Link-Based Viewing of Multiple Web API Repositories

Devis Bianchini, Valeria De Antonellis, and Michele Melchiori

Dept. of Information Engineering University of Brescia
Via Branze, 38 - 25123 Brescia (Italy)
{devis.bianchini|valeria.deantonellis|michele.melchiori}@unibs.it

Abstract. Web API sharing, fueled by large repositories available online, is becoming of paramount relevance for agile Web application development. Approaches on Web API sharing usually rely on single Web API repositories, which provide complementary Web API descriptions, on which different search facilities can be implemented. In this paper, we propose a framework for defining a linked view over multiple repositories and for searching their content. In particular, we apply Linked Data principles to publish repository contents and identify semantic links across them in order to exploit complementary Web API descriptions. We discuss how Web API sharing across multiple repositories, based on such a link-based view, may benefit from selection criteria that combine several aspects in Web API characterization. A preliminary evaluation based on two popular public Web API repositories is presented as well.

1 Introduction

Selection and aggregation of reusable Web APIs is gaining momentum as a new development style for building new, value-added applications. Web APIs are specific kinds of digital resources that can be shared over the Web. They may be general purpose, provided by third parties, and they can be used as a productivity tool for on-the-spot problems, that is, situational applications that require few resources in terms of investment costs and development time. Existing approaches for Web API selection rely on APIs registered within the `ProgrammableWeb`[1] repository, because of its popularity. However, although `ProgrammableWeb` constitutes a well-known meeting point for the community of mashup developers, it is mainly focused on a feature-based description of Web APIs, classifying them through categories and tags and storing features like the adopted data formats, protocols and the list of mashups that have been developed using the Web APIs. There exist different repositories that emphasize distinct aspects to be considered for Web API sharing. For instance, `Mashape`, a cloud API hub[2] leveraging a twitter-like organization, associates each Web API with the list of developers who adopted or declared their interests for it and technical details

[1] `http://www.programmableweb.com/`
[2] `https://www.mashape.com/`

H. Decker et al. (Eds.): DEXA 2014, Part I, LNCS 8644, pp. 362–376, 2014.
© Springer International Publishing Switzerland 2014

for Web API invocation are provided as well. Similarly, other repositories (e.g., `theRightAPI`) provide a community-oriented perspective on Web APIs. In this scenario, distinct repositories act as information silos where: (i) complementary or partially overlapping Web API characterizations are stored; (ii) the same Web APIs or mashups are registered multiple times within different repositories; (iii) similarity between Web APIs and mashups across different repositories cannot be exploited to enrich search results. Various approaches have been proposed to integrate Web APIs with the Linked Data cloud, but they require models that are difficult and time-consuming to build [1] and generally are based on a single repository. On the other hand, publishing Web API repository contents as Linked Data may be useful to overtake the issues highlighted above. Moreover, publishing Web API descriptions as (open) Linked Data offers the opportunity for semantically enriching them by making explicit their social, technical and terminological aspects with links from the Web API description to relevant parts of the Web of Data. A further completion could be, when available, linking them to descriptions specifically conceived for consuming and producing linked data [2–4].

In this paper, we present a novel framework built according to Linked Data principles aimed at enhancing effective cross-repository Web API browsing and search for mashup development. The framework is based on an unified model for Web APIs providing a linked view on the contents of each repository. Such a model makes the content of repositories machine processable and accessible through non-proprietary tools (e.g., SPARQL endpoints). Therefore, it enables Web API search in a transparent way with respect to the localization of each repository. This unified model is based on three modeling perspectives that we introduced in [5] for Web APIs in a single repository. Within the framework, a *Linker* is in charge of identifying and storing semantic links across repositories according to identity and similarity criteria. Semantic links are published as Linked Data and can be exploited to perform Web API search over multiple repositories. thus taking benefit from existing approaches on query processing over Linked Data [6] also in the Web API selection scenario.

The paper is organized as follows: in Section 2 we provide the motivations of our approach, through the presentation of the running example and the discussion about related work; in Sections 3 and 4 we detail the design of the unified vocabulary and linking criteria on which the Linker is based; in Section 5 we describe how to exploit semantic links for Web API search through the Web API Search Engine; we discuss implementation issues and preliminary evaluation of the approach in Section 6; finally, Section 7 closes the paper.

2 Motivations

2.1 Running Example

As a running example, we refer to the `ProgrammableWeb` (PW) and `Mashape` (MP) repositories. We chose these two repositories since they are the most popular ones and provide complementary and partially overlapping features in Web

API descriptions so better illustrating the advantages and the problems arising in using both of them. In particular, let us consider a web designer who is charge of including a face recognition functionality to access the private area of his/her web site. Since developing this kind of applications from scratch would require very specific competencies, let us suppose that the designer looks for a Web API in the PW repository. Using the `face` and `recognition` keywords, the designer finds 15 matches. Among them, the `LambdaLabs Face` API is not assigned to any mashup. Since the goal of the designer's search on PW is to make easier the integration of a new Web API in an existing web site, the `LambdaLabs Face` API may not be considered. Nevertheless, on the MP repository, the same Web API has 1385 consumers and 1304 followers (that is, developers who declared their interest in the Web API). It is the most popular Web API out of the 106 APIs retrieved on MP using the same keywords. This additional information could influence the designer's choice. It is thus very important for the web designer to be able to identify and exploit correspondences between Web APIs in different repositories. However, this task may be not trivial. In fact, the same Web API may be classified into different categories or tagged with different tags across distinct repositories (for instance, the `LambdaLabs Face` API is classified in the `media` category on MP, while it is classified in the `Photos` category on PW). Moreover, the categorizations adopted in the two repositories are very different (68 categories on PW, 18 categories on MP, none of them organized in a hierarchy, 10 overlapping categories, but no explicit semantics). Finally, in an ideal situation, there should be specific properties to be used for identifying the same Web API description across different repositories, such as the URL. Unfortunately, data in Web API repositories is not often as complete as it could be: for example, the `LambdaLabs Face` API is registered with slightly different names on the two repositories and it is associated with the `http://api.lambdal.com/` URL on PW and to the `http://www.lambdal.com/developers/` URL on MP. Therefore, to merge together the search results coming from the two Web API repositories, the web designer must carefully inspect and compare each API description. We propose an approach that defines semantic links based on identity and similarity criteria between elements across different repositories in order to overcome these issues.

2.2 Related Work

Related efforts in the literature about Linked Web services or Linked Web APIs for discovery purposes focused on the semantic annotation of service features [7] through domain ontologies to publish service descriptions on the Web of Data [2, 8, 9], on the enrichment of the characterization of Web APIs to enable them to consume and produce Linked Data [4, 10], and on publishing as linked resources each single API invocation [11, 3].

The iServe platform described in [9] proposes adopting the Minimal Service Model (MSM), that is, a simple RDF(S) integration ontology that captures the maximum common denominator between existing conceptual models for services. In this model, Linked Data techniques are used for semantic annotation of the

I/O parameters of the interfaces of Linked Web services or Linked Web APIs with ontologies taken from the Web of Data. Comparison between I/Os based on semantic annotations is performed to infer semantic relationships between Web APIs. In [1], an interactive Web-based interface enables domain experts to rapidly create semantic models of services and Web APIs, using an expressive vocabulary that can be seen as an evolution of the MSM and includes also lowering and lifting rules, that is mappings, between the semantic model and the concrete API or service, that in this way is also able to consume and produce RDF data. All these approaches are characterized by a complex semantic model, where the semantic annotation of I/Os usually inhibits their wide adoption. This is especially critical in Web API repositories that have highly variable content. Moreover, the description of Web APIs at this level of details is not always available and, although semi-automatic solutions have been proposed in [1], the contents of repositories vary too fastly to enable this kind of approaches.

With respect to these approaches, our aim is to publish as Linked Data sources the Web API repositories as a whole, enabling discovery of Web APIs as resources. Therefore, we rely on information stored within multiple and public available repositories without requiring a semantic annotation activity of Web APIs. Similarity and identity links are automatically set to enable cross-repository resource sharing. Moreover, our unified view is not focused on Web APIs only, but also on mashups where APIs have been included according to the multiple perspectives described in [5], thus enabling more in-depth searching facilities. Approaches, which discuss how to process or generate Linked Data, such as Linked Open Services [11, 3] and Linked Data Services [4, 10], have not been designed for Web API search on multiple repositories. Linked Open Services (LOS) and Linked Data Services (LDS) are still focused on I/O modeling.

Among the most popular approaches and tools for link identification, we mention the Silk framework [12], a toolkit to link entities across different data sources. Nevertheless, the user-defined metrics presented in [12] can be built combining generic similarity measures only, and cannot be specifically applied to our domain of interest.

3 Modeling Web Mashup Resources

In order to obtain a link-based view of repository content we look for same/similar resources and define links among them. For comparing resources we assume a multi-perspective model introduced in [5] where features characterizing resources in existing repositories are considered. According to the multi-perspective model a resource can be described through:

- a functional characterization of the Web API obtained through a top-down classification, according to fixed categories, a bottom-up tagging through semantic tags associated by designers, and a set of technical features (e.g., protocols or data formats) used to further characterize the API (*component perspective*);

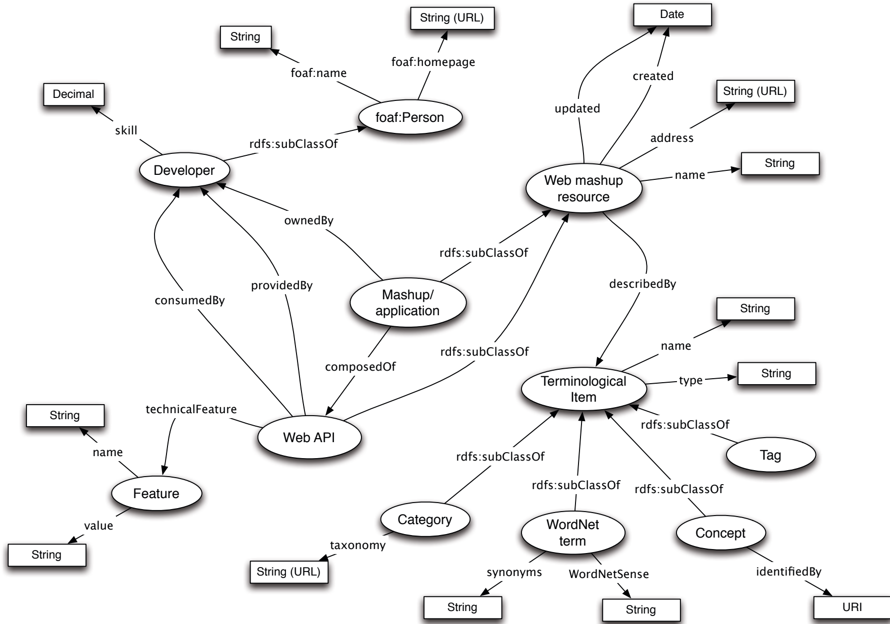

Fig. 1. RDF representation of the Web mashup resource unified model

– any existing mashups that include the Web API, described through the other APIs that compose them and tags associated by the designers with mashups as well (*application perspective*);
– the ratings assigned to the Web API by designers who used it in their own mashups (*experience perspective*).

The basic concepts and relationships of the multi-perspective model are represented in Figure 1. In the model, both a Web API and a mashup (composed of APIs) are defined as subclasses of *Web mashup resource*, denoted by its URL, a human-readable name, the date in which resource has been created, and the time of the last upgrade. Each Web mashup resource is associated with a set of *terminological items*, which correspond to: (a) categories extracted from top-down classifications imposed within a given repository where the resource is registered; (b) a term with an explicit semantics, either a term extracted from a terminological thesaurus like WordNet or a concept extracted from an ontology in the Semantic Web context; and (c) a simple keyword or tag without an explicit representation of semantics. Each terminological item is described by a name, a type (namely, `C` for categories, `WD` for WordNet terms, `O` for ontological concepts, `K` for simple keywords or tags) and a set of properties:

– if the item is a category, the item name corresponds to the category name and it has a property that identifies the taxonomy or the classification to which the category belongs;

- if the item is extracted from WordNet, it has two properties, namely the list of synonyms of the term and the meaning associated with the WordNet sense to which the term belongs;
- if the item is a concept extracted from an ontology, its name corresponds to the concept name; it is further specified through its URI, which identifies the definition of the concept within the ontology where it is provided;
- if the item is a keyword or a tag without an explicit semantics, the item has no properties and its name corresponds to the keyword or the tag name.

We distinguish keywords and tags as follows: tags are assigned by designers, aimed at classifying resources in a folksonomy-like style, but keywords are recurrent terms extracted from resource textual descriptions using common IR techniques. Each Web mashup resource is also associated with ratings assigned by designers, who may be characterized by their development skill, either self-declared, as shown in [5], or estimated, using techniques such as those proposed in [13]. Designers may be either Web API providers, Web mashup owners, or Web API consumers who rate resources according to their personal opinion. Web APIs can be further characterized through technical features, such as protocols or data formats.[3]

To enrich the model, we rely on external ontologies, when available, following the methodological guidelines suggested in [14]. For instance, we modeled designers using the `foaf:Person` class from the FOAF (Friend of a Friend) ontology,[4] where contact information of each `foaf:Person` is composed of the name (`foaf:name`) and the homepage address (`foaf:homepage`).

4 Defining Links Among Web Mashup Resources

Formally, we represent a link $\mathcal{L}$ between Web mashup resources, either within the same repository or across different repositories, as follows:

$$\mathcal{L} = \langle \texttt{type}, s_URI, t_URI, \texttt{conf}, [\texttt{when}] \rangle \tag{1}$$

where s_URI and t_URI are the URIs of the resources that are source and target of the link, respectively, $\texttt{conf} \in [0, 1]$ is the confidence to set the link (obtained through identity and similarity metrics evaluation depending on the link type, as we describe below) and the optional element `when` denotes when the link has been established and therefore published as Linked Data. The `when` clause is exploited to filter out or check links older than a predefined number of days. During Web API search, the designer can choose a threshold $\texttt{conf}' \in [0, 1]$ such that only links with $\texttt{conf} \geq \texttt{conf}'$ are considered. We have identified two link types, `sameAs` and `simAs`, which we describe as follows:

- `sameAs` link between Web mashup resources, to denote that two resources registered in the repositories refer to the same software artifact; this kind of link

[3] See, for example, the technical description of the `LambdaLabs Face` API at: `http://www.programmableweb.com/api/lambdalabs-face`

[4] `http://xmlns.com/foaf/spec/`

makes sense only across different repositories, since we assume that the publication of the same Web mashup resource in the same repository is not allowed;
- `simAs` link between two Web mashup resources, established on the basis of a comparison between their terminological items; a further specification of this kind of link is provided to distinguish among similar APIs and similar mashups; in the former case, also technical features (such as protocols and data formats) are taken into account to establish the link; in the latter one, we must also consider that the more two mashups are similar, the more similar the Web APIs that compose them are; the `simAs` link can be set either within the same repository or across different repositories.

Each kind of link is identified through proper metrics evaluation, detailed in the following. A threshold $\tau \in [0, 1]$ is set in order to publish each link as shown in Section 6. In our approach, we fixed $\tau = 0.5$ to avoid storing loosely related resources. If the result of metrics evaluation is equal or greater than $\tau = 0.5$, the `conf` parameter introduced in Equation (1) is set to this value and the link is stored in the *Link Repository* as we discuss in Section 6.

The `sameAs` link between Web mashup resources. We say that two Web mashup resources across different vocabularies *reference* the same software component if they present the same URL. In this case, the link is set and `conf` = 1.0. Unfortunately, URL mismatches like the ones described in the motivating example make the use of additional similarity computations necessary. In particular, to manage these mismatches, the criteria used to set the `sameAs` link between Web mashup resources is based upon a *host name similarity* metric (*HostSim*). This metric is computed between the host names of two URLs (e.g., `http://api.lambdal.com` and `http://www.lambdal.com/developers/`, except for the scheme (e.g., `http://`) and the fragment (e.g., `/developers/`) [15]. Each host name is composed of substrings separated by dots, e.g., `www.lambdal.com`. Host name similarity is computed using the Dice formula in Equation (2):

$$HostSim(URI_1, URI_2) = 2 \cdot \frac{|URI_1 \cap URI_2|}{|URI_1| + |URI_2|} \in [0, 1] \tag{2}$$

where $|URI_1 \cap URI_2|$ denotes the maximum number of common substrings within URI_1 and URI_2 (in the exact order) starting from the top-level domain (e.g., `com`), $|URI_i|$ denotes the number of substrings within URI_i. The Dice formula is used to normalize the metric in the [0,1] range. For instance, *HostSim*(`www.lambdal.com`, `api.lambdal.com`) $= 2 \cdot \frac{2}{3+3} = 0.67$, because the two host names share the two `lambdal.com` substrings (underlined) on a total of 3+3 substrings. Since the same domain may be assigned to several Web mashup resources, the *HostSim* computation is also applied to the URLs of the designers who provided or own the resource (for Web APIs or mashups, respectively) and is combined with the string similarity between resource names, using one of the classical string distance metrics. It is out of the scope of our framework to propose a particular metric for string comparison. The framework includes the most common metrics [16]. In the future we may explore tuning strategies further. The overall

metric applied on two Web mashup resources res_1 and res_2, provided or owned by designers d_1 and d_2, respectively, is defined as:

$$\begin{aligned} 0.4 \cdot HostSim(res_1.\texttt{address}, res_2.\texttt{address}) + \\ 0.4 \cdot HostSim(d_1.\texttt{homepage}, d_2.\texttt{homepage}) + \\ 0.2 \cdot StringSim(res_1.\texttt{name}, res_2.\texttt{name}) \in [0, 1] \end{aligned} \quad (3)$$

where, for instance, res_1.`address` is the value of property `address` for the resource res_1; in our preliminary evaluation, for $StringSim()$ computation, we chose the Levenshtein distance metric. On the basis of the results of this experimentation, in the equation (3) $HostSim()$ is weighted more than string similarity between resource names. In fact, the latter is relevant only if we are within the same or very close domains. The value of the overall metric in this case will be assigned to the `conf` parameter.

The `simAs` link between Web mashup resources. The similarity between two Web mashup resources is deeply rooted in the comparison of their terminological items, that is:

$$TermSim(res_1, res_2) = \frac{2 \cdot \sum_{t_1 \in \mathcal{T}_1, t_2 \in \mathcal{T}_2} itemSim(t_1, t_2)}{|\mathcal{T}_1| + |\mathcal{T}_2|} \in [0, 1] \quad (4)$$

where we denote with $\mathcal{T}_i$ the set of terminological items used to characterize res_i, t_1 and t_2 are terminological items, $|\mathcal{T}_i|$ denotes the number of items in the set $\mathcal{T}_i$ and $itemSim(\cdot)$ values are aggregated through the Dice formula. The point here is how to compute $itemSim(t_1, t_2) \in [0, 1]$ given the different types of involved terminological items.

The algorithm for the $itemSim(\cdot)$ calculation is shown in Algorithm (1). When the types of t_1 and t_2 coincide, proper metrics from the literature are used for the comparison (see rows 1-8). In all the other cases, a comparison between the names of terminological items using the $StringSim(\cdot)$ metric is performed (row 18), except for the case of WordNet terms, that are expanded considering all the synonyms (rows 10-17)in order to look for a better matching term in the synset. In this version of the framework, we considered only the concept name in the case of ontological concepts. Future work will be devoted for refining this part, by expanding the set of terms extracted from the ontological concept with the name of other concepts connected through semantic relationships in the ontology.

For establishing `simAs` links between Web APIs, the $TermSim(\cdot)$ metric is equally balanced with the technical feature similarity $TechSim(\cdot) \in [0, 1]$, which evaluates how much the two Web APIs have common technical features, that is, $ApiSim(res_1, res_2) = 0.5 \cdot TermSim() + 0.5 \cdot TechSim()$ ($\in [0, 1]$). The value of the $ApiSim()$ metric in this case will be assigned to the `conf` parameter. Feature values are compared only within the context of the same feature. For example, if res_1 presents {`XML`, `JSON`, `JSONP`} as data formats and {`REST`} as protocol,

Algorithm 1. The $itemSim(\cdot)$ calculation algorithm

```
Input  : Two terminological items t1 and t2.
Output: The calculated itemSim(t1, t2) value.
1  if (t1.type == C) and (t2.type == C) then
2      itemSim(t1, t2) = Sim_cat(t1, t2) (using the Sim_cat defined in [5]);
3  else if (t1.type == WD) and (t2.type == WD) then
4      itemSim(t1, t2) = Sim_tag(t1, t2) (using the Sim_tag defined in [5]);
5  else if (t1.type == O) and (t2.type == O) then
6      itemSim(t1, t2) = H-MATCH(t1, t2) (using the H-MATCH function
       given in [17]);
7  else if (t1.type == K) and (t2.type == K) then
8      itemSim(t1, t2) = α · StringSim(t1, t2) (using the Levenshtein measure);
9  else
10     if t1.type == WD then
11         t1.bagOfWords = expandWithSynonyms(t1);
12     else
13         t1.bagOfWords = t1.name;
14     if t2.type == WD then
15         t2.bagOfWords = expandWithSynonyms(t2);
16     else
17         t2.bagOfWords = t2.name;
18     itemSim(t1, t2) = max_{i,j}{StringSim(t1^i, t2^j)}, where t1^i ∈ t1.bagOfWords
                         and t2^j ∈ t2.bagOfWords;
19 return itemSim(t1, t2);
```

while res_2 presents {XML, JSON} as data formats and {REST, Javascript, XML} as protocols, the $TechSim()$ value is computed as

$$\frac{2 \cdot [|\{\texttt{XML}, \texttt{JSON}, \texttt{JSONP}\} \cap \{\texttt{XML}, \texttt{JSON}\}| + |\{\texttt{REST}\} \cap \{\texttt{REST}, \texttt{Javascript}, \texttt{XML}\}|]}{|\{\texttt{XML}, \texttt{JSON}, \texttt{JSONP}\}| + |\{\texttt{XML}, \texttt{JSON}\}| + |\{\texttt{REST}\}| + |\{\texttt{REST}, \texttt{Javascript}, \texttt{XML}\}|} = 0.67 \tag{5}$$

In this example, XML is used both as data format and as XML-RPC protocol and it is considered separately in the two cases.

For establishing simAs links between mashups, the $TermSim(\cdot)$ metric is equally weighted with the mashup composition similarity $MashupCompSim(\cdot) \in [0, 1]$, which measures the degree of overlapping between two mashups as the number of common or similar APIs between them, that is

$$MashupCompSim(res_1, res_2) = \frac{2 \cdot \sum_{i,j} ApiSim(res_1^i, res_2^j)}{|res_1| + |res_2|} \tag{6}$$

where res_1^i and res_2^j are two Web APIs, used in res_1 and res_2 mashups, respectively, $|res_1|$ (resp., $|res_2|$) denotes the number of Web APIs in res_1 (resp.,

res_2) mashup and $ApiSim() = 1.0$ by construction when the two Web APIs are the same. Therefore, $MashupSim(res_1, res_2) = 0.5 \cdot TermSim() + 0.5 \cdot MashupCompSim()$ ($\in[0,1]$). The value of the $MashupSim()$ metric in this case will be assigned to the `conf` parameter.

5 Exploiting Links among Web mashup Resources

In this section, we present an applicative scenario for link-based view on repositories by defining and implementing a Web API search process. The process exploits the representation of repositories given according to the model of Section 3. The search process is formalized in Algorithm (2). A developer submits a query by specifying a set of keywords, a set of desired technical features, and an optional set of Web APIs that have been selected by the developer to be aggregated with the Web API to search for. Once results are obtained by merging the contents coming from considered repositories matching the query, they are filtered with respect to a set of required technical features, their appropriateness with respect to a given mashup (see below) or their popularity based on the number of developers and followers who are interested in a given Web API of the result. A query is formally defined as $\mathcal{Q} = \langle \mathcal{K}_\mathcal{Q}, \mathcal{F}_\mathcal{Q}, \mathcal{M}_\mathcal{Q} \rangle$, where $\mathcal{K}_\mathcal{Q}$ is the set of keywords, $\mathcal{F}_\mathcal{Q}$ is a set (possibly empty) of pairs $\langle$`tech_feature=value`$\rangle$, and $\mathcal{M}_\mathcal{Q}$ (possibly empty) is a mashup (that is, a set of Web APIs).

In the prototype implementation, in order to build an answer $\mathcal{R}(\mathcal{Q})$, a set of SPARQL queries are issued on the Virtuoso Universal Server[5]. Additionally, since we assume that the web designer is not confident with SPARQL, the Web interface lets the designer insert the components of the query $\mathcal{Q}$ and generates corresponding SPARQL queries.

When a query $\mathcal{Q}$ is evaluated, the content of each repository of Web APIs is inspected searching for those Web APIs that include in their terminological equipments at least one of the keywords specified in $\mathcal{Q}$ (rows 1-3). The inspection of terminological equipments is performed through the InTERM predicate that checks the presence of a keyword in the name attribute of a terminological item (see the model in Fig. 1) and, in case of a WordNet term, in the set of synonyms. In a second phase, for each retrieved Web API the `simAs` links are used to include in the result also those Web APIs that are similar to the retrieved ones (rows 4-5). In a third phase, descriptions of pairs of Web APIs related by `sameAs` links are merged by building a unified representation. Moreover, the original descriptions are removed from the result (rows 6-12).

An API r^i belonging to the query result $\mathcal{R}(\mathcal{Q})$ is modeled as 4-tuple:

$$\langle api_URIs, r^i_\mathcal{M}, r^i_\mathcal{D}, r^i_\mathcal{F} \rangle \tag{7}$$

where api_URIs are the URIs assigned to the Web API in the repositories, $r^i_\mathcal{M}$ is the set of mashups where the Web API has been used, identified by their URIs, $r^i_\mathcal{D}$ is the set of developers, consumers and followers who used the Web API (e.g., developers of mashups which contain the Web API are used for PW), $r^i_\mathcal{F}$ is the set of technical features of the Web API.

[5] `http://virtuoso.openlinksw.com/`

Algorithm 2. Linked Web API search

Input : the query $\mathcal{Q} = \langle \mathcal{K}_\mathcal{Q}, \mathcal{F}_\mathcal{Q}, \mathcal{M}_\mathcal{Q} \rangle$.
Output: $\mathcal{R}(\mathcal{Q})$, where $r^i \in \mathcal{R}(\mathcal{Q})$ is a 5-tuple $\langle$api_URIs,$r^i_\mathcal{M}$,$r^i_\mathcal{D}$,$r^i_\mathcal{F}$ $\rangle$.

```
1  foreach Web API Repository S do
2      foreach k∈K_Q do
3          R(Q) ← Web APIs W from S such that InTERM(k,W);
4  foreach Web API W∈ R(Q) do
5      add W′ to R(Q) such that W simAs W′ with conf ≥ conf';
6  foreach Web API W∈ R(Q) do
7      foreach Web API W′∈ R(Q) do
8          if (W sameAs W′ with conf ≥ conf') then
9              build W″ by merging W and W′;
10             add W" to R(Q);
11             remove W from R(Q);
12             remove W′ from R(Q);
13 if F_Q ≠ ∅ then
14     filter R(Q) with respect to the set F_Q;
15 if M_Q ≠ ∅ then
16     rank Web APIs in R(Q) according to their appropriateness wrt M_Q;
17     rank equally appropriate Web APIs in R(Q) according to their
       popularity;
18 else
19     rank Web APIs in R(Q) according to their popularity;
20 return R(Q);
```

The last search steps of Algorithm (2) concern filtering and ranking of search results. Retrieved Web APIs are filtered out according to the set of required features $\mathcal{F}_\mathcal{Q}$ if specified in $\mathcal{Q}$ (rows 13-14). Finally, search results are ranked according to their appropriateness with respect to the target mashup $\mathcal{M}_\mathcal{Q}$ if specified in $\mathcal{Q}$ (rows 12-14) and according to their popularity (row 16). Specifically, we define the similarity between two mashups $\mathcal{M}_1$ and $\mathcal{M}_2$ (as sets of Web APIs) using a formula according to the same rationale of Equation (4):

$$MashupSim(\mathcal{M}_1, \mathcal{M}_2) = \frac{2 \cdot |\mathcal{M}_1 \cap \mathcal{M}_2|}{|\mathcal{M}_1| + |\mathcal{M}_2|} \tag{8}$$

where $|\mathcal{M}_1 \cap \mathcal{M}_2|$ denotes the number of common Web APIs in the two mashups and $|\mathcal{M}_i|$ the number of Web APIs in the mashup $\mathcal{M}_i$. Given the set $r^i_\mathcal{M}$ of mashups of a search result r^i, if $r^i_\mathcal{M} \neq \emptyset$, the appropriateness of r^i with respect to the mashup $\mathcal{M}_\mathcal{Q}$ is given by $max_j\{MashupSim(\mathcal{M}_\mathcal{Q}, \mathcal{M}_j)\}$, where $\mathcal{M}_j \in r^i_\mathcal{M}$. If $r^i_\mathcal{M} = \emptyset$, then ranking based on appropriateness is not performed. Popularity of a result r^i is measured as the number of developers in $r^i_\mathcal{D}$. For ranking purposes, the designer may choose, through the search interface, to give priority to the appropriateness or to the popularity of results.

Example. Let us consider again the `LambdaLabs Face` face recognition API. In particular, suppose that after retrieving Web APIs from repositories (rows 1-3), this API is present in both PW and MP. In particular, the descriptions of the API in these sets is given in Table 1. Note that total number of followers and consumers of this API in `Mashape` is reported and that the number of developers that used the API in `ProgrammableWeb` is 0.

Table 1. Descriptions of `LambdaLabs Face` face recognition API

	PW	**MP**
URI	{`http://api.lambdal.com`}	{`http://www.lambdal.com/developers`}
$r^i_{\mathcal{M}}$	{}	{}
$\lvert r^i_{\mathcal{D}} \rvert$	0	2689
$r^i_{\mathcal{F}}$	$\{REST, JSON\}$	{}

A `sameAs` link is already set in the Link Repository based on the linking criteria presented in the previous section. Hence, these Web API descriptions are merged and added to $\mathcal{R}(\mathcal{Q})$ (rows 9-10). Specifically, the merged description presented in the query result will be, according to (7):

$$\langle \{\texttt{http://api...}, \texttt{http://www...}\}, \{\}, r^i_{\mathcal{D}}, \{REST, JSON\} \rangle$$

Because $r^i_{\mathcal{M}}$ is empty, the ranking based on mashup appropriateness is not performed, as explained above. On the contrary, if $\mathcal{M}_{\mathcal{Q}}$ is empty, the ranking based on the popularity, given by $|r^i_{\mathcal{D}}|$, of this API will be rather high.

Preliminary evaluation. We performed a preliminary evaluation of the search process based on classical IR measures of precision and recall. The aim is to check the capability of our approach to provide improved search results with respect to the separated use of the available repositories. A more extensive experimentation on the system implementing the process will be performed as future work. As a proof of concept, we started from two popular repositories, namely `ProgrammableWeb` (PW) and `Mashape` (MP), considered for the running example in Section 2.1. Observing the two repositories and their differences, we note that their search results strongly depend on the tags and categories used for search. Experiments have been run on an Intel laptop, with 2.53 GHz Core 2 CPU, 2GB RAM and Linux OS. We manually selected all the relevant Web APIs on face recognition stored within the PW and MP repositories and we collected the sets of tags and categories for the two Web APIs. We then issued several queries using different subsets of tags and categories: on the PW repository only, on the MP repository only, and on both the repositories through our approach. In Table 2 we report an excerpt of results of this preliminary experimentation, using the following subsets of tags: ⟨`face`,`recognition`⟩ and ⟨`facial`,`detection`⟩. We note that even if we consider the union of the results from the PW and MP repositories, queried separately, our approach presents better precision and recall results. The other aspect that can take advantage of

Table 2. Preliminary evaluation results

	⟨face,recognition⟩		⟨facial,detection⟩	
	Precision	**Recall**	**Precision**	**Recall**
ProgrammableWeb	0.72	0.64	(no results)	(no results)
Mashape	0.68	0.69	0.47	0.43
Union of PW and MP results (invoked separately)	0.73	0.70	0.40	0.41
Results from the joint use of PW and MP through our	0.93	0.91	0.77	0.70

our approach is the identification of corresponding Web APIs across different repositories: for instance, 75% of the face detection Web APIs that have been registered in both the repositories present different URLs. The action to reconcile Web APIs, if manually performed, would be time-consuming and error-prone, due to the dynamic nature of the two sources.

6 The Framework Architecture

The framework architecture is shown in Figure 2. The *Web API meta-repository* aims at enabling uniform access to the contents of individual repositories. It is based on the RDF Quad Store of the Virtuoso Universal Server, on which our approach is implemented and it includes the unified vocabulary and the *Link Repository*, to store similarity and identity links across distinct Web API repositories.

Once an RDF vocabulary for our unified model has been defined, the Virtuoso Sponger tool[6] has been properly configured in order to retrieve resources from the Web API repositories, according to their conceptualization in the vocabulary. In this way, resources are directly retrieved from the repositories by relying on the Virtuoso update procedures. These procedures are executed offline and are combined with link maintenance strategies mentioned in Section 3. To perform its tasks, Sponger relies on the *Virtuoso Content Crawler (VCC)*, which executes a periodic update of cached contents of PW and MP repositories, and a set of *cartridges* which are used to extract RDF tuples from the retrieved sources. A cartridge is composed of an interface for invocation (Cartridge Hook), an extractor to obtain (non-RDF) data from the source and a mapper that looks for correspondences between data extracted from the source and the RDF vocabulary that has been built for the source. The mapper is based on an XSLT document. Cartridges are stored within the *Cartridge Registry*. Cartridges have been designed to invoke specific methods made available by public repositories to query their contents[7]. It is clear that the effort of configuring a cartridge has to be done only one time for each repository that we want to include.

Currently, ProgrammableWeb and Mashape cartridges are available. Other proprietary Web API repositories may be provided by enterprises, which can inte-

[6] http://www.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtSponger

[7] See, for instance, http://api.programmableweb.com for the ProgrammableWeb repository or http://www.mashape.com/mashaper/mashape#!documentation for the Mashape repository.

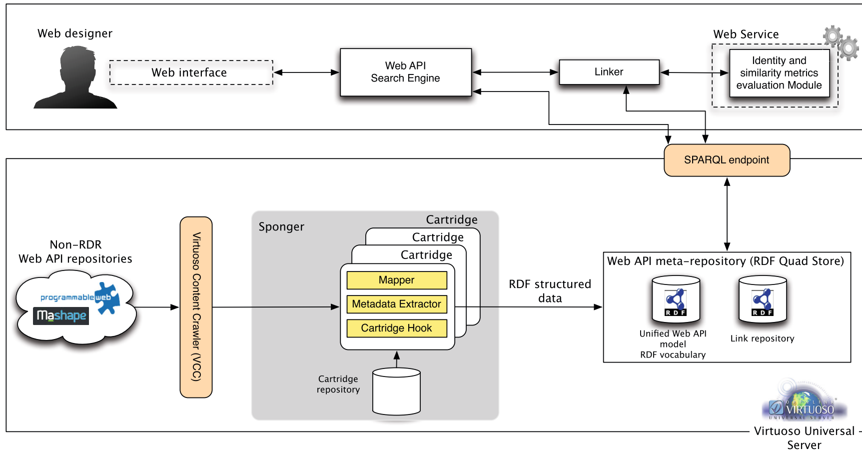

Fig. 2. The framework architecture

grate their own components with Web APIs made available within public repositories, thus adopting an "app store" development approach to reduce investment costs and development time for those applications that do not shape strategic decisions. In this case, new cartridges must be added to the system according to a modularized architecture. The contents stored within the Web API meta-repository are only accessible through the *SPARQL Endpoint*. The *Linker* is in charge of populating the Link Repository by performing evaluation of the metrics designed to identify semantic links across the repositories. To this aim, the Linker relies on the *Identity and Similarity Evaluator*, which implements the metrics as a Web Service.

Finally, the *Web API Search Engine* implements the search process by issuing SPARQL queries on the Virtuoso Universal Server in order to access the contents of different repositories. Due to the dynamic nature of Web API repositories, a links maintenance mechanism has been combined with the update procedures implemented by the Virtuoso Content Crawler.

7 Concluding Remarks

In this paper, we discussed an approach to link, according to Linked Data principles, contents of multiple Web API repositories, and how to exploit these links for Web API search purposes. The approach is based on a unified model for Web mashup resources. As future work, we plan to quantify how the productivity of Web designers is increased through the use of multiple repositories for Web API selection, where different repositories focus on complementary Web mashup resource descriptions. We have run preliminary experiments to test effectiveness of Web API search in terms of precision and recall, using two popular public repositories as a proof of concept.

References

1. Taheriyan, M., Knoblock, C., Szekely, P., Ambite, J.L.: Rapidly Integrating Services into the Linked Data Cloud. In: Proc. of the International Semantic Web Conference (ISWC), pp. 559–574 (2012)
2. Taheriyan, M., Knoblock, C., Szekely, P., Ambite, J.: Semi-Automatically Modeling Web APIs to Create Linked APIs. In: Proceedings of the ESWC 2012 Workshop on Linked APIs (2012)
3. Speiser, S., Harth, A.: Integrating linked data and services with linked data services. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part I. LNCS, vol. 6643, pp. 170–184. Springer, Heidelberg (2011)
4. Krummenacher, R., Norton, B., Marte, A.: Towards linked open services and processes. In: Proceedings of the Third Future Internet Conference, pp. 68–77 (2010)
5. Bianchini, D., De Antonellis, V., Melchiori, M.: A Multi-perspective Framework for Web API Search in Enterprise Mashup Design. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) CAiSE 2013. LNCS, vol. 7908, pp. 353–368. Springer, Heidelberg (2013)
6. Hartig, O., Langegger, A.: A database perspective on consuming linked data on the web. In: Datenbank-Spektrum, pp. 57–66 (2010)
7. Bianchini, D., De Antonellis, V., Melchiori, M., Salvi, D.: Semantic-enriched service discovery. In: Proc. of the 22nd International Conference on Data Engineering (ICDE), pp. 38–47 (2006)
8. Bianchini, D., De Antonellis, V.: Linked Data Services and Semantics-enabled Mashup. In: On Semantic Search on the Web, pp. 281–305. Springer (2012)
9. Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecky, J., Domingue, J.: iServe: a Linked Services Publishing Platform. In: Proceedings of ESWC Ontology Repositories and Editors for the Semantic Web (2010)
10. Norton, B., Krummenacher, R.: Consuming Dynamic Linked Data. In: Proc. of First International Workshop on Consuming Linked Data (2010)
11. Speiser, S., Harth, A.: Towards Linked Data Services, in: Proc. of the 9th International Semantic Web Conference, ISWC (2010)
12. Volz, J., Bizer, C., Gaedke, M., Kobilarov, G.: Discovering and Maintaining Links on the Web of Data. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 650–665. Springer, Heidelberg (2009)
13. Malik, Z., Bouguettaya, A.: RATEWeb: Reputation Assessment for Trust Establishment among Web Services. VLBD Journal 18, 885–911 (2009)
14. Villazón-Terrazas, B., Vilches, L., Corcho, O., Gómez-Pérez, A.: Methodological Guidelines for Publishing Government Linked Data. Springer, Heidelberg (2011)
15. Qi, X., Nie, L., Davison, B.: Measuring Similarity to Detect Qualified Links. In: Proc. of the 3rd Int. Workshop on Adversarial Information Retrieval on the Web, pp. 49–56 (2007)
16. Cohen, W., Ravikumar, P., Fienberg, S.: A comparison of string distance metrics for name-matching tasks. In: IJCAI-2003 Workshop on Information Integration on the Web, pp. 73–78 (2003)
17. Castano, S., Ferrara, A., Montanelli, S.: Matching Ontologies in Open Networked Systems: Techniques and Applications. Journal on Data Semantics 2, 25–63 (2006)