1.GraphQL

1.1) Introduction

- New API standard that was invented and open sourced by Facebook
- Enables declarative data fetching
- Graph QL server exposes single endpoint and response to the queries
- A query language for APIs
- A more efficient alternative to REST
    1. Increased mobile usage creates need for efficient data loading
    2. Variety of different front end frameworks and platforms on client side
    3. Fast development speed expectation for rapid feature development
- Graph QL not only for React ,can be use any programming language and frameworks

The main terms used most commonly in GraphQL are:

- **Schema** — The contract between the GraphQL client and the GraphQL server
- **Query** — Similar to GET call in REST and used by the client to query the fields
- **Mutations** — It is similar to a POST/PUT call in REST and is used by the client for any insert/update operation

1.2) GraphQL vs REST

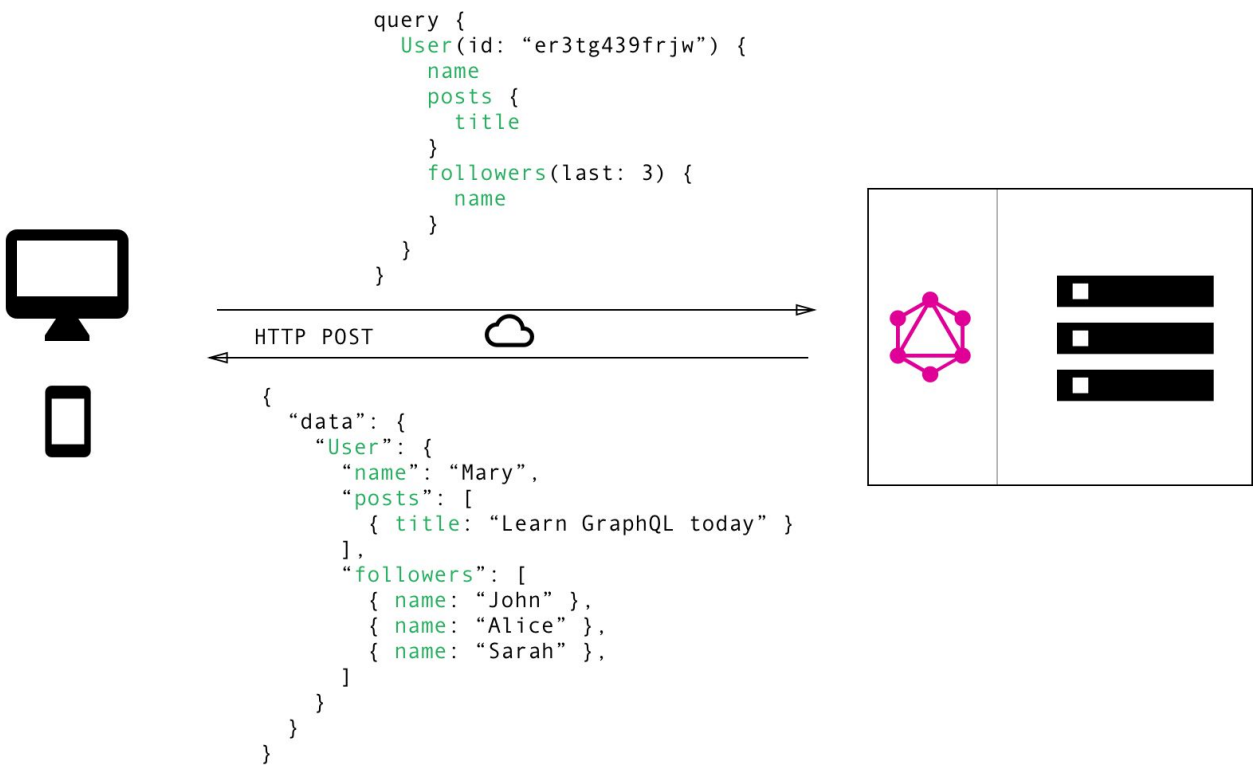
- Great ideas in REST:stateless server and structured access to resource
- Strictly specification - but the concept was widely interpreted
- Rapidly changing requirement with client side don't go well with static nature of REST
- GrpahQL was developed to cope with need to more **flexibility and efficiency** in client-server communication

Data Fetching with REST vs GraphQL

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.

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.

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.



Using GraphQL, the client can specify exactly the data it needs in a query. Notice that the structure of the server's response follows precisely the nested structure defined in the query.

No more Over- and Underfetching

One of the most common problems with REST is that of over- and underfetching. This happens because the only way for a client to download data is by hitting endpoints that return *fixed* data structures. It's very difficult to design the API in a way that it's able to provide clients with their exact data needs.

Overfetching: Downloading superfluous data

Overfetching means that a client downloads more information than is actually required in the app. Imagine for example a screen that needs to display a list of users only with their names. In a REST API, this app would usually hit the /users endpoint and receive a JSON array with user data. This response however might contain more info about the users that are returned, e.g. their birthdays or addresses - information that is useless for the client because it only needs to display the users' names.

Underfetching and the n+1 problem

Another issue is *underfetching* and the *n+1*-requests problem. Underfetching generally means that a specific endpoint doesn't provide enough of the required information. The client will have to make additional requests to fetch everything it needs. This can escalate to a situation where a client needs to first download a list of elements, but then needs to make one additional request per element to fetch the required data.

As an example, consider the same app would also need to display the last three followers per user. The API provides the additional endpoint /users/<user-id>/followers. In order to be able to display the required information, the app will have to make one request to the /users endpoint and then hit the /users/<user-id>/followers endpoint for *each* user.

Rapid Product Iterations on the Frontend

A common pattern with REST APIs is to structure the endpoints according to the views that you have inside your app. This is handy since it allows for the client to get all required information for a particular view by simply accessing the corresponding endpoint.

The major drawback of this approach is that it doesn't allow for rapid iterations on the frontend. With every change that is made to the UI, there is a high risk that now there is more (or less) data required than before. Consequently, the backend needs to be adjusted as well to account for the new data needs. This kills productivity and notably slows down the ability to incorporate user feedback into a product.

With GraphQL, this problem is solved. Thanks to the flexible nature of GraphQL, changes on the client-side can be made without any extra work on the server. Since clients can specify their exact data requirements, no backend engineer needs to make adjustments when the design and data needs on the frontend change.

Insightful Analytics on the Backend

GraphQL allows you to have fine-grained insights about the data that's requested on the backend. As each client specifies exactly what information it's interested in, it is possible to gain a deep understanding of how the available data is being used. This can for example help in evolving an API and deprecating specific fields that are not requested by any clients any more.

With GraphQL, you can also do low-level performance monitoring of the requests that are processed by your server. GraphQL uses the concept of *resolver functions* to collect the data that's requested by a client. Instrumenting and measuring performance of these resolvers provides crucial insights about bottlenecks in your system.

Benefits of a Schema & Type System

GraphQL uses a strong type system to define the capabilities of an API. All the types that are exposed in an API are written down in a *schema* using the GraphQL Schema Definition Language (SDL). This schema serves as the contract between the client and the server to define how a client can access the data.

Once the schema is defined, the teams working on frontend and backends can do their work without further communication since they both are aware of the definite structure of the data that's sent over the network.

Frontend teams can easily test their applications by mocking the required data structures. Once the server is ready, the switch can be flipped for the client apps to load the data from the actual API.

1.3) Core Concepts

The Schema Definition Language (SDL)

GraphQL has its own type system that's used to define the *schema* of an API. The syntax for writing schemas is called Schema Definition Language (SDL).

Here is an example of how we can use the SDL to define a simple type called Person:

```
type Person {
  name: String!
  age: Int!
}
```

This type has two *fields*, they're called name and age and are respectively of type String and Int. The ! following the type means that this field is *required*.

It's also possible to express relationships between types. In the example of a *blogging* application, a Person could be associated with a Post:

```
type Post {
  title: String!
  author: Person!
}
```

Conversely, the other end of the relationship needs to be placed on the Person type:

```
type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}
```

Note that we just created a *one-to-many*-relationship between Person and Post since the posts field on Person is actually an *array* of posts.

Fetching Data with Queries

When working with REST APIs, data is loaded from specific endpoints. Each endpoint has a clearly defined structure of the information that it returns. This means that the data requirements of a client are effectively *encoded* in the URL that it connects to.

The approach that's taken in GraphQL is radically different. Instead of having multiple endpoints that return fixed data structures, GraphQL APIs typically only expose *a single endpoint*. This works because the structure of the data that's returned is not fixed. Instead, it's completely flexible and lets the client decide what data is actually needed.

That means that the client needs to send more *information* to the server to express its data needs - this information is called a *query*.

```
{
  allPersons {
    name
  }
}
```

The allPersons field in this query is called the *root field* of the query. Everything that follows the root field, is called the *payload* of the query. The only field that's specified in this query's payload is name.

This query would return a list of all persons currently stored in the database. Here's an example response:

```
{
  "allPersons": [{ "name": "Johnny" },
```

```
    { "name": "Sarah" },
    { "name": "Alice" }
  ]
}
```

Notice that each person only has the name in the response, but the age is not returned by the server. That's exactly because name was the only field that was specified in the query.

If the client also needed the persons' age, all it has to do is slightly adjust the query and include the new field in the query's payload

One of the major advantages of GraphQL is that it allows for naturally querying *nested* information. For example, if you wanted to load all the posts that a Person has written, you could simply follow the structure of your types to request this information:

```
{
  allPersons {
    name
    age
    posts {
      title
    }
  }
}
```

Queries with Arguments

In GraphQL, each *field* can have zero or more arguments if that's specified in the *schema*. For example, the allPersons field could have a last parameter to only return up to a specific number of persons. Here's what a corresponding query would look like:

```
{
  allPersons(last: 2) {
    name}}
```

Writing Data with Mutations

Next to requesting information from a server, the majority of applications also need some way of making changes to the data that's currently stored in the backend. With GraphQL, these changes are made using so-called *mutations*. There generally are three kinds of mutations:

- creating new data
- updating existing data
- deleting existing data

Mutations follow the same syntactical structure as queries, but they always need to start with the mutation keyword. Here's an example for how we might create a new Person:

```
mutation {
  createPerson(name: "Bob", age: 36) {
    name
    age
  }
}
```

Notice that similar to the query we wrote before, the mutation also has a *root field* - in this case it's called createPerson. We also already learned about the concepts of arguments for fields. In this case, the createPerson field takes two arguments that specify the new person's name and age.

Like with a query, we're also able to specify a payload for a mutation in which we can ask for different properties of the new Person object. In our case, we're asking for the name and the age - though admittedly that's not super helpful in our example since we obviously already know them as we pass them into the mutation. However, being able to also query information when sending mutations can be a very powerful tool that allows you to retrieve new information from the server in a single roundtrip!

The server response for the above mutation would look as follows:

```
"createPerson": {
  "name": "Bob",
  "age": 36,
}
```

One pattern you'll often find is that GraphQL types have unique *IDs* that are generated by the server when new objects are created. Extending our Person type from before, we could add an id like this:

```
type Person {
  id: ID!
  name: String!
  age: Int!
}
```

Now, when a new Person is created, you could directly ask for the id in the payload of the mutation, since that is information that wasn't available on the client beforehand:

```
mutation {
  createPerson(name: "Alice", age: 36) {
    id
  }
}
```

Resolvers

Resolvers are functions written in the GraphQL server corresponding to each field in GraphQL query/mutations. When a request comes to the server, it will invoke resolvers' functions corresponding to fields mentioned in the query.

Realtime Updates with Subscriptions

Another important requirement for many applications today is to have a *realtime* connection to the server in order to get immediately informed about important events. For this use case, GraphQL offers the concept of *subscriptions*.

When a client *subscribes* to an event, it will initiate and hold a steady connection to the server. Whenever that particular event then actually happens, the server pushes the corresponding data to the client. Unlike queries and mutations that follow a typical "*request-response*-cycle", subscriptions represent a *stream* of data sent over to the client.

Subscriptions are written using the same syntax as queries and mutations. Here's an example where we subscribe on events happening on the Person type:

```
subscription {
  newPerson {
    name
    age
  }
}
```

After a client sent this subscription to a server, a connection is opened between them. Then, whenever a new mutation is performed that creates a new Person, the server sends the information about this person over to the client:

```
{
  "newPerson": {
    "name": "Jane",
    "age": 23
  }
}
```

Defining a Schema

Now that you have a basic understanding of what queries, mutations, and subscriptions look like, let's put it all together and learn how you can write a schema that would allow you to execute the examples you've seen so far.

The *schema* is one of the most important concepts when working with a GraphQL API. It specifies the capabilities of the API and defines how clients can request the data. It is often seen as a *contract* between the server and client.

Generally, a schema is simply a collection of GraphQL types. However, when writing the schema for an API, there are some special *root* types:

```
type Query { ... }
```

```
type Mutation { ... }

type Subscription { ... }
```

The Query, Mutation, and Subscription types are the *entry points* for the requests sent by the client. To enable the allPersons-query that we saw before, the Query type would have to be written as follows:

```
type Query {
  allPersons: [Person!]!
}
```

allPersons is called a *root field* of the API. Considering again the example where we added the last argument to the allPersons field, we'd have to write the Query as follows:

```
type Query {
  allPersons(last: Int): [Person!]!
}
```

Similarly, for the createPerson-mutation, we'll have to add a root field to the Mutation type:

```
type Mutation {
  createPerson(name: String!, age: Int!): Person!
}
```

Notice that this root field takes two arguments as well, the name and the age of the new Person.

Finally, for the subscriptions, we'd have to add the newPerson root field:

```
type Subscription {
  newPerson: Person!
}
```

Putting it all together, this is the *full* schema for all the queries and mutation that you have seen in this chapter:

```
type Query {
  allPersons(last: Int): [Person!]!
}

type Mutation {
  createPerson(name: String!, age: Int!): Person!
}

type Subscription {
  newPerson: Person!
}

type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}

type Post {
  title: String!
  author: Person!
}
```

Limitations

GraphQL is indeed a great way to build and query APIs, but it does have certain limitations. A few of them are:

- The nested queries having multiple fields could lead to performance issues. GraphQL queries have to be carefully designed as the control is with the client and it could ask anything.
- Web caching is easier with REST compared to GraphQL, as the latter has a single endpoint.
- Retrieving objects recursively (to infinite length) is not supported in GraphQL. One has to specify to what depth it needs the data to get the recursive data.

Conclusion

Though GraphQL is becoming popular, it is not always the best choice, and it is not the alternative for REST web services. REST has its own advantages over GraphQL in terms of web cache, performance, etc.

Keeping the above limitations in mind will help you choose the right tool for your needs.