**Слайд 4**

В качестве примера проекта мы посмотрим на некогда успешно работающий сайт-магазин [unicornadoptions.com](https://www.unicornadoptions.com/), который предлагает возможность приобрести набор для приручения единорога.

**Слайд 5-6**

Посетитель этого сайта может выбрать понравившийся набор для приручения единорога, затем посмотреть, что именно входит в этот набор: игрушка, сертификат, значок. Далее у покупателя есть возможность добавить товар в корзину, просмотреть ее содержимое и оформить заказ. Простой сайт, которых неисчислимое множество. Сайт работает с одним бекендом, с разными клиентами: браузер, телефон и тд. И использует для этого API о котором мы и поговорим.

**Слайд 7**

Давайте представим себе, какой API общего назначения (то есть один API для всех магазинов поверх платформы) мы бы могли создать, чтобы обеспечить функциональность магазинов. Сконцентрируемся пока сугубо на получении данных.

**Слайд 8**

Для страницы продукта на таком сайте должно возвращаться название продукта, его цена, картинки, описание, дополнительная информация и многое другое. Стандартное решение для такой задачи поверх HTTP API — это описать ресурс /products/:id, который на GET-запрос возвращает данные продукта, цену продукта, изображение, описание, доп. Информацию и еще множество данных для отображения.

**Слайд 9**

Давайте теперь посмотрим на страницу каталога продуктов. Для этой страницы понадобится ресурс-коллекция */products*. Вот только в отображении коллекции продуктов на странице каталога нужны не все данные продуктов, а лишь цена, название и основное изображение. Например, описание, дополнительная информация, второстепенные изображения и прочее нас не интересуют.

Допустим, для простоты, мы решаем использовать одинаковую модель данных продукта для ресурсов /products и /products/:id. В случае коллекции таких продуктов потенциально будет несколько. Схему ответа можно представить следующим образом:

**Слайд 10**

А теперь давайте посмотрим на «полезную нагрузку» ответа от сервера для коллекции продуктов. Вот что в действительности используется клиентом среди более чем двух десятков полей:

Очевидно, что если я хочу держать модель продукта простой, возвращая одинаковые данные, то в итоге сталкиваюсь с over-fetching проблемой, получая в некоторых случаях больше данных, чем мне необходимо. В данном случае это проявилось на странице каталога продуктов, но вообще, любые экраны UI, которые так или иначе связаны с продуктом, потребуют от него потенциально только части (а не всех) данных.

**Слайд 11:**

Давайте теперь рассмотрим страницу корзины. В корзине, кроме самих продуктов, есть еще их количество (в этой корзине), цена, а также суммарная стоимость всего заказа:

Если продолжать подход простого моделирования HTTP API, то корзина может быть представлена через ресурс */carts/:id*, представление которого ссылается на ресурсы продуктов, добавленных в эту корзину:

**Слайд 12:**

Теперь, например, для того чтобы отрисовать корзину с тремя продуктами на фронтенде, необходимо сделать четыре запроса: один для того, чтобы загрузить саму корзину, и три запроса, чтобы загрузить данные по продуктам (название, цену и артикул SKU).  
  
Вторая проблема, которая у нас возникла — under-fetching. Разграничение ответственности между ресурсами корзины и продукта привело к необходимости делать дополнительные запросы

**Слайд 13:**

Конечно же, такое решение не подходит для продакшена. Один из способов избавиться от проблемы — это добавить поддержку проекций для корзины. Дальше отдельные проекции для единорогов, для мобильного клиента и вот мы городим огород раздувая проект и делая его не пригодным.

**Слайд 14-15**

В 2012 году в процессе разработки мобильного приложения с подобной проблемой столкнулась компания Facebook. Инженерам хотелось достичь минимального количества обращений мобильного приложения к серверу, при этом на каждом шаге получая только нужные данные и ничего, кроме них. Результатом их усилий стал GraphQL, представленный в 2015 году на конференции React Conf. GraphQL — это язык описания запросов, а также среда исполнения этих запросов.

**Слайд 16:**

Рассмотрим типичный подход к работе с GraphQL-серверов.

Описываем схему

Схема данных в GraphQL определяет типы и связи между ними и делает это в строго-типизированной манере. Например, представим себе простую модель социальной сети. Пользователь User знает про своих друзей friends. Пользователи живут в городе City, и город знает про своих жителей через поле citizens. Вот что является графом такой модели в GraphQL:

Мы не можем напрямую обратиться в любую точку графа, нужна какая то точка входа или несколько. Поэтому мы должны дополнительно сказать как получить пользователя по имени, город по имени, как в рамках этих типов получить сложные запросы.

**Слайд 17 – 18**

Посмотрим где же запросы, где QL в GraphQL.

Переведем на этот язык такой вопрос: *«Для пользователя с именем Vanya Unicorn, хочу узнать имена его друзей, а также название и население города, в котором Ваня проживает»*:

Обратите внимание, как форма запроса «созвучна» с формой ответа. Возникает ощущение, что этот язык запросов создавался для JSON. Со строгой типизацией. И все это делается за один запрос HTTP POST — не нужно делать несколько обращений к серверу.

Открываю GraphiQl: 1ый запрос

Давайте посмотрим, как это выглядит на практике. Откроем стандартную консоль для GraphQL-сервера, которая называется Graph*i*QL («графикл»). Для запроса на корзину я выполню следующий запрос: *«Хочу получить корзину по идентификатору 1, интересуют все позиции этой корзины и информация по продуктам. Из информации важны название, цена, инвентарный номер и изображения (причем только первое). Также меня интересует количество этих продуктов, какова их цена и общая стоимость в рамках корзины»*.

{

 cart(id: 1) {

   items {

     product {

       title

       price

       sku

       images(limit: 1)

     }

     quantity

     total

   }

   subTotal

 }

}

Модно, удобно, молодежно!

**Слайд 19**

* **Гибкая выборка.** Клиент может составить запрос под свои конкретные требования.
* **Эффективная выборка.** В ответе возвращаются только запрошенные данные.
* **Более быстрая разработка.** Много изменений на клиенте могут происходить без необходимости менять что-либо на серверной стороне. Например, исходя из нашего примера, запросто можно показать другое представление корзины для мобильного web.
* **Полезная аналитика.** Так как клиент обязан в запросе указывать поля явно, сервер точно знает, какие поля действительно нужны. А это важная информация для deprecation-политики.
* **Работает поверх любого источника данных и транспорта.** Важно, что GraphQL позволяет работать поверх любого источника данных и любого транспорта. В данном случае HTTP — это не панацея, GraphQL может также работать через WebSocket

**Слайды 20**

Сегодня GraphQL-сервер можно сделать практически на любом языке. Наиболее полная версия GraphQL-сервера — [GraphQL.js](https://github.com/graphql/graphql-js) для Node-платформы. В Java-комьюнити эталонной реализацией является [GraphQL Java](https://www.graphql-java.com/).

**Слайд 21**

Давайте посмотрим, как создать GraphQL-сервер на конкретном жизненном примере.  
  
Рассмотрим упрощенную версию интернет-магазина на основе микросервисной архитектуры с двумя компонентами:

* Cart-сервис, обеспечивающий работу с пользовательской корзиной. Хранит данные в реляционной БД и использует SQL для доступа к данным. Очень простой сервис, без лишней магии :)
* Product-сервис, обеспечивающий доступ к продуктовому каталогу, из которого, собственно, и наполняется корзина. Предоставляет HTTP API для доступа к продуктовым данным.

Оба сервиса реализованы поверх классического Spring Boot и уже содержат всю базовую логику

Мы же намерены создать GraphQL API поверх Cart-сервиса. Этот API призван обеспечить доступ к данным корзины и добавленным в нее продуктам.

Открытие схемы:

В этой самой схеме, как вы помните, есть «точки входа» или запросы верхнего уровня. Они определяются через поле *query* в схеме. Назовем наш тип для точек входа *EntryPoints*:

schema {

 query: EntryPoints

}

Определим в нем поиск корзины по идентификатору как первую точку входа:

type EntryPoints {

 cart(id: Long!): Cart

}

Cart — это и есть не что иное как *поле* в терминах GraphQL. id — параметр этого поля со скалярным типом Long. Восклицательный знак ! после указания типа означает, что параметр обязательный.  
  
Самое время определить и тип Cart:

type Cart {

 id: Long!

 items: [CartItem!]!

 subTotal: BigDecimal!

}

Кроме стандартного идентификатора id в корзину входят ее элементы *items* и сумма за все товары subTotal. Обратите внимание, что *items* определены как список, о чем свидетельствуют квадратные скобки []. Элементы этого списка являются типами CartItem. Наличие восклицательного знака после названия типа поля ! указывает, что поле обязательное. Это значит, что сервер обязуется вернуть непустое значение для этого поля, если оно было запрошено.  
  
Осталось посмотреть на определение типа CartItem, в который входит ссылка на продукт (*productId*), сколько раз он добавлен в корзину (quantity) и сумма продукта, пересчитанная на количество (total):

type CartItem {

 productId: String!

 quantity: Int!

 total: BigDecimal!

}

Здесь всё просто — все поля скалярных типов и являются обязательными.  
  
Такая схема выбрана не случайно. В Cart-сервисе уже определена корзина Cart и ее элементы CartItem с точно такими же названиями и типами полей, как и в схеме GraphQL. Модель корзины использует библиотеку Lombok для автогенерации геттеров/сеттеров, конструкторов и других методов. JPA используется для персистенции в БД.

Итак, корзина (Cart) и элементы корзины (CartItem) описаны и в GraphQL-схеме, и в коде, и «совместимы» между собой по набору полей и их типам. Но этого еще недостаточно для того, чтобы наш сервис заработал.  
  
Нам необходимо уточнить, как именно будет работать точка входа "cart(id: Long!): Cart".

Итак, корзина (Cart) и элементы корзины (CartItem) описаны и в GraphQL-схеме, и в коде, и «совместимы» между собой по набору полей и их типам. Но этого еще недостаточно для того, чтобы наш сервис заработал.  
  
Нам необходимо уточнить, как именно будет работать точка входа "cart(id: Long!): Cart". Для эттго реализованы специальные бины – GraphQLQueryResolvers. Не будем на этом останавливаться, нас интересует сам GraphQL, тем более никто ничего не поймет

Этих изменений нам достаточно для получения работающего приложения. После перезапуска Cart-сервиса в консоли GraphiQL начнет успешно исполняться следующий запрос:

{

 cart(id: 1) {

   items {

     productId

     quantity

     total

   }

   subTotal

 }

}

На заметку

* На данном этапе разработки нашего приложения внутренняя и внешняя модель предметной области полностью идентичны. Речь идет о классах Cart и CartItem и их непосредственном использовании в GraphQL-резолверах. В боевых приложениях эти модели рекомендуется разделять. Для GraphQL-резолверов должна существовать отдельная от внутренней предметной области модель.

На основе уже готовой схемы мы можем выполнять и более сложные запросы:

{

 cart(id: 1) {

   items {

     product {

       title

       price

       sku

       images

     }

     quantity

     total

   }

   subTotal

 }

}

Самое время показать и Document Explorer, часть Graph*i*QL-консоли, которая строится на основе GraphQL-схемы и показывает информацию по всем определенным типам. Вот как выглядит Document Explorer для типа CartItem:

## **Изменение данных через мутации**

До сих пор мы рассматривали сугубо выборку данных. Но GraphQL позволяет органично организовать не только получение данных, но и их изменение. Для этого существует механизм мутаций. В схеме для этого отведено специальное место — поле mutation:

schema {

 query: EntryPoints,

 mutation: Mutations

}

Например, добавление продукта в корзину может быть организовано через такую мутацию:

type Mutations {

   addProductToCart(cartId: Long!,

                    productId: String!,

                    count: Int = 1): Cart

}

Это похоже на определение поля, ведь у мутации также есть параметры и возвращающееся значение.  
  
Реализация мутации в коде сервера с помощью GraphQL SPQR выглядит следующим образом:

@GraphQLMutation(name = "addProductToCart")

**public** Cart **addProductToCart**(

       @GraphQLArgument(name = "cartId") Long cartId,

       @**GraphQLArgument**(name = "productId") String productId,

       @**GraphQLArgument**(name = "quantity", defaultValue = "1") **int** quantity) {

**return** cartService.addProductToCart(cartId, productId, quantity);

}

Конечно же, основная часть полезной работы делается внутри cartService. А задача этого метода-прослойки — связать ее с API. Как и в случае с выборкой данных, благодаря аннотациям @GraphQL\* очень просто понять, какая именно генерируется GraphQL-схема из этого определения метода.  
  
В консоли GraphQL теперь можно выполнить запрос-мутацию на добавление определенного продукта в нашу корзину в количестве 2:

mutation {

 addProductToCart(

     cartId: 1,

     productId: "59eb83c0040fa80b29938e3f",

     quantity: 2) {

   items {

     product {

       title

     }

     quantity

     total

   }

   subTotal

 }

}

Вот мы и посмотрели какой GraphQL интересный, мощный и гибкий инструмент. Но мы с вами инженеры и наша задача помимо плюсов найти еще и минусы.

**Слайд 22:**

Представим себе работу с графом пользователей в рамках классической социальной сети, такой как Facebook. Если такая система предоставляет GraphQL API, то клиенту ничего не мешает послать запрос следующего характера:

{

 user(name: "Vova Unicorn") {

   friends {

     name

     friends {

       name

       friends {

         name

         friends {

            name

            ...

         }

       }

     }

   }

 }

}

**Слайд 23:**

На 5-6 уровне вложенности полноценное выполнение такого запроса приведет к выборке всех в мире пользователей. Сервер уж точно не справится с такой задачей за один присест и скорее всего просто напросто «упадет».  
  
Есть ряд мер, которые следует обязательно предпринять для того, чтобы обезопаситься от подобных ситуаций:

* **Ограничить глубину запроса.** Иными словами, нельзя позволять клиентам просить данные произвольной вложенности.
* **Ограничить сложность запроса.** Назначив вес на каждое поле и подсчитав сумму весов всех полей в запросе, можно принимать или отклонять такие запросы на сервере.

В зависимости от используемого языка пути включения данных ограничений могут отличаться, и мы сейчас не будем на этом останавливаться.

**Слайд 24:**

Давайте посмотрим на полный список недостатков GraphQL

 **необходимость в дополнительных проверках отказоустойчивости**. Вы сами могли убедиться, что для того чтобы поддерживать безопасную работу такого API, необходимо строить дополнительную защиту, прибегая к анализу сложности и глубины запросов.

 GraphQL **плохо обстоят дела с кэшированием данных**. В GraphQL нет таких богатых возможностей по кэшированию, как в HTTP API. Заголовки *Cache-Control* или *Last-Modified* широко используемые в HTTP не помогают в случае с GraphQL API. Вы также не можете воспользоваться кэшированием на промежуточных узлах, типа proxy и gateways (Varnish, Fastly и другие). С одной стороны, GraphQL обеспечивает эффективность выполнения запроса, но с другой стороны, плохо обеспечивает кэширование.

 **Обработка ошибок** в GraphQL требует дополнительного контракта и соглашений вне стандарта. Имена ошибок и их семантику нужно изобретать самостоятельно.

 Вы **не можете работать с произвольными ресурсами**. GraphQL — не универсальное решение для всех типов данных. Вы можете работать с JSON и XML, но, например, загружать файлы на сервер вы вряд ли будете через GraphQL, потому что он не предназначен для этого.

 В GraphQL **нет понятия идемпотентности операции (при вызове операции сколько угодлно раз, значение возвращается одно и то же)**. Например, для изменения состояния на сервере в HTTP можно применять *PUT* для идемпотентных операций и *POST* для не-идемпотентных. Это отличие важно, потому что идемпотентные операции можно запросто повторять. Так вот в GraphQL соглашение про идемпотентность не является частью стандарта. Все детали нужно выносить в объяснение и документацию.

 **Нужно придумывать имена операциям**. Например, операцию удаления можно назвать по-разному: «delete» или «kill», «annihilate» или «terminate», ну и так далее. Между разными GraphQL API такие соглашения будут разными. Конкретно с HTTP этот пример идеально ложился бы просто на использование метода *DELETE*.

**Слайд 25:**

Мы посмотрели на GraphQL, на REST, когда то в моде был SOAP, а есть еще gRPC. Так что же лучше? Я сейчас не буду рассказывать про плюсы и минусы каждого подхода, поскольку это займет еще минимум 4 слайда и несколько страниц текста, по которому я всё рассказываю. Самое главное, что надо понять – для каждого дела свой инструмент.

Благодаря своей сильной связанности RPC хорошо годится для внутренних микросервисов, но это не вариант для сильного внешнего API или API-сервиса.

SOAP  —  это хлопотно, но его обширный функционал в плане безопасности по-прежнему незаменим для биллинговых операций, систем бронирования и платежей.

REST обладает самой высокой абстракцией и лучшим моделированием API. Но он, как правило, тяжелее в плане нагрузки на сеть и многословнее  —  недостаток, если вы работаете на мобильных устройствах.

GraphQL  —  большой шаг вперед с точки зрения извлечения данных, но не у всех достаточно времени и возможностей, чтобы его осваивать.