Обзор на GraphQL и Prisma

Prisma

Prisma — это ORM нового поколения с открытым исходным кодом для Node.js и TypeScript. Она состоит из следующих инструментов:

- **Prisma Client**: Автогенерируемый и типобезопасный клиент базы данных
- **Prisma Migrate**: Декларативное моделирование данных и миграции с возможностью пользовательского редактирования
- **Prisma Studio**: Современный пользовательский интерфейс для просмотра и редактирования данных

Prisma

Prisma Client

Auto-generated and type-safe database client.

Prisma Migrate

Declarative data modeling and customizable migrations.

Prisma Studio

Modern UI to view and edit your application data.

Prisma – зачем?

Работа с базами данных — одна из самых сложных областей разработки приложений. Моделирование данных, миграция схем и написание запросов к базе данных — это задачи, с которыми разработчики приложений сталкиваются каждый день.

Слоган Prisma звучит следующим образом:

Разработчики приложений должны думать о данных, а не о SQL

Моделирование данных в Prisma Schema

```
model Post {
  id        Int     @id @default(autoincrement())
  title     String
  content   String?
  published Boolean @default(false)
  author    User?   @relation(fields: [authorId], references: [id])
  authorId  Int?
}

model User {
  id    Int     @id @default(autoincrement())
  email String  @unique
  name  String?
  posts Post[]
}
```

Моделирование данных в Prisma Schema

Каждая из этих моделей описывает таблицу в соответствующей базе данных и служит основой для сгенерированного доступа к данным с API, предоставляемого Prisma Client. В WebStorm существует расширение Prisma, которое предоставляет подсветку синтаксиса, автодополнение, быстрые исправления и множество других возможностей, чтобы сделать моделирование данных волшебным и приятным занятием ✨.

Миграция баз данных с Prisma Migrate

Prisma Migrate преобразует Prisma schema в SQL, необходимый для создания и изменения таблиц в вашей базе данных. Его можно запустить команду ***prisma migrate*** из API Prisma CLI.

Далее представлены сгенерированные SQL схемы согласно установленному поставщику данных в конфигурации Prisma.

Рассмотрим на примере диалекта Postgre SQL.

Миграция баз данных с Prisma Migrate

```sql
CREATE TABLE "Post" (
    "id" SERIAL NOT NULL,
    "title" TEXT NOT NULL,
    "content" TEXT,
    "published" BOOLEAN NOT NULL DEFAULT false,
    "authorId" INTEGER,

    PRIMARY KEY ("id")
);
```

```sql
CREATE TABLE "User" (
    "id" SERIAL NOT NULL,
    "email" TEXT NOT NULL,
    "name" TEXT,

    PRIMARY KEY ("id")
);
```

```sql
CREATE UNIQUE INDEX "User.email_unique" ON "User"("email");

ALTER TABLE "Post" ADD FOREIGN KEY ("authorId") REFERENCES "User"("id")
ON DELETE SET NULL ON UPDATE CASCADE;
```

Генерация объектов в Prisma

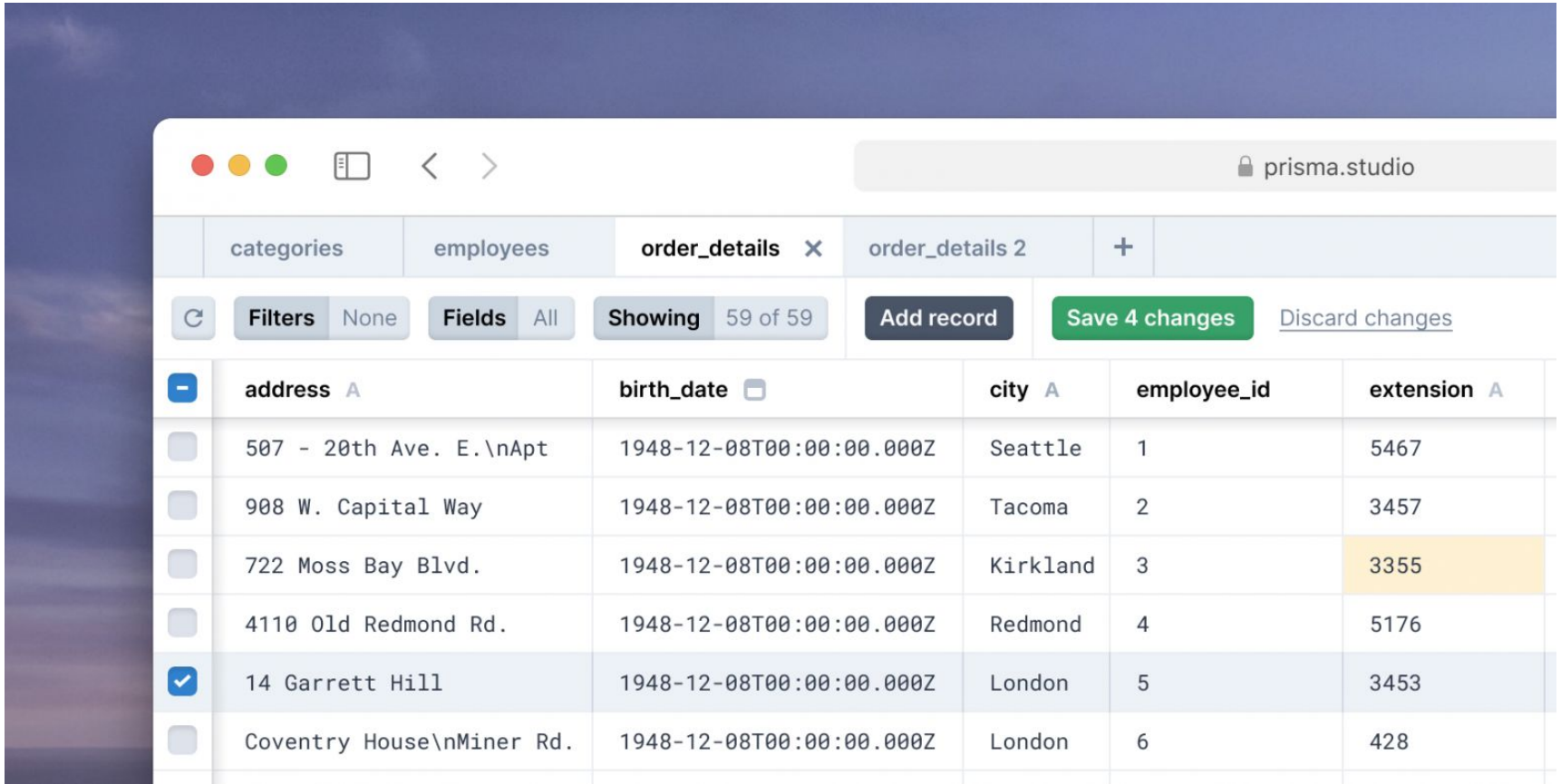
Основным преимуществом работы с Prisma Client является то, что он позволяет разработчикам мыслить объектами и поэтому предлагает привычный и естественный способ рассуждать о своих данных.

В Prisma Client нет концепции экземпляров модели. Вместо этого он помогает сформировать запросы к базе данных, которые всегда возвращают простые объекты JavaScript. Благодаря генерируемым типам вы получаете автозаполнение и для этих запросов.

Также в качестве бонуса для разработчиков TypeScript: Все результаты запросов Prisma Client полностью типизированы. Фактически, Prisma обеспечивает самые сильные гарантии безопасности типов среди всех ORM на TypeScript.

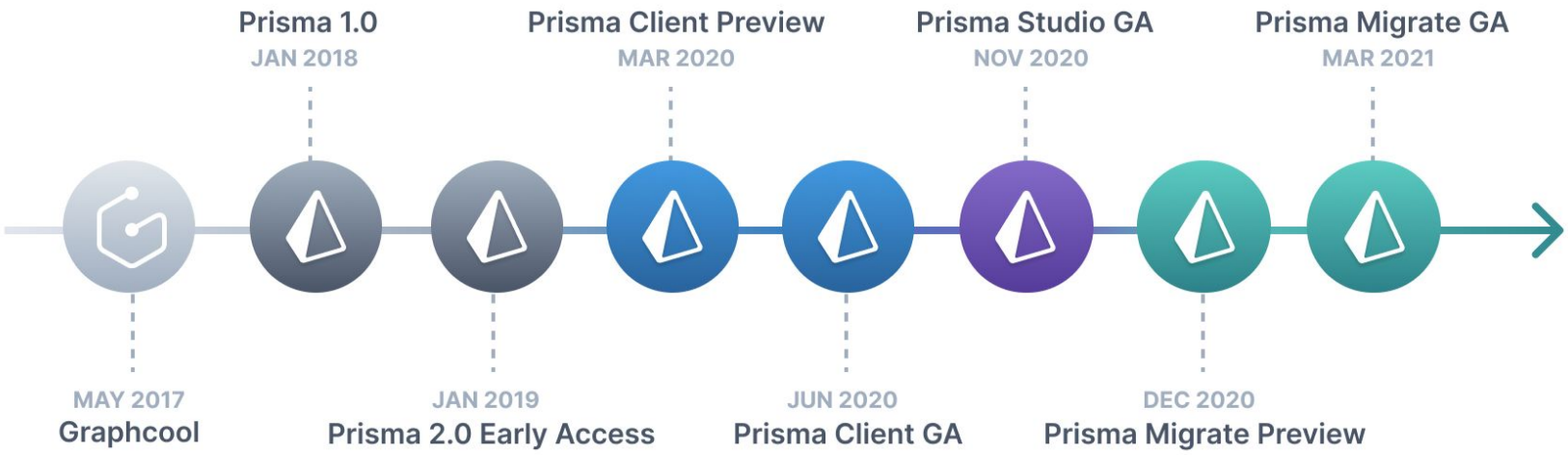
Prisma Studio

Prisma также поставляется с современным интерфейсом администратора для вашей базы данных



Prisma

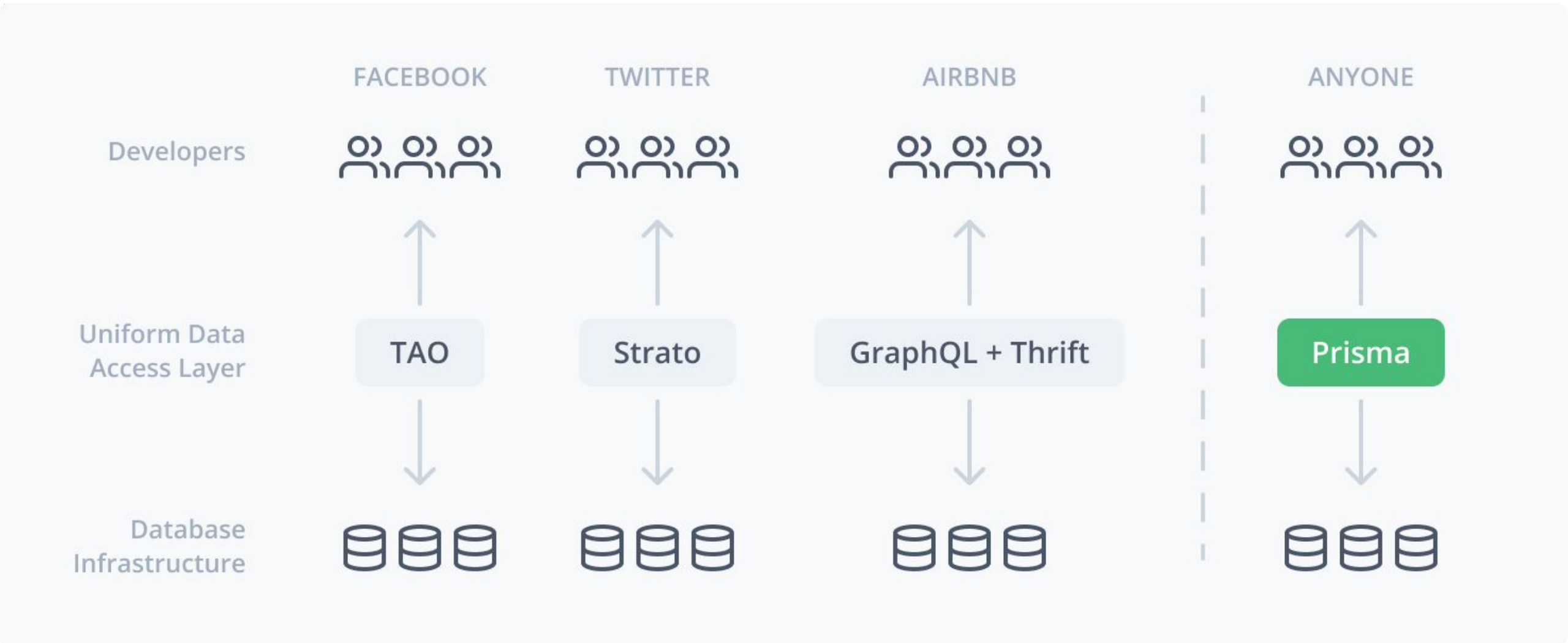
За последние три года Prisma претерпела значительные изм



Prisma & GraphQL

Prisma — это результат опыта, приобретённый благодаря экосистеме GraphQL, и знаний о слоях данных компаний всех размеров, от небольших стартапов до крупных предприятий.

Используемая тысячами компаний с момента первого выпуска три года назад, Prisma прошла боевые испытания и готова к эксплуатации в критически важных приложениях.


FACEBOOK
TWITTER
AIRBNB
ANYONE
Developers
Uniform Data Access Layer
TAO
Strato
GraphQL + Thrift
Prisma
Database Infrastructure

Что же такое этот GraphQL?

tl:dr;

GraphQL это **синтаксис, который описывает как запрашивать данные**, и, в основном, используется клиентом для загрузки данных с сервера.

GraphQL имеет три основные характеристики:

- Позволяет клиенту точно указать, какие данные ему нужны.
- Облегчает агрегацию данных из нескольких источников.
- Использует систему типов для описания данных.

Задача GraphQL

GraphQL был разработан в Facebook ещё в 2012-ом году, для решения проблем с ограничениями традиционных REST API интерфейсов.

Например представьте, что вам нужно отобразить список записей (posts), и под каждым опубликовать список лайков (likes), включая имена пользователей и аватары. На самом деле, это не сложно, вы просто измените API posts так, чтобы оно содержало массив likes, в котором будут объекты-пользователи.

```
[
  {
    title: 'My First Post',
    likes: [
      {
        name: 'Sacha',
        avatar: 'sacha.jpg'
      },
      {
        name: 'Mike',
        avatar: 'mike.jpg'
      }
    ]
  },
  {
    title: 'Another Great Post',
    likes: [
      {
        name: 'Julia',
        avatar: 'julia.jpg'
      },
      {
        name: 'Sacha',
        avatar: 'sacha.jpg'
      }
    ]
  },
]
```

My First Post

Liked by:

 **Sacha**

 **Mike**

Another Great Post

Liked by:

 **Julia**

 **Sacha**

Задача GraphQL

При разработке мобильного приложения, оказалось что из-за загрузки дополнительных данных приложение работает медленнее. Так что вам теперь нужно *два* endpoint, один возвращающий записи с лайками, а другой без них.

Добавим ещё один фактор: оказывается, записи хранятся в базе данных MySQL, а лайки в Redis! Что же теперь делать?! Экстраполируйте этот сценарий на то множество источников данных и клиентских API, с которыми имеет дело Facebook, и вы поймёте почему старый добрый REST API достиг своего предела.

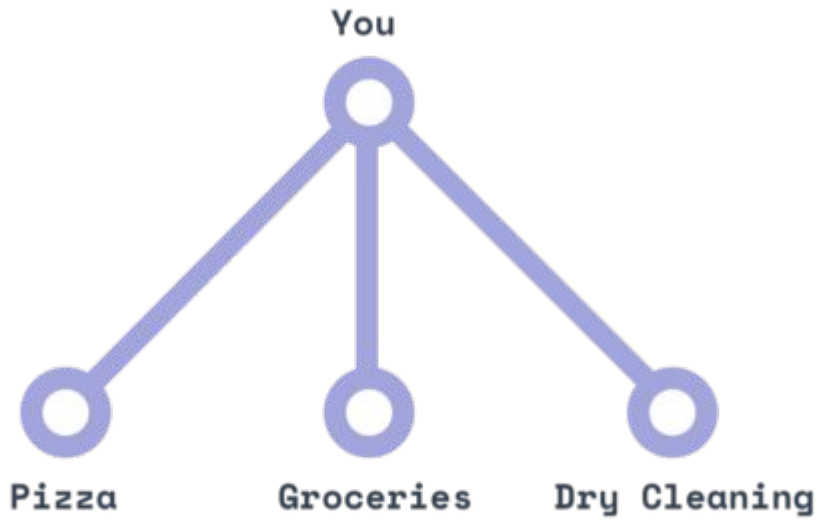
Решение

Facebook придумал концептуально простое решение: вместо того, чтобы иметь множество "глупых" endpoint, лучше иметь один "умный" endpoint, который будет способен работать со сложными запросами и придавать данным такую форму, какую запрашивает клиент.

Фактически, слой GraphQL находится между клиентом и одним или несколькими источниками данных; он принимает запросы клиентов и возвращает необходимые данные в соответствии с переданными инструкциями. Запутаны? Время метафор

Решение

Пользоваться старой REST-моделью это как заказывать пиццу, затем заказывать доставку продуктов, а затем звонить в химчистку, чтобы забрать одежду. Три магазина – три телефонных звонка.



Решение

GraphQL похож на личного помощника: вы можете передать ему адреса всех трех мест, а затем просто запрашивать то, что вам нужно («принеси мне мою одежду, большую пиццу и два десятка яиц») и ждать их получения.

Из чего состоит GraphQL

- На практике GraphQL API построен на трёх основных строительных блоках:
- на **схеме** (schema)
- **запросах** (queries)
- и **распознавателях** (resolvers).

Запросы (queries)

(query и request одинаково переводится как "запрос". Далее будет подразумеваться query)

Когда вы о чём-то просите вашего персонального помощника, вы выполняете **запрос**. Это выглядит примерно так:

Мы объявляем новый запрос при помощи ключевого слова query,
также спрашивая про поле stuff

```
query {
  stuff
}
```

Запросы (queries)

Самое замечательное в запросах GraphQL является то, что они поддерживают вложенные поля, так что мы можем пойти на один уровень глубже:

```
query {
  stuff {
    eggs
    shirt
    pizza
  }
}
```

Как можно заметить,
клиенту при формировании запроса
не нужно знать откуда поступают данные.
Он просто спрашивает о них,
а сервер GraphQL заботится об остальном.

Запросы (queries)

Также стоит отметить, что поля запроса могут быть **массивами**. Например вот общий шаблон запроса списка сообщений:

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

Запросы (queries)

Поля запроса также могут содержать аргументы. Например, если необходимо отобразить конкретный пост, можно добавить аргумент id к п

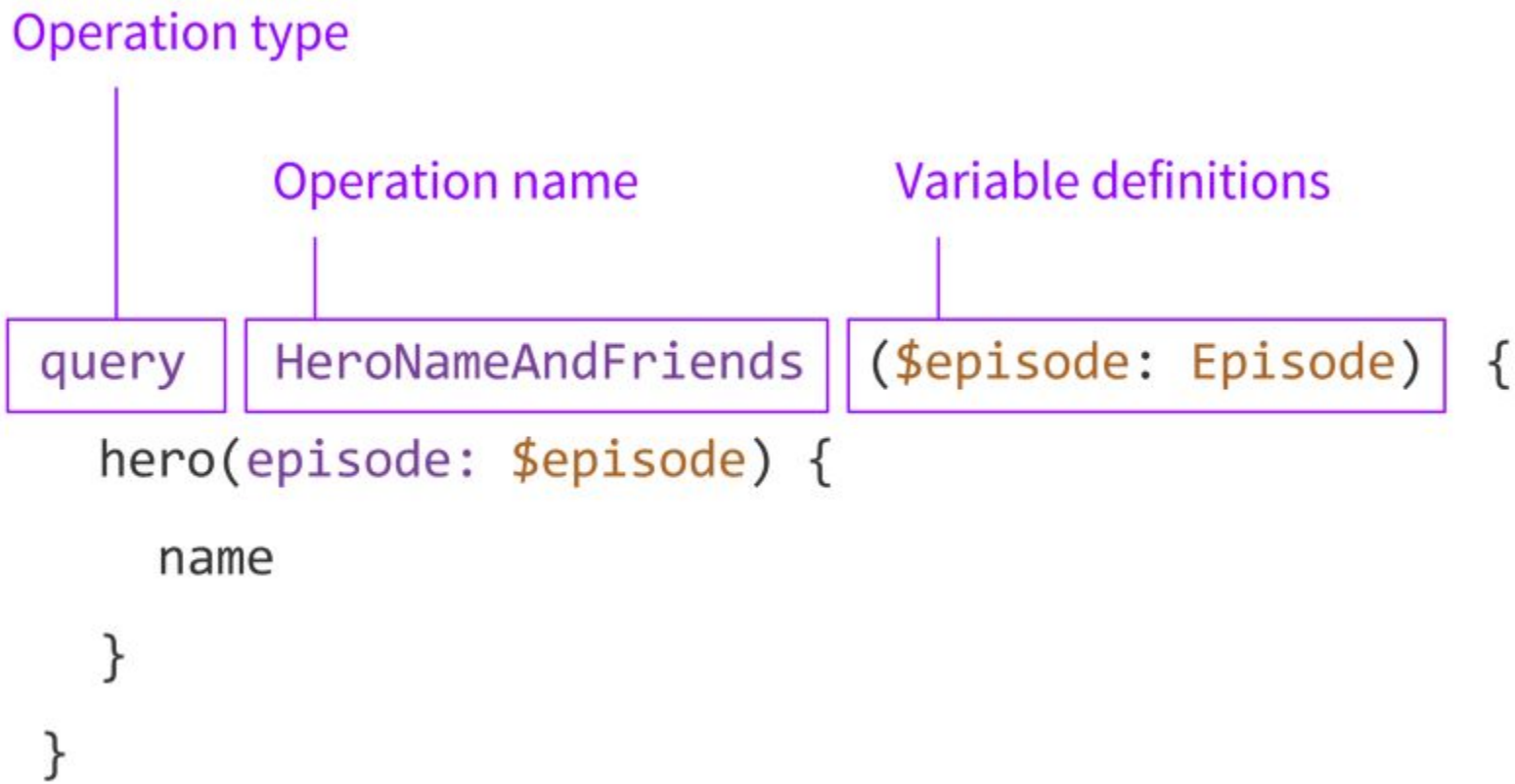
```
query {
  post(id: "123foo"){
    title
    body
    author{
      name
      avatarUrl
      profileUrl
    }
  }
}
```

Запросы (queries)

Наконец, если нужно сделать аргумент id динамическим, можно определить переменную, а затем использовать её в запросе (обратите внимание, также мы сделали запрос именованным):

```
query getMyPost($id: String) {
  post(id: $id){
    title
    body
    author{
      name
      avatarUrl
      profileUrl
    }
  }
}
```

Запросы (queries)



Запросы (queries)

Для того чтобы лучше понять как это работает, можем воспользоваться GraphQL Explorer’ом на примере уже готового приложения

https://is-web-y23-lecture-6.herokuapp.com

Запросы (queries)

Обратите внимание, когда мы открываем Explorer автоматически предлагает вам возможные имена полей, полученные при помощи GraphQL API Introspection.

Распознаватели (resolvers)

Даже самый лучший в мире личный помощник не сможет принести ваши вещи из химчистки если вы не дадите ему адрес.

Подобным образом, сервер GraphQL не может знать что делать с входящим запросом, если ему не объяснить при помощи **распознавателя** (**resolver**).

Распознаватели (resolvers)

Используя распознаватель GraphQL понимает, как и где получить данные, соответствующие запрашиваемому полю. К примеру, распознаватель для поля **запись** может выглядеть вот так:

```
Query: {
  post(root, args) {
    return Posts.find({ id: args.id });
  }
}
```

Распознаватели (resolvers)

Мы помещаем наш распознаватель в раздел Query потому что мы хотим получать запись (post) в корне ответа. Но также мы можем создать распознаватели для подполей, как например для поля auth

```
Query: {
  post(root, args) {
    return Posts.find({ id: args.id });
  }
},
Post: {
  author(post) {
    return Users.find({ id: post.authorId})
  }
}
```

Распознаватели (resolvers)

Обратите внимание что ваши распознаватели не ограничены в количестве возвращаемых объектов. К примеру, вы захотите добавить поле commentsCount к объекту Post:

```
Post: {
  author(post) {
    return Users.find({ id: post.authorId})
  },
  commentsCount(post) {
    return Comments.find({ postId: post.id}).count()
  }
}
```

Распознаватели (resolvers)

Обратите внимание что ваши распознаватели не ограничены в количестве возвращаемых объектов. К примеру, вы захотите добавить поле commentsCount к объекту Post:

```
Post: {
  author(post) {
    return Users.find({ id: post.authorId})
  },
  commentsCount(post) {
    return Comments.find({ postId: post.id}).count()
  }
}
```

Распознаватели (resolvers)

Ключевое понятие здесь то, что схема запроса GraphQL и структура вашей базы данных никак не связаны. Другими словами, в базе данных может не существовать полей author или commentsCount, но мы можем "симулировать" их благодаря силе распознавателей.

Как было показано выше, вы можете писать любой код внутри распознавателя. Так что вы можете изменять содержимое базы данных; такие распознаватели называют изменяющими (mutation).

Посмотреть работу мутаций можно так же на примере ранее показанного приложения.

Схема (schema)

Всё взаимодействие становится возможным благодаря типизированной схеме данных GraphQL.

Я призываю вас заглянуть в документацию GraphQL *(англ)*, если вы хотите узнать больше.

А пока что просто создадим готовую схему на основе той, что уже описана в приложении.

https://www.prisma.io/docs/concepts/components/prisma-schema/generators

Исходники приложения:

https://github.com/Nditah/node-graphql-lesson-04