

Взгляд на GraphQL Afisha Daily

Рассмотрим плюсы и минусы, после
использования в production

Никита Ларионов (17.04.2024)

Оглавление

- Краткий обзор GraphQL
- SDL и обзор его возможностей
- Известные недостатки
- Причины выбора
- Apollo Client
- А что на сервере?
- Инструменты которые используем
- Приобретенный опыт и проблемы
- Рекомендации
- Полезная инфа и ссылки

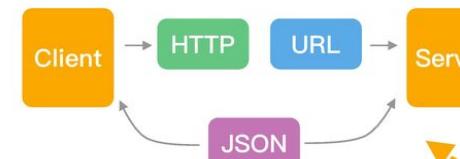


API Protocols

● Webhooks

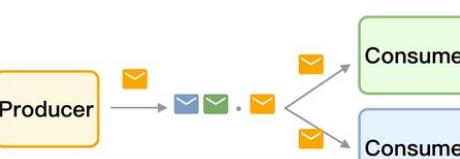
● REST

REST is an architectural style for designing networked applications, using stateless communication and standard HTTP methods



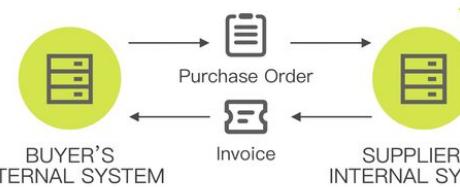
● EDA

Event-Driven Architecture (EDA) is a trending software architecture pattern nowadays



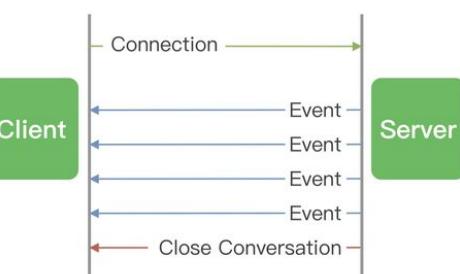
● EDI

EDI (Electronic Data Interchange) is a set of standards for exchanging structured business data between organizations electronically without human intervention



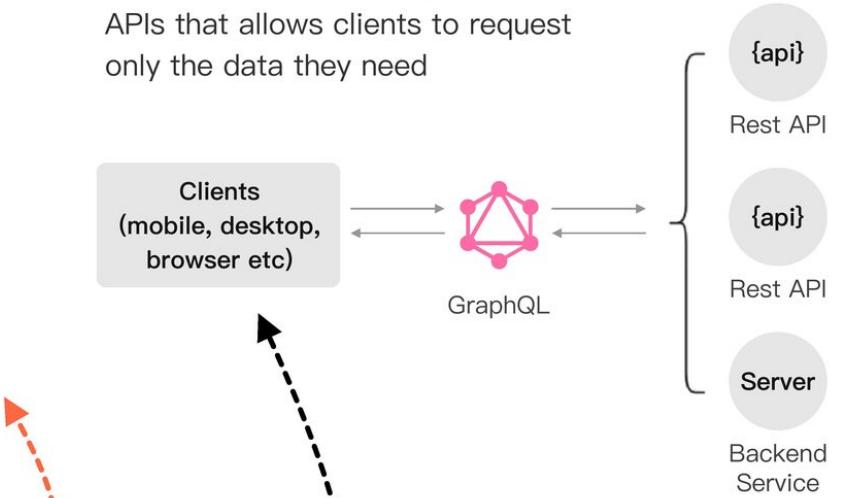
● SSE

SSE (Server-Sent Events) is a simple and efficient standard for server-push notifications over an HTTP connection



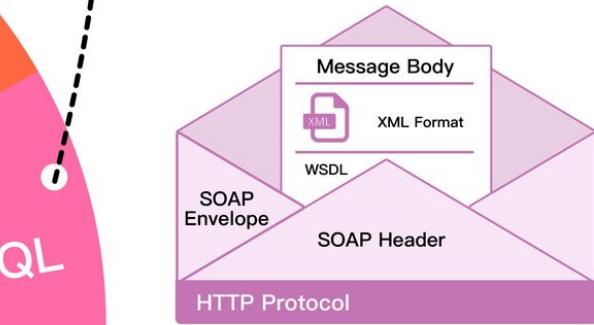
● GraphQL

GraphQL is a query language for APIs that allows clients to request only the data they need



● SOAP

SOAP is a protocol for exchanging structured information using XML



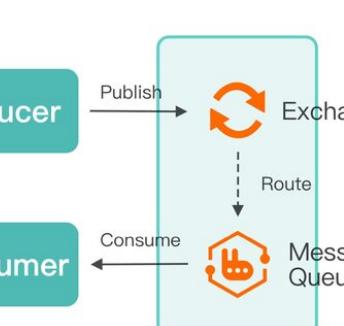
● WebSocket

WebSockets provide a full-duplex communication channel over a single, long-lived connection, allowing for real-time data exchange



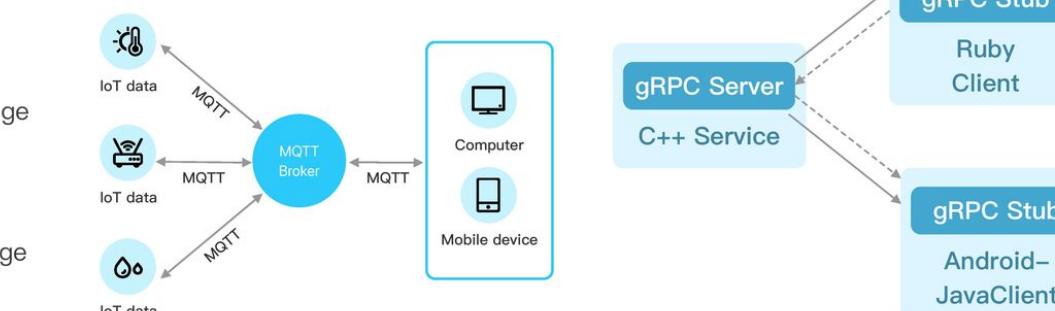
● AMQP

AMQP is an open-standard protocol for message-oriented middleware, facilitating message routing, queuing, and delivery



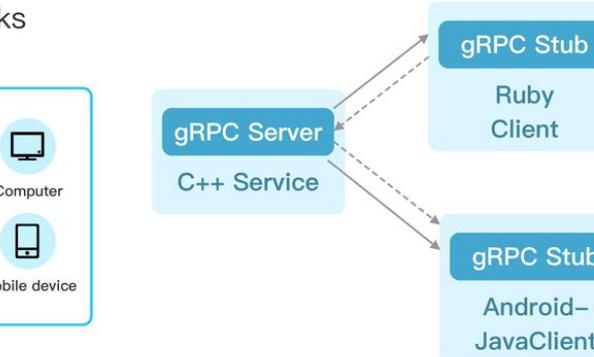
● MQTT

MQTT is a lightweight publish-subscribe messaging protocol designed for low-bandwidth, high-latency, or unreliable networks



● gRPC

gRPC is a high-performance, open-source framework for RPCs using Protocol Buffers



Краткий обзор GraphQL



Краткий обзор GraphQL

Что? зачем? почему?

- Это альтернативный способ взаимодействия клиента с сервером посредством HTTP, дополнение или альтернатива к REST API
- Поддерживается большинством языков программирования
- Упрощает взаимодействие с сервером при работе с множеством полей, фильтров и агрегатов
- Предоставляет более широкие возможности выбора данных
- Позволяет использовать единую точку входа для получения данных
- Знакомые ручки теперь становятся функциями с параметрами в запросе
- Имеет множество реализаций, таких как *Relay*, *Apollo* и другие

Краткий обзор GraphQL

Плюсы в студию!

- Гибкость запросов
- Единая точка входа
- Авто документация, кодогенерация, интроспекция
- Меньшее количество запросов - один запрос, много ресурсов
- Легкое добавление новых полей и переиспользование
- Сильная типизация, ввод -> вывод
- Вид возвращаемых данных определяется исключительно запросом с клиента, что обеспечивает независимость от версий

Краткий обзор GraphQL

Гибкость запросов

- Возможно кто-то сталкивался с ситуацией в API, когда нужно запросить 15-1000 полей?
- Пример запроса в [Blizzard Entertainment API \(WoW\)](#)
- Пример запроса в [HeadHunter API](#)

Краткий обзор GraphQL

Гибкость запросов - неудобные API
Blizzard Entertainment (WoW)

```
1  const axios = require('axios');
2
3  const realm = 'realmName';
4  const characterName = 'characterName';
5
6  const apiKey = 'yourApiKey';
7  const locale = 'en_US';
8
9
10 const fields = [
11   'achievements', 'appearance', 'reputation', 'quests',
12   'stats', 'titles', 'talents', 'professions',
13   'petSlots', 'pets', 'mounts', 'status',
14   'pvp', 'progression', 'feed', 'guild',
15   'hunterPets', 'overview', 'stats'
16 ];
17
18 const filters = [
19   { name: 'guild', params: ['achievements'] },
20   { name: 'pvp', params: ['ratings'] },
21   { name: 'achievements', params: ['statistics'] },
22   { name: 'quests', params: ['recent'] },
23   { name: 'progression', params: ['raids'] },
24   { name: 'talents', params: ['selected'] },
25   { name: 'items', params: ['equipped'] },
26   { name: 'reputation', params: ['standing'] },
27   { name: 'titles', params: ['selected'] },
28   { name: 'statistics', params: ['category'] },
29   { name: 'audit', params: ['profile'] }
30 ];
31
32 const url = `https://us.api.blizzard.com/wow/character/${realm}/${characterName}
33 ?fields=${fields.join('%2C')}
34 &filters=${filters.map(f => `${f.name}(${f.params.join(',')})`).join(',')}`
35 &locale=${locale}&apikey=${apiKey}`;
36
37 axios.get(url)
38 .then(response => {
39   console.log(response.data);
40 })
41 .catch(error => {
42   console.error(error);
43 });
44
```

Краткий обзор GraphQL

Гибкость запросов - неудобные API

Blizzard Entertainment (WoW)

- А если нужно фильтровать еще и поля?
- Blizzard разработали способ фильтрации полей в REST API
- Они предлагают добавлять точку к полю запроса конкретных значений
- Например `stats.agility` - для запроса значения ловкости персонажа
- Гибко, но не всегда удобно

Краткий обзор GraphQL

Ладно пойдем дальше...

Есть еще неудобные API? Полно!

- В веб разработке, особенно у ресурсов с огромным количеством данных, есть множество справочников в API
- Например рассмотрим HeadHunter, известный сервис поиска работы
- Проблема его API, в том, что все `related` сущности надо тянуть отдельными запросами
- Вы сначала запрашиваете одну сущность, получаете ее `ID`, потом запрашиваете следующую сущность по этому `ID`
- Учитывая этот момент, желание написать парсер вакансий у вас быстро пропадет...

Краткий обзор GraphQL

Гибкость запросов

HeadHunter API

Чтобы получить вакансии, потом вакансию, а далее информацию о работодателе, нам нужно будет три функции и вызывать их нужно будет последовательно

```
1
2
3 async function getVacancies() {
4   try {
5     const response = await axios.get('https://api.hh.ru/vacancies', {
6       params: {
7         text: 'Data Scientist',
8         area: 1 // ID города Москва
9       }
10    });
11    return response.data;
12  } catch (error) {
13    console.error('Ошибка при получении списка вакансий:', error);
14    return null;
15  }
16}
17
18 // Пример запроса для получения подробной информации о вакансии по ее ID
19 async function getVacancyDetails(vacancyId) {
20   try {
21     const response = await axios.get(`https://api.hh.ru/vacancies/${vacancyId}`);
22     return response.data;
23   } catch (error) {
24     console.error('Ошибка при получении подробной информации о вакансии:', error);
25     return null;
26   }
27}
28
29 // Пример запроса для получения информации о работодателе по его ID
30 async function getEmployerDetails(employerId) {
31   try {
32     const response = await axios.get(`https://api.hh.ru/employers/${employerId}`);
33     return response.data;
34   } catch (error) {
35     console.error('Ошибка при получении информации о работодателе:', error);
36     return null;
37   }
38}
```

Краткий обзор GraphQL

Гибкость запросов

HeadHunter API

```
32
33
34 // Пример использования функций
35 async function main() {
36   const vacancies = await getVacancies();
37   if (vacancies) {
38     const firstVacancyId = vacancies.items[0].id; // Предположим, что выбираем первую вакансию из списка
39     const vacancyDetails = await getVacancyDetails(firstVacancyId);
40     console.log('Подробная информация о вакансии:', vacancyDetails);
41   }
42 }
43
44 main();
45
```

Просто запросили три объекта. А если наша цепочка состоит из 5-10 шагов?

Можно создать хранилище, связывать объекты между собой и получить нормализованный Redux...

Конечно, есть [OpenAPI Generator](#) для генерации клиента, но это не делает API удобнее

Краткий обзор GraphQL

Гибкость запросов - Query (НН)

А вот так будет выглядеть запрос в [НН.ru](#)
Если представить, что у него есть GraphQL API

```
1
2  < query {
3    < vacancies(text: "Data Scientist", area: 1) {
4      < items {
5        id
6        name
7        description
8        employer {
9          id
10         name
11         description
12         website
13       }
14     }
15   }
16 }
```

Краткий обзор GraphQL

Гибкость запросов - Query (WoW)

Вернемся к Blizzard :)
Представим, что они
тоже у себя запилили GraphQL

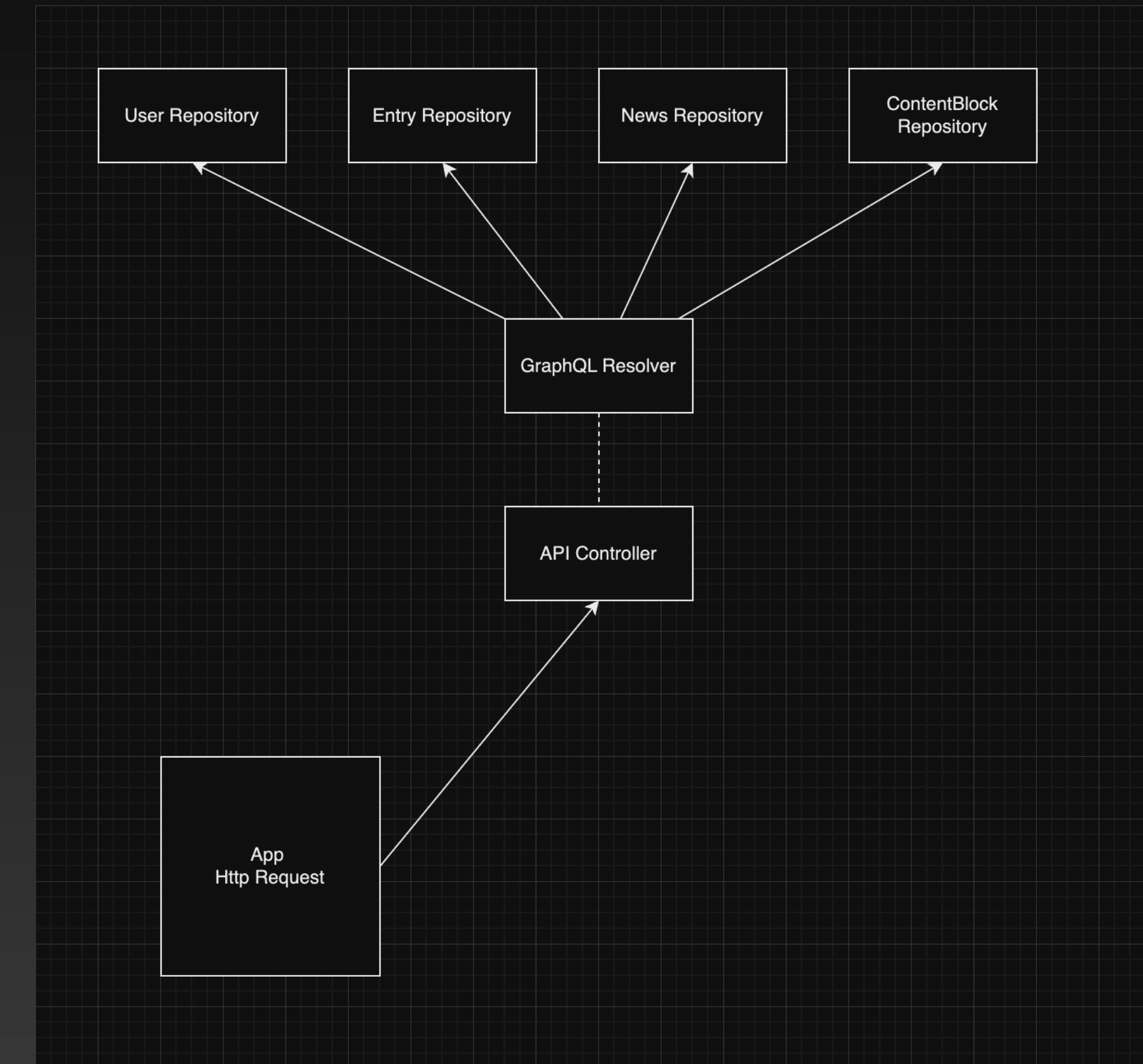
```
1
2 < query ($includeTalents: Boolean!) {
3   < character(realm: "realmName", name: "characterName") {
4     < achievements(filter: { category: "general", completed: true }) {
5       id
6       name
7       points
8       description
9     }
10
11   < talents @include(if: $includeTalents) {
12     selected
13     tree
14     < talents {
15       name
16       description
17     }
18   }
19
20   < stats {
21     health
22     power
23     speed
24   }
25 }
26 }
27 }
```

Краткий обзор GraphQL

Единая точка входа

Любой http запрос с query
попадает в одну точку и может запрашивать
данные из разных источников

Нашему беку,
нужно только определить
для этого [Resolvers](#). Это достаточно быстро!



Краткий обзор GraphQL

Авто документация

- Концепция GraphQL строится на собственном SDL ([Schema Definition Language](#))
- Данный язык имеет свой синтаксис для описания схемы на сервере
- Схема определяет данные и возможные над ними операции
- Включая [скалярные типы](#), [enum](#), [unions](#), [interfaces](#), [модификаторы типов](#)
- Имеется отдельный [Input Type](#) - он нужен для объединения параметров
- Все выше перечисленное позволяет автоматический генерировать документацию, помечать определенные поля на сервере как `deprecated` и оставлять к ним комментарии
- SDL дает нам широкие возможности для различной [кодогенерации](#)

Краткий обзор GraphQL

Меньшее количество запросов

- В одной query - мы можем запросить несколько полей, вызвать несколько функций (поля с параметрами, то что мы привыкли называть в REST-API умной ручкой)
- Другими словами GraphQL позволяет объединять несколько запросов в один http запрос
- Следовательно мы делаем меньше запросов к серверу, передавая всю информацию внутри тела запроса, по сути количество данных и вложенных полей не ограничено
- В формате отправляемых данных технология нас тоже никак не ограничивает
- Все включается, к чему мы привыкли **base64**, **multi-part-data**, **файлы**
- Сокращение запросов к серверу и количество передаваемых байтов, существенное :)

Краткий обзор GraphQL

Легкое добавление новых полей

- На стороне сервера для добавления нового поля достаточно поправить пару строк, чтобы изменить **схему** и выкатить новую версию API
- На стороне клиента нужно **перегенерить код** из новой схемы и в сам запрос добавить поле
- Чтобы не дублировать поля, GraphQL предлагает нам **Fragments**
- **Фрагменты** позволяют использовать кусочки запросов полей во внутри других query
- Таким образом в структуре проекта всегда есть две отдельные папки **fragments** и **queries**

GQL SDL обзор возможностей



GQL SDL обзор возможностей

Базовые элементы



GQL SDL обзор возможностей

Базовые элементы - Query

- Query - это по сути тело запроса + переменные, запрос выполняется поверх HTTP методов
- Для каждой query, мы обязаны задать уникальное название (**Operation Name**)
- Передать словарь с переменными (**Variables**), которые доступны в контексте query
- Переменные в HTTP передаются просто словарем рядом с запросом
- Внутри запроса указываем названия моделей, ручек
- Можно передать параметры к операциям из variables или указать значения хардкодом
- У каждого **Field** есть свои **мета данные**

GQL SDL обзор возможностей

Базовые элементы - Query

Пример как query выглядит в теле HTTP запроса

```
{  
  hero {  
    name  
  }  
}
```

```
{  
  "data": {  
    "hero": {  
      "name": "R2-D2"  
    }  
  }  
}
```

GQL SDL обзор возможностей

Базовые элементы - Query params

Передача **параметров** в ручку

```
{  
  human(id: "1000") {  
    name  
    height  
  }  
}
```

```
{  
  "data": {  
    "human": {  
      "name": "Luke Skywalker",  
      "height": 1.72  
    }  
  }  
}
```

GQL SDL обзор возможностей

Базовые элементы - Query params

Передача **параметров** в отдельное поле

```
{  
  human(id: "1000") {  
    name  
    height(unit: FOOT)  
  }  
}
```

```
{  
  "data": {  
    "human": {  
      "name": "Luke Skywalker",  
      "height": 5.6430448  
    }  
  }  
}
```

GQL SDL обзор возможностей

Базовые элементы - Query Alias

Aliases - как переиспользовать параметизированные query

```
{  
    empireHero: hero(episode: EMPIRE) {  
        name  
    }  
    jediHero: hero(episode: JEDI) {  
        name  
    }  
}
```

```
{  
    "data": {  
        "empireHero": {  
            "name": "Luke Skywalker"  
        },  
        "jediHero": {  
            "name": "R2-D2"  
        }  
    }  
}
```

GQL SDL обзор возможностей

Базовые элементы - Query Fragments

Fragments - переиспользование выборки полей

```
{  
  leftComparison: hero(episode: EMPIRE)  
    ...comparisonFields  
  }  
  rightComparison: hero(episode: JEDI) ·  
    ...comparisonFields  
}  
  
fragment comparisonFields on Character {  
  name  
  appearsIn  
  friends {  
    name  
  }  
}
```

```
{  
  "data": {  
    "leftComparison": {  
      "name": "Luke Skywalker",  
      "appearsIn": [  
        "NEWHOPE",  
        "EMPIRE",  
        "JEDI"  
      ],  
      "friends": [  
        {  
          "name": "Han Solo"  
        },  
        {  
          "name": "Leia Organa"  
        },  
        {  
          "name": "C-3PO"  
        },  
        {  
          "name": "BB-8"  
        }  
      ]  
    }  
  }  
}
```

GQL SDL обзор возможностей

Базовые элементы - Query Fragments

Fragments - использование переменных

```
query HeroComparison($first: Int = 3) {  
    leftComparison: hero(episode: EMPIRE)  
        ...comparisonFields  
    }  
    rightComparison: hero(episode: JEDI) .  
        ...comparisonFields  
}  
  
fragment comparisonFields on Character .  
    name  
    friendsConnection(first: $first) {  
        totalCount  
        edges {  
            node {  
                name  
            }  
        }  
    }  
},  
{  
    "data": {  
        "leftComparison": {  
            "name": "Luke Skywalker",  
            "friendsConnection": {  
                "totalCount": 4,  
                "edges": [  
                    {  
                        "node": {  
                            "name": "Han Solo"  
                        }  
                    },  
                    {  
                        "node": {  
                            "name": "Leia Organa"  
                        }  
                    },  
                    {  
                        "node": {  
                            "name": "C-3PO"  
                        }  
                    }  
                ]  
            },  
            "name": "Luke Skywalker"  
        },  
        "rightComparison": {  
            "name": "Darth Vader",  
            "friendsConnection": {  
                "totalCount": 4,  
                "edges": [  
                    {  
                        "node": {  
                            "name": "Han Solo"  
                        }  
                    },  
                    {  
                        "node": {  
                            "name": "Leia Organa"  
                        }  
                    },  
                    {  
                        "node": {  
                            "name": "C-3PO"  
                        }  
                    }  
                ]  
            },  
            "name": "Darth Vader"  
        }  
    },  
    "errors": []  
}
```

GQL SDL обзор возможностей

Базовые элементы - Fragments

Fragments позволяют связываться
на определенные типы
Droid, Human

```
query HeroForEpisode($ep: Episode!) {  
  hero(episode: $ep) {  
    name  
    ... on Droid {  
      primaryFunction  
    }  
    ... on Human {  
      height  
    }  
  }  
}
```

```
{  
  "ep": "JEDI"  
}
```

```
{  
  "data": {  
    "hero": {  
      "name": "R2-D2",  
      "primaryFunction": "Astromech"  
    }  
  }  
}
```

GQL SDL обзор возможностей

Базовые элементы - Query

Подробнее про переменные

```
query HeroNameAndFriends($episode: Episode!) {
  hero(episode: $episode) {
    name
    friends {
      name
    }
  }
}

"episode": "JEDI"
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

GQL SDL обзор возможностей

Базовые элементы - Query

Дефолтные значения у переменных

```
query HeroNameAndFriends($episode: Episode = JEDI) {  
  hero(episode: $episode) {  
    name  
    friends {  
      name  
    }  
  }  
}
```

GQL SDL обзор возможностей

Базовые элементы - Directives

Директивы: получаем поле только если...

```
query Hero($episode: Episode, $withFriends: Boolean!) {  
  hero(episode: $episode) {  
    name  
    friends @include(if: $withFriends) {  
      name  
    }  
  }  
}  
  
{  
  "episode": "JEDI",  
  "withFriends": false  
}
```



GQL SDL обзор возможностей

Базовые элементы - Mutation

Мутации - изменяем данные на сервере

```
mutation CreateReviewForEpisode($ep: Episode!, $review: CreateReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}

{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```

GQL SDL обзор возможностей

Базовые элементы - Meta Fields

Мета данные

Позволяют завязываться на определенный подтип сущности

```
{  
  search(text: "an") {  
    __typename  
    ... on Human {  
      name  
    }  
    ... on Droid {  
      name  
    }  
    ... on Starship {  
      name  
    }  
  }  
}
```

```
{  
  "data": {  
    "search": [  
      {  
        "__typename": "Human",  
        "name": "Han Solo"  
      },  
      {  
        "__typename": "Human",  
        "name": "Leia Organa"  
      },  
      {  
        "__typename": "Starship",  
        "name": "TIE Advanced x1"  
      }  
    ]  
  }  
}
```

GQL SDL обзор возможностей

Type concept - Введение



GQL SDL обзор возможностей

Type concept - Введение

- Система типов в GraphQL SDL - похожа на Typescript
- Основной упор в системе - описание объектов, обязательности полей
- Тип описания аргументов - нужно воспринимать как ordered list
- Query и Mutations - объявляются отдельно, соответственно различны
- Для описания примитивных полей - используются скалярные типы
- Доступны для использования Enum - перечисления
- Коллекции - списки, описываются скобочками []
- Интерфейсы используются для описания агрегатов и моделей
- Объединения (Union) - аналогичны по сути с Typescript
- Input Types - позволяют описать параметры в виде объектов
- Обязательность полей или коллекций описывается модификаторами ! И !!

GQL SDL обзор возможностей

Type concept - Скалярные типы

- GraphQL `Int` - целое число (32 бита)
- GraphQL `Float` - число с плавающей точкой (64 бита)
- GraphQL `String` - строка в формате UTF-8
- GraphQL `Boolean` - true/false
- GraphQL `ID` - строка

GQL SDL обзор возможностей

Type concept - Базовые элементы

Схема в основном описывается с помощью
Type и Interfaces

```
interface Character {  
    id: ID!  
    name: String!  
    friends: [Character]  
    appearsIn: [Episode]!  
}
```

```
type Human implements Character {  
    id: ID!  
    name: String!  
    friends: [Character]  
    appearsIn: [Episode]!  
    starships: [Starship]  
    totalCredits: Int  
}  
  
type Droid implements Character {  
    id: ID!  
    name: String!  
    friends: [Character]  
    appearsIn: [Episode]!  
    primaryFunction: String  
}
```

GQL SDL обзор возможностей

Type concept - Базовые элементы

Input Types - удобный способ передавать
параметры в виде объектов

```
input ReviewInput {  
    stars: Int!  
    commentary: String  
}
```

```
mutation CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {  
    createReview(episod  
    stars  
    commentary  
}  
}  
  
{  
    "ep": "JEDI",  
    "review": {  
        "stars": 5,  
        "commentary": "This is a great movie!"  
    }  
}
```

GQL SDL обзор возможностей

Type concept - Базовые элементы

Аргументы похожи на определение
функций в TS

```
type Starship {  
    id: ID!  
    name: String!  
    length(unit: LengthUnit = METER): Float  
}
```

GQL SDL обзор возможностей

Type concept - Базовые элементы

Enum и Union аналогично

```
enum Episode {  
    NEWHOPE  
    EMPIRE  
    JEDI  
}
```

```
union SearchResult = Human | Droid | Starship
```

GQL SDL обзор возможностей

Type concept - Базовые элементы

Валидация типов и runtime ошибки

```
query HeroForEpisode($ep: Episode!) {
  hero(episode: $ep) {
    name
    primaryFunction
  }
}

{
  "ep": "JEDI"
}
```

```
{
  "errors": [
    {
      "message": "Cannot query field \"primaryFunction\"",
      "locations": [
        {
          "line": 4,
          "column": 5
        }
      ]
    }
  ]
}
```

GQL SDL обзор возможностей

Type concept - Базовые элементы

Валидация типов и runtime ошибки

```
# INVALID: favoriteSpaceship does not exist on Character
{
    hero {
        favoriteSpaceship
    }
}
```

```
{  
  "errors": [  
    {  
      "message": "Cannot query field \"favoriteSpaceship\".  
      "locations": [  
        {  
          "line": 4,  
          "column": 5  
        }  
      ]  
    }  
  ]
```

```
# INVALID: hero is not a scalar, so fields are needed
{
    hero
}
```

```
{  
  "errors": [  
    {  
      "message": "Field \\\"hero\\\" of type \\\"Character\\\" is required.",  
      "locations": [  
        {  
          "line": 3,  
          "column": 3  
        }  
      ]  
    }  
  ]  
}
```

GQL SDL обзор возможностей

Type concept - Execution Time

- Без использования type system, query, mutation не может быть исполнены
- Каждое поле или ручка - требует отдельного **resolver** на сервере
- **Resolver** - определяет то как определенное поле или объект запрашивает данные
- Для вложенных объектов и полей **resolver**-ы переиспользуются
- В случае скалярного значения - **resolver** отработал и все
- В случае вложенных nested запросов - **resolver** исполняется до конца вглубь
- Важно, что каждое поле сервер может обрабатывать асинхронно

GQL SDL обзор возможностей

Type concept - Introspection

- Схема определяет какие query ее поддерживают
- GraphQL позволяет нам использовать `IntrospectionSystem`
- Система позволяет писать unit тесты, тестируя соответствие query и schema-ы
- Валидировать на лету написание запросов в IDE
- Проверять в момент исполнения запросов соответствие контрактов

Проблемы GraphQL



Проблемы GraphQL

Известные недостатки

- Проблема N+1 запроса на сервере
- Проблемы отслеживания ошибок во вложенных запросах (Nested Queries)
- Открытый для чтения данных из инспектора
- Требует ordered-list при работе с параметрами
- Сложно интегрировать в существующую схему модели стороннего API
- Строгость NotNull, Nullable
- Полиморфизм и __typeNames
- Джениерики? А все просто - их нету :)
- Неймспейсы? Их тоже нет

Проблемы GraphQL

Известные недостатки

- Головоломки со строгой типизацией, сгенеренными типами
- Бекенд разработчики вынуждены поддерживать обратную совместимость в версиях
- Пагинация требует дополнительно передачи offset у курсора
- Кэширование - сложность идентификаторов ключей
- Каждый запрос для сервера отличается, несмотря на то, что работает с той-же сущностью

Проблемы GraphQL

Известные недостатки - N+1

- Вы хотите получить автора и всего его книги
- Хранилища наподобие SQL хранят книги и авторов в **разных таблицах**
- Каждый resolver будет запрашивать данные через отдельную ORM модель
- Так как resolver будет вызван для каждого ID
- То будет N запросов - в виде `SELECT * FROM books WHERE author_id == ID`
- То есть нам придется выполнять запрос к базе данных **для каждого автора**
- Мы теряем возможность использовать оптимизацию `SELECT * FROM books WHERE Author_id in (1,2,3)`
- Решение конечно есть - некий магический **dataloader (Batching)**
- Его применение, вынуждает нас добавлять еще один слой сложности

Проблемы GraphQL

Известные недостатки - N+1

Что такое Batch и Data loader?

- DataLoader - запоминает все одиночные запросы
- Аккумулирует запросы для каждой сущности
- Собирает массив идентификаторов

Проблемы GraphQL

Известные недостатки - N+1

И как решается N+1?

- В результате мы можем производить запрос связанных объектов произвольного уровня
- Например: запросим всех авторов, а потом к ним все связанные книги
- Но дополнительно запрашивать будем и в кейсах, где не нужно

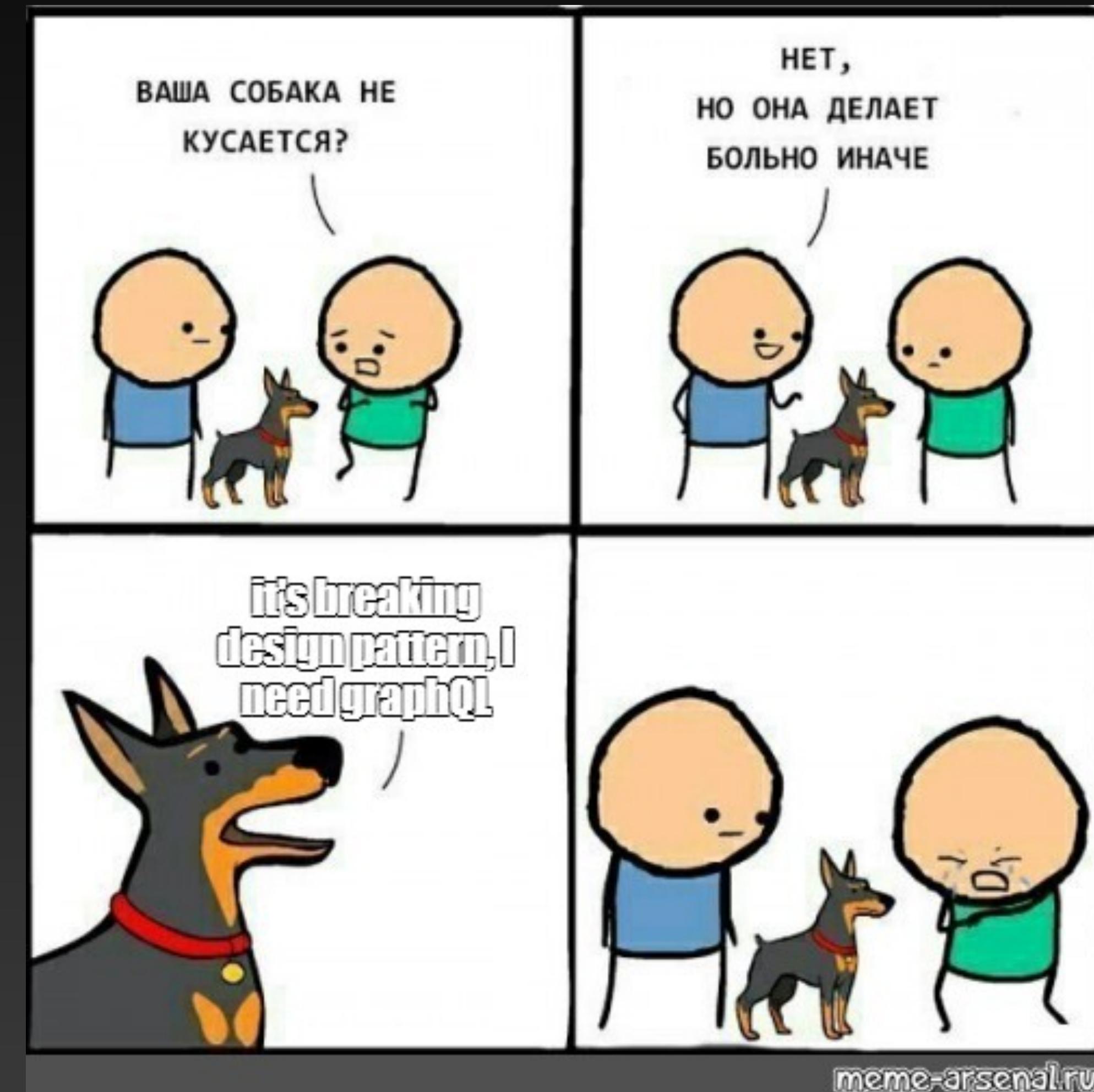
Проблемы GraphQL

Известные недостатки - N+1

Выводы

- Здесь нужно понимать, что если перейдем к работе с SQL
- На каждый уровень вложенности объектов будет по одному агрегированному запросу
- А иногда требуется чтобы это был все равно один запрос
- Иначе будет вопрос: почему так медленно отвечает ар?

Проблемы GraphQL



Проблемы GraphQL

Проблемы с отслеживанием ошибок

- Сложно определить проблему на определенном уровне вложенности
- Запросы клиента к новой версии API с использованием старой схемы вызывают ошибки и несоответствия
- Изменения в API могут вызывать непонятные ошибки, требующие внимательного анализа
- Для эффективного анализа ошибок требуется хорошее логирование и время для привыкания к процессу

Проблемы GraphQL

Безопасность из коробки

- Любой может посмотреть в инспекторе запрос, скопировать, повторить
- Есть решение [Persisted queries](#) - запросы без BODY, только хэши
- Схему надо прятать и точно не гонять по сети
- Разный бэд софт хорошо умеет сканировать apiEndPoints GraphQL (особенно если у вас /graphql)
- GraphQL также подтвержден различным атакам:
- IDOR, DOS, Brute, SSRF, SQL Injection, OS Command Injection, XSS, ByPass, CSRF

Проблемы GraphQL

Аккуратность при работе с параметрами

- Порядок параметров в query файлах, где лежат запросы - имеет важность
- Можно принудительно сортировать параметры

Проблемы GraphQL

Сложность интеграции в схему
внешних API и моделей

- Как говорилось ранее, отсутствие неймспейсов - модели и типы могут задублироваться и перетирать друг-друга
- Одно из решений: [Apollo Federation](#)
- Дает возможность спрятать разные GraphQL API в один [GateWay](#)
- Хорошо подходит под микросервисную архитектуру
- Смысл в том, чтобы предоставить клиенту ощущение, что он работает с одним API

Проблемы GraphQL

Нет гибкости в статус кодах

- В основном код статусов не так много: 500, 401, 400, 200
- GraphQL [transport agnostic](#) - ему все равно, как он получает запросы
- Если не сработал какой-то один resolver, code status будет 200 или 500

Проблемы GraphQL

Полиморфизм

- В случае когда на бекенде есть дженерики
- Не так просто, задать разную бизнес логику для подтипов на клиенте
- `__typename` у подтипов будет разный
- Нужно проверять `__typename` и делать по ним switch case

Проблемы GraphQL

Головоломки со строгой типизацией

- Если сущности описаны по разному и используются по разному и встречаются - больно
- `Nullable` или `NotNullable` в разных запросах сложно обрабатывать - больно
- Совмещение выводимых и локальных types при кодогенерации - больно

Проблемы GraphQL

Головоломки со строгой типизацией

Приходится использовать подобные конструкции

```
113  ↘ const mapDataToProps = ({  
114    entry,  
115    position,  
116    isRubricPage,  
117  }: DictionaryCardData): DictionaryPropTypes => {  
118    ↘ switch (entry.__typename) {  
119      ↘ case 'Entry':  
120        ↗ return EntryMapper(entry, position, isRubricPage);  
121      ↘ case 'News':  
122        ↗ return NewsMapper(entry, position);  
123      ↘ case 'EntryArchive':  
124        ↗ return EntryArchiveMapper(entry, position, isRubricPage);  
125      ↘ case 'NewsArchive':  
126        ↗ return NewsArchiveMapper(entry, position);  
127      ↘ default:  
128        ↗ return {  
129          ↗ position: 0,  
130          ↗ href: '',  
131          ↗ title: '',  
132        };  
133    }  
134  };  
135  
136  export default mapDataToProps;  
137
```

Причины выбора

Пушка бомба?



Причины выбора

Аргументы

- У Afisha Daily уже был написан REST-API для CMS
- GraphQL удобно ложиться поверх готового REST-API*
- Уже был небольшой опыт работы с graphql и хорошая экспертиза в Афише

Apollo Client

Наш выбор

- Нужен был готовый клиент с плюшками в стек с React NextJS (SSR/SPA)
- Нормализация из коробки за счет **Apollo Cache** и **Cache Policy**
- Генерация клиентского кода для запросов ([graphql-code-generator](#))
- Гибкая настройка каждого этапа запроса - **Apollo Link aka (Middleware)**

Apollo Client

SSR Конфигурация

- На каждый SSR Request - нужно создавать новый клиент с *In Memory Cache*
- Передавать кэш в момент гидрации
- Чтобы избавиться на клиенте от лишних запросов в браузере
- На клиенте происходит первый запрос данных из кэша

Apollo Client

SSR Конфигурация

```
26  /*
27   * Initializes apollo client with cache for using in Next.js scope
28  */
29  export const initializeApollo = (initialState?: NormalizedCacheObject) => {
30    const _apolloClient = client ?? createApolloClient();
31
32    // If your page has Next.js data fetching methods that use Apollo Client,
33    // the initial state gets hydrated here
34    if (initialState) {
35      // Get existing cache, loaded during client side data fetching
36      const existingCache = _apolloClient.extract();
37    const data = merge(initialState, existingCache, { You, 12 месяцев назад • Refactor apollo client,
38      arrayMerge: (destinationArray, sourceArray) => [
39        ...sourceArray,
40        ...destinationArray.filter(d =>
41          sourceArray.every(s => !isEqual(d, s))
42        ),
43      ],
44    });
45    // Restore the cache using the data passed from
46    // getStaticProps/getServerSideProps combined with the existing cached data
47    _apolloClient.cache.restore(data);
48  }
49
50  // For SSG and SSR always create a new Apollo Client
51  if (typeof window === 'undefined') return _apolloClient;
52
53  // Create the Apollo Client once in the client
54  if (!client) client = _apolloClient;
55  return _apolloClient;
56 };
57
```

Apollo Client

SSR Конфигурация

```
146
147 √ export const handleNewsListPageRequest: PageRequestHandler<
148   {
149     news?: News[];
150     siteConfiguration?: DeepPartial<SiteConfiguration> | null;
151   },
152   RubricRouterCtx
153 √ > = async ({getData, context, client}) => {
154   const {pageNumber} = getRubricRouterCtx(context);
155
156   if (!isSafePageNumber(pageNumber)) return badRequest(context);
157
158   const data = await getData({pageNumber, rubricSlug: 'news'});
159
160   const newsListIsEmpty = data?.news?.length === 0;
161
162   √ if (newsListIsEmpty) {
163     return notFound(context);
164   }
165
166   √ return {
167     apolloCache: client.cache.extract(),
168     experiments: context.res.locals.experiments,
169     configuration: data?.siteConfiguration,
170   };
171 };
```

А что на сервере?

C# HotChocolate

- Наработанная экспертиза в его использовании со стороны Афиши
- Встроенный свой data-loader для решения проблемы N+1
- Хорошая производительность
- Полная поддержка .NET
- Гибкая конфигурация, богатая документация
- Активное сообщество разработчиков

Инструменты которые используем

Общий список

- Apollo DevTools (Просмотр кэша в браузере)
- Graphql-codegen (Генерация клиента)
- Генерим correlationId - uuid для отслеживания конкретного запроса
- Playground для запросов (HotChocolate)
- GraphQL Plugin IDE Jetbrains (Синтаксис, валидация, запуск генерации кода)
- Apollo extension VS Code (Аналогично тому что выше, ток без генерации)
- Набор библиотек для persisted query (пара пакетов прм, отдельная база и CI Job)

Приобретенный опыт и проблемы

Первая боль

- Сложность в логировании SSR/Client (нужен изоморфный логгер)
- В коробке при сериализации ошибки - не выводится вся глубина данных
- Много времени ушло на ресерч и правильную конфигурацию клиента apollo (SSR / browser)
- Cache policy и существование кэша - тайна для новичков в команде
- Без DevTools непонятно, как устроен кэш
- Не получилось разделить mobile/desktop полностью в рамках одной схемы
- Полиморфизм и __turrename - порождает массивный код
- Кэширование на бекенде потребовало значительного рефакторинга

Приобретенный опыт и проблемы

Боль спустя время

- Настройка и обслуживание Persisted Query заняло много времени
- Утечка памяти в SSR (Apollo Client)
- Были инциденты с деплоем - когда клиент и сервер различались
- Денормализованные данные - не писались в кэш

Приобретенный опыт и проблемы

Persisted Query подробнее

- Persisted Query - убирает query тело из BODY в HTTP запросе
- На сервер отправляется только сгенеренный от query хэш
- Хэши хранятся в отдельной базе, при запросе сервер - поднимает тело запроса из базы
- При развертывании приложения - мы обновляем базу через CI
- Сервер может принимать полные запросы внутри сети

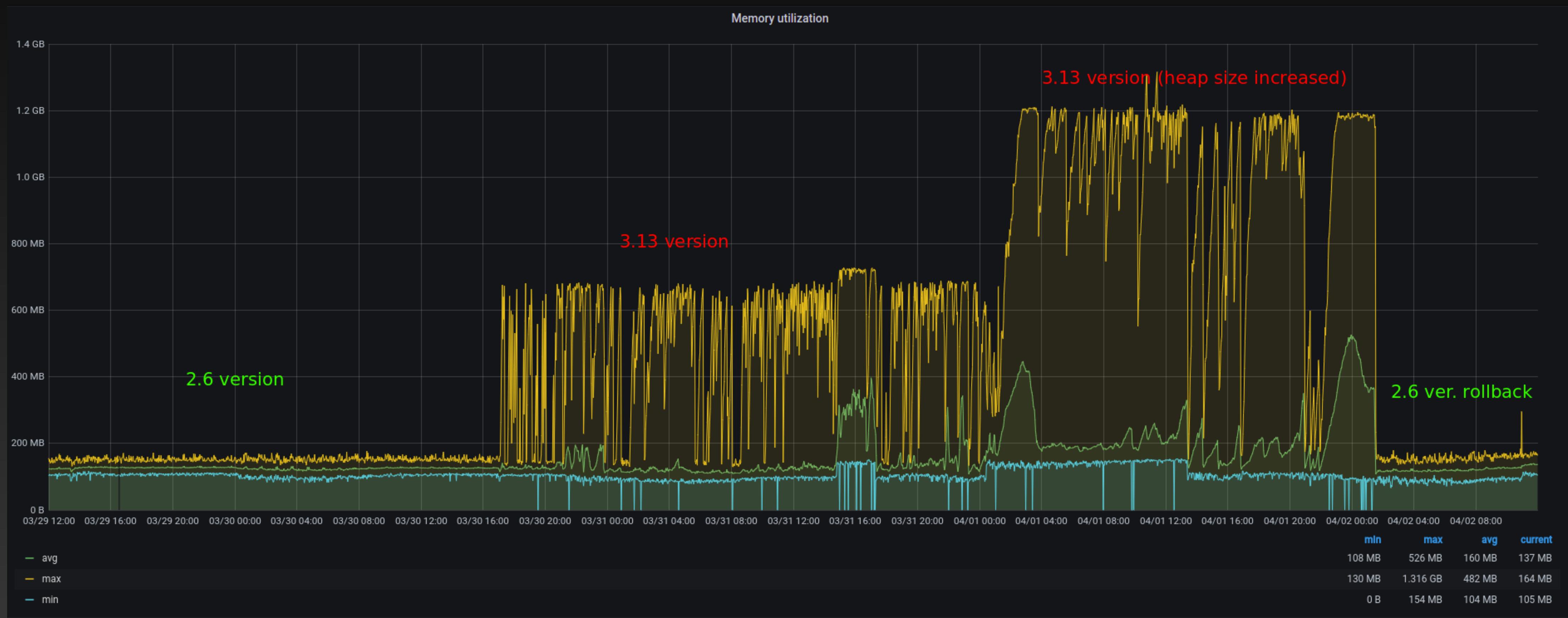
Приобретенный опыт и проблемы

Боль спустя время - Дублирование клиента

- У нас единая кодовая база для Desktop / Mobile (Webpack entry root)
- При сборке приложение через webpack резолвер подменяет путь в imports
- На выходе генерируется по клиенту для каждой платформы
- В Mobile есть что-то от Desktop, и наоборот
- Излишний код клиента попадает в выходной bundle - проблема еще не решена

Приобретенный опыт и проблемы

Боль спустя время - Утечка памяти в SSR



Приобретенный опыт и проблемы

Боль спустя время - Утечка памяти в SSR

- Проблема из-за сериализации, десериализации данных при чтении и записи в cache
- Долго ждали фикса от Apollo
- В версии 3.9 пофиксили (Ждали 1.5 года)
- После фикса GC стал очищаться, но несколько раз в сутки, вызывая timeout
- Контроль за потреблением памяти процессом - стал не тривиальной задачей

Рекомендации



Рекомендации

Перед тем как использовать - 1

- Не стоит брать новую технологию, если требуется стабильное решение в короткий срок
- Можно попробовать на мелких микросервисных ручках в отдельных микрофронтах
- Как маленькое дополнение к REST-API

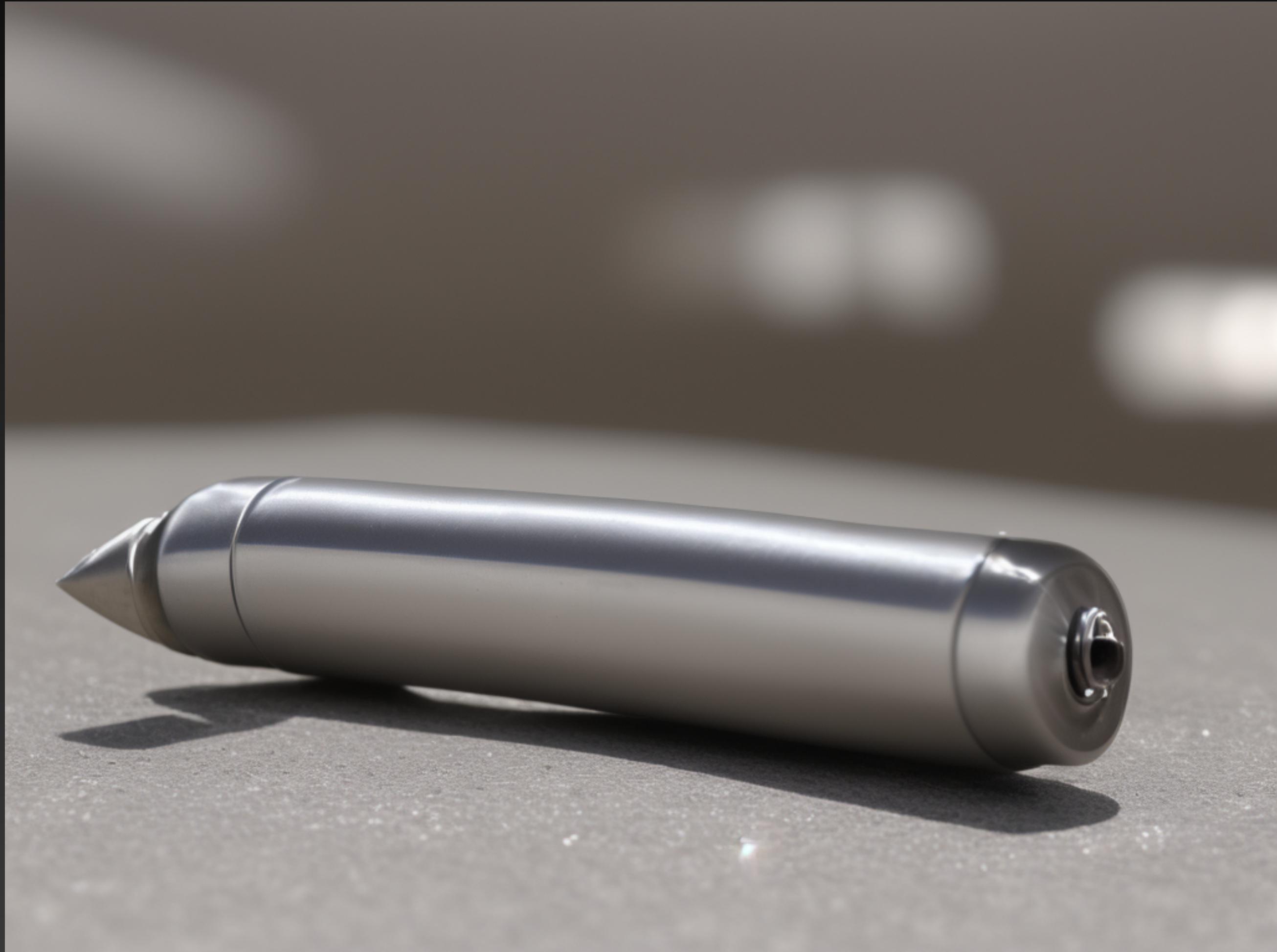
Рекомендации

Перед тем как использовать - 2

- Варить свой клиент - достаточно муторное занятие (знаем из опыта [eda.ru](#))
- Стоит учитывать наличие highload при выборе graphql
- Нужно приличное время на освоение и экспертизу

Рекомендации

Серебряная пуля?



Рекомендации

Нет!

- Что-то стало удобнее, что-то в обратном направлении добавляет мелких хлопот
- Научившись варить graphql - уже не жалуемся
- Есть ощущение единого образа в работе с API - это конечно плюс!
- Остаются нерешенные до конца, проблемы с которыми мы живем

Полезные ресурсы

QR CODE

