GraphQL Schema Generation for Data-Intensive Web APIs

Carles Farré, Jovan Varga, and Robert Almar

Universitat Politècnica de Catalunya, BarcelonaTech
{farre,jvarga}@essi.upc.edu, robert.almar@est.fib.upc.edu

Abstract. Sharing data as a (non-)commercial asset on the web is typically performed using an Application Programming Interface (API). Although Linked Data technologies such as RDF and SPARQL enable publishing and accessing data on the web, they do not focus on mediated and controlled web access that data providers are willing to allow. Thus, recent approaches aim at providing traditional REST API layer on top of semantic data sources. In this paper, we propose to take advantage of the new GraphQL framework that, in contrast to the dominant REST API approach, exposes an explicit data model, described in terms of the so-called GraphQL schema, to enable precise retrieving of only required data. We propose a semantic metamodel of the GraphQL Schema. The metamodel is used to enrich the schema of semantic data and enable automatic generation of GraphQL schema. In this context, we present a prototype implementation of our approach and a use case with a real-world dataset, showing how lightly augmenting its ontology to instantiate our metamodel enables automatic GraphQL schema generation.

Keywords: GraphQL, Data-Intensive Web APIs, Semantic Metamodel

1 Introduction

Ontology-driven approaches have proven successful to deal with data integration challenges thanks to the inherent simplicity and flexibility of ontologies, which make them suitable to define the required unified view [9]. Ontologies may be represented in the form of Resource Description Framework (RDF) [4] triples and then made web-accessible via SPARQL endpoints [3]. Nevertheless, web data access at SPARQL endpoints is uncontrolled and unpredictable, compromising the performance, the response time, and even the availability of the service [2]. Moreover, exposing data directly as RDF triples is quite distinct and even disconnected from the "mainstream" Web API development built on HTTP and JSON [8]. Therefore, it is unsurprising that recent approaches aim at providing a REST API layer on top of semantic data sources [5,7,8].

The GraphQL framework proposed by Facebook [6] represents an alternative to the REST APIs. One of its key benefits is that it enables the retrieval of only required data with a single request. This feature is enabled by describing the available data in terms of a *GraphQL schema*. In this way, instead of using

Farré, C.; Varga, J.; Almar, R. GraphQL schema generation for data-intensive web APIs. A: International Conference on Model and Data Engineering. "Model and Data Engineering, 9th International Conference, MEDI 2019: Toulouse, France, October 28-31, 2019: proceedings". Berlín: Springer, 2019, p. 184-194.
The final authenticated version is available online at https://doi.org/10.1007/978-3-030-32065-2_13

several REST method calls, the user of a GraphQL service can define custom requests that precise the data to be retrieved by using the schema information. Therefore, the GraphQL Schema is a foundation for the flexibility introduced.

In this paper, we couple GraphQL and ontologies to take the most out of the two technologies. In particular, we propose GQL as an RDF-formalized semantic metamodel for GraphQL schema and explain how to take advantage of this metamodel to support the semi-automatic generation of a GraphQL schema from a given RDF ontology. Our contributions can be summarized as follows.

- We define GQL, a semantic metamodel for GraphQL Schema.
- We provide a prototype that, given a semantic schema model instantiating GQL, automatically generates the corresponding GraphQL Schema.
- We present a use case with a real-world dataset in order to demonstrate the feasibility of our approach. The use case shows how lightly augmenting the dataset ontology to instantiate our metamodel enables automatic GraphQL schema generation. Moreover, we explain how our prototype is also able to generate a proof-of-concept GraphQL runtime service that answers the queries posed against its GraphQL schema using the available data.

The rest of the paper is structured as follows. Section 2 summarizes the key aspects of GraphQL. Section 3 describes our semantic metamodel for GraphQL Schema: the GQL Metamodel. Section 4 demonstrates, through a use case, how to take advantage of the GQL Metamodel to enable the generation of GraphQL schemas and services. Section 5 discusses the related work. Finally, Section 6 presents the conclusions and further work.

2 GraphQL

This section provides a short introduction on GraphQL that we have synthesized from the original documentation published by Facebook [6]. A *GraphQL service* provides web-based access to its underlying data sources. The *GraphQL query language* allows defining the request payloads to be sent to the GraphQL service unique endpoint. Such payloads are JSON-like sentences that may contain one or more operations. There are three types of operations: queries, mutations, and subscriptions. In this paper, we focus on the querying capabilities of GraphQL.

GraphQL queries must be written in terms of the *GraphQL schema* that describes the data sources that the GraphQL service exposes. A GraphQL schema defines a set of object *types*, which are similar to object classes but without operations. In this way, an object type specifies a list of attributes, named *fields*, each of which yields one or more values of a specific type. This latter may be either a *scalar* type (String, Integer, etc.) or another object type.

For the sake of an example, Figure 1(a) presents a (fragment of the) GraphQL schema that we obtain for our use case dataset about movies (for further details see Section 4). Figure 1(b) presents a corresponding GraphQL query that, for a given `film` (`id: "../film/100"`[1]), returns its `title`, `date`, director's name

[1] Shorthand for "http://data.linkedmdb.org/resource/film/100"

Fig. 1. Example GraphQL schema (a), query (b), and result (c).

(`director_name`), and the names of the genres (`film_genre_name`) to which it belongs. Finally, Figure 1(c) shows a JSON result returned by the GraphQL service after processing the GraphQL query in Figure 1(b). This example illustrates two of the main features of the GraphQL query language:

1. GraphQL queries define templates for the data to be retrieved by selecting the specific scalar fields that are of interest to the user.
2. GraphQL queries can be nested, i.e., for each field of the object type in a GraphQL query, even if it is a list of objects, a GraphQL (sub) query can be defined. The query displayed in Figure 1(b) contains two subqueries: `director { director_name }` and `genre { film_genre_name }`, which retrieve the names of the objects `director` and `genre` associated with a `film`.

The GraphQL schema in Figure 1(a) defines several "normal" object types (`film`, `director`, ...) together with the `Query` type. Every GraphQL schema must provide a `Query` type to define the available *entry points* for querying the GraphQL service. Consequently, any valid GraphQL query needs to be rooted in one of the supplied entry points. In the example, we show two possible entry

points, namely `allfilms` and `getfilm`. The former entry point allows querying the whole collection of films, whereas the latter one allows the direct access to just one film giving its id, as it is the case of the query shown in Figure 1(b).

3 A Semantic Metamodel

In this section, we define *GQL* as a semantic metamodel for GraphQL schema. The metamodel abstraction level is needed as each API has its own model (i.e., schema) defined in terms of GraphQL schema, which in turn, has its data instances, and such setting can strongly benefit from metamodeling [12]. The metamodel is formalized in RDF and, given the challenge of metamodeling, we first explain the modeling principles, after which we explain the metamodel design and its elements.

3.1 Modeling Principles

As GraphQL schema represents the data model of the data exposed by an API, we designed GQL by following the modeling principles already applied in similar cases. Our approach is inspired by the RDF Data Cube vocabulary, which is the W3C recommendation, and its extension the QB4OLAP vocabulary that aim at representing multidimensional schema to enable OLAP analysis on the SW [13].

GQL includes the following kinds of elements – class, class instance, schema property, and data property (see Figure 3). The class and schema property elements are used to define an API schema and, thus, they are instantiated only once. In RDF, the former is defined as an instance (via rdf:type) of RDFS class (i.e., rdfs:Class), while latter is an instance of RDFS property (i.e., rdf:Property), e.g., see lines 1 and 2 in Figure 2. The class instance elements are predefined instances (via rdf:type) of a previously defined class kind in the API schema, e.g., see line 3 in Figure 2. Finally, the data property elements are instantiated at the API schema level and, if data are also represented with RDF, can be used to retrieve the data instances automatically. For this reason, a data property element is defined as an instance (i.e., via rdf:type) of both the RDFS class (so that it can be instantiated at the schema level) and the subclass of RDFS property so that it can be used as the property in GQL, e.g., see lines 4 and 5 in Figure 2. At the schema level, a data property element is instantiated as an RDFS property that can be used to retrieve concrete data instances with which it is used and we provide further details on this in the following subsections.

```
1 gql:Scalar rdf:type rdfs:Class .
2 gql:hasModifier rdf:type rdf:Property .
3 gql:Int rdf:type gql:Scalar .
4 gql:Field rdf:type rdfs:Class .
5 gql:Field rdfs:subClassOf rdf:Property .
```

Fig. 2. Example of Instance Level Triples

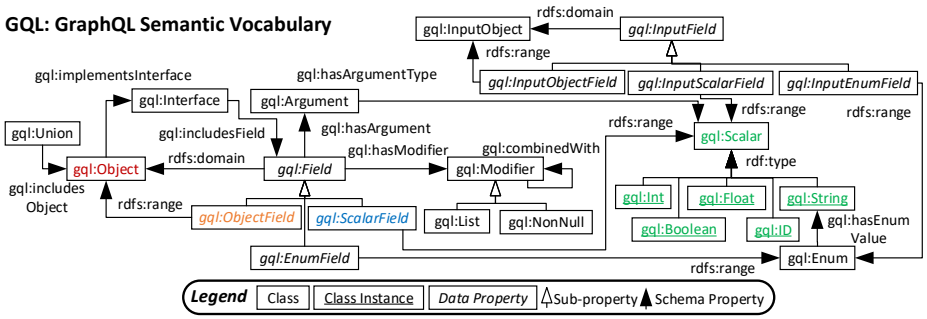

Fig. 3. GQL Vocabulary

3.2 GQL Metamodel

GraphQL Schema is organized around GraphQL types. To define our GQL metamode, we first introduce the gql:Type class as the superclass for all GraphQL types. Then, we introduce the following classes:

- gql:Scalar representing primitive values such as string.
- gql:Object organizing scalars or other objects in a single struct.
- gql:Enum representing a set of allowed scalar values.
- gql:InputObject representing complex structs to be used with field arguments (see the next subsection for more details on field arguments).
- gql:Interface representing an abstract type defining a list of fields to be implemented by an object.
- gql:Union representing a list of possible types for a certain field.
- gql:List representing a list of elements of a certain type.
- gql:NonNull denoting that a certain filed cannot take a null value.

For the last two GraphQL types above we introduce the superclass gql:Modifier as they represent modifiers over the other types.

The complete GQL metamodel (excluding gql:Type) is depicted in Figure 3.[2] In addition to classes for the previously introduced types, the gql:Argument class represents an argument for an object field that determines the value to be returned for that object field. Next, we introduce the data property elements of GQL. As in RDF, each property is directed, a property has its source and target, defined via domain (i.e., rdfs:domain) and range (i.e., rdfs:range) properties.

An object can have different fields that are represented with the gql:Field data property element. A field is always related to an object as its domain, and can relate to another object, scalar, or enum as its range. Thus we define three gql:Field sub-properties – gql:ObjectField having another object as range, gql:ScalarField having a scalar as range, and gql:EnumField having an enum as range. In the same way, we define gql:InputField with its sub-properties gql:InputObjectField, gql:InputScalarField, and gql:InputEnumField that define field for gql:InputObject.

[2] Metamodel triples: `http://www.essi.upc.edu/~jvarga/gql/gql.ttl`

Furthermore, GQL defines the following data properties. If object implements an interface, gql:implementsInterface is used to link the two. Moreover, a union of objects is defined via gql:includesObject that links a union with an object that it includes. The argument for a field is specified via gql:hasArgument linking a field with an argument. Furthermore, the argument is linked with its scalar type via gql:hasArgumentType. Each field can also be linked to a modifier via gql:hasModifier where modifiers can be mutually interlinked via gql:combinedWith, e.g., a field can be a list of non-null values. An enum can be linked to a string value via gql:hasEnumValue.

Considering the predefined class instances, the primitive values considered by GraphQL are defined as instances of gql:Scalar via rdf:type. These include gql:Int, gql:Float, gql:String, gql:Boolean and qgl:ID, and we explicitly define all of them to comply with the GraphQL specification. Note that gql:ID represents a unique identifier used to identify objects.

4 Automation

In this section, we explain how using the GQL Metamodel enables the semi-automatic generation of GraphQL schemas from a given ontology. The generation process consists of three steps:

1. *Annotation of the dataset schema* with the GQL Metamodel so that it becomes a valid instantiation of the metamodel.
2. *Automatic generation of a GraphQL schema* from the annotated ontology.
3. *Automatic generation of a GraphQL service* that exposes the generated GraphQL schema and the available data related to the annotated ontology.

The automation of step 1 is a non-trivial task that would require the definition and identification of many mappings from ontology patterns to GraphQL constructs. Nevertheless, a user familiar with the dataset (e.g., dataset publisher) requires small manual efforts to perform this task as it involves only the ontology TBox (i.e., the dataset schema) that is typically only a small part of the whole dataset (see the sequel for the details in our use case). Thus, currently step 1 is manually performed, while its automation is part of the future work.

To automate steps 2 and 3, we implemented a prototype. This prototype is a Web application with a simple interface to provide the required inputs, namely the necessary parameters to connect to the RDF triple store where the input ontology is stored. As an output, the prototype produces a GraphQL schema and a ready-to-deploy GraphQL service implementation. [3]

To illustrate the feasibility of our approach, we present a use case with the real-world dataset from the Linked Movie Database[4], which contains a total of 6,148,121 triples. This dataset contains information on 53 different concepts with a total of 222 different properties. For the purpose of the use case example, we

[3] Prototype available at `https://github.com/genesis-upc/Ontology2GraphQL`.
[4] `https://old.datahub.io/dataset/linkedmdb`

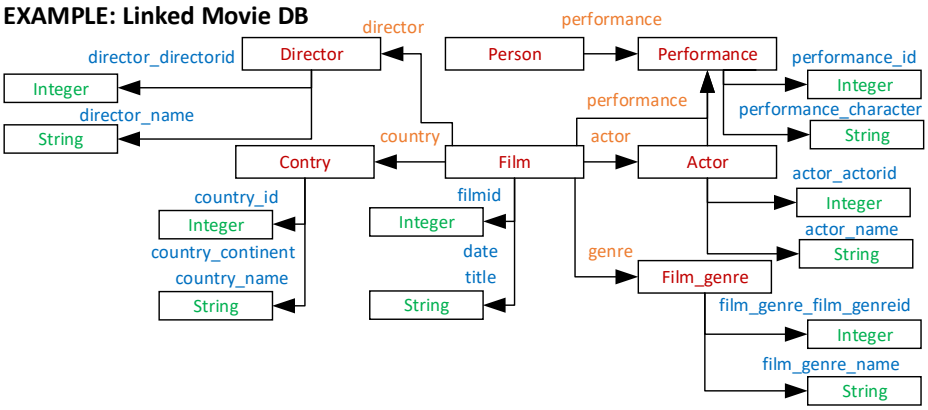

Fig. 4. Use Case dataset

have selected 7 concepts, namely Director, Person, Performance, Country, Film, Film_genre, and Actor, as it is shown in Figure 4. For these concepts, we consider a total of 20 properties whose names label the arrows in Figure 4: filmid, title, genre, etc. Notice that the range of these properties can either be a concept (e.g., Film_genre in the case of the property genre of Film) or a scalar field (e.g., String in the case of the property title of Film).

4.1 Step 1: Annotation of the Dataset

The purpose of this step it to add the necessary meta-data annotations (i.e., TBox statements) in the form of extra RDF triples so that the resulting ontology is an instance of our GQL metamodel. Accordingly, for each concept of the input ontology, we should add a triple of the form: *concept* rdf:type gql:Object

For example, Figure 5 shows the triples added for two properties (dc:title and movie:genre) that have a film (movie:film) as their domain. Notice that the triples added in lines 1-2 and 5-7 are not part of the annotation with any GQL concept but are generic meta-data annotations defining the schema for the dataset. As this schema information was missing in the dataset, we defined it based on the data instances (i.e., ABox). The semantic enrichment specific for GQL extension is presented in lines 3-4 and 8-10 in Figure 5, and this enriched schema information is necessary for the automation procedures that we describe below. In general, the number of triples to be added depends on the semantic quality of the original ontology (i.e., if the schema information is available) but should generally be small. In particular, for our use case, the resulting annotated ontology[5] consists of 91 new triples, where only 46 triples are related to GQL-specific annotations. The remaining 45 triples correspond to the definition of the ontology TBox (i.e., dataset schema) that was in missing.

[5] `https://git.io/linkedmdb.ttl`

```
dc:title rdf:type rdf:Property .
dc:title rdfs:domain movie:film .
dc:title rdf:type gql:ScalarField .
dc:title rdfs:range gql:String .
movie:genre rdf:type rdf:Property .
movie:genre rdfs:domain movie:film .
movie:genre rdfs:range movie:film_genre .
movie:genre rdf:type gql:ObjectField .
movie:genre gql:hasModifier ex:l2 .
ex:l2 rdf:type gql:List .
```

Fig. 5. Example of added triples

4.2 Step 2: Automatic Generation of a GraphQL Schema

Taking the annotated ontology, our prototype generates a GraphQL schema. For example, Figure 1(a) depicts a fragment of the GraphQL schema generated from the annotated ontology obtained in the previous subsection. [6]

The GraphQL schema generation is fully automated following the next steps:

- For each gql:Object in the ontology, a GraphQL Type is prouduced.
- For each gql:ScalarField, the corresponding scalar field is produced.
- For each gql:ObjectField, the corresponding object field is produced with its modifiers (i.e., single object or list).

In addition, our prototype also deals with the following enrichments:

- **Identifiers**. The prototype adds a scalar field `idInstance` to each Type. This field is required by the GraphQL service implementation in order to properly identify each object that can be retrieved.
- **Query entry points**. As we explained in Section 2, GraphQL schemas must define the so-called entry points. Our prototype automatically generates a pair of entry points for each Type: one to retrieve all the instances of the type and the other one to retrieve a single instance given its identifier.

4.3 Step 3: Automatic Generation of a GraphQL Service

Our prototype is also able to produce a proof-of-concept GraphQL-service implementations for the GraphQL schemas that it generates. This is again a fully automatic procedure. These GraphQL service implementations are based on the graphql-java framework[7]. Assuming that all the data is stored in an RDF triple store such as Virtuoso, resolvers are automatically implemented in terms of SPARQL queries against the triple store. Such resolvers are functions that specify how the types, fields and entry points in the schema are connected to the data sources, and they are called at runtime to fetch the corresponding data. Our

[6] The whole GraphQL schema can be found at `https://git.io/linkedmdb.graphql`.
[7] `https://github.com/graphql-java/graphql-java`

current implementation allows generating resolvers for the following GraphQL-schema elements: scalar fields, object fields, interfaces, lists of scalar fields, lists of object fields, list of interfaces, and nulls. The following elements are not yet supported: unions, input types, non-id field arguments, and enumerations.

In the context of our use case, the prototype is able to generate a GraphQL service able of answering any GraphQL query posed to the GraphQL Schema that it supports. Consequently, the resulting GraphQL service can retrieve data for 7 types with their 20 properties stored in 1,287,354 triples.

5 Related Work

Providing "a GraphQL interface for querying and serving linked data on the Web" is the aim of the tool HyperGraphQL[8]. However, in this case, the GraphQL Schema itself needs to be generated manually with additional custom (i.e., tool-specific) annotations to link the exposed data with the underlying Linked Data sources (SPARQL endpoints and or local files with RDF dumps).

GraphQL-LD [11] represents another proposal aimed at providing a bridge between GraphQL users and Linked Data publishers. In this case, the approach consists of two complementary methods: one for transforming GraphQL queries to SPARQL, and a second one for converting SPARQL results to a "GraphQL query compatible response". In the case of GraphQL-LD, no GraphQL Schema is generated nor made available to end users. In this way, one of the key foundations of the whole GraphQL approach, the GraphQL Schema, is entirely missing.

In [10] we find the so far unique proposal that has addressed the automatic generation of GraphQL schemas, to the best of our knowledge. The approach consists in defining one-to-one mappings from elements of UML Class diagrams to GraphQL schema constructs. However, in our case, the ability to have both the meta-data and the data in the same triple store allows us to generate proof-of-concept GraphQL services automatically that can expose such data.

6 Conclusions and Further Work

In this paper we have presented a semantic-based approach to generate GraphQL schemas. At the core of this approach lies the definition of a semantic metamodel for GraphQL Schema. We have also shown how with some little initial effort, e.g., manually adding just 91 triples into a 6M-triple dataset, we can automatically generate a GraphQL schema and service able of retrieving the data stored in 1,287,354 triples.

As future work, we plan to advance towards the automation of the annotation of ontologies. In a broader context, we want to integrate our approach and tool in the framework described in [1] to tackle the generation and evolution of data-intensive Web APIs.

[8] `https://www.hypergraphql.org`

Acknowledgements

This work is funded by the Spanish project TIN2016-79269-R.

References

1. Abelló, A., Ayala, C.P., Farré, C., Gómez, C., Oriol, M., Romero, O.: A data-driven approach to improve the process of data-intensive API creation and evolution. In: Proceedings of the CAiSE-Forum-DC. pp. 1–8 (2017)
2. Aranda, C.B., Hogan, A., Umbrich, J., Vandenbussche, P.: SPARQL web-querying infrastructure: Ready for action? In: 12th International Semantic Web Conference, Part II. pp. 277–293 (2013). https://doi.org/10.1007/978-3-642-41338-4_18
3. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. Int. J. Semantic Web Inf. Syst. **5**(3), 1–22 (2009). https://doi.org/10.4018/jswis.2009081901
4. Cyganiak, R., et al.: Resource description framework (RDF): Concepts and abstract syntax. `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/` (2014)
5. Daga, E., Panziera, L., Pedrinaci, C.: A basilar approach for building web apis on top of SPARQL endpoints. In: Third Workshop on Services and Applications over Linked APIs and Data. pp. 22–32 (2015)
6. Facebook, Inc.: GraphQL (June 2018), `http://facebook.github.io/graphql`
7. Groth, P.T., Loizou, A., Gray, A.J.G., Goble, C.A., Harland, L., Pettifer, S.: Api-centric linked data integration: The open PHACTS discovery platform case study. J. Web Sem. **29**, 12–18 (2014). https://doi.org/10.1016/j.websem.2014.03.003
8. Meroño-Peñuela, A., Hoekstra, R.: Automatic query-centric API for routine access to linked data. In: 16th International Semantic Web Conference, Vienna, Austria, Part II. pp. 334–349 (2017). https://doi.org/10.1007/978-3-319-68204-4_30
9. Nadal, S., Abelló, A.: Integration-Oriented Ontology. In: Encyclopedia of Big Data Technologies, pp. 1–5. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-63962-8_13-1
10. Rodríguez-Echeverria, R., Izquierdo, J.L.C., Cabot, J.: Towards a UML and IFML mapping to graphql. In: ICWE 2017 International Workshops. pp. 149–155 (2017). https://doi.org/10.1007/978-3-319-74433-9_13
11. Taelman, R., Sande, M.V., Verborgh, R.: Graphql-ld: Linked data querying with graphql. In: ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks (2018), `http://ceur-ws.org/Vol-2180/paper-65.pdf`
12. Varga, J., Romero, O., Pedersen, T.B., Thomsen, C.: Analytical metadata modeling for next generation BI systems. Journal of Systems and Software **144**, 240–254 (2018). https://doi.org/10.1016/j.jss.2018.06.039
13. Varga, J., Vaisman, A.A., Romero, O., Etcheverry, L., Pedersen, T.B., Thomsen, C.: Dimensional enrichment of statistical linked open data. J. Web Sem. **40**, 22–51 (2016). https://doi.org/10.1016/j.websem.2016.07.003