Introduction to GraphQL

He Dai

Background

Back in 2012, The engineers in Facebook began an effort to rebuild Facebook's native mobile applications.

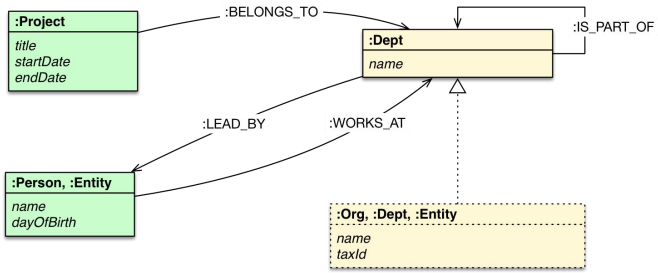
At the time, their iOS and Android apps were thin wrappers around views of their mobile website. While this brought them close to a platonic ideal of the "write once, run anywhere" mobile application, in practice it pushed their mobile-webview apps beyond their limits. As Facebook's mobile apps became more complex, they suffered poor performance and frequently crashed.

As they transitioned to natively implemented models and views, they found they need an API data version of News Feed — which up until that point had only been delivered as HTML. They evaluated their options for delivering News Feed data to their mobile apps, including RESTful server resources and FQL tables (Facebook's SQL-like API). They were frustrated with the differences between the data they wanted to use in apps and the server queries they required. They don't think of data in terms of resource URLs, secondary keys, or join tables; they think about it in terms of a graph of objects and the models they ultimately use in their apps like NSObjects or JSON.

There was a considerable amount of code to write on both the server to prepare the data and on the client to parse it. This frustration inspired a few of them to start the project that ultimately became GraphQL, which is simple enough to parse the data between the backend and frontend.

What is GraphQL?

An application- level query language provides a common interface between the client and the server for data fetching and manipulations.



Graph Database

The initial idea of GraphQL was based on Graph database. They both have direct relationship between different data layer.

But GraphQL is not a query language for graph databases. It's a declarative, strongly typed application-level query language that we can use with any backend—SQL, MongoDB, Redis, etc.. Example here from Facebook, get some information about a user: (left is request, right is response)

```
{
  user(id: 4802170) {
    id
    name
    isViewerFriend
    profilePicture(size: 50)  {
      uri
      width
      height
    }
    friendConnection(first: 5) {
      totalCount
      friends {
        id
        name
      }
    }
  }
}
```

```
{
  "data": {
    "user": {
      "id": "4802170",
      "name": "Lee Byron",
      "isViewerFriend": true,
      "profilePicture": {
        "uri": "cdn://pic/4802170/50",
        "width": 50,
        "height": 50
      },
      "friendConnection": {
        "totalCount": 13,
        "friends": [
          {
            "id": "305249",
            "name": "Stephen Schwink"
          },
          {
            "id": "3108935",
            "name": "Nathaniel Roman"
```

GraphQL fits between our backend and our frontend. It lets us define a model for our data, and a mapping between that model and our backends.

Key concepts of the GraphQL query language are:

- Hierarchical
- Product-centric
- Strong-typing
- Client-specified queries
- Introspective

I would like to highlight the concept strong-typing here which means that GraphQL introduces an application level type system. It's a contract between the client and server which means that your server in the background may use different internal types. The only thing here what matters is that the GraphQL server must be able to

receive GraphQL queries, decide if that it is syntactically correct and provide the specific data for that.

One of the co-creator of GraphQL, Nick Schrock, explains Facebook's motivation behind creating GraphQL:

"GraphQL is unapologetically driven by the requirements of views and the front-end engineers that write them. We start with their way of thinking and requirements and build the language and runtime necessary to enable that."

Why GraphQL?

The most essential reason is we want to manage the increasingly complex data requirements of modern web/mobile apps.

The best practice for web development has changed significantly over the last few years. While just a few years ago most websites used to be rendered on the server and have only relatively little client-side logic, the opposite is true of new apps today. Single-page applications and clients that implement complex logic are the new reality.

In a Ruby app, our model, our view, and our controller all live on the server. Getting data from our models to our view is not an issue, because almost all the data is right there on the server, where the page is rendered. In modern javascript apps, that is no longer the case: our controllers and our views now live mostly on the client, but most of the data is still on the server.

With all but the most custom RESTful APIs, fetching data from the server is both costly and complicated: the latency is high, and chances are we will either fetch more data than we need or make more roundtrips we would like—or both! That's where GraphQL come to the rescued.

With GraphQL, we can describe the required data in a more natural way, where helps when our client needs a flexible response format to avoid extra queries and/or massive data transformation with the overhead of keeping them in sync. It can speed up development, because in application structures like top-down rendering in React, the required data is more similar to our component structure.

Using a GraphQL server makes it very easy for a client side developer to change the response format without any change on the backend.

GraphQL vs REST API

Now, I would like to make a comparison between GraphQL and REST API.

So, what is REST again? Let’s make a recap: REST, an acronym for Representational State Transfer, is an architectural style rather than a formal protocol.

Objects in a typical REST system are addressable by URI and interacted with using verbs in the HTTP protocol. An HTTP GET to a particular URI fetches an object and returns a server-specified set of fields. An HTTP PUT edits an object; an HTTP DELETE deletes an object; and so on.

REST APIs are resource based. For instance, we want to address our resources like `GET /users/1/friends`, which is a unique path for them. It tells us very well that we are looking for the friends of the user whose id equals 1.

It is obvious to see that the advantages of REST APIs are that they are cacheable and transparent. But everything has two sides, Here the disadvantage comes that it's hard to specify and implement advanced requests with includes, excludes and especially with linked resources. We have to implement some customed endpoint with special words just like this:
`GET /users/1/friends/1/dogs/1?include=user.name,dog.age`

Here GraphQL comes to challenge REST. The key idea is that the code which best knows what data is needed for a UI is not on the server side but on the client side. The UI component knows the best of what data it wants.

REST delivers all the data a UI might need about a resource and it's up to the client to go look for the bits it actually wants to show. If that data is not in a resource it already has, the client needs to go off to the server and request some more data from another URL. With GraphQL the UI gets exactly the data it needs instead, in a shape handy for the UI.

As we can see here, a GraphQL query looks like this:

```
 {
   user(id: 4000) {
     id,
     name,
     isViewerFriend,
     profilePicture(size: 50)  {
       uri,
       width,
       height
     }
   }
}
```

And we get response like this:

```
{
  "user" : {
    "id": 4000,
    "name": "Some name",
    "isViewerFriend": true,
    "profilePicture": {
      "uri": "http://example.com/pic.jpg",
      "width": 50,
      "height": 50
    }
  }
}
```

This says we want an object of type user with id 4000. We are interested in its id, name and isViewerFriend fields. Also we need another object it is connected to: the profilePicture. We want the uri, width and height fields of this.

To talk with the backend, there is only a single endpoint that receives all these queries. Very unRESTful indeed!

In contrast, If we want to get such data by REST API, the whole query probably will looks like this:

```
/users/4000?
  field=id&
  field=name&
  field=isViewerFriend&
  field=profilePicture&
  filter:profilePicture.size=50&
  field=profilePicture.uri&
  field=profilePicture.width&
  field=profilePicture.height
```

See? Which is really nasty and the the result of this query would look the same as in the GraphQL example. It's important to notice that this REST API is not fully normalized -- the profilePicure data is not behind a separate URL that the client then needs to go to. Instead, the object is embedded in the result for the sake of convenience and performance, which is not the usual case. It need backend developer to implement specific endpoints for the requirement like that.

The drawback of REST API is exactly the advantage of GraphQL. That’s the reason why GraphQL was invented for. Here are the situations that GraphQL could solve where REST could not:

- Need to have single HTTP endpoint for managing CRUD operation and performing custom actions.
- Need to have meta data about available services and its data contracts.
- Need to fetch linked resources(domain objects) of application in single call, helps to avoid multiple custom endpoints and round trips.
- Need to have backward compatibility for services and its data contracts.
- Need to have custom query capability with filtering, sorting, pagination and selection set for targeted domain objects.
- Need to go away from a SOAP web services.

Conclusion

GraphQL is an application layer query language from Facebook. With GraphQL, we could define our backend as a well-defined graph-based schema. Then client applications can query their dataset as they are needed. It showed the overwhelming benefits than the current REST API, which is also the reason for inventing that from Facebook. Though there are some steep learning curve at the beginning of GraphQL, like we have to implement a framework named Relay in our frontend to deal with the data through GraphQL. But the history tendency just like that, no pain no gain, we take the benefits only after implement something new for improving the performance.

I believe someday GraphQL will take place of REST API eventually, just like the transformation from Angular to React.

Reference:

1. https://facebook.github.io/react/blog/2015/05/01/graphql-introduction.html
2. http://graphql.org/
3. http://facebook.github.io/graphql/
4. https://neo4j.com/developer/graph-db-vs-rdbms/