as an alternative to REST

Agenda

Fundamentals and key concepts 1

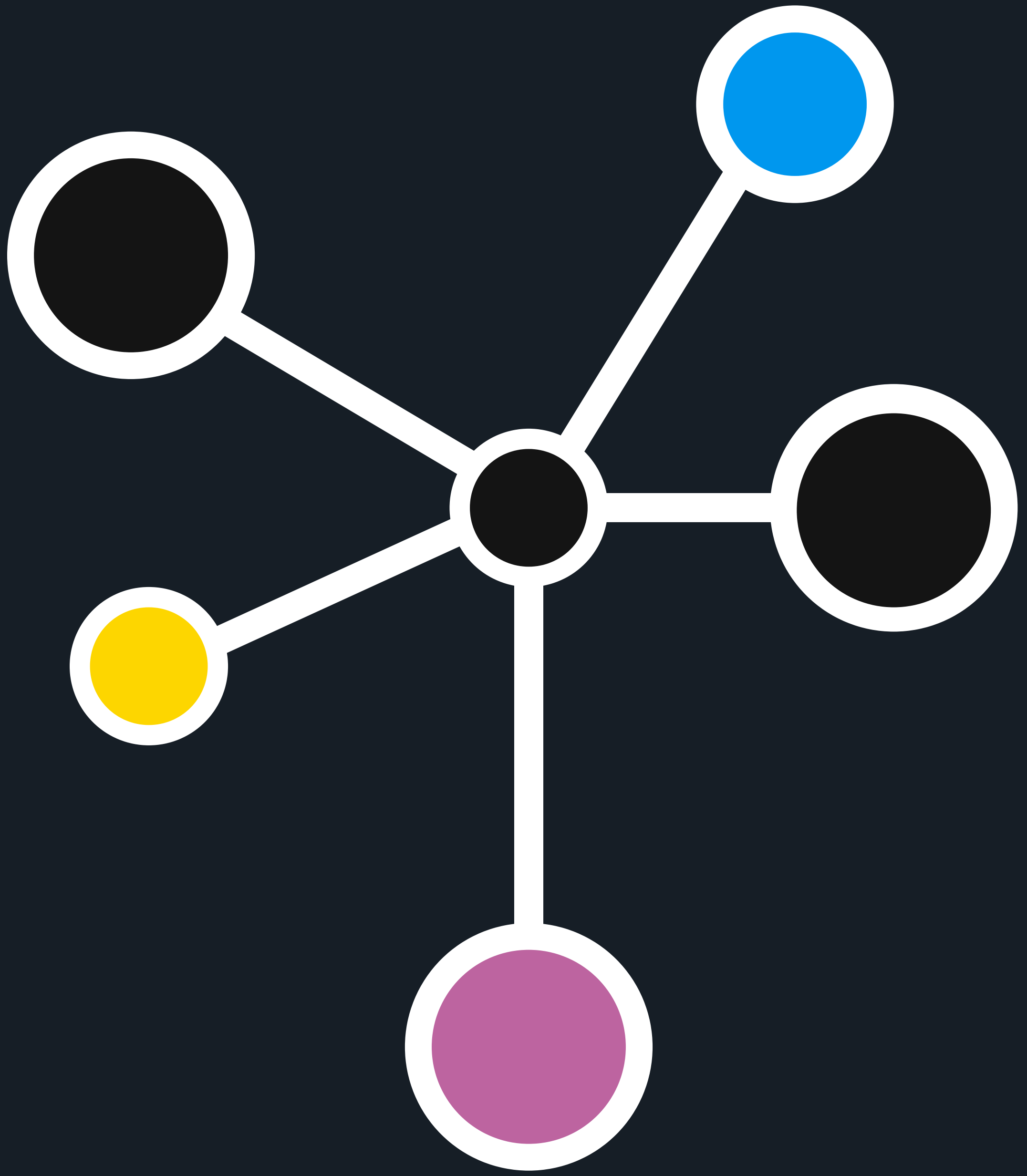
Main concepts in SOA 2

Benchmarking (NodeJS) and available libs 3

Conclusions 4

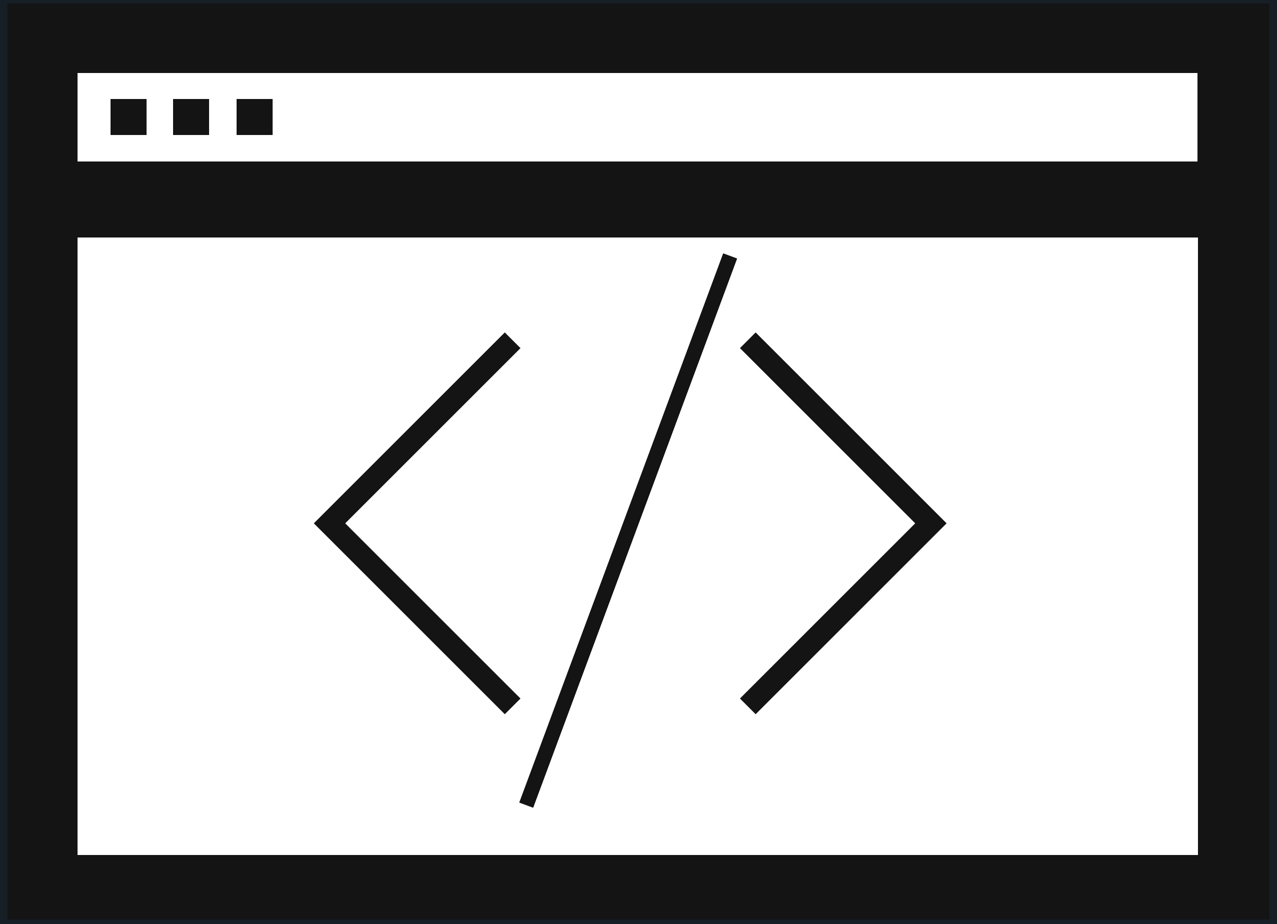
Fundamentals and concepts

What is GraphQL?



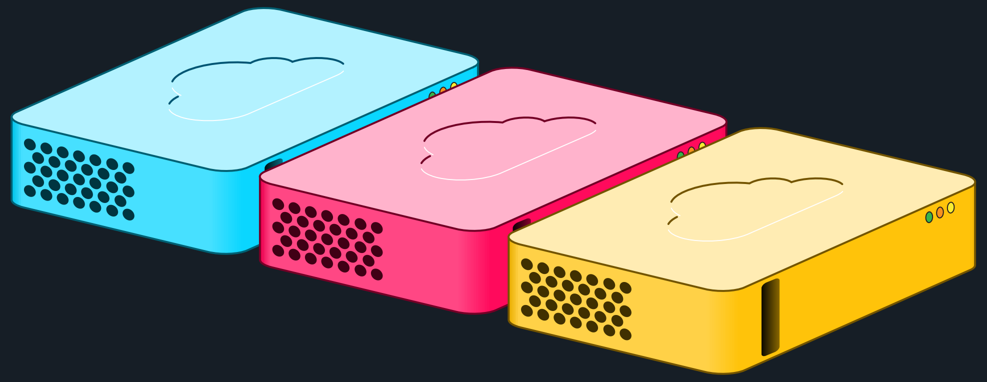
A graph-based query language for your APIs

What is GraphQL?

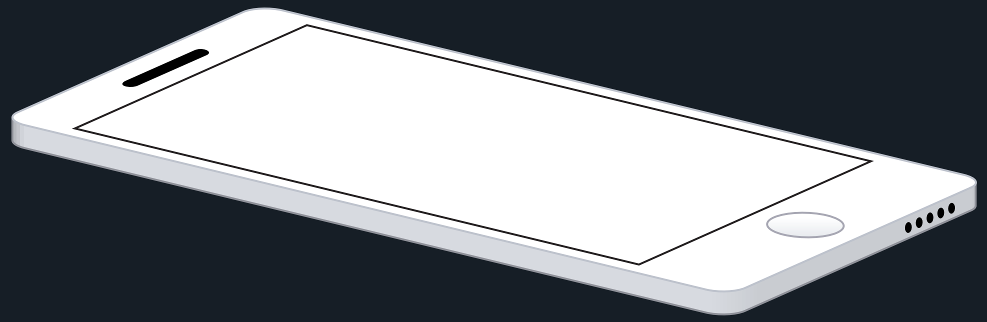


A server implementation

What is GraphQL?



```
query getUser($id: ID!, $includeLeisure: Boolean!)
{
  user(_id: $id){
    ...userFields
    leisure @include(if: $includeLeisure) {
      name
      ... on Movie {
          duration
      }
      __typename
    }
  }
}
```



A way to get many resources with a single request

What is GraphQL?

A specification with implementations for different languages

What is GraphQL?

Owing to Dgraph, a database query language

What it is not

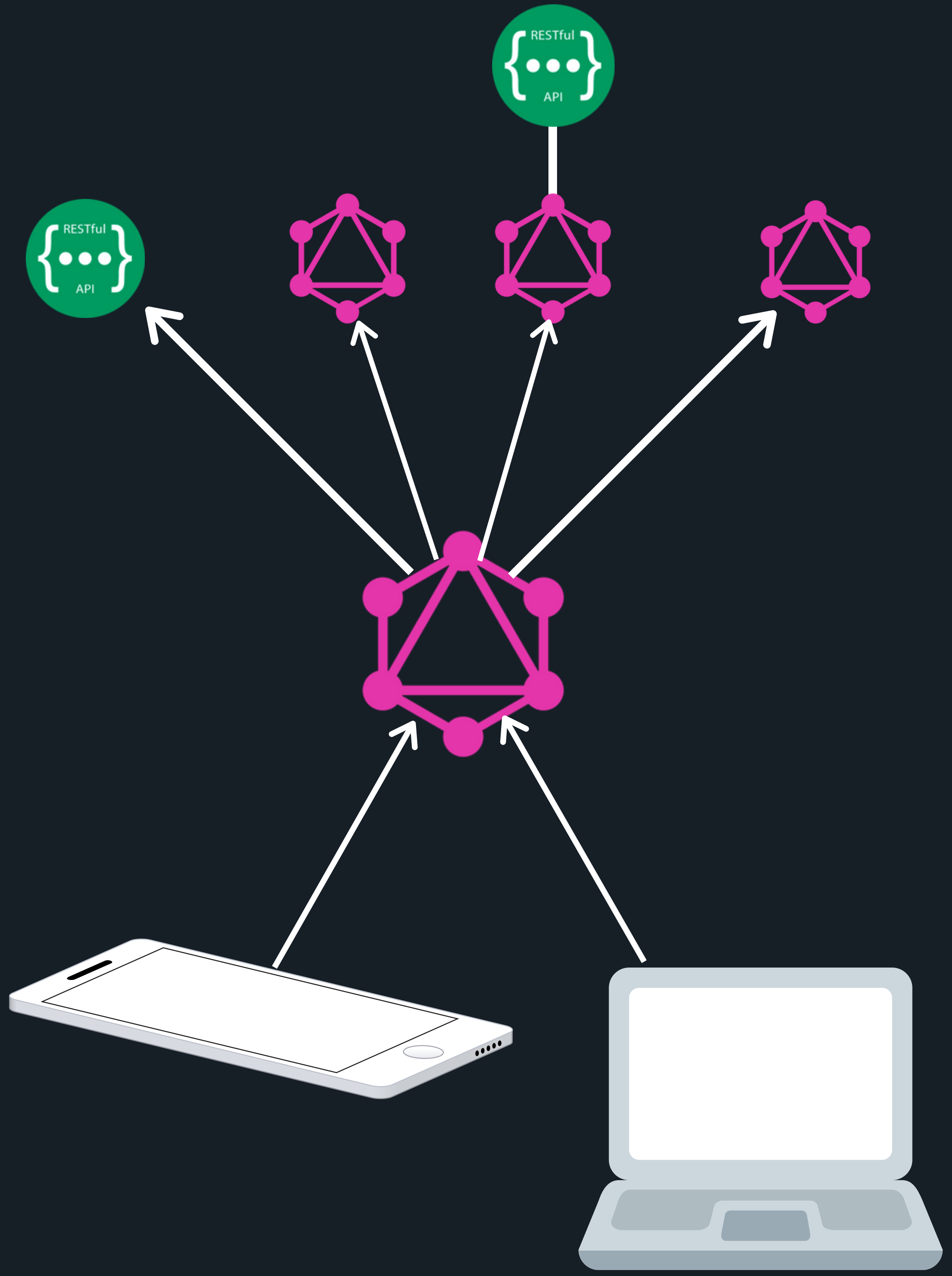
A graph database. The specification is not limited to specific databases

What it is not

A solution to manage client state.
But it may become a replacement
for global states

How to replace global state management libraries

What it is not



Necessarily a replacement for REST APIs. Both can work together

REST interaction scheme

http://domain.com/resource

http://domain.com/resource/1

http://domain.com/resource?page=1&limit=1

http://domain.com/resource?page=1&limit=100&name=myname

http://domain.com/v1/resource?page=1&limit=1&fields=name,age

URI request representing a resource

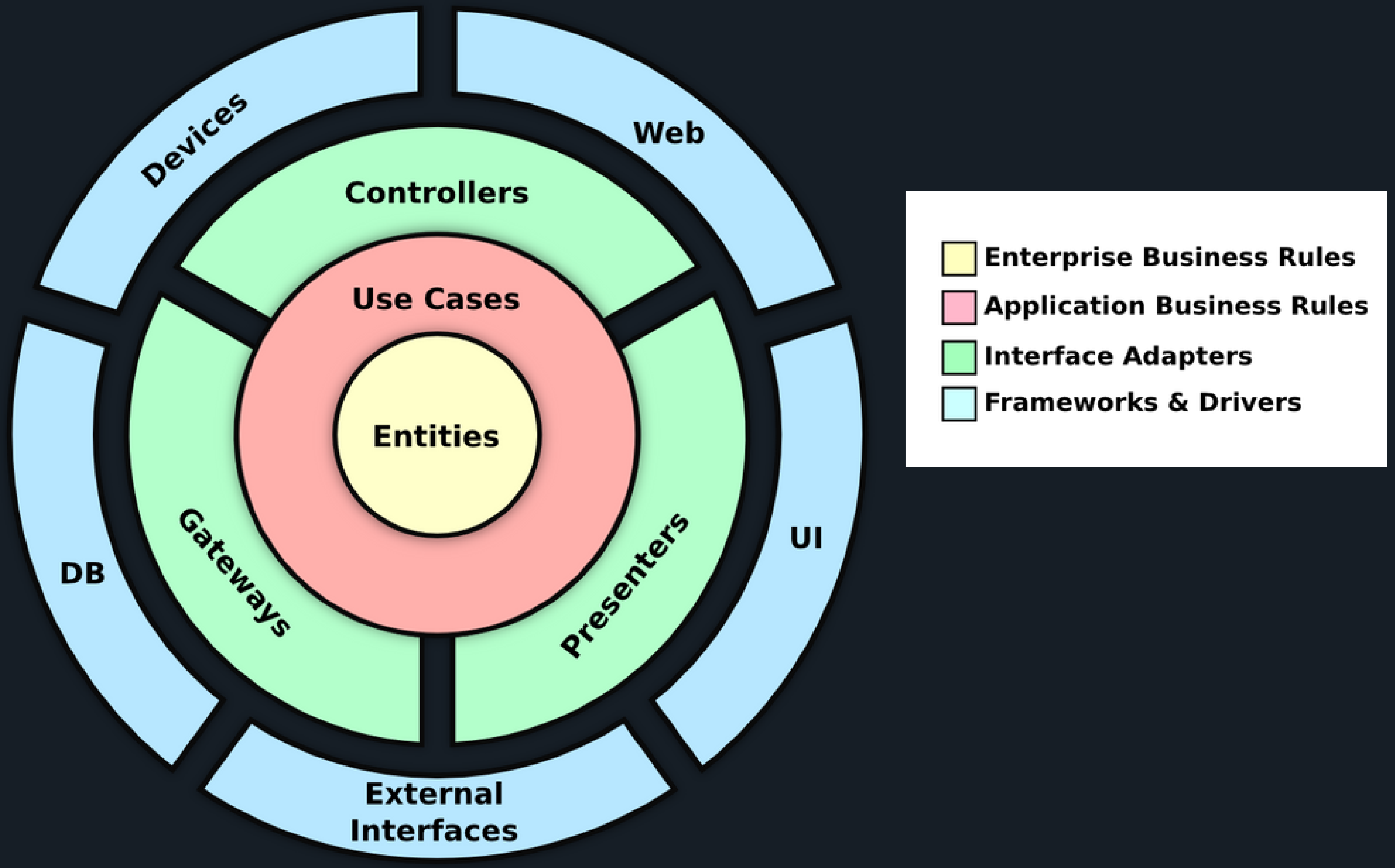
```
import {usersDB} from '@data-access';

export function buildGetUsers() {
  return async function getUsers() {
    const headers = {
      'Content-Type': 'application/json'
    }
    try {
      const users = await (await usersDB).findAll();
      return {
        headers,
        statusCode: 200,
        body: users
      }
    } catch (e) {
      console.log(`${new Date()} : An error when getting users has occured`);
      return {
        headers,
        statusCode: 400,
        body: {
          error: e.message
        }
      }
    }
  }
}
```

Server's implementation that calculates a JSON response

```
{
    "next": {
        "page": 1,
        "limit": 1
    },
    "results": [
        {
            "_id": "e8b74ea8-082c-41e3-b3af-138c70f42d7f",
            "username": "asdasdasd",
            "password":
"$argon2i$v=19$m=4096,t=3,p=1$Dx3E72xDrD/4XkCEJoFfww$RsOXj2
LKop54bC6wEOpCePR7J2bVANFUmTJp7qB+BqY"
        }
    ]
}
```

JSON response

REST clean architecture



Uncle Bob's clean architecture

GraphQL interaction scheme

```
query getUser($id: ID!, $includeLeisure: Boolean!) {
  user(_id: $id){
    ...userFields
    leisure @include(if: $includeLeisure) {
      name
      ... on Movie {
        runningTime
      }
      __typename
    }
  }
}
```

Write and run queries

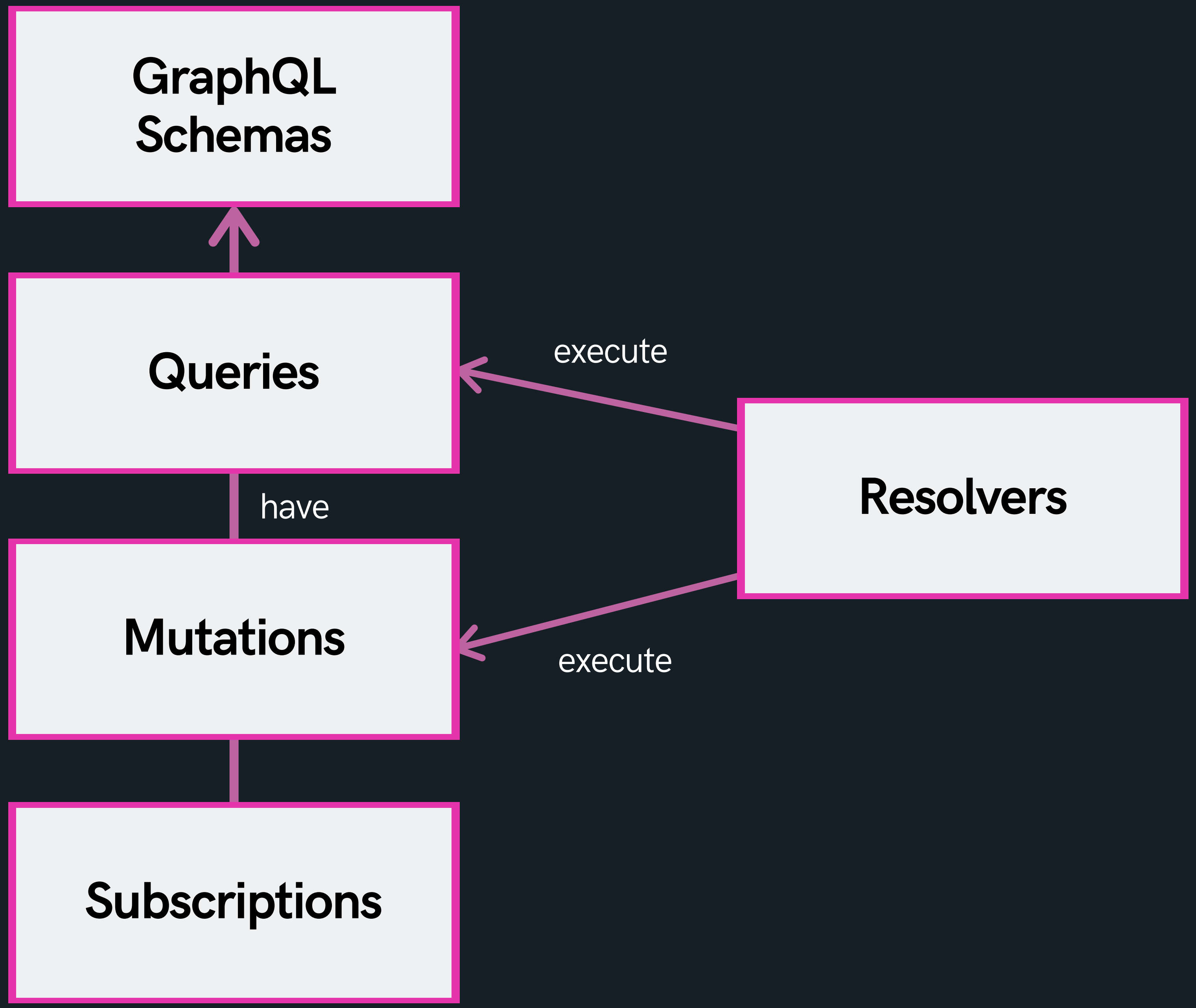
```
  type User {
    _id: ID
    username: String!
    password: String!
    createdAt: Date!
    role: ROLE!
    leisure: [Leisure!]!
  }

type Query {
    users: [User]
    paginatedUsers(first: Int, after: ID): PaginatedUserResult
    user(_id: ID!): User
  }

 user: async (_: any, {_id} : {_id: string})=>{
               const user = await
dbClient.collection('users').findOne({_id: new ObjectId(_id)});
               return user;
 }
```

GraphQL server (schema types and their implementations) resolves queries and mutations

```
{
    "data": {
        "user": {
            "_id": "606456492a3c5e0e30190cdf",
            "username": "new user",
            "createdAt": 1617188425637,
            "leisure": []
        }
    }
}
```

GraphQL client. Get exactly what you ask for

GraphQL clean architecture



GraphQL fundamentals

How do we define schemas, queries and mutations?

GraphQL fundamentals

Scalar types

- Int: A signed 32-bit integer.
- Float: A signed double-precision floating-point value.
- String: A UTF-8 character sequence.
- Boolean: true or false.
- ID: The ID scalar type represents a unique identifier, often used to refetch an object or as the key for a cache. The ID type is serialized in the same way as a String; however, defining it as an ID signifies that it is not intended to be human-readable.
- Custom scalars (like Date, JSON and so on)

Object types

```
type User {
   _id: ID
   username: String!
   password: String!
   createdAt: Date!
   role: ROLE!
   leisure: [Leisure!]!
 }
```

Queries

```
   type Query {
     users: [User]
    }

Query: {
       users: async ()=>{
          ...db call....
          return users;
       }
}
```

GraphQL fundamentals

Mutations

```
type Mutation {
    createUser(
      username: String!
      password: String!
      role: ROLE!
    ): Boolean
}
```

```
Mutation: {
          createUser: async (par) => {
              try{
                   ...implementation
                  return true;
              }catch(err){
                  return false;
              }
          }
}
```

Arguments

```
type Query {
    paginatedUsers(first: Int, after: ID): PaginatedUserResult
  }
```

Enums

```
enum ROLE {
    USER
    MODERATOR @deprecated(reason: "Use 'User' instead")
    ADMIN
  }
```

GraphQL fundamentals

Interfaces

```
interface Leisure {
   name: String!
 }
```

```
Leisure: {
        __resolveType(obj: any) {
          if (obj.runningTime) {
             return 'Movie';
          }
          return 'Magazine';
        },
}
```

Union types

```
union Leisure = Movie | Magazine
```

Input types

```
input LeisureInput {
   id: ID!
   name: String!
 }
```

```
type Mutation {
   addMagazineLeisure(leisureInput: LeisureInput!): Boolean
 }
```

GraphQL fundamentals

Fragments

A concise way to aggregate reusable fields

Aliases

Directives

A way add additional logic to schemas

Subscriptions

Literally publisher/subscriber pattern (not really a part of the specification, poorly supported)

GraphQL fundamentals

Validation

GraphQL types system can predetermine whether a query is valid or not. If not end users get an error message.

Validation

GraphQL fundamentals

Introspection allows clients to ask a GraphQL schema for information about what queries it supports

Introspection

Introspection

Key concepts

1

Uses stateless interactions

REST service store state information on the server. Clients maintain this information

2

Explicitly uses HTTP methods for communication

3

Uses standard HTTP status codes

4

Manipulates resources

REST represents objects exposed as resources. A unique URL identifies each resource

5

Provides a hypermedia-driven API

REST services return links to available resources

6

Server centric and most likely version dependent

RESTful key concepts

Key concepts

1

View centric

Designed to satisfy frontend application requirements

2

Communicates over HTTP (Post method) by means of hierarchical queries

3

Manipulates strictly typed objects

A GraphQL server defines a specific types system

4

Introspective

The type system itself is queryable. Tools are built around that capability.

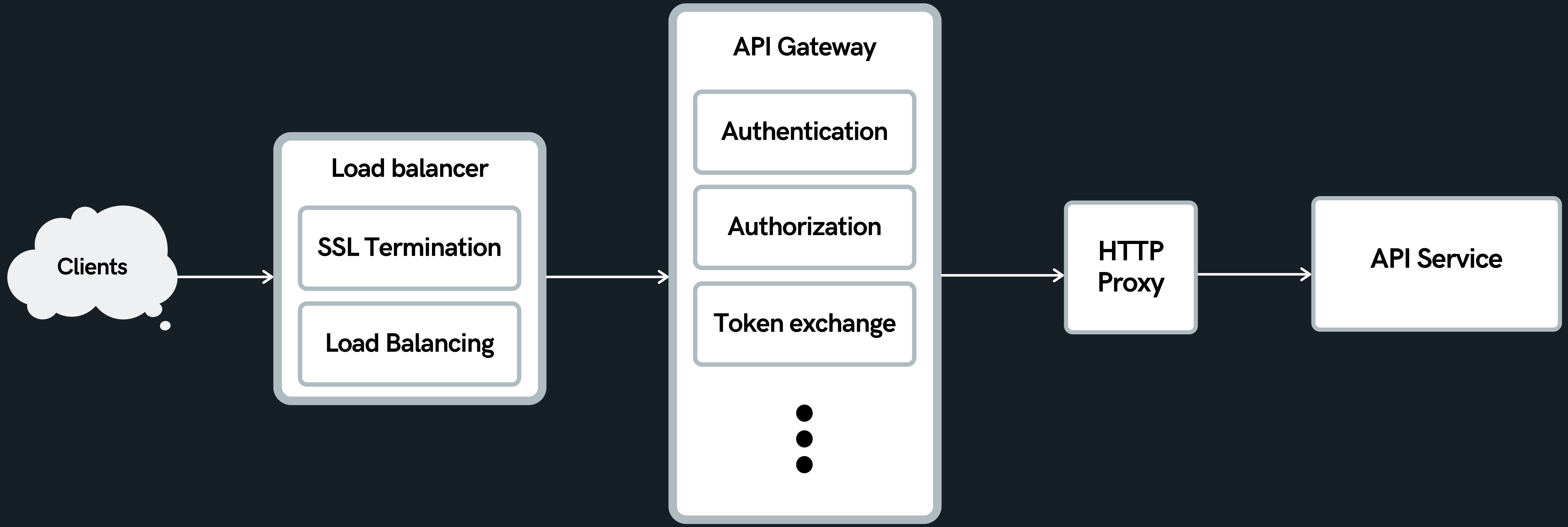
5

Version free

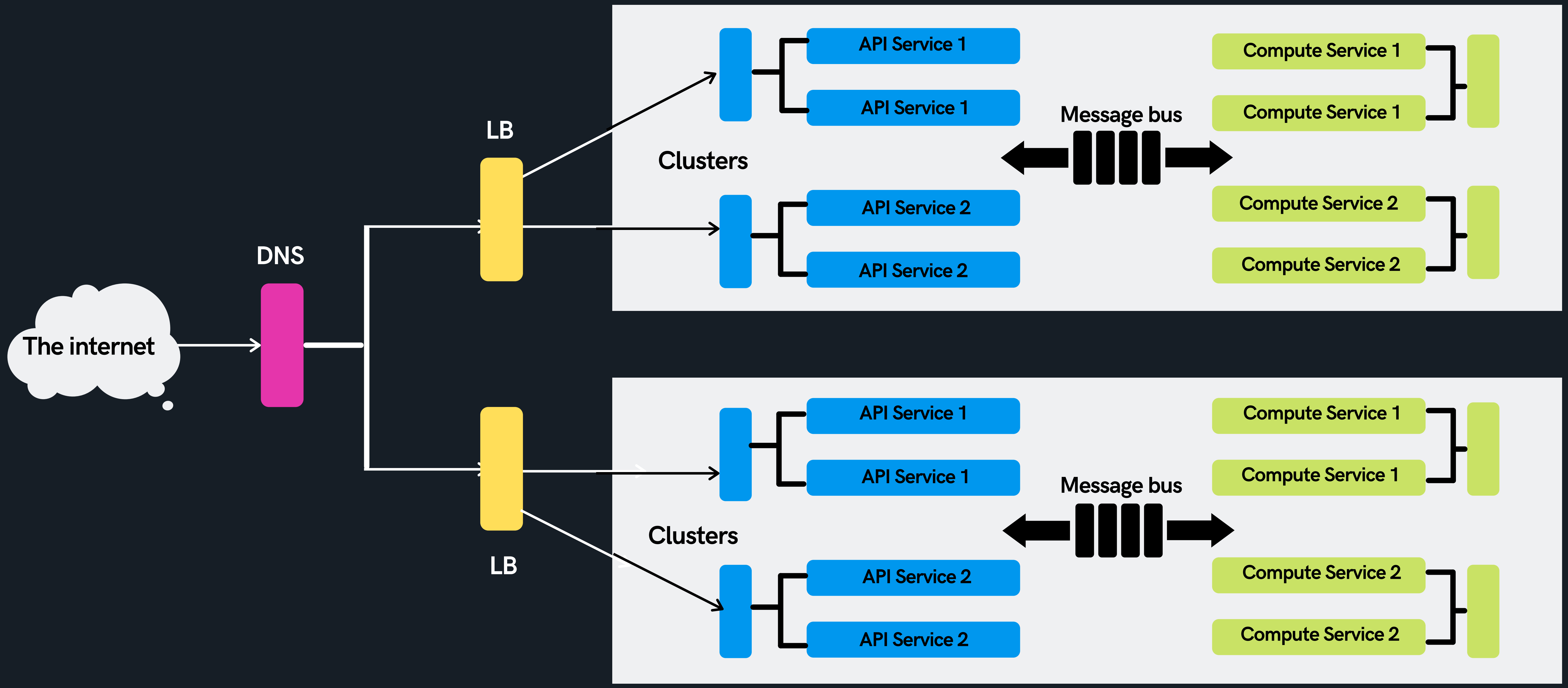
GraphQL key concepts

Main concepts in SOA

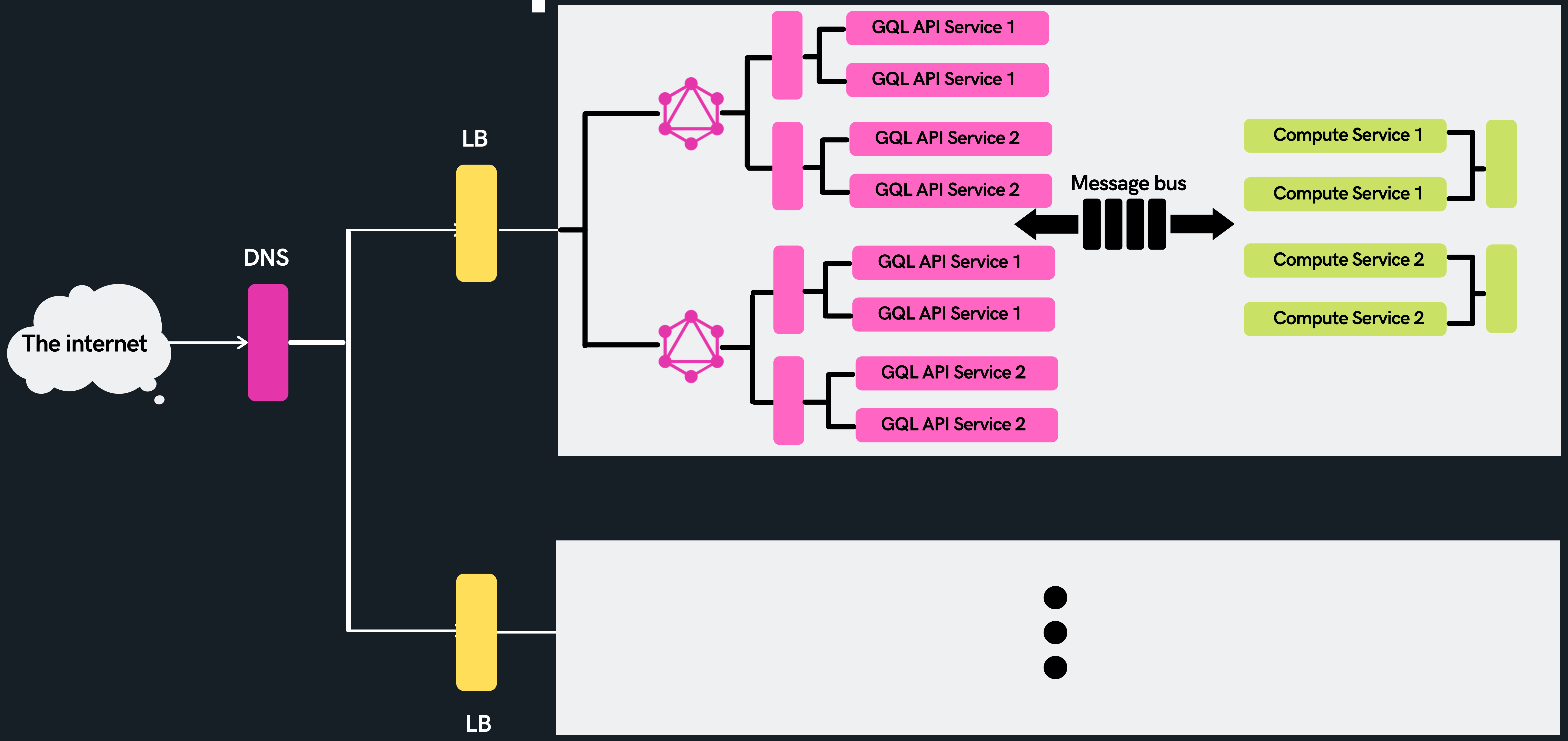
Topology of an API Service



Typical SOA for REST



SOA for GraphQL



Apollo Federation

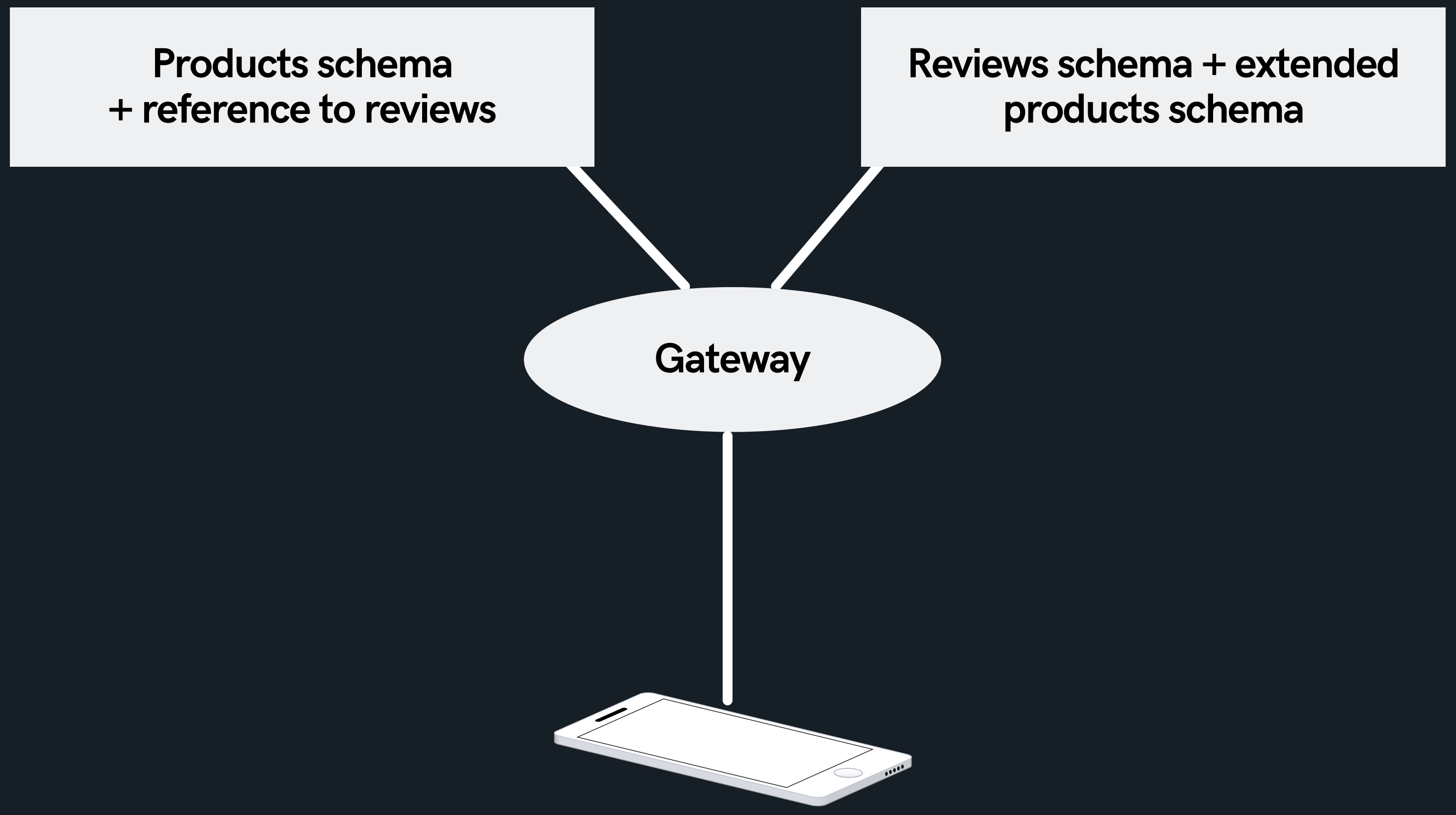
Every GraphQL API should have one graph but allow teams to work on different products without interfering with each other.

So, one graph but multiple graphs?

Federation

Apollo Federation

Clients should consume one cohesive graph. But the server implementations should be federated



Federation

Benchmarking and available libs

Benchmarking (Node JS)

Framework	Requests/s	Latency/ms	Throughput/Mb
uWebSockets-graphql+jit	7898.0	0.08	48.59
benzene-http	6176.4	0.28	38.69
fastify-REST	5384.4	0.30	43.19
express-REST	3758.2	1.03	30.38
mercurius+graphql-compose	3741.4	0.73	23.42

GraphQL vs REST benchmarking (NodeJS)

Benchmarking (Node JS)

Framework	Requests/s	Latency	Throughput/Mb
apollo-server-fastify+graphql-jit	3446.6	1.25	21.68
express-gql	3391.4	1.31	21.45
apollo-server-express	1662.4	2.70	10.56

- JIT optimization helps with performance problems
- Apollo server does have overhead
- Type graphql adds overhead
- It is possible to achive similar to REST performance, however it takes a lot of tweaking and extra code

GraphQL JIT

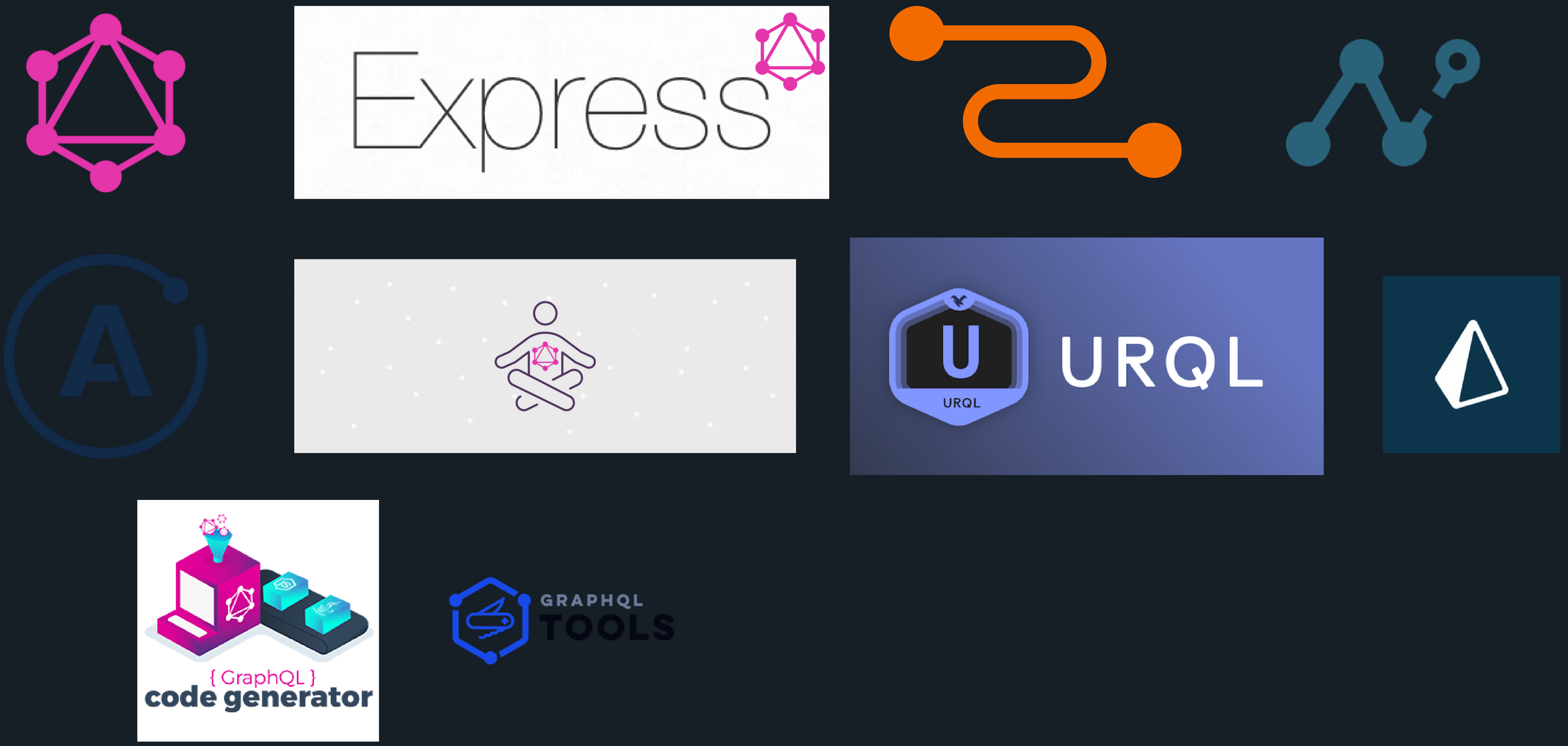
GraphQL vs REST benchmarking (NodeJS)

Benchmarking

"I think the TL;DR of this issue is that GraphQL has some overhead ... In other words GraphQL.js does runtime type checking and sub-selection and this has some cost." - Lee Byron

The GraphQL issue

GraphQL libraries



Libraries (JS/TS)

Libraries

Conclusions

Conclusions

Developers experience	API Gateway	SOA
++ Brilliant for organizing frontend and backend developers interaction without extra tools + Saves the day for frontend developers	– Poor support from existing gateways. Even though, out of the box gateway is fairly decent, mostly, gateway level solutions must be implemented from scratch	++ Perfect for data composition in parallel ++ Easy to set up inter-service communication
+ API-first design: great tools (Swagger, Apiary and so on) ? May well be inconvenient for frontend developers	++ API Gateways take away from REST endpoints common tasks (OAuth, API keys, throttling, security)	– – It is difficult to combine data without extra logic for each resource + Wide variety of microservice oriented frameworks and libraries

Conclusions

Authentication and Authorization	Caching	Versioning and data fetching
? Bare graphql specification doesn't cover this. Client/Server providers give necessary tools and techniques to achieve that.	- - Virtually impossible to utilize network caching (Only one endpoint for all operations) + Object types caching is possible. And there is a specification on that.	+ No API versioning. It should be avoided and tools are provided + No over-fetching or under-fetching. Always get what you request
++ Major standards supported by API Gateways and frameworks.	++ Network caching is easy. Common tools can be used + Services can cache data similarly to GraphQL	- v0, v1, v2, ... ? Over- and Under-fetching or just the right data with a long fields query in URIs. However, there are many solutions for that.

Conclusions

Maturity	Learning	Performance
? Not mature enough (e.g. subscriptions are poorly supported in some cases, not enough util libraries). + Rapid development and improvement of core frameworks	- A lot of new types, concepts, caching peculiarities and federation tricks to learn - Error handling is overcomplicated	- May require extra optimizing libraries and some tweaking to achive close to REST performance ? Without ORMs or DataLoaders, be careful with N+1
++ Has been with us for ages. Plenty of frameworks, libs and best practices	? Still a lot to learn. Though, no extra types and complex features	+ Relatively predictable thanks to best practices