ISRES Publishing

The Eurasia Proceedings of Science, Technology, Engineering & Mathematics (EPSTEM)
ISSN: 2602-3199

The Eurasia Proceedings of Science, Technology, Engineering & Mathematics (EPSTEM), 2024

Volume 32, Pages 230-237

IConTES 2024: International Conference on Technology, Engineering and Science

GraphQL: A Comprehensive Analysis of Its Advantages, Challenges, and Best Practices in Modern API Development

Ina Papadhopulli
Polytechnic University of Tirana

Hakik Paci
Polytechnic University of Tirana

Abstract: GraphQL, a query language and execution engine for application programming interfaces (APIs), has rapidly gained traction as a flexible and efficient alternative to traditional representational state transfer (REST) APIs. This paper aims to increase awareness of GraphQL by providing a comprehensive overview of its status in the industry, comparing its features and capabilities with those of REST, and addressing common concerns regarding security and performance. We explore the distinct advantages of GraphQL, such as its ability to allow clients to request precisely the data they need and its streamlined approach to data fetching and mutation. Additionally, we examine the inherent challenges and potential performance bottlenecks associated with GraphQL, offering best practices and solutions to mitigate these issues. The paper also delves into security considerations specific to GraphQL, emphasizing the importance of proper query validation, authorization, and rate limiting. Through detailed analysis, this paper seeks to elucidate the benefits and drawbacks of GraphQL, ultimately demonstrating why it is a powerful tool for modern API development and how it can be effectively implemented to enhance data interaction and overall application performance.

Keywords: GraphQL, APIs, REST, Performance, Security

Introduction

Representational State Transfer (REST) services are commonly employed for facilitating communication between web applications. REST services are utilized in scenarios such as mobile applications communicating with backend servers, web applications built with frameworks like React or Vue communicating with backend servers, or even when two backend servers interact. REST APIs provide access to server functionality through endpoints that clients can invoke. However, REST APIs have inherent limitations, including over-fetching, where more data than necessary is retrieved, and under-fetching, where insufficient data is retrieved for client needs.

Graph Query Language (GraphQL) was originally developed at Facebook in 2012 before they open-sourced it around 2016. It is a query language for APIs and a query runtime engine (The GraphQL Foundation). It acts as a middleware layer between frontend and backend data sources, including databases. Most people use it as a better API specification in their applications for their back-end systems (Doolittle, 2023). GraphQL enables clients to precisely specify the data they require in a single query, thereby mitigating issues related to over-fetching and under-fetching of data (Mukhiya et al., 2019). With GraphQL, a schema is defined that outlines how clients can query and mutate data. This schema does not replace the database schema but complements it. The database schema dictates how data is structured and stored, while the GraphQL schema specifies how clients interact with that data. When a GraphQL query is executed, the GraphQL server resolves the query by retrieving data from the database based on the query's specifications. This approach ensures optimal performance by fetching only necessary data.

- This is an Open Access article distributed under the terms of the Creative Commons Attribution-Noncommercial 4.0 Unported License, permitting all non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

- Selection and peer-review under responsibility of the Organizing Committee of the Conference

© 2024 Published by ISRES Publishing: www.isres.org

While GraphQL facilitates precise data retrieval, it still relies on a database for storing and retrieving data. GraphQL itself is not a database technology but rather a query language for APIs that enhances control over data fetching. In contrast to SQL queries, which require direct interaction with database tables and knowledge of the database schema, GraphQL queries are schema-agnostic. The GraphQL server handles query resolution and fetches data from appropriate data sources, which can include SQL databases, NoSQL databases, or other APIs. It achieves this through the use of resolvers, which are functions that map GraphQL queries to the underlying data sources. Resolvers ensure that the data requested by the client is efficiently retrieved and formatted according to the schema defined for the API. This abstraction allows clients to interact with a single GraphQL endpoint while the server manages the complexities of data retrieval from multiple sources (Microsoft, July 2024). GraphQL puts more control in the hands of the client, allowing front-end developers to fetch data in a way that suits their UI requirements more precisely, reducing the need for multiple endpoints or transformations.

GraphQL is compatible with various programming languages such as Java, Python, PHP, Ruby, JavaScript, etc., underscoring its versatility and adoption across different technology stacks. The rest of the paper is structured as follows: the next section provides a detailed description of GraphQL and compares it with other query languages and REST. The following two sections discuss the challenges associated with GraphQL and best practices for security and performance. The final two sections present the conclusions and recommendations for future work.

Graphql Comparison with Other Query Languages and Status

GraphQL

GraphQL is a query language for APIs and a runtime for executing those queries by using a type system you define for your data. To get/mutate data we need to define a schema. Schema provides flexibility to consumers regarding which attributes they want in the response. It is the contract between the consumer and provider on how to get and alter the data for the application. Query and Mutation are the root types in the schema representing read and write operations, respectively. Additionally, GraphQL supports the use of custom scalar types and enumerations, enabling developers to define precise data structures and enforce specific data formats.

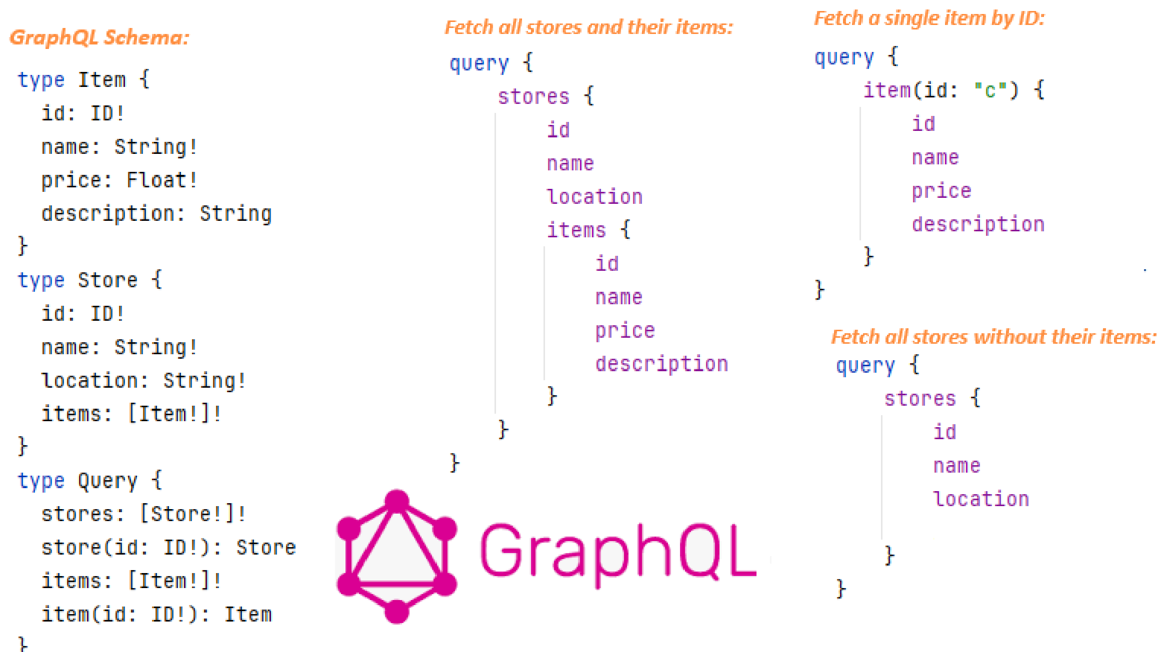

Figure 1. Example of GraphQL schema, queries for a store that has items

GraphQL offers the resolver mechanism that is a central concept that defines how to fetch the data for the fields in a query. Each field in a GraphQL schema is backed by a resolver function, which determines how that field's data should be obtained. Resolvers can be defined for queries, mutations, and individual fields within types. It is a powerful feature that enables precise, field-level data resolution. Allowing custom logic for individual fields

provides flexibility and control over how data is fetched and processed, making it particularly suitable for complex and nested data structures. If this feature is used appropriately, it can reduce the load on the server and improve the overall performance of the API.

Suppose we don't need the items as shown in the request: '*Fetch all stores without their items'* (Figure 1). If a resolver is not used, the items will still be retrieved from the data source but will not be included in the response since the user has not specified them in the request. Using a resolver for the Store ensures that the items are not fetched from the data source at all, increasing code efficiency. While the same behavior can be achieved with REST APIs, it is not as straightforward as it is with the GraphQL resolver mechanism.

GraphQL: Current Status and Developments

Over the past decade, several query languages have emerged as alternatives for API data access, each with its own strengths and use cases. SPARQL, for instance, is designed for querying Resource Description Framework (RDF) data and is widely used in semantic web applications. Cypher, developed by Neo4j, excels in handling graph databases. Gremlin, part of the Apache TinkerPop framework, provides a versatile graph traversal language that can operate across various graph databases (Quina-Mera et al., 2023). Among these, GraphQL has emerged as the most successful and widely adopted query language in recent years (Seifer et al., 2019).

Initially released publicly in 2015, GraphQL has since garnered significant attention and adoption within the development community. It is an open-source technology supported by an active and expanding community (GraphQL.com, 2024), making it suitable for teams of all sizes. Major industry players such as Microsoft, IBM, AWS, and Atlassian leverage GraphQL extensively in their applications (The GraphQL Foundation, 2024).

Despite its growing acceptance and adoption as a powerful tool for API development, GraphQL currently lacks a widely established scientific community dedicated to its study and advancement. While there is a noticeable trend of publications in reputable venues, there remain notable gaps in the existing literature. Specifically, shortcomings persist in terms of the maturity of empirical evidence supporting GraphQL's efficacy, validation through realistic use cases, and the comprehensive evaluation of its diverse quality characteristics and underutilized features (Quina-Mera et al., 2023). Addressing these gaps is crucial for enhancing the understanding and maximizing the potential of GraphQL in real-world applications.

GraphQL vs REST

Despite the many types of APIs, debates about two major paradigms have dominated the conversation in recent years: REST and GraphQL. Both are popular approaches for designing APIs, each with its own advantages and ideal use cases. However, they differ significantly in how they manage data traffic (IBM, June 2024). REST has different HTTP methods and separate endpoints for each API. In GraphQL we have Query and Mutation and there is only one endpoint. All calls go to the same endpoint and are HTTP POST methods. REST does not need Schema, GraphQL does. The choice between GraphQL and REST depends on the project's specific requirements and the data interactions' complexity. Some projects even use both, utilizing GraphQL for complex queries and REST for simpler endpoints.

When to Use GraphQL?

- *Complex Queries*: Ideal for retrieving complex or nested data structures in a single request, reducing over-fetching and under-fetching.
- *Flexible Data Retrieval*: Beneficial when API consumers need the flexibility to dictate the structure of the response, allowing clients to request only the data they need.
- *Rapid Development*: Supports agile development environments and frequent API updates, allowing schema changes without impacting existing clients.
- *Multiple Data Sources*: Simplifies the process of aggregating data from multiple sources or services into a single endpoint.
- *Strongly Typed Schema*: Uses a schema to define data structure, beneficial for validating requests and ensuring client-server data contract understanding.

When to Use REST?

- *Simple and Standardized*: Well-understood by many developers and suitable for simple needs met with standard HTTP methods (GET, POST, PUT, DELETE).
- *Caching*: Effectively leverages HTTP caching mechanisms to improve performance.
- *Stateless Operations*: Inherently stateless, with each client request containing all necessary information, simplifying design and scaling.
- *Standard Methods*: Uses standard HTTP methods and status codes, making it easy to understand and use.
- *Simple Relationships*: Easier to implement and maintain if the data model is straightforward without deeply nested or complex relationships.

GraphQL Security Challenges and Best Practices

Security should be considered carefully when utilizing GraphQL. GraphQL itself does not inherently pose security risks; however, if we do not follow the design guidelines and best practices related to security, we might expose vulnerabilities and compromise data security. (Shrey, 2024).

Common Security Issues Associated with GraphQL

a) *Introspection Queries*: This is a special type of query that allows clients to query the schema itself for information about types, fields, and operations that are available in the GraphQL API. Usually, the schema is not supposed to be public. If introspection queries are enabled, it would lead to:

1) Exposure to internal structure
2) Discovery of hidden fields and endpoints
3) Facilitation of targeted attacks.

b) *Excessive Data Exposure*: Situations where more data than necessary is exposed through the API. This is often done unintentionally, and this can lead to various security and privacy issues. Causes of excessive data exposure:

i. Over-fetching: The client requests more data than needed. This can happen even though with GraphQL we can specify exactly what data we want.
ii. Unprotected fields: All fields can be accessed by the clients if not explicitly protected in the schema.

c) *SQL Injection*: This type of attack can happen when user inputs are directly used in SQL queries without proper validation and sanitization. GraphQL allows for complex queries with parameters. This gives the possibility to attackers to write malicious queries to manipulate the SQL queries executed by the server.

```
query {
    item(id: "'); DROP TABLE store; ") {
        id
        name
        price
        description
    }
}
```

The resolver could generate the following SQL query:

```
SELECT * FROM item WHERE id = ''); DROP TABLE store;
```

Figure 2. Example of a SQL injection for the schema defined in Figure 1

d) *Denial of Service (DoS) Attacks*: An attacker aims to make the GraphQL API unavailable to legitimate users by overwhelming the server with a large number of expensive or complex queries. This type of attack can be achieved through various methods, including:

i. Batched Queries and Resource Exhaustion: Attackers can send batched queries with multiple operations into a single request, causing significant resource consumption on the server.
ii. Aliases-based attack: By using the aliases feature, the attacker can circumvent rate-limiting measures that typically count the number of HTTP requests from an IP address.

iii. Deep Recursive Query Attack: Types can reference each other leading to potentially infinite recursion. This can crash the server due to excessive resource consumption.

Best Practices to Mitigate Security Risks

a) *Rate Limiting and Throttling*: Implement rate limits to prevent abuse and overuse of the API.
b) *Query Complexity Analysis*: Evaluate and limit the complexity of GraphQL queries to protect the API from resource exhaustion and DoS attacks.
c) *Introspection Control*: Disable introspection. If it is needed for development or debugging purposes, ensure it is disabled in production.
d) *Authentication and Authorization*: Define roles and specify the permissions for each role. Check permissions within resolvers to ensure users can only perform actions they are authorized for.
e) *Logging*: Log information to detect anomalous behavior, track API usage, and gather information about errors.
f) *Monitoring*: Enhance security through real-time alerts, user behavior analytics, resource utilization analysis, and more.
g) *Avoid SQL Injection*:

   i. Always use parameterized queries
   ii. Validate and sanitize inputs
   iii. Use Object-Relational Mappers (ORMs) because they handle parameterization automatically.
   iv. Apply least privilege principles

GraphQL Performance Challenges and Best Practices

GraphQL offers a powerful and flexible way to query and manipulate data, but with this power comes several performance challenges. This section explores these challenges, particularly when dealing with large graphs and high volumes of API requests, and outlines solutions to mitigate them.

Dealing with Large Graphs

Handling large graphs in GraphQL can lead to performance bottlenecks, especially when deep nesting or complex queries are involved. Some strategies to manage large graphs effectively include:

a) *Efficient Data Loading*: Utilize techniques like data loaders to batch and cache requests, reducing the number of database queries.
b) *Pagination and Filtering*: Implement pagination and filtering to limit the amount of data returned in a single query, thus avoiding over-fetching.
c) *Query Complexity Analysis*: Use tools to analyze and limit query complexity, preventing overly complex or expensive queries from degrading performance.

Maintaining GraphQL APIs

Maintaining a GraphQL API with a high volume of requests presents several challenges, including schema management, performance optimization, and ensuring high availability. Here are some key aspects and solutions:

Schema Visualization and Management

- *Schema Visualization*: Tools for visualizing the schema help understand how objects are connected and identify potential bottlenecks. Visualization aids in managing and optimizing the schema.
- *Checks and Insights*: Improved schema-checking tools, such as Apollo Studio, offer validation, change tracking, and usage analytics. These tools ensure that the schema remains well-defined, compatible, and error-free.

Architectural Approaches

There are two primary architectural approaches for implementing GraphQL: Monolith and Supergraph (Federated GraphQL) (Apollo Graph Inc., 2024). Understanding these architectural approaches is crucial for making informed decisions about how to implement GraphQL in a way that best suits the needs of a given application or organization. Each approach has its advantages and disadvantages:

Monolith Architecture:

- *Advantages:*

  - *Simplicity*: Easier to set up and manage with a single schema and resolver map.
  - *Performance*: Direct inter-resolver communication minimizes latency.
  - *Deployment*: Only one service to manage, reducing overhead.
  - *Consistency*: Uniform coding practices and schema design across the codebase.

- *Disadvantages:*

  - *Scalability*: Harder to scale as the application grows.
  - *Flexibility*: Riskier updates affecting the entire system.
  - *Team Collaboration*: Potential for conflicts in a single codebase.
  - *Resilience*: Failure in one part can impact the whole application.

Supergraph Architecture:

- *Advantages*:

  - *Scalability*: Independent scaling of subgraphs addresses specific performance bottlenecks.
  - *Flexibility*: Isolated changes to subgraphs without affecting the entire system.
  - *Team Collaboration*: Teams can work independently on different subgraphs.
  - *Resilience*: Failures in one subgraph do not affect the entire system.

- *Disadvantages*:

  - *Complexity*: More complex setup and management with additional infrastructure.
  - *Performance*: Potential latency from inter-subgraph communication.
  - *Deployment*: More challenging deployment coordination of multiple services.
  - *Consistency:* Requires robust schema management across subgraphs.

Query Batching

Modern applications often require several requests to render a single page. Not only does this cause a performance overhead (different components may be requesting the same data) but it can also cause a consistency issue. Query batching allows multiple operations to be combined into a single request, reducing network overhead and improving performance. This technique ensures efficient client-server communication by minimizing the number of network requests (Apollo Graph Inc, 2024).

Persisted Queries

Persisted Queries involve storing pre-defined queries on the server side, reducing the payload size and improving performance. Clients send unique identifiers instead of full query strings, which the server uses to retrieve and execute the stored queries. This approach also enhances security by preventing malicious queries and ensuring consistent authorization.

High Availability and Reliability

Ensuring high availability and reliability involves implementing redundant systems and failover mechanisms. It also includes setting up performance alerts and monitoring metrics to proactively address issues before they impact users.

Conclusion

GraphQL can be considered a flexible and efficient alternative to traditional REST APIs. Through this comprehensive analysis, we have explored the distinct advantages of GraphQL, including its ability to allow clients to request precisely the data they need and its streamlined approach to data fetching and mutation. We have also highlighted the inherent challenges and potential risks associated with GraphQL, providing best practices and solutions to mitigate these issues. Implementing robust security measures, such as query complexity analysis, rate limiting, and proper authentication and authorization, is crucial to safeguard against potential vulnerabilities like DoS attacks and SQL injection.

In addressing performance challenges, strategies such as efficient data loading, pagination, and query batching can be used especially when dealing with large graphs and high volumes of requests. Additionally, adopting appropriate architectural approaches, whether monolith or supergraph and leveraging tools for schema visualization and management can enhance the maintainability and scalability of GraphQL APIs. Organizations can effectively manage these challenges by adhering to best practices in security, performance optimization, and architectural design. GraphQL offers great potential, ensuring robust and efficient API implementations that meet the dynamic needs of today's applications.

Recommendations

While this analysis has provided a comprehensive overview of GraphQL's advantages, challenges, and best practices, several areas need further exploration. Future work could focus on developing more sophisticated tools for automated query complexity analysis and enhanced introspection control to strengthen GraphQL API security further. Additionally, empirical studies evaluating GraphQL's performance in large-scale, real-world applications would provide valuable insights into its scalability and efficiency. Investigating the integration of GraphQL with emerging technologies such as machine learning could also reveal new opportunities for innovation and optimization. Finally, establishing a more robust scientific community dedicated to the study and advancement of GraphQL will be crucial for driving its evolution and addressing any emerging challenges in the field.

Scientific Ethics Declaration

The authors declare that the scientific ethical and legal responsibility of this article published in EPSTEM Journal belongs to the authors.

Acknowledgments or Notes

* This article was presented as an oral presentation at the International Conference on Technology, Engineering and Science (www.icontes.net) held in Antalya/Turkey on November 14-17, 2024.

References

Apollo Graph Inc. (2024, June 25). *Query batching*. https://www.apollographql.com/docs/router/executing-operations/query-batching/

Shrey, A. (2024,). Hacking GraphQL APIs for fun & profit. *Toptal Global Talent Conference, 2024*

Doolittle, J. (2023). APIs with GraphQL. *IEEE Software*, *40*(2), 118-120.

Mukhiya, S., Rabbi, F., Ka I Pun, V., Rutla, A., & Lamo, Y. (2019). A GraphQL approach to healthcare information exchange with HL7 FHIR. *The 9th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2019),* 4-7,

The GraphQL Foundation. (2024, June 21). *Introduction to GraphQL*. Retrieved from https://graphql.org/learn/

GraphQL.com. (2024, July 10). Community. Retrieved from https://graphql.com/community/

The GraphQL Foundation. (2024, June 25). *Who's using GraphQL*? Retrieved from https://graphql.org/users/

Apollo Graph Inc. (2024, July 07). Optimizing GraphQL APIs for operational success. Retrieved from https://www.apollographql.com/events

Seifer, P., Härtel, J., Leinberger, M., Lämmel, R., & Staab, S. (2019). Empirical study on the usage of graph query languages in open source Java projects. *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE'19) ACM,* 152–166.

IBM (2024, June 01). *GraphQL vs. REST API: What's the difference?* Retrieved from: https://www.ibm.com/blog/graphql-vs-rest-api/

Microsoft. (2024, July 09). *Configure a GraphQL resolver*. Retrieved from https://learn.microsoft.com/en-us/azure/api-management/configure-graphql-resolver

Quiña-Mera, A., Fernandez, P., García, J. M., & Ruiz-Cortés, A. (2023). GraphQL: A systematic mapping study. *ACM Computing Surveys*, *55*(10), 1-35.

Author Information

Ina Papadhopulli
Polytechnic University of Tirana
"Mother Tereza" Square, Tirana, 1001, Albania
Contact e-mail: *ipapadhopulli@fti.edu.al*

Hakik Paci
Polytechnic University of Tirana
"Mother Tereza" Square, Tirana, 1001, Albania

To cite this article:

Papadhopulli, I., & Paci, H. (2024). GraphQL: A comprehensive analysis of its advantages, challenges, and best practices in modern API development. *The Eurasia Proceedings of Science, Technology, Engineering & Mathematics (EPSTEM), 32,* 230-237.