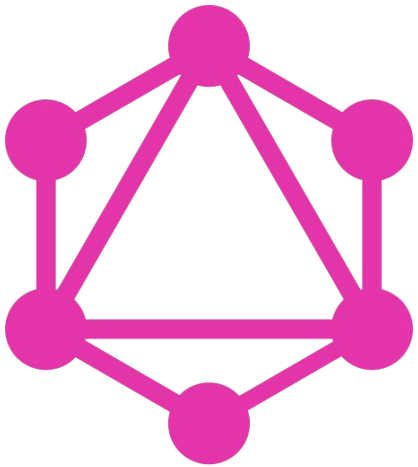
GraphQL

A Gentle Introduction



Saverio Mattia **Merenda**
Pasquale **Castelluccia**

08/05/2025

Lesson Agenda

- **The Limits of Traditional REST APIs** (The Problem)
- **What is GraphQL?** (Introduction & Origins)
- **How GraphQL addresses these Limits** (The Solution)
- **The Heart of GraphQL:** The Schema Definition Language
- **Interacting with Data:** Queries & Mutations
- **Direct Comparison:** GraphQL vs REST (Practical Scenario)

Social Media Data Retrieval Use Case

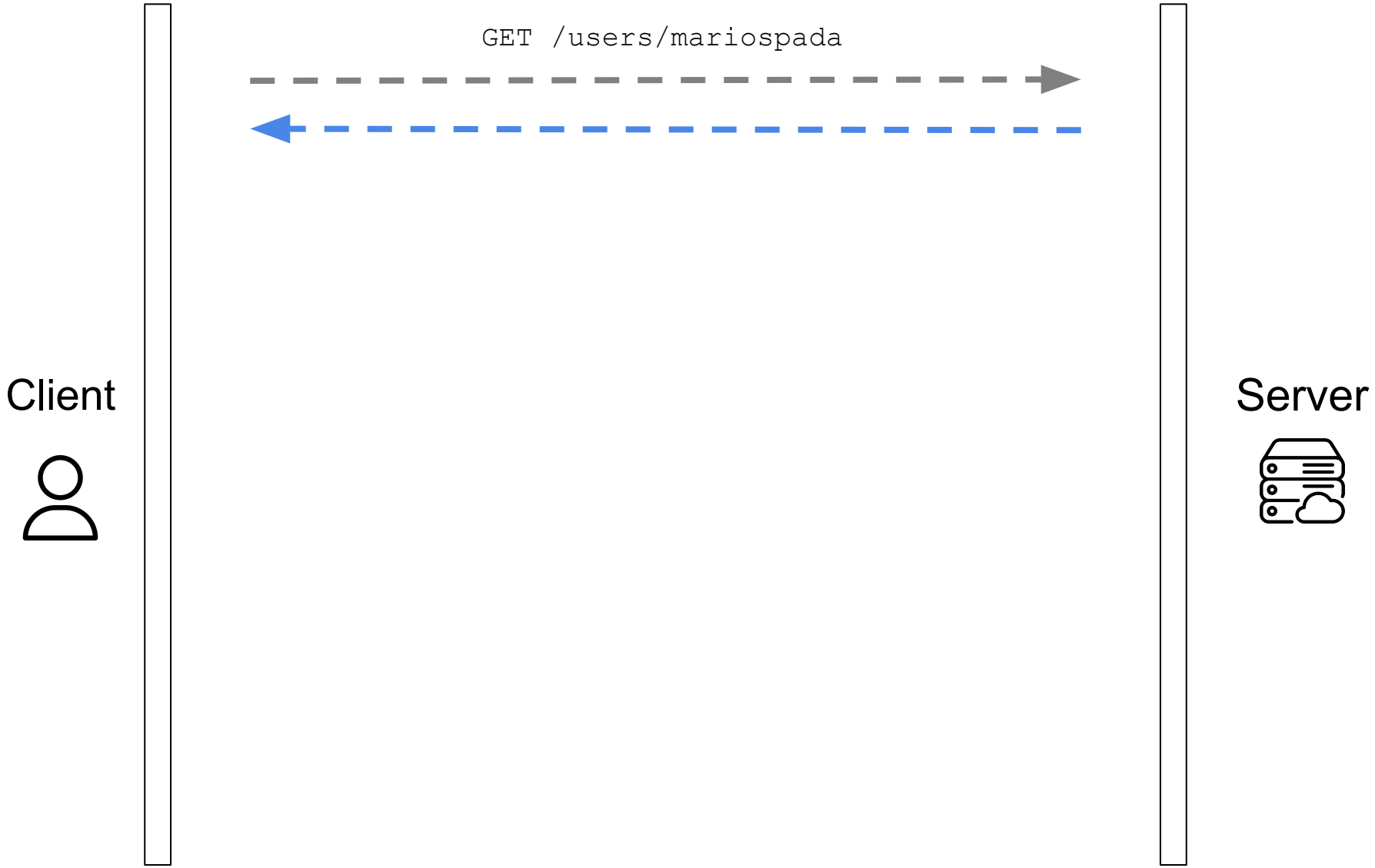
- **Goal:** fetch a user's profile within its recent activity in one cohesive flow.
- **Data Requirements**:
  - User: `id`, `username`, `profilePictureUrl`
  - Posts (most recent 3):
    - for each post `id`, `text`, `timestamp`
  - Comments (most recent 2 per post):
    - for each comment `id`, `text`, `commenterUsername`

What is REST?

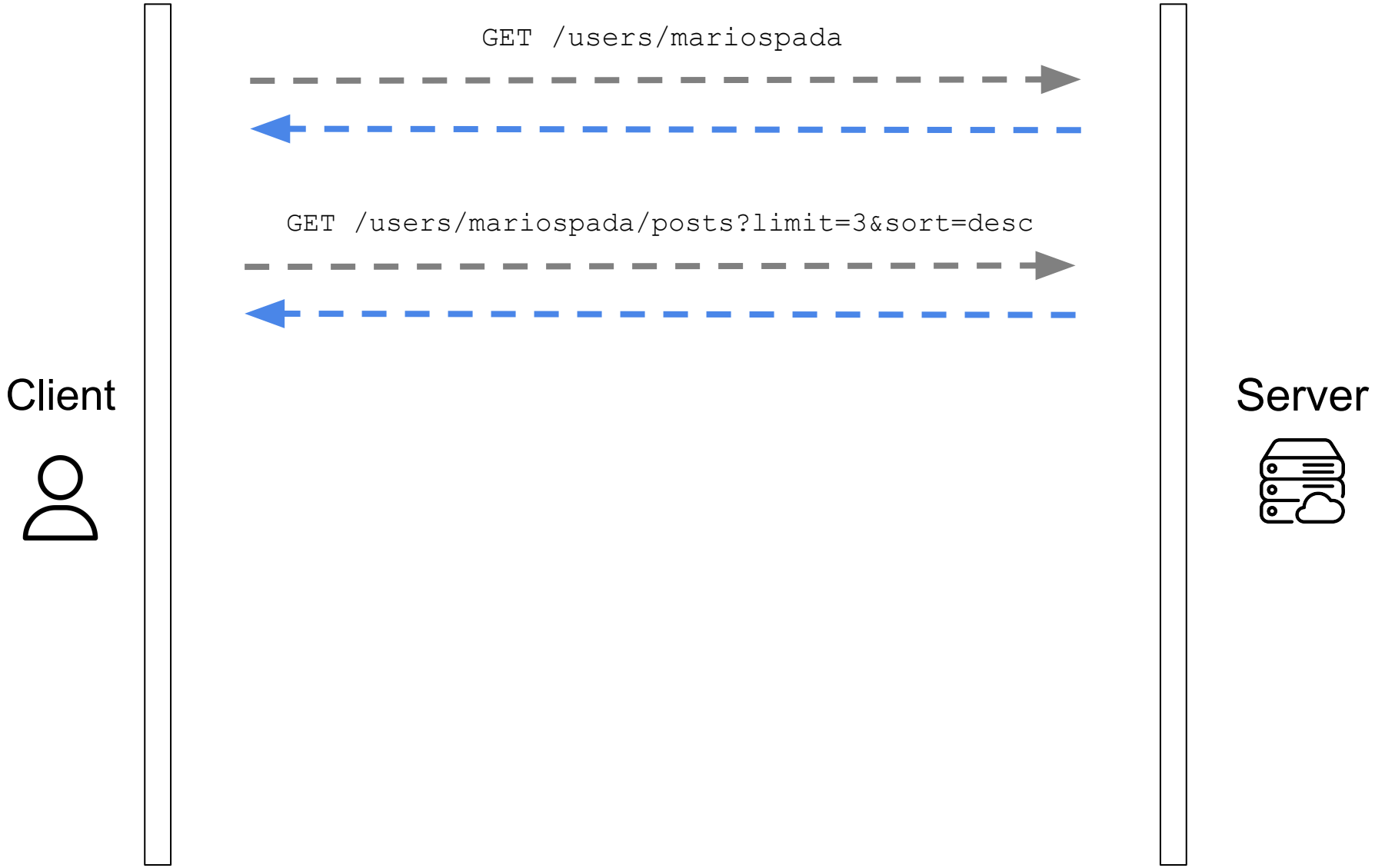
- **REST (REpresentational State Transfer):** an architectural style for designing networked applications, widely used for web APIs.
- **Resource-Oriented:** focuses on *resources* (like users, posts, comments) identified by unique URLs (e.g., `/users/123`).
- Uses standard HTTP verbs for actions, e.g., GET to retrieve a resource; POST to create a new resource; PUT to update an existing resource.
- **Stateless**: each request from client to server must contain all necessary information; the server doesn't store client session state between requests.
- **Client-Server Separation:** client (e.g., app) and server (API) are independent and evolve separately.

REST Approach

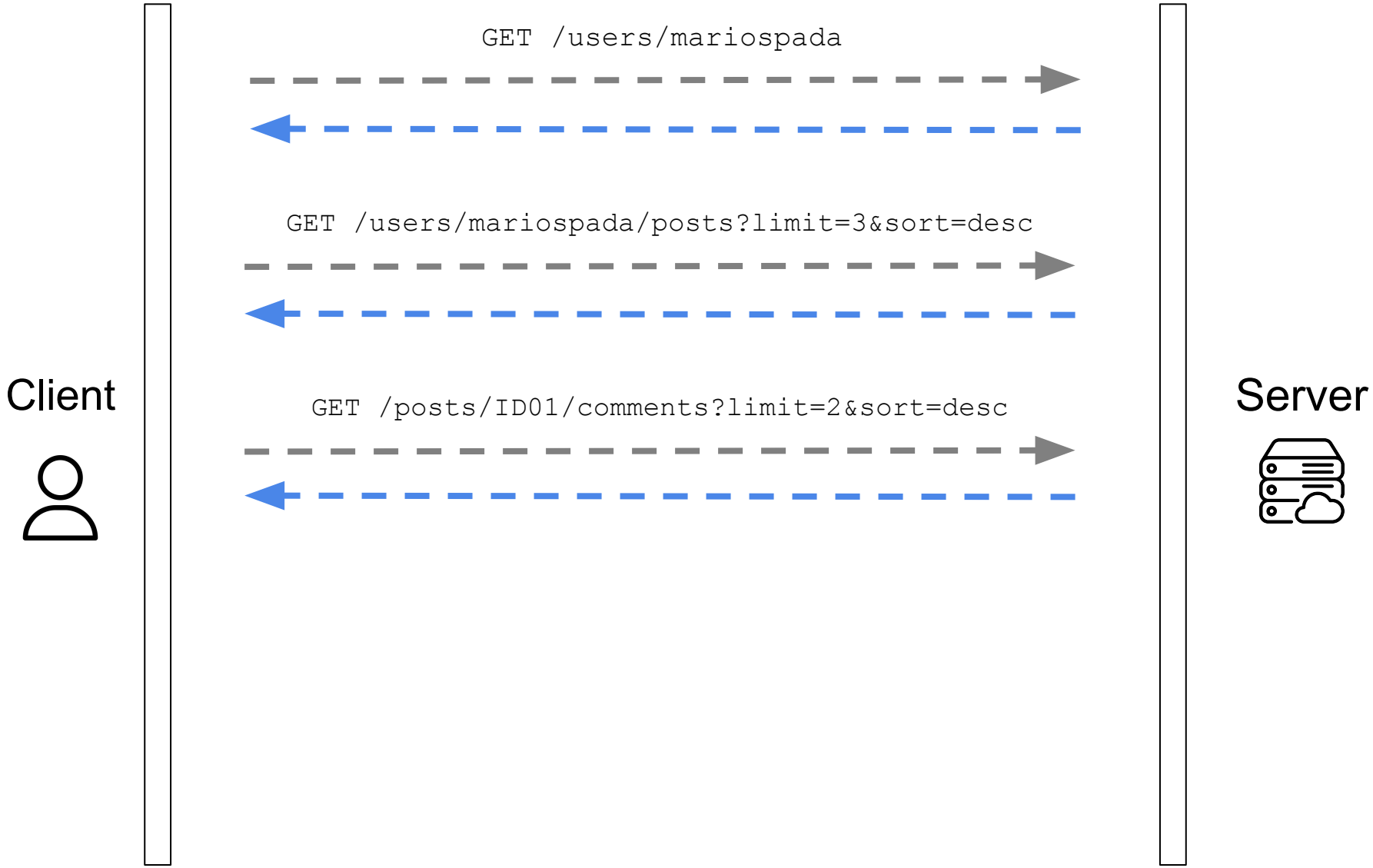
REST Approach



REST Approach



REST Approach



REST Approach

REST Approach

REST Approach

- `GET /users/{userId}`
  - Returns full user resource (often includes extra fields causing over-fetching).
- `GET /users/{userId}/posts?limit=3&sort=desc`
  - Retrieves last 3 posts (may include likes, shares causing unused data).
- For each post (e.g., post1, post2, post3):
  - `GET /posts/{postId}/comments?limit=2&sort=desc`
  - Three separate calls to fetch comments for each post (under-fetching without nested support).

The Problem (Why)

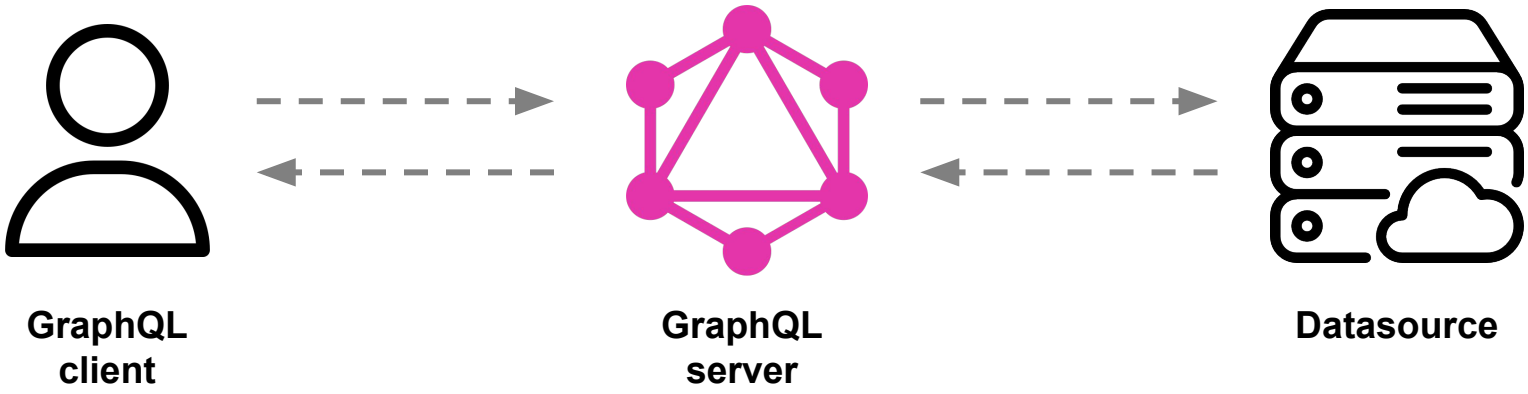
- **Inefficient Data Fetching**
  - Complex features (e.g., News Feed) required data from multiple sources with nested relationships.
  - REST often meant numerous round-trips to different endpoints, slowing load times and consuming precious mobile bandwidth.
- **Over-fetching & Under-fetching**
  - Over-fetching: clients received extra, unused data—wasting bandwidth and client-side processing.
  - Under-fetching: single endpoints didn’t supply all required fields, forcing additional API calls.

The Problem (Why)

- **Tight Frontend–Backend Coupling**
  - UI changes on the client required corresponding backend endpoint updates.
  - Slowed development cycles and hindered rapid iteration.
- **Compromised Mobile UX**
  - High network usage and latency degraded performance on native mobile apps.
  - Inconsistent experience on unstable cellular connections.

GraphQL: not just another API

- GraphQL is a **query language** for APIs and a **server-side runtime**.
- A **powerful** alternative to REST, born from specific needs for **efficiency** and **flexibility**.
- **Focus:** allowing clients to request exactly the data they need, nothing more, nothing less.



Who, When, Where

- Developed internally by **Facebook** (now Meta).
  - Key engineers: Nick Schrock, Lee Byron, Dan Schafer.
- **2012:** internal development begins to rethink API architecture for the shift from HTML5 mobile apps to fully native clients.
- **2015:** public release of the specification draft and reference implementation as an open-source project.
- **2018:** GraphQL project transferred to the newly formed GraphQL Foundation under the Linux Foundation.

The Solution (What)

- **Declarative Queries:** client specifies exactly which fields and nested relationships it needs.
- **Single Endpoint:** all requests go through one unified API endpoint, simplifying your network layer.
- **Predictable Responses:** server returns JSON matching the structure of the client's query.
- **Strongly-Typed Schema:** a type system defines the API contract, enabling powerful tooling and validation.
- **Data Aggregation:** seamlessly combine data from databases, microservices, and legacy REST APIs behind one GraphQL interface.

Schema Definition Language (SDL)

- **Core Role of the Schema**
  - Serves as a rigorous contract between client and server.
  - Defines every API capability unambiguously.
- **Strong Typing**
  - Every object, field, and argument has a specific type.
  - Validates client requests before execution and ensures predictable response structure.
- **Clear Contract**
  - Enumerates all data types, fields, and supported operations (queries, mutations, subscriptions).
  - Aligns frontend and backend teams around a single authoritative API definition.

SDL: Object Types

- Built-in primitives: `Int`, `Float`, `String`, `Boolean`, `ID`.
- Use the type keyword followed by a PascalCase name.
- Enclose field definitions in `{}`.
- Mirrors the shape of your domain objects.

```
# Defining a Book type
type Book {
  # fields go here
}
```

SDL: Field Definitions

- Inside an object type, list fields in *camelCase.*
- Each field has the form `name: Type`.
- No commas between fields.

```
type Book {
  title: String
  pageCount: Int
}
```

SDL: Non-Null Modifier

- Append ! to any type to mark it *non-nullable.*
- The server guarantees a value or returns an error.

```
type Book {
  title: String!    # title is required
  pageCount: Int
}
```

SDL: List Modifier

- Wrap a type in `[]` to indicate an array of that type.
- Combine with `!` to enforce non-null lists or non-null elements.

```
type Author {
  name: String!
}

type Book {
  title: String!
  pageCount: Int
  # list itself and each Author are non-null
  authors: [Author!]!
}
```

Interacting with Data

- **Query**
  - Read-only operation.
  - Analogous to `GET` in REST.
  - Ideal for fetching data without side effects.
- **Mutation**
  - Write operation (create/update/delete).
  - Analogous to `POST/PUT/PATCH/DELETE` in REST.
  - Used whenever you need to change server-side state.

Query Syntax

- Fetching exactly **what you need**.
- Mirrors the shape of the desired JSON response.
- Request only the fields and nesting you require.

```
query GetUserAndPosts {
  user(id: "123") {      # Root field with an argument
    username             # Scalar field
    posts {              # Nested list of objects
      text               # Scalar field inside list
    }
  }
}
```

Query Syntax


```
{
  "data": {
    "user": {
      "username": "mariospada",
      "posts": [
        {
          "text": "Excited to share my first GraphQL query!"
        },
        {
          "text": "Just had the best coffee this morning."
        },
        {
          "text": "Looking forward to the weekend plans."
        }
      ]
    }
  }
}
```

Mutation Syntax

- Calls a mutation field defined in the schema (e.g., `createPost`).
- Uses `Input` types for complex arguments.
- Specifies exactly which fields to return after execution.

```
mutation CreateNewPost {
  createPost(input: { text: "Hello GraphQL!", visibility: PUBLIC }) {
    id     # Return the new post's ID
    text   # Return the new post's text
  }
}
```

Execution Order

- **Queries**
  - Fields resolve in parallel (where possible).
  - Maximizes efficiency for data fetching.
- **Mutations**
  - Root fields execute serially, in request order.
  - Ensures predictable, ordered state changes.
  - Note: does not imply automatic transactional guarantees across fields.

Back to Social Media Scenario

- **Goal:** fetch a user's profile within its recent activity in one cohesive flow.
- **Data Requirements**:
  - User: `id`, `username`, `profilePictureUrl`
  - Posts (most recent 3):
    - for each post `id`, `text`, `timestamp`
  - Comments (most recent 2 per post):
    - for each comment `id`, `text`, `commenterUsername`

REST Approach

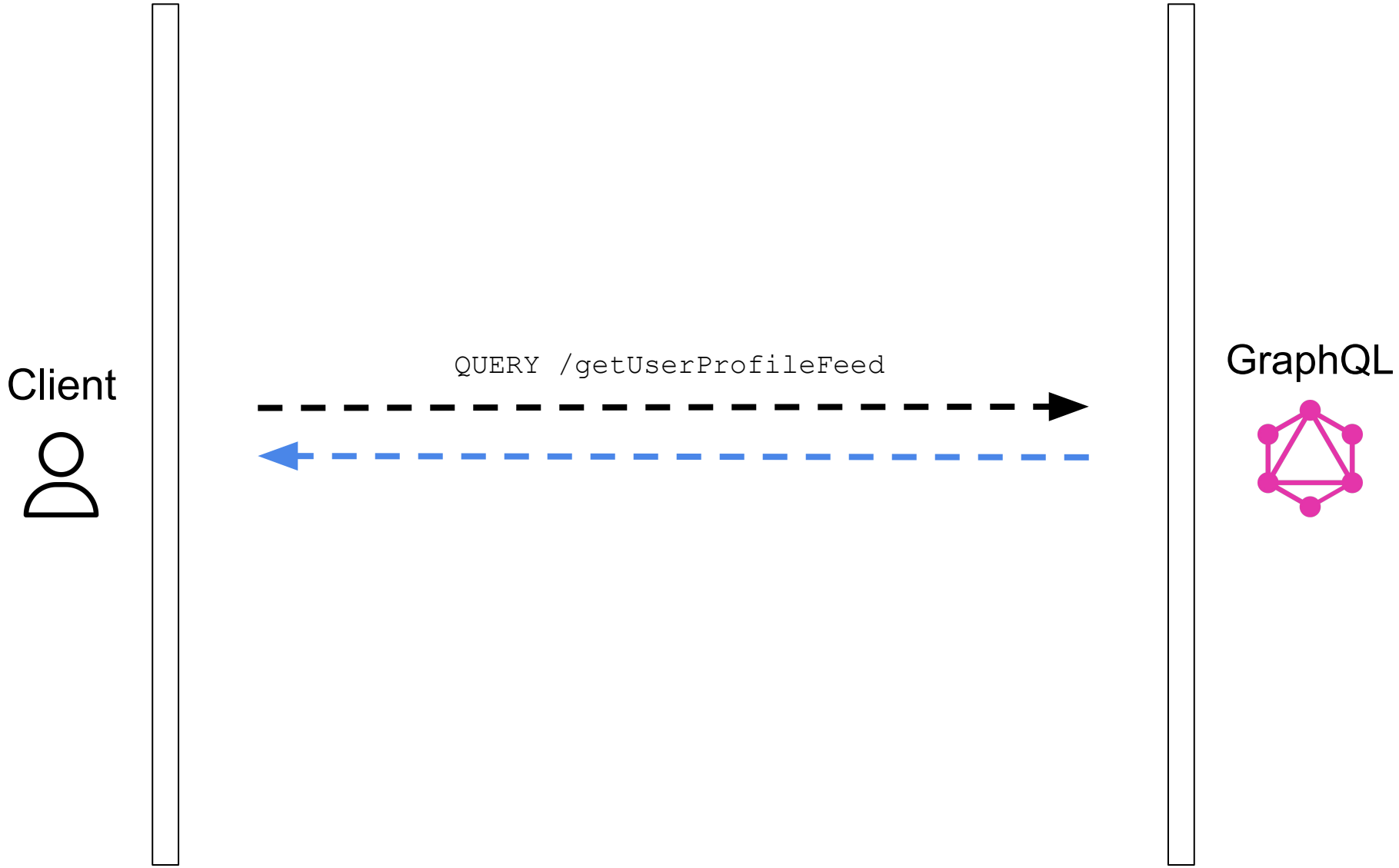
GraphQL Approach

```
query GetUserProfileFeed {
  user(id: "{userId}") {      # Specify the user
    id
    username
    profilePictureUrl
    posts(last: 3) {          # Request the last 3 posts
      id
      text
      timestamp
      comments(last: 2) {     # Request the last 2 comments per post
        id
        text
        commenterUsername
      }
    }
  }
}
```

GraphQL Approach


```
{
  "data": {
    "user": {
      "id": "123",
      "username": "mariospada",
      "profilePictureUrl": "https://example.com/avatars/mariospada.png",
      "posts": [
        {
          "id": "post01",
          "text": "Just tried GraphQL for the first time – awesome!",
          "timestamp": "2025-05-07T14:23:00Z",
          "comments": [
            {
              "id": "comment456",
              "text": "Welcome to the GraphQL club!",
              "commenterUsername": "graphqlFan"
            },
            {
              "id": "comment457",
              "text": "Glad you're enjoying it!",
              "commenterUsername": "dev_guru"
            }
          ]
        },
        ...
      ]
    }
  }
}
```

GraphQL Approach



GraphQL Approach

- **Minimal Latency:** one network request for all data, i.e., reduced round-trip time.
- **Precise Payloads:** no over-fetching; transfer only requested fields, i.e., smaller, optimized responses.
- **Shifted Complexity:** server resolves complex, nested queries across multiple data sources.
- **Developer Productivity:** frontend teams can evolve data requirements independently, without backend endpoint changes.

REST vs GraphQL

	REST	GraphQL
API calls	Multiple (e.g., GetUser, GetPosts, GetComments x3)	Single
Endpoint	Multiple (e.g., /users/{id}, /users/{id}/posts, /posts/{id}/comments)	Single (e.g., /graphql)
Recovered Data	Fixed structure per endpoint; likely unnecessary fields (Over-fetching)	Exactly the fields specified in the query; no superfluous data
Network Latency	Major due to multiple round trips	Minor due to single round trip
Payload size	Potentially greater due to over-fetching	Minor, optimized for the specific query
Client logic	Must orchestrate multiple calls, filter/join data	Simpler data retrieval logic; response matches query
Flexibility	Low; tied to predefined endpoints	High; client defines data needs per query

Lab time!

References

1. https://hygraph.com/learn/graphql
2. https://blog.mobcoder.com/graphql-vs-rest-api-is-a-comprehensive-comparison-for-modern-development/
3. https://en.wikipedia.org/wiki/GraphQL
4. https://www.expeed.com/mastering-data-fetching-with-graphql-overcome-over-fetching-under-fetching/
5. https://www.mulesoft.com/api-university/graphql-and-how-did-it-evolve-from-rest-api
6. https://medium.com/@amoljadhav_48655/simplifying-api-client-integration-the-shift-from-rest-to-graphql-965fbcb5485d