

Extending Ontology Engineering Practices to Facilitate Application Development

Paola Espinoza-Arias¹[0000–0002–3938–2064], Daniel Garijo²[0000–0003–0454–7145],
and Oscar Corcho¹[0000–0002–9260–0753]

Ontology Engineering Group, Universidad Politécnica de Madrid
{`pespinoza, ocorcho`}@`fi.upm.es`, `daniel.garijo@upm.es`

Abstract. Ontologies define data organization and meaning in Knowledge Graphs (KGs). However, ontologies have generally not been taken into account when designing and generating Application Programming Interfaces (APIs) to allow developers to consume KG data in a developer-friendly way. To fill this gap, this work proposes a method for API generation based on the artefacts generated during the ontology development process. This method is described as part of a new phase, called ontology exploitation, that may be included in the last stages of the traditional ontology development methodologies. Moreover, to support some of the tasks of the proposed method, we developed OATAPI, a tool that generates APIs from two ontology artefacts: the competency questions and the ontology serialization. The conclusions of this work reflect that the limitations found in the state-of-the-art have been addressed both at the methodological and tooling levels for the generation of APIs based on ontology artefacts. Finally, the lines of future work present several challenges that need to be addressed so that the potential of KGs and ontologies can be more easily exploited by application developers.

Keywords: Ontology Engineering · Application Development · Application Programming Interface · Ontology Artefacts.

1 Introduction

Over recent years Knowledge Graphs (KGs) have started being generated and adopted by many organizations to integrate data, facilitate interoperability, and generate new insights and recommendations. KGs are commonly structured according to ontologies, which allow data to be unambiguously defined with a shared and agreed meaning, as well as to infer new knowledge. However, despite their adoption, KGs are still challenging to consume by application developers.

On the one hand, developers face a *production-consumption challenge*: there is a gap between the ontology engineers who design an ontology and may intervene in KG creation and the application developers who want to consume its contents [7]. Ontologies may be complex, and the resources generated during their development (use cases, requirements, etc.) are often not made available to their users (e.g. application developers). As a result, developers usually need to

duplicate some of the effort already done by ontology engineers when they were understanding the domain, interacting with domain experts, taking modeling decisions, etc. On the other hand, application developers face *technical challenges*: many of them are not familiar with Semantic Web standards such as OWL and SPARQL, and hence those KGs that are exclusively based on Semantic Web technologies remain hardly accessible to them [18]. Developers (and in particular application developers) are mostly used to data representation formats like JSON and Application Programming Interfaces (APIs) for accessing data.

In order to address both production-consumption and technical challenges, multiple approaches have been proposed by the Semantic Web community, ranging from Semantic RESTful APIs [15] which are compatible with Semantic Web and REST; to tools to create Web APIs on top of SPARQL endpoints [3, 12, 1, 4]. Outside the Semantic Web community, approaches like GraphQL¹ are gaining traction among developers due to their flexibility to query and retrieve data from public endpoints. However, we have not identified so far any work focused on generating APIs based on ontology artefacts. These artefacts are any intermediate or final resource generated during the ontology development process (e.g. competency questions, SPARQL queries, ontology serialization, etc.).

The main goal of this work is to advance in this direction facilitating the KG data consumption by application developers who are not experts in ontologies, while reusing some of those intermediate or final resources of the ontology development process. Therefore, we focus on the following research questions: RQ1: *Is it possible to generate APIs based on the artefacts created by ontology engineers during the ontology development process?*, RQ2: *Is it possible to automatically generate APIs that are indistinguishable from APIs developed by application developers in real use cases?*. To answer our RQs, we propose a novel **method for API generation based on ontology artefacts**, which proposes a set of activities to define, specify, implement, validate, and deploy APIs. In addition, we develop a **proof-of-concept tool for supporting API generation**, which allows building a set of APIs and SPARQL queries based on two ontology artefacts: competency questions and ontology serialization.

The remainder of this paper is organized as follows. We begin by describing the related work (Sect. 2). Then, we present the proposed method in Sect. 3 and how to support its execution in Sect. 4. In Sect. 5, we present the results of our evaluation. Finally, we discuss future work and conclusions in Sect. 6.

2 Related Work

We consider in the related work the most relevant and well-known methodologies for ontology development with a special focus on the stages and activities they propose, and the ontology artefacts they produce. Several approaches have been proposed for developing ontologies (relevant surveys are reported in [2, 10, 11]). There are heavyweight methodologies that require time and resource-consuming activities (e.g. METHONTOLOGY [6], On-To-Knowledge [16], or NeOn [17]),

¹ <https://spec.graphql.org/June2018>

and lightweight methodologies based on agile techniques that allow building ontologies that are always ready to be used (e.g. SAMOD [13] or LOT [14]). These methodologies propose, at a lower or higher level of detail, similar core phases to develop ontologies. At the beginning of the development, the methodologies propose identifying the requirements, expressed as competency questions (CQs), that the ontology must meet. Next, they propose generating a model or intermediate representation of the ontology containing the terms and the relationships between them. Then, they propose formalizing the model (using an ontology implementation language such as OWL) and, finally, testing whether the formal representation of the ontology answers the CQs. Also, if best practices for ontology publishing are followed,² they suggest publishing the formal model along with its human-readable documentation. At the end, some methodologies identify a maintenance stage to allow fixing bugs and updating the ontology. In addition, several ontology artefacts are generated during the development process suggested by these methodologies. Among them we identified the competency questions (CQs) to specify the ontology requirements, the ontology serialization to formalize the ontology model, the glossary of terms to extract key terms and definitions from documents or data analyzed, etc. However, despite the evolution of ontology development methodologies during the years from less detailed descriptions of the steps involved to more documented phases, activities, tasks, guidelines, and techniques for ontology development, there is a lack of detail on how to use an ontology after it has been generated.

Moreover, as part of the related work we conducted a study, reported in [5], in which we analyzed existing approaches, techniques, and tools for the generation of APIs from ontologies. Our findings revealed that most of the tools and technologies do not consider ontologies or ontology artefacts for building APIs. In fact, most of the approaches we analyzed generate APIs from SPARQL queries. Only OBA [8] and OWL2OAS³ allow generating an OAS⁴ document from the OWL ontology. Moreover, OBA generates automatically SPARQL query templates needed to execute CRUD operations, and it also provides the server-side functionality for the API. Furthermore, we found that in all the tools and technologies analyzed the effort was focused on the technological support to automatically generate APIs rather than on a methodology to design them. Thus, we can state that there is not a methodological approach to build APIs taking as input the ontology artefacts produced during the ontology development process.

3 Method for API generation based on ontology artefacts

Existing ontology development methodologies (e.g. LOT, NeOn, among others mentioned in Sect. 2) usually involve phases such as requirements specification, ontology implementation, ontology publication, and maintenance. However, none of these methodologies pays much attention to how the ontology will be consumed after it has been generated. To fill this gap, we propose a new *ontology*

² <https://www.w3.org/TR/swbp-vocab-pub>

³ <https://github.com/RealEstateCore/OWL2OAS>

⁴ OpenAPI Specification (OAS) <https://swagger.io/specification>

exploitation phase encompassing any task where the ontology must be used; therefore, it may include tasks such as RDB2RDF mapping definition, RDF data generation, data consumption mechanisms provision, among others.

In this work, we focus on providing data consumption mechanisms through APIs, as none of the well-established methodologies provide details on how to generate APIs for KG data consumption. Thus, the ontology exploitation phase describes the *API generation* process. Figure 1 illustrates this phase as an extension of the LOT methodology.⁵ However, this phase can be adopted by any methodology as it makes use of artefacts that are commonly generated in all these methodologies and it should be considered at the end of the development process. As Figure 1 shows, the actors involved in the ontology exploitation phase are ontology developers (aka ontology engineers) and application developers. In addition, the inputs of the phase are the ontology artefacts produced during the previous phases of the ontology development process, and the output is an API.

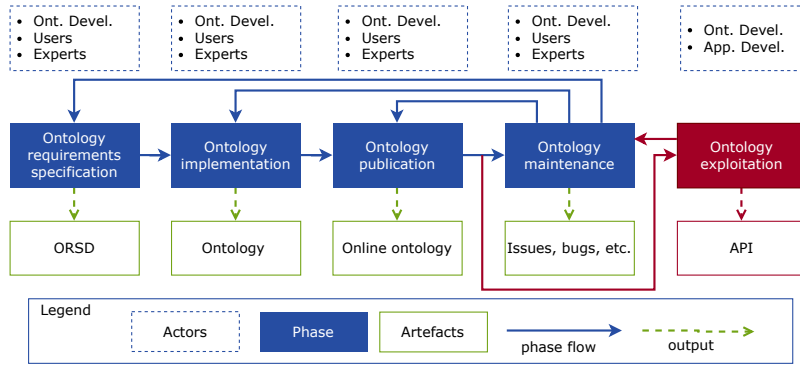


Fig. 1. Ontology exploitation phase as an extension of the LOT Methodology. New elements are illustrated in magenta.

To define the APIs, our method proposes a set of activities to build APIs from ontology artefacts. This method is inspired in the workflow for designing and building Web APIs presented in [9]. Figure 2 summarizes the five activities of the method. Below we describe each activity and to illustrate its execution we present some examples using two artefacts of an ontology for the representation of the local businesses of a municipality developed in the context of the Open Cities⁶ project. The artefacts from the Local Business Census ontology are: its ontology serialization⁷ and its CQs.⁸

⁵ For readability, we call the LOT's activities phases. Thus, we describe the methodology as a set of phases involving activities, these activities involve several tasks, and so on.

⁶ Open Cities (Ciudades Abiertas) <https://ciudades-abiertas.es>

⁷ <http://vocab.ciudadesabiertas.es/def/comercio/tejido-comercial>

⁸ <https://github.com/CiudadesAbiertas/vocab-comercio-censo-locales/tree/master/requirements>

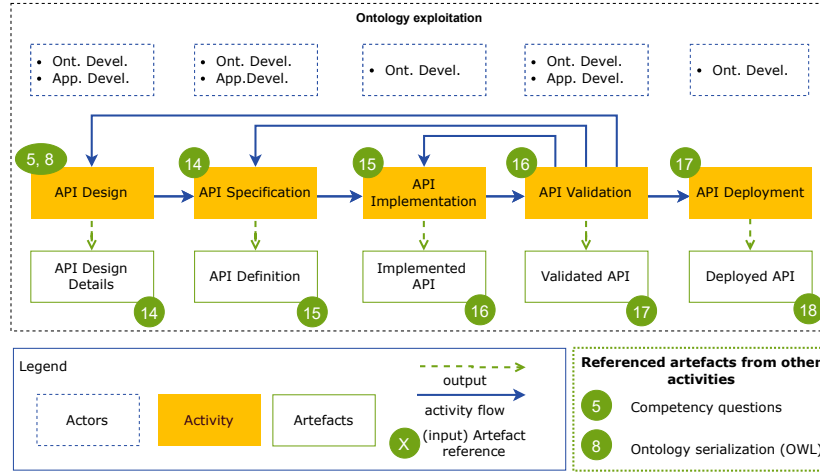


Fig. 2. Activities of the method for API generation based on ontology artefacts. This figure follows the convention defined by the LOT methodology.

1) API design: This activity focuses on deciding how the API will behave and defining the resources along with the operations that the API will provide. To this end, we propose a set of tasks to guide ontology engineers and application developers in making these decisions and defining resources and operations taking ontology artefacts as the main input. Figure 3 shows the tasks of this activity and their inputs and outputs. Each task is described below.

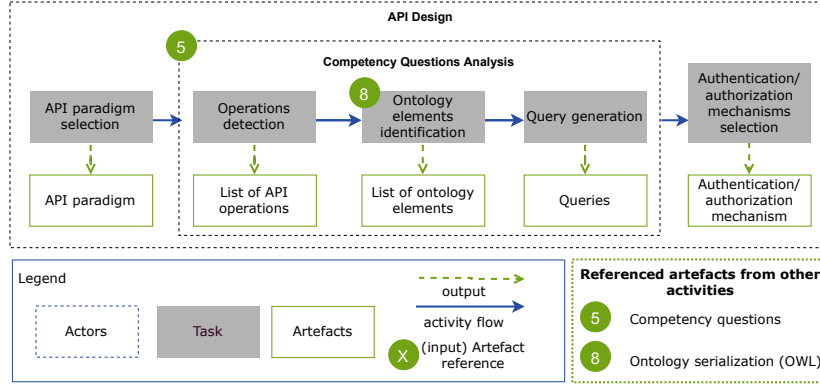


Fig. 3. Tasks and subtasks involved in the API design activity

1.1) API paradigm selection: This task aims to choose what style will follow the API. The selected paradigm may be a request-response one such as those oriented to a resource-style, hypermedia-style, query-style, among others. The selection on which paradigm to use depends on the decision of ontology engineers and application developers, after an analysis of the existing paradigms and how

they fit in the requirements that need to be addressed by the ontology, maintainability, a specific application behaviour, etc. In the Semantic Web community the most common paradigms adopted are REST (resource-style) and GraphQL (query-style), as we have detected in our study reported in [5]. Thus, this method describes how to carry out the activities and tasks according to both paradigms.

Example

In this first task, let us assume we select the resource-oriented REST paradigm to generate the API of the Local Business Census ontology.

1.2) Competency questions analysis: This task concentrates on analyzing the CQs to find the terms required for the API design. To this end, the following three subtasks should be executed:

- **Operations detection.** This subtask aims to identify which operations will be implemented in the API according to the intent of the CQs. To detect operations, the ontology engineer must analyze specific terms that request something in the CQs. Table 1 shows some examples of common terms and their correspondence with the operations of the REST and GraphQL paradigms. In general, CQs consider read-only operations rather than data changes requests. However, if a CQ intends to perform data changes the operation detection should also consider terms related to write operations.

Table 1. Common competency questions terms denoting operations in APIs.

| Term \ Paradigm | REST | GraphQL |
|--|--------|----------|
| get, list, which, what, who, where, when, obtain, give | GET | query |
| add, insert | POST | mutation |
| update, change | PUT | mutation |
| delete, remove | DELETE | mutation |

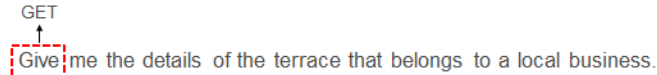
Also, for those CQs aiming to solve boolean or counting questions a different strategy is required. Table 2 shows examples of common terms defining these CQs and their correspondence with the operations of each API paradigm. In general, to solve these queries, CQs can be related to a read operation whose resulting data must be processed to deliver the expected responses. Thus, to solve counting questions, API consumers may count the resulting data on the client-side to get the number of elements. As for boolean questions, API consumers may analyze the resulting data to determine whether the query response is true (more than zero results), or false (zero results).

Table 2. Example of operations detection for counting and boolean queries

| Term | Query type | REST | GraphQL |
|---------------|------------|------|---------|
| how many | counting | GET | query |
| is, was, were | boolean | GET | query |

Example

For the CQs analysis, we choose one question from the set of CQs of the Local Business Census ontology. Then, we analyze each term of the CQ and detect a coincidence with one of the terms presented in Table 1. The image below shows the CQ, the term denoting an operation (Give), and the detected operation (GET):



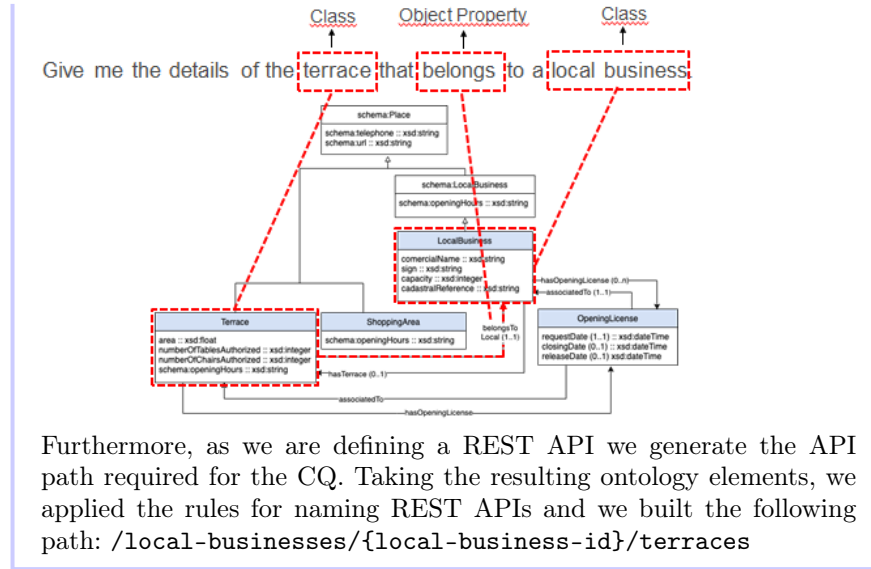
The diagram shows the text "Give me the details of the terrace that belongs to a local business." where the word "Give" is enclosed in a red dashed box. An upward-pointing arrow connects "Give" to the word "GET" above it.

- **Ontology elements identification.** This subtask aims to distinguish which ontology elements are required by the API. To do so, it uses the ontology serialization as input to verify whether the labels of the ontology elements match the terms identified in the CQs. This verification is essential to detect how these elements are related and, depending on the operation, the structure of the expected API input/output. An alternative way to identify the ontology elements may be to analyze the fragment identifier of their URIs. This alternative can be applied when URIs are defined with short and compact names in natural language. However, this alternative may not be applicable when URIs follow an opaque strategy (which obfuscates the name of the ontology elements) because it hinders the elements identification. Moreover, in those cases where the terms identified do not match ontology elements it would be necessary to check for their synonyms. To do this, the ontology engineer may reuse the glossary of terms generated at the beginning of the ontology development. Thus, some terms found in the CQ can be compared with those in the glossary to find the corresponding ontology elements. Another option for identifying terms would be to use existing open glossaries (domain dependent or independent). Finally, when the selected API paradigm is REST it will be necessary to use the elements identified in this subtask to define the API paths required for each CQ. To this end, we designed a set of rules for naming REST APIs based on ontology artefacts. These rules are described in our GitHub repository.⁹

Example

To identify the ontology elements, we analyze the CQ and the ontology serialization. In the image below, we illustrate the ontology serialization as an excerpt of the diagram of the Local Businesses Census ontology. As shown in this image, some terms from the CQ match the labels of the ontology serialization, and, as a result, we identified the **Terrace** and **LocalBusiness** classes and the **belongsToLocal** object property.

⁹ <https://github.com/oeg-upm/oatapi/blob/main/Additional%20Resources>



- **Query generation.** This subtask intends to define the query to manage the data and perform the functionality required by the CQ. To this end, the SPARQL queries generated for the ontology validation may be reused, if available. This validation strategy is common in ontology development methodologies as it allows ontology engineers to verify whether the ontology fulfills the requirements against a proof of concept dataset. However, if queries had not been defined or are not available they must be generated. Therefore, for the query generation, two steps should be carried out. First, take the operation, detected in the first subtask, and determine what SPARQL operation should be included in the query. Second, match the ontology elements, detected in the second subtask, with the query structure and format the query accordingly. The query structure should be defined according to the CQ intention and its expected answer.

Example

We manually build the SPARQL query to fetch the data, as follows:

```

1 PREFIX escom:<http://vocab.ciudadesabiertas.es/def/comercio/
  tejido-comercial#>
2 CONSTRUCT { ?terrace ?predicate ?object }
3 WHERE { {
4   SELECT DISTINCT ?terrace
5     WHERE {?terrace escom:perteneceALocal ?local-business-id}}
6   ?terrace ?predicate ?object }
```

Note that the `?local-business-id` variable will be replaced by the value specified in the request as it corresponds to the `{local-business-id}` template defined in the API path. Also, as the operation is `GET`, the SPARQL operation corresponds to `SELECT`. This query includes a

CONSTRUCT statement, since the result of the request will be composed of all the values of the properties of each instance found.

1.3) Authentication and authorization mechanisms selection: This task aims to define which security mechanisms the API must handle. Depending on operations and data that the API will manage, it may be necessary to provide a security level to control who is allowed and what functions they may execute. For example, it may be necessary to restrict personal data access, or write operations may be allowed only to certain users. The security methods to be provided may be basic (e.g. HTTP requests including username and password) or more reliable (e.g. OAuth 2.0 protocol which is based on the provision of a token access).

Example

Finally, since this API does not involve confidential data, does not require accounting mechanisms, and the CQs correspond to read operations, we decided not to use a security mechanism.

2) API specification: The purpose of this activity is to describe the decisions made on the API design. To do so, a common practice is using an Interface Description Language (IDL) which allows documenting several aspects of the API according to a syntax provided in a machine-readable format (e.g. OAS). Using a description language allows to quickly generate the API documentation because its syntax can be translated into, for example, an HTML document that may include interactions showing how the API works. However, other specifications¹⁰ can be adopted as long as they provide facts about the exposed resources, allowed methods, message codes that the API handle, result formats, expected inputs/outputs, among others. It is important to provide a good specification to help application developers learn and understand how the API works.

Example

Assume that we write the API details according to the specification template we designed. The following table presents an extract of this template filled in with the details of the API path in terms of the allowed operation, its input, and the expected result. The full specification of this example is available in our repository along with our API Specification template.

| API path | Operation | Input | Output |
|--|-----------|-------------------|--|
| /local-businesses/{local-business-id}/terraces | GET | local-business-id | an array of terrace instances (each instance will contain its attribute values of area, table number, opening hours, number of authorized chairs, among others.) |

3) API implementation: This activity refers to building the API functionality according to its design and specification. Thus, it is necessary to build the server-side logic to handle each request and deliver the response from the SPARQL

¹⁰ An API specification template we have developed is available in <https://github.com/oeg-upm/oatapi/tree/main/Additional%20Resources>

endpoint. Also, if a security mechanism is required, the API behavior needs to be configured based on the user executing the request. Figure 4 illustrates the interactions that can occur between the client and the endpoint. In this figure, the API acts as a middle layer that allows clients to make requests. These requests are processed by the server to query the endpoint which returns data. Last, the server delivers data to the client in the appropriate format via the API.

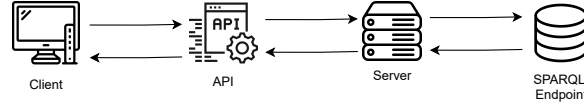


Fig. 4. Interactions between a client and SPARQL endpoint

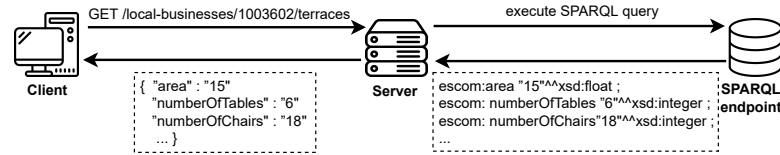
Example

Suppose we develop the server-side functionality to handle the API calls and provide the expected responses. This functionality includes executing SPARQL queries according to each API call, serializing the results to deliver the response in a developer-friendly format (e.g. JSON), and configuring the messages to retrieve when executing the requests.

4) API validation: This activity is carried out jointly by the ontology engineer and application developer, and it is concerned with testing whether the API calls perform the requested behaviour, which is determined by the answers of the CQs. If this validation fails it will be necessary to go back to the previous activities to review and refine the design, specification or implementation of the API.

Example

Suppose the server we implemented in the previous activity is up, and we develop a test script that acts as a client to make API calls against this server. We illustrate the API validation by the following figure showing the interactions between the client, the server, and the SPARQL endpoint.



First, the client executes the GET call. Then, the server receives this call requesting data from the local business 10003602, and executes the SPARQL query that was earlier defined in our example. Next, the SPARQL endpoint returns the attributes and values of the terraces that belong to this local business (for simplicity, few attributes and values in Turtle are shown). Lastly, the server fetches data from the endpoint, formats that data into the requested serialization (JSON), and delivers it to the client. This validation is successful since the response contains the expected data described in the output column from our API specification example.

5) API deployment: This last activity refers to making the API available online. Moreover, in this activity it is important to provide access to the API documentation to help developers learn and understand how the API works. Thus, the API specification, represented in a human-readable format, should also be made available. Finally, it is advisable to make the API findable on the Web to ease its promotion and discovery. To this end, ontology engineers may publish the API in registries (e.g. Programmable Web,¹¹ BioCatalogue,¹² etc.)

Example

Finally, suppose we put the server on-line, make the API specification available on the Web, and register the API in the Programmable Web registry.

4 Supporting the API generation process

To automate some activities and tasks of our method, we developed OATAPI (Ontology Artefacts to API). This proof-of-concept tool¹³ takes as input a set of CQs and the ontology serialization. Then, it analyzes both artefacts and delivers a set of REST API paths and SPARQL queries that allow getting data to solve the CQs. We decided to build REST APIs since REST allows us to directly use SPARQL queries to access data that is hosted in traditional SPARQL endpoints. Thus, OATAPI performs by default the paradigm selection from the API design activity. This tool also automates the CQ analysis task from the API design, which is formalized in Algorithm 1. It begins by loading the CQs and the ontology serialization into OATAPI. Then, the algorithm executes the following steps:

1. **Competency questions breakdown.** It consists of splitting each CQ into pieces to be able to analyze its terms, and tagging each piece according to its respective part-of-speech and dependency tree labels. Each piece considers only the base form of the terms to perform the analysis.
2. **Operations identification.** It detects which pieces from step 1 match terms denoting operations. To that end, the algorithm compares each piece with the terms presented in Table 1.
3. **Ontology elements identification.** It identifies the elements in two steps:
 - (a) Load the ontology elements into a directed multigraph. Classes are considered as nodes and object properties as edges. Thus, OATAPI requires that object properties contain domain and range restrictions such that edges contain information about the node they come from and go to.
 - (b) Take each CQ's piece and check if it matches the label of the ontology elements. If the ontology does not contain labels, the algorithm checks the fragment URI identifier of the ontology elements (only in case the fragment is defined with names in natural language).
4. **API path generation.** It builds the paths by executing the following stages:
 - (a) Search for coincidences between the ontology elements identified in step 3 and the nodes from the multigraph.

¹¹ <https://www.programmableweb.com>

¹² <https://www.biocatalogue.org>

¹³ OATAPI is publicly available in <https://github.com/oeg-upm/oatapi>

- (b) Obtain the shortest path between the nodes found in step 4(a).
 - (c) Take the nodes of the shortest path and generate the API path. To build the path, the algorithm considers the order of the nodes and follows the rules for naming REST APIs based on ontology artefacts we designed.
5. **Query generation.** It builds SPARQL CONSTRUCT queries using the nodes obtained in step 4(b) and the operations identified in step 2.

Algorithm 1 : Pseudocode for the competency questions analysis

Input: Competency questions (CQs) and ontology serialization (onto)

Output: API paths (APIpaths) and SPARQL queries (queries)

```

1: loadCompetencyQuestions(CQs);
2: loadOntology(onto);
3: for all cq in CQs do
4:    $CQ \leftarrow cqBreakdown(cq)$ ;
5:    $operation \leftarrow operationIdentification(CQ)$ ;
6:    $ontologyElements \leftarrow ontologyElementsIdentification(onto)$ ;
7:    $APIpaths, nodesFound \leftarrow APIpathGeneration(ontologyElements)$ ;
8:    $queries \leftarrow queryGeneration(operation, nodesFound)$ ;
9: end for
10: return APIpaths, queries;
```

It is worth mentioning that the stages described in step 4 present a common case when there are matches between ontology elements and nodes. However, there are cases where only one node is detected, where an edge matches an object property, or both. Moreover, step 5 was also described for a common case when a shortest path is found; but, the query structure will change when other cases occur (e.g. when only one or two nodes are found). These cases are supported by OATAPI, but have omitted from the manuscript due to space constraints.

5 Evaluation

We evaluated¹⁴ our work by means of two experiments. The first one aimed to test whether participants of a questionnaire are able to detect if the shown API paths were manually or automatically built, or if it is not possible to determine that (they are indistinguishable). The second intended to test whether the API paths manually generated are similar to the automatically built ones.

5.1 Experiment 1

Experiment settings. For this evaluation, we generated a testbed which contains: 1) a set of CQs and ontologies (their IRIs) from different domains, 2) API paths generated by developers in real use cases for these CQs and ontologies. Next, taking as input the CQs and ontologies from the testbed, we executed OATAPI to automatically build their API paths, and we also included these resulting paths in the testbed. Then, to compare the API paths gathered in our testbed, we created a questionnaire. First, the participants of the experiment

¹⁴ The resources and results of our evaluation are available in our repository <https://github.com/oeg-upm/oatapi/tree/main/Evaluation>

were asked about their background and if they knew what is the correct response of executing a GET operation in some REST API calls. Then, they were asked for each CQ and its API paths. For each pair of API paths, they were asked to determine how the APIs were generated (automatically or manually), if possible; and of the API paths shown, which path they would prefer to use.

Results. Regarding the background of the 20 participants, most of them were used to web services and REST APIs, had developed 1-5 web APIs, and were familiar with Semantic Web technologies. As for general questions about REST APIs, most of them knew the correct answer of executing different API calls. With respect to the last part of the questionnaire, Table 3 summarizes the evaluation results of the API paths presented to the participants. Each column header contains the options available in the questionnaire. The "1st" represents the API paths generated by developers, and the "2nd" those paths built by OATAPI.

Table 3. Summary of evaluation results of manually vs automatically built API paths

| 1st automatically - 2nd manually | 2nd automatically - 1st manually | Both man- ually | Both auto- matically | Indistin- guishable |
|-------------------------------------|-------------------------------------|--------------------|-------------------------|------------------------|
| 50% | 10% | 5% | 10% | 25% |
| 25% | 15% | 15% | 10% | 35% |
| 35% | 15% | 15% | 5% | 30% |
| 45% | 15% | 5% | 10% | 25% |
| 40% | 20% | 5% | 5% | 30% |
| 35% | 10% | 5% | 10% | 40% |
| 30% | 25% | 5% | 5% | 35% |
| 30% | 15% | 15% | 5% | 35% |
| 30% | 20% | 10% | 5% | 35% |

From this table it can be seen that, in general, regarding the way APIs were generated, participants mostly selected the first (first column) and the last (fourth column) option. As for the first option, participants believed that the "1st" API path was built automatically and the "2nd" was generated manually. However, this option is not true because the "1st" are the API paths built by developers and not automatically, as we explained earlier. As for the last option, the participants were not able to determine how the API paths were constructed (indistinguishable). These results allow us to conclude that automatically generated API paths are indistinguishable from manually built ones.

5.2 Experiment 2

Experiment settings. For this evaluation, we generated a testbed which contains: 1) a set of CQs and ontologies, 2) API paths manually built, according to the steps of our method, for these CQs and ontologies, and 3) API paths built by OATAPI taking these CQs and ontologies as input. Then, we use the ROUGE (Recall-Oriented Understudy for Gisting Evaluation) metric to compare the similarity between the API paths from our testbed.

Results. Table 4 summarizes the results of the similarity comparison between the API paths from our testbed. The first column contains the prefixes of the analyzed ontologies. The values shown in the precision, recall, and F-measure columns represent the sum of the result of each metric value obtained after executing each comparison. These values are round numbers since we obtained 1

or 0 for these metrics after evaluating each couple of paths. This occurred because we evaluated API paths as one token; therefore ROUGE can only determine if there is a full syntactical match or not between the paths evaluated.

Table 4. Results of similarity between manually and automatically built API paths.

| Ontology | Precision | Recall | F-measure | API paths evaluated |
|------------|-----------|--------|-----------|---------------------|
| VGO | 15 | 15 | 15 | 22 |
| ESCOM | 8 | 8 | 8 | 8 |
| SWO | 0 | 0 | 0 | 3 |
| SAREF4ENVI | 1 | 1 | 1 | 2 |
| NOISE | 3 | 3 | 3 | 3 |
| PPROC | 5 | 5 | 5 | 5 |
| ESAIR | 3 | 3 | 3 | 3 |
| ESBICI | 3 | 3 | 3 | 4 |

From this results it can be observed that 38 of the 50 API paths evaluated are similar; it corresponds to the 76% of the API paths. These results allow us to conclude that OATAPI is able to generate API paths similar to those built manually by following the steps proposed in our method.

6 Conclusions and Future Work

In this work, we proposed a method designed to extend ontology engineering practices for building APIs. The method provides details on the activities and tasks required to build APIs making use of ontology artefacts, including examples on how to carry them out. From our method, we can conclude that it is possible to generate APIs based on the artefacts created by ontology engineers during the ontology development process (answer to RQ1). We also conclude that making ontology artefacts publicly available is important because these artefacts can be analyzed and reused not only for API generation, but to investigate new solutions for ontology documentation, formalization, testing, or exploitation.

In addition, we developed OATAPI to automate some of the API design tasks of our method. From the evaluation results, we can conclude that it is possible to automatically build API paths that are indistinguishable from those manually generated by application developers (answer to RQ2). We can also conclude that it is possible to automatically generate similar API paths than those manually built following our proposed method. From our evaluation we also conclude that the definition of CQs should be as precise as possible since verbose and ambiguous expressions hinder the analysis of these questions in terms of identifying ontology elements, and as a result limit API generation.

Although OATAPI allows generating the API paths and SPARQL queries for the CQs there is still work to be done to improve its functionality. Possible directions could be 1) using inference for ontology elements identification to detect those that are not directly defined but are inherited from the parent classes/properties; and 2) enabling synonym detection to ease the ontology elements detection by reusing, for example, the glossary of terms. We aim to move OATAPI from a prototype to a production tool that can be used in real-world scenarios and, as a result, to get wide user feedback, especially from application developers. This feedback will allow us to refine both the method and the

tool to continue working on providing resources to facilitate the development of applications that consume KG data.

References

1. Badenes-Olmedo, C., Espinoza-Arias, P., Corcho, O.: R4R: Template-based REST API Framework for RDF Knowledge Graphs. In: Proceedings of the ISWC 2021 Posters, Demos and Industry Tracks: From Novel Ideas to Industrial Practice co-located with 20th International Semantic Web Conference, Virtual Conference, 2021. CEUR Workshop Proceedings (2021)
2. Corcho, O., Fernández-López, M., Gómez-Pérez, A.: Methodologies, tools and languages for building ontologies. Where is their meeting point? *Data & knowledge engineering* **46**(1), 41–64 (2003)
3. Daga, E., Panziera, L., Pedrinaci, C.: A BASILar approach for building web APIs on top of SPARQL endpoints. In: CEUR Workshop Proceedings. vol. 1359, pp. 22–32 (2015)
4. Daquino, M., Heibi, I., Peroni, S., Shotton, D.: Creating RESTful APIs over SPARQL endpoints using RAMOSE. *Semantic Web* **23**(2), 195–213 (2022)
5. Espinoza-Arias, P., Garijo, D., Corcho, O.: Crossing the Chasm Between Ontology Engineering and Application Development: A Survey. *Journal of Web Semantics* **70**, 100655 (2021)
6. Fernández-López, M., Gómez-Pérez, A., Juristo, N.: Methontology: from ontological art towards ontological engineering (1997)
7. Fletcher, G., Groth, P., Sequeda, J.: Knowledge Scientists: Unlocking the data-driven organization. arXiv preprint arXiv:2004.07917 (2020)
8. Garijo, D., Osorio, M.: OBA: An Ontology-Based Framework for Creating REST APIs for Knowledge Graphs. In: Pan, J.Z., Tamma, V., d’Amato, C., Janowicz, K., Fu, B., Polleres, A., Seneviratne, O., Kagal, L. (eds.) *The Semantic Web – ISWC 2020*. pp. 48–64. Springer International Publishing, Cham (2020)
9. Jin, B., Sahni, S., Shevat, A.: Designing Web APIs: Building APIs That Developers Love. O’Reilly Media, Inc. (2018)
10. Keet, M.: *An Introduction to Ontology Engineering*, vol. 1. Maria Keet (2018)
11. Kotis, K.I., Vouros, G.A., Spiliotopoulos, D.: Ontology engineering methodologies for the evolution of living and reused ontologies: status, trends, findings and recommendations. *The Knowledge Engineering Review* **35** (2020)
12. Meroño-Peñuela, A., Hoekstra, R.: grlc Makes GitHub Taste Like Linked Data APIs. In: *European Semantic Web Conference*. pp. 342–353. Springer (2016)
13. Peroni, S.: A simplified agile methodology for ontology development. In: *Proceedings of the 13th OWL: Experiences and Directions Workshop and 5th OWL reasoner evaluation workshop*, Bologna, Italy. pp. 55–69. Springer (2016)
14. Poveda-Villalón, M., Fernández-Izquierdo, A., Fernández-López, M., García-Castro, R.: LOT: An industrial oriented ontology engineering framework. *Engineering Applications of Artificial Intelligence* **111**, 104755 (2022)
15. Salvadori, I., Siqueira, F.: A Maturity Model for Semantic RESTful Web APIs. In: *2015 IEEE International Conference on Web Services*. pp. 703–710. IEEE (2015)
16. Staab, S., Studer, R., Schnurr, H.P., Sure, Y.: Knowledge processes and ontologies. *IEEE Intelligent systems* **16**(1), 26–34 (2001), iEEE
17. Suárez-Figueroa, M.C., Gómez-Pérez, A., Fernandez-Lopez, M.: The neon methodology framework: A scenario-based methodology for ontology development. *Applied ontology* **10**(2), 107–145 (2015)
18. Verborgh, R., Vander Sande, M.: The Semantic Web identity crisis: in search of the trivialities that never were. *Semant. Web J.* **11** (1), 19–27 (2020)