# CRAFTS: Configurable REST APIs For Triple Stores

Guillermo Vega-Gorgojo [a,b]
[a] *Group of Intelligent and Cooperative Systems, Universidad de Valladolid, Spain*
[b] *Departamento de Teoría de la Señal y Comunicaciones e Ingeniería Telemática, E.T.S. Ingenieros de Telecomunicación, Universidad de Valladolid, Spain*
*E-mail: guiveg@tel.uva.es*

**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. Web developers can then use a CRAFTS API to read and write Linked Open Data. CRAFTS automatically handles the translation of API calls into SPARQL queries, delivering results in JSON format. 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 seven different applications, with more than 940 users, and more than 106K API calls.

Keywords: REST API, triple stores, Linked Open Data, SPARQL, data access

## 1. Introduction

With the embracement 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.[1] While there are different ways of accessing Linked Open Data [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 [4, 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 API 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], 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. Novel contributions include:

- A uniform API for managing CRAFTS APIs, sending parametrized SPARQL queries, and performing read and write operations over RDF resources (Section 2.2).

[1] https://lod-cloud.net/

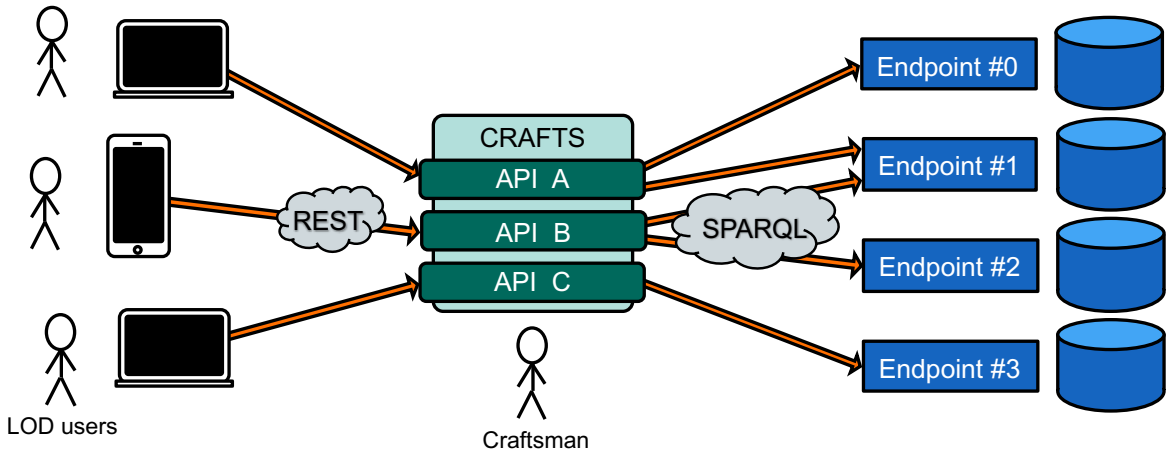


Figure 1. Overview of CRAFTS.

– An architecture with the logical components of CRAFTS for providing the desired functionalities (Section 2.3).
– A reference implementation of CRAFTS, available in a GitHub repository. A working test site of CRAFTS is also included (Section 2.4).
– A showcase scenario (Section 3) and a brief description of the main applications that currently exploit CRAFTS (Section 4).
– A discussion of CRAFTS and its relationship to other competing approaches (Section 5).

The rest of the paper is organized as follows: Section 2 provides a technical description of CRAFTS, including its requirements, a description of the API of CRAFTS, its logical architecture, and implementation details. Section 3 presents a showcase scenario. Section 4 reports about the uptake of CRAFTS so far. The paper ends with a discussion and future work lines in Section 5.

## 2. Technical description of CRAFTS

Configurable REST APIs For Triple Stores (CRAFTS) is a novel tool for accessing Linked Open Data through REST APIs. As graphically depicted in Figure 1, CRAFTS is purposed for two types of users: *craftsmen* and *LOD users*. Craftsmen are knowledge engineers that are in charge of setting up APIs. Once configured, an API provides transparent access to one or several endpoints. CRAFTS automatically handles the translation of API calls into SPARQL queries, delivering results in JSON format. Thus, LOD users can employ a CRAFTS API through regular REST calls without requiring knowledge of RDF, OWL, or SPARQL.

### *2.1. Requirements*

CRAFTS is designed to comply with the following requirements:

**R1** *Generic approach for creating REST APIs over Linked Open Data.* CRAFTS should allow the provision of REST APIs on top of SPARQL endpoints. The mechanism for configuring an API should be domain-agnostic and flexible, allowing a high degree of control of the data to expose through the API.

**R2** *Expose RDF resources.* Once configured, a CRAFTS API should work as a plain REST API. Specifically, RDF resources will be accesible at specific URLs through HTTP methods [11]. API body requests and responses in CRAFTS will be formatted in JSON, the main interchange format in modern REST APIs [12].

**R3** *Support read and write operations.* A CRAFTS API should not be limited to read operations, i.e. GET calls. A CRAFTS API should also allow write operations such as PUT, PATCH, or DELETE. Write operations imply the modification of RDF data in a triple store, typically supported through SPARQL Update [13]. As a result, CRAFTS will only allow a write operation if the API configura-

Table 1
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`* | Resource model excerpt | 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`*, `iris`*, `ns`, `nspref` | Resource model | Get a set of resources with IRIs `iris` 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 |

tion includes valid authentication credentials for using SPARQL Update in the target endpoints.

**R4** *Support parametrized SPARQL queries.* A CRAFTS API should support parametrized SPARQL queries. 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.

**R5** *Access to multiple triple stores.* A single CRAFTS API can be configured to transparently access one or several endpoints if desired.

### *2.2. The API of CRAFTS*

In order to comply with the requirements in Section 2.1, CRAFTS provides a uniform API that is summarized in Table 1. This API is documented with the OpenAPI specification [14], 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 [12]. 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 1 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. 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 3.
- `queryTemplates`: an array of SPARQL query templates. Each one has an `id`, a previously configured `endpoint`, an array of `parameters`, an array of `variables`, and a `template`. The latter is a Mustache[2] template that must produce a valid SPARQL query after replacing the template placeholders with the parameters. The array of `variables` have to coincide with the projected variables from the SPARQL query.

[2]https://mustache.github.io/

Calls C4–8 in Table 1 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 [15] 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. Appendix A includes a non-trivial CRAFTS API configuration object, while Section 3 presents a thorough showcase scenario that exploits such API configuration.

### *2.3. 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 1. 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*.

```
Api { 
   apiId*                string
   endpoints*
                         [ {
                           id*                 string
                           sparqlURI*          string($uri)
                           graphURI            string($uri)
                           httpMethod          string
                                               Enum:
                                               [ GET, POST ]
                           authInfo
                                               {
                             user*             string
                             password*         string
                             type*             string
                                               Enum:
                                               [ basic, digest ]
                                               }
                           sparqlUpdate
                                               {
                             id                string
                             sparqlURI         string($uri)
                             graphURI          string($uri)
                             httpMethod        string
                                               Enum:
                                               [ GET, POST ]
                             authInfo
                                               {
                               user*           string
                               password*       string
                               type*           string
                                               Enum:
                                               [ basic, digest ]
                                               }
                                               }
                         }]
   model*
                         [ {
                           id*                 string
                           types*
                                               [ {
                             label*            string
                             endpoint*         string
                             inferred          boolean
                             restrictions
                                               [string]
                             targetId          string
                             embed             boolean
                             writeonly         boolean
                                               }]
                           dprops*
                                               [ {
                             label*            string
                             endpoint*         string
                             iri*              string($uri)
                             restrictions
                                               [string]
                             writeonly         boolean
                                               }]
                           oprops*
                                               [ {
                             label*            string
                             endpoint*         string
                             iri*              string($uri)
                             inv               boolean
                             restrictions
                                               [string]
                             targetId          string
                             embed             boolean
                             writeonly         boolean
                                               }]
                         }]
   queryTemplates*
                         [ {
                           id*                 string
                           endpoint*           string
                           description         string
                           template*           string
                           variables*
                                               [string]
                           parameters*
                                               [ {
                             label*            string
                             type*             string
                                               Enum:
                                               [ iri, string, number, integer, boolean ]
                             optional          boolean
                                               }]
                         }]
}
```

Figure 2. JSON schema of a CRAFTS API configuration.

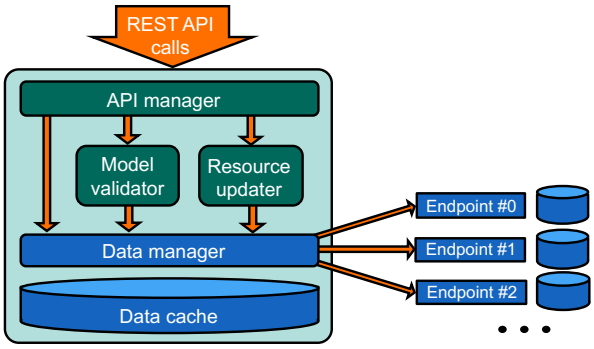


Figure 3. Logical architecture of CRAFTS.

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, 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. 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 `iris` 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, 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 .
 {{#restrictions}} {{{.}}} {{/restrictions}}
 FILTER (?iri IN ( {{{iris}}} ))
}
```

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 {
 {{^inv}} ?iri <{{{propiri}}}> ?value . {{/inv}}
 {{#inv}} ?value <{{{propiri}}}> ?iri . {{/inv}}
 {{#restrictions}} {{{.}}} {{/restrictions}}
 FILTER (?iri IN ( {{{iris}}} ))
}
```

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 manager* for the target `iri`. 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 [16]: (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 operations are dispatched to the *Data manager*. For any writing call, the *Resource*

Table 2
Prefixes and namespaces employed in the showcase of CRAFTS.

| Prefix | Namespace |
|---|---|
| axis | http://epsg.w3id.org/ontology/axis/ |
| crs | http://epsg.w3id.org/data/crs/ |
| 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/ |
| rdfs | http://www.w3.org/2000/01/rdf-schema# |
| schema | http://schema.org/ |
| tree | https://datos.iepnb.es/recurso/sector-publico/medio-ambiente/ifn/tree/ |

*updater* keeps track of the RDF resources that were modified and makes a clean cache request to the *Data manager*.

### *2.4. Implementation*

I have coded a full functional version of CRAFTS in JavaScript for the popular Node.js platform.[3] 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[4] web framework to easily handle web requests. The whole API of CRAFTS (see Table 1) is annotated with the OpenAPI specification [14]. This is exploited by the express-openapi-validator[5] 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 autogenerated API documentation served with Swagger UI Express.[6] The utility functions of Underscore[7] are employed for handling JavaScript collections. Mustache[8] is used for templating SPARQL queries, as described in Section 2.3.

The source code of CRAFTS is available on GitHub.[9] I have also set up a live version of CRAFTS[10] for anyone who wants to use it. The OpenAPI specification of CRAFTS is browsable (and actionable) at this URL.[11] This version incorporates user management functionality; any registered user can create their own APIs and perform read operations in any API published in CRAFTS. Write operations are only allowed for API creators (i.e. craftsmen). Note that is not necessary to be a registered user in order to use an API in CRAFTS –every API has read and write tokens that can be employed with the Bearer authentication scheme [17]. In this way, a craftsman can share the API read or write tokens to provide access to the API without requiring to sign up in CRAFTS.

## 3. 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 2 includes the namespaces employed along this showcase.

*Creation of a CRAFTS API.* Any registered user can create an API by using call C3 with a configuration in the body request. We will use the configuration included in Appendix A. Briefly, it includes two endpoints: `crossforest` and `dbpedia`. The former is a large forestry dataset (almost 200M triples) built in the context of the EU

[3] https://nodejs.org
[4] http://expressjs.com/
[5] https://www.npmjs.com/package/express-openapi-validator
[6] https://www.npmjs.com/package/swagger-ui-express
[7] https://www.npmjs.com/package/underscore
[8] https://www.npmjs.com/package/mustache
[9] https://github.com/guiveg/crafts
[10] https://crafts.gsic.uva.es
[11] https://crafts.gsic.uva.es/docs/
[12] See footnote 10.

Table 3
Sample parameter values for testing `treesinbox`.

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

Cross-Forest project[13] with institutional repositories from Spain and Portugal. The latter endpoint is the well-known English DBpedia [18]. 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

As described in Section 2.3, the *Model Validator* will check that the configuration is correct, including probe queries to `crossforest` and `dbpedia`.

*Submitting a parametrized query.* Once the API `test` is set up, we can use the query template `treesinbox`, aimed for discovering trees of a specific species in a bounding box. For this showcase we will use the parameter values in Table 3. Call C9 is purposed 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&offset=0&species=https://datos.iepnb.es/def/sector-publico/medio-ambiente/ifn/Species43

This request is processed by the *Data manager*, producing the SPARQL query in Listing 3. 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 [19]. For the sake of readability, the actual answer to the query in Listing 3 is presented in Table 4.

[13] https://crossforest.eu/

Table 4
Answer to the SPARQL query in Listing 3.

| tree | tsps | 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 | 38.385372 | -3.688194 |

Listing 3: SPARQL query obtained after applying parameters in Table 3 to the Mustache template `treesinbox` in Appendix A.

```
SELECT DISTINCT ?tree ?tsps ?tlat ?tlng WHERE {
?tree a ifn:Tree, ?tsps ;
    pont:hasPosition ?pos .
?pos pont:hasCoordinateReferenceSystem crs:4326 ;
    axis:1 ?tlat ;
    axis:2 ?tlng .
?tsps rdfs:subClassOf+ ifn:Plantae .
 FILTER (?tsps IN (ifn:Species43))
 FILTER (?tlat > 37)
 FILTER (?tlat < 40)
 FILTER (?tlng > -6)
 FILTER (?tlng < -3)
}
LIMIT 5
```

*Requesting a resource (one endpoint).* We will use the first result in Table 4 to make an RDF resource read call (C4) for `tree:10-2898-A-1-2` and model `Tree`. The resulting URL is https://crafts.gsic.uva.es/apis/test/resource?id=Tree&iri=https://datos.iepnb.es/recurso/sector-publico/medio-ambiente/ifn/tree/05-0810-A-4-11

This request is received by the *Data manager*. This component will then use the `Tree` model to send a series of SPARQL queries to the `crossforest` endpoint (provided there are misses in the *Data cache*). Table 5 shows the actual SPARQL queries that will be produced after applying the Mustache templates in Listing 1 and 2 to each element in the `types`, `dprops`, and `oprops` arrays of `Tree`. While the queries are straightforward, it is interesting to note the inclusion of `restrictions` in `species` and `position` (check the origin of those restrictions in Appendix A). Also note that the latter two queries are referred to a different RDF resource, `pos:10-2898-A-1-2-23030-4326`. This is for embedding position data in the requested tree, as defined in `position` by setting `embed=true`

Table 5
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 )) } |

and `targetId=Position`.[14] With the obtained responses, the *Data manager* will return the JSON object in Listing 4.

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

```
{
  "iri": "tree:10-2898-A-1-2",
  "species": "ifn:Species43",
  "diameter_mm": 327,
  "height_M": 6.9,
  "position": {
    "iri": "pos:10-2898-A-1-2-23030-4326",
    "latWGS84": 39.393602,
    "lngWGS84": -5.648659
  }
}
```

*Requesting a resource (two endpoints).* Similarly to the previous case, we 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

The interesting thing here is that data is coming from two endpoints, as shown in Table 6. 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. `dbr: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`.

```
{
  "iri": "ifn:Species43",
  "scientificName": {
    "la": "Quercus pyrenaica"
  },
  "vulgarName": [
    {
      "en": "Pyrenean oak"
    },
```

[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 6 to produce Listing 5.

Table 6
SPARQL queries produced for obtaining the output in Listing 5.

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

```
    {
      "es": "Rebollo"
    },
    {
      "de": "Pyrenaen-Eiche"
    },
    {
      "fr": "Chene tauzin"
    }
  ],
  "dbpedia": {
    "iri": "dbr: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"
  }
}
```

*Creating a resource.* We will use call C5 to create an RDF resource with IRI `tree:0` and model `Tree`.[15] In the request body we include the JSON object in Listing 6, 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

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

```
{
  "iri": "tree:0",
  "species": "ifn:Species23",
  "position": {
    "iri": "pos: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*

[15] The remaining examples of this showcase requires setting credentials for SPARQL Update in the `crossforest` endpoint.

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 . }
```

*Updating a resource.* We are going to update the tree created in the previous step, `tree:0`, with diameter and height measures. For this purpose we 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 . }
```

*Deleting a resource.* We 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/ifn/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` 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`.

```
DELETE DATA {
  tree:0 a ifn:Species23 ;
    pont:hasPosition pos:0 ;
    ifn:hasDBH1InMillimeters 100 ;
    ifn:hasTotalHeightInMeters 10 . }
```

## 4. Uptake of CRAFTS

The test site of CRAFTS[16] has been up since February 2021. I presented CRAFTS in an online seminar in March 2021. Since then, there has been some significant activity that is tracked with the Google Analytics platform.[17] Table 7 summarizes the collected data (obtained in July 2021).

In this period of barely five months, a small group of craftsmen has registered and released seven active APIs. Remarkably, more than 900 LOD users have accessed CRAFTS in 2.9K sessions and making 106K calls. The bulk of this workload corresponds to three APIs: Forest Ex-

[16] See footnote 10.
[17] https://www.google.com/analytics

Table 7
Uptake of the test site of CRAFTS.

| Item | Value |
|---|---|
| # of craftsmen | 6 |
| # of LOD users | 946 |
| # of sessions | 2,932 |
| # of active APIs | 7 |
| # of API calls | 106,423 |

plorer (84.7% of all the calls), EducaWood (12.5%), and LocalizARTE (1.4%). There is a dedicated web application for each of these APIs, as described below.

Forest Explorer [20] is a web application that offers an interactive map for browsing forestry data from the Cross-Forest and English DBpedia endpoints. This application has been 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...). Despite the increased size and complexity of the Cross-Forest dataset (71GB vs 11GB), the refactored version of Forest Explorer is much simpler.[18] A source lines of code (SLOC) analysis of the two versions with CLOC[19] 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 [21] 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. Noteworthy, the web application is developed by a Master student in Telecommunications Engineering with no expertise on Semantic Web technologies.

LocalizARTE[20] is a web application for ubiquitous learning in the Cultural Heritage domain. Teachers can 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 use the application 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. As in the EducaWood case, a Master student with no expertise on Semantic Web technologies develops LocalizARTE and uses the corresponding CRAFTS API.

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, 51% of the users are located in Spain, 21% in Portugal, 3% in United States, 2% in India, 2% in Vietnam, 1% in the Netherlands, and the rest in other 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 to fill after some significant activity in For-

[18] Note that the refactoring was carried out in two days by the same developer that created the former version (me).
[19] https://github.com/AlDanial/cloc
[20] https://localizarte.gsic.uva.es/

Table 8
Distribution of call types and latencies in the test site of CRAFTS.

| Call ID | % of calls | Avg. time |
|---|---|---|
| C0 | 0.2% | 0.02 s. |
| C1 | 1.2% | 0.02 s. |
| C2 | 0.2% | 13.05 s. |
| C3 | 0.0% | 0.04 s. |
| C4 | 11.9% | 0.21s. |
| C5 | 0.1% | 0.25 s. |
| C6 | 0.1% | 0.31 s. |
| C7 | 0.1% | 0.24 s. |
| C8 | 50.0% | 0.21 s. |
| C9 | 36.2% | 1.07 s. |

est Explorer.[21] With respect to EducaWood and LocalizARTE, the user base is smaller (~100 users) and more local (mainly Spain), as we have not yet initiated their promotion. This is likely to change given that EducaWood has received in July 2021 the third award to the *Aporta Challenge 2020 – The value of data in digital education*,[22] a national challenge to promote the use of data that is sponsored by the Spanish Ministry of Economy.

To conclude this section, Table 8 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. 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 unfrequent 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.

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

[22] https://datos.gob.es/en/desafios-aporta/aporta-challenge-2020

## 5. Discussion

CRAFTS has demonstrated to be very flexible for accessing Linked Open Data. 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, e.g. including position data of the requested tree in Listing 4. Moreover, an API can be configured to seamlessly integrate data from several endpoints; Listing 5 showcases a non-trivial example of species data integration from the Cross-Forest and English DBpedia endpoints. 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 API of CRAFTS is completely generic, domain-agnostic, and predictable. CRAFTS exposes an interactive frontend of the API contents,[23] benefitting from the use of the OpenAPI specification. As a result, a CRAFTS API can be accessed as any regular REST API. LOD users can easily send parametrized SPARQL queries by selecting a template in an API and setting appropriate parameter values, as demonstrated in Section 3. 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 4 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.

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 [22] and Linked Data Fragments [3]. The Linked Data Platform is a W3C recommendation for providing read and write access to Linked Data

[23] See footnote 11.

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 [23] and SPARQL transformer [24]. 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. Note that CRAFTS has a different goal since it aims to customize the data to LOD users' needs without requiring to follow the RDF data model. SPARQL transformer proposes a new JSON serialization of SPARQL results in order to simplify the output, although it does not support SPARQL query formulation, a much more demanding task for LOD users.

Closer to the scope of CRAFTS, RAMOSE [4], grlc [9], and BASIL [10] are tools for creating REST APIs on top of triple stores. All of them provide a functionality similar to the parametrized SPARQL queries of CRAFTS: knowledge engineers will define the queries to be supported, and LOD users will make API calls instead of authoring SPARQL queries. A difference in CRAFTS is the use of Mustache as a well-known templating mechanism for parametrized SPARQL queries, while the aforementioned tools employ their own custom conventions. Further, none of them supports operations over LOD resources, one of the most distinguishing capabilities of CRAFTS that is also supported by OBA.

OBA [8] is a recent proposal for creating REST APIs over triple stores. Starting from an ontology, OBA can 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. A prominent difference with CRAFTS is that OBA is an ontology-driven API generator; this is of course appealing for automating the creation of an API, 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. in the case of the Forest Explorer API we only gather tree species information from the English DBpedia, so it is impractical to take the whole DBpedia ontology for this purpose.

Another important difference between OBA and CRAFTS 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 3. Other relevant differences include the native support for caching in CRAFTS and partial updates of resources through PATCH requests.

Ongoing pilots with CRAFTS demonstrates the usefulness of this tool for supporting the consumption, integration, and generation of Linked Open Data through REST APIs. Remarkably, this is performed by a group of LOD users that do not need to know Semantic Web technologies. Future work includes the exploitation of CRAFTS in a number of application cases, especially in the forestry and educational domains. 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.[24]

## Acknowledgements

This work has been partially funded by the European Commission through Cross-Forest (CEF 2017-EU-IA-0140) and Virtual Forests (Erasmus+

[24] https://graphql.org/

2020-1-ES01- KA226-HE-095836) and Spanish Ministry of Science and Innovation through REFORM (ERANET SUMFOREST PCIN-2017-027) projects.

## Appendix A. Example of a CRAFTS API configuration

```
{
  "apiId": "test",
  "endpoints": [
    {
      "id": "crossforest",
      "sparqlURI": "https://crossforest.gsic.uva.es/sparql
           ",
      "graphURI": "http://crossforest.eu",
      "httpMethod": "GET"
    },
    {
      "id": "dbpedia",
      "sparqlURI": "http://dbpedia.org/sparql",
      "graphURI": "http://dbpedia.org",
      "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",
          "embed": false,
          "restrictions": [
            "?type rdfs:subClassOf+ <http://crossforest.eu
                 /ifn/ontology/Plantae> ."
          ],
          "endpoint": "crossforest"
        }
      ]
    },
    {
      "id": "Position",
      "oprops": [],
      "dprops": [
        {
          "label": "latWGS84",
          "iri": "http://epsg.w3id.org/ontology/axis/1",
          "endpoint": "crossforest"
        },
        {
          "label": "lngWGS84",
          "iri": "http://epsg.w3id.org/ontology/axis/2",
          "endpoint": "crossforest"
        }
      ],
      "types": []
    },
    {
      "id": "Species",
      "oprops": [
        {
          "label": "dbpedia",
          "targetId": "DbpediaSpecies",
          "iri": "http://schema.org/sameAs",
          "restrictions": [
            "FILTER contains(str(?value), \"dbpedia\" )"
          ],
          "embed": true,
          "endpoint": "crossforest"
        }
      ],
      "dprops": [
        {
          "label": "scientificName",
          "iri": "http://crossforest.eu/ifn/ontology/
               hasAcceptedName>/<http://crossforest.eu/
               ifn/ontology/name",
          "endpoint": "crossforest"
        },
        {
          "label": "vulgarName",
          "iri": "http://crossforest.eu/ifn/ontology/
               vulgarName",
          "endpoint": "crossforest"
        }
      ],
      "types": []
    },
    {
      "id": "DbpediaSpecies",
      "oprops": [
        {
          "label": "image",
          "iri": "http://dbpedia.org/ontology/thumbnail",
          "endpoint": "dbpedia"
        }
      ],
      "dprops": [
        {
          "label": "comment",
          "iri": "http://www.w3.org/2000/01/rdf-schema#
               comment",
          "restrictions": [
            "FILTER (lang(?value) = \"es\" OR lang(?value)
                 = \"en\" OR lang(?value) = \"\")"
          ],
          "endpoint": "dbpedia"
        }
      ],
      "types": []
    }
  ],
  "queryTemplates": [
    {
```

```
      "id": "treesinbox",
      "description": "Obtain trees (variable "tree"),
           their species (variable "tsps") and their
           coordinates (variables "tlat", "tlng") in a
           box (parameters "lngwest", "lngeast", "
           latsouth", "latnorth") of species "species".
           This template query can be paginated with the
           optional parameters "limit" and "offset"",
      "template": "SELECT DISTINCT ?tree ?tsps ?tlat ?tlng
            WHERE {
?tree a <https://datos.iepnb.es/def/sector-publico/medio-
     ambiente/ifn/Tree>, ?tsps ;
    <http://crossforest.eu/position/ontology/hasPosition>
        ?pos .
?pos <http://crossforest.eu/position/ontology/
     hasCoordinateReferenceSystem> <http://epsg.w3id.org/
     data/crs/4326> ;
    <http://epsg.w3id.org/ontology/axis/1> ?tlat ;
    <http://epsg.w3id.org/ontology/axis/2> ?tlng .
?tsps rdfs:subClassOf+ <https://datos.iepnb.es/def/sector-
     publico/medio-ambiente/ifn/Plantae> .
{{#species}} FILTER (?tsps IN (<{{{species}}}>)) {{/
     species}}
{{#latsouth}} FILTER (?tlat > {{latsouth}}) {{/latsouth}}
{{#latnorth}} FILTER (?tlat < {{latnorth}}) {{/latnorth}}
{{#lngwest}} FILTER (?tlng > {{lngwest}}) {{/lngwest}}
{{#lngeast}} FILTER (?tlng < {{lngeast}}) {{/lngeast}}}
{{#limit}} LIMIT {{limit}} {{/limit}}
{{^limit}} LIMIT 100 {{/limit}}
{{#offset}} OFFSET {{offset}} {{/offset}}",
      "variables": [
        "tree",
        "tlat",
        "tlng",
        "tsps"
      ],
      "parameters": [
        {
          "label": "species",
          "type": "iri",
          "optional": true
        },
        {
          "label": "lngwest",
          "type": "number",
          "optional": true
        },
        {
          "label": "lngeast",
          "type": "number",
          "optional": true
        },
        {
          "label": "latnorth",
          "type": "number",
          "optional": true
        },
        {
          "label": "latsouth",
          "type": "number",
          "optional": true
        },
        {
          "label": "limit",
          "type": "integer",
          "optional": true
        },
        {
          "label": "offset",
          "type": "integer",
          "optional": true
        }
      ],
      "endpoint": "crossforest"
    }
  ]
}
```

## References


[1] T. Berners-Lee, Linked Data – Design Issues, 2006, URL: http://www.w3.org/DesignIssues/LinkedData.html, revised 18-06-2009, last visited July 2021.

[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* **5**(3) (2009), 1–22, DOI: 10.4018/jswis.2009081901.

[3] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck and P. Colpaert, Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web, *Journal of Web Semantics* **37–38** (2016), 184–206, DOI: 10.1016/j.websem.2016.03.003.

[4] M. Daquino, I. Heibi, S. Peroni and D. Shotton, Creating Restful APIs over SPARQL endpoints with RAMOSE, *Semantic Web* (2021), In press.

[5] A.-S. Dadzie and E. Pietriga, Visualisation of Linked Data – Reprise, *Semantic Web* **8**(1) (2017), 1–21, DOI: 10.3233/SW-160249.

[6] R.T. Fielding, Architectural styles and the design of network-based software architectures, PhD thesis, University of California, Irvine, 2000.

[7] T. Bray (ed.), The JavaScript Object Notation (JSON) Data Interchange Format, Standards Track, RFC 8259, Internet Engineering Task Force (IETF), 2017, URL: https://datatracker.ietf.org/doc/html/rfc8259, last visited July 2021.

[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)*, J.Z. Pan, V. Tamma, C. d'Amato, K. Janowicz, B. Fu, A. Polleres, O. Seneviratne and L. Kagal, eds, LNCS, Vol. 12507, Springer, Cham, Switzerland, 2020, pp. 48–64.

[9] A. Meroño-Peñuela and R. Hoekstra, grlc makes GitHub taste like linked data APIs, in: *Proceedings of the 13th European Semantic Web Conference (ESWC 2016)*, H. Sack, G. Rizzo, N. Steinmetz, D. Mladenić, S. Auer and C. Lange, eds, LNCS, Vol. 9989, Springer, Cham, Switzerland, 2016, pp. 342–353.

[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, 2015, pp. 22–32, co-located with the 12th European Semantic Web Conference (ESWC 2015).

[11] R. Fielding and J. Reschke (eds), Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, Standards Track, RFC 7231, Internet Engineering Task Force (IETF), 2014, URL: https://datatracker.ietf.org/doc/html/rfc7231, last visited July 2021.

[12] T. Johnson, Documenting APIs: A guide for technical writers and engineers, 2020, URL: https://idratherbewriting.com/learnapidoc/, last visited July 2021.

[13] P. Gearon, A. Passant and A. Polleres (eds), SPARQL 1.1 Update, Recommendation, W3C, 2013, URL: http://www.w3.org/TR/2013/REC-sparql11-update-20130321/, last visited July 2021.

[14] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky and U. Sarid (eds), OpenAPI Specification, Technical Report, Version 3.0.3, OPENAPI initiative, 2020, URL: https://spec.openapis.org/oas/v3.0.3, last visited July 2021.

[15] P. Bryan (ed.), JavaScript Object Notation (JSON) Patch, Standards Track, RFC 6902, Internet Engineering Task Force (IETF), 2013, URL: https://datatracker.ietf.org/doc/html/rfc6902, last visited July 2021.

[16] L. Dusseault, L. Lab and J. Snell, PATCH Method for HTTP, Standards Track, RFC 5789, Internet Engineering Task Force (IETF), 2010, URL: https://datatracker.ietf.org/doc/html/rfc5789, last visited July 2021.

[17] M. Jones and D. Hardt, The OAuth 2.0 Authorization Framework: Bearer Token Usage, Standards Track, RFC 6750, Internet Engineering Task Force (IETF), 2012, URL: https://datatracker.ietf.org/doc/html/rfc6750, last visited July 2021.

[18] 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* **6**(2) (2015), 167–195, DOI: 10.3233/SW-140134.

[19] A. Seaborne, K. Grant Clark, L. Feigenbaum and E. Torres (eds), SPARQL 1.1 Query Results JSON Format, Recommendation, W3C, 2013, URL: http://www.w3.org/TR/2013/REC-sparql11-results-json-20130321/, last visited July 2021.

[20] G. Vega-Gorgojo, J.M. Giménez-García, C. Ordóñez and F. Bravo, Pioneering easy-to-use forestry data with Forest Explorer, *Semantic Web* (2021), 1–14, In press. DOI: 10.3233/SW-210430.

[21] J. Andrade-Hoz, G. Vega-Gorgojo, I. Ruano-Benito, M.L. Bote-Lorenzo, J.I. Asensio-Pérez, F. Bravo and C. Ordóñez, EducaWood: a Socio-Semantic Annotation System for Environmental Education, in: *Proceedings of the 16th European Conference on Technology Enhanced Learning (EC-TEL 2021)*, LNCS, Springer, Cham, Switzerland, 2021.

[22] S. Speicher, J. Arwe and A. Malhotra (eds), Linked Data Platform 1.0, Recommendation, W3C, 2015, URL: http://www.w3.org/TR/2015/REC-ldp-20150226/, last visited July 2021.

[23] G. Kellogg, P.A. Champin and D. Longley (eds), JSON-LD 1.1: A JSON-based Serialization for Linked Data, Recommendation, W3C, 2020, URL: https://www.w3.org/TR/2020/REC-json-ld11-20200716/, last visited July 2021.

[24] 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)*, C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois and F. Gandon, eds, LNCS, Vol. 11779, Springer, Cham, Switzerland, 2019, pp. 454–470.