (for returning the information about a region in particular) for the GET operation; the '`/regions`' path for POST;[9] and the '`/regions/{id}`' path for PUT and DELETE operations.

In OAS, the *schema* to be followed by an object in an operation is described through its properties. For example, we can define a *Region* as a simple object with a *label*, a *type* and a *partOfRegion* property which indicates that a region is part of another region. The associated schema would look as follows in OAS:

[6] https://github.com/OAI/OpenAPI-Specification.
[7] http://spec.openapis.org/oas/v3.0.3.
[8] https://github.com/OAI/OpenAPI-Specification/blob/master/IMPLEMENTATIONS.md.
[9] Alternatively, '`/regions/id`' may be used to allow developers to post their own ids.

```
Region:
 description: A region refers to an extensive, continuous
       part of a surface or body.
 properties:
   id:
     nullable: false
     type: string
   partOfRegion:
     description: Region where another region is included in.
     items:
       $ref: '#/components/schemas/Region'
     nullable: true
     type: array
   label:
     description: Human readable description of the resource
     items:
       type: string
     nullable: true
     type: array
   type:
     description: type of the resource
     items:
       type: string
     nullable: true
     type: array
 type: object
```

Note that the *partOfRegion* property will return objects that follow the *Region* schema (as identified by '`#/components/schemas/Region`'). The *nullable* parameter indicates that the target property is optional.

The main OAS structure maps naturally to the way classes and properties are specified in ontologies and vocabularies. Therefore, in OBA we support RDFs class expressivity by mapping each ontology class to a different path in the API specification;[10] and adding each object property and data type property in the target ontology to the corresponding schema by looking into its domain and range. Complex class restrictions consisting on multiple unions and intersections are currently not supported. Documentation for each path and property is included in the `description` field of the OAS by looking at the available ontology definitions (e.g., `rdfs:comment` annotations on classes and properties). Unions in property domains are handled by copying the property into the respective class schemas (e.g., if the domain of a property is '`Person or Cat`', the property will be added in the `Person` and `Cat` schemas); and properties declared in superclasses are propagated to their child class schemas. Properties with no domain or range are by default excluded from the API, although this behavior can be configured; and property chains are currently not supported. By default,

[10] We follow the best practices for RESTful API design: paths are in non-capital letters and always in plural (e.g., `/regions`, `/persons`, etc.).

all properties are *nullable* (optional). The full mapping between OAS and OWL supported by OBA is available online.[11]

Finally, we also defined two filtering features in OBA when generating the OAS to help interacting with the API. First, we allow specifying a subset of classes of interest to include in an API, since ontologies may contain more classes than the ones we may be interested in. Second, by default OBA will define a parameter on each GET path to retrieve entities of a class based on their label.

As a result of executing the OBA specification generator, we create an OAS in YAML format[12] that can be inspected by ontology engineers manually or using an online editor.[13] This specification can be implemented with the OBA server (Sect. 2.2) or by other means (e.g., by implementing the API by hand).

2.1.2 Generating SPARQL Query Templates and JSON-LD Context

The OBA Specification Generator also creates a series of templates with the queries to be supported by each API path. These queries will be used by the server for automatically handling the API calls. For example, the following query is used to return all the information of a resource by its id (`?_resource_iri`):

```
#+ summary: Return resource information by its resource_iri
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
CONSTRUCT {
    ?_resource_iri ?predicate ?obj .
    ?obj a ?type .
    ?obj rdfs:label ?label
}
WHERE {
    ?_resource_iri ?predicate ?obj
    OPTIONAL {
        ?obj  a ?type
        OPTIONAL {
            ?obj rdfs:label ?label
        }
    }
}
```

The individual id `?_resource_iri` acts as a placeholder which is replaced with the URI associated with the target path (we reuse GRLC [11] to define parameters in a query[14]). Returned objects will have one level of depth within the graph (i.e., all the outgoing properties of a resource), in order to avoid returning very complex objects. This is useful in large knowledge graphs such as DBpedia [1], where returning all the sub-resources included within an instance may be too costly. However, this default behavior can be customized by editing

[11] https://oba.readthedocs.io/en/latest/mapping/.
[12] https://yaml.org/.
[13] https://editor.swagger.io/.
[14] https://github.com/KnowledgeCaptureAndDiscovery/OBA_sparql/.

the proposed resource templates or by adding a custom query (further explained in the next section).

Together with the query templates, OBA will generate a JSON-LD context file from the ontology, which will be used by the server to translate the obtained results back into JSON. We have adapted owl2jsonld [13] for this purpose.

2.2 OBA Service Generator

Once the OAS has been generated, OBA creates a script to set up a functional server with the API. We use OpenAPI generator,[15] one of the multiple server implementations for OAS made available by the community; to generate a server with our API as a Docker image.[16] Currently, we support the Python implementation, but the architecture is flexible enough to change the server implementation in case of need. OBA also includes a mechanism for enabling pagination, which allows limiting the number of resources returned by the server.

OBA handles automatically several aspects that need to be taken into account when setting up a server, including how to validate and insert complex resources in the knowledge graph, how to handle authentication; how to generate unit tests and how to ease the access to the server by making clients for developers. We briefly describe these aspects below.

2.2.1 Converting SPARQL Results into JSON

We designed OBA to generate results in JSON, one of the most popular interchange formats used in web development. Figure 2 shows a sequence diagram with the steps we follow to produce the target JSON in GET and POST requests. For example, for the GET request, we first create a SPARQL CONSTRUCT query to retrieve the result from a target knowledge graph. The query is created automatically using the templates generated by OBA, parametrizing them with information about the requested path. For example, for a GET request to `/regions/id`, the `id` will replace `?_resource_iri` in the template query as described in Sect. 2.1.2.

As shown in Fig. 2, the construct query returns a JSON-LD file from the SPARQL endpoint. We frame the results to make sure they follow the structure defined by the API and then we transform the resultant JSON-LD to JSON. In order to transform JSON-LD to JSON and viceversa, we keep a mapping file with the API path to ontology class URI correspondence, which is automatically generated from the ontology. The URI structure followed by the instances is stored in a separate configuration file.

2.2.2 Resource Validation and Insertion

OBA uses the specification generated from an input ontology to create a server with the target API. By default, the server is prepared to handle GET, POST,

[15] https://github.com/OpenAPITools/openapi-generator.
[16] https://oba.readthedocs.io/en/latest/server/.

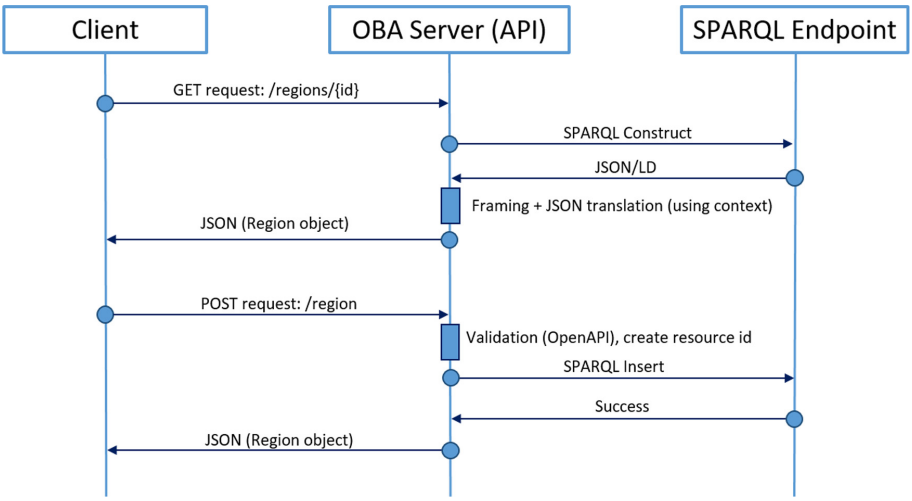

Fig. 2. Sample GET and POST request through the OBA server.

PUT and DELETE requests, which are addressed with CONSTRUCT, INSERT, UPDATE and DELETE SPARQL queries respectively. However, POST, PUT and DELETE requests need to be managed carefully, as they modify the contents of the target knowledge graph.

For POST and PUT, one of the main issues to address is handling *complex objects*, i.e., objects that contain one or multiple references to other objects that do not exist in the knowledge graph yet. Following our above example with regions, what would happen if we received a POST request with a new region where *partOfRegion* points to other regions that do not exist yet in our knowledge graph? For example, let us consider that a developer wants to register a new region `Marina del Rey` that is part of `Los Angeles`, and none of them exist in the knowledge graph. One way would be requiring the developer to issue a new request to register each parent region before registering the child one (e.g., a POST request first to register `Los Angeles` region and then another POST request for `Marina del Rey`); but this makes it cumbersome for developers to use the API. Instead, OBA will recursively validate, insert and generate ids for all subresources that do not have an id already. Hence, in the previous example OBA would register `Los Angeles` first, and then `Marina del Rey`. If a subresource already has an id, it will not be registered as a new resource. When a new resource is created, the server assigns its id with a uuid, and returns it as part of the JSON result.

This recursive behavior is not desirable for DELETE requests, as we could potentially remove a resource referenced by other resources in the knowledge graph. Therefore, OBA currently deletes only the resource identified by its id in the request.

Finally, OBA defines a simple mechanism for different users to contribute and retrieve information from a knowledge graph. By default, users are assigned a *named graph* in the target knowledge graph. Each named graph allows users submitting contributions and updates independently of each other (or collaboratively, if they share their credentials). User authentication is supported by default through Firebase,[17] which leverages standards such as OAUth2.0[18] for easy integration. However, we designed authentication to be extensible to other authentication methods, if desired. OBA may also be configured so users can retrieve all information from all available graphs; or just their own graph. User management (i.e., registering new users) is out of the scope of our application.

2.2.3 Support for Custom Queries

OBA defines common template paths from an input ontology, but knowledge engineers may require exposing more complex queries to web developers. For example, knowledge engineers may want to expose advanced filtering (e.g., return regions that start with "Eu"), have input parameters or complex path patterns (e.g., return only regions that are part of another region). These paths are impossible to predict in advance, as they depend on heterogeneous use cases and requirements. Therefore, in OBA we added a module to allow supporting *custom queries* and let knowledge engineers expand or customize any of the queries OBA supports by default.

To add a custom query, users need to follow two main steps. First, create a CONSTRUCT query in a file; and second, edit the OAS with the path where the query needs to be supported. OBA reuses GRLC's query module [11] to support this feature, as GRLC is an already well established application for creating APIs from SPARQL queries. An example illustrating how to add custom queries to an OAS in OBA can be found online.[19]

2.2.4 Generating Unit Tests

OBA automatically generates unit tests for all the paths specified in the generated OAS (using Flask-Testing,[20] a unit test toolkit for Flask servers[21] in Python). Units tests are useful to check if the data in a knowledge graph is consistent with the classes and properties used in the ontology, and to identify unused classes. By default, OBA supports unit tests for GET requests only, since POST, PUT and DELETE resources may need additional insight of the contents stored in the target knowledge graph. However, this is a good starting point to test the different calls to be supported by the API and detect any inconsistencies. Knowledge engineers may extend the test files with additional tests required

[17] https://firebase.google.com/docs/auth/.
[18] https://oauth.net/2/.
[19] https://oba.readthedocs.io/en/latest/adding_custom_queries/.
[20] https://pythonhosted.org/Flask-Testing/.
[21] https://flask.palletsprojects.com/en/1.1.x/.

by their use cases. Unit tests are generated as part of the server, and may be invoked before starting up the API for public consumption.[22]

2.2.5 Generating Clients for API Exploitation

The OpenAPI community has developed tools to generate clients to support an OAS in different languages. We use the OpenAPI generator in OBA to create clients (software packages) to ease API management calls for developers. For example, below is an example of a code snippet using the Python client for one of the APIs we generated with OBA[23] and available through `pip`. The client retrieves the information of a region (with id '*Texas*') and returns the result as a JSON object in a Python dictionary without the need of issuing GET requests or writing SPARQL:

```
import modelcatalog
# modelcatalog is the Python package with our API
api_instance = modelcatalog.ModelApi()
region_id = "Texas"
try:
    # Get a Region by its id. The result is a JSON object
    region = api_instance.regions_id_get(region_id)
    print(region)
except ApiException as e:
    print("Exception when calling ModelApi->regions_id_get: %s\n" % e)
```

3 Usage Examples and Performance Analysis

In this section we demonstrate the full capabilities of OBA in an incremental manner through three different examples of increasing complexity (Sect. 3.1); and a performance analysis to measure the overhead added by OBA (Sect. 3.2). All the resources described in this section are accessible online.[24]

3.1 Illustrating OBA's Features Through Examples

Drafting an API for an Ontology Network: The simplest way in which OBA can be used is by generating a draft OAS from a given ontology (without generating a server). We have tested OBA with ten different ontologies from different domains to generate draft specifications, and we have found this feature very useful in our work. Drafting the API allows knowledge engineers discuss potential errors for misinterpretation, as well as easily detect errors on domains and ranges of properties.

[22] https://oba.readthedocs.io/en/latest/test/.
[23] https://model-catalog-python-api-client.readthedocs.io/en/latest/.
[24] https://oba.readthedocs.io/en/latest/examples/.

Generating a GET API for a Large Ontology: DBpedia [1] is a popular knowledge graph with millions of instances over a wide range of categories. The DBpedia ontology[25] contains over 680 classes and 2700 properties; and creating an API manually to support them becomes a time consuming task. We demonstrated OBA by creating two different APIs for DBpedia. The first API contains all the paths associated with the classes in the ontology, showing how OBA can be used by default to generate an API even when the ontology has a considerable size. Since the resultant API is too big to browse manually, we created a Python client[26] and a notebook[27] demonstrating its use. The second API has just a selected group of classes by using a filter, as in some cases not all the classes may need to be supported in the desired API. OBA does a transitive closure on the elements that are needed as part of the API. For example, if the filter only contains "Band", and it has a relationship with "Country" (e.g., *origin*), then by default OBA will also import the schema for "Country" into the API specification to be validated accordingly. In the DBpedia example, selecting just 2 classes (`dbpedia:Genre` and `dbpedia:Band`) led to the inclusion of more than 90 paths in the final specification.

Generating a Full Create, Delete, Update, Delete (CRUD) API: OKG-Soft [5] is an open knowledge graph with scientific software metadata, developed to ease the understanding and execution of complex environmental models (e.g., in hydrology, agriculture or climate sciences). A key requirement of OKG-Soft was for users to be able to contribute with their own metadata in collaborative manner, and hence we used the full capabilities of OBA to support adding, editing and deleting individual resources. OKG-Soft uses two ontologies to structure the knowledge graph, which have evolved over time with new requirements. We used OBA to maintain an API release after each ontology version, generating an OAS, updating it with any required custom queries and generating a server with unit tests, which we executed before deploying the API in production. Having unit tests helped detecting and fixing inconsistencies in the RDF, and improved the overall quality of the knowledge graph. Authenticated users may use the API for POST, PUT and DELETE resources;[28] and we use the contents of the knowledge graph for scientific software exploration, setup and execution in different environments. An application for browsing the contents of the knowledge graph is available online.[29]

The three examples described in this section demonstrate the different features of OBA for different ontologies: the ability to draft API specifications, the capabilities of the tool to be used for large ontologies and to filter classes when

[25] https://wiki.dbpedia.org/services-resources/ontology.
[26] https://github.com/sirspock/dbpedia_api_client.
[27] https://github.com/sirspock/dbpedia_example/.
[28] https://model-catalog-python-api-client.readthedocs.io/en/latest/endpoints/.
[29] https://models.mint.isi.edu.

required; and the support for GET, POST, PUT and DELETE operations while following the best practices for RESTful design.

3.2 Performance Analyses

We carried out two main analyses to assess 1) the overhead added by OBA when framing results to JSON and 2) the performance of the API for answering multiple requests per second. Both tests have been performed in two separate machines (one with the API, another one with the SPARQL endpoint) with the same conditions: 8 GB of RAM and 2 CPUs. The analyses retrieve a series of results from a SPARQL endpoint (Fuseki server) by doing SPARQL queries and comparing them against an equivalent request through an OBA-generated API (GET queries, without reverse proxy caching). All requests retrieve individuals of various classes of a knowledge graph (e.g., GET all datasets, get all persons) and not individual resources. The corresponding SPARQL queries use CONSTRUCTs.

Table 1 shows that OBA (without enabling reverse proxy caching) adds an overhead below 150 ms for the majority of the queries with respect to a SPARQL endpoint (below 50ms); and between 150 and 200 ms for 8% of the queries. Overall, this overhead is barely noticeable when using the API in an application.

Table 2 shows the performance of the OBA server when receiving 5, 10 and 60 requests per second. When enabling reverse proxy caching (our recommended option), the API can handle 60 queries/second with a delay of less than 200 ms. Without cache, performance degrades when receiving more than 10 requests per second. This may be improved with an advanced configuration of the Python server used in our implementation.

Table 1. Time delay added by OBA when transforming RDF results into JSON

Time	No. requests		% of requests	
	Endpoint	OBA API	Endpoint	OBA API
[0s, 50ms]	59	0	98.33%	0.00%
[50ms, 100ms]	1	0	1.67%	0.00%
[100ms, 150ms]	0	0	0.00%	0.00%
[150ms, 200ms]	0	55	0.00%	91.67%
[200ms, 250 ms]	0	5	0.00%	8.33%
[250ms, 350 ms]	0	0	0.00%	0.00%
>350ms	0	0	0.00%	0.00%

Table 2. API performance for different number of requests and reverse proxy caching

Time	5 requests/second		10 requests/second		60 requests/second	
	Cache	No cache	Cache	No cache	Cache	No cache
[0s, 100ms]	100%	0.00%	99.83%	0%	99.89%	0%
[100ms, 200ms]	0%	88.67%	0%	0.28%	0.11%	0.24%
[200ms, 300ms]	0%	11.33%	0.17%	1.11%	0%	0%
[300ms, 1s]	0%	0%	0%	3.64%	0%	1.2%
[1s, 5s]	0%	0%	0%	95.00%	0%	6.10%
>5s	0%	0%	0%	0%	0%	92.44%

4 Adoption, Impact and Limitations

We developed OBA to help developers (not familiar with SPARQL) accessing the contents of knowledge graphs structured by ontologies. With OBA, non-expert web developers may use clients in the languages they are more familiar with (e.g., Python, JavaScript, etc.); generated with the OBA Service Generator. Web developers with more knowledge on using APIs may use the API created with the OBA server, while knowledge engineers may choose to query the SPARQL endpoint directly.

OBA builds on the work started by tools like Basil [3] and GRLC [11] - pioneers in exposing SPARQL queries as APIs- to help involve knowledge engineers in the process of data retrieval using the ontologies they designed. In our experience, generating a draft API from an ontology has helped our developer collaborators understand how to consume the information of our knowledge graphs, while helping the ontology engineers in our team detect potential problems in the ontology design.

In fact, similar issues have been raised in the Semantic Web community for some time. For example, the lack of guidance when exploring existing SPARQL endpoints[30] has led to the development of tools such as [9] to help finding patterns in SPARQL endpoints in order to explore them. The Semantic Web community has also acknowledged the difficulties experienced by developers to adopt RDF,[31] which have resulted in ongoing efforts to improve materials and introductory tutorials.[32]

OBA helps addressing these problems by exploiting Semantic Web technologies while exposing the information to developers following the REST standards they are familiar with. OBA allows prototyping APIs from ontologies, helps maintaining APIs (having an API per version of the ontology), helps validating API paths, assists in the creation of unit tests and documents all of the API

[30] https://lists.w3.org/Archives/Public/semantic-web/2015Jan/0087.html.
[31] https://lists.w3.org/Archives/Public/semantic-web/2018Nov/0036.html.
[32] https://github.com/dbooth-boston/EasierRDF.

schemas automatically. In addition, the tool is thoroughly documented, with usage tutorials and examples available online.[33]

We end this section by discussing assumptions and limitations in OBA. For instance, OBA assumes that the target endpoint is modeled according to the ontology used to create the API; and changes in the ontology version will lead to a new version of the API (hence keeping track of which version supports which operations). OBA also assumes that two classes in an ontology network don't have the same local name, as each class is assigned a unique path. As per current limitations, OBA simplifies some restrictions in the ontology, such as complex axioms in property domains and ranges (e.g., having unions and intersections at the same time as a property range), to help creating the OAS. In addition, large ontologies may result in extensive APIs, which will work appropriately handling requests, but may be slow to render in a browser (e.g., to see documentation of a path). Finally, by default OBA does not handle reification or blank nodes, although they can be partially supported by creating custom queries.

OBA is proposed as a new resource, and therefore we don't have usage metrics from the community so far.

5 Related Work

The Semantic Web community has developed different approaches for helping developers access and manipulate the contents of knowledge graphs. For instance, the W3C Linked Platform [8] proposes a framework for handling HTTP requests over RDF data; and has multiple implementations such as Apache Marmotta[34] or Trellis.[35] Similarly, the Linked Data Templates specification[36] defines a protocol for read/write Linked Data over HTTP. The difference between these efforts and our approach is that OBA creates an OAS from ontology terms that provides an overview of the contents in a knowledge graph; and also simplifies validating resources with a documented OAS that is popular among developers.

Other efforts like Basil [3], GRLC [11], r4r[37] and RAMOSE[38] create REST APIs from SPARQL queries in order to ease access to knowledge graphs. However, in these efforts there is no validation of posted data according to a schema or ontology; knowledge engineers have to manually define the queries that need to be supported; and additional work is required to transform a result into an easy to use JSON representation. In OBA, posted resources are validated against the OpenAPI schemas, a first version of the API is created automatically from the ontology, and all the results are returned in JSON according to the generated OAS.

[33] https://oba.readthedocs.io/en/latest/quickstart/.
[34] http://marmotta.apache.org/.
[35] https://www.trellisldp.org/about.html.
[36] https://atomgraph.github.io/Linked-Data-Templates/.
[37] https://github.com/oeg-upm/r4r.
[38] https://github.com/opencitations/ramose.

Other efforts have attempted to improve the serialization of SPARQL results. For example, SPARQL transformer [7] and SPARQL-JSONLD[39] both present approaches for transforming SPARQL to user-friendly JSON results by using a custom mapping language and JSON-LD frames [6] respectively. In [14] the authors use GraphQL,[40] which is gaining popularity among the developer community, to generate SPARQL queries and serialize the results in JSON. In fact, some triplestores like Stardog have started to natively support interfaces for GraphQL.[41] These approaches facilitate retrieving parseable JSON from a knowledge graph, but developers still need to be familiar with the underlying ontologies used to query the data in those knowledge graphs. In OBA, an OAS with all available calls is generated automatically.

Parallel to the development of OBA, a recent effort has started to map OWL to OAS.[42] However, this approach focuses only on the mapping to OAS, while OBA also provides an implementation for creating an API server.

Finally, [4] proposes to define REST APIs to access the classes and properties of an ontology. This is different from our scope, which uses the ontology as a template to create an API to exploit the contents of a knowledge graph. To the best of our knowledge, our work is the first end-to-end framework for creating REST APIs from OWL ontologies to provide access to the contents of a knowledge graph.

6 Conclusions and Future Work

In this paper we have introduced the Ontology-Based APIs framework (OBA), a new resource for creating APIs from ontologies by using the OpenAPI Specification. OBA has demonstrated to be extremely useful in our work, helping setting up and maintaining API versions, testing and easy prototyping against a target knowledge graph. We believe that OBA helps bridging the knowledge gap between ontology engineers and developers, as it provides the means to create a guide (a documented API) that illustrates how to exploit a knowledge graph using the tools and standards developers are used to.

We are actively expanding OBA to support new features. First, we are working towards supporting additional mappings between OWL and OAS, such as complex domain and range axiomatization. Second, we are working to support accepting and delivering JSON-LD requests (instead of JSON only), which is preferred by some Semantic Web developers. As for future work, we are exploring the possibility of adding support for GraphQL, which has gained popularity lately, as an alternative to using SPARQL to retrieve and return contents. Finally, an interesting approach worth exploring is to combine an ontology with existing tools to mine patterns from knowledge graphs to expose APIs with the most common data patterns.

[39] https://github.com/usc-isi-i2/sparql-jsonld.
[40] https://graphql.org/.
[41] https://www.stardog.com/categories/graphql/.
[42] https://github.com/hammar/owl2oas.

Acknowledgements. This work was funded by the Defense Advanced Research Projects Agency with award W911NF-18-1-0027 and the National Science Foundation with award ICER-1440323. The authors would like to thank Yolanda Gil, Paola Espinoza, Carlos Badenes, Oscar Corcho, Karl Hammar and the ISWC anonymous reviewers for their thoughtful comments and feedback.

References

1. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: a nucleus for a web of open data. In: Aberer, K., et al. (eds.) ASWC/ISWC -2007. LNCS, vol. 4825, pp. 722–735. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76298-0_52
2. Cyganiak, R., Lanthaler, M., Wood, D.: RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C (2014) http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/
3. Daga, E., Panziera, L., Pedrinaci, C.: A BASILar approach for building Web APIs on top of SPARQL endpoints. In: Proceedings of the Third Workshop on Services and Applications over Linked APIs and Data, co-located with the 12th Extended Semantic Web Conference (ESWC 2015), **1359**, 22–32 (2015)
4. Dirsumilli, R., Mossakowski, T.: RESTful encapsulation of OWL API. In: Proceedings of the 5th International Conference on Data Management Technologies and Applications, DATA 2016, SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT, pp. 150–157 (2016)
5. Garijo, D., Osorio, M., Khider, D., Ratnakar, V., Gil, Y.: OKG-Soft: an Open Knowledge Graph with Machine Readable Scientific Software Metadata. In: 2019 15th International Conference on eScience (eScience), San Diego, CA, USA, pp. 349–358. IEEE (2019) https://doi.org/10.1109/eScience.2019.00046
6. Kellogg, G., Champin, P.A., Longley, D.: JSON-LD 1.1. Candidate recommendation, W3C (2020) https://www.w3.org/TR/2020/CR-json-ld11-20200417/
7. Lisena, P., Meroño-Peñuela, A., Kuhn, T., Troncy, R.: Easy web API development with SPARQL transformer. In: Ghidini, C., et al. (eds.) ISWC 2019. LNCS, vol. 11779, pp. 454–470. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30796-7_28
8. Malhotra, A., Arwe, J., Speicher, S.: Linked data platform 1.0. W3C recommendation, W3C (2015) http://www.w3.org/TR/2015/REC-ldp-20150226/
9. Mihindukulasooriya, N., Poveda-Villalón, M., García-Castro, R., Gómez-Pérez, A.: Loupe - an online tool for inspecting datasets in the linked data cloud. In: Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, PA, USA. CEUR Workshop Proceedings, CEUR-WS.org, **1486** (2015)
10. Patel-Schneider, P., Motik, B., Parsia, B.: OWL 2 web ontology language structural specification and functional-style syntax (2 edn). W3C recommendation, W3C (2012) http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/
11. Meroño-Peñuela, A., Hoekstra, R.: GRLC makes gitHub taste like linked data APIs. In: Sack, H., Rizzo, G., Steinmetz, N., Mladenić, D., Auer, S., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9989, pp. 342–353. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47602-5_48
12. Seaborne, A., Harris, S.: SPARQL 1.1 query language. W3C recommendation, W3C (2013) http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

13. Soiland-Reyes, S.: Owl2Jsonld 0.2.1 (2014) https://doi.org/10.5281/ZENODO.10565
14. Taelman, R., Vander Sande, M., Verborgh, R.: GraphQL-LD: linked data querying with GraphQL. In: Proceedings of the 17th International Semantic Web Conference: Posters and Demos, pp. 1–4 (2018)
15. Vrandečić, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Commun. ACM **57**(10), 78–85 (2014)
16. Zhu, Q., et al.: Collective multi-type entity alignment between knowledge graphs. In: Proceedings of The Web Conference 2020, WWW '20, Association for Computing Machinery, New York, NY, USA, pp. 2241–2252 (2020)