REST vs GraphQL

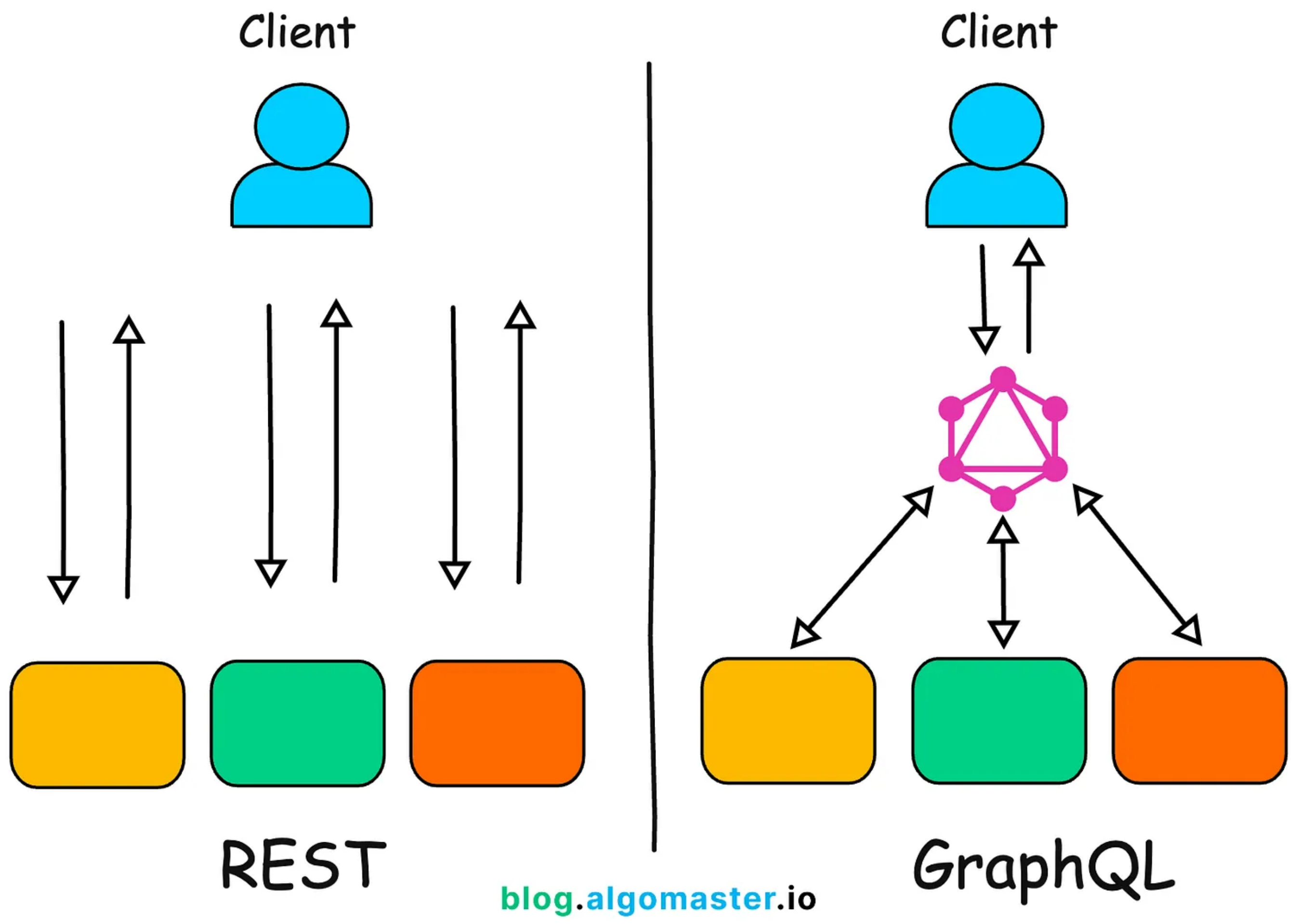
Which one should you use?

ASHISH PRATAP SINGH
MAR 11, 2025

APIs are the backbone of modern applications, acting as the bridge between **client applications and backend servers**.

Among the many API design choices, **REST** and **GraphQL** have emerged as two dominant approaches.

Both offer powerful ways to retrieve and manipulate data, but they are built on fundamentally different philosophies.



REST, a time-tested architectural style, structures APIs around **fixed endpoints and HTTP methods**, making it intuitive and widely adopted.

On the other hand, GraphQL, a newer query language developed by Facebook, takes a more **flexible and efficient approach**, allowing clients to request exactly the data they need in a single request.

In this article, we'll break down REST and GraphQL, compare their differences, and help you decide which one is best suited for your use case.

---

1. What is REST?

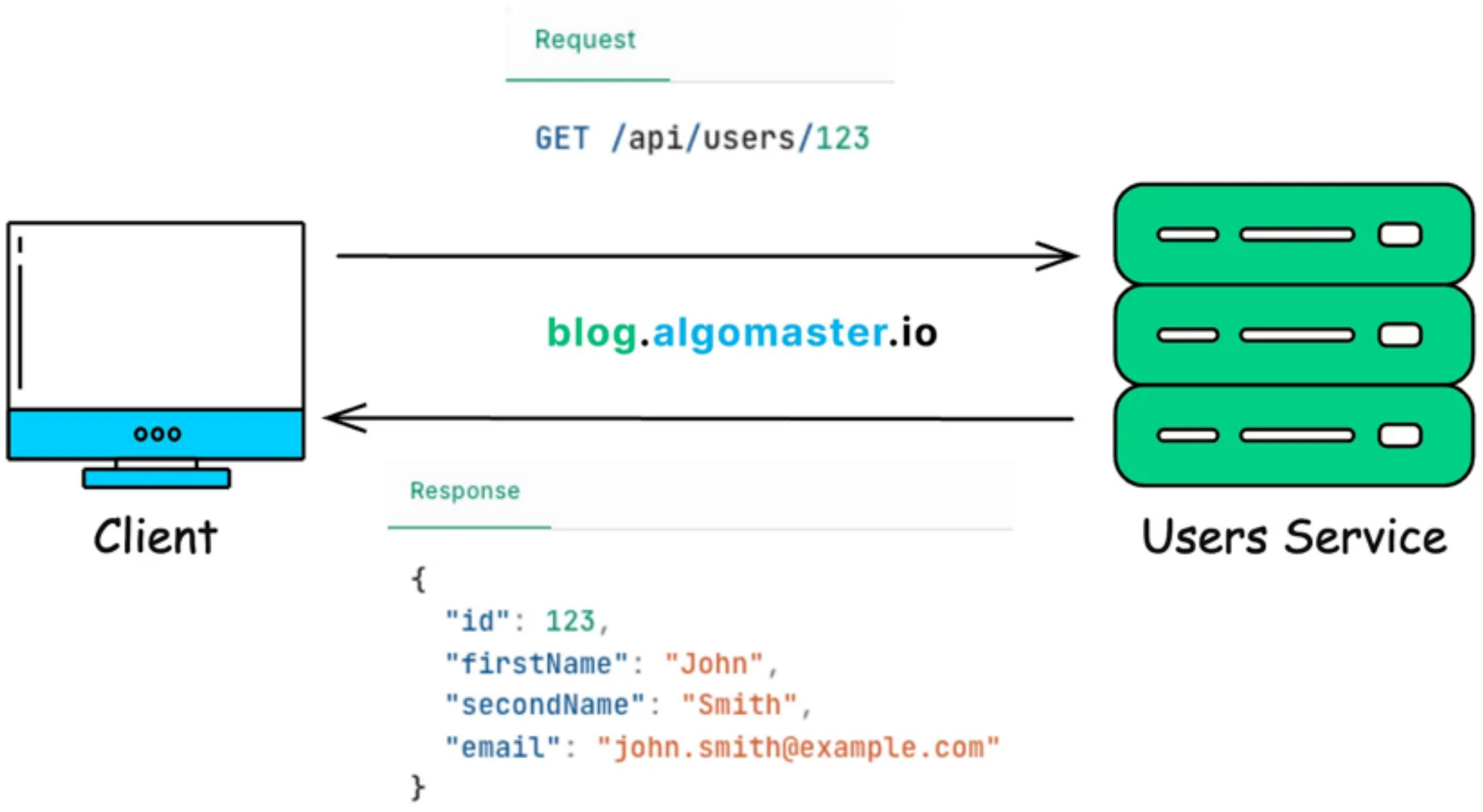
REST emerged in the early 2000s as a set of architectural principles for designing networked applications.

REST is not a protocol or standard but rather a **set of guiding principles** that leverage the existing **HTTP protocol** to enable communication between clients and servers.

At its core, REST is built around **resources**. Each resource (such as a user, order, or product) is uniquely identified by a **URL** (Uniform Resource Locator), and clients interact with these resources using a **fixed set of HTTP methods**.

- **GET** → Retrieve a resource (e.g., `GET /api/users/123` to fetch user data).
- **POST** → Create a new resource (e.g., `POST /api/users` to add a new user).
- **PUT/PATCH** → Update an existing resource (e.g., `PUT /api/users/123` to update user details).
- **DELETE** → Remove a resource (e.g., `DELETE /api/users/123` to delete a user).

For example, let's say a client needs information about a specific user with **ID 123**.



```
GET /api/users/123
```

```
{
  "id": 123,
  "firstName": "John",
  "secondName": "Smith",
  "email": "john.smith@example.com"
}
```

- The client makes a request
- The server responds with a JSON representation of the user

REST APIs typically **return data in JSON** and use **HTTP status codes** to communicate the outcome of the request:

- **200 OK** → Success
- **201 Created** → Resource successfully created
- **400 Bad Request** → Client error (e.g., missing required fields)
- **404 Not Found** → Requested resource does not exist
- **500 Internal Server Error** → Unexpected server issue

Benefits of REST

- **Simplicity and Intuitive Design**: The resource-based model aligns well with most business domains, making REST intuitive for developers.
- **Statelessness**: Each request contains all the information needed to complete it, making REST scalable across distributed systems.
- **Cacheability**: HTTP's caching mechanisms can be leveraged to improve performance.
- **Scalability:** REST APIs can be easily scaled using load balancers and CDNs.
- **Mature Ecosystem**: With nearly two decades of widespread use, REST enjoys robust tooling, documentation, and developer familiarity.

Drawbacks of REST

- **Over-fetching:** REST endpoints often return **more data than needed**, leading to inefficient network usage. For example, if a mobile app only needs a user's name and email, but the API response includes additional fields like address, phone number, and metadata, it results in **wasted bandwidth**.
- **Under-fetching**: If an API doesn't return related data, the client may need to **make multiple requests** to retrieve all required information. For example, to get user details and their posts, a client might have to make:
    1. `GET /api/users/123` (fetch user)
    2. `GET /api/users/123/posts` (fetch user's posts)
- **Versioning issues**: When APIs evolve, maintaining backward compatibility becomes difficult. REST APIs often require **versioned URLs** (`/v1/users`, `/v2/users`), adding maintenance overhead.
- **Rigid Response Structure:** The server defines how data is returned, and clients must adapt to it—even if they only need a subset of the data.

---

2. What is GraphQL?

For years, **REST** was the de facto standard for building APIs. However, as applications grew more complex, REST began to show limitations—especially in scenarios where clients needed fine-grained control over the data they fetched.

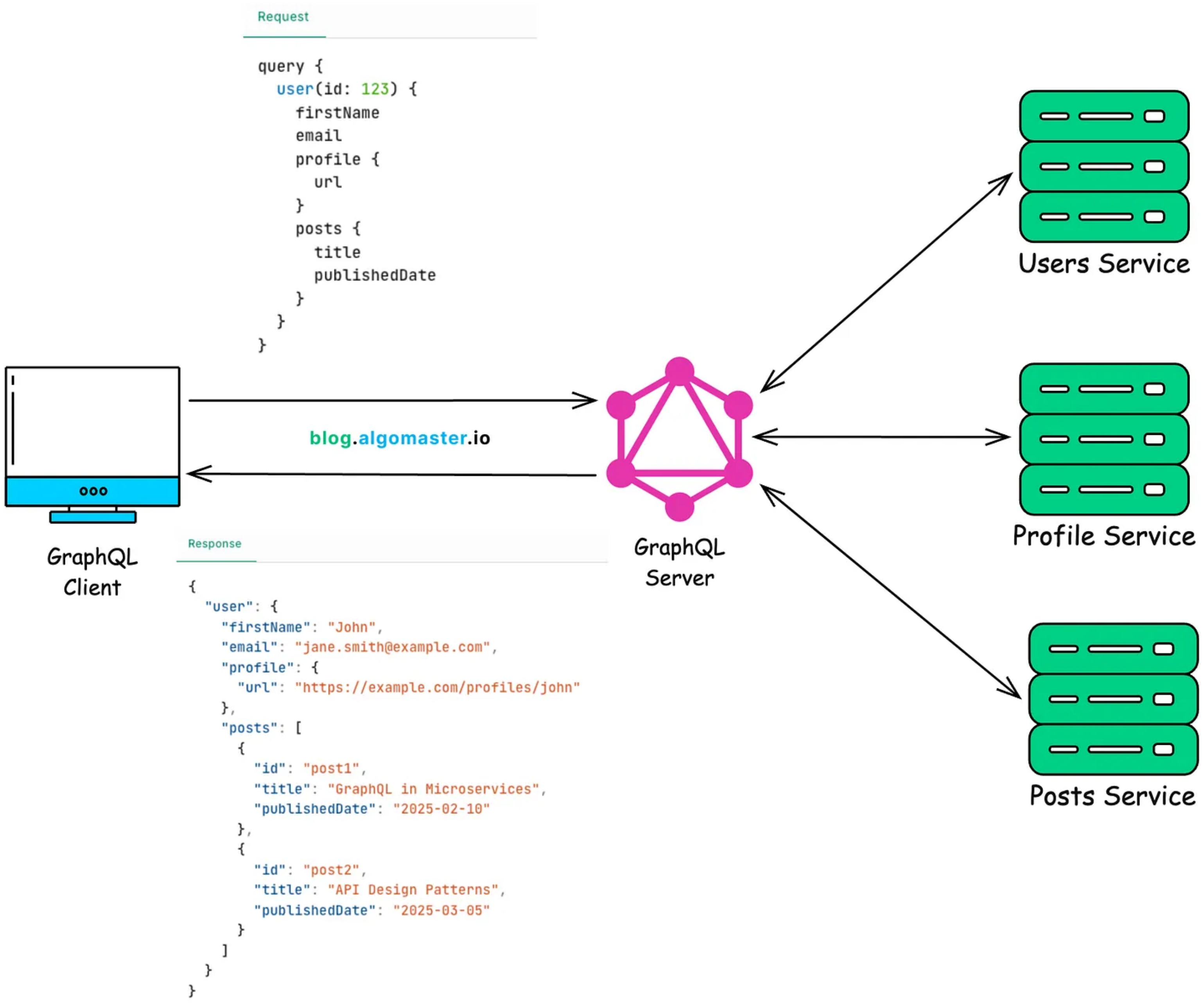
To address these challenges, **Facebook introduced GraphQL in 2015**, offering a more flexible and efficient approach to data retrieval.

How GraphQL Works

Unlike REST, which organizes APIs around **fixed endpoints and HTTP methods**, GraphQL is a **query language** that allows clients to request exactly the data they need—nothing more, nothing less.

> A **single GraphQL endpoint** (`/graphql`) replaces multiple REST endpoints, allowing clients to structure their own queries instead of relying on predefined responses.

Example:



```
query {
  user(id: 123) {
    firstName
    email
    profile {
      url
    }
    posts {
      title
      publishedDate
    }
  }
}
```

```
{
  "user": {
    "firstName": "John",
    "email": "jane.smith@example.com",
    "profile": {
      "url": "https://example.com/profiles/john"
    },
    "posts": [
      {
        "id": "post1",
        "title": "GraphQL in Microservices",
        "publishedDate": "2025-02-10"
      },
      {
        "id": "post2",
        "title": "API Design Patterns",
        "publishedDate": "2025-03-05"
      }
    ]
  }
}
```

Here, the query asks for a **specific user's firstName, email, profileUrl and posts**, all within a **single request.**

GraphQL aggregates the data from multiple services and returns precisely the requested data.

It solves the problems of **over-fetching** (getting unnecessary data) and **under-fetching** (requiring multiple requests to retrieve related data).

Unlike REST, where API responses are **loosely structured** and may vary across versions, **GraphQL enforces a strict schema** that defines the shape of the data.

A simple GraphQL schema for the above example might look like this:

```
type User {
  id: ID!
  firstName: String!
  lastName: String!
  email: String!
  profile: Profile!
  posts: [Post!]
}

type Profile {
  id: ID!
  url: String!
}

type Post {
  id: ID!
  title: String!
  publishedDate: String!
  content: String!
  author: User!
}

type Query {
  user(id: ID!): User
  posts: [Post!]!
}
```

Three Core Functionalities of GraphQL

GraphQL provides three core functionalities:

1. Queries → Fetch Data

Similar to GET requests in REST, GraphQL queries allow clients to request specific fields of data.

Clients have full control over what they retrieve, avoiding unnecessary data fetching.

Example: Fetching specific user and post details in a single request

```
query {
  user(id: 123) {
    name
    email
    posts {
      title
      content
    }
  }
}
```

2. Mutations → Modify Data

Equivalent to **POST, PUT, PATCH, or DELETE** in REST. Used to **create, update, or delete** resources in the API.

Example: Creating a new post

```
mutation {
  createPost(title: "GraphQL vs REST", content: "GraphQL solves many of
REST's limitations...", publishedDate: "2025-03-10") {
    id
    title
    content
  }
}
```

The response will contain the newly created post with its **ID, title, and content**.

3. Subscriptions → Real-Time Updates

Unlike REST, which requires polling or WebSockets for real-time updates, GraphQL subscriptions enable clients to listen for changes and receive updates automatically when data is modified.

Ideal for chat applications, live feeds, stock market updates, and notifications.

Example: Listening for new posts

```
subscription {
  newPost {
    title
    content
```

```
      author {
        name
      }
    }
  }
```

Whenever a **new post is created**, all subscribed clients will **receive instant updates**.

How GraphQL Differs from REST

Both GraphQL and REST rely on **HTTP requests and responses**, but they differ in how they structure and deliver data.

- REST centers around resources (each identified by a URL).
- GraphQL centers around a schema that defines the types of data available.

In REST, the **API implementer** decides which data is included in a response. If a client requests a blog post, the API might also return related **author details**, even if they aren't needed.

With GraphQL, the **client decides** what to fetch. This makes GraphQL more flexible but also introduces challenges in **caching and performance optimization**.

Benefits of GraphQL

1. **Precise Data Fetching**: Clients can request only the fields they need, reducing over-fetching and under-fetching.
2. **Single Request for Multiple Resources**: Related data can be retrieved in one request, solving REST's `n+1` query problem.
3. **Strong Typing**: GraphQL APIs use a schema to define available data, making them easier to explore and document.
4. **Real-time Data with Subscriptions:** GraphQL natively supports real-time data updates through subscriptions, enabling clients to receive automatic notifications whenever data changes on the server.
5. **API Evolution Without Versioning**: New fields can be added without breaking existing queries, avoiding REST-style `/v1`, `/v2` versioning issues.

Drawbacks of GraphQL

1. **Complex Setup & Tooling**: Unlike REST, which can be used with basic HTTP clients (cURL, browsers), GraphQL requires a GraphQL server, schema, and resolvers.
2. **Caching challenges**: REST APIs leverage HTTP caching (e.g., browser caching,

CDNs), but GraphQL queries use POST requests, making caching trickier.

3. **Increased Server Load:** Since clients can request arbitrary amounts of data, GraphQL APIs must be carefully optimized to prevent performance issues.
4. **Security Risks:** Unoptimized queries (e.g., deeply nested requests) can lead to costly database scans, increasing the risk of denial-of-service (DoS) attacks.

Performance Risks with GraphQL

Imagine a mobile app introduces a **new feature** that unexpectedly triggers a **full table scan** on a critical database table.

With REST, this scenario is less likely because API endpoints are predefined, and developers control how data is exposed.

With GraphQL, the client **constructs the query**, which could inadvertently request massive amounts of data. If a poorly designed query is executed on a high-traffic service, it could **bring down the entire database**.

To mitigate this, GraphQL APIs require **strict query rate limiting, depth restrictions, and cost analysis mechanisms**—adding additional complexity to the implementation.

3. Which One Should You Pick?

There is no **one-size-fits-all** answer. **REST** remains a great choice for simple APIs, while **GraphQL** is powerful for complex applications with varying data needs.

Ultimately, it's not about which is better, but which is better for your specific needs.

Here's a quick guide:

Use **REST** if:

- Your API is simple and doesn't require flexible queries.
- You need caching benefits from HTTP.
- You need a standardized, well-established API approach.
- You're integrating with third-party services.
- Your team is already familiar with REST and need faster implementation.

Use **GraphQL** if:

- You need flexible and efficient data fetching.
- Your API serves multiple clients (mobile, web, IoT) with different data needs.

- Real-time updates are required (GraphQL subscriptions).
- You want to avoid API versioning issues.
- Your application requires deeply nested data

Can You Use Both REST and GraphQL?

Absolutely! REST and GraphQL are **not mutually exclusive**, and many organizations implement a **hybrid approach** to get the best of both worlds:

- GraphQL for client-facing applications where flexibility, performance, and dynamic querying are essential.
- REST for admin interfaces, third-party integrations, and internal microservices where statelessness, caching, and simplicity are beneficial.

Thank you for reading!

If you found it valuable, hit a like ❤️ and consider subscribing for more such content every week.

P.S. If you're enjoying this newsletter and want to get even more value, consider becoming a paid subscriber.

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all premium courses on algomaster.io.

There are group discounts, gift options, and referral bonuses available.

Checkout my Youtube channel for more in-depth content.

Follow me on LinkedIn and X to stay updated.

Checkout my GitHub repositories for free interview preparation resources.

I hope you have a lovely day!

See you soon,

Ashish

178 Likes · 7 Restacks

← Previous

Next →

Discussion about this post

Comments Restacks

Write a comment...

r gupta 11 Mar

Liked by Ashish Pratap Singh

Thanks! for the great article, Ashish

On the client side, do we always need a GraphQL client responsible for building the GraphQL queries that are being sent to the server?

As GraphQL queries have a standard format, I guess GraphQL client should always be used as a component on the client side.

LIKE (4) REPLY SHARE

1 reply by Ashish Pratap Singh

1 more comment...