GraphQL

Как уговорить сервер подстроиться под приложение

Александр Газаров

Лидер сообщества Android, Райффайзен Банк

Обо мне

Android – с 2010 года

Java ME – с 2006 года

- Ведущий разработчик – лидер сообщества, вице-президент;
- Ответственный за приложение Android для малого и среднего бизнеса;
- Автор GraphQL-клиента Raiffeisen.

О чём пойдёт рассказ

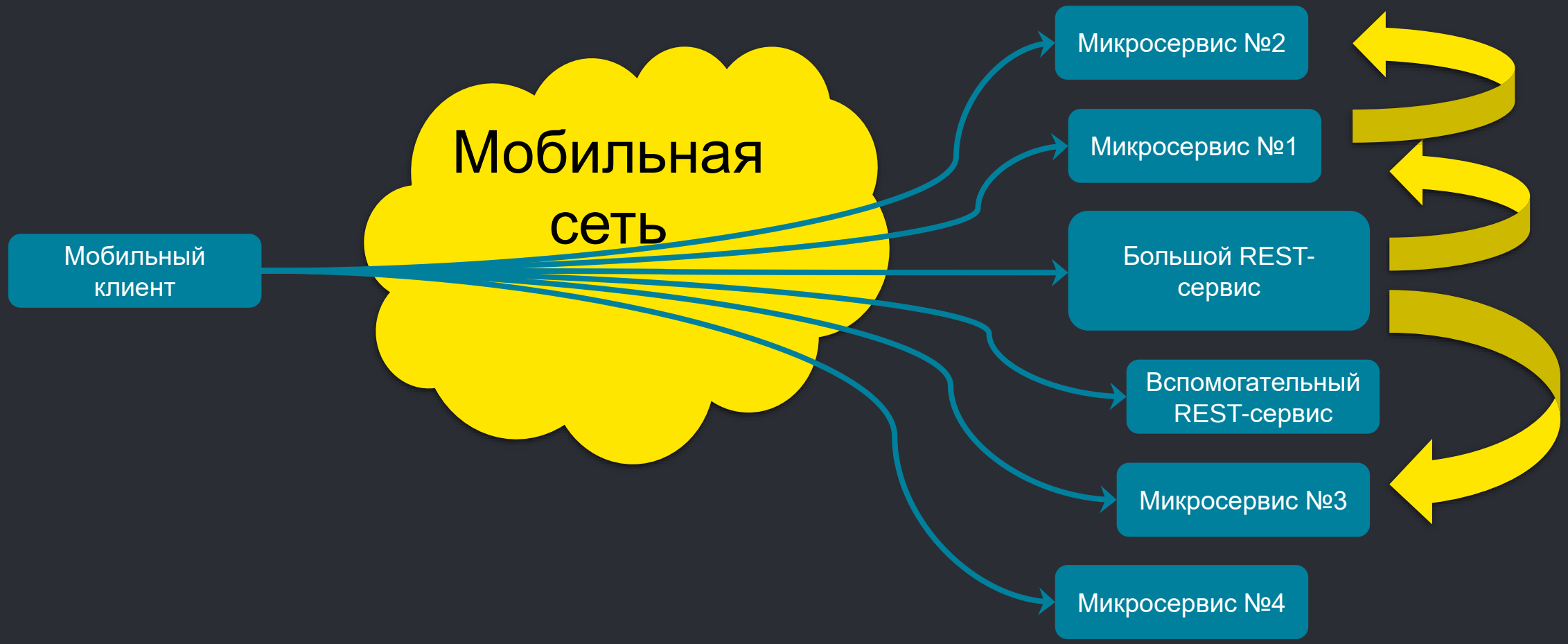
- Введение в GraphQL;
- Обзор клиентов и их возможностей;
- Проблемы с многомодульностью и их решение;
- Подведение итогов.

Введение в GraphQL

Что у нас за зверь такой

Как растёт наша серверная часть

И почему с ней становится сложно работать



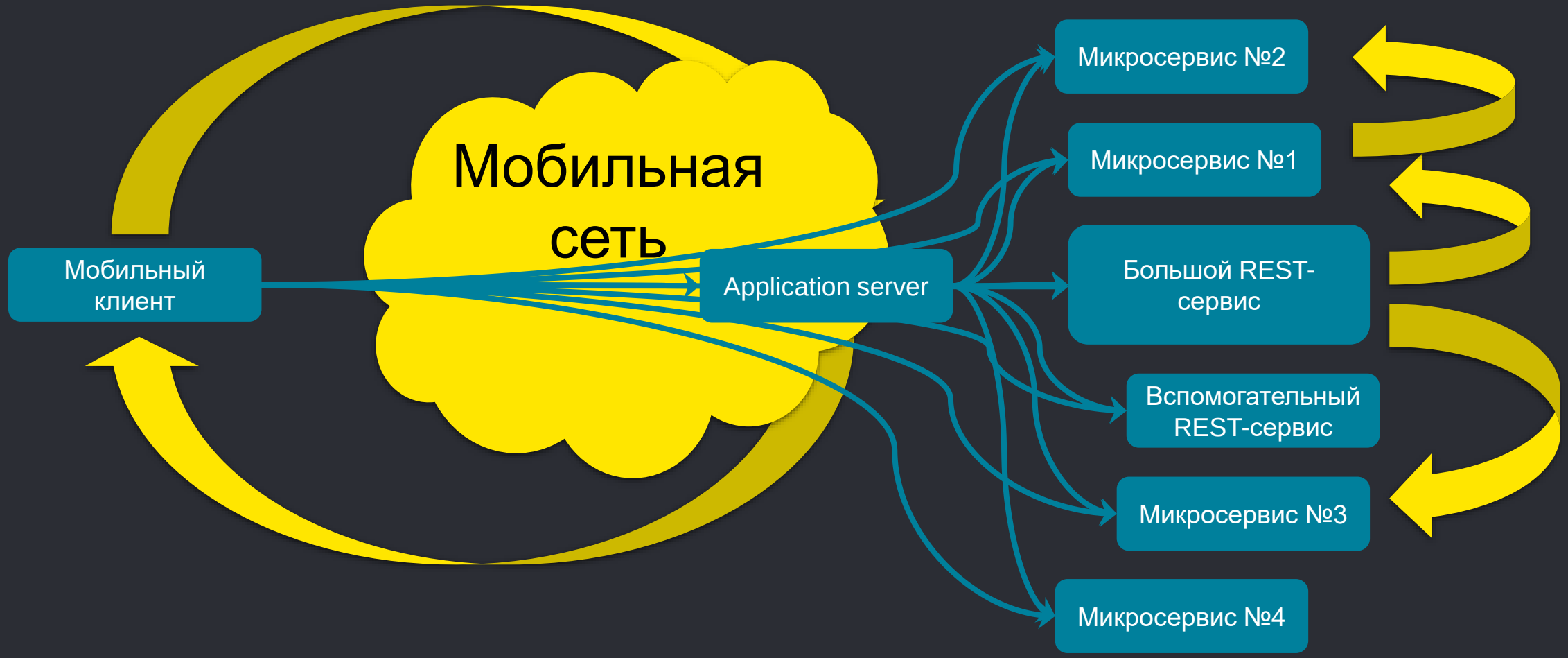
Вот серверная часть и подросла

И появились проблемы

- Необходимость в нескольких запросах, иногда по цепочке;
- Лишние данные в запросах, неактуальные для мобильного клиента;
- Множественность точек входа как результат микросервисной архитектуры;
- Трудная замена устаревших сервисов;
- Разрозненная и неактуальная документация по сервисам.

Что, если ввести application server

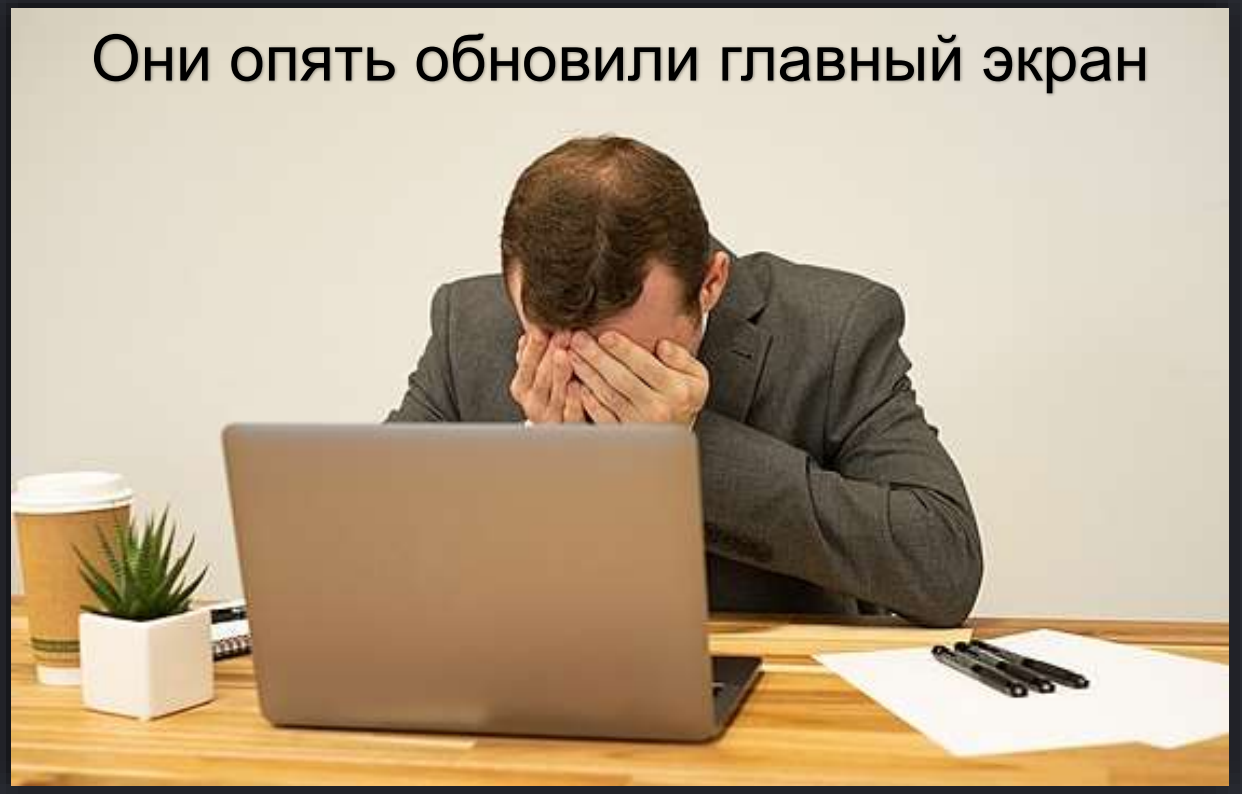
Пусть кто-то другой ходит скрести по сусекам



Application server нам поможет

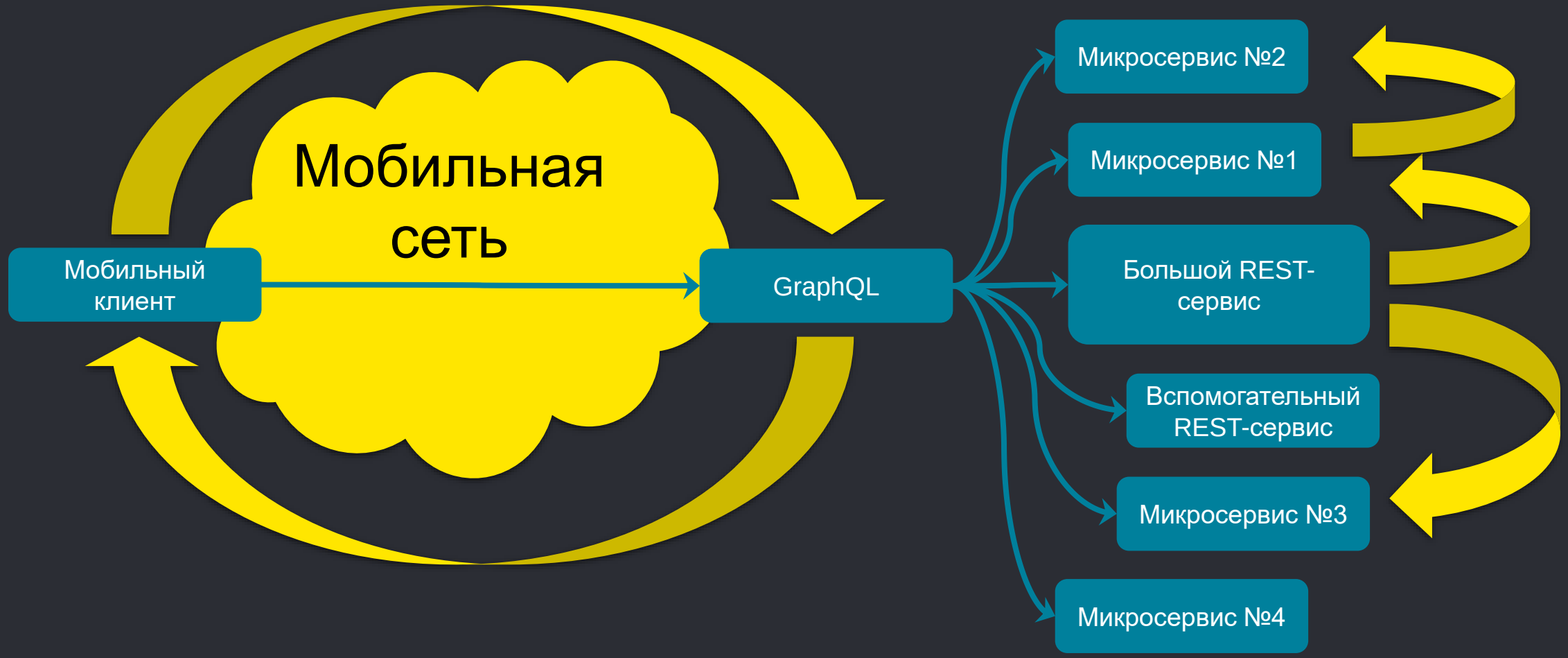
Но это не точно

- Необходимость в постоянных зеркальных изменениях;
- Балансирование между недостаточной оптимизацией и затратами на поддержку.



GraphQL спешит на помощь

Наш сервер поумнел и возмужал



Как построена работа с GraphQL-сервисом

Что там вообще ходит по сети?

```
type Query {
    organization(orgId: String!):
Organization
    organizations: [Organization!]!
}

type Organization {
    id: String!
    shortName: String!
    intName: String!
}
```

Это схема

```
{
    organizations {
        shortName
    }
}
```

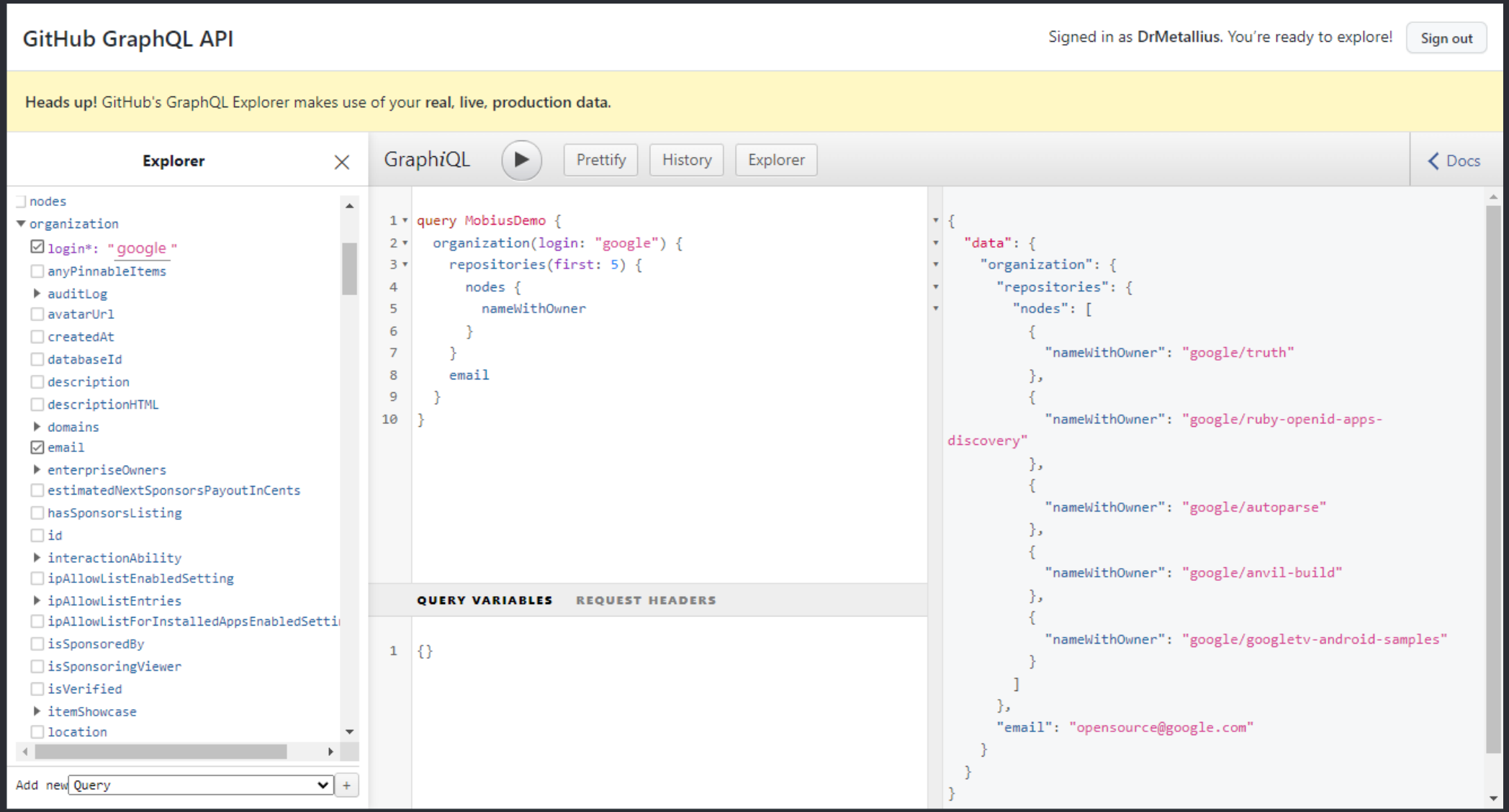
А это запрос

И вот ответ

```
{
  "data": {
    "organizations": [
      {
        "shortName": "ООО Промышленные конструкции"
      },
      {
        "shortName": "ООО ЗЕЛЬГРОС"
      },
      {
        "shortName": "ООО Дом изысканного чая"
      }
    ]
  }
}
```

Как всё выглядит в песочнице

На примере GitHub



Итоги работы GraphQL-сервиса

Мобильному клиенту дышится легче

- Один запрос, чтобы править всеми;
- Только необходимые клиенту данные в ответе;
- Единая точка входа;
- Прозрачность изменения сервисов для клиента;
- Обязательное наличие базовой документации.

Сравнение подходов

Столкнём лоб в лоб

	Микросервисы	GraphQL	Application server
Минимизация запросов	Нет	Да	Частично
Экономия трафика	Нет	Да	Частично
Устойчивость к изменениям в API сторонних сервисов	Нет	Да	Да
Версионирование	Требуется	Не требуется	Требуется
Единая точка входа	Нет	Да	Да
Самодокументируемость	Нет	Да	Нет
Затраты на поддержку	Отсутствуют	Средние	Большие

Сервисов мало?
GraphQL ни к чему

Но до него продукт ещё дорастёт

Клиенты GraphQL

Кто же будет в сеть ходить?

Схема для наших экспериментов

Чтобы экспериментировалось как следует

```
type Query {
    organization(orgId: String!): Organization
    organizations: [Organization!]!
    movements(first: Int, offset: Int, movementFilter: MovementFilter): [Movement!]!
    accounts: [Account!]!
    account(number: String!): Account
}
type Movement {
    amount: BigDecimal
    operationType: OperationType
    counterparty: String
}
type Organization {
    id: String!
    shortName: String!
    intName: String!
}

enum OperationType {
    INCOMING, OUTGOING
}

input MovementFilter {
    operationType: OperationType
}

scalar BigDecimal
```

Apollo Android

К версии 3.0 поддержка Kotlin наладилась

Пишем наш запрос

```
query MovementQuery($number: String!, $first: Int, $offset: Int, $filter: MovementFilter) {
    account(number: $number) {
        movements(first: $first, offset: $offset, filter: $filter) {
            amount
            counterparty
        }
    }
}
```

Генерируется класс запроса

```
data class MovementQuery(
    val number: String,
    val first: Int,
    val offset: Int,
    val filter: MovementFilter
)
```

Заполняем и шлём

```
apollo.query(
    MovementQuery(
        number = number,
        first = limit,
        offset = offset,
        filter = MovementFilter(
            operationType = Optional.Present(OUTGOING)
        )
    )
)
```

GraphQL Kotlin с Ktor

Работаем поверх OkHttp

Генерируется класс запроса

```
class MovementQuery(
    public override val variables: MovementQuery.Variables
)
```

Заполняем и шлём

```
client.execute(
    MovementQuery(
        variables = MovementQuery.Variables(
            number = number,
            first = limit,
            offset = offset,
            filter = MovementFilter(
                operationType = OUTGOING
            )
        )
    )
)
```

```
data class Variables (
    val number: String,
    val first: Int,
    val offset: Int,
    val filter: MovementFilter
)
```

Общие недостатки существующих клиентов

Кодогенерация портит малину

- Отсутствие интеграции кодогенерации с IDE;
- Замедление времени сборки.

Автор изображения – Vysotsky (Wikimedia)

Клиент Raiffeisen

А что делаем мы?

```
@Serializable
class MovementQuery(
    @Arg("number", "\$number") val account: Account
)

@Serializable
class Account(
    @Arg("first", "\$first")
    @Arg("offset", "\$offset")
    @Arg("filter", "\$filter")
    val movements: List<Movement>
)

@Serializable
class Movement(
    val amount: BigDecimal,
    val counterparty: String
)
```

И ничего не генерируется

```
@Serializable
class MovementQueryVars(
    val number: String,
    val first: Int,
    val offset: Int,
    val filter: MovementFilter
)
```

Заполняем и шлём

```
val result: MovementQuery = client.query(
    MovementQueryVars(
        number = number,
        first = limit,
        offset = offset,
        filter = MovementFilter(
            operationType = OUTGOING
        )
    )
)
```

Клиент Raiffeisen

Переменных нет? Всё ещё проще

```
@Serializable
class MovementQuery(
    val accounts: List<Account>
)

@Serializable
class Account(
    @Arg("first", "30")
    @Arg("offset", "0")
    val movements: List<Movement>
)

@Serializable
class Movement(
    val amount: BigDecimal,
    val counterparty: String
)
```

```
val result: MovementQuery = client.query()
```

Шлём и ничего не заполняем

Сравнение клиентов

Столкнём лоб в лоб

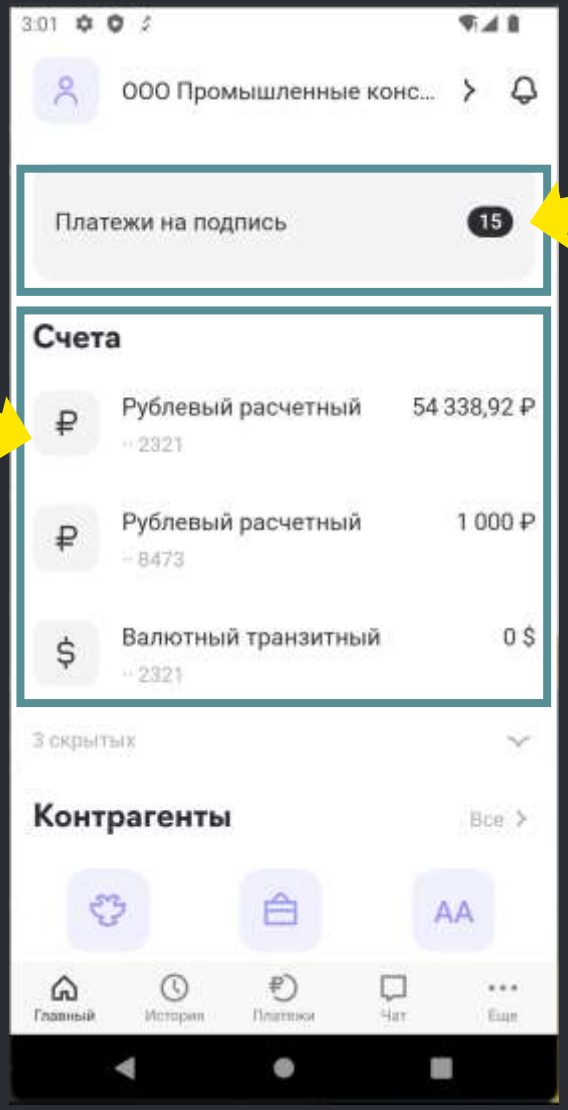
	Apollo Android	GraphQL Kotlin	Raiffeisen GraphQL
Кодогенерация	Да	Да	Нет
Библиотека для сериализации	Собственные инструменты	Jackson, Kotlin Serialization	Kotlin Serialization
Работа с OkHttp	Да	Да, через Ktor	Да
Проверка по схеме	Да	Да	Нет
Кэширование	Да	Нет	Нет

Проблемы с многомодульностью и их решение

Когда приложение большое, экономии достигать сложнее

Главный экран приложения

И его участники из разных модулей, каждый из которых рождает свой запрос

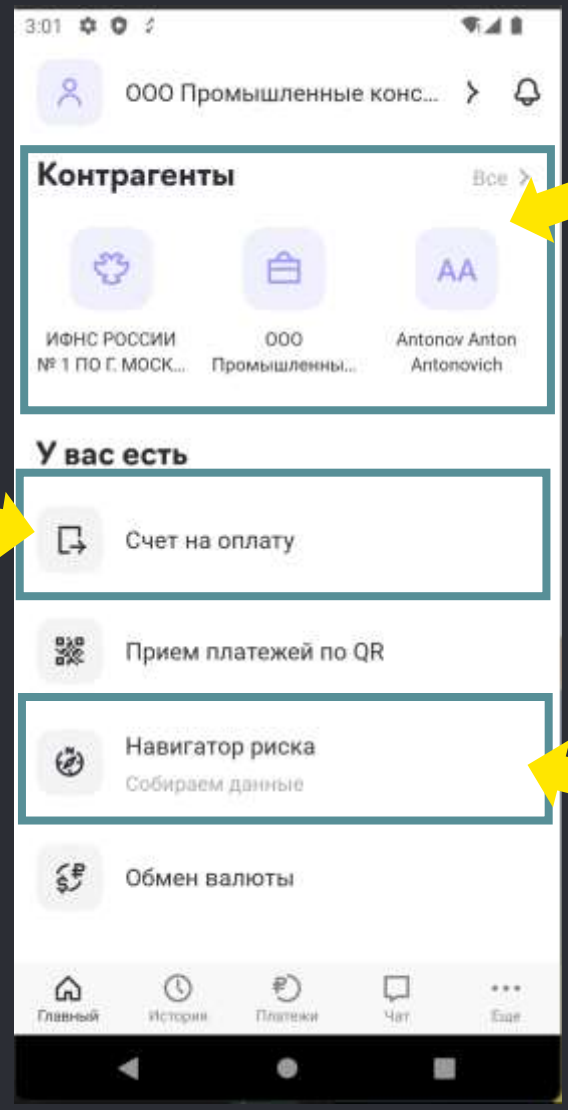


Платежи на подпись

Здесь грузим счета и аресты

И бухгалтерию…

А тут счета на оплату



Здесь контрагентов

И все прочие сервисы

И навигатор риска

Как бороться в новых условиях?

Вся экономия – странный предмет. Если и есть, то её сразу нет!

- Многочисленность независимых модулей = многочисленность независимых запросов;
- Отсутствие автоматической выгоды при подключении GraphQL.

Пусть так и останется

Отправим на сервер несколько запросов в массиве

Но осилит ли сервер?

Совместим несколько запросов в один на клиенте, потом ответ разделим на части

Но осилит ли клиент?

Пакетирование запросов

Метаморфозы начинаются

```
{
    organizations {
        shortName
    }
}
```

+

```
{
    organizations {
        accounts
    }
}
```

=

```
{
    organizations {
        shortName
        accounts
    }
}
```

или

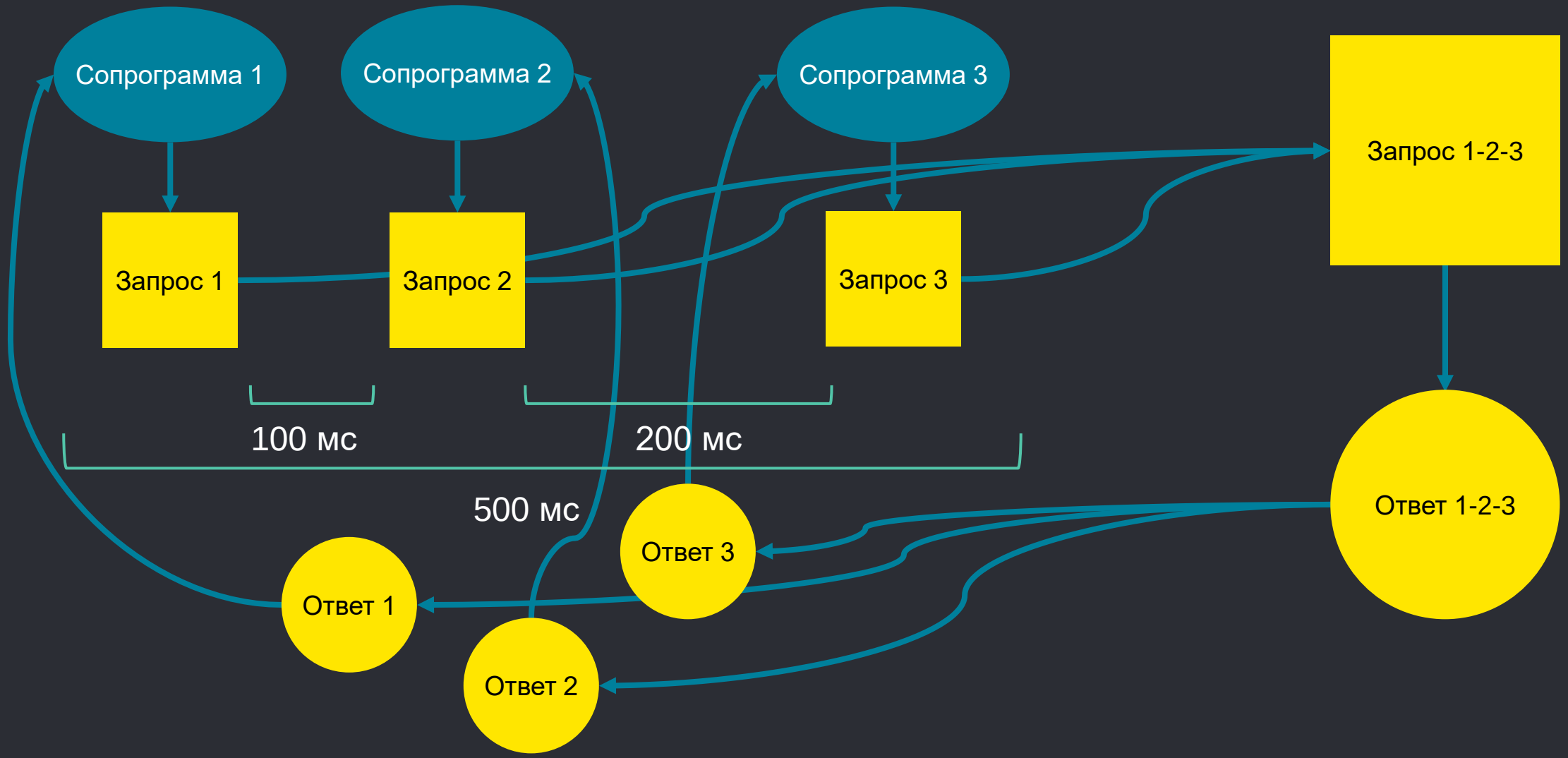
```
[
    {
        organizations {
            shortName
        }
    },
    {
        organizations {
            accounts
        }
    }
]
```

Слияние в один запрос (пакетирование на уровне запроса)
Плюсы: любой сервер может обработать
Минусы: клиенту нужно трудиться, вероятны монстроподобные запросы

Отправка массивом (пакетирование на транспортном уровне)
Плюсы: клиентский код намного проще
Минусы: не по стандарту, не всякий сервер может обработать

Как работает наш клиент

Делаем хитрые приседания



Сравнение подходов к пакетированию

Режим пакетирования включать строго по требованию

	Обычный клиент	Пакетирующий на уровне транспорта (Apollo Android, GraphQL-Kotlin)	Пакетирующий на уровне запроса (Raiffeisen GraphQL)
Соответствие стандартам	Да	Нет	Да
Моментальность отправки и получения ответа	Да	Нет	Нет
Сложность отладки	Низкая	Средняя	Высокая
Эффективность работы в условиях многомодульности	Нет	Да	Да

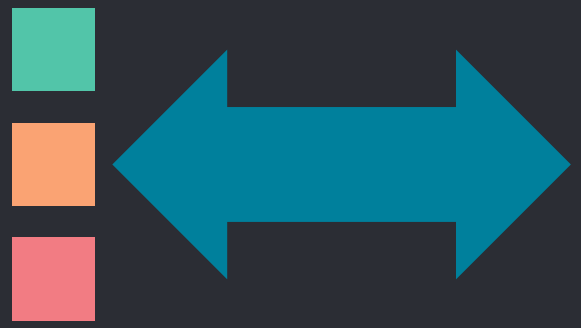
Дополнительные альтернативы

Не пакетированием единым

Обычный клиент, не пакетирующий, где нет необходимости

Keep
It
Simple,
Stupid

HTTP/2, который поддерживает мультиплексирование запросов



Automated Persisted Queries

```
{
    organizations {
        shortName
    }
}
```

1d62fb65d6759a9eedfa4d5d612…

Подведение итогов

Считаем числа

Главный экран до и после

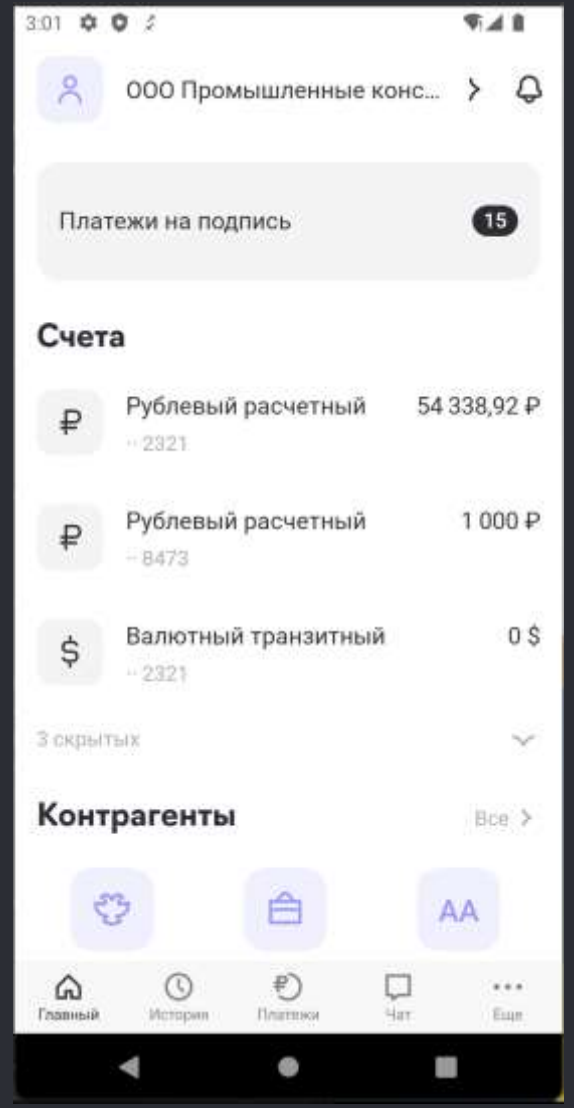
Пожинаем плоды

Было:

Счета
Депозиты
Аресты × кол-во счетов
Контрагенты
Пластиковые карты
Истории
Бухгалтерия

Стало:

Запрос на GraphQL



Экономия трафика

Симулируем EDGE и смотрим, что получается

Было:

7 запросов об организациях и счетах (≈266 КБ)

52,15 секунд

Стало:

3 аналогичных запроса (≈110 КБ)

8,74 секунд

Потребление аккумулятора меньше

Загрузка экрана шустрее

Мысли о свободном доступе в будущем

Когда заматереет как следует

Спасибо за внимание!