Proceedings of the 7th Workshop on Linked Data in Linguistics (LDL-2020), pages 52–60
Language Resources and Evaluation Conference (LREC 2020), Marseille, 11–16 May 2020

A Lime-Flavored REST API for Alignment Services

Manuel Fiorelli, Armando Stellato
University of Rome "Tor Vergata", Department of Enterprise Engineering, via del Politecnico 1, 00133 Roma, Italy
fiorelli@info.uniroma2.it, stellato@uniroma2.it

Abstract

A practical alignment service should be flexible enough to handle the varied alignment scenarios that arise in the real world, while minimizing the need for manual configuration. MAPLE, an orchestration framework for ontology alignment, supports this goal by coordinating a few loosely coupled actors, which communicate and cooperate to solve a matching task using explicit metadata about the input ontologies, other available resources and the task itself. The alignment task is thus summarized by a report listing its characteristics and suggesting alignment strategies. The schema of the report is based on several metadata vocabularies, among which the Lime module of the OntoLex-Lemon model is particularly important, summarizing the lexical content of the input ontologies and describing external language resources that may be exploited for performing the alignment. In this paper, we propose a REST API that enables the participation of downstream alignment services in the process orchestrated by MAPLE, helping them self-adapt in order to handle heterogeneous alignment tasks and scenarios. The realization of this alignment orchestration effort has been performed through two main phases: we first described its API as an OpenAPI specification (a la API-first), which we then exploited to generate server stubs and compliant client libraries. Finally, we switched our focus to the integration of existing alignment systems, with one fully integrated system and an additional one being worked on, in the effort to propose the API as a valuable addendum to any system being developed.

Keywords: Lime, OntoLex, VocBench, MAPLE, Ontology Matching

1. Introduction

Ontology matching (Euzenat & Shvaiko, 2013) is the task of computing an alignment between two (or more) ontologies that consists of correspondences between semantically related concepts. We consider a broader definition of the task, to cover thesauri and datasets, in general. We argued (Fiorelli et al., 2019) that a practical matching system should be flexible enough to recognize different matching scenarios and handle each of them with a suitable strategy possibly benefiting from additional support resources. Our framework MAPLE[1] achieves that goal by exploiting explicit metadata about the input ontologies and other available resources. MAPLE uses a combination of metadata vocabularies, including DCMI Metadata Terms[2], FOAF[3], VoID[4], DCAT[5] and Lime (Fiorelli et al., 2015). The latter is the metadata module of the OntoLex-Lemon model[6] (McCrae et al., 2017; Cimiano, McCrae, & Buitelaar, 2016), which is becoming a cornerstone of the growing Linguistic Linked Open Data cloud (Chiarcos, Nordhoff, & Hellmann, 2012), moving beyond its original focus on ontology lexicons. MAPLE uses Lime metadata to determine how lexical information is represented (i.e. the so-called *lexicalization model*), the degree of linguistic compatibility of the input ontologies (e.g. supported natural languages, relative coverage, relative expressiveness, etc.), as well to find suitable language resources (e.g. a wordnet) in some natural language to support synonym expansion. MAPLE compiles a *task report* that summarizes the characteristics of the given matching scenario and hints at possible matching strategies. This task report is intended to help a downstream matching system configure itself in order to manage the given matching scenario as best as possible. In this paper, we will refer to such a matching system as an *alignment service*, meaning a web service for the computation of alignments between datasets (in general). The contribution of this paper is precisely a REST API (Fielding, 2000) that an alignment service shall implement in order to comply with MAPLE. We used the OpenAPI[7] format to describe this API explicitly, ensuring that the produced specifications are both machine-readable and human friendly. These specifications establish a contract that make it possible for a user to invoke any alignment system for which a compliant server has been developed. We validated our work through the implementation of a sever for one alignment system, while planning an analogous one for an additional system.

2. Background

2.1 LIME: Linguistic Metadata

LIME is the module of OntoLex-Lemon dedicated to the description of lexicalized datasets and language resources such as wordnets. LIME extends VoID, by defining subclasses of *void:Dataset* based on the different roles that these datasets play form the view point of the ontology-lexicon interface.

A *lime:LexicalizationSet* is a dataset consisting of *lexicalizations* for a given *reference dataset* in some natural *language*, optionally using a *lexicon*, and expressed using a specific *lexicalization model*. A *lexicalization set* can describe the fact that an ontology (the *reference dataset*) contains RDFS labels (hence, RDFS is the *lexicalization model*) in English. If the ontology also contains labels in Italian, it would be necessary to introduce a second *lexicalization set*. Only if the *lexicalization model* is OntoLex-Lemon, then the *lexicalization set* shall reference a *lexicon*, providing the (reified) *lexical entries*. A *lexicalization set* may include metadata such as the number

[1] http://art.uniroma2.it/maple/
[2] https://www.dublincore.org/specifications/dublin-core/dcmi-terms/
[3] http://xmlns.com/foaf/spec/
[4] https://www.w3.org/TR/void/
[5] https://www.w3.org/TR/vocab-dcat-2/
[6] https://www.w3.org/2016/05/ontolex/
[7] https://www.openapis.org/

of lexicalizations, the percentage of the reference dataset being covered and the average number of lexicalizations for the entities in the reference dataset.

Wordnets are represented in OntoLex-Lemon by mapping: i) each synset to an *ontolex:LexicalConcept*, ii) each word to an *ontolex:LexicalEntry*, *iii)* each sense to an *ontolex:LexicalSense*. While similar to the one of lexicalized datasets, the structure of wordnets is characterized by the use of lexical concepts and specific properties to bind these to lexical entries thorough lexical senses. Therefore, their metadata deserved a dedicated class, called *lime:ConceptualizationSet*, which relates a *lime:Lexicon* (describing the collection of lexical entries) to an *ontolex:ConceptSet* (describing the collection of lexical concepts). The description of a conceptualization set may include the number of lexical concepts, the number of lexical entries, the average ambiguity and average synonymy.

A *conceptualization set* (and the associated datasets) can be used for synonym expansion; given a word: i) find matching lexical entries (usually one per POS tag), ii) for each matched lexical entry, find the associated lexical concepts, and iii) retrieve other lexical entries associated to any of these lexical concepts.

2.2 MAPLE: MAPping architecture based on Linguistic Evidences

MAPLE is a framework facilitating the orchestration of different, loosely coupled actors with the aim to support a *robust matching system*. A user defines a *matching task* as a pair of datasets, say D_{left} and D_{right}. The purpose of MAPLE is to facilitate the configuration of a downstream alignment system to solve this task. To this end, MAPLE provides an *orchestrator* that analyzes the input datasets, in order to infer the characteristics of the task and hint at promising alignment strategies.

The *orchestrator* looks up the two datasets in a *metadata registry* to retrieve metadata about them: indeed, the descriptions of the two datasets jointly characterize (perhaps indirectly) the matching task.

MAPLE specifies a *metadata application profile* that a compliant registry must obey to, while the actual implementation of the registry is part of the integration with other systems. In this manner, it is possible to adopt and switch different strategies to acquire and store the metadata (e.g. automatic profiling, manual addition or retrieval of metadata published alongside the datasets). The *orchestrator* and other downstream components in the processing chain are completely unaware of the chosen strategy.

The *orchestrator* uses the metadata about the input datasets, to determine which information is available, how it is represented, and the extent of overlap between the two datasets.

The *orchestrator* first determines the nature of the input datasets (i.e. their metamodel), identifying whether they are ontologies, thesauri and other RDF datasets. This knowledge is important to set the goal of the alignment (e.g. matching OWL classes vs matching SKOS concepts), while different combinations of dataset types may require different matching algorithms or dedicated configurations (e.g. taxonomy is encoded by the property *rdfs:subClassOf* in OWL ontologies and by the properties *skos:broader* and *skos:narrower* in SKOS thesauri).

MAPLE doesn't commit on any alignment technique, nonetheless it makes some assumptions:

- the seeding role of natural language lexicalizations
- the possibility to use wordnets for synonym expansion (and, in the future, for translation)

The *orchestrator* finds the *lexicalization sets* for the input datasets (see Section 2.1) and produces a ranked list of pairs of *lexicalization sets*. The *orchestrator* also tries to construct a *synonymizer* using a suitable wordnet included in the metadata registry. The order of the aforementioned list is determined by a complex scoring formula taking into account metrics about the lexicalization sets and, if available, about the *synonymizer*.

The *orchestrator* will compile a *task report* with the output of its analysis, which can be communicated to the *alignment system*.

3. Use Case and Requirements

As a software framework, MAPLE needs to be integrated into other systems, which in turn must implement or consume interfaces defined by MAPLE. Figure 1 illustrates a concrete use case applying MAPLE to VocBench 3[8] (Stellato et al., 2017; Stellato et al., in press), an open-source web application supporting collaborative editing of ontologies, thesauri and lexicons, complying with Semantic Web representation standards.

In this use case, the *matching task* comprises two datasets that are managed as two projects in VocBench.

VocBench provides an implementation of the *metadata registry* that covers locally managed datasets and remote ones (which are not associated with a VocBench project).

The *task report* produced by the orchestrator provided by MAPLE is returned to the *user* for explanation and refinement. The (possibly refined) task report is sent to the *alignment service* for the actual execution of the alignment task. The need for accepting the *task report* as an input instead of obtaining it from the orchestrator is motivated by the necessity to include the user in the loop.

In addition to the *task report*, the alignment service may accept some configuration parameters. The configuration is split in two: a *system configuration* that does not depend on the (explicit) choice of a *matcher*, and a *matcher configuration* that is bound to a specific matcher. *Matchers* and *configuration schemas* are clearly dependent on the alignment service, whose interface must include operations for retrieving them.

The computation of an alignment can be a slow task; therefore, it should be handled asynchronously without blocking the application (and thus the user) who submitted

[8] http://vocbench.uniroma2.it/

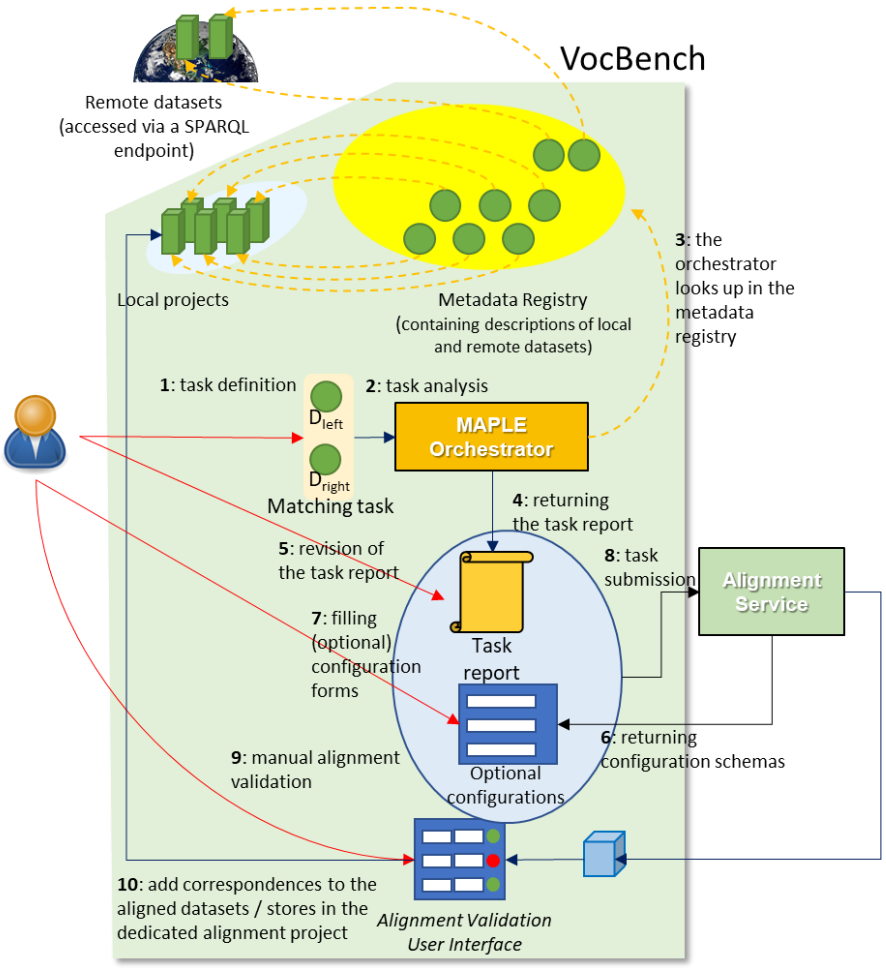

Figure 1: Use case integrating MAPLE, VocBench 3 and an external alignment service

it. Additionally, the alignment service shall support the submission of multiple tasks.

When an alignment task is completed, the user should be able to *download the alignment* into an alignment validation user interface. Validated alignments can then be integrated into either the left or right dataset. Moreover, the user can store that alignment into an *EDOAL project* dedicated to the alignment between these two datasets.

4. API Design Methodology

We designed a resource-centric API without using hypermedia (see Section 5.2), which is required by a strict compliance to the REST architectural style (Fielding, 2008). This kind of API, often called "pragmatic REST", comprises a collection of resources associated with endpoints (i.e. URLs) that can be operated on through standard HTTP verbs (e.g. GET to retrieve the representation of a resource, POST to create a new resource in a collection, etc.).

We analyzed the use case described in Section 3 to identify the resources, their representation and the verbs supported by each of them.

For the development of our API, we adopted the *API-first* approach: i.e. we started from the specifications of the API itself using the OpenAPI format rather than from the implementation of a reference server. These API specifications are a first-class artifact of the development process and, as such, they can be version controlled, verified, validated and published. Interoperability between clients and servers is guaranteed by the compliance to the same specifications. In fact, compliance to a given API is facilitated by tools that generate *client libraries* (to consume the API) and *server stubs* (to ease the implementation of the API) from the API definition. One such tool is Swagger Codegen[9], which supports tens of different programming languages. Moreover, a lot of API tools can be configured for a certain API by simply loading its definition in OpenAPI format.

5. API Definition

The design of a REST API is focused on the identification of the *resources* in the domain of interest, their *representations* and the *HTTP verbs* that they support. Each kind of resource is often associated with two paths (or endpoints): i) the collection of resources of that kind (e.g. */matchers*), ii) each specific resource of that kind (e.g. */matchers/1*). We represented the resources and the request bodies using JSON, which is currently the de facto standard for web APIs. In our API, some resources are read-only, because they reflect the capabilities of a specific alignment

[9] https://swagger.io/tools/swagger-codegen/

```
{
  "service": "Genoma REST API",
  "version": 1,
  "status": "active",
  "documentation": "https://../Home",
  "specs": [
    "http://../alignment-services-
1.0.0.yaml"
  ],
  "configuration" : {
    ...
  }
}
```

Listing 1: Representation of the root resource

service or the result of a computation, while others can be manipulated. This distinction manifests in the support for verbs other then *GET*.

5.1 Resources

5.1.1 API root

The path / is the root of the API namespace. Performing a GET on this path returns a JSON object like the one in Listing 1.

The object contains metadata about the *implementation of the alignment service* such as its name (*service)*, *version*, compliance to different *specs* and an optional system *configuration* schema (see Section 5.1.2). The *specs* property is an array of URLs for locating API definitions in the OpenAPI format. This array must contain at least the URL of the description of our REST API. Humans (e.g. developers) interfacing with this service may benefit from a reference to the *documentation* of the service.

When a sever has been just launched, it is not obvious when it is ready to accept requests. An approach to answer this question is to first attempt to retrieve the representation of the root: it can be assumed that a sever is not ready as long the service doesn't respond at all. Once the representation of the root path is returned, the property *status* tells whether the service is *starting*, *active*, *busy* (i.e. no longer accepting task submissions), *shutting down* or *failed*.

5.1.2 Matchers

The design goal of MAPLE is to disburden the user from manual configuration of the matching process to the maximum extent possible. However, an *alignment service* may support an option for manually choosing between different *matchers* (i.e. often associated with different combinations of matching techniques).

The path */matchers* is the collection of all available matchers, whereas the path */matchers/{id}* represents an individual matcher.

Listing 2 illustrates the JSON object describing a matcher, which contains its identifier (*id*), a textual *description* and an optional *configuration* schema.

The configuration schema defines the "shape" of the JSON object that represents the actual *matcher configuration* in a task submission (see Section 5.1.3). Moreover, the configuration schema can be used to produce a suitable user

```
{
  "id": "example-matcher",
  "description": "example matcher",
  "configuration": {
    "type": "object",
    "properties": {
      "structuralFeatures": {
        "description": "whether to use
structural features or not",
        "type": "boolean",
        "default": true
      },
      "synonymExpansion": {
        "description": "whether to do
synonym expansion or not",
        "type": "boolean",
        "default": true
      }
    }
  }
}
```

Listing 2: Representation of a matcher

interface to edit the configuration. Instead of reinventing the wheel, we adopted a subset of JSON Schema[10].

If the alignment service does not support manual selection and configuration of the matcher, this collection should be empty.

5.1.3 Tasks

The computation of an alignment is managed as an asynchronous *task*, which needs to be modeled explicitly.

The path */tasks* is the collection of all tasks ever submitted to the alignment service. The description of individual tasks can be obtained from the endpoint */tasks/{id}*. Listing 3 contains a JSON object that exemplifies the representation of a task.

The *id* identifies this task and it can also be found inside the path associated with the task. The properties *leftDataset* and *rightDataset* reference the two datasets to align. The service may differentiate between the *submission time*, when the task was first queued into the system, and the *start time*, when the computation started (pragmatically, when the service allocated computing resources for the task). An *end time* is also included when the execution ends. In fact, the task *status* makes it possible to differentiate between a task that is just *submitted*, *running*, *failed* or *completed*. When a task is *running*, its *start time* is non null and the service is computing the alignment. The task will

```
{
  "id": "c27d77380cf4[…]020871d5f95c2",
  "leftDataset":
"http://example.org/void.ttl#EuroVoc",
  "rightDataset":
"http://example.org/void.ttl#TESEO",
  "submissionTime": "202-02-
10T18:00:00+01:00",
  "startTime": "202-02-10T18:00:30+01:00",
  "status": "running",
  "progress": 60
}
```

Listing 3: Representation of a task

[10] https://json-schema.org/

```
{
  "taskReport": {
    "leftDataset": { "@id": "http://example.org/void.ttl#TESEO",
      "conformsTo": "http://www.w3.org/2004/02/skos/core#",
      "uriSpace": "http://www.senato.it/teseo/tes/",
      "sparqlEndpoint": "http://localhost:7200/repositories/TESEO_core"
    },
    "rightDataset": { "@id": "http://example.org/void.ttl#EuroVoc",
      "conformsTo": "http://www.w3.org/2004/02/skos/core#",
      "uriSpace": "http://eurovoc.europa.eu/",
      "sparqlEndpoint": "http://localhost:7200/repositories/EuroVoc_core"
    },
    "supportDatasets": [{
        "@id": " http://example.org/void.ttl#TESEO_it_lexset",
        "@type": "http://www.w3.org/ns/lemon/lime#LexicalizationSet"
        "sparqlEndpoint": "http://localhost:7200/repositories/TESEO_core",
        "referenceDataset": "http://example.org/void.ttl#TESEO",
        "lexiconDataset": null,
        "lexicalizationModel": "http://www.w3.org/2008/05/skos-xl",
        "lexicalizations": 3378, "references": 3378,
        "avgNumOfLexicalizations": 1, "percentage": 1,
        "languageTag": "it",
      }, {
        "@id": "http://example.org/void.ttl#EuroVoc_it_lexset",
        "@type": "http://www.w3.org/ns/lemon/lime#LexicalizationSet"
        "sparqlEndpoint": "http://localhost:7200/repositories/EuroVoc_core",
        "referenceDataset": " http://localhost:7200/repositories/EuroVoc_core",
        "lexiconDataset": null,
        "lexicalizationModel": "http://www.w3.org/2008/05/skos-xl",
        "lexicalizations": 18545, "references": 7282,
        "avgNumOfLexicalizations": 2.546, "percentage": 1,
        "languageTag": "it",
      }, {
         "@id": "http://.../omw/MultiWordNet-it-lexicon",
         "@type": "http://www.w3.org/ns/lemon/lime#Lexicon",
         "sparqlEndpoint": "http://localhost:7200/repositories/OMW_core",
         "languageTag": "it", "lexicalEntries": 43011
       }, {
         "@id": "http://.../omw/pwn30-conceptset",
         "@type": "http://www.w3.org/ns/lemon/ontolex#ConceptSet",
         "sparqlEndpoint": "http://localhost:7200/repositories/OMW_core",
         "concepts": 117659
       }, {
         "@id": "http://.../void.ttl#MultiWordNet_ConceptualizationSet",
         "@type": "http://www.w3.org/ns/lemon/lime#ConceptualizationSet",
         "sparqlEndpoint": "http://localhost:7200/repositories/OMW_core",
         "lexiconDataset": "http://.../omw/MultiWordNet-it-lexicon",
         "conceptualDataset": "http://.../omw/pwn30-conceptset",
         "conceptualizations": 63133, "concepts": 35001, "lexicalEntries": 43011,
         "avgSynonymy": 0.537, "avgAmbiguity": 1.468
       }],
    "pairings": [{
        "score": 0.7836831074710862,
        "source": {"lexicalizationSet": "http://example.org/void.ttl#EuroVoc_it_lexset" },
        "target": {"lexicalizationSet": "http://example.org/void.ttl#TESEO_it_lexset" },
        "synonymizer": {
          "lexicon": "http://example.org/void.ttl#OMW_Lexicon",
          "conceptualizationSet": "http://.../void.ttl#MultiWordNet_ConceptualizationSet"
        }
      }]
  }
}
```


Listing 4: Representation of a task submission (this example doesn't include neither a matcher nor configurations)

eventually end either by completing the computation of the alignment or by *failing* for some *reason*. The reason is expressed as a JSON object with at least the property *message*, which shall contain a textual description of the failure. For a running task, the property *progress* contains the percentage (expressed as an integer between 0 and 100) of the task that has been carried on. A *completed* task is associated with an alignment that can be retrieved by means of a GET on the path */tasks/{id}/alignment*. The response is

formatted according the format[11] of the Alignment API (David et al., 2011).

The submission of a *task* to the system can be made as a POST to the collection path */tasks*. The body of the request (see Listing 4) represents the submission in terms of a *task report* and, optionally, a *system configuration*, a *matcher* and (only if a *matcher* is provided) a *matcher configuration*.

Let us describe the content of a submission in reverse order, starting from the optional parts. If the alignment service allows that, the parameter *matcher* can be used to manually specify the matching algorithm (see Section 5.1.2). In this case, it is also possible to specify a *matcher configuration* as a JSON object, which shall conform to the configuration schema included in the representation of the matcher. Independently from the choice of a matcher, the user can also provide a *system configuration* as another JSON object, which shall conform to the configuration schema included in the representation of the root resource (see Section 5.1.1).

The *task report* is the only mandatory part of a task submission.

At the beginning of the report, the properties *leftDataset* and *rightDataset* contain the descriptions of the two datasets to match. The descriptions use properties that are in most cases eponym for properties defined by widely adopted metadata vocabularies. The description of a dataset includes its identifier (*"@id"*) (in the metadata registry), which is used in the rest of the task report to mention that dataset. The property *uriSpace* contains the namespace of the dataset (corresponding to *void:uriSpace*), while the property *sparqlEndpoint* contains the address of a SPARQL endpoint that provides access to the actual content of the dataset (corresponding to *void:sparqlEndpoint*). The property *conformsTo* (corresponding to *dcterms:conformsTo*) contains the URI of a modeling vocabulary that defines the type of the dataset (in the example, both datasets are SKOS thesauri).

The property *supportDatasets* is an array of JSON objects describing other potentially useful datasets. Like the ones above, these descriptions also include further properties that are bound to specific dataset types (*@type*).

In the example in Listing 4, the first two support datasets are *lime:LexicalizationSet*s that provide SKOS-XL labels in Italian for each of the input datasets. Indeed, Italian is the only natural language shared by these datasets, and consequently it is suggested as the basis for a monolingual matching scenario. The description of these lexicalization sets includes several properties borrowed from Lime to represent metrics.

The other three datasets define a subset of Open Multilingual Wordnet[12] (Bond & Paik, 2012) for Italian: i) the *ontolex:ConceptSet* describes the set of lexical concepts (i.e. synsets), ii) the *lime:Lexicon* describes the set of words in Italian, iii) the *lime:ConceptualizationSet* describes the bindings between these words and these concepts (i.e. word senses).

At the end of the report, the property *pairings* contains a ranked list of pairs of lexicalizations for each of the input datasets. Each pairing suggests a different strategy to compare the input datasets from a lexical viewpoint. If available (as in the example), the pairing also includes a *synonymizer* describing a strategy for synonym expansion (see Section 2.1).

The response of this HTTP request is the description of the task just created: using the identifier contained in this description, it is possible to poll the alignment service for updates on the status of the task.

5.2 Linking

Hypermedia is one of the defining characteristics of the REST architectural style, which is neglected by "pragmatic" realizations like ours. The principle is that the representations of the resources should include links to other resources and, in general, make it explicit to the clients the available affordances. The design constraint HATEOAS (Hypermedia as the Engine of Application State) requires that any state transition of the applications should be guided by these links. Without hypermedia, the usage protocol of the API should be encoded in the clients, and possibly communicated through an out-of-band mechanism.

OpenAPI 3 (the version we used to define our API) introduced the notion of links: these are not implemented using hypermedia in the API responses, but are expressed in the API definitions at the operation level. Simplifying, a link tells how part of the response of one operation can be used as argument for another operation. In other words, these links allow for describing (part of) the usage protocol of the API.

Within our API definitions, for example, we used links to tell that the ID contained in the response of creation operations can be used as an argument of operations for retrieving the details of a resource or for deleting it.

6. Implementation Report

The OpenAPI definition of the *alignment services* API is available online[13].

In Section 3, we gave the overall picture of our use case, integrating VocBench, MAPLE and remote alignment services. Our REST API meets all functional requirements elicited in that section; however, the VocBench user interface is not complete yet:

- users can't choose a matcher and specify its configuration nor can they specify a system configuration
- the task report generated by MAPLE can't be inspected or refined by users

The limitations above are clearly deficiencies of the components using the proposed alignment services API rather than a problem of the API itself: in fact, the

[11] http://alignapi.gforge.inria.fr/format.html
[12] http://compling.hss.ntu.edu.sg/omw/
[13] http://art.uniroma2.it/maple/specs/alignment-services-1.0.0.yaml

capabilities of the API (currently) exceed the ability of other systems to consume them.

We have already implemented a compliant server for the ontology matching tool GENOMA (Enea, Pazienza, & Turbati, 2015) using Swagger Codegen. Additionally, we planned the integration of another matching system called Naisc[14] (McCrae & Buitelaar, 2018).

7. Evaluation

The focus of our research effort is to provide concrete reusable support to alignment systems, separating the vertical discovery and exploration of efficient alignment techniques from the assessment of the alignment scenario and consequent fine tuning of these techniques to the situation. While the former is clearly not our goal – and thus requires no evaluation, as it mostly depends on the specific considered systems complying with our framework – we conducted an evaluation of the consistency of our approach and implementation in terms of specifications and API validation. Additionally, we provide a qualitative analysis based on our experiences in applying the API-first approach to the development of API-compliant components.

7.1 Verification of the Specifications

We used an online validator[15] to verify that our API definition conformed to the OpenAPI format. A non-conforming API definition might still be quite useful as a documentation for humans; nonetheless, this verification step is necessary to ensure that tooling[16] based on the OpenAPI format (e.g. code generators, testing frameworks, etc.) correctly process our API definition.

The validator confirmed that our API was valid, but it warned of not better specified circular references. We analyzed the API definition and, by revalidating a carefully edited definition, we ascertained that these circular references arose in the data model: in particular, in the data type *Schema*, which represents a JSON Schema that describes a *system configuration* (see Section 5.1.1) or a *matcher configuration* (see Section 5.1.2). Indeed, *Schema* is defined recursively: i.e. this data type occurs in its own definition. Let us consider Listing 2, in which the property *configuration* holds a *Schema* object. This schema models a JSON object that has the properties *structuralFeatures* and *synonymExpansion*. The value of each property is described recursively through a "nested" JSON Schema. In the example, the recursion terminates immediately, because both properties expect a primitive boolean value. However, a more complex configuration property might require several levels of nested JSON objects. Another source of recursion is the definition of *array* properties, whose *items* are modeled recursively with *Schema* objects.

Currently, we are aware of these negative consequences of circular references:

- The documentation generated by Swagger UI[17] doesn't display recursive data types correctly
- We were reported of problems with the OpenAPI Generator[18]. For GENOMA, we used Swagger Codegen without any issue: since GENOMA does not support custom configurations yet, there might be latent problems that we did not discover.

Unfortunately, removing recursion from the definition of *Schema* requires some redundancy in its definition and moreover, sacrificing the support for arbitrary nesting levels. We need to collect several more examples of configuration objects (see Section 9) to make an informed choice about whether the limitations introduced by a non-recursive definition are acceptable.

7.2 Validation of the API

The verification process described in the previous section is about whether we "built it right". However, it does not tell anything about whether we "built the right thing". With this regard, we should point out that we implemented (see Section 6) the use case described in Section 3, allowing the users of VocBench to actually interact with external alignment services using MAPLE. This has increased our confidence that the API we have defined is appropriate for its purpose. As pointed out in Section 9, we believe that onboarding of additional alignment systems should not affect the overall structure of the API, but mainly allow us to better understand the representation of configuration objects.

7.3 Qualitative Evaluation of the API-first approach

Swagger Codegen supports over 20 different languages for the generation of server stubs and over 40 different languages for the generation of client libraries.

The variety of server stubs simplifies the integration of matching systems implemented using different programming languages. In case of GENOMA (see Section 6), which is written in Java, we eventually decided to generate a sever stub utilizing the Spring framework. In fact, the generated stub provided the complete scaffolding of the server, leaving us just to provide the implementation of the operations of the API inside pre-generated methods. The generated code dealt with mapping of URL paths and parameters, clearly facilitating conformance to the API. With this regard, we should mention the automatic generation of a domain model from the JSON schemas (in the API definition) that model parameters and response bodies. This domain model uses standard Java types (e.g. Strings) instead of more specific types (e.g. RDF4J's IRI). This is advantageous since the alignment systems may use different libraries for the same purpose (e.g. RDF4J [19], Apache Jena[20] or OWLAPI[21] as RDF middleware). Initially, we were concerned about losing our customizations when regenerating the server because of changes of the API. However, we discovered that the generator produces a Java interface (which should not be edited at all) and a class implementing that interface (where the developer shall place its code). It is possible to regenerate the sole interface, while the IDE easily identifies

[14] https://github.com/insight-centre/naisc
[15] https://apitools.dev/swagger-parser/online/
[16] https://openapi.tools/
[17] https://swagger.io/tools/swagger-ui/
[18] https://openapi-generator.tech/
[19] https://rdf4j.org/
[20] https://jena.apache.org/
[21] http://owlcs.github.io/owlapi/

necessary changes to the class (e.g. new methods, changed method signatures, etc.). Currently, VocBench (see Section 3) is the only consumer of our API. In this case, we could not use the code generator precisely because of the generated domain model, which conflicted with the one already used for the communication with MAPLE: we preferred to implement the client manually, while the fact that the server was generated accordingly acted as a conformance check.

8. Related Work

Shvaiko and Euzenat (2013) analyzed the results of recent evaluation campaigns for ontology alignment[22], concluding that future growth of the field requires addressing eight challenges. Our work focuses on four of those:

- *matcher selection combination and tuning*: not explicitly addressed by MAPLE, but the *task report* is intended to help the alignment service to adapt and fine tune itself in order to fit the characteristics of the given matching scenario;
- *user involvement*: while striving to automate most of the configuration, our approach also foresees human intervention on both the task report and the configuration of the alignment service, as well as during the later validation of an alignment;
- *explanation of matching results*: in fact, our approach focuses on the visibility into the process for setting up and configuring the alignment service for a certain task;
- *alignment management: infrastructure and support*: the REST API presented in this paper and, even more, the overall integration described in Section 3 deal with the infrastructure supporting the management aspects, such as execution of alignment tasks, alignment validation and storage of links.

The Alignment Server, bundled with the Alignment API (David et al., 2011), offers a REST API[23] that can be compared to ours. In fact, the API of the Alignment Server has a wider scope: aiming at managing ontology networks, the Alignment Server supports computation, validation, storage and retrieval of alignments. Our API is focused on supporting the computation of alignments, while the rest is covered by the overall platform described in Section 3.

SEALS[24] (Semantic Evaluation At Large Scale) (Gutiérrez, García-Castro, & Gómez-Pérez, 2010) and HOBBIT[25] (Holistic Benchmarking of Big Linked Data) (Röder, Kuchelev, & Ngonga Ngomo, 2019) are two European projects whose outcome is a sustainable infrastructure for the execution of evaluation campaigns of semantic technologies in a scalable, systematic, repeatable and transparent manner. Consequently, their focus is more on i) unaided execution of heterogeneous systems against shared tests cases and ii) storage and comparison of test results. They also describe procedures to package the systems under test, and they offer a sophisticated platform to execute the resulting packages. Conversely, we don't deal with the provisioning of computing resources to the alignment services, which are assumed to be up and running on a (remote) machine. Moreover, our approach prescribes that the alignment service is actively aided by its clients, which submit a task report and, optionally, a matcher and some configuration parameters.

The integrated architecture described in Section 3 is close to the architecture of GOMMA, a "generic infrastructure for managing and analyzing life science ontologies and their evolution" (Kirsten et al., 2011). With respect to GOMMA, our whole architecture (including VocBench) covers storing versions of ontologies and mappings, and the invocation of alignment services. We do not cover diffing of ontologies (and mappings) and their evolution yet.

9. Future Work

We represented the resources defined by our API using JSON (see Section 5), while the schema of the task submission (see Section 5.1.3) is informally based on Lime and other metadata vocabularies (i.e. by the use of property names that match the names of the metadata properties). We will investigate JSON-LD[26] to preserve the use of JSON, while making that correspondence explicit through a JSON-LD context (referenced by the responses of our API).

By disseminating our API, we hope to on-board further alignment services beyond the two mentioned in Section 6. We believe that these services shouldn't require (substantial) changes to the operations (i.e. path + HTTP verb), since these are mainly defined from the viewpoint of client systems (i.e. that invoke the alignment service). Conversely, additional alignment services will help us to better understand and improve custom configurations (both at *system level* and *matcher level*), which are specific to an alignment service. Firstly, as the diversity of custom configurations increase, we will test the adequacy of the chosen subset of JSON Schema. Problematic areas include support for complex property values (e.g. structured values, polymorphism, etc.) and complex dependencies between configuration parameters (e.g. mutual exclusiveness between properties, conditional enablement of configuration properties, etc.). More varied configuration schemas will secondly give us the opportunity to understand if there are recurring patterns that deserve being part of a (possibly optional) standard configuration.

The use case presented in Section 3 includes manual evaluation of alignments aimed at improving their quality. We will investigate automatic alignment evaluation (performances, quality etc..) as well, even though instead of reinventing the wheel it could be interesting to see if we can integrate existing solutions such as HOBBIT.

10. Conclusions

MAPLE addresses the need for robustness in alignment systems through a metadata-based approach. In this paper, we concentrated on the interface that an alignment service should implement in order to comply with MAPLE and

[22] Such as the ones organized by OAEI (Ontology Alignment Evaluation Initiative) http://oaei.ontologymatching.org/
[23] http://alignapi.gforge.inria.fr/rest.html
[24] http://www.seals-project.eu/
[25] https://project-hobbit.eu/
[26] https://json-ld.org/

benefit from its services. Following the API-first methodology, we started from the specifications of the API as a machine-readable artifact using the OpenAPI format. Then, we implemented the API for the alignment system GENOMA and planned the same for the system Naisc.

11. Acknowledgements

This work has been supported by the PMKI project, under the 2016.16 action of the ISA^2 Programme (https://ec.europa.eu/isa2/). ISA^2 is a programme of the European Commission for the modernization of public administrations in Europe through the eGovernment solutions.

12. Bibliographical References

Bond, F., & Paik, K. (2012). A survey of wordnets and their licenses. Proceedings of the 6th Global WordNet Conference (GWC 2012). Matsue, Japan, January, 9-13, 2012, (pp. 64-71).

Chiarcos, C., Nordhoff, S., & Hellmann, S. (Eds.). (2012). Linked Data in Linguistics. Springer. doi:10.1007/978-3-642-28249-2

Cimiano, P., McCrae, J. P., & Buitelaar, P. (2016). Lexicon Model for Ontologies: Community Report, 10 May 2016. Community Report, W3C. Retrieved from https://www.w3.org/2016/05/ontolex/

David, J., Euzenat, J., Scharffe, F., & Trojahn dos Santos, C. (2011). The Alignment API 4.0. Semantic Web Journal, 2(1), 3-10.

Enea, R., Pazienza, M. T., & Turbati, A. (2015). GENOMA: GENeric Ontology Matching Architecture. In M. Gavanelli, E. Lamma, & F. Riguzzi (A cura di), Lecture Notes in Computer Science (Vol. 9336, p. 303-315). Springer International Publishing. doi:10.1007/978-3-319-24309-2_23

Euzenat, J., & Shvaiko, P. (2013). Ontology Matching (2 ed.). Springer-Verlag Berlin Heidelberg. doi:10.1007/978-3-642-38721-0

Fielding, R. T. (2000). REST: architectural styles and the design of network-based software architectures. University of California. Retrieved from https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Fielding, R. T. (2008, October 20). REST APIs must be hypertext-driven. Retrieved from Untangled: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Fiorelli, M., Stellato, A., Lorenzetti, T., Schmitz, P., Francesconi, E., Hajlaoui, N., & Batouche, B. (2019). Metadata-driven Semantic Coordination. In E. Garoufallou, F. Fallucchi, & E. William De Luca (Eds.), Metadata and Semantic Research (Communications in Computer and Information Science) (Vol. 1057). Springer, Cham. doi:10.1007/978-3-030-36599-8_2

Fiorelli, M., Stellato, A., Mccrae, J. P., Cimiano, P., & Pazienza, M. T. (2015). LIME: the Metadata Module for OntoLex. In F. Gandon, M. Sabou, H. Sack, C. d'Amato, P. Cudré-Mauroux, & A. Zimmermann (Eds.), The Semantic Web. Latest Advances and New Domains (Lecture Notes in Computer Science) (Vol. 9088, pp. 321-336). Springer International Publishing. doi:10.1007/978-3-319-18818-8_20

Gutiérrez, M. E., García-Castro, R., & Gómez-Pérez, A. I. (2010). Executing evaluations over semantic technologies using the SEALS Platform. Proceedings of the International Workshop on Evaluation of Semantic Technologies (IWEST 2010). Shanghai, China: CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-666/paper11.pdf

Kirsten, T., Gross, A., Hartung, M., & Rahm, E. (2011). GOMMA: a component-based infrastructure for managing and analyzing life science ontologies and their evolution. Journal of Biomedical Semantics. doi:10.1186/2041-1480-2-6

McCrae, J. P., & Buitelaar, P. (2018). Linking Datasets Using Semantic Textual Similarity. Cybernetics and Information Technologies, 8(1), 109-123. doi:10.2478/cait-2018-0010

McCrae, J. P., Bosque-Gil, J., Gracia, J., Buitelaar, P., & Cimiano, P. (2017). The OntoLex-Lemon Model: Development and Applications. In I. Kosem, C. Tiberius, M. Jakubíček, J. Kallas, S. Krek, & V. Baisa (Ed.), Electronic lexicography in the 21st century. Proceedings of eLex 2017 conference., (pp. 587-597).

Röder, M., Kuchelev, D., & Ngonga Ngomo, A.-C. (2019). HOBBIT: A platform for benchmarking Big Linked Data. Data Science. doi:10.3233/DS-190021

Shvaiko, P., & Euzenat, J. (2013, January). Ontology Matching: State of the Art and Future Challenges. IEEE Transactions on Knowledge and Data Engineering, 25(1), 158-176. doi:10.1109/TKDE.2011.253

Stellato, A., Fiorelli, M., Turbati, A., Lorenzetti, T., van Gemert, W., Dechandon, D., . . . Costetchi, E. (in press). VocBench 3: a Collaborative Semantic Web Editor for Ontologies, Thesauri and Lexicons. Semantic Web. Accepted manuscript at http://www.semantic-web-journal.net/content/vocbench-3-collaborative-semantic-web-editor-ontologies-thesauri-and-lexicons-1

Stellato, A., Turbati, A., Fiorelli, M., Lorenzetti, T., Costetchi, E., Laaboudi, C., . . . Keizer, J. (2017). Towards VocBench 3: Pushing Collaborative Development of Thesauri and Ontologies Further Beyond. In P. Mayr, D. Tudhope, K. Golub, C. Wartena, & E. W. De Luca (Ed.), 17th European Networked Knowledge Organization Systems (NKOS) Workshop. Thessaloniki, Greece, September 21st, 2017, (pp. 39-52). Retrieved from http://ceur-ws.org/Vol-1937/paper4.pdf