Journal Pre-proof

Crossing the chasm between ontology engineering and application development: A survey

Paola Espinoza-Arias, Daniel Garijo, Oscar Corcho

PII: S1570-8268(21)00030-5
DOI: https://doi.org/10.1016/j.websem.2021.100655
Reference: WEBSEM 100655

To appear in: *Web Semantics: Science, Services and Agents on the World Wide Web*

Received date : 14 January 2021
Revised date : 26 March 2021
Accepted date : 2 June 2021

Please cite this article as: P. Espinoza-Arias, D. Garijo and O. Corcho, Crossing the chasm between ontology engineering and application development: A survey, *Web Semantics: Science, Services and Agents on the World Wide Web* (2021), doi: https://doi.org/10.1016/j.websem.2021.100655.

Crossing the Chasm Between Ontology Engineering and Application Development: A Survey

Paola Espinoza-Arias[a,*], Daniel Garijo[b] and Oscar Corcho[a]

[a]Ontology Engineering Group, Universidad Politécnica de Madrid, MAD, Spain
[b]Information Sciences Institute, University of Southern California, CA, United States

ARTICLE INFO

Keywords:
Ontology
OWL
Ontology Engineering
Web API
Application Development
Knowledge Graph

ABSTRACT

The adoption of Knowledge Graphs (KGs) by public and private organizations to integrate and publish data has increased in recent years. Ontologies play a crucial role in providing the structure for KGs, but are usually disregarded when designing Application Programming Interfaces (APIs) to enable browsing KGs in a developer-friendly manner. In this paper we provide a systematic review of the state of the art on existing approaches to ease access to ontology-based KG data by application developers. We propose two comparison frameworks to understand specifications, technologies and tools responsible for providing APIs for KGs. Our results reveal several limitations on existing API-based specifications, technologies and tools for KG consumption, which outline exciting research challenges including automatic API generation, API resource path prediction, ontology-based API versioning, and API validation and testing.

1. Introduction

Knowledge Graphs (KGs) have become a crucial asset for structuring data and factual knowledge in private and public organizations. Several prominent KGs have been generated over the years to improve search capabilities, empower business analytics, ease decision making, etc. [48]. Industry KGs have been created by companies like Google, Microsoft, Facebook, eBay, or IBM to make their services "smarter" and add value to users [73]. Open KGs such as DBpedia [64] cover a wide variety of domains, and crowdsourced KGs like Wikidata [100] are actively maintained by an international community of curators. Domain-specific KGs have been used to open data by public administrations of several countries (e.g. national administrations: US [46], UK [87], and local administrations: Zaragoza in Spain [27], Bologna in Italy [16]); by libraries (e.g. by the Spanish [99], British [22], and French [89] National Libraries); by the life sciences community (e.g., the Monarch initiative to integrate data of genes, diseases, phenotypes, variants, and genotypes across species [88], and DisGeNET [79] to describe data about genes and variants associated to human diseases); among others.

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 a KG and the application developers who want to consume its contents [34]. KGs are commonly organized by ontologies [91], which are used to structure data without ambiguities, provide shared meaning and infer new knowledge. Ontologies are usually developed following well defined methodologies [15, 37, 56, 59], which identify use cases and competency questions that drive their design. However, ontologies can become complex, and the resources used in their development (use cases, requirements, discussion logs, etc.) are often not made available to 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 a *technical challenge*: many of them are not familiar with Semantic Web standards such as OWL [69] and SPARQL [85], and hence KGs based on Semantic Web technologies remain hardly accessible to them [98]. Developers (and in particular web developers) are mostly used to data representation formats like JSON [6]; and Application Programming Interfaces (APIs) for accessing their data. APIs allow the communication and interaction between services without having to provide details about how they are implemented. The de facto architectural style for building APIs is the scalable and resource-oriented REpresentational State Transfer (REST) architectural style [33].

In order to address both data representation and technical challenges, multiple approaches have been proposed in recent years by the Semantic Web community, ranging from Semantic RESTful APIs [83] compatible with Semantic Web and REST; to tools to create Web APIs on top of SPARQL endpoints [41, 20, 70, 84]. Outside the Semantic Web community, approaches like GraphQL [35] are gaining traction among developers due to their flexibility to query and retrieve data from public endpoints. However, to the best of our knowledge there is no framework to compare the capabilities and differences of these existing efforts.

The contribution of this paper is a systematic literature review to analyze and compare existing API-based specifications and tools for 1) making KG data more accessible to application developers, and 2) helping ontology engineers

*Corresponding author
pespinoza@fi.upm.es (P. Espinoza-Arias); dgarijo@isi.edu (D. Garijo); ocorcho@fi.upm.es (O. Corcho)
ORCID(s): 0000-0002-3938-2064 (P. Espinoza-Arias); 0000-0003-0454-7145 (D. Garijo); 0000-0002-9260-0753 (O. Corcho)

guide application developers in KG consumption. In our review, we introduce two comparison frameworks for analyzing existing specifications, technologies and tools designed to address any of these points; and outline their limitations and remaining research challenges. Our effort goes beyond existing guidelines for building Semantic RESTful Technologies [13], as we discuss the features of nine different specifications and nineteen technologies and tools rather than recommending one of them based on a series of requirements.

The rest of the paper is structured as follows. Section 2 describes three typical examples that highlight the challenges introduced above and motivate the research questions addressed in our survey. Section 3 follows with an explanation of the methodology used in our literature review. Section 4 describes the different specifications, technologies and tools found; and Section 5 compares their features and capabilities. Finally, we answer our research questions and discuss open research challenges in Section 6, and conclude the paper in Section 7.

2. Motivating Examples and Research Questions

In order to illustrate the challenges described in the previous section, we use one of the many open data projects being carried out in Spain. Ciudades Abiertas,[1] (i.e., Open Cities) is a project where several Spanish cities (A Coruña, Madrid, Santiago de Compostela and Zaragoza) are working together to create a shared set of ontologies to provide homogeneous data access in their open data portals and APIs. A total of eleven ontologies have been created in several domains, like local business census, inhabitant demographics, budgets, etc.

Thanks to this initiative, city councils, industry and citizens have been able to use open data to develop applications e.g., to display the empty retail units in a specific city area, to showcase the education level of the inhabitants of a city regarding a specific year, district, sex or age range; etc.

2.1. Accessing and manipulating KG data

For our first example we will focus on an ontology for representing data about local businesses (see Appendix C.1 for an overview diagram).[2] This ontology is easy to follow by an ontology engineer, as it consists of four main concepts (local business, opening license, terrace, shopping area), some datatype and object properties (economic activity type, operational time period, area, capacity, etc.), and SKOS concepts which represent thesauri terms (e.g. thesaurus of the situation type of local businesses[3]). However, developers who intend to build an application with data described according to this ontology may not consider it so simple. These developers may have several questions prior to consuming data, such as how to retrieve common data patterns needed for their applications (e.g. empty retail units)?; or how to operate with the semantic data serializations resulting from query execution (in a format like JSON)?

2.2. Understanding complex ontology-based data

For our second example (see Appendix C.2 to see an overview diagram), let us consider an ontology for representing the census of inhabitants of an area,[4] which has a certain degree of complexity even for experienced ontology engineers. The ontology reuses the RDF Data Cube Vocabulary [19], which represents multidimensional data such as official statistics. The ontology also involves understanding a large amount of concepts (dimensions, measures, slices, etc.), properties and lists of concepts that may be challenging for application developers who are not used to this type of representation.

In this scenario, the ontology engineers who designed the open data KG may be concerned on how to expose data represented with this ontology in a developer-friendly manner through an API. Therefore, they may have several questions like the classes that should be exposed to ensure usability; the API paths should be provided to ease data cube access; or whether dimensions, measures, etc. should be included in those API paths.

2.3. Dynamic data needs

Some city councils are implementing an open-data-by-default policy, which usually implies that they are already the main consumers of their own open data [27]. Application developers inside the city council will thus not only perform read operations on the data, but will also need to perform changes.

Developers may also have some additional questions because data usually exposed through APIs (e.g. a resource or a list of resources) may not enough for their needs. Therefore, these developers may need to know how to define API calls or queries to handle specific data for their applications (e.g. to get local businesses in active situation that have a terrace with an annual operating period).

2.4. Research questions

The examples described above showcase three typical scenarios that ontology engineers and application developers face often. Similar scenarios may occur when developers need to consume KG data structured following an ontology or an ontology network (i.e., ontology-based KG data). Each of the three examples contributes to motivate a research question (RQ), as described below.

RQ1: How can KG consumption by application developers be facilitated?[5]

- *RQ1.1: Are there any API-based methodologies / methods / processes to ease KG consumption by application developers?*

[1] https://ciudadesabiertas.es/

[2] http://vocab.ciudadesabiertas.es/def/comercio/tejido-comercial

[3] http://vocab.linkeddata.es/datosabiertos/kos/comercio/tipo-situacion

[4] http://vocab.ciudadesabiertas.es/def/demografia/cubo-padron-municipal

[5] Methods and their corresponding implementations (tools/technologies) have been separated into two sub-research questions for clarity.

- *RQ1.2: Are there any technologies that ease / automate the execution of the API-based methodologies / methods / processes to consume KGs?*

RQ2: How can ontology engineers be guided to create APIs that ease ontology-based KG consumption?

- *RQ2.1: Are there any methodologies / methods / processes to help ontology engineers creating APIs that ease ontology-based KG consumption?*
- *RQ2.2: Are there any technologies that ease / automate the execution of methodologies / methods / processes to help ontology engineers creating APIs that ease ontology-based KG consumption?*

RQ3: Are there any tools to help application developers creating APIs on demand?

3. Survey Methodology

In this section we describe the process followed to identify approaches and associated technologies that have been used to expose ontology-based KG data as APIs. Our methodology is based on the guidelines defined by Kitchenham and Charters [58] for conducting systematic literature reviews. These guidelines define a process which consists of three phases namely planning, conducting and reporting the review.

3.1. Planning the review

The main objective of this phase is to describe how the review has been carried out. To do so, the following points should be addressed: (a) the research questions; (b) the source selection and search; (c) the inclusion and exclusion criteria; and (d) the selection procedure. Since the research questions have already been defined in subsection 2.4, this section elaborates points (b)-(d).

3.1.1. Source selection and search

We used Scopus [24], a well-known database of peer-reviewed literature, to perform our review. Scopus contains specialized journals and venues that are relevant for our survey such as:

- *Journals*: Semantic Web Journal: Interoperability, Usability, Applicability (SWJ), Journal of Web Semantics: Science, Services and Agents on the World Wide Web (JWS), among others.
- *Proceedings of Conferences*: International Semantic Web Conference (ISWC), World Wide Web Conference (WWW), Extended Semantic Web Conference (ESWC), SEMANTiCS, Conference on Software Engineering and Knowledge Engineering (SEKE), among others.

We queried Scopus for potential candidates using the following query to search in titles, abstracts and keywords of related articles: (TITLE-ABS-KEY ((ontology OR OWL OR "linked data" OR "semantic data" OR "knowledge graph") AND (API OR "web API") AND (tool OR technology OR method OR methodology OR process))).

To avoid systematic bias [58], we included non-scientific literature describing relevant work in this area such as related W3C Recommendations and Technical Specifications, and existing tools developed by the Semantic Web community:

- Linked Data Platform (LDP) [2]
- Linked Data API specification (LDA) [39]
- Solid [5]
- Pubby [18]
- Puelia [40]
- ELDA [25]
- Linked Data Templates [54]

We also contacted experts and researchers working in the area and asked them whether they knew of any additional efforts, including unpublished results or ongoing work. As a result we collected the following additional efforts:

- Linked Open Data Inspector (LODI) [32]
- AtomGraph Processor [55]
- RESTful-API for RDF data (R4R) [3]
- Restful API Manager Over SPARQL Endpoints (RAMOSE) [21]
- OWL2OAS Converter [44]
- Ontology Based APIs Framework (OBA) [38]
- Community Solid Server [96]
- Walder [47]
- LDflex [97]

3.1.2. Exclusion and inclusion criteria

The standardized exclusion (EC) and inclusion (IC) criteria for scientific literature review was defined as follows:

- EC1: articles not written in English.
- EC2: articles not describing a methodology / method / process for API generation from ontologies / Linked Data / Knowledge Graphs.
- EC3: the full-text of articles does not give details about the methodology / method / process.
- EC4: articles with an extended version that presents more details about the same methodology / method / process.
- EC5: articles referring to semantic annotation of APIs.

- EC6: articles which reuse a methodology / method / process but do not make any changes to it.
- EC7: duplicated articles (when retrieved from the database).
- EC8: articles describing programming APIs for handling RDF.
- IC1: articles including open source code or free-access demo (if a software tool is presented in the article).

The exclusion and inclusion criteria for selection of non-scientific literature, unpublished, or ongoing work were:

- EC9: works not written in English
- EC10: works not describing the methodology / method / process followed to make available Knowledge Graph data represented with ontologies as APIs.
- IC2: works providing the source code or a demo with free access (if software is described or included in a work).

3.1.3. Selection procedure

This process was carried out by one of the authors and was validated by the rest. The validation consisted on several meetings where the authors discussed the findings and resolved any potential differences.

The literature selection was manually performed in three sequential phases described below. It should be noted that the exclusion and inclusion criteria was applied on each phase of our survey.

1. **Phase 1:** screening titles and abstracts that are relevant for our study
2. **Phase 2:** diagonal reading (i.e., reading the introduction and conclusions, and looking for tables or images throughout the study that highlight and provide relevant information) of selected articles from the previous phase.
3. **Phase 3:** full text reading on the remaining articles from the previous phase. As a result, the final set of articles for our survey was retrieved.

Finally, for the selection process of non-scientific literature, unpublished, or ongoing work, we manually applied the specific exclusion and inclusion criteria (EC0, EC10, and IC2) to review the W3C Recommendations, Technical Specifications, and existing efforts suggested by the experts and researchers we contacted.

3.2. Review process

Our search in Scopus retrieved 845 publications [28]. Figure 1a shows the phases of the literature selection process and the number of articles resulting after applying the exclusion and inclusion criteria in each phase. As a result, our literature review process resulted in 13 articles, summarized in Appendix A.

Figure 1b illustrates the process followed to select non-scientific literature, unpublished, or ongoing work. We (the authors of this survey) suggested 7 relevant works to be included and the experts and researchers we contacted suggested an additional 9 works. As a result of this second review, 15 works were selected after applying the exclusion and inclusion criteria (EC9, EC10, and IC2).

While performing Phases 2 and 3 of the literature selection process, we found several articles describing ontology-based applications developed with well-known API libraries for managing RDF such as rdflib,[6] Apache Jena [11], OWL API [49], Sesame API (now RDF4J) [7], or JOPA [62]. We discarded these libraries, according to EC8, as they aim to manage RDF data and ontologies from a specific programming language (Python, Java, etc.). Rather, we focus on Web APIs that allow application developers to directly access data without having to rely on a specific programming language, queries, or transformation of the results obtained from an endpoint. A further comparison on API libraries for managing RDF is presented in [63].

Similarly, we excluded LDflex [97] from our final selection, as despite providing front-end developers with an abstraction to RDF data and SPARQL queries, it is a library for a specific programming language (JavaScript), and hence out of the scope of this manuscript.

4. Approaches for APIs generation

In this section we present our findings in two main categories: 1) specifications, i.e., set of rules and descriptions on how to define and implement APIs, and 2) technologies and tools, i.e., systems that have been developed to implement specifications or provide solutions for KG consumption.

4.1. Specifications

In our study, we found several descriptions of the design and details on how to implement APIs. In the following subsections we begin by presenting a summary of those that have been defined in the Semantic Web community.

4.1.1. SPARQL Protocol

The SPARQL Protocol and RDF Query Language [14] describes the means for conveying SPARQL queries and updates to a SPARQL processing service and returning the results via HTTP to the entity that requested them. It was the first standard to provide access to RDF data. Therefore, most of the projects that had published RDF data use this protocol through a server implementation. The latest version is the SPARQL 1.1 Protocol [31].

4.1.2. SPARQL 1.1 Graph Store Protocol (GSP)

Protocol [74] that describes HTTP operations for managing a collection of RDF graphs from a SPARQL triplestore. To this end, GSP describes a mapping between HTTP methods and SPARQL queries. This protocol can be viewed as a

[6] https://github.com/RDFLib/rdflib

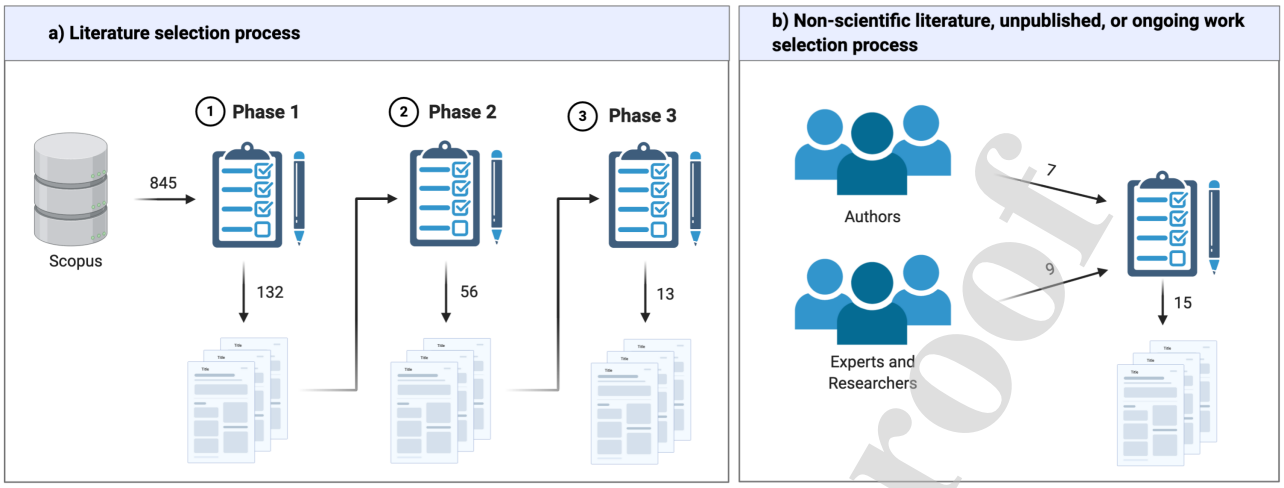

Figure 1: a) Literature selection process includes reviewing the articles from Scopus in three phases: Phase 1) Screening titles and abstracts, Phase 2) Diagonal reading, Phase 3) Full text reading. The exclusion and inclusion criteria defined for scientific literature were applied in each phase; as a result 13 articles were retrieved. b) Non-scientific literature, unpublished, or ongoing work selection process includes reviewing works suggested by the authors of this survey and the experts contacted. After applying the exclusion and inclusion criteria, 15 works were obtained. Summarizing, a total of 28 works resulted from the selection process.

lightweight alternative to the SPARQL 1.1 protocol in combination with the full SPARQL 1.1 Query and SPARQL 1.1 Update languages.

4.1.3. Linked Data API (LDA)

Specification that defines a configurable API layer intended to support the creation of simple RESTful APIs over RDF triplestores [39] . This configuration must be provided by means of an RDF file that follows a specific vocabulary and processing model to describe the SPARQL endpoint, variables, pagination, queries and all the details needed for the API generation.

4.1.4. Hydra Vocabulary

Lightweight vocabulary designed to create hypermedia-driven Web APIs [60]. Hydra defines a set of common concepts to create generic APIs; enabling servers to advertise valid state transitions to a client. Clients can use this information to construct HTTP requests to achieve a goal by modifying the state of the server.

4.1.5. Linked Data Platform (LDP)

Specification that defines a set of rules for HTTP operations on web resources to provide an architecture for read-write Linked Data on the Web [2]. LDP provides details on how to configure HTTP access to manage resources (HTTP resources) and containers (collections of resources). Resources can be RDF sources and non-RDF sources (e.g. binary or text documents). Containers are defined only for RDF resources and they can be Basic, Direct, and Indirect. Basic containers contain triples of arbitrary resources, and must be described by a fixed structure using and a specific vocabulary.[7] Direct containers specialize Basic containers by introducing membership triples which allows the subject and predicate of the triple to be configured using the container definition. Indirect containers are similar to Direct containers but they also are capable of having members whose objects have any URI.

4.1.6. Linked Data Templates (LDT)

Protocol that specifies how to read-write Linked Data based on operations backed by SPARQL 1.1 [54]. LDT defines an ontology with the core concepts and properties required to describe applications. The ontology must be reused to design application ontologies that contain API paths, operations, SPARQL queries, and state change instructions for the desired application. State changes intend to cover the hypermedia definition provided in the REST architecture [33], which states that web resources should specify their next state.

4.1.7. Social Linked Data specification (Solid)

Specification [5] that describes implementation guidelines for servers and client applications to enable decoupling data from services. Solid provides support for a decentralized Web where users can store their personal data on Solid-compliant servers and choose which applications can access such data. Likewise, Solid-compliant applications allow managing any user's data stored on the aforementioned servers. This specification extends the Linked Data Platform to provide a REST API for read and write operations on resources and containers. Solid also provides a WebSocket-based API with a publish/subscribe mechanism to notify clients of changes affecting a given source in real time.

In addition, during the review process we found other specifications defined by the Software Development community and that are relevant for our study since they are reused in solutions proposed by the Semantic Web commu-

[7] https://www.w3.org/ns/ldp#

nity.

4.1.8. OpenAPI Specification (OAS)

Formerly known as the Swagger Specification, OAS [50] defines how to describe REST APIs in a programming language-agnostic interface in order to allow humans and machines to discover and understand the details of a service. OAS has become the choice of reference by many developers due to its community support and the amount of available tools for creating API documentation, server and client generation and testing.

4.1.9. GraphQL

Specification [35] that uses a declarative query language to allow clients accessing the data they need on demand. In GraphQL, queries define the available entry points for querying a GraphQL service. GraphQL has become popular among the developer community as an alternative to REST-based interfaces, as it presents a flexible model rather than a static API. However, developers must be familiar with the schema used to represent the queried data.

4.2. Technologies and Tools

The state of the art describes several technologies and tools for generating APIs to enable KG consumption. In the following subsections we present a brief description of each solution.

4.2.1. KG stores

Several graph databases (e.g. Neo4j [71]) and triple-stores (e.g. Fuseki [1], Blazegraph [92], GraphDB [75]) can be used for KG storage. As a representative example (resulting from our literature review) we include OpenLink Virtuoso [26], a hybrid data store and application server supporting the SPARQL 1.1 Protocol that has been widely used in the Semantic Web community. Virtuoso can be configured as an implementation backend on some of the specifications presented in the previous section, e.g., as a Linked Data Platform client and server.

4.2.2. Pubby

Linked Data compliant server that adds a simple HTML interface and dereferenceable URIs on top of SPARQL endpoints [18]. Thanks to Pubby, users can navigate the contents of an endpoint interactively in their browser, without having to issue any SPARQL queries. Pubby handles content negotiation and includes an extension to describe the provenance of each request made to the server [45].

4.2.3. Puelia

PHP implementation of the Linked Data API [40]. Puelia allows handling incoming requests by reading a configuration file and executing the corresponding SPARQL queries defined in such file. The RDF data retrieved from the SPARQL endpoint is returned to the client in several formats (e.g. Turtle, JSON, etc.).

4.2.4. Epimorphics Linked Data API Implementation (ELDA)

Java implementation of the Linked Data API [25]. ELDA provides a way to create APIs to access RDF data using RESTful URLs; as well as a mechanism to create resource-specific views for browsing these data. As with Puelia, in ELDA all URLs are translated into SPARQL queries to get data from a target SPARQL endpoint.

4.2.5. Linked Open Data Inspector (LODI)

Linked Data server that provides HTML views and content negotiation of resources on a SPARQL endpoint [32]. LODI was inspired by Pubby, but it includes extra functionalities such as more detailed and customizable views for developers, map-based location graphs in case resources contain geospatial attributes, automatic detection and display of image files, and custom configuration for host portal information.

4.2.6. Apache Marmotta

Linked Data server [36] compliant with the SPARQL Protocol 1.1 (providing a SPARQL endpoint). Marmotta was one of the first ools which implemented the Linked Data Platform specification, with support for LDP Basic Containers and content negotiation. Moreover, Marmotta is a Linked Data development environment which includes several modules and libraries for building Linked Data applications.

4.2.7. Building Apis SImpLy (BASIL)

Framework designed for building Web APIs on top of SPARQL endpoints [20]. In BASIL, a set of SPARQL queries and their related endpoints must be defined. In addition, API parameters can be included according to a SPARQL variable naming convention. This convention allows using parameters in configurable templates to parametrize SPARQL as an API. Then, BASIL generates the API paths to retrieve the data and the Swagger specification documentation of the API.

4.2.8. Git repository linked data API constructor (GRLC)

Server implementation that takes SPARQL queries and translates them to Linked Data Web APIs [70]. These queries can be stored in GitHub repositories, local filesystem, or listed as online available URLs into a YAML file. In addition, these queries must include SPARQL decorators[8] (or tags) to add metadata and comments, e.g. to define the specific HTTP method to be executed, the query-specific endpoint, pagination, among others. Then, GRLC takes each query and translates it into one API operation and generates a JSON Swagger-compliant specification and a Swagger-UI to provide the interactive API documentation. In addition, GRLC has recently included a mechanism (provided by SPARQL Transformer [65]) to translate a JSON structure, defined according to specific rules, into a SPARQL query. This mechanism allows transforming SPARQL query results into

[8] https://github.com/CLARIAH/grlc/tree/dev#decorator-syntax

a JSON serialization.

4.2.9. AtomGraph Procesor

Linked Data processor and server for SPARQL endpoints [55] (earlier known as Graphity [53]). AtomGraph uses an ontology for HTTP request matching and response building. This ontology contains Linked Data Templates that map URI templates to the SPARQL queries needed to request matching and response building. The SPARQL queries are included into the application ontology using the SPIN-SPARQL Syntax model.[9]

4.2.10. JSON-QB API

Interface for developers that reuses statistical data stored as RDF Data cubes [103] [90]. JSON-QB only works for data represented with the W3C RDF Data Cube vocabulary, and has evolved into CubiQL,[10] a GraphQL service for querying multidimensional Linked Data Cubes.

4.2.11. Open Semantic Framework (OSF)

Framework designed to create and manage domain specific ontologies; and to maintain, curate and access the stored data [67]. Data access is enabled through a REST API based on prefabricated SPARQL query templates.

4.2.12. Trellis

Linked Data server which supports high scalability, large quantities of data, data redundancy and high server loads [51]. Trellis follows the Linked Data Platform specification for resource management and has several extensions[11] implementing persistence layers and service components e.g. Trellis-Cassandra for distributed storage. Trellis has been included in the Solid Project[12] Test Suite.[13]

4.2.13. Ontology-Based APIs (OBA)

Framework designed to generate an OpenAPI specification from an ontology or ontology network (specified in OWL) [38]. Once a target OpenAPI specification is generated, OBA also provides the means to create a REST API server to handle requests, deliver the resulting data in JSON format (following the ontology structure) and validate the API against an existing KG. OBA automatically generates SPARQL templates for common operations from the source ontology; but also accepts custom queries needed by users. Custom queries are specified following the conventions established by GRLC and Basil.

4.2.14. RESTful-API for RDF data (R4R)

Template-based framework that creates RESTful APIs over SPARQL endpoints using customized queries [3]. R4R is both a server and a working environment: once started, R4R runs a web service that can be updated when new resources are added without having to restart the server. The workspace in R4R defines all available resources to its service, and contains the SPARQL queries and the templates required for managing input queries and resources obtained from a target endpoint.

4.2.15. OWL2OAS

Converter designed for translating OWL ontologies into OpenAPI Specification documents [44]. This tool generates API paths for the concepts of the ontology and their schemas. In addition, OWL2OAS provides JSON-LD context for the aforementioned schemas which is based on the object and data properties defined in the ontology.

4.2.16. Ontology2GraphQL

Web application that generates a GraphQL schema and its corresponding GraphQL service from a given RDF Ontology [30]. To this end, the ontology for data representation must be manually annotated with a GraphQL Metamodel (GQL), which includes several classes representing the GraphQL types that compose a GraphQL schema (e.g. object, list, enumeration, among others). Therefore, each ontology class is mapped to an instance of GQL Object class. Object and datatype properties are defined as instances of GQL ObjectField and ScalarField classes respectively. Finally, there are several GQL properties required to specify more details on properties, for example, the GQL hasModifier property can be used to define that an object property will manage an array of the elements.

4.2.17. Restful API Manager Over SPARQL Endpoints (RAMOSE)

Framework designed to create REST APIs over SPARQL endpoints through the creation of textual configuration files [21]. Such files enable querying SPARQL endpoints via Web RESTful API calls that return either JSON or CSV-formatted data. To provide this configuration, a hash-format syntax[14] based on a simplified version of Markdown is required.

4.2.18. Community Solid Server

Server implementation of the Solid specifications [96]. It aims to provide support for data pods, which allows storing personal data in an accessible manner. Solid makes it possible to decouple personal data storage from services, and therefore users are free to decide which applications can access to their pods. As a result, users can keep total control of their data.

4.2.19. Walder

Framework that allows configuring a website or Web API on top of Knowledge Graphs (e.g. SPARQL endpoint, Solid pod, etc.) [47]. To this end, users must define a configuration file with the details of the data source, paths, operations, etc. allowed for the API. Walder reuses the Comunica framework [93], more precisely the graphql-ld-comunica en-

[9] https://spinrdf.org/sp.html
[10] https://github.com/Swirrl/cubiql
[11] https://github.com/trellis-ldp/trellis-extensions
[12] https://solidproject.org/
[13] https://github.com/solid/test-suite
[14] https://github.com/opencitations/ramose#Hashformat-configuration-file

gine,[15] to execute the queries needed to get the required data. Walder uses GraphQL-LD [94], a query language which allows extending GraphQL queries with a JSON-LD context. Comunica then takes the GraphQL queries and translates them, based on the JSON-LD context, into SPARQL queries to retrieve the desired data.

5. Analysis of specifications, technologies and tools for API definition and generation

In this section we introduce the frameworks designed to perform a systematic comparison of the specifications, technologies and tools described in Section 4. We also discuss the results obtained when applying these frameworks to compare the specifications, technologies and tools considered in this survey.

5.1. Criteria for comparing API specifications

Table 1 summarizes the set of criteria defined in the framework to compare the existing specifications. These criteria highlight relevant information to help us to answer the research questions outlined in Section 2.4 and to describe the research challenges discussed in Section 6. We are interested in the *year* when specifications were created in order to understand their evolution over time. We want to know if specifications are officially recognized by an authority (i.e., whether they are a *standard* or not) or if they have just been adopted by a community without going through a standardization process. We also consider relevant the *endpoints* supported by specifications, since this allows detecting the different KG data sources (e.g. RDF data dump, SPARQL endpoint, among others). We also consider *configuration formats*, as they give us an idea of details needed to implement a target specification.

In addition, we evaluate if specifications support *configurable queries*, which indicate the degree of freedom offered by a specification to manage specific data needs of an application. We analyze the file formats (*media types*) that developers must expect when using such specification, and whether specifications consider developer-friendly formats or not. We assess the *operations* for resource management provided by a specification, i.e., create, read, update, delete support (CRUD[16]). We also take into account the *authentication* techniques proposed by the specifications, since it determines how the specification manages the access to the API methods.

We consider the *versioning* support to understand whether a specification manages KGs that evolve over time, both in terms of their contents and schema (ontology). We analyze the *status codes*, since they provide information of the methods suggested by the specification in order to properly provide details to clients about the execution of API calls. We examine the *resources* supported by the specification, as it allows us to understand if the specification manages single resource, a collection of resources, or nested resources (i.e., if a resource contains a subcollection of resources). Finally, *reference* provides information about the origin of the specification details provided in this comparison for provenance purposes.

Table 1
Comparison criteria for specifications

Criteria	Description
Year	Year the specification was made available
Standard	Specification recognition type
Endpoint	Type of point where resources are available and requests are submitted
Configuration format	The file format to be provided with the details of the specification
Configurable queries	Support for customized queries
Media types	File formats supported for data interchange
Operations	Allowed methods for managing resources
Authentication	Allowed methods to verify the identity of a user or process
Versioning	Support for version management
Status codes	Type of response messages to a client's request
Resources	Type of resources allowed such as single, collection, or nested resources
Reference	The provenance of the specification details.

5.2. API specification comparison

Table 2 shows the comparison between the specifications according to each criterion defined in subsection 5.1. The symbol "-" indicates that the criterion is not described or detailed in the specification source.

Specifications began to appear in the *year* 2008, when the SPARQL protocol 1.0 [95] was defined; and span until 2019, when the latest Solid specification draft has been made available. Three of the analyzed specifications are W3C Recommendations (SPARQL 1.1 Protocol, Graph Store Protocol, and Linked Data Platform) and three are not recommendations but were defined in W3C Community groups: Linked Data Templates by the Declarative Linked Data Apps group,[17] Hydra by the Hydra group,[18] and Solid by the Solid Community group.[19] OpenAPI and GraphQL are now considered *de facto* specifications, as despite not being officially recognized by a standardization body, they are widely used by the developer community [80].

[15] https://github.com/rubensworks/graphql-ld-comunica.js

[16] In this work, CRUD is understood as the HTTP POST, GET, PUT, and DELETE operations.

[17] https://www.w3.org/community/declarative-apps/

[18] https://www.w3.org/community/hydra/

[19] https://www.w3.org/community/solid/

Table 2
Analysis of API Specifications according to our comparison framework.

Specification	Year	Standard	Endpoint	Configuration format	Configurable queries	Media types	Operations	Authentication	Versioning	Status codes	Resources	Reference
SPARQL Protocol	2008	W3C Recommendation	SPARQL Endpoint	-	Yes	RDF formats and XML, JSON, or CSV/TSV	GET and POST	HTTP authentication	-	HTTP Status codes [RFC2616 specification]	Single SPARQL URL endpoint	https://www.w3.org/TR/sparql11-protocol
Linked Data API	2010	De facto	SPARQL Endpoint	RDF format	Yes	HTML, Turtle, JSON, RDF+XML, XML and XSLT	GET	-	-	-	Single, multiple	https://github.com/UKGovLD/linked-data-api
OpenAPI Specification	2011	De facto	Any	JSON and YAML	Yes	Any in compliance with RFC6838 specification (the most common is JSON)	CRUD, HEAD, OPTIONS, PATCH, and TRACE	HTTP authentication, OAuth2, API key, and OpenID Connect Discovery	Yes	HTTP Status Codes	Single, multiple, nested	https://github.com/OAI/OpenAPI-Specification
SPARQL 1.1 Graph Store Protocol	2013	W3C Recommendation	SPARQL Endpoint	-	Yes	Turtle, RDF/XML or N-Triples	CRUD	HTTP authentication	-	HTTP Status codes [RFC2616 specification]	Single SPARQL URL endpoint	https://www.w3.org/TR/sparql11-http-rdf-update
Hydra	2013	No	-	JSON-LD	-	JSON-LD	CRUD	-	-	HTTP Status Codes	Single, multiple	http://www.hydra-cg.com/spec/latest/core
Linked Data Platform	2015	W3C Recommendation	Linked Data server	-	Yes	RDF (Turtle is required) and non-RDF formats like HTML and JSON	CRUD, HEAD, OPTIONS, and PATCH	HTTP authentication	-	HTTP Status Codes	Single, multiple, nested	https://www.w3.org/TR/ldp
GRAPHQL Spec	2015	De facto	Any	GRAPHQL Schema	Yes	Any serialization format (the most common is JSON)	Query, mutation, and subscription	-	Not needed	-	Single URL endpoint	https://spec.graphql.org/June2018
Linked Data Templates	2016	No	SPARQL Endpoint	RDF format	Yes	RDF format	CRUD	-	-	HTTP Status Codes	Single, multiple	https://atomgraph.github.io/Linked-Data-Templates
Solid	2019	No	RDF data	-	Yes	Any in compliance with IANA media types	CRUD, HEAD, OPTIONS, and PATCH	TLS connections, HTTP/1.1 Authentication, and Web Access Control	-	HTTP Status Codes	Single, multiple, nested	https://github.com/solid/specification

Most of the analyzed specifications from the Semantic Web community support SPARQL *endpoints*. Solid goes one step further, aiming to support any RDF data source such as RDF data dumps and Linked Data documents in addition to SPARQL endpoints. OpenAPI and GraphQL allows specifying any given endpoint, leaving to the implementations the support for query languages.

Specification settings are usually provided in different *configuration formats*. LDA and LDT must be configured in an RDF file which contains the URI templates required for the API and the required SPARQL queries. OpenAPI and GraphQL are configured in JSON, a developer-friendly format [86], [82], but OpenAPI also supports YAML. In summary, the OpenAPI configuration file must contain the schemas and the API paths that will be implemented, and the GraphQL configuration must define the GraphQL schemas which describes the data sources and the GraphQL queries that define the available entry points for querying a GraphQL service. Finally, Hydra requires a JSON-LD [57] configuration file. Such format aims to represent Linked Data as JSON with minimal changes, thus it intends to be an alternative for developers interested in semantic data. The Hydra configuration file must contain several details of the API such as the URL of the endpoint, supported schemas, allowed operations, among others.

All specifications, depending on the operations and resources supported, allow *configurable queries*. For LDA, LDP, and LDT, such queries must be written in the SPARQL query language. GraphQL requires queries written in the GraphQL query language; and OpenAPI supports multiple languages, e.g. SQL, since it doesn't restrict the type of endpoint. As for query execution, the specifications provide data results in several *media types*. SPARQL 1.1 Graph Store Protocol, Hydra, and LDT specifications only provide results in RDF formats (e.g. Turtle, JSON-LD, etc.), while the remaining specifications also support non-RDF formats (e.g. HTML, CSV, JSON, among others).

Regarding allowed *operations*, the most limited specification is Linked Data API since it only supports reading data (GET). The SPARQL Protocol initially supported GET and POST operations, but after the SPARQL 1.1 Graph Store Protocol was introduced, its support was expanded to full CRUD. Hydra and Linked Data Templates support the configuration of CRUD methods, while Linked Data Platform, Solid and OpenAPI, in addition to CRUD, also support HEAD (to ask for information about resources), OPTIONS (to describe the communication options of a resource), and PATCH (to partially update resources) methods. Moreover, OpenAPI supports the TRACE method, which allows to follow the path that a HTTP request follows to the server and it is generally used for diagnostic purposes. GraphQL supports operations with different names but that may be equated to HTTP methods: query implements a GET, mutation implements a POST, and subscription implements a PUT, PATCH, or DELETE.

Most specifications support HTTP *authentication*. Solid also allows TLS connections for data pods through the https URI scheme, HTTP/1.1 authentication and it must conform to the Web Access Control specification. Solid clients must support HTTP/1.1 Authentication. OpenAPI allows configuring other authentication mechanisms like API key, OAuth2, among others. In order to control the changes in the API, only OpenAPI provides an specific attribute to define the *versioning*, by following a Semantic Versioning 2.0.0[20] convention. In GraphQL it is not needed to specify versions since the specification strongly encourages the provision of tools to allow for the evolution of APIs. To this end, GraphQL tools must allow API providers to specify that a given type, field, or input field has been deprecated and will disappear at some point in time; thus they must notify clients by, for example, message responses detailing on changes. Therefore, GraphQL systems may allow for the execution of requests which at some point were known to be free of any validation errors, and have not changed since. The remaining specifications leave versioning up to the implementations.

The OpenAPI, LDP, and Solid specifications support single, collection, and nested *resources* since they led implementers to freely define them. LDA, Hydra and LDT support single and collection of resources only. In the case of the SPARQL Protocol and SPARQL 1.1 Graph Store Protocol both support a single URL referring to the SPARQL endpoint. In a similar manner, GraphQL only requires a single URL. Finally, almost all specifications support reusing the *status codes* defined by the HTTP protocol.[21] Therefore, implementers may provide relevant messages when dealing with clients such as successful requests (2xx), bad requests (4xx), etc. LDA and GraphQL do not provide details about response messages; however, as technologies that implement such specifications are served over HTTP they may reuse HTTP status codes.

5.3. Criteria for comparing API generation technologies and tools for KG consumption

Table 3 describes the criteria proposed in the framework to compare API generation technologies and tools. Since some of these criteria are the same as those defined for comparing specifications in subsection 5.1, we present below only the new criteria that we included to compare technologies and tools.

The first new criterion considered is the *Interface Description Language* which outlines which convention is followed by a technology or tool to define APIs. We also assess what is the *input* required by the technology or tool for generating APIs, e.g., an ontology, queries, etc; and the expected result (*output*) after executing a given technology or tool (e.g., data formats, API specification file, a server, etc.). In addition, we analyze whether technologies or tools provide *control over the JSON structure* as it helps us to detect which ones allow users to manage such files. As for the *source* is only an informative column that indicates where the technology or tool code, demo, or repository is available

[20] https://semver.org/

[21] https://www.ietf.org/assignments/http-status-codes/http-status-codes.xml

Table 3
Comparison criteria for technologies and tools

Criteria	Definition
Year	Year when the technology or tool was made available
Interface Description Language	Specification to document functional and non-functional aspects of the API
Input	Files needed for the API generation
Output	Result of the technology or tool execution
Operations	Allowed methods for managing resources
Configuration format	The file format to be provided with the details of the technology or tool
Configurable queries	Support for customized queries
Authentication	Allowed methods to verify the identity of a user or process
Resources	Type of resources allowed such as single, collection, or nested resources
Versioning	Support for version management
Control over the JSON structure	Support for JSON management
Source	The source code / repository of the technology or tool
Last release	Year when the last version of the technology or tool was released
Language	Programming language used for create the technology or tool

for a technology or tool. Moreover, we are interested in the *last release date* when a technology or tool was updated in order to know whether it is still maintained. Finally, we also are interested in the *language* selected for the development of the technology or tool as it may help us to understand if there is a preferred option to implement them.

5.4. Results of comparison of API generation technologies and tools for KG consumption

Table 4 presents the comparison between the technologies and tools according to the criteria described in subsection 5.3. The symbol "-" indicates that the criterion is not described or detailed in the technology or tool source.

One of the first tool to appear was OpenLink Virtuoso, in the *year* 2008, which became a popular technology to store and manage RDF data. Since then, several alternatives appeared over the years to ease data consumption by providing interfaces based on the REST paradigm and taking advantage of the HTTP protocol. The most recent tool reported in our survey is Walder, released in 2020, which allows generating APIs for consuming RDF data from several sources, in an effort to integrate decentralized endpoints.

Most of the assessed technologies and tools use as *Interface Description Languages* (IDL) the specifications presented in subsection 4.1. However, some of them support other API description blueprints. For example, JSON-QB API requires users to define the API following an ad-hoc specification (JSON-qb API specification). R4R allows users to manually describe the API but does not restrict the use of a specific IDL (e.g. users can provide an OpenAPI-compliant file). RAMOSE requires users to define a hash-format configuration file that contains the details of the API. Pubby and LODI do not require any IDL as they only provide HTML views of resources.

In general, all included technologies and tools require as *input* the URL of the SPARQL endpoint and the SPARQL queries needed to implement the allowed API methods, but some technologies and tools differ slightly in their needs. The R4R framework, in addition to the aforementioned inputs, requires users to define JSON templates that allow the responses of SPARQL queries or requested resources to be translated into JSON. The AtomGraph Processor requires an application ontology that must follow the structure described in the Linked Data Templates protocol. Such ontology must define the API details, the SPARQL queries to request matching, and the application behavior. The JSON-QB-API technology only requires the URL of the SPARQL endpoint, as SPARQL queries are generic (designed to handle data described with the Data Cube vocabulary) and provided automatically. Apache Marmotta and Trellis have a similar input configuration as both only require the RDF source (e.g. an RDF data dump).

Other tools start from OWL ontologies. For example, OBA and OWL2OAS require an OWL ontology as input, which they convert to an OpenAPI specification. OBA also accepts the URL of a target SPARQL endpoint, as it generates a server to handle the client requests. OBA allows users to specify which classes should be excluded in the final API by using a YAML configuration file, while OWL2OAS requires the ontology to be annotated with an ad-hoc Boolean property to define which classes or properties should be taken into account or not in the API. The Ontology2GraphQL application needs an ontology annotated with the GraphQL meta model; this ontology must be stored in a Virtuoso instance which must also contain the RDF data. Finally, Walder supports any RDF source (RDF dump, Linked Data documents, SPARQL endpoint) as input, as well as the GraphQL-LD queries and the specific JSON-LD contexts required for the execution of API operations.

Table 4: Analysis of API generation technologies and tools for KG consumption according to our comparison framework.

Technology or Tool	Year	Interface Description Language	Input	Output	Operations	Configurable queries	Configuration	Authentication	Resources	Versioning	Control over JSON	Source	Last release	Language
OpenLink Virtuoso	2008	SPARQL Protocol and LDP	SPARQL Queries	RDF or non-RDF	CRUD	Yes	LDP must be activated in the WebDAV content section	OAUth, WebID and Digest Authentication via SQL Accounts.	Single	-	-	http://vos.openlinksw.com/owiki/wiki/VOS/VirtLDP	2018	C
Pubby	2008	None	SPARQL Endpoint	HTML, Turtle and RDF/XML	GET	No	Turtle file	-	Single	-	-	https://github.com/cygri/pubby	2011	Java
Puelia	2010	LDA	SPARQL Endpoint and SPARQL Queries	Turtle, RDF/XML, JSON and XML	GET	Yes	RDF/Turtle files	-	Single, multiple, nested	-	-	https://code.google.com/archive/p/puelia-php/	2010	PHP
ELDA	2011	LDA	SPARQL Endpoint and SPARQL Queries	HTML, XML, JSON, RDF/XML, and Turtle	GET	Yes	RDF format, typically writen in Turttle	-	Single, multiple, nested	-	-	https://github.com/epimorphics/elda	2018	Java
Apache Marmotta	2013	LDP	RDF source	RDF and non-RDF	CRUD	No	LDP module must be included in the pom.xml file	Custom authentication and authorization mechanism	Single, multiple	Yes	-	http://marmotta.apache.org/platform/ldp-module.html	2018	Java
BASIL	2015	Swagger	SPARQL Endpoint and SPARQL Queries	Swagger-compliant API, data results in XML, JSON, CSV and RDF	GET and POST	Yes	A file with the connection parameters to the database	HTTP basic authentication	Single, multiple	-	-	https://github.com/the-open-university/basil	2021	Java
GRLC	2016	Swagger	SPARQL Endpoint and SPARQL Queries	JSON Swagger-compliant specification, Swagger-UI, data results in CSV, JSON, Turtle, and HTML	GET and POST	Yes	List of queries in .rq format or as URLs into a YAML file	User and password to the SPARQL endpoint (if required), and an access token to communicate with GitHub API.	Single, multiple	-	Yes	https://github.com/CLARIAH/grlc	2020	Python
AtomGraph Processor	2016	LDT	Application ontology	RDF serializations	CRUD	Yes	A Turtle file with the application ontology	HTTP basic authentication	Single	-	-	https://github.com/AtomGraph/Processor	2021	Java
LODI	2017	None	SPARQL Endpoint and SPARQL queries	HTML and N3	GET	No	Turtle file	-	Single	-	-	https://github.com/marfersel/LODI	2018	Node.js
JSON-QB API	2017	JSON-qb API specification	SPARQL Endpoint	JSON	GET	No	-	-	Single, multiple	Yes	-	https://github.com/OpenGovIntelligence/json-qb-api-implementation	-	Java
OSF	2017	-	ontology and SPARQL Endpoint	JSON	CRUD	Yes	-	-	-	-	-	https://github.com/structureddynamics/OSF-Web-Services	2017	PHP

Table 4: Analysis of API generation technologies and tools for KG consumption according to our comparison framework.

Technology or Tool	Year	Interface Description Language	Input	Output	Operations	Configurable queries	Configuration	Authentication	Resources	Versioning	Control over JSON	Source	Last release	Language
Trellis	2017	LDP	RDF source	RDF serializations and HTML	CRUD, PATCH, HEAD and OPTIONS	No	A YAML file with the application configuration	Basic and token-based	Single, multiple, nested	Yes	-	https://github.com/trellis-ldp/trellis	2021	Java
Ontology2 GraphQL	2019	GRAPHQL schema	Ontology annotated with the GQL meta model	GraphQL schema and a ready-to-deploy GraphQL service implementation	GET	Yes	RDF data and the annotated ontology stored in the same Virtuoso triplestore	-	Single API endpoint	-	Yes	https://github.com/genesis-upc/Ontology2GraphQL	-	Java
R4R	2019	Static HTML manually generated	SPARQL Endpoint, SPARQL Queries, and JSON templates	Swagger-compliant RESTful API, and requested data in JSON	GET	Yes	The resources configured in the workspace containing the SPARQL queries and JSON (Velocity) templates	HTTP basic authentication	Single, multiple, nested	-	Yes	https://github.com/oeg-upm/r4r	2020	Java
OBA	2020	OpenAPI	OWL ontology and SPARQL Endpoint	YAML OpenAPI-compliant specification, SPARQL templates, a server, and requested data in JSON	CRUD	Yes	YAML file	OAUth2.0	Single, multiple, nested	Yes	-	https://github.com/KnowledgeCaptureAndDiscovery/OBA	2021	Java
OWL2OAS	2020	OpenAPI	OWL ontology	OpenAPI specification in YAML or JSON	GET	-	Not required	-	Single	Yes	-	https://github.com/RealEstateCore/OWL2OAS	2020	C#
RAMOSE	2020	A hash-format file	SPARQL Endpoint and SPARQL queries	HTML documentation of the API, a dashboard for the API monitoring, and data requested in CSV or JSON	GET and POST	Yes	Hash-format file	-	Single, multiple	Yes	Yes	https://github.com/opencitations/ramose	2021	Python
Community Solid Server	2020	Solid	-	-	CRUD, PATCH, HEAD and OPTIONS	-	-	-	-	-	-	https://github.com/solid/community-server	2021	Typescript
Walder	2020	OpenAPI	RDF source, and the GraphQL-LD queries + JSON-LD context	Data requested in RDF serializations and HTML	GET	Yes	YAML OpenAPI-compliant file with Walder-specific extensions	-	Single, multiple, nested	Yes	Yes	https://github.com/KNowledgeOnWebScale/walder	2020	JavaScript

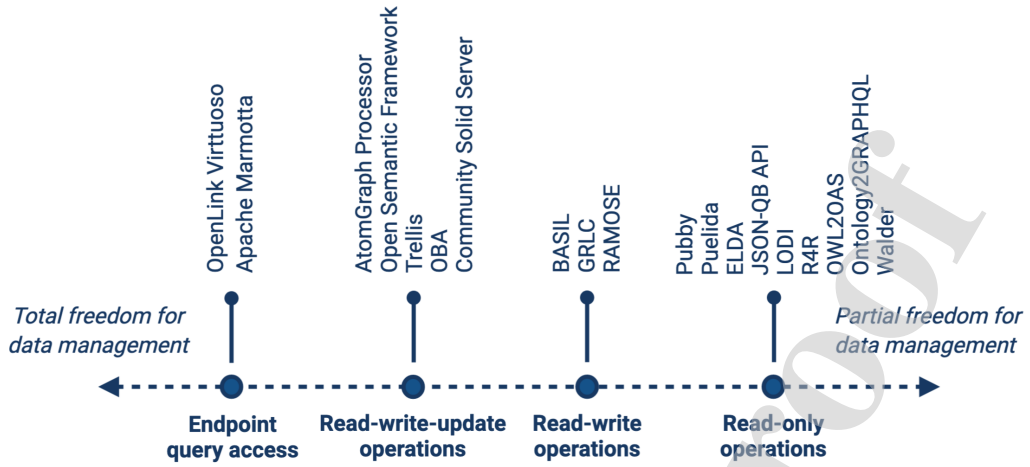

Figure 2: Data management levels offered by API generation technologies and tools for KG consumption.

When executing the analyzed technologies and tools, different types of *outputs* are generated. Virtuoso, Pubby, Puelia, ELDA, Apache Marmotta, AtomGraph Processor, LODI, Trellis, and Walder provide data results in RDF and non-RDF serializations. Tools such as BASIL, GRLC, OWL2OAS, and OBA generate OpenAPI-compliant APIs, and provide the requested data in JSON format. BASIL and GRLC also provide data in other formats such as CSV, XML, and RDF. Likewise, JSON is the output format for JSON-QB-API, R4R, and OSF results. OBA and OWL2OAS generate the API schemas and paths from an OWL ontology, but OBA is the only tool which generates, without human intervention, the basic SPARQL query templates needed to handle the KG data when executing the API methods. Ontology2GraphQL translates the annotated ontology into a GraphQL schema and provides a GraphQL service implementation; however, the annotation process must be performed manually. Finally, RAMOSE generates an HTML with the API documentation, a dashboard for monitoring the API, and provides data results as CSV or JSON files.

The evaluated technologies and tools also allow different *operations* to be carried out. Figure 2 shows the data management levels that such technologies and tools offer on a scale from full query access to a read only query level. Data management levels depend on the allowed operations, thus technologies and tools that support read-only operations provide less freedom than those that allow query execution at the endpoint level. Trellis and the Solid Community Server also allow HEAD (to ask for information about resources), OPTIONS (to describe the communication options of a resource), and PATCH (to partially update resources) operations.

Few technologies and tools provide details on how to manage user *authentication*. This depends on the allowed operations, since, in general, reading operations do not need to authenticate users. Virtuoso and Marmotta allow complete freedom for data management, and thus provide more details on their authentication methods (OAuth for Virtuoso, ad-hoc authentication and authorization mechanisms for Marmotta[22]). Among those technologies and tools allowing writing operations such as update or delete, some of them provide support for authentication mechanisms. For example, basic HTTP authentication is supported by AtomGraph Processor, Trellis, BASIL, and R4R; whereas GRLC requires an access token to communicate with the GitHub API, and the user and password of the SPARQL endpoint, if required. OBA supports OAuth2.0 by default, but authentication can be extended to other methods (which need to be configured by hand).

As for *configurable queries*, almost all technologies and tools define their own mechanisms to allow users defining custom queries. For example, Basil, GRLC, and OBA use ad-hoc decorators in queries to parametrize them and align them to their exposed APIs; while Ontology2GraphQL and Walder accept GraphQL queries. The analyzed technologies and tools also use different *configuration formats*. RDF is the most common choice, but Trellis, OBA, and Walder use YAML configuration files. Technologies like Marmotta and Virtuoso, which support LDP, require to activate the LDP mode by providing specific configuration settings applied to a java file (Project Object Model file) and to its configuration utility (Conductor) respectively. BASIL requires a configuration file (.ini) with connection parameters to the database that must be configured with some required database queries together with a MySQL server. R4R requires to configure the SPARQL queries (.sparql) and JSON templates (.json.vm) both stored into the specific directory that will be taken as the source for the resource path generation. GRLC requires to specify a collection of SPARQL queries (.rq files) into a GitHub repository, but it also allows users to provide such queries as a YAML file containing a list of URLs of SPARQL queries online available. RAMOSE requires a hash-format (.hf) file described according to a simplified version of Markdown syntax.

All the assessed technologies and tools manage single

[22] https://marmotta.apache.org/platform/security-module.html

resources. Marmotta, BASIL, GRLC, JSON-QB-API, and RAMOSE also provide collection of resources Besides single and collection, Puelia, ELDA, Trellis, OBA, R4R, and Walder provide nested resources. Therefore, these last tools allow defining more specific paths for data consumption.

As for *versioning*, tools like JSON-QB-API, OBA, R4R, OWL2OAS, and RAMOSE allow users to specify the API version in the API documentation, but they do not implement control over different versions. In contrast, Apache Marmotta and Trellis manage data versioning through the Memento protocol[23] a variant on content negotiation which enables accessing a resource version that existed around a specific datetime. Ontology2GraphQL and Walder assume that the server must manage data versioning, and hence do not support versioning.

Only five tools provide *control over the JSON structure*. GRLC allows developers to pose queries as a JSON object for specifying what data will be retrieved from the endpoint and what shape the results should follow. R4R allows configuring JSON templates that map the SPARQL query results to compose the desired JSON output. RAMOSE also allows users transforming each key-value pair of the final JSON result according to the rule specified in the call URL. Such transformations rules can be used to convert the output into an array or into a JSON object. Lastly, since Ontology2GraphQL and Walder use GraphQL, both allow managing JSON according to the developer needs. This gives more flexibility to developers issuing queries to KGs, but at the same time forces them to be familiar with the ontology used to represent the information in detail.

Most of the analyzed technologies and tools show changes over their *last release* compared to when they first were made available. Early technologies and tools like Virtuoso, ELDA, and Marmotta have evolved over time in contrast to Puelia, which shows no change. As for Ontology2GraphQL and JSON-QB API, they do not have any release in their source repositories. Since the latest changes observed in the repositories of both tools date from the same year in which they were made available, they may not be currently maintained. Most recent tools have recent releases, which may mean that they are evolving as people begin to use them and new requirements and enhancements are implemented. Finally, regarding the programming *languages* for the development of technologies and tools, Java is the preferred option (used by 10 implementations), followed by PHP, and Python (each selected by 2 implementations), and lastly C, C#, JavaScript, Node.js, and TypeScript (each chosen by 1 implemetation).

5.5. Specification, technology and tool evolution over the years

The rationale for the appearance and evolution of the specifications, technologies and tools included in our survey can be better understood by looking at them in chronological order. Figure 3 shows a timeline illustrating existing specifications, and technologies and tools over the years. SPARQL endpoints were the first and the most common means to provide access to data represented with ontologies. SPARQL endpoints offer access to RDF data using the SPARQL Protocol and RDF Query Language, which was officially standardized in 2008. Thanks to the SPARQL 1.1 Graph Store Protocol (which became a W3C recommendation in 2013), many SPARQL endpoints also provide update and fetch of RDF data via mechanisms of the HTTP protocol. Today, hundreds of SPARQL endpoints have been made available on the web to expose over one thousand public datasets [68].

For illustration purposes, let us assume that we have a KG with local business census data, accessible through a SPARQL endpoint. Let us also assume that we want to retrieve data of the business `"CortField"` which has the identifier `"CortFieldID"`. With a SPARQL endpoint, developers have to issue SPARQL queries to obtain data they need, like the query provided in Appendix B.1 to get data of the business `"CortField"`. As a result of the SPARQL query execution, data will be obtained in a RDF serialization.

In order to ease access and navigation over Semantic Web resources in SPARQL endpoints, a new generation of technologies and tools emerged to provide HTML access to RDF data by dereferencing URI resources. The first and most popular technology providing such features was Pubby, released in 2008, and the latest technology was LODI, launched in 2017. In our example, Pubby or LODI can be executed on top of the SPARQL endpoint so developers can resolve the URI of `"CortField"` without having to issue a SPARQL query, e.g. by browsing `"http://example.org/resource/LocalComercial/CortFieldID"` in a browser.

Several efforts followed by taking advantage of the REST principles to provide developers with a well-known interface for RDF data consumption. The Linked Data API (LDA) specification was proposed in 2010 to define read-only RESTful APIs over RDF triplestores. The most popular tools implementing LDA are Puelia and ELDA, released in 2010 and 2011 respectively. Thanks to these tools, developers can configure API paths to be translated into SPARQL queries that select resources or define views with the specific resource attributes they need. For example, to get data of the local business `"CortField"` developers may issue a request to `"http://example.org/doc/localbusiness/CortFieldID"`, which will trigger a query similar to the one specified in Appendix B.1 and return the corresponding results.

In 2013, Hydra was defined as a vocabulary to combine REST with Linked Data principles focused on describing APIs using JSON-LD. Two years later, the Semantic Web community proposed the Linked Data Platform (LDP) specification to address the read-only limitations of the Linked Data API specification. LDP became a W3C recommendation in 2015, defining a protocol for full read-write Linked Data. Several technologies and tools included support for LDP like Virtuoso, Apache Marmotta,[24] or Trellis; which were released between 2008 and 2017. In our example, local business data could be handled in Apache Marmotta and or-

[23] `https://tools.ietf.org/html/rfc7089`

[24] Marmotta was released before 2015, but in this study we mention the year when the first version (3.3.0) compliant with the LDP specification was launched.

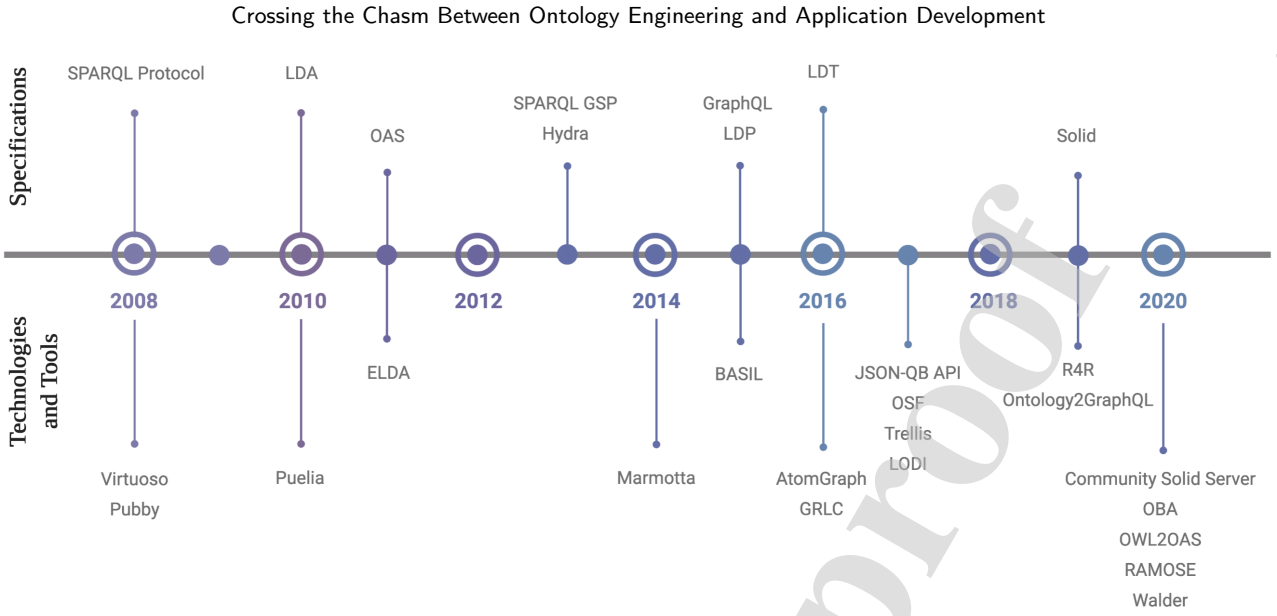

Figure 3: API specifications and API generation technologies and tools timeline. Above the line, specifications are listed according to the year they appeared. Below the timeline, technologies and tools are listed according to the year when they were released.

ganized into a Basic Container (e.g. `"http://example.org/ldp/localbusinesses"` container). Therefore, developers may retrieve items from this container by invoking the get method and the API path of the desired item. For instance, developers can access item `"CortField"` by requesting the API path `"http:// example.org/ldp/localbusinesses/CortField"` in the desired RDF serialization.

Another relevant REST-based approach is the Linked Data Templates (LDT) protocol, presented in 2016. LDT allows users to read-write RDF data based on details that must be specified in an application ontology. Unlike the LDP specification, LDT allows users to define the next state of resources needed for the desired application. This protocol has been implemented in 2016 by the AtomGraph Processor technology. Going back to our example, ontology engineers would have to configure AtomGraph's application ontology with the details of the desired resource, for instance, the local business item template (`ldt:Template`) including the API path (`ldt:match "/localbusiness/{id}"`), and the SPARQL query (`ldt:Query`) to perform the supported operations (e.g. `"get"`). Developers would request the method and path `"GET /localbusiness/CortFieldID"` and retrieve data of local business `"CortField"` in RDF.

The next generation of technologies and tools relied on interfaces to make it easier for non-Semantic Web developers to interact with KGs in their "native" languages (JSON and Interface Description Languages). To this end, some of these technologies and tools reused the OpenAPI specification, released in 2011, due to its wide adoption by application developers. Most of the initial efforts focused on providing support for GET, but some of them have evolved into partial or full CRUD. In this regard, the first effort providing Swagger-compliant APIs was BASIL, released in 2015, followed by tools such as GRLC, OWL2OAS, and OBA that were introduced in subsequent years. It is worth mentioning that, from 2017 to 2020, tools like JSON-QB API, R4R, and RAMOSE have also been proposed to generate developer-friendly APIs, but they follow other specifications to define them.

To illustrate these efforts, let us consider we use GRLC with a GitHub repository where we define and store the SPARQL queries needed for data consumption. As a result of executing GRLC, it generates an API path for each query and a JSON Swagger-compliant specification. The path structure conforms to the GitHub repository structure. For instance, if the query file to select data of local businesses is named `"localbusinesses.rq"` (this example query is provided in Appendix B.3), stored in the repository `"examplerepository"` of the `"GitHubUser"` account, then the corresponding API path would be `"http://api/GitHubUser/examplerepository/localbusinesses"`, where `api` corresponds to the service where GRLC runs. By requesting this API path developers will get local business data in formats supported by the SPARQL endpoint. For example, results can be retrieved in JSON, but this resulting format includes irrelevant metadata that conforms with the query structure (e.g. the header metadata which contains the list of fields of the query results) rather than just providing data according to the structure of the ontology that describes them. To get results into a friendly JSON format users can provide queries in JSON using SPARQL Transformer [65].

A new generation of technologies and tools was developed in parallel to these efforts after the GraphQL Specification (originally developed at Facebook in 2012), was released openly in 2015. GraphQL proposed a flexible way to define APIs under the principle that what you need is exactly what you get, and has been adopted in efforts like Ontology2GraphQL and Walder, released in 2019 and 2020 respectively. Unlike Ontology2GraphQL, Walder requires defining an API by reusing the OpenAPI spec and to spec-

ify the necessary queries in GraphQL plus a JSON-LD context. For instance consider we use Walder to consume the local business census data. Ontology engineers would need to configure an OAS file which includes the URL of the datasource (e.g. our SPARQL endpoint), API paths (e.g. `"/localbusiness/{value}"`), required parameters (e.g. `"value"` which allows providing the identifier of the specific local business to be requested), the allowed operations (e.g. `"get"`); and the GraphQL queries and JSON-LD context required to implement the operations (an example query and JSON-LD context are provided in Appendix B.2). By doing so, developers may request, for example, `"/localbusiness/CortFieldID"` and obtain data of the business `"CortField"` as HTML, RDF, or JSON-LD.

More recently, other approaches are beginning to emerge with the aim of exploiting the knowledge contained in ontologies (those used to describe KGs data) and facilitate the work of developers. The goal is to generate specifications and APIs from ontologies, with minimal human intervention. The most representative solution from this is OBA, released for the first time in 2020. In our example, users need to provide OBA with the local business census ontology[25] and the YAML configuration file which contains the URL of the SPARQL endpoint, the list of classes to be included in the API (e.g. local business represented by the `"LocalComercial"` class), and the allowed methods (e.g. `"get"`). After executing OBA, developers would get the OAS document with the schemas and API paths, the SPARQL queries for implementing the methods, and a server. As a result, developers may request, for instance, `"/localescomercialles/CortFieldID"` and get back data of local business `"CortField"` in JSON format which follows the ontology structure.

The last generation of technologies and tools is focused on providing APIs to ease decentralizing the Web. To this end, users require to handle their personal data in servers under their control, applications require consuming data from several RDF sources (data dump, SPARQL endpoints, etc.), dealing with different authorization mechanisms, among others. The Solid specification appeared in 2019, and still continues as an ongoing draft, as a set of guidelines to implement servers and clients to support the aforementioned features for a decentralized Web. The Community Solid Server is the official beta implementation of such specification, released by the end of 2020. In our example, developers can use the Solid server to get the local business data invoking, e.g., the `"/localbusinessCortFieldID.ttl"` path to retrieve data of `"CortField"` in Turtle format.

6. Discussion and Research Challenges

In this section we discuss our findings by addressing the research questions defined in subsection 2.4. Based on this discussion we outline a set of open research challenges that we consider necessary to ease KG consumption by application developers.

[25] http://vocab.ciudadesabiertas.es/def/comercio/tejido-comercial

6.1. Answering research questions

RQ1.1: Are there API-based methodologies / methods / processes to ease KG consumption by application developers?

Our findings highlight that several specifications have been proposed to provide details on how to define and implement APIs to ease KG consumption. Most of these specifications have been proposed by the Semantic Web community and they are aligned with the REST principles. LDA, Hydra, LDP, LDT, and Solid specifications allow defining read-only, read-write, and full CRUD APIs on single, collection, or nested resources, which are retrieved in several formats. In addition, we found that two specifications from the Software Engineering field (OpenAPI and GraphQL) have been adopted to provide developers with a well-known interface to consume data from KGs. Unlike OpenAPI, the GraphQL spec does not follow the REST paradigm but a more flexible strategy for data consumption over a single endpoint using HTTP.

Almost all the analyzed specifications (LDA, LDP, LDT, Hydra, and Solid) require SPARQL queries, and therefore assume that a Semantic Web expert familiar with the ontology used for modeling the data in a KG is involved in configuring its corresponding API. Similarly, GraphQL also requires developers to know the data structure (ontology) before defining the schema needed for data querying.

RQ1.2: Are there technologies that ease / automate the execution of the API-based methodologies / methods / processes to consume KGs?

Our survey indicates that there are several technologies to automate the API generation to provide developers with a friendly interface for KG consumption. Most of these technologies implement the API specifications described in our review. We also detected that almost all technologies take as input the queries required to retrieve the desired resources for the API generation. However, there are technologies (AtomGraph Processor, OSF, Ontology2GraphQL) which require as input an ontology annotated with specific details to generate the API. In contrast, OBA and OWL2OAS generate the API specification from the OWL ontology that has been developed to describe and organize the KG data. Moreover, OBA also generates automatically the SPARQL queries needed to execute general CRUD operations. All the assessed technologies provide developers with APIs that must be generated by experts in Semantic Web technologies.

RQ2.1: Are there methodologies / methods / processes to help ontology engineers creating APIs that ease ontology-based KG consumption?

Our review revealed that there is no evidence of a formally defined methodology, method, or process to help ontology engineers generate APIs to ease for application developers the ontology-based data consumption. All found efforts are focused on API specifications for KG consumption; but most of them do not consider ontologies as a first-class citizen for designing APIs. Found efforts also do not take into account the experience that the ontology engineer has gained on the target domain; or the artefacts generated

during the ontology development process.

RQ2.2: Are there technologies that ease / automate the execution of methodologies / methods / processes to help ontology engineers creating APIs that ease ontology-based KG consumption?

We found two technologies (OBA and OWL2OAS) that take into account the OWL ontology to generate the APIs. However, in both technologies the authors are focused on the technological support to automatically generate basic APIs rather on the methodology for designing them. In addition, Ontology2GraphQL allows generating a GraphQL schema from an ontology; however it requires users to learn the metamodel necessary to manually annotate such ontology. This technology also needs users to define the queries needed for consume the KG data.

A methodological approach and a technology which implements it are still missing. Both must be focused on helping ontology engineers to actively participate in the API design and implementation from the beginning of the ontology development process in such a manner that at the end of this process other resulting artefact will be the API.

RQ3: Are there tools to help application developers to create APIs on demand?

Surveyed studies revealed that the most recent approaches like GRLC, R4R, RAMOSE, OBA and Walder allow application developers configuring APIs to fulfill their application requirements. However, not all of them allow configuring full CRUD operations or handling nested resources, which hampers the flexibility developers require for building their applications. In addition, in terms of usability, most of these tools require developers to know the ontology behind the KG to design the API, and the query language to pose the required queries to implement the desired methods. For instance, in Walder developers are required to learn the GraphQL-LD language (in addition to the ontology) to design the GraphQL schemas. GRLC has recently included a functionality to allow users to pose queries in JSON, but it requires developers to learn the notation needed to define such queries and also to know the ontology behind the KG data. Many of the reviewed technologies allow doing queries on demand using GraphQL or SPARQL requiring developers to learn these query languages. In summary, the heterogeneity and learning curve of the technologies included in this survey may be challenging for non Semantic Web experts to create APIs over existing KGs on demand.

6.2. Open research challenges

Our systematic review uncovers major research challenges that deserve further investigation. We describe these challenges below:

Automated API generation. Our review showed several specifications and technologies to generate APIs from OWL ontologies and SPARQL queries. However, it is important to consider other inputs that could be reused/added to the API generation process.

- *API generation from ontology engineering artefacts.* There is a lack of investigations or implementations regarding the use of the artefacts that are generated during the ontology development process. These artefacts could be use cases, user stories, or competency questions (defining the functional ontology requirements as proposed in [42]), designed to motivate and assess an ontology. For example, the competency question defined for the local businesses ontology: "What are the local business located in district X?" could be used to automatically generate the required API to answer it and, as a result, to ease the KG consumption by the application developers. Experiments should be conducted in order to test if these artefacts could help application developers to understand the ontology and as a result support them in configuring the custom APIs needed for their applications.

- *API generation from application requirements.* Application developers may want to consume KG data for purposes that are different to those proposed or motivated when the ontology to represent such data was developed. Therefore, it is necessary to investigate alternatives to provide developers with the mechanisms to generate ad-hoc APIs to consume the data that they need for their applications. One alternative could be to allow developers reuse application use cases, requirements, types of users involved, etc., in order to generate the API paths and methods that are required for the application implementation. Several initiatives proposed to transform natural language into knowledge base queries [23] can be used/adapted to generate the queries needed for implementing the ad-hoc API methods. Also, there is an opportunity to explore how language models (e.g. GPT-3 [8]) can be used in the translation of uses cases into API paths.

API version management. None of the surveyed approaches address how changes to an ontology may affect its corresponding KG and API. In some cases like GraphQL, the specification claims to not require managing API versions, since it assumes that the server must handle them and ensure backward compatibility. However, this results in version management having to be handled by API providers. Technologies like Apache Marmotta and Trellis offer resource versioning since they implement the Memento protocol. However, both technologies do not detail how changes are managed in terms of ontology evolution. Therefore, new techniques are needed to detect ontology changes and propagate them into their corresponding APIs, ensuring that applications will not crash when the underlying ontology is updated. Existing work in ontology evolution [102, 76, 77] can be reused and extended to help meet this challenge.

API simplification through lightweight ontologies. Complex ontologies make the API generation process more difficult, since they contain axioms and restrictions that work for defining abstract classes and properties to represent upper-level or domain knowledge, but that are not practical for ontology-based application development. We can illustrate this problem with the SOSA/SSN ontology [43], a W3C recommendation which allows users to represent sensor data. Although this ontology provides several ontology modules intended to supply a lightweight ontology version to those users who do not need extensive axiomatization nor more

specialized entities, it still contains complex representations. For example, to represent the time interval when a sensor observation was measured, users can employ the `sosa:Observation` class and `sosa:phenomenomTime` property. Such property has the `time:TemporalEntity` class as range which is reused from the W3C Time ontology [17]. This class has a `time:Interval` subclass which can be related (using the `time:hasBeginning` and `time:hasEnding` properties) to two instants of time (`time:Instant`). The first instant represents the start of the interval and the second represents its end, each interval must be represented by the `time:Instant` class which contains several datatype properties describing the temporal position of the interval (e.g. `time:inXSDDateTimeStamp`). However, simple applications that require just the start and end of an observation do not require such a verbose mechanism and could be simplified by creating API abstractions on top of the standard representation.

Many W3C recommendations and well-known ontologies have complex mechanisms to describe data. Therefore, providing mechanisms to automatically translate a heavyweight ontology into a lightweight version requires to be investigated (e.g., following existing work for identifying the most relevant ontology concepts [78]; or methods for graph summarization [81, 12]) . This translation would be useful to simplify the API generation and also to provide developers with a reduced version of the ontology prior to consuming the KG data.

API resource path prediction. Some of the analyzed technologies allow automatically defining basic API paths, while others allow their customization. We consider that one challenge is the prediction of relevant paths based on the data available in a KG. This would help automatically retrieving the most relevant resources in a KG, based on e.g., their number of connections, frequency or other metrics from graph theory and network analysis [29] e.g. centrality, connectivity, community detection etc. [48]. In this prediction scenario, using the ontology is relevant since data need to follow the structure defined by it. Automatically generating the queries necessary to implement the methods of each predicted API path is also a challenge that still needs to be addressed.

API validation and testing. Following a Test Driven Development (TDD) [4] approach is a common practice in the Application Development field. Therefore, applying such testing approach to APIs helps ensuring that APIs are aligned with their functional requirements, allowing developers to validate the permissions granted to users when executing certain operations. TDD allows developers creating test requests by defining the API resource paths together with their required operations. Users can then implement the missing functionality and run the API tests until they pass, refining them iteratively in case of errors. One initial effort in this direction has been proposed in OBA, allowing users to produce and perform automated unit tests to evaluate the API paths that are automatically generated. However, these tests are basic and they only support GET requests.

7. Conclusions

The growing number of Knowledge Graphs on the Web reaffirms the great importance they have within the business strategies of public and private organizations. Easing KG consumption by application developers is a big challenge since most developers are not sufficiently aware of semantic technologies and find it difficult to develop applications for which KG data can be exploited.

This article contributed with a systematic literature review concerning API-based solutions for KGs consumption. We proposed two comparison frameworks to analyze the existing specifications, technologies and tools which implement them. We presented, compared and discussed approaches to ease KG consumption through APIs; and we found that most of the existing research works focus on API generation from queries whereas recently some tools have been exploring how to generate APIs from OWL ontologies.

Our results indicate the need for improvements in this research field. To this end, the challenges we outlined provide some ideas to alleviate some of the limitations we found in this work. We believe that it is necessary for the Semantic Web community to discuss these challenges and join forces to propose other alternatives that could ease the work of developers when generating applications with ontology-based data. Many developers today are not familiar with Semantic Web technologies and, as a consequence, the great potential of the semantic representations and data has not been fully exploited. Therefore, as a community it becomes crucial that we prioritize application developers as the key users of our KGs and find new solutions that allow bridging the gap between developers and Semantic Web experts.

CRediT authorship contribution statement

Paola Espinoza-Arias: Conceptualization, Methodology, Investigation, Writing - Original Draft. **Daniel Garijo:** Conceptualization, Methodology, Validation, Supervision, Writing - Original Draft. **Oscar Corcho:** Conceptualization, Validation, Supervision, Writing - review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been funded by a Predoctoral grant from the I+D+i program of the Universidad Politécnica de Madrid, the Spanish project DATOS 4.0: RETOS Y SOLUCIONES (TIN2016-78011-C4-4-R), by the Defense Advanced Research Projects Agency (DARPA) with award W911NF-18-1-0027 and the National Institutes of Health (NIH) with Award number 1R01AG059874-01. The authors would like to thank Carlos Badenes Olmedo and José Luis Redondo García for all their valuable comments and feedback.

Table 5
Studies resulting from the literature review process

Authors	Title	Technology / Tool / Specification
Farré C., Varga J. and Almar R. [30]	GraphQL Schema Generation for Data-Intensive Web APIs	Ontology2GraphQL
Jansen G. et al. [52]	DRAS-TIC linked data: Evenly distributing the past	Trellis LDP
Zeginis D. et al. [103]	Facilitating the exploitation of Linked open Statistical data: JSON-QB API requirements and design criteria	JSON-QB API
Mayer S. et al. [67]	An Open Semantic Framework for the Industrial Internet of Things	OSF
Meroño-Peñuela A. and Hoekstra R. [70]	grlc makes GitHub taste like linked data APIs	GRLC and Swagger
Yu L. and Liu Y. [101]	Using Linked Data in a heterogeneous Sensor Web: challenges, experiments and lessons learned	ELDA and LDA
Car N.J. [10]	A method and example system for managing provenance information in a heterogeneous process environment-a provenance architecture containing the Provenance Management System (PROMS)	ELDA and LDA
Daga E., Panziera L. and Pedrinaci C. [20]	A BASILar approach for building web APIs on top of SPARQL endpoints	BASIL and Swagger
Bukhari A.C. et al. [9]	ICyrus: A semantic framework for biomedical image discovery	Virtuoso and Pubby
Lopes P. and Luís Oliveira J. [66]	COEUS: "semantic web in a box" for biomedical applications	Pubby
Lanthaler M. and Gütl C. [60]	Hydra: A vocabulary for hypermedia-driven web APIs	Hydra
Lapi E. et al. [61]	Identification and utilization of components for a linked open data platform	Virtuoso and SPARQL Protocol
Narvaez N. and Piedra N [72]	A Linked Data approach to guarantee the semantic interoperability and integrity of university academic data	Apache Marmotta

Appendix

A. Literature results

Table 5 presents the studies resulting from the literature review process together with the authors and the resulting technologies, tools, or specifications that we analyzed in this work.

B. Query Examples

B.1. SPARQL example

List 1 presents the SPARQL query to get the instance of local business `"CortField"`.

Listing 1: SPARQL query example

```
PREFIX escom:<http://vocab.ciudadesabiertas.es/def/comercio/tejido-comercial#>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
SELECT ?localBusiness
WHERE
{
    ?localBusiness a escom:LocalComercial.
    ?localBusiness dc:identifier "CortFieldID"
}
```

B.2. Walder example

List 2 shows an excerpt of the OAS configuration, using the Walder-specific extensions, providing the GraphQL query and JSON-LD context that allows getting the name and capacity of a local business.

Listing 2: Query example in Walder

```
x-walder-query:
    graphql-query: >
    {
     name @single
     capacity @single
     (id:$id)
    }

    json-ld-context: >
    {
      "escom": "http://vocab.ciudadesabiertas.es/def/comercio/tejido-comercial#",
      "dc": "http://purl.org/dc/elements/1.1/",
      "name": "escom:nombreComercial",
      "capacity": "escom:aforo",
      "id": "dc:identifier"
    }
```

B.3. GRLC example

List 3 shows the `"localbusineess.rq"` query annotated with the SPARQL notation to describe the operation implemented by the query, the URL of the SPARQL endpoint, and a summary of the operation.

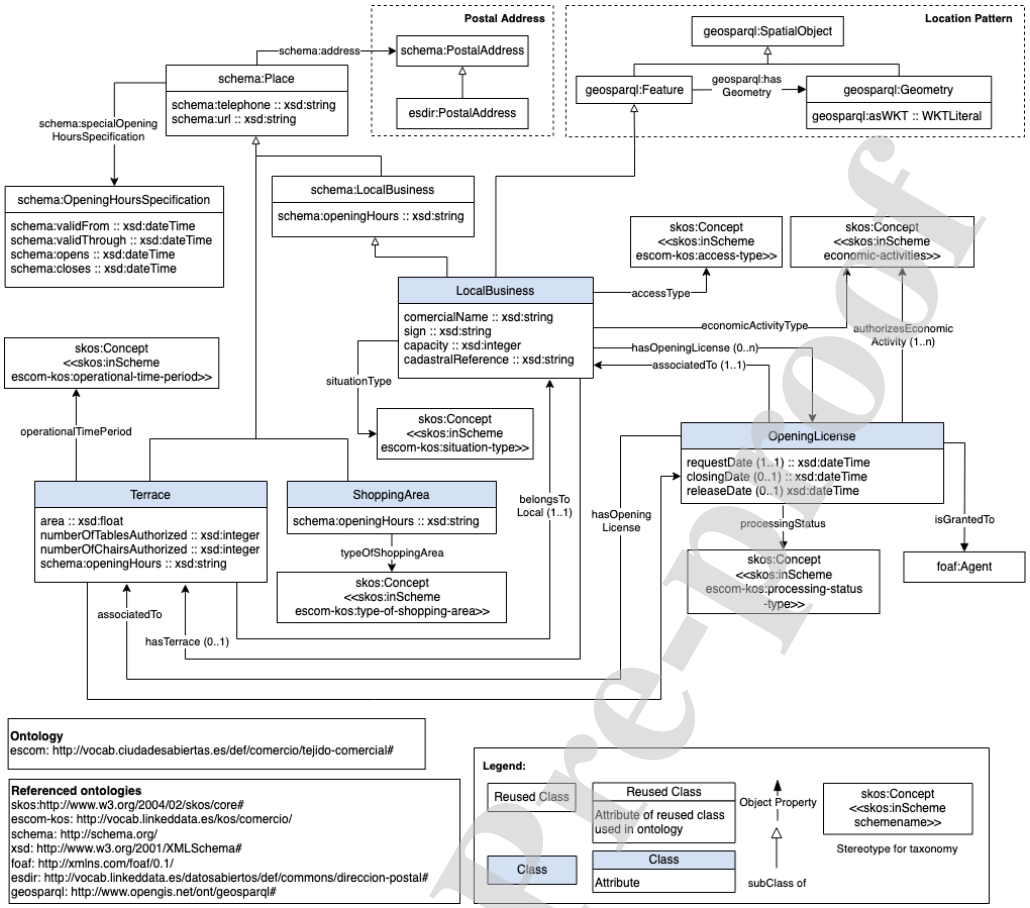

Figure 4: Diagram of the ontology for the representation of data from the census of local business premises and terraces, as well as their associated economic activities and activity licenses. Elements in blue correspond to the new classes defined for this ontology.

Listing 3: GRLC query example

```
#+ summary: Returns the instance of "CortField" local busi
ness
#+ endpoint: http://example.com/sparql
#+ method: GET

PREFIX escom:<http://vocab.ciudadesabiertas.es/def/comer
cio/tejido-comercial#>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
SELECT ?localBusiness
WHERE
{
    ?localBusiness a escom:LocalComercial.
    ?localBusiness dc:identifier "CortFieldID"
}
```

C. Ontology Diagrams

C.1. Diagram of the ontology for the census of local businesses

Figure 4 shows the diagram of the ontology mentioned in the motivating example presented in subsection 2.1. For readability purposes, the class and property names in both diagrams correspond with the English values of the ontology elements' labels. However, the naming strategy followed in this ontology uses Spanish terms. The English version of the ontology documentation is available on the Web.[26]

C.2. Diagram of the ontology for the data cubes representation of census of inhabitants

Figure 5 shows the diagram of shows the Population By Age data cube represented by the ontology mentioned in the motivating example of subsection 2.2. The diagrams of the remaining six data cubes defined in this ontology are available in its HTML documentation.[27] For readability purposes we translated to English the original names of the data cube instances (`ex:DSD_PopulationByAge` and `ex:DS_PopulationByAge`) and the measure employed to represent the number of persons (`espad-medida:persons-number`). However, the naming strategy followed in this ontology uses Spanish terms.

[26] http://vocab.ciudadesabiertas.es/def/comercio/tejido-comercial/index-en.html

[27] http://vocab.ciudadesabiertas.es/def/demografia/cubo-padron-municipal/index-en.html

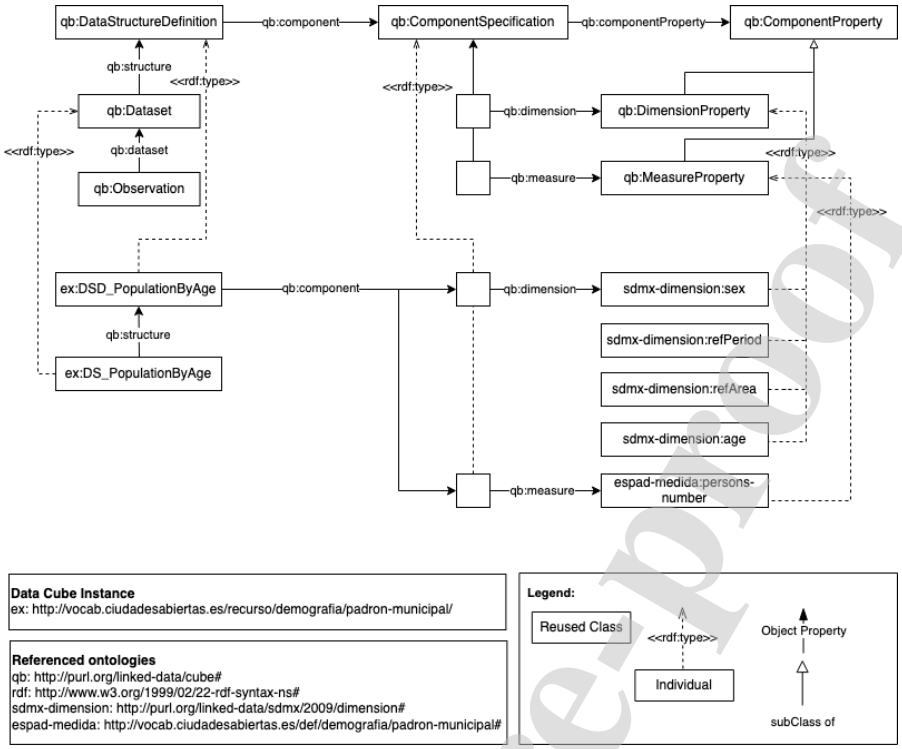

Figure 5: Diagram of the Population By Age data cube defined in the ontology for the data cubes representation of census of inhabitants.

References

[1] Apache, 2014. Apache Jena Fuseki. URL: https://jena.apache.org/documentation/fuseki2/. accessed: 2020-03-12.

[2] Arwe, J., Speicher, S., Malhotra, A., 2015. Linked Data Platform 1.0. W3C Recommendation. W3C. URL: https://www.w3.org/TR/2015/REC-ldp-20150226/.

[3] Badenes-Olmedo, C., 2019. RESTful API for RDF data. doi:10.5281/zenodo.3543320. accessed: 2020-01-20.

[4] Beck, K., 2003. Test-Driven Development: By Example. Addison-Wesley Professional.

[5] Berners-Lee, T., Capadisli, S., Verborgh, R., Kjernsmo, K., Bingham, J., Zagidulin, D., 2019. The Solid Ecosystem, Editor's Draft. URL: https://solid.github.io/specification/. accessed: 2020-09-11.

[6] Bray, T., et al., 2014. The JavaScript Object Notation (JSON) Data Interchange Format .

[7] Broekstra, J., Kampman, A., Van Harmelen, F., 2002. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema, in: International semantic web conference, Springer. pp. 54–68.

[8] Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al., 2020. Language Models are Few-Shot Learners. arXiv preprint arXiv:2005.14165 .

[9] Bukhari, S.A.C., Nagy, M.L., Ciccarese, P., Krauthammer, M., Baker, C.J.O., 2015. iCyrus: A Semantic Framework for Biomedical Image Discovery, in: SWAT4LS.

[10] Car, N., 2013. A method and example system for managing provenance information in a heterogeneous process environment–a provenance architecture containing the Provenance Management System (PROMS), in: 20th International Congress on Modelling and Simulation, pp. 824–830.

[11] Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K., 2004. Jena: Implementing the Semantic Web Recommendations, in: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, pp. 74–83.

[12] Čebirić, Š., Goasdoué, F., Kondylakis, H., Kotzinos, D., Manolescu, I., Troullinou, G., Zneika, M., 2019. Summarizing semantic graphs: a survey. The VLDB Journal 28, 295–327.

[13] Cheron, A., Bourcier, J., Barais, O., Michel, A., 2019. Comparison Matrices of Semantic RESTful APIs Technologies, in: International Conference on Web Engineering, Springer. pp. 425–440.

[14] Clark, K., Torres, E., Feigenbaum, L., 2008. SPARQL Protocol for RDF. W3C Recommendation. W3C. Https://www.w3.org/TR/2008/REC-rdf-sparql-protocol-20080115/.

[15] Corcho, O., Fernández-López, M., Gómez-Pérez, A., 2003. Methodologies, tools and languages for building ontologies. Where is their meeting point? Data & knowledge engineering 46, 41–64.

[16] Corradi, A., Foschini, L., Ianniello, R., 2014. Linked Data for Open Government: The Case of Bologna, in: 2014 IEEE Symposium on Computers and Communications (ISCC), IEEE. pp. 1–7.

[17] Cox, S., Little, C., 2017. Time Ontology in OWL. W3C Recommendation. W3C. Https://www.w3.org/TR/2017/REC-owl-time-20171019/.

[18] Cyganiak, R., Bizer, C., 2007. Pubby – A Linked Data Frontend for SPARQL Endpoints. URL: http://wifo5-03.informatik.uni-mannheim.de/pubby/. accessed: 2020-03-08.

[19] Cyganiak, R., Reynolds, D., 2014. The RDF Data Cube Vocabulary. W3C Recommendation. W3C. Https://www.w3.org/TR/2014/REC-vocab-data-cube-20140116/.

[20] Daga, E., Panziera, L., Pedrinaci, C., 2015. A BASILar approach for building web APIs on top of SPARQL endpoints, in: CEUR Workshop Proceedings, pp. 22–32.

[21] Daquino, M., Heibi, I., Peroni, S., Shotton, D., 2020. Creating Restful APIs over SPARQL endpoints with RAMOSE. arXiv preprint arXiv:2007.16079 .

[22] Deliot, C., 2014. Publishing the British National Bibliography as

Linked Open Data. Catalogue & Index 174, 13–18.
[23] Diefenbach, D., Lopez, V., Singh, K., Maret, P., 2018. Core techniques of question answering systems over knowledge bases: a survey. Knowledge and Information systems 55, 529–569.
[24] Elsevier, . Scopus. URL: https://www.scopus.com. accessed: 2020-02-12.
[25] Epimorphics, 2011. Elda: The linked-data API in Java. URL: http://epimorphics.github.io/elda/index.html. accessed: 2020-03-09.
[26] Erling, O., Mikhailov, I., 2009. RDF Support in the Virtuoso DBMS, in: Networked Knowledge-Networked Media. Springer, pp. 7–24.
[27] Espinoza-Arias, P., Fernández-Ruiz, M.J., Morlán-Plo, V., Notivol-Bezares, R., Corcho, O., 2020. The Zaragoza's Knowledge Graph: Open Data to Harness the City Knowledge. Information 11, 129.
[28] Espinoza-Arias, P., Garijo, D., Corcho, O., 2021. Scopus records. URL: https://doi.org/10.5281/zenodo.4433203, doi:10.5281/zenodo.4433203.
[29] Estrada, E., 2012. The Structure of Complex Networks: Theory and Applications. Oxford University Press.
[30] Farré, C., Varga, J., Almar, R., 2019. GraphQL schema generation for data-intensive web APIs, in: International Conference on Model and Data Engineering, Springer. pp. 184–194.
[31] Feigenbaum, L., Torres, E., Clark, K., Williams, G., 2013. SPARQL 1.1 Protocol. W3C Recommendation. W3C. Https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/.
[32] Fernández-Sellers, M., 2015. Linked Open Data Inspector. URL: https://github.com/marfersel/LODI/. accessed: 2020-03-12.
[33] Fielding, R.T., Taylor, R.N., 2002. Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology (TOIT) 2, 115–150.
[34] Fletcher, G., Groth, P., Sequeda, J., 2020. Knowledge Scientists: Unlocking the data-driven organization. arXiv preprint arXiv:2004.07917 .
[35] Foundation, G., 2015. GraphQL. URL: https://spec.graphql.org/June2018/. accessed: 2020-10-10.
[36] Foundation, T.A.S., 2013. Apache Marmotta. URL: http://marmotta.apache.org/. accessed: 2020-07-10.
[37] Garcia, A., O'Neill, K., Garcia, L.J., Lord, P., Stevens, R., Corcho, O., Gibson, F., 2010. Developing Ontologies within Decentralised Settings, in: Semantic e-Science. Springer, pp. 99–139.
[38] Garijo, D., Osorio, M., 2020. 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, Springer International Publishing, Cham. pp. 48–64. doi:10.1007/978-3-030-62466-8_4.
[39] Google, 2010a. Linked Data API. URL: https://github.com/UKGovLD/linked-data-api. accessed: 2020-03-02.
[40] Google, 2010b. Puelia. URL: https://code.google.com/archive/p/puelia-php/. accessed: 2020-03-09.
[41] Groth, P., Loizou, A., Gray, A.J., Goble, C., Harland, L., Pettifer, S., 2014. Api-centric linked data integration: The open phacts discovery platform case study. Journal of Web Semantics 29, 12 – 18. URL: http://www.sciencedirect.com/science/article/pii/S1570826814000195, doi:https://doi.org/10.1016/j.websem.2014.03.003. life Science and e-Science.
[42] Grüninger, M., Fox, M.S., 1995. Methodology for the Design and Evaluation of Ontologies .
[43] Haller, A., Janowicz, K., Cox, S., Lefrançois, M., Taylor, K., Le Phuoc, D., Lieberman, J., García-Castro, R., Atkinson, R., Stadler, C., 2018. The SOSA/SSN ontology: a joint WeC and OGC standard specifying the semantics of sensors observations actuation and sampling. Semantic Web 1, 1–19.
[44] Hammar, K., 2020. The OWL2OAS Converter. URL: https://github.com/RealEstateCore/OWL2OAS. accessed: 2020-06-15.
[45] Hartig, O., Zhao, J., Mühleisen, H., 2010. Automatic integration of metadata into the web of linked data, in: Proceedings of the Demo Session at the 2nd Workshop on Trust and Privacy on the Social and Semantic Web (SPOT) at ESWC, pp. 2–4.
[46] Hendler, J., Holm, J., Musialek, C., Thomas, G., 2012. US Government Linked Open Data: Semantic.data.gov. IEEE Intelligent Systems , 25–31.
[47] Heyvaert, P., De Meester, B., Pandit, H., Verborgh, R., 2020. Walder. URL: https://github.com/KNowledgeOnWebScale/walder. accessed: 2020-10-20.
[48] Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Gayo, J.E.L., Kirrane, S., Neumaier, S., Polleres, A., et al., 2020. Knowledge Graphs. arXiv preprint arXiv:2003.02320 .
[49] Horridge, M., Bechhofer, S., 2011. The OWL API: A Java API for OWL ontologies. Semantic web 2, 11–21.
[50] Initiative, O., 2011. OpenAPI Specification. URL: https://swagger.io/specification/. accessed: 2020-10-05.
[51] Jansen, G., Coburn, A., Soroka, A., Marciano, R., 2019a. Using Data Partitions and Stateless Servers to Scale Up Fedora Repositories, in: 2019 IEEE International Conference on Big Data (Big Data), IEEE Computer Society. pp. 3098–3102.
[52] Jansen, G., Coburn, A., Soroka, A., Thomas, W., Marciano, R., 2019b. DRAS-TIC Linked Data: Evenly Distributing the Past. Publications 7, 50.
[53] Jusevicius, M., 2014. Graphity: Generic processor for declarative linked data applications, in: SWAT4LS.
[54] Jusevičius, M., 2016. Linked Data Templates, in: Proceedings of the XML London conference, pp. 50–55.
[55] Jusevičius, M., 2016. AtomGraph Processor. URL: https://github.com/AtomGraph/Processor. accessed: 2020-08-15.
[56] Keet, M., 2018. An Introduction to Ontology Engineering. volume 1. Maria Keet.
[57] Kellogg, G., Lanthaler, M., Sporny, M., 2014. JSON-LD 1.0. W3C Recommendation. W3C. Https://www.w3.org/TR/2014/REC-json-ld-20140116/.
[58] Kitchenham, B., Charters, S., 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering. Engineering 45, 1051.
[59] Kotis, K.I., Vouros, G.A., Spiliotopoulos, D., 2020. Ontology engineering methodologies for the evolution of living and reused ontologies: status, trends, findings and recommendations. The Knowledge Engineering Review 35.
[60] Lanthaler, M., Gütl, C., 2013. Hydra: A Vocabulary for Hypermedia-Driven Web APIs. LDOW 996.
[61] Lapi, E., Tcholtchev, N., Bassbouss, L., Marienfeld, F., Schieferdecker, I., 2012. Identification and Utilization of Components for a Linked Open Data Platform, in: 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops, IEEE. pp. 112–115.
[62] Ledvinka, M., Kostov, B., Křemen, P., 2016. JOPA: Efficient Ontology-Based Information System Design, in: Sack, H., Rizzo, G., Steinmetz, N., Mladenić, D., Auer, S., Lange, C. (Eds.), The Semantic Web, Springer International Publishing, Cham. pp. 156–160. doi:10.1007/978-3-319-47602-5_31.
[63] Ledvinka, M., Křemen, P., 2020. A comparison of object-triple mapping libraries. Semantic Web 11, 483–524. URL: https://content.iospress.com/articles/semantic-web/sw190345, doi:10.3233/SW-190345. publisher: IOS Press.
[64] Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., Van Kleef, P., Auer, S., et al., 2015. DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. Semantic web 6, 167–195.
[65] Lisena, P., Meroño-Peñuela, A., Kuhn, T., Troncy, R., 2019. Easy Web API Development with SPARQL Transformer, in: Ghidini, C., Hartig, O., Maleshkova, M., Svátek, V., Cruz, I., Hogan, A., Song, J., Lefrançois, M., Gandon, F. (Eds.), The Semantic Web – ISWC 2019, Springer International Publishing, Cham. pp. 454–470.
[66] Lopes, P., Oliveira, J.L., 2012. COEUS:"semantic web in a box" for biomedical applications. Journal of biomedical semantics 3, 11.
[67] Mayer, S., Hodges, J., Yu, D., Kritzler, M., Michahelles, F., 2017. An Open Semantic Framework for the industrial Internet of Things. IEEE Intelligent Systems 32, 96–101.

[68] McCrae, J.P., Abele, A., Buitelaar, P., Cyganiak, R., Jentzsch, A., Andryushechkin, V., Debattista, J., Nasir, J., 2007. Linked Open Data Cloud. URL: https://lod-cloud.net/. accessed: 2020-12-10.

[69] McGuinness, D.L., Van Harmelen, F., et al., 2004. OWL Web Ontology Language Overview. W3C recommendation 10, 2004.

[70] Meroño-Peñuela, A., Hoekstra, R., 2016. grlc Makes GitHub Taste Like Linked Data APIs, in: European Semantic Web Conference, Springer. pp. 342–353.

[71] Miller, J.J., 2013. Graph Database Applications and Concepts with Neo4j, in: Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA.

[72] Narvaez, E., Piedra, N., 2018. Un enfoque de linked data para garantizar la interoperabilidad semántica e integridad de datos académicos universitarios (A linked data approach to guarantee the semantic interoperability and integrity of university academic data), in: Proceedings of the 3rd International Workshop on Semantic Web 2018 co-located with 15th International Congress on Information (INFO 2018), CEUR-WS.org. pp. 50–62.

[73] Noy, N., Gao, Y., Jain, A., Narayanan, A., Patterson, A., Taylor, J., 2019. Industry-scale Knowledge Graphs: Lessons and Challenges. Queue 17, 48–75.

[74] Ogbuji, C., 2013. SPARQL 1.1 Graph Store HTTP Protocol. W3C Recommendation. W3C. Https://www.w3.org/TR/2013/REC-sparql11-http-rdf-update-20130321/.

[75] Ontotext, 2015. GraphDB. URL: https://graphdb.ontotext.com/. accessed: 2020-03-12.

[76] Osborne, F., Motta, E., 2018. Pragmatic Ontology Evolution: Reconciling User Requirements and Application Performance, in: International Semantic Web Conference, Springer. pp. 495–512.

[77] Pernisch, R., Dell'Aglio, D., Serbak, M., Bernstein, A., 2020. ChImp: Visualizing Ontology Changes and theirImpact in Protégé, in: Fifth International Workshop on Visualization and Interaction for Ontologies and Linked Data, CEUR Workshop Proceedings. pp. 47–60.

[78] Peroni, S., Motta, E., d'Aquin, M., 2008. Identifying Key Concepts in an Ontology, through the Integration of Cognitive Principles with Statistical and Topological Measures, in: Asian Semantic Web Conference, Springer. pp. 242–256.

[79] Piñero, J., Ramírez-Anguita, J.M., Saüch-Pitarch, J., Ronzano, F., Centeno, E., Sanz, F., Furlong, L.I., 2020. The DisGeNET knowledge platform for disease genomics: 2019 update. Nucleic acids research 48, D845–D855.

[80] Postman, 2020. State of the API Report. URL: https://www.postman.com/state-of-api. accessed: 2021-03-22.

[81] Pouriyeh, S., Allahyari, M., Liu, Q., Cheng, G., Arabnia, H.R., Atzori, M., Mohammadi, F.G., Kochut, K., 2019. Ontology Summarization: Graph-Based Methods and Beyond. International Journal of Semantic Computing 13, 259–283.

[82] Safris, S., 2019. A Deep Look at JSON vs. XML, Part 1: The History of Each Standard. URL: https://www.toptal.com/web/json-vs-xml-part-1. accessed: 2021-03-25.

[83] Salvadori, I., Siqueira, F., 2015. A Maturity Model for Semantic RESTful Web APIs, in: 2015 IEEE International Conference on Web Services, IEEE. pp. 703–710.

[84] Schröder, M., Hees, J., Bernardi, A., Ewert, D., Klotz, P., Stadtmüller, S., 2018. Simplified sparql rest api, in: Gangemi, A., Gentile, A.L., Nuzzolese, A.G., Rudolph, S., Maleshkova, M., Paulheim, H., Pan, J.Z., Alam, M. (Eds.), The Semantic Web: ESWC 2018 Satellite Events, Springer International Publishing, Cham. pp. 40–45.

[85] Seaborne, A., Harris, S., 2013. SPARQL 1.1 Query Language. W3C Recommendation. W3C. Https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[86] Severance, C., 2012. Discovering JavaScript Object Notation. Computer 45, 6–8.

[87] Shadbolt, N., O'Hara, K., Berners-Lee, T., Gibbins, N., Glaser, H., Hall, W., et al., 2012. Linked Open Government Data: Lessons from Data.gov.uk. IEEE Intelligent Systems 27, 16–24.

[88] Shefchek, K.A., Harris, N.L., Gargano, M., Matentzoglu, N., Unni, D., Brush, M., Keith, D., Conlin, T., Vasilevsky, N., Zhang, X.A., et al., 2020. The Monarch Initiative in 2019: An integrative data and analytic platform connecting phenotypes to genotypes across species. Nucleic acids research 48, D704–D715.

[89] Simon, A., Wenz, R., Michel, V., Di Mascio, A., 2013. Publishing bibliographic records on the web of data: Opportunities for the bnf (french national library), in: Extended Semantic Web Conference, Springer. pp. 563–577.

[90] Stasiewicz, A., Waqar, M., 2016. Deliverable 3.2: Report on OpenGovIntelligence ICT tools - first release. Technical Report. National University of Ireland (NUIG). OpenGovIntelligence Project. URL: https://ec.europa.eu/futurium/sites/futurium/files/d3.2_693849_report_on_opengovintelligence_ict_tools_first_release.pdf.

[91] Studer, R., Benjamins, V.R., Fensel, D., 1998. Knowledge engineering: Principles and methods. Data & knowledge engineering 25, 161–197.

[92] SYSTAP, L., 2015. Blazegraph. URL: https://blazegraph.com/. accessed: 2020-03-12.

[93] Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R., 2018a. Comunica: a Modular SPARQL Query Engine for the Web, in: Proceedings of the 17th International Semantic Web Conference. URL: https://comunica.github.io/Article-ISWC2018-Resource/.

[94] Taelman, R., Vander Sande, M., Verborgh, R., 2018b. GraphQL-LD: linked data querying with GraphQL, in: ISWC2018, the 17th International Semantic Web Conference, pp. 1–4.

[95] Torres, E., Clark, K., Feigenbaum, L., 2008. SPARQL Protocol for RDF. W3C Recommendation. W3C. Https://www.w3.org/TR/2008/REC-rdf-sparql-protocol-20080115/.

[96] Van Herwegen, J., Taelman, R., Verborgh, R., 2020. Community Solid Server. URL: https://github.com/solid/community-server. accessed: 2020-11-20.

[97] Verborgh, R., Taelman, R., 2020. LDflex: A Read/Write Linked Data Abstraction for Front-End Web Developers, in: International Semantic Web Conference, Springer. pp. 193–211.

[98] Verborgh, R., Vander Sande, M., 2020. The Semantic Web identity crisis: in search of the trivialities that never were. Semantic Web , 1–9.

[99] Vila-Suero, D., Villazón-Terrazas, B., Gómez-Pérez, A., 2013. datos.bne.es: A library linked dataset. Semantic Web 4, 307–313.

[100] Vrandečić, D., Krötzsch, M., 2014. Wikidata:A Free Collaborative Knowledge Base. Communications of the ACM 57, 78–85.

[101] Yu, L., Liu, Y., 2015. Using Linked Data in a heterogeneous Sensor Web: challenges, experiments and lessons learned. International Journal of Digital Earth 8, 17–37. doi:10.1080/17538947.2013.839007.

[102] Zablith, F., Antoniou, G., d'Aquin, M., Flouris, G., Kondylakis, H., Motta, E., Plexousakis, D., Sabou, M., 2015. Ontology evolution: a process-centric survey. The knowledge engineering review 30, 45–75.

[103] Zeginis, D., Kalampokis, E., Roberts, B., Moynihan, R., Tambouris, E., Tarabanis, K.A., 2017. Facilitating the Exploitation of Linked Open Statistical Data: JSON-QB API Requirements and Design Criteria, in: HybridSemStats@ ISWC.

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: