GraphQL

Query Language for APIs.

Developed by Facebook in 2012

Unlike REST, where the server defines what you get, in GraphQL the client decides what data it wants.

```
{
    "name": "Aniket",
    "email": "ani@example.com",
    "address" : "example address"
}
```

Think of REST as a fixed thali in a restaurant 🍱 (you always get all items).
GraphQL → you order only what you want.

Unlike REST (multiple endpoints), GraphQL exposes a single endpoint (/graphql) where the client specifies exactly what data it needs.

Why GraphQL Was Created (Problems with REST)

Over-fetching

REST may return too much data.

Example: GET /users/123 → returns full profile, but mobile app only needed name and photo.

Under-fetching

Sometimes one REST call isn't enough.

Example: To show user's profile + recent posts → need GET /users/123 + GET /users/123/posts.

Multiple round trips → slow.

Versioning Mess

REST often needs /v1/users, /v2/users as APIs evolve.

GraphQL avoids versioning → flexible queries.

GraphQL has 3 core operations:

Query → Read data (like GET in REST).

Mutation → Write/Update/Delete data (like POST/PUT/DELETE in REST).

Subscription → Real-time updates (like WebSockets).

Real-World Example: E-commerce App (Amazon/Flipkart)

With REST:
/user/123 → name, email, etc.
/user/123/orders → list of orders
/product/{id} → product details for each order

Multiple round-trips → slow, inefficient.

Query (Read Data)

Query

```
{
  user(id: "123") {
    name
    email
    orders {
      id
      total
      items {
        product {
          name
          price
        }
        quantity
      }
    }
  }
}
```

```
{
  "data": {
    "user": {
      "name": "Aniket",
      "email": "Aniket@example.com",
      "orders": [
        {
          "id": "567",
          "total": 4599,
          "items": [
            {
              "product": { "name": "Laptop", "price": 4599 },
              "quantity": 1
            }
          ]
        }
      ]
    }
  }
}
```

Response

Only data needed → clean, efficient.

Mutation Example (Place Order):

Mutation

```
mutation {
  createOrder(userId: "123", items: [{productId: "X1", quantity: 2}]) {
    id
    total
    status
  }
}
```

Response

```
{
  "data": {
    "createOrder": {
      "id": "999",
      "total": 4200,
      "status": "CONFIRMED"
    }
  }
}
```

Subscription Example (Real-time Tracking like Swiggy):

```
subscription {
  orderStatus(orderId: "999") {
    status
    estimatedDeliveryTime
  }
}
```

Whenever order status updates →
server pushes new data automatically (no polling).

```
type User {
  id: ID!
  name: String!
  email: String!
  orders: [Order]
}

type Order {
  id: ID!
  total: Int!
  items: [OrderItem]
}

type OrderItem {
  product: Product
  quantity: Int!
}

type Product {
  id: ID!
  name: String!
  price: Int!
}
```

Schema & Types

GraphQL uses Schema Definition Language (SDL).

Strongly typed: every query must match schema.

Client asks → server checks against schema → returns valid data.

Benefits of GraphQL

Client controls data → No over/under fetching.
One endpoint → Simpler networking.
Strong typing → Safer APIs, better tooling.
Versioning not needed → Just deprecate fields.
Great for mobile apps → Less data = faster.
Introspection → Clients can ask API "what fields exist?"

Challenges of GraphQL

Complexity on server side → Need resolvers for each field.
Caching harder → REST can cache by URL (/users/123). GraphQL queries vary.
Performance risk → Clients can ask for huge nested data.
Learning curve for dev teams.

When to Use GraphQL

Complex systems with lots of relationships (social networks, e-commerce, dashboards).
When clients (mobile/web) need different shapes of data.
For real-time apps (chat, live tracking).

Stick to REST if:

Simple CRUD APIs.

Heavy caching needed (CDNs work best with REST).

Small team with simple requirements.

DEMO