COMPARATIVE ANALYSIS OF WEB APPLICATION PERFORMANCE IN CASE OF USING REST VERSUS GRAPHQL

Milena Vesić[7]
Nenad Kojić[8]

DOI:

Abstract: *Web applications are the most common type of application in modern society since they can be accessed by a large number of users at any time from any device. The only condition for their use is an Internet connection. Most applications run using the HTTP protocol and client-server architecture. This architecture is based on the use of API (Application programming interface), most often REST architecture (Representational State Transfer). If there are several different functionalities on the website that fill their content with data from the web server, for most of them a special HTTP request must be generated with one of the existing methods (GET, POST, PUT, DELETE). This way of communication can be a big problem if the connection to the Internet is weak, there are a lot of HTTP requests because you have to wait for each request to be executed and for the web server to return the data.*
In this paper, one implementation of GraphQL is presented. GraphQL is an open-source data query and manipulation language for APIs. GraphQL enables faster application development and has less server code. The key advantage is the number of HTTP requests because all the desired data of the page is obtained with one request. This paper will show a comparative analysis on the example of a real website in the case of using the REST architecture and GraphQL in the case of different qualities of Internet connections, code complexity and the number of required requests.

.
Keywords*: Web page, GraphQL, REST, Performance.*

INTRODUCTION

Web applications are popular solutions today, because they can be accessed at any time, from anywhere, from computers or mobile devices, and they can be accessed as well as Internet pages using a Web browser (Kalmanek, 2010.). Web applications work on the principle of client-server communication. The client communicates with the server using the HTTP protocol (Fielding, 1998.), and in most cases using REST (Neumann, 2018.) for Uniform Resource Identifier (URI). The evolution of Web applications and programming languages has led to the evolution of communication between client and server (Kalmanek, 2010.). Certainly, the API enables this communication, but not necessarily using the REST architectural style (Neumann, 2018.). The paper will describe GraphQL (Freeman, 2019.) as a query language for APIs that allows clients to request what they need and nothing more than that data. GraphQL is not tied to a specific database or storage mechanism, but supports existing code and data.

[7] Academy of Technical and Art Applied Studies Belgrade (ATUSS) - Department School of Applied Studies for Information and Communication Technologies Belgrade, Zdravka Celara 16, Serbia
[8] Academy of Technical and Art Applied Studies Belgrade (ATUSS) - Department School of Applied Studies for Information and Communication Technologies Belgrade, Zdravka Celara 16, Serbia

Web applications can be implemented in one of two basic ways: server-side rendering and client server architecture. This is independent of which programming language is used, which technology or code writing architecture. Recently, the client-server architecture is becoming increasingly popular, which is further supported by the great popularity of front-end frameworks for programming client-side programming languages.

HTTP request/response (Fielding, 1998.) plays a key role in such communication. The user experience is based on the design of the application, the content and the speed of the application. The speed of the application is mostly based on the number of HTTP requests and the amount of content that is exchanged through them. For this reason, from the point of view of the speed of application and user experience (Neumann, 2018.), the number of HTTP requests needs to be minimized as much as possible. On the other hand, SEO techniques and web application quality assessment by the web search engine also expect a reduced number of HTTP requests.

The required reduction leads to the need to either change the way the code is written and the individual HTTP requests to the group. This is possible to some extent because the logic by which the related data is conditionally retrieved requires sequentiality in the work.

Therefore, reducing the number of HTTP requests is only possible (Neumann, 2018.). The possible solution is therefore not sought in technologies for reducing requests, but in changing the way of organization and data structure that is returned from the server side to the client part of the web application. GraphQL is therefore one of the candidates that can change this (Freeman, 2019.).

The possibilities of GraphQL will be analyzed in this paper using comparative analysis in relation to the classic REST way of communication of the client part of the web application with the server (Brito, 2020.). This will be observed in relation to different Internet speeds, the volume of data in the database in terms of the number of records in the tables and the number of generated HTTP requests and the time required to implement these requests on the client part of the application.

This paper is organized through four chapters: After the Introduction where the basic ideas and goals are given, the second chapter gives the key features of GraphQL that are basic for its implementation and represent some of the key features that provide developers the ability to implement them better than classical REST form of communication. The third chapter describes the methodology of the observed empirical results, which compares the use of GraphQL and the classical REST form of communication with the server in different operating conditions. Finally, a conclusion and further guidance in the investigation.

GRAPHQL

GraphQL is a query language for APIs (Application programming interfaces) and "server runtime" for executing these queries with existing data (Freeman, 2019.).

The way applications are created has been evaluated in the last twenty years, the biggest change being perhaps the "single page" applications running in the Browser, unlike the earlier multi-page applications running on the server. And for client-server communication, the REST (REpresentational State Transfer) architectural style is still mostly used. REST uses the HTTP (HyperText Transfer Protocol) transmission protocol and the Uniform Resource Locator

(URL) addressing mechanism. It communicates with resources using HTTP methods (GET, POST, PUT, DELETE) to send, retrieve, delete and update.

Using the REST architectural style to display data in different blocks of a single page application, it would be necessary to send multiple HTTP requests (Brito, 2020.). The following image shows an example of how many requests would be forwarded to the server to display all the data on the Gmail homepage. Each block (marked in green) would be populated with data that would be obtained as a single response for each HTTP request.

This way of communication can be a problem if the connection to the Internet is bad, because you have to wait for each request to be executed and for the server to return the data. The difference between REST architectural style and GraphQL is precisely in the number of requests (Brito, 2020.), GraphQL would get all this data by sending one request.

Another advantage is that GraphQL only retrieves the required data. It enables faster application development, has less server code because GraphQL is a layer between the client and the server that does not allow "bad" requests to reach the server side.

GraphQL is not tied to any specific database (Brito, 2019.), it doesn't even have to work with a database.

The GraphQL service is created by defining the types and fields of those types, as well as creating functions for each field and each type (https://graphql.org/learn/). When the GraphQL service starts, it can receive queries that it validates and executes. The received query is first checked to ensure that it refers only to defined types and fields, and then a function is run to get the result.

In addition to these feature keys, GraphQL has several very important features that give it a very significant competitive advantage, some of which are (Freeman, 2019.):

- GraphQL addresses the server looking for the values of the explicitly specified object fields. Unlike SQL where asterisk (*) can be used to select all columns of a table, GraphQL does not have this option, but only those fields whose values are needed are listed.
- The way they communicate (client-server) is not explicitly specified. The SSH, FTP, Web Socket, or HTTP method can be used as the most common mode for the transport mechanism.
- Each query can have an operation type (query or mutation) and an operation name. If they are not specified, then the type of operation will be query, which means that data is required from the server. Operation names are optional, but make the code more readable and easier to debug.
- The server response always corresponds to the request format which is a characteristic of GraphQL. The values required do not have to be simple data types (scalar data types - String, Integer, Float, Boolean or ID), but the value can be an object or a string (complex data types).
- When sending a query, arguments can also be passed. For example, if there is a need to display data for a specific person, then the ID of that person by whom his record is unique can also be forwarded by query.
- GraphQL allows you to send a set of arguments when sending a query, that is, each field nested inside the object can have its own arguments.

- In a system such as REST, only one set of arguments can be passed - query parameters and the URL segment in the request. In GraphQL, each field and nested object can get its own set of arguments, making GraphQL a complete replacement for creating multiple data retrieval APIs. Arguments can even be passed to scalar fields to implement data transformations once on the server, instead of on each client separately.
- Because the result fields match the field name in the query, but do not include arguments, the same field with different arguments cannot be requested directly. That's why aliases are needed - they allow you to rename a field result to anything.
- Fragments are reusable units, that is, a list of fields of a certain type. When there is a need to search for the same group of data in a query, then fragments are created. If fragments were not used in the query, there would be code repetitions. The snippet concept is often used to break complex application data requirements into smaller parts, especially when many UI components have to be combined with different fragments into one initial data set.
- Union is a complex data type. It is used in situations where the server is expected to return results of different types for the search term. It can be compared to if else flow control. It is most often used in searches.
- Working with arguments and their values in most web applications means that these values are dynamic, not "hardcoded".
- It would not be good to pass dynamic arguments directly to a query array, because then the client-side code would have to dynamically manipulate the query array during execution and serialize it into a GraphQL-specific format. Instead, GraphQL has a first-class way to factory-process dynamic values and pass them as a separate dictionary to variables.

Directives: Directives allow us to dynamically change the structure and form of queries using variables. The directive can be added to a field or fragment and can affect the execution of the query in any way the server wants. The core GraphQL specification includes exactly two directives that must be supported by any GraphQL server implementation that complies with the specifications:

@include (if: Boolean) The field is included in the result only if the argument is true.
@skip (if: Boolean) The field is skipped if the argument is true.

Directives can be useful for getting out of situations where otherwise string manipulation would have to be done to add and remove fields in the query. Server implementations can also add experimental features by defining completely new directives.

Mutations: One important thing is addressed to mutations. Mutations are a type of operation that is used when it is necessary to change the data on the server, or when writing, modifying or deleting data.

The difference between queries and mutations is that the functions called by the query are executed in parallel, while the fields called in the mutation are called one after the other. Although changes to the server can be made by sending a query, it is better to use mutations because it is by convention.

Errors: Using REST architectural style the result of the request is either 100% success or 100% failure. In GraphQL, queries are executed partially, that is, part of the queries will be executed successfully regardless of the need for a field that cannot receive a response from the server. In the answer, in addition to the requested data, the property of errors for the requested field

whose data cannot be returned is also obtained. The reason for this is that REST uses multiple requests, while GraphQL uses one.

Checking whether the request was successfully executed using the REST architectural style was done by checking the status code, and if the code would start with the number 2, then the request was successfully implemented. Using GraphQL the status code will start with the number 2, but it may happen that part of the query is not successfully implemented. Errors are processed by checking to see if there is an error property in the response.

This type of error occurs when the search box server is disabled. If the field name is changed to start with a capital letter P, the answer will be different because something that does not exist is required.

Scheme: GraphQL services can be written in any language. Because it cannot rely on a specific programming language syntax, such as JavaScript, GraphQL uses a schema language. The language of the GraphQL schema is similar to the query language. If the types that will be used in the application are not known, by setting a query for the __schema field, which is always available in the basic query type, a list of defined types is obtained.

Some of important things are also addressed to the queries. We can view each field in the GraphQL query as a function or method of the previous type that returns the next type. In fact, GraphQL does just that. Each field on each type has a function called a resolver provided by the GraphQL server programmer. When the field is executed, the corresponding resolver is called to produce the next value. At the top level of each GraphQL server is a type that represents all possible entry points in the GraphQL API, often called the Root or Query type. As each field is resolved, the resulting value is placed in the key / value folder with the field name (or alias) as the key, and the resolved value as the value. This continues from the bottom query field all the way to the original query root type field.

Resolve: "Resolve" functions are like small routers. They determine how the types and fields in the schema are connected on the server side. GraphQL "resolve" functions can contain arbitrary code, which means that the GraphQL server can communicate with any type of server code, even with another GraphQL server. For example, the Person type can be stored in an SQL database, while the City type can be stored in MongoDB, or it can even be serviced by a micro service.

Perhaps the biggest feature of GraphQL is that it hides all server complexity from the client. No matter how many server pages the application uses, all the client will see is one GraphQL endpoint with a simple self-documenting API for the application.

RESULTS

An example of two web-oriented applications is shown to show the difference when using REST architectural style and GraphQL.

Both data storage applications use the PostgreSQL version 11.4 database. The graphical interface pgAdmin version 4.24 was used for data manipulation.

A web application that uses GraphQL as a query language, i.e. a way of communicating between the client and server side, uses the Node.js version 10.15.3 executable as support for

the server part of the code, while GraphiQL, the reference implementation of GraphQL IDE, is used to test servers and queries.

REST architectural style is applied in a web application that uses AJAX requests for client-server communication, i.e. requests are created in the client part of the application in JavaScript programming language, while PHP is used as the server programming language.

Both applications should display all the data from the database on the application's home page, with different client-server communication technologies having a different number of requests to the server.

Using GraphQL, no matter how many records there are in the database, the number of requests will always be one, while using the REST architectural style, the number of requests differs depending on how many records there are within the tables. This happens because one collection of data, which was received in response to an HTTP request, i.e. its element has its own data collection for which it is necessary to send a new HTTP request to the server.

Results of comparison of REST and GraphQL in the execution speed of requests at different flow rates in examples with 10 and 100 records per table in the database. In the example with 10 records per table, using REST architectural style the number of requests is 12, while in the example with 100 records, the number of requests is 102. Using GraphQL in both examples the number of requests is 1.

Table 1. Execution time of requests for different flow rates with 10 records per table

Flow rate	REST	GraphQL
36.59 Mbps	802ms	174ms
10 Mbps	988ms	139ms
128 Kbps	1.61s	907ms

Table 2. Execution time of requests for different flow rates with 100 records per table

Flow rate	REST	GraphQL
36.59 Mbps	6.48s	122ms
10 Mbps	6.63s	210ms
128 Kbps	11.25s	7.93s

Figure 1. Execution speed of requests at different flow rates with 10 records per table in the database

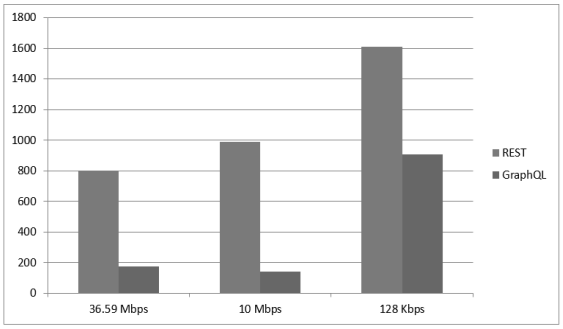
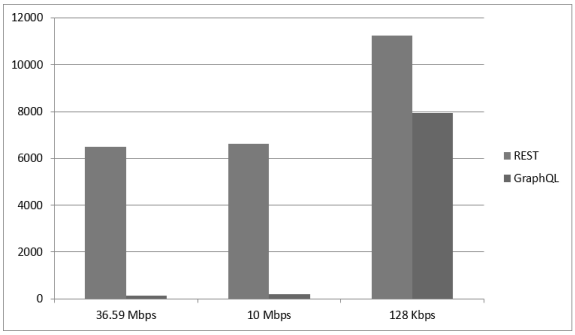


Figure 2. Execution speed of requests at different flow rates with 100 records per table in the database



Results of comparison of REST and GraphQL in the execution speed of requests at different flow rates in examples with 500 and 1000 records per table in the database. In the example with 500 records per table, using REST architectural style the number of requests is 502, while

in the example with 1000 records, the number of requests is 1002. Using GraphQL in both examples the number of requests is 1.

Figure 3. Execution speed of requests at different flow rates with 500 records per table in the database

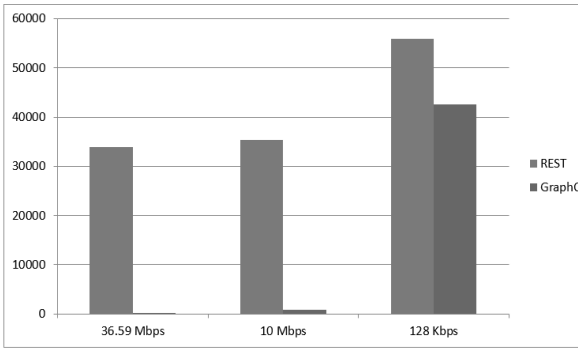


Figure 4. Execution speed of requests at different flow rates with 1000 records per table in the database

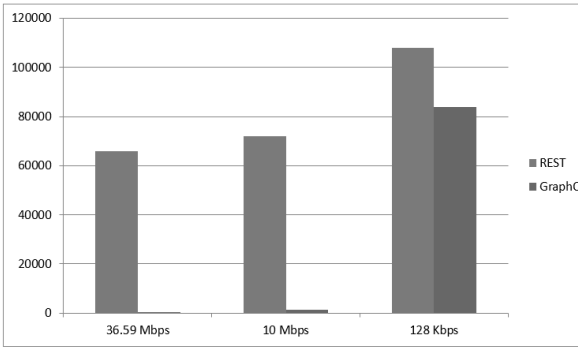


Table 3. Execution time of requests for different flow rates with 500 records per table

Flow rate	REST	GraphQL
36.59 Mbps	33.98s	211ms
10 Mbps	35.32s	777ms
128 Kbps	55.96s	42.66s

Table 4. Execution time of requests for different flow rates with 1000 records per table

Flow rate	REST	GraphQL
36.59 Mbps	1.1min	352ms
10 Mbps	1.2min	1.37s
128 Kbps	1.8min	1.4min

CONCLUSION

This paper presents the possibilities of GraphQL and the possibilities of reducing the number of HTTP requests if it is used and compared with REST technology. The real conditions of different faster Internet connections, different number of records in the database and all this from the angle of the total time required to show the user the desired content in the website are analyzed. It has been shown that the use of GraphQL achieves a large reduction in the total time and number of HTTP requests, which is very important for the user experience and the quality of the web application. Further work will focus on analysis with a larger number of tables and records in the database and implementation with different front-end frameworks.

REFERENCES

Brito, G., & Valente, M. T. (2020, March). REST vs GraphQL: A Controlled Experiment. In 2020 IEEE International Conference on Software Architecture (ICSA) (pp. 81-91). IEEE.

Brito, G., Mombach, T., & Valente, M. T. (2019, February). Migrating to GraphQL: A practical assessment. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 140-150). IEEE.

Fielding, R., & Gettys, J. (1998). Hypertext Transfer Protocol-HTTP/1.1.

Freeman, A. (2019). Understanding GraphQL. In Pro React 16 (pp. 679-705). Apress, Berkeley, CA.

Introduction to GraphQL, https://graphql.org/learn/

Kalmanek, C. R., Misra, S., & Yang, Y. R. (Eds.). (2010). Guide to reliable Internet services and applications. Springer Science & Business Media.

Porcello, E., & Banks, A. (2018). Learning GraphQL: declarative data fetching for modern web apps. " O'Reilly Media, Inc.".

Neumann, A., Laranjeiro, N., & Bernardino, J. (2018). An analysis of public REST web service APIs. IEEE Transactions on Services Computing.

Sud, K. (2020). Understanding REST APIs. In Practical hapi (pp. 1-11). Apress, Berkeley, CA.