A Comparative Study Between Graph-QL& RESTful Services in API Management of Stateless Architectures

Mr.Sayan Guha and Mrs.Shreyasi Majumder

Data Architect, AI and Analytics Practice, Cognizant Technology Solutions, India

ABSTRACT

A stateless architecture design is a web architecture design that typically does not persist data in any database and such applications also does not require any kind of backup storage. Data that flows through a stateless service is data in transition and such data is never stored in any data store. The processing requests that arrive to such architecture does not rely on information gathered or persisted from any previous session. API (Application programming interface) which consists of subroutines, definitions & procedures that can access data on the applications are the communication points between applications and management of API endpoints using stateless architecture is less complex as there is no server side retention of the client session and each client sends requisite information in each request to the server. GraphQL and RESTful services are means of designing such API architecture. This paper discusses and explains in detail both GraphQL and REST API architecture design and management methods and does an analysis of the potential benefits of GraphQL over REST in Stateless architectural API designs.

KEYWORDS

API, RESTful, URI, GraphQL, Stateless, Schema Definition Language (SDL), HTTP, Mutation.

1. Introduction

Since inception of API based web services to aid seamless integration & data exchange across applications on the foundations of stateless architecture, REST based API web services have been the de-facto standard & preferred architectural style for design and management of the web services.

Although there are scenarios, where performance of a web service API, tailored need of the data consumer & ease of retrieval of data using one single API endpoint lays a strong underpinning for using GRAPHQL as suitable alternative of REST for Web service computing.

In this paper, we aim to perform a comparative analysis of scenarios where GRAPHQL could evolve as an alternative architectural style for API based stateless application architectures. We aimed to infer based on observations with variable data volumes simulating an experiment with social media posts. We have performed the experiment across many iterations and factoring the complexities of number of API endpoints & increasing data volume in each iteration to depict the behavioural response of both GRAPHQL & REST in terms of throughput. The results depict potential benefits of GRAPHQL over REST & identifies areas of further research and improvement with GRAPHQL based approachWe have organized the remainder of the paper as follows. In Section 2, we discussed API management in stateless architecture as a brief introduction to the context. In Section 3, related work has been discussed in REST based API management scenarios along with the shortcomings and security vulnerabilities. In Section 4, we aim to provide a concise introduction to GRAPHQL based API management approach along with resolution to the conventional problems encountered in REST based approach. We aimed to explain the supported types of GRAPHQL & its usage scenarios to facilitate an appreciative understanding of GRAPHQL objects and ease the understanding of to the experiment in the subsequent Section 5.

In Section 6, we present the performance evaluation results factoring the complexity of number of API & increasing data volume and relative analysis of GRAPHQL Vs REST. We summarize our work in Section 7.

2. API MANAGEMENT IN STATELESS ARCHITECTURE

MoussaTaifi et al (2016) [1] says the challenge of HPC (High Performance computing) applications are required to be improved on the fronts of reliability and performance owing to the existing difficulties for existing performance tuned APIs and provided a solution provided by Stateless architecture at scale.

In a Stateless architecture, deploying APIs to multiple number of concurrent users and to multiple servers is a good practice. Any server can handle any request as no session information is being stored from previous sessions.

3. RELATED WORK

JacekKopecký et al (2016) [2] mentioned that the existing misuse HTTP protocols has paved the way to RESTful services into the picture for effective API management. HTTP operations like GET, PUT, POST, DELETE & PATCH are united under the umbrella of combined operations under the name of "RESTfulness" adopting the acronym of REST invented by Roy Thomas Fielding (2010) [3].

Roy Thomas Fielding (2010) [3] mentioned in his article that RESTful Web Services is a client-server architectural style that provides a behavioural model for client applications and web services. REST describes a number of design principles and constraints, such as stateless communication and the use of uniform interfaces and self-descriptive messages applied in REST-based services.

The services created in accordance to the style of REST architecture are typically referenced a RESTful web services. The basic principle of a RESTful web service the exposure a set of resources, i.e., any information source, uniquely identified by a Uniform Resource Identifier (URI) and can be accessed through web. The below shows the working principle of RESTful Web Service in API management.

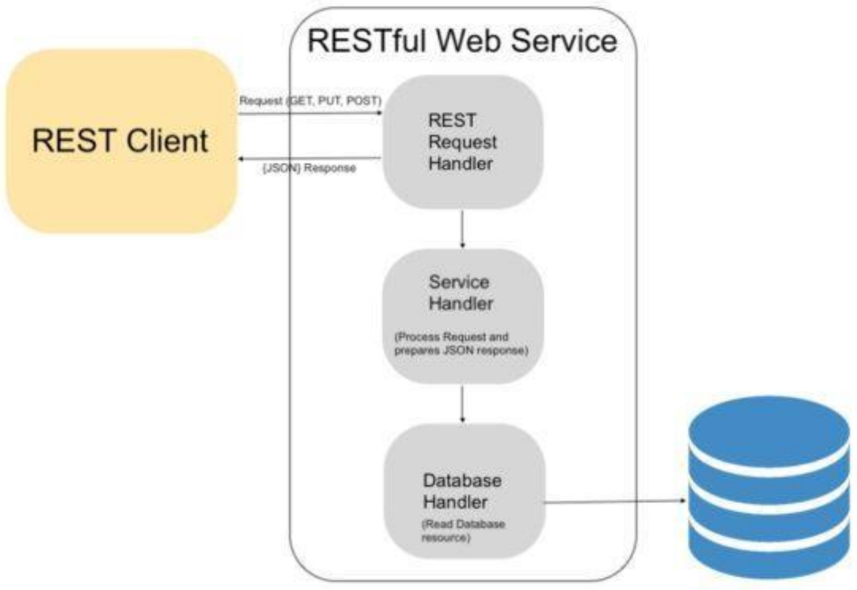


Figure 1: Runtime architecture of RESTful Web Services Source: Website, phppot.com

3.1. Limitations of RESTful Services in Stateless Architecture

Below limitations exist while working with RESTful services in Stateless architectural communications:

i. Incapability to heavyweight data transfer: According to FestimHalili et al (2018) [4] one of the major drawbacks of RESTful Web Services lies in the incapability of handling heavy data transfer. This inability actually makes REST services lightweight and rely on lightweight data transfer over a common interface –the URL.

ii. Reliability on fixed data structures: REST relies on fixed data structures. In RESTful, services making multiple calls to different REST endpoints is required and separate code to transform & merge the data from each response is required before using them to render your views. REST relies on the fixed data structures and iterative process to get the desired response.

iii. Security challenges: API management done using REST has proven to be vulnerable to security threats over the web. According to NishuPrasher (2018) [5] the possibilities of the following security vulnerability possibilities in his thesis.

Table 1. REST Security vulnerability possibilities

Possibility	Scenario
Injection Attacks & message altering	Unreliable SQL injection into API by a command
Security Assurance	SQL injection, parameter & path disclosure
Authentication-based attacks	Inadequate authentication. Hacking of web tokens.
Denial of service (DoS) and buffer overflows	API key/ Access token hacking if no threshold on too many requests
Cross-site scripting/cross-site request forgery	Also known as, XSS attack. REST APIs are vulnerable when malevolent code is injected as input to web services
Man-in-the-middle (MITM) attacks	Absence of TLS layer security in a REST API. Lack of transport level encryption
Replay attacks and spoofing.	REST APIs are vulnerable to spoofing of the valid transactions and the attacker could replay one valid transactions as would like.
Insecure direct object references.	REST APIs expose IDs to get resources. This results in direct exposure to internal objects
Sensitive data exposure.	Non Encrypted data exposure
Missing function level access control	Weak authentication validation in sensitive request handlers

4. GRAPHQL IN API MANAGEMENT

GraphQL is an alternative and all new approach to interact with Web APIs. It is an open source data manipulation and querying language for APIs. It is dynamic, single endpoint interactive query based language to interact with APIs. Client systems using GraphQL can talk to the server on exactly what they need, the queries are written to interact the web on exactly what is required - nothing more or less.
As an alternative to conventional RESTful service, GraphQL developed by Facebook® in 2012 & publicly released in 2015.

On 7 November 2018, Facebook® transferred the GraphQL project to the lately established GraphQL Foundation, hosted by the non-profit Linux Foundation.

Olaf Hartig et al (2017) [6] in his conference paper explains GraphQL as new type of Web-based data access interfaces that presents an alternative to the notion of REST-based interfaces & owing to this advantages over REST, since its release GraphQL has gained significant momentum and has been adopted by an increasing number of users.

4.1. GRAPHQL based API management – An alternative approach

Kristopher Sandoval (2017) [7] in his research blog discussed about the potential benefits of using GraphQL. He also mentions that because GraphQL is extremely powerful, several providers who need stable readability with quick speed and indexing have used it. Most of the use cases for GraphQL are therefore those who require high data throughput with ease of sorting & represented clearly by its highest profile users.

Data Fetching using Single Endpoint

Data Fetch using GraphQL is a paradigm shift compared to working with REST APIs. REST APIs considered multiple specific endpoints to load the data. In GraphQL APIs typically a single endpoint is exposed and which in turns out to be more flexible for the client system to decide what data is actually needed. The below figure explains the difference of data fetch between REST API and GRAPHQL APIs.

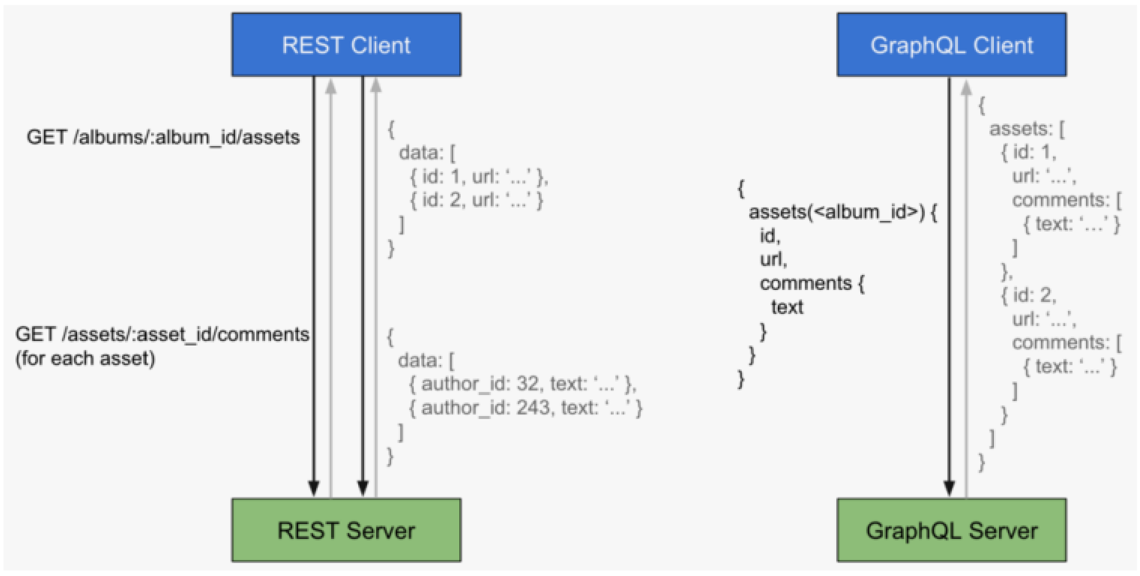

REST Client
GET /albums/:album_id/assets
{ data: [{ id: 1, url: '...' }, { id: 2, url: '...' }] }
GET /assets/:asset_id/comments (for each asset)
{ data: [{ author_id: 32, text: '...' }, { author_id: 243, text: '...' }] }
REST Server
GraphQL Client
{ assets(<album_id>) { id, url, comments { text } } }
{ assets: [{ id: 1, url: '...', comments: [{ text: '...' }] }, { id: 2, url: '...', comments: [{ text: '...' }] }] }
GraphQL Server

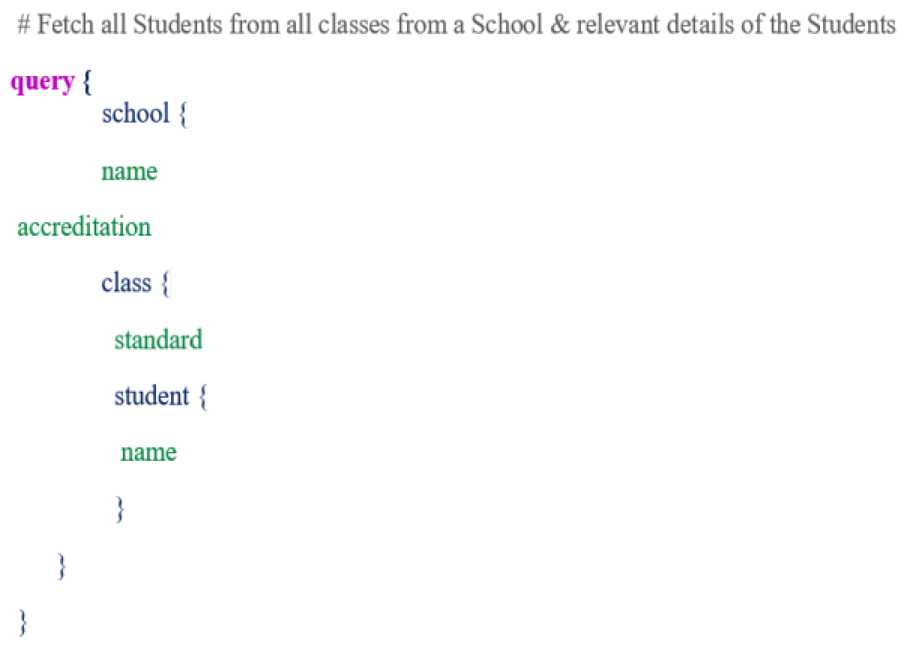
Figure 2: REST API Vs GraphQL API Source: Website, pinterest.com

Resolution to Underfetching&Overfetching

Overfetching means the API consumer downloads more information than actually required for his custom need. In Rest API, the only way user obtains the data is by accessing multiple endpoints. The Response might have additional information that the user requirement. This situation is **Overfetching**.

Underfetching means API consumer downloads less information than actually required for his custom need. In Rest API, the only way user obtains the data is by accessing multiple endpoints. The Response might have less information that the user requirement. In this situation, the client will make additional request until the retrieval of required information. This situation is **Underfetching**.

In GraphQL client, obtain exactly the data they need from an API. The below figure explains a sample GraphQL request query to fetch authors and relevant articles details in a single GraphQL query. It does not require two separate endpoints.



```
# Fetch all Students from all classes from a School & relevant details of the Students

query {
          school {

          name

accreditation

          class {

            standard

            student {

             name

            }

     }

}
```

Figure 3: GraphQL query to fetch students all classes in a school

Schema Definition Language (SDL)

Olaf Hartig et al (2019) [8] in his research paper has focussed on repurpose of schemas for graph databases that are based on the Property Graph model based the Schema Definition Language (SDL), originally meant as a language to define a so-called GraphQL schema that specifies different types of objects that can be queried when accessing a particular Web API.

Therefore, the authors rearticulate definition of Schema Definition Language (SDL) from data usage standpoint -GraphQL uses a schema to articulate the shape of the data graph. This schema defines and describes a hierarchy of objects called types, populated from the backend database storage.

Olaf Hartig et al (2018) [9] in his morning paper blog have put together many perspectives of quantitative definition of GraphQL schemas. One of the most popularly used definition explains GraphQL as edge-labelled multigraph where each node is associated with an object type and comprising of dictionary of properties. Property keys derives their as field names from set F. GraphQL schemas constructed over three sets: Fields (F), Arguments (A), and Types (T)
As per the authors GraphQL graph over (F, A, T) is a representation of a tuple

$G = (N, E, \tau, \lambda, r)$ where

- N is a set of nodes
- E is a set of edges of the form (u, f [α], v) where u, v € N, f € F and α is the partial mapping from A to Values
- τ: N $\rightarrow O_T$ is a function, which assigns a type to every node in the system.
- λ is a partial function that assigns a scalar value v € Values or a sequence [v1, v2...vn] of scalar values where (vi € Values) of some pairs of the form (u, f [α]) where u € N, f € F and α is the partial mapping from A to Values.
- r € N is the distinguished node called root node of the graph system.

Variety of Supported Type Definitions in GRAPHQL

In order to have a better understanding of our work about relative study of GRAPH-QL vis-à-vis REST services in API management, we considered a quick appreciative recognition of various supported type definitions associated with GRAPHQL & the related scenario of their application.

Table 2. GRAPHQL supported types

Type	Type description	Includes	Definition	Primitive type	Advanced type	Heterogeneous type	Data retrieval	Insert / Update in API change
Scalar	Scalar types are the primitive types, single type for every scalar type	Int	Signed 32-bit integer.	X				
		Float	Signed double-precision floating-point value.	X				
		String	UTF-8 compliant character sequence	X				
		Boolean	True / False OR 1/0 decision enabling type	X				
		ID	A unique identifier and is serialized as a string	X				
Object	Either a field or another object type of a combination		Object type can constitute another Object type along with other scalars		X	X		
Query	Object types by means of which data can be retrieved from multiple variables for a particular schema		An explicit mention of the required fields mentioned in the query		X	X	X	
Mutation	API, which can alter data, types useful either by inserting or updating data already in the database.		Type constitutes "Create" & " Update type" within Type Mutation		X	X		X

5. EXPERIMENT SCENARIO: RETREIVE SOCIAL MEDIA POST DATA

As we discussed the shortcomings of data fetching using REST API based calls & categorically tried to explain the ease with which GRAPHQL has brought a paradigm shift to the fetching of user and related data from any website using typically a single endpoint. In the experiment, we have aimed to bring forth the advantage of right fetching of the data that is required compared the limitations of over fetching and underfetching using REST API based GET calls.

GRAPHQL Based Query Engine

Based on all the work that has been accomplished with GRAPHQL, we conceptualize GRAPHQL query engine would work as per the below block diagram.

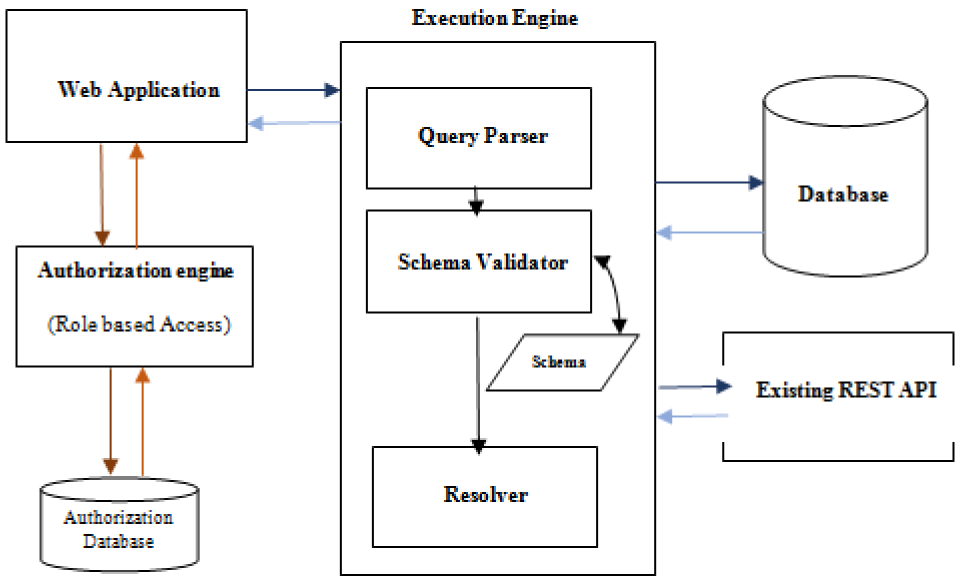


Figure 4: Conceptual GRAPHQL Query Execution Engine

- Schema is a tailored type language, which will return results back to the user. The user/client will request any number of fields and GRAPHQL server will return only the fields expected.
- The query optimization parser once parses the values, validated to be good by the schema validator,
- The resolvers proceeds towards processing the query & will return requisite data fields only.

In the process of performing the experiment we have studied previous work of JobineshPurushothaman (2018) [10] on his complete guide to building a polyglot GraphQL Server lays down a foundational understanding to use GraphQL to its best for retrieving Social Media Posts.

Our work derives inspiration from the same where he has provided lightweight understanding in his paper.

In addition to the above, we have studied the work of Sebastian Eschweiler (2018) [11] on his lightweight approach by which a server can be implemented of its own and can be made ready to execute queries to retrieve data from any website.

All codes provided in the below example are not in executable state and only for educational purpose used by the authors. It can help the reader to understand the approach to the retrieval but is not a verbatim source code.

```
/ Provisioning of resolver functions for schema field & Type definitions

consttypeDefs = `

typequery{

    post (id: Int!): Post
                user (id: Int!): User
       },
typepost   {
    Post_id: Int
                        user: User
                        title: String
                   },
typeuser   {
    id: Int
                        name: String
                        email: String
                       posts: [Post]
                   },
    `;
/ declare variable definitions for post and users for different users

/ Author names used for demo purpose

varpost  =  [
        {
          Post_id: 1,
          user: 1,
          title: ‘Sky is the limit’,
          },
        {
          Post_id: 2,
          user: 2,
          title: 'Where knowledge is free and head is held high’
          }
]
```

```
varuser  =  [
          {
          User_id: 1,
          name: 'ShreyasiMajumder',
          email: 'shmaj@yahoo.com'
          },
          {
          id: 2,
          name: 'SayanGuha',
          email: 'sguha@yahoo.com'
          }
          ];

/ Variables functions to getPost&getUser

vargetPost=function (root, {id})
      {
          returnpost.filter
           (post => {
               return post. Post_id === id;
          }) [0];
      };

vargetUser= function (root, {id})
          {
          returnuser.filter
             (user => {
                 return user. User_id === id;
             }) [0];
          };

/ Provisioning of resolver functions for schema fields
   const resolvers =  {

query:  {
   post: getPost,
   user: getUser,
   },
   User:  {
   posts: (user) => filter (post, {userId: user. id}),
   },
Post:  {
   user: (post) => find (user, {postid: post.id}),
   },
   };
/ End of code required to create the API endpoints for data retrieval to work
```

Figure 5: Sample code to create API endpoints to retrieve data using GRAPHQL

The authors hereby also draws attention to the work by **Erik Wittern et al (2018) [12]** y on the best utilization of GRAPHQL to query of the API endpoints. They mentioned to have analysed corpuses for common schema characteristics, naming conventions, and worst-case response sizes.

Authors have extended the same understanding with the above case study, evaluated the retrieval time for the same API endpoint, and made a comparitive study with RESTful service based data retrieval in terms of throughput time.

Extending the code shared already in Figure 4, we depict the retrieval of data by user query.

```
/ Sample query definition to get all Post info and user information for the post in a
  single query

queryInfoPostUser  {
  post(id:1)
{
  Post_id
  title
  user
{
  User_id
  name
  email
   } }
   }
/ Sample query for data retrieval from a social media website & related details of the
```

Figure 6: Sample query to retrieve data from Social Media Website & related details of the user

The query would provide the results and related user details in one query and working with one API endpoint

```
"data":
{
"post":
{"Post_id": 1,
  "title":'Sky is the limit',
  "user": {
  "User_id": 1,
  "name":'ShreyasiMajumder',
  "email" : 'shmaj@yahoo.com'
   } }
  }
```

Figure 7: Data Output from the Data retrieval query obtained from one API endpoint

6. PERFORMANCE EVALUATION: REST VS GRAPHQL

Our work in evaluation of performance of REST Vs GRAPHQL is most closely related to MatheusSeabra et al (2019) [13] who has done a deep performance comparison study between REST and GRAPHQL in his conference paper.

MatheusSeabra et al (2019) [13] mentioned that through research of performance metrics of response time and the average transfer rate between the requests, it was possible to deduce the particularities of each architectural model in terms of performance metrics. We observed that migrating to GraphQL resulted in an increase in performance in two-thirds of the tested application.

Authors have carried out a Proof of Concept not by migrating from REST to GRAPHQL but creating sample API endpoints for REST and GRAPHQL and tested the same in open source playground and tested the throughput in terms of response time for data retrieval in 3 iterations

i. Iteration1: We carried out Iteration1 with a data volume of 1,000 data records through one REST API and one GRAPHQL API endpoints.
ii. Iteration 2: We carried out Iteration1 with a data volume of 10,000 data records through two REST API and one GRAPHQL API endpoints.
iii. Iteration 3: We carried out Iteration1 with a data volume of 100,000 data records through three REST API and one GRAPHQL API endpoints.

Performance evaluation for REST and GRAPHQL in the experiment based on the complexity definition assumed by the authors as combination of data volume and API endpoint weightage as below

Table 3. Complexity definition for Data retrieval & Data Volume (following the experiment scenario)

Complexity Consideration Perspective	Technology Platform	No API Endpoint Consideration	SIMPLE	MEDIUM	COMPLEX
Complexity of Data retrieval	REST	1	X		
		2		X	
		3			X
	GRAPHQL	1	X		
		1		X	
		1			X
Complexity Consideration Perspective	Technology Platform	Volume of Data	SIMPLE	MEDIUM	COMPLEX
Complexity of Data Volume (no of records)	REST	1000	X		
		10000		X	
		1000000			X
	GRAPHQL	1000	X		
		10000		X	
		1000000			X

Performance evaluation in response time (throughput) is as below.

The results show:

Iteration 1: Approximately marginal or no difference when we considered one API endpoint for both REST and GRAPHQL with the same volume of 1000 data records.

Iteration 2: 35percentage lesser response time using GRAPHQL where 10 times higher data volume considered with respect to Iteration1 and our Iteration 2 was executed with 10000 data records. In Iteration 2, we considered two API endpoints in REST compared to only one API endpoint required in GRAPHQL.

Iteration 3: When we increase the volume of data 100 times to the initial volume and execute Iteration 3 with 100,000 data records with three REST API endpoints being considered compared to only one API endpoint as required GRAPHQL. The authors have observed an approximate 40% less response time required in the experiment results in GRAPHQL.

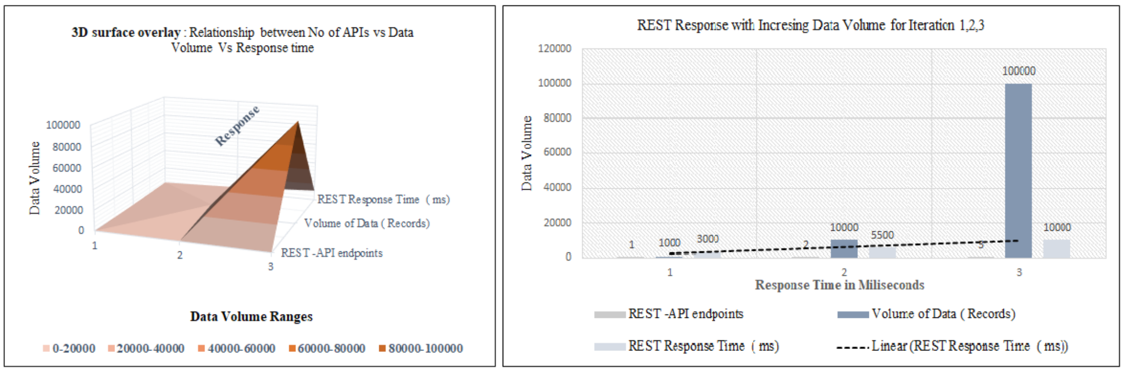


Figure 8: REST Experiment API Results

Left – Relationship between the three dimensions (Volume, No of APIs and REST Response Time Right – Linear Trend of REST Response Time Plotted on data volume considered for three iterations of the experiment

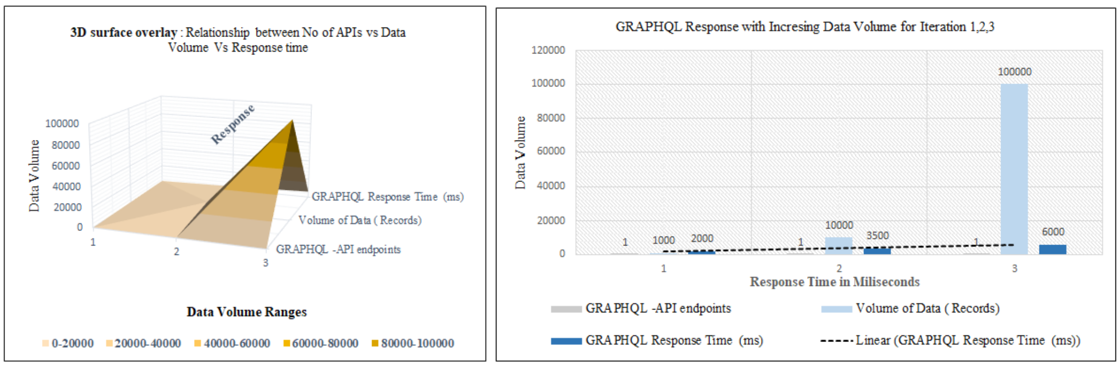


Figure 9: GRAPHQL Experiment API Results

Left – Relationship between the three dimensions (Volume, No of APIs and GRAPHQL Response Time Right – Linear Trend of GRAPHQL Response Time Plotted on data volume considered for three iterations of the experiment

The authors therefore considers

- The single API endpoint utilization in GRAPHQL as one of the key factors to determine the ease of availability of data as well as the response time with a constant data volume.
- High data volume experiment to check the relationship between the No of API Endpoints Vs the Response time to conclude and understand the implications of high volumes vs number of API endpoints. REST services produces low performance in response time compared to GRAPHQL.

Authors therefore share one more observation with only two parameters No of API Endpoints & Response Time in REST and GRAPHQL testing the performance with only 1 Iteration in this case viz. 100,000 data records, which confirms the understanding of linear increase of REST in response time compared to GRAPHQL as below.

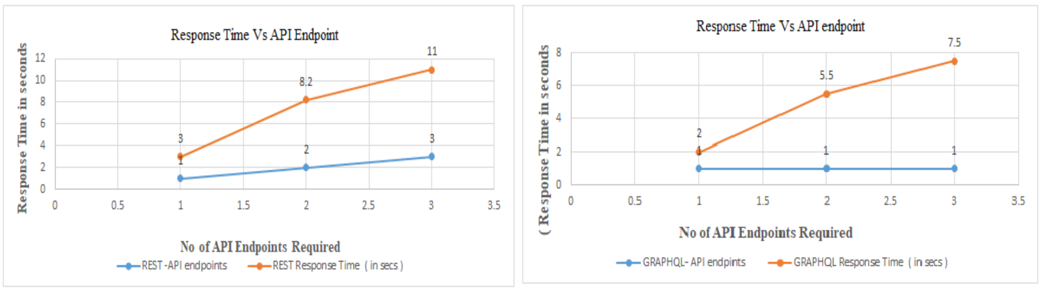


Figure 10: No. API Endpoints Vs Response time relationship for a high volume

7. Limitations and Areas of Improvement

The authors have observed that the performance of GRAPHQL considerably increases compared to the REST based API framework in situations where considerable data volume is considered. Researchers plausibly considers REST as the defacto standard in APIs, which have very less number of entities. The trade-off between the cost and architectural decision will favour REST with success & ease of use in simple request-response architectural designs.

GRAPHQL on the other hand suits situations where data is required at scale, large number of entities are involved, and expected growth of the data is manifold. The response time in such cases with only one API interfaced for data retrieval makes is relatively better choice for selection over REST.

In the experiment, with a gradual increasing data volume, we have captured the performance of GRAPHQL relative to REST. Our results depict GRAPHQL capabilities of data retrieval are considerably better in such situation.

We have not considered the caching implications of GRAPHQL, which could turn out to be costly, in cases where we need to write tailored GRAPHQL queries & therefore cannot store results cached from previous data operations. Our work lies within the boundaries of potential benefits of GRAPHQL in response time where data volume is huge and number of entities are large. The GRAPHQL based technology platform have limitations in single request–response based interaction & caching capabilities These areas of improvement of GRAPHQL ,we consider as subject to further research.

8. CONCLUSION

As studied, we have observed that GRAPHQL is increasing acceptability as preferred API management technology where performance metrics of response time and utilization of lesser number of API endpoints are key measuring criteria with high data volume. We acknowledge that REST has become an industry standard for companies and API management using REST endpoints have matured over period of time and GRAPHQL have a learning curve associated with it and with improved tooling functions over a period of time in future applications of GRAPHQL in fields like Business Intelligence will increase manifold.

Our work, we trust will motivate upcoming avenues of future research where performance, data driven design and performance flexibility with lesser API interaction would take precedence.

REFERENCES

[1] MoussaTaifi, Yuan Shi. &YasinCelik (2015) "JENERGY: A Fault Tolerant Stateless Architecture for High Performance Computing", https://www.researchgate.net/publication/303837779_JENERGY_A_Fault_Tolerant_Stateless_Architecture_for_Hig_Performance_Computing

[2] JacekKopecký, Paul Fremantle & Rich Boakes (2014) "A history and future of Web APIs", https://www.researchgate.net/publication/274527941_A_history_and_future_of_Web_APIs

[3] Roy Thomas Fielding (2014) "Architectural Styles and the Design of Network-based Software Architectures", https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[4] FestimHalili&YasinCelik (2018) "Web Services: A Comparison of Soap and Rest Services", https://www.researchgate.net/publication/323456206_Web_Services_A_Comparison_of_Soap_and_Rest_Services.

[5] NishuPrasher (2018) "Security Assurance of REST API based applications", https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2502569/19973_FULLTEXT.pdf?sequence=1&isAllowed=y.

[6] Olaf Hartig& Jorge Pérez (2017) "An Initial Analysis of Facebook's GraphQL Language", https://www.researchgate.net/publication/316686431_An_Initial_Analysis_of_Facebook's_GraphQL_Language.

[7] Kristopher sandoval (2018) "7 Unique Benefits of Using GraphQL in Microservices", https://nordicapis.com/7-unique-benefits-of-using-graphql-in-microservices/

[8] Olaf Hartig& Jorge Pérez (2017) "An Initial Analysis of Facebook's GraphQL Language", https://www.researchgate.net/publication/316686431_An_Initial_Analysis_of_Facebook's_GraphQL_Language.

[9] Olaf Hartig& Jorge Pérez (2018) "Semantics and complexity of GraphQL", https://blog.acolyer.org/2018/05/21/semantics-and-complexity-of-graphql/

[10] JobineshPurushothaman (2018) "Building a Polyglot GraphQL Server", https://static.rainfocus.com/oracle/oow18/sess/1526618246355001wDNO/PF/DEV6113_Purushothaman_15404417239460019Dyp.pdf.

[11] Sebastian Eschweiler (2018) "Creating a GraphQL Server with Node.js and Express", https://medium.com/codingthesmartway-com-blog/creating-a-graphql-server-with-node-js-and-express-f6dddc5320e1

[12] Erik Wittern, Alan Cha, James C. Davis, Guillaume Baudart& Louis Mandel (2018) "An Empirical Study of GraphQL Schemas", https://arxiv.org/pdf/1907.13012.pdf

[13] MatheusSeabra, Marcos Felipe Nazário, & Gustavo Pinto (2019) "REST or GraphQL? A Performance Comparative Study", https://www.researchgate.net/publication/335784769_REST_or_GraphQL_A_Performance_Comparative_Study

AUTHORS

SayanGuha completed his Bachelors in Electronics & Communication Engineering in 2006. He have been serving Information Technology industry supporting Data Modelling & Architecture across business domains of Retail, Banking, and Insurance & Telecom. He is currently working with Cognizant technology Solutions and his area of interests are in the fields of Big Data Integration, API based data integration & Advanced analytics.

ShreyasiMajumder completed her Bachelors in Computer Science & Engineering in 2005. She have been serving Information Technology industry supporting across business domains Retail, Manufacturing, Insurance & Telecom business domains. She is currently working with Cognizant Technology Solutions and her areas of interests are in the fields of Business Intelligence, Design thinking, Big Data & API based data integration and Advanced Analytics.