Linked Data Wrapping as a Service

Iker Azpeitia, Jon Iturrioz and Oscar Díaz
Onekin Research Group, Computer Science Faculty, University of the Basque Country UPV/EHU, Pº Manuel Lardizabal, 1 - 20018 Donostia-San Sebastián. Tel: 943 015245 Fax: 943 015590
Spain
E-mail: iker.azpeitia@ehu.eus, jon.iturrioz@ehu.eus, oscar.diaz@ehu.eus

Abstract. Platform as a service (PaaS) allows customers to develop, run, and manage applications without being involved in maintaining the associated infrastructure. We believe Linked Data Wrappers (LDWs) might well benefit from this kind of approach. LDWs have been proposed to integrate Open APIs into the Linked Data Cloud. Frequently, LDWs are developed in-house and its lifetime is that of the containing application. We advocate for LDWs to be externalized into third-party platforms. Besides the PaaS benefits, this approach addresses one of LDWs’ main hurdles: maintenance. API upgrades are LDWs’ Damocles sword. A LDW platform might well offer support for collaborative LDW definition and maintenance, extending their lifetime beyond their breakout applications, and in so doing, becoming a sound Semantic layer on top of existing Open APIs. This paper contributes to this vision through: (1) identifying a set of requirements for LDW platforms; (2) instantiating these requirements for Yahoo’s YQL; and (3) validating the extent to which this approach accounts for collaborative maintenance.

Keywords: Web APIs, Linked Data, Wrappers, YQL

1. Introduction

Open APIs are an important source of current data. The importance of APIs for external data consumption should not be underestimated. According to a report by the Harvard Business Revue, Salesforce generates 50% of its revenue through APIs, Expedia generates 90%, and eBay, 60%, to name a few [44]. Unfortunately, the lack of services directly exposing Linked Data is far too common. According to Programmable Web, a leading API yellow-page site, only 0.5% of APIs export their data using an RDF data format, being JSON-LD the preferable RDF format [20]. This might be due to several circumstances: technical (i.e. mapping the underlying data representation to Linked Data formats might not be trivial), social (i.e. no demand on Linked Data representation by the service community) or financial (i.e. no clear business model). Fortunately, in case the data is available under a liberal license [19], developers can wrap these services to expose Linked Data. We focus on this kind of wrappers, hereafter, referred to as Linked-Data Wrappers (LDWs) [18,26].

Commonly, LDWs are regarded as by-products of larger endeavors, e.g. developing a mashup application. Each application develops its own LDW, and its usage tends to be limited to this application. This is most unfortunate since the encapsulation of LDWs within their breakout applications prevents LDWs from being reused somewhere else, and hence, becoming truly RDF end-points. This prevents the Web of Data to tap into this too important source of current data.

This work advocates for a different architecture. LDWs are mainly data-driven, not contaminated by application semantics. This increases the chances of LDWs to be reused by applications other than the breakout application. Unfortunately, the LDW lifecycle is that of the containing application. The idiosyn-
The democratic and short-lived nature of some semantic applications (e.g., mashups), might lead to abandon the application, and thus, the LDW. However, LDW usefulness might well outlive their breakout applications. This grounds for LDWs to be externalized in a dedicated Linked-Data Platform [67].

Besides serving RDF resource dereferencing, a LDW Platform should also handle LDW collaborative definition and management. The expected benefits of LDW Platforms include:

- built-in infrastructure. Developers only need to care about the LDW definition. Deployment is handle by the LDW Platform.
- collaborative development. LDW continuous effort pays off if benefits go beyond breakout developers. So far, most LDWs are seldom used outside their research projects. By externalizing LDW definition, multiple developers can contribute to keep LDWs up and running.
- higher-level programming. Platforms can abstract developers from the heterogeneity of API requests and its optimization, making LDW definition more declarative, and hence, more accessible.

The hopes are in supporting one of the desires of the Web of Data: “the overall goal is one in which storage with the necessary functionality is a ubiquitous commodity, and application growth becomes dramatic as the provision of storage is decoupled from the design and deployment of applications” [24] (italics are ours). LDW-as-a-service might well pave the way towards this vision. A video about the different services is available at [http://rdf.onekin.org/](http://rdf.onekin.org/). This paper contributes to this vision through: (1) identifying a set of requirements for LDW platforms; (2) instantiating these requirements for Yahoo’s YQL; and (3) validating the extent to which this approach accounts for collaborative maintenance.

The rest of the paper is structured along the main landmarks of the Design Science methodology [45]. Section 2 introduces the practice, i.e. Linked-Data Applications. Section 3 identifies the problem that rises within this practice, i.e., LDW fragility. Section 4 draws six requirements for dedicated LDW Platforms. Section 5 provides some motivating scenarios. Section 6 to 11 introduce an instantiation of this approach: the SYQL platform. This platform is evaluated w.r.t. collaborative maintenance in Section 12. Related work and conclusions end the paper.

2. The practice: Linked Data Applications

A general architecture of Linked Data Applications exhibits three layers: Presentation layer, Logic layer and Data layer (see Figure 1). The Data layer provides tools to expose traditional data sources in RDF data formats. They include wrappers for the databases and LDWs (a.k.a. RDFizers) for transforming data from...
other formats (e.g., XML, JSON, and HTML) into RDF. Then, when all data is accessible as Linked Data, it might be stored in storages or accessed via Web APIs such as SPARQL endpoints. These data might be manipulated and integrated to access in a refined form via a SPARQL query interface by application code in the Logic layer.

From a data consumption perspective, three main architectural patterns have been identified. First, the Crawling Pattern where data is loaded in advance. Second, the Federated Query Pattern in which complex queries are submitted to a fixed set of data sources. And finally, the On-The-Fly Dereferencing Pattern where URIs are dereferenced at the moment that the application requires the data. We focus on the last one. This pattern retrieves up to date data but performance is affected when the application must dereference many URIs. Therefore, this approach might not scale up when bulky data sets need to be retrieved but it might fit scenarios where medium number of RDF resources need to be returned, frequently on demand. This is a common scenario when tapping into Open APIs.

As an example, consider the Flickr API. This Open API facilitates programmatic access to user pictures. Output formats include XML and JSON but not Linked Data. Figure 2 broadly describes the wrapping endeavor, i.e. moving from a JSON document in the left to a JSON-LD resource in the right. Notice that while a document is retrieved, a resource is dereferenced, i.e. resource content is obtained by dereferencing its URI. Hence, wrapping main tasks include:

- lowering: i.e. mapping the URI (e.g. \texttt{http://rdf.onekin.com/flickr/videoobject/itemNumber}) to the corresponding API call (e.g. \texttt{https://api.flickr.com/services/rest/?method=flickr.photos.getInfo&photoId=itemNumber}).

- API key handling: an API key is a code passed in by programs calling an API to identify the calling program, its developer, or its user to the website (e.g. Flickr). Normally, API keys serve to limit the number of times the API can be invoked in a certain period of time. For Flickr, this accounts for 3600 calls an hour. LDWs call APIs. Hence, they might require to get first an API key.

- lifting: creation of the Linked Data Resource from the API result (see the property-mapping arrow). Not all API data need to be exposed as Linked Data through a property mapping, and some semantic properties might be obtained from different API data as a calculation.

- interlinkage: most current APIs behave as data silos with no interlinkage with other resources. Hence, moving to the Linked Data Cloud might require not only a change in the output format but also setting links with other URI-addressable related sources. Figure 2 illustrates this scenario (see the association-mapping arrow). The resource holds references to the vivoweb ontology (through the property \texttt{schema:additionalType}), the weather report of the video’s location (through the \texttt{km4c:hasWeatherReport} property) and the...
video’s topics on DBpedia (through the property `schema:about`).

– metadata: information is embedded into the resource describing the resource provenance and linking the wrapper description (see the metadata box).

So far, some efforts have been devoted to LDW development [63,68,69]. However, LDW maintenance has been largely overlooked. This moves us to the next section.

3. The problem: LDW fragility

As any other piece of software, LDWs need to be maintained at the risk of becoming progressively obsolete. Indeed, examples can be found of LDWs that properly worked at the time they were launched but they were no longer up at the time of this writing: Flickr wrapper [26], GoogleArt project to RDF [6], OAPI2LOD IATI parser [12] or Twitter wrapper [14]. This provides evidences of LDW fragility. Next, we analyze the consequences and causes.

**Consequences** are twofold. First, the erosion of the role of LDWs as a sustainable foundation for the Web of Data. LDWs are well positioned to overcome the scarcity of interlinkage in the LOD Cloud. A recent study about the Linked Data Cloud concludes that only 56% of the 1014 datasets studied have external links [64]. In the same vein, Käfer et al. observe that, unlike the HTML world with an estimate of 25% in the number of new hyperlinks in a week period, LD seemed much more static [46]. The authors indicate that “this seems counter-intuitive in that LD itself is fundamentally comprised of URIs and thus, links”. Here, LDWs might offer a way out by defining wrappers that enhance interlinkage around LOD nodes. However, the steady introduction of new LOD nodes requires this interlinkage effort to be also continuous. Current LDW in-house development practices do not offer much help.

Second, LDW-intensive applications deteriorate. In an app setting, Bavota et al. [22] provide empirical evidence about the relation between the success of apps (in terms of user ratings), and the change- and fault-proneness of the underlying APIs. There is nothing to suggest that LDW-intensive applications are different. LDW fragility is a main risk for applications wanted to build a sound Semantic layer on top of existing Open APIs.

**As for the causes**, LDWs’ external dependencies are listed first. LDWs might exhibit dependencies with APIs, data ontologies and the LOD Cloud. Upgrades on these resources can make LDWs fall apart. By far, the most problematic is API upgrades. While use of static libraries can be maintained for as long as the client developer wishes, this is not the case with APIs where LDW developers are now in charge of pushing changes, in most cases, without warning. As noted in Espinha et al. [37] “the promise of loosely coupled web service APIs comes, in fact, at the cost of having changes forced upon the client developers”. Li et al. [54] identify sixteen API change patterns. Some patterns involve the API call (e.g. restrict access to API or parameter change) while other patterns affect the API response (e.g. change XML tag or structure). The bottom line is that similar problems will likely emerge for LDWs. In addition, the LOD Cloud and data ontologies might also suffer changes. Linked Open Vocabularies (LOV) database [9] is a case in point. This database stores every different version of a vocabulary over time. For instance, LOV reports 26 different versions of schema.org, 10 versions of FOAF, 3 DBpedia ontology versions or 13 Dublin Core Metadata versions. Each version might entail an upgrade on the LDW using the ontology.

This problem exacerbates when one considers:

– the frequency of change (APIs are reckoned to evolve regularly),

– the shortage of people involved (traditionally, research groups which might lack the resources for keeping LDWs up and running),

– the cost of keeping LDWs up and running (i.e. LDW infrastructure), or

– the limited lifetime of some applications.

Though little can be done as for LDWs’ external dependencies, some means can be envisaged to limit the above matters. Specifically, this work advocates to externalize LDWs into a dedicated platform. Next, we tackle the requirements for this solution.

4. Requirements for a LDW Platform

This section draws some (meta)requirements to battle back the aforementioned exacerbating factors. For each requirement, we look at existing solutions, if any. We ground the derived requirements on research on Software Engineering and the Semantic Web. Table[1] maps factors and addressing meta-requirements. Thus,
for example, allowing public inspection of LDWs (MR6), their quality (MR7) and their code (MR2 and MR3) it is a hope that new people is involved in LDW maintenance.

The vision is for LDWs to be externalized through a separated platform. Platform-as-a-Service (PaaS) is a category of cloud computing services that provides a platform allowing customers to develop, run and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app [48]. Here, the apps are the LDWs. This leads to our first requirement:

**MR1. Offer LDWs as services**

Rather than embedding LDWs as part of the application code, LDW should be available for others to tap into. One step further would be to facilitate the transition of LDW consumers to LDW developers. That is, striving to engage LDW consumers into their maintenance on the hope of increasing LDW longevity. This results in the second requirement:

**MR2. Allow for LDW code to be collaboratively developed**

Both “serviceness” and collaboration are not new to the semantic community. Indeed, different initiatives can be pigeonholed along these two dimensions. “Serviceness” indicates whether LDWs can be deployed on either a proprietary server or a third party server. “Collaboration” reflects whether LDW maintenance is only conducted by the LDW creator or rather it is open to the whole community. **D2RQ [1]** illustrates the first quadrant (see Table 2). This platform provides a proprietary server where LDW creators can customize automatically generated LDWs. **Bio2RDF [23]** opens maintenance to developers other than the creator. If we move to PaaS, **TWC LOGD [34]** is a popular approach with a focus mainly on creators. Finally, **DBpedia** illustrates a PaaS approach where the community can create and maintain the wrappers (wikitext templates) [53]. Unfortunately, this approach is DBpedia specific.

Previous requirements call for LDWs to be externalized in a dedicated platform. This platform is a LDW Platform. As such, it should comply with the W3C standard for resource management [67]. The difference stems from resources being dynamically obtained from API data at the time they are dereferenced. In other words, RDF resources are dynamically obtained through API calling, i.e. the platform does not hold data but LDWs. The vision is to move forward one of the desires of the Web of Data: “the overall goal is one in which storage with the necessary functionality is a ubiquitous commodity, and application growth becomes dramatic as the provision of storage is decoupled from the design and deployment of applications” [24]. But, what would it be the offerings of this dedicated platform? This results in five additional requirements.

**MR3. Easy LDW definition, even for the less skillful**

Different guidelines address the distinct concerns risen during LDW definition [41,42,64]. However, if the challenge expands beyond definition to sharing and re-use, other non-functional requirements come into play: understandability and affordability. A common approach to address these two issues is increasing the abstract level at which LDW are defined. Declarativeness increases understandability (good for consumers) as well as facilitating development (good for providers).

Being a dedicated platform, LDW Platforms can abstract developers from the heterogeneity of API requests and its optimization. This efforts is being pioneered by Yahoo’s YQL [62]. YQL is a query engine, which is hosted by Yahoo!, and exposed as a REST endpoint. YQL aims at hiding APIs’ specifics into a uniform table-like metaphor. To this end, petitions are specified in terms of a SQL-like language: the Yahoo Query Language (see later).

But even SQL can put some users off. Different compromises between expressiveness and learnability can be found in the literature depending on the target audience. Domain-specific approaches focus on specific data sources (e.g. **Wikipedia** or relational databases) which permit lowering and lifting to be built-in. This accounts for more declarative wrapper specifications that easy user involvement. In the case of **DBpedia**, this specification is realized in terms of wiki
templates, akin to the wiki origins of this initiative. In relational databases, wrapping is specified through R2RML ontology [32], where “TripleMaps” objects map tables and columns into RDF classes and properties, respectively. Departing from declarative specifications, other authors resort to general-purpose procedural languages (e.g. Bio2RDF), wrapper ontologies (e.g. TWC LOGD), or a mixture (e.g. SWEET [56]), depending on the target audience (i.e. programmers for Bio2RDF vs. the Semantic Web community for Karma [69]).

MR4. Allow for LDW registration.

LDWs must be registered on the Platform for the LDW’s URIs to be dereferenceable. The risk of errors expanding beyond the original authors to potential LDW consumers makes registration be specially demanding. In addition, backward compatibility issues should be considered. LDWs might be upgraded while on use. Hence, upgrades should not conflict with existing LDW consumers.

MR5. Allow for resource lookup.

URI’s are transformed to the underlying API calls and the results are lifted to a Linked Data representation. The result should be enriched with appropriate ontology metadata and provenance [27]. In addition, credential management should be cared of. Credentials are codes requested by the servers to verify the calls are being made through a valid account. API keys are the most common mechanism. An API key is a code passed in by programs calling an API to identify the calling program, its developer, or its user to the website. For instance, Flickr assigns an API key credential to every account and all requests must be accompanied by this API key. Two alternatives exist to credential provision. Credentials can be provided by the LDW producer at LDW specification time. Alternatively, credentials can be provided by the LDW consumer (i.e. the Linked Data Application) at dereferencing time.

MR6. Allow for LDW lookup.

This work intends to spur LDW sharing and reuse. This moves to the very front the ability to find and understand LDWs and APIs alike. Web Developers have to deal with heterogeneous and sometimes not up to date APIs’ textual descriptions. Finding, interpreting and invoking APIs requires extensive human involvement due to the lack of API machine-processable descriptions [57]. A little structure in architecting and documenting the APIs could greatly benefit application developers and reduce the amount of manual effort required when integrating multiple APIs (LDWs being a case in point) [70]. Here is where the Semantic community shines up. The use of ontologies for LDW description will provide an homogeneous semantic description (most important in a sharing setting) which would reduce developers effort while facilitating LDWs’ discoverability, composition and automated invocation. Existing initiatives include VoID (Vocabulary of Interlinked Datasets) [19] and Hydra [51].


Externalizing LDWs pays off as long as third-party applications reuse them. This requires trust which, in turn, calls for data quality. The quality of Linked Data is crucial if a LD Application ecosystem wants to be developed [33]. From this perspective, the quality of a LDW is that of the Linked Data it produces. Data quality is commonly conceived as “fitness for use for a certain application” [24]. That is, the quality range very much depends on the purposes of the application at hand. Linked Data is not an exception. Its use will certainly be affected by its quality. The fact that this Linked Data is obtained through a LDW does not change this point. However, if LDWs are externalized in a separated platform, this platform can act as a “quality checker” about LDW-sourced Linked Data. Promoting LDW usage implies providing measurement about the quality that are relevant to consumers [72]. Informing about this quality is then a major assignment for LDW Platforms.
To conclude, we advocate for a LDW Platform that facilitates the (re)use and collaborative development of LDWs. Next section builds the case for LDW externalization through different scenarios.

5. Building the case for LDW externalization

This section does not address LDW realization. Rather, it addresses the opportunities brought by externalizing LDW management into a dedicated platform. Figure 3 shows the vision. Applications do still encompass the three layers as described in the previous section. However, LDW concerns are externalized into a separated platform. This implies:

- at development time, LDWs are specified at the LDW Platform. Being a specialized platform, utilities can be offered to speed-up both specification and deployment. From the application’s perspective, API resources are now accessed as native RDF resources. From the developers’ perspective, the LDW is specified outside the application boundaries but in the LDW Platform.
- at runtime, LDWs are enacted at the time resources are looked up. From the application’s perspective, no difference should exist between static RDF resources, and RDF resources dynamically assembled.

This section introduces different usage scenarios, namely, annotation plug-ins, semantic mashups and RDF visualizers. Specifically, we use WordPress [16], LinkedWidgets [10] and LODmilla [13] as representatives of Content Management Systems (CMSs), mashup platforms, and RDF graph visualizers, respectively. For each platform, we developed an application where LDW needs are externalized. In this way, other developers can tap into existing LDWs and adapt them to their own needs. Backward compatibility is always required. We start by CMSs.

5.1. CMS platforms: WordPress

A CMS is a computer application that supports the creation and modification of digital content using a common user interface. WordPress is a popular open-source CMS. Here, content owners care about their pages ranking high in search engines. Recently, Google, Yahoo and Bing join forces to provide a standard for Rich Snippets, i.e. a JSON-LD formatted sample of a site’s content on Search Engine Result Pages [40]. Rich Snippets are used for organizations, reviews, people, recipes, events or videos. For instance, reference to an organization can be annotated with metadata about the business address, geographic coordinates, telephone, etc. Once Web content is marked up along the snippet directives, search engines can offer a more detailed account of your website, making it more enticing for users to click on, and easier for search engines to extract information [66]. The importance of Rich Snippets is highlighted by the fact that WordPress offers over two hundred plug-ins for WordPress sites to be annotated this way. Bloggers are provided with a snippet editor that inlays the corresponding Rich Snippet in the blog page when referring to let’s say, people or organizations.

So far, bloggers are prompted to introduce this metadata manually. However, it is not rare for this meta-
data already be available via an API. In this case, it is possible to develop a plug-in that obtains this information automatically from the website API rather than prompting the user. Figure 4 (a) provides an example. A new post is being edited that inlays a video taken from Flickr. (e.g. the video with ID '27376196615'). The plug-in provides annotation mark up (e.g. \[FlickrVideoObject id=videoID\]) for obtaining the Rich Snippet out of an API call to Flickr. Not only does this alleviate bloggers from introducing the metadata manually, but also avoids mismatches between the metadata provided by bloggers and the metadata already available through APIs. Unfortunately, these APIs rarely provide their output in JSON-LD. Therefore, the WordPress plug-in needs to handle the API call as well as the mapping from the API format to Rich Snippet.

Rather than embedding this wrapping functionality as part of the plug-in, we advocate for this functionality to be detached into a LDW Platform from where it can be re-used. Provided this is the case, our sample plug-in is reduced to the snippet in Figure 4 (b): the video ID is scrapped from the post rendering; the LDW URI is constructed (e.g. 'http://rdf.onekin.org/flickr/videoobject/' + video ID); finally, the URI is dereferenced and the Rich Snippet is obtained and embedded.

5.2. Semantic mashups: LinkedWidgets

Semantic mashups are mashup applications using RDF as its background data model, and SPARQL for tasks execution [47]. These applications offer new functionality by combining, aggregating, and transforming data available on the Web of Data. The benefits brought by Semantic technologies w.r.t traditional Web mashups, is the use of the RDF data model as the unified data model for combining data from heterogeneous data resources. Different tools have been proposed to empower end-users to create mashups [38,55]. LinkedWidgets is a case in point.

In LinkedWidgets, mashups are modeled as widgets that are orchestrated using a pipe-like approach. Figure 5 depicts a LinkedWidget mashup that involves arranging somebody’s Flickr pictures into Google Maps. It looks like Yahoo! Pipes but the novelty comes for the underlying data exchange technology: JSON-LD. This adds a semantic layer to the data and makes it machine-readable.

Although JSON-LD certainly improves interoperability, the low number of APIs offering this format requires a wrapping effort. This is the case for our sample problem. Flickr APIs need to be consulted to obtain the person’s pictures (flickr.person.ldw) and the picture metadata (flickr.videoobject.ldw) to be later displayed in the Google map. This wrapping effort might put some users off. Here, the notion of LDW as-a-service can alleviate this burden. Specifically, the LDW flickr.videoobject.ldw could have well be made available as a result of our previous WordPress plug-in scenario. Rather than starting from scratch, the LinkedWidgets user can tap into this effort, and extends flickr.videoobject.ldw to include the geo coordinates. Backward compatibility (i.e. keep existing properties) should ensure the WordPress plug-in keeps functioning.

5.3. RDF graph visualizers: LODmilla

Although the Semantic Web and the Linked-Open Data (LOD) Cloud are meant for machine processable data, their use by humans is being argued as being usually more accurate and richer than any human-
readable representation[58]. Linked Data exploration is being supported through different visualization tools[30,29]. An example is LODmilla. It permits navigate and explore multiple LOD datasets, save LOD views and share them with other users. For our purposes, the point to note is that LODmilla allows for links to be navigated dynamically, exploring the LOD in a personal way.

The question is to extent the exploration out of the existing LOD. There exists plenty of APIs out there to tap into. The current LOD can be idiosyncratically extended with dynamic resources obtained through API calls. All it is needed is the existence of LDWs that permit to close the chasm between API native format and JSON-LD, including interlinkage with existing LOD sources. Let’s take an example. We can initiate the exploration at a given Flickr user, and thereupon retrieve his pictures. This is straightforward with the two previously developed LDWs. In addition, we can interlinkage flickr.videoobject.ldw resources to other resources either LOD-based (e.g. DBpedia resources) or API-obtained (e.g. weather forecast through the Wunderground API[15]). Figure 6 depicts how this exploration looks like at LODmilla.

The exploration starts at a given Flickr person. Pictures are next explored (through the flickr.videoobject.ldw LDW). Using LODmilla facilities, users decide which picture properties to show up: ‘schema:interactionStatistic’ (i.e. the number of interactions for the video), ‘schema:about’ (i.e. the subject matter of the video), ‘km4c:hasWeatherReport’ (i.e. the weather report of the location where the video was taken), etc. Notice that ‘schema:about’ and ‘km4c:hasWeatherReport’ are links to DBpedia (LOD sourced) and Wunderground (API sourced), respectively. This permits to keep exploring on the basis of these resources. Specifically, ‘km4c:hasWeatherReport’ resources hold weather reports. This example requires a LDW for turning Wunderground API into a data set. In addition, the flickr.videoobject.ldw LDW needs to be upgraded adding two links ‘schema:about’ and ‘km4c:hasWeatherReport’ to point to DBpedia and Wunderground resources, respectively.

The bottom line is that LDWs permit to combine in the very same graph LOD-sourced resources and API-sourced resources, hence introducing a dynamicity that it is seldom obtained using LOD alone. In this way, LODmilla users can save the exploration to be next run periodically, and observe how dynamic data (e.g. interaction counters, weather forecast) changes.

Figure 7 summarizes our vision: LDWs are created, reused and upgraded by the community as developers confront wrapping needs in distinct scenarios.
As indicated in Section 6, we regard LDW Platforms as supporting five main requirements: LDW definition (MR3), LDW registration (MR4), Resource lookup (MR5), LDW lookup (MR6) and LDW quality (MR7). Next sections provide the details. A video showing the platform at work can be found at http://rdf.onekin.org/.

7. LDW definition

Section 6 introduces the distinct concerns that arise during LDW development. This section looks into two approaches to LDW definition available in SYQL. Each offer a different balance between expressiveness and learnability, and hence target different audiences. For API programmers, we draw on YQL. For less programming-oriented users, we resort to annotation.

7.1. LDWs as YQL's ODTs

YQL is a query engine, which is hosted by Yahoo!, and exposed as a REST endpoint. Petitions are specified in terms of a SQL-like language: the Yahoo Query Language. Here, we will use "YQL" to denote both, i.e. the platform and the language, unless the context does not make clear which one we refer to. This subsection provides an YQL outline to subsequently show how this framework is used for LDW implementation.

7.1.1. YQL basics

YQL aims at hiding APIs’ specifics into a uniform table-like metaphor. To this end, it resorts to a SQL-like syntax. As an example, the following YQL statement retrieves Flickr data about the photo whose ID is 27376196615 (see Figure 9):

```
select * from flickr.photos.info where photo_id= "27376196615" and api_key = "4fb031bf5b2f138576d011ff37f31565"
```

This setting is achieved through three mechanisms: the Yahoo Query Language, Open Data Tables (ODT), and the YQL Console.

The YQL Language. YQL includes SELECT, INSERT, UPDATE and DELETE statements that permits to handle API petitions à la SQL. Behind the curtains,
YQL maps these statements into the corresponding API methods. To this end, developers should provide Open Data Tables.

Open Data Tables (ODTs). Broadly, ODTs are syntactic sugar for API parameters. Figure 10 shows the `flickr.photos.info` ODT. Main tags include `<meta>` and `<bindings>`. The former contains descriptive information about the ODT such as author, description or documentation link (lines 3-8). Bindings (lines 9-17) indicate how SQL operations are mapped into API calls. An entry exists for each operation (e.g. `<select>`, `<insert>`). The snippet illustrates the SELECT case (lines 10-16): `<url>` accounts for the URL pattern to invoke (line 11) whereas `<inputs>` denotes the possible YQL statement input fields (lines 12-15). Each field (e.g. `photo_id`) accounts for variables to be instantiated when SELECT is enacted. ODTs hold all the intricacies of the underlying APIs. Specifically, benefits can be obtained from reusing of the authorization and authentication code from YQL, given the many API access control mechanisms. In this way, YQL offloads processing that programmers would normally do on the client/server side to the YQL engine. Besides those provided by YQL itself (known as “built-in tables”), ODTs can be provided by developers (known as “community tables”). A full list of community tables can be found at \[http://www.datatables.org/\].

The YQL Console. The YQL Console [61] enables to run YQL statements interactively from a browser (see Figure 9). Community tables are listed on the left. Choose an ODT and next, provide a YQL statement (e.g. SELECT) for the output to show up. If right, the REST counterpart of this query appears at the bottom, ready to be embedded in the application. Next, we look at how this approach can be extended for YQL queries to output JSON-LD, and for URIs to be dereferenced in terms of REST calls.

7.1.2. LDWs as ODTs

This subsection resorts to YQL expressiveness to define LDWs. As an example, we tap into the ODT in
Figure 10 and turned it into a LDW (see Figure 11). The process includes three main steps: lowering, lifting and credential handling.

**Lowering** (i.e. mapping the URI’s (e.g. http://rdf.onekin.com/flickr/videoobject/{itemNumber}) to the corresponding API call (e.g. https://api.flickr.com/services/rest/?method=flickr.photos.getInfo&photo_id={itemNumber})). YQL’s sampleQuery tag is used to describe the lowering through the URI pattern (line 8) and some URI examples (line 9). When the SYQL platform receives a URI (e.g. http://rdf.onekin.org/flickr/videoobject/27376196615), it dynamically identifies the ODT at hand through pattern matching against the registered URIPatterns. The lowering mapping from the URIPattern to the ODT input parameters is realized through pattern matching (i.e. line 8 to line 17 {photo_id} binding).

**Lifting** (i.e. creation of the Linked Data Resource from the API result). YQL’s function tag is recast for lifting: each YQL tuple (i.e. oneXML converted to oneJSON, line 28) is to be turned into an RDF resource serialized as JSON-LD (i.e. oneJSONLD, line 29). The lifting function holds <inputs> and <execute> tags. The former indicates the function’s parameters which are set to <pipe> (i.e. holds a result “tuple” of the ODT table described à la XML) (line 24) and <key> (i.e. to cast the URI for the returned RDF resource) (line 25). As for <execute> (lines 27-46), it holds the JavaScript code that obtains JSON-LD from the XML tuple (i.e. from oneXML pipe input to oneJSONLD). The line 28 parses the oneXML input to a JSON object. The lines 31 and 32 create the namespace and the type of the resource respectively. Lines 33-39 create some RDF properties from oneJSON parameters. Interlinkage is also described here by constructing URIs out of existing parameters, e.g. km4c:hasWeatherReport links to the Wunderground resource about the locality’s weather report (line 40). Noteworthy, this interlinkage is realized through another LDW!

**Credential Handling.** Credentials are codes requested by the servers to verify the calls are being made through a valid account. The question arise about whether these keys should be provided by either the LDW producer or the LDW consumer. When performance is not an issue (the number of invocations per API key is limited), LDW consumers can stick to the producer’s API key. In this case, the API key is embedded in the LDW itself (see Figure 11 lines 18-19). In this way, all looks up will reuse the same API key. In a more demanding setting, the extensive use of the same API key could cause a capacity bottleneck so LDW producers might resort to API keys being provided by consumers at lookup time (see Section 9).

**Demo time**

Let’s check this out. Go to the YQL Console at http://developer.yahoo.com/yql/console/

Paste the following snippet removing the line breaks:

```
use "https://raw.githubusercontent.com/onekin/ldw/master/flickr/flickr.videoobject.ldw" as flickrLDW;
select * from flickrLDW where photo_id= "27376196615" | flickrLDW.lifting ("http://rdf.onekin.org/flickr/videoobject/27376196615");
```

The ‘use’ clause holds the path to the Flickr LDW. Sentence select retrieves XML documents associated with photo “27376196615”. The result is piped (using the “|” operator) to a flickrLDW.lifting function that
returns its JSON-LD counterpart. More examples of LDW's can be found at [https://github.com/onekin/ldw](https://github.com/onekin/ldw).

### 7.2. LDWs by Example

Lowering and lifting might be complex enough to require the full-expressiveness of JavaScript. However, it is not unusual for this mapping between XML documents and JSON-LD resources to be concisely described using annotations, i.e. mappings between XML tags and ontology concepts. This is specially interesting for some scenarios (e.g. mashup applications) where the target audience might lack JavaScript skills. This subsection introduces an annotation facility built on top of the YQL Console so that LDWs are created out of annotations on top of API outputs (i.e. XML documents).

Figure 12 depicts how the YQL Console is being extended with a third tab: **Annotation View**. This tab...
mimics the “Formatted View” tab but now each XML tag is transformed into a mapping button. Buttons’ background color is used to denote whether the tag has already been annotated or not (deep blue and pale blue indicates whether the tag has been annotated or not, respectively). Click to indicate how the tag is lifted to its RDF counterpart. Annotation options include 12:

- Class type mapping (window 1): the tag element accounts for a resource (e.g. the 27376196615 Flickr video). Users are prompted to introduce the ontology and the class to define the RDF resources’ type.
- Property mapping (window 2): the tag element accounts for an RDF property (e.g. the views tag is mapped to the schema:interactionCount property).
- Association mapping (window 3): the tag element accounts for an RDF association. In this

---

1 The triggered operations are context aware, depending on both the tag annotation depth and the previously realized operations.
scenario, the XML element is used to create an interlink to another LOD resource. In the example, the photo’s <media> element is lifted to VivoWeb URI: the http://vivoweb.org/ontology/core#video is created from the video value. It is hypothesized that LDW exposure to a large community might well imply more heterogeneous data needs, hence increasing the pressure for interlinkages.

– Nested resource mapping (window 4): the tag element accounts for a resource. As an example, the <owner> tag in a <photo> stands for a Person resource held in schema:creator. This resource holds the name, username and location properties.

Demo time

Let’s define a LDW through annotation. The annotation facility is supported as a Firefox 47.0 plug-in on top of the YQL Console. Follow these steps:

1. install the Greasemonkey Firefox add-on [7].
3. navigate to the YQL Console.
4. click on the flickr.photos.info ODT.
5. add to the WHERE clause of the SELECT statement the additional predicate: ‘ and api_key= “4fb031bf5b2f13857d011fj37f31565” ’.
6. click the “test” button.

The resulting output should look like the screen-shot in Figure [2] Click on the Annotation View. The XML document is layered with a set of buttons. Proceed to annotate the XML tags to be turned into properties. Normally, the root of the document stands for the class. Click on its button to introduce the annotation ontology. A subset of ontologies from the Linked Open Vocabularies service (http://lov.okfn.org/dataset/lov/) are offered, though this can be configured. Once the annotation ontology is chosen, continue with the rest of the XML tags. At any time, the outcome of the annotation so far can be obtained by moving to the Semantic View tab. Moving back and forth between the Annotation View and the Semantic View increases user awareness about how LDWs are being developed. Once satisfied with the output, click the Generate button to obtain the LDW. Figure [1] displays the output automatically generated from the annotation.

Looking at the generated code, we observe how lifting is supported in the JavaScript part as a set of assignments: data from the XML document (right side) is assigned to a property (left side). As for lowering, it is specified through the <sampleQuery> using URIPattern and URIExample conventions (lines 8-9). Code generation goes as follows:

– URIPattern. It is automatically derived from the YQL query example and the resource’s class name along the URI template [59]: http://rdf.onekin.org/[Service]/[ClassName]/[{param-name}]* where /Service/ is the name of the Web Service obtained from the ODT table name (i.e. flickr.photos.info). Next, /ClassName/ stands for the resource’s class being lifted (i.e. videoobject). Finally, /param-name/ is obtained from the properties of the WHERE part of the query (i.e. photo_id and api_key). For our example, the outcome is: http://rdf.onekin.org/flickr/videoobject/{photo_id}/{api_key}. It is worthy to note that the API key should not be in the URI template since API keys are interchangeable and URIs must be stable. Therefore, the “/api_key/” must be removed from the URI.

– URIExample. It is obtained from the values of the WHERE part of the YQL query (see line 9). For the same reason as aforementioned, the API key value is removed from the URIExample and pasted as default value in the api_key input parameter (line 19). This is an example of credentials provided by the LDW producer.

The aim is for the generated code to be self sufficient (if credentials are not required the URIPattern and URIExample are not modified). Should more elaborated mappings be needed, programmers can tune this code at wish. Now, we are ready to registry this LDW by clicking the Register button of Figure [1].

8. LDW registration

Once defined, LDWs need to be registered before being stored at the Github repository. Registration also takes place through the YQL Editor. Besides setting the different registries, LDW registration also includes quality verifications. After all, this is a reuse architecture where eventual errors expand beyond the original authors to potential LDW consumers. So far, three types of verifications are conducted, namely, syntactic, dereferenced-based and backward compatibility. Failure to meet any of them prevents the LDW from being registered.
Syntactic verification. It checks whether LDWs are schema compliant. Through an XML Schema parser, distinct syntactic errors are pointed out: no URIPattern, no URIExample, lifting <function> does not exist; LDW badly parameterized.

Dereference verification. LDW definitions include URIExamples. At registration time, LDWs are put to the test using these URIExamples. Possible errors include: not enough credentials, badly formatted URI, no resource returned, or JavaScript errors.

Backward compatibility verification. LDWs are shared artifacts. This involves that the very same LDW may be being enhanced at the time it is being in use in other applications. This calls for backward compatibility so that existing applications receive what they expect. This is fleshed out by enhancements adding but never removing properties/associations of the LDW at hand. In addition, the URIPattern structure must also be preserved.

Demo time
LDWs are edited through the YQL Editor (see Figure 11). This editor holds a “Register” button. Click on it to move to the Verification Console (see Figure 13). This console displays the result of the verification process. If verification issues are arisen, they must be solved on the editor. Syntactic issues indicate the lack of essential parts (e.g. the URI example is lost). Dereferencing issues appear if there are errors on either the API call or the lifting process. Finally, compatibility issues imply that previously existing properties have been removed.

To check the XML parser, you could remove the URIExample from the current LDW and next click the “Register” button. The registration window should alert about the lack of the URIExample. Similarly, a dereference error could be produced by introducing an incorrect JavaScript code into the lifting function. For example, write the “5 = 6” incorrect assignment at the beginning of the lifting function. This will result in the creation of a void LD resource. Once corrected, acknowledgement messages should pop up in the Verification Console (see Figure 13).

9. Resource lookup
Once registered, the LDW Platform starts dereferencing URIs that conform to the LDW’s URIPattern.
URI dereferencing involves five main tasks (see Figure 14):

1. LDW retrieval, where the wrapper is downloaded from the LDW repository;
2. lowering, where the YQL select statement is prepared;
3. API calling, where the select statement is enacted, and the XML document obtained;
4. lifting, where the XML document is turned into an RDF resource; and finally
5. metadata enrichment, where dataset and provenance metadata are added.

The latter deserves further attention. The ability to track the data origin is a key component in building trustworthy, reliable applications [27]. Hence, SYQL resorts to the Provenance Ontology (http://purl.org/net/provenance/ns#) to enrich resources (see Figure 2). Specifically, the following properties are introduced: prv:usedGuideline, to indicate how the data has been created (e.g. pointing to the LDW URL); prv:performedBy, to denote who performs the wrapping process (e.g. SYQL http://rdf.onekin.org); and prv:usedData, to describe the data source. Furthermore, dataset metadata is supported through the void:inDataset property that links the resource to its VoID. This URI is also dereferenceable.

Credentials can be embedded into the LDW itself as shown in Subsection 7.1.2. Here, dereferencing is realized through an HTTP call. For instance, look up the http://rdf.onekin.org/flickr/videoobject/27376196615 URI. In this case, the LDW contains the API key, and the resource is returned. Alternatively, API keys can be provided at lookup time by LDW consumers. This is the scenario depicted in Figure 14 through the alt cases.

As an example, consider the lookup of a Flickr object but this time the API key is to be programmatically provided by the LDW consumer. The request holds a lookup URI and a request header (i.e. Authorization header) providing the API key value (i.e. schema:value) using Hydra credential description (see later):

```
GET /flickr/videoobject/27376196615
HTTP/1.1 Host: rdf.onekin.org Accept: application/ld+json
Authorization = {"hydra:supportedProperty": [{"hydra:property": {"schema:name": "api_key", "schema:value": "2c894ba749b4137b6f7ab127c86890ec"}]}]
```

For security reasons, the Authorization header should be encrypted but for understanding sake, we will not
encrypt it. SYQL recovers the api_key from the header and embeds it in the API call (see Figure 14). This API key takes precedence over the one embedded in the LDW.

**Demo time**

Consumer-provided API keys are provided programmatically. To simulate an application call, we resort to Hurl. Figure 15 shows the case for resource http://rdf.onekin.org/flickr/videoobject/27376196615. The Authorization header corresponds to the one described in the previous Hydra snippet. Launch the request and the response should show the 27376196615 Flickr video’s description.

So far, so good. But dereferencing might fail. Check it again introducing an incorrect API key value. For example, replace the “2c894ba749b4137b6f7ab127c86890ec” value with the incorrect “1234567890” API key. In this case, SYQL does not recover the Flickr video but still includes metadata about the data source (see metadata box in Figure 2). Consumers can peer at this metadata to find a way out. This moves us to the next section.

Fig. 15. Using Hurl to mimic resource lookup using consumer-provided keys.

10. LDW lookup

Finding, interpreting and invoking LDWs might be facilitated by the use of ontologies for LDW description [35]. This will provide an homogeneous semantic description (most important in a sharing setting) which would reduce developers effort while facilitating LDWs’ discoverability, composition and automated invocation. To this end, SYQL resort to the combined use of VoID (Vocabulary of Interlinked Datasets) [19] and Hydra [51].

VoID increases discoverability and facilitates metadata consumption from multiple LDWs [42,64]. Since resources are generated on the fly through an underlying API, it is not possible to work out statistical information (e.g. the number of entities stored in the API’s service). However, other structural metadata is provided. Namely, the example and the base URIs, the pattern of dereferenceable URIs, the ontologies and properties used, and the API URL (see Figure 16).

VoID is general purpose, i.e. it describes no matter the datasource. If this datasource is a LDW, more specific information can be provided. This grounds the use of Hydra. Hydra is a lightweight vocabulary to describe Web APIs and to augment Linked Data with hypermedia controls. It enables the creation of interoperable Web APIs that are accessible by generic clients. SYQL resorts to hydra:ApiDocumentation to describe the LDW itself (rather than the underlying API). The
derivative `hydra:ApiDocumentation` describes basic API concepts: the supported class (i.e., the class of the RDF resources and its properties), the lookup operation and the required credentials.

**Demo time**

A common scenario for LDW lookup is when consumers face problems in dereferencing URIs. If difficulties arise, SYQL does not return an empty answer but metadata is provided, specifically `void:inDataset`. This property holds a LDW resource. Let’s consider the case for the following LDW: `http://rdf.onekin.org/flickr/videoobject/{photo_id}`. Dereferencing this URI returns the resource in Figure 16. Property `hydra:apiDocumentation` holds an Hydra resource which, in turn, holds `hydra:supportedClass`. Figure 17 shows this information using the Hydra Console.

11. LDW quality checker

Externalizing LDWs pays off as long as third-party applications reuse them. This requires trust which, in turn, calls for data quality. From this perspective, the quality of a LDW is that of the Linked Data it produces. SYQL focuses on two main data quality dimensions: accessibility and intrinsic properties. Both dimensions are of importance for consumers and producers alike. For potential LDW consumers, to decide whether to use the LDWs. For producers, to find ways to improve their LDWs. Hereafter, Zaveri et al.’s quality framework is used.

**Accessibility subdimensions.** It involves aspects related to “the access, authenticity and retrieval of data to obtain either the entire or some portion of the data (or from another linked dataset) for a particular use case” [72]. Table 3 outlines how SYQL addresses these subdimensions along Zaveri et al.’s framework. For instance, availability is checked out by dereferencing the LDW’s sample URIs. So, sample URIs act as a sort of regression testing bucket.

**Intrinsic subdimensions.** This refers to whether information correctly (syntactically and semantically), compactly and completely represents the real world, and whether information is logically consistent in itself, independently of the user’s context. Table 4 indicates how these aspects find their way in SYQL.

Since LDWs rest on third-party resources (i.e., APIs), availability and representation are beyond the control of LDW developers. This substantiates the introduction of a “health checker” that periodically monitors LDWs to inform about current data quality mea-
Table 3

Accessibility subdimensions. “Abr” stands for the abbreviation used in [72]

<table>
<thead>
<tr>
<th>Subdimension</th>
<th>Abr</th>
<th>Metric</th>
<th>SYQL realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>A3</td>
<td>Dereferenceability of the URI</td>
<td>Checking sample URIs work</td>
</tr>
<tr>
<td>Interlinking</td>
<td>I1</td>
<td>Detection of good quality interlinks</td>
<td>Checking sample URI output interlinks are dereferenceable</td>
</tr>
<tr>
<td></td>
<td>I2</td>
<td>Existence of links to external data providers</td>
<td>Checking sample URI output interlinks exist</td>
</tr>
<tr>
<td>Security</td>
<td>S1</td>
<td>Usage of digital signatures</td>
<td>Using credentials at URI lookup time</td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>Authenticity of a dataset</td>
<td>Returning information about provenance and VoID</td>
</tr>
<tr>
<td>Performance</td>
<td>P2</td>
<td>Low latency</td>
<td>(minimum) Delay between submission of a request by the user and reception of the response from the system</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>High throughput</td>
<td>(maximum) Number of answered HTTP-requests per second</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>Scalability of a data source</td>
<td>Detection of whether the time to answer an amount of ten requests divided by ten is not longer than the time it takes to answer one request</td>
</tr>
</tbody>
</table>

Table 4

Intrinsic subdimensions. “Abr” stands for the abbreviation used in [72]

<table>
<thead>
<tr>
<th>Subdimension</th>
<th>Abr</th>
<th>Metric</th>
<th>SYQL realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntactic validity</td>
<td>SV1</td>
<td>No syntax errors of the document</td>
<td>Checking RDF/XML syntax errors</td>
</tr>
<tr>
<td></td>
<td>SV2</td>
<td>Syntactically accurate values</td>
<td>Classes and properties pertain to a dereferenceable ontology</td>
</tr>
<tr>
<td>Consistency</td>
<td>CS4</td>
<td>Deprecated properties or classes</td>
<td>Detection of owl:DeprecatedClass, owl:DeprecatedProperty or the deprecated word into descriptions</td>
</tr>
<tr>
<td>Conciseness</td>
<td>CN1</td>
<td>High intensional conciseness</td>
<td>Check for redundant attributes</td>
</tr>
<tr>
<td>Completeness</td>
<td>CM2</td>
<td>Property completeness</td>
<td>Rate of XML tags lifted in the LDW</td>
</tr>
<tr>
<td></td>
<td>CM4</td>
<td>Interlinking completeness</td>
<td>Rate of XML tags used to link to datasets</td>
</tr>
</tbody>
</table>

SYQL provides such a health checker. Figure 18 depicts a possible output for a list of registered LDWs: red square indicates that the LDW does not work (i.e. returns an error status code); orange square means that the LDW works but the result is empty; blue square indicates that the LDW works but fails in some quality subdimensions; finally, green square denotes that the LDW works and passes all the quality filters.

**Demo time**

Go to the health-checker console at [http://rdf.onedin.org/ldw/page/healthchecker/](http://rdf.onedin.org/ldw/page/healthchecker/). Click on the flickr:videoobject LDW to check it out. Some “warnings” pop up (see Figure 18):

- Accessibility dimension. The schema:additional Type contains a broken link (the I1 subdimension). Rationales: the “Video” class in the [http://vivoweb.org/ontology/core#](http://vivoweb.org/ontology/core#) ontology has the first letter in uppercase whereas the source word used in the mapping has the first letter in lowercase (see the Association mapping in Figure 12). Go to YQL Editor, change this first letter to capital in the lifting function code. Check out again.

- Intrinsic dimension. Four warnings reported. First (the SV2 subdimension), the “lat” and “lon” properties are not in an ontology. They should refer to “geo:latitude” and “geo:longitude”. Second (the CS4 subdimension), the schema:interactionCount is deprecated. Third (the CM2 subdimension), the ratio of mapped attributes to properties is very low. And fourth (the CM4 subdimension), the ratio of used attributes to create interlinkages is very low.

12. Evaluation

This work addresses LDW fragility. The focus is not so much on development but facilitating LDW maintenance. The strategy: externalizing LDW development. The rationales: a dedicated platform might make this effort sharable while introducing dedicated functionality geared towards facilitating LDW devel-
Fig. 18. Health-checker console. Quality assessment for `flickr.videoobject`: red (4 LDWs), orange (6 LDWs), blue (8 LDWs) and green (10 LDWs) squares warn about the LDWs’ states.

Development. This section evaluates to what extent this aim is achieved.

The success of the approach rests on a wide variety of concerns, from the LDW Platform throughput to the willingness of LDW authors to share their efforts or how to engage third parties. The former was addressed in a previous publication [43]. This section moves the focus to engaging third parties in maintaining LDWs. From this perspective, quality-in-use becomes paramount. Third parties should find far easier to extend/maintain an existing LDW than developing their own from scratch. This begs the question of how “usable” is SYQL as for extending/maintaining existing LDWs: `flickr.videoobject.ldw` (see Figure 11). Subjects need to address the following maintenance scenarios:

- **API evolution.** This might impact both the lowering and lifting of LDWs. Two scenarios are considered:

  * **Task 1.1:** API key expiration. No data is retrieved from the API. Subjects should update LDW’s API key. Difficulties include identifying and changing credentials.
  * **Task 1.2:** API resulting document structure changes. A property lifting stops working. Difficulties include recreating the mapping between the attribute and the property.

- **Ontology upgrades.** This might impact class mappings and property mappings.

  * **Task 2.1:** Switching `dc:subject` for `dcterms:subject`[^5]. Difficulties include transforming an Attribute mapping to an Association mapping.
  * **Task 2.2:** Class definition. Retype RDF resources as `schema:VideoObject`. To increase discoverability and reusability, the use of general purpose ontologies as de-facto standards is recommended. In addition, more specific classes into the class hierarchy is recommended too. For instance, `schema:VideoObject` is a sub-

[^5]: This is a real case where Dublin Core refined the `dc` namespace by `dcterms`. The `dcterms:subject` range suggests to use a non-literal value (e.g. `http://dbpedia.org/resource/Spain`) instead of a literal value (e.g. Spain).
Table 5

Time spent on each task (in minutes). Times are rounded up to a half minute

<table>
<thead>
<tr>
<th>Task</th>
<th>Subtask</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1.1</td>
<td>API related maintenance. Expired credential</td>
<td>3.5</td>
<td>2.5</td>
<td>2</td>
<td>3.5</td>
<td>4.5</td>
<td>4.5</td>
<td>2.5</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3.2</td>
</tr>
<tr>
<td>Task 1.2</td>
<td>API related maintenance. Changed path</td>
<td>7</td>
<td>5</td>
<td>5.5</td>
<td>6.5</td>
<td>6.5</td>
<td>5</td>
<td>4.5</td>
<td>7</td>
<td>4.5</td>
<td>6.5</td>
<td>5.8</td>
</tr>
<tr>
<td>Task 2.1</td>
<td>Ontology related maintenance. Property evolution</td>
<td>3</td>
<td>2</td>
<td>2.5</td>
<td>3.5</td>
<td>3</td>
<td>3.5</td>
<td>2</td>
<td>3</td>
<td>2.5</td>
<td>3.5</td>
<td>2.9</td>
</tr>
<tr>
<td>Task 2.2</td>
<td>Ontology related maintenance. Class redefinition</td>
<td>1</td>
<td>1.5</td>
<td>1</td>
<td>2</td>
<td>2.5</td>
<td>2</td>
<td>1.5</td>
<td>1.5</td>
<td>1</td>
<td>1.5</td>
<td>1.6</td>
</tr>
<tr>
<td>Task 3.1</td>
<td>Cloud related maintenance. Increase interlinkage</td>
<td>7.5</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>5.5</td>
<td>7.5</td>
<td>5</td>
<td>7</td>
<td>5.5</td>
<td>5</td>
<td>6.1</td>
</tr>
</tbody>
</table>

class of schema:MediaObject which is a subclass of schema:CreativeWork, and so on. In the sample wrapper, resources are typed as schema:MediaObject. Subjects had to type resources as pertaining to schema:VideoObject subclass.

- Linked Data Cloud evolution. New nodes might enrich existing LDWs with additional interlinkage.
  - Task 3.1: A new interlinkage to a Linked Data Cloud node. Let’s suppose a consumer is interested in knowing where the videos were taken. The subjects must create a new property (e.g., rdfs:seeAlso) which points to the place by means of the coordinates the picture metadata holds (e.g., http://linkedgeodata.org/api/3/intersects/36.507014,-4.881641/1). The Linkedgeodata service provides information about points of interest in a circular area [11]. This task is similar to Task 2.1, but now subjects should locate and use two attributes (i.e., latitude and longitude) and compose a URI.

Next subsection describes the experiment.

12.1. The experiment

Measures. The evaluation is focused on two of the quality-in-use model characteristics proposed by the ISO/IEC 25010 standard [8].

- Efficiency, which relates to the resources spent in relation to the accuracy and completeness with which users achieve goals. A main indicator of efficiency is task completion time.
- Satisfaction, which relates to the degree to which a user is satisfied with their perceived achieve-ment of pragmatic goals, including the results of use and the consequences of use. It was assessed through specifically designed questionnaires measured by attitude rating scales such as SUMI [49].

Setting. In order to eliminate differences in the perception of the sample LDW due to hardware or bandwidth differences, the study was conducted in a laboratory of the Computer Science Faculty of San Sebastián. All participants used computers with the same features (i.e., Intel Core 2 1.86 GHz, 3 GB RAM and Windows XP Professional SP3) and a clean installation of Firefox 45.0.1.

Subjects. The experiment was conducted among 10 graduate students applying in a Master in Web Engineering. The majority of participants were male (80%). Regarding age, all participants were in the 22-25 age range. This experiment was realized at the end of 10 hours course in Web Programming issues, where students were familiarized with the YQL Console, the YQL language and ODT specifications. As part of the Master degree, students followed a 30 hour Semantic Web course, where Linked Data concepts and RDF syntax were introduced. All of them were acquainted with XML and JSON, but not with JSON-LD. Six students were expert JavaScript programmers and four had basic skills.

Instrument. A questionnaire served to gather users’ experience. It consisted of two parts, one to gather the participants’ background and one to evaluate efficiency and satisfaction. In order to measure efficiency, participants had to annotate the start time and the finishing time of each task. Satisfaction was measured using 7 questions with a 5-point Likert scale (1=completely disagree, 5=completely agree). Additionally, an open question gathered the subjects’ feelings using the tool.

Procedure. Before starting, a 45 minute talk was given, introducing the purpose. A user-guide sheet was distributed among participants with all this information. Next, subjects were faced with the aforementioned tasks.

---

6The efficacy, safety and context coverage has not been evaluated in this experiment.
Table 6
Satisfaction assessment: from 1 ("total disagreement") to 5 ("total agreement")

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>I easily pinpoint to the property I want to annotate</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3.6</td>
</tr>
<tr>
<td>I easily realize whether properties were annotated or not</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3.2</td>
</tr>
<tr>
<td>Defining instances types was easy</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3.8</td>
</tr>
<tr>
<td>Defining property mapping was easy</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4.2</td>
</tr>
<tr>
<td>Defining association mapping was easy</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3.8</td>
</tr>
<tr>
<td>Pre-views help fixing mapping errors</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3.8</td>
</tr>
<tr>
<td>The Semantic View tab is useful</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4.2</td>
</tr>
</tbody>
</table>

Efficiency results. Table 5 shows the average time performing each task. The experiment was arranged along the three sources of LDW fragility, namely:

- API evolution. Task 1.1 requires more than 3 minutes in average. It implies editing the LDW code on the YQL Editor, and change a parameter value (i.e. the API key). This is not so demanding per se but it is the first contact with the LDW code. So, subjects had to explore and understand the wrapper definition (i.e. the XML syntactic sugar and the JavaScript source code). Next, Task 1.2. It took 5.8 minutes on average. Main challenge was to explore the API response on the search for the missing property (as a result of API evolution) within the XML structure. On the other hand, subjects spent time interacting with the annotation tool.

- Ontology upgrades. Tasks 2.1 and 2.2 involve interacting with the annotation tool to swap properties (i.e. from dc:subject for dcterms:subject) and class membership (i.e. from schema:MediaObject to schema:VideoObject), respectively. Subjects spent 2.9’ for Task 2.1. and 1.6’ for Task 2.2. The reduction in time w.r.t. Task 1.2 (which conceptually is not so different) can be presumably due to the increasing familiarization with the tool.

- Linked Data Cloud expansion. Task 3.1 involved composing a URI out of the object coordinates. This requires moving to the YQL Editor, and exercise JavaScript skills. This might well explain time dispersion (from 4 to 8 minutes) where skillful subjects outperform their colleagues less knowledgeable about JavaScript.

Satisfaction results. An evaluation questionnaire was prepared to ascertain the satisfaction of subjects in using the annotation facility. This facility is realized through the “Annotation View” and the “Semantic View” tabs in the YQL Console (see Figure 12). Table 6 displays the results using a Likert scale from 1 (“total disagreement”) to 5 (“total agreement”). The weakest results are obtained for property searching (3.6 avg. points) or to know what is being annotated (3.2 avg. points). This may be due to scalability matters when scrolling large XML documents in search for a given element. Color conventions (dark blue for unannotated, light blue for annotated) might also be too faint to easily spot what properties have not yet being annotated. By contrast, pop-up windows for setting either resources’ type, property mapping and association mapping are found intuitive enough with 3.8, 4.2 and 3.8 points, respectively. Showing the semantic counterpart for the annotation at hand (i.e. pre-views) was also of interest (3.8 avg. points). In general, the Semantic View tab was highly regarded (4.2 avg. points).

Summing it up, this first evaluation suggests that students were able to upgrade someone else’s LDW using the annotation approach. This not only reduce the wrapping effort (as compared with directly using JavaScript) but also increases the user base able to contribute (no need to know JavaScript). This first evaluation should be complemented with a more “naturalistic evaluation” that explores the effectiveness of the approach in its real environment. At this respect, we believe the ultimate spur to back LDW platforms should not be found in altruistically maintaining broken LDWs but on promoting reusing practices that make existing LDWs valuable for developments other than the original ones.

13. Related work

This section frames SYQL into other attempts to facilitate LDW development: TWC LOGD [34], xCurator [71], LOD Laundromat [23], D2RQ [11], Virtuoso Sponger [36], Bio2RDF [28], DBpedia [53], SA-REST [65], Karma [69], SWEET [56] and LIDS/LOS services [63,68]. These platforms are compared along four di-
Data Sources. There are several initiatives to wrap heterogeneous data sources to the Linked Data. **D2RQ** platform wraps relational databases, **DBpedia** wraps Wikipedia HTML pages, and **SA-REST** focuses on Web Services. But it is the wrapping of REST API’s where more initiatives showed up in accordance with the popularity of this approach (75% of Programmable Web reported APIs are available through REST). More encompassing approaches such as **Virtuoso Sponger** or **Bio2RDF**, offer a middleware for a variety of data sources (relational, Web Service or REST). Unlike other approaches, data is not obtained on the fly but periodically uploaded. **SYQL** focuses on wrappers upon REST services, and in general, any source that produces XML (e.g. XHTML).

Wrapper Language. Approaches attempt to find a compromise between expressiveness and learnability. More declarative wrapper specifications easy user involvement focusing on specific data sources (e.g. Wikipedia or relational databases). In the case of **DBpedia**, this specification is realized in terms of wiki templates, akin to the wiki origins of this initiative. In **D2RQ** wrapping is specified by R2RML ontology, where “TripleMaps” objects map relational databases’ tables and columns into RDF classes and properties, respectively. Departing from declarative specifications, other authors resort to general-purpose procedural languages (e.g. **SYQL**, **Bio2RDF** and **Virtuoso Sponger**), wrapper ontologies (e.g. **Karma**, **TWC LOGD**), or a mixture (e.g. **SA-REST**, **SWEET** and **LIDS/LOS**), depending on the target audience (i.e. API programmers for **SYQL** vs. the Semantic Web community for **Karma**).

Creation time. This dimension refers to the time the target RDF data is created from the data source. Broadly, this dimension is related with the obsolescence of the data source. For volatile data (e.g. REST data sources), RDF resources are created when they are requested on the fly. For more stable data (i.e. CSV, semistructured or RDF Datasets files), wrapping might happen at loading time. Finally, some platforms such as **Bio2RDF**, **Virtuoso Sponger** or **DBpedia**, allow for RDF data to be loaded periodically in search of a higher throughput.

Tooling. Promoting collaborative LDW development involves appropriate tools. This includes the existence of publicly available LDW repositories that permit clone&own, code generators, assistive editing, testing and debugging capabilities as well as cloud deployment. **RBA** [60] is a tool for semi-automatically generating customized R2RML mappings from a database. **Virtuoso Sponger**, **Bio2RDF** and **SYQL** enjoy a LDW repository which is realized through a GitHub repository. In contrast, **DBpedia** shares wrappers as wiki pages. **SWEET** offers an ontology assisted annotation recommender based on Wik-
### Table 8
LDW Platform’s metarequirements compliance

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TWC LOGD</td>
<td>Yes (automatic)</td>
<td>Yes</td>
<td>Yes (SPARQL)</td>
<td>Yes (VoID, metadata)</td>
<td>No</td>
</tr>
<tr>
<td>xCurator</td>
<td>Yes (semiautomatic)</td>
<td>Yes</td>
<td>Yes (SPARQL and browsing)</td>
<td>Yes (provenance)</td>
<td>Yes (duplicates)</td>
</tr>
<tr>
<td>LOD Laundromat</td>
<td>No (build-in)</td>
<td>No</td>
<td>Yes (SPARQL and triple query)</td>
<td>Yes (metadata)</td>
<td>Yes (syntax and duplicates)</td>
</tr>
<tr>
<td>D2RQ</td>
<td>Yes (automatic)</td>
<td>Yes</td>
<td>Yes (SPARQL)</td>
<td>Yes (VoID, provenance)</td>
<td>No</td>
</tr>
<tr>
<td>Virtuoso Sponger</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (SPARQL, URI deref)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Bio2RDF</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (SPARQL)</td>
<td>Yes (VoID, metadata)</td>
<td>No</td>
</tr>
<tr>
<td>DBpedia</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (SPARQL, URI deref)</td>
<td>Yes (VoID)</td>
<td>Yes (syntax)</td>
</tr>
<tr>
<td>SA-REST</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (API invocation with RDF)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>KARMA</td>
<td>Yes (semiautomatic)</td>
<td>Yes</td>
<td>Yes (API invocation with RDF)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SWEET</td>
<td>Yes (semiautomatic)</td>
<td>Yes</td>
<td>No</td>
<td>Yes (metadata)</td>
<td>No</td>
</tr>
<tr>
<td>LIDS/LOS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (URI deref)</td>
<td>Yes (VoID, provenance, Hydra)</td>
<td>Yes (syntax and semantics)</td>
</tr>
<tr>
<td>SYQL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (URI deref)</td>
<td>Yes (VoID, provenance, Hydra)</td>
<td>Yes (syntax and semantics)</td>
</tr>
</tbody>
</table>

son [31]. Karma illustrates the most ground-breaking stance. Through a kind of “programming-by-example” approach, Karma permits end users to generate LDW automatically out of a set of examples of API calls.

### 14. Discussion

Platforms can serve different aims, and hence, being driven by different requirements. Platform comparison can then be unfair if the requirements of the comparison are not those that drive the platform design. Nevertheless, this comparison is needed to show out the additional contributions, and what is also important, the extent to which existing platform can embrace the new requirements. To this end, this work identifies some key requirements for facilitating LDW sharing and collaborative editing (see Table[1]). This section addresses the extent to which the aforementioned platforms fulfill these requirements.

Table[8] holds the output where each requirement admits two values (i.e. “yes” or “no”) according to these criteria:

- **LDW definition.** Yes: users can define their own wrappers. No: there is no way to define wrappers;
- **LDW registration.** Yes: the system allows to register user wrappers. No: registering user wrappers is not allowed;
- **Resource lookup.** Yes: individual resources are dispatched[7] No: resources are not accessible through their URI;
- **LDW lookup.** Yes: LDWs are RDF resources[8] No: LDWs are not semantically described;
- **Quality checking.** Yes: an implicit or explicit LD quality assessment is used[9] No: none quality assessment is performed.

**TWC LOGD, D2RQ and Bio2RDF** cover all the requirements except the quality checking. **LOD Laundromat** is a fully automatized RDF to RDF datasets cleaner. Hence, users cannot define or maintain wrappers. **Virtuoso Sponger and Karma** allow to create wrappers but do not focus on quality and maintenance. **SA-REST** consumes wrappers and data into a proxy server so they are not publicly provided nor validated. **SWEET** and **LIDS/LOS** focus on APIs semantic de-

---

[7] For instance, dereferencing URIs or by means of a SPARQL endpoint.

[8] LDW descriptions include VoID (i.e. dataset data), Hydra (APIs documentation), provenance or other type of metadata.

[9] Quality checking could be realized by removing duplicate entities, detecting syntax errors, or reporting semantic inconsistencies.
scription. No contribution on quality or wrapper maintenance. Only two platforms cover all requirements: DBpedia and xCurator.

The platforms more similar to SYQL include Bio2RDF, xCurator, DBpedia and TWC LOGD. Bio2RDF shares the vision of an open community of wrapper developers. Bio2RDF allows developers to program in their preferred programming language which lowers technological barriers but complicates reusability. By contrast, SYQL aims to promote both collaborative maintenance and the engagement of the API community. Its declarativeness and popularity among API programmers make YQL’s ODTs our bet.

xCurator offers a semiautomatic LDWs development while the maintenance process gathers consumers’ feedback. The main difference with SYQL lies in openness. SYQL is totally open to Web users: everybody can create and modify wrappers. In the xCurator platform data consumers can report data problems while system administrators are in charge of modifying wrappers.

DBpedia wrappers are syntactically validated whereas data is assessed by selected consumers detecting and reporting errors \(^{[21,50]}\). The main difference stems from DBpedia being Wikipedia specific while SYQL is agnostic.

TWC LOGD also faces backward compatibility but with a different approach. Upgrades are incrementally created adding new properties. That is, if there are \(n\) different upgrades, there will be \(n\) different wrappers. In this way, each consumer can pick up his favorite version. By contrast, SYQL only keeps a single wrapper version, though developers can resort to version control to create new LDWs out of previous versions. Backward compatibility forces no properties removal. This could lead to redundant values, and hence, lower data quality.

Though similarities exist, SYQL departs from Bio2RDF, xCurator, DBpedia and TWC LOGD in its focus: APIs. These platforms follow a Crawling Pattern \(^{[2]}\) whereby data is loaded in advance. This brings a main advantage: SPARQL end-points. By contrast, SYQL explores the On-The-Fly Dereferencing Pattern where URIs are dereferenced at the moment the data is required. The increasing importance of APIs so sustains. Nevertheless, an open issue is how to extend SYQL to also account for SPARQL capabilities.

15. Conclusions

APIs are a main asset of the Web, but not of the Web of Data. LDWs permit to close the chasm by offering APIs as part of the Linked Data Cloud. Unluckily, most attempts so far do not sustain over time, and they are seldom maintained. This work promotes a vision where LDWs become a valuable resource of the Web of Data. This sustains the existence of dedicated platforms for LDW management. A dedicated platform can bring the following benefits: built-in infrastructure (i.e. developers do not need to care about deployment), higher-level programming (i.e. offering facilities for a more declarative LDW definition), and collaborative development (i.e. multiple developers can contribute to keep LDWs up and running). This work introduces some of the facilities these platforms can offer in terms of LDW development, maintenance, reuse and deployment. This vision is fleshed out through SYQL, a LDW Platform built on top of Yahoo’s YQL.

We hypothesize SYQL can substantially enlarge LDW lifespan. This has yet to be proven. However, the benefits are worthwhile: keeping in sync all, APIs, LDWs and the Linked Data Cloud. First evaluations suggest that using API output documents as a basis for annotating the different wrapping concerns, seems intuitive enough for most subjects. This not only reduce the wrapping effort (as compared with directly using JavaScript) but also increases the user base able to contribute (no need to know JavaScript). Nevertheless, the ultimate spur should not be found in altruistically maintaining broken LDWs but on promoting reusing practices that make existing LDWs valuable for developments other than the original ones. Future work includes extend SYQL’s On-The-Fly Dereferencing Pattern to account also for SPARQL query capabilities, and to support read-write LDWs.

Acknowledgement. This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2014-58131-R.

References


