An Overview of GraphQL: Core Features and Architecture

Vlatko Spasev, Ivica Dimitrovski, and Ivan Kitanovski

Faculty of Computer Science and Engineering, Skopje
http://www.finki.ukim.mk {vlatko.spasev, ivica.dimitrovski, ivan.kitanovski}@finki.ukim.mk

Abstract. GraphQL is a query language used for querying Web-based APIs. It is a relatively new way of accessing and querying API's that proves to be quite successful in many parameters compared with the most used so far REST architecture, more and more large companies are starting to use it. In this paper, we will present the core features of GraphQL and then through some examples we will present the advantages and disadvantages of using GraphQL as an API query language. After making a deep overview of the core features, the Subscription option will be discussed followed by the Protection mechanisms that are most used to protect the GraphQL server. At last, we will explain and analyze how big companies, Facebook, Twitter and Coursera, implemented GraphQL in their platforms and the problems that they encountered during the implementation.

Keywords: GraphQL · API · REST · JSON

1 Introduction

An API presents an interface that allows the applications to make interaction with some external remote service and fetch or send some related data. The client does not need to know and understand the internal business logic of the service, only the endpoints are enough to get the desired information. For this data transfer, the APIs can use different architectures. One of the most used in the past was the SOAP, which is a message protocol that allows communication between remote application elements. But, as the web and mobile application market was growing and the need of lightweight apps grows, the SOAP was replaced with the REST API architecture.
Since the introduction and definition of REST principles in 2000 with Roy Fielding's doctoral dissertation, more and more web-based APIs have begun to use it, slowly replacing the SOAP messaging protocol. Today, most web-based APIs still use REST as an architectural style. Most of the web services that use the REST architecture are using compact and easy to use JSON. That is data interchange format that uses human-readable text to transmit and store the data that is formatted as attribute-value pairs. Beside all advantages and disadvantages, the lastest migrations in the big applications from REST to GraphQL, may lead

to losing the throne of REST. In 2012, as a need to implement a new News Feed API for their mobile app, Facebook began giving the shape of GraphQL. In 2015, the GraphQL specification became available for the first time.

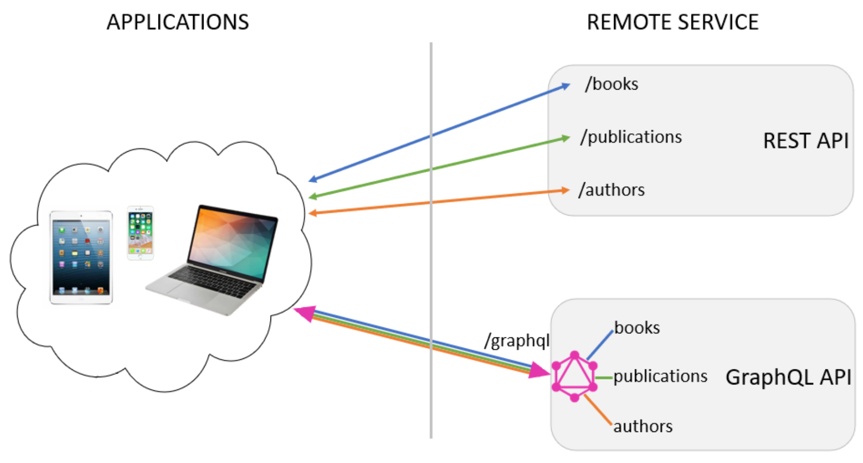

Fig. 1. Endpoint explanation between REST and GraphQL

The simplest way to present GraphQL is to explain it as a query language used for querying Web-based APIs [5] [8]. GraphQL provides an approach for developing web-based APIs. The main difference and most important one between REST and GraphQL is the way the clients fetch the data. As can be seen on Figure 1, in the case of REST, we get or post data using different endpoints for different entities while when using GraphQL we hit only one endpoint. It is not tied to any specific database or data storage mechanism. GraphQL servers are available for most backend frameworks and programming languages used today. In this paper, in Section 2 the GraphQL in a nutshell will be explained. In Section 3, a review of GraphQL will be presented with it's advantages and disadvantages. Section 4 will explain some of the most used protection mechanisms and Section 5 will give an overview of GraphQL Subscription option. An explanation on how big companies implemented GraphQL in their platforms is presented in Section 6. The concluding remarks are presented in Section 7.

2 Graphql

In this section, we will explain about the core characteristics of GraphQL. It uses a strong type system to define the capabilities of an API. The types that are exposed in an API are written down in a human-readable schema using the

Schema Definition Language (SDL). This schema defines a hierarchy of types with fields that are populated from the application back-end data stores[11]. This schema acts as a contract between the client and the server, defining how a client can access the data using the API. After defining the schema, the backend and frontend teams can continue working without any additional communication about this topic because they are both aware of the definite structure of the data[4]. This is one of the biggest benefits of using GraphQL compared to REST architecture. When using REST, server applications offer a list of API endpoints that can be called by the clients. Unlike REST, GraphQL servers export a database which can be queried by clients. This is possible because of the defined schema, which is a multi-graph [1]. In this multi-graph, the nodes are objects, which referre to entities from real life, and these objects have a list of fields[7]. Each of the fields has a type and a name. Edges in the graph appear when there is a relation between two objects, i.e. one object as a field has another object type[16]. For better illustration of the usage of this language, we will present a simple application schema with three object types: `Book`, `Author` and `Publication`.

```
type Book {
    id: ID!
    title: String!
    author: Author
    content: String!
    publication: Publication!
    year_of_publication: Int
}

type Author {
    id: ID!
    name: String!
    email: String
    year_of_birth: Int
    author_on: [Book!]!
}

type Publication {
    id: ID!
    name: String!
    published: [Book!]!
}
```

Fig. 2. Schema with Book, Author and Publication types

The object types are defined using the key word type. As you can see on Fig. 2, the Book object type has six fields: `id`, `title`, `author`, `content`, `publication` and `year of publication`. These fields are the only fields that can appear in any part of a GraphQL query that operates on the `Book` object type. The first field, `id`, is of type `ID` and it represents a unique identifier that is often used to fetch an object. This type is serialized as String, but going this way means that it is not human-readable. The exclamation mark (!) that is presented after the field type means that the field is non-nullable[14]. So, the GraphQL service whenever you query that field will give you a value. The third and fifth fields, `author` and `publication`, are of type `Author` and `Publication` that are other object types in our schema. The other fields are of scalar types `String` and `Int`. `Author` and `Publication` object types have the same ID field type and the other fields to explain the exact entity characteristics. In both objects we can see the array of `Book` objects. These fields are also non-nullable. The fields defined like this means that the list can not be null and also it can not contain null items.

If we declare the list like this

```
    author_on: [Book!]
```

it will mean that the list cannot contain null items but the field can be null. If the exclamation mark is positioned outside the brackets, without having exclamation mark inside the bracket, that means that the list can not be null but it is allowed to be empty or contain null items.
When we use REST APIs the data is fetched through specifically defined endpoints. This endpoints has a well-defined structure for access and also for the data that they return [2]. But, when we talk about GraphQL, this is radically different. All the endpoints that are defined in REST API with the defined data structure are replaced in GraphQL with a single endpoint. This is crucial because the structure of the data that is returned when using GraphQL is not fixed. The client of the API has the opportunity to ask the server for the exact data it needs using the query. This is one of the most important features of GraphQL. In addition to the defined scalar types, the GraphQL schema can contain more than just those types. The query and mutation are the other most used types. Every GraphQL schema must contain a query type. This type is mandatory.

```
type Query {
    book(id: ID!): Book
    allBooks: [Book]
    author(id: ID!): Author
    allAuthors(last: Int): [Author]
}
```

Fig. 3. Query options

The query type represents the entry point to the GraphQL API data. This query object, as showed in Figure 3, defines what the client can ask for from the database and also the way that the client should do that (provide the arguments if needed) [18]. In Fig. 3, book is a query that accepts a non-null ID as argument and will return the `Book` object that has the provided `ID` as `id`. For example, the `allBooks` query does not take any arguments and returns all the books that we have records fro in the database. The other queries that are for the `Author` model, also accept the `ID` as an argument so it can return some specific object and query for returning all the objects. Using this query's options, the clients can ask for the exact data they need.

```
query BookTitle{
    book(id: 123){
        title
    }
}

query BookTitleAndAuthor{
    book(id: 123){
        title
        author{
            name
        }
    }
}

query BookTitleAuthorPub{
    book(id: 123){
        title
        author{
            name
        }
        publication{
            name
        }
    }
}

query BooksByAuthor{
    author(id: 321){
        name
        author_on{
            title
        }
    }
}
```

Fig. 4. Example of clients queries

On Fig. 4 is example of the four queries where a client asks for specific data. In the first query, `BookTitle`, the client asks only for the title of the book that has the provided `id`. We will not get all the information about the book like content, author etc., only the title. This way we get only the informations that we will

use. In addition, the second query shows the example where we need the title of the book and the name of the author of that book. We get this information with only one call. With the third query, the client asks for the title of the book, name of the author of the book and the name of the publication that published the book. In the last query example the client asks for some specific author with provided id and only the title of all his books. This is a very practical and easy way of fetching data. Going this way we make less calls compared with REST API architecture and we get only the fields that we need, but we will talk more about this in the next section.
To make all these query's work, we need to define a resolver function for each query that is defined in the `Query` type [3][9]. The resolver function need to be defined for every field in the schema. This function is called each time when some object type needs to be retrieved by GraphQL server. Every GraphQL server has an execution algorithm. The implementation of this algorithm is what transforms the query from the client into the results that need to be returned, by going through every field in the schema executing their resolver function to determine its result. If the field produces some of the scalar values like for example a string or number then the execution is done. But in the situations when the field produces an object value, in that situation the query will contain another selection of fields which apply to that object. This will continue until scalar values are reached because the GraphQL queries always end at scalar values.

```
type Mutation{
    addBook(id: ID!, title: String!, author: Author,
        content: String!, publication: Publication!,
        year_of_publication: Int) : Book
}
```

Fig. 5. Mutation for Book object

So far we have talked about how to define the scheme and the ways in which the client will be able to access the data he needs. Now, let's explain how the client will be able to add, update and delete data. This is possible using the mutation type, as shown on Figure 5. The GraphQL server may or may not include a mutation type, it depends on the implementation. Compared to REST architecture, this is similar to POST, UPDATE and DELETE requests. In REST, we do not use the GET request to modify data by convention. The situation with GraphQL is the same, but if we like, we can make any query to be able to make some data modification. But that is not a practice. Instead, for making changes in data we use mutation. Same as queries, the client can ask for nested fields if the mutation field returns an object type. This can be useful for fetching the new state of an object after update operation.

3 GraphQL in parallel to REST

We will use this section to explain the advantages and disadvantages of using this API query language. The two most important advantages that we will start with are number of API endpoints and the fields that the returned JSON response has. When using a REST API, we have specific endpoints that we access for fetching, creating, uploading or deleting data. For the Book example we presented earlier, the endpoints are shown in Figure 6.

```
/books/book_id/
/books/
/authors/author_id/
```

Fig. 6. Example REST endpoints

Using the first endpoint, we get the information about the specific book that we will specify in the URL. The second API endpoint returns all the books we have records for, the third endpoint return the information about the specific author that we specify in the URL. Going this way, we need to create endpoints for all the necessary data that we will use in our app. These endpoints have a specific structure and we need to follow it within the constraints of REST. All this endpoints that we have while using REST are replaced with the single endpoint when using GraphQL. This is very practical in the development process and we avoid the situation of uncertainty which API we should call. In almost all of the discussions covered about this situation with reducing the number of endpoints, there was also bad examples and according to that incorrect conclusions. It is not adequate for a test case in which you want to prove the reduction on number of endpoints to take the applications that are simple and with only one API endpoint. That way, because one endpoint can not be mapped in less that one GraphQL endpoint, leads you to wrong conclusions and can affect future research.
Beside the number of endpoints, let's see and discuss about the returned JSON response and the number of fields that it contains. When using the REST architecture over and underfetching are one of the most common problems. This happens because the client is making a request to a specific endpoint and gets fixed data structures. This data in most situation contain more data that the client app needs in that particular situation. This problem is called overfetching. Lets explain this in details with the already mentioned app about the books. Imagine that we have a screen in which we have a list of all the books represented only with their name and name of the author. If the app uses the REST API, then the client will call the allBooks API endpoint and will fetch all the books that we have records for in the database. But, the response will contain more information that the app actually needs. The response beside the title and author name will additionally has the information about the id of each book, year and name of the publishing house. The app will only use the information that should be shown on the screen and the other data in the response will be

discarded. Nowadays, when we have a lot of apps and the app speed is very important, fetching all that data will take time and at the end the app will not use all of it. One of the possible solutions to this problem is to design that particular API endpoint to return only the data that the app will use. But, after some time if there is a change in the app requirements or there is a change that needs to be implemented on the UI and not only the title of the book and the author but we also need to show a year when the book was published, then beside the changes in the frontend of the app we will need to make changes in the backend, to restructure the API to return the year of publishing of the book too. Neither practical nor correct. The next problem that is also close connected here is the problem of underfetching that we mentioned before, and it means that the client can not get all the required data from specific endpoint[19]. In this situation, the client should make additional requests to different endpoints to get all the data that he need in that situation. For illustration, if we want to show the books per author, then we should first call the endpoint to get all the authors, and then for each author id, we should call the other endpoint for getting the books for that specific author.
Using GraphQL, these two problems are solved. According to the app needs, the client will fetch only the data that is needed on that particular screen using the possibilities that the query language offers. Any changes in requirements or UI will be quickly fixed, only with changes in the query and without any intervention on the server side. The underfetching problem is also solved with using a nested object query using the object relationship between the two objects. On this way, we add on query depth, that will be explained in the next section. Another important thing that should be mentioned here is about the backend analytics that GraphQL has. According to this feature that GraphQL offers, we will have an accurate insight into what customers most often ask from our API. With this informations, we can improve the performance of those fields that are most in demand while some of the fields may be deprecated. Additionally, it may affect the decision to add new query options and features.
Another advantage of using GraphQL is it's detailed error messages [6]. If an error occurs while processing some GraphQL query, than the server will provide to the client a detailed error message referring to the exact part of the query that has a fault. The client can choose what kind of error message to receive, be it a stack trace, an application-specific error code or a plain text. Compared to REST based API, this is more elegant and understandable instead of checking the status code of the response and trying to understand what went wrong with the request. Usually, some of the REST API endpoints even not return the right status code and lead the client in a wrong direction while trying to solve the problem.
Using GraphQL, we can define some data not to be exposed in defined circumstances making them on some way private. This is not a case when using REST where we have to decide to show all that particular data or nothing, not being able to return at least the public parts of that particular data.
When working with REST, in addition to writing the code for the endpoints

and defining their behavior the developers need to write an API documentation for the users that will use their API endpoints to have clear understanding of how it works and what data they receive. Clean, understandable and up-to-date documentation is what every client needs when try to integrate some API endpoints. This is not a problem any more when using GraphQL because it keeps the documentation up-to-data all the time. With every change in the code, field, type or query changes, the documentation will be also updated.
All this mention characteristics together with the ability of sharing code, fast application prototyping and the perfect solution for microservice architecture give the part of the most important advantages that GraphQL users enjoy.
GraphQL has some disadvantages as well. Giving the opportunity to clients to ask for the data they need, can be a double-edged sword. The ability of nesting fields can add on query depth and will make it complex and time consuming for GraphQL to return a response. For complex queries, REST API will be a better solution, even if it requires fetching the required data with few requests.
Also, for the clients to ask for the data they need, they would need time to clearly understand the schema and the fields that are available to them. Instead of this, while using REST the only thing we should know are the endpoints. Here, it's good to be mentioned that while it is the best solution for microservice architecture, using GraphQL for small applications will be not the appropriate solution. REST seems to be the choice because there will be no great benefit from using the GraphQL querys in that situations.
The last important disadvantage that should be mentioned about GraphQL is the caching mechanism. With the caching we store copies of frequently accessed data in several places along the request-response [10]. The implementation of caching is also important because than way we increase the performance of the application and also decrease the load on the server and reduce bandwidth and latency. The REST APIs do not have problems with caching since the caching is one of the constraints of the REST, but implementing caching in GraphQL is not an easy thing because of the single endpoint. There are some solutions for this like using a batching technique, persisted queries etc.

4 Protection mechanisms

Since we explained the basics and advantages and disadvantages of using GraphQL, in this section we will cover the part about how the GraphQL cope with evil clients and big nested queries. We will explain some of the protection mechanism that keeps the server from going down.
Using a timeout for query execution is the most simple strategy that can be implemented. Going this way, the server does not know anything about the query, all he knows is the maximum time allowed for one query to be resolved. The developer can configure the server and set the time in which the query will get chance to be executed. This is simple to be implemented and it is widely used, but sometimes the server may still have some damage even the execution reach the execution time.

In the previous section when the nested queries were explained, we mention the query depth. The developer, knowing the schema, can affect on the maximum depth that the query can has. If the query from the client exceed the maximum depth that the developer set, than the query will be rejected and the client will receive the error message that explain this problem. This strategy is better that the timeout strategy because here the execution on the query will not even start if the depth of the query exceed the maximum depth set by the developer.
But, this is still not the safest way. While we make some restrictions with this to protect the server and we reject the queries that exceed the max depth, there can be still come to execution some query that does not exceed the set up depth but it may be some complex query that requires big amount of complex compute fields.
To avoid executing complex queries, the GraphQL offers a query complexity mechanism. This is performed with defining the complexity of each field and on that way calculate the total complexity of the received query. Different fields can have different complexity. This is useful when we have objects that are more complex than some others. On that way, the server can easily calculate the query complexity and reject the queries that exceed the defined query complexity. If the developer did not set a field complexity, it will be set to 1 by default. There is an option for complexity to be dynamically calculated in the runtime, for example depending on how many child items we have at the moment about that object and will be fetched with that call.

5 Subscription

Most of the applications that we use today have some real time functionality in it that we receive as notifications. Whether it's about some product that is back in stock, someone send us a message, the online ordered parcel arrived in neighborhood etc, the applications that handle this things should be updated in real-time when this events happens so they can inform the users. GraphQL subscription helps in this cases with the opportunity to send the only the data that clients asked for after some specific event happens. This is possible as a result of establishing a bi-directional communication channel between the client and the server using a subscription query [12]. In this query, the client specify which data he wants to receive back from the server when some specific event is triggered. Subscriptions are very similar to the queries that we talk about before, but the difference is that here the client did not receive the data immediately but when the event specified in the subscription query is triggered. The subscriptions need to be added in the schema as shown on Figure 7.

```
type Subscription {
    bookAdded(author: Author!): Book
}
```

Fig. 7. Example subscription when book is added

In this first subscription option the client can subscribe on the new book added from the author specified, this is useful for creating a notifications for new books of some favorite author of the client, and the second option is for making subscription when new book is being published by some publication house. When some of these events happened, then the query that is stored is being execute and the client receive the information that he defined in the query when make the subscription.

6 How big companies implement GraphQL

Beside it is a new technology, GraphQL has already been implemented in a lot of famous and top used applications. We have choose three of them that we want to present how they implement GraphQL in their code. The transformation to GraphQL was not an easy step, especially because they have a lot of users and data. Their API was using REST architecture before and on that way making it more hard for developers to fetch the data they needed.

6.1 Facebook

We will start with analyzing how Facebook uses GraphQL because they are its creators. The way they are using GraphQL can be expressed in three ways. Facebook considers the data in the application as a graph. This means that post, comments, profiles are object types representing the nodes and there is a relationships between them making the edges in that graph [15]. GraphQL represents the single source of truth. This means that adding a new feature in the application should not affect on reimplementing the data fetching, security etc. The third way is the thin API layer. The idea behind this is, beside GraphQL offers a practical and easy way of fetching data, the backend does not need to be build around it. On the contrary, the backend together with the business logic should be built separately and then the API layer should be put on the top of it. On this way also, there can be made changes in the API without affecting on business logic and vice versa.
Authorization and permissions are another topic that should be mentioned here. This is done in a separate layer called "business logic" that is positioned under the external interfaces and over storage layer. At Facebook, each type of data has its own model object. So, there is a single function that is called when the clients ask for an objects from the server and the authorization logic is implemented here. This function by convention accepts an object called viewer which contains all the information and permissions roles that the client that asks for data has. Caching is solved on a very simple way, remembering the objects that was already fetched. If object with that ID was already fetched, then we don't make a new request but just reuse the saved one. The DataLoader does this by default.

6.2 Twitter

Twitter began using GraphQL step-by-step. Because Twitter is a microservice company, GraphQL was right choice for their platform. As many other companies, Twitter also uses Thrift, the Remote Procedure Call system. Using Thrift, the communication between microservices was using types because each of the services exposes an typed interface. These types were not GraphQL types, but it helps GraphQL to fits in their architecture as shown on Figure 8 [17].

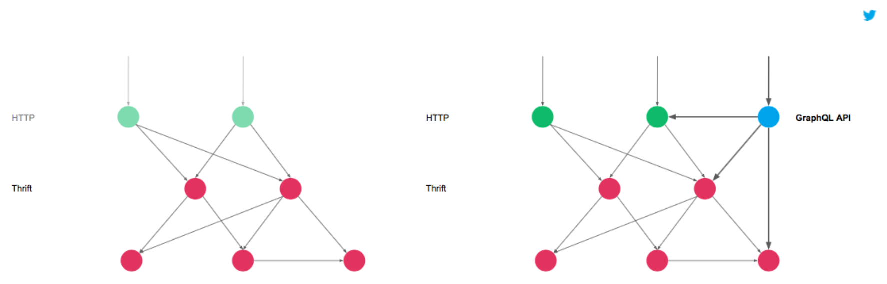

Fig. 8. Twitter before (left) and after (right) GraphQL

At first, they implemented in TweetDeck and Twitter Lite, and then on their Android and iOS applications. There are many challenges that Twitter faced with when they started implementing GraphQL. By its nature, GraphQL returns status code of 200 even when there is something wrong with the query. Twitter manage this problem with not tracking the status codes but the number of exceptions generated during the query execution [17]. From the "over-nested" and complex queries they cope on that way they measure the query complexity and depth. First, the query complexity is checked and if it exceeds the defined complexity, it is rejected. Then, they check the query depth.
Twitter copes with the authentication with the help of the Twitter Fronted layer. This layer is positioned between the Twitter services and the outside world. All the requests come to this layer and their validation is done here. After the authentication, the GraphQL API redirects the bundle to the other services included so they don't worry about authentication. Having the lower-level services responsible for authorizations, the GraphQL API server doesn't worry about any of this.

6.3 Coursera

Coursera uses GraphQL with a middleware which function is to dynamically translate their REST API to GraphQL. Several round trip the clients need to

perform to fetch some specific data from the Coursera REST based API. They start their migration to GraphQL to solve some of their problems and to enable better experience to the clients. They try to migrate to GraphQL two times, but they failed in the first attempt. The reason for this was because of a bug, their course catalog was rolled back to a previous version, causing the schema that they used in GraphQL not to be synced with the data [13]. In situation like this, manually updating the schema was almost impossible.
After the failure, in their second attempt, they went with a different approach. Instead of GraphQL, the REST APIs were threated as the source of truth and having the GraphQL schema based on it. They conclude that the GraphQL layer should be automatically and deterministically built based on the current architecture.
The result was services able to dynamically inform what REST resources are running on it. At last, they defined a GraphQL to REST request translations, so they get a functional GraphQL server, never more than five minutes out of date because the GraphQL server every five minutes pings every downstream service.

7 Conclusion

To sum up the things represented in this paper, we can firmly say that using GraphQL may lead to a completely new way of developing the applications. Compared with the most used API architecture, REST, GraphQL reduces the JSON size more that 90%. This is an important feature in this "application battle" environment. The reason for this is the nature of REST where the client receives all the data associated with that endpoint, while the core of GraphQL is to return only the fields requested by the client. On this way, we maximize the efficiency and there is no discarded data. Also, there is no more problem of over- and under-fetching with GraphQL because all the data needed is fetched with a single call, using the option of nesting the fields in the query.
Before choosing GraphQL, should be think twice, mainly because it may not fit on the application purposes and architecture so the GraphQL advantage will not be experienced. Will REST experience the SOAP destiny, remains to be seen. However, we think that is too early to talk about REST retirement because it is still the most used one worldwide. Proclaiming the GraphQL as a better REST has yet to be proven.

References

1. G. Brito, T. Mombach, M. T. Valente : Migrating to GraphQL: A Practical Assessment, https://homepages.dcc.ufmg.br/ mtov/pub/2019-saner-graphql.pdf
2. G. Brito, M. T. Valente : REST vs GraphQL: A Controlled Experiment, https://arxiv.org/pdf/2003.04761.pdf
3. P. Mbanugo : GraphQL: Schema, Resolvers, Type System, Schema Language, and Query Language, https://www.telerik.com/blogs/graphql-schema-resolvers-type-system-schema-language-query-language, Last accessed 23.04.2020

4. O. Hartig, J.Perez : Semantics and Complexity of GraphQL , http://olafhartig.de/files/HartigPerez_WWW2018_Preprint.pdf
5. https://graphql.org/, Last accessed 27.05.2020
6. GraphQL: Core Features, Architecture, Pros and Cons, url: https://www.altexsoft.com/blog/engineering/graphql-core-features-architecture-pros-and-cons/, Last accessed 05.05.2020
7. O. Hartig, J.Perez : An Initial Analysis of Facebook's GraphQL Language, http://repositorio.uchile.cl/bitstream/handle/2250/169110/An-initial-analysis-of-facebooks-GraphQL-language.pdf?sequence=1isAllowed=y
8. R. Taelman, M.V. Sande, R. Verborgh, GraphQLLD: Linked Data Querying with GraphQL: https://biblio.ugent.be/publication/8578324/file/8579408.pdf
9. D.M.Vargas, A.F.Blanco, A.C.Vidaurre, J.P.S.Alcocer, M.M.Torres, A. Bergel : Deviation Testing: A Test Case Generation Technique for GraphQL APIs, http://bergel.eu/MyPapers/Mene18a-GraphQL.pdf
10. T. Poniatowicz, GraphQL vs REST - Caching, https://blog.graphqleditor.com/grapqhl-vs-rest-caching/. Last accessed 30.04.2020
11. Y.W. Kim, M.P. Consens, O. Hartig : An Empirical Analysis of GraphQL API Schemas in Open Code Repositories and Package Registries. http://ceur-ws.org/Vol-2369/short04.pdf
12. P. Mbanugo, An Introduction to GraphQL: Subscriptions, https://www.telerik.com/blogs/introduction-to-graphql-subscriptions. Last accessed 24.05.2020
13. W. Wattearachchi, The Significance of GraphQL — Part 2 (How Facebook, Coursera and Artsy use GraphQL?), https://blog.usejournal.com/the-significance-of-graphql-part-2-how-facebook-coursera-and-artsy-use-graphql-86abe9ab9cb2. Last accessed 22.05.2020
14. M. Biehl : GraphQL API Design 2nd edn. API-University Press, (April 2018)
15. S. Stubailo, How Facebook organizes their GraphQL code, https://www.apollographql.com/blog/graphql-at-facebook-by-dan-schafer-38d65ef075af. Last accessed 19.05.2020
16. O. Hartig, J. Hidders : Defining Schemas for Property Graphs by using the GraphQL Schema Definition Language, https://www.researchgate.net/publication/333888921_Defining_Schemas_for _Property_Graphs_by_using_the_GraphQL_Schema_Definition_Language
17. GraphQL at Twitter, https://about.sourcegraph.com/graphql/graphql-at-twitter. Last accessed 19.05.2020
18. J. Werbrouck, M. Senthilvel, J. Beetz, P. Pauwels : Querying Heterogeneous Linked Building Data with Context-expanded GraphQL Queries, https://www.researchgate.net/publication/334083989_Querying_Heterogeneous _Linked_Building_Data_with_Context-expanded_GraphQL_Queries
19. D.A. Kus, I. Koren, R. Klamma : A Link Generator for Increasing the Utility of OpenAPI-to-GraphQL Translations, https://www.researchgate.net/publication341478510_A_Link_Generator_for _Increasing_the_Utility_of_OpenAPI-to-GraphQL_Translations