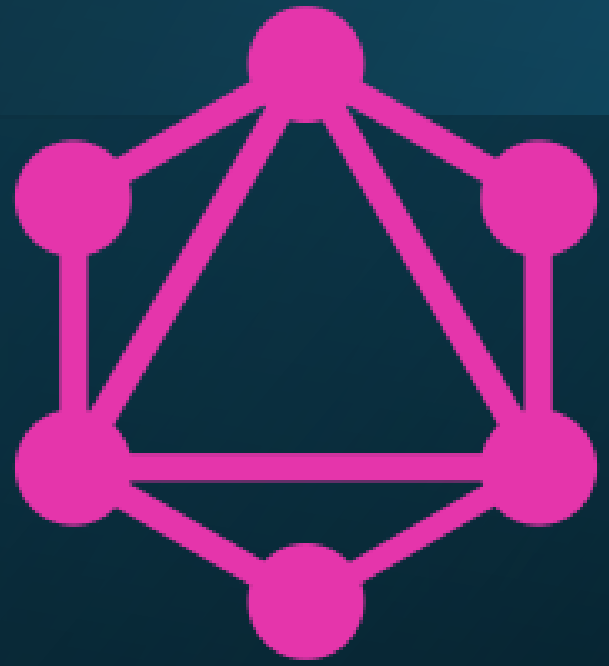
Системный Дизайн

технологии интеграции

GraphQL

01



Для чего

У **RESTful API** – как протокола межсервисного взаимодействия есть несколько проблемных мест:

- Нет общей для всех схемы и **introspection**
- **Quering** вложенных друг в друга сущностей
- Многословность и недостаточность информации одновременно и т.д.

GraphQL был призван решить это проблемы.

Два типа запросов

GraphQL – это язык запросов, который позволяет получать данные, изменять их и описывать интерфейс взаимодействия.

Запросы могут быть:

- Query - запросы чтения
- Mutations - запросы на изменения

Query позволяют выбрать ровно те данные, которые нужны

```
query {
  me {
    name
  }
}
```

запрос

```
{
  "data": {
    "me": {
      "name": "Marc"
    }
  }
}
```

результат

Query позволяют выбрать связанные данные

запрос

```
query {
  me {
    name
    friends(first: 2) {
      name
      age
    }
  }
}
```

результат

```
{
  "data": {
    "me": {
      "name": "Marc",
      "friends": [{
        "name": "Robert",
        "age": 30
      }, {
        "name": "Andrew",
        "age": 40
      }]
    }
  }
}
```

Спецификация https://spec.graphql.org/October2021/

GraphQL Scheme позволяет описать формат данных

```
type Shop {
  name: String!
  # Where the shop is located, null if online only.
  location: Location
  products: [Product!]!
}

type Location {
  address: String
}

type Product {
  name: String!
  price: Price!
}
```

В запросе можно передавать параметры

```
query {
  # 1. The shop field re
  shop(id: 1) {
    # 2. field location
    # Returns a `Locatio
    location {
      # 3. field address
      # Returns a String
      address
    }
  }
}
```

Пример описания запроса

```
type Query {
   shop(id: ID!): Shop!
}
```

```
type Query {
  shop(owner: String!, name: String!, location: Location): Shop!
}
```

Попробуем

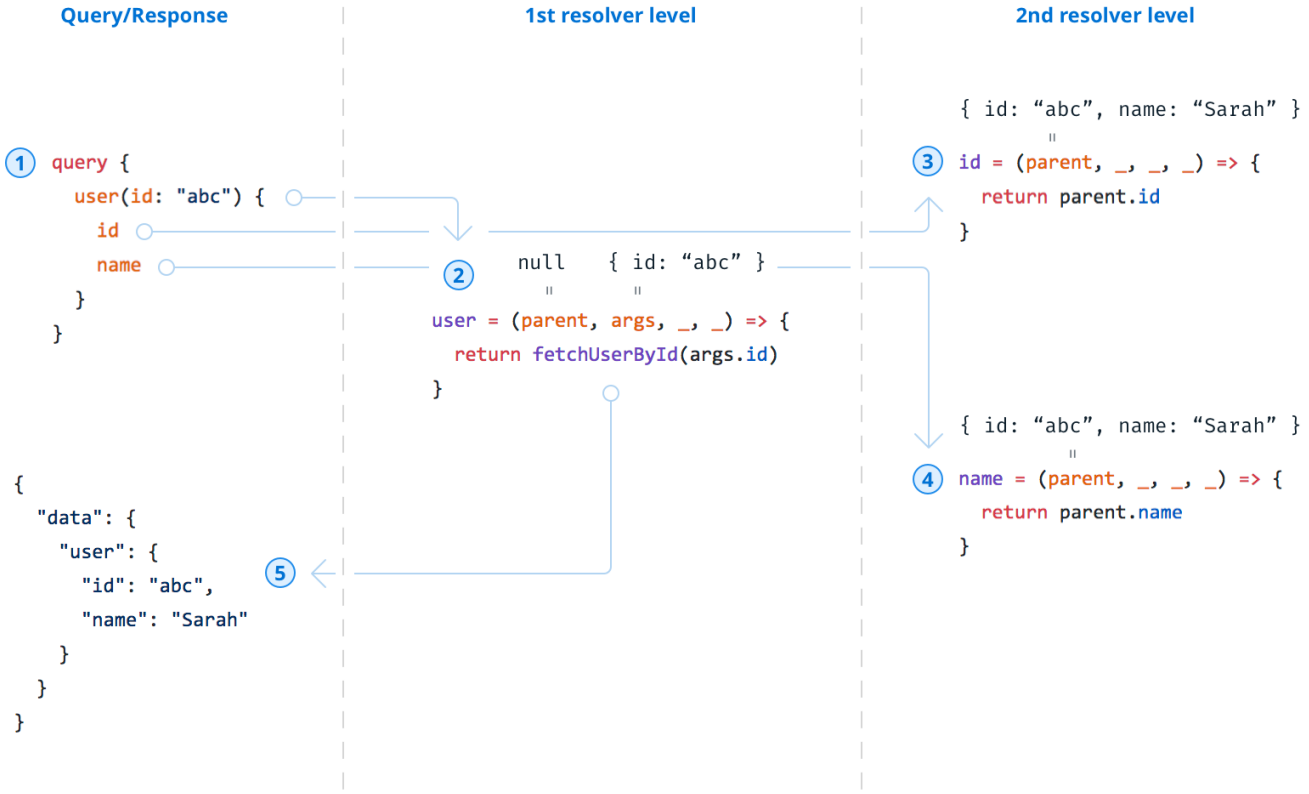
https://countries.trevorblades.com/

```
{
  country(code: "RU") {
    name
    native
    capital
    emoji
    currency
    languages {
      code
      name
    }
  }
}
```

Как устроен GraphQL?

https://gist.github.com/loganpowell/56af8f0e2b68af21f88833075092f864

Resolver



Изменение данных: мутации

```
type Mutation {
  addProduct(name: String!, price: Price!): AddProductPayload
}

type AddProductPayload {
  product: Product!
}
```

```
mutation {
  addProduct(name: String!, price: Price!) {
    product {
      id
    }
  }
}
```

Subscriptions

схема

```
schema {
        query: Queries,
        mutation: Mutations,
        subscription: Subscriptions
}
type Subscriptions {
        productAdded(cartId: String!): Cart
}
```

подписка

```
subscription {
productAdded(cart: 1) {
  items {
    product ...  }
  subtotal
 }
}
```

нотификация

```
{
 "data": {
  "productAdded": {
   "items": [
    { "product": ..., "subTotal": ... },
    { "product": ..., "subTotal": ... },
    { "product": ..., "subTotal": ... },
    { "product": ..., "subTotal": ... }
   ],
   "subTotal": 289.33
  }
```

Опасность в применении GraphQL:

Нарушение контекста

GraphQL **провоцирует** проектировать API, которые сильно связны с деталями реализации.

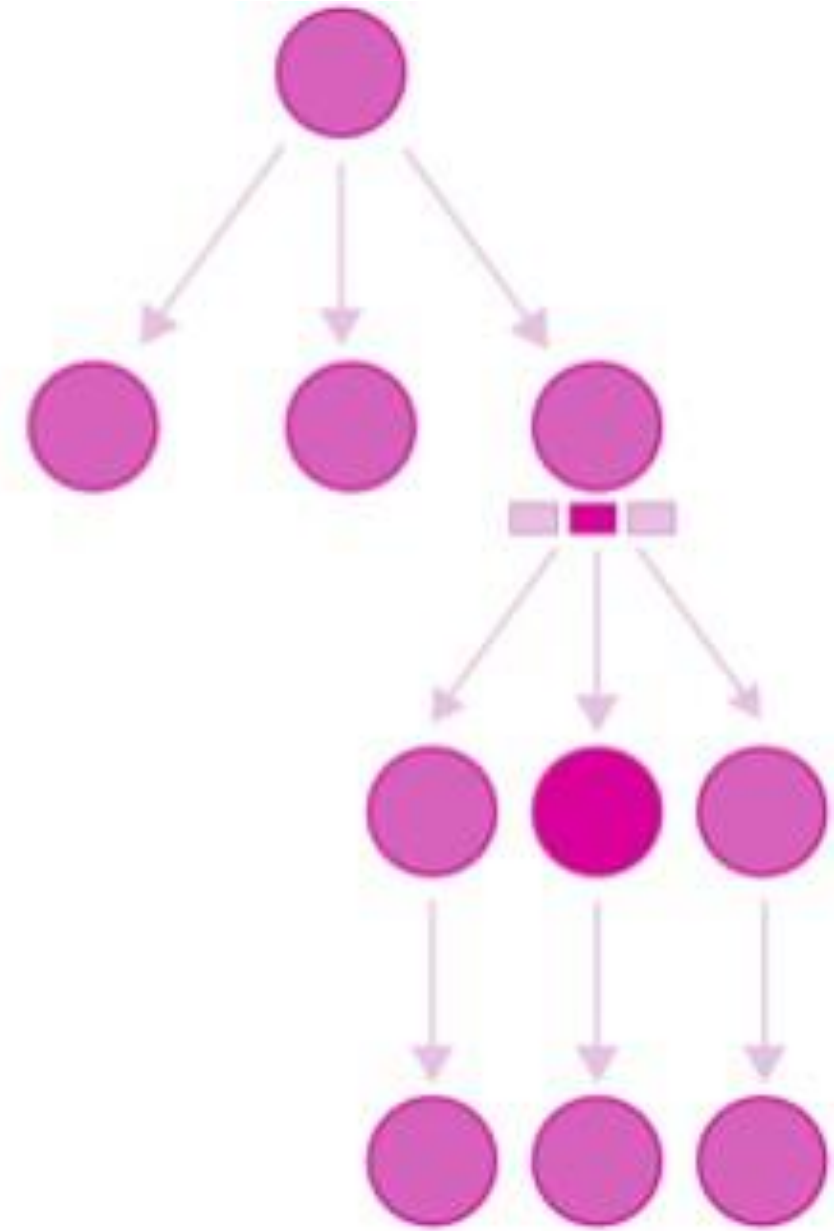
Т.е. есть **множество** инструментов, которые позволяют автоматически строить Scheme-у из табличек в БД.

Это является **антипаттерном**.

Проблемы производительности

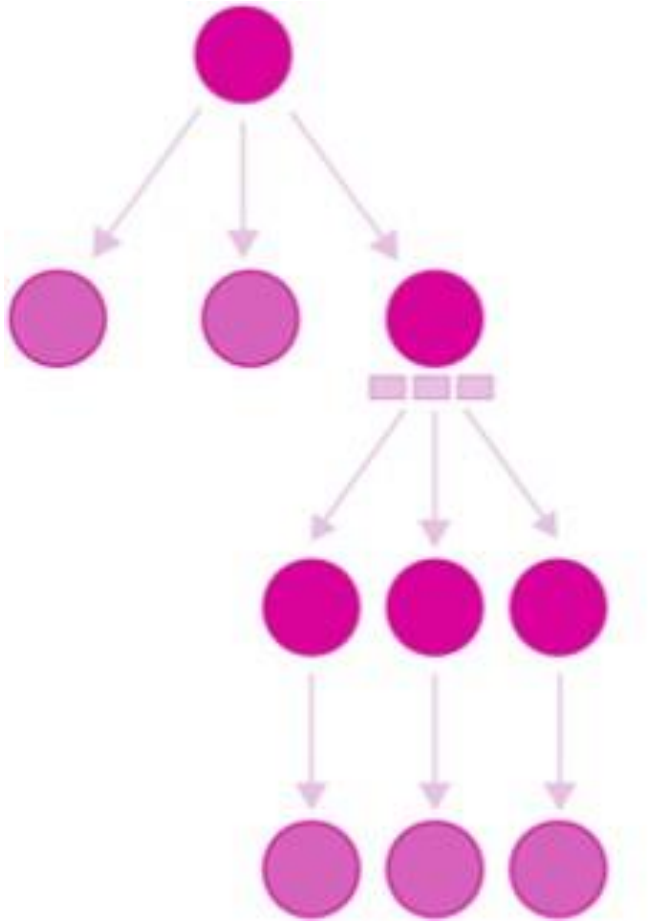
Опасность в применении GraphQL

```
query {
  name
  age
  friends(first: 3) {
    bestFriend {
      name {
    }
  }
}
```



Наивная реализация resolver-а приведет к N+1 запросам.

Проблемы с производительности



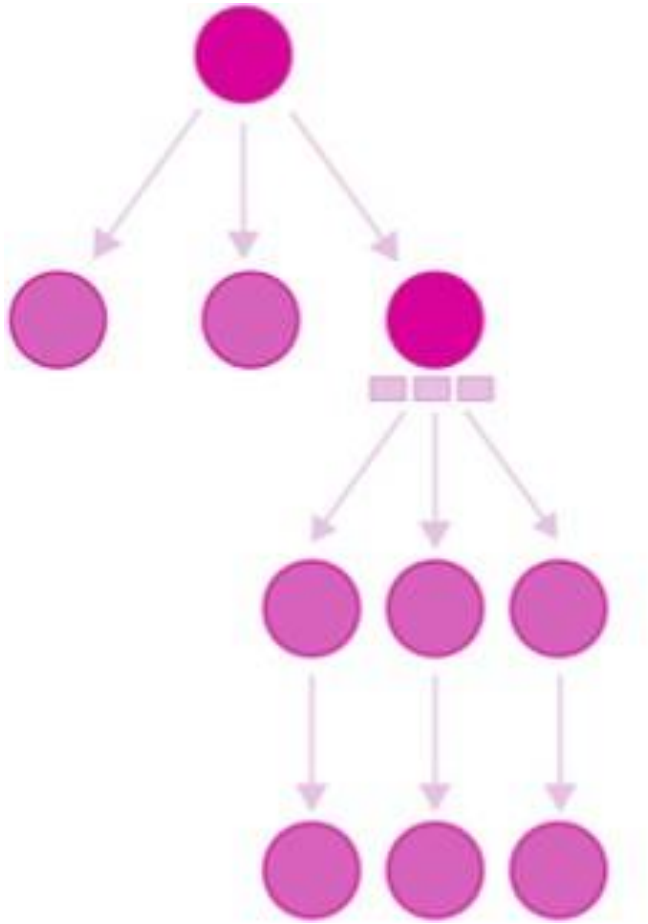
```
SELECT * FROM users WHERE id = 1
```

```
SELECT * FROM friends WHERE user_id = 1 LIMIT 3

/*
  Load the first 3 first friends using their ID
*/
SELECT * FROM users WHERE id IN (2, 3, 4)
```

```
/*
  Each resolver loads the best friend for a
friend.
*/
SELECT * FROM users WHERE id = 2
SELECT * FROM users WHERE id = 3
SELECT * FROM users WHERE id = 4
```

Чтобы сделать за оптимальное количество запросов, необходимо LazyLoading и паттерн DataLoader.

Проблемы с производительности



```
SELECT * FROM users WHERE id = 1

/*
  Load the first 3 first friends IDs
*/
SELECT * FROM friends WHERE user_id = 1 LIMIT 3

/*
  Load the first 3 first friends using their ID
*/
SELECT * FROM users WHERE id IN (2, 3, 4)

/*
  Load the best friend of each
*/
SELECT * FROM users WHERE id IN (5, 6, 7)
```

GraphQL для фронтового API

- GraphQL удобен для построения Frontend.
- При изменениях структуры UI достаточно просто переделать запрос
- Можно очень гибко менять логику без доработок на backend

GraphQL для фронтового API

Возможность запускать произвольные запросы с фронта – **очень опасна**.

Потому что можно как произвольно, так и не произвольно положить систему.

```
query {
  product {
    variants {
      product {
        variants {
          product {
            # We can do that for a while
            variants {}
          }
        }
      }
    }
  }
}
```

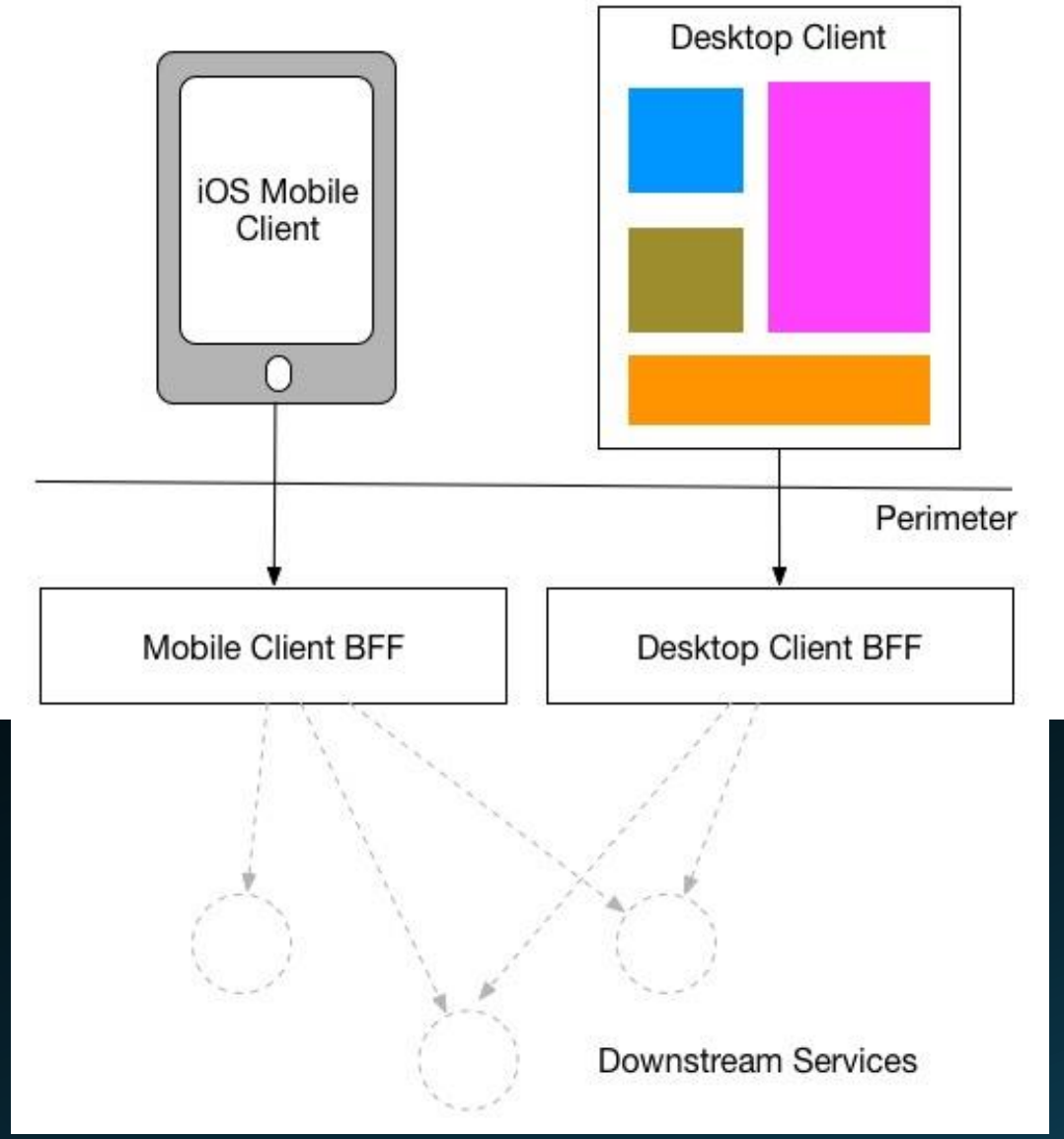
GraphQL для фронтового API

Возможность интроспекции – это полное раскрытие схемы и методов. Которое бывает не всегда полезным. В зрелых сервисах из-за security issues не очень любят отдавать полную схему

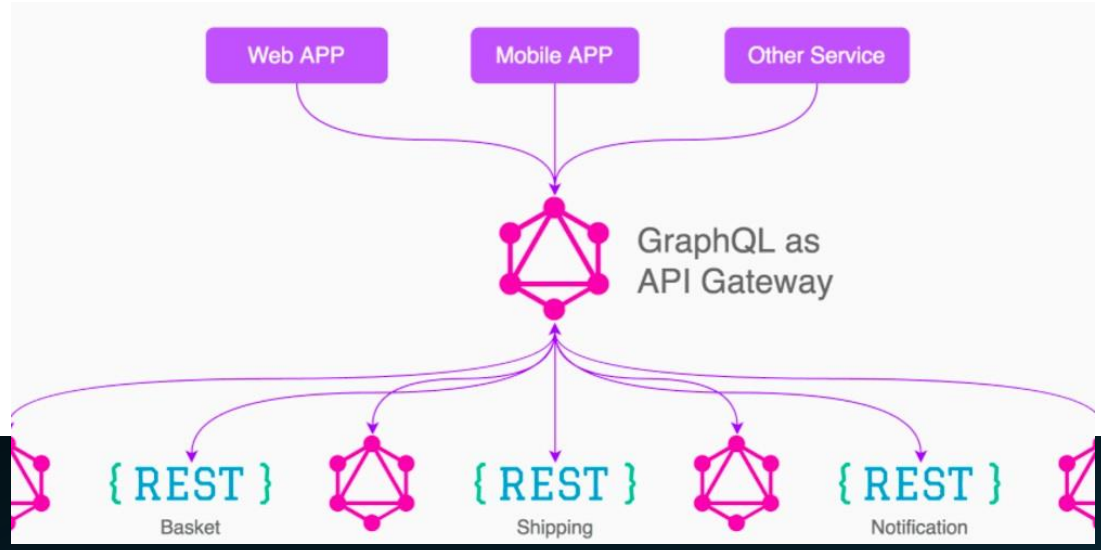
```
directive featureFlagged(name: String!)
  on FieldDefinition | TypeDefinition

type Query {
  viewer: User
}

type User {
  name: String
  secretField: SecretType @featureFlagged(name: "secret")
}

type SecretType @featureFlagged(name: "secret") {
  name: String!
}
```

Backend For Frontend

https://samnewman.io/patterns/architectural/bff/



GraphQL в качестве BFF или API Composer

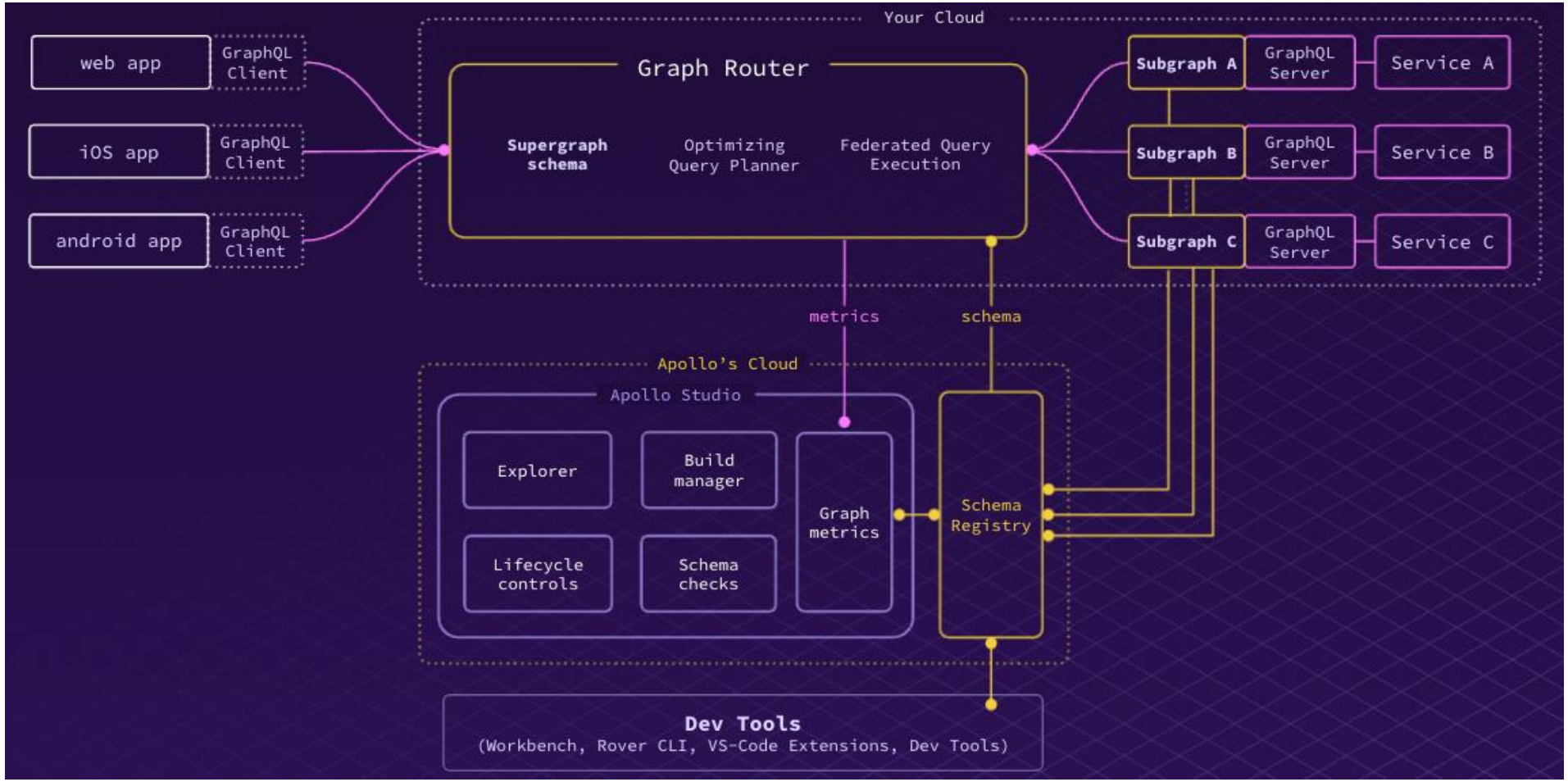


Чаще всего GraphQL выступает как прослойка или прокся между API Gateway/BFF и сервисами.

GraphQL в качестве внешнего API

- Также может быть оправдано использование GraphQL для внешнего API (когда API – отдельный продукт).
- Shopify, Github - самые яркие представители.

https://shopify.dev/api/admin-graphql#top
https://developer.github.com/v4/



GraphQL
как единый API к нескольким доменам

https://github.com/apollographql/supergraph-demo-fed2/blob/main/docs/media/supergraph.png

Инструменты: Apollo GraphQL

https://www.apollographql.com/
https://github.com/apollographql

Библиотеки (на примере C++)

- Libgraphqlparser
  A GraphQL query parser in C++ with C and C++ APIs.

- agoo-c -
  A high performance GraphQL server written in C.

  cppgraphqlgen -
  C++ GraphQL schema service generator.

- CaffQL -
  Generates C++ client types and request/response serialization from a GraphQL introspection query.

https://github.com/chentsulin/awesome-graphql

GraphQL over HTTP

Вообще протокол graphql не зависит от транспорта и мы можем передавать запросы/ответы в любом удобном нам виде.

Есть рекомендация, которой можно придерживаться: https://github.com/graphql/graphql-over-http/blob/main/spec/GraphQLOverHTTP.md

Пример

08_graphsql

Схема

```
schema {
  query: Query
}
type Query {
  author(id: Int): Author
  allAuthors: [Author]
  search(term1: String!,term2: String!): [Author!]!
}
type Author  {
  id: Int
  first_name:String!
  last_name:String!
  email:String!
  title:String!
}
query {
    search(term1:"A", term2:"B"){
        id,
        first_name,
        last_name,
        email,
        title
    }
}
```

Генерируем заготовки кода

```
schemagen -s
database/schema.graphql -p GQL
-n database
```

Делаем resolver для типов и запросов

```
struct AuthorImpl
{
int getId([[maybe_unused]] service::FieldParams
&&params) const
std::string getFirst_name([[maybe_unused]]
service::FieldParams &&params) const
std::string getLast_name([[maybe_unused]]
service::FieldParams &&params) const
std::string getEmail([[maybe_unused]]
service::FieldParams &&params) const
std::string getTitle([[maybe_unused]]
service::FieldParams &&params) const
};

struct QueryImpl
{
std::shared_ptr<Author>
getAuthor(std::optional<int> &&idArg) const;
std::vector<std::shared_ptr<Author>>
getAllAuthors() const;
std::vector<std::shared_ptr<Author>>
getSearch(std::string &&term1Arg, std::string
&&term2Arg) const;
};
```

Делаем фабрику для создания сервиса

```
std::shared_ptr<graphql::service::Request> GetService()
{
 std::shared_ptr<Query> query =
std::make_shared<Query>(std::make_shared<QueryImpl>());
 auto service = std::make_shared<Operations>(std::move(query));
 return service;
}
```

Вставляем вызов сервиса

```
std::ostream &ostr = rsp.send();
auto service = graphql::database::object::GetService();

try
{
	graphql::peg::ast query;

	std::istream_iterator<char> start{request.stream() >> std::noskipws}, end{};
	std::string input{start, end};

	query = graphql::peg::parseString(std::move(input));

	if (!query.root)
	{
		std::cerr << "Unknown error!" << std::endl;
		std::cerr << std::endl;
	}

	ostr << graphql::response::toJSON(service->resolve({query, ""}).get())
	<< std::endl;
}
catch (const std::runtime_error &ex)
{
		std::cerr << ex.what() << std::endl;
}
```

GraphQL плюсы

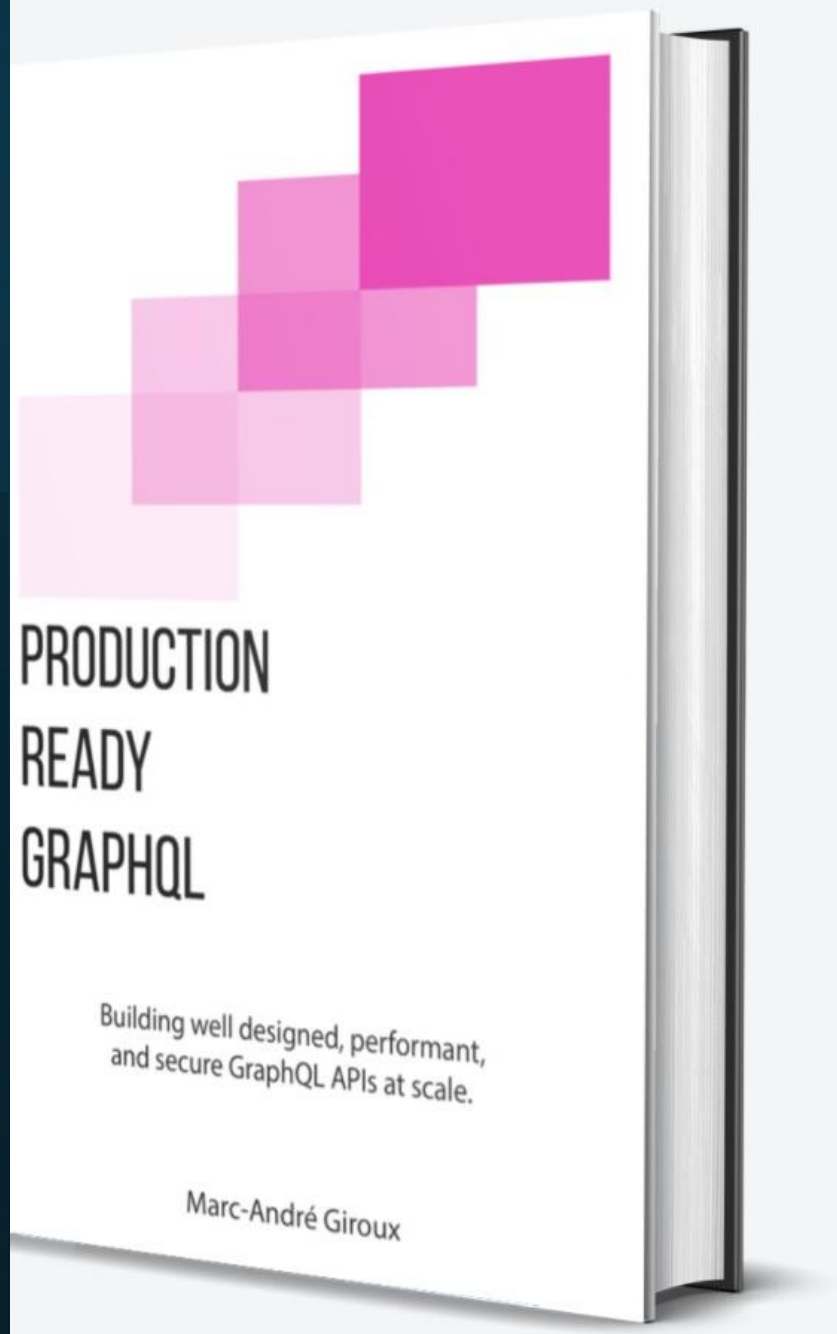
- Отсутствие недовыборки данных (как в REST)
- Хорошо сочетается с концепцией bounded context
- Повышенная наглядность
- Легче объединять данные из нескольких сервисов
- Развитие интроспекции и инструментальных средств
- Декларативная выборка данных

GraphQL минусы

- Если у нас уже есть сервисы REST, то добавление сервера GraphQL потребует больше работы на бэкенде.
- Кэширование HTTP GET не работает по умолчанию.
- Ограничение доступа и управление производительностью является более сложным для публичных API.
- Неэффективный текстовый транспорт

https://book.productionreadygraphql.com/

Production ready graphql




GRPC

Протоколы для межсервисного взаимодействия

Много сервисов – это много API

- Приходится много раз интегрироваться с разными API
- Приходится много раз писать обвязку кода
- Для одного клиентского запроса приходится много раз ходить по сети

Что привело к росту интереса к протоколам, которые

- Имеют IDL из коробки с проверкой типов и автоматической валидацией
- Избавляющие разработчиков от необходимости писать повторяющийся код и думать про транспортный уровень
- Более производительный, как по размеру передаваемых данных, так и по использованию сетевых соединений

В результате возник ряд RPC-протоколов Thrift, gRPC, Avro …

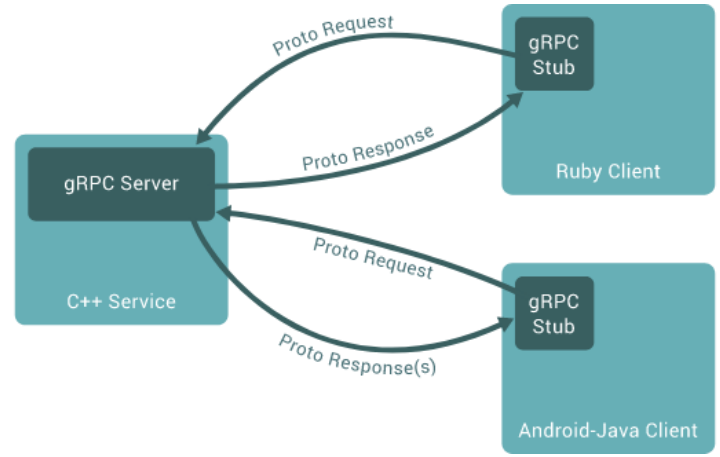
RPC

Какие виды rpc бывают?

RPC system	Transport	Language support	Notes
Sun/ONC RPC	Custom TCP or UDP	Various (Mainly C)	Early standard for RPC. Used in NFS (**N**etwork **F**ile **S**ystem). Custom "Rpcgen" IDL.
CORBA	Custom TCP	Various	Early standard for RPC and distributed objects. Custom IDL.
DCOM	Custom TCP	Various (Windows Only)	Distributed objects. Proprietary Microsoft serialization. Supports distributed garbage collection. Custom IDL.
Java RMI	Custom TCP	Java	Distributed objects. Uses Java serialization.
XML-RPC, SOAP	HTTP 1.1	Various	Specifies the communication protocol, not the client and server programming models. Uses XML for message encoding.
JSON-RPC	HTTP 1.1	Various	Specifies the communication protocol, not the client and server programming models. Uses JSON for message encoding. Technically a subset of REST.
Apache Thrift	Custom TCP	Various	Custom IDL that is similar to protobuf. No streaming.
Apache Avro	Custom TCP *or* HTTP 1.1	Various	Code generation not required. Data includes schema. IDL is formatted as JSON.
Go net/rpc	HTTP 1.1	Go	Uses Go's GOB encoding.
Cap'n Proto	Custom TCP	Various (Mainly C++)	Distributed objects. Promise pipelining. Strives for zero-overhead serialization. Custom IDL.
Twirp	HTTP 1.1	Various	No streaming. Can use JSON as encoding. Uses Protobuf as IDL.

gRPC

- В качестве транспортного протокола используется **HTTP/2**
- Используется IDL и формат данных на основе **protobuf**
- Использует кодогенерацию для stub-ов.



- RPC – remote procedure call – подход, при котором вызов другого сервиса в коде не отличим от локального вызова для разработчика

Client Code

```
// issue RPC
answer = stub.doFoo(
    bar, baz, 3.14159);
if (answer.isError()) {
    // handle failure
}
```

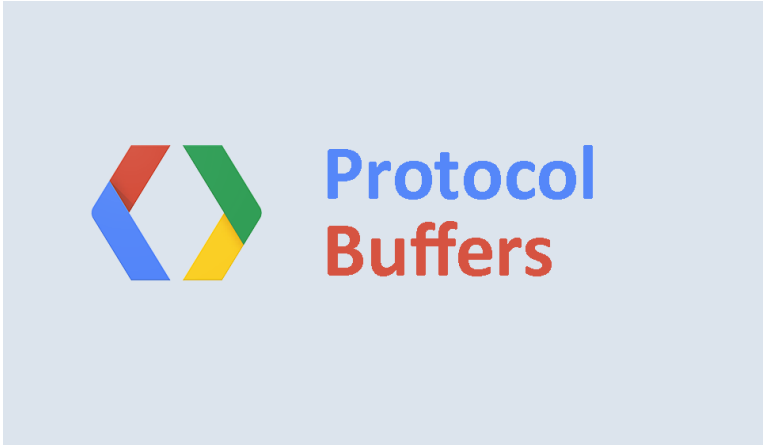
Server Code

```
class Service {
  // implements doFoo
  public FooAnswer doFoo(
      String bar, Frobnitz baz, div double) {
    // validate request
    if (!validate(bar, baz, double)) {
      return new FooAnswer(
          error("bad request!"));
    }
    // do some processing and return answer
    return new FooAnswer(
        baz.classify(bar, div));
  }
}
```

1. Client serializes `bar`, `baz`, and `3.14159` to bytes and sends on network connection to server.
2. Server serializes `FooAnswer` result and sends on network connection to client.

Protobuf

- Protocol buffers (protobuf) - это спецификация и набор библиотек
  - IDL для описания контракта взаимодействия
  - Кодогенерация клиента и сервера
- Старается быть максимально оптимизированным при передаче данных по сети.

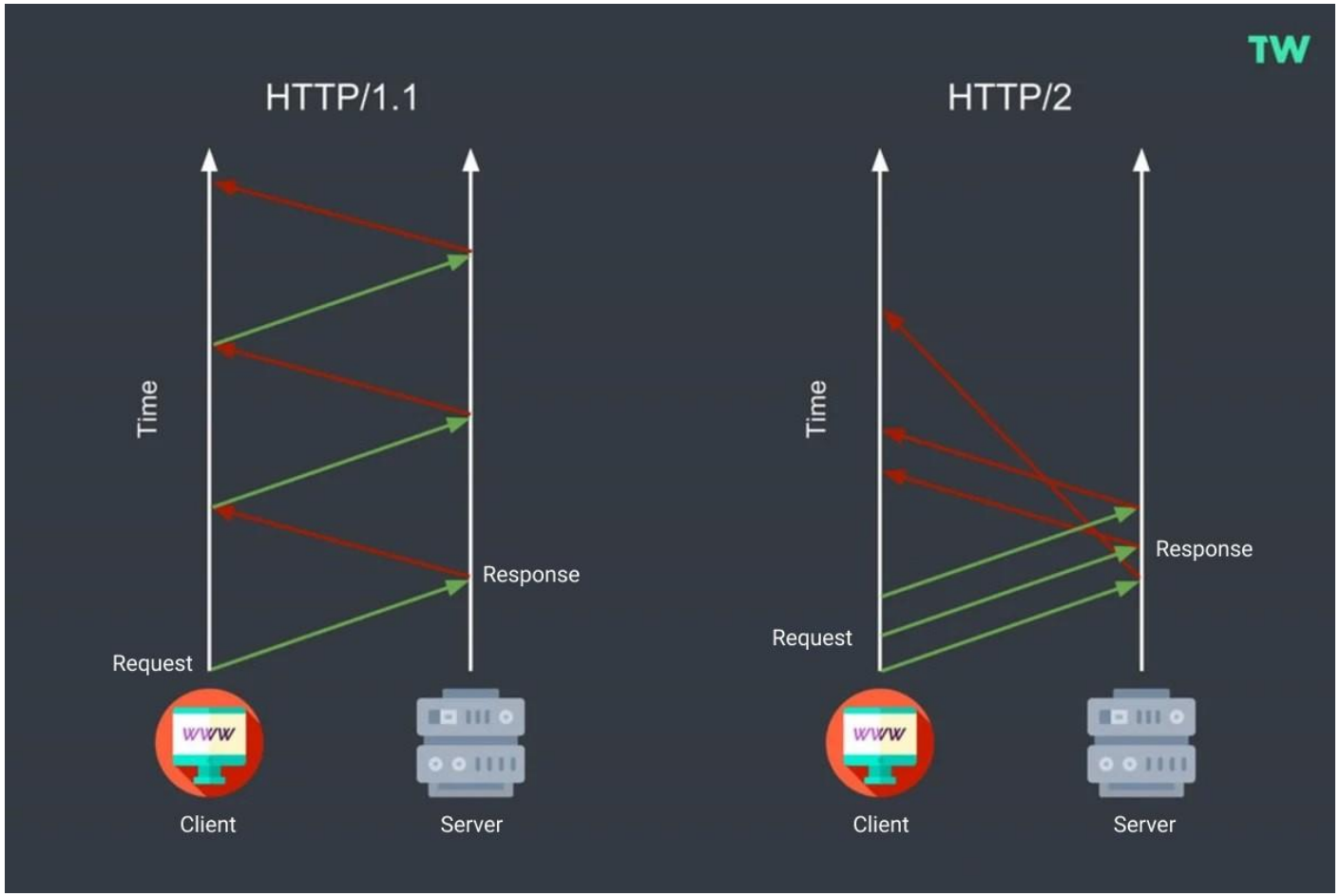


HTTP/2

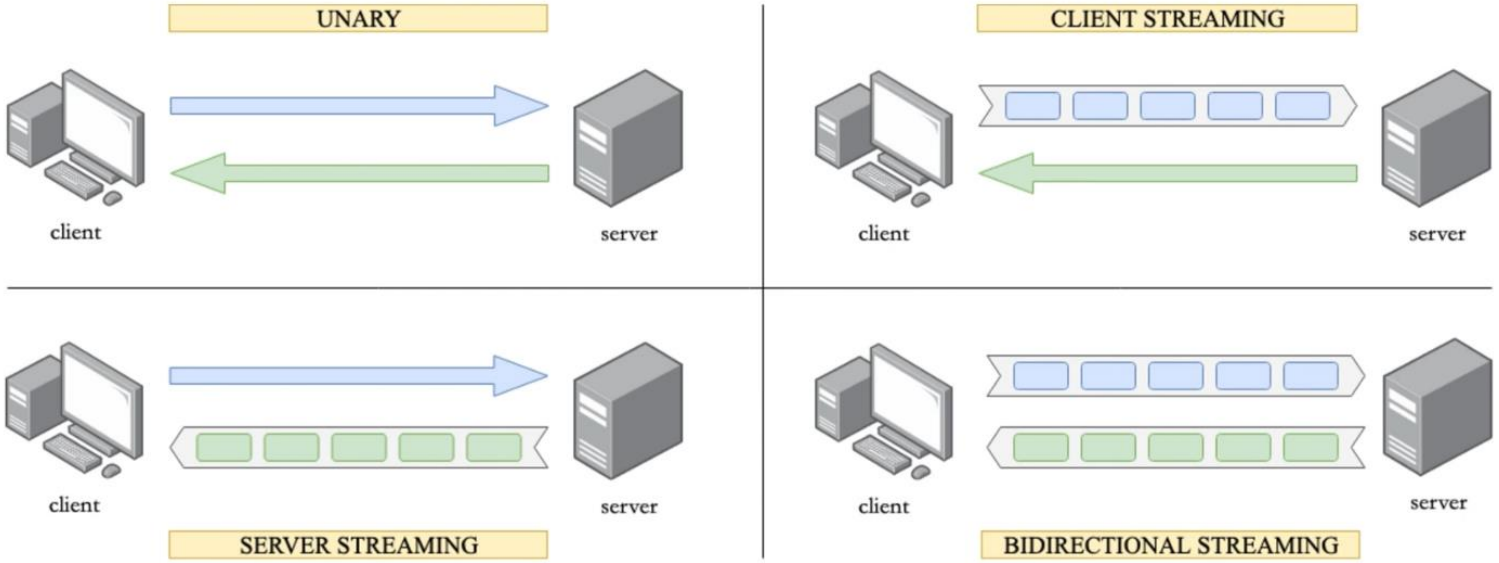
- gRPC использует HTTP/2 в качестве транспорта
- Протокол стал бинарным и появилась компрессия данных
- Появилось мультиплексирование запросов в рамках одного TCP коннекта
- Возможность push с серверной стороны
- Приоритизация ресурсов

gRPC использует HTTP 2

https://itnext.io/a-minimalist-guide-to-grpc-e4d556293422



4 типа gRPC


UNARY
client
server
CLIENT STREAMING
client
server
client
server
SERVER STREAMING
client
server
BIDIRECTIONAL STREAMING

Пример описания сервиса

https://developers.google.com/protocol-buffers

```
// ProductInfo.proto
syntax = "proto3";
package ecommerce;

service ProductInfo {
 rpc addProduct(Product) returns (ProductID);
 rpc getProduct(ProductID) returns (Product);
}

message Product {
 string id = 1;
 string name = 2;
 string description = 3;
}

message ProductID {
 string value = 1;
}
```

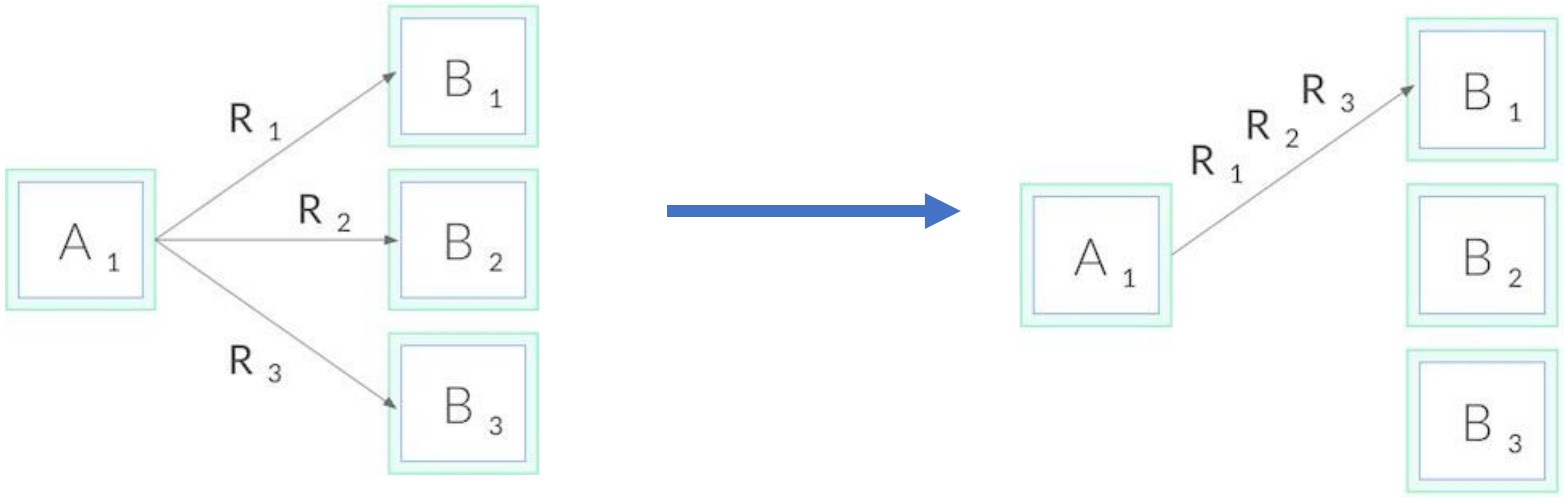
Пример gRPC

08_grpc

Балансинг в gRPC

- **Раньше:** слали запросы и потому что каждый раз создавался новый connection это запрос прилетал одному из сервисов.

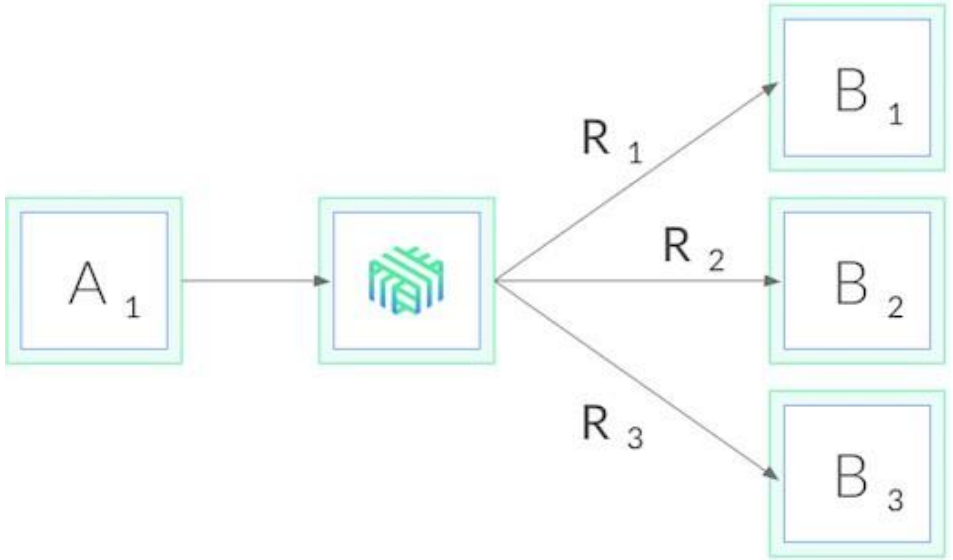
- **Теперь:** все запросы в рамках одного коннекшна идут на один и тот же инстанс.



https://kubernetes.io/blog/2018/11/07/grpc-load-balancing-on-kubernetes-without-tears/

Балансинг в gRPC

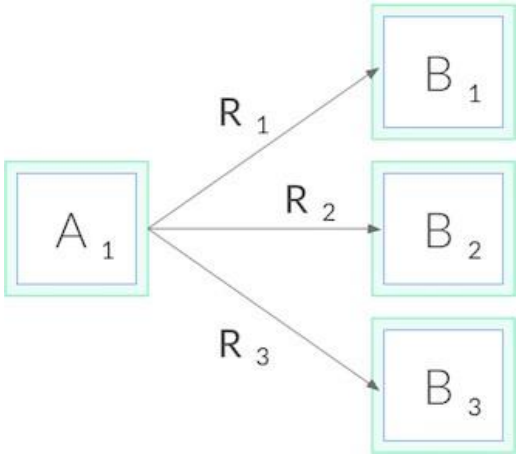
Решение1:
использовать балансинг на проксе L7 уровня, который будет разбирать протокол и понимать, что это уже другой запрос, и направлять его на другой бекенд.



https://kubernetes.io/blog/2018/11/07/grpc-load-balancing-on-kubernetes-without-tears/

Балансинг в gRPC

Решение2:
использовать client side балансинг и в grpc есть для этого xds механизм. В этом случае клиент должен знать, где находятся все бекенды и сам распределять запросы.

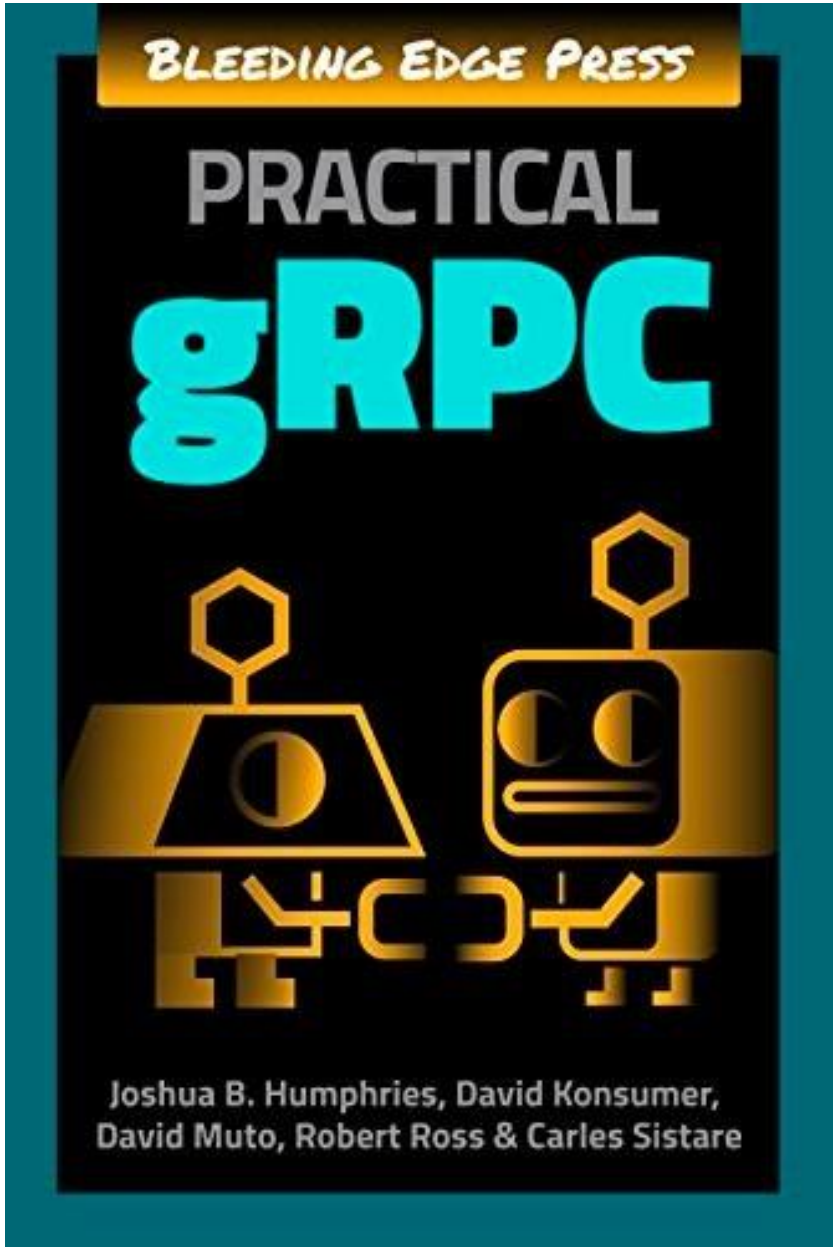


https://kubernetes.io/blog/2018/11/07/grpc-load-balancing-on-kubernetes-without-tears/

Критика gRPC

- Сложный runtime.
  В случае багов в runtime вообще непонятно, что делать.
- Есть проблемы с load balancing
- Streaming практически никому на самом деле не нужен, а его наличие значительно усложняет рантайм и создает баги.
- Бинарный протокол не сильно помогает дебагу, а инструменты для работы с grpc сложны и очень плохо развиты на данный момент
- Чаще всего время на создание соединения и передачу данных не является бутылочным горлышком.

Что почитать?



На сегодня все

ddzuba@yandex.ru