Towards the automatic deployment of data in Linked Data Platforms

Noorani Bakerally, Antoine Zimmermann

Univ Lyon, IMT Mines Saint-Étienne, CNRS, Laboratoire Hubert Curien UMR 5516, F-42023 Saint-Étienne, France
{noorani.bakerally,antoine.zimmermann}@emse.fr

Abstract. The Linked Data Platform (LDP) 1.0 is the W3C Recommendation for exposing linked data in a RESTful manner. Current solutions for LDP exist but provide no support for deploying data from external sources in LDPs. In this paper, we address this issue by providing a generic LDP resource generator which can automatize data deployment both from RDF and heterogeneous sources into LDP repositories. We demonstrate the effectiveness and flexibility of this tool using concrete cases of deploying real datasets in an LDP.

Keywords: RDF, Linked Data, Linked Data Platform

1 Introduction

The LDP standard [4] has been a step towards standardizing RESTful access to linked data. It specifies the use of HTTP to access, update, create and delete resources from servers which expose their resources as linked data. Linked data platforms complying with the LDP standard, which we refer to as LDPs, can be useful in numerous context, specially in open data where there is a need to facilitate sharing and exploitation of heterogeneous data sources by providing a homogeneous view and access to them via an LDP. However, deploying data in LDPs is still complex. LDPs are data-driven systems and deploying them involves both data and system deployment. Current LDP solutions address only the latter. There is currently no support for deploying data in LDP which we believe is the reason for the rare adoption of the LDP standard in spite of the numerous LDP solutions (cf. Sec. 2). To deploy data in LDP repositories, manual development of LDP resource generators is required to transform data resources into LDP resources and materialize them in LDP repositories. The development of LDP resource generators involves two main phases: design and implementation. During the design phase, design decisions related to LDP design are taken. The design of LDP mainly include aspects such as LDP resources IRI, their content and organization in terms of LDP containers. During the implementation phase, these decisions are encoded in the LDP resource generator. While encoding the design decisions, if they are tightly coupled with the implementation, it may be difficult both to maintain and reuse the design. To address these issues, we propose a generic LDP resource generator, which we refer to as an LDPizer, to automatize the deployment of RDF and heterogeneous data sources in LDPs using a design document as input. In the rest of this paper, we discuss related work in Sec. 2, in Sec. 3 we describe the LDPization process and finally in Sec. 4, we demonstrate the effectiveness and flexibility of the LDPizer in deploying real datasets on an LDP.

2 Related Work

The LDP standard provides a set of rules for read-write linked data via HTTP. Data resources exposed via LDPs are referred as LDP Resources (LDPR). There are two main types of LDPRs: LDP RDF Source (LDP-RS) and LDP Non-RDF Source (LDP-NR). The state of LDP-RS is fully represented in RDF while that of LDP-NR is not represented in RDF. An LDPC is a special type of LDP-RS which represents a collection of LDPRs. Current LDP solutions exist and our analysis of them is restricted to those mentioned in the LDP implementation conformance report[1] which shows their degree of conformance to the LDP standard. We categorize these solutions into LDPR management systems (Callimachus, Carbon LDP, Fedora Commons, Apache Marmotta, Virtuoso, gold, rww-play) and LDP frameworks (Eclipse Lyo,LDP4j. LDP management system can be seen as a repository for LDPRs on top of which CRUD operations, adhering to the LDP standard, are allowed through HTTP methods. LDP frameworks are solutions which can be used to build custom applications which implement LDP interactions. Based on our analysis, there are no LDP solutions which provide support for directly deploying data to LDP repositories.

3 Generic LDPization Process

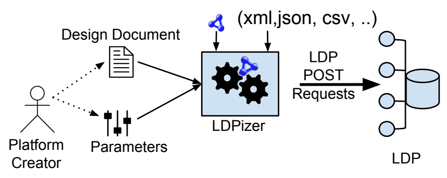

Fig. 1. General View of the LDPization Process

Fig. 1 shows a general view of the LDPization process. The LDPizer takes as input a design document and some parameters. The design document include links to data sources. Using it, the LDPizer transforms the data into LDPRs and finally generate POST requests for them that are sent to an LDP for materialization purposes. We provide an open source[2] implementation of the LDPizer. The architecture of the LDPizer uses principles from model-driven engineering as it

[1] https://www.w3.org/2012/ldp/hg/tests/reports/ldp.html
[2] https://github.com/noorbakerally/LDPizer

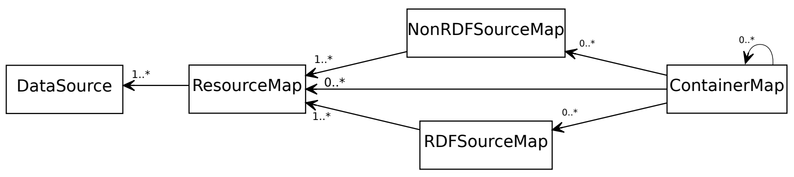

Fig. 2. Simplified Abstract LDP Design Model

allows the decoupling of the design from the implementation, thus enhancing the maintainability and reusability of the design. The design document contains a declarative description of LDP design aspects such as organization of resources in terms of LDPCs, LDPR IRIs or LDPR content. We provide a vocabulary[3] to declaratively describe the design. A simplified abstract model of the vocabulary is shown in Fig. 2.

The core of this model is the `ResourceMap` which can have one or more `DataSources`. The `DataSoure` of a `ResourceMap` can be a an RDF or non-RDF data source. In the latter case, the lifting rule for the data must be provided. Currently, the LDPizer support only lifting rules expressed in SPARQL Generate [2] as it is the only RDFizer which provides a Web API[4]. A `ResourceMap` has two main attributes, the `resourceQuery` and an optional `graphQuery`. For a particular `ResourceMap`, the LDPizer uses its `resourceQuery` to select a set of resources from its `DataSources`. Then using its `graphQuery`, the LDPizer generates an RDF graph for each of the selected resources. For each selected resource, the LDPizer creates an LDPR. If the `ResourceMap` is related to a `ContainerMap` or `RDFSouceMap`, the LDPR is an LDPC or LDP-RS else it is an LDP-NR. Two main aspects of an LDPR is its IRI and content. Per the LDP standard, when creating an LDPR, a slug [1, §9.7] may be provided to indicate some preference about the resource IRI but the final IRI is generated by the LDP. Therefore, a slug template can be provided. When processing the slug template of an LDPR, the IRI and graph (if any) of the resource for which the LDPR has been generated are used as input. The slug template can contain SPARQL expressions. For the content of the LDPR, if the LDPR is an LDPC or LDP-RS, its content is the RDF graph of the resource for which it is being generated. In case of LDP-NR, its content is the content of its corresponding resource which the LDPizer download using the URL of the resource. For each LDPRs, the LDPizer uses its IRI and content, create a POST request and sends it to the LDP where the LDPR is materialized. For the content, if the LDPR is an LDP-RS, its content is the graph of the resource for which the LDP-RS was generated. For LDP-NR, its content is the content of the resource at its URL for which the LDP-NR was generated. Finally, for each LDPRs, the LDPizer uses its slug and content, create a POST request and sends it to the LDP where the LDPR is materialized.

[3] https://github.com/noorbakerally/LDPDesignVocabulary/blob/master/vocabulary.owl

[4] http://ci.emse.fr/sparql-generate/language-api.html

4 Demonstration Scenario

The objective of our demonstration is to show the effectiveness and genericity of the LDPizer in deploying both RDF and heterogeneous data on LDPs and also the reusability of the design document when the data sources use the same vocabularies. In our demonstration, we use Apache Marmotta as the LDP. We show the effectiveness of the LDPizer by using a design document[5] to deploy the data catalog of *Paris Open data Portal*[6] on an LDP. Then, to show the maintainability and reusability of that design document, we only change its data sources and then use it[7] for deploying a different data catalog from *Toulouse Open data Portal*[8]. This is possible because both Paris and Toulouse use the DCAT standard [3] to describe their catalogs. We demonstrate the genericity of the LDPizer by deploying a different dataset using a different design document from the first ones. For this, we use an RDF graph[9] extracted from LinkGeoData[10] using a SPARQL CONSTRUCT query[11] which relates to geographical places in Paris. Finally, we illustrate that the LDPizer can also deploy non-RDF data by using a design document[12] to deploy the JSON dataset[13] for bicycle stations in Paris streets.

5 Future Work

Several improvements can be envisaged as we presented only a preliminary version of the design vocabulary and LDPizer. There are several aspects such as entailment regimes, blank nodes or external resources which we intend to consider in future versions.

Acknowledgments This work is supported by grant ANR-14-CE24-0029 from *Agence Nationale de la Recherche* for project OpenSensingCity.

References

1. J. Gregorio and B. de hOra. The Atom Publishing Protocol. Technical report, IETF, 2007.
2. M. Lefrançois, A. Zimmermann, and N. Bakerally. A SPARQL extension for generating rdf from heterogeneous formats. In *14th ESWC 2017*, 2017.
3. F. Maali and J. Erickson. Data Catalog Vocabulary (DCAT). Technical report, W3C, January 16 2014.
4. S. Speicher, J. Arwe, and A. Malhotra. Linked Data Platform 1.0. Technical report, W3C, February 26 2015.

[5] `https://github.com/noorbakerally/ISWC2017Demo/blob/master/ParisCatalog.dd.ttl`
[6] `https://opendata.paris.fr/`
[7] `https://github.com/noorbakerally/ISWC2017Demo/blob/master/ToulouseCatalog.dd.ttl`
[8] `https://data.toulouse-metropole.fr`
[9] `https://github.com/noorbakerally/ISWC2017Demo/blob/master/ParisGeo.ttl`
[10] `http://linkedgeodata.org/`
[11] `https://github.com/noorbakerally/ISWC2017Demo/blob/master/ParisGeo.rq`
[12] `https://github.com/noorbakerally/ISWC2017Demo/blob/master/ParisStationVelib.dd.ttl`
[13] `https://opendata.paris.fr/explore/dataset/stations-velib-disponibilites-en-temps-reel/`