GraphQL for Archival Metadata: an Overview of the EHRI GraphQL API

Mike Bryant

February 19, 2020

GraphQL for Archival Metadata: An Overview of the EHRI GraphQL API

Mike Bryant
Department of Digital Humanities,
King's College London
United Kingdom
Email: michael.bryant@kcl.ac.uk

***Abstract*—The European Holocaust Research Infrastructure (EHRI) portal provides transnational access to archival metadata relating to the Holocaust. A GraphQL API has recently been added to the portal in order to expand access to EHRI data in structured form. The API defines a schema which mediates access to EHRI's graph-based data store, catering to both targeted and bulk metadata retrieval across a range of interrelated data types. This short paper provides an overview of the GraphQL API and illustrates a number of use-cases for the capturing of structured archival metadata.**

***Index Terms*—Archives, APIs, Structured data.**

I. Introduction

The European Holocaust Research Infrastructure (EHRI) Portal[1] offers access to information about Holocaust-related archival material held in almost 500 institutions worldwide. Like many other cultural heritage aggregation projects, EHRI seeks to make its data accessible in a form that can be effectively utilised by researchers with a diverse range of needs, including, for computationally-oriented research, sources of structured data relating to collection descriptions and other contextual information. This paper introduces the portal's GraphQL API, describing the original rationale and providing a general overview of its implementation as a database extension and the limitations this involves. We conclude with some use-cases and example queries which illustrate the scope of the API and some of its capabilities.

II. Related work

Much related work (including [1], [2]) has discussed structured data interfaces to cultural heritage and archival material, building on many years of research on bibliographical and archival metadata standards. To a significant extent, these pioneering efforts focused on interoperability, discovery, and distribution of metadata between cooperating *institutions*, rather than directly between the institutions and their user constituencies.

[1]https://portal.ehri-project.eu

More recent work has been motivated by what is often perceived as the transformative potential for data-driven innovation afforded by the availability of open, machine-readable information. Open data and open government initiatives in countries such as the U.S., the U.K., India, Ireland and Australia have, at least by association, put some pressure on other taxpayer-funded institutions to likewise make more of their information available in ways aligned with open data best practices. In the past several years a number of archival institutions, including the National Archives and Records Administration (NARA), the UK's National Archives, and the State Records of New South Wales have created APIs to some or all of their metadata catalogues, as have archival data aggregators such as Archives Portal Europe (APE) and, most notably, Europeana, who have invested significant effort in developing semantic standards and Linked Open Data (LOD) [3], [4].

Yet, in the context of EHRI and other cultural-heritage projects, there is often a presumptive leap that the end-users of the APIs are similar to those more broadly conceived as the constituents of the project at large. Edmonds and Garnett investigate the take-up and use of APIs for cultural heritage data and, through interviews with a number of practitioners, conclude that there is a mismatch between what the developers of APIs *imagine* their users need, and what those users actually want [5]. While this point may seem orthogonal to a narrower discussion concerning the technical merits of various approaches to delivering APIs, there is some overlap in the area where specific API implementations can help bridge the gap between technical and non-technical users and produce a better experience for both.

Since the EHRI portal first went live in May 2015 we have implemented a number of structured data interfaces in addition to the portal website itself. These include:

- a search interface for a subset of EHRI's archival

data, based on the JSON-API specification[2]
- an Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) server interface for archival descriptions, supporting both Dublin Core and EAD 2002 XML [6]
- integration of historically-relevant geographical data into the Wikimedia Foundation's Wikidata platform[3]

While it still demands a degree of technical knowledge of the user, the EHRI GraphQL implementation was intended to, in the words of one of Edmonds and Garnett's interviewees, "slide the APIs a bit closer to the humanists", primarily by taking advantage of its intrinsic schema-based documentation and an ecosystem of high-quality third-party tools. In addition, we wanted to expand the level of detail offered by the API to the point where it could be the primary source of data for projects focused on computationally-oriented research.[4]

While a full description of the GraphQL language is beyond the scope of this paper (see [7], for a detailed analysis) a brief overview is provided in the following section for context.

III. Overview of the GraphQL language

The GraphQL query language was developed internally at Facebook from 2012 and announced publicly in 2015, with the release of a draft language specification.[5] The language was conceived with several broad goals:

- to reduce overhead of data transfer relative to REST-like web service models, in terms of both the amount of data transferred unnecessarily, and the number of separate queries required to do it
- reduce the potential for errors caused by invalid queries on the part of the client
- support an evolving data model without the need for API versioning

These goals were largely driven by Facebook's mobile applications, which benefit from efficiency in both power and data usage, and have many versions in use concurrently.

GraphQL differs significantly from both low-level database query language specifications like SPARQL, and high-level resource-oriented web-service protocols such as OAI-PMH, operating on a level of abstraction between the two. A key distinguishing feature is a schema which allows the definition of complex data types that represent domain objects within an implementation system. These complex data types may have attributes consisting of not just scalar and array values, but other complex types, forming a directed object graph.

An example schema, simplified for clarity, is shown in listing 1 describing `DocumentaryUnit` and `Repository` types. and a root `Query` type which defines top-level query items, which serve as entry points into the object graph:

```
type Repository {
  id: ID!
  name: String!
  documentaryUnits: [
     DocumentaryUnit]
  itemCount: Int!
  address: Address
}

type Address {
    street: String
    city: String
}

type DocumentaryUnit {
  id: ID!
  repository: Repository!
  parent: DocumentaryUnit
  ancestors: [DocumentaryUnit]
  children: [DocumentaryUnit]
  descriptions: [Description]
}

type Description {
  id: ID!
  languageCode: String!
  scopeAndContent: String
}

type Query {
  repository(id: ID!): Repository
  repositories: [Repository]
  documentaryUnit(id: ID!):
     DocumentaryUnit
  documentaryUnits: [
     DocumentaryUnits]
}
```

Listing 1. A simple schema describing two archival domain objects, `DocumentaryUnit` and `Repository`, their related `Address` and `Description` types, and a root-level `Query` type defining entry points into the object graph.

When a user queries a GraphQL system they do so by listing the scalar attributes they require from the object graph, using brace nesting to span object types through named relationships defined by the schema.

[2]http://jsonapi.org

[3]https://www.wikidata.org

[4]For example, comparing structured metadata regarding the time periods covered by an archival collection with those gleaned from an analysis of its unstructured textual description.

[5]https://facebook.github.io/graphql/

Unlike SQL or SPARQL, where queries are a form of relational calculus, returning tuples as a subselection of the query space, GraphQL response data is returned in a hierarchical form that corresponds directly to the structure of the originating query.

A simple example query is shown in listing 2:

```
query example1 {
  repository(id: "us-005578") {
    name
    address {
      city
    }
    itemCount
  }
}
```

Listing 2. An example GraphQL request fetching a domain object (the `Repository` type) by unique ID reference, and `city` property of its related `Address` type.

```
{
  "data" {
    "repository" {
      "name": "USHMM",
      "address": {
        "city": "Washington␣D.C."
      }
      "itemCount": 9574
    }
  }
}
```

Listing 3. A typical response to the request shown in listing 2.

In this example, the id argument is provided to retrieve a Repository object with ID "us-005578", requesting the name attribute, the city attribute of its related address, and the number of archival descriptions it contains.

The above examples omit many features of GraphQL (such as Interface, Union, and Enum types, field aliases, fragments, and graph mutations) but the interested reader can learn more via the referenced papers and the draft specification.

Since GraphQL is just a specification, implementing a compatible system is the responsibility of the data provider, typically by building upon middleware that facilitates schema definition and validation, whilst delegating actual data retrieval operations to the implementer in a manner agnostic to the specific storage mechanisms used.

IV. The EHRI Portal GraphQL Implementation

The portal's GraphQL implementation is an interface to EHRI's metadata persistence layer, described in [8, Section 4]. The persistence layer, which handles validation, access control, and audit logging for the various types of information managed by the the portal, is likewise based on a Neo4j graph database, one of the most popular implementations of the property graph model [9]. Building the GraphQL API on top of EHRI's persistence layer means that GraphQL query execution can inherit the same set of access control and permission rules as the portal itself, preventing access to data that is, for example, user-generated and private, or restricted due to other concerns.

Because query execution runs *within* the Neo4j database server environment directly, however, it can exploit the close conceptual similarities between GraphQL schemas and the property graph model.[6] As query execution takes place, scalar object attributes fetched in a GraphQL query can be mapped directly to node properties on the underlying graph, whilst object-to-object relationships typically correspond to graph relationships and are navigated using Neo4j's native index-free traversals.

While this database-native approach has limitations (discussed below) it is simple and efficient compared to an architecture involving separate networked services, and facilitates features such as streaming responses, discussed below in section VII-D.

V. Advantages of the GraphQL approach

We perceive a number of advantages to the GraphQL API, both from the perspective of the user and the data provider:

A. Documentation and integration with third-party tools

Providing users of an API with high-quality error messages and feedback about potentially malformed requests adds considerably to the challenge - and overall effort - of developing domain-specific structured data interfaces. At the opposite end of the spectrum, while the generic SPARQL query language can generally (depending on the implementation) provide a degree of syntax-level feedback, higher-level validation that is able to reason about queries on a logical or semantic level remains uncommon, in part due to the open world assumption [10]. With its closed-world, domain-specific schemas, however, GraphQL API implementations can leverage open-source libraries for parsing and schema validation that provide effective error handling and feedback for user queries on both the syntactic and semantic level, with minimal effort on behalf of the implementer.

[6]The property graph, in which both nodes and relationships within a graph database can hold an arbitrary number of (usually scalar) key/value attributes, has a close conceptual overlap to GraphQL, and makes similar closed-world assumptions.

Moreover, because a GraphQL schema is itself able to be interrogated via standard GraphQL queries, third-party tools can interact with APIs for the purpose of diagnostics, introspection, or automatically generating documentation describing its intended use. These include the Graph*i*QL interface[7], which allows users to explore the schema interactively in their browser with direct feedback, auto-completion, syntax highlighting, and inline documentation.

B. The schema as a mediated view on an internal data model

Data providers are typically required to maintain both internal and external data models, with the latter representing a simplified, mediated view of the former. There are a number of reasons for this: internal data is often stored in legacy systems that are themselves the result of evolution in working practices and requirements over long time scales. Internal data models typically include considerably more administrative metadata contributing to resource maintenance and the institutional audit trail, and often do not adhere rigorously to, or pre-date, applicable metadata standards. Additionally, internal data may not be fit for public consumption in other ways, and may be partial, fragmentary, incompletely described, or privacy-sensitive.

It is rare, then, that data providers can "open up" their raw data to users without a considerable degree of mediation, simplification, and contextualisation. GraphQL, being agnostic to specific storage and retrieval technologies, is well suited to a bridging role, providing a coherent interface to legacy systems and allowing data providers to expand or evolve their outward-facing schema in a backwards- and forwards-compatible manner without explicit API versioning. In EHRI's case, the internal and external data models have a high degree of correspondence, yet the public schema is still a considerably more simple and limited interface, with a number of affordences for convenience and ease of navigation.

VI. Limitations of the EHRI implementation

While we believe the GraphQL API is a valuable addition to the suite of structured data interfaces available on the EHRI portal it does, as currently implemented, have a number of limitations.

A. API scope and expandability

Implementing GraphQL as an extension of EHRI's Neo4j-based metadata storage system prevents us from easily integrating, for example, full-text search capabilities handled by a dedicated search engine (in EHRI's case, Apache Solr.) More broadly, implementing GraphQL in this manner prevents us from expanding it into a single endpoint that acts as a facade to multiple heterogeneous backend systems, a scenario for which the language is well suited due to its retrieval-agnostic nature. We believe at this point however that the benefits offered by direct database integration outweigh this downside, in combination with the other APIs offered the EHRI portal.

[7]Deployed for the EHRI Portal at: https://portal.ehri-project.eu/api/graphql/ui

B. Sparse data and lack of filtering

An additional limitation regards the handling of sparse data. Since GraphQL object traversals behave like LEFT JOINs in SQL (or SPARQL's OPTIONAL patterns), with absent data present in the response as a `NULL` value, the task of filtering data at the top of the object hierarchy when some condition on the child (or deeper) is unmet falls to the client. For example, it is not possible to retrieve a particular property for a given data type - for example, the `scopeAndContent` field for archival descriptions - whilst omitting from the query response those descriptions where `scopeAndContent` field is missing. For EHRI's archival data, where fields defined in the conceptual metadata standards are very commonly absent in item descriptions this can result in a poor "signal-to-noise ratio" in the response data that obliges users to perform secondary filtering themselves.

VII. Use-cases

The section discusses two use-cases related to the archival domain.

A. Retrieval of contextual data

The GraphQL API was implemented in order to make more easily accessible data that is difficult to expose or fetch via other means, such as a REST-style resource-oriented API. For archival data, the display of items *in context* - that is, in a manner that emphasises their place in the archival hierarchy consisting of holding institution, fonds, and (for an integration project such as EHRI) a varying number of intermediate levels of description - complicates the implementation of resource-oriented data retrieval methods due to the *potential* size of this object graph. For an archival description nested 10 levels deep, fetching the accompanying context data (9 parent items) requires either making many individual resource requests or including the full context data in the original payload, where much of it may be superfluous most of the time. The problem of under-fetching or over-fetching of data with resource-oriented APIs, one of the issues GraphQL was designed to solve, is therefore highly relevant for archival data where contextual information is so important.

With EHRI's GraphQL API, delivering the ability to navigate and retrieve contextual data with greater precision, flexibility, and succinctness was an important goal. Listing 4 provides an example, whereby the retrieval of an item by ID includes the language-specific name of its repository and parent item(s), along with the number of items they each contain:

```
fragment data on Described {
  id
  description(languageCode: "eng")
      {
    name
  }
}

query contextExample($id: ID!) {
  DocumentaryUnit(id: $id) {
    repository {
      ...data
      itemCount
    }
    ancestors {
      ...data
      itemCount
    }
    ...data
  }
}
```

Listing 4. An EHRI GraphQL query for a documentary unit item and its archival context. The query accepts a single parameter: the ID of the requested item. The `ancestors` field will retrieve a JSON array value containing the item's parent data, while the `repository` field fetches it's institutional context. This example also shows the use of a GraphQL feature called *fragments*, where we exploit polymorphism in the schema between items that have multi-lingual descriptions, fetching just the `id` and English `name` in each case.

When composing the GraphQL query, the client can request as much or as little of the item's context - retrieved by traversing its object graph - as necessary for a given situation. To account for the varying degrees of nesting present in a hierarchy of archival descriptions the schema makes available the `ancestors` field, which provides a reified list representation of the internally recursive *parent-to-parent* traversal.

B. Extracting relationships and interconnected descriptions

Since part of EHRI's mission is connecting collections held across different institutions (in particular, copy collections and those with shared provenance), exposing such relationships, along with their contextual metadata, is a key part of the GraphQL API. Relationships, or "links" in the terminology of the GraphQL API, are first-class data items that can be retrieved individually or as a collection via a top-level GraphQL field. The API also exposes links that connect individual items from the items themselves, in addition to materialising these connections via a `relations` field which manifests as an object through which both the context (type of relationship and the period for which it was active) and the connected item can be explored:

```
query relationsExample($id: ID!) {
  DocumentaryUnit(id: $id) {
    related {
      item {
        id
        type
      }
      context {
        description
        dates {
          startDate
          endDate
        }
      }
    }
  }
}
```

Listing 5. A query which lists related items for a given archival unit, along with the context (description, dates, if applicable) of the relationship.

C. Extracting administrative metadata

The provenance of archival descriptions - when they were written and by whom - can provide valuable context to the researcher in guiding their discovery and use of sources. When integrating collection descriptions from many sources, EHRI tries to preserve as much of the original administrative metadata as possible within the constraints of ISAD(G), but also adds another layer of digital provenance reflecting the management operations undertaken by EHRI itself. This "born-digital" metadata can be minimal, in cases where EHRI has harvested collection descriptions as structured data from partner institutions, or extensive, such as when EHRI has created completely new descriptions (or other types of digital record) from scratch.

Ongoing acts of metadata curation are captured within the EHRI portal as a stream of "system events" (named to distinguish them from the historical kind), available as a data field on all principal item types.

```
query eventsExample($id: ID!) {
  DocumentaryUnit(id: $id) {
        systemEvents {
          eventType
          timestamp
          logMessage
        }
      }
    }
```

Listing 6. A query which extracts metadata about the digital provenance of items within the EHRI portal, such as when the were created or updated.

```
    {
      "data" {
        "DocumentaryUnit" {
          "systemEvents": [
            {
                "eventType": "updated",
                "timestamp": "
                   2017-11-10T14
                   :14:36.442Z",
                "logMessage": "Expanded
                   ␣biographical␣
                   history"
            },
            ...
          ]
        }
      }
    }
```

Listing 7. A typical (abridged) response to the request shown in listing 6.

D. Bulk data retrieval

Both EHRI's REST-style search API and the portal website itself restrict the user to paginated browsing or search operations which return a fixed amount of data per request, a restriction imposed both for technical and user-interface concerns. While often necessary, mediating a user's access to the data in this manner, via the imperfect mechanisms of search and browse, nonetheless presents a barrier to conducting research on the dataset as a whole, akin to viewing a landscape through a keyhole. An additional motivation behind the development of the GraphQL API, therefore, was to provide the means to extract bulk data with minimal friction imposed by the API itself.

The GraphQL API accomplishes this by executing the user's query and streaming the in-progress result tree back to them on-the-fly, as it is constructed by traversing the Neo4j database. This allows clients to retrieve a holistic view of EHRI data - for example, the titles or textual content of over 200,000 archival descriptions - in a single query.

On-the-fly result generation is practical in large part because we are able to restrict the schema's data-retrieval operations to those with predictable space complexity, such as index iteration and graph traversals, whilst avoiding potentially expensive sorting or aggregation routines. While these limitations reduce the expressiveness of the available queries, especially compared to SPARQL or SQL, and push the responsibility for complex aggregation or filtering onto the user, we believe it is a worthwhile trade-off to avoid mandatory pagination or resumption tokens when requesting bulk data.

VIII. Summary

This paper has described the rationale behind the EHRI project's GraphQL API, given a brief overview of GraphQL itself, and presented EHRI's implementation of the API as a mediated view of its Neo4j-based data store. We have described the main benefits and limitations we see in the context of an API tailored towards users wishing to extract structured data for research purposes, and given examples of how the API facilitates access to metadata that provides enriched context to archival descriptions via the relationships between them and their digital provenance.

References

[1] Carl Lagoze and Herbert Van de Sompel. The Open Archives Initiative: Building a Low-barrier Interoperability Framework. In *Proceedings of the 1st ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '01, pages 54–62, New York, NY, USA, 2001. ACM.

[2] Cliff Lynch, Savas Parastatidis, Neil Jacobs, Herbert Van de Sompel, and Carl Lagoze. The OAI-ORE Effort: Progress, Challenges, Synergies. In *Proceedings of the 7th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '07, pages 80–80, New York, NY, USA, 2007. ACM.

[3] Antoine Isaac and Bernhard Haslhofer. Europeana Linked Open Data – data.europeana.eu. *Semantic Web*, 4(3):291–297, January 2013.

[4] Cesare Concordia, Stefan Gradmann, and Sjoerd Siebinga. Not just another portal, not just another digital library: A portrait of Europeana as an application program interface. *IFLA Journal*, 36(1):61–69, March 2010.

[5] Jennifer Edmond and Vicky Garnett. APIs and Researchers: The Emperor's New Clothes? *International Journal of Digital Curation*, 10(1):287–297, May 2015.

[6] Herbert van de Sompel, Michael L. Nelson, Carl Lagoze, and Simeon Warner. Resource harvesting within the OAI-PMH framework, December 2004.

[7] Olaf Hartig and Jorge Pérez. An Initial Analysis of Facebook's GraphQL Language. Preprint available on author's website, 2017.

[8] Tobias Blanke, Michael Bryant, Michal Frankl, Conny Kristel, Reto Speck, Veerle Vanden Daelen, and René Van Horik. The european holocaust research infrastructure portal. *J. Comput. Cult. Herit.*, 10(1):1:1–1:18, January 2017.

[9] Marko A. Rodriguez and Peter Neubauer. The graph traversal pattern. *CoRR*, abs/1004.1001, 2010.

[10] Jesús M. Almendros-Jiménez, Antonio Becerra-Terón, and Alfredo Cuzzocrea. Syntactic and semantic validation of sparql queries. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 349–352, New York, NY, USA, 2017. ACM.