American Research Journal of Computer Science and Information Technology

Volume 8, Issue 1, 31-35 Pages
Research Article | Open Access
ISSN (Online)- 2572-2921
DOI : 10.21694/2572-2921.25006

Comparison of API Construction Approaches: REST vs. GraphQL in Modern Applications

Danylo Sereda

B.Sc. in Automation and Computer-Integrated Technologies, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine.

ABSTRACT

With the increasing complexity and scalability of modern applications, choosing the optimal API architectural style has become a key task for developers. REST and GraphQL are two popular approaches, each with its own unique features, advantages, and limitations. This article is devoted to comparing these two approaches in order to facilitate the decision-making process for developers and companies. We analyze in detail the differences between REST and GraphQL, highlighting their strengths and weaknesses in different usage contexts. Special attention is paid to how these differences can affect productivity, development time, and end user satisfaction. Understanding these aspects will allow you to choose the approach that best meets the requirements of a particular project, minimizing risks and contributing to effective application development.

KEYWORDS: *applications, REST, GraphQL, performance, efficient development, risks.*

INTRODUCTION

With the rapid advancement of technologies and the increasing complexity of modern applications, developers and technical managers face critical decision-making regarding application architecture. One such crucial decision is selecting the appropriate API architectural style. REST and GraphQL, as the two most prominent methodologies, offer distinct approaches to data exchange between client and server. However, understanding their specific characteristics and optimal use cases can present a significant challenge for professionals.

The objective of this article is to provide a clear and comprehensive comparative analysis of these two approaches, with a focus on their application in modern software development. We aim to assist developers and technical managers in making well-informed decisions tailored to their project requirements. To achieve this goal, the study focuses on addressing several key tasks: we thoroughly examine the core concepts of REST and GraphQL, evaluate their strengths and weaknesses across different use cases, and analyze real-world implementations and case studies of both paradigms.

Furthermore, particular emphasis is placed on identifying scenarios where one approach may prove more advantageous than the other. This study provides valuable insights that can serve as a practical guide for professionals seeking to optimize development workflows and enhance the overall efficiency of their projects.

REST

REST (Representational State Transfer) is an architectural approach that facilitates data interaction via the HTTP protocol and represents information in multiple formats, including XML, JSON, and others [3]. This methodology simplifies access to web services by leveraging existing standards rather than introducing additional data processing layers into the network stack. As a lightweight alternative to the more complex SOAP protocol, REST operates on the principle of self-descriptive resources accessible through HTTP. In this paradigm, services are treated as addressable resources that can be consumed as needed.

To retrieve information about a user with ID 1, a client would access the URI *http://www.example.com/rest/user/1* through a web browser or API call. RESTful services utilize standard HTTP methods – including GET (retrieve), PUT (update), POST (create), and DELETE (remove) – to perform CRUD (Create, Read, Update, Delete) operations on resources. In this architecture, each URI (Uniform Resource Identifier) serves as a unique address for resource identification and manipulation [6] (fig.1.) .

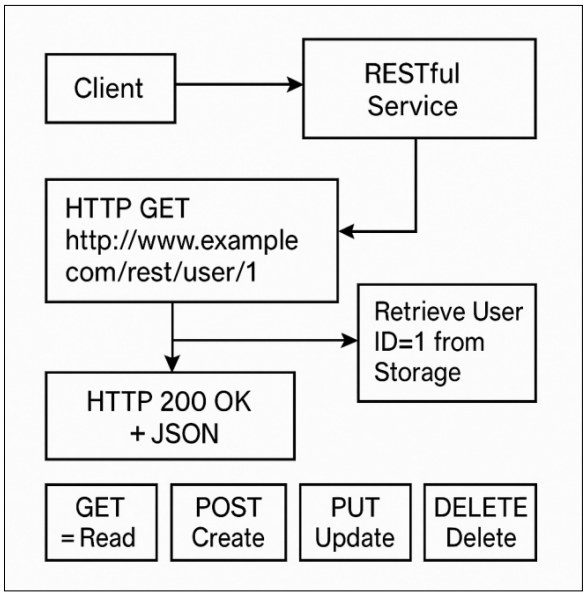

Figure 1. Diagram showing the process of obtaining user information [6]

In his seminal 2000 doctoral dissertation, Roy Fielding first formalized the API architectural style that has since become the dominant paradigm for web services. Now universally recognized as REST (Representational State Transfer), this approach is defined by six mandatory architectural constraints that must be satisfied for full RESTful compliance. Compared to SOAP's rigid specifications, REST offers a more flexible framework for distributed systems design.

This approach delivers server-side information in standardized, widely-adoptable formats — predominantly JSON and XML. As a self-descriptive architectural style, REST was specifically engineered for large-scale consumption by diverse API clients. Its constrained design inherently facilitates straightforward access to server-hosted data resources while maintaining platform independence.

The architecture enables independent evolution of client and server components through a strict separation of concerns. Client-server interactions are conducted via standardized protocols, ensuring uniform access across diverse devices and applications. A fundamental characteristic is its stateless nature — all necessary processing data is contained within each request, thereby relieving the server of any obligation to maintain session information.

The system's layered architecture is further enhanced by data caching capabilities and the server's ability to transmit executable code to the client side. It should be noted that in practice, many services only partially adhere to these architectural principles, failing to implement the complete set of RESTful requirements [6].

HATEOAS (Hypermedia as the Engine of Application State) constitutes a fundamental constraint in REST architecture. This principle enables independent client-server evolution while maintaining seamless interoperability. The hypermedia approach mandates that every REST API response includes machine-interpretable metadata, comprehensively describing all available interface capabilities and state transitions [9].

The HTTP infrastructure is leveraged efficiently within this approach, where large-scale services are decomposed into discrete resources while maintaining an RPC-style foundation. This architectural paradigm enables both API developers and consumers to evolve their systems independently while guaranteeing uninterrupted interoperability.

Even high-quality REST API interfaces often fail to meet the highest standards today. HATEOAS represents the most advanced form of REST architecture, but its implementation comes with significant challenges. Modern API clients typically lack sufficient intelligence and functionality to fully support this approach, making it particularly difficult to achieve this level of REST maturity.

HATEOAS primarily serves as a guiding principle that shapes the strategic evolution of API architecture in accordance with REST's core concepts.

GRAPHQL

In contrast to traditional REST APIs, GraphQL introduces a fundamentally novel approach to data interaction (Fig. 2). This innovative query language and specification effectively addresses issues of data over-fetching and under-fetching. Clients gain unprecedented flexibility by requesting precisely the data they require, substantially simplifying API consumption. Designed specifically for information retrieval optimization, GraphQL delivers more efficient and intuitive data interaction capabilities.

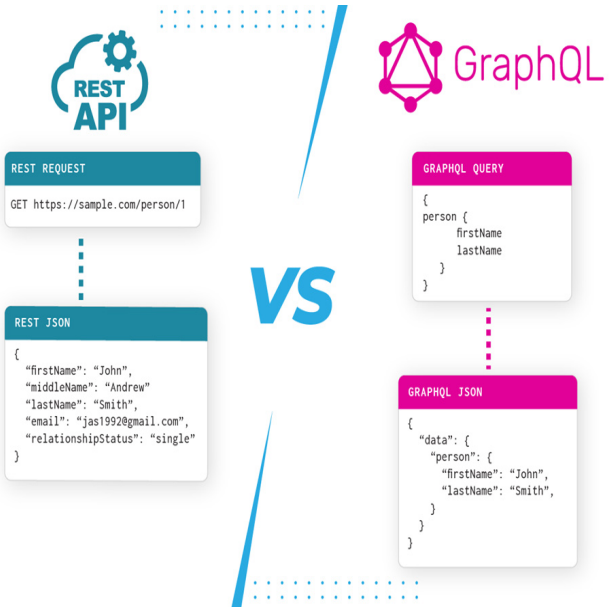

Figure 2. RESTful API compared to GraphQL [7]

Mastering GraphQL has become increasingly accessible to professionals of varying skill levels due to its intuitive syntax and conceptual framework [7]. With high-quality learning resources, even novice developers can successfully implement this technology. A foundational understanding of web development—including HTTP principles, API interactions, and data processing—significantly facilitates

the learning process. Integrating GraphQL's server and client components is notably streamlined with proficiency in JavaScript or other supported programming languages. Familiarity with development ecosystem tools, particularly Node.js and the npm package manager, provides additional advantages.

Facebook's development team introduced GraphQL to the world – an innovative query language and API runtime that surpasses traditional REST APIs in flexibility and efficiency. This technology enables developers to eliminate problems of data over-fetching and under-fetching by requesting precisely the required information. With fundamental knowledge in data modeling and web development, one can begin learning GraphQL. Even novice programmers can quickly master this technology and implement it in their own projects with moderate perseverance [8].

Precise data retrieval is the principal advantage of GraphQL technology. Traditional RESTful APIs often suffer from two opposing issues: they either return excessive data or provide insufficient information.

GraphQL addresses this dilemma elegantly. Users define the exact scope of the required data within their queries. This approach ensures that clients receive precisely what they requested—no more, no less.

The ability to request data with pinpoint precision eliminates common shortcomings of REST architecture. Instead of receiving bulky payloads of unnecessary information or making additional requests to obtain missing details, GraphQL enables clients to construct perfectly balanced queries.

There exists a specialized method of communication with databases for data retrieval and analysis—declarative queries [8]. Complex information requirements can be expressed without delving into the technical intricacies of the process. Users articulate their needs using a dedicated language that renders the querying process intuitive and accessible.

This approach allows users to focus on the essence of the query. The simplicity of expressing required data is a key advantage. The variety of possible formulations provides flexibility. Both concise and elaborate expressions are equally effective. The structure of queries may vary significantly in scope and complexity [3].

In contrast to the ambiguity often associated with REST APIs, GraphQL's strongly typed schema offers developers a precise blueprint of the data. This technology not only defines explicit types and structures for both server and client sides but also helps eliminate uncertainties in communication between application components.

One of GraphQL's key capabilities is the consolidation of heterogeneous data sources into a single access point, which significantly simplifies data handling. The clear definition of the schema ensures stable interactions within the system, thereby enhancing development efficiency.

With GraphQL, we can gather all the necessary information from a single source, avoiding the need to query multiple disparate resources. This greatly streamlines the data retrieval process.

Thanks to a unified interface, in contrast to REST with its multiple endpoints, API administration becomes significantly simpler—this is one of the key advantages of GraphQL technology.

The ability to check the availability of fields in GraphQL allows developers to gradually phase out deprecated elements instead of removing them abruptly [3]. This approach ensures evolutionary development and version management without interrupting the operation of existing client applications, as GraphQL structures can evolve over time without causing disruptions for users.

The implementation of GraphQL entails significant initial effort. Developing and configuring the GraphQL server schema is considerably more complex and time-consuming compared to building a traditional RESTful API.

Mastering GraphQL can present a significant challenge for beginners. The need to learn specific concepts and adhere to particular patterns requires considerable initial time and effort investments.

However, once the initial learning curve is overcome, GraphQL provides access to powerful functionality [7]. The increased flexibility and efficiency in query handling ultimately offset the initial complexities and additional configurations, offering more optimal ways of data retrieval.

The initial setup of GraphQL requires considerable effort, surpassing the complexity of creating RESTful interfaces. Beginners will need to delve into concepts and guidelines, which may present challenges in the early stages.

Even with GraphQL, there is a risk of system overload due to excessive queries. Poorly formulated queries can retrieve redundant information, placing unnecessary load on the server. While GraphQL minimizes the number of unnecessary server requests, the issue is not completely eliminated.

To prevent performance issues, careful schema design and query optimization are essential. It is important to take additional steps to avoid data over-fetching, especially when requesting too much information at once.

To achieve optimal performance, it is crucial to request only the necessary information by carefully designing the schema structure and optimizing the queries.

Security when working with GraphQL requires special attention. Developers must implement protective mechanisms against potential threats, particularly when dealing with complex queries with excessive nesting [4]. Among the effective security measures, query rate limiting and pre-validation stand out, as they help minimize potential vulnerabilities in API interfaces.

Unlike RESTful APIs, caching strategies in GraphQL require special attention due to the uniqueness of client queries, which creates challenges in performance optimization. The overload protection system includes time limits on the number of allowed requests and validation to ensure queries comply with established parameters. Efficient response caching becomes a challenging task when each query is unique, significantly complicating the caching process compared to traditional API interfaces. The distinguishing features of REST and GraphQL are presented in Table 1.

Table 1. Comparing REST и GraphQL (compiled by the author).

Criterion	REST	GraphQL
Description	An architectural style widely adopted as the de facto standard for API design.	A query language specifically designed to address common API integration challenges.
Architecture	Server-centric (clients interact with predefined endpoints).	Client-centric (queries are dynamically tailored to client needs).
Data Fetching	Often results in over-fetching or under-fetching of data.	Eliminates issues of over-fetching and under-fetching.
Response Format	Supports XML, JSON, and YAML response formats.	Exclusively returns responses in JSON format.
Response Structure	Determined by the server.	Defined by the client's query requirements.
Caching	Built-in response caching by default.	Lacks automatic caching mechanisms.
Security	No native support for type checking or automatic documentation	Provides type safety guarantees and generates documentation automatically

Consider the development of a library system that tracks authors and their works. When using traditional REST architecture, the client-side application in React is forced to make separate requests to different endpoints—retrieving data about literary works and separately about their creators. In contrast, GraphQL allows for combining these interrelated information needs into a single composite query.

For projects with dynamically changing requirements and the need for complex data queries, GraphQL is the preferred choice. However, if the application is characterized by relative simplicity and a clearly defined resource model, the REST approach may prove to be a more suitable solution.

The uniqueness of queries in client requests necessitates a thoughtful approach to storing responses in order to optimize the servicing process.

A poorly designed GraphQL schema can lead to serious consequences: decreased performance, difficulties in handling heavy traffic, and challenges in serving numerous users. To prevent ongoing technical difficulties and the need for regular adjustments, it is critical to pay special attention to data structuring and establishing correct relationships between different informational components [6].

Developing an efficient schema involves certain resource costs; however, these investments are justified considering the potential issues that may arise from insufficiently careful data architecture planning.

Transitioning to GraphQL is a challenging task for development teams. This journey can be especially difficult for specialists accustomed to working with RESTful APIs. The new technology requires immersion in unfamiliar concepts and approaches, creating certain obstacles in the learning process. Mastering this tool becomes a true test, requiring time and effort to adapt to a different programming paradigm.

CONCLUSION

In this article, we have provided a comparative analysis of two key approaches to API development—REST and GraphQL—to assist developers and technical managers in making informed decisions when selecting a technology for their projects. The main concepts of both approaches were examined based on a review of current scientific publications, technical documentation, and real-world case studies.

Our analysis shows that REST, as a more mature and widely used technology, offers simplicity and reliability in application architectures where data has stable and predictable structures. Its natural integration with the HTTP protocol and rich ecosystem make it the preferred choice for many traditional server-side solutions.

GraphQL, on the other hand, stands out for its flexibility and efficiency in cases requiring dynamic and complex queries. This approach is particularly useful for mobile applications and complex interfaces, where optimizing network requests is critically important. However, it requires more complex setup and understanding, which may increase the initial design complexity of the system.

Thus, the choice between REST and GraphQL should be made based on the specific needs and context of the project. REST is suitable for projects with predictable data structures and minimal changes, while GraphQL can be more effective for dynamic systems with evolving data requirements. The research shows that neither approach is universally superior: their application depends on the specific technical and business requirements of the project.

REFERENCES

1. Kozhanov P.S., Gotskaya I.B. Comparative analysis of approaches to client-server interaction organization in modern web applications, using REST API and GRAPHQL

as examples // Modern Science-Intensive Technologies. - 2024. - Vol. 5 (2). - pp. 284-293.

2. Tonkushin M.V., Gudkov K.V. Comparative analysis of GraphQL and REST technologies // Modern Information Technologies. - 2019. - Vol. 29. - pp. 127-131.
3. Gridin V.N., Anisimov V.I., Vasiliev S.A. Methods for improving the performance of modern web applications // Izvestiya of South Federal University. Technical Sciences. - 2020. - Vol.4. - pp. 193-200.
4. Buna S. GraphQL in action (1st edition.) — Manning Publications. - 2021. - 375 p.
5. Ananchenko I.V., Churikov E.A. Optimization of HTTP requests by transitioning from REST API to GraphQL // Current Issues in Modern Science: Proceedings of the III International Scientific and Practical Conference (Penza, September 25, 2022). Penza: Science and Enlightenment (IP Gulyaev G.Y.). - 2022. – pp. 11-14.
6. Kumari S., Rath S. K. Performance comparison of soap and rest based web services for enterprise application integration // 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI). — 2015. — pp. 1656-1660.
7. Mikula M., Denkovski M. Comparison of REST and GraphQL Web Technology Performance // Journal of Computer Science-Institute of Science. - 2020. - pp. 309-316.
8. Serrano N., Hernantes J., Gallardo G. Service-oriented architecture and legacy systems //IE software. – 2014. – Vol. 31 (5). – pp. 15-19.
9. Subramanian H. "Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs" — Packt Publishing. - 2019. - p. 378.
10. Tihomirovs J., Grabis J. Comparison of soap and rest based web services using software evaluation metrics //Information technology and management science. – 2016. –Vol. 1 (19). – pp. 92-97.

Citation: Danylo Sereda, "Comparison of API Construction Approaches: REST vs. GraphQL in Modern Applications", American Research Journal of Computer Science and Information Technology, Vol 8, no. 1, 2025, pp. 31-35.