GraphQL vs. REST: Choosing the Right API for Frontend

Vivek Jain

Software Development Manager, Comcast, PA, USA
vivek65vinu@gmail.com

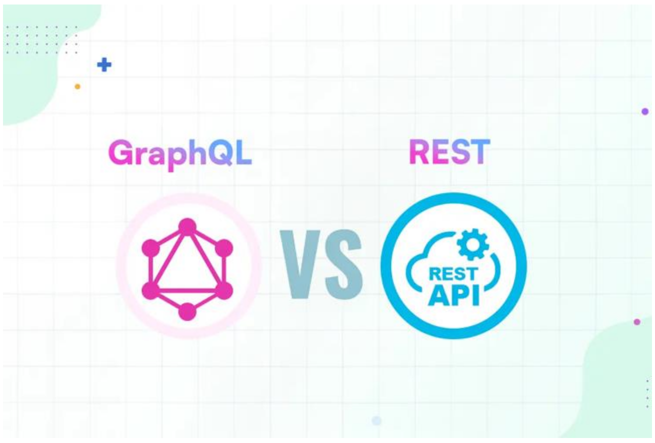
Abstract

Application Programming Interfaces (APIs) are the backbone of modern web and mobile application development, facilitating communication between frontend and backend systems. Representational State Transfer (REST) has been a widely adopted architectural style, but GraphQL, introduced by Facebook, has emerged as an alternative with unique advantages. This paper evaluates GraphQL and REST in terms of performance, scalability, development ease, and use cases, providing insights to guide developers in choosing the optimal API for their frontend requirements.

Keywords: REST, GraphQL, API Design, Frontend Development, Data Query Optimization

1. Introduction

The growing demand for dynamic and interactive applications has pushed developers to reconsider traditional API approaches. REST, a stateless client-server model, has dominated API design for over a decade. However, REST's limitations in complex and data-intensive scenarios have paved the way for GraphQL. This paper compares GraphQL and REST by analyzing their characteristics, strengths, and weaknesses to assist in selecting the right API for frontend development.



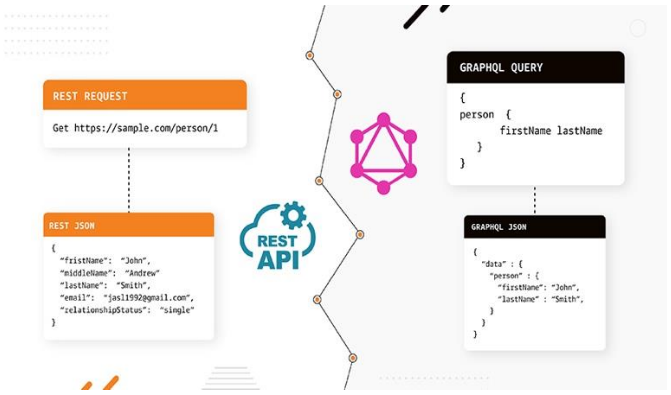
2. Overview of REST and GraphQL

2.1 REST

REST is an architectural style that utilizes HTTP methods (GET, POST, PUT, DELETE) to interact with resources identified by URLs. Each API endpoint in REST corresponds to a fixed resource, making it simple and easy to implement. However, issues like over-fetching and under-fetching of data often arise in complex applications.

2.2 GraphQL

GraphQL is a query language and runtime for APIs that allows clients to specify the exact data they need. Unlike REST, GraphQL exposes a single endpoint and relies on a schema to define available types and operations. This flexibility can improve efficiency but adds complexity to server implementation.



```
Get https://sample.com/person/1
```

```
{
  "fristName":  "John",
  "middleName":  "Andrew"
  "lastName":  "Smith",
  "email":  "jasl1992@gmail.com",
  "relationshipStatus":  "single"
}
```

```
{
person  {
        firstName lastName
   }
}
```

```
{
  "data" : {
    "person" : {
      "firstName": "John",
      "lastName" : "Smith",
    }
  }
}
```

3. Key Comparisons

3.1 API Versioning

• **REST:** Requires separate endpoints for different versions, increasing maintenance overhead.

• **GraphQL:** Avoids explicit versioning by extending the schema, enabling backward compatibility.

3.2 Performance

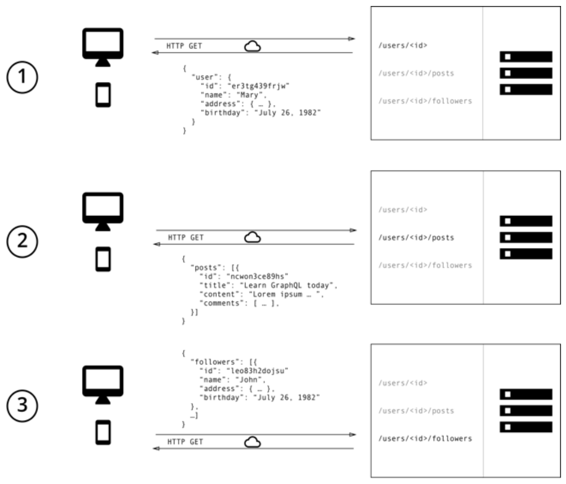
• **REST:** Better suited for simple, well-defined use cases. However, chaining multiple requests for related data can increase latency.

• **GraphQL:** Optimized for complex and nested queries but may introduce server-side performance bottlenecks due to dynamic query execution.

3.3 Data Fetching

• **REST:** Often results in over-fetching or under-fetching due to fixed endpoints. With a REST API, you would typically gather the data by accessing multiple endpoints.

In the example, these could be /users/<id> endpoint to fetch the initial user data. Secondly, there's likely to be a /users/<id>/posts endpoint that returns all the posts for a user. The third endpoint will then be the /users/<id>/followers that returns a list of followers per user.



```
{
  "user": {
    "id": "er3tg439frjw"
    "name": "Mary",
    "address": { … },
    "birthday": "July 26, 1982"
  }
}
```

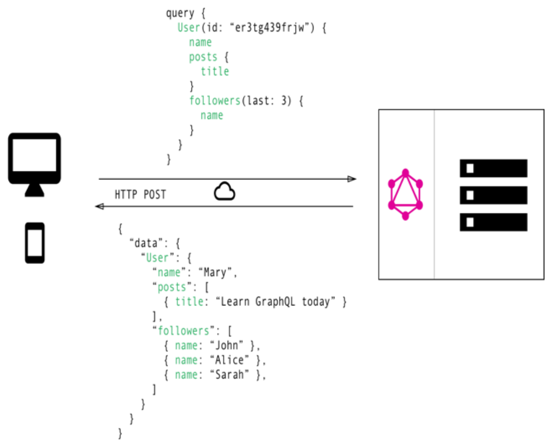
```
{
  "posts": [{
    "id": "ncwon3ce89hs"
    "title": "Learn GraphQL today",
    "content": "Lorem ipsum … ",
    "comments": [ … ],
  }]
}
```

```
{
  "followers": [{
    "id": "leo83h2dojsu"
    "name": "John",
    "address": { … },
    "birthday": "July 26, 1982"
  },
  …]
}
```

With REST, you have to make three requests to different endpoints to fetch the required data. You're also *overfetching* since the endpoints return additional information that's not needed.

• **GraphQL:** Allows clients to request only the required fields, reducing payload size and improving performance, especially for low-bandwidth environments.

In GraphQL on the other hand, you'd simply send a single query to the GraphQL server that includes the concrete data requirements. The server then responds with a JSON object where these requirements are fulfilled.



```
query {
  User(id: "er3tg439frjw") {
    name
    posts {
      title
    }
    followers(last: 3) {
      name
    }
  }
}
```

```
{
  "data": {
    "User": {
      "name": "Mary",
      "posts": [
        { title: "Learn GraphQL today" }
      ],
      "followers": [
        { name: "John" },
        { name: "Alice" },
        { name: "Sarah" },
      ]
    }
  }
}
```

3.4 Error Handling

• **REST:** Utilizes HTTP status codes, which are intuitive for developers.

• **GraphQL:** Returns all errors in a single response, which may require additional handling on the client side.

3.5 Tooling and Ecosystem

• **REST:** Mature tooling (Postman, Swagger) and widespread adoption.

• **GraphQL:** Growing ecosystem with tools like Apollo Client and Relay but requires a learning curve for schema design.

4. Use Cases

4.1 When to Use REST

• Applications with simple data requirements and well-defined resources.

• Scenarios where caching is critical and can be implemented via HTTP standards.

• Legacy systems or teams with REST expertise.

4.2 When to Use GraphQL

• Applications with complex or nested data relationships, such as social media platforms.

• Scenarios demanding high flexibility in frontend development.

• Projects requiring rapid iteration and schema evolution.

5. Challenges

5.1 REST Challenges
• Lack of flexibility in handling complex queries.
• Over-fetching or under-fetching leading to inefficiencies.

5.2 GraphQL Challenges
• Increased server-side complexity.
• Potential for over-querying, impacting server performance.
• Steeper learning curve for developers new to GraphQL.

7. Conclusion

The choice between GraphQL and REST depends on the application's specific needs. REST remains a reliable option for straightforward use cases with predictable data patterns. In contrast, GraphQL excels in scenarios requiring flexible, efficient data retrieval and frequent schema evolution. By understanding the trade-offs, developers can make informe decisions to optimize both development workflows and user experience.

References

[1] Roy T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," University of California, Irvine, 2000.

[2] Lee Byron and Dan Schafer, "GraphQL: A Data Query Language," Facebook, 2015.

[3] M. Fowler, "API Design: REST vs GraphQL," ThoughtWorks Insights, 2019.

[4] J. Resig, "Understanding API Evolution," ACM SIGWEB, 2020.