Abstract—Massive amounts of Linked Open Data are readily available to anyone who wants to use them. Unfortunately, Semantic Web technologies such as SPARQL and RDF remain unfamiliar to the majority of web developers, more used to REST APIs. This paper addresses the challenge of accessing Linked Open Data through REST APIs. Configurable REST APIs For Triple Stores (CRAFTS) is the tool devised for this purpose. CRAFTS allows knowledge engineers to configure REST APIs over multiple triple stores. CRAFTS automatically handles the translation of API calls into SPARQL queries, delivering results in JSON format. Web developers can then use a CRAFTS API to read and write Linked Open Data. The API of CRAFTS is uniform, domain-independent, and described with the OpenAPI specification. A reference implementation of CRAFTS is published in a GitHub repository, and a live test site is readily available since February 2021. CRAFTS is currently employed in eight different applications, with more than 2.4K users, and more than 260K API calls.

Index Terms—REST API, triple stores, Linked Open Data, SPARQL, data access.

I. INTRODUCTION

With the embrace of the Linked Data principles [1], [2], a data deluge is available across all domains. This is especially evident through the evolution of the Linked Open Data (LOD) cloud from 12 datasets in 2007 to 1,301 in 2021 [3]. While there are different ways of accessing LOD [3], SPARQL querying has widespread adoption among Semantic Web practitioners. Unfortunately, SPARQL has a very steep learning curve, is error-prone, and tedious, even for experts [1], [5].

Web developers are not familiar with Semantic Web technologies such as SPARQL, RDF, and OWL. Instead, they are used to Representational State Transfer (REST) [6] Application Programming Interfaces (APIs) and the JavaScript Object Notation (JSON) data interchange format [7]. Therefore, the challenge for the Semantic Web community is how to make accessible the LOD cloud to web developers. Ideally, they should employ regular REST APIs that allow them to read and write Linked Open Data from multiple triple stores and using JSON for data exchanges. There are several proposals of API generators for Linked Open Data, notably RAMOSE [4].

Corresponding author: G. Vega-Gorgojo (email: guiveg@tel.uva.es).

OBA [8], grlc [9], and BASIL [10]. However, they do not comply with all the aforementioned requirements.

In this paper, this challenge is addressed with the proposal of CRAFTS, a tool that can be configured to access Linked Open Data through REST APIs. Contributions include:

- A uniform API for managing CRAFTS APIs, sending parametrized SPARQL queries, and performing read and write operations over RDF resources (Section III-B).
- An architecture with the logical components of CRAFTS for providing the desired functionalities (Section III-C).
- A reference implementation of CRAFTS, available in a GitHub repository. A working test site of CRAFTS is also included (Section III-D).
- A showcase scenario (Section IV) and a brief description of the main applications that currently exploit CRAFTS (Section V).
- A discussion of CRAFTS and its relationship to other competing approaches (Section VI).

The rest of the paper is organized as follows: Section II analyses existing approaches for accessing LOD, particularly API generators. Section III provides a technical description of CRAFTS, including its requirements, a description of the API of CRAFTS, its logical architecture, and implementation details. Section IV presents a showcase scenario. Section V reports about the uptake of CRAFTS so far. Section VI includes a discussion. The paper ends with a conclusion and future work lines in Section VII.

II. RELATED WORK

Access to LOD is challenging even for knowledge engineers. Beyond expertise with W3C standards such as RDF and SPARQL, accessing LOD requires familiarity with the ontologies employed to annotate data, as well as some domain knowledge (not trivial in many specialized domains). Due to this, the Semantic Web community has proposed several approaches to facilitate the access to triple stores. Some proposals define an HTTP interface over Linked Data, notably Linked Data Platform [11] and Linked Data Fragments [3]. The Linked Data Platform is a W3C recommendation for providing read and write access to Linked Data resources. The main focus is on describing resource representations with triples, but its application
to a non-RDF model such as JSON is not covered. Linked Data Fragments is a framework that defines so-called Triple Pattern Fragments for consuming Linked Data in an efficient way. Triple Pattern Fragments can be used as a limited Linked Data API that provides read-only access to RDF resources that returns triples.

Other approaches propose new serializations of Linked Data and SPARQL results to JSON such as JSON-LD [12] and SPARQL transformer [13]. JSON-LD is a syntax to serialize Linked Data in JSON format. JSON-LD is especially relevant for publishers, since it provides a way to convert an RDF graph into JSON. SPARQL transformer proposes a new JSON serialization of SPARQL results in order to simplify the output, but it does not support SPARQL query formulation, a much more demanding task for web developers.

Web developers commonly employ REST APIs and JSON as the main interchange format [14]. It is thus desirable to provide a similar solution for accessing LOD. There are several proposals in the literature that support the creation of REST APIs on top of triple stores: RAMOSE [4], grlc [9], BASIL [10], and OBA [8]. To facilitate their comparison, I will employ the following set of desirable features of API generators for LOD:

- Generic approach: an API generator should provide a domain independent solution for the provision of REST APIs over LOD sources.
- Flexible configuration of REST APIs: an API generator should allow a high degree of control in the specification of an API.
- Ontology-driven API generation: an API can be configured by using an ontology as input.
- Expose RDF resources: an API can provide access to RDF resources through HTTP methods [15].
- Support read operations: an API can provide read access to exposed RDF resources through HTTP GET calls.
- Support write operations: an API can provide write access to exposed RDF resources with operations such as POST, PUT, or DELETE.
- Support partial updates: an API can provide partial updates to exposed RDF resources through HTTP PATCH calls.
- Support parametrized queries: an API can provide a way to specify input parameters for submitting parametrized SPARQL queries.
- Use JSON as interchange format: an API should use JSON to format body requests and responses.
- Caching SPARQL queries: an API can cache responses from triple stores to limit network exchanges.
- OpenAPI documentation: an API can be documented with the OpenAPI specification [16].
- Access to multiple endpoints: an API can access one or several endpoints.

Table I summarizes the features of the surveyed API generators for LOD. All of them offer generic approaches for the creation of REST APIs and employ JSON as interchange format. RAMOSE, grlc, and BASIL essentially allow the provision of APIs that encapsulate parametrized SPARQL queries. In this way, knowledge engineers will define the queries to be supported, and web developers will make API calls instead of authoring SPARQL queries. However, none of them allow the exposition of RDF resources over an API – this is why the flexibility of these tools is considered “Low”.

In contrast to the previous proposals, OBA is mainly purposed for exposing RDF resources over REST APIs. OBA employs an ontology as input to automatically generate an API for a single target endpoint. The ontology is analyzed to extract the classes and properties in order to create an OpenAPI specification for the API. OBA employs a series of predefined query templates to map API calls into SPARQL queries. Read and write operations are supported since OBA maps GET, POST, PUT, and DELETE requests to CONSTRUCT, INSERT, UPDATE, and DELETE SPARQL queries, respectively; however, partial updates are not supported. OBA generates one-to-one mappings between an API call and a SPARQL query. The simplicity of this translation facilitates the automatic generation of an API from an ontology, but flexibility is somewhat reduced, basically allowing the retrieval of the datatype and object properties of RDF resources. OBA can also accommodate parametrized SPARQL queries, requiring a knowledge engineer to author them.

All the surveyed proposals except RAMOSE support OpenAPI documentation. None of them use caching to limit exchanges with triple stores. Moreover, they do not provide access to multiple endpoints from an API.
This analysis serves to identify a group of API generators (RAMOSE, grlc, and BASIL) providing a basic functionality that is limited to the exposition of parametrized SPARQL queries. On a next step, OBA also allows the exposition of RDF resources—supporting read and write operations—, as well as ontology-driven API generation that can be appealing in some situations. However, there are cases that require additional features, such as enhanced flexibility in the specification of an API, support for partial updates, caching SPARQL queries, and access to multiple triple stores.

III. TECHNICAL DESCRIPTION OF CRAFTS

Configurable REST APIs For Triple Stores (CRAFTS) is a novel tool for accessing LOD through REST APIs. As graphically depicted in Figure 1, different types of users are involved. LOD users are interested in accessing Linked Open Data, but they require appropriate applications in order to fulfill their user goals. Web developers are programmers that develop applications for the World Wide Web. They commonly use REST APIs and JSON, but are not fluent in Semantic Web technologies. Web developers can employ a CRAFTS API by submitting regular REST calls without requiring knowledge of RDF, OWL, or SPARQL. Craftsman are knowledge engineers that are in charge of setting up CRAFTS APIs. Once configured, an API provides transparent access to one or several endpoints—Figure 1 shows that API A is configured to access endpoints #0 and #1; API B accesses endpoints #1 and #2; and API C just accesses endpoint #3. CRAFTS automatically handles the translation of API calls into SPARQL queries, delivering results in JSON format.

A. Requirements

CRAFTS is designed to fulfill the requirements in Table II. CRAFTS should allow the provision of REST APIs on top of SPARQL endpoints in a domain-agnostic way (R0). The mechanism for configuring an API should be very flexible (R1), allowing a high degree of control of the data to expose through the API. Once configured, a CRAFTS API should work as a plain REST API. RDF resources will be accessible at specific URLs through HTTP methods (R2). A CRAFTS API should also allow read operations, i.e., HTTP GET, over RDF resources (R3), but also write operations such as PUT or DELETE (R4). Write operations imply the modification of RDF data in a triple store, typically supported through SPARQL Update [17]. Partial updates will be also supported through HTTP PATCH (R5). A CRAFTS API should also support parametrized SPARQL queries (R6). The aim is twofold: (1) to discover IRIs of RDF resources that can be accessed through the API, and (2) to provide a simple way of querying an endpoint without requiring knowledge of SPARQL. API body requests and responses in CRAFTS will be formatted in JSON (R7). Since API calls will propagate into SPARQL queries to the source endpoints, queries will be cached to limit future exchanges with the endpoints (R8). CRAFTS will be documented with the OpenAPI specification [16] (R9), a standard and programming language-agnostic interface description for REST APIs. Adhering to OpenAPI seems a sensible decision, given that OpenAPI is the most popular specification for REST APIs, with the largest community, and the best tooling [14]. Finally, CRAFTS will allow the configuration of APIs that transparently access one or several endpoints if desired (R10).

Table II: Requirements of CRAFTS.

| ID | Requirement |
|----|-------------|
| R0 | Generic approach for creating REST APIs over LOD |
| R1 | Flexible specification of REST APIs |
| R2 | Expose RDF resources |
| R3 | Support read operations |
| R4 | Support write operations |
| R5 | Support partial updates |
| R6 | Support parametrized queries |
| R7 | Use JSON as interchange format |
| R8 | Cache SPARQL queries |
| R9 | OpenAPI documentation |
| R10 | Access to multiple endpoints |
Table III: Specification summary of the API exposed by CRAFTS. Path parameters are marked with curly braces { } and are always required. Query parameters marked with * are required.

| ID | Operation | Path | Query parameters | Schema | Description |
|----|-----------|------|------------------|--------|-------------|
| C0 | GET       | /apis |                 | List of APIs | Get the list of available APIs |
| C1 | GET       | /apis/{apiId} |                 | CRAFTS API configuration | Get the API apiId |
| C2 | PUT       | /apis/{apiId} |             | CRAFTS API configuration | Create or replace the API apiId |
| C3 | DELETE    | /apis/{apiId} |          |             | Delete the API apiId |
| C4 | GET       | /apis/{apiId}/resource | id*, iri* | Resource model | Get the resource with IRI iri from API apiId using model id |
| C5 | PUT       | /apis/{apiId}/resource | id*, iri* | Resource model | Create or replace the resource with IRI iri from API apiId using model id |
| C6 | PATCH     | /apis/{apiId}/resource | id*, iri* | JSON PATCH | Update the resource with IRI iri from API apiId using model id |
| C7 | DELETE    | /apis/{apiId}/resource | id*, iri* |             | Delete the resource with IRI iri from API apiId using model id |
| C8 | GET       | /apis/{apiId}/resources | id*, iri*, ns, nspref | Resource model | Get a set of resources with IRIs iri from API apiId using model id. IRIs can be abbreviated by using prefix nspref and namespace ns |
| C9 | GET       | /apis/{apiId}/query | id* | Query results | Send a parametrized SPARQL query using the template id from API apiId with the desired parameter values |

B. The API of CRAFTS

In order to comply with the requirements in Table II, CRAFTS provides a uniform API that is summarized in Table III. This API is documented with the OpenAPI specification, as required (R9). Thus, the API of CRAFTS will be described using the terminology defined in OpenAPI.

A REST API supports a number of different calls, each one includes a path and an operation. A path identifies a web resource exposed by the API; it is defined with a URL relative to the server. An operation is one of the HTTP methods that will be applied to the target resource such as GET, PUT, DELETE... A call can be further qualified with different types of parameters. A path parameter is a variable part of the URL that is denoted with curly braces { }; when a client makes an API call, any path parameter has to be substituted with an actual value. Query parameters appear at the end of the request URL after a question mark ?, with different name=value pairs separated by ampersands &. Write operations (POST, PUT, and PATCH) typically require a request body with the representation of the web resource to create or update. An API call will return a response consisting of an HTTP status code and an optional response body – the latter is especially relevant for read operations through GET. Schemas are used to define the structure of request and response bodies.

API calls C0–3 in Table III are aimed for the management of CRAFTS APIs. The design of this block is relatively straightforward: call C0 serves to obtain the list of available APIs, while the remaining calls C1–3 refer to a specific CRAFTS API by using the path parameter apiId. More importantly, a CRAFTS API configuration has to be provided in the request body of call C2 so as to create or replace a CRAFTS API through a PUT operation [5]. Any valid configuration has to follow the schema in Figure 2. Specifically, a configuration is a JSON object with the following keys and values:

- **apiId**: the identifier of the API. It has to be unique for a CRAFTS site.
- **endpoints**: an array of SPARQL endpoints that serve Linked Open Data for the API. Every endpoint requires an identifier and a SPARQL URI. Further access information can be included such as a graph URI, authentication credentials, or SPARQL Update activation.
- **model**: an array of resource models that guides data exchanges with the endpoints. Every request for an RDF resource with the API (calls C4–8) must reference a resource model with an identifier id. A resource model defines a mapping of RDF data to a JSON object: datatype properties are specified in dprops, object properties in oprops, and class membership in types. As for the required parameters, label will be the key in the mapped JSON object, endpoint refers to the source of data, and iri identifies the corresponding datatype or object property. The rest of parameters are optional and will be illustrated in Section IV.
- **queryTemplates**: an array of SPARQL query templates. Each one has an id, a previously configured endpoint, an array of parameters, and a template. The latter is a Mustache template that must produce a valid SPARQL query after replacing the template placeholders with the parameters. The array of variables has to coincide with the projected variables from the SPARQL query.

---

2 https://mustache.github.io/
Figure 2: JSON schema of a CRAFTS API configuration.
Calls C4–8 in Table III are used for easily accessing Linked Open Data. They all have the same overall structure, valid for any successfully configured API in CRAFTS: path parameter apiId identifies the API, query parameter id refers to a resource model, and query parameter iri identifies the target RDF resource. The operation determines whether it will be retrieved (GET), created or replaced (PUT), updated (PATCH), or deleted (DELETE). Resource representations will be exchanged in request and response bodies, as required by the operation. They will be all formatted in JSON following the schema defined in the corresponding resource model. As for partial updates of resources (call C6), JSON PATCH is used to express the sequence of operations to apply to the target RDF resource. Since C8 allows the retrieval of a set of RDF resources in a single call, it can be used to reduce the number of C4 calls.

Finally, C9 is purposed for easily sending parametrized SPARQL queries, as well as for discovering IRIs of RDF resources to be used in calls C4–8. The Appendix includes a non-trivial CRAFTS API configuration object, while Section IV presents a thorough showcase scenario that exploits such API configuration.

C. Architecture

The logical architecture of CRAFTS is graphically depicted in Figure 3. Incoming calls are first checked if they comply with the specification in Table III. Valid calls are dispatched to the API manager, the component in charge of the management of the APIs in CRAFTS. Depending on the call type, the API manager may rely on other components in order to produce an answer. Specifically, the API manager can handle by itself calls C0, C1, and C3, corresponding to list, read, and delete operations of existing APIs in CRAFTS. Every call with a request body (C2, C5, and C6) is forwarded to the Model Validator to check the correctness of the included data. RDF resource read operations (C4 and C8) and parametrized queries (C9) are handled by the Data manager. Calls C5, C6, and C7 are RDF resource write operations and are processed by the Resource updater. Figure 4 shows the flowcharts of a representative set of the calls supported by CRAFTS (C2, C4, C5, and C9).

The Model validator is used to detect errors in the request bodies of the received calls. Note that syntactic errors in request bodies are detected upfront due to the use of JSON schema validation. Instead, this component focuses on higher-level problems. In the case of call C2 (see Figure 4), the API configuration provided in the request body is thoroughly analyzed to check wrong references to endpoints, duplicate ids and labels, and wrong references to model elements. Moreover, all the endpoints are tested with probe queries, including triple insertions and deletions if a sparqlUpdate configuration is present, as well as test queries for every resource model and query template included in the configuration. With respect to calls C5 and C6, the Model Validator checks whether the provided request bodies are compliant with the corresponding model element in the API configuration.

The Data manager handles all data exchanges with the endpoints. Regarding resource read calls (C4 and C8), the Data manager will map those requests into SPARQL queries and return JSON data in response (see Figure 4). The target model element of the API drives this data extraction process. As for the types array, a SPARQL query will be created for each element in this array using the Mustache template in Listing 1, with the iri of the request and the inferred and restrictions values of the model type element, if included. Similarly, the Mustache template in Listing 2 will be used for the dprops and oprops arrays of the model element. Answers to these queries are stored in the Data cache, so as to limit the requests to the endpoints. In this way, the Data manager first checks the Data cache before querying an endpoint. With respect to call C9 (see Figure 4), the Data manager first identifies the target query template in the API configuration, and constructs the actual query with the received parameters. The Data cache is, again, first checked for an answer hit before querying the endpoints.

Listing 1: Template query for retrieving class membership for a set of RDF resources (iris).

```
SELECT DISTINCT ?iri ?type
WHERE {
  ?iri a{{#inferred}}/rdfs:subClassOf*{{/inferred}} ?type .
  {?iri }{#restrictions} ?{(.).} {{(restrictions)}}
  FILTER (?iri IN ( {{{iri}}} ))
}
```

Listing 2: Template query for retrieving the values of a target object or datatype property (propiri) for a set of RDF resources (iris).

```
SELECT DISTINCT ?iri ?value
WHERE {
  {?iri }{#inferred} ?value .
  {?iri }{#inferred} ?iri .
  {?iri }{#restrictions} ?{(.).} {{(restrictions)}}
  FILTER (?iri IN ( {{{iri}}} ))
}
```

The Resource updater handles resource writing calls (C5, C6, and C7). As a first step for any of these calls, the Resource updater makes a resource read request to the Data...
Figure 4: Flowcharts for a subset of the calls exposed by CRAFTS. (a) Creation of a CRAFTS API (C2). (b) Retrieval of a resource (C4). (c) Parametrized SPARQL query (C9). (d) Creation or replacement of a resource (C5).
manager for the target iri (see Figure 3). Upon arrival of the corresponding JSON object, the Resource updater will prepare a sequence of graph update operations in the endpoints that will be dispatched to the Data manager. In the case of a resource deletion (C7), the obtained JSON object will be analyzed to obtain the triples to be removed in the target endpoints through DELETE DATA operations. For a fresh resource creation (C5), the request body is used to produce the triples to be included in the target endpoints through INSERT DATA operations. In the case of a resource replacement (C5 too), existing triples will be removed and then the new ones will be inserted. Regarding call C6, the strategy is somewhat different to comply with the atomicity requirement of PATCH requests [19]: (1) the operations of the patch are sequentially applied to a copy of the obtained JSON object; (2) if the sequence is valid, the Resource updater compares the two JSON objects and obtains the triples to be deleted and inserted in the endpoints; and (3) the corresponding graph update operation is dispatched to the Data manager. For any writing call, the Resource updater keeps track of the RDF resources that were modified and makes a clean cache request to the Data manager.

D. Implementation

I have coded a full functional version of CRAFTS in JavaScript for the popular Node.js platform [7]. The code is organized in several files that reflect the logical architecture in Figure 3. The implementation effort has been considerably reduced by the integration of a number of Node.js libraries. Notably, CRAFTS uses the Express.js [8] web framework to easily handle web requests. The whole API of CRAFTS (see Table III) is annotated with the OpenAPI specification [10]. This is exploited by the express-openapi-validator to perform a syntactic validation of incoming calls, again simplifying the code of CRAFTS. The OpenAPI specification of CRAFTS is also used to provide an auto-generated API documentation served with Swagger UI Express [9]. The utility functions of Underscore are employed for handling JavaScript collectons. Mustache is used for templating SPARQL queries, as described in Section III-C. A configuration file is used to adjust parameters such as the port of the Node.js server or the timespan of cached elements.

The source code of CRAFTS is available on GitHub [7]. I have also set up a live version of CRAFTS [11] for anyone who wants to use it. The OpenAPI specification of CRAFTS is browsable (and actionable) at this URL [11]. This version provides two authentication schemes: Basic [20] and Bearer authentication [21]. Basic authentication is aimed to craftsmen, although anybody with a valid email account can sign up (an email will be sent for activating the user). A registered user—typically a craftsman—can then use their credentials to make any call to CRAFTS through the Basic authentication scheme. A user will be authorized to make any call to an API of their own, e.g. updating the API (C2) or reading a resource (C4). A user will only be authorized to make read calls (C1, C4, and C8–9) of others’ APIs, but they will not be allowed to make write calls (C2–3 and C5–7) of others’ APIs.

The Bearer authentication scheme is aimed for web developers to access a CRAFTS API without requiring user registration. Every API includes read and write tokens that are only shown to the owner of the API through call C1. The read token of an API can be used to make calls C4 and C8–9 with Bearer authentication, while the write token can be used to make calls C2–3.

IV. SHOWCASE OF CRAFTS

This section illustrates the capabilities of CRAFTS in a showcase scenario. It will be demonstrated with the test site of CRAFTS [12], although any other deployment can be used. Table IV includes the namespaces employed along this showcase.

| Prefix | Namespace |
|--------|-----------|
| axis   | http://epsg.w3id.org/ontology/axis/ |
| cors   | http://epsg.w3id.org/data/cors/   |
| dbo    | http://dbpedia.org/ontology/      |
| dbr    | http://dbpedia.org/resource/      |
| ifn    | https://datos.iepnb.es/def/sector-publico/medio-ambiente/ifn/ |
| pont   | http://crossforest.eu/position/ontology/ |
| pos    | https://datos.iepnb.es/ recurso/sector-publico/medio-ambiente/ifn/position/ |
| rdfse  | http://www.w3.org/2000/01/rdf-schema# |
| schema | http://schema.org/                |
| tree   | https://datos.iepnb.es/ recurso/sector-publico/medio-ambiente/ifn/tree/ |

See footnote 10.

[1] https://nodejs.org
[2] https://expressjs.com/
[3] https://www.npmjs.com/package/express-openapi-validator
[4] https://www.npmjs.com/package/swagger-ui-express
[5] https://www.npmjs.com/package/underscore
[6] https://www.npmjs.com/package/mustache
[7] https://github.com/guiveg/crafts
[8] https://www.npmjs.com/package/express
[9] https://www.npmjs.com/package/swagger-ui-express
[10] https://www.npmjs.com/package/underscore
[11] https://crafts.gsic.uva.es/docs/
[12] See footnote 10
crossforest and dbpedia. The former is a large forestry dataset (almost 200M triples) built in the context of the EU Cross-Forest project[^1] with institutional repositories from Spain and Portugal. The latter endpoint is the well-known English DBpedia.[^22]. The configuration includes a model with four different elements: Tree, Position, Species, and DbpediaSpecies. There is also a query template, treesinbox. Since the id of the API in the configuration is test, the URL of this PUT call should be [https://crafts.gsic.uva.es/apis/test](https://crafts.gsic.uva.es/apis/test).

As described in Section III-C, the Model Validator will check that the configuration is correct, including probe queries to crossforest and dbpedia.

b) Submitting a parametrized query: Once the API test is set up, I can use the query template treesinbox, aimed for discovering trees of a specific species in a bounding box. For this showcase I will use the parameter values in Table V. Call C9 is prepared for parametrized queries, so the resulting URL will be [https://crafts.gsic.uva.es/apis/test/query? id=treesinbox&lngwest=-6&lngeast=-3&latnorth=40& latsouth=37&limit=5&species=https://datos.iepnb.es/def/sector-publico/medio-ambiente/ifn/Species43](https://crafts.gsic.uva.es/apis/test/query?id=treesinbox&lngwest=-6&lngeast=-3&latnorth=40&latsouth=37&limit=5&species=https://datos.iepnb.es/def/sector-publico/medio-ambiente/ifn/Species43)

This request is processed by the Data manager that first obtains the query template treesinbox from the API configuration. Next, the Data manager prepares a JSON object with the query parameters enclosed in the URL (species, lngwest, lngeast, latnorth, latsouth, and limit). The Mustache library is then called with the query template and the parameter object, obtaining the SPARQL query in Listing 3. Note that all the parameters of treesinbox are optional, so this query template uses sections such as `{{#latsouth}}` FILTER (`?tlat > {{latsouth}}`) `{{/latsouth}}` to check the existence of a latsouth key in the parameter object and then include the text embedded in the section, substituting latsouth with the value of latsouth in the parameter object, i.e., FILTER (`?tlat > latsouth`). Since treesinbox is linked to the crossforest endpoint, the Data manager will first check if there is a hit of this query for crossforest in the Data cache. In case of a miss, the Data manager will submit the query to the crossforest endpoint. Query results will be returned using the standard serialization of SPARQL results in JSON format.[^23].

| Parameter | Value            |
|-----------|------------------|
| id        | treesinbox       |
| species   | ifn:Species43    |
| lngwest   | -6               |
| lngeast   | -3               |
| latnorth  | 40               |
| latsouth  | 37               |
| limit     | 5                |

[^1]: [https://crossforest.eu/](https://crossforest.eu/)

Table V: Sample parameter values for testing treesinbox.

For the sake of readability, the actual answer to the query in Listing 3 is presented in Table VI.

| tree      | tpsps | tlat  | tlng |
|-----------|-------|-------|------|
| tree:10-2898-A-1-2 | ifn:Species43 | 39.393602 | -5.648659 |
| tree:13-1518-A-1-3 | ifn:Species43 | 38.443551 | -4.273078 |
| tree:18-1252-A-1-31 | ifn:Species43 | 37.116059 | -3.350195 |
| tree:45-1089-A-1-10 | ifn:Species43 | 39.577024 | -4.433454 |
| tree:23-0073-A-1-18 | ifn:Species43 | 38.365372 | -3.688194 |

[^2]: Listing 3: SPARQL query obtained after applying parameters in Table V to the Mustache template treesinbox in the Appendix.

```sparql
SELECT DISTINCT ?tree ?tpsps ?tlat ?tlng WHERE {
?tree a ifn:Tree; ?tpsps ;
post:hasPosition ?pos .
?pos post:hasCoordinateReferenceSystem crs:4326 ;
axis:1 ?tlat ;
axis:2 ?tlng ;
?tpsps rdfs:subClassOf+ ifn:Plantae .
FILTER (?tlat > 37) FILTER (?tlng > -6) FILTER (?tlat < 40) FILTER (?tlng < -3) }
LIMIT 5
```

[^3]: Table VI: Answer to the SPARQL query in Listing 3.

[^4]: This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see [https://creativecommons.org/licenses/by/4.0/](https://creativecommons.org/licenses/by/4.0/)

[^5]:[^14]: Species data can be also embedded in a tree by setting `embed=true` in the `species` element of the `types` array. This will result on new queries, specifically the ones in Table VIII to produce Listing 5.
Table VII: SPARQL queries produced for obtaining the output in Listing 4.

| Key                        | Endpoint | Query |
|----------------------------|----------|-------|
| species                    | crossforest | SELECT DISTINCT ?iri ?type WHERE {
|                            |          | ?iri a ?type .
|                            |          | ?type rdfs:subClassOf* ifn:Plantae .
|                            |          | FILTER (?iri IN ( tree:10-2898-A-1-2 )) } |
| diameter_mm                | crossforest | SELECT DISTINCT ?iri ?value WHERE {
|                            |          | ?iri ifn:hasDBH1InMillimeters ?value .
|                            |          | FILTER (?iri IN ( tree:10-2898-A-1-2 )) } |
| height_M                   | crossforest | SELECT DISTINCT ?iri ?value WHERE {
|                            |          | ?iri ifn:hasTotalHeightInMeters ?value .
|                            |          | FILTER (?iri IN ( tree:10-2898-A-1-2 )) } |
| position                   | crossforest | SELECT DISTINCT ?iri ?value WHERE {
|                            |          | ?iri pont:hasPosition ?value .
|                            |          | ?value pont:hasCoordinateReferenceSystem crs:4326 .
|                            |          | FILTER (?iri IN ( tree:10-2898-A-1-2 )) } |
| position.latWGS84          | crossforest | SELECT DISTINCT ?iri ?value WHERE {
|                            |          | ?iri axis:1 ?value .
|                            |          | FILTER (?iri IN ( pos:10-2898-A-1-2-23030-4326 )) } |
| position.lngWGS84          | crossforest | SELECT DISTINCT ?iri ?value WHERE {
|                            |          | ?iri axis:2 ?value .
|                            |          | FILTER (?iri IN ( pos:10-2898-A-1-2-23030-4326 )) } |

Listing 4: Answer to the resource request with id=Tree and iri=tree:10-2898-A-1-2.

```json
{
"iri": "https://datos.iepnb.es/recurso/sector-publico/medio-ambiente/ifn/tree/10-2898-A-1-2",
"species": "https://datos.iepnb.es/def/sector-publico/medio-ambiente/ifn/Species43",
"diameter_mm": 327,
"height_M": 6.9,
"position": {
"iri": "https://datos.iepnb.es/recurso/sector-publico/medio-ambiente/ifn/position/10-2898-A-1-2-23030-4326",
"latWGS84": 39.393602,
"lngWGS84": -5.648659
}
}
```

d) Requesting a resource (two endpoints):

Similarly to the previous case, I can make an RDF resource read call (C4) for ifn:Species43 and model Species. The resulting URL is [https://crafts.gsic.uva.es/apis/test/resource?id=Species&iri=https://datos.iepnb.es/def/sector-publico/medio-ambiente/ifn/Species43](https://crafts.gsic.uva.es/apis/test/resource?id=Species&iri=https://datos.iepnb.es/def/sector-publico/medio-ambiente/ifn/Species43)

The interesting thing here is that data is coming from two endpoints, as shown in Table VIII. In the Cross-Forest dataset, tree species are linked to resources in other datasets using schema:sameAs. This is exploited in the element dbpedia of the oprops array that embeds data from DbpediaSpecies, as defined in the configuration. db:Quercus_pyrenaica is discovered for ifn:Species43, enabling the formulation of queries dbpedia:comment and dbpedia:image. The Data manager integrates all the received answers to produce the JSON object in Listing 5.

Listing 5: Answer to a resource request with id=Species and iri=ifn:Species43.

```json
{
"iri": "https://datos.iepnb.es/def/sector-publico/medio-ambiente/ifn/Species43",
"scientificName": {
"la": "Quercus pyrenaica"
},
"vulgarName": {
"en": "Pyrenean oak",
"es": "Rebollo"
},
"de": "Pyrenaen-Eiche"
},
"fr": "Chene tauzin"
},
"dbpedia": {
"iri": "http://dbpedia.org/resource/Quercus_pyrenaica",
"comment": [
{"es": "Quercus pyrenaica, llamado vernacularmente melojo en algunas partes..."},
{"en": "Quercus pyrenaica, commonly known as Pyrenean oak ..."}],
"image": "http://commons.wikimedia.org/wiki/Special:FilePath/Melojar_soto.JPG?width=300"
}
```

e) Creating a resource:

I will use call C5 to create an RDF resource with IRI tree:0 and model Tree. In the request body I include the JSON object in Listing 4 while the target URL is [https://crafts.gsic.uva.es/apis/test/resource?id=Tree&iri=https://datos.iepnb.es/recurso/sector-publico/medio-ambiente/ifn/tree/0](https://crafts.gsic.uva.es/apis/test/resource?id=Tree&iri=https://datos.iepnb.es/recurso/sector-publico/medio-ambiente/ifn/tree/0)

The remaining examples of this showcase requires setting credentials for SPARQL Update in the crossforest endpoint.
Table VIII: SPARQL queries produced for obtaining the output in Listing 5.

| Key             | Endpoint | Query |
|-----------------|----------|-------|
| scientificName  | crossforest | SELECT DISTINCT ?iri ?value WHERE { ?iri ifn:hasAcceptedName/ifi:name ?value . FILTER (?iri IN ( ifn:Species43 )) } |
| vulgarName      | crossforest | SELECT DISTINCT ?iri ?value WHERE { ?iri ifn:vulgarName ?value . FILTER (?iri IN ( ifn:Species43 )) } |
| dbpedia         | crossforest | SELECT DISTINCT ?iri ?value WHERE { ?iri schema:sameAs ?value . FILTER (lang(?value) = "es" OR lang(?value) = "en" OR lang(?value) = ") FILTER (?iri IN ( dbr:Quercus_pyrenaica )) ) |
| dbpedia.comment | dbpedia   | SELECT DISTINCT ?iri ?value WHERE { ?iri rdfs:comment ?value . FILTER (?iri IN ( ifn:Species43 )) } |
| dbpedia.image   | dbpedia   | SELECT DISTINCT ?iri ?value WHERE { ?iri dbo:thumbnail ?value . FILTER (?iri IN ( ifn:Species43 )) } |

Listing 6: Request body of a sample resource creation request with id=Tree and iri=tree:0.

```
{  
"iri": "https://datos.iepnb.es/recurso/sector-publico/medio-ambiente/ifi/tree/0",  
"species": "https://datos.iepnb.es/def/sector-publico/medio-ambiente/ifi/Species23",  
"position": {  
"iri": "https://datos.iepnb.es/recurso/sector-publico/medio-ambiente/ifi/position/0",  
"latWGS84": 40,  
"lngWGS84": 0  
}  
}
```

This can be seen as the reverse of a read resource call. The main focus here is the correctness of the included request body that must adhere to the API model. Thus, the Model validator checks that species and position exist elsewhere in the types, dprops, and oprops arrays of Tree. It also finds that position refers to a Position model with latWGS84 and lngWGS84 in the corresponding dprops array. After this validation, the Resource updater will send a read request for resource tree:0 and model Tree to the Data manager. If the response contains any data, the Resource updater will prepare DELETE DATA operations with the triples to be removed. As this is a fresh resource creation, the Resource updater only needs to produce the set of triples to be inserted. Listing 7 shows the INSERT DATA operation that will be sent. Since all the insertions apply to the crossforest endpoint, a single operation is needed. Also note that multiple RDF resources can be created with a single call—in this case tree:0 and pos:0.

Listing 7: SPARQL Update query to fulfill the request in Listing 6

```
INSERT DATA {  
tree:0 a ifn:Species23 ;  
pont:hasPosition pos:0 .  
pos:0 axis:1 40 ;  
axis:2 0 .  
}
```

f) Updating a resource: I am going to update the tree created in the previous step, tree:0, with diameter and height measures. For this purpose I will use call C6 with id=Tree, iri=tree:0, and Listing 8 in the request body. The target URL is thus the same as in the previous step.

Listing 8: JSON PATCH for updating tree:0.

```
{  
"op": "add",  
"path": "/diameter_mm",  
"value": 100  
},  
{  
"op": "add",  
"path": "/height_M",  
"value": 10  
}
```

Listing 8 is a JSON PATCH that is checked with the Model validator. Since diameter_mm and height_M are defined in the dprops array of Tree and their values are literals, the patch is validated and the request is forwarded to the Resource updater. This component sends a read request for the target resource to the Data manager. Then, it creates a copy of the returned JSON object (see Listing 6) and applies each operation included in the patch. If no errors are found, the Resource updater obtains the triples to be deleted and inserted by comparing the original and the modified JSON objects. In this case there are no removals, so the update is completed with the INSERT DATA operation in Listing 9.

Listing 9: SPARQL Update query to fulfill the request in Listing 8

```
INSERT DATA {  
tree:0 ifn:hasDBH1InMillimeters 100 ;  
ifn:hasTotalHeightInMeters 10 .  
}
```

g) Deleting a resource: I conclude the showcase with the deletion of tree:0. This can be done with call C7, again using the URL https://crafts.gsic.uva.es/apis/test/resource?id=Tree&iri=https://datos.iepnb.es/recurso/sector-publico/medio-ambiente/ifi/tree/0. No request body is needed here, so the Resource updater can easily handle this call. As in previous resource write operations, it sends a read request for resource tree:0.
Table IX: Uptake of the test site of CRAFTS (February to December 2021).

| Item                  | Value |
|-----------------------|-------|
| # of active APIs      | 8     |
| # of LOD users        | 2,422 |
| # of sessions         | 6,903 |
| # of API calls        | 259,667 |

and model Tree to the Data manager. With the obtained response, the Resource updater prepares the triples to be removed and sends the DELETE DATA operation in Listing 10. Note that the deletion is not propagated to other RDF resources such as pos:0. Deleting this latter resource will require another explicit C7 call with iri=pos:0 and id=Position.

Listing 10: SPARQL Update query for deleting tree:0.

```
define a instance of ifn:Species23 ;
pon:hasPosition pos:0 ;
ifn:hasDBH1InMillimeters 100 ;
ifn:hasTotalHeightInMeters 10 .
```

V. UPTAKE OF CRAFTS

The test site of CRAFTS\textsuperscript{16} was launched in February 2021. Since then, different projects are using CRAFTS through eight active APIs. Three of them are based on previous works, while the remaining five correspond to new projects based on CRAFTS from the beginning. Activity in the test site is tracked with the Google Analytics platform\textsuperscript{17}. Table IX summarizes the collected data (obtained in December 2021). Remarkably, more than 2.4K LOD users have accessed CRAFTS in 6.9K sessions and making 259.7K calls. The bulk of this workload corresponds to three APIs: Forest Explorer (87.5% of all the calls), EducaWood (9.0%), and LocalizARTE (0.7%). There is a dedicated web application for each of these APIs, as described below.

Forest Explorer \textsuperscript{24} is a web application that offers an interactive map for browsing forestry data from the Cross-Forest and English DBpedia endpoints. This application was refactored in March 2021 for using a CRAFTS API instead of a custom SPARQL query formulation logic. The refactoring was motivated by (1) major changes in the Cross-Forest ontology and (2) the release of new data (particularly evident with the inclusion of forestry data from Portugal). The CRAFTS API for Forest Explorer includes: three bootstrapping query templates for obtaining all available species, soil usages, and provinces; three additional query templates for discovering patches, plots, and trees in a geographical area; and a model with 11 elements (Tree, Species, Province...). Since I am the developer of the original and the refactored versions of Forest Explorer, it is possible to analyze the impact of using CRAFTS.

The refactoring required a modest effort, two working days, dedicated to replace the custom SPARQL querying code with common REST calls to the new CRAFTS API. More importantly, the refactored version is much simpler, despite the increased complexity in the ontology and the increased size of the Cross-Forest dataset (71GB vs 11GB). A source lines of code (SLOC) analysis of the two versions with CLOC\textsuperscript{18} revealed that the CRAFTS version of Forest Explored employs 10.0% SLOC less. Focusing on the component in charge of data exchanges with the endpoints alone, SLOC reduction is 28.8%.

EducaWood \textsuperscript{25} is a socio-semantic annotation system intended for environmental learning in Secondary and Higher Education. The CRAFTS API of EducaWood is configured for retrieving land cover maps and forestry data from the Cross-Forest endpoint, as well as species information from the English DBpedia. This API uses a third endpoint to publish annotations of trees and other ecosystem structures such as dead wood and microhabitats. While consumption of forestry data from the Cross-Forest endpoint is similar to the case of Forest Explorer, the creation of social tree annotations is the most distinctive feature of EducaWood.

LocalizARTE \textsuperscript{26} is a supporting web application of Casual Learn \textsuperscript{27}, a ubiquitous learning mobile application in the Cultural Heritage domain. Teachers can use LocalizARTE to annotate sites of interest and educational activities associated to them, e.g. take a photo of the southern rose window of the Cathedral of León. Learners can then use Casual Learn with their mobile devices to carry out educational activities in close proximity. LocalizARTE makes use of a CRAFTS API to reduce the effort required for the creation of Cultural Heritage sites. Specifically, the API uses the English and Spanish DBpedia endpoints, defines two query templates, and includes three elements: Place, Image, and Category. The template queries are purposed for finding the closest sites of interest to a given point. A subsequent resource read call serves to extract the coordinates, label, comment, image, and Wikipedia categories of the found sites. A teacher can reuse all this data when annotating a Cultural Heritage site.

EducaWood and LocalizARTE are developed by two Master students with no expertise in Semantic Web technologies. In these two cases I played the role of craftsman and configured the corresponding CRAFTS APIs. I prepared a short guide for using each API, similar to the showcase in Section IV but succincter. The students followed the guide and they had no problems in using the APIs to build LocalizARTE and EducaWood (even using the write calls of the API in this latter case).

The majority of LOD users are inadvertently making API calls to CRAFTS through the provided web applications. The distribution of LOD users roughly corresponds to the reported percentages of calls per API. In the case of Forest Explorer, 64% of the users are located in Spain, 10% in Portugal, 3% in United States, and the rest in other

\textsuperscript{16}See footnote 10
\textsuperscript{17}https://www.google.com/analytics
\textsuperscript{18}https://github.com/AlDanial/cloc
countries around the world. As the Cross-Forest dataset includes data from Iberian forests, the majority of users are forestry professionals from Spain and Portugal and outside our contact network—some of them send us feedback by social media and through a web form that is requested after some significant activity in Forest Explorer. With respect to EducaWood and LocalizARTE, the user base is smaller (~100 users) and more local (mainly Spain), as their promotion is not yet initiated.

To conclude this section, Table X shows the distribution of call types and their average latency in the test site of CRAFTS. The API of Forest Explorer accounts for the immense majority of C8 and C9 calls. Template queries (C9) in Forest Explorer correspond to relatively complex geospatial requests for retrieving features (i.e., patches, plots, and trees) in a bounding box. Isolating the template query for gathering patches, the latency of a fresh request of this type is ~1.5 seconds, while cached requests are served in ~6 milliseconds. This illustrates that the generation cost of a SPARQL query is negligible compared to the cost of querying an endpoint, thus motivating the use of a cache. C8 calls are very responsive and efficient, given that Forest Explorer tries to pack 100 resources in a single C8 call. The API of EducaWood is responsible for all write resource calls (C5, C6, and C7) with a good latency. Creating or updating an API (C2) is an infrequent operation, but it is slow. This is because CRAFTS sends SPARQL probes for testing that the API configuration is correct, e.g., 47 queries are sent when creating the API of Forest Explorer.

### VI. Discussion

CRAFTS has demonstrated to be very flexible for accessing LOD. When configuring an API, a craftsman has a high degree of control of which data to expose and in which form. The configuration goal should be to provide meaningful data access to LOD users. This implies filtering out unnecessary classes and properties of the sources in the API model. The use of embedding can be very helpful to integrate different RDF subgraphs into a single JSON object. This is quite desirable for web developers so as to
crowd all the data about a resource with a single read call (C4), e.g., Listing 4 illustrates how all the relevant data about a tree, including its position, is obtained with a single API call. In this way, web developers do not need to be aware of the existence of different RDF subgraphs, and the result is a familiar JSON object. Moreover, an API can be configured to seamlessly integrate data from several endpoints; Listing 3 showcases a non-trivial example of species data integration from the Cross-Forest and English DBpedia endpoints.

The flexibility of CRAFTS comes at a price. When producing an API configuration, the craftsman should have sufficient knowledge of the underlying sources so as to ensure that RDF data is mapped to JSON structures as required. Domain experts and web developers should also participate in the creation of an API by identifying which data to extract and in which particular form. In this regard, APIs can be built in an incremental way, testing intermediate versions, and gathering feedback from the different user groups at different stages. Indeed, web developers can skip many details of an API, such as property IRIs and restrictions of models, or the Mustache templates of parametrized queries.

The API of CRAFTS is completely generic, domain-agnostic, and predictable. CRAFTS includes a very thorough validation process to assist the creation of API configurations; this is supported through the use of OpenAPI for detecting syntactic errors, as well as through the checks of the Model validator component. The OpenAPI specification is also exploited to expose an interactive frontend of the API contents. As a result, a CRAFTS API can be accessed as any regular REST API. Web developers can easily send parametrized SPARQL queries by selecting a template in an API and setting appropriate parameter values, as demonstrated in Section V. They will discover IRIs of RDF resources in the obtained answers, enabling the usage of regular REST API calls for reading and writing Linked Open Data resources. The showcase scenario and the APIs introduced in Section V give a glimpse of the versatility of CRAFTS. Noteworthy, two Master students with no training on Semantic Web topics are developing web applications on top of CRAFTS APIs.

Alternatives to CRAFTS (see Section II) also require knowledge engineers to configure APIs. In the case of RAMOSE, grile, and BASIL, somebody needs to author the parametrized SPARQL queries using custom conventions, while CRAFTS employs Mustache as a well-known templating mechanism. Regarding OBA, the use of an ontology-driven API generator may reduce the need of knowledge engineers for creating APIs, but in practice some customization is needed (as discussed in [8]). Moreover, the scope of an API does not necessarily match the scope of an ontology, e.g., the Forest Explorer API only gathers tree species information from the English DBpedia, so it is impractical to take the whole DBpedia ontology for this purpose.

---

19More than one week using the application and a session length of more than five minutes.

20See footnote 11
While OBA and CRAFTS share several features (see Table I), an important difference is that the former employs one-to-one mappings between an API call and a SPARQL query. In contrast, CRAFTS uses one-to-many mappings, as illustrated in Listings 5 and 6; this allows a more granular operation that enables the integration of data from several endpoints – this feature is not currently supported by OBA. Further, configuring an API model in CRAFTS is not as demanding as authoring SPARQL queries, since the former basically entails the inclusion of datatype and object properties; more fine-grained decisions can be made to support embedding and restrictions, as illustrated in Section IV. Other relevant differences include the native support for caching in CRAFTS and partial updates of resources through PATCH requests.

VII. Conclusion and future work

CRAFTS is a tool that allows the provision of REST APIs over LOD. Craftsmen are knowledge engineers that can set up APIs with a high degree of flexibility. CRAFTS APIs can be configured to transparently access multiple triple stores, supporting read and write operations over RDF resources, encapsulating parametrized SPARQL queries, and delivering results in JSON format. Since web developers are already used to REST APIs and JSON, they can employ CRAFTS APIs to access LOD.

Ongoing pilots with CRAFTS demonstrates the usefulness of this tool for supporting the consumption, integration, and generation of LOD. Future work includes the exploitation of CRAFTS in a number of application cases, especially in the forestry and educational domains. User studies with web developers are planned to gather more evidence about their difficulties employing CRAFTS APIs. Bootstrapping approaches for automating the generation of API configurations are also worth exploring, possibly using a combination of ontology- and data-driven strategies. New functionalities are also envisioned, such as adding support for other popular formats like GraphQL.

ACKNOWLEDGEMENTS

This work has been partially funded by the European Commission through Cross-Forest (CEF 2017-EU-1A-0140) and Virtual Forests (Erasmus+ 2020-I-ES01-KA226-HE-095836) and Spanish Ministry of Science and Innovation through H2O (PID2020-112584RB-C32) project grants.

APPENDIX

Example of a CRAFTS API configuration

```json
{
  "apiId": "test",
  "endpoints": [
    {
      "id": "crossforest",
      "sparqlURI": "https://crossforest.gsic.uva.es/sparql",
      "graphURI": "http://crossforest.eu",
      "httpMethod": "GET"
    }
  ],
  "model": [
    {
      "id": "Tree",
      "oprops": [
        {
          "label": "position",
          "targetId": "Position",
          "iri": "http://crossforest.eu/position/ontology/hasPosition",
          "embed": true,
          "restrictions": ["?value <http://crossforest.eu/position/ontology/hasCoordinateReferenceSystem> <http://epsg.w3id.org/data/crs/4326> ."]
        },
        {
          "endpoint": "crossforest"
        }
      ],
      "dprops": [
        {
          "label": "diameter_mm",
          "iri": "https://datos.iepnb.es/def/sector-publico/medio-ambiente/ifn/hasDBH1InMillimeters",
          "endpoint": "crossforest"
        },
        {
          "label": "height_M",
          "iri": "https://datos.iepnb.es/def/sector-publico/medio-ambiente/ifn/hasTotalHeightInMeters",
          "endpoint": "crossforest"
        }
      ],
      "types": [
        {
          "label": "species",
          "targetId": "Species",
          "endpoint": "crossforest"
        },
        {
          "label": "scientificName",
          "targetId": "ScientificName",
          "endpoint": "crossforest"
        }
      ]
    }
  }
}
```

[2] https://graphql.org/
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160610, IEEE Access

REFERENCES

[1] T. Berners-Lee, “Linked Data – Design Issues,” 2006, URL: http://www.w3.org/DesignIssues/LinkedData.html revised 18-06-2009, last visited January 2022.

[2] C. Bizer, T. Heath, and T. Berners-Lee, “Linked data – the story so far,” International Journal on Semantic Web and Information Systems, Special Issue on Linked Data, vol. 5, no. 3, pp. 1–22, 2009, DOI: 10.4018/jswis.2009081901.

[3] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonckx, and P. Colpaert, “Triple Pattern Fragments: a low-cost knowledge graph interface for the Web,” Journal of Web Semantics, vol. 37–38, pp. 184–206, Mar. 2016, DOI: 10.1016/j.jwebsem.2016.03.003.

[4] M. Daquino, I. Heibi, S. Peroni, and D. Shotton, “Creating restful APIs over SPARQL endpoints with RAMOSE,” Semantic Web, vol. 13, no. 2, pp. 195–213, 2022, DOI: 10.3233/SW-210439.

[5] A.-S. Dadzie and E. Pietriga, “Visualisation of linked data – reprise,” Semantic Web, vol. 8, no. 1, pp. 1–21, 2017, DOI: 10.3233/SW-160249.

[6] R. T. Fielding, “Architectural styles and the design of networked software architectures,” Ph.D. dissertation, University of California, Irvine, 2000.

[7] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format,” Internet Engineering Task Force (IETF), Standards Track RFC 8259, Dec. 2017, URL: https://datatracker.ietf.org/doc/html/rfc8259, last visited January 2022.

[8] D. Garijo and M. Osorio, “OBA: An ontology-based framework for creating REST APIs for knowledge graphs,” in Proceedings of the 19th International Semantic Web Conference (ISWC 2020), ser. LNCS, H. Sack, G. Rizzo, N. Steinmetz, D. Mladenic, S. Auer, and C. Lange, Eds., vol. 12507, Cham, Switzerland: Springer, Nov. 2020, pp. 48–64.

[9] A. Meronio-Penuela and R. Hoekstra, “grlc makes GitHub taste like linked data APIs,” in Proceedings of the 13th European Semantic Web Conference (ESWC 2016), ser. LNCS, H. Sack, A. Polleres, O. Seneviratne, and L. Kagal, Eds., vol. 9670, Portoroz, Slovenia, Jun. 2015, pp. 22–32, 2017, DOI: 10.3233/SW-160249.

[10] E. Daga, L. Panziera, and C. Pedrinaci, “A BASILar approach for building web APIs on top of SPARQL endpoints,” in Proceedings of the Third Workshop on Services and Applications over Linked APIs and Data (SALAD2015), vol. 1359, Portoroz, Slovenia, Jun. 2015, pp. 22–32, 2017, DOI: 10.3233/SW-160249.

[11] S. Speicher, J. Arwe, and A. Malhotra, “Linked Data Platform 1.0,” W3C, Recommendation, Feb. 2015, URL: http://www.w3.org/TR/2015/REC-ldp-20150226/ last visited January 2022.

[12] G. Kellogg, P. A. Champin, and D. Longley, “JSON-LD 1.1: A JSON-based Serialization for Linked Data,” W3C, Recommendation, Jul. 2020, URL: https://www.w3.org/TR/2020/REC-json-ld11-20200716/ last visited January 2022.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
[13] P. Lisena, A. Meroño-Peñuela, T. Kuhn, and R. Troncy, “Easy web api development with sparql transformer,” in Proceedings of the 18th International Semantic Web Conference (ISWC 2019), ser. LNCS, C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois, and F. Gandon, Eds., vol. 11779. Cham, Switzerland: Springer, Oct. 2019, pp. 454–470.

[14] T. Johnson, “Documenting APIs: A guide for technical writers and engineers,” 2020, URL: https://idratherbewriting.com/learnapidoc/ last visited January 2022.

[15] R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,” Internet Engineering Task Force (IETF), Standards Track RFC 7231, Jun. 2014, URL: https://datatracker.ietf.org/doc/html/rfc7231 last visited January 2022.

[16] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky, and U. Sarid, “OpenAPI Specification,” OPENAPI initiative, Tech. Rep. Version 3.0.3, Feb. 2020, URL: https://spec.openapis.org/oas/v3.0.3 last visited January 2022.

[17] P. Geaon, A. Passant, and A. Polleres, “SPARQL 1.1 Update,” W3C, Recommendation, Mar. 2013, URL: http://www.w3.org/TR/2013/REC-sparql11-update-20130321/ last visited January 2022.

[18] P. Bryan, “JavaScript Object Notation (JSON) Patch,” Internet Engineering Task Force (IETF), Standards Track RFC 6902, Apr. 2013, URL: https://datatracker.ietf.org/doc/html/rfc6902 last visited January 2022.

[19] L. Dusseauit, L. Lab, and J. Snell, “PATCH Method for HTTP,” Internet Engineering Task Force (IETF), Standards Track RFC 5789, Mar. 2010, URL: https://datatracker.ietf.org/doc/html/rfc5789 last visited January 2022.

[20] J. Reschke, “The ‘Basic’ HTTP Authentication Scheme,” Internet Engineering Task Force (IETF), Standards Track RFC 7617, Sep. 2015, URL: https://datatracker.ietf.org/doc/html/rfc7617 last visited February 2022.

[21] M. Jones and D. Hardt, “The OAuth 2.0 Authorization Framework: Bearer Token Usage,” Internet Engineering Task Force (IETF), Standards Track RFC 6750, Oct. 2012, URL: https://datatracker.ietf.org/doc/html/rfc6750 last visited January 2022.

[22] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer et al., “Dbpedia – a large-scale, multilingual knowledge base extracted from wikipedia,” Semantic Web, vol. 6, no. 2, pp. 167–195, 2015, DOI: 10.3233/SW-140134.

[23] A. Seaborne, K. Grant Clark, L. Feigenbaum, and E. Torres, “SPARQL 1.1 Query Results JSON Format,” W3C, Recommendation, Mar. 2013, URL: http://www.w3.org/TR/2013/REC-sparql11-results-json-20130321/ last visited January 2022.

[24] G. Vega-Gorgojo, J. M. Gímenez-Garcia, C. Ordonez, and F. Bravo, “Pioneering easy-to-use forestry data with forest explorer,” Semantic Web, vol. 13, no. 2, pp. 147–162, 2022, DOI: 10.3233/SW-210430.

[25] J. Andrade-Hoz, G. Vega-Gorgojo, I. Ruano-Benito, M. L. Bote-Lorenzo, J. I. Asensio-Pérez, F. Bravo, and C. Ordoñez, “Edu-cwood: a socio-semantic annotation system for environmental education,” in Proceedings of the 16th European Conference on Technology Enhanced Learning (EC-TEL 2021), ser. LNCS, vol. 12884. Cham, Switzerland: Springer, Sep. 2021, pp. 368–372.

[26] P. García-Zarza, A. Ruiz-Calleja, M. L. Bote-Lorenzo, G. Vega-Gorgojo, E. Gómez-Sánchez, and J. I. Asensio-Pérez, “Hacia la anotación y realización de tareas de aprendizaje ubicuo en el contexto de historia del arte,” in Proceedings of the XV Jornadas de Ingeniería Telemática (JITEL 2021), Oct. 2021, pp. 164–167.

[27] A. Ruiz-Calleja, P. García-Zarza, G. Vega-Gorgojo, M. Bote-Lorenzo, E. Gómez-Sánchez, J. Asensio-Pérez, S. Serrano-Iglesias, and A. Martínez-Monés, “Casual learn: A linked data-based mobile application for learning about local cultural heritage,” Semantic Web, pp. 1–15, 2021, in press. DOI: 10.3233/SW-212907.