Available online www.jsaer.com

***Journal of Scientific and Engineering Research*, 2019, 6(10):283-288**

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Graph Query Language: A Data Consolidation Layer

Khirod Chandra Panda

Asurion Insurance, USA
Email: khirodpanda4bank@gmail.com

Abstract API integration is a crucial aspect of modern software development, enabling smooth communication between applications and external services. Among the various integration methods available, GraphQL has emerged as a powerful API query language, offering flexibility and efficiency in retrieving data. This article aims to explore API integration using GraphQL in-depth, covering its core concepts, advantages over traditional REST APIs, implementation strategies, best practices, real-world use cases, and its future in software development. GraphQL revolutionizes the way APIs are queried and executed. Unlike traditional REST APIs, which limit clients' control over the data they receive, GraphQL allows clients to specify exactly what data they require. This approach reduces unnecessary data fetching, leading to more effective communication between clients and servers. In a GraphQL Web API, a GraphQL schema defines the types of data objects that can be queried, while resolver functions fetch the relevant data from underlying sources based on the queries. This architecture not only enables efficient data access but also facilitates data integration. When the GraphQL schema accurately represents the semantics of data from various sources, and resolver functions can retrieve and structure data accordingly, GraphQL can act as a unified interface for accessing and integrating diverse data sources. However, current approaches to GraphQL for data integration lack semantic awareness and formal methods for defining GraphQL APIs based on ontologies. To bridge this gap, a framework is proposed in which a global domain ontology guides the generation of a GraphQL server. This framework includes an algorithm for generating a GraphQL schema based on ontology and generic resolver functions based on semantic mappings.

Keywords Batching, Data, DoS, GraphQL, Introspection , Mutation, Query ,Schema

1. Introduction

In today's interconnected digital landscape, the seamless exchange of data and functionality across diverse systems has become essential. This trend has elevated the role of Application Programming Interfaces (APIs) in modern software development to new heights. APIs act as bridges that allow applications to communicate with external services, enabling collaboration, innovation, and efficiency. While traditional Representational State Transfer (REST) APIs have been widely used, GraphQL has emerged as a revolutionary alternative, offering a fresh perspective on API integration. This article explores the realm of API integration using GraphQL, examining its core principles, advantages over REST, and its potential to redefine communication between applications.

The evolution of APIs has been transformative, evolving from simple data endpoints to complex ecosystems that facilitate intricate interactions. The advent of REST APIs introduced a standardized approach to structuring

and requesting data, enabling applications to communicate effectively over the web. However, REST APIs have encountered challenges such as over-fetching or under-fetching data, resulting in performance issues and suboptimal response times. In contrast, GraphQL, a query language for APIs developed by Facebook in 2012 and released to the public in 2015, addresses these limitations by offering a more flexible and efficient method of data retrieval.

GraphQL empowers clients to precisely request the data they need, reducing the need for multiple requests and eliminating redundant data transfers. Unlike REST, where each endpoint corresponds to a specific data structure, GraphQL allows clients to craft queries that specify their exact data requirements. This dynamic nature of GraphQL enables developers to tailor responses to meet the unique needs of user interfaces, thereby enhancing performance and reducing network overhead.

Additionally, GraphQL's introspective nature allows clients to query the schema itself, facilitating self-documentation and simplifying the discovery process. As software systems become increasingly complex and interconnected, GraphQL emerges as a solution that enhances data exchange efficiency and streamlines the development process. It enables the creation of more agile and responsive applications, making it a valuable tool in modern software development.

Is dynamic nature of GraphQL empowers developers to shape responses according to the unique needs of user interfaces, optimizing performance and reducing network overhead. Furthermore, GraphQL's introspective nature enables clients to query the schema itself, facilitating self-documentation and simplifying the discovery process. As software systems become increasingly complex and interconnected, GraphQL emerges as a solution that enhances data exchange efficiency and streamlines the development process, allowing for more agile and responsive applications.

2. Literature Review

Today, Software-as-a-Service (SaaS) has received significant attention as one of the three main service models of cloud computing (i.e., Platform-as-a-Service or (PaaS) and Infrastructure as-a-Service or (IaaS) [1, 2]). SaaS utilizes the internet to deliver applications to its users, which are managed by a provider. GraphQL schemas and GraphQL resolver functions are basic building blocks in the implementations of GraphQL servers. The former describe how users can retrieve data using GraphQL APIs. The latter contain program code including how to access data sources and structure the obtained data according to the schema. We introduce GraphQL schemas and GraphQL resolver functions in Section 2.1 and Section 2.2, respectively.

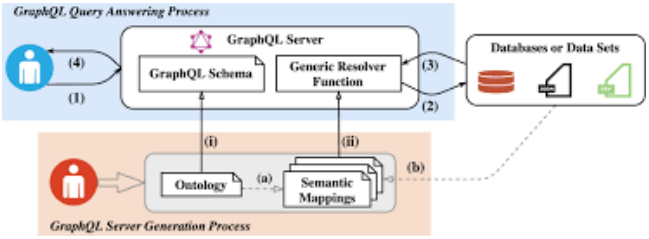

Figure 1: Ontology based GraphQL Server

2.1. GraphQL Schema

In a GraphQL API, the GraphQL schema defines types, their fields, and the value types of the fields. Such a schema represents a form of vocabulary supported by a GraphQL API rather than specifying what the data instances of an underlying data source may look like and what constraints.

must be guaranteed [3]. There are six different (named) type definitions in GraphQL, which are scalar type, object type, interface type, union type, enum type and input object type. Figure 2 depicts a GraphQL schema example.

```
const Accessory = gql`

type Accessory {
    VendorItemInformation:VendorItemInformation
    HorizonItemInformation:HorizonItemInformation
    IsNoLongerProduced:String
    IsNotCurrentlyAvailableForOrder:String
    IsCovered:String
    IsMandatoryNoSplitingFromPrimary:String
    IsOptionalAccessoryRequired:String
    IsSimCard:String
    ItemTypeID:String
    IsInKit:String
    IsInStock:String
    Quantity:Int
    AvailabilityMessage:String

        # Added for GetShippingOrder
        Priority: String
        VendorItemDescription: String
        Type: String
        SKU: String
        IsMandatory: Boolean
    }

    type AccessoryList {
        Accessory : [Accessory]
    }
`

module.exports = [Accessory];
```

Figure 2: GraphQL Schema Definition

An object type represents a list of fields and each field has a value of a specific type such as object type or scalar type. A scalar is used to represent a value such as a string

2.1. GraphQL Resolver Function

In a GraphQL API, apart from the GraphQL schema defining types, their fields, and the value types of the fields, resolver functions are responsible for populating the data for fields of types in the GraphQL schema.

```
const query = {
    Query: {
        GetAccountSummary: async (_source, args, data) => {
            Object.freeze(GetKeyFromObject(data,'ALPHA'));
            return data.dataSources.getAccountSummaryAPI.getAccountSummary(args, GetKeyFromObject(data,'ALPHA'));
        }
    }
};

module.exports = {
    query
};
```

Figure 3: Query pattern in GraphQL.

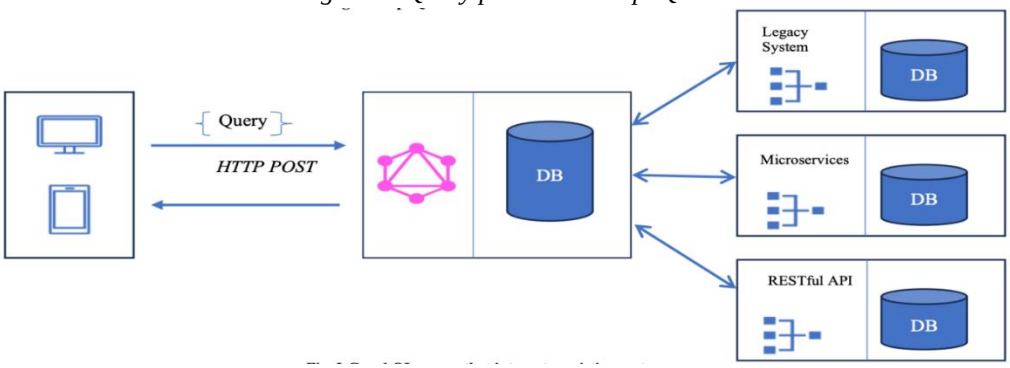

Figure 4: GraphQL. As a Data consolidator Layer

```
const mutation = {
  Mutation: {
    FindAddresses: async (_source, args, data ) => {
      Object.freeze(GetKeyFromObject(data,'ALPHA'));
      return data.dataSources.findaddressesAPI.findAddresses(args,GetKeyFromObject(data,'ALPHA'));
    },
    CreateAddress: async (_source, args, data) => {
      Object.freeze(GetKeyFromObject(data,'ALPHA'));
      return data.dataSources.createAddressAPI.createAddress(args,GetKeyFromObject(data,'ALPHA'));
    },
    GetShippingAddress: async (_source, args, data) => {
      Object.freeze(GetKeyFromObject(data,'ALPHA'));
      return data.dataSources.getShippingAddressAPI.getShippingAddress(args,GetKeyFromObject(data,'ALPHA'));
    },

    UpdateShippingAddress: async (_source, args,  data) => {
      Object.freeze(GetKeyFromObject(data,'ALPHA'));
      return data.dataSources.updateShippingAddressAPI.updateShippingAddress(args,GetKeyFromObject(data,'ALPHA'));
    }
  }
};

module.exports = {
  mutation
};
```

Figure 5: Mutation pattern in GraphQL

3. Advantages of GraphQL

Flexible data retrieval: GraphQL allows clients to request exactly the data they need, reducing over-fetching and under-fetching of data.

Efficient communication: By enabling clients to specify their data requirements, GraphQL facilitates more efficient communication between clients and servers.

Reduced network overhead: The ability to tailor responses to match client needs minimizes unnecessary data transfers, improving network performance.

Dynamic queries: Clients can craft queries to suit their specific requirements, allowing for dynamic and responsive interactions with the API.

Introspection: GraphQL's introspective nature allows clients to query the schema itself, facilitating self-documentation and simplifying the discovery process.

Single endpoint: Unlike REST APIs, which often require multiple endpoints for different resources, GraphQL typically uses a single endpoint for all data retrieval operations.

Ecosystem growth: The GraphQL ecosystem has grown significantly, with various libraries, tools, and adopters supporting its development and adoption across different programming languages and industries.

4. Difficulties and Factors to Consider When Implementing GraphQL

As GraphQL is not opinionated around how to design schemas rather has provided guidelines for designers, developers to follow , it is left to implementor as to how best he understand the ecosystem and designs the details . This leads to unnecessary complexity as and when system grows.

***4.1 Depth of Query*:** GraphQL's flexibility empowers clients to design queries that precisely match their data requirements, allowing for deeply [4] nested fields. This capability enhances the customization of responses, enabling clients to retrieve exactly the data they need. However, this flexibility can also lead to performance issues if not managed carefully.

When clients construct deeply nested queries, the server may need to perform complex operations to fulfill these requests. This can result in longer query execution times and increased server load, ultimately impacting the overall responsiveness of the application.

To mitigate these challenges, developers can implement caching strategies and query optimization techniques. Caching allows the server to store the results of frequently executed queries, reducing the need to recompute them for subsequent requests. Query optimization involves analyzing and restructuring queries to make them more efficient, such as reducing the depth of nested queries or batching multiple requests into a single query.

By carefully managing query complexity and implementing optimization strategies, developers can leverage GraphQL's flexibility while ensuring optimal performance and responsiveness in their applications.

***4.2 Security Concern*:** By default, GraphQL server exposes the schema to be introspected. This leads to exposing the domain of the application [5] and if not properly managed, hackers can use that definition to attach the server with malicious query / mutation. And this can lead to a Denial of Service (DoS) Attack

GraphQL tends to be incredibly verbose in its error messages. Attackers can use the information provided in the errors' description to build a valid query/ attack.

4.3 Batching Concern: - GraphQL's support for query batching [6] enables the aggregation of multiple queries into a single HTTP request, enhancing efficiency by reducing the number of network round trips. However, this approach can introduce vulnerabilities if not managed carefully.

One potential risk is application-level Denial of Service (DoS) attacks. In this scenario, an attacker could exploit query batching to send a large number of queries in a single request, overwhelming the server and causing it to become unresponsive. This can occur due to the translation of a single network call into a high volume of queries or object requests, leading to a strain on the server's CPU and memory resources.

To mitigate the risk of DoS attacks, developers can implement measures such as query complexity analysis and rate limiting. Query complexity analysis involves evaluating the complexity of incoming queries and rejecting those that exceed a predefined threshold. Rate limiting restricts the number of queries a client can send within a certain timeframe, preventing an excessive number of queries from being processed simultaneously.

By implementing these security measures, developers can protect their GraphQL APIs from potential DoS attacks and ensure the availability of their applications to legitimate users.

5. Conclusion

GraphQL represents a significant advancement in the field of API development, showcasing innovation and adaptability in its approach. Its core strengths lie in its flexible data retrieval capabilities, which allow clients to request precisely the data they need, reducing over-fetching and under-fetching of data. This precision-driven approach results in more efficient communication between frontend and backend systems, leading to a more streamlined development process and improved user experiences.

Furthermore, GraphQL's ability to reshape frontend-backend collaboration has had a profound impact on how data-driven applications are built and experienced. By providing a unified interface for data retrieval, GraphQL simplifies the development process and enables developers to iterate quickly on their applications.

As we conclude our exploration of GraphQL, it is crucial to recognize its transformative potential. However, it is also important to remain aware of the challenges that come with adopting GraphQL, such as query complexity and performance optimization. By understanding and addressing these challenges, developers can fully harness the power of GraphQL and continue to innovate in the field of API development.

References

[1]. Peter M. Mell and Timothy Grance. 2011. The NIST Definition of Cloud Computing. Technical Report. National Institute of Standards and Technology, Gaithersburg.

[2]. W. T. Tsai, X. Y. Bai, and Y. Huang. 2014. Software-as-a-service (SaaS): Perspectives and challenges. Sci. China Info. Sci. 57, 5 (2014), 1–15. https://doi.org/10.1007/s11432-013-5050-z

[3]. O. Hartig and J. Hidders, Defining Schemas for Property Graphs by Using the GraphQL Schema Definition Language, in: Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) co-located with SIGMOD/PODS '19: International Conference on Management of Data, GRADES-NDA'19, Association for Computing Machinery, 2019, pp. 1–11. doi:10.1145/3327964.3328495

[4]. "Query complexity and depth - Async-graphql Book". Online, Available: https://async-graphql.github.io/async-graphql/en/depth_and_complexity.html

[5]. S. Yerushalmi, "GraphQL API vulnerabilities and common attacks | Imperva," Blog, Sep. 05, 2019. Online Available: https://www.imperva.com/blog/graphql-vulnerabilities-common-attacks/

[6]. "Batching client GraphQL queries | Apollo GraphQL blog." Online, Available: https://www.apollographql.com/blog/batching-client-graphql-queries