Linked Data Wrapper Curation: A Platform Perspective

Iker Azpeitia *, Jon Iturrioz and Oscar Díaz
Onekin Research Group, Computer Science Faculty, University of the Basque Country UPV/EHU, Pº Manuel Lardizábal, 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. Linked Data Wrappers (LDWs) turn Web APIs into RDF end-points, leveraging the LOD cloud with current data. This potential is frequently undervalued, regarding LDWs as mere by-products of larger endeavors, e.g. developing mashup applications. However, LDWs are mainly data-driven, not contaminated by application semantics, hence with an important potential for reuse. If LDWs could be decoupled from their breakout projects, this would increase the chances of LDWs becoming truly RDF end-points. But this vision is still under threat by LDW fragility upon API upgrades, and the risk of unmaintained LDWs. LDW curation might help. Similar to dataset curation, LDW curation aims to clean up datasets but, in this case, the dataset is implicitly described by the LDW definition, and “stains” are not limited to those related with the dataset quality but also include those related to the underlying API. This requires the existence of LDW platforms that leverage existing code repositories with additional functionalities that cater for LDW definition, deployment and curation. 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 facilitates LDW curation.

Keywords: Linked Data Wrappers, Curation, YQL, Web APIs

1. Introduction

Web 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 [51]. This explains the exponential growth in API figures [79]. Unfortunately, less than 0.5% of APIs export their data using an RDF data format [40], being JSON-LD the preferable RDF format [26]. 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, producers can wrap these services to expose Linked Data. Indeed, in the 2017’s LOD cloud diagram [14] 36 datasets qualified as wrappers[1]. We focus on this kind of wrappers, hereafter referred to as Linked Data Wrappers (LDWs) [31,32].

Commonly, LDWs are regarded as by-products of larger endeavors, e.g. developing a mashup applica-

*Corresponding author. E-mail: iker.azpeitia@ehu.eus.
tion. Each application develops its own LDW, and its usage tends to be limited to this application. Hence, the LDW lifecycle is that of the containing application. Indeed, it is not rare the case 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 [31], GoogleArt project to RDF [8], OAPI2LOD IATI parser [15], GeoNames wrapper [75] or Twitter wrapper [17].

The problem is then not so much about LDW development but about unmaintained LDWs. This is unfortunate since it undermines the role of LDWs as a sustainable foundation for both the Web of Data and Semantic Applications. Causes may be many-fold: lack of interest, lack of recognition, lack of usage, lack of resources, etc. In this work, we address three main causes:

1. LDWs’ lifecycles are coupled to those of the breakout projects. Once projects are over, so is the maintenance of the attached LDWs,
2. LDW maintenance penalty is high. This is mainly due to LDW fragility upon API upgrades,
3. the shortage of people involved. Traditionally, this is the case of research groups which might lack the resources for keeping LDWs up and running.

To lessen these causes, we resort to LDW curation. Curation is not new to the LOD world. Evidences on LOD’s mistakes and incompatibilities [49] gave rise to the interest in data curation. More to the point, the fact that a dataset’s own quality might impact the quality of other datasets that link to it, is being argued as “an incentive to clean stains in LOD that goes beyond that of the original dataset creators” [28]. If this is so for explicit datasets, similar concerns can be risen from implicit datasets, i.e. LDWs. Different projects (e.g. Virtuoso Sponger [42] and Bio2RDF [33]) resort to GitHub repositories for developers to clone LDWs; next, curate them in a different GitHub branch, and finally, send a pull request to modify the master distribution.

LDWs are code and hence, they can resort to general facilities for code artifacts, e.g. code repositories like GitHub. But, can we do better? After all, LDWs realize the definition of implicit datasets whose “stains” are not limited to those preventing the code from functioning but also those related with the quality of the dataset being obtained. Beyond general-purpose code repositories like GitHub, LDW-specific platforms could well cater for the specifics of LDWs. This includes LDW deployment but also supporting specific functionalities for LDW curation. Such platforms can act as repositories where LDWs can outlive their original applications, and most importantly, where third parties can tap into. Re-use increases the number of actors interested in keeping LDWs in shape, inspiring others to share LDW maintenance burden (i.e. the curators). 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 facilitates LDW curation.

The rest of the paper is structured as follows. Section 2 introduces the practice, i.e. Linked-Data Wrapping. Section 3 provides some motivating scenarios. Section 4 draws main requirements for LDW Platforms. Sections 5 to 8 flesh out this approach by introducing the SYQL platform. This platform is evaluated w.r.t. LDW curation in Section 9. Finally, SYQL is confronted with other platforms with related aims. Conclusions end the paper.

2. The practice: Linked-Data Wrapping

Broadly, LDWs are mainly used in two scenarios: Web of Data and Semantic Applications. This section outlines the importance of LDWs in these two scenarios. Next, we introduce LDW definition as either coupled artifacts or separated artifacts.

Web of Data. LDWs are being used to extent the LOD cloud with current data. We conducted a search upon https://datahub.io for the keyword “wrapper” in March, 2017: 36 datasets qualified as wrappers. But LDW usefulness does not stop at introducing current data but also help to add “interlinkage layers” on top of existing LOD nodes. The need for interlinkage layers is evidenced by a 2014 study that concludes that only 56% of the 1014 LOD datasets studied have external links [70]. More to the point, the steady introduction of new LOD nodes requires this interlinkage to be a continuous effort. In-
Fig. 1. General architecture of Linked Data Applications (taken from Euclid [5]).

deed, a 2016 survey about the quality of links between LD datasets, concludes that 7.9% of the links were actually dead [65]. This sustains the need for continuously revising LOD interlinkage. LDWs can help by repairing/enhancing existing datasets with the broken/missing links.

**Semantic Applications.** They are grounded on the existence of quality datasets. Web APIs are a most important source of current data. Unfortunately, API providers (i.e. eBay, Amazon) lack a clear demand for RDF, while consumers stick to JSON/XML due to the learning curve and lifting effort to move to RDF. This chicken-and-egg “cold-start” problem could be mitigated if LDWs were in place. If API providers do not yet have a business case for leveraging their APIs to RDF, LDWs can temporarily take their place, providing the basis for semantic applications to thrive. Once semantic applications are available, this would make the case for API providers to take over, and natively provide RDF, making (some) LDWs redundant.

2.1. LDWs embedded in 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 [5]. First, the **Crawling Pattern** where data is loaded in advance [45]. Second, the **Federated Query Pattern** in which complex queries are submitted to a fixed set of data sources [71]. 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 [7], 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 web APIs.

As an example, consider the **Flickr API**. This API facilitates programmatic access to pictures and videos [7]. 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 [66,76,78]:

- lowering: i.e. mapping the URI (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}`).
- API key handling: an API key is a code passed in by programs calling an API to identify the calling program, its producer, 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

3API producers enforce a request rate limit to prevent abuse of the service. If you exceed these thresholds, the API may stop working for you temporarily. Rates might be set on different basis: consumer-based (e.g. Twitter sets a maximum of 450 calls per 15’), resource-based (e.g. Facebook sets a maximum of 4800 calls per page and day for each active user) or operation-based (Youtube sets 1M calls per day for reads and 2000 for uploads).
for 3600 calls an hour. LDWs call APIs. Hence, they might require to first get an API key.

- **lifting**: creation of the Linked Data Resource from the API result (see the “property-mapping” arrow in Figure 2). 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 video’s location (through the schema:locationCreated property) and the video’s topics on DBpedia (through the property schema:about).

- **metadata** (see the metadata box in Figure 2): this includes a link to the wrapper description (line 12), and provenance data (line 13-16).

The resulting code ends up being embedded in the Data layer of applications. Once applications are over, the interest in keeping up LDWs frequently vanishes.

### 2.2. LDWs as separated artifacts

LDWs have a value on their own right. The fact that they are mainly data-driven, increases their potential of reuse in different scenarios. Therefore, a broad literature exists on defining LDWs on their own: **TWC LOGD** [39], **xCurator** [80], **LOD Laundromat** [28], **D2RQ** [1], **Virtuoso Sponger**, **Bio2RDF** [60], **DBpedia** [60], **SA-REST** [72], **Karma** [78], **SWEET** [63] and **LIDS/LOS** services [66,76]. This subsection compares these platforms along five dimensions: the data source being wrapped, the wrapper language, the creation time, tool availability, and finally, data curation support. First four dimensions are collected in Table 1 while curation support is displayed in Table 2.

**Data Sources.** There are several initiatives to wrap heterogeneous data sources to Linked Data. **D2RQ** wraps relational databases (RDB), **DBpedia** converts Wikipedia HTML pages, and **SA-REST** focuses on Web Services. But it is the wrapping of REST API’s where more initiatives showed up. More encompassing approaches such as **Virtuoso Sponger** or **Bio2RDF**, offer a middleware for a variety of data sources (relational database, Web Service or REST).

**Wrapper Language.** **DBpedia** resorts to wiki templates, akin to the wiki origins of this initiative. In **D2RQ**, wrapping is specified through the 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. **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. 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 Dataset files), wrapping might happen at loading time. Finally, some platforms such as **Bio2RDF**, **Virtuoso Sponger** or **DBpedia**, allow

---

**Fig. 2.** From Flickr API output (left) to URI-addressable resources (right). Besides the data itself, RDF resources include dataset and provenance metadata.
Table 1
LDW approaches

<table>
<thead>
<tr>
<th>Data source</th>
<th>Wrapper language</th>
<th>Creation time</th>
<th>Tooling</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOD Laundromat</td>
<td>RDF datasets</td>
<td>Hidden n/a</td>
<td>Load time</td>
</tr>
<tr>
<td>TWC LOGD</td>
<td>CSV</td>
<td>RDF</td>
<td>Load time</td>
</tr>
<tr>
<td>xCurator</td>
<td>Semistructured</td>
<td>Hidden n/a</td>
<td>Load time</td>
</tr>
<tr>
<td>D2RQ</td>
<td>RDB</td>
<td>R2RML ontology</td>
<td>On the fly</td>
</tr>
<tr>
<td>Virtuoso Sponger</td>
<td>RDB and REST</td>
<td>Procedural (C++, Java, ...)</td>
<td>Periodically</td>
</tr>
<tr>
<td>Bio2RDF</td>
<td>Semistructured, RDB and REST</td>
<td>Procedural (PHP, Java or Ruby)</td>
<td>Periodically</td>
</tr>
<tr>
<td>DBpedia</td>
<td>Wikipedia articles</td>
<td>WikiText template</td>
<td>Periodically</td>
</tr>
<tr>
<td>SA-REST</td>
<td>Web services</td>
<td>RDFa upon SAWSDL ontology</td>
<td>On the fly</td>
</tr>
<tr>
<td>Karma</td>
<td>REST</td>
<td>Karma ontology</td>
<td>On the fly</td>
</tr>
<tr>
<td>SWEET</td>
<td>REST</td>
<td>hREST upon MicroWSMO ontology</td>
<td>On the fly</td>
</tr>
<tr>
<td>LIDS/LOS</td>
<td>REST</td>
<td>Ontology and procedural</td>
<td>On the fly</td>
</tr>
</tbody>
</table>

Table 2
Spotting and Cleaning activities during data curation

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Spotting</th>
<th>Cleaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOD Laundromat</td>
<td>Datasets</td>
<td>Automatic</td>
</tr>
<tr>
<td>TWC LOGD</td>
<td>Datasets</td>
<td>Consumers</td>
</tr>
<tr>
<td>xCurator</td>
<td>LDW</td>
<td>Consumers</td>
</tr>
<tr>
<td>Virtuoso Sponger</td>
<td>LDW</td>
<td>Developers</td>
</tr>
<tr>
<td>Bio2RDF</td>
<td>LDW</td>
<td>Developers</td>
</tr>
</tbody>
</table>

for RDF data to be loaded periodically in search of a higher throughput.

Tooling. Promoting collaborative LDW development involves dedicated 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 [53] is a tool for semi-automatically generating customized R2RML mappings from databases. Virtuoso Sponger and Bio2RDF resort to GitHub as the LDW repository. In contrast, DBpedia shares wrappers as wiki pages. SWEET offers an ontology-assisted annotation recommender based on Watson [36]. Karma resorts to “programming-by-example” (PbE) where users generate LDWs out of a set of examples of API calls.

Curation. Table 2 outlines main projects in the LOD area where curation is being addressed. For our purposes, the main insights come from who conducts the curation, specifically, who conducts two of its main tasks: detection (i.e. spotting the stain) and intervention (i.e. cleaning the stain). Ideally, both tasks should be automated. Unfortunately, curation is not fully automated for most data types, requiring user intervention. Here, the user can be limited to the platform administrator or extended to any consumer. The amplitude of the curator spectrum very much depends on the complexity of the dataset or the LDW at hand, but also on the richness of the stains to be spotted. For instance, LOD Laundromat is a curation service for RDF datasets. Being an automatic curation service, it detects and repairs only a fixed number of issues. Alternatively, xCurator allows for consumers to spot stains that are next handled by system administrators. In the same vein, TWC LOGD allows for consumers to generate personal versions of datasets as needed. If we move to the LDW realm, both Virtuoso Sponger and Bio2RDF resort to GitHub repositories for developers to clone LDWs; next, curate them in a different GitHub branch, and finally, send a pull request to modify the master distribution.

LDWs are code and hence, they can resort to general code repositories like GitHub. This work advocates for dedicated platforms that leverage existing repositories to account for LDW definition, deployment and curation.

3. The case for LDW Platforms

The vision is for LDW concerns to be externalized into a separated platform (see Figure 3). Implications are many-fold:

– 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, LDWs are specified outside the application boundaries but in the LDW Platform.
Fig. 3. Externalizing LDW concerns into a dedicated platform.

- at maintenance time, LDWs are curated at the LDW platform. Facilities should be provided for visualizing the current functioning status of LDWs, and to spot (and amend) eventual malfunctions. Curators might or might not coincide with the original LDW developers, so code understandability becomes a critical feature.

- 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, RDF visualizers and semantic mashups. Specifically, we use WordPress [21], LODmilla [16] and LinkedWidgets [13] as representatives of Content Management Systems (CMSs), RDF graph visualizers and mashup platforms, respectively. For each platform, we developed an application where LDW needs are externalized. In this way, other developers can tap into existing LDWs. We start by CMSs.

3.1. 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 the schema.org ontology in order to mark up structured data in the Web [46]. Search Engines consume so annotated content and show it in a flashy way. The term “Rich Snippet” was coined by Google to refer to those schema.org formatted samples of a site’s content. Once Web content is marked up along the snippet directives, search engines can offer a more detailed account of Web sites, making it more enticing for users to click on, and easier for Search Engines to extract information [73]. 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 metadata 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 JSON-LD. Rather than embedding this wrapping functionality as part of the plug-in, we advocate for this functionality to be detached into an 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 (line 2); the LDW URI is constructed (e.g. ‘http://rdf.onekin.org/flickr/videoobject’ + video ID) (line 3); finally, the URI is dereferenced (lines 4-6), and the Rich Snippet is obtained (line 8). That WordPress plug-in is available at https://github.com/onekin/ldw/blob/master/Flickr.WPplugin.php.

3.2. RDF graph visualizers: LODmilla

Linked Data exploration is being supported through different visualization tools [44][55]. 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.
Fig. 4. WordPress editor. (a) Post editor: Post including annotation mark-up for obtaining the Rich Snippet (e.g. \[FlickrVideoObject id=videoID\]). (b) Plug-in editor: code requests the LDW service flickr.videoobject.ldw available at https://github.com/onekin/ldw/blob/master/flickr/flickr.videoobject.xml

Fig. 5. LODmilla exploration graph. Merging LOD-sourced data (e.g. DBpedia) and API-sourced data (e.g. Flickr, GeoPlanet [23] & Wunderground [20]) in the same graph through LDWs.

The question is to extend 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 videos. This is straightforward with the previously developed LDW. In addition, we can interlinkage flickr.videoobject.ldw resources to other resources either LOD-based (e.g. DBpedia resources) or API-obtained (e.g. GeoPlanet resources). Figure 5 depicts how this exploration looks like at LODmilla.

The exploration starts at a given Flickr person (35092116@N00 node in Figure 5). Videos are next explored (through the flickr.videoobject.ldw LDW). Using LODmilla facilities, users decide which video properties to show up: schema:interactionCount (i.e. the number of interactions for the video), schema:about (i.e. the subject matter of the video), etc. This permits to keep exploring on the basis of these resources. Specifically, schema:locationCreated resources hold places information and from places the weather reports are retrieved. This example requires an LDW for turning GeoPlanet API into a data set. Notice that the flickr.videoobject.ldw LDW needs to be upgraded adding two links schema:about and schema:locationCreated to point to DBpedia and GeoPlanet 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.

3.3. Semantic mashups: LinkedWidgets

Semantic mashups are mashup applications using RDF as its background data model, and SPARQL for tasks execution [54]. These applications offer new functionality by combining, aggregating, and trans-
performing 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 [44,62]. LinkedWidgets is a case in point.

In LinkedWidgets, mashups are modeled as widgets that are orchestrated using a pipe-like approach. Figure 6 depicts a LinkedWidget mashup that involves arranging somebody’s Flickr video 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.

Though 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 videos (flickr:person.ldw) and the video 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 help. Specifically, the LDW flickr:videoobject.ldw could have well be made available as a result of the previous use cases. If so, LinkedWidgets developers can tap into this LDW when creating their widgets.

4. Requirements for LDW Platforms

LDW platforms aim at becoming single-stop solutions for LDW management. Specifically, LDW platforms should account for three main stakeholders: producers (i.e. those who develop LDWs from scratch), consumers (i.e. those who re-use someone else’s LDWs) and curators (i.e. those who perform some kind of LDW upgrading). The rest of this section identifies requirements for each stakeholder. Requirements are motivated by the existing literature, outlining different ways in which the requirement is being addressed so far. The intention is not to provide an exhaustive account but just to motivate the need.

4.1. Producer Requirements

4.1.1. Allow for LDW definition

Mechanisms should be provided to address the specifics of LDW development such as lowering, lifting, or interlinkage (see Section 6.1). Platforms offer a possibility of abstracting developers from the heterogeneity of API requests and its optimization, making LDW definition more declarative, and hence, more effective.

LDW definition admits different compromises between expressiveness and learnability. 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 LDW specifications that ease user involvement. In the case of DBpedia, this is realized in terms of wiki templates, akin to the wiki origins of this initiative. In relational databases, wrapping is specified through the R2RML ontology [37], 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), depending on the target audience (i.e. programmers for Bio2RDF vs. the Semantic Web community for Karma).

4.1.2. Allow for LDW deployment

Deployment starts by registering the LDW into the platform. At this point, some checks are made about LDW syntactic correctness [60] and credential availability. Credentials are codes requested by the API 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 producer, or its user.

API key provision admits two alternatives. The API key can be provided by the LDW producer at LDW specification time. Alternatively, the API key can be obtained from the LDW consumer at dereferencing time. This mimics the handling of credentials in stored procedures in Data Base Management Systems [58].

4.2. Consumer Requirements

4.2.1. Allow for LDW discovery

LDW discovery helps consumers to identify potential LDW services. The use of ontologies become paramount in so far as providing an homogeneous semantic description (most important in a sharing setting) [40]. For LDWs, LIDS and LOS are two approaches to document LDW inputs and outputs using Query Graph Patterns [60,76]. Next, SPARQL can be used to query these patterns, though the complexity of this notation makes it not the most accessible option. In a similar vein but with a more affordable notation, Karma resorts to an RDF language to describe inputs, outputs and their relationships where models can be queried using SPARQL [79].

4.2.2. Allow for LDW lookup

Once LDWs of interest are located, consumers need to go down to the nitty-gritty. Here, LDWs can be documented along their dual nature: implicit dataset definition vs. services. As for the former, LDWs can be characterized by their dataset content and dataset quality. Here, LDW producers can tap into the Vocabulary of Interlinked Datasets (VoID) [25], an RDF Schema vocabulary for expressing metadata about RDF datasets. VoID increases discoverability and facilitates metadata consumption from multiple LDWs [47,79]. In addition, De Baton et al. evidence the importance of the quality of Linked Data if an LD Application ecosystem wants to be developed [38]. Reusable resources should provide information about their quality not only to ease the process of selection but also to increase the chances of reuse. Therefore, if LDWs are going to become reusable, quality information should be provided. Accordingly, W3C’s Data Quality Vocabulary (DQV) [2] is being proposed to assess the dataset quality via a number of observed properties [3]. As implicit definition of datasets, LDWs can be qualified along DQV.

On the other hand, as services, LDWs need to be invoked and its service quality characteristics reported. Invocation wise, producers can resort to W3C’s Hydra Core Vocabulary. This lightweight vocabulary permits to create hypermedia-driven web APIs [59]. By specifying a number of concepts commonly used in web APIs, it enables a server to advertise valid state transitions following REST best practices. This approach can be extended to LDWs.

4.2.3. Allow for resource lookup

An LDW Platform is an Linked-Data Platform. As such, it should comply with the W3C standard for resource management [64,74]. Specifically, LDW platforms should support resource lookup. Compared with explicit dataset, the difference stems from resources being dynamically obtained from API data at the time they are dereferenced.

4.3. Curator Requirements

4.3.1. Allow for spotting stains

Means are needed to make the community aware of stains in LDWs. Stains are not limited to those related with the quality of the dataset being obtained but also include those preventing the code from functioning (e.g. API upgrades). As for the former, data curation is tackled in LOD Laundromat, Bio2RDF or TWC
Fig. 8. SYQL main components: SYQL Engine, YQL Engine, Health Monitor, LDW Repository and three HTML consoles.

LOGD (see Section 2.2). As for spotting code faults, inspiration can be drawn from incident management systems (e.g. JIRA [67]) and on-line Linked Data validators. For example, W3C’s RDF Validation service [19] and Vafu validation service [18] check whether Semantic Web data is correctly published according to best practices.4 Besides automatic issue detection, users can detect and notify issues to be curated [27,56].

4.3.2. Allow for cleaning up stains

Once stains are spotted, mechanisms should be in place to ease a prompt repair. Different attempts have been conducted to facilitate LDW maintenance to developers other than the authors. Declarativeness is one way forward. DBpedia introduces wikitext templates, i.e. DBpedia-specific wrappers along the lines of Wikipedia templates. Bio2RDF supports open source PHP scripts, Java programs and Ruby gems into a single GitHub repository, facilitating scripts modification by anyone wishing to improve the quality of RDF conversions. D2RQ platform provides a propriety server where LDW authors can customize automatically-generated LDWs.5 Our scenario departs from the previous ones in the data source being wrapped, i.e., APIs rather than databases or Wikipedia)

5. A platform for LDW management: architecture

This section fleshes out the LDW platform vision though SYQL (Semantic YQL). SYQL is heavily based on YQL. Besides the technical facilities, YQL allows us to tap into an existing community. At the time of this writing (April 2017), the YQL community exhibits the following figures [24]: 150 contributors, 3291 commits, 36 open and 17 closed issues, 25 open and 403 closed pull requests, 722 stars, and 462 forks. We believe LDW concerns are not so alien to API programmers. By moving to YQL, our hope is to tap into this sibling community.

Figure 8 outlines SYQL’s architecture. Care is taken for the three stakeholders: consumers (developing applications where resources are dereferenced); producers (defining and deploying LDWs from scratch) and curators (curating and tracking LDW functioning status). Next sections delve into how these stakeholders’ needs are considered in SYQL. A video about

---

4Best practices are those defined by the Linked Data principles [29], the Best Practice Recipes [80] and the Cool URIs [69].

5The generate-mapping tool creates a D2RQ mapping file by analyzing the schema of an existing database where table names and column names are used as default values. Next, administrators can customize these default mappings to curate the generated code.
the different services is available at [http://rdf.onekin.org/](http://rdf.onekin.org/). First, we provide a brief about YQL.

5.1. YQL basics

YQL is a query engine, which is hosted by Yahoo, and exposed as a REST endpoint. Requests 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.

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 (or video) whose ID is 27376196615 (see Figure 9):

```
select * from flickr.photos.info where photo_id = "27376196615" and api_key = "4fb031bf5b2f138576d01ff37f31565"
```

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 permit to handle API requests à la SQL. Behind the curtains, YQL maps these statements into the corresponding API methods. To this end, producers 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 pos-
sensible 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 producers (known as “community tables”). A full list of community tables can be found at [http://www.datatables.org/](http://www.datatables.org/).

**The YQL Console.** The YQL Console \[22\] enables to run YQL statements interactively from a browser (see Figure 9). Community tables are listed on the left. Once an ODT table is selected, YQL statements (e.g. `SELECT`) can be enacted, and the results show up at the bottom canvas. In addition, the REST-call counterpart of the query is also provided, ready to be embedded in the application. Next, we look at how this approach can be extended for URIs to be dereferenced in terms of YQL queries and for YQL queries to output JSON-LD.

### 6. Addressing Producer Requirements

Producers develop LDWs from scratch. At design time, they look for APIs (or even better, YQL ODTs) that meet their data needs. At implementation time, they resort to a wrapping template for addressing lifting and lowering (i.e. LDW definition). At deployment time, they register LDWs with the platform (i.e. LDW deployment).

#### 6.1. LDW definition

SYQL resorts to YQL expressiveness to define LDWs, and provides a “wrapping template” to guide developers. The template accounts for the three main steps: lowering, lifting and credential handling. As an example, we tap into the ODT in Figure 10 and turned it into an LDW (see Figure 11).

**Lowering** (i.e. mapping the URI’s (e.g. `http://rdf.onkin.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 6) and some URI examples (line 7). When the SYQL platform receives a URI (e.g. `http://rdf.onkin.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 6 to line 15 `[dc-terms:identifier]` binding). Worth noticing, the URI parameter is annotated along the `dcterms:ontology` (line 6). This will turn useful during LDW discovery.

**Lifting** (i.e. creation of the Linked Data Resource from the API result). YQL’s `function` tag is recast for lifting. Specifically, the wrapping template advises each XML tuple to be turned into an RDF resource which is serialized as JSON-LD (i.e. `oneJSONLD`, line 26). The lifting function holds `<input>` 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 21).
and `<key>` (i.e. to cast the URI for the returned RDF resource) (line 22). As for `<execute>` (lines 24-39), it holds the JavaScript code that obtains JSON-LD from the XML tuple (i.e. from `oneXML` pipe input to `oneJSONLD`). The line 25 parses the `oneXML` input to a JSON object. The lines 28 and 29 create the namespace and the type of the resource, respectively. Line 30 creates an RDF property from an `oneJSON` parameter. Interlinkage is also described here by constructing URIs out of existing parameters. Specifically, line 31 links the video to a `vivoweb` ontology class type and line 32 links to a `GeoPlanet` resource about the locality. Lines 33-36 create an embedded `schema:Person` resource that is linked to the video through the `schema:creator` association (line 37).

API key handling. API keys 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 default value in line 16). 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. Here, LDW producers might resort to API keys which are provided by consumers at lookup time (see an example at Appendix A).

6.2. LDW deployment

Once defined, LDWs need to be deployed before being stored at the GitHub repository. Deployment also
takes place through the YQL Editor. Besides setting the different registries, LDW deployment 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, two types of verifications are conducted, namely, syntactic and dereferenced-based. 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, no resource returned, or JavaScript errors.

Quality issues are detected as well but they do not prevent registration. Instead, they are shown in the Health Checker to warn about quality issues (see Section 8).

7. Addressing Consumer Requirements

Consumers build applications out of LDWs. At design time, consumers look for LDWs that meet their data and quality service needs (i.e. LDW discovery). At implementation time, consumers need help to create the environment for calling LDWs. Finally, at runtime, consumers’ applications dereference individuals obtained through LDWs (i.e. resource lookup).

7.1. LDW discovery

SYQL does not support LDW discovery. Rather, it relies on the Datahub portal for LDW discovery. Refer to [https://datahub.io/organization/linked-data-wrappers](https://datahub.io/organization/linked-data-wrappers) for details. The aim: increasing the visibility of SYQL LDWs.

Visibility, and eventually the recognition that goes by using LDWs, might turn rather important. Recognition is being reported as one of the main spurs for sharing [68]. The Semantic Web community has so understood when the Semantic Web Journal announced in 2012 the first “Special Call” for Linked Dataset descriptions as a way not only to disseminate but also to acknowledge the effort and importance of these resources [48]. In the same way that explicit datasets, LDWs might avail of these initiatives.

7.2. LDW look-up

For LDW description, SYQL resorts to the combined use of VoID, Hydra and DQV. Figure 12 sets the two main resource types: individuals and LDWs. LDWs exhibit a two-fold nature. As implicit definition of datasets, they can be characterized through VoID. As REST services, LDWs might be documented through Hydra. Figure 13 shows the VoID description for an LDW dataset along the structure depicted in Figure 12.

LDW look-up might be conducted by both humans and agents. The former, to be informed about LDW characteristics. For easy access, SYQL turns (part of) this information into an HTML page: the Health Checker (see Section 8). In addition, and similar to the role of WSDL for Web Services, interpreting and invoking LDWs might be facilitated by the use of standards for LDW description. Hydra allows data to be enriched with machine-readable affordances which enable interaction. By specifying a number of concepts commonly used in web APIs, it enables a server to advertise valid state transitions following REST best practices.

Specifically, SYQL resorts to Hydra for a main purpose: credential provision. Credentials can be provided by either producers (at deployment time) and consumers (at resource look-up time). The former scenario might lead to a capacity bottleneck if a large number of resource look-ups are based on the same API key. Alternatively, SYQL might also avail of API keys provided by consumers at lookup time. This is when Hydra comes into play. The LDW’s Hydra document holds an RDF credential description to be used at the time resources are looked up. Consumer-provided keys take precedence over producer-provided keys. Appendix A provides an example.
7.3. Resource look-up

Once deployed, 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, and the API key provided;
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.

As for the latter, Figure 2 shows an example along the structure depicted in Figure 12: void:inDataset holds a link to VoID dataset metadata (line 12); prv:usedData describes the data source (line 14); prv:usedGuideline indicates how the data has been created (e.g. pointing to the LDW URL) (lines 15 and 16).

8. Addressing Curator Requirements

Curators keep LDWs in shape. At design time, curators become aware of LDW stains. At implementation time, curators clean stains by upgrading the LDW at hand. Finally, at deployment time, LDWs are checked to be fully functional.

8.1. Allow for spotting stains

SYQL introduces the Health Checker, a daemon that periodically checks LDWs for stains, and renders the output as a Web page. Figure 15 illustrates the case for the 10 LDWs developed so far: green denotes that the LDW works and passes all the quality filters; yellow indicates that the LDW works but still holds some stains; finally, red indicates that the LDW does not work (i.e. returns an error status code).

Stains can refer to either the functioning status or data quality issues. Hereafter, Zaveri et al.’s quality framework is used.

8.1.1. Functioning-status stains

This mainly corresponds to the Accessibility dimension in the Zaveri et al.’s quality framework. 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”. Table 3 indicates how quality aspects find their way in SYQL. For instance, availability is checked out by dereferencing the LDW’s sample URI. So, sample URIs act as a sort of regression testing bucket. In addition, SYQL keeps an aggregate of how LDWs behave in the last 10 calls w.r.t. latency (Zaveri et al.’s P2 subdimension), throughput (P3) and scalability (P4). Back to Figure 15, click on the flickr:videoobject LDW for its quality measures to
Fig. 14. URI lookup sequence diagram. The “alt” deviation tackles the consumer-provided API-key scenario.

Table 3

Quality dimensions. “Abr” stands for the abbreviation used in [81].

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Subdimension</th>
<th>Abr</th>
<th>Metric</th>
<th>SYQL realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessibility</td>
<td>Availability</td>
<td>A3</td>
<td>Dereferenceability of the URI</td>
<td>Sample URIs work</td>
</tr>
<tr>
<td></td>
<td>Interlinking</td>
<td>I1</td>
<td>Detection of good quality interlinks</td>
<td>Number of broken links</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I2</td>
<td>Existence of links to external data producers</td>
<td>Number of external links</td>
</tr>
<tr>
<td></td>
<td>Performance</td>
<td>P2</td>
<td>Low latency</td>
<td>Minimum request to response delay</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P3</td>
<td>High throughput</td>
<td>Number of requests per second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P4</td>
<td>Scalability of a data source</td>
<td>Average throughput of the last ten calls</td>
</tr>
<tr>
<td>Intrinsic</td>
<td>Syntactic validity</td>
<td>SV2</td>
<td>Syntactically accurate values</td>
<td>Detection of null values</td>
</tr>
<tr>
<td></td>
<td>Semantic accuracy</td>
<td>SA2</td>
<td>No inaccurate values</td>
<td>Notifications via GitHub comments</td>
</tr>
<tr>
<td></td>
<td>Consistency</td>
<td>CS4</td>
<td>owl:DeprecatedProperty not used</td>
<td>Number of deprecated properties</td>
</tr>
<tr>
<td></td>
<td>Conciseness</td>
<td>CN1</td>
<td>High intensional conciseness</td>
<td>Number of redundant attributes</td>
</tr>
<tr>
<td></td>
<td>Completeness</td>
<td>CM2</td>
<td>Property completeness</td>
<td>Rate of XML elements lifted in the LDW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CM4</td>
<td>Interlinking completeness</td>
<td>Rate of XML elements used to interlink</td>
</tr>
<tr>
<td>Contextual</td>
<td>Trustworthiness</td>
<td>T7</td>
<td>Reputation of the dataset</td>
<td>Number of derefs and ratings in GitHub</td>
</tr>
</tbody>
</table>

show up: contains no interlinks (the I2 subdimension), 770 millisecond latency, 1.3 calls/second throughput, and 977 millisecond average elapsed time for the last ten calls (“scalability”).

In addition to Zaveri et al’s characteristics, we include two issues of concern for APIs: the expiration of the API key, and the return of no value by the API. Both scenarios are also noted in the Health Checker window (see Figure 15 under the heading “API Dimension”).

8.1.2. Data-quality stains

This mainly corresponds to the Intrinsic and Contextual dimensions in the Zaveri et al.’s quality framework. **Intrinsic.** It 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. Back to the example, Figure [15] reports three warnings. First, the SA2 subdimension: consumers report 2 issues through GitHub. Second, the CM2 subdimension: the ratio of properties per XML attributes is low. Finally, the CM4 subdimension: the
ratio of interlinks per XML attributes is low. The values are computed along the formulae proposed by Zaveri et al.

**Contextual.** This dimension tackles aspects that highly depend on the context of the task at hand. For trustworthiness, SYQL supports the reputation of the dataset (i.e. “assignment of explicit trust ratings to the dataset by humans or analyzing external links or page ranks”). This can be worked out based on human rating and rate of LDW reuse. SYQL works out these measures from LDWs’ GitHub repositories, specifically from how users rate the LDW’s code. As for LDW reuse, SYQL keeps a counter of the number of times the LDW is being used from different IPs. Back to the example, Figure 15 indicates that the sample LDW has been subject to 15 dereferentiations from 2 different IPs where the LDW has received 4 thumbs up and 1 thumbs down.

Worth mentioning, some of Zaveri et al.’s features are met “by construction”. That is, the fact that datasets are obtained out of API calls ensures the fulfillment of the following features:

- human-readable properties and metadata (U1): Figure 2 shows the automatically generated metadata. In addition, the description, author name, etc. are extracted from the LDW definition.
- exemplary URIs (U2): the URIExample is compulsory for the lowering process (see line 7 in Figure 11).
- regular expression that matches the URI of the dataset (U3): the URIPattern also is compulsory.
and allows to derive a regular expression (see line 6 in Figure 11).

- indication of the vocabularies used (U5): the vocabularies are taken from the @context property in the lifting function (see line 28 in Figure 11).

- provision of the data in different serialization formats (V1): the SYQL server performs content negotiation and dispatches the requested serialization format: JSON-LD, RDF/XML, Notation3, N-Quads, N-Triples or Turtle.

8.2. Allow for cleaning up stains

Producers start from scratch. By contrast, curators do not start afresh but depart from someone else’s code. This moves to the forefront understandability.

SYQL resorts to JavaScript for LDW implementation. This might put some curators off. To fight this back, SYQL performs reverse engineering (hereafter re-engineering), i.e. LDW code is processed for extracting knowledge about how the lifting has been con-

Reverse engineering is “the processes of extracting knowledge or design information from anything man-made and re-producing it or re-producing anything based on the extracted information” [41].
Table 4
SYQL realization of the requirements for LDW platforms

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Requirement</th>
<th>SYQL Realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer</td>
<td>Allow for LDW definition</td>
<td>Wrapping template on top of ODT tables</td>
</tr>
<tr>
<td></td>
<td>Allow for LDW deployment</td>
<td>LDWs deployed as YQL services</td>
</tr>
<tr>
<td>Consumer</td>
<td>Allow for LDW discovery</td>
<td>LDWs are publicized as “intensional datasets” at the Datahub portal</td>
</tr>
<tr>
<td></td>
<td>Allow for LDW lookup</td>
<td>Dereferenceable VoID &amp; Hydra documentation</td>
</tr>
<tr>
<td></td>
<td>Allow for resource lookup</td>
<td>URI dereferencing</td>
</tr>
<tr>
<td>Curator</td>
<td>Allow for spotting stains</td>
<td>Health Checker</td>
</tr>
<tr>
<td></td>
<td>Allow for cleaning up stains</td>
<td>Code-to-annotation re-engineering</td>
</tr>
</tbody>
</table>

9. Evaluation

This section evaluates the extent to which SYQL fulfills the requirements for LDW platforms. Table 4 outlines SYQL realization of the requirements for LDW platforms. All in all, this work’s main issue is not so much about LDW definition or quicker resource lookup, but the one of extending LDW lifecycle through curation. The challenge is not about accomplishing the change (after all, the LDW is already there) but the mechanisms available for detecting the change (i.e. the Health Checker) and conducting the change over someone else’s code (i.e. the code-to-annotation reverse engineering approach). We are not aware of other approaches that tackle similar issues. Hence, evaluation-by-comparison is not possible. Hence, we evaluate Quality-in-Use as for the curation perspective. Nevertheless, and for completeness sake, we also outline previous evaluations from the perspective of producers and consumers.[50]

**Producer Stakeholder.** We evaluated the effectiveness of LDW definition in SYQL. The task: creating an LDW on top of Flickr. The subjects: 15 MSc students knowledgeable about JavaScript and RDF. Effectiveness was evaluated in terms of “quality of solution”, i.e. a measure of the outcome of the user’s interaction with the system. The criterium for success was for the LDW successfully conducting resource lookup. The outcome: 13 out of 15 subjects were able to develop an LDW that successfully handled resource lookup. Refer to [50] for further details.

**Consumer Stakeholder.** We analyzed resource look-up throughput in SYQL. Specifically, we measured the latency introduced by LDWs w.r.t. directly invoking the API for the Flickr case. Dereferencing was conducted 1000 times with one call per second. The experiment was repeated three times at different hours of the day. Outcomes indicated that wrapping involves a three-fold overhead compared with direct API calling. Notice that this includes the cost of lifting...
and lowering. Additionally, scalability was measured. Concretely, SYQL was invoked with different dereferencing petition loads: 10, 50, 250 and 2000 threads. The result time increases logarithmically. This is due to the load balancing performed by the YQL platform. Refer to [50] for further details.

**Curator Stakeholder.** We evaluate how successful SYQL is in facilitating users the curation of third-party LDWs. From this perspective, Quality-in-Use becomes paramount. Specifically, the evaluation entails facing subjects with the curation of someone else’s LDWs: flickr.videoobject.ldw (see Figure 11). Curation scenarios include:

- 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.
  * Task 1.2: API resulting document structure changes. This causes property lifting to stop working. Challenges include recreating the mapping between the attribute and the property.

- Ontology upgrade. This might impact class mappings and property mappings.
  * Task 2.1: Switching dc:subject for dcterms:subject
  Difficulties include transforming an Property 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 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. schema:locationCreated) which points to the place where the video has been taken (e.g. http://linkedgeodata.org/api/3/intersects/Zarautz). The LinkedGeoData service provides information about places [77].

Next subsection describes the experiment.

9.1. The experiment

**Measures.** The evaluation focuses on two of the Quality-in-Use model characteristics proposed by the ISO/IEC 25010 standard [9]:

- 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 achievement 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 [55].

**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 52.0.

9 Besides API, the LOD Cloud and data ontologies might also suffer changes. Linked Open Vocabularies (LOV) database [12] 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.

10 This is a real case: 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).

11 The efficacy, safety and context coverage has not been evaluated in this experiment.
Table 5
Time spent on each task (in minutes). Times are rounded up to a half minute

<table>
<thead>
<tr>
<th>Task 1.1 API evolution. Expired credential</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
<th>S8</th>
<th>S9</th>
<th>S10</th>
<th>S11</th>
<th>S12</th>
<th>avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.5</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.8</td>
</tr>
<tr>
<td>Task 1.2 API evolution. Changed path</td>
<td>7</td>
<td>6.5</td>
<td>5.5</td>
<td>6.5</td>
<td>6</td>
<td>4.5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>6.5</td>
<td>5.5</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Task 2.1 Ontology upgrade. Property evolution</td>
<td>4</td>
<td>2.5</td>
<td>2</td>
<td>3</td>
<td>3.5</td>
<td>3.5</td>
<td>2.5</td>
<td>2</td>
<td>3</td>
<td>3.5</td>
<td>3</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>Task 2.2 Ontology upgrade. Class redefinition</td>
<td>1.5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2.5</td>
<td>3</td>
<td>1.5</td>
<td>1</td>
<td>2</td>
<td>1.5</td>
<td>2</td>
<td>1.8</td>
</tr>
<tr>
<td>Task 3.1 Cloud evolution. Increase interlinkage</td>
<td>3.5</td>
<td>4</td>
<td>3</td>
<td>2.5</td>
<td>3.5</td>
<td>3</td>
<td>3.5</td>
<td>2.5</td>
<td>3</td>
<td>3.5</td>
<td>2.5</td>
<td>4</td>
<td>3.2</td>
</tr>
</tbody>
</table>

Table 6
Satisfaction assessment: from 1 (“total disagreement”) to 5 (“total agreement”)

<table>
<thead>
<tr>
<th>I easily pinpoint to the property I want to annotate</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
<th>S8</th>
<th>S9</th>
<th>S10</th>
<th>S11</th>
<th>S12</th>
<th>avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3.5</td>
</tr>
<tr>
<td>I easily realize whether properties were annotated or not</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
</tr>
<tr>
<td>Defining instances types was easy</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3.9</td>
</tr>
<tr>
<td>Defining property mapping was easy</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4.1</td>
</tr>
<tr>
<td>Defining association mapping was easy</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3.9</td>
</tr>
<tr>
<td>Pre-views help fixing mapping errors</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>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3.7</td>
</tr>
<tr>
<td>The Semantic View tab is useful</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4.2</td>
</tr>
</tbody>
</table>

Subjects. The experiment was conducted among 12 graduate students applying in a Master in Web Engineering. The majority of participants were male (75%). Regarding age, all participants were in the 22-26 age range. This experiment was realized at the end of a 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. Seven students were expert JavaScript programmers and five 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).

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.

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 less than one minute on average. It implies changing the API key. Next, Task 1.2. It took 6 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.
- Ontology upgrade. Tasks 2.1 and 2.2 involve interacting with the annotation tool to swap properties (i.e. from dc:subject to dcterms:subject) and class membership (i.e. from schema:MediaObject to schema:VideoObject), respectively. Subjects spent 2.9’ for Task 2.1, and 1.8’ 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 now the mapping operates upon already annotated XML elements, fewer in quantity and hence, easier to spot.
- Linked Data Cloud expansion. Task 3.1 was twofold: composing a URI out of the object name, and selecting the association that links the annotated resource with a composed URI. This required moving to the YQL annotator, create a new resource from a string (e.g. Zarautz) and select the association (i.e. schema:locationCreated).

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 16). Table 6 displays the results using a Likert scale from 1
Table 7

LDW Platform’s requirement compliance

<table>
<thead>
<tr>
<th>LDW Definition/Deployment</th>
<th>LDW Discovery</th>
<th>LDW lookup (metadata type)</th>
<th>Resource lookup (mechanisms)</th>
<th>Quality checking</th>
<th>LDW curation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LOD Laundromat</strong></td>
<td>No (built-in)</td>
<td>Yes (datasets)</td>
<td>Yes (Metadata)</td>
<td>Yes (Intrinsic)</td>
<td>No</td>
</tr>
<tr>
<td><strong>TWC LOGD</strong></td>
<td>Yes (automatic)</td>
<td>No</td>
<td>Yes (VoID)</td>
<td>Yes (Intrinsic)</td>
<td>No (Personal enhancements)</td>
</tr>
<tr>
<td><strong>xCurator</strong></td>
<td>Yes (semi-automatic)</td>
<td>No</td>
<td>Yes (SPARQL &amp; URI deref)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>D2RQ</strong></td>
<td>Yes (semi-automatic)</td>
<td>No</td>
<td>Yes (SPARQL)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Virtuoso</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes (SPARQL &amp; URI deref)</td>
<td>No</td>
<td>Yes (GitHub pull requests)</td>
</tr>
<tr>
<td><strong>Sponger</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Bio2RDF</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes (SPARQL &amp; URI deref)</td>
<td>Yes (Intrinsic)</td>
<td>Yes (Wiki editions)</td>
</tr>
<tr>
<td><strong>DBpedia</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes (ad-hoc URI deref)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>SA-REST</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Karma</strong></td>
<td>Yes (semi-automatic)</td>
<td>Yes</td>
<td>Yes (ad-hoc URI deref)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>SWEET</strong></td>
<td>Yes</td>
<td>Yes (metadata)</td>
<td>Yes (ad-hoc URI deref)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>LIDS/LOS</strong></td>
<td>Yes</td>
<td>No (in Datahub)</td>
<td>Yes (VoID, DQV &amp; Hydra)</td>
<td>Yes (URI deref)</td>
<td>No</td>
</tr>
<tr>
<td><strong>SYQL</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes (URI deref)</td>
<td>Yes (Intrinsic, Accessibility &amp; Contextual)</td>
<td>Yes (Annotations)</td>
</tr>
</tbody>
</table>

("total disagreement") to 5 ("total agreement") for the 12 subjects (S1, S2, etc). The weakest results are obtained for property searching (3.5 avg. points) or the awareness of what is being annotated (3.1 avg. points). This may be due to scalability matters when scrolling large XML documents in search for a given element. Color conventions (i.e. 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 3.9, 4.1 and 3.9 points, respectively. Showing the semantic counterpart for the annotation at hand (i.e. pre-views) was also of interest (3.7 avg. points). In general, the Semantic View tab was highly regarded (4.2 avg. points).

10. Comparing SYQL with other platforms

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 platforms can embrace the new requirements. This section addresses the extent to which the aforementioned platforms fulfill these requirements.

Table 7 holds the output where each requirement admits two values (i.e. “yes” or “no”) according to these criteria:

- **LDW definition/deployment.** Yes: users can define their own wrappers. No: there is no way to define wrappers;
- **LDW discovery.** Yes: facilities are provided to query LDWs (or datasets, in the absence of). No: no query facilities;
- **LDW lookup.** Yes: LDWs are RDF resources. No: LDWs are not semantically described;

---

12LDW descriptions include VoID (i.e. dataset data), Hydra (APIs documentation) or other type of metadata.
None of the listed systems cover all the requirements. Systems aim constraints the requirements their fulfill. **LOD Laundromat** is a fully automatized RDF to RDF datasets cleaner. Hence, users cannot define or maintain wrappers. **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 description. Issues of quality or maintenance are not tackled. **Virtuoso Sponger** and **Karma** allow to create wrappers but do not focus on quality and maintenance. **D2RQ** is a server to be locally installed, therefore it ignores discoverability. **SYQL** has been mainly influenced by five developments: **Bio2RDF**, **xCurator**, **DBpedia**, **TWC LOGD** and **Karma**. Next, we provide a deeper comparison.

**Bio2RDF** shares the vision of an open community of wrapper producers. It allows producers to program in their preferred programming language which lowers technological barriers but complicates reusability. By contrast, **SYQL** aims to promote both LDW sharing and the engagement of the API community. Its declarativeness and popularity among API programmers make YQL's ODTs our bet. In addition, **SYQL** provides an re-engineering and annotation tool to engage consumers in curating wrappers.

**xCurator** offers a semiautomatic wrapper development while the maintenance process gathers consumers’ feedback. The main difference with **SYQL** lies in openness. **SYQL** is totally open: everybody can create and curate wrappers. By contrast, in **xCurator** data consumers can report data problems but only administrators can curate wrappers.

**DBpedia** wrappers are syntactically validated whereas generated data is assessed by selected consumers detecting and reporting errors [27][57]. Users can ask for edition rights in order to curate wrappers [4]. The main difference stems from **DBpedia** being Wikipedia-specific while **SYQL** is agnostic.

**TWC LOGD** also faces upgrading 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 that way, each consumer can pick up his favorite version. By contrast, **SYQL** only keeps a single wrapper version, though producers can resort to **GitHub**’s version control to create new LDWs out of previous versions.

**Karma** also addresses API-based LDWs. Both **Karma** and **SYQL** resort to annotations. However, **Karma** illustrates a generative endeavor (from annotations to code) whereas **SYQL** is a re-engineering effort (from code to annotations). This difference stems from the different targeted audiences: **Karma** targets LDW producers whereas **SYQL** aims at helping curators in cleaning someone else’s LDWs.

11. Conclusions

This work identifies a problem (i.e. unmaintained LDWs), focuses on three possible causes (i.e. LDW application coupling, LDW-maintenance penalty and people shortage), hypothesizes that these causes could be lessen by introducing LDW curation, realizes this vision through an LDW platform (i.e. **SYQL**), and finally, evaluates how successful **SYQL** is in easing third parties in curating existing LDWs. This work (1) proves the feasibility of this vision through **SYQL**, (2) introduces the notion of “the health checker” and “LDW re-engineering” as new mechanisms to support LDW curation, and (3) provides a first validation about the appropriateness of a code-to-annotation re-engineering approach to facilitate curation by third parties.

It should be mentioned that LDW curation is not free from the threats that also jeopardize other repository initiatives. As expert S. Leonelly puts it: “There is no reliable business model to finance the curation and maintenance of data repositories ... Crowdsourcing models are promising in this respect because data producers ensure that the deposited data are accurate and reusable, but these models are still not widely deployed” [61]. This work explores this approach for LDW curation. To this end, both LDWs and the **SYQL** code are available at **GitHub** [10][11].

Our next follow-on is to disseminate the existence of **SYQL**, looking for synergies with dataset repositories such as **Datathub**. We would like also to tackle discrepancy handling when different curation alternatives arise. So far, **SYQL** provides no mechanisms for

---

11For instance, dereferencing URIs or by means of a SPARQL endpoint.
discrepancy resolution. Overall, the expected benefit is bringing the currency of APIs to the LOD cloud by providing a sustainable basis for API access. The current growth in API numbers makes this source of current data far too important for the LOD cloud to overlook.

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

Appendix A SYQL at work.

This Appendix describes the steps to define, deploy and curate the running example used in this paper. A video is available at http://rdf.onekin.org/

Preliminaries

1. Open an account in Yahoo. Alternatively, you can use the sample account: User: ana.fiss@yahoo.es Password: ldw-onekin.

SYQL client-side is supported as a Firefox 52.0 plug-in on top of the YQL Console/Editor.

Producer. LDW definition:

1. Go to the YQL editor: https://developer.yahoo.com/yql/editor/
2. Log in Yahoo.
3. Click on the New button.
4. Click on the "LDW template" to have a head-start on defining the LDW.
5. Edit the lowering, the API handling and the lifting parts. Code for the running example can be found at https://github.com/onekin/ldw/blob/master/flickr.videodemo.ldw
6. Click on the Save button. At this time, you can give a name to your LDW. Next, click on the Deploy button.
7. Go to the Health console http://rdf.onekin.org/ldw/page/healthchecker/ to check out the newly created LDW.

Curator. LDW curation:

1. Go to the Health console http://rdf.onekin.org/ldw/page/healthchecker/ to check one LDW worth curating. For instance, flickr.videodemo.ldw has a broken interlink. Let’s solve this out.
2. Go to the YQL console: https://developer.yahoo.com/yql/console/
3. Log in Yahoo.
4. Select the Community LDWs radio button.
5. Click on the flickr.videodemo.ldw LDW.
6. Go to the Annotation View. Here, annotation interlays will show up after re-engineering flickr.videodemo.ldw code.
7. Go to the annotation that accounts for the broken interlink: photo/location/locality/content. Update the Association mapping’s URI to http://rdf.onekin.org/geo/place/{VALUE}. A preview of the impact on the instance resource can be obtained by moving to the Semantic View tab.
8. Once satisfied with the output, click the Generate button to obtain the code counterpart. This moves you to the YQL Editor Console
9. Click on the Redeploy button.
10. Go to the Health console http://rdf.onekin.org/ldw/page/healthchecker/ to check the interlink works.

Consumer. Resource look-up. Consumers do not need to install anything in order to dereference resources in SYQL.

1. Go to the Health console http://rdf.onekin.org/ldw/page/healthchecker/
2. Unfold the flickr.videodemo LDW.
3. Click on the URI Example for this resource to be displayed using Online JSON Viewer 14.

This example illustrates resource look-up using the producer’s API key. Alternatively, the API key can be programmatically provided by the LDW consumer. This requires the HTTP request to hold a look-up URI and an Authorization header providing the API key value (i.e. schema:value) along the Hydra credential description:

```
GET http://rdf.onekin.org/flickr/videoobject/27376196615
Authorization = "{hydra:supportedProperty": [{"hydra:property": {"schema:name": "api_key", "schema:value": "2c894ba749b4137b6f7ab127c86890ec"}}]}
```

14 http://jsonviewer.stack.hu
For security reasons, the Authorization header should be encrypted but for the sake of a better understanding, we have not encrypted it. SYQL recovers the api_key from the header and embeds it in the API call. This API key takes precedence over the one embedded in the LDW. Readers can check this out through Hurl.

References

M. Knuth, J. Hercher, H. Sack. Collaboratively patching.


