A Simple API to the KnowledgeStore

Ian Hopkinson[1], Steven Maude[1], and Marco Rospocher[2]

[1] ScraperWiki, ic2, Liverpool Science Park, 146 Brownlow Hill, Liverpool L3 5RF
[2] Fondazione Bruno Kessler, 18 Via Sommarive, 38123 - Trento Povo (TN), ITALY

Abstract. RDF and SPARQL technologies have not gained widespread acceptance amongst web developers. We describe a simple API which is used to provide access to the KnowledgeStore, a scalable, fault-tolerant, and Semantic Web grounded storage system for interlinking structured and unstructured data, developed in the contect of the FP7 NewsReader EU project. The simple API wraps a set of parameterised SPARQL queries to access the KnowledgeStore RDF structured content, and calls to the KnowledgeStore CRUD endpoint to retrieve unstructured resources. Responses are delivered as JSON, JSONP, HTML or CSV. The API is largely self-documenting. The API is written using the Flask Python library, and includes an extensive suite of tests. It is modular, so that new SPARQL queries can be added easily or it could be used as a template for providing an API to any SPARQL endpoint. The simple API was succesfully exploited by 38 web developers, many of them unfamiliar with RDF and SPARQL technologies, to build web applications on top of the KnowledgeStore.

Keywords: SPARQL, Virtuoso, Python

1 Introduction

NewsReader [1] is an FP7 project which seeks to provide a ”news recorder” which processes news articles to extract ”events” using a combination of natural language processing and semantic web technologies with the aim of enhancing our ability to ”understand” the news. News articles are marked up using an NLP pipeline. Events are extracted from the output of the pipeline, and they are coreferenced across multiple documents.

The original articles, marked up articles and RDF event descriptions are stored in the KnowledgeStore [2,3], a scalable, fault-tolerant, and Semantic Web grounded storage system for interlinking structured and unstructured data. Specifically, the KnowledgeStore consists of a HBase database, for storing (marked up) articles and metadata, and a Virtuoso [4] triplestore, containing details of the events and their participants. In addition a subset of DBpedia [5] is included in the triplestore component to provide background knowledge.

The KnowledgeStore exposes the triplestore component via a SPARQL endpoint, while documents and document metadata are made available through a CRUD (create, retrieve, update, delete) endpoint. Both endpoints are part of the KnowledgeStore HTTP ReST API. However, SPARQL (and RDF) has not gained widespread acceptance amongst web developers, and in addition there are issues with managing free form

SPARQL queries against a large sized triplestore ($\sim$300M triples). Therefore there is a need to provide a Simple API to provide access to this resource, this we describe in the following sections.

2 Methods

Python is a well-established language for general computing with a rich, accessible ecosystem of libraries. In this work we use the Flask [6] web serving library which provides simple routing to help build web applications. Although there are Python libraries designed to interact with SPARQL and RDF they do not yet appear to be stable or well-supported. Therefore we handle the SPARQL queries and responses using the requests library. A useful feature of the requests [7] library is that caching can be implemented very easily via [8]. This allows us to reduce load on the SPARQL endpoint and improve responsiveness to user queries.

The Simple API is currently on-line[1] and its code is available[2] and released under the Apache License.

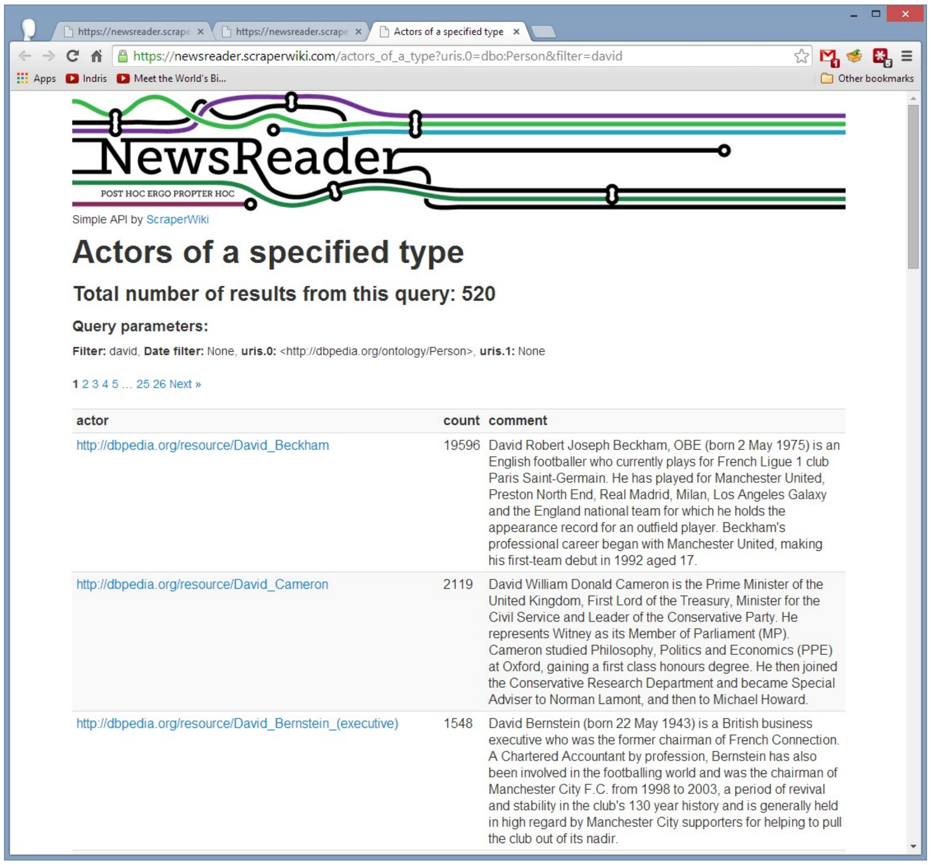
The API supports the following queries:

– actors_of_a_type
– describe_uri
– event_details_filtered_by_actor
– event_label_frequency_count
– get_document
– get_document_metadata
– get_mention_metadata
– people_sharing_event_with_a_person
– properties_of_a_type
– property_of_actors_of_a_type
– summary_of_events_with_actor
– summary_of_events_with_actor_type
– summary_of_events_with_event_label
– summary_of_events_with_framenet
– summary_of_events_with_two_actors
– types_of_actors

Which can variously take these parameters:

– callback = a function with which to wrap the JSON response to make JSONP
– datefilter = YYYY, YYYY-MM or YYYY-MM-DD, filter to a year, month or day
– filter = a character string on which to filter, it can take combinations such as bribery+OR+bribe
– limit = a number of results to return
– offset = an offset into the returned results

[1] https://newsreader.scraperwiki.com/
[2] http://www.bitbucket.org/scraperwikids/newsreader_api_flask_app


https://newsreader.scrape
https://newsreader.scrape
Actors of a specified type
https://newsreader.scraperwiki.com/actors_of_a_type?uris.0=dbo:Person&filter=david
Apps
Indris
Meet the World's Bi...
Other bookmarks
NewsReader
POST HOC ERGO PROPTER HOC

Simple API by ScraperWiki

Actors of a specified type

Total number of results from this query: 520

Query parameters:

Filter: david, **Date filter:** None, **uris.0:** <http://dbpedia.org/ontology/Person>, **uris.1:** None

1 2 3 4 5 ... 25 26 Next »

actor	count	comment
http://dbpedia.org/resource/David_Beckham	19596	David Robert Joseph Beckham, OBE (born 2 May 1975) is an English footballer who currently plays for French Ligue 1 club Paris Saint-Germain. He has played for Manchester United, Preston North End, Real Madrid, Milan, Los Angeles Galaxy and the England national team for which he holds the appearance record for an outfield player. Beckham's professional career began with Manchester United, making his first-team debut in 1992 aged 17.
http://dbpedia.org/resource/David_Cameron	2119	David William Donald Cameron is the Prime Minister of the United Kingdom, First Lord of the Treasury, Minister for the Civil Service and Leader of the Conservative Party. He represents Witney as its Member of Parliament (MP). Cameron studied Philosophy, Politics and Economics (PPE) at Oxford, gaining a first class honours degree. He then joined the Conservative Research Department and became Special Adviser to Norman Lamont, and then to Michael Howard.
http://dbpedia.org/resource/David_Bernstein_(executive)	1548	David Bernstein (born 22 May 1943) is a British business executive who was the former chairman of French Connection. A Chartered Accountant by profession, Bernstein has also been involved in the footballing world and was the chairman of Manchester City F.C. from 1998 to 2003, a period of revival and stability in the club's 130 year history and is generally held in high regard by Manchester City supporters for helping to pull the club out of its nadir.

Fig. 1. Example of API response

- output = json—html—csv
- uris.[n] = a URI to a thing, e.g. dbpedia:David_Beckham

Queries look like this:

- `http://newsreader.scraperwiki.com:5000/actors_of_a_type?uris.0=dbo:Person&filter=david`
- `http://newsreader.scraperwii.com:5000/summary_of_events_with_framenet?uris.0=framenet:Omen`

A typical response is shown in Figure 1.

The application contains two modules: A *viewer module*, based on Flask, contains routing patterns which handle the URL queries the user can present, converting them into calls to the queries module. The viewer module also converts the query responses to the required format and similarly presents any error messages. The *queries module* defines a parent SPARQLQuery class which contains functions to process the query arguments supplied by the viewer module and feed them into SPARQL query templates. Each SPARQL query is implemented in a separate file which defines a class based on the parent SPARQLQuery class. The key component of this file is a template for the query into which parameters are inserted. It also contains fields for a description of the query, an example query, and the required and optional parameters. This separation of concerns makes implementing new queries straightforward.

The queries module also defines a CountQuery class and a CRUDQuery class, derived from the SPARQLQuery class. Each SPARQL query also defines a count query so that the user can be informed of the number of results returned by a query, and the results can be paged. The CRUDQuery class is defined to cover the three queries that use the CRUD endpoint (thus, the Simple API has also the effect of hiding the two separate KnowledgeStore endpoints behind a common interface).

The filter arguments for queries are implemented in a modular fashion such that they can be added to suitable SPARQL queries with little effort. If no filter arguments are supplied, then the statements implementing the filter are not applied.

Visiting the API root provides an HTML documentation page which is built dynamically from the queries that have been defined. In early development the example queries were used directly as a manual integration test. The advantage of this approach is that text required for documentation is largely derived directly from executable code, and ”optional” descriptive text is entered close to where the thing it describes is implemented.

During the Hack Day (see Section 2.1) we identified that URIs containing the period character outside of the prefix were not handled by the SPARQL endpoint, although expanding prefixes and wrapping in $<$ and $>$ worked for these URIs. Therefore we implemented prefix expansion.

The API was implemented on REST principles which implies that it holds no state information, meaning that queries must be delivered ”live” - i.e. they cannot ask the user to come back later for a response. This presents some problems for the Simple API since the underlying triplestore is large and queries can be potentially slow running. We mitigated these risks by optimising the available SPARQL queries - we were able to achieve speed increases up to a factor of 10 by re-ordering execution order for SPARQL statements and using some Virtuoso built-in (e.g. BIND) rather than generic functions.

The API contains moderate levels of testing which we implemented using the nosetests [9] library. We used the mock library to emulate error states on the SPARQL endpoint in order to demonstrate correct handling of these error states.

The Bootstrap [10] framework is used to style the HTML output of the API, and the jinja [11] templating library is used to build HTML pages from the SPARQL query responses. Since the majority of SPARQL queries implemented use the SELECT keyword, the output is tabular in form. This means that the output of new queries of this form can easily be inserted into a single template which handles all tabular output. Graph-like output is more difficult to handle generically and for this reason it is currently only presented as JSON.

2.1 Applications

The Simple API was used by 38 web developers, many of them unfamiliar with RDF and SPARQL technologies, for a Hack Day. The SPARQL endpoint received 30,000 queries during the course of the day, with a peak of 20 queries per second. The SPARQL server rebooted automatically three times during the day and only one query timed out. This highlights the protective functionality of the Simple API. Users, in general, only had access to pre-determined queries which were well understand in terms of their execution costs and had been optimised upfront. The penalty for this was reduced flexibility, but for a SPARQL-naive audience this is no penalty. It does mean the onus is on the authors of the Simple API to provide the right suite of queries.

2.2 Further work

The Simple API is a new piece of work, and we have identified a number of areas for improvement.

Currently the API does not output graph responses from SPARQL queries as HTML, it simply provides the raw query response in JSON. During our Hack Day we observed that users frequently made describe_uri queries; typically for a limited range of entity types. Therefore there is the potential to provide HTML templates for certain types of query returning graph responses.

In principle, we can combine the results of multiple SPARQL or CRUD queries inside the API, to provide results which could not be returned by a single SPARQL query.

During the Hack Day users created a number of applications. These were good proofs-of-concept, though somewhat unpolished given the time constraints. We would like to build some sample applications on top of the API.

We made the decision early on to provide output in paged form, this is in keeping with how web APIs typically deliver responses. This is an issue for our implementation of the Simple API since we use the LIMIT and OFFSET statements to page through results, Virtuoso has a a maximum OFFSET (currently set to 10,000) therefore for queries returning a large number of results we cannot page to all the results. The maximum OFFSET is in place for a reason: performance degrades when handling large offsets. There are a couple of approaches to dealing with this problem, one would be to return entire result sets, without using the LIMIT and OFFSET keywords. Alternatively, SQL databases use cursors to handle this problem and it may be possible to apply this approach using the Virtuoso triplestore.

3 Lessons learned

The core developers for the Simple API were effectively SPARQL-naive at the start of the process but with a background in SQL. This was remedied by Bob Ducharme's book[12], Learning SPARQL and consulting with other members of the team. The biggest challenge was that learners typically learn against toy datasets rather than a system containing hundreds of millions of triples, such as the KnowledgeStore. This slows their skill acquisition.

Conceptually the triple-matching style of SPARQL querying is alien but in some ways more straightforward than SQL particularly when carrying out the equivalent of a join. Using SELECT queries means that responses are returned in familiar tables shapes for which we found generic processing patterns.

The second challenge for RDF naive developers is the feeling that they are forever being pointed somewhere else for an answer to a query. A URI is not an answer, it's a pointer to an answer. Discovering the rdfs:label and rdfs:comment types helps resolve this tension, and should highlight to triplestore developers their importance.

By the end of the development process the naive members were writing functional SPARQL queries. The SPARQL-expert members of the team were able to significantly optimise those queries using a combination of query ordering, a wider knowledge of SPARQL constructs and better knowledge of the particular triplestore implementation.

Non-RDF lessons learned:

– It is easy to make everything look better with Bootstrap;
– Testing web applications using mocking is surprisingly straightforward with the appropriate libraries, and very useful.

4 Conclusions

We have described the development of a Simple API which can be used to access a complex resource containing both a triplestore accessed via a SPARQL endpoint and a document collection with a CRUD API. The Simple API has been used in a Hack Day where it enabled users to access content with the need to know SPARQL. Furthermore, the API ensured a responsive system by limiting the available queries.

Acknowledgments

The research leading to this paper was supported by the European Union's 7th Framework Programme via the NewsReader Project (ICT-316404).

References

1. NewsReader, http://www.newsreader-project.eu/
2. Francesco Corcoglioniti, Marco Rospocher, Roldano Cattoni, Bernardo Magnini, Luciano Serafini: Interlinking Unstructured and Structured Knowledge in an Integrated Framework. 7th IEEE International Conference on Semantic Computing (ICSC), Irvine, CA, USA, 2013
3. KnowledgeStore, http://knowledgestore.fbk.eu
4. Virtuoso, http://virtuoso.openlinksw.com/
5. DBpedia, http://dbpedia.org/
6. Flask, http://flask.pocoo.org/
7. Requests, http://docs.python-requests.org/en/latest/
8. requests_cache, http://requests-cache.readthedocs.org/
9. nosetests, https://nose.readthedocs.org/en/latest/
10. Bootstrap, http://getbootstrap.com/
11. jinja, http://jinja.pocoo.org/
12. Ducharme, B., Learning SPARQL, O'Reilly, (2013).