Undefined 1 (2012) 1–6
IOS Press

Facebook Linked Data via the Graph API

Editor(s): Pascal Hitzler, Kno.e.sis Center, Wright State University, Dayton, OH, USA; Krzysztof Janowicz, University of California, Santa Barbara, USA
Solicited review(s): Michael Hausenblas, DERI Galway, Ireland; Ivan Herman, W3C; Amit Joshi, Kno.e.sis Center, Wright State University, USA

Jesse Weaver [a,*] and Paul Tarjan [b]
[a] *Tetherless World Constellation, Rensselaer Polytechnic Institute, 110 8th St., Troy, NY, USA*
E-mail: weavej3@cs.rpi.edu
[b] *Facebook Inc., 1601 Willow Road, Menlo Park, CA, USA*
E-mail: pt@fb.com

Abstract. Facebook's Graph API is an API for accessing objects and connections in Facebook's social graph. To give some idea of the enormity of the social graph underlying Facebook, it was recently announced that Facebook has 901 million users, and the social graph consists of many types beyond just users. Until recently, the Graph API provided data to applications in only a JSON format. In 2011, an effort was undertaken to provide the same data in a semantically-enriched, RDF format containing Linked Data URIs. This was achieved by implementing a flexible and robust translation of the JSON output to a Turtle output. This paper describes the associated design decisions, the resulting Linked Data for objects in the social graph, and known issues.

Keywords: Linked Data, Facebook, Graph API, Turtle, JSON

1. Introduction

Facebook's Graph API[1] "presents a simple, consistent view of the Facebook social graph, uniformly representing objects in the graph (e.g., people, photos, events, and pages) and the connections between them (e.g., friend relationships, shared content, and photo tags)" [6]. To give some idea of the enormity of the social graph underlying Facebook, it was recently announced that Facebook has 901 million users [7], and the social graph consists of many types beyond just users. Therefore, publishing the social graph as Linked Data would significantly contribute to the Web of Data.

Until recently, the Graph API provided data to applications in only a JSON [4] format. In 2011, an effort was undertaken to provide the same data in a semantically-enriched, RDF format containing Linked Data (HTTP(S)) URIs that dereference in accordance with httpRange-14 [2]. The effort had three primary restrictions: (1) to make only minimal changes to existing code, (2) to make the solution robust enough to require little (if any) maintenance over time, and (3) to avoid (for the present time) XML formats. This was achieved by implementing a flexible and robust translation of the JSON output to a Turtle [1] output made accessible via HTTP content negotiation. This paper describes the associated design decisions, some of the resulting Linked Data, and known issues.

Although the Graph API has many features, due to constraints on paper length, the focus of this paper is on Linked Data data about objects (in exclusion of connections) in the social graph. However, as a rule of thumb, any JSON that can be obtained from the Graph API can also be obtained as Linked Data RDF in the Turtle syntax. Therefore, one need only consult the Graph API documentation [6] to learn about available data and method of access. The Linked Data represents

*Corresponding author. E-mail: weavej3@cs.rpi.edu.

[1] It is a common mistake to confuse the Graph API with the Open Graph Protocol (OGP) [10]. The OGP is a standard for placing metadata in web pages which, when followed, allows Facebook to integrate web pages into its internal, social graph. The Graph API is an API which provides access to the internal, social graph.

0000-0000/12/$00.00 © 2012 – IOS Press and the authors. All rights reserved

only the underlying graph which does not connect to external, non-information resources on the web, and as such, it may be considered only four-star Linked Data.

The following is basic information regarding the Linked Data:

- The base **URL** for accessing the Linked Data is `http://graph.facebook.com/`, although the HTTPS scheme can also be used wherever the HTTP scheme is used.
- Concerning **versioning**, since the published Linked Data reflects the dynamic state of the underlying social graph, it is not appropriate to give the data itself a version number. It can be said, however, that the current methodology of publication is the first one employed by Facebook and was publicly announced in September 2011.
- Regarding **availability**, it should be noted that the Graph API respects permissions. Some basic information is public for some types of objects in the social graph. The OAuth 2.0 protocol [8] must be used and appropriate permissions obtained in order to access any other information.
- **Metrics** and **statistics** related to the Linked Data are proprietary and therefore cannot be included in this paper.

For abbreviation in this paper, all relative URIs are resolved against `http://graph.facebook.com/`. We also define the following prefixes for CURIEs [3].

- `rdf:`, `rdfs:`, `owl:`, `xsd:`, and `foaf:` have their usual definitions.
- `:` (the empty prefix) is the prefix for `http://graph.facebook.com/schema/~/`.
- `api:` is the prefix for `tag:graph.facebook.com,2011:/` (tag URIs discussed in section 3).
- any other prefixes used herein, e.g. *`user`*`:`, are defined as `http://graph.facebook.com/schema/`*`user`*`#`.

All examples are given in Turtle syntax with ellipses sometimes placed in long URIs or literals to shorten the appearance in this paper.

The remainder of this paper is organized as follows. Section 2 describes the method of converting JSON output to RDF triples, followed by section 3 which discusses the details of minting and supporting Linked Data URIs in conformance with httpRange-14. Section 4 describes how the RDF is semantically enriched by exposing underlying object schema (in the broadest sense) as ontologies and using ontology terms wherever possible. Section 5 gives examples of the Linked Data, section 6 discusses known issues, and section 7 concludes the paper.

2. Converting JSON to RDF

The primary interchange format of the Graph API is JSON, a well-known, standard format for lightweight interchange of data [4]. A detailed description of JSON would be superfluous herein due to JSON's wide adoption. It is sufficient to say that JSON consists of two kinds of sets of key/value pairs: JSON objects[2] in which keys are strings; and arrays in which the set of keys forms a finite, counting sequence of non-negative integers beginning with zero. The values of the pairs can be JSON objects, arrays, or primitives. JSON primitives include strings, numbers, booleans, and `null`.

Given that XML syntaxes were not an option, the only standard RDF syntax RDF/XML could not be used. Turtle is a *de facto* standard, RDF syntax, and anticipating its actual standardization in RDF 1.1 [5], Turtle was the apparent best choice in lieu of RDF/XML.[3]

Design Decision 1. *Turtle was chosen as the RDF syntax to which the JSON output should be translated.*

Translating JSON objects and arrays to triples is a relatively straightforward process. A JSON object (or array) is assigned a URI or blank node as its RDF identifier to be used as subject in RDF triples. The key/value pairs of the JSON object (or array) are used to formulate predicate/object pairs in RDF triples, where the key must be translated into a URI and the value must be translated into a sensible RDF term (URI, blank node, or literal).

Primitive values are translated into RDF literals using heuristics to determine a possible datatype URI with which to type the literal, with the exception that JSON strings that form URIs are translated to URIs. Commonly employed datatype URIs are `xsd:integer`, `xsd:decimal`, `xsd:double`, `xsd:boolean`, and `xsd:dateTime`. The exception is

[2] The overloaded word "object" is avoided herein. Henceforth, "instance" is used to refer to objects in the Facebook social graph, "JSON object" is used to refer to non-array sets of key/value pairs in JSON, and "object" is used to refer to the object position of an RDF triple where the context makes its meaning clear.

[3] JSON-LD was considered but disregarded since its conventions varied too widely from the existing JSON format.

that instance identifiers (which often look like integers) are preserved as strings. In anticipation of RDF 1.1, regular strings are not explicitly typed as `xsd:string` but rather are left as plain literals which, under RDF 1.1, are implicitly understood to have datatype `xsd:string`.

The real difficulty lies in assigning URIs in such a way that their dereference behavior complies with httpRange-14. After some discussion, the following decision was made.

Design Decision 2. *Wherever possible, hash URIs should be used in preference over slash URIs.*

The reasoning behind this is simple: it is less work to support hash URIs than slash URIs. Supporting hash URIs consists only of publishing the data to be fetched, but supporting slash URIs also entails setting up redirection. However, sometimes slash URIs are necessary, as is illustrated in section 4.

3. Linked Data URIs for instances

Every instance in the social graph has a unique identifier. If an identifier *id* is a single, positive integer – call such an identifier a primitive identifier – then information about the instance it identifies can be obtained from `/id`. Some instances in the social graph (e.g., statuses) are identified by concatenating integers together with underscores. At present, when such composite identifiers are used in a HTTP(S) request for application/json, then the HTTP(S) response returns with code 200 OK containing only the text `false`.

The Graph API follows a single convention in describing instances in the social graph, and that is, instances are translated into JSON objects in which there must be a key/value pair where the key is the string "id" and the value is a string representation of the identifier. We utilize this convention to assign RDF URIs to instances.

As mentioned, information about an instance with primitive identifier *id* can be found at `/id`. Thus, the simplest solution for minting a Linked Data URI for the instance is to append a fragment to the URI. The common conventions of using fragments `#this` and `#me` were considered, but in the end, the empty fragment `#` was chosen so as to require the least amount of modification to the URI possible.

URI Pattern 1. *An instance in the social graph with primitive identifier* ***n*** *is identified in RDF by the URI* `/`***n***`#`.

Instances with composite identifiers pose a particular problem. If they are assigned URIs in the same manner as with instances having primitive identifiers, then dereferencing `/id#` will simply return `false` in JSON with code 200 OK, but in reality, there is some information about the instance identified with *id*. Thus, the dereferencing behavior would be somewhat undesirable.

The most correct solution would be to change the Graph API to actually return data about the instance identified by the composite identifier, but that would violate the constraint that only minimal changes be made to existing code. The possibility of allocating a blank node was considered, but that would lead to well-known problems with blank node proliferation. Thus, a global identifier is needed for which dereferencing would not be a concern. Tag URIs [9] are employed for this very purpose.

Tag URIs are URIs of the form `tag:`*`host`*`,`*`date`*`:`*`remainder`*, where by convention, the URI is controlled by whomever owns the host name *host* at the time *date*, and *remainder* should be interpreted in that context. Tag URIs in the Graph API use host graph.facebook.com and date 2011.

URI Pattern 2. *An instance in the social graph with a composite identifier* ***i*** *is identified in RDF by the URI* `api:id_`***i***. *(Recall that the* `api:` *prefix is defined as* `tag:graph.facebook.com,2011:/`.*)*

Sometimes the JSON output contains JSON objects (or arrays) that do not contain an "id" key (note that arrays always meet this criterion). There are some special cases that are treated differently, but in general, such a JSON object (or array) is considered to represent an anonymous instance – usually something serialized in the JSON that is not directly represented as an instance in the social graph – and therefore a blank node is allocated to represent it.

Design Decision 3. *If a JSON object (or array) cannot be determined to represent an identifiable instance, as a last resort, it should be represented with a blank node in RDF.*

4. Ontologies and their Linked Data URIs

Every instance in the social graph has an associated type, and some types have static properties associated with them. When converting from JSON to RDF, the converter uses identifiers to look up associated types.

Alternatively, if the JSON object describing the instance has an explicit "type" key, the value associated with the key is used. The typing information is used to enrich property URIs to have type-specific semantics whenever possible. If the JSON object contains the "type" key, then a corresponding `rdf:type` triple is included in the RDF. As mentioned before, though, the challenge is in minting the URIs such that their dereference behavior complies with httpRange-14.

In order to support dereferencing of ontology URIs, the schema information needs to be published on the web. Thus, the special `/schema` path was created for accessing information about type schema. For a given type *class*, the schema description can be found at `/schema/`*`class`*.

URI Pattern 3. *A type* ***c*** *is identified by the URI* ***c:****type*.

Example. *Type/class* `user:type`

```
user:type a rdfs:Class ;
  rdfs:label "user" .
```

For a given type, it can be determined whether properties are associated with it, and sometimes these type-specific properties have associated descriptions and richer semantics that can be used to enrich the Linked Data descriptions.

URI Pattern 4. *A type-specific property* ***p*** *associated with type* ***c*** *is identified by the URI* ***c:p***.

When a JSON object is converted to RDF, if a type can be found for the represented instance, then each key in the JSON object is checked against the type schema to see if a rich semantics can be found. If it can, then the key is translated into a type-specific property URI. However, sometimes a type cannot be found for an instance, or the JSON object has a key that does not identify a type-specific property. In this case, the key is translated into a generic property URI.

URI Pattern 5. *A generic property for a key* ***p*** *is identified by the URI* ***:p***.

Example. *Generic property* `:name`

```
:name a rdf:Property ;
  rdfs:label "name" ;
  rdfs:comment "A tag having no ...." .
```

In general, a generic property *prop* has a limited semantics described as: "A tag having no semantics beyond the conventional semantics of the JSON key '*prop*' as used in the Facebook Graph API." In other words, a generic property has the equivalent semantics of its corresponding JSON key. It follows that a type-specific property is a subproperty of the generic property with the same local name.

Design Decision 4. *For any type-specific property* ***c:p****, it holds that* ***c:p*** `rdfs:subPropertyOf` ***:p***.

Example. *Type-specific property* `user:name`

```
user:name a rdf:Property ;
  rdfs:label "name" ;
  rdfs:comment "The user's full ...." ;
  rdfs:domain user:type ;
  rdfs:subPropertyOf :name .
```

There are some special generic properties with stronger semantics. One is `:id` because the conventional semantics of the JSON key "id" can be codified to some extent.

Example. *Special generic property* `:id`

```
:id a owl:InverseFunctionalProperty ;
  rdfs:label "id" ;
  rdfs:comment "A tag having no ...." ;
  rdfs:range xsd:string .
```

The other special case is the handling of non-negative integers. These properties are defined specifically for converting numeric keys in JSON objects and are an attempt to improve upon the design of RDFS container membership properties by making the index number explicit in the RDF.

Example. *Special generic property* `:_0`

```
:_0 a rdf:Property ;
  rdfs:label "_0" ;
  rdfs:comment "A tag having no ...." ;
  rdfs:subPropertyOf api:has ;
  api:index 0 .
```

The property `api:has` is analogous to `rdfs:member`, and the property `api:index` is used to explicitly state the numeric index of the property. Note that these numeric-indexing properties are used only for JSON objects with numeric keys; arrays use only `api:has`.

Note that generic property URIs are the only slash URIs used. If hash URIs were used, then the document retrieved using the fragment-stripped URI would need to contain information about all possible terms in the namespace. However, a JSON key could be any conceivable finite string, and since there are infinitely many finite strings, the fetched document would need to be infinitely large, making the use of hash URIs infeasible for this particular purpose. Thus, slash URIs must be used because then the entire URI is received by the server and can be used to generate information for the single term.

Design Decision 5. *Fetching a description of the generic property URI* `:p` *results in 303 redirection to* `/schema?tag=p`.

5. Examples

Having completed the description of Linked Data publication, this section provides some examples. The best way to find more examples is to do public keyword search using `http://graph.facebook.com/search?q=keywords`, requesting text/turtle in HTTP content negotiation. The first example is of public data available for users, and the second is an example of data for a photo. The third example shows how Facebook Linked Data URIs can be used to augment a person's FOAF profile.

Example. *User instance, public description*

```
</1340421292#> user:id "1340421292" ;
  user:name "Jesse Weaver" ;
  user:first_name "Jesse" ;
  user:last_name "Weaver" ;
  user:username "jrweave" ;
  user:gender "male" ;
  user:locale "en_US" .
```

Example. *Photo instance description*

```
api:id_1340421292_3145012107816 a photo:type;
 photo:id "1340421292_3145012107816" ;
 photo:from </1340421292#> ;
 :message "If you thought they ...." ;
 photo:picture <https://....jpg> ;
 photo:link <http://www.facebook.com/...> ;
 photo:icon <https://....gif> ;
 :actions [
   api:has [
     :name "Comment" ;
     :link <http://www.facebook.com/...> ] ;
   api:has [
     :name "Like" ;
     :link <http://www.facebook.com/...> ] ];
 :privacy [
   :description "Public" ;
   :value "EVERYONE" ] ;
 photo:created_time
   "2012-05-03T17:54:16+00:00"^^xsd:dateTime;
 photo:updated_time
   "2012-05-03T17:56:10+00:00"^^xsd:dateTime.
```

Example. *FOAF profile augmentation*

```
# base URI defined just to improve clarity
@base <http://graph.facebook.com/> .
<http://www.cs.rpi.edu/~weavej3/foaf.rdf#me>
  owl:sameAs </1340421292#> ;
  rdfs:seeAlso </1340421292?metadata=1> ;
  foaf:depiction </1340421292/picture> ;
  foaf:account
    <http://www.facebook.com/jrweave> .
```

6. Known Issues

6.1. Lack of External Links

Although it includes links to many information resources, the Facebook Linked Data does not link to external, non-information resources. This is because the Linked Data is provided by a dynamic translation of the existing JSON output to a Turtle output, the process of which cannot dynamically determine how to link Facebook instances to external instances on the Web of Data.

However, since they change infrequently, Facebook ontologies could conceivably link to external ontologies. For example, it seems intuitively true that `user:type rdfs:subClassOf foaf:Agent`. Such links could be easily supported by maintaining a static document for each Facebook type, to which the dynamic pages could refer by including the triple – for example – `user:type rdfs:seeAlso <file-URI>`.

6.2. HTTPS URIs

As shown in previous examples, relative URIs are used in the Turtle output to identify instances (with primitive identifiers), but the Turtle output declares no explicit base URI. Therefore, the URI of the document becomes the base URI against which all relative URIs are to be resolved, and that URI could use either the HTTP or HTTPS scheme. Therefore, instances are identified by two URIs differing only in whether the scheme is HTTP or HTTPS. While this is technically valid, it is certainly poor practice for the same publisher to publish multiple URIs for the same thing. However, this can easily be remedied by including a base URI declaration in the Turtle output, stating which URI should be used for resolution.

Of greater concern, though, is the fact that some of the Linked Data can only be retrieved using HTTPS with an access token specified in the URL query (respecting permissions). This is not so much a problem for instance data but rather for connections, a topic which has not been covered herein due to limitations on paper length. For obvious reasons, Linked Data URIs should not include private access tokens. Without access tokens, dereferencing such URIs results in 400 Bad Request, but it is odd for a server to state that a URI originally provided by that server is malformed. This is the most significant issue. The best solution would be to support HTTP authentication and

return 401 Unauthorized, although the simplest solution would be to return a 403 Forbidden.

6.3. Tag URIs

As mentioned in section 3, URIs with the "tag" scheme are utilized to produce global identifiers for which dereferencing is unnecessary. Although this approach is completely valid, the idea of using non-dereferenceable URIs seems to contradict the entire purpose of Linked Data. However, they have been used in the Linked Data primarily for identifying instances with composite identifiers.

The best solution is to make information about the instance identified by composite identifier *id* available at `/id`, whereas now, only the JSON primitive `false` is returned, or if Turtle is requested, empty content is returned. This behavior is allowable but not ideal. Therefore, an alternative solution to using tag URIs would be to use the HTTP(S) URIs that dereference to no (empty) data. This seems like a more correct solution than using tag URIs, but it was disregarded since it may add some maintenance burden in the future. Specifically, there is no current guarantee as to the kind or form of data that will ever be published at `/id`, and thus, it seemed wise to avoid placing any constraints on future data published at `/id` by expecting Linked-Data-friendly behavior. It is fortunate, though, that whenever tag URIs identifying instances are used, associated Linked Data are usually (if not always) included for those instances.

Aside from composite identifiers, tag URIs have been used to identify two properties: `api:has` and `api:index`. The issue with these terms is that there is no published description of them anywhere on the web. The best solution would be to use HTTP(S) URIs that correctly dereference to their descriptions, which would be most easily supported by publishing a static document meant for defining such special terms.

6.4. Empty fragment

Instances with primitive identifiers are identified with URIs of the form `/id#`. Utilizing the empty fragment for identifying things is perfectly valid, but it seems somewhat of an odd choice considering that the fragment is meant to serve as a local identifier.

7. Conclusion

Facebook's Graph API provides access to an enormous amount of data, and exposing the data (respecting permissions) as Linked Data significantly grows the Web of Data. The Linked Data is dynamically formed by translating the JSON output to a Turtle output. The RDF/Turtle output is semantically richer than the JSON output, having explicit semantics made accessible as ontologies utilizing the RDFS and OWL vocabularies. The URIs in the RDF are designed to dereference correctly according to current Linked Data standards, with some exception concerning HTTPS URIs returning 400 Bad Request. Known issues have been discussed and possible solutions presented.

Acknowledgements

Thanks to James A. Hendler and Gregory Todd Williams for reviewing an earlier version of this paper.

References

[1] D. Beckett, T. Berners-Lee, and E. Prud'hommeaux. Turtle. Working draft, W3C, Aug. 2011. http://www.w3.org/TR/2011/WD-turtle-20110809/.

[2] T. Berners-Lee and N. Mendelsohn. ISSUE-14: What is the range of the HTTP dereference function? - Technical Architecture Group Tracker. http://www.w3.org/2001/tag/group/track/issues/14, May 2012.

[3] M. Birbeck and S. McCarron. CURIE Syntax 1.0. Candidate recommendation, W3C, Jan. 2009. http://www.w3.org/TR/2009/CR-curie-20090116/.

[4] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, IETF, July 2006. http://www.ietf.org/rfc/rfc4627.txt.

[5] R. Cyganiak and D. Wood. RDF 1.1 Concepts and Abstract Syntax. Working draft, W3C, Aug. 2011. http://www.w3.org/TR/2011/WD-rdf11-concepts-20110830/.

[6] Facebook. Graph API - Facebook Developers. https://developers.facebook.com/docs/reference/api/, Mar. 2012.

[7] M. Hachman. Facebook Now Totals 901 Million Users, Profits Slip. http://www.pcmag.com/article2/0,2817,2403410,00.asp, Apr. 2012.

[8] E. Hammer, D. Recordon, and D. Hardt. The OAuth 2.0 Authorization Framework. Draft 26, IETF, May 2012. http://www.ietf.org/id/draft-ietf-oauth-v2-26.txt.

[9] T. Kindberg and S. Hawke. The 'tag' URI Scheme. RFC 4151, IETF, Oct. 2005. http://www.ietf.org/rfc/rfc4151.txt.

[10] P. Tarjan and D. Recordon. The Open Graph protocol. http://ogp.me/, Mar. 2012.