The DataTank: an Open Data adapter with semantic output

Miel Vander Sande
Ghent University - IBBT
ELIS - Multimedia Lab
9050 Ghent, Belgium
miel.vandersande@ugent.be

Pieter Colpaert
iRail NPO
9000 Ghent, Belgium
pieter@irail.be

Davy Van Deursen
Ghent University - IBBT
ELIS - Multimedia Lab
9050 Ghent, Belgium
davy.vandeursen@ugent.be

Erik Mannens
Ghent University - IBBT
ELIS - Multimedia Lab
9050 Ghent, Belgium
erik.mannens@ugent.be

Rik Van de Walle
Ghent University - IBBT
ELIS - Multimedia Lab
9050 Ghent, Belgium
rik.vandewalle@ugent.be

ABSTRACT

The idea of Open Data states that by making data sets freely available on the Internet, data owners can benefit from a huge community. In this paper, we extend The DataTank framework, a data adapter for publishing local Open Data as a Web API, to produce semantic output. A data set and its content are identified by a unique URI, exploiting the REST interface and differentiating between IR and NIR. Also, a data set can be requested as RDF in multiple notations. Furthermore, ontology information can be added to a data model, thus creating machine-understandable RDF. This ontology information is made externally reusable and changeable. The DataTank now produces semantic output, while the original architecture remains unchanged.

1. INTRODUCTION

Today the Web contains an enormous amount of data used in websites or applications. Unfortunately, this data is locked away on servers, only available to the owner. Recently, the idea of Open Data arose, stating that certain data should be freely available on the Web for anyone to use or redistribute. This way, anybody can start building applications with this data, thus addressing the creativity, input and workload of a huge community. As a result, the return from this community is way larger than a single company can handle, while the investment is a lot less.

Tim Berners-Lee, the creator of the Web, defined participation in Open Data as five stars [1]. First, put data on the Web with an open license. Second, it is structured and machine readable. Third, it has a non-proprietary format. Fourth, use URIs to identify things, so people can point at it, and return something meaningful in a machine-readable language (e.g., RDF). Fifth, link your data to data from others. In this research we focus on helping governments (and others) to easily go from the second star to the fourth, making the step into valuable Open Data smaller and faster.

The DataTank[1] is our open-source data adapter platform. It publishes local data on-the-fly as a web API. This makes the original data directly and remotely usable for developers. This maximizes the potential of your data, while the extra effort of the user is minimized. Our goal is to extend this platform with meta-data description (making it machine-understandable) and output in the RDF standard, while keeping the original philosophy of minimal effort in mind.

2. RELATED WORK

Several works on publishing Open Data as Web APIs exist, but they all differ in their field of application. Web API layer tools [3] focus on publishing Open Data through a standard interface into multiple standard formats. They offer flexibility and get good performance results. However, they duplicate the data, lowering the owner's control. Furthermore, they do not consider any semantic output.

Closely related is the catalogue software CKAN [4], which mainly provides Open Data storage and access. They offer a good user interface for managing data sets and support linked data as well. Access is provided through an API that can be queried and provide multiple output formats. Meta data can be added to the data sets and versioning is included. Although RDF can be retrieved, this is only possible if the source is originally in RDF. Also, the web interface they offer is not RESTful and every data set is stored on their server.

In the area of RDF conversion is the RDF extension for Google Refine [2] is a popular solution, but only works with off-line datasets. More web-oriented approaches are database abstraction systems [5]. A virtual RDF abstraction is created on top of relational databases, to serve a SPARQL processor. However, the abstraction is based on database schemas and manually defined mappings. They therefore require the data source to be a relational database, or at least imported into one.

[1]developed by iRail V.Z.W - http://thedatatank.com

3. THE DATATANK AS DATA ADAPTER

The DataTank is a PHP server application forming a data adapter between the original data source and the data consumer (e.g., App developer). Each dataset is available as a *Resource*, which is part of a virtual directory, called a *Package*. This is then accessible through the URI `http://<host>/<package>/<resource>`. We can break the philosophy down in three main features: RESTful, data storage avoidance and multiple format support. First, the data in the adapter can be addressed through a RESTful interface. This defines the whole data flow, as it is build with a CRUD[2] functionality in mind. Data sets are read, added, changed or deleted by sending a corresponding HTTP GET, PUT, POST or DELETE request to the server. Second, the data is not stored, thus giving data owners more insight in how much, when, where and by whom their data is used. Only the location of the data source is stored, implying an on-the-fly approach when converting and publishing the data. Third, data can be converted from multiple data formats to

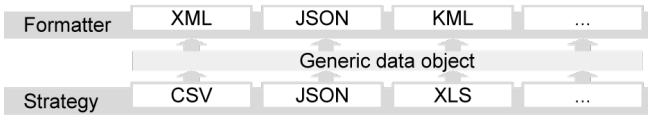

Figure 1: Basis architecture of a read operation, supporting multiple formats.

multiple data formats. This is supported by implementing a loosely coupled module structure, so extra formats can be added easily. As illustrated in Figure 1, input formats are handled by a *Strategy*, containing parsing logic for a structured data format(e.g., CSV). A resource will dynamically create a generic data object based on the original data, using the right strategy. Output formatting is handled by a *Formatter*, implementing serialisation logic for turning the generic data object into a specific web format (e.g., XML).

4. PRODUCING SEMANTIC OUTPUT

In this research, we extended The DataTank to produce four star data, implying an RDF conversion. For this, we focused on three points. First, we created a clear distinction in URIs between things and their representations on the web. Second, we implemented a ontology mapping module, which maps the generic data object model to an ontology. Third, we combined the data and the mapping to generate a RDF graph.

We manipulate RDF and OWL models using RAP (RDF API for PHP)[3]. We chose this library because of its resource-centric manipulation methods, database persistent model, parsing in multiple notations and serialization in multiple notations. We extended the API to work with the database abstraction layer of The DataTank, for using database persistent models.

4.1 Differentiating between an IR and a NIR

The RESTful interface of The DataTank allows data sources to be browsed hierarchically. For example, a row-column value of a tabular data source can be accessed by requesting `.../<package>/<resource>/<rownumber>/<columnlabel>`.

[2]Create, Read, Update, Delete

[3]`http://www4.wiwiss.fu-berlin.de/bizer/rdfapi/`

This automatically defines a unique URI for every entity in a data set. In line with the fourth star of Open Data, we exploit these URIs to enable semantic descriptions about entities embedded in a data set.

Before these URIs can be used in the Semantic Web, we require a different URI for identifying a NIR (Non Information Resource) then for identifying an IR (Information Resource). A NIR is something that we can describe, but has no physical form on the Web (e.g., the White House). An IR, is typically serialised data describing a NIR (e.g., the string "White House"). This distinction is important, since they are semantically different and we might want to describe them separately. In The DataTank, a request to a URI returns a web document, which is always an IR. We add a file extension to create a separate URIs for IRs. This way, users can immediately access the data in their desired format.

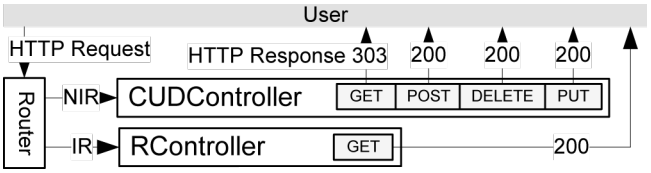

Figure 2: HTTP Request/Response cycle with dereferencable URIs

Although we determined a clear distinction in URIs, the question remains: what to respond in case a NIR is requested? To tackle this issue, we respond with a HTTP 303 response, redirecting the user to the corresponding IR URI. Therefore, we split the Controller into the *RController* and the *CUDController*, as shown in Figure 2. The former handles all HTTP requests on IRs. It only needs to process GET requests, since IRs only need to be read. The latter handles all HTTP requests on NIRs. In case of a GET request, the URI is rewritten by adding the *.about* extension, a HTTP 303 Redirect response is assembled and this response is send back to the user.

4.2 Ontology mapping

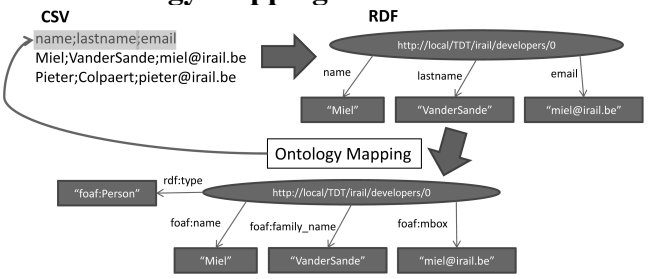

Figure 3: Using ontology mapping to derive RDF

RDF uses classes and properties defined in ontologies for describing data to a machine. Therefore, we implemented a way to map the internal data model to ontology members. This mapping needs to be easily definable, reusable and externally changeable. As illustrated in Figure 3, ontology mapping is done on the PHP class structure, so the result is the same for every instance. Implementing this raises some issues. In The DataTank, an instance of the generic class *stdClass* is created, to which the fields of a data set are dynamically added. As a result, the class definition is only

known at runtime, preventing a straight-forward mapping. Furthermore, because of the loosely typed nature of PHP, we do not know in advance which type a certain property will contain.

To solve these issues and acquire our goals, we chose to describe the data model in an own ontology. For every member in the data object, we take the class path and define it as *owl:Class* or *rdf:Property* (depending whether it's a class or a property in PHP). This approach facilitates the mappings, since we can add any ontology member to the class path by using *owl:equivalentClass* or *rdf:equivalentProperty*. An example is shown in Listing 1.

```
@prefix owl:<http://www.w3.org/2002/07/owl#>.
@prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix foaf:<http://xmlns.com/foaf/0.1/>.
@base <http://datatank.demo.ibbt.be/TDTInfo/Ontology/>.

<developers> a owl:Class
<developers/stdClass> a owl:Class;
        owl:equivalentClass foaf:Person .
<developers/stdClass/name> a rdf:Property;
        owl:equivalentProperty foaf:name .
<developers/stdClass/lastname> a rdf:Property;
        owl:equivalentProperty foaf:lastname .
<developers/stdClass/email> a rdf:Property;
        owl:equivalentProperty foaf:mbox .
```

Listing 1: The ontology of the resource "developers" with FOAF mapping in N3

Now that we covered the modeling and mapping, we use the existing REST architecture to publish this ontology. A fixed resource *Ontology* in the package *TDTInfo*[4] was created to handle the requests. We can browse the ontology using this type of URIs: `.../TDTInfo/Ontology/<package>/<classpath>`. Every request is treated by the architecture like a normal resource request, but instead of accessing data sources, it calls the *OntologyProcessor*. This class implements the CRUD structure to manage ontologies. They are stored in the database and manipulated through RAP.

To visualise our ontology mapping system, we created a graphical web client called The Semantifier. This application interacts with the ontology through the REST API.

4.3 Formatting RDF output

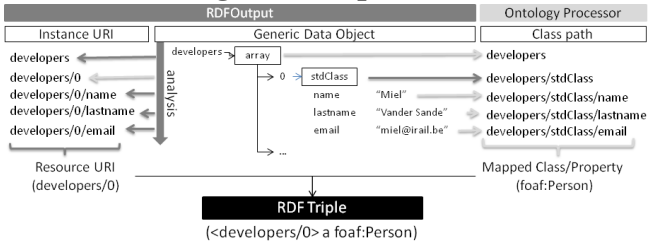

Figure 4: Creating RDF triples by analyzing the generic object model

When a resource is requested in a certain web format (e.g, JSON), the generic data object is passed to the right formatter (e.g., JSONFormatter) to return the parsed to the user. We create RDF output using this principle. We created a formatter for each of the most common notations (e.g., N3, RDF/XML). The serialization is already supported by RAP, hence we only need to generate a RDF model from the data. As a result, the generic data object is first passed to the class *RDFOutput*, which creates and returns a model.

To create a RDF model, we analyse the generic data object recursively. While iterating through the object, an instance URI and class path are constructed for each object or property. This class path is used to retrieve the corresponding mapping from *OntologyProcessor*. An instance URI, a mapping and a value are used to produce an RDF triple, as shown in Figure 4.

5. USE CASE: OGD WIEN

As an example, we took the hospitals dataset in the JSON format. By sending an HTTP PUT request to our The DataTank demo server, supplying the file's URI as a parameter, we created the resource *hospitals* in the package *vienna*. Once this resource was created, the data can already be accessed in various formats (e.g., `.../vienna/hospitals.xml`). Next, we created an ontology using The Semantifier and mapped Schema.org and Dublin Core concepts to its fields. This makes the dataset automatically compatible with all datasets described with Schema.org concepts. By requesting an RDF extension, we can get an on-the-fly created RDF representation of the hospitals in Vienna, each with their own unique, dereferencable URI and mapped properties. An article about this use case can be found on the *Open Government Data Wien* website[5] and live demo can be found on our demoserver[6].

6. CONCLUSIONS

Data owners can benefit from a huge community of consumer by publishing their data sets as Open Data. The DataTank is a data adapter that publishes local datasets as a RESTful Web API. By extending this framework, we added semantic output for every data set. The REST URIs were exploited to create a distinction between Information Resources (IR) and Non Information Resources (NIR). Also, a solution was implemented for requesting NIRs. We offered a way to easily describe the data model and corresponding ontology mapping. This information is made reusable and changeable by using the REST interface. Finally, a recursive method was introduced for creating an RDF model .

7. REFERENCES

[1] T. Berners-Lee. Linked data - design issues, 2006.

[2] V. P. Fadi Maali, Richard Cyganiak. Re-using cool uris: Entity reconciliation against lod hubs. In *Proceedings of the Linked Data on the Web Workshop 2011 (LDOW2011)*, 2011.

[3] C. O. Jeffrey Cafferata. Rotterdam open data store (rods), 2011.

[4] D. D. Rufus Pollock. Ckan: apt-get for the debian of data. 2011.

[5] S. S. Sahoo, W. Halb, S. Hellmann, K. Idehen, T. T. Jr, S. Auer, J. Sequeda, and A. Ezzat. A survey of current approaches for mapping of relational databases to rdf, 01 2009.

[4] the standard package for retrieving system information in The DataTank

[5] `http://data.wien.gv.at/apps/datatank.html` (german)

[6] `http://datatank.demo.ibbt.be/vienna/hospitals`