SOI: 1.1/TAS DOI: 10.15863/TAS

International Scientific Journal

Theoretical & Applied Science

p-ISSN: 2308-4944 (print) **e-ISSN:** 2409-0085 (online)

Year: 2021 **Issue:** 01 **Volume:** 93

Published: 23.01.2021 **http://T-Science.org**

QR – Issue **QR – Article**

Vadim Andreevich Kozhevnikov
Peter the Great St. Petersburg Polytechnic University
Senior Lecturer
vadim.kozhevnikov@gmail.com

Donat Vitalievich Shergalis
Peter the Great St. Petersburg Polytechnic University
student
donutellko@gmail.com

MIGRATING FROM REST TO GRAPHQL HAVING LONG-TERM SUPPORTED CLIENTS

Abstract: *This article describes a concept of a new approach in API architecture. The article also contains an analysis of some existing architectural styles and technologies and compares them to this new approach.*

Key words: *API architecture, GraphQL, REST.*

Language: *English*

Citation: Kozhevnikov, V. A., & Shergalis, D. V. (2021). Migrating from rest to graphql having long-term supported clients. *ISJ Theoretical & Applied Science, 01 (93),* 180-185.

Soi: http://s-o-i.org/1.1/TAS-01-93-31 ***Doi***: crossref https://dx.doi.org/10.15863/TAS.2021.01.93.31

Scopus ASCC: *1700.*

Introduction

Such technologies as SOAP (Simple Object Access Protocol) and REST (Representative State Transfer) have been the main architectural approaches for data exchange. The obvious difference is the way data is encoded. In the case of SOAP, the only supported format is XML. While REST does not define the format, so developer can transmit JSON, HTML, XML and binary data. Each approach has its own advantages and disadvantages, but REST is the priority choice for modern developers due to the lightness and better readability of the JSON format, flexibility and ease of use, especially in web development [1].

However, when working with REST, developers also face certain limitations and disadvantages. The main ones are the so-called over fetching and under fetching, when, in response to a request, the REST client receives extra data, or vice versa, an insufficient amount of it, so client needs to perform another request. This often happens due to changes in application design or functionality. In case of insufficient data, the back-end developer has to modify the code to add the required data, while cases of redundancy are often ignored. The problem is aggravated, when there are different types of clients – for example, Android, iOS and web applications.

The need for small changes on the server side slows down the development process. As one of the approaches to solve the problem, Facebook created for internal use the GraphQL query language, which allows developers on the client side to select the data they need using a special query language [2].

After the specification was published, developers became interested in a new flexible approach, but faced a number of obstacles when trying to implement this technology in the systems they are developing [3]. The main ones were:

- the inability to upgrade existing versions of mobile and desktop applications to use GraphQL;
- the need to maintain both protocols during the process of migration [4];
- developers of client applications are unwilling to learn and implement new technology.

To solve these problems, the author suggests creating a service that will become an intermediary between the GraphQL server and the REST client. The author expects this can solve the listed problems, and also have additional advantages, compared to using each technology separately.

The aim of the article

The purpose of this work is to study the possibility and reasonability of creating the service described earlier. To do this, we need to compare GraphQL to other architectures and protocols, then make a conclusion about the feasibility of development of such technology, based on a comparison of approaches and the expected usability.

Benefits of GraphQL in compare to REST

Some of the benefits obtained by using GraphQL have already been listed in the introduction, but we will consider them in more detail [5, 6]:

The client can specifically indicate what data he needs and in what form.

This allows us to save the number of network calls, calls to the database, memory and file system, save traffic and get rid of unnecessary transfor-mations, conversions and sorts.

Let's say there is a service that displays the user's top 25 photos in high quality, sorted by the number of likes and with several top comments. Then, on the profile page, we need to display a preview of the last five photos, sorted by date, so we add a new endpoint. Then we decided to create mobile versions for both screens, where we display less photos, do not display comments, and thumbnails are used instead of full-size images. This would require at least two modifications to backend code, which might be considered impractical due to lack of time. As a result, extra requests are made on the server side to get comments for images, and the user wastes time and traffic on downloading images in higher quality.

If we use GraphQL, in both cases developers of client application would have just to change their requests: the sorting method and the number of requested photos, remove comments from list of requested fields, and change link to the image to the link to the preview.

It simplifies the aggregation of data from multiple sources in a single query.

Let's say we have a service that provides information about accounts and a service that provides information about cards. To get information about the accounts and the cards linked to them, you have to either make two requests and compare their results on the client side, or create another service that aggregates the information and provides it in the form we need. When using GraphQL, one service called BFF (Backend For Frontend) is a generic aggregator.

The type system is used to describe the data.

A schema is a contract between a client and a server; it allows you to specify field types for requests and responses, lists of possible values (enum), and their mandatory presence (nullability).

Disadvantages of GraphQL

However, GraphQL also has disadvantages in comparison with REST [5, 7]:

Need to manage additional constraints.

Since consumers of the GraphQL API have ability to choose the data that he wants to receive, a security issue arises. For example, a malicious user can send a request to the server to obtain complete information about all users in order to use them for purposes that are contrary to the interests of the company. Or send a lot of resource-intensive requests in order to cause a denial of service. To prevent both attacks, developers need to set additional restrictions, anticipating possible ways of abuse. Also, some implementations have a Persistent Queries mechanism that allows you to set all possible queries and refer to them by a unique identifier.

The missing field is indistinguishable from null.

For example, there is a mutation request to update user information, allowing the username, address, and phone number to be changed. These fields may or may not be present in the request so that the customer does not have to send an address and phone number if they want to change their name. However, if the user wants to completely remove the address value by passing null, then the server will decide that the address field should not be changed.

The difference between the input and output format.

To illustrate, let's use one of the principles of REST – statelessness. For example, in one request, the client receives a certain context, which he must transfer in the next request. In the case of REST, the server and client exchange an identical object. And in the case of GraphQL, for this purpose, separate types for input and output must be defined in the schema, and the client must compose a request using the fields from the response.

Polymorphism is unsupported for mutations.

We can inherit one object from another or use union for several objects, but we cannot inherit input objects used in mutation. For example, we need to transfer information about the account holder. If the owner is an individual, then his last name, first name and patronymic are needed, and for a legal entity – the name of the company. In the case of REST, the client can pass an additional field containing the type of the account holder, and the server can deserialize to the desired type based on this field. And when using GraphQL, you have to create a separate query for each type.

Lack of namespaces.

Each GraphQL service has a single schema. In the case of a large project, the scheme can reach significant volumes, and orientation in it will require additional documentation, and the likelihood of name collisions increases. REST, in turn, does not use such concept as type of transferred objects, so these types are managed by clients and servers separately.

Limiting the use of the Backend Driven UI approach.

With this approach, the server side is in charge of controlling the user interface: the application is a collection of widgets. The list and order of widgets displayed on each screen, the contents of each widget, as well as the way to navigate between screens, are received by application from the server. When using GraphQL, the developer will have to decide how to transfer the changing query from the server to the device, or in response to each change, edit the schema by creating or modifying the existing query.

As you can see, both REST and GraphQL have their advantages and disadvantages, but the decision to use one or another approach should be made taking into account the specifics of a particular project.

GraphQL migration issues

In case developers decide to start using GraphQL in an existing product, they may face a number of problems [9]. Let's take an example to illustrate possible problems.

Let's say a company has mobile apps for Android and iOS platforms, as well as web versions of apps for computers and mobile devices. Some users are unable to update the application due to the fact that their version of the operating system is no longer supported by the application, but the company does not want to lose profit from these users.

The company periodically redesigns the application, so the format of data presentation on a large number of screens changes, in connection with which the formats of requests to the server and responses from it might also be changed. Due to the large number of available platforms, the workload on the server-side developers increases greatly, since it is necessary to develop services taking into account the differences in the presentation of the result on each platform.

Using GraphQL would reduce the burden on the backend developers, since in this case they just need to create a number of generic sources and provide a schema for which the developers of client applications will write queries.

However, the introduction of new technology means that it is necessary to teach every developer on each platform to use it, and then maintain two versions of the API simultaneously for a long time [8] – GraphQL for new versions of the application, and REST for old ones.

Proposed technology

As a solution to the listed problems, the author proposes creating an additional service as an adapter between REST clients and GraphQL servers. This service stores mappings (the correspondence between requests of two types), and when receiving a REST request, finds the corresponding GraphQL request template in its database, fills it with data from the REST request and sends it to the GraphQL server for execution, then returns the response to the client.

In addition to the described basic functionality, it is also possible to implement the following features:

If there is no match for the REST request, the request is passed to the gateway service – thus, the described service can become the gateway and the only access point to the system.

If necessary, casting the response of the GraphQL service to a different form can be implemented in order to maintain backward compatibility.

Advantages of the proposed technology

Let's consider the advantages of this approach in comparison with using REST or GraphQL:

– Flexibility and power of the GraphQL language – developers have the ability to write full-fledged GraphQL queries and develop the back end in accordance with all the principles of GraphQL.

– Backward compatibility on the client – the old version of the application can continue to function after the full transition to using GraphQL on the server.

– Developers of client applications do not need to learn new technology and implement it – interaction with the server is still carried out through the usual REST.

– Responsibilities for writing, debugging and editing queries can also be assumed by analysts and support staff, without requiring the participation of developers.

– Possibility of gradual migration – requests that were not implemented in GraphQL are passed to the old services, and then can implicitly replaced by the new implementation.

– Such service can access many different GraphQL services, allowing you to delineate areas of responsibility and solve the naming problem.

– It is possible to use existing caching mechanisms on the client and intermediate nodes, not suitable for use with GraphQL.

– A similar system can be implemented for any protocol of communication with the client – for example, instead of REST, the SOAP protocol can be used.

– To add and change the mapping, it is not necessary to reload the service, which allows you to make and check changes much faster than when you would have changed the code. Mappings can be updated periodically, upon detection of changes or by another event.

– The ability to edit a query on the server in real time, which is impossible when using the Persistent Queries mechanism, in which the client uses the hash code of one of the predefined immutable queries.

– Settings can be taken from any type of source – git or other version control system, database, config servers, local yaml files, etc.

– Storing mappings in the version control system will allow you to use an existing role model, including streamlined processes for reviewing changes, as well as protect mappings from unauthorized changes and loss, and use versioning.

– The response format can be easily manipulated using JSON formatting technologies such as the JOLT library [10].

– If you migrate from this solution to using GraphQL without intermediaries, developers will have tested GraphQL queries ready to be used. And to support old versions of the application after migration, you will not have to support the old REST cluster, but only this service.

– Reduces the amount of code responsible for data representation in services. This introduces a new level of abstraction that allows developers to write cleaner code.

– This service as a separate layer can be tested independently of the rest of the system using autotests: for this, "stubs" can be used, called instead of real services on the testing environment.

Disadvantages of the proposed technology

– An additional element in the call chain slows down the execution of requests.

– Service and mapping sources are new potential points of failure.

– If the source of mappings is unavailable, the service cannot be restarted – an additional source must be provided.

– With increasing number of mappings, the complexity of system support can significantly increase.

– We need a new mechanism for testing changes. Those actions that were previously covered by unit tests in services can now only be tested using autotests and stub services.

As you can see, the number and significance of expected advantages significantly exceeds the number and significance of expected disadvantages, and therefore the author consider it expedient to develop such a system.

Description of proposed service

The implementation of the proposed system should have the following functionality:

– Load mappings from the source, which is the git repository, the link to which is specified in the settings.

– Accept REST requests for GET, POST, PUT and DELETE methods.

– Find the corresponding mapping for the request by the request method and request URI. The request URI specified in the mapping can have a path variable, that is, for the request `GET /api/users/123/accounts?currency=RUR` the following mapping should be found: `GET /api/users/${userId}/accounts`

– Use path variables, query params and request headers as variables to form a request according to the template specified in the mapping. So, in the previous example, based on the request, the values of two variables will be set: `userId=123 and currency=RUR`.

– Also use body to get variable values. For example, when receiving a request with the following body

```
{"name": "Aleksandrov", "address": {"street": "Lubyanka", "house": "1"}
```

the values of the variables will be obtained `name=Alexandrov; address/street=Lubyanka; address/house=1`

– Substitute variable values into the query template stored in the mapping, in place of placeholders with the corresponding name. For example, the following request from the mapping:

```
{users (id: "${userId}") {accounts (currency_eq: "${currency}") {number}}
```

should be filled in like this:

```
{users (id: "123") {accounts (currency_eq: "RUR") {number}}
```

– If the mapping was not found, the request must be sent unmodified (with the URI, query params, headers and request body preserved) to the API Gateway address specified in the settings.

– Transformation of the received response is carried out if there is a transformation rule in the mapping. The transformation uses the technology specified in the mapping. The transformation rules are specified in the format corresponding to the specified technology. Supported technologies are selected by the developer.

A diagram illustrating the interaction between the client and the described service is shown in Figure 1.

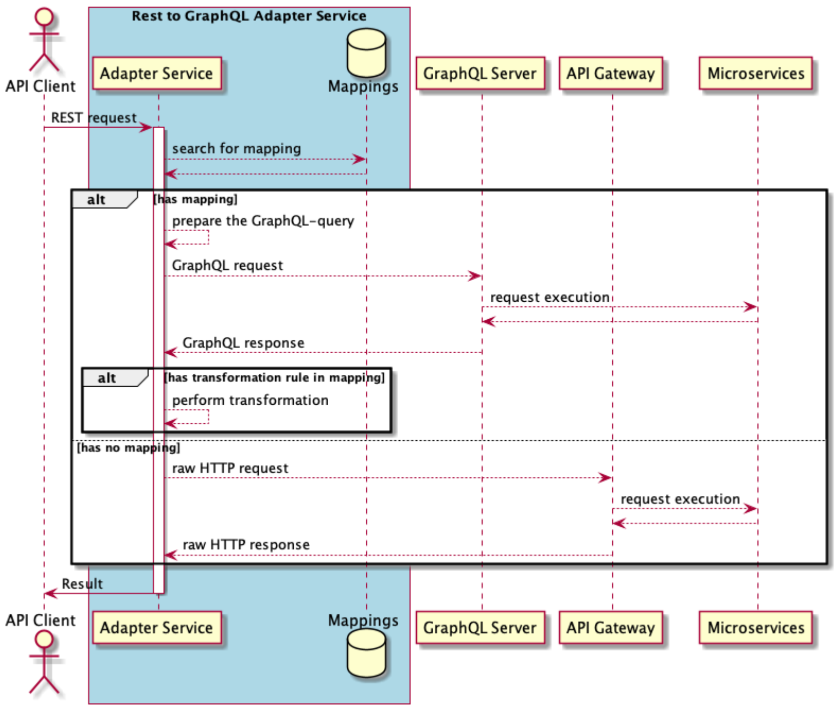

Figure 1 – sequence diagram

Conclusion

In this article we explored REST and GraphQL, their advantages and disadvantages, compared them to each other and modeled a process of migration from REST to GraphQL. Based on this data we described a service that would superpose both approaches, not only combining most of their advantages, but also having additional benefits.

The description can be treated as a specification for implementation of such service.

References:

1. Fielding, R.T. (n.d.). *Architectural Styles and the Design of Network-based Software Architectures.* [Accessed at 26.01.2021]. Retrieved from https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
2. (n.d.). *Introduction to GraphQL*. [Accessed at 26.01.2021]. Retrieved from https://graphql.org/learn/
3. Sandoval, K. (n.d.). *Federated Data With HyperGraphQL.* [Accessed at 26.01.2021]. Retrieved from https://nordicapis.com/how-to-make-graphql-work-for-you/
4. Mayogra, J. (n.d.). *Gradually migrating an app from REST to GraphQL.* [Accessed at: 26.01.2021]. Retrieved from https://graphql.college/gradually-migrating-a-node-and-react-app-from-rest-to-graphql
5. (n.d.). *Honest Engineers. Why use GraphQL, good and bad reasons*. [Accessed at 26.01.2021]. Retrieved from https://honest.engineering/posts/why-use-graphql-good-and-bad-reasons
6. (n.d.). *The Ultimate Guide to API Architecture: REST, SOAP or GraphQL?* [Accessed at 26.01.2021]. Retrieved from https://da-14.com/blog/ultimate-guide-api-architecture-rest-soap-or-graphql
7. Soharev, P. (n.d.). *What's wrong with GraphQL (Chto ne tak s GraphQL)* [in Russian]. [Accessed at 26.01.2021]. Retrieved from https://habr.com/ru/post/425041/
8. Haldar, M. (n.d.). *Migrating Existing REST APIs to GraphQL*. [Accessed at: 26.01.2021]. Retrieved from https://blog.bitsrc.io/migrating-existing-rest-apis-to-graphql-2c5de3db647d

9. Fletcher, C. (n.d.). *A look at Trello: adopting GraphQL and Apollo in a legacy application*. [Accessed at 26.01.2021]. Retrieved from https://atlassian.com/engineering/a-look-at-trello-adopting-graphql-and-apollo-in-a-legacy-application
10. Simpson, M. (n.d.). *JOLT*. [Accessed at 26.01.2021]. Retrieved from https://github.com/bazaarvoice/jolt