http://127.0.0.1:8080/#/1

Typical REST API. Server have no idea about how data will be used (it’s ok) and sends whatever it knows about some entity. All entities are atomic.

http://127.0.0.1:8080/#/2

Common issues

Over-fetching – it’s case when client receive too much. What can happen:

- system of entities is not perfect

- obsolete API

- System was designed to work with other service, but we must use it’s api for UI

- system designed to work with Desktop application, but company stared to develop mobile version of application and it has to use Desktop api

UI has issue: to much data and not really clear what it is and how to use it.

Typical solution is tu introduce tons of filters and parameters to query. And they become complex.

And developers has to create they own language of queries. => nobody except them will understand this language.

Under-fetching - it’s situation when you have to create another fetches after initial fetch to get whole information for your work.

One request has part of information => you have to investigate it => create another request for another byte of it.

Additional roud-trip to server => additional networking => additional resources and issues with performance.

Solution => expands. But this solution is completely custom. There is no any standard on how it should be used across different projects.

API changes and evaluation:

How to maintain changes:

- Versioning. When api has some version and you have to maintain old versions because some of you clients are using it

- Deprecation – when at some point we would say to client: in one month we will remove this field. And we have to manage it => additional costs

- Maintenance -> additional costs

http://127.0.0.1:8080/#/3

API Gateway

People were trying to fix those issues with different patterns like “API Gateway” .

All internal apis are covered with big api which keeps clear on what we expose and what we not expose.

But this kind of api have to work with different kind of clients who has different requirements. For instance mobile applications,

browsers and external service (browsers could create tons of requests per session, mobile could create one request per half an hour if it’s in background mode,

external server has completely different data requirements).

And peoples have to create different endpoints for different clients. => all endpoints are custom and we have to maintain all of them.

http://127.0.0.1:8080/#/4

QGL approach

It’s the same process: client asks for data from BE but with small difference. Client defines what exact data it wants to receive.

How companies are reaching it?

1. Your company should have one unified graph, instead of multiple graphs created by each team

2. There should be a single source of truth for registering and tracking the graph.

http://127.0.0.1:8080/#/6

GraphQL BE

Graph ql be should follow some protocol/rules which are defined by specification.

http://127.0.0.1:8080/#/10

Type system

Big advantage of GQL responses that they are type save.

It very familiar to typescript and it’s easy to integrate to TS.

Great feature that type system is exposed to outside from API and all your clients will be typed.

On BE you have to explain GQL what type is it. All future examples will be for JS.

GQL has interfaces, unions, enums.

Graphql trying to represent your system like a graph. Graph has no start and end. It has nodes that are connected somehow. And you have to traverse your graph from some point.

And graphQL has suggested start points: QUERY MUTATION SUBSCRIPTIONS. May be in some situation you could think about them like about root types.

http://127.0.0.1:8080/#/11

Mutations:

Interesting! all mutations will be executed in the order of definition

http://127.0.0.1:8080/#/12

Graphiql - small application build on top of graphql.

It’a absolutely generic and has no idea about api. It only knows one endpoint and it knows how to load types.

Example of fragments

query {

allUsers{

...Foo

comments{

...Foo

}

}

}

fragment Foo on Node {

id

}

If you have tons of components each component will be exposed like …Fragment1 …fragment2 etc

query {

query {

allUsers{

...UserFragment

comments{

...CommentFragment

}

}

}

fragment UserFragment on User {

id

name

}

fragment CommentFragment on Comment {

id

text

}

Introspection

You could go really deep in types

query {

\_schema {

queryType {

} }

\_\_type(name: "User"){

fields{

name

}

}

MUTATION

mutation UpdateUserComment($id: ID!, $comment: String!) {

updateUser(id: $id, comments: [{text:$comment}]) {

id

comments{

text

}

}

}

{

"id": "cju6uoeoy05q701705863fk00",

"comment": "bla-bbla"

}

Subscriptions???

http://127.0.0.1:8080/#/14

So. On FE you have all types, all handlers to change data, schemas, documentation. With the help of introspection you could grab all data about types

=> GraphQl applications are completely BE Agnostic

GQL do not care about all your technologies on BE and simply resolves what FE is needed

Transport: client defines transport. For instance client could set WS or set TCP connection and send binaries. And you steel use graphql.

http://127.0.0.1:8080/#/15

As result: BE shows possibilities to client

Client asks BE about what it needs

http://127.0.0.1:8080/#/16

More features

Deferred values. N+1Problem. Entity1 connected to Entity2. an d you have to create two quaries. Instead of that you could wait untill GQL

quary on BE for all inforamtion and will return you whole information that you need

Projecitons. Help to build efficient DB projeciotns. Query is already source of projeciton.

Middleware. There are tons of libraries that allows to handle queries, interupt them, intro

Query resucers. Endpoint allows to abuse properties of query and use it in non correct way. Reducers on BE allows to analyse query before it actually run

and decrease complexity of it to single value (for instance)

You could set a treshold on complexity of a quary. For instance depth.

http://localhost:8080/#/17

Why you should use graphQL

Better developer experience - you have schemas, strong typing, code completeon, documentation. and all this provided by endpoint.

Complex schemas - GraphQL has ability to server really complex schemas (Facebook, github)

Different kind of clients - mobile, services, desktops

Hide complexity from clients (microservices) - you could have extremely complex BE but client will never know about it.

Slow network with big lattency and interruptions - GraphQl shrinks data in responses, libraries support different transport and caches => all this could work in case of bad network

http://localhost:8080/#/19

Why you do NOT use GraphQL

REST already can do the same - if you want to have filtered set of properties use fields. Yes it's a custom solution, but solution. In many cases it's enought.

If you need static typization => use JSON schemas

Complexity could grow in uncontrollable way.

1) You have graph and somebody should serve it. Otherwise it will grow like mad.

2) If you have small application all those mutations, subscriptions, etc are adding extracomplexity and maintain costs

3) Errors and Binary files are not really trivial in GQL.

Learning curve - you have to change your mind. Instead of atomic REST endpoints you have to have sunbaked graph of entities

Web cache with REST - Web cache on Transport level could solve tons of problems. But not incase of one endoint because all calls are going to it

Schemas are static - You can not change schema dynamically. You do not have dynamic types. All flexibility of GQL is bounded by schema.