Comparison of Graph and REST APIs

Marko Gluhak[1], Marjan Heričko[1]

[1] *University of Maribor, Faculty of Electrical Engineering and Computer Science, Koroška cesta 46, Maribor, Slovenia*

Abstract

As APIs have become more important it is crucial to offer flexibility in their use. REST API has been the dominating architectural style over the past two decades. With the API Gateway and other improvements, it has been able to keep pace with the ever-changing customer demands. Novel approaches such as the Graph API offer such features as a native property of their design philosophy. They offer benefits to API consumers and providers which are still not understood in full. In this paper we explore Graph API, how it works, performs, and what are the differences between it and the standard REST API. We experiment and assess the performance of GitHub, GitLab and Star Wars APIs in REST and Graph forms. Conclusion is that neither approach is ideal in all situations, but they supplement each other where the other falls short.

Keywords

REST API, Graph API, API principles, API performance

1. Introduction

As APIs (Application Programming Interface) have proliferated in the IT industry as the standard for communication between applications, the REST architectural style has become the favorite for lacking strict rules in favor of guidelines to follow [1]. In the recent years of big data, there have been different optimizations of the architectural style in the form of API Gateways and API Management platforms [2]. These offer a central access point to an API in the form of a product, tailored endpoints to suit the use cases of customers, etc. A novel approach pioneered by Facebook [3] called the Graph API has appeared to offer a centralized entry to an API by design. The Graph API aims to bridge the issues suffered by REST developers and clients and not necessarily displace the old architectural standard. Its hype cycle is on the rise [4], as more and more big players are adopting this approach. There have also been efforts to wrap REST services in Graph wrappers to offer the outward behavior of a graph instead of a REST API [5]. It does come with its own shortcomings, however, in the form of new learning curves, demand for quality schema design [6], security challenges [7], and performance tradeoffs [8]. When correctly leveraged, the Graph API can yield great results for those willing to understand and develop such services [9].
The goal of this paper is to better understand the new style of designing APIs. Our motivation is to offer a good starting point to researchers and practitioners for the decision-making process, whether they would like to provide or consume services via REST or Graph API. To this end, we have proposed three research questions:

RQ1: What are the differences between Graph and REST API approaches?
RQ2: Are there performance benefits to using a particular approach when it is appropriate?
RQ3: Are there differences in testing the two APIs?

We will answer the following questions with a synthesis of knowledge, drawing from all the larger digital libraries, namely: IEEE, Web of Science, ScienceDirect, SpringerLink, ACM, and Google

SQAMIA 2023: Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 10--13, 2023, Bratislava, Slovakia
marko.gluhak@um.si (M. Gluhak); marjan.hericko@um.si (M. Heričko)
0009-0000-2185-4011 (M. Gluhak); 0000-0002-1094-0085 (M. Heričko);

Scholar. To understand performance differences, we test three well-known APIs in both REST and Graph forms. Answers are elaborated on in Chapter 4 of this paper. In Chapter 3, we will look at the lessons learned from our research and our experiment. Specifically, focusing on the differences between two approaches, the performance they offer, and how it is perceived in testing Chapter 2 offers a baseline on REST API principles and goes into more detail on the novel Graph API approach, including how it is made up of schemas and how to formulate a request. We by no means offer a comprehensive guide on how to design production-ready Graph APIs, but rather an entry point into a new principle. We also explore the current industry players offering Graph APIs.

1.1. Related work

There has been an attempt to quantify the performance benefits of Graph APIs. Previous works have addressed different aspects of the design approach. The work of Eunggi, Kiwoong, and Jungmee [8] delves into the handling of complexity. Their findings report that REST overtakes Graph when a call is made to more than 20 resources at once. Although their tests are limited to a single system, what is evident is that the graph has a higher standard deviation and lower response sizes. Yet Naman and Ida [10] experiment with a thorough set of tests for various loads of users. They also test for various GET complexities and POST and PUT methods. Their findings are more focused on the API gateway implementations (Ocelot and HotChocolate). They find that REST is more performant, outperforming graphs in nearly all situations and loads. The same was found by Armin, Benny, and Takaichi [11] in their experiment. Graph does show better results in resource utilization with both memory and processing power. Contrary to previous works, the contributions of Jaime, Evelin, and Andres [12] show that Graph is the most performant option. They test for different programming languages (C#, Java, and PHP) and various user loads, and graph is better in response times and throughput. The work of Suresh et al. [13] presents a Graph solution for transforming a REST API into Graph for the sake of achieving performance, resource utilization and solving other issues of REST that are presented in this paper.

2. Characteristics of REST and Graph APIs

Both REST API and Graph API approaches aim to accomplish the same goal: to provide clients with requested data. To better understand them, we will first present how each provides services to customers and some of their key principles. At the end of this section, we will also discuss some of the key players that are in the market for providing Graph APIs.

2.1. REST API

In his thesis, Roy Thomas Fielding first coined the term "Representational State Transfer (REST) as an architectural style for distributed hypermedia systems. REST provides a set of architectural constraints. When applied, the constraints emphasize scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. [1]. There are six core design principles when designing REST interfaces: a uniform interface, client-server, stateless, cacheable, layered systems, and code on demand.

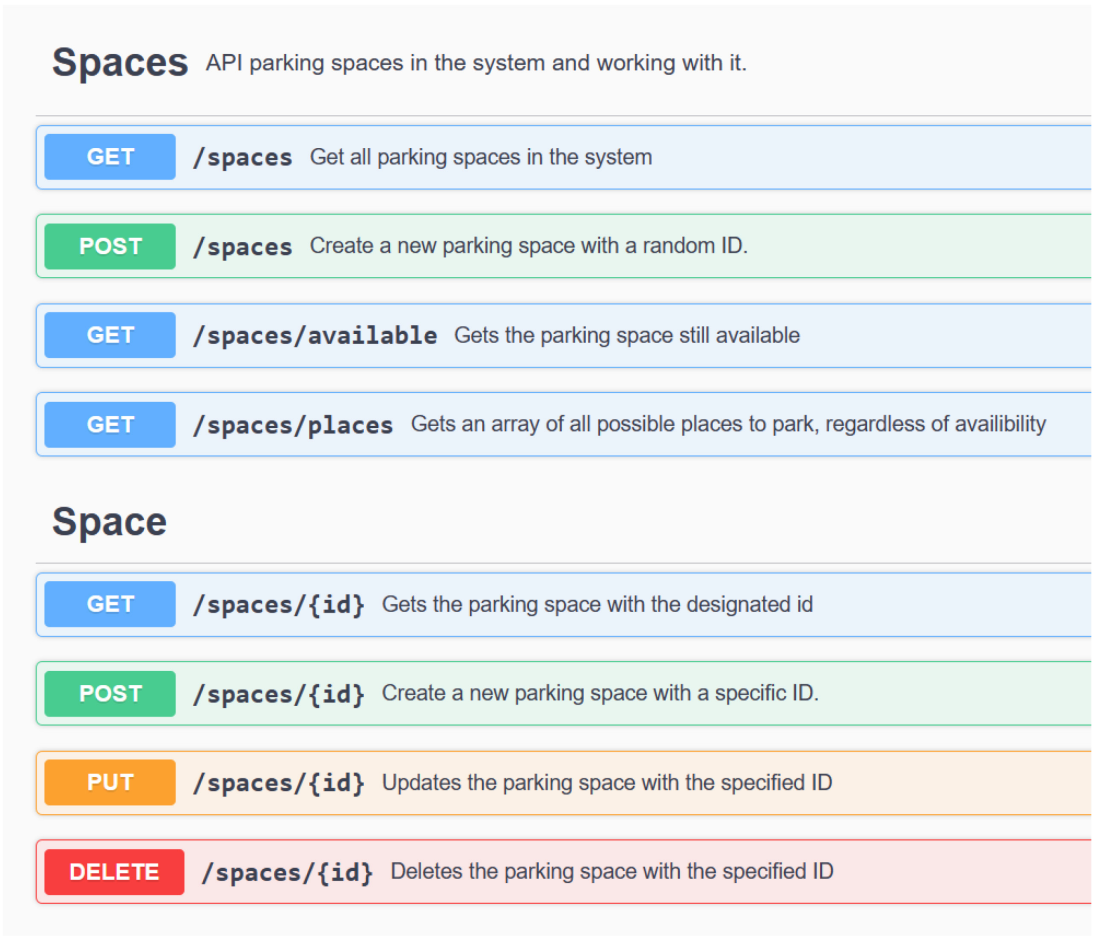
Among these, the first is the most fundamental to a RESTful API. Providing a uniform interface is the basis on which the API is built. It should adhere to the following constraints: identification of resources, manipulation of resources through representation, self-descriptive messages, and hypermedia as the engine of application state (HATEOAS).

This then brings us to how we expose our data over such interfaces. Every resource has its own Uniform Resource Identifier (URI) that corresponds to an exposed Uniform Resource Locator (URL) [2]. This puts an emphasis on how we design paths to our resources. An example path of an API looks like this: https://foo.com/api/employees/123. It retrieves a single employee with an ID of 123 from our API, which stores employees. We receive the full employee object only if we want to know his name. If we now want to know his active projects, it depends on how we have designed our API. It could be as simple as calling https://foo.com/api/employees/123/projects. Or it could be a separate call to another API with our employees' ID. If we only need to know how long a project the employee is currently working on is going to take, we need to know where that data is stored and how the tree structure leads to it. Even then, we are inevitably running into the problem of fetching too much or too little data [8]. A major drawback of current REST APIs.

One of the remedies has been to implement patterns via the API Gateway. These patterns allow us to abstract how our APIs are connected between themselves and provide use-case-driven APIs. Some of the patterns include the API Facade, API Composition, Two-phase Transaction Management and Synchronous to Asynchronous Mediation [2]. Most API management platforms allow us to easily create these use cases without writing code through the developer portal. However, these use cases still require maintenance and resources [2].

Another undefined area with REST is its service description. It's now been standardized with the Open API initiative, but there are still many tools that go about generating the description. Swagger, as viewed in Figure 1, is one such tool. It offers us a description of a basic parking service. What endpoints does the API respond on and what are the HTTP methods for achieving them? From the figure, we observe the four basic verbs. GET, which is used for fetching resources. POST for creating them, PUT for updating them, and DELETE for deleting them. There are other verbs such as HEAD, TRACE, PATCH, OPTIONS, and CONNECT that are less commonly used.

In the next chapter, we'll present how the Graph API handles these areas.


Spaces API parking spaces in the system and working with it.

GET /spaces Get all parking spaces in the system

POST /spaces Create a new parking space with a random ID.

GET /spaces/available Gets the parking space still available

GET /spaces/places Gets an array of all possible places to park, regardless of availibility

Space

GET /spaces/{id} Gets the parking space with the designated id

POST /spaces/{id} Create a new parking space with a specific ID.

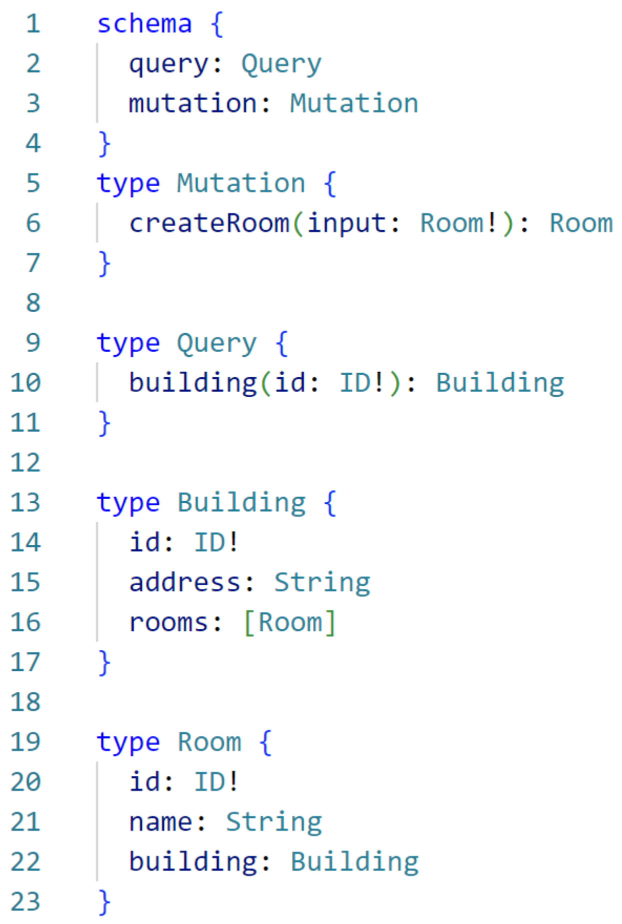
PUT /spaces/{id} Updates the parking space with the specified ID

DELETE /spaces/{id} Deletes the parking space with the specified ID

Figure 1 Example REST API description with Swagger. Source: author's contribution.

2.2. Graph API

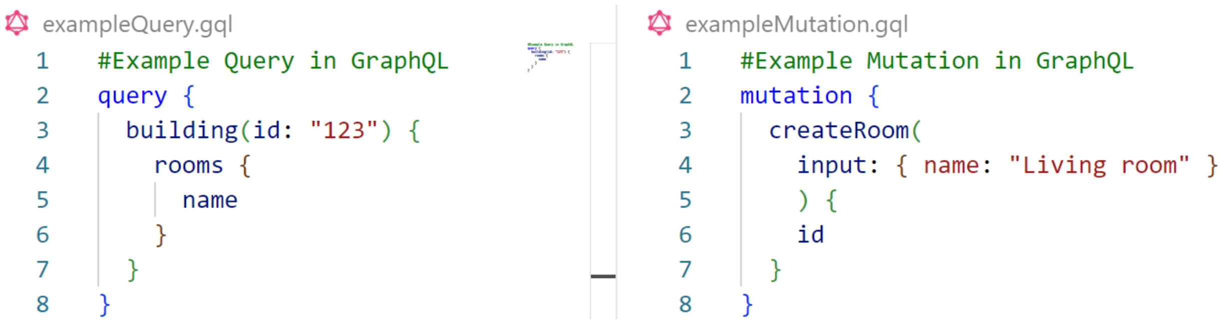
Graph API offers an alternative to the REST approach to building APIs. Graph Query Language (GraphQL) is a query language that produces APIs that facilitate flexible access to data. Although using GraphQL is not necessary to provide a Graph API, it is often the case. One of the key advantages Graph APIs have over REST APIs is that clients receive only relevant data and can create new queries that suit their needs without any changes on the backend [14]. One of the main identified necessities for this approach to work is to have interconnected data and defined relationships between them [6], [15]. This requires well-defined schemas for clients to be able to formulate requests efficiently. An example of a GraphQL schema can be observed in Figure 2. We have defined a schema using the Schema Definition Language (SDL). The schema defines two possible operations, query and mutation, along with their corresponding root elements, Query and Mutation. In our case, the names are the same, so the schema field could have been omitted and it would remain a valid scheme. What we have done with this is explicitly name the root elements of our operations [16]. Along with these operations, we have also defined two types of objects, Building and Room, that have their own unique fields. We have also created a relationship between them. That is done by creating an array of rooms in the building and a building field in the Room.



```
schema {
  query: Query
  mutation: Mutation
}
type Mutation {
  createRoom(input: Room!): Room
}

type Query {
  building(id: ID!): Building
}

type Building {
  id: ID!
  address: String
  rooms: [Room]
}

type Room {
  id: ID!
  name: String
  building: Building
}
```

Figure 2 Example of a simple GraphQL schema. Source: author's contribution.

Now that we have a simple schema, we may look at how we can extract the data from an API that would be exposing its data over it. In Figure 3, we now see a query and a mutation being formulated. It is only valid because of the schema we have defined. The query would only return the names of rooms in the building with an id of 123. The mutation would create a room with the name "Living room" and give us back the ID it was created with. Why GraphQL is so intuitive for clients to adopt is the ability to introspect the schema behind the API. This also allows developers to build tools around it, as intelligent suggestions can be made about the API itself at runtime [16].


exampleQuery.gql

```
#Example Query in GraphQL
query {
  building(id: "123") {
    rooms {
      name
    }
  }
}
```

exampleMutation.gql

```
#Example Mutation in GraphQL
mutation {
  createRoom(
    input: { name: "Living room" }
  ) {
    id
  }
}
```

Figure 3 Examples of GraphQL Query (left) and GraphQL Mutation (right). Source: author's contribution.

These requests are normally enveloped in the body of a POST request. Although there is precedent for mapping certain requests as GET parameters, there are certain limitations to this approach, but the benefits of keeping the API discoverable sometimes outweigh them. Facebook is the creator of GraphQL, and they are using GET for most simple requests that do not require complex queries [17]. This also comes into play when we begin to add authentication and access tokens into the mix [15]. To support these, they must also be included in the schema, which further adds to the complexity of designing a good GraphQL schema [6].

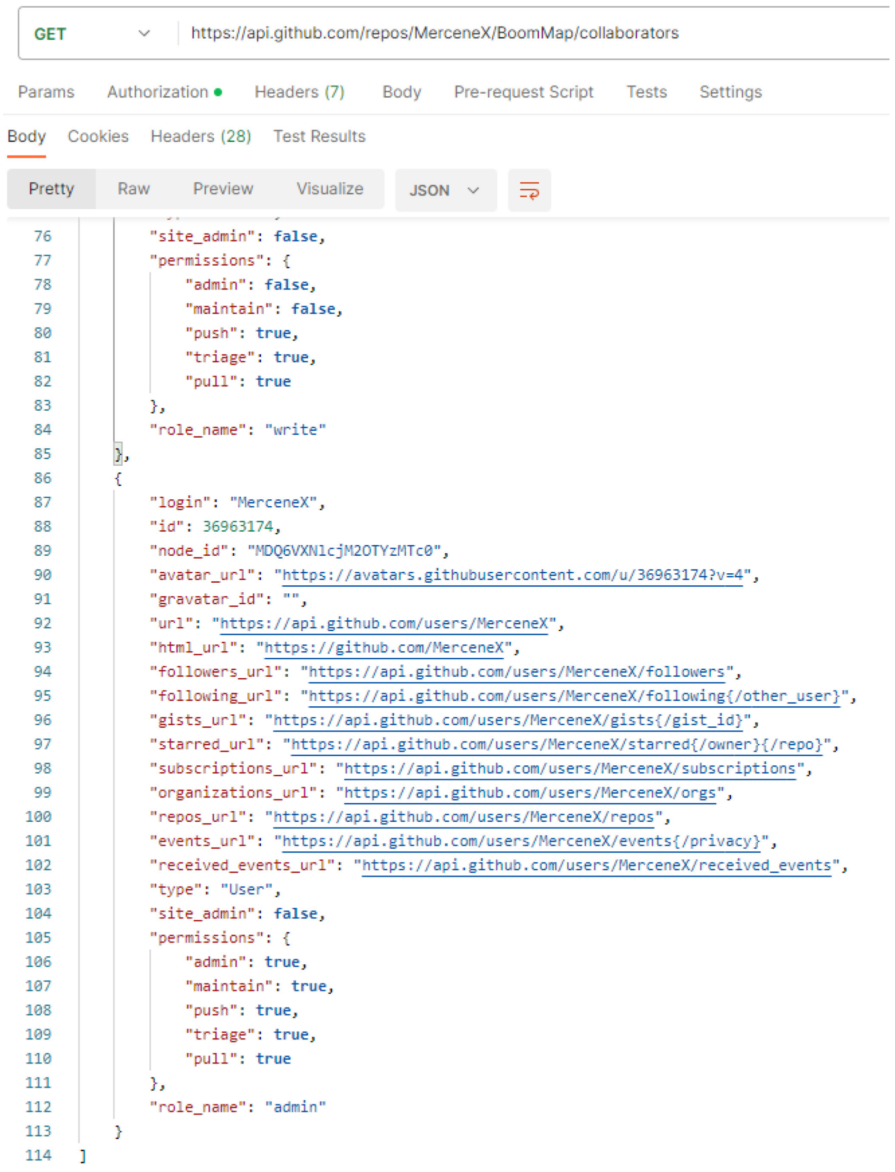
As Gartner has pointed out, Graph APIs are in the Innovation Trigger stage [4]. Meaning early adopters such as Facebook have been in the game for a while. Their need for the technology eventually led to new tools such as GraphQL itself when some other early attempts led to other shortcomings [7]. Currently, most of the big players in API management platforms are also offering their own take on the Graph architecture. Microsoft Azure [18], AWS [19], and Google Apigee [20] are all offering some form of support for graph APIs. But as it currently appears, the Graph API will not be replacing the REST API anytime soon. Graph-based APIs are meant to overcome certain shortcomings of REST. Most big adopters of the Graph API are offering it alongside REST. And research has been made in the direction of transforming REST into Graph when reasonable [5], [21-23].

3. Differences between consuming APIs

The key differences in the two approaches to API design that stand out are the philosophies in which the challenges of request performance are addressed. The REST API is usually quicker in providing responses, but they are less tailored to the needs of the customer. It also requires clients to know where certain resources are available. Graph API, on the other hand, exposes a singular endpoint and accepts requests via a query in the body of a POST request.

3.1. Over fetching and under fetching

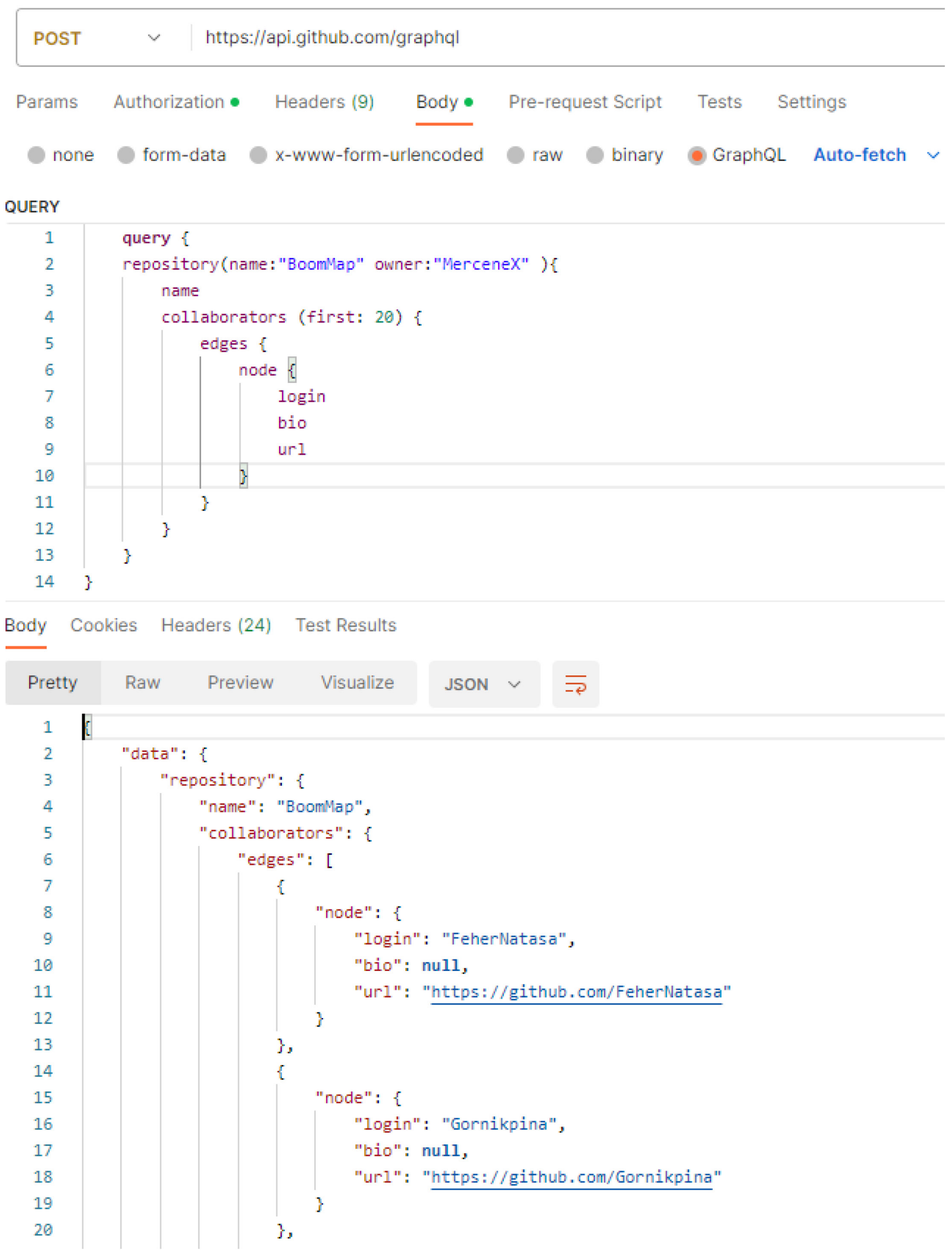
With the REST API, to find the repository information from which we can extract the collaborators, we must visit the following endpoint: https://api.github.com/repos/{owner}/{repo}/collaborators. The {owner} is replaced by the username of the owner, and the {repo} is replaced by the name of the repository we wish to visit. We are presented with a full array of collaborators and 25 fields for each of them, visible in Figure 4. Among them, only two of them are relevant to our needs. This illustrates one of the common problems with REST, as we have no way of applying specificity to our request. It leads to acquiring more data than is necessary for our given needs. What is worse is the fact that this endpoint does not include all the user data but only a subset that the provider decided to include. There is no bio field for a user. This is the problem of "under fetching," where the endpoint does not provide all the data needed for our use case. To obtain this data, we must visit another endpoint, https://api.github.com/users/{user}", where the {user} is replaced with the user we wish to query. Where we are again served with 32 fields, of which only one is relevant to our use case. If we are authenticated with a token from the same user, the number of fields is 43. This is the problem of "overfetching" more data than needed.



```
        "site_admin": false,
        "permissions": {
            "admin": false,
            "maintain": false,
            "push": true,
            "triage": true,
            "pull": true
        },
        "role_name": "write"
    },
    {
        "login": "MerceneX",
        "id": 36963174,
        "node_id": "MDQ6VXNlcjM2OTYzMTc0",
        "avatar_url": "https://avatars.githubusercontent.com/u/36963174?v=4",
        "gravatar_id": "",
        "url": "https://api.github.com/users/MerceneX",
        "html_url": "https://github.com/MerceneX",
        "followers_url": "https://api.github.com/users/MerceneX/followers",
        "following_url": "https://api.github.com/users/MerceneX/following{/other_user}",
        "gists_url": "https://api.github.com/users/MerceneX/gists{/gist_id}",
        "starred_url": "https://api.github.com/users/MerceneX/starred{/owner}{/repo}",
        "subscriptions_url": "https://api.github.com/users/MerceneX/subscriptions",
        "organizations_url": "https://api.github.com/users/MerceneX/orgs",
        "repos_url": "https://api.github.com/users/MerceneX/repos",
        "events_url": "https://api.github.com/users/MerceneX/events{/privacy}",
        "received_events_url": "https://api.github.com/users/MerceneX/received_events",
        "type": "User",
        "site_admin": false,
        "permissions": {
            "admin": true,
            "maintain": true,
            "push": true,
            "triage": true,
            "pull": true
        },
        "role_name": "admin"
    }
]
```

Figure 4 GitHub Rest API response to our request for a list of collaborators for a repository. Source: author's contribution.

Graph API gives us the tools to specify exactly the fields we wish to extract from it. Not only is it available, but it is also mandatory. This could lead to APIs being prohibitive for new developers to adopt. But since the schema is a mediator with its introspection, it allows developers to easily discover which fields they may call. In the Graph API, with a single well-defined query in the body of a POST request, we are presented with a tailored response of only the needed data. Giving the client the power to design a desired response from the API. Such is evident from Figure 5, whereby, by specifying the "login", "bio", and "url" fields, we are presented only with those. Fetching exactly the data we need.



```
query {
repository(name:"BoomMap" owner:"MerceneX" ){
    name
    collaborators (first: 20) {
        edges {
            node {
                login
                bio
                url
            }
        }
    }
}
}
```

```
{
    "data": {
        "repository": {
            "name": "BoomMap",
            "collaborators": {
                "edges": [
                    {
                        "node": {
                            "login": "FeherNatasa",
                            "bio": null,
                            "url": "https://github.com/FeherNatasa"
                        }
                    },
                    {
                        "node": {
                            "login": "Gornikpina",
                            "bio": null,
                            "url": "https://github.com/Gornikpina"
                        }
                    },
```

Figure 5 GitHub Graph API response example to specific field requirements. Source: author's contribution.

3.2. Performance

The key differences in the two approaches also lead us to believe there are performance differences. To evaluate our hypothesis, we have devised several test scenarios on platforms where APIs are available in both REST and Graph forms. Based on real use case scenarios presented in Table 1. We performed tests for all scenarios were using Apache JMeter with settings observed in Table 2.

Settings remained the same across use cases, except for UC2 and UC3 at GitHub where we increased the ram-up period to 10 minutes to spread out requests over. We did this due to rate limiting and we were not able to perform more than twenty requests in succession. We assume this did not impact our results as the aim was not to test load capabilities of GitHub or GitLab APIs. Regardless, our aim was to collect the following metrics:

- Average, minimum, and maximum response times and their standard deviation (milliseconds)

- Size of response (bytes)
- Size of request (bytes)
- Throughput (requests per second) – only Star Wars API

Table 1
Test scenarios of API platforms

	Use case 1	Use case 2	Use case 3
GitHub	Retrieve a repository and all its collaborators (GET)	Create a new issue for a specific repository (POST)	Update the issue, created in the use case 2 scenario (PUT)
GitLab	Retrieve current user's owned projects (GET)	Create a new issue for a specific project (POST)	Update the issue, created in the use case 2 scenario (PUT)
Star Wars API	Retrieve a specific starship (GET)	Retrieve a character and their homeworld with its terrain description (GET)	/

Table 2
Thread group settings for Apache JMeter

	GitHub	GitLab	StarWars
Number of Threads (users)	100	100	100
Ramp-up period (seconds)	2/600/600	2	2
Loop Count	1	1	1

In the case of the Star Wars API, we were able to stress the system because it was a self-hosted instance in a Docker container on a VirtualBox virtual machine with resources listed in Table 3. Hence, we also collected the throughput of the requests.

Table 3
Virtual machine specifications

Property	Value
Processor	4
RAM	4096MB
HDD	25GB
OS	Ubuntu Server 22.0.4 LTS

Results are visualized in Figure 6a, presenting our findings for response times on all three APIs and throughput on the Star Wars API. In Figures 6b and 6c, we skewed the standard deviation visualization for Graph on Use Case 2. This was done because the results were making other results difficult to read and interpret. Actual numbers for standard deviation are 3120,84 and 9759,93.

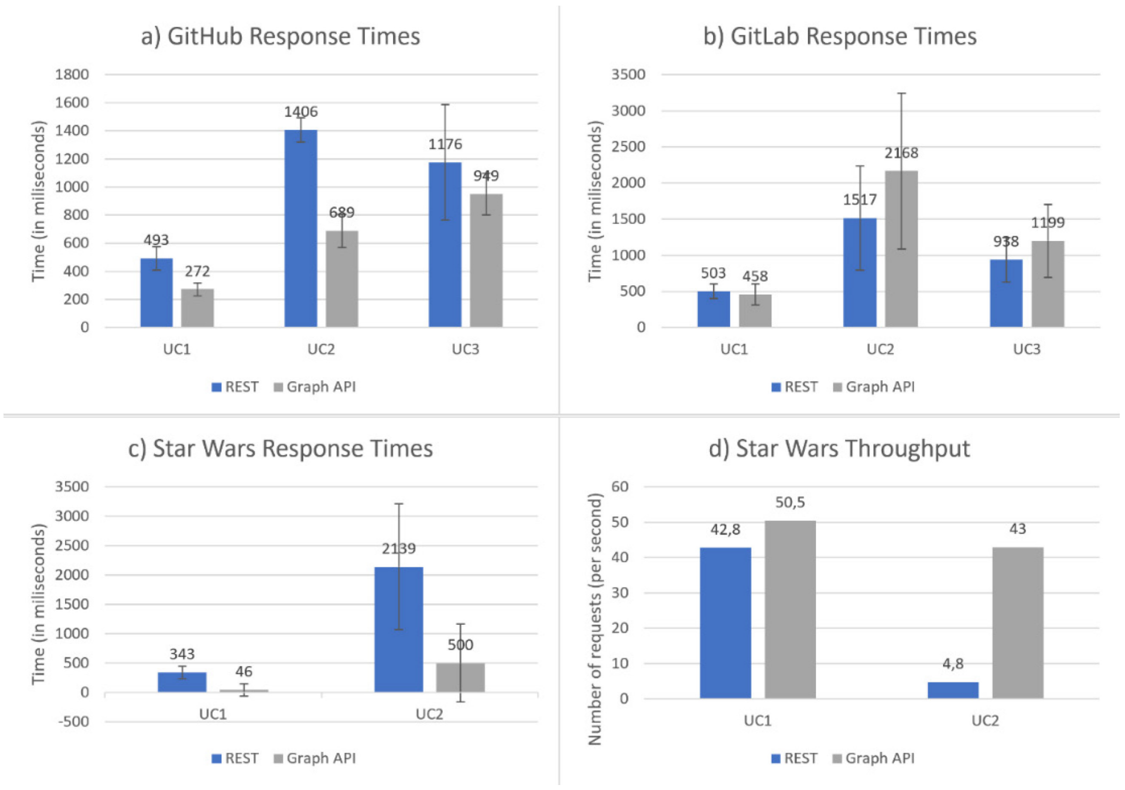


Figure 6 JMeter test results visualized. Source: author's contribution.

In Figure 6, we observe that in the case of GitHub and Star Wars API graphs, the more performant option is faster to give requested responses. Although in the case of GitHub, it is worth mentioning that the two REST requests were individually faster than graph, we required two of them to achieve our use case. It also stands out that graph was usually the more consistent choice, resulting in a lower standard deviation in all but a few use cases. Which runs counter to what a related study found [8].
The next surprise came in the form of Graph being better behaved under moderate load. When a system has limited resources and REST requires more requests than graph, it can result in severely degraded performance. Which is nicely presented in Figure 6d, where the difference in simple queries is minor and very big when REST requires just one more request per resource.

4. Discussion

In this chapter, we will address the research questions presented in the introduction. We will answer them with the knowledge we have acquired from literature and our experiment.

4.1. RQ1: What are the differences between Graph and REST API approaches?

The key difference between the two approaches is that Graph provides a singular endpoint where it accepts requests in a POST body. However, for simpler queries, GET can still be used. REST presents different endpoints that are more tailored to the use cases of an application. Which does have its benefits, but the main drawbacks are the over- and under-fetching of data. The amount of data we retrieve from the server is predetermined by its developers. Graph gives that power of decision-making to the user. Which can be a great thing when we are exposing a large API with related data to the public. As maintainers of an API, it is utopian to foresee and prepare all use cases, and even more so to maintain all those endpoints. One of the challenges Graph API provider faces is limiting how much data users can extract from it. Due to the interconnectivity, it is crucial to implement the cursor pattern and enforce limiting numbers of entries per request [6].

REST API providers have been addressing this issue with the advent of API management platforms. These platforms allow us to create use cases from APIs, compose them with no code, and expose and version all these endpoints with ease. This gave the REST API the API Gateway pattern, which has essentially the same goal as a graph API's native feature: to provide a singular point of access to our API.

4.2. RQ2: Are there performance benefits to using a particular approach when it is appropriate?

As we have discovered in Section 3.2, no approach is indefinitely better than another. There are situations where one is preferable to the other. When we need simple resources presented quickly to the customer, REST appears to be more effective. The responses were usually quicker than those of the Graph counterpart. But if we require more complex queries that require combining more endpoints, Graph appears to be the more efficient option. There also seems to be a difference in how much computational power each API requires. Graph trades a more process-heavy server for a leaner client with less required bandwidth. This leads us to believe that the Graph API is more processor-intensive than REST since it takes more time to traverse a graph than to simply respond with a pre-prepared query in REST. Although it has been shown that Graph is more efficient than REST in the utilization of its resources [11].

Graph API also liberates the client from having to process received data, shifting the responsibility from the client to the server. This also results in lower bandwidth requirements, as less data is transferred overall.

4.3. RQ3: Are there differences in testing the two APIs?

In testing, we experienced no considerable differences in performance, load, or stress testing with JMeter. It requires a different approach to payload transfer setup and transferring more data to the server via the larger request body. In unit testing, there should be no perceived differences between the two approaches. However, Graph could turn out to be more friendly to developers when it comes to preparing JSON strings to compare responses to, as they are leaner and more focused on the data we wish to retrieve and test.

In Table 4, we can see the key differences between two approaches to APIs. Both are flexible and replicate each other in the definition of the service, and they both support the same key CRUD (Create, Read, Update, and Delete) operations. Although the Graph API is not cacheable, its resource utilization and customizable response make it better performing. Both approaches are testable in the same way, although a different philosophy in forming the request is dictated by the API type. Even so, REST remains the industry standard.

Table 4
Comparison of key differences between two API approaches.

	REST	Graph API
Open API Specification	Yes	Partial
Schema definition	Partial	Yes
Supports CRUD operations	Yes	Yes
Data field selection	Fixed	Flexible
Resource utilization efficiency	No	Yes
Automatic caching	Yes	No
Testable with HTTP requests	Yes	Yes
Is the industry standard	Yes	No
Self documenting	No	Yes
Data format	Multiple formats	JSON Only
Maturity	Matured	Growing
Communication	Synchronous	Synchronous /Asynchronous
Endpoints	Might require many	One

5. Conclusion

We have conducted an exploratory study of the Graph API and how it compares to the REST API in key areas of conceptualization, performance, use cases, and performance testing. We can draw conclusions from the works of previous authors and add our own experiment. We conclude that REST and Graph API both have a place in the industry as complementary principles. This is further supported by the fact that some providers offer both API options. It is worth mentioning that the Graph API has an edge when working with larger, interconnected data sets that are exposed to the public, where use cases are hard (and numerous) to predict. It also alleviates the pressure on client devices in terms of computational power and required bandwidth. REST, however, remains a better option when we know the use cases of our customers and it is feasible to prepare them beforehand in more closed and controlled environments.

We suggest the following areas for further research: When does the Graph API become better in terms of performance to warrant the extra expense of computational power? How interconnected must the data be to make use of the relationship-focused approach of the Graph API? When do use-case generation tools like API management platforms cause bigger expenses than the additional computational power of Graph APIs? In this paper, we have limited ourselves mostly to the performance analysis of retrieving data from the server.

Acknowledgements

The authors acknowledge financial support from the Slovenian Research Agency (Research Core Funding No. P2-0057).

References

[1] R. Fielding, “Architectural Styles and the Design of Network-Based Software Architectures,” Jan. 2000.
[2] B. De, “API Management,” API Management, 2017, doi: 10.1007/978-1-4842-1305-6.
[3] D. Ritter and C. Herrmann, “A graph API for complex business network query and traversal,” Communications in Computer and Information Science, vol. 285 CCIS, pp. 52–63, 2012, doi: 10.1007/978-3-642-29166-1_5/COVER.
[4] “Hype Cycle for APIs and Business Ecosystems, 2021.” https://www.gartner.com/document/4004085 (accessed May 28, 2023).

[5] E. Wittern, A. Cha, and J. A. Laredo, “Generating GraphQL-Wrappers for REST(-like) APIs,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 10845 LNCS, pp. 65–83, Sep. 2018, doi: 10.1007/978-3-319-91662-0_5.
[6] E. Wittern, A. Cha, J. C. Davis, G. Baudart, and L. Mandel, “An Empirical Study of GraphQL Schemas,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 11895 LNCS, pp. 3–19, 2019, doi: 10.1007/978-3-030-33702-5_1/TABLES/4.
[7] J. Weaver and P. Tarjan, “Facebook Linked Data via the Graph API,” Semant Web, vol. 4, no. 3, pp. 245–250, Jan. 2013, doi: 10.3233/SW-2012-0078.
[8] E. Lee, K. Kwon, and J. Yun, “Performance Measurement of GraphQL API in Home ESS Data Server,” in 2020 International Conference on Information and Communication Technology Convergence (ICTC), IEEE, Oct. 2020, pp. 1929–1931. doi: 10.1109/ICTC49870.2020.9289569.
[9] M. Bryant, “GraphQL for archival metadata: An overview of the EHRI GraphQL API,” in 2017 IEEE International Conference on Big Data (Big Data), IEEE, Dec. 2017, pp. 2225–2230. doi: 10.1109/BigData.2017.8258173.
[10] N. Vohra and I. B. Kerthyayana Manuaba, “Implementation of REST API vs GraphQL in Microservice Architecture,” Proceedings of 2022 International Conference on Information Management and Technology, ICIMTech 2022, pp. 45–50, 2022, doi: 10.1109/ICIMTECH55957.2022.9915098.
[11] A. Lawi, B. L. E. Panggabean, and T. Yoshida, “Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System,” Computers 2021, Vol. 10, Page 138, vol. 10, no. 11, p. 138, Oct. 2021, doi: 10.3390/COMPUTERS10110138.
[12] J. Sayago Heredia, E. Flores-García, and A. R. Solano, “Comparative Analysis Between Standards Oriented to Web Services: SOAP, REST and GRAPHQL,” Communications in Computer and Information Science, vol. 1193 CCIS, pp. 286–300, 2020, doi: 10.1007/978-3-030-42517-3_22/TABLES/3.
[13] S. K. Mukhiya, F. Rabbiab, V. K. I. Punax, A. Rutle, and Y. Lamo, “A GraphQL approach to Healthcare Information Exchange with HL7 FHIR,” Procedia Comput Sci, vol. 160, pp. 338–345, Jan. 2019, doi: 10.1016/J.PROCS.2019.11.082.
[14] A. Freeman, “Understanding GraphQL,” Pro React 16, pp. 679–705, 2019, doi: 10.1007/978-1-4842-4451-7_24.
[15] M. Seshadri, “FACEBOOK GRAPH API 26 1 Overview of the Facebook Graph API,” 2010, Accessed: May 30, 2023. [Online]. Available: https://graph.facebook.com/704520608
[16] “GraphQL.” http://spec.graphql.org/October2021/#sec-Schema (accessed Jun. 01, 2023).
[17] “Overview - Graph API.” https://developers.facebook.com/docs/graph-api/overview (accessed Jun. 01, 2023).
[18] “Add a GraphQL API to Azure API Management | Microsoft Learn.” https://learn.microsoft.com/en-us/azure/api-management/graphql-api?tabs=portal (accessed Jun. 01, 2023).
[19] “GraphQL | How to Create a Serverless GraphQL API on AWS | Amazon Web Services (AWS).” https://aws.amazon.com/graphql/serverless-graphql-server/ (accessed Jun. 01, 2023).
[20] “How to manage GraphQL APIs in Apigee | Google Cloud Blog.” https://cloud.google.com/blog/products/api-management/how-to-manage-graphql-apis-in-apigee (accessed Jun. 01, 2023).
[21] “GitHub - yarax/swagger-to-graphql: Swagger to GraphQL API adapter.” https://github.com/yarax/swagger-to-graphql (accessed May 30, 2023).
[22] “GitHub - aweary/json-to-graphql: Create GraphQL schema from JSON files and APIs.” https://github.com/aweary/json-to-graphql (accessed May 30, 2023).
[23] “GitHub - IBM/openapi-to-graphql: Translate APIs described by OpenAPI Specifications (OAS) into GraphQL.” https://github.com/ibm/openapi-to-graphql (accessed May 30, 2023).