grlc Makes GitHub Taste Like Linked Data APIs

Albert Meroño-Peñuela[1,2(✉)] and Rinke Hoekstra[1,3]

[1] Department of Computer Science,
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
{albert.merono,rinke.hoekstra}@vu.nl
[2] Data Archiving and Networked Services, KNAW, Amsterdam, The Netherlands
[3] Faculty of Law, University of Amsterdam, Amsterdam, The Netherlands

Abstract. Building Web APIs on top of SPARQL endpoints is becoming common practice. It enables universal access to the integration favorable data space of Linked Data. In the majority of use cases, users cannot be expected to learn SPARQL to query this data space. Web APIs are the most common way to enable programmatic access to data on the Web. However, the implementation of Web APIs around Linked Data is often a tedious and repetitive process. Recent work speeds up this Linked Data API construction by wrapping it around SPARQL queries, which carry out the API functionality under the hood. Inspired by this development, in this paper we present `grlc`, a lightweight server that takes SPARQL queries curated in GitHub repositories, and translates them to Linked Data APIs on the fly.

Keywords: SPARQL · Git · GitHub · Linked Data APIs · RESTFul

1 Introduction

Despite their known benefits for data integration, the Linked Data technologies of RDF and SPARQL still operate in a niche. There is a gap with what average Web-applications and developers have come to expect. RDF and SPARQL remain relatively unknown to the wider Web community. But they are still a requirement for accessing Linked Data. Both have steep learning curves that many developers refuse to face. The W3C specification of SPARQL 1.1 has a limited adoption even within the Linked Data community [16]. Triggered by requirements of Linked Open Data publishers, such as the UK government,[1] the current best practice solution to this problem is the deployment of a custom Web API on top of a Linked Data source. For instance, as an interface to a SPARQL endpoint. This use of APIs to apply the basic principle of encapsulation, and their deployment in large scale Linked Data applications [4], has proved to solve another problem: the expressiveness of SPARQL allows for highly inefficient or

Pronounced as in "garlic", written lowercase with no vowels.
[1] https://github.com/UKGovLD/linked-data-api.

H. Sack et al. (Eds.): ESWC 2016 Satellite Events, LNCS 9989, pp. 342–353, 2016.
DOI: 10.1007/978-3-319-47602-5_48

just computationally expensive queries. Abstracting over curated, 'well behaved' queries allows for more efficient query answering and caching. It is an important step in safeguarding query response times.

However, the deployment and configuration of a Linked Data API is still a cumbersome task. Effective Linked Data APIs require careful management of SPARQL queries, reliable storage, abundant documentation, and the overhead of software maintenance. The latter has been recently addressed by Daga et al. [1], who propose a system that builds API operations automatically by taking a SPARQL query, a short description, and an endpoint location as input. However, the question of how to effectively store and organise such API-translated SPARQL queries remains. As shown in this paper, users require organised APIs that adapt to their existing query curation workflows. Moreover, such APIs might need to coexist in systems where SPARQL queries are already being used by other Linked Data applications. This requires a paradigm shift where not just the data, but the queries themselves also become *first class citizens*.

In several Linked Data projects such as CEDAR [11] and CLARIAH-SDH [6] we followed a practice of storing, curating, and publishing illustrative SPARQL queries of their use cases using GitHub repositories. These queries are then used by various client applications to access Linked Data. An analysis of GitHub shows that this is quite common: it currently hosts at least 5000 SPARQL queries, and potentially many more. A search on 'sparql extension:rq' produces 4987 results; 'query extension:rq' gives 4386 results. The seach for 'rdf language:SPARQL' returns over 46k results, and searching for 'SELECT' queries gives over 280k files, but these include a large number of syntax templates used by Audacity, Virtuoso and ClioPatria. From an e-Science perspective, sharing research questions as concrete queries on GitHub has a huge potential for the reproducibility of research outcomes.

In this paper, we investigate how the current practice of curating queries in open GitHub repositories can be decoupled from the custom built applications that use them to interact with Linked Data. This way we can to lower the costs of (1) constructing APIs for Linked Data and (2) developing applications that interact with Linked Data. Concretely, the contribution of this paper is:

- A mapping between a Swagger RESTful API specification and SPARQL query repositories accessible through the GitHub API;
- a decorator syntax to enrich SPARQL queries in Git repositories with metadata about their intended use (Sect. 3.3); and
- a description of the `grlc` service, that automatically exposes such enriched SPARQL queries in GitHub repositories as Linked Data APIs (Sect. 4)

As for the rest of the paper, we survey relevant related work in Sect. 2, evaluate our approach in two use cases in Sect. 5, and conclude in Sect. 6.

2 Related Work

The organization and management of SPARQL queries is central to the study of their efficiency, nature, and use at improving Linked Data applications. SPARQL

query logs have been used to study differences between queries by humans and machines [13]. These logs are also useful to understand semantic relatedness of queried entities [7]. Saleem et al. [14] propose to "create a Linked Dataset describing the SPARQL queries issued to various public SPARQL endpoints". Loizou et al. [9] identify (combinations of) SPARQL constructs that constitute a performance hit, and formulate heuristics for writing optimized queries. To the best of our knowledge, no previous work addresses the use of collaborative code platforms to ease deployment of Web APIs.

The Semantic Web has developed significant work on the relationship between Linked Data and Web Services [3,12]. In [15], authors propose to expose REST APIs as Linked Data. These approaches suggest the use of Linked Data technology on top of Web services. Our work is related to results in the opposite direction, concretely the Linked Data API specification[2] and the W3C Linked Data Platform 1.0 specification, which "describes the use of HTTP for accessing, updating, creating and deleting resources from servers that expose their resources as Linked Data"[3]. Kopecký et al. [8] address the specific issue of writing (updating, creating, deleting) these Linked Data resources via Web APIs. However, our work is more related to providing Linked Data access interfaces that function as SPARQL wrappers, like the OpenPHACTS Discovery Platform for pharmacological data [4], and the BASIL server [1]. These approaches build Linked Data APIs compliant with the Swagger RESTful API specification[4] on top of SPARQL endpoints. Our contribution proposes an additional decoupling of Linked Data APIs with SPARQL query curation infrastructures, in order to lower the costs of building and maintaining such APIs.

3 From GitHub Repositories to Linked Data APIs

Git is becoming increasingly popular for maintaining projects that revolve around other content than code: a wide variety of projects use Git repositories to store data and queries over them. This development can be largely attributed to the popularity of GitHub[5], a cloud-based Git repository hosting service [10]. For example, we use GitHub to store important SPARQL queries and templates for the CEDAR and CLARIAH projects. This has brought two key outcomes. First, it contributes to *better maintainability* of the life cycle of SPARQL queries. By leveraging git features and GitHub's infrastructure, queries become easily reusable (since they get unique, dereferenceable URIs), their provenance better traceable [2], their development (through frictionless branching) less error-prone, and their versioning effortless. Second, it lowers *coupling* between SPARQL queries and applications, by separating their development and maintenance workflows while keeping queries accessible. As a consequence, queries are less frequently hard-coded and retyped. The goal of `grlc`

[2] https://github.com/UKGovLD/linked-data-api.
[3] https://www.w3.org/TR/2015/REC-ldp-20150226/.
[4] https://github.com/OAI/OpenAPI-Specification.
[5] https://github.com/.

is to profit from this the decoupling of applications and queries to streamline the infrastructure for building and exposing Linked Data APIs.

This section investigates how the organisational characteristics of GitHub repositories can be used to build, manage and maintain a Swagger-spec compliant API. First, in Sect. 3.2 we study the requirements of Swagger-compliant APIs and map them to elements of the GitHub API. Since these elements are insufficient for a complete API spec, in Sect. 3.3 we propose to complete it with non-intrusive SPARQL decorators.

3.1 The Swagger Specification and User Interface

The OpenAPI Specification, previously known as the Swagger[6] Specification, is a standard[7] specification for machine-readable data structures that describe, produce, consume, and visualize RESTful Web services. Given such a data structure, a variety of tools can generate API code, documentation, and test cases.

The OpenAPI specification allows for the declarative description of API resources, such as operation names (called *paths*), their human-readable descriptions, available access methods (e.g. HTTP `GET`, `POST`), parameters, available output formats, and expected responses. The standard is ongoing work, and there exists abundant documentation on all supported features[8]. Client applications can read these declarative resource descriptions, and consume services without knowledge about their specific implementations. The Swagger UI[9] is an example of such a client application. It reads Swagger service specifications, and produces a web-base user interface that displays the contents of the API and offers a variety of ways of interaction.

3.2 Mapping Swagger and GitHub

We propose to align the metaphor of the *repository* with that of the *API*, since both share abstract notions of organizing files and operations in a way that is meaningful for their users. For this reason, in this section we study a possible mapping between the two. Table 1 shows the mapping between the attribute requirements of the Swagger RESTful API specification (see Sect. 3.1), and how these correspond with either attributes of the GitHub API (repository organisation elements) or attributes of the SPARQL usage decorator (usage metadata elements). The latter are discussed in Sect. 3.3.

3.3 SPARQL Decorators

To complete the mapping of the GitHub API to the Swagger RESTful API specification shown in Table 1, we propose *SPARQL decorators* (or tags) to add metadata in queries as comments. These decorators are used by `grlc` to build more

[6] See http://swagger.io/ and https://github.com/swagger-api.
[7] See https://openapis.org/.
[8] For a complete overview of the specification, see http://swagger.io/specification/.
[9] See https://github.com/swagger-api/swagger-ui.

Table 1. Mappings between the Swagger RESTful API and the GitHub API/SPARQL decorators. Such decorators, and the query itself, are parsed through accessing any file with the extension `.rq` in the repo via `GET` http://raw.githubusercontent.com/:owner/:repo/master/

Swagger attribute	Scope	Description	Mapping
Swagger version	API	Version number of compliant Swagger RESTful API specification	**Static**: independent of the LDA. Currently version 2.0 of the Swagger RESTful API spec is supported
API version	API	Version number of the API	**GitHub API**: last repo release from the release API through `GET /repos/:owner/:repo/releases/latest`
Title	API	Title of the API	**GitHub API**: name of the repository through `GET /repos/:owner/:repo`
Contact name	API	Author and contact information	**GitHub API**: login name of the repository owner through `GET /repos/:owner/:repo`
Contact URL	API	URL to be followed for additional information	**GitHub API**: link to the HTML page of the repository owner through `GET /repos/:owner/:repo`
License	API	License under which the API is released	**Repository file**: a link to the raw `LICENSE` file of the repo if it exists; empty otherwise
Host	API	Host name to compose the API calls	**grlc parameter**: supplied host name in grlc's configuration; `localhost` by default
Base path	API	Base path to compose the API calls	**GitHub API**: the string `/:owner/:repo` in `GET /repos/:owner/:repo`
Schemes	API	Supported schemes to compose the API calls	**Static**: `http` is supported
Path name	Operation	Name of the API operation	**GitHub API**: the file name, without the extension, of any `.rq` file found in the repository
Path method	Operation	HTTP method for the operation (`GET`, `POST`)	**Static**: `GET` is supported
Path tags	Operation	Tags under which the operation will be classified	**SPARQL decorator**: the parsed list of the decorator *tags* in `.rq` files
Path description	Operation	Description of the API operation	**SPARQL decorator**: the parsed *description* decorator in `.rq` files
Path parameters	Operation	Parameters of the operation	**SPARQL decorator**: all parameter placeholders parsed in the query (see Sect. 4)
Path responses	Operation	Responses of the operation	**SPARQL decorator**: response codes on success, datatypes of parameters (see Sect. 4)

accurate and descriptive Linked Data APIs (see Sect. 4). We assume SPARQL queries organised as `.rq` files in git repositories. Each of these files will translate into an API operation. We propose to comment them in the first file lines, with the syntax depicted in the following example[10]:

```
#+ summary: A brief summary of what the query does
#+ method: GET
#+ endpoint: http://example.org/sparql
#+ tags:
#+     - UseCase1
#+     - Awesomeness
```

This indicates the *summary* of the query (which will document the API operation), the `http` *method* to use (`GET`, `POST`, etc.), the *endpoint* to send the query, and the *tags* under which the operation falls in. The latter helps to keep operations organized within the API. In addition, we suggest to include two special files in the repository. The first is a `LICENSE` file containing the license for the SPARQL queries and the API. The second is the `endpoint.txt` file, with the URI of a default endpoint to direct all queries of the repository. When parsing the repository (see Sect. 4) the target endpoint will be the one indicated by the `#+ endpoint` decorator, the `endpoint.txt` file, or http://dbpedia.org/sparql, in this order of preference.

4 grlc

`grlc`[11] is a thin gateway that automatically builds complete, well documented, and neatly organized Linked Data APIs on the fly, with no input required from users beyond a GitHub user and repository name. To do so, it implements the GitHub API mappings proposed in Sect. 3.2, and uses the SPARQL decorators described in Sect. 3.3. The public online instance of `grlc` is located at http://grlc.io/.

`grlc` provides three basic operations: *(1)* generates the Swagger spec of a specified GitHub repository; *(2)* generates the Swagger UI (see Sect. 3.1) to provide an interactive user-facing frontend of the API contents; and *(3)* translates `http` requests to call the operations of the API against a SPARQL endpoint with several parameters. If the GitHub repository at https://github.com/:owner/:repo contains decorated SPARQL queries, `grlc` uses these, together with organisational repo information from the GitHub API, to build the API interface automatically. Assuming that `grlc` is running at `:host`, these operations are available at the following routes:

- `http://:host/:owner/:repo/spec`: JSON Swagger-compliant specification, using the mappings of Sect. 3
- `http://:host/:owner/:repo/api-docs`: Swagger-UI, rendered using such mappings, as shown in Fig. 1.

[10] Additional examples can be found at https://github.com/CEDAR-project/Queries and https://github.com/CLARIAH/wp4-queries.

[11] Source code at https://github.com/CLARIAH/grlc.

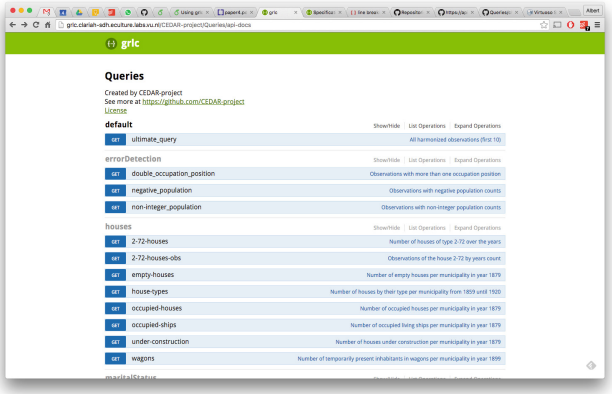

Fig. 1. Screenshot of the Swagger user interface generated by `grlc`

– `http://:host/:owner/:repo/:operation?p_1=v_1...p_n=v_n`: `http GET` request to `:operation` with parameters $p_1, ..., p_n$ taking values $v_1, ..., v_n$.

`grlc` composes the Swagger spec as follows: *(1)* the user requests the URI `http://:host/:owner/:repo/spec`[12] to a host running `grlc`; *(2)* `grlc` issues the `http GET` request to the GitHub API at `https://api.github.com/repos/:owner/:repo`, using the owner and repo names indicated in the previous step; *(3)* for each `.rq` file described in the response, `grlc` derefences https://raw.githubusercontent.com/:owner/:repo/master/file.rq to get the SPARQL file contents; *(3)* `grlc` parses these file contents to extract: *(a)* the values of the decorators (if any), and *(b)* any parameter placeholders in the query; *(4)* `grlc` uses all the gathered data to compose the Swagger spec, and returns it to the client as JSON. The composition of the Swagger UI is analogous: first the JSON spec is composed and, after, it is used to render the Swagger UI template[13].

4.1 Parameter Mapping

It is often useful for SPARQL queries to be parameterized. This happens when a resource in a basic graph pattern (BGP) can take specific values that affect the result of the query. Previous work has investigated how to map these values to parameters provided by the API operations [1,4].

`grlc` is compliant with BASIL's convention for Web API parameters mapping[14]. This means that some "parameter-declared" SPARQL variables of a query are interpreted by `grlc` as parameter placeholders. Hence, operations of the

[12] Requested from any `http` compliant client: a Web browser, `curl`, etc.
[13] https://github.com/swagger-api/swagger-ui.
[14] See https://github.com/the-open-university/basil/wiki/SPARQL-variable-name-convention-for-WEB-API-parameters-mapping.

```
SELECT (SUM(?pop) AS ?tot) FROM <urn:graph:cedar-mini:release> WHERE {
        ?obs a qb:Observation.
        ?obs sdmx-dimension:refArea ?_location_iri.
        ?obs cedarterms:Kom ?_kom_iri.
        ?obs cedarterms:population ?pop.
        ?slice a qb:Slice.
        ?slice qb:observation ?obs.
        ?slice sdmx-dimension:refPeriod ?_year_integer.
        ?obs sdmx-dimension:sex ?_sex_iri.
        ?obs cedarterms:residenceStatus ?_residenceStatus_iri.
        FILTER (NOT EXISTS {?obs cedarterms:isTotal ?total }) }
```

Listing 1.1. Example of a parametrized SPARQL query (prefixes ahave been omitted).

```
SELECT DISTINCT ?_sex_iri FROM <urn:graph:cedar-mini:release> WHERE {
        ?obs sdmx-dimension:sex ?_sex_iri. }
```

Listing 1.2. Rewritten query by `grlc` to retrieve plausible values for the parameter `?_sex_iri`.

form `http://:host/:owner/:repo/:operation?p_1=v_1...p_n=v_n`, are executed by `grlc` by: *(1)* retrieving the raw SPARQL query from https://raw.githubusercontent.com/:owner/:repo/master/:operation.rq; and *(2)* rewriting this query by replacing the placeholders by the parameter values $v_1, ..., v_n$ supplied in the API request. After this, the query is submitted to the endpoint indicated by the methods described in Sect. 3.3. The endpoint results are forwarded to the client application.

An example of such a parameterized query[15] is shown in Listing 1.1. SPARQL variable names staring with `?_` and `?__` indicate mandatory and optional parameters, respectively. If they end with `iri` or `integer`, they are expected to be mapped to IRIs and literal (integer) values, respectively. These parameters are replaced by the user provided parameter values $v_1, ..., v_n$ by `grlc`'s query rewriting engine.

Parameter Enumerations. In order to guide the user at providing valid parameter values, `grlc` tries to fill the enumeration `get->parameters->enum` attribute of an operation in the Swagger specification. This (optional) attribute contains an array with all possible values that a parameter can take. To generate this enumeration, `grlc` sends an additional SPARQL query to the endpoint, replacing the original BGP by the triple pattern where the parameter appears. For instance, to fill the enumeration of parameter `?_sex_iri` in Listing 1.1, `grlc` retrieves its plausible values from the endpoint with the query shown in Listing 1.2. Figure 2 shows an example of how the Swagger UI displays these plausible parameter values.

[15] The original query can be found at https://github.com/CEDAR-project/Queries/blob/master/residenceStatus_params.rq.

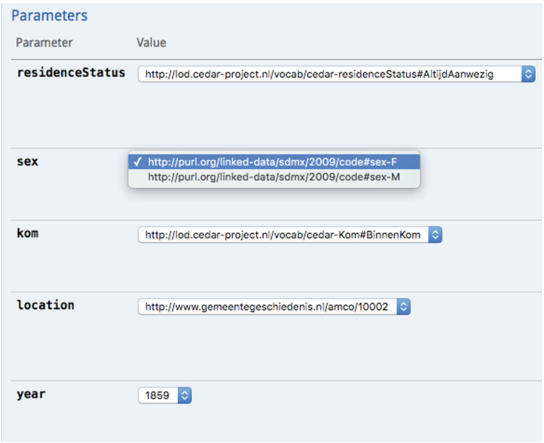

Fig. 2. Screenshot of the Swagger user interface rendering parameter enumerations generated by `grlc`

4.2 Content Negotiation

`grlc` supports content negotiation at two different levels: by *request*, and by *URL*. By request, `grlc` checks the value of the `Accept` header in incoming `http` requests. By URL, `grlc` checks whether a route calling an API operation ends with a trailing `.csv`, `.json` or `.html`.

In both cases, the corresponding `Accept http` header is used in the request to the SPARQL endpoint, delegating support of specific content types to each endpoint. When the response from the server is received, `grlc` sets the `Content-Type` header of the client response to match that received by the endpoint, and therefore it only proxies both requests and responses.

4.3 Caching

Building a Swagger specification retrieving data from the GitHub API can be a costly operation. One GitHub request is needed to retrieve the repository contents; n additional GitHub requests are needed for the n queries in the repository; and m extra endpoint requests per query are required when such query contains m parameters. Thus, the performance of `grlc` can be severely affected by cumulative network latency.

In order to mitigate this cost, `grlc` implements a simple *cache* that maintains the following data structure:

`{ <repo_uri> : { 'date' : <date>, 'spec' : <spec> } }`

`<repo_uri>` is the URI of a requested GitHub repository; `<date>` is the timestamp at which `grlc` generated the Swagger spec for that repository for the last time; and `<spec>` is the JSON data structure containing the Swagger spec itself.

Every time there is an incoming request to generate a spec for the repository located at `<repo_uri>`, `grlc` checks whether there is a cached spec for `<repo_uri>` in the

cache. If there is, `grlc` compares: *(a)* the date at which the repository `<repo_uri>` was updated for the last time, by requesting the value of `pushed_at` to the GitHub API; and *(b)* the date `<date>` of the cached spec. If the cached copy is more recent than the last GitHub update, `grlc`'s cache is up to date and the cached spec can be used instead of generating it from scratch.[16] If there is no `<repo_uri>` in the cache, or its date is older than the last GitHub update, the spec is generated from scratch, as described in Sect. 4, and the cache is updated. This makes `grlc` about 20 times faster to generate API specs, at the cost of building them from scratch when the cache is empty or outdated.

5 Preliminary Evaluation

In this section we evaluate requirements satisfied by `grlc` in two use cases.

Dutch Historical Census Data. The CEDAR project[17] has published the Dutch historical censuses (1795–1971) as 5-star Linked Data [11]. Key queries and templates to interrogate this dataset are available at GitHub[18]. These queries are used in various client applications[19,20]. Before `grlc`, we decided to implement a minimal effort Web API using our own instance of BASIL[21]. However, the queries needed to be retyped in the system, and caused ramifications with respect to the ones in our existing applications. Moreover, it was not possible to mimic the organisation these queries had in the original GitHub repo in the API spec. After `grlc`, we could create this API without interfering with the original applications and queries, effectively reusing them. Furthermore, `grlc` permitted an ecosystem where SPARQL and non-SPARQL savvy applications coexist.

Born Under a Bad Sign. In CLARIAH[22], querying structured humanities data from combined sources is central. This particular use case focuses on validating the hypothesis that prenatal and early-life conditions have a strong impact on socioeconomic and health outcomes later in life, by using 1891 census records of Canada and Sweden. These were converted to Linked Data with QBer [6], and analyzed in the statistical environment `R`. Before `grlc`, loading the data to be analyzed implied the manual download of a SPARQL query resultset in a file, and then loading this file in `R`. This was mitigated with the `R` SPARQL package [5]. However, this resulted in hard-coded, hardly reusable, and difficult to maintain queries. After better organising these queries in a GitHub repository, an API using them became immediately available through `grlc`. As shown in Fig. 3, the `R` code became *clearer* due to the decoupling with SPARQL; and *shorter*, since a `curl` one-liner calling a `grlc` enabled API operation sufficed to retrieve the data.

[16] Note that the cache currently does not track updates to the underlying data. This means that the parameter enumerations 'grlc' generates can become outdated for more dynamic datasets.
[17] http://www.cedar-project.nl/.
[18] https://github.com/CEDAR-project/Queries.
[19] YASGUI-based browsing: http://lod.cedar-project.nl/cedar/data.html.
[20] Drawing historical maps with census data: http://lod.cedar-project.nl/maps/map_CEDAR_women_1899.html.
[21] https://github.com/the-open-university/BASIL.
[22] http://clariah.nl/.

```
## using grlc API call
library(RCurl)
canada <- getURL("http://grlc.clariah-sdh.eculture.labs.vu.nl/clariah/wp4-
canada <- read.csv(textConnection(canada))
sweden <- getURL("http://grlc.clariah-sdh.eculture.labs.vu.nl/clariah/wp4-
sweden <- read.csv(textConnection(sweden))

fit_canada_base <- lm(log(hiscam) ~ log(gdppc), data=canada)
fit_canada <- lm(log(hiscam) ~ log(gdppc) + I(age^2) + age, data=canada)
fit_sweden_base <- lm(log(hiscam) ~ log(gdppc), data=sweden)
fit_sweden <- lm(log(hiscam) ~ log(gdppc) + I(age^2) + age, data=sweden)
```

Fig. 3. The use of `grlc` makes Linked Data accessible from any `http` compatible application

6 Conclusion and Future Work

In this paper we presented `grlc`, a novel approach to automatically build Linked Data APIs by using SPARQL queries stored and documented in git repositories. Our approach addresses two pitfalls of current practice in constructing Linked Data APIs: *(1)* the coupling of SPARQL curation workflows and the API infrastructure, which hampers query reuse and forces query retyping and ramifications; and *(2)* the common lack of organisation in Linked Data APIs. `grlc` maps the Swagger specification with GitHub API features and a proposed SPARQL decorator notation, and builds and maintains Linked Data APIs automatically with minimal effort. We argue that this approach enables a better coexistence of SPARQL and non-SPARQL savvy applications, and allows developers to switch their efforts from API infrastructure to applications.

We plan to extend this work in several ways. First, we will support additional repository elements and SPARQL decorators. Second, we will add compatibility with other collaborative coding platforms, like Bitbucket and GitLab, enabling private APIs and authentication. Third, we will investigate ways to map API results pagination and the SPARQL keywords `LIMIT` and `OFFSET`. Fourth, we will investigate the use of `grlc`-built APIs for the retrieval of generic endpoint metadata (schema, VoID, etc.) without the need of SPARQL. Finally, we plan to create a `grlc` companion to facilitate the curation of SPARQL queries in Git repositories.

Acknowledgments. This work was funded by the CLARIAH project of the Dutch Science Foundation (NWO) and the Dutch national programme COMMIT. The work on which this paper is based has been supported by the Computational Humanities Programme of the Royal Netherlands Academy of Arts and Sciences. For further information, see http://ehumanities.nl. We want to thank the reviewers for their valuable comments and suggestions.

References

1. Daga, E., Panziera, L., Pedrinaci, C.: A BASILar approach for building web APIs on top of SPARQL endpoints. In: Services and Applications over Linked APIs and Data SALAD2015 (ISWC 2015), vol. 1359, CEUR Workshop Proceedings (2015). http://ceur-ws.org/Vol-1359/

2. De Nies, T., Magliacane, S., Verborgh, R., Coppens, S., Groth, P., Mannens, E., Van de Walle, R.: Git2PROV: Exposing version control system content as W3C PROV. In: Poster and Demo Proceedings of the 12th International Semantic Web Conference (2013). http://www.iswc2013.semanticweb.org/sites/default/files/iswc_demo_32_0.pdf
3. Fielding, R.T.: Architectural styles and the design of network-based software architectures (2000)
4. Groth, P., Loizou, A., Gray, A.J., Goble, C., Harland, L., Pettifer, S.: API-centric linked data integration: the open PHACTS discovery platform case study. Web Semant.: Sci. Serv. Ag. World Wide Web **29**, 12–18 (2014). http://www.sciencedirect.com/science/article/pii/S1570826814000195, life Science and e-Science
5. van Hage, W.R., with contributions from: Tomi Kauppinen, Graeler, B., Davis, C., Hoeksema, J., Ruttenberg, A., Bahls., D.: SPARQL: SPARQL client (2013). http://CRAN.R-project.org/package=SPARQL. Rpackageversion1.15
6. Hoekstra, R., Meroño-Peñuela, A., Dentler, K., Rijpma, A., Zijdeman, R., Zandhuis, I.: An ecosystem for linked humanities data. In: Proceedings of the 1st Workshop on Humanities in the Semantic Web (WHiSe 2016), ESWC 2016 (2016, under review)
7. Huelss, J., Paulheim, H.: What SPARQL query logs tell and do not tell about semantic relatedness in LOD. In: Gandon, F., et al. (eds.) ESWC 2015. LNCS, vol. 9341, pp. 297–308. Springer, Heidelberg (2015). doi:10.1007/978-3-319-25639-9_44
8. Kopecký, J., Pedrinaci, C., Duke, A.: Restful write-oriented API for hyperdata in custom RDF knowledge bases. In: 2011 7th International Conference on Next Generation Web Services Practices (NWeSP), pp. 199–204, October 2011
9. Loizou, A., Angles, R., Groth, P.: On the formulation of performant SPARQL queries. Web Semant.: Sci. Serv. Ag. World Wide Web **31**, 1–26 (2015). http://www.sciencedirect.com/science/article/pii/S1570826814001061
10. McMillan, R.: From collaborative coding to wedding invitations: github is going mainstream. Wired Magazine, 9 February 2013. http://www.wired.com/2013/09/github-for-anything/all
11. Meroño-Peñuela, A., Guéret, C., Ashkpour, A., Schlobach, S.: CEDAR: the Dutch historical censuses as linked open data. Semant. Web - Interoper. Usabil. Appl. (2015, in press)
12. Pedrinaci, C., Domingue, J.: Toward the next wave of services: linked services for the web of data. J. Univ. Comput. Sci. **16**(13), 1694–1719 (2010)
13. Rietveld, L., Hoekstra, R.: Man vs. machine: differences in SPARQL queries. In: Proceedings of the 4th USEWOD Workshop on Usage Analysis and the Web of of Data, ESWC 2014 (2014). http://usewod.org/files/workshops/2014/papers/rietveld_hoekstra_usewod2014.pdf
14. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.-C.N.: LSQ: the linked SPARQL queries dataset. ISWC 2015. LNCS, vol. 9367, pp. 261–269. Springer, Heidelberg (2015). doi:10.1007/978-3-319-25010-6_15
15. Speiser, S., Harth, A.: Integrating linked data and services with linked data services. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part I. LNCS, vol. 6643, pp. 170–184. Springer, Heidelberg (2011)
16. Vandenbussche, P.Y., Aranda, C.B., Hogan, A., Umbrich, J.: Monitoring the status of SPARQL endpoints. In: 12th International Semantic Web Conference on Proceedings of the ISWC 2013 Posters and Demonstrations Track (ISWC 2013), pp. 81–84. CEUR-WS (2013)