Data Federation for a Project Management Solution through a GraphQL Middleware

Diana Anca Vasiliev
Babeş-Bolyai University
Cluj-Napoca, Romania dianaanca94@gmail.com

Ana-Maria Ghiran
Babeş-Bolyai University
Cluj-Napoca, Romania ana.ghiran@ubbcluj.ro

Robert Andrei Buchmann
Babeş-Bolyai University
Cluj-Napoca, Romania robert.buchmann@ubbcluj.ro

Abstract

Data integration has been under research from the early stages of data management solutions, but it becomes subject to more intense scrutiny with the expanding opportunities opened by Web interoperability technologies. In distributed environments, there are many scenarios in which data from different sources must be accessed through a joint approach, overcoming heterogeneity of data location and format. This paper reports on design decisions with a GraphQL middleware to enable consistent access and delivery of data spanning over heterogeneous sources, in a project management context. Compared to other service-based architectures employing a REST-based API, a GraphQL solution mitigates issues such as overfetching or underfetching when retrieving data in a client application. As proof-of-concept, a prototype of a project management module in an academic management system was implemented to integrate data entities available in legacy data sources.

Keywords: data integration, data federation, GraphQL, Apollo, project management.

1. Introduction

The connectedness of distributed data that is available nowadays for legacy reasons demands for integration across heterogeneous sources, which are distinct in their original purpose, business rules and context. In order to solve the heterogeneity issues, there are two traditional approaches that can be employed:

- one based on data transformations in order to store them into a consolidated repository (physically merging data);
- one where data sources keep their data and an integration layer is developed to expose it in a consolidated view (logically merging data).

These broad strategies quickly expanded as vendors of data integration tools are offering a variety of technological instantiations. Examples of these integration tools are spanning from those encountered in the ETL (extract-transform-load) area to those that are employing a service-based architecture or to data fabrics and middleware enabled by graph-based solutions. Data fabrics and knowledge graphs have been identified by Gartner [9] as technological trends in supporting data management frameworks that facilitate not only a uniform query approach, but also finding relationships across diverse data assets. However, they still reside on a common data format like RDF (Resource Description Framework) [19] to which data sources must be converted and semantically lifted. In cases that involve both internal and external platforms, depending on the number of participants,

it might be challenging to find the tools for data conversion or to impose compliance to a uniform data format, even if it is a standard one. For such situations, a middleware solution in the form of a service-based architecture [5] can be a better data fetching approach.

In this paper, in answer to the challenge of integrating legacy project management data in a novel integrated system, we opted for a GraphQL [6] middleware solution to provide consistent access over multiple isolated and diversely formatted data sources. This enables a decentralized architecture over the data sources without concerns regarding investments in compatible solutions or a conversion tool for transforming data, while leveraging access to a wide spectrum of data types.

A data integration system consists of three essential elements: a global schema, a set of source schemas, which include schemas of all resources, and a mapping between the global schema and the source schemas, along with their corresponding relationships [4]. A GraphQL solution is an appropriate interpretation for the above definition. By using GraphQL, the business domain can be modelled as a graph schema prescribing different types of nodes with connections one to each other.

In comparison with other middleware techniques, the main benefit of using a GraphQL API for data integration purposes is that, in order to retrieve the needed data, it is sufficient to send a single query that includes data filters and joins recognized by the schema prescribed by the GraphQL Server. Therefore, GraphQL provides a single endpoint used for "resource fetching, creation and modification" [13] allowing for a dynamic and loose coupling. Moreover, it enforces a data schema and in case of errors a detailed message that refers to the exact incorrect part of the query is provided. This means that the clients define the exact and nested data requirements in typed queries, that are resolved by the server against multiple backend services or data sources - which can be different databases, object storages, or APIs. Clients will receive only the needed data in a single request capable of expressing virtual entity joins, and in this way data excess does not overload the network [22]. By tailoring the query according to information defined and exposed by a schema documentation, the user can define custom requests to retrieve the accurate data for various needs [7]. In terms of performance and usability, GraphQL has great advantages when compared to the REST API paradigm - by allowing the users to specify exactly what data they need in the GraphQL query, the request-response time can be smaller.

The remainder of the paper is structured as follows: Section 2 formulates the problem statement and provides an overview of the proposed solution indicating the technology architecture and a conceptual data model, Section 3 describes furthermore the employed research method while Section 4 presents the implementation details in a running example. The paper ends with related works and conclusions.

2. Problem Statement and Solution Overview

In the face of diverse competition and emancipated consumers, companies are striving to base their decisions on data-driven analytics, which solicits a data integration solution. Most of the time, this requires collaboration between both internal information systems and other parties' components - i.e., it is no longer enough to perform analysis only on company's own data. In these cases, data remains unaltered and distributed, consequently it does not accumulate in a large data warehouse but it's made available in a service-based architecture.

REST [8] has been considered as a standard technical solution that allowed remote data access through HTTP by exposing resources identified by URLs through API endpoints. For each data requirement, the REST architect must conceive in advance the combination of HTTP method, parameters and route URL (custom endpoints that address specific needs for the client), but this has the disadvantage that the number of endpoints can grow significantly, which means increased burden for data management systems especially when client application requirements evolve over time.

Many of today's applications are consuming large amounts of data, therefore there is a call for more efficient management of data. This problem, that initially affected only giant companies like Facebook or Netflix, who had to deal with almost endless user data feeds,

is now becoming a challenge for more common data-centric organizations, including the host organization where our research is being performed on legacy academic management systems, out of which this paper focuses on a GraphQL-driven project management app.

GraphQL [6] enables servers to use a schema to describe how data is organized and also allows client applications to specify exactly the required data fields, data filters and entity joins therefore it engages a more efficient communication between the client and the server, compared to REST. Through this work, we advocate a recipe for implementing both sides of a GraphQL-based data integration architecture - a technical contribution that may help developers to implement more robust data integration applications.

The work followed the Design Science methodology [20] in order to implement a proof-of-concept that consists of a GraphQL middleware to gather data from MySQL databases and a JSON server, exposing as a result a virtual federated data graph governed by a unified schema across the existing data sources. The result is a gateway that exposes a federated data graph, which provides a unified interface for extracting data from the available legacy sources without changing them, hence reducing the time spent searching for information across different systems.

Figure 1 shows an overview of the proof-of-concept architecture (left) and also the technological choice (Apollo Platform [1]) underlying the GraphQL implementation (right). Apollo consists of two main components: an Apollo Server is actually the GraphQL Middleware while the Apollo Client will be used in order to run queries against the Apollo Server and obtain the results as query-specific Swift types for the front-end Mac OS application.

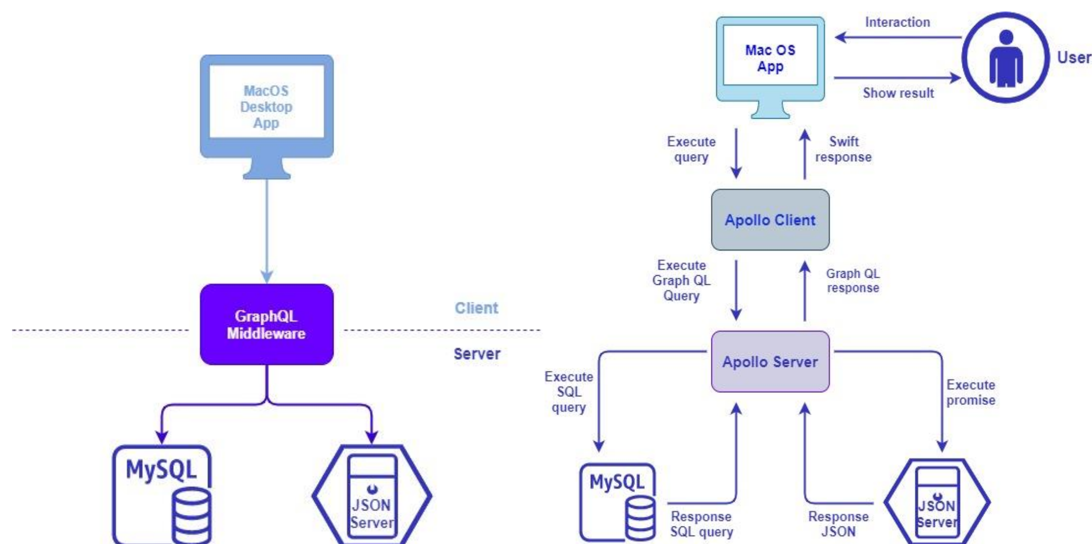

Fig. 1. Apollo-based solution architecture

With GraphQL, a unifying schema is defined to capture the structure of data spanning across multiple data sources, thus exposing a graph shape for the integrated data (virtual) repository. The unifying schema specifies all the types and fields available in the data graph - it represents a collection of type definitions that will determine the queries that can be executed against the data. A declarative model helps creating a GraphQL integrated API that can be used across several data providers. The data sources can be easily added, deleted or changed, without changing the client's perspective.

However, the uptake of GraphQL is still slow because on the server side it still requires more effort than setting up a REST API, as it needs to manage an increased burden - e.g. to ensure that GraphQL queries do not result in nested (cyclic) operations that can bring down the server or enable DDoS attacks.

3. Engineering Methodology

The Design Science (DS) methodology [20] was adopted to address the pragmatic problem of integrating legacy data sources in a project management context. DS is preoccupied with the design and the investigation of the artifacts in a specific context, with the main goal of solving real-life problems, while following a disciplined engineering cycle potentially informing design theories.

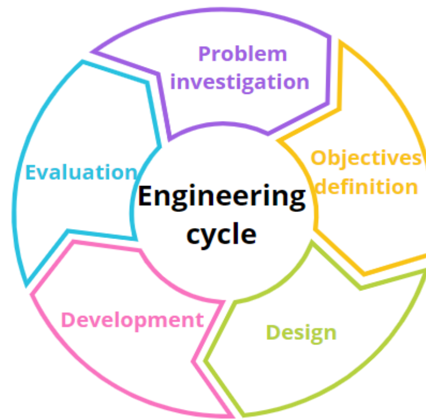

Fig. 2. Engineering cycle diagram

The DS methodology encompasses an engineering cycle as shown in Figure 2. In the *investigation step* a detailed understanding of the encountered problem was achieved – the existence of heterogeneous data sources that should be used for a project management app instead of recreating the data in a convenient redundant repository. In the *objectives definition* part, specific requirements are gathered, in this case the need to realize the app to serve some basic project management tasks. The *design phase* implies the design decisions on how the integration is to be achieved (to be detailed in the next section). In the *development part* the prototype is developed. The *evaluation phase* assesses artifact qualities according to a relevant set of criteria – a taxonomy being suggested in [15]. Hereby reported evaluation is limited to the fit with requirements, with more recent experimentation and comparison with alternative implementations being discussed in [16].

4. Proof of concept

This section presents a running example for a case-based project management scenario: an app needed in an organization leading multiple projects with numerous tasks corresponding to each project and human resources that have been traced in a legacy system. The employees' information and details are accessible through a JSON-based traditional API, the information corresponding to a specific project is mostly kept on a MySQL database. Project tasks are detailed more granularly in an additional MySQL database, created for administrative compliance reasons in relation to accounting.

One key requirement is that an employee can be assigned to a specific project and afterwards can create their own tasks to work on. Also, each employee can have a specific role on a project such as a developer, a project manager, a tester and so on. Usually, the objectives might also describe the costs involved, the risks that might appear along the way, the success metrics and the time that is necessary in order to complete the proposed artifact. In the design phase, the architecture shown in Figure 1 was proposed, with granular design activities on the front-end (MacOS) and on the back-end data schemas.

In the following, we will focus on the design and development phase.

4.1. Design details

The case-relevant data comes from various sources, those being two MySQL databases and a JSON data service based on LowDB [12].

- The *Projects Database* contains the entities "project" and "client". The table

"project" has the following fields with their corresponding data types: id (int), project_name (varchar), start_date (date), end_date (date), project_descr (varchar), employee_id (int), budget (int), client_id (int). The table "client" has the following fields: id (int) – primary key, name (varchar), address (varchar), details (varchar) email (varchar). The Foreign Key that joins table "project" with table "client" is the field "client_id".

- The *Tasks Database:* contains the tables "task" and "activity". The table "task" is composed of: id (int) set as primary key, task_name (varchar), priority (int), description (varchar), start_date (date), end_date (date), project_id (int), status (varchar), employee_id (int). The table "activity" includes the following fields with their corresponding data types: id (int) – primary key, activity_name (varchar), task_id (int), priority (int), description (varchar), start_date (date), end_date (date), status (varchar), employee_id (int). The Foreign Key that joins table "task" with table "activity" is the field "task_id".

The data served by the JSON data service is structured as follows:

- Employee object with attributes id, employee_code, employee_name, account_id, role_id;
- User_account object with attributes id, email, firstname, lastname;
- Role with attributes id and role_name.

The MySQL and JSON data sources are exposed as a single unifying data graph via the *Apollo GraphQL Server*. From this federated data graph, the *Apollo GraphQL Client* uses the unified query interface in order to obtain any combination of data from the backend data sources. The engineering recipe is further desccribed for both sides.

The GraphQL Server

In order to design the conceptual model of the data graph, the first step is to locate the foreign keys used to express the joins between data entities, which will become relationships between the graph nodes. The following adjustments and mappings prepare for this:

- In the Projects Database, the foreign key that links the "project" table with the "client" table ("client_id" field) becomes the "HAS_CLIENT" relationship;
- In the Tasks Database, the foreign key that joins the "task" table with the "activity" table ("task_id" field) becomes the "HAS_ACTIVITY" relationship;
- In the JSON data service, the links between the "employee" table and the tables "user_account" and "role", are provided by the conventional foreign keys "user_account_id" and "role_id" which become the relationships "HAS_ACC" and "HAS_ROLE";

Furthermore, the relationships across the data sources must be made explicit: the foreign key between table "project" and table "task" becomes the relationship "HAS_TASK", the foreign key between the table "employee" and tables "task" and "activity" becomes the relationships "WORKS_ON_TASK" and "WORKS_ON_ACTIVITY".

The conceptual model diagram of the virtual federated data graph with the nodes and their relationships can be consulted in Figure 3.

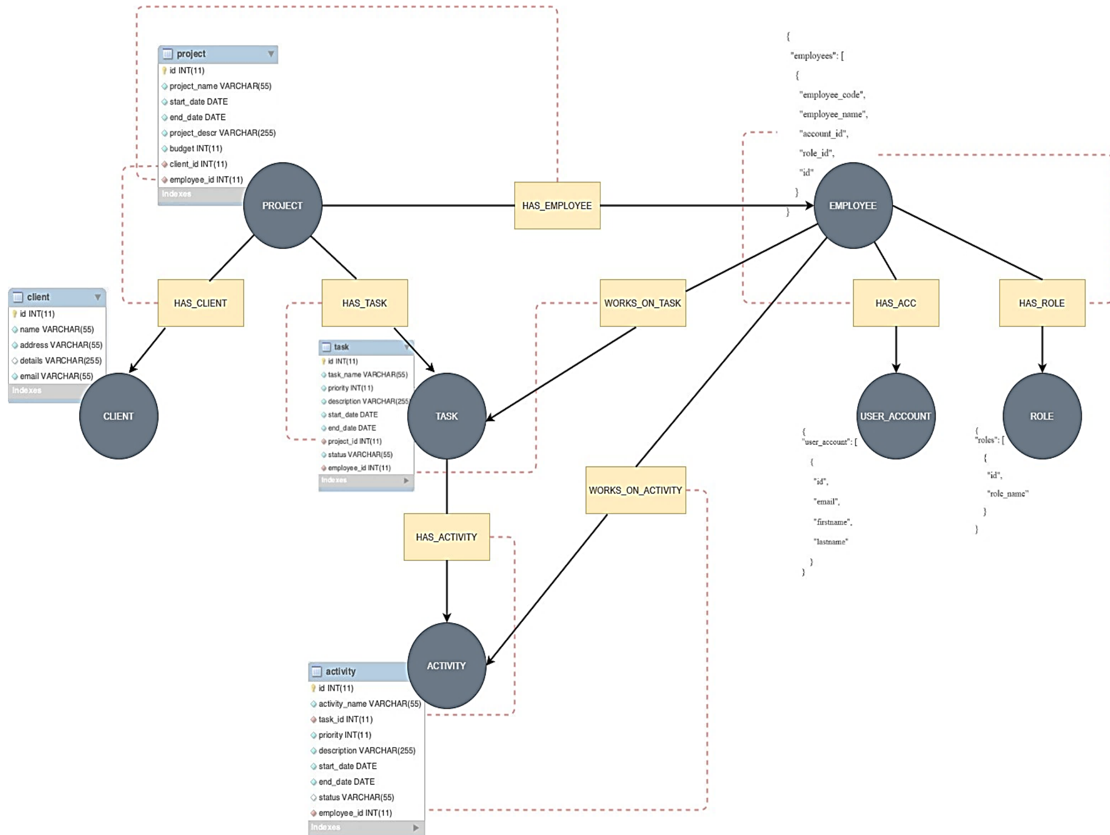

Fig. 3. Conceptual representation of the virtual federated data graph

The GraphQL Client

The MacOS front-end application benefits from the client-side of the Apollo framework in order to initiate queries and mutations on the heterogeneous data sources.

The data structures are located in an API.swift file that is generated by Apollo Codegen, which performs conversions of the application-specific GraphQL queries and mutations into Swift-formatted queries and mutations. This conversion is constrained by the GraphQL schema prescribed by the GraphQL server and accessed by Apollo Codegen, meaning that the application-defined queries and mutations will comply with the schema. This interaction is presented in Figure 4.

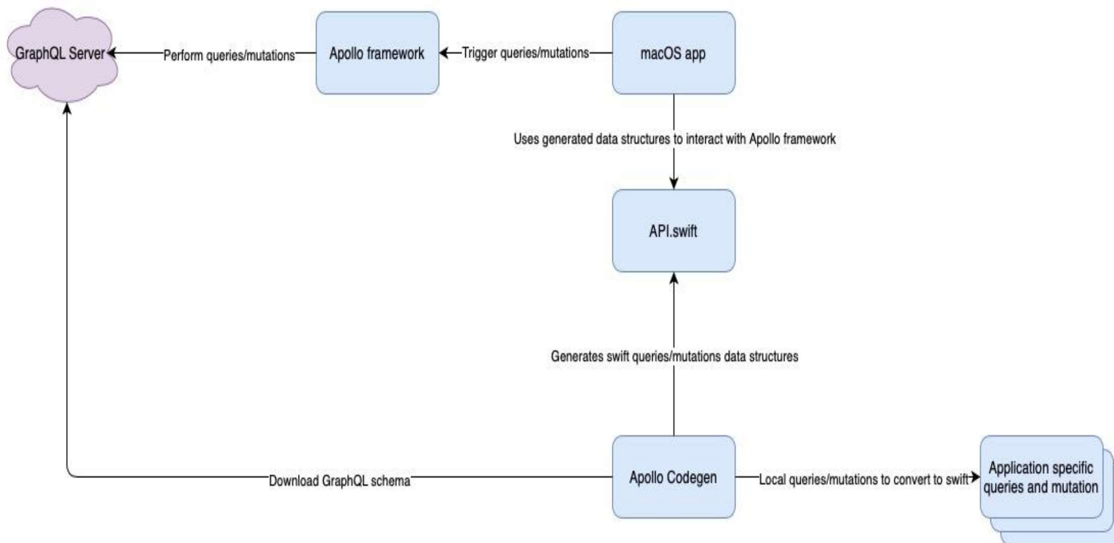

Fig. 4. Interactions of the Apollo Components

4.2. Implementation details

The GraphQL Server (Middleware)

In order to exploit the Apollo Server, dependencies such as “apollo-server” and “graphql”, are needed – these are the core GraphQL schema management tool and the Apollo framework customization. They were implemented in a Node.js project. The first step was to implement the schema by defining, in GraphQL terminology, the *types*, *queries* and *mutations*. The schema describes the whole structure and will form a federated data graph.

The *type definitions* are composed of *object types* for every needed entity, *query definitions* and *mutation definitions* with their necessary *input types.* The *object type* definition includes the necessary fields that correspond to the ones defined in the actual database schema and it can include other object types as fields. The *query type* defines the entry points for the read operations, whereas the *mutation type* defines entry points for write operations. The *input types* represent special object types that allow passing arguments to queries and mutations.

For example, the *object type* for “Projects” will include fields corresponding to the ones defined in the MySQL database (id, project_name, start_date, end_date, project_descr, budget), but also the employee that works on the project, the client that owns the project results and a list of assigned tasks. Figure 5 shows the object type for the project table, with its corresponding *query and its mutation definition* based on the defined *input types.*

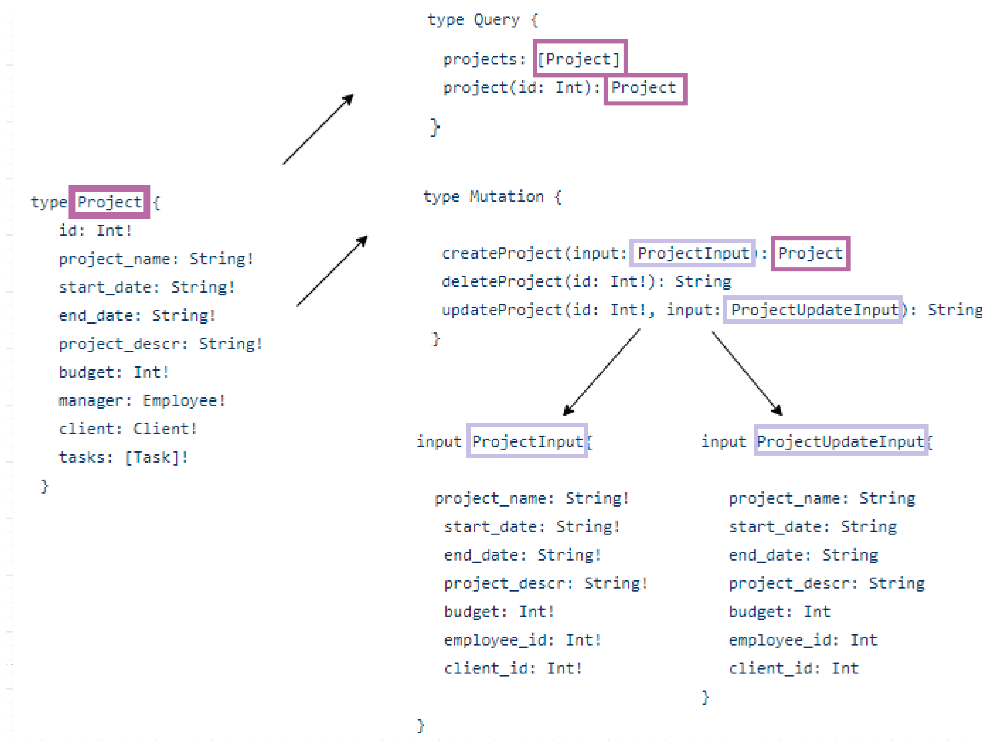

Fig. 5. Object type, query and mutation definitions with input types for Project

The *query type* for the project model will return a list of projects or a particular project based on a specified id.

The project's *mutation definition* type has 2 input types: “ProjectInput” - used for a create mutation where all the fields are mandatory, and “ProjectUpdateInput” - used for an update mutation where all the fields are optional, because the user might only want to modify and change the information from a specific field. Similar input types is used for all the models.

Inside the *mutation type***,** all the available mutations for a particular object type are defined. For the project object type, the defined mutations are createProject, updateProject and deleteProject. Similar mutation definitions are prescribed for the other object types as

well.

After defining the types, queries and mutation, the next step was to create the so-called *resolvers***,** the actual middleware mechanisms through which the data will be transmitted. For each query and mutation that was declared in typedef, a resolver is needed to obtain the data from the real providers and deliver it to the app.

The requests for data are based on the popular Promise mechanism - asynchronous code acting as a proxy for a value that will eventually become available.

First, data from the JSON service is fetched and for this, according to the REST routes already available there for CRUD operations.

The employee object is the most complex object in the JSON server, as it incorporates data from the user_account and role objects, having the "account_id" and "role_id" fields with the same values as in the corresponding objects. Hence, in order to link the data from the user_account and role objects to the employee object, a resolver was needed for the employee. This resolver creates a link between the user_account node, the role node and the employee node. The employee resolver is composed of GET requests based on the ids of each user_account and role objects, mapped to their corresponding fields in the employee object (account_id and role_id), as it is represented in Figure 6.

```
const userAccount_by_id_request = RequestById(Paths.UserAccount, "GET")

const roles_by_id_request = RequestById(Paths.Roles, "GET")

  Employee: {
    account: async parent => {
     return await request(userAccount_by_id_request(parent.account_id))
    },
    role: async parent => {
     return await request(roles_by_id_request(parent.role_id))
    }
  },
```

Fig. 6. Requests used to build the Employee resolver

After the employee resolver creation, in order to query the employee object with its new structure including the account_id and role_id, two query resolvers were created. One performs a GET request that fetches a list of the employee data, and the other fetches a particular employee's data, based on id. Query resolvers are presented in Figure 7.

```
const employee_request = Request(Paths.Employees)
const employee_by_id_request = RequestById(Paths.Employees, "GET")

    Query: {
        employees: async () => {
           return await request(employee_request)
        },
        employee: async (parent, {id}) => {
          return await request(employee_by_id_request(id))
        },
```

Fig. 7. Queries for employee data

Besides the Read operations implemented in queries, Create, Update and Delete operations in the employee data are prescribed by mutation types: e.g., the Delete operation is based on a DELETE request that will be accessed through the deleteEmployee mutation, see Figure 8.

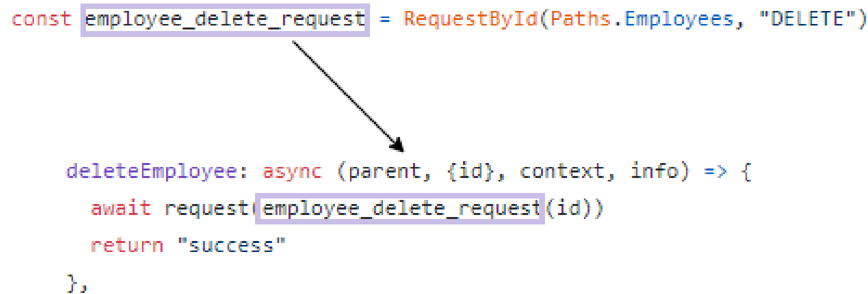

Fig. 8. Employee mutation for DELETE operation

The project client data is fetched here by using the SELECT statement over the SQL source into a client query, and the client is selected by a specific id. The (project) client also has a query resolver that fetches all clients available in the data source. Both queries for the client are represented in Figure 9.

```
clients: async (parent, args, context, info) => {
  return await query(`SELECT * FROM projects.client`)
},
client: async (parent, {id}, context, info) => {
  return (await query(`SELECT * FROM projects.client WHERE id = ${id}`))[0]
},
```

Fig. 9. Client queries

The GraphQL Client (Apollo-based MacOS application)

To communicate with the GraphQL server the Apollo-iOS framework [1] facilitates the integration, by converting between GraphQL and the Swift language [3], therefore we have a Swift language API for the GraphQL server that can be easily integrated in applications on Apple platforms.

Once the basic MacOS application is set up, the Apollo-iOS framework can be obtained from the official Apollo GraphQL Git repository [2]. In order to be able to perform the conversions, the generated code must be validated against the schema which is hosted on the GraphQL Server. Therefore, the schema needs to be downloaded and integrated into the client project so that the Apollo-iOS framework can assure the correctness of API functionality. A command line tool was created to automate the download of the latest schema each time the MacOS application is built. To download the schema programmatically Apollo provides ApolloSchemaDownloader as part of ApolloCodegenLib. It will download the schema from the specified host and store it in JSON format in the application folder for later use. Once the automatic download is done, it is added as a build step in the MacOS application Build Phases, to be executed each time the application is built.

To allow easy communication with GraphQL Server, Apollo iOS will generate a Swift API for the server, thus the client can interact with it in the Swift language. Apollo iOS regenerates the Swift API each time the application is built to account for changes. As the framework by itself isn’t performing any conversion, an additional setup is needed to implement the API generation process. ApolloCodegen API from ApolloCodegenLib is used to create the Swift API for our GraphQL server. In its essence, ApolloCodegen generates swift code to interact with GraphQL server and this code will contain all queries,

mutations and data structures in Swift format.

Once the connection is created, there two main API methods that can be used for interaction with the server:

- GraphQLQuery performs a query against the server. For the query parameter we can use any query generated by ApolloCodegen located in API.swift file; the fetching will respond back with a failure or success containing the subfields specified when the query type was defined;
- GraphQLMutation performs a mutation against the server. For the mutation parameter we can use any generated mutation by ApolloCodegen located in API.swift file; the mutation will respond back with a failure or success containing the subfields of the mutation type.

With this setup, the Swift data source specific to the needs of the macOS application was created – using the models in the created GraphQL server: Project, Employees, Clients and Roles. For each component, a list of all available entries is shown, with the ability to view, edit and delete each separate entry, as well as to create new ones. With GraphQL overfetching is largely avoided – e.g., when fetching entries for a particular model, only the requested fields are obtained, additional data will only be downloaded when viewing/editing the details.

The MacOS front-end follows the Master-Detail UI design, allowing to shift from abstract information to concrete details, as suggested in Figure 10, and GraphQL allows this dynamic while hiding the heterogeneity of the data sources.

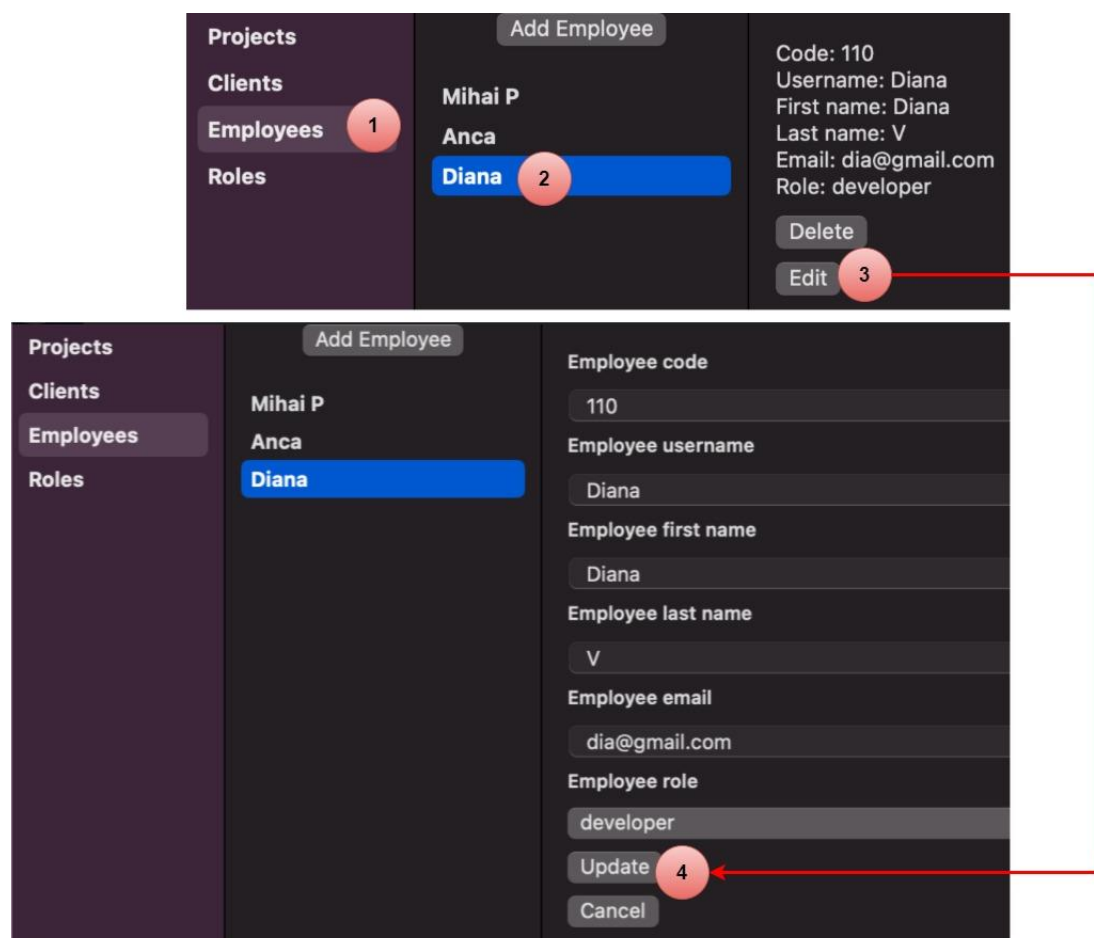

Fig. 10. Projects list and specific project details in the macOS application

5. Related Works

While in practice GraphQL has already gained traction, in research projects it is adopted slower, as a technological workaround rather than an artifact-building method. Besides the

literature that includes technical specification [6] or reference works for GraphQL [14], we could only find few papers that tackled this relatively new technology for fetching data and even fewer papers that presented case studies. Some are comparing between REST-based services and GraphQL just to highlight the latter's benefits, since GraphQL was designed from the beginning to alleviate the issues identified in the REST services.

Vazquez et al. [17] implemented a GraphQL API to replace the REST API for the Observatory of University Employability and Employment (OEEU). Their system included components for data gathering and analysis and components for visualizations to disseminate knowledge obtained from the analysis of the gathered data pertaining to university's employability. The nature of the collected data required a high number of endpoints in a REST- based architecture, which created difficulties in managing the server and even posed challenges in scalability. Migrating to GraphQL turned out to be easier to maintain as changes in the client requirements do not demand changes on the server side. However, they did not present how GraphQL was adopted on the client side.

Guo Y. et al. [10] proposed a GraphQL-based server for managing data that comes in real-time but, likewise, they miss a complete description of their implementation. Vogel et al. [18] presented a case study to migrate a smart home management system API to GraphQL and Wittern et al. [21] proposed a tool to create GraphQL wrappers from REST APIs with OpenAPI Specification.

Hartig and Perez [11] inspected GraphQL regarding its complexity by defining the semantics of the query language on a logical data model. The central issue was the possibility to have recursive and deeply nested queries which could result in a heavy burden on the server and cause denial-of-service attacks - a solution was provided to calculate the total size of a GraphQL response in polynomial time.

The work at hand is subordinated to a Design Science project interested in experimenting with data integration and data linking techniques across recent technologies.

6. Conclusion

In this paper, we presented engineering details for building a data graph middleware for project management needs using existing heterogeneous data sources – a MacOS proof-of-concept was developed to showcase integration of data coming from MySQL databases and a JSON RESTful data service.

By integrating the information from various data sources into a federated virtual data graph, the client app accesses a single endpoint in a flexible way that generally avoids overfetching and underfetching while having a certain degree of data validation, an approach that can deal with requirements that can change over time while remining within a certain semantic space of connected entities available in legacy systems.

As a future improvement, the tool will be extended by testing integration with additional data sources based on XML, RDF and MongoDB to compare performance of the mitigated data access compared to accessing the native data sources. An initial step in this direction was published [16].

Acknowledgements

Further technical and theoretical aspects of the work are detailed in the master dissertation of D. A. Vasiliev titled *Proof of Concept for Data Federation through a GraphQL Middleware*, defended at University Babeș-Bolyai.

The work of Dr. Ghiran and Dr. Buchmann was supported by project POC/398/1/1/124155 - co-financed by the European Regional Development Fund (ERDF) through the Competitiveness Operational Programme for Romania 2014-2020.

References

1. Apollo Data Graph Platform Official Documentation (2021) https://www.apollographql.com/ , Accessed June 18,2021

2. Apollo GraphQL Git repository (2021), https://github.com/apollographql/apollo-ios , Accessed June 18, 2021
3. Apple Developer Official Documentation (2021), https://developer.apple.com/ Accessed June 18, 2021
4. Bantini, C., Scannapieco, M. Data Quality: Concepts, Methodologies and Techniques, pp. 133-136, Springer (2006)
5. Demirkan, H., Kauffman, R.J., Vayghan, J.A., Fill, H.-G., Karagiannis, D. and Maglio, P.P., Service-oriented technology and management: perspectives on research and practice for the coming decade. The Electronic Commerce Research and Applications Journal. v7 i4. 356-376. (2008)
6. Facebook. GraphQL. Specifications (2018), https://spec.graphql.org/June2018/ Accessed June 18, 2021
7. Farré, C., Varga, J. and Almar, R., GraphQL schema generation for data-intensive web APIs. In International Conference on Model and Data Engineering, pp. 184-194. Springer, Cham (2019)
8. Fielding, R.T. Architectural styles and the design of network-based software architectures (Vol. 7). Irvine: University of California, Irvine. (2000)
9. Gartner - Gartner Magic Quadrant for Data Integration Tools (2021), https://www.gartner.com/en/documents/3989223/magic-quadrant-for-data-integration-tools, Accessed June 18, 2021
10. Guo, Y., Deng, F. and Yang, X. Design and Implementation of Real-Time Management System Architecture based on GraphQL. In IOP Conference Series: Materials Science and Engineering (Vol. 466, No. 1, p. 012015). IOP Publishing (2018)
11. Hartig, O. and Pérez, J., Semantics and complexity of GraphQL. In Proceedings of the 2018 World Wide Web Conference, pp. 1155-1164. (2018)
12. Lowdb, https://github.com/typicode/lowdb, Accessed April 1st 2021
13. Mukhiya, S. K., Rabbi, F., Ka I Pun, V., Rutle, A., Lamo, Y. A GraphQL approach to Healthcare Information Exchange with HL7 FHIR, The 9th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2019), Volume 160, pp. 338-345, Elsevier (2019)
14. Porcello, E. Banks, A. Learning GraphQL: declarative data fetching for modern web apps. O'Reilly Media, Inc. (2018)
15. Prat, N., Comyn-Wattiau, I., Akoka, J., Artifact Evaluation in Information Systems Design-Science Research-a Holistic View. PACIS2014, Proceedings 23, pp.1-16. (2014)
16. Vasiliev, D.A., Ghiran, A.M., Buchmann, R.A. Evaluation of Data Integration Plans based on Graph Data, 25th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (in press), Elsevier (2021)
17. Vázquez-Ingelmo, A., Cruz-Benito, J. and García-Peñalvo, F.J. Improving the OEEU's data-driven technological ecosystem's interoperability with GraphQL. In Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality, pp. 1-8. (2017)
18. Vogel, M., Weber, S. and Zirpins, C. Experiences on migrating RESTful web services to GraphQL. In International Conference on Service-Oriented Computing, pp. 283-295. Springer, Cham, (2017)
19. W3C: The RDF official resource page (2021) http://www.w3.org/RDF/ Accessed June 18, 2021
20. Wieringa, R.J. Design science methodology for information systems and software engineering. Springer (2014)
21. Wittern, E., Cha, A. and Laredo, J.A. Generating GraphQL-wrappers for REST(-like) APIs. In International Conference on Web Engineering, pp. 65-83. Springer, (2018)
22. Wittern, E., Cha, A., Davis, C., J., Baudart, G., Mandel, L. An Empirical Study of GraphQL Schemas, Lecture Notes in Computer Science, vol 11895, pp. 3-19. Springer (2019)