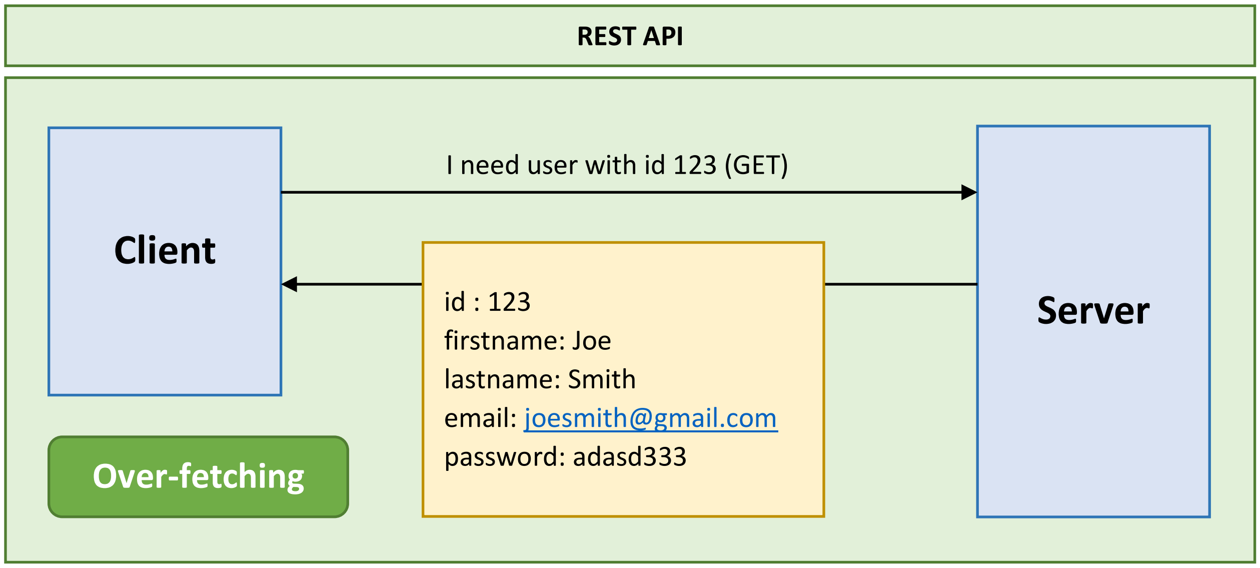
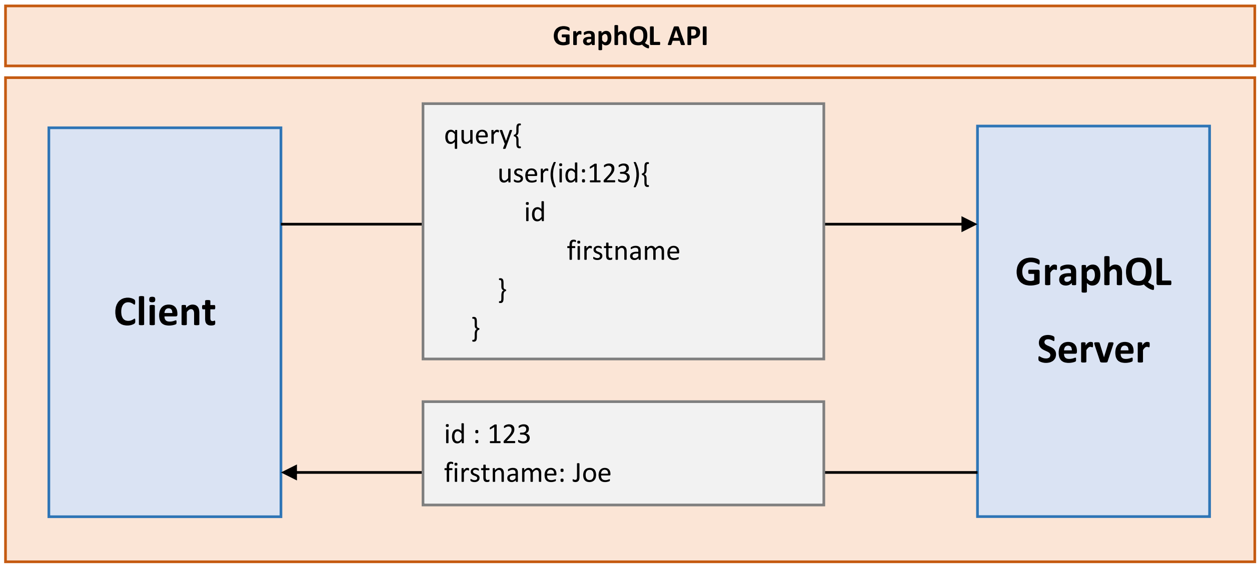
Why GraphQl ?



```
id : 123
firstname: Joe
lastname: Smith
email: joesmith@gmail.com
password: adasd333
```

Suppose I want only id and firstname of user, but server is sending full user details. I don't need all details of user. This problem is called **OVER FETCHING**.



```
query{
      user(id:123){
         id
             firstname
      }
   }
```

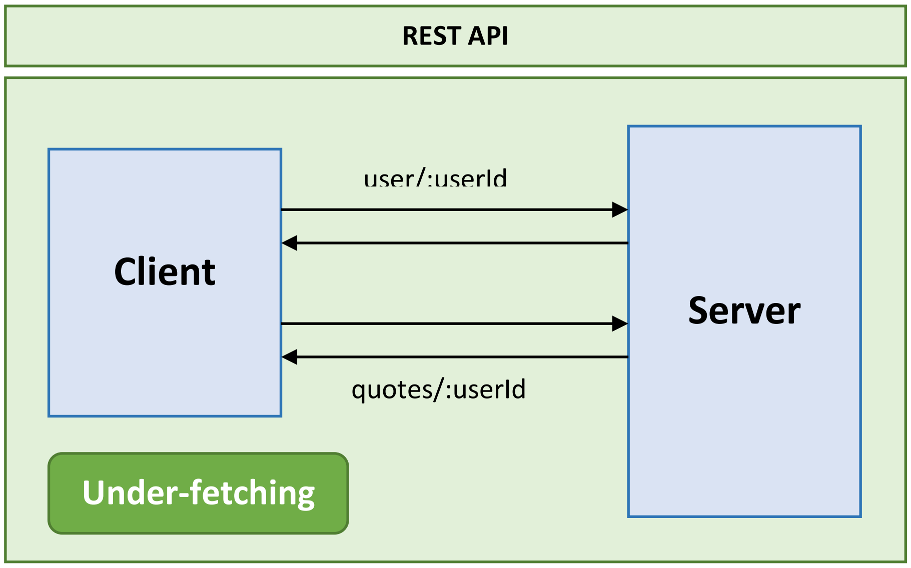
```
id : 123
firstname: Joe
```

User API

firstname: Joe
email:
joesmith@gmail.com

Quotes API

name: 'qoute name'
Desc: '.....'

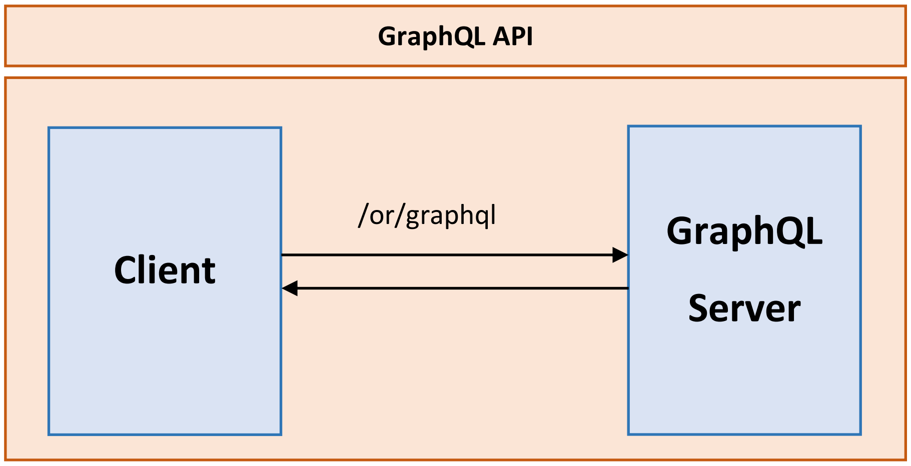


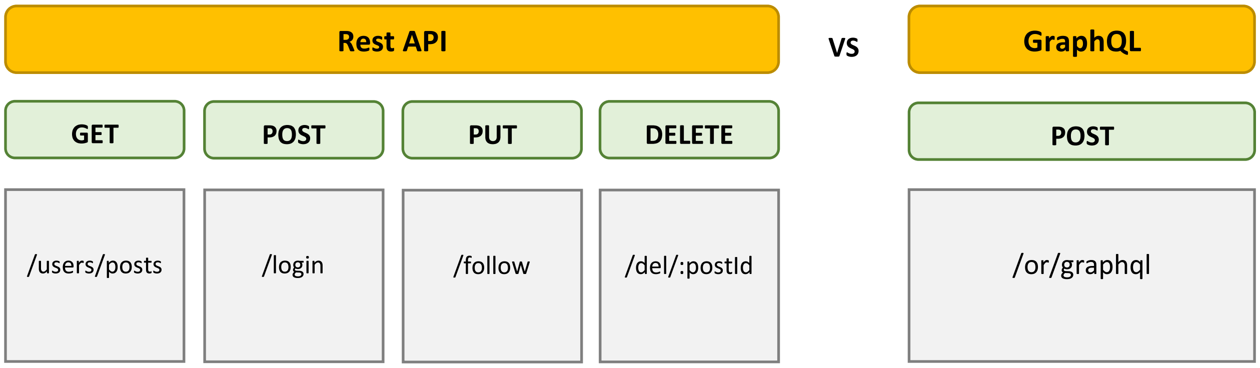
Let' say, we have to show firstname, email & quotes created by user. So here we have to call 2 APIs fo getting data because user details & quotes are two different entities.

1. user/:userId API fetchs firstname & email of user
2. quotes/:userId API fetchs quote info

graphQl API

```
query{
    user(id:123){
        firstname
        email
        quotes{
            name
        }
    }
}
```





In GraphQL, there's only one endpoint, usually an HTTP POST endpoint

Query	**Mutation**	**Resolver**
To get data	To post / update / delete data	Has logic to handle query or mutation operation

There have so many problems like the following:

- Over-fetching
- Multiple requests for multiple resources
- Waterfall network requests on nested data
- Each client need to know the location of each service

One solution can be possible to be create a separate API for fetching only id and firstname. But it will not be accurate solution because then for every small queries you need to create separate APIs. Suppose future you need to how only email, then you have to create another API for showing only email. SO this is not a permanent solution.

GraphQl solves this over fetching & Multiple APIs problems.

GraphQL: a server-side runtime and query language for your API.

GraphQL simplifies the task of aggregating data from multiple sources or APIs and then resolving the data to the client in a single API call. On the other hand, API technologies like REST would require multiple HTTP calls to access data from multiple sources.

Advantages of GraphQl

No Over-Fetching or Under-Fetching

With GraphQL, developers can fetch only what is required. Nothing less, nothing more. This solves the issues that arise due to over-fetching and under-fetching.

Over-fetching happens when the response fetches more than what is required. Consider the example of a blog home page. It displays the list of all blog posts (just the title and URLs). However, to display this list, you need to fetch all the blog posts (along with body data, images, etc.) through the API, and then show only what is required, usually through UI code. This impacts the performance of your app and consumes more data, which is expensive for the user.

With GraphQL, you define the exact fields that you want to fetch (i.e., Title and URL, in this case), and it fetches the data of only these fields.

Under-fetching, on the other hand, is not fetching adequate data in a single API request. In this case, you need to make additional API requests to get related or referenced data. For instance, while displaying an individual blog post, you also need to fetch the referenced author's entry, just so that you can display the author's name and bio.

GraphQL handles this well. It lets you fetch all relevant data in a single query.

Saves Time and Bandwidth

GraphQL allows making multiple resources request in a single query call, which saves a lot of time and bandwidth by reducing the number of network round trips to the server. It also helps to save waterfall network requests, where you need to resolve dependent resources on previous requests. For example, consider a blog's homepage where you need to display multiple widgets, such as recent posts, the most popular posts, categories, and featured posts. With REST architecture, displaying these would take at least five requests, while a similar scenario using GraphQL requires a single GraphQL request.

Schema Stitching for Combining Schemas

Schema stitching allows combining multiple, different schemas into a single schema. In a microservices architecture, where each microservice handles the business logic and data for a specific domain, this is very useful. Each microservice can define its own GraphQL schema, after which you'd use schema stitching to weave them into one that is accessed by the client.

Apollo Client

Apollo Client is a fully-featured caching GraphQL client with integrations for React, Angular, and more. It allows you to easily build UI components that fetch data via GraphQL.