

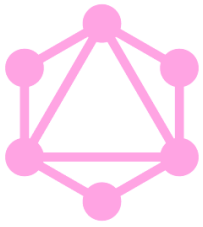
GRAPHQL КАК СПОСОБ ОРГАНИЗАЦИИ BACK-END API

ОПИСАНИЕ ПРЕЗЕНТАЦИИ

- Краткий обзор технологии GraphQL.
- Хорошо знакомые с GraphQL скорее всего не найдут ничего нового.
- Будут ссылки в виде QR-кодов.
- Картинок не будет :(

СОДЕРЖАНИЕ

1. Какие проблемы решает GraphQL и как?
2. GraphQL в микросервисном приложении.
3. Проблемы GraphQL.
4. Зрелость GraphQL и его реализаций.



QUICK FACTS

- GraphQL — спецификация, включающая в себя язык запросов, язык описания схемы и др.
- Создана в Facebook в 2012.
- Первая версия спецификации выпущена в 2015, текущая — в 2021.
- С 2019 развитием занимается некоммерческая организация GraphQL Foundation.
- Спецификация выпущена под лицензией Open Web Foundation.

НАЗНАЧЕНИЕ GRAPHQL

Решить проблемы типичного RESTful API.

ПРИМЕР RESTFUL API

API филиалов

```
GET branches/br-1

{
  "id": "br-1",
  "name": "Московский"
}
```

API отделов

```
GET departments/dep-1

{
  "id": "dep-1",
  "branch_id": "br-1",
  "name": "Консалтинг"
}
```

API сотрудников

```
GET employees/emp-1

{
  "id": "emp-1",
  "dep_id": "dep-1",
  "family": "Обломов",
  "given": "Илья",
  "patronymic": "Ильич",
  "photo": "/9j/4AAQSkZJRgABAQ..."
}
```

- API 100% нормализовано.
- На back-end оно реализуется при помощи простых операций, таких как `findById` и `findAllByIdIn`.

ЗАДАЧА

По известному ID сотрудника получить название его филиала.

1. РЕШЕНИЕ «В ЛОБ»

Выполнить 3 последовательных запроса к API.

НЕДОСТАТКИ

- Утроенное время выполнения операции.
- Код на клиенте перестаёт быть тривиальным.
- Вероятность получения несогласованных данных.

2. ДЕНОРМАЛИЗАЦИЯ API

Можно добавить в данные сотрудника соответствующие поля:

```
GET employees/emp-1?have-dep-branch-data=true
```

```
{
  "id": "emp-1",
  "dep_id": "dep-1",
  "family": "Обломов",
  "given": "Илья",
  "patronymic": "Ильич",
  "photo": "/9j/4AAQSkZJRgABAQ...",
  "dep_branch_id": "br-1",
  "dep_branch_name": "Московский"
}
```

2. ДЕНОРМАЛИЗАЦИЯ API

Можно добавить в данные сотрудника соответствующие поля:

```
GET employees/emp-1?have-dep-branch-data=true
```

```
{
  "id": "emp-1",
  "dep_id": "dep-1",
  "family": "Обломов",
  "given": "Илья",
  "patronymic": "Ильич",
  "photo": "/9j/4AAQSkZJRgABAQ...",
  "dep_branch_id": "br-1",
  "dep_branch_name": "Московский"
}
```

- А есть более общее решение?

3. ИЕРАРХИЧЕСКИЕ ПОЛЯ

Добавим информацию об отделе и филиале непосредственно в данные сотрудника:

```
GET employees/emp-1?have-dep=true&have-dep-branch=true
```

```
{
  "id": "emp-1",
  "department": {
    "dep_id": "dep-1",
    "name": "Консалтинг",
    "branch": {
      "id": "br-1",
      "name": "Московский"
    }
  },
  "family": "Обломов",
  "given": "Илья",
  "patronymic": "Ильич",
}
```

ПРОБЛЕМА ЭФФЕКТИВНОСТИ

Наш запрос возвращает много ненужных данных, включая такие «тяжёлые» поля, как photo.

- Избыточная нагрузка на сервер.
- Избыточная нагрузка на клиента.
- Избыточный трафик.

ПАРАМЕТРЫ ВЫБОРКИ

Добавим в запрос параметры, регулирующие выборку данных:

```
GET employees/emp-1?have-dep=true&have-dep-branch=true  
&have-name=false&have-photo=false&have-dep-name=false
```

```
{  
  "id": "emp-1",  
  "department": {  
    "dep_id": "dep-1",  
    "branch": {  
      "id": "br-1",  
      "name": "Московский"  
    }  
  }  
}
```

ЗАДАЧА РЕШЕНА!

Но с ростом API начинают накапливаться другие проблемы...

ПРОБЛЕМЫ НА КЛИЕНТЕ

- API стало громоздким: необходимо помнить значительный объём конфигурационных параметров.
- При добавлении новых полей в API нужно их явно отключать. Или делать новые поля отключёнными по умолчанию.

ПРОБЛЕМЫ НА СЕРВЕРЕ

- API стало громоздким: необходимо **поддерживать** значительный объём конфигурационных параметров.
- Реализация перестала быть тривиальной, помимо выборки данных добавилась логика их композиции.

НУЖНО ОБЩЕЕ РЕШЕНИЕ

- Для клиента: иметь некий специализированный язык (DSL) и с его помощью указывать, какие данные должен возвращать запрос.
- Для сервера: иметь некий framework, который взял бы на себя обработку параметров запроса, декомпозицию выборки данных и композицию результатов запроса.

GRAPHQL

Как вариант такого решения.

GRAPHQL НА КЛИЕНТЕ

GraphQL вводит язык запросов (query language):

Запрос (QL)

Ответ (JSON)

```
query {  
  employee(id: "emp-1") {  
    id  
    department {  
      branch {  
        name  
      }  
    }  
  }  
}
```

```
{  
  "id": "emp-1",  
  "department": {  
    "branch": {  
      "name": "Московский"  
    }  
  }  
}
```

- В параметрах запроса — ID сотрудника.
- А также иерархический список полей, которые необходимо вернуть.

GRAPHQL НА СЕРВЕРЕ: ФРЕЙМВОРК

GraphQL-фреймворк:

1. Анализирует запрос.
2. Разбивает его на отдельные простые операции по выборке одного или нескольких однотипных объектов.
3. Производит композицию итогового результата из отдельных простых объектов.

GRAPHQL НА СЕРВЕРЕ: РАЗРАБОТЧИК

- Разработчик реализует специальные процедуры — резольверы, `resolvers`, которые возвращают единичные объекты или коллекции однотипных объектов.
- Резольверы реализуются, как правило, простыми операциями, такими как `findById` и `findAllByIdIn`.

PROFIT!

Практически с теми же трудозатратами мы получаем API с гораздо большими возможностями по выборке данных.

КОМБИНАЦИЯ ЗАПРОСОВ...

... в одном обращении к серверу

```
query {  
  employee(id: "emp-1") {  
    id  
    departmentId  
  }  
  department(id: "dep-1") {  
    name  
  }  
  branch(id: "br-1") {  
    name  
  }  
}
```

И так далее: в нашем распоряжении
полноценный язык запросов.

ПРОМЕЖУТОЧНЫЙ ИТОГ

- Для клиента — существенное обогащение API.
- Для back-end — примерно тот же уровень трудозатрат, что в простом нормализованном RESTful API.

GRAPHQL И СИСТЕМНЫЙ АНАЛИЗ

Проектирование API тоже упростилось.

- Проектировщик объявляет типы объектов и связи между ними.
- Комбинирование объектов в запросе — забота клиента API.
- Это касается только выборки данных!
Изменение данных на сервере не сильно изменилось по сравнению, например, с JSON-RPC.

СЛЕДСТВИЕ

GraphQL хорошо работает в «классическом» web-приложении, когда 80-90% запросов — на чтение.

В некоторых случаях (например, какая-нибудь CRM-система) использование GraphQL не даёт никакой выгоды.

SDL — ЯЗЫК ОПИСАНИЯ API

Спецификация GraphQL включает в себя
schema definition language , SDL.

Это язык описания вашего API во всей его
полноте.

В мире REST неким аналогом является
спецификация OpenAPI.

SDL — ЯЗЫК ОПИСАНИЯ API

Пример простейшей схемы.

```
type Employee {           # Типы данных
  id: ID!
  name: String
  friends: [Employee!] # Связи между типами
}

type Query {               # Операции чтения данных
  getEmployee(id: ID!): Employee
}

type Mutation {           # Операции обновления данных
  addEmployee(name: String): Employee
  updateEmployee(id: ID!, name: String): Employee
  deleteEmployee(id: ID!): ID!
}
```

ИНТРОСПЕКЦИЯ

GraphQL включает в себя API по исследованию схемы со стороны клиентов:

Запрос (QL)

Ответ (JSON)

```
query {  
  __type(name: "Employee") {  
    fields {  
      name  
    }  
  }  
}
```

```
{  
  "data": {  
    "__type": {  
      "fields": [  
        {  
          "name": "id"  
        },  
        {  
          "name": "name"  
        }  
      ]  
    }  
  }  
}
```

Опять же, аналогия в REST — это OpenAPI.

ДОКУМЕНТАЦИЯ В SDL

Схема может включать в себя документацию, которая является first-class feature.

```
"""
    Тип описывает сотрудника.
    Сотрудник имеет идентификатор и имя.
"""
type Employee {
  "Идентификатор сотрудника."
  id: ID!

  "Имя сотрудника."
  name: String
}
```

МЕТАДААННЫЕ В SDL

Схема может включать в себя метаданные посредством добавления в схему директив:

```
type Employee {  
  name: String  
  numOfLogins: Int @needRole(role: "ADMIN")  
}
```

@needRole — пользовательская директива.

ВЕРСИОННОСТЬ API

Вспомним, как это делается в REST:

```
https://myapi.io/v1/employees
```

- Клиент API явно указывает версию API в запросе.
- Сервер поддерживает множественные версии, кроме совсем устаревших.

ВЕРСИОННОСТЬ В GRAPHQL

- Вместо набора фиксированных версий предлагается «continuous evolution».
- Изменения в API вносятся таким образом, чтобы не ломать обратную совместимость.

CONTINUOUS EVOLUTION I

Нельзя менять семантику и формат существующих элементов API.

В официальной документации это сформулировано как «always avoiding breaking changes».

CONTINUOUS EVOLUTION II

При добавлении в API новых элементов, существующие запросы не меняют поведения:

```
# В существующий тип...
type Employee {
  firstName: String

  # ... добавили новое поле
  givenName: String
}

# На ранее созданный запрос новое поле не влияет
query {
  employees {
    firstName
  }
}
```

CONTINUOUS EVOLUTION III

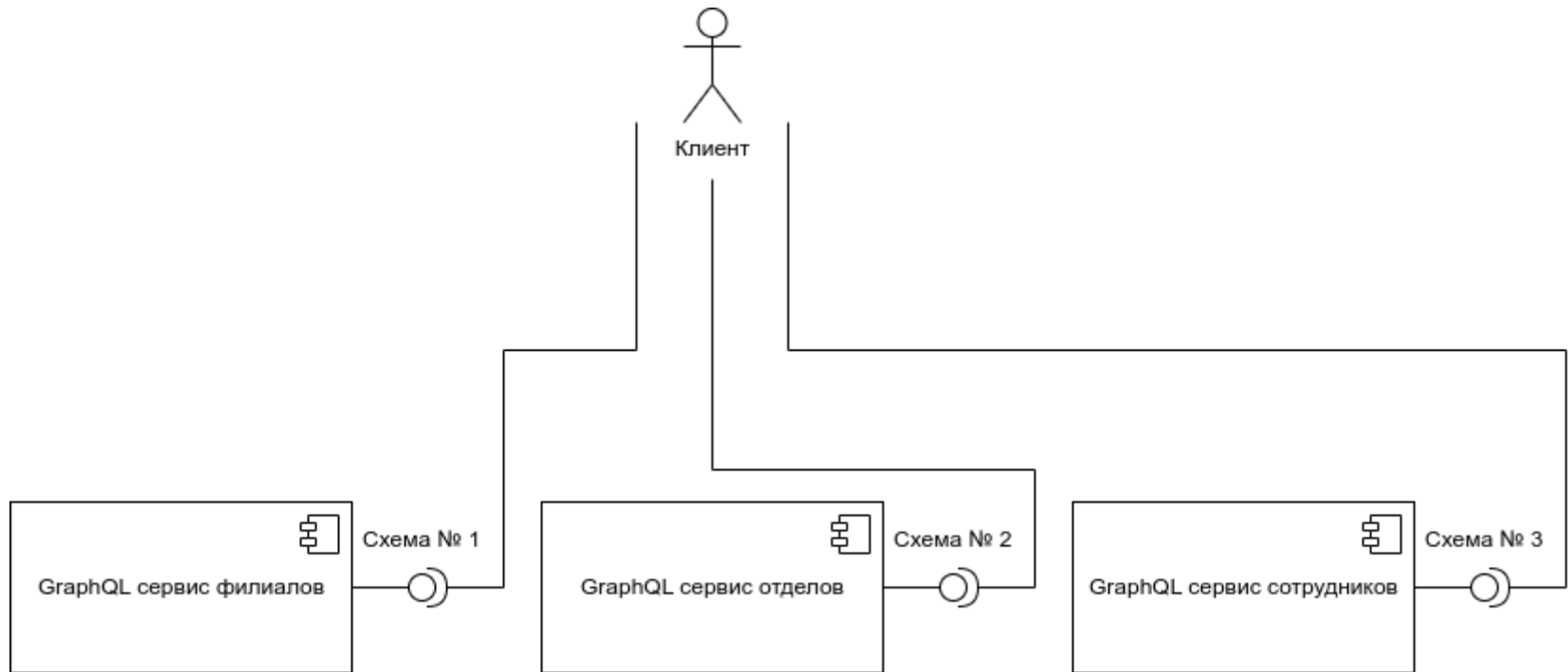
Устаревшие поля помечаются как deprecated и спустя какое-то время удаляются из схемы:

```
type Employee {  
  # Устаревшее, но пока не удалённое поле  
  firstName: String @deprecated(reason: "Use givenName")  
  
  # Новое поле  
  givenName: String  
}
```

Клиенты API должны отслеживать deprecation.

GRAPHQL В МНОГОСЕРВИСНЫХ ПРИЛОЖЕНИЯХ

ПРИМЕР ПРИЛОЖЕНИЯ



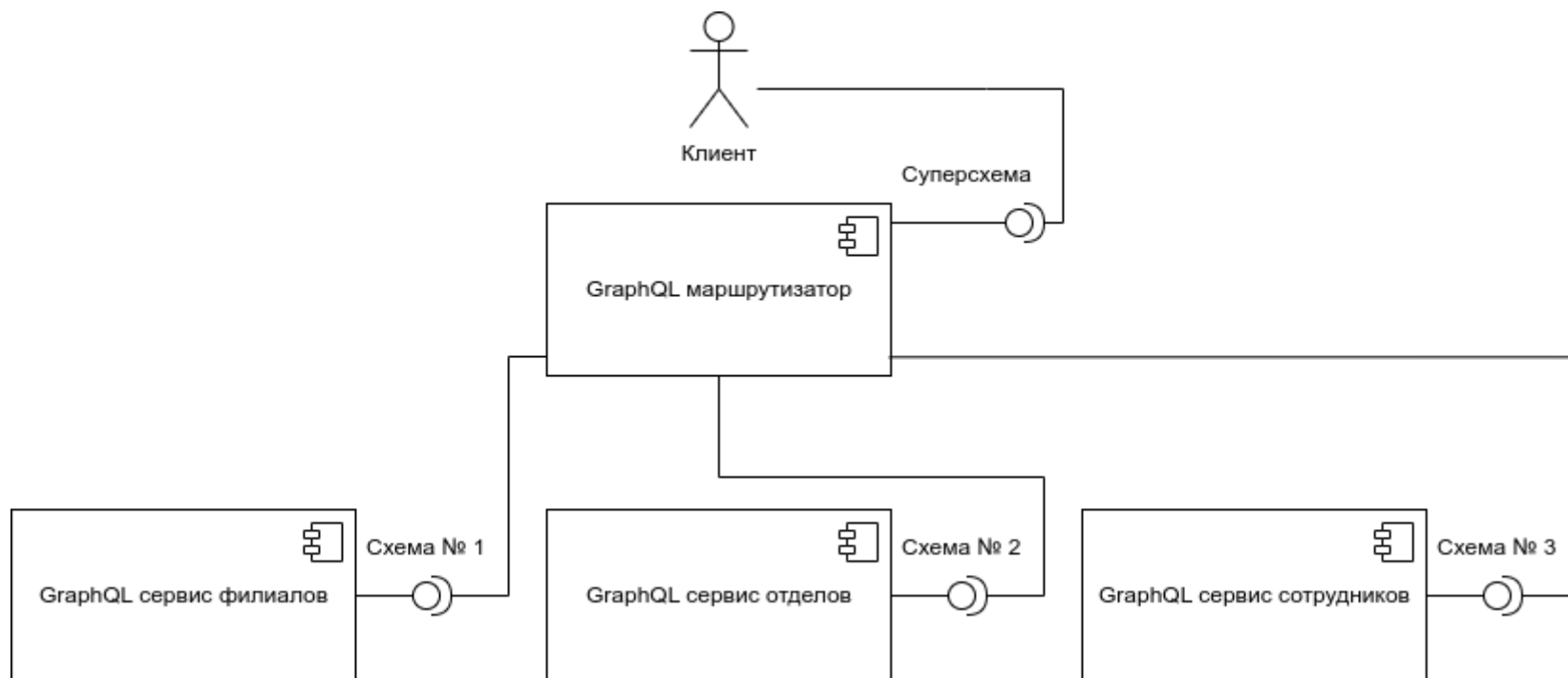
Клиент и три модуля с GraphQL API.

ПРОБЛЕМЫ

- Клиент хранит ссылки на все компоненты и их SDL-схемы и выбирает нужные в каждом отдельном запросе.
- Невозможно комбинировать запросы к разным API в одном обращении к back-end:

```
query {  
  departments { id name }  
  employees { id departmentId } # Нельзя, другое API!  
}
```

ДОБАВИМ МАРШРУТИЗАТОР



Маршрутизатор выступает как единая точка
входа в API.

УЖЕ ЛУЧШЕ

- Клиент хранит только ссылку на маршрутизатор и совокупную SDL-схему.
- Теперь можно комбинировать запросы к разным API в одном обращении к back-end:

```
query {  
  departments { id name }  
  employees { id departmentId } # Можно!  
}
```

GRAPHQL-МАРШРУТИЗАТОР

- Специализированное ПО, способное производить разбор GraphQL-запросов (обычный nginx не годится).
- Отправляет запросы нужным сервисам и объединяет их результаты в одно целое.
- Не должен вносить заметную задержку в запрос к back-end.

ВСЁ ЕЩЁ ЕСТЬ ПРОБЛЕМЫ

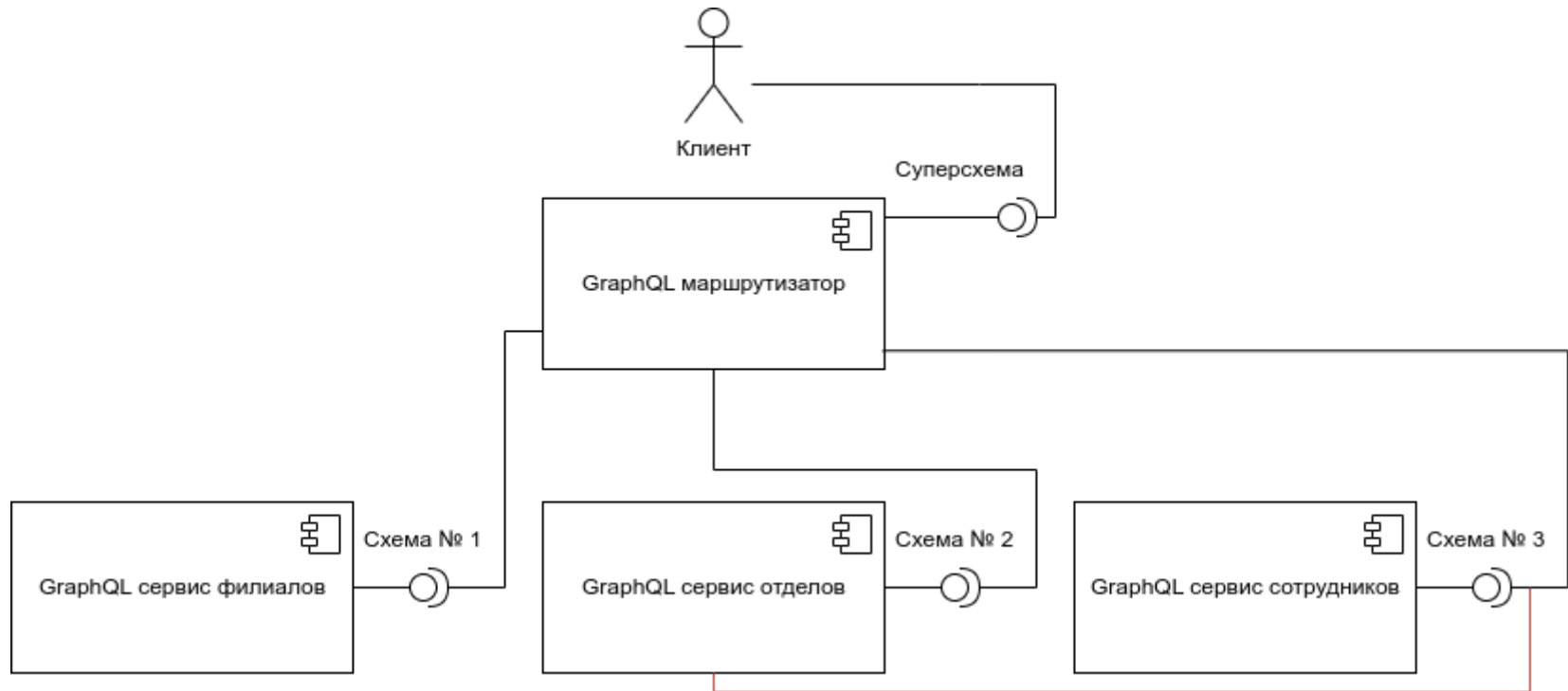
- Так плохо, нет интеграции между API:

```
query {  
  departments { id name }  
  employees { id departmentId }  
}
```

- А вот так было бы хорошо:

```
query {  
  departments {  
    id  
    name  
    employees { # Хочется получить  
      id        # список сотрудников  
    }           # прямо в отделе  
  }  
}
```

ДОБАВИМ СВЯЗЬ МЕЖДУ СЕРВИСНЫМИ МОДУЛЯМИ



Сервис отделов может обращаться непосредственно к сервису сотрудников.

СНОВА ПРОБЛЕМЫ

- Увеличивается связность между модулями и их зависимость друг от друга.
- Увеличивается сложность модулей, в них появляется GraphQL-клиент и логика композиции данных.

Есть ли общее решение?.

API FEDERATION

Интеграция нескольких поставщиков API.

APOLLO API FEDERATION V1

- Вводятся специальные SDL директивы: @key, @external и др.
- SDL-схемы сервисов модифицируются, чтобы обозначить связь между API.
- В сервисы добавляется поддержка federation-запросов (как правило, с помощью библиотеки).
- Сама связь реализуется в GraphQL-маршрутизаторе.

ПРИМЕР API FEDERATION

Модификация схемы сервиса отделов:

```
type Department @key(fields: "id") {  
  id: ID!  
  name: String!  
}
```

Директива @key определяет главный ключ сущности Department.

ПРИМЕР API FEDERATION

Модификация схемы сервиса сотрудников:

```
type Employee {  
  id: ID!  
}  
  
type Department @key(fields: "id") {  
  id: ID! @external  
  employees: [Employee!]  
}
```

- Мы импортируем тип Department и добавляем в него новое поле employees.
- Также в код сервиса добавляем резольвер для нового поля.

ИНТЕРЕСНЫЙ ЭФФЕКТ

- Сервис отделов **не знает**, что в его тип Department добавилось новое поле!

Можно добавлять функциональность к legacy-сервисам без модификации последних.

API FEDERATION

Даёт возможность по-новому организовать API
вашего приложения:

- API реализуется небольшими сервисными модулями.
- Добавляется разметка связей между API.
- Все эти детали скрыты от клиента, который имеет дело с одним совокупным API.

API FEDERATION

Новые возможности по рефакторингу приложения.

- «Как использовать GraphQL Federation для инкрементальной миграции с монолита (Python) на микросервисы (Go)» (Хабр, май 2021).

API FEDERATION

Новые возможности по рефакторингу приложения.

- «GraphQL Federation, или Как не выстрелить себе в ногу» (доклад на Highload++ Весна2021).

ПРОБЛЕМЫ GRAPHQL

...проистекают из его достоинств.

COMPLEXITY OVERHEAD

- Добавляет сложность в проект.
- Повышает требования к разработчикам.

Следствие: GraphQL может оказаться бесполезным для небольшого API с малым числом сценариев использования.

PERFORMANCE OVERHEAD

Работа с GraphQL, в основном на стороне сервера, приносит некоторые накладные расходы как по CPU, так и по ОЗУ.

ПРОБЛЕМА N+1

«Наивная» реализация резольверов в типичном GraphQL фреймворке приводит к проблеме N+1.

Решение: использовать шаблон проектирования DataLoader.

DATALOADER

Ускорение работы с помощью простых приёмов:

- Отложенная загрузка отдельного объекта (вместо объекта возвращается его Future).
- Пакетная загрузка однотипных объектов.
- Опциональное кеширование ранее загруженных объектов.

УЯЗВИМОСТЬ К DDOS-АТАКАМ

- GraphQL по своей природе, в силу наличия QL уязвим к атакам.
- Язык запросов предоставляет клиенту очень широкие полномочия.
- Злонамеренный клиент может этим воспользоваться.

ПРИМЕР DDOS АТАКИ

Рассмотрим схему:

```
type User {  
  id: ID!  
  friends: [User!]!  
}  
  
type Query {  
  users: [User!]!  
}
```

ПРИМЕР DDOS АТАКИ

Тогда следующий запрос...

```
query {  
  users {  
    friends {  
      friends {  
        friends {  
          # ... и так K раз  
        }  
      }  
    }  
  }  
}
```

... должен вернуть N^K объектов.

ОГРАНИЧЕНИЕ УРОВНЯ ВЛОЖЕННОСТИ

Реализация GraphQL может ограничивать максимальный уровень вложенности объектов в запросе.

В целом это работает плохо:

- Некоторые идиомы QL требуют значительного уровня вложенности в GraphQL-запросе.
- Часто можно подобрать «тяжёлый» запрос, не требующий большой вложенности.

ОГРАНИЧЕНИЕ ПРОДОЛЖИТЕЛЬНОСТИ

Возможно также ограничивать время работы каждого отдельного запроса.

Должна быть поддержка со стороны фреймворка, чтобы обеспечить корректное освобождение ресурсов, когда запрос принудительно останавливается.

ОГРАНИЧЕНИЕ СЛОЖНОСТИ

Выглядит перспективнее вычислять и ограничивать вычислительную «сложность» запроса.

Но как это сделать? На уровне отдельного резольвера это невозможно в силу декомпозиции.

Оценить сложность запроса можно только в процессе его выполнения — нужна поддержка со стороны фреймворка.

GRAPHQL В ПУБЛИЧНОМ API

Если ваше API предполагает доступ со стороны неопределённого круга лиц, GraphQL может оказаться не лучшим выбором.

Объём усилий по защите API может обесценить пользу от использования GraphQL.

ЗРЕЛОСТЬ

GraphQL — это спецификация.

Поэтому говоря о зрелости технологии надо
разделять

- саму спецификацию GraphQL
- и её реализации на разных платформах.

ЗРЕЛОСТЬ СПЕЦИФИКАЦИИ

В целом — production-ready.

Но некоторой функциональности не хватает.

И не устранены небольшие недоделки.

НЕТ ВЫГРУЗКИ (UPLOADING) ФАЙЛОВ

Спецификация не рассматривает данный аспект.

Приходится использовать дополнительную точку
входа или дополнительные компоненты
(например, Minio).

Некоторые реализации (например, Netflix DSG)
предлагают свои решения.

НЕТ NAMESPACES

В объёмном API, особенно при использовании API Federation это требует внимательности от проектировщиков.

INPUT FIELD DEPRECATION

Нельзя пометить поле в input-типе или аргумент в запросе как `@deprecated`.

Выглядит как банальная недоделка, пропущенная по недоразумению.

Исправлено в черновике (working draft) спецификации GraphQL, которая на момент написания в стадии prerelease.

ОТСУТСТВУЕТ RBAC

- Отсутствуют стандартные механизмы разделения доступа на основе ролей пользователей, role-based access control.
- Официальный сайт рекомендует делегировать авторизацию на уровень бизнес-логики.
- Есть разнообразные решения от независимых вендоров.

ЗРЕЛОСТЬ РЕАЛИЗАЦИЙ

- Существуют для **всех популярных платформ**.
- У них не всегда удовлетворительные стабильность, производительность и пр.

GRAPHQL-GO

- Библиотека для Golang, реализующая GraphQL сервер.
- Используется как reference implementation и в качестве основы для других GraphQL фреймворков.

GOLANG GRAPHQL БИБЛИОТЕКИ

- Другие GraphQL-фреймворки
- Есть свои плюсы и минусы в каждом из них.

APOLLO ROUTER

- Реализация GraphQL-маршрутизатора от компании Apollo.
- Реализует спецификацию Apollo Federation v2.
- Написан на Rust.
- Отправляет телеметрию в Apollo, но это (если верить документации) отключаемо.

GraphQL API ШЛЮЗЫ

Cloud Native Landscape — Mozilla Firefox

Cloud Native Landscape x +

https://landscape.cncf.io/card-mode?category=api-gateway 110%


CNCF Cloud Native Interactive Landscape

The cloud native landscape (png, pdf), serverless landscape (png, pdf), and member landscape (png, pdf) are dynamically generated below. Please open a pull request to correct any issues. Greyed logos are not open source. Last Updated: 2023-01-16T14:21:42.

You are viewing 19 cards with a total of 75,900 stars, market cap of \$2.6T and funding of \$579.1M.

Landscape Card Mode Members Serverless Wasm


CNCF Incubating Projects (1)



EMISSARY INGRESS

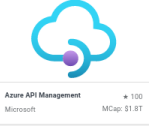
Emissary Ingress ★ 3,375
Cloud Native Computing Foundation (CNCF) Funding: \$3M

CNCF Member Products/Projects (11)



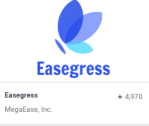
3SCALE

3Scale ★ 281
Red Hat MCAP: \$130.8B



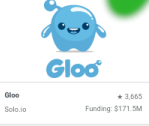
Azure API Management

Microsoft ★ 100
MCAP: \$1.8T



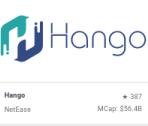
Easegress

MegaEase, Inc. ★ 4,970




Gloo

Solo.io ★ 3,665
Funding: \$171.5M




Hango

NetEase ★ 387
MCAP: \$56.4B



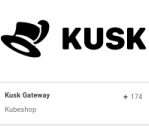
Higress

Alibaba Cloud ★ 790
MCAP: \$304.8B




Kong

Kong ★ 33,809
Funding: \$169.1M



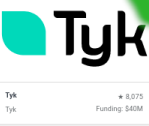
KUSK

Kusk Gateway ★ 174
Kubeshop




MuleSoft

Salesforce ★ 340
MCAP: \$149.5B



Tyk

Tyk ★ 8,075
Funding: \$40M



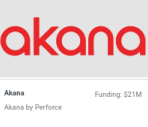
WSO2 API Microgateway

WSO2 API Microgateway ★ 233
WSO2 Funding: \$133.9M

Example filters


- Cards by age
- Open source landscape
- Member cards
- Cards by stars
- Cards from China
- Certified K8s/KCSP/KTP
- Cards by MCAP/Funding
- Cards without
- bestpractices.dev

Non-CNCF Member Products/Projects (7)




akana

Akana ★ 221M
Funding: \$21M




APIOAK

APIOAK ★ 384




APISIX

The Apache Software Foundation ★ 11,053




saaras.io

EnRoute OneStep Ingress ★ 177
Saras




GRAVITEE.io

Gravitee ★ 1,558
Funding: \$41M



krakenD

KrakenD ★ 5,377



Reactive Interaction Gateway

Accenture ★ 352
MCAP: \$177.8B

Crunchbase data is used under license from Crunchbase to CNCF. For more information, please see the [license](#) info.

СПАСИБО ЗА ВНИМАНИЕ!

