



Максимов Иван

Ведущий разработчик технологической платформы ЕФР

[maksimov.ial@open.ru](mailto:maksimov.ial@open.ru)

[sungam3r@yandex.ru](mailto:sungam3r@yandex.ru)



<https://join.skype.com/invite/nY26BKiywjW>



<https://t.me/sungam3r>

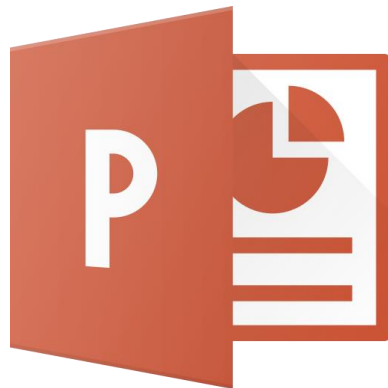


<https://github.com/sungam3r>

# Где можно посмотреть?



<https://github.com/sungam3r/graphql-story>

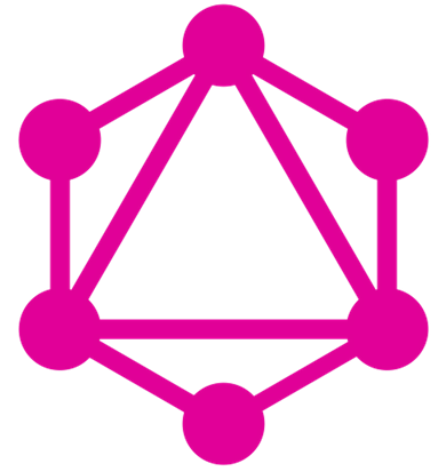
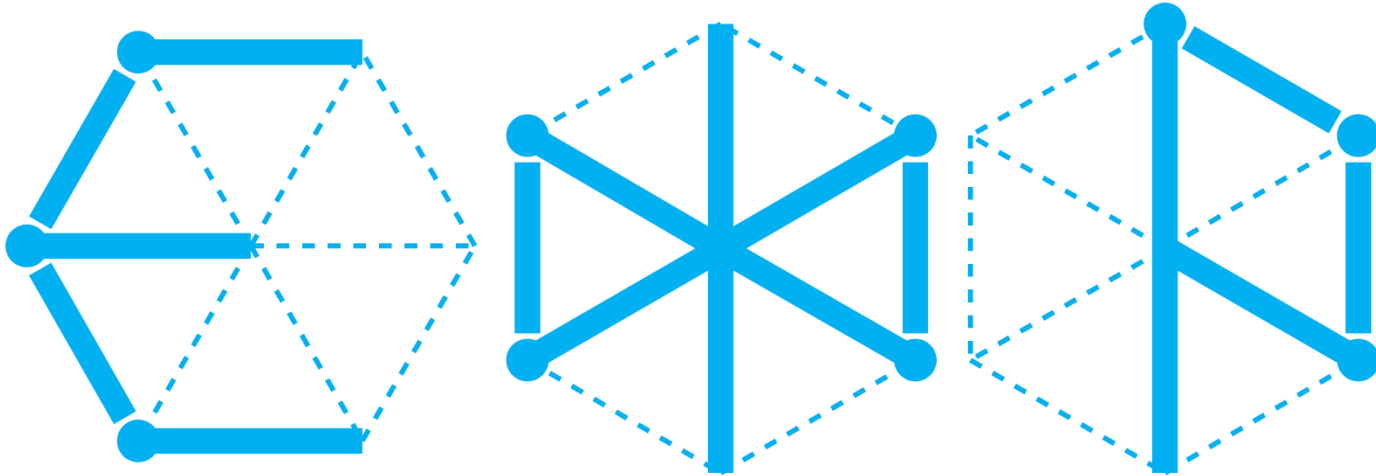


+



# Жизни нет без GraphQL

## Обзор концепции и возможностей



Единое фронтальное Решение

GraphQL

# Нельзя объять необъятное



за 30 минут

# Чего **не будет** в этой презентации

- Сложных концепций, выходящих за базовый уровень языка
- Подробностей механизмов работы описываемых технологий
- Особенности реализации на различных платформах
- Исходного кода

будет только немного примеров GraphQL запросов и их схем



# Цели этой презентации

- Дать общее представление о том, что же такое GraphQL
- Рассказать о предпосылках появления GraphQL
- Описать цели использования и место GraphQL в ЕФР

# ЧАСТЬ 0

Как сделать микросервисы на самом деле  
переиспользуемыми

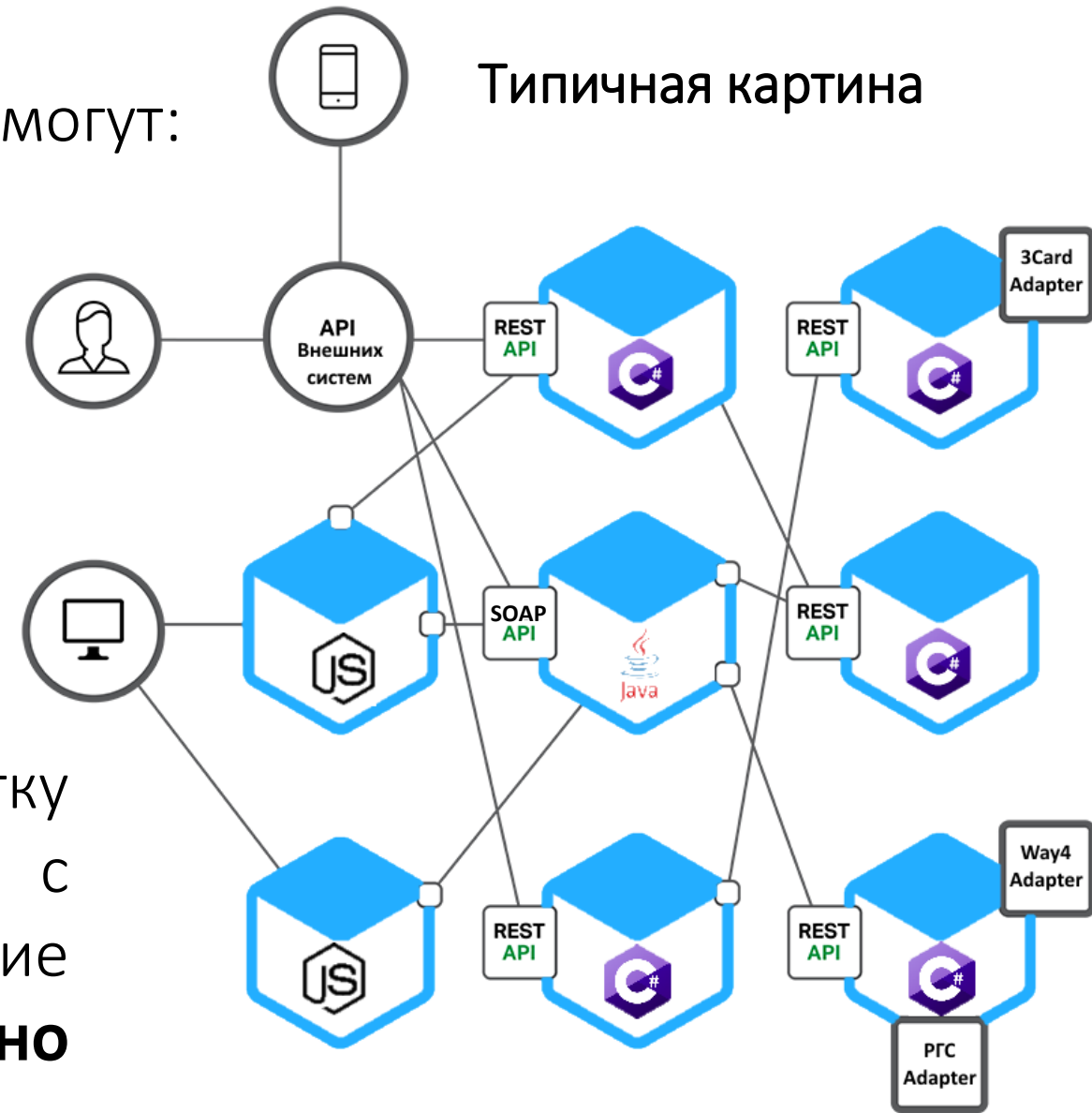
# Как микросервисы должны взаимодействовать?

Микросервисы в крупном приложении могут:

- Быть реализованы разными людьми
- Использовать разные технологии
- Не иметь единого стандарта API

Как не допустить «казуса строительства Вавилонской башни?»

Чтобы избежать издержек на доработку новых сервисов и интеграцию с внешними системами, взаимодействие внутри и снаружи платформы **должно быть унифицировано.**





# Взаимодействие – это операции над данными

Основная идея - чтобы через что-то взаимодействовать это что-то нужно сначала описать.  
Возникает понятие сущности.

Пример - сущность “клиент” с точки зрения учетной системы:

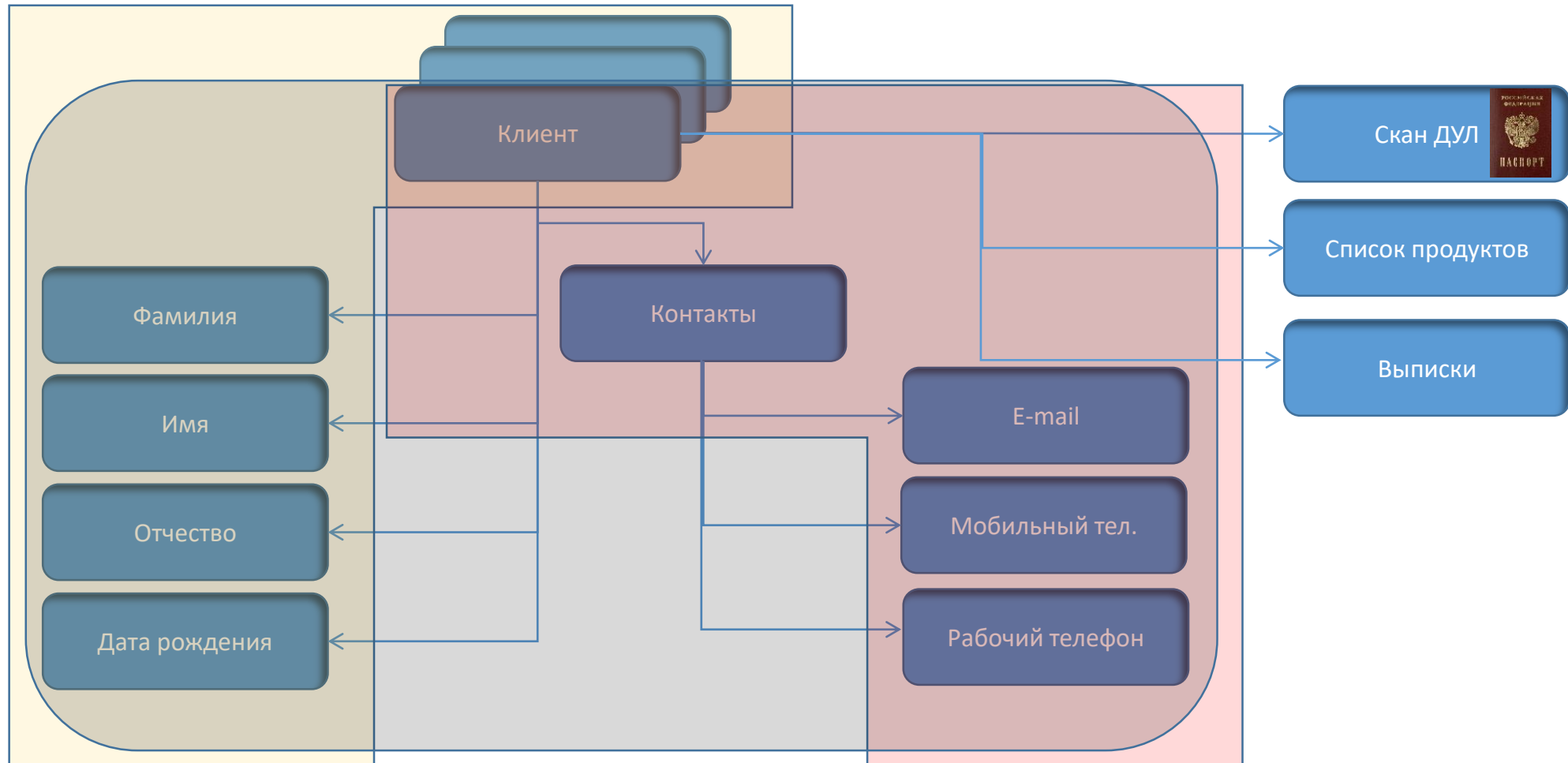


- Фамилия
- Имя
- Отчество
- Дата рождения
- Серия и Номер паспорта
- Сканы ДУЛ
- Адрес регистрации
- Адрес для корреспонденции
- Адрес проживания
- Место работы
- Номер телефона
- Финансовый номер телефона
- Место работы
- ...



В реальной системе у сущности могут быть многие десятки свойств, а в некоторых случаях и сотни

# Проекции данных



Любая сущность (модель) описывается полным набором своих атрибутов и каждый потребитель интересуется проекцией этой модели на свою предметную область. Больше потребителей – больше проекций.

# Какие варианты решения?

1. Давайте всегда отдавать полную модель данных, а клиент на своей стороне будет использовать нужное ему подмножество!
2. Давайте разобьем полную модель на более мелкие, а их на еще более мелкие, а клиент пусть запрашивает нужные ему части!
3. Давайте предоставим каждому клиенту удобный для него (свой) срез данных, чтобы он получал только то, что ему нужно!

# Отдаём полную модель => Overfetching

```
179 <xs:complexType name="BundleHeadDtoInput">...
192 </xs:complexType>
193
194 <xs:complexType name="ProductsRequestInput">...
202 </xs:complexType>
203
204 <xs:complexType name="GetTermsByProductTypeRequestInput">...
211 </xs:complexType>
212
213 <xs:complexType name="CreateBundleRequestInput">...
219 </xs:complexType>
220
221 <xs:complexType name="SendBundleRequestInput">...
227 </xs:complexType>
228
229 <xs:complexType name="UpdateBundleRequestInput">
230 <xs:sequence>
231 <xs:element name="bundleId" type="xs:long" minOccurs="1" maxOccurs="1" />
232 <xs:element name="bundleHead" type="BundleHeadDtoInput" minOccurs="1" maxOccurs="1" />
233 <xs:element name="products" type="ProductUpdateDtoInput" minOccurs="1" maxOccurs="unbounded" />
234 </xs:sequence>
235 </xs:complexType>
236
237 <xs:complexType name="ProductUpdateDtoInput">...
245 </xs:complexType>
246
247
248 <xs:simpleType name="PeriodUnit" final="restriction" >...
256 </xs:simpleType>
257
258 <xs:simpleType name="ValidationErrorType" final="restriction" >...
269 </xs:simpleType>
270
271 <xs:simpleType name="ProductType" final="restriction" >...
280 </xs:simpleType>
281
282 <xs:simpleType name="TimeUnit" final="restriction" >...
290 </xs:simpleType>
291
292 <xs:simpleType name="BundleState" final="restriction" >...
300 </xs:simpleType>
301
302 </xs:schema>
```



Нерациональное использование сети и излишняя нагрузка на CPU при обработке

# Канонические (полные) модели данных создают проблемы

- Потенциально создают проблему overfetching - эволюция/рост модели ведёт к деградации клиентов
- Решение проблемы через опциональные поля на самом деле создает семейство неявных контрактов – методы могут возвращать различные неявные проекции
- Любая неопределенность создает проблемы при анализе, археологии и эксплуатации

# Отдаём мелкие модели => Underfetching

The screenshot shows the SITI.LINK website, an electronics retailer. The page features a sidebar with category icons, a main banner for HONOR 20, and a 'Топ продаж' (Top Sales) section with three smartphones: Redmi Note 7, ZenFone MAX M2, and ZenFone Max Pro M1. Overlaid on the right is the Chrome DevTools Network tab, which is highlighted with a red rectangle. The Network tab shows a list of 244 requests, including many small image files (e.g., 3.7 KB, 4.3 KB, 1.1 KB) and scripts. A red box highlights the 'Filter' and 'Type' dropdowns, and another red box highlights the summary at the bottom: '244 requests | 127 KB transferred | Finish: 4.71 s | DOMContentLoaded: 2.21 s | Load: 2.25 s'. The 'Filter' dropdown is set to 'All', and the 'Type' dropdown is set to 'XHR'. The 'Finish' time is 4.71 s, 'DOMContentLoaded' is 2.21 s, and 'Load' is 2.25 s.

Name	Status	Type	Initiator	Size	Time	Waterfall
?random=1561569...	200	script	conversion...	1.9 KB	320 ms	
dis.aspx?p=30891&...	200	docu...	Id.js:1	384 B	642 ms	
325522_v01_s.jpg	200	jpeg	Other	3.7 KB	281 ms	
1032077_v01_s.jpg	200	jpeg	Other	4.3 KB	279 ms	
csc-event?p=0%3A...	200	gif	Other	373 B	422 ms	
collect?tid=UA-558...	200	gif	VM652:3	469 B	261 ms	
underscore.js	200	script	forms.js?v...	(fro...	0 ms	
jquery.js	200	script	forms.js?v...	(fro...	0 ms	
raven.js	200	script	forms.js?v...	(fro...	0 ms	
?vid=1358580423	200	xhr	jquery.js:1	1.1 KB	253 ms	
?random=1003333...	302	gif	Other	1.9 KB	260 ms	
?type=pageview&u...	200	gif	Other	629 B	255 ms	
?random=1003333...	302	gif	googlelead...	1.8 KB	195 ms	
1072517_v01_s.jpg	200	jpeg	Other	(fro...	3 ms	
1089214_v01_s.jpg	200	jpeg	Other	(fro...	4 ms	
1135601_v01_s.jpg	200	jpeg	Other	(fro...	4 ms	
?random=1003333...	200	gif	www.goo...	574 B	109 ms	
collect?tid=UA-558...	(pen...	text/...	VM652:3	0 B	Pend...	
collect?t=dc&aip=1...	(pen...	Other		0 B	Pend...	

244 requests | 127 KB transferred | Finish: 4.71 s | DOMContentLoaded: 2.21 s | Load: 2.25 s

Нерациональное использование сети и излишняя нагрузка на CPU при обработке, ОПЯТЬ!



# Отдаём каждому клиенту свой срез данных

- Если используем REST/SOAP, каждая проекция требует кодирование
- С ростом количества проекций (клиентов) сложность поддержки растёт лавинообразно
- Фиксированные проекции создают серьезные проблемы при планировании
- Скрытая на уровне менеджмента зависимость команд создает значительно отличные от нуля издержки
- Найти хороший баланс фиксированных проекций данных при росте числа потребителей — как взятие интегралов. Это искусство и не все берутся.

# Что будем делать





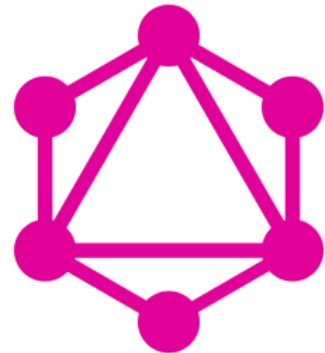


**Секрет успеха поставщика заключается в том, чтобы обеспечить потребителей качественным товаром.**

**Ну и еще важно не пускаться во все тяжкие с нарушением обратной совместимости.**

# Какие **ещё** варианты решения?

4. Давайте дадим каждому клиенту возможность самому сказать, что ему и только ему нужно! При этом он должен знать, из чего ему выбрать. Пусть мы будем для него супермаркетом, где он за 1 раз сможет получить все данные в свою продуктовую тележку.



GraphQL



# Что мы ожидали от использования GraphQL



- Уменьшение издержек на управление бэклогом команд. Остается только расширение контрактов



- Решение проблемы over- & underfetching, снижение нагрузки



- Ускорение и стандартизация процесса интеграции с внешними системами



- Строго типизированное потребление упростит рефакторинг

# ЧАСТЬ 1

## Введение в GraphQL

# Что такое GraphQL в двух словах

GraphQL (Graph Query Language) - это синтаксис, который описывает, как запрашивать данные.

В основном, используется клиентом для загрузки данных с сервера, выполняя роль языка API.


Официальная спецификация GraphQL

<https://github.com/graphql/graphql-spec>

- \* GraphQL не имеет ничего общего с графовыми базами данных и базами данных вообще
- \*\* кроме получения данных можно их менять (mutation) и подписываться на изменения (subscription)
- \*\*\* строго говоря, не только синтаксиса, но и семантики, валидации, выполнения и т.п.
- \*\*\*\* GitHub использует GraphQL public API v4 после REST API v3

# GraphQL – довольно молодая технология

- Создан в Facebook как внутренняя разработка в **2012** году
- Первая спецификация как открытого стандарта - **июль 2015**
- Пятая release-спецификация - **июнь 2018**
- Перешёл под патронаж GraphQL Foundation **7 ноября 2018**
- Working Draft шестой спецификации - **11 ноября 2019**



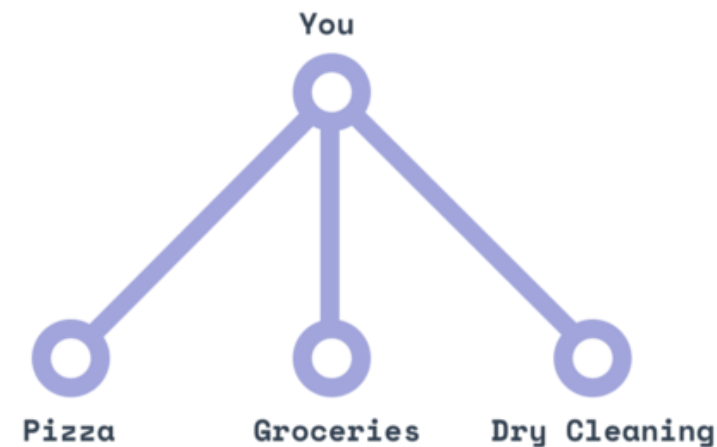
На 28 ноября 2019 технология проделала путь длиной 7 лет

# Основные характеристики GraphQL

- Open Source – спецификация + реализации под популярные платформы
- Кроссплатформенный язык
- Позволяет клиенту точно указать, какие данные ему нужны - клиент получает **ТОЛЬКО** то, что он заказал, и ничего больше  
это значительно снижает объем передаваемого трафика
- Облегчает агрегацию данных из нескольких источников, выступая в роли API Gateway
- Использует систему типов для описания и валидации данных, множество типов составляет схему (Schema, SDL)
- Позволяет гибко строить запросы, сохраняя при этом **жёстко формализованные** контракты
- Схема (контракты) доступна через интроспекцию и может публиковаться:
  - для машинной обработки
  - как часть документации, так как схема самодокументируема

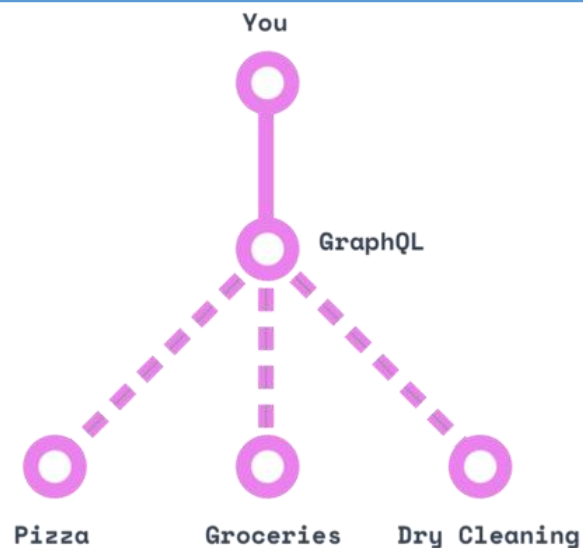


# GraphQL vs REST



REST - много "глупых" / "атомарных" endpoint'ов:

- Набор возвращаемых полей жестко задан
- Клиент при необходимости сам собирает (агрегирует) требуемый набор данных
- Необходимость версионирования ресурсов
- Клиент не знает схему ресурсов (помогает Swagger)
- Структура возврата может поменяться → ошибки/проблемы производительности



GraphQL - один "умный" / "комплексный" endpoint:

- Определена схема возвращаемых данных, но не конечный набор полей
- Унифицирует разнородную природу различных источников данных
- Нет версионирования, есть @deprecated
- Клиенту известна полная схема всех данных
- Структура возврата **не может** поменяться, она фиксирована **самим клиентом**

\* Нужно заметить, что как параметризацию набора возвращаемых полей, так и API Gateway, средства документирования **можно** обеспечить и в рамках REST. Разница в том, что в GraphQL всё это – **сущность первого рода, предоставляемая из коробки**. GraphQL и REST могут сосуществовать в одном приложении, возможен плавный переход.



# API Design: GraphQL vs REST

## REST

В REST вы должны получить понимание потребностей ваших пользователей **ДО** реализации API

## GraphQL

С помощью GraphQL вы можете **отложить** момент понимания того, как пользователи используют ваши API, до тех пор, пока вы не начнете профилировать запросы, оценивая их сложность и выявляя медленные запросы

# Пример запроса REST и GraphQL

Задача – получить с сервера некоторый ресурс так, чтобы получить лишь указанное подмножество всех полей.

Так это может выглядеть в REST:

**GET /users/135?fields=id,name,isViewer,avatar\_\_size\_\_50,avatar.uri,avatar.width,avatar.height**

А вот так в GraphQL:

```
{
  user(id: 135) {
    id,
    name,
    isViewer,
    avatar(size: 50) {
      uri,
      width,
      height
    }
  }
}
```

Вопросы к REST:

- 1) Как быть с вложенными объектами – 2,3...10 уровня?
- 2) Как получить атрибут ресурса, на который мы получили ссылку (ID) в теле ответа?
- 3) Как быть с параметризацией отдельных полей? Ведь параметры тоже могут быть объектами...
- 4) Какие символы выбирать в качестве разделителей?
- 5) Это что же, писать ещё один парсер?

Решаемы ли эти вопросы в REST? Конечно!

Но хочется ли их решать (снова)? Вряд ли... GraphQL дает ответы и готовое решение.

# Вопросы за рамками GraphQL

Следующие вопросы находятся за рамками\* непосредственно спецификации самого GraphQL:

- Конкретный вид транспорта (HTTP, TCP, AMQP, JMS, голубиная почта, ...)
- Аутентификация/авторизация
- Контроль потребляемых клиентом/сервером ресурсов (rate limiting)
- Paging – страничная разбивка результатов
- Интернационализация / локализация
- Передача файлов и бинарных данных
- Кэширование

\*но которые тем не менее достаточно **хорошо проработаны** средствами вокруг спецификации GraphQL

## ЧАСТЬ 2

# Основные строительные блоки GraphQL

# Составляющие GraphQL

Будут затронуты лишь основные вопросы, больше информации на <https://graphql.org/learn/> и странице спецификации GraphQL <https://graphql.github.io/graphql-spec/>

---

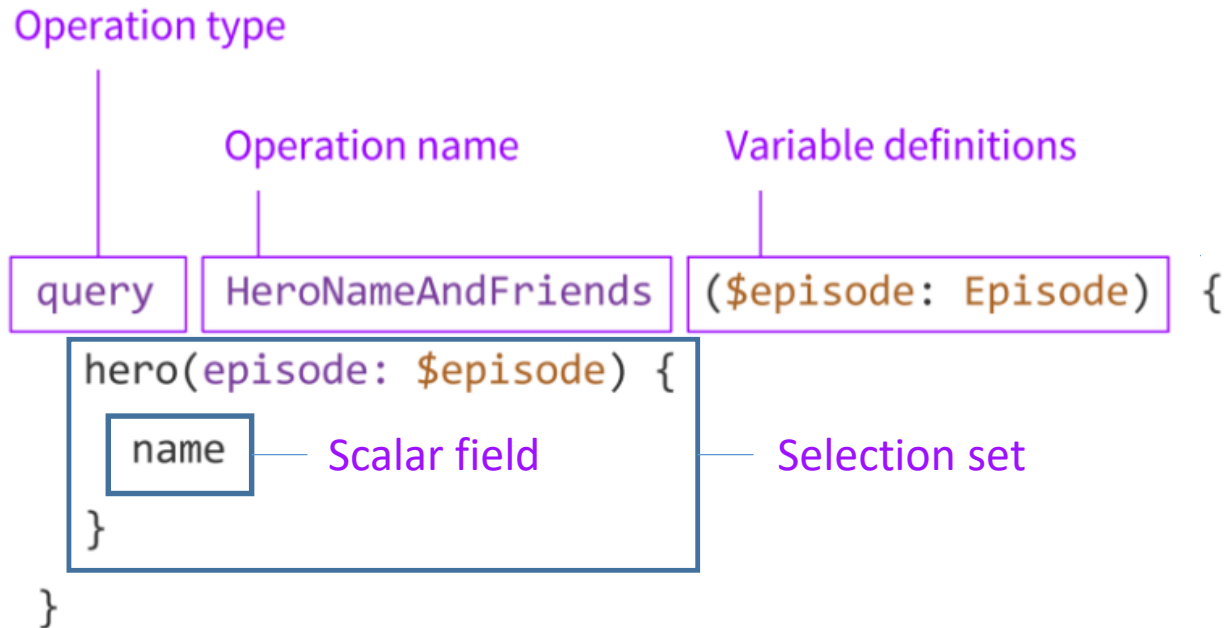
## О чем поговорим

Строительный блок	Описание
<b>Query</b>	Запросы к GraphQL серверу
<b>Field</b>	Поля данных – то, что запрашивается
<b>Type</b>	Типы данных в GraphQL
<b>SDL</b>	Schema Definition Language (аналог WSDL для GraphQL)
<b>Argument</b>	Аргументы полей
<b>Alias</b>	Псевдонимы полей
<b>Mutation</b>	Мутации – способ изменения данных
<b>Subscription</b>	Подписки на изменения данных
<b>Resolver</b>	Выполняют реальную работу по получению данных
<b>Directive</b>	Директивы – декларация произвольной метаданных

# Queries - запросы

Query – аналогия GET для REST: **query** { **stuff** } и имеет вложенную природу **query** { **stuff** { **eggs** **shirt** **pizza** } }

Структура запроса GraphQL:



Пример запроса GraphQL:

```
query {  
  posts { # комментарий: это массив  
    title  
    body  
    author { # мы можем пойти глубже  
      name  
      avatarUrl  
      profileUrl  
    }  
  }  
}
```

Составные поля

Простые поля

# Fields - поля

Просто говоря, весь GraphQL состоит в том, чтобы запросить определенные поля у нужных объектов. Поля возвращаются в ответе в формате JSON.

Запрос:

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

← родительское поле  
← дочернее поле

Ответ:

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

Запрос имеет точно такую же форму как и результат (data – объект-контейнер, ещё есть errors). Поле может быть массивом.

Запрос:

```
{
  hero {
    name
    # это комментарий
    friends {
      name
    }
  }
}
```

← поле 3-его уровня вложенности

Ответ:

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

# Types - система типов GraphQL

Поля в GraphQL существуют не сами по себе, а в рамках некоторого типа, то есть поле принадлежит типу. Всего существует 8 типов полей, из которых 2 – специальные типы-модификаторы:

- **1. Scalar** – скаляры, примитивные типы Int/String/Boolean/Float (как в JavaScript), не имеющие полей
- **2. Object** – составные типы (структуры), имеют набор полей
- **3. Union** – позволяет указать поле как возвращающее один из набора разнородных типов
- **4. Interface** – задает общий набор полей для типов-наследников
- **5. Enum** – перечисление (особый скаляр), задает фиксированный набор вариантов выбора
- **6. Input** – составные типы для входных аргументов запросов (аналогия Object)
- **7. List []** – тип-модификатор, применённый к другому типу, позволяет указать массив этого типа
- **8. NonNull !** – тип-модификатор, примененный к другому типу, позволяет требовать обязательное наличие поля этого типа, **так как по умолчанию все поля в GraphQL необязательные**

Пример типа с двумя полями:

Обязательное строковое поле



Не обязательное поле-массив, элементы  
которого - обязательные значения другого  
типа Episode



```
type Character {  
  name: String!  
  appearsIn: [Episode!]  
}
```



# SDL – язык описания GraphQL схем

```
directive @isAuthenticated on FIELD_DEFINITION
directive @hasScope(scope: [String]) on
FIELD_DEFINITION
```

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

# A mechanical creature in the Star Wars universe.

```
type Droid implements Character {
  id: String!
  name: String
  friends: [Character]
  appearsIn: [Episode]
  primaryFunction: String
}
```

# One of the films in the Star Wars Trilogy.

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

```
type Human implements Character {
  id: String!
  name: String
  friends: [Character]
  appearsIn: [Episode]
  homePlanet: String
}
```

```
input HumanInput {
  name: String!
  homePlanet: String
}
```

```
type Query {
  hero: Character @isAuthenticated
  human(id: String!): Human
  droid(id: String!): Droid
}
```

```
type Mutation {
  title : String
  createHuman(human: HumanInput!): Human @hasScope(scope:
["admin"])
}
```



# Arguments - аргументы


Аргументы позволяют параметризовать то или иное поле (поле – аналогия метода в привычном понимании)

Пример получения персонажа по его идентификатору:

Запрос:

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

Аргумент и его значение



Ответ:

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

Аргументы могут иметь значение по умолчанию, определённое на поле внутри типа, тогда клиенту не обязательно задавать в запросе этот аргумент


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

# Aliases – псевдонимы полей

Позволяют запросить одно и то же поле несколько раз, как правило с разными аргументами.

Запрос:

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



Клиент использовал в запросе 2 псевдонима

Ответ:

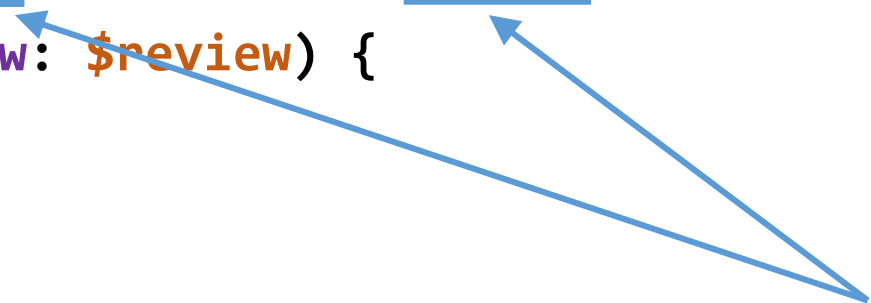
```
{  
  "data": {  
    "empireHero": {  
      "name": "Luke Skywalker",  
    },  
    "jediHero": {  
      "name": "R2-D2",  
    }  
  }  
}
```

Можно рассматривать псевдонимы как еще один способ повышения производительности через сокращение числа обращений к серверу.

# Mutations – способ изменения данных

Mutation – аналогия POST для REST. Мутация по синтаксису полностью аналогична query, просто использует ключевое слово **mutation** вместо **query**.

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



Входные аргументы мутации

Мутация точно так же как query способна возвращать результат в виде набора полей:

```
{  
  "data": {  
    "createReview": {  
      "stars": 5,  
      "commentary": "This is a great movie!"  
    }  
  }  
}
```

# Subscriptions – подписка на изменение данных

Subscription – аналог Push-уведомлений, позволяющий получать обратную связь от сервера только по тем объектам, в изменениях которых заинтересован клиент. Сервер начинает потоком передавать клиенту события (данные).

Пример подписки на сообщение в чате:

```
subscription NewMessages {  
  newMessage(roomId: 123) {  
    sender  
    text  
  }  
}
```

Пример события от сервера:

```
{  
  "data": {  
    "newMessage": {  
      "sender": "AZorin",  
      "text": "Pizza has arrived!"  
    }  
  }  
}
```

Подписки позволяют добиться нескольких вещей:

- 1) Для клиента – **интерактивность** при взаимодействии с сервером, данные приходят «сами и сразу»
- 2) Для сервера – **снижение нагрузки** на обслуживание polling-клиентов
- 3) Для обеих сторон – **снижение объема** передаваемого трафика

# Resolvers – рабочая лошадка GraphQL

Resolver – это код, поэтому тут ничего не будет, как я и обещал 😊

Главное – GraphQL-сервер запускает resolver'ы ТОЛЬКО для тех полей, которые запросил клиент, и запускает их на выполнение ПАРАЛЛЕЛЬНО для query и ПОСЛЕДОВАТЕЛЬНО для mutation.

# Директивы

Директивы – это способ расширения схемы без вмешательства в её графовую структуру.

- Могут быть подключены к любому элементу схемы, даже к самим директивам. 😊
- Могут быть предназначены как сугубо для сервера или для клиента, так и для обоих.
- Могут выступать как в виде пассивного блока, декларирующего дополнительную метainформацию на элементе схемы, так и в активном виде, встраиваясь в цепочку обработки запроса.

---

Могут быть использованы для:

- 1) Аутентификации - **directive @isAuthenticated on FIELD\_DEFINITION**
- 2) Авторизации - **directive @hasScope(scope: [String]) on FIELD\_DEFINITION**
- 3) Ограничения длины входных/выходных данных - **@length(max: Int!)**
- 4) Маскирования, шифрования и другой произвольной модификации ответа
- 5) Условного включения/исключения полей из ответа сервера - **@skip / @include**
- 6) Упорядочения/группировки обработчиков полей на сервере
- 7) Интернационализации (локализации) API
- 8) Постепенного вывода из эксплуатации отдельных частей схемы - **@deprecated(reason: "use `total`")**

\*Спецификация GraphQL имеет пробел в разделе, задающем возможность возврата информации о директивах через интроспекцию, мы ведём работу над расширением спецификации.

# ЧАСТЬ 3

## Экосистема GraphQL (обзорно)



# Спецификация

- <https://github.com/graphql/graphql-spec>
- <https://graphql.github.io/graphql-spec/>

Спецификация простым языком описывает практически всё, что требуется 99% разработчиков, чтобы понять и начать работать с GraphQL на любой платформе.

Спецификация выложена в открытом доступе на GitHub, регулярно обновляется, ведутся дискуссии, оказывается помощь по возникающим вопросам.

При работе над ЕФР мы предлагали интересующие нас поправки и предложения к спецификации. Часть уже принята, часть находится в процессе обсуждения.

# Обучающие материалы

<https://graphql.org/>

Пошаговое введение в GraphQL с интерактивными примерами.

## Introduction

## Queries and Mutations

Fields

Arguments

Aliases

Fragments

Operation Name

Variables

Directives

Mutations

Inline Fragments

## Schemas and Types

Type System

Type Language

Object Types and Fields

Arguments

The Query and Mutation Types

Scalar Types

Enumeration Types

Lists and Non-Null

Interfaces

Union Types

Input Types

## Validation

## Execution

## Introspection

## BEST PRACTICES

## Introduction

## Thinking in Graphs

## Serving over HTTP

## Authorization

## Pagination

## Caching

# Серверы GraphQL

- JS - <https://github.com/graphql/graphql-js>
- .NET - <https://github.com/graphql-dotnet/graphql-dotnet>
- Java - <https://github.com/graphql-java/graphql-java>
- Node - <https://www.apollographql.com/>
- И многие, многие другие

Полный список тут - <https://graphql.org/code/#server-libraries>,  
представлены десятки проектов под популярные платформы  
разработки

# Клиенты GraphQL

- JS - <https://graphql.org/code/#javascript-1>
- .NET - <https://github.com/graphql-dotnet/graphql-client>
- Java - <https://github.com/apollographql/apollo-android>
- Python - <https://github.com/graphql-python/gql>
- И многие, многие другие

Полный список тут - <https://graphql.org/code/#graphql-clients>,  
представлены десятки проектов под популярные платформы  
разработки

Popular Servers

Name	Language	★
<a href="#">apollo-server</a>	JavaScript	8500
<a href="#">graphql</a>	Go	5700
<a href="#">Graphene</a>	Python	5200
<a href="#">graphql-yoga</a>	TypeScript	5100
<a href="#">express-graphql</a>	JavaScript	4900
<a href="#">graphql-ruby</a>	Ruby	4000
<a href="#">graphql-java</a>	Java	3800
<a href="#">GraphQL for .NET</a>	C#/.NET	3800
<a href="#">gqlgen</a>	Go	3600
<a href="#">graphql-php</a>	PHP	3400
<a href="#">absinthe</a>	Elixir	3000
<a href="#">graphql-go</a>	Go	3000
<a href="#">Juniper</a>	Rust	1900
<a href="#">Sangria</a>	Scala	1600
<a href="#">Iacinia</a>	Clojure	1300
<a href="#">Thunder</a>	Go	914
<a href="#">graphql-elixir</a>	Elixir	837
<a href="#">Siler</a>	PHP	837
<a href="#">GraphQL.Net</a>	C#/.NET	795
<a href="#">Hot Chocolate</a>	C#	736
<a href="#">koa-graphql</a>	JavaScript	719

♥ Web API

♥ Backend

Popular Clients

Name	Language	★
<a href="#">Relay</a>	JavaScript	13800
<a href="#">apollo-client</a>	TypeScript	12600
<a href="#">graphiql</a>	JavaScript	9600
<a href="#">GraphQL Playground</a>	TypeScript	5400
<a href="#">graphql-editor</a>	TypeScript	4100
<a href="#">urql</a>	TypeScript	3800
<a href="#">Apollo iOS</a>	Swift	2200
<a href="#">graphql-request</a>	TypeScript	2000
<a href="#">apollo-android</a>	Java	2000
<a href="#">Altair</a>	TypeScript	1800
<a href="#">Lokka</a>	JavaScript	1500
<a href="#">graphql-hooks</a>	JavaScript	1000
<a href="#">machinebox/graphql</a>	Go	463
<a href="#">nanographql</a>	JavaScript	357
<a href="#">GQL</a>	Python	304
<a href="#">Graphql</a>	Go	294
<a href="#">re-graph</a>	Clojure	260
<a href="#">Grafoo</a>	TypeScript	250
<a href="#">GraphQL.Client</a>	C#	244
<a href="#">nodes</a>	Java	208
<a href="#">sgqlc</a>	Python	160

♥ Frontend

♥ Backend + generated

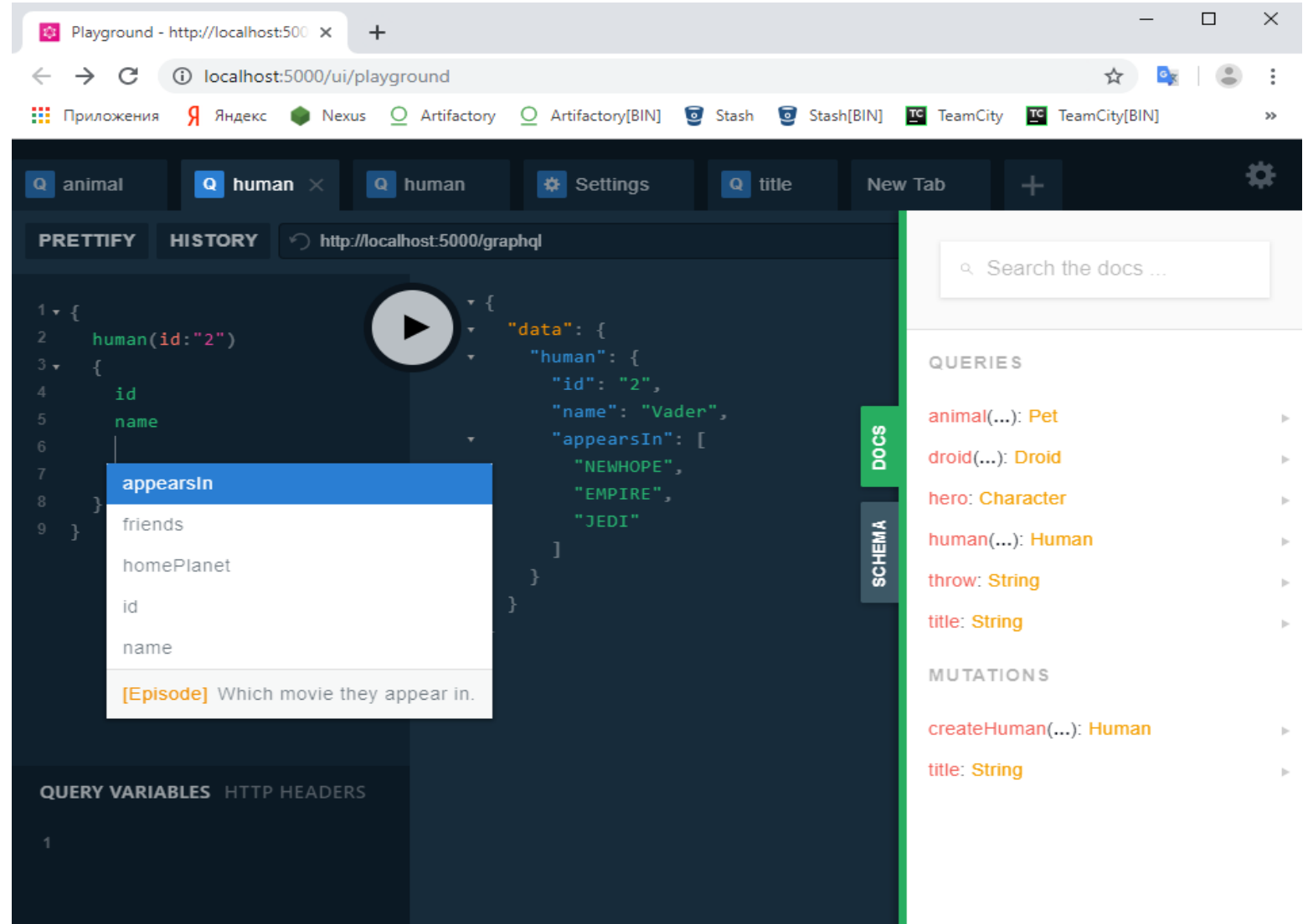
Сервер и клиент на любой вкус и цвет

# In-browser IDE для GraphQL

GraphQL Playground <https://github.com/prisma/graphql-playground>

Аналог Postman, только для GraphQL:

- Подключается за 1 клик
- Многовкладочный интерфейс
- Настройки интерфейса и поведения
- Написание/выполнение запросов
- Подсветка синтаксиса
- Подсказки редактора
- Форматирование запроса/ответа
- История запросов
- Работа с переменными запроса
- Задание HTTP заголовков
- Метрики времени выполнения
- Поддержка интроспекции
- Интерактивная документация



# GraphQL

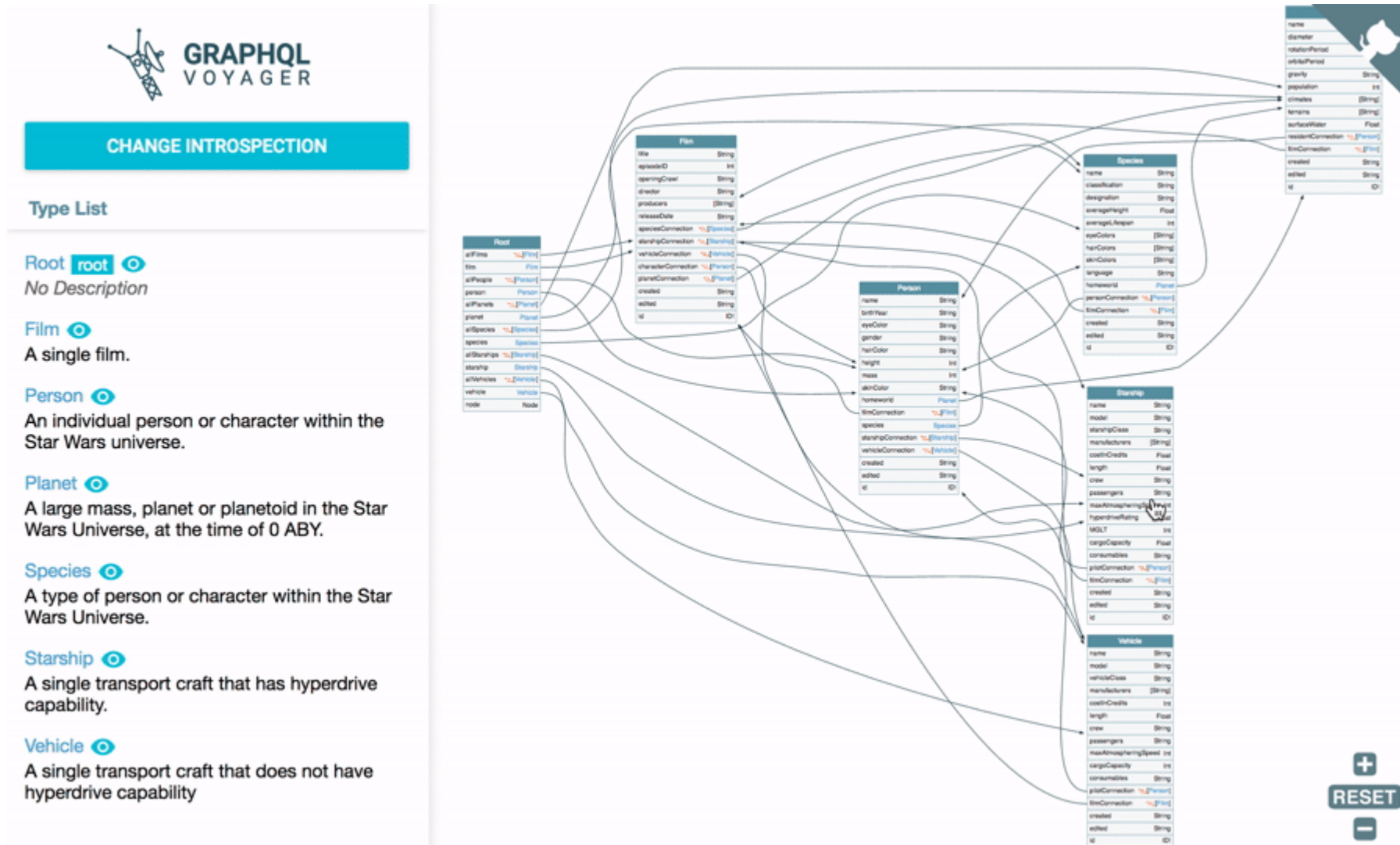
<https://github.com/graphql/graphql>

1



# Voyager – интерактивный граф схемы

<https://github.com/APIs-guru/graphql-voyager>



Полезный инструмент для:

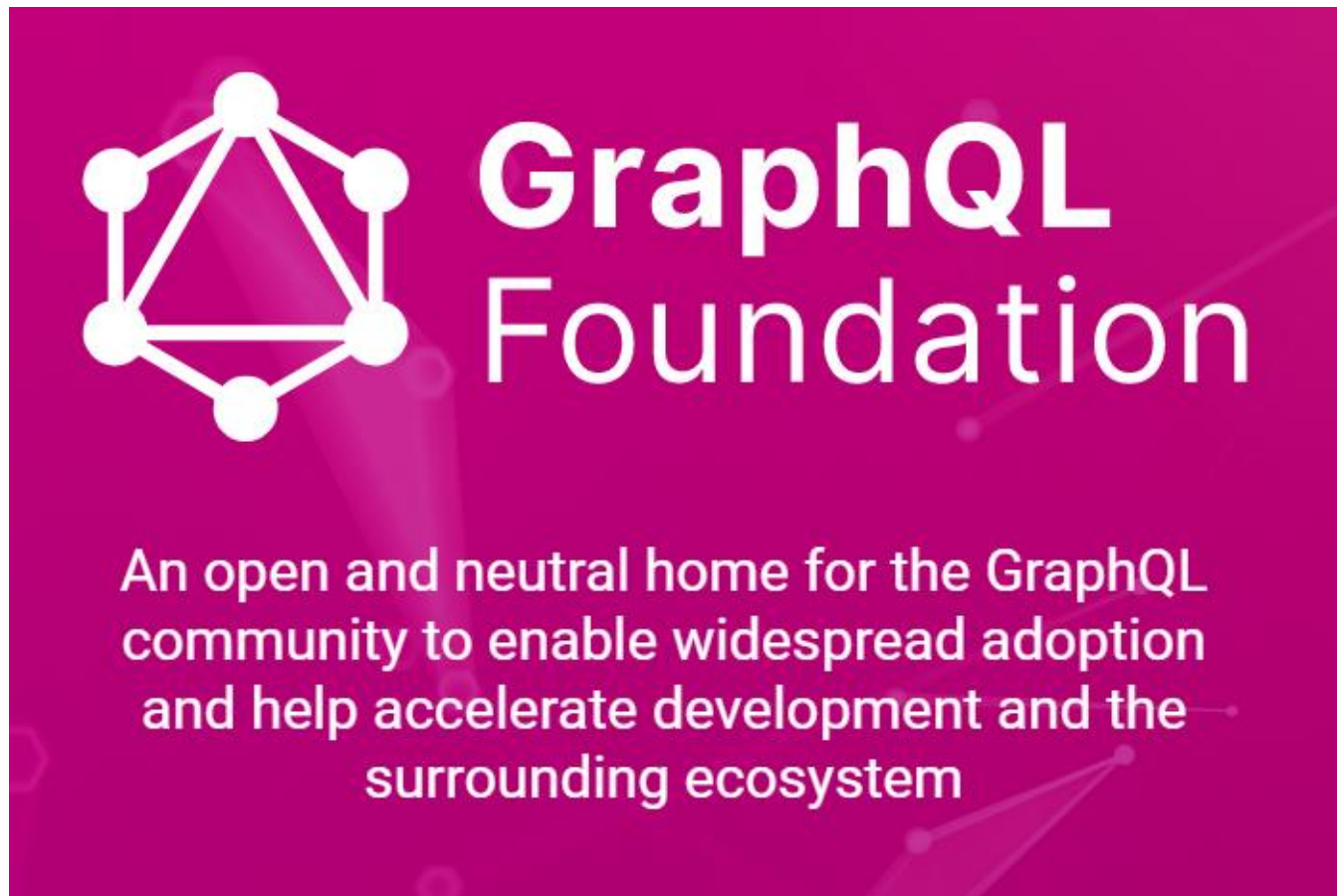
- Разработчика
- Аналитика
- Поддержки
- Клиентов сервера

Подключается в 1 клик,  
содержит встроенную  
актуальную документацию.



# GraphQL Foundation

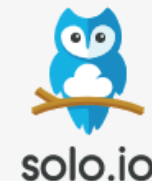
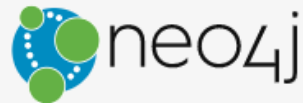
<https://gql.foundation/>



GraphQL Foundation - это независимая организация, основанная компаниями, занимающимися разработкой технологий и приложений. GraphQL Foundation поощряет вклад, руководство и совместное инвестирование от широкой группы в независимые от поставщиков события, документацию, инструменты и поддержку GraphQL.

# GraphQL Foundation Members

<https://gql.foundation/members/>



Это только участники, а пользователей-организаций гораздо больше, например, Netflix.

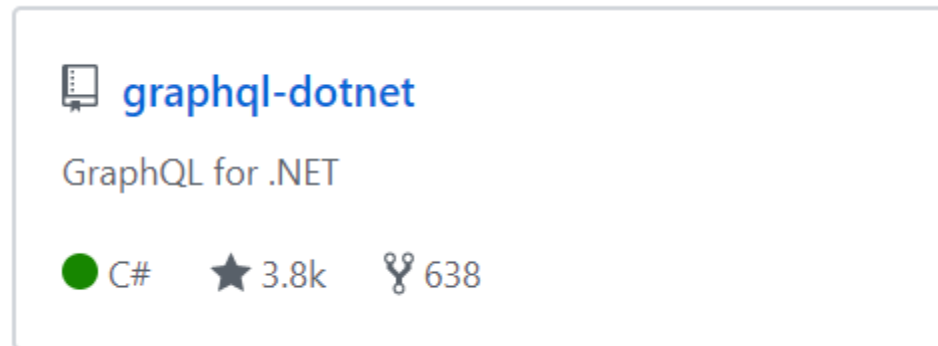
# GraphQL RFC Working Group

<https://github.com/graphql/graphql-wg>

GraphQL WG (Рабочая группа) - это ежемесячное 3-часовое собрание в формате видеоконференции разработчиков, сопровождающих саму спецификацию, а также популярные библиотеки и инструменты GraphQL.

Основная задача GraphQL WG - обсудить и согласовать предлагаемые дополнения к спецификации GraphQL. Кроме того, группа может обсуждать и сотрудничать по другим соответствующим техническим темам, касающимся основных проектов GraphQL.

Нами было получено предложение участия в GraphQL WG благодаря нашему вкладу и регулярной поддержке проекта graphql-dotnet – самого популярного на данный момент GraphQL сервера для платформы .NET на GitHub.



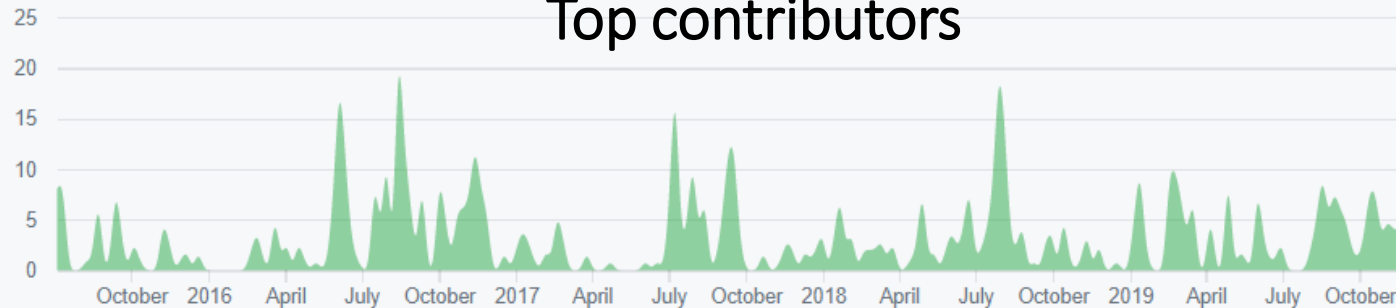
# GraphQL .NET Server

<https://github.com/graphql-dotnet/graphql-dotnet>

Мы являемся частью глобального GraphQL Community и членом организации graphql-dotnet, принимая участие в развитии GraphQL сервера и сопутствующих инструментов для платформы .NET на GitHub.

Contributions to master, excluding merge commits

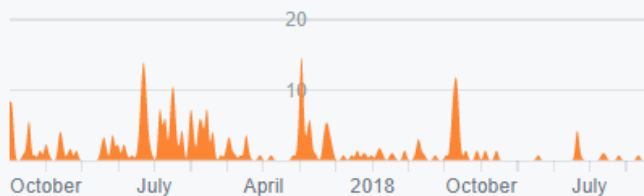
## Top contributors



**joemcbride**

#1

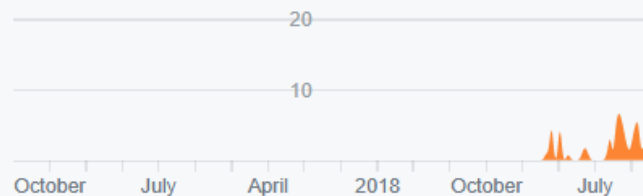
356 commits 134,983 ++ 86,455 --



**sungam3r**

#2

69 commits 8,639 ++ 3,245 --



## GraphQL Core Engine



**graphql-dotnet**

GraphQL for .NET

C#

★ 3.8k

🔗 638

## GraphQL ASP.NET Core Hosting/Transport



**server**

ASP.NET Core Server + WebSockets Transport

C#

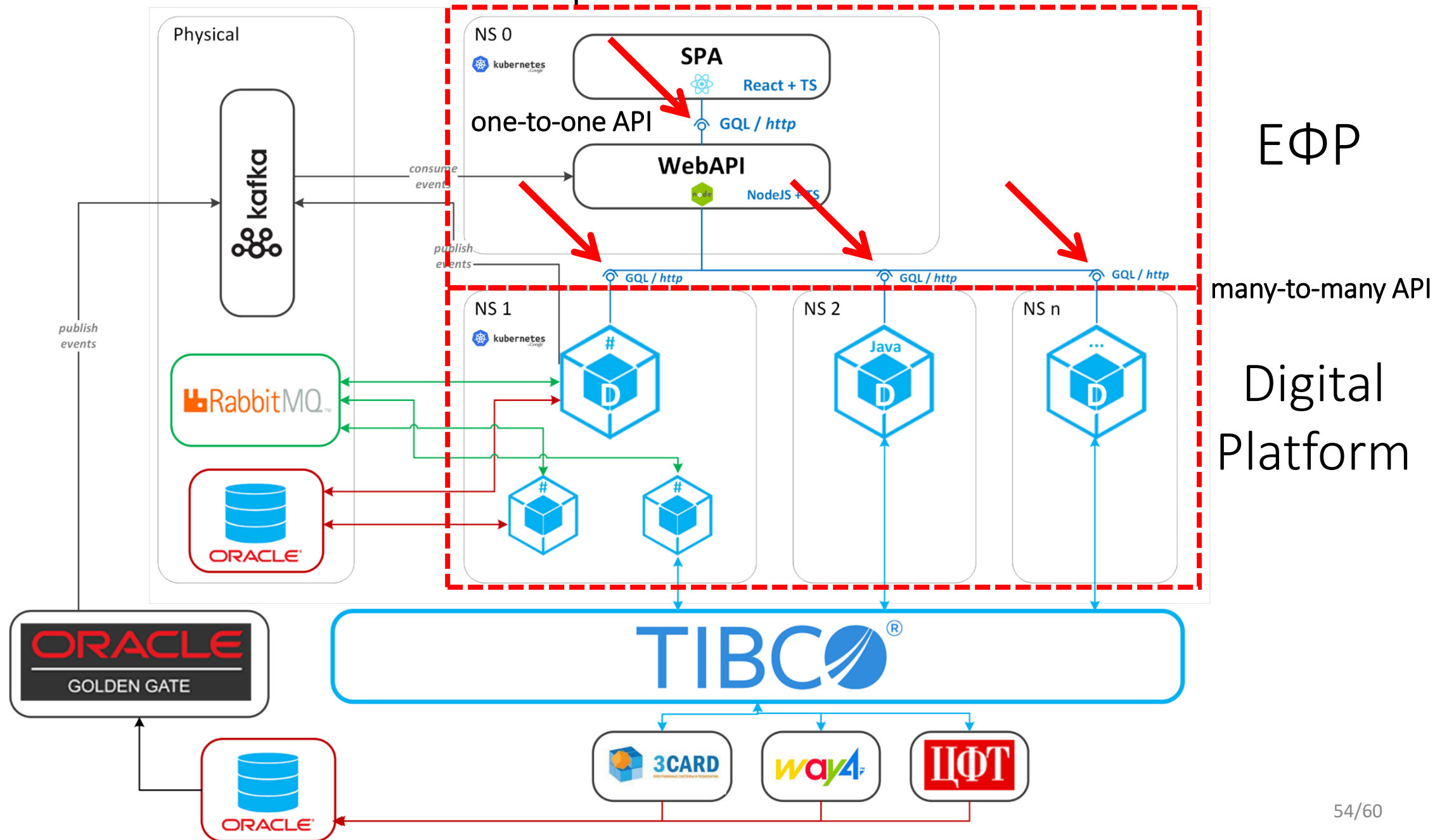
★ 238

🔗 81

# ЧАСТЬ 4

## GraphQL в ЕФР

# Общая схема ЕФР



# Paradigm Shift



First Wave (2000)

## CUSTOMER-SPECIFIC APIs

**one-to-one:** few large established customers



Second Wave (2010)

## GENERIC APIs

**one-to-many:** many mid- or small- size customers



Third Wave (2020)

## AUTONOMOUS APIs

**many-to-many &  
machine-to-machine:**  
automatic, later  
autonomous APIs

# Типы сервисов в ЕФР (.NET Core)

## Доменные сервисы

- Для **внешнего** потребления
- Обеспечивают высокоуровневый API
- Используемый транспорт – HTTP (ASP.NET Core)
- Экспортируют GraphQL схему в качестве контрактов для работы с ними
- Собственный .NET SDK для GraphQL
- Встроенный Playground на тест-контуре
- Встроенный Voyager на тест-контуре

## Технические сервисы

- Для **внутреннего** потребления
- Обеспечивают продвинутые сценарии взаимодействия: RPC, FireAndForget, Events
- Используемый транспорт – AMQP (RabbitMQ)
- Экспортируют JSON схему внутреннего формата для работы с ними
- Собственный .NET SDK для RabbitMQ

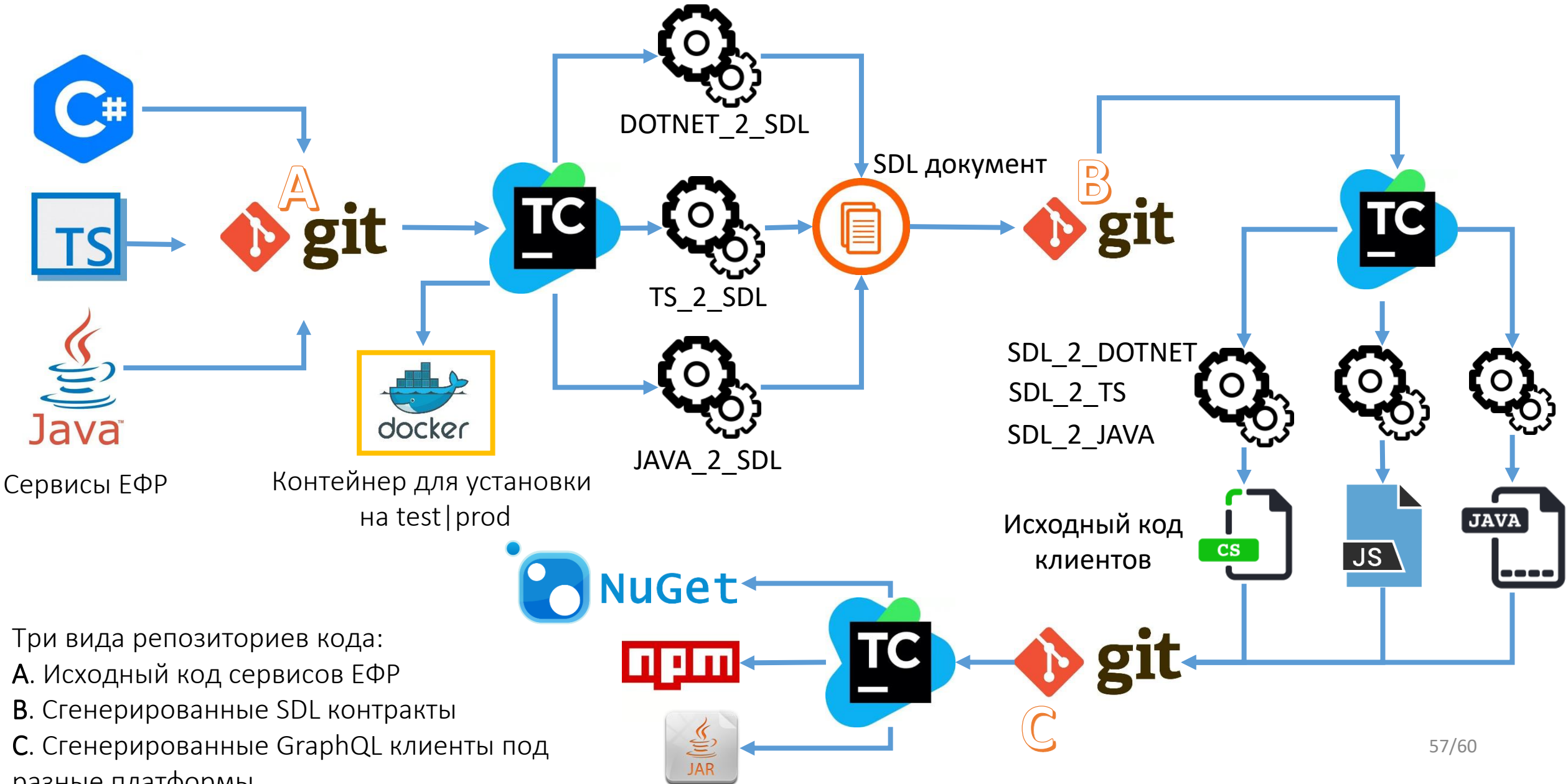
1 домен  $\approx$  1 команда разработки, но это не точно



\* Ведутся работы по взаимодействию сервисов по спецификации GraphQL через транспорт Apache Kafka



# Обмен GraphQL контрактами



# Что дальше?

Что ещё предстоит сделать и/или находится в разработке.

- Картография и мониторинг сервисов
- Анализ зависимостей на основе контрактов и кода
- Аутентификация, авторизация, API Keys, ролевая модель
- GraphQL SecurityProxy – аналог SSL offloader, только для ролей
- Kafka транспорт для GraphQL Subscriptions
- Распределённая трассировка/профилирование

# Что осталось за кадром

**Introspection** – получение схемы в runtime

**DataLoader** – решение проблемы N+1

**Fragments** - именованные и встроенные фрагменты

**Variables** – переменные, передаваемые отдельно от запроса

**Validations** – валидация входных/выходных данных

**Batching** – группировка запросов в пакеты

**HTTP GraphQL Spec** – спецификация передачи GraphQL по HTTP

**Tracing** – распределенная трассировка и мониторинг взаимодействия сервисов

**Localization** - интернационализация API

**Query Cost Analysis/Rate Limiting** – контроль потребляемых клиентом ресурсов

**Paging** - страничная разбивка результатов

**Caching** – кеширование результатов запроса клиентом/сервером

**и многое, многое другое...**

Спасибо за внимание