Строим GraphQL-сервер

Pavel Chertorogov :: @nodkz

Чтоб листать презентацию используйте ПРОБЕЛ.

А то Reveal.js со своим листанием
влево-вправо-вверх-вниз
ещё много кому ломает голову.

Ссылка на эту презентацию

http://bit.ly/holy-graphql

Коротко о себе

Коротко о себе

- В универе дипломку писал по SOAP и WSDL 😬

Коротко о себе

- В универе дипломку писал по SOAP и WSDL 😬
- В веб-разработке с 2001 года (ужос 17 лет мучений) 🙀

Коротко о себе

- В универе дипломку писал по SOAP и WSDL 😬
- В веб-разработке с 2001 года (ужос 17 лет мучений) 🙀
- Фронтендер и бэкендер в одном флаконе 💑

Коротко о себе

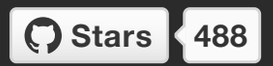
- В универе дипломку писал по SOAP и WSDL 😬
- В веб-разработке с 2001 года (ужос 17 лет мучений) 🙀
- Фронтендер и бэкендер в одном флаконе 💏
- Использую GraphQL с 2015 года 💃

Мой OpenSource

- graphql-compose — генерация GraphQL-схем 

- mongodb-memory-server — MongoDB для тестов 

- react-relay-network-layer — NetworkLayer для Relay 

- lvovich — склонение городов 

- и пачка других

Section #1

Зачем что-то менять?

В чем профит от использования GraphQL?

GraphQL приходит на смену REST API

GraphQL приходит на смену REST API

Он тупо удобнее 😇

У REST API часто адовая документация

Самодельная и без интерактива

Not Secure | https://taldau.stat.gov.kz/ru/Api/Dev

3. Информация о доступных типов периодов GetPeriodList

http://taldau.stat.gov.kz/ru/Api/GetPeriodList?indexId=2709379

где нужно указать:

indexId - указывается идентификатор показателя.

Ответ сервера возвращается в формате JSON.

[
{"id":7,"name":"Год"},
{"id":9,"name":"Квартал с накоплением"}
]

Описание полей ответа:

- **id** - тип периода. В дальнейшем используется в качестве параметра periodId для других запросов;
- **name** - название типа периода.

Страшненькая и неудобная

1. Данные показателя GetIndexTreeData

http://taldau.stat.gov.kz/ru/Api/GetIndexTreeData?
p_measure_id=1&p_index_id=2709379&p_period_id=7&p_terms=741880&p_term_id=741880&p_dicIds=67&idx=0&parent_id=
где следует указать:

- **p_index_id** - идентификатор показателя;
- **p_period_id** - идентификатор типа периода;
- **p_terms** - список элементов разделенных через запятую для выборки (termIds из GetSegmentList);
- **p_term_id** - указывается один главный элемент по которому нужна детализация (один из p_terms);
- **p_dicIds** - список справочников через запятую (dicId из GetSegmentList);
- **idx** - индекс разрезности (idx из GetSegmentList);
- **parent_id** - идентификатор родительского элемента. Для получения корневого элемента следует оставлять пустым.

dateList

Ответ сервера возвращается в формате JSON.

[{"
id":"741880",
"rownum ":16,
"text ":"РЕСПУБЛИКА КАЗАХСТАН",
"leaf":"false",
"expanded":"true",
"measureName":"",
"y122011":"28243052700000",
"y122012":"31015186600000",
"y122013":"35999025100000"

Частенько используют Swagger для стандартизации, документации и интерактива вашего API

Частенько используют Swagger для стандартизации, документации и интерактива вашего API

Но он тоже не айс 🤪

В Swagger есть интерактив, стандарт да и выглядит покрасивше

pet Everything about your Pets — Find out more: http://swagger.io

POST `/pet` Add a new pet to the store

PUT `/pet` Update an existing pet

GET `/pet/findByStatus` Finds Pets by status

Multiple status values can be provided with comma separated strings

Parameters — Try it out

Name	Description

status * required

`array[string]`

(query)

Status values that need to be considered for filter

Available values : available, pending, sold

Но от этого не легче 😭

Как вообще этой "фигней" пользоваться?

Resume Show/Hide | List Operations | Expand Operations

POST	/resume/{resumeId}/personal
POST	/resume/{resumeId}/additional
DELE	/resume/{resumeId}/additional/{additionalId}
DELE	/resume/{resumeId}/experience
POST	/resume/{resumeId}/experience
DELE	/resume/{resumeId}/experience/{experienceId}
POST	/resume/{resumeId}/position
POST	/resume/{resumeId}/experience/{experienceId}/recommendation
DELE	/resume/{resumeId}/experience/{experienceId}/recommendation/{recommendationId}
DELE	/resume/{resumeId}/language
POST	/resume/{resumeId}/language
DELE	/resume/{resumeId}/language/{languageId}
DELE	/resume/{resumeId}/photo

POST /resume/{resumeId}/photo

POST /resume/{resumeId}/contact

GET /resume

POST /resume

DELETE /resume/{resumeId}

GET /resume/{resumeId}

POST /resume/{resumeId}/copy

POST /resume/{resumeId}/confirmemail

GET /resume/{resumeId}/state

POST /resume/{resumeId}/state

POST /resume/{resumeId}/date

POST /resume/{resumeId}/searchstate

POST /resume/{resumeId}/skill

POST /resume/{resumeId}/uilanguage

POST /resume/{resumeId}/education

DELETE /resume/{resumeId}/education/{educationId}

POST /resume/{resumeId}/training

DELETE /resume/{resumeId}/training/{trainingId}

Минусы REST API, даже со Swagger'ом:

Минусы REST API, даже со Swagger'ом:

- Каждый ендпоинт описывается примитивными типами, нет "сложных" типов

Минусы REST API, даже со Swagger'ом:

- Каждый ендпоинт описывается примитивными типами, нет "сложных" типов
- Нет связей, не известно как оптимальнее всего запросить связанные ресурсы

Минусы REST API, даже со Swagger'ом:

- Каждый ендпоинт описывается примитивными типами, нет "сложных" типов
- Нет связей, не известно как оптимальнее всего запросить связанные ресурсы
- Жирные ответы, если не прикрутить фильтр по полям

Минусы REST API, даже со Swagger'ом:

- Каждый ендпоинт описывается примитивными типами, нет "сложных" типов
- Нет связей, не известно как оптимальнее всего запросить связанные ресурсы
- Жирные ответы, если не прикрутить фильтр по полям
- Нет возможности построить сложный агрегационный запрос

Минусы REST API, даже со Swagger'ом:

- Каждый ендпоинт описывается примитивными типами, нет "сложных" типов
- Нет связей, не известно как оптимальнее всего запросить связанные ресурсы
- Жирные ответы, если не прикрутить фильтр по полям
- Нет возможности построить сложный агрегационный запрос
- Частенько много сетевых запросов

Минусы REST API, даже со Swagger'ом:

- Каждый ендпоинт описывается примитивными типами, нет "сложных" типов
- Нет связей, не известно как оптимальнее всего запросить связанные ресурсы
- Жирные ответы, если не прикрутить фильтр по полям
- Нет возможности построить сложный агрегационный запрос
- Частенько много сетевых запросов
- Боль с вложенными агрументами

Самый жирный минус:

Самый жирный минус:

С REST API фронтендеры пишут кучу бойлерплейт кода, для получения связанных данных между ресурсами.

Самый жирный минус:

С REST API фронтендеры пишут кучу бойлерплейт кода, для получения связанных данных между ресурсами.

Часто гадают на кофейной гуще 🤔

Самый жирный минус:

С REST API фронтендеры пишут кучу бойлерплейт кода, для получения связанных данных между ресурсами.

Часто гадают на кофейной гуще 🤔

Часто с матами 🤬

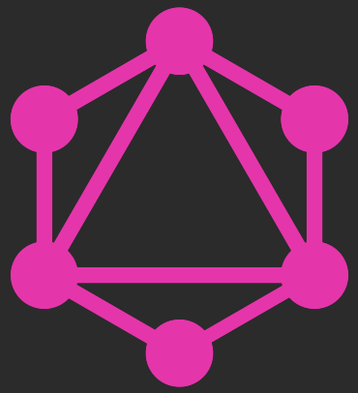
Ведь бэкендеры хорошо знают связи между данными,

но способ передачи знаний обычно плохо отработан

GraphQL чума!

Когда много связанных данных

Что такое GraphQL?



Это *удобный* формат для общения между сервером и клиентом

GraphiQL ▶ Prettify History

```
{
  viewer {
    # Гибкость агрументентов
    customer(filter: {customerID: "AROUT"}) {
      # Выбор полей в ответе
      customerID
      companyName
      companyName
      # Вложенные запросы
      # Описание связей между ресурсами/моделями
      orderList(limit: 3, skip: 10) {
        # Фрагменты (для компонентного подхода)
        ...OrderData
      }
    }
    # Запросить несколько ресурсов в одном запросе
    # Гибкость агрументентов
    order1: order(filter: {orderID: 11001}) {
      # Фрагменты (для компонентного подхода)
      ...OrderData
    }
    # Запросить несколько ресурсов в одном запросе
    # Гибкость агрументентов
    order2: order(filter: {orderID: 11002}) {
      # Фрагменты (для компонентного подхода)
      ...OrderData
    }
  }
}

# Фрагменты (для компонентного подхода)
fragment OrderData on Order {
  # Выбор полей в ответе
  orderID
  orderDate
  freight
}
```

```
{
  "data": {
    "viewer": {
      "customer": {
        "customerID": "AROUT",
        "companyName": "Around the Horn",
        "orderList": [
          {
            "orderID": 10864,
            "orderDate": "1998-02-02T00:00:00.000Z",
            "freight": 3.04
          },
          {
            "orderID": 10953,
            "orderDate": "1998-03-16T00:00:00.000Z",
            "freight": 23.72
          },
          {
            "orderID": 11016,
            "orderDate": "1998-04-10T00:00:00.000Z",
            "freight": 33.8
          }
        ]
      },
      "order1": {
        "orderID": 11001,
        "orderDate": "1998-04-06T00:00:00.000Z",
        "freight": 197.3
      },
      "order2": {
        "orderID": 11002,
        "orderDate": "1998-04-06T00:00:00.000Z",
        "freight": 141.16
      }
    }
  }
}
```

Viewer **Customer** ✕

Search Customer...

Тип Customer содержит данные по заказчику. Также можно запросить связанные с каждым заказчиком списки его счетов с возможностью пагинации.

IMPLEMENTS

Node

FIELDS

customerID: String

Customer unique ID

companyName: String

contactName: String

contactTitle: String

address: CustomerAddress

_id: MongoID!

id: ID!

The globally unique ID among all types

orderConnection(
  first: Int
  after: String
  last: Int
  before: String
  sort: SortConnectionOrderEnum = _ID_DESC
): OrderConnection

QUERY VARIABLES

```
orderList(
  skip: Int
  limit: Int = 1000
  sort: SortFindManyOrderInput
): [Order]
```

GraphiQL ▶ Prettify History

```graphql
{
  viewer {
    # Гибкость агрументов
    customer(filter: {customerID: "AROUT"}) {
      # Выбор полей в ответе
      customerID
      companyName
      # Вложенные запросы
      # Описание связей между ресурсами/моделями
      orderList(limit: 3, skip: 10) {
        # Фрагменты (для компонентного подхода)
        ...OrderData
      }
    }
    # Запросить несколько ресурсов в одном запросе
    # Гибкость агрументов
    order1: order(filter: {orderID: 11001}) {
      # Фрагменты (для компонентного подхода)
      ...OrderData
    }
    # Запросить несколько ресурсов в одном запросе
    # Гибкость агрументов
    order2: order(filter: {orderID: 11002}) {
      # Фрагменты (для компонентного подхода)
      ...OrderData
    }
  }
}

# Фрагменты (для компонентного подхода)
fragment OrderData on Order {
  # Выбор полей в ответе
  orderID
  orderDate
  freight
}
```

QUERY VARIABLES


```json
{
  "data": {
    "viewer": {
      "customer": {
        "customerID": "AROUT",
        "companyName": "Around the Horn",
        "orderList": [
          {
            "orderID": 10864,
            "orderDate": "1998-02-02T00:00:00.000Z",
            "freight": 3.04
          },
          {
            "orderID": 10953,
            "orderDate": "1998-03-16T00:00:00.000Z",
            "freight": 23.72
          },
          {
            "orderID": 11016,
            "orderDate": "1998-04-10T00:00:00.000Z",
            "freight": 33.8
          }
        ]
      },
      "order1": {
        "orderID": 11001,
        "orderDate": "1998-04-06T00:00:00.000Z",
        "freight": 197.3
      },
      "order2": {
        "orderID": 11002,
        "orderDate": "1998-04-06T00:00:00.000Z",
        "freight": 141.16
      }
    }
  }
}
```


Documentation Explorer

Search Schema...

A GraphQL schema provides a root type for each kind of operation.

ROOT TYPES

query: Query

mutation: Mutation

Фишки GraphQL:

Фишки GraphQL:

- Описание бэкендерами связей между ресурсами с фильтрацией

Фишки GraphQL:

- Описание бэкендерами связей между ресурсами с фильтрацией
- Выбор полей фронетнедерами в ответе из коробки

Фишки GraphQL:

- Описание бэкендерами связей между ресурсами с фильтрацией
- Выбор полей фронетнедерами в ответе из коробки
- Вложенные запросы

Фишки GraphQL:

- Описание бэкендерами связей между ресурсами с фильтрацией
- Выбор полей фронетнедерами в ответе из коробки
- Вложенные запросы
- Гибкость агрументов на любом уровне вложенности

Фишки GraphQL:

- Описание бэкендерами связей между ресурсами с фильтрацией
- Выбор полей фронетнедерами в ответе из коробки
- Вложенные запросы
- Гибкость агрументов на любом уровне вложенности
- Получение нескольких ресурсов в одном запросе

Фишки GraphQL:

- Описание бэкендерами связей между ресурсами с фильтрацией
- Выбор полей фронетнедерами в ответе из коробки
- Вложенные запросы
- Гибкость агрументов на любом уровне вложенности
- Получение нескольких ресурсов в одном запросе
- Фрагменты (для компонентного подхода)

Фишки GraphQL:

- Описание бэкендерами связей между ресурсами с фильтрацией
- Выбор полей фронетнедерами в ответе из коробки
- Вложенные запросы
- Гибкость агрументов на любом уровне вложенности
- Получение нескольких ресурсов в одном запросе
- Фрагменты (для компонентного подхода)
- Поддерживает сложные типы (статическая типизация 🌶)

Фишки GraphQL:

- Описание бэкендерами связей между ресурсами с фильтрацией
- Выбор полей фронетнедерами в ответе из коробки
- Вложенные запросы
- Гибкость агрументов на любом уровне вложенности
- Получение нескольких ресурсов в одном запросе
- Фрагменты (для компонентного подхода)
- Поддерживает сложные типы (статическая типизация 🌶)
- Экзотика полиморфизма (Interfaces, Union types)

GraphQL — это не база данных!

GraphQL помогает бэкендерам задать конву - описание структуры, типов и связей в ваших данных.

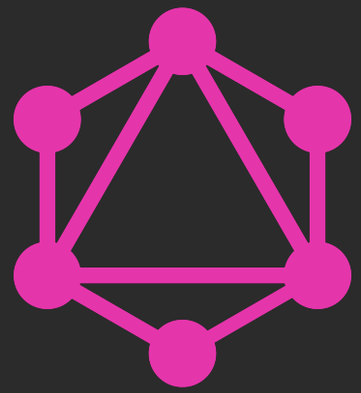
Работает с любыми базами данных, на любом языке программирования!

Работает с любыми базами данных, на любом языке программирования!

Грубо говоря, GraphQL требует от бекендеров описать набор функций для получения данных 👌

На сервере объявляете о своих *возможностях*

в предоставлении данных (бэкендеры создают GraphQL схему).



На клиенте заявляете о своих *потребностях*

в получении данных (фронтендеры пишут GraphQL запросы).

С GraphQL как в ресторане:

— Вот вам пожалуйста меню, выбирайте.

— Будьте любезны салатик 🥗, рыбку на пару 🐟
и бокальчик красного 🍷.

С REST API как в армии:

— Жри что дали, а то голодным останешься,
И наваливают кучу чего-то 🤐

С REST API как в армии:

— Жри что дали, а то голодным останешься,
И наваливают кучу чего-то 🤐

PS. Swagger — это контрактная армия 😂

Вроде все сыты остались,
но как мы видим есть нюанс

Концептуальная разница GraphQL и REST API в том

Концептуальная разница GraphQL и REST API в том

что логику получения связанных ресурсов перенесли с клиента на сервер.

Концептуальная разница GraphQL и REST API в том

что логику получения связанных ресурсов перенесли с клиента на сервер.

Сняли жуткий головняк с фронтендеров 👍

Более подробнее расписано тут

- Что такое GraphQL?
- Swagger vs GraphQL

Section #2
Экосистема GraphQL

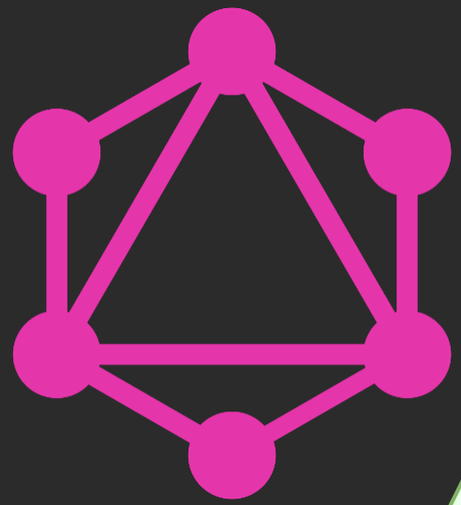
В этой секции речь пойдет:

- GraphQL Schema
- GraphQL Типы
- Что такое SDL и интроспекция?
- Четыре подхода к написанию GraphQL-схем
- GraphQL на вашем сервере
- Запускаем сервер на NodeJS

Ссылки кликабельные и ведут на детальные статьи самописного производства 😉

GraphQL-сервер в NodeJS

graphql package



server package

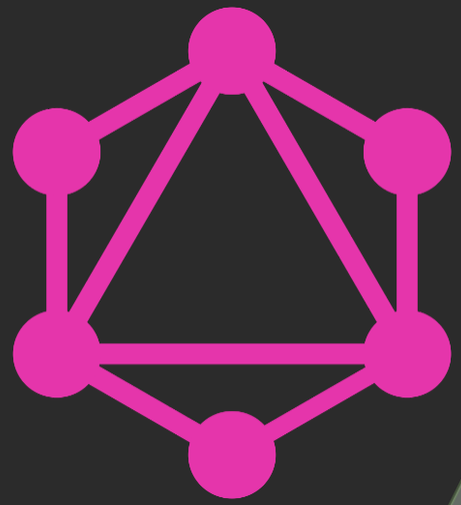
Define GraphQL Schema

Execute GraphQL Query

Serve Client Requests

GraphQL Schema

graphql package



server package

Define GraphQL Schema

Execute GraphQL Query

Serve Client Requests

GraphQL Schema — это

GraphQL Schema — это

описание ваших типов данных на сервере,

GraphQL Schema — это

описание ваших типов данных на сервере, связей между ними

GraphQL Schema — это

описание ваших типов данных на сервере,
связей между ними
и логики получения этих самых данных.

GraphQL-схема это точка входа, это корень всего вашего API.

И у этого корня три "головы" 

- `query` — для операций получения данных
- `mutation` — для операций изменения данных
- `subscription` — для подписки на события

Hello world schema (build phase)

```
import { GraphQLSchema, GraphQLObjectType, GraphQLString } from 'graphql';

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'RootQueryType',
    fields: {
      hello: {
        type: GraphQLString,
        resolve: () => 'world',
      }
    }
  }),
  // mutation: { ... },
  // subscription: { ... },
});
```

Когда вы описываете структуру своих данных и методы получения.

Hello world schema (runtime phase)

```
1 import { graphql } from 'graphql';
2 import { schema } from './your-schema';
3
4 const query = '{ hello }';
5 const result = await graphql(schema, query);
6
7 // returns: { data: { hello: "world" } }
```

Hello world schema (runtime phase)

```
1 import { graphql } from 'graphql';
2 import { schema } from './your-schema';
3
4 const query = '{ hello }';
5 const result = await graphql(schema, query);
6
7 // returns: { data: { hello: "world" } }
```

Когда вы выполняете клиентский запрос на вашей схеме методом graphql() который:

Hello world schema (runtime phase)

```
1 import { graphql } from 'graphql';
2 import { schema } from './your-schema';
3
4 const query = '{ hello }';
5 const result = await graphql(schema, query);
6
7 // returns: { data: { hello: "world" } }
```

Когда вы выполняете клиентский запрос на вашей схеме методом graphql() который:

- производит парсинг GraphQL-запроса

Hello world schema (runtime phase)

```
1 import { graphql } from 'graphql';
2 import { schema } from './your-schema';
3
4 const query = '{ hello }';
5 const result = await graphql(schema, query);
6
7 // returns: { data: { hello: "world" } }
```

Когда вы выполняете клиентский запрос на вашей схеме методом graphql() который:

- производит парсинг GraphQL-запроса
- производит валидацию запроса на соответствие GraphQL-схемы

Hello world schema (runtime phase)

```
1 import { graphql } from 'graphql';
2 import { schema } from './your-schema';
3
4 const query = '{ hello }';
5 const result = await graphql(schema, query);
6
7 // returns: { data: { hello: "world" } }
```

Когда вы выполняете клиентский запрос на вашей схеме методом graphql() который:

- производит парсинг GraphQL-запроса
- производит валидацию запроса на соответствие GraphQL-схемы
- выполняет запрос, пробегаясь по дереву схемы

Hello world schema (runtime phase)

```
1 import { graphql } from 'graphql';
2 import { schema } from './your-schema';
3
4 const query = '{ hello }';
5 const result = await graphql(schema, query);
6
7 // returns: { data: { hello: "world" } }
```

Когда вы выполняете клиентский запрос на вашей схеме методом graphql() который:

- производит парсинг GraphQL-запроса
- производит валидацию запроса на соответствие GraphQL-схемы
- выполняет запрос, пробегаясь по дереву схемы
- валидирует возвращаемый ответ

GraphQL по натуре строго типизированный

GraphQL по натуре строго типизированный

Шаг влево, шаг вправо от схемы —

GraphQL по натуре строго типизированный

Шаг влево, шаг вправо от схемы —

☠ РАССТРЕЛ ☠

☠ РАССТРЕЛ ☠

☠️ РАССТРЕЛ ☠️

- фронтендера если криво запросил данные

☠️ РАССТРЕЛ ☠️

- фронтендера если криво запросил данные
- бекендера если криво вернул данные

GraphQL-схема своей строгостью заставляет Бэкендеров и Фронтендеров 💑 жить дружно 👨‍❤️‍👨

💋❤️🍻

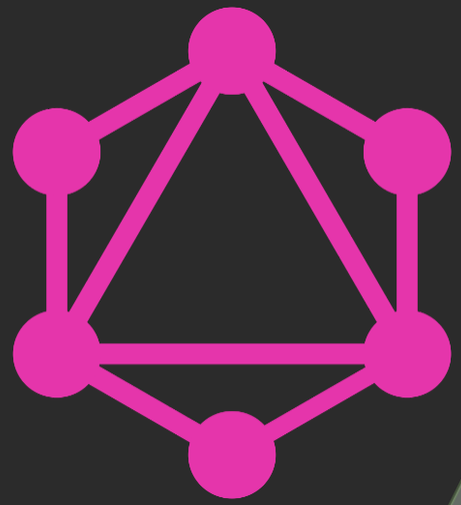
GraphQL-схема своей строгостью заставляет Бэкендеров и Фронтендеров 💑 жить дружно 👨‍❤️‍💋‍👨

💋❤️🍻

И винить во всех грехах GraphQL, т.к. теперь он крайний!

GraphQL-типы

graphql package



server package

Define GraphQL Schema

Execute GraphQL Query

Serve Client Requests

GraphQL-схема содержит в себе описания всех типов, полей и функции получения данных.

GraphQL-тип содержит в себе поля
Поля содержат в себе GraphQL-тип

Тип — Поля — Тип — Поля — Тип — Поля — ...

Тип — Поля — Тип — Поля — Тип — Поля — ...

Думаете все так просто? 🤔

Тип — Поля — Тип — Поля — Тип — Поля — ...

Думаете все так просто? 🤔

Конечно просто!

Тип — Поля — Тип — Поля — Тип — Поля — ...

Думаете все так просто? 🤔

Конечно просто!

Через месяц другой использования GraphQL

😈

Тип — Поля — Тип — Поля — Тип — Поля — ...

Думаете все так просто? 🤔

Конечно просто!

Через месяц другой использования GraphQL

😈

Погнали ускорять понимание!

Есть две группы типов:

- Output — для вывода данных
- Input — для ввода данных

Есть две группы типов:

- Output — для вывода данных
- Input — для ввода данных

А еще модификаторы и аннотации 🤯

Система типов состоит из:

Система типов состоит из:

- Scalar types (Output, Input)

Система типов состоит из:

- Scalar types (Output, Input)
- Custom scalar types (Output, Input)

Система типов состоит из:

- Scalar types (Output, Input)
- Custom scalar types (Output, Input)
- Object types (только Output)

Система типов состоит из:

- Scalar types (Output, Input)
- Custom scalar types (Output, Input)
- Object types (только Output)
- Input types (только Input)

Система типов состоит из:

- Scalar types (Output, Input)
- Custom scalar types (Output, Input)
- Object types (только Output)
- Input types (только Input)
- Enumeration types (Output, Input)

Система типов состоит из:

- Scalar types (Output, Input)
- Custom scalar types (Output, Input)
- Object types (только Output)
- Input types (только Input)
- Enumeration types (Output, Input)
- Lists and Non-Null (модификаторы типов для Output, Input)

Система типов состоит из:

- Scalar types (Output, Input)
- Custom scalar types (Output, Input)
- Object types (только Output)
- Input types (только Input)
- Enumeration types (Output, Input)
- Lists and Non-Null (модификаторы типов для Output, Input)
- Interfaces (только Output)

Система типов состоит из:

- Scalar types (Output, Input)
- Custom scalar types (Output, Input)
- Object types (только Output)
- Input types (только Input)
- Enumeration types (Output, Input)
- Lists and Non-Null (модификаторы типов для Output, Input)
- Interfaces (только Output)
- Union types (только Output)

Система типов состоит из:

- Scalar types (Output, Input)
- Custom scalar types (Output, Input)
- Object types (только Output)
- Input types (только Input)
- Enumeration types (Output, Input)
- Lists and Non-Null (модификаторы типов для Output, Input)
- Interfaces (только Output)
- Union types (только Output)
- Root types (только Output)

Система типов состоит из:

- Scalar types (Output, Input)
- Custom scalar types (Output, Input)
- Object types (только Output)
- Input types (только Input)
- Enumeration types (Output, Input)
- Lists and Non-Null (модификаторы типов для Output, Input)
- Interfaces (только Output)
- Union types (только Output)
- Root types (только Output)
- Directives (аннотации для типов и рантайма)

Scalar types

5 базовых скалярных типов

- `GraphQLInt` — целое число
- `GraphQLFloat` — число с плавающей точкой
- `GraphQLString` — строка в формате UTF-8
- `GraphQLBoolean` — true/false
- `GraphQLID` — строка; уникальный индетификатор

Custom scalar types

Не хватает 5 скалярных типов?

Хочется сразу работать с Date а не циферками?

Date, Email, URL, LimitedString, Password, SmallInt ...

Custom scalar types

Не хватает 5 скалярных типов?

Хочется сразу работать с Date а не циферками?

Date, Email, URL, LimitedString, Password, SmallInt ...

Их можно объявить самостоятельно!

Объявляем свой Custom scalar type

```
1  import { GraphQLScalarType, GraphQLError } from 'graphql';
2
3  export default new GraphQLScalarType({
4    // 1) --- ОПРЕДЕЛЯЕМ МЕТАДАННЫЕ ТИПА ---
5    // У каждого типа, должно быть уникальное имя
6    name: 'DateTimestamp',
7    // Хорошим тоном будет предоставить описание для вашего типа,
8    // чтобы оно отображалось в документации
9    description: 'A string which represents a HTTP URL',
10
11   // 2) --- ОПРЕДЕЛЯЕМ КАК ТИП ОТДАВАТЬ КЛИЕНТУ ---
12   // Чтобы передать клиенту в GraphQL-ответе значение вашего поля
13   // вам необходимо определить функцию `serialize`,
14   // которая превратит значение в допустимый json-тип
15   serialize: (v: Date) => v.getTime(), // return 1536417553
```

1. Определяем метаданные типа
2. Как сериализуем для отправки клиенту
3. Как де-сериализуем значение от клиента

Объявляем свой Custom scalar type

```
1  import { GraphQLScalarType, GraphQLError } from 'graphql';
2
3  export default new GraphQLScalarType({
4    // 1) --- ОПРЕДЕЛЯЕМ МЕТАДАННЫЕ ТИПА ---
5    // У каждого типа, должно быть уникальное имя
6    name: 'DateTimestamp',
7    // Хорошим тоном будет предоставить описание для вашего типа,
8    // чтобы оно отображалось в документации
9    description: 'A string which represents a HTTP URL',
10
11   // 2) --- ОПРЕДЕЛЯЕМ КАК ТИП ОТДАВАТЬ КЛИЕНТУ ---
12   // Чтобы передать клиенту в GraphQL-ответе значение вашего поля
13   // вам необходимо определить функцию `serialize`,
14   // которая превратит значение в допустимый json-тип
15   serialize: (v: Date) => v.getTime(), // return 1536417553
```

1. Определяем метаданные типа
2. Как сериализуем для отправки клиенту
3. Как де-сериализуем значение от клиента

Объявляем свой Custom scalar type

```
1  import { GraphQLScalarType, GraphQLError } from 'graphql';
2
3  export default new GraphQLScalarType({
4    // 1) --- ОПРЕДЕЛЯЕМ МЕТАДАННЫЕ ТИПА ---
5    // У каждого типа, должно быть уникальное имя
6    name: 'DateTimestamp',
7    // Хорошим тоном будет предоставить описание для вашего типа,
8    // чтобы оно отображалось в документации
9    description: 'A string which represents a HTTP URL',
10
11   // 2) --- ОПРЕДЕЛЯЕМ КАК ТИП ОТДАВАТЬ КЛИЕНТУ ---
12   // Чтобы передать клиенту в GraphQL-ответе значение вашего поля
13   // вам необходимо определить функцию `serialize`,
14   // которая превратит значение в допустимый json-тип
15   serialize: (v: Date) => v.getTime(), // return 1536417553
```

1. Определяем метаданные типа
2. Как сериализуем для отправки клиенту
3. Как де-сериализуем значение от клиента

Объявляем свой Custom scalar type

```
1  import { GraphQLScalarType, GraphQLError } from 'graphql';
2
3  export default new GraphQLScalarType({
4    // 1) --- ОПРЕДЕЛЯЕМ МЕТАДАННЫЕ ТИПА ---
5    // У каждого типа, должно быть уникальное имя
6    name: 'DateTimestamp',
7    // Хорошим тоном будет предоставить описание для вашего типа,
8    // чтобы оно отображалось в документации
9    description: 'A string which represents a HTTP URL',
10
11   // 2) --- ОПРЕДЕЛЯЕМ КАК ТИП ОТДАВАТЬ КЛИЕНТУ ---
12   // Чтобы передать клиенту в GraphQL-ответе значение вашего поля
13   // вам необходимо определить функцию `serialize`,
14   // которая превратит значение в допустимый json-тип
15   serialize: (v: Date) => v.getTime(), // return 1536417553
```

1. Определяем метаданные типа
2. Как сериализуем для отправки клиенту
3. Как де-сериализуем значение от клиента

Объявляем свой Custom scalar type

```
1  import { GraphQLScalarType, GraphQLError } from 'graphql';
2
3  export default new GraphQLScalarType({
4    // 1) --- ОПРЕДЕЛЯЕМ МЕТАДАННЫЕ ТИПА ---
5    // У каждого типа, должно быть уникальное имя
6    name: 'DateTimestamp',
7    // Хорошим тоном будет предоставить описание для вашего типа,
8    // чтобы оно отображалось в документации
9    description: 'A string which represents a HTTP URL',
10
11   // 2) --- ОПРЕДЕЛЯЕМ КАК ТИП ОТДАВАТЬ КЛИЕНТУ ---
12   // Чтобы передать клиенту в GraphQL-ответе значение вашего поля
13   // вам необходимо определить функцию `serialize`,
14   // которая превратит значение в допустимый json-тип
15   serialize: (v: Date) => v.getTime(), // return 1536417553
```

1. Определяем метаданные типа
2. Как сериализуем для отправки клиенту
3. Как де-сериализуем значение от клиента

Объявляем свой Custom scalar type

```
1  import { GraphQLScalarType, GraphQLError } from 'graphql';
2
3  export default new GraphQLScalarType({
4    // 1) --- ОПРЕДЕЛЯЕМ МЕТАДАННЫЕ ТИПА ---
5    // У каждого типа, должно быть уникальное имя
6    name: 'DateTimestamp',
7    // Хорошим тоном будет предоставить описание для вашего типа,
8    // чтобы оно отображалось в документации
9    description: 'A string which represents a HTTP URL',
10
11   // 2) --- ОПРЕДЕЛЯЕМ КАК ТИП ОТДАВАТЬ КЛИЕНТУ ---
12   // Чтобы передать клиенту в GraphQL-ответе значение вашего поля
13   // вам необходимо определить функцию `serialize`,
14   // которая превратит значение в допустимый json-тип
15   serialize: (v: Date) => v.getTime(), // return 1536417553
```

1. Определяем метаданные типа
2. Как сериализуем для отправки клиенту
3. Как де-сериализуем значение от клиента

Объявляем свой Custom scalar type

```
1  import { GraphQLScalarType, GraphQLError } from 'graphql';
2
3  export default new GraphQLScalarType({
4    // 1) --- ОПРЕДЕЛЯЕМ МЕТАДАННЫЕ ТИПА ---
5    // У каждого типа, должно быть уникальное имя
6    name: 'DateTimestamp',
7    // Хорошим тоном будет предоставить описание для вашего типа,
8    // чтобы оно отображалось в документации
9    description: 'A string which represents a HTTP URL',
10
11   // 2) --- ОПРЕДЕЛЯЕМ КАК ТИП ОТДАВАТЬ КЛИЕНТУ ---
12   // Чтобы передать клиенту в GraphQL-ответе значение вашего поля
13   // вам необходимо определить функцию `serialize`,
14   // которая превратит значение в допустимый json-тип
15   serialize: (v: Date) => v.getTime(), // return 1536417553
```

1. Определяем метаданные типа
2. Как сериализуем для отправки клиенту
3. Как де-сериализуем значение от клиента

Объявляем свой Custom scalar type

```
1  import { GraphQLScalarType, GraphQLError } from 'graphql';
2
3  export default new GraphQLScalarType({
4    // 1) --- ОПРЕДЕЛЯЕМ МЕТАДАННЫЕ ТИПА ---
5    // У каждого типа, должно быть уникальное имя
6    name: 'DateTimestamp',
7    // Хорошим тоном будет предоставить описание для вашего типа,
8    // чтобы оно отображалось в документации
9    description: 'A string which represents a HTTP URL',
10
11   // 2) --- ОПРЕДЕЛЯЕМ КАК ТИП ОТДАВАТЬ КЛИЕНТУ ---
12   // Чтобы передать клиенту в GraphQL-ответе значение вашего поля
13   // вам необходимо определить функцию `serialize`,
14   // которая превратит значение в допустимый json-тип
15   serialize: (v: Date) => v.getTime(), // return 1536417553
```

1. Определяем метаданные типа
2. Как сериализуем для отправки клиенту
3. Как де-сериализуем значение от клиента

Объявляем свой Custom scalar type

```
1  import { GraphQLScalarType, GraphQLError } from 'graphql';
2
3  export default new GraphQLScalarType({
4    // 1) --- ОПРЕДЕЛЯЕМ МЕТАДАННЫЕ ТИПА ---
5    // У каждого типа, должно быть уникальное имя
6    name: 'DateTimestamp',
7    // Хорошим тоном будет предоставить описание для вашего типа,
8    // чтобы оно отображалось в документации
9    description: 'A string which represents a HTTP URL',
10
11   // 2) --- ОПРЕДЕЛЯЕМ КАК ТИП ОТДАВАТЬ КЛИЕНТУ ---
12   // Чтобы передать клиенту в GraphQL-ответе значение вашего поля
13   // вам необходимо определить функцию `serialize`,
14   // которая превратит значение в допустимый json-тип
15   serialize: (v: Date) => v.getTime(), // return 1536417553
```

1. Определяем метаданные типа
2. Как сериализуем для отправки клиенту
3. Как де-сериализуем значение от клиента

Object types

Самый часто используемый конструктор типов в GraphQL - это `GraphQLObjectType`. Output-тип со списком полей:

```
const AuthorType = new GraphQLObjectType({
  // Уникальное имя вашего типа в рамках всей GraphQL-схемы. Обязательный парам
  name: 'Author',
  // Описание типа для документации (интроспекции). Желательно указывать.
  description: 'Author data with related data',
  // Интерфейсы реализуемые текущим типом (смотрите секцию `Interfaces`). Можно
  interfaces: [],
  // Объявление полей, рекомендую не лениться и сразу объявлять через () => ({}
  // это позволяет в будущем избежать проблемы с hoisting'ом (когда у вас два т
  // Обязательный параметр, должно быть указано как минимум одно поле
  fields: () => ({
    id: { type: GraphQLInt },
    name: { type: GraphQLString }, //  <--- FieldConfig
  }),
});
```

Object types

Самый часто используемый конструктор типов в GraphQL - это `GraphQLObjectType`. Output-тип со списком полей:

```
const AuthorType = new GraphQLObjectType({
  // Уникальное имя вашего типа в рамках всей GraphQL-схемы. Обязательный парам
  name: 'Author',
  // Описание типа для документации (интроспекции). Желательно указывать.
  description: 'Author data with related data',
  // Интерфейсы реализуемые текущим типом (смотрите секцию `Interfaces`). Можно
  interfaces: [],
  // Объявление полей, рекомендую не лениться и сразу объявлять через () => ({}
  // это позволяет в будущем избежать проблемы с hoisting'ом (когда у вас два т
  // Обязательный параметр, должно быть указано как минимум одно поле
  fields: () => ({
    id: { type: GraphQLInt },
    name: { type: GraphQLString }, //  <--- FieldConfig
  }),
});
```

Object types

Самый часто используемый конструктор типов в GraphQL - это `GraphQLObjectType`. Output-тип со списком полей:

```
const AuthorType = new GraphQLObjectType({
  // Уникальное имя вашего типа в рамках всей GraphQL-схемы. Обязательный парам
  name: 'Author',
  // Описание типа для документации (интроспекции). Желательно указывать.
  description: 'Author data with related data',
  // Интерфейсы реализуемые текущим типом (смотрите секцию `Interfaces`). Можно
  interfaces: [],
  // Объявление полей, рекомендую не лениться и сразу объявлять через () => ({}
  // это позволяет в будущем избежать проблемы с hoisting'ом (когда у вас два т
  // Обязательный параметр, должно быть указано как минимум одно поле
  fields: () => ({
    id: { type: GraphQLInt },
    name: { type: GraphQLString }, //  <--- FieldConfig
  }),
});
```

Object types

Самый часто используемый конструктор типов в GraphQL - это `GraphQLObjectType`. Output-тип со списком полей:

```
const AuthorType = new GraphQLObjectType({
  // Уникальное имя вашего типа в рамках всей GraphQL-схемы. Обязательный парам
  name: 'Author',
  // Описание типа для документации (интроспекции). Желательно указывать.
  description: 'Author data with related data',
  // Интерфейсы реализуемые текущим типом (смотрите секцию `Interfaces`). Можно
  interfaces: [],
  // Объявление полей, рекомендую не лениться и сразу объявлять через () => ({}
  // это позволяет в будущем избежать проблемы с hoisting'ом (когда у вас два т
  // Обязательный параметр, должно быть указано как минимум одно поле
  fields: () => ({
    id: { type: GraphQLInt },
    name: { type: GraphQLString }, //  <--- FieldConfig
  }),
});
```

Object types

Самый часто используемый конструктор типов в GraphQL - это `GraphQLObjectType`. Output-тип со списком полей:

```
const AuthorType = new GraphQLObjectType({
  // Уникальное имя вашего типа в рамках всей GraphQL-схемы. Обязательный парам
  name: 'Author',
  // Описание типа для документации (интроспекции). Желательно указывать.
  description: 'Author data with related data',
  // Интерфейсы реализуемые текущим типом (смотрите секцию `Interfaces`). Можно
  interfaces: [],
  // Объявление полей, рекомендую не лениться и сразу объявлять через () => ({}
  // это позволяет в будущем избежать проблемы с hoisting'ом (когда у вас два т
  // Обязательный параметр, должно быть указано как минимум одно поле
  fields: () => ({
    id: { type: GraphQLInt },
    name: { type: GraphQLString }, //  <--- FieldConfig
  }),
});
```

Object types — FieldConfig

- `type` — Output-тип возвращаемого значения (Scalar, Enum, OutputObject, Interface, Union, List, NonNull)
- `description` — описание поля для документации
- `deprecationReason` — строка, помечает поле как устаревшее
- `args` — набор аргументов, которые может принимать поле для уточнения возвращаемого значения в resolve-методе:
  - `type` — Input-тип (Scalar, Enum, InputObject, List, NonNull)
  - `defaultValue` — значение по-умолчанию
  - `description` — описание аргумента для документации
- `resolve` — метод получения данных, для текущего поля

Object types — FieldConfig — resolve(source, args, context, info)

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Object types — Example

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // объявляем поле `authors`
    authors: { // FieldConfig:
      // устанавливаем, что это поле вернет нам массив авторов (тип который объ
      type: new GraphQLList(AuthorType),
      args: {
        // позволяем клиентам указать кол-во возвращаемых записей с помощью арг
        limit: {
          // принимаемое значение от клиента - Int
          // если будет передано неверное значение, то GraphQL прервет запрос н
          type: GraphQLInt,
          // если клиент не указал значение в запросе, тогда применится значени
          defaultValue: 10,
```

Подробно и с деталями — читать тут

Input types

GraphQL для входящих аргументов полей может использовать следующие Input-типы:

- `GraphQLScalarType` — как встроенные Int, String, так и ваши кастомные скалярные типы
- `GraphQLEnumType` — это особый вид скаляров, о котором речь пойдет в другом разделе
- `GraphQLInputObjectType` — это сложные объекты, которые состоят из набора именованных полей
- а также любой микс из уже озвученных Input-типов с модификаторами `GraphQLList` и `GraphQLNonNull`

Input types — GraphQLInputObjectType

```
const ArticleInput = new GraphQLInputObjectType({
  // уникальное имя для типа
  name: 'ArticleInput',
  // текстовое описание для всего типа
  description: 'Article data for input',
  // объявляем поля, рекомендую не лениться и сразу объявлять через () => ({})
  // это позволяет в будущем избежать проблемы с hoisting'ом,
  // когда у вас два типа импортят друг-друга
  fields: () => ({
    // объявлеяем поле `title`
    title: {
      // тип поля String!
      type: new GraphQLNonNull(GraphQLString),
      // значение по-умолчанию `Draft`
      defaultValue: 'Draft',
```

Подробно и с деталями — читать тут

Input types — GraphQLInputObjectType

```
const ArticleInput = new GraphQLInputObjectType({
  // уникальное имя для типа
  name: 'ArticleInput',
  // текстовое описание для всего типа
  description: 'Article data for input',
  // объявляем поля, рекомендую не лениться и сразу объявлять через () => ({})
  // это позволяет в будущем избежать проблемы с hoisting'ом,
  // когда у вас два типа импортят друг-друга
  fields: () => ({
    // объявлеяем поле `title`
    title: {
      // тип поля String!
      type: new GraphQLNonNull(GraphQLString),
      // значение по-умолчанию `Draft`
      defaultValue: 'Draft',
```

Подробно и с деталями — читать тут

Input types — GraphQLInputObjectType

```
const ArticleInput = new GraphQLInputObjectType({
  // уникальное имя для типа
  name: 'ArticleInput',
  // текстовое описание для всего типа
  description: 'Article data for input',
  // объявляем поля, рекомендую не лениться и сразу объявлять через () => ({})
  // это позволяет в будущем избежать проблемы с hoisting'ом,
  // когда у вас два типа импортят друг-друга
  fields: () => ({
    // объявлеяем поле `title`
    title: {
      // тип поля String!
      type: new GraphQLNonNull(GraphQLString),
      // значение по-умолчанию `Draft`
      defaultValue: 'Draft',
```

Подробно и с деталями — читать тут

Input types — GraphQLInputObjectType

```
const ArticleInput = new GraphQLInputObjectType({
  // уникальное имя для типа
  name: 'ArticleInput',
  // текстовое описание для всего типа
  description: 'Article data for input',
  // объявляем поля, рекомендую не лениться и сразу объявлять через () => ({})
  // это позволяет в будущем избежать проблемы с hoisting'ом,
  // когда у вас два типа импортят друг-друга
  fields: () => ({
    // объявлеяем поле `title`
    title: {
      // тип поля String!
      type: new GraphQLNonNull(GraphQLString),
      // значение по-умолчанию `Draft`
      defaultValue: 'Draft',
```

Подробно и с деталями — читать тут

Input types — GraphQLInputObjectType

```
const ArticleInput = new GraphQLInputObjectType({
  // уникальное имя для типа
  name: 'ArticleInput',
  // текстовое описание для всего типа
  description: 'Article data for input',
  // объявляем поля, рекомендую не лениться и сразу объявлять через () => ({})
  // это позволяет в будущем избежать проблемы с hoisting'ом,
  // когда у вас два типа импортят друг-друга
  fields: () => ({
    // объявлеяем поле `title`
    title: {
      // тип поля String!
      type: new GraphQLNonNull(GraphQLString),
      // значение по-умолчанию `Draft`
      defaultValue: 'Draft',
```

Подробно и с деталями — читать тут

Input types — GraphQLInputObjectType

```
const ArticleInput = new GraphQLInputObjectType({
  // уникальное имя для типа
  name: 'ArticleInput',
  // текстовое описание для всего типа
  description: 'Article data for input',
  // объявляем поля, рекомендую не лениться и сразу объявлять через () => ({})
  // это позволяет в будущем избежать проблемы с hoisting'ом,
  // когда у вас два типа импортят друг-друга
  fields: () => ({
    // объявлеяем поле `title`
    title: {
      // тип поля String!
      type: new GraphQLNonNull(GraphQLString),
      // значение по-умолчанию `Draft`
      defaultValue: 'Draft',
```

Подробно и с деталями — читать тут

Input types — GraphQLInputObjectType

```
const ArticleInput = new GraphQLInputObjectType({
  // уникальное имя для типа
  name: 'ArticleInput',
  // текстовое описание для всего типа
  description: 'Article data for input',
  // объявляем поля, рекомендую не лениться и сразу объявлять через () => ({})
  // это позволяет в будущем избежать проблемы с hoisting'ом,
  // когда у вас два типа импортят друг-друга
  fields: () => ({
    // объявлеяем поле `title`
    title: {
      // тип поля String!
      type: new GraphQLNonNull(GraphQLString),
      // значение по-умолчанию `Draft`
      defaultValue: 'Draft',
```

Подробно и с деталями — читать тут

Input types — GraphQLInputObjectType

```
const ArticleInput = new GraphQLInputObjectType({
  // уникальное имя для типа
  name: 'ArticleInput',
  // текстовое описание для всего типа
  description: 'Article data for input',
  // объявляем поля, рекомендую не лениться и сразу объявлять через () => ({})
  // это позволяет в будущем избежать проблемы с hoisting'ом,
  // когда у вас два типа импортят друг-друга
  fields: () => ({
    // объявлеяем поле `title`
    title: {
      // тип поля String!
      type: new GraphQLNonNull(GraphQLString),
      // значение по-умолчанию `Draft`
      defaultValue: 'Draft',
```

Подробно и с деталями — читать тут

Enumeration types

Также называемые Enums, это особый вид скаляра, который ограничен определенным набором допустимых значений (key-value).

- На клиентской стороне вы работаете только с ключами (`key`)
- На серверной стороне только со значениями (`value`)

Enumeration types — Example

```
const GenderEnum = new GraphQLEnumType({
  name: 'GenderEnum',
  values: {
    // key    value
    //  ↓       ↓
    MALE: { value: 1 },
    FEMALE: { value: 2 },
    CHUCK_NORRIS: {
      value: 3,
      description: "Значение для особо уважаемого человека",
      deprecationReason: `
        Какой-то ненормальный уже уволенный сотрудник завел это значение.
        Не используйте это поле, если не хотите повторить его судьбу.
      `,
    }
```

`key` — для клиентской стороны

`value` — доступно на сервере в resolve-методах

Подробно и с деталями — читать тут

Lists and Non-Null

Дополнительные модификаторы к типам (еще их называют "wrapping types"):

- `GraphQLList` — задает массив указанного типа
- `GraphQLNonNull` — указывает на то, что возвращаемое или получаемое значение не может быть пустым

По спецификации GraphQL любое Output поле является nullable а Input аргумент — optional, т.е. сервер может вернуть/принять значение указанного типа, либо null.

При этом вы можете комбинировать модификаторы:

SDL	JS	Значение
`[Int!]`	new GraphQLList(new GraphQLNonNull(GraphQLInt))	null или массив чисел
`[Int]!`	new GraphQLNonNull(new GraphQLList(GraphQLInt))	массив чисел или null, пустой массив
`[Int!]!`	new GraphQLNonNull(new GraphQLList(new GraphQLNonNull(GraphQLInt))	массив чисел или пустой массив

SDL	JS	Значение
`[[Int]]`	new GraphQLList(new GraphQLList(GraphQLInt))	целочисленный массив-масивов

Interfaces

`GraphQLInterfaceType` — это именованный абстрактный тип который представляет собой набор именованных полей и их аргументов.

`GraphQLObjectType` может реализовать в дальнейшем этот интерфейс, при этом должны быть объявлены все поля и аргументы, которые определены в интерфейсе.

Interfaces — Definition

```
const EventInterface = new GraphQLInterfaceType({
  name: 'EventInterface',
  fields: () => ({
    ip: { type: GraphQLString },
    createdAt: { type: GraphQLInt },
  }),
  resolveType: value => {
    if (value instanceof ClickEvent) {
      return 'ClickEvent';
    } else if (value instanceof SignedUpEvent) {
      return 'SignedUpEvent';
    }
    return null;
  },
});
```

Interfaces — Definition

```
const EventInterface = new GraphQLInterfaceType({
  name: 'EventInterface',
  fields: () => ({
    ip: { type: GraphQLString },
    createdAt: { type: GraphQLInt },
  }),
  resolveType: value => {
    if (value instanceof ClickEvent) {
      return 'ClickEvent';
    } else if (value instanceof SignedUpEvent) {
      return 'SignedUpEvent';
    }
    return null;
  },
});
```

Interfaces — Definition

```
const EventInterface = new GraphQLInterfaceType({
  name: 'EventInterface',
  fields: () => ({
    ip: { type: GraphQLString },
    createdAt: { type: GraphQLInt },
  }),
  resolveType: value => {
    if (value instanceof ClickEvent) {
      return 'ClickEvent';
    } else if (value instanceof SignedUpEvent) {
      return 'SignedUpEvent';
    }
    return null;
  },
});
```

Interfaces — Definition

```
const EventInterface = new GraphQLInterfaceType({
  name: 'EventInterface',
  fields: () => ({
    ip: { type: GraphQLString },
    createdAt: { type: GraphQLInt },
  }),
  resolveType: value => {
    if (value instanceof ClickEvent) {
      return 'ClickEvent';
    } else if (value instanceof SignedUpEvent) {
      return 'SignedUpEvent';
    }
    return null;
  },
});
```

Interfaces — Using in Output types

```
const ClickEventType = new GraphQLObjectType({
  name: 'ClickEvent',
  interfaces: [EventInterface], // <--Тип может использовать несколько интерфей
  fields: () => ({
    ip: { type: GraphQLString },
    createdAt: { type: GraphQLInt },
    url: { type: GraphQLString },
  }),
});
```

```
const SignedUpEventType = new GraphQLObjectType({
  name: 'SignedUpEvent',
  interfaces: [EventInterface],
  fields: () => ({
    ip: { type: GraphQLString },
    createdAt: { type: GraphQLInt },
    login: { type: GraphQLString },
  }),
});
```

Теперь представим, что у нас в схеме есть `search` метод, который возвращает массив `EventInterface`

```
query {
  search {
    __typename # <----- магическое поле, которое вернет имя типа
    ip
    createdAt
  }
}
```

```
# получим следующий ответ:
# search: [
#   { __typename: 'ClickEvent', createdAt: 1536854101, ip: '1.1.1.1' },
#   { __typename: 'ClickEvent', createdAt: 1536854102, ip: '1.1.1.1' },
#   { __typename: 'SignedUpEvent', createdAt: 1536854103, ip: '1.1.1.1' },
# ]
```

При этом GraphQL позволяет дозапросить поля для конкретных типов через фрагменты:

```
query {
  search { // имеет тип [EventInterface]
    __typename
    ip
    createdAt
    ...on ClickEvent { // фрагмент уточнения типа (resolveType)
      url
    }
    ...on SignedUpEvent {
      login
    }
  }
}
```

```
# получим следующий ответ:
# search: [
#   { __typename: 'ClickEvent', createdAt: 1536854101, ip: '1.1.1.1', url: '/li
#   { __typename: 'ClickEvent', createdAt: 1536854102, ip: '1.1.1.1', url: '/re
#   { __typename: 'SignedUpEvent', createdAt: 1536854103, ip: '1.1.1.1', login:
# ]
```

Подробно и с деталями — читать тут

Union types

Это другой абстрактный тип в GraphQL,

когда у вас нет общих полей.

```
1  import { GraphQLUnionType } from 'graphql';
2
3  const SearchRowType = new GraphQLUnionType({
4    name: 'SearchRow',
5    description: 'Search item which can be one of the following types: Article, C
6    types: () => ([ ArticleType, CommentType, UserProfileType ]),
7    resolveType: (value) => {
8      if (value instanceof Article) {
9        return ArticleType;
10     } else if (value instanceof Comment) {
11       return CommentType;
12     } else if (value instanceof UserProfile) {
13       return UserProfileType;
14     }
15   },
```

Union types — всегда надо использовать фрагменты

```
query {
  search(q: "text") {
    __typename # <----- магическое поле, которое вернет имя типа для каждой зап
    ...on Article { # <----- типизированный фрагмент
      title
      publishDate
    }
    ...on Comment { # <----- типизированный фрагмент
      text
      author
    }
    ...on UserProfile { # <----- типизированный фрагмент
      nickname
      age
    }
```

```
# получим следующий ответ:
# search: [
#   { __typename: 'Article', publishDate: '2018-09-10', title: 'Article 1' },
#   { __typename: 'Comment', author: 'Author 1', text: 'Comment 1' },
#   { __typename: 'UserProfile', age: 20, nickname: 'Nick 1' },
# ]
```

Подробно и с деталями — читать тут

Root types

Как вы уже знаете GraphQL-запросы иерархические и описывают древовидную структуру:

- если скалярные типы описывают листья вашего графа,
- Object-типы описывают промежуточные узлы,
- то Root-типы описывают корневые узлы (корни).

Root-типов всего три и они строятся с помощью обычного `GraphQLObjectType`:

- `Query` — этот тип описывает все доступные операции ("точки входа") для чтения. Если запросили несколько полей, то они выполняются параллельно.
- `Mutation` — для операции записи. Если запросили несколько полей, то они выполняются последовательно.
- `Subscription` — подписки, особый вид операций позволяющий клиентам подписаться на события произошедшие на сервере.

Root-типы непосредственно добавляются в `GraphQLSchema`:

```
import { GraphQLSchema, GraphQLObjectType, graphql } from 'graphql';

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({ name: 'Query', fields: { getUserById, findMany
  mutation: new GraphQLObjectType({ name: 'Mutation', fields: { createUser, rem
  subscriptions: new GraphQLObjectType({ name: 'Subscription', fields: ... }),
  // ... и ряд других настроек
});

// как схема готова, на ней можно выполнять запросы
const response = await graphql(schema, `query { ... }`);
```

Ваше API начинается именно с этих типов:

```
query {
  getUserById { ... }
  findManyUsers { ... }
}
```

Подробно и с деталями — читать тут

Directives

Директивы в GraphQL это дополнительные аннотации, которые:

- TypeSystemDirective — могут использоваться при описании схемы в SDL
- ExecutableDirective — могут использоваться в рантайме

TypeSystemDirective (при описании схемы)

- `@deprecated(reason: "Use 'newField'")` — встроенная в GraphQL директива
- Prisma добавляет свои `@unique`, `@default`, `@relation` и прочие
- Apollo graphql-tools дает удобную помогайку для создания директив через шаблон `visitor`

TypeSystemDirective (при описании схемы) — отличная штука, если хотите расширить возможности SDL

```
type Story {
  id: ID! @unique
  text: String!
  author: User! @relation(name: "WrittenStories")
}
```

Пример из Prisma

ExecutableDirective (в рантайме)

- `@skip(if: true)`, `@include(if: true)` — чтобы запрашивать часть графа по условию, идет в пакете graphql из коробки.
- `@defer` — фишка которая позволяет через лонг-полинг отложить получение данных из вашего запроса (надо шаманить сервер, в Apollo есть готовое решение).

Хотите свою директиву в рантайме?

```
const UppercaseDirective = new GraphQLDirective({
  name: 'uppercase',
  description: 'Provides default value for input field.',
  locations: [DirectiveLocation.FIELD],
});
```

```
const AuthorType = new GraphQLObjectType({
  name: 'Author',
  description: 'Author data with related data',
  fields: () => ({
    id: { type: GraphQLInt },
    name: {
      type: GraphQLString,
      resolve: (source, args, context, info) => {
        if (info.fieldNodes?.[0].directives?.[0]?.name?.value === 'uppercase')
          return source.name.toUpperCase();
        }
        return source.name;
      },
    },
  }),
```

Хотите свою директиву в рантайме?

```
const UppercaseDirective = new GraphQLDirective({
  name: 'uppercase',
  description: 'Provides default value for input field.',
  locations: [DirectiveLocation.FIELD],
});
```

```
const AuthorType = new GraphQLObjectType({
  name: 'Author',
  description: 'Author data with related data',
  fields: () => ({
    id: { type: GraphQLInt },
    name: {
      type: GraphQLString,
      resolve: (source, args, context, info) => {
        if (info.fieldNodes?.[0].directives?.[0]?.name?.value === 'uppercase')
          return source.name.toUpperCase();
        }
        return source.name;
      },
    },
  }),
```

Ужос! Индусский код!

Directives — итог

Директивы при создании схемы — OK!

Директивы в рантайме — только встроенные!

Подробно и с деталями — читать тут

Вся система типов очень подробно и доходчиво

расписана у меня в гитхабе

Вся система типов очень подробно и доходчиво

расписана у меня в гитхабе

на русском! 😀

Вся система типов очень подробно и доходчиво

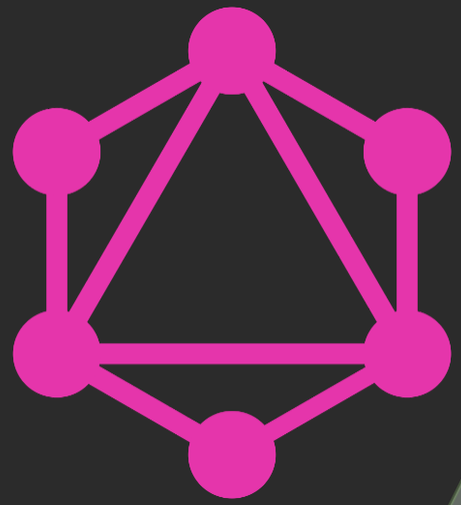
расписана у меня в гитхабе

на русском! 😀

Специально для вас в рамках подготовки к HolyJS

Что такое SDL и интроспекция?

graphql package



server package

Define GraphQL Schema

Execute GraphQL Query

Serve Client Requests

SDL — Schema Definition Language

Это простой формат описания GraphQL-схемы.

Он не зависит ни от какого языка программирования.

SDL

```
# Output-тип
type Post {
  id: Int!
  title: String!
  publishedAt: DateTime!
  comments(limit: Int = 10): [Comment]
}

# Input-тип
input Credentials {
  login: String!
  password: String!
}

# Enum-тип
```

SDL

```
# Output-тип
type Post {
  id: Int!
  title: String!
  publishedAt: DateTime!
  comments(limit: Int = 10): [Comment]
}

# Input-тип
input Credentials {
  login: String!
  password: String!
}

# Enum-тип
```

SDL

```
# Output-тип
type Post {
  id: Int!
  title: String!
  publishedAt: DateTime!
  comments(limit: Int = 10): [Comment]
}

# Input-тип
input Credentials {
  login: String!
  password: String!
}

# Enum-тип
```

SDL

```
# Output-тип
type Post {
  id: Int!
  title: String!
  publishedAt: DateTime!
  comments(limit: Int = 10): [Comment]
}

# Input-тип
input Credentials {
  login: String!
  password: String!
}

# Enum-тип
```

SDL

```
# Output-тип
type Post {
  id: Int!
  title: String!
  publishedAt: DateTime!
  comments(limit: Int = 10): [Comment]
}

# Input-тип
input Credentials {
  login: String!
  password: String!
}

# Enum-тип
```

SDL

```
# Output-тип
type Post {
  id: Int!
  title: String!
  publishedAt: DateTime!
  comments(limit: Int = 10): [Comment]
}

# Input-тип
input Credentials {
  login: String!
  password: String!
}

# Enum-тип
```

SDL

```
# Output-тип
type Post {
  id: Int!
  title: String!
  publishedAt: DateTime!
  comments(limit: Int = 10): [Comment]
}

# Input-тип
input Credentials {
  login: String!
  password: String!
}

# Enum-тип
```

SDL

```
# Output-тип
type Post {
  id: Int!
  title: String!
  publishedAt: DateTime!
  comments(limit: Int = 10): [Comment]
}

# Input-тип
input Credentials {
  login: String!
  password: String!
}

# Enum-тип
```

SDL

```
# Output-тип
type Post {
  id: Int!
  title: String!
  publishedAt: DateTime!
  comments(limit: Int = 10): [Comment]
}

# Input-тип
input Credentials {
  login: String!
  password: String!
}

# Enum-тип
```

SDL

```
# Output-тип
type Post {
  id: Int!
  title: String!
  publishedAt: DateTime!
  comments(limit: Int = 10): [Comment]
}

# Input-тип
input Credentials {
  login: String!
  password: String!
}

# Enum-тип
```

Интроспекция — это описание всех типов в вашей GraphQL-схеме.

Это ваша схема без resolve-методов.

Можно сравнить с `МЕНЮ в ресторане`

Видно что можно заказать, но не видно как и из чего готовится.

Пример интроспекции в формате SDL

```
type Query {
  book(id: Int): Book
  author(name: String): Author
}

# Author model
type Author {
  name: String!
}

type Book {
  id: Int!
  name: String!
  authors: [Author]
}
```

Вся ваша схема, только без `resolve`-методов.

Зачем нужна интроспекция клиенту?

- для IDE (GraphiQL, GraphQL Playground, Altair GraphQL Client)
- для линтеров, проверяющих корректность запросов в коде
- для тайпчекеров (Flowtype, TypeScript)
- для связывания микросервисной архитектуры

Для тупого выполнения запросов интроспекция не нужна.

Генерация интроспекции в формате SDL

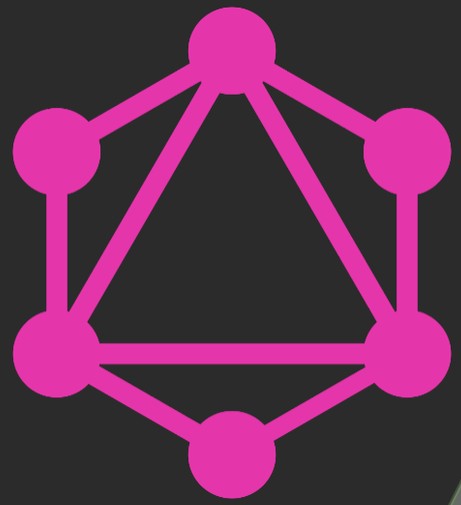
```
1 import fs from 'fs';
2 import { printSchema } from 'graphql';
3 import schema from './your-schema';
4
5 fs.writeFileSync('./schema.graphql', printSchema(schema));
```

Генерация интроспекции в формате JSON

```
1  import fs from 'fs';
2  import { getIntrospectionQuery } from 'graphql';
3  import schema from './your-schema';
4
5  async function prepareJsonFile() {
6    const result = await graphql(schema, getIntrospectionQuery());
7    fs.writeFileSync('./schema.json', JSON.stringify(result, null, 2));
8  }
9
10 prepareJsonFile();
```

4 подхода к построению схем

graphql package



server package

Define GraphQL Schema

Execute GraphQL Query

Serve Client Requests

- `graphql` — жестко и квадратно, нельзя редактировать типы.
- `graphql-tools` — описываете типы в SDL и отдельно резолверы.
- `graphql-compose` — упрощенный синтаксис создания типов, можно использовать SDL. Позволяет читать и редактировать типы. Удобно для написания собственных функций генераторов.
- `type-graphql` — свежак, использует декораторы поверх ваших классов (TypeScript).

`graphql` — vanilla

```
import {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLList,
  GraphQLNonNull,
} from 'graphql';
import { authors, articles } from './data';

const AuthorType = new GraphQLObjectType({
  name: 'Author',
  description: 'Author data',
  fields: () => ({
    id: { type: GraphQLInt },
```

https://github.com/graphql/graphql-js

`graphql` — vanilla

```
import {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLList,
  GraphQLNonNull,
} from 'graphql';
import { authors, articles } from './data';

const AuthorType = new GraphQLObjectType({
  name: 'Author',
  description: 'Author data',
  fields: () => ({
    id: { type: GraphQLInt },
```

https://github.com/graphql/graphql-js

`graphql` — vanilla

```
import {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLList,
  GraphQLNonNull,
} from 'graphql';
import { authors, articles } from './data';

const AuthorType = new GraphQLObjectType({
  name: 'Author',
  description: 'Author data',
  fields: () => ({
    id: { type: GraphQLInt },
```

https://github.com/graphql/graphql-js

`graphql` — vanilla

```
import {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLList,
  GraphQLNonNull,
} from 'graphql';
import { authors, articles } from './data';

const AuthorType = new GraphQLObjectType({
  name: 'Author',
  description: 'Author data',
  fields: () => ({
    id: { type: GraphQLInt },
```

https://github.com/graphql/graphql-js

`graphql` — vanilla

```
import {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLList,
  GraphQLNonNull,
} from 'graphql';
import { authors, articles } from './data';

const AuthorType = new GraphQLObjectType({
  name: 'Author',
  description: 'Author data',
  fields: () => ({
    id: { type: GraphQLInt },
```

https://github.com/graphql/graphql-js

`graphql` — vanilla

```
import {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLList,
  GraphQLNonNull,
} from 'graphql';
import { authors, articles } from './data';

const AuthorType = new GraphQLObjectType({
  name: 'Author',
  description: 'Author data',
  fields: () => ({
    id: { type: GraphQLInt },
```

https://github.com/graphql/graphql-js

graphql-tools — typeDefs & resolvers

```
1  import { makeExecutableSchema } from 'graphql-tools';
2  import { authors, articles } from './data';
3
4  const typeDefs = `
5    "Author data"
6    type Author {
7      id: Int
8      name: String
9    }
10
11   "Article data with related Author data"
12   type Article {
13     title: String!
14     text: String
15     "Record id from Author table"
```

https://github.com/apollographql/graphql-tools

graphql-tools — typeDefs & resolvers

```
1  import { makeExecutableSchema } from 'graphql-tools';
2  import { authors, articles } from './data';
3
4  const typeDefs = `
5    "Author data"
6    type Author {
7      id: Int
8      name: String
9    }
10
11   "Article data with related Author data"
12   type Article {
13     title: String!
14     text: String
15     "Record id from Author table"
```

https://github.com/apollographql/graphql-tools

graphql-tools — typeDefs & resolvers

```
1  import { makeExecutableSchema } from 'graphql-tools';
2  import { authors, articles } from './data';
3
4  const typeDefs = `
5    "Author data"
6    type Author {
7      id: Int
8      name: String
9    }
10
11   "Article data with related Author data"
12   type Article {
13     title: String!
14     text: String
15     "Record id from Author table"
```

https://github.com/apollographql/graphql-tools

graphql-tools — typeDefs & resolvers

```
1  import { makeExecutableSchema } from 'graphql-tools';
2  import { authors, articles } from './data';
3
4  const typeDefs = `
5    "Author data"
6    type Author {
7      id: Int
8      name: String
9    }
10
11   "Article data with related Author data"
12   type Article {
13     title: String!
14     text: String
15     "Record id from Author table"
```

https://github.com/apollographql/graphql-tools

graphql-tools — typeDefs & resolvers

```
1  import { makeExecutableSchema } from 'graphql-tools';
2  import { authors, articles } from './data';
3
4  const typeDefs = `
5    "Author data"
6    type Author {
7      id: Int
8      name: String
9    }
10
11   "Article data with related Author data"
12   type Article {
13     title: String!
14     text: String
15     "Record id from Author table"
```

https://github.com/apollographql/graphql-tools

graphql-tools — typeDefs & resolvers

```
1  import { makeExecutableSchema } from 'graphql-tools';
2  import { authors, articles } from './data';
3
4  const typeDefs = `
5    "Author data"
6    type Author {
7      id: Int
8      name: String
9    }
10
11   "Article data with related Author data"
12   type Article {
13     title: String!
14     text: String
15     "Record id from Author table"
```

https://github.com/apollographql/graphql-tools

graphql-tools — typeDefs & resolvers

```
1  import { makeExecutableSchema } from 'graphql-tools';
2  import { authors, articles } from './data';
3
4  const typeDefs = `
5    "Author data"
6    type Author {
7      id: Int
8      name: String
9    }
10
11   "Article data with related Author data"
12   type Article {
13     title: String!
14     text: String
15     "Record id from Author table"
```

https://github.com/apollographql/graphql-tools

`graphql-compose` — sugared vanilla + SDL

```
1  import { TypeComposer, schemaComposer } from 'graphql-compose';
2  import { authors, articles } from './data';
3
4  // SDL
5  const AuthorType = TypeComposer.create(`
6    "Author data"
7    type Author {
8      id: Int
9      name: String
10   }
11 `);
12
13 // Sugared vanilla
14 const ArticleType = TypeComposer.create({
15   name: 'Article',
```

https://github.com/graphql-compose/graphql-compose

`graphql-compose` — sugared vanilla + SDL

```
1  import { TypeComposer, schemaComposer } from 'graphql-compose';
2  import { authors, articles } from './data';
3
4  // SDL
5  const AuthorType = TypeComposer.create(`
6    "Author data"
7    type Author {
8      id: Int
9      name: String
10   }
11 `);
12
13 // Sugared vanilla
14 const ArticleType = TypeComposer.create({
15   name: 'Article',
```

https://github.com/graphql-compose/graphql-compose

`graphql-compose` — sugared vanilla + SDL

```
1  import { TypeComposer, schemaComposer } from 'graphql-compose';
2  import { authors, articles } from './data';
3
4  // SDL
5  const AuthorType = TypeComposer.create(`
6    "Author data"
7    type Author {
8      id: Int
9      name: String
10   }
11 `);
12
13 // Sugared vanilla
14 const ArticleType = TypeComposer.create({
15   name: 'Article',
```

https://github.com/graphql-compose/graphql-compose

`graphql-compose` — sugared vanilla + SDL

```
1  import { TypeComposer, schemaComposer } from 'graphql-compose';
2  import { authors, articles } from './data';
3
4  // SDL
5  const AuthorType = TypeComposer.create(`
6    "Author data"
7    type Author {
8      id: Int
9      name: String
10   }
11 `);
12
13 // Sugared vanilla
14 const ArticleType = TypeComposer.create({
15   name: 'Article',
```

https://github.com/graphql-compose/graphql-compose

`graphql-compose` — sugared vanilla + SDL

```
1  import { TypeComposer, schemaComposer } from 'graphql-compose';
2  import { authors, articles } from './data';
3
4  // SDL
5  const AuthorType = TypeComposer.create(`
6    "Author data"
7    type Author {
8      id: Int
9      name: String
10   }
11 `);
12
13 // Sugared vanilla
14 const ArticleType = TypeComposer.create({
15   name: 'Article',
```

https://github.com/graphql-compose/graphql-compose

`graphql-compose` — sugared vanilla + SDL

```
1  import { TypeComposer, schemaComposer } from 'graphql-compose';
2  import { authors, articles } from './data';
3
4  // SDL
5  const AuthorType = TypeComposer.create(`
6    "Author data"
7    type Author {
8      id: Int
9      name: String
10   }
11 `);
12
13 // Sugared vanilla
14 const ArticleType = TypeComposer.create({
15   name: 'Article',
```

https://github.com/graphql-compose/graphql-compose

`type-graphql` — decorators (TypeScript)

```typescript
import 'reflect-metadata';
import {
  ObjectType, Field, ID, String, type ResolverInterface,
} from 'type-graphql';
import { authors, articles } from './data';

@ObjectType({ description: 'Author data' })
class Author {
  @Field(type => ID)
  id: number;

  @Field(type => String, { nullable: true })
  name: string;
}

```

https://github.com/19majkel94/type-graphql

`type-graphql` — decorators (TypeScript)

```typescript
import 'reflect-metadata';
import {
  ObjectType, Field, ID, String, type ResolverInterface,
} from 'type-graphql';
import { authors, articles } from './data';

@ObjectType({ description: 'Author data' })
class Author {
  @Field(type => ID)
  id: number;

  @Field(type => String, { nullable: true })
  name: string;
}

```

https://github.com/19majkel94/type-graphql

`type-graphql` — decorators (TypeScript)

```typescript
import 'reflect-metadata';
import {
  ObjectType, Field, ID, String, type ResolverInterface,
} from 'type-graphql';
import { authors, articles } from './data';

@ObjectType({ description: 'Author data' })
class Author {
  @Field(type => ID)
  id: number;

  @Field(type => String, { nullable: true })
  name: string;
}

```

https://github.com/19majkel94/type-graphql

`type-graphql` — decorators (TypeScript)

```typescript
import 'reflect-metadata';
import {
  ObjectType, Field, ID, String, type ResolverInterface,
} from 'type-graphql';
import { authors, articles } from './data';

@ObjectType({ description: 'Author data' })
class Author {
  @Field(type => ID)
  id: number;

  @Field(type => String, { nullable: true })
  name: string;
}

```

https://github.com/19majkel94/type-graphql

`type-graphql` — decorators (TypeScript)

```typescript
import 'reflect-metadata';
import {
  ObjectType, Field, ID, String, type ResolverInterface,
} from 'type-graphql';
import { authors, articles } from './data';

@ObjectType({ description: 'Author data' })
class Author {
  @Field(type => ID)
  id: number;

  @Field(type => String, { nullable: true })
  name: string;
}

```

https://github.com/19majkel94/type-graphql

`graphql-tools`, `graphql-compose`, `type-graphql`

`graphql-tools`, `graphql-compose`, `type-graphql`

только строят GraphQL-схему

`graphql-tools`, `graphql-compose`, `type-graphql`

только строят GraphQL-схему

с помощью пакета `graphql`

`graphql-tools`, `graphql-compose`, `type-graphql`

только строят GraphQL-схему

с помощью пакета `graphql`

В runtime опять-таки используется `graphql`

На закуску — Генераторы схем

Prisma

Prisma — ORM прослойка на GraphQL, генерирует базу (Postgres, MySQL, more to come) и GraphQL API со всеми базовыми CRUD операциями. Дальше вы можете строить свой уникальный GraphQL API (используя подход `graphql-tools`), либо пользоваться уже сгенерированным.

Hasura

Hasura — работает на интроспекции Postgres, плюс задает пермишенны и реляции между таблицами. Захотите свою кастомную схему, опять будите брать подход `graphql-tools` и ститчить (склеивать) вместе несколько схем, либо использовать knex для хитрого получения данных.

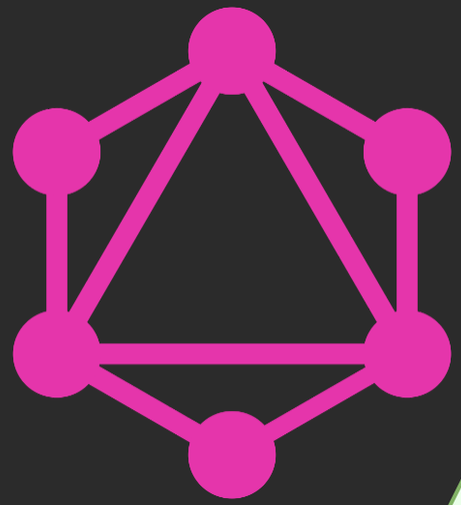
graphql-compose-mongoose

graphql-compose-mongoose — на базе ваших mongoose-моделей для MongoDB генерирует типы и резолверы (кусочки для схемы). А дальше вы с помощью подхода `graphql-compose` собираете свою схему сразу так, как вам нужно.

А еще есть graphql-compose-elasticsearch

GraphQL-сервер

graphql package

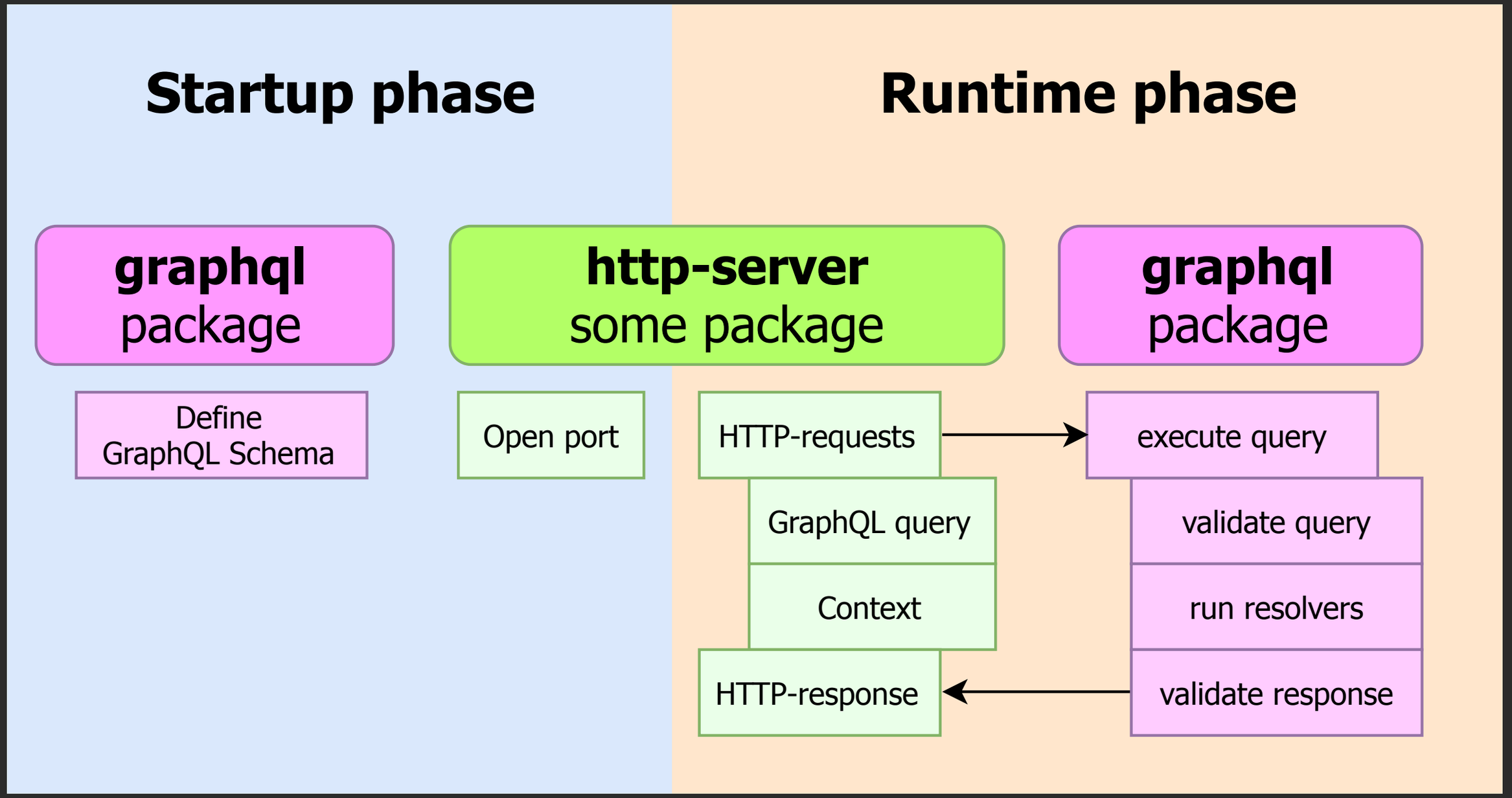


server package

Define GraphQL Schema

Execute GraphQL Query

Serve Client Requests


Startup phase
Runtime phase
graphql
package
http-server
some package
graphql
package
Define
GraphQL Schema
Open port
HTTP-requests
GraphQL query
Context
HTTP-response
execute query
validate query
run resolvers
validate response

Пакет `graphql` ничего не знает о сети, авторизации, не слушает никакой порт.

```
1 import { graphql } from 'graphql';
2 import { schema } from './your-schema';
3
4 const query = '{ hello }';
5 const result = await graphql(schema, query);
```

т.к. GraphQL-сервер реализуется на другом уровне абстракции.

в других пакетах

Каковы обычно требования к серверу?

Каковы обычно требования к серверу?

По некому протоколу обслуживать множество запросов от разных клиентов.

Каковы обычно требования к серверу?

По некому протоколу обслуживать множество запросов от разных клиентов.

Это может быть `http(s)` или `websockets`, либо вообще что-то экзотическое типа `ssh`, `telnet`.

Что должен делать HTTP-сервер

- открыть порт и слушать http-запросы от клиентов
- инициализировать GraphQL-схему
- вытаскивать GraphQL-запрос с переменными из полученных http-запросов
- формировать `context` с данными текущего пользователя и глобальными ресурсами
- отправить на выполнение GraphQL-схему, запрос и `context` в пакет `graphql`
- из полученных данных от `graphql` сформировать http-ответ и отправить клиенту

По-желанию попутно сделать всякие операции, типа:

- парсинга кук
- проверки токенов
- базовая авторизация
- превращение persistent query по id в GraphQL-запрос
- логирования запросов
- кеша запросов
- отлова ошибок и отправки их в систему мониторинга.

Популярные сервера:

- express-graphql
- koa-graphql
- apollo-server@1.x.x
- apollo-server@2.x.x
- graphql-yoga

TL;DR: берите `apollo-server@2.x.x`

express-graphql

```
1  import express from 'express';
2  import graphqlHTTP from 'express-graphql';
3  import schema from './schema';
4
5  const app = express();
6
7  app.use('/graphql', graphqlHTTP(req => async ({
8    schema,
9    graphiql: true,
10   context: await prepareSomehowContextDataFromRequest(req),
11 })));
12
13 app.listen(3000);
```

Это даже не сервер, а `middleware` для `express`.

Минусы: ограничение запроса в 100kb, нет подписок.

koa-graphql

```
1  import Koa from 'koa';
2  import mount from 'koa-mount';
3  import graphqlHTTP from 'koa-graphql';
4  import schema from './schema';
5
6  const app = new Koa();
7
8  app.use(mount('/graphql', graphqlHTTP(req => async ({
9    schema,
10   graphiql: true,
11   context: await prepareSomehowContextDataFromRequest(req),
12 }))));
13
14 app.listen(4000);
```

Такое же middleware как `express-graphql`, только для `Koa`

apollo-server@1.x.x

```
1  import express from 'express';
2  import bodyParser from 'body-parser';
3  import { graphqlExpress, graphiqlExpress } from 'apollo-server-express';
4  import schema from './schema';
5
6  const app = express();
7  app.use('/graphql', bodyParser.json(), graphqlExpress(req => async ({
8    schema,
9    context: await prepareSomehowContextDataFromRequest(req),
10 })));
11 app.get('/graphiql', graphiqlExpress({ endpointURL: '/graphql' }));
12
13 app.listen(5000);
```

Больше гибкости и наворотов,

по сравнению с `express-graphql`

apollo-server@2.x.x

```
import { ApolloServer, gql } from 'apollo-server';

const typeDefs = gql`
  type Query {
    hello: String
  }
`;

const resolvers = {
  Query: {
    hello: () => 'world',
  },
};

const server = new ApolloServer({
```

apollo-server@2.x.x

```
import { ApolloServer, gql } from 'apollo-server';

const typeDefs = gql`
  type Query {
    hello: String
  }
`;

const resolvers = {
  Query: {
    hello: () => 'world',
  },
};

const server = new ApolloServer({
```

Стал ближе к `graphql-tools`

apollo-server@2.x.x

```
1  import { ApolloServer, gql } from 'apollo-server';
2
3  const typeDefs = gql`
4    type Query {
5      hello: String
6    }
7  `;
8
9  const resolvers = {
10   Query: {
11     hello: () => 'world',
12   },
13 };
14
15 const server = new ApolloServer({
```

Стал ближе к `graphql-tools`

Можно сразу передавать `typeDefs` и `resolvers`

apollo-server@2.x.x

- Subscriptions через PubSub
- поддержка загрузки файлов
- persisted queries
- всё ещё можно передать `schema`
- можно не ставить Express или Koa
- и еще удобнее сделали интеграцию со своими платными сервисами

При написании этой версии

всё что можно "повзаимвствовали" у `graphql-yoga` 😊

graphql-yoga

```
1  import { GraphQLServer } from 'graphql-yoga'
2
3  const typeDefs = ...;
4  const resolvers = ...;
5
6  const server = new GraphQLServer({
7    typeDefs,
8    resolvers,
9    context: (req) => prepareSomehowContextDataFromRequest(req),
10 });
11 server.start({
12   port: 7000,
13   endpoint: '/graphql',
14   playground: '/playground',
15 })
```

Шайба в шайбу с `apollo-server@2.x.x` ☺️

Важно про context

Задача сервера сформировать `context` для того, чтобы можно было разделить пользователей друг от друга при выполнении GraphQL-запросов.

```
1 const server = new ApolloServer({ schema,
2   context: ({ req }) => prepareSomehowContextDataFromRequest(req),
3 });
```

`context` это объект с данными текущего пользователя и глобальными ресурсами, которые будут доступны в резолверах вашей GraphQL-схемы

Подробнее о серверах, читать тут

Запускаем сервер

Запускаем сервер на NodeJS

```
import { ApolloServer } from 'apollo-server';
import { GraphQLSchema, GraphQLObjectType, GraphQLString } from 'graphql';

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      hello: {
        type: GraphQLString,
        args: {
          name: { type: GraphQLString, defaultValue: 'world' },
        },
        resolve: (source, args, context) => {
          return `Hello, ${args.name} from ip ${context.req.ip}`;
        },
```

Запускаем сервер на NodeJS

```
import { ApolloServer } from 'apollo-server';
import { GraphQLSchema, GraphQLObjectType, GraphQLString } from 'graphql';

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      hello: {
        type: GraphQLString,
        args: {
          name: { type: GraphQLString, defaultValue: 'world' },
        },
        resolve: (source, args, context) => {
          return `Hello, ${args.name} from ip ${context.req.ip}`;
        },
```

Запускаем сервер на NodeJS

```
import { ApolloServer } from 'apollo-server';
import { GraphQLSchema, GraphQLObjectType, GraphQLString } from 'graphql';

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      hello: {
        type: GraphQLString,
        args: {
          name: { type: GraphQLString, defaultValue: 'world' },
        },
        resolve: (source, args, context) => {
          return `Hello, ${args.name} from ip ${context.req.ip}`;
        },
```

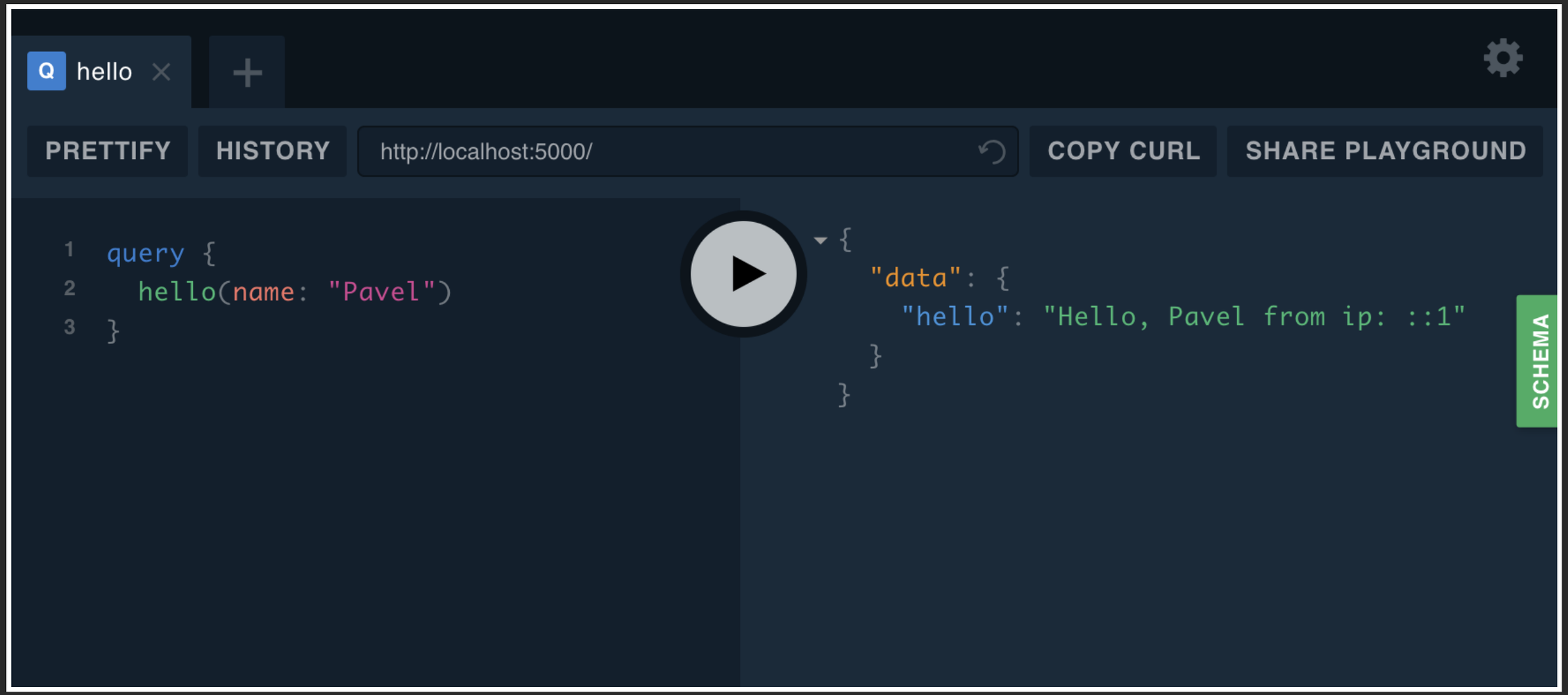
Запускаем сервер на NodeJS

```
import { ApolloServer } from 'apollo-server';
import { GraphQLSchema, GraphQLObjectType, GraphQLString } from 'graphql';

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      hello: {
        type: GraphQLString,
        args: {
          name: { type: GraphQLString, defaultValue: 'world' },
        },
        resolve: (source, args, context) => {
          return `Hello, ${args.name} from ip ${context.req.ip}`;
        },
```

Запускаем сервер на NodeJS

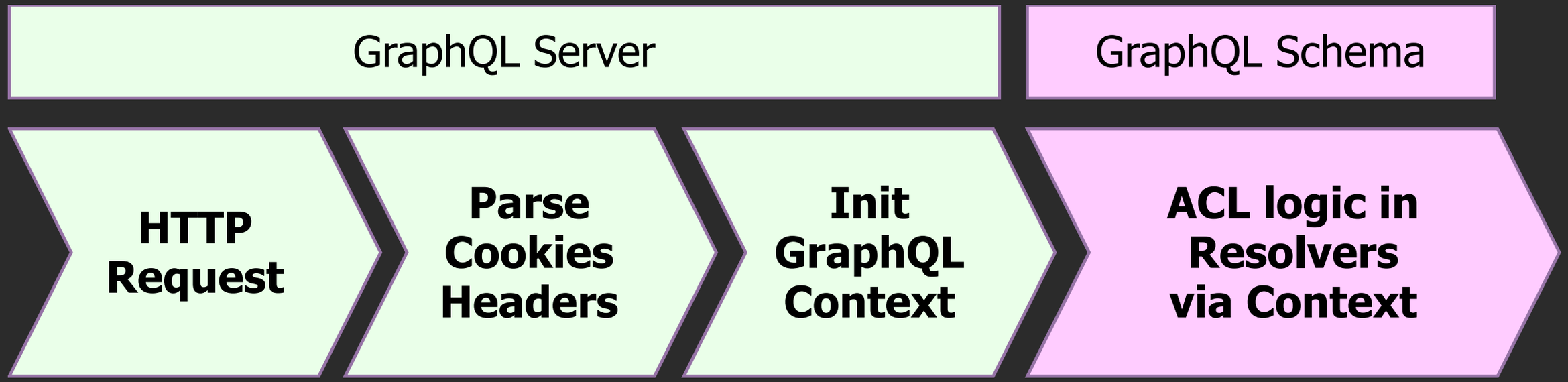
```
import { ApolloServer } from 'apollo-server';
import { GraphQLSchema, GraphQLObjectType, GraphQLString } from 'graphql';

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      hello: {
        type: GraphQLString,
        args: {
          name: { type: GraphQLString, defaultValue: 'world' },
        },
        resolve: (source, args, context) => {
          return `Hello, ${args.name} from ip ${context.req.ip}`;
        },
```

Результат в браузере



Section #3
АВТОРИЗАЦИЯ

Auth flow diagram



Но сперва вспомним, что такое...

Аутентификация

Идентификация

Авторизация

Но сперва вспомним, что такое...

Аутентификация	процедура проверки подлинности пользователя путём сравнения введённого им логина и пароля.
Идентификация	процедура распознания пользователя по токену, сессии или кукам.
Авторизация	процедура проверки прав доступа к ресурсам на выполнение определённых действий.

1. Аутентификация — Sign In

- **Без GraphQL**
  - отправляем логин/пароль через старый добрый POST, получаем токен
  - используем OAuth, получаем токен
- **Через GraphQL**
  - крутим query или mutation, через аргументы передаем логин/пароль в респонсе получаем токен; плюс сразу можем получить вагон данных

В результате Аутентификации должен быть TOKEH

2. Идентификация — JWT, cookie

В мире GraphQL для генерации токенов сильно прижился JWT

Читать про JSON Web Token в википедии.

JWT-токен состоит из:

. .

Наример:

Внутрянка:

JWT-токен состоит из:

header. . .

Наример:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

Внутрянка:

base64({ "alg": "HS256", "typ": "JWT" }).

JWT-токен состоит из:

header.payload.

Наример:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOjEsImlhdCI6MTU0MTI1MDE2M30.

Внутрянка:

base64({ "alg": "HS256", "typ": "JWT" }).
base64({ "sub": 1, "iat": 1541250163 }).

JWT-токен состоит из:

header.payload.sign

Наример:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOjEsImlhdCI6MTU0MTI1MDE2M30.
M7xYg8GuEgwbqTrta0xnN7WmNEXOCKiQGDdogt_Kduk

Внутрянка:

base64({ "alg": "HS256", "typ": "JWT" }).
base64({ "sub": 1, "iat": 1541250163 }).
HMACSHA256(base64UrlEncode(header) + "." +
base64UrlEncode(payload), JWT_SECRET_KEY)

С JWT легко работать на сервере

```
import jwt from 'jsonwebtoken';

const JWT_SECRET_KEY = 'qwerty ;)';

// Генерация токена
const token = jwt.sign({ sub: 2 }, JWT_SECRET_KEY);

// Проверка токена
const payload = jwt.verify(token, JWT_SECRET_KEY);
// { "sub": 1, "iat": 1541250163 }
// sub - это subject, например id пользователя
// iat - это время генерации токена
// а еще есть `iss`, `aud`, `exp`, `nbf`, `jti` - смотри спеку
```

Где хранить TOKEN на клиенте?

- `Если браузер` — то в куках с флагом httpOnly

- `Если мобильное приложение` — то в "AsyncStorage"

Где хранить TOKEN на клиенте?

- `Если браузер` — то в куках с флагом httpOnly
  - делается только бэкендером, фронтендеру ничего делать не нужно

- `Если мобильное приложение` — то в "AsyncStorage"

Где хранить TOKEN на клиенте?

- `Если браузер` — то в куках с флагом httpOnly
  - делается только бэкендером, фронтендеру ничего делать не нужно
  - если фронтендер в браузере хоть как-то сохранит токен, то привет XSS ☠️
- `Если мобильное приложение` — то в "AsyncStorage"

Где хранить TOKEN на клиенте?

- `Если браузер` — то в куках с флагом httpOnly
  - делается только бэкендером, фронтендеру ничего делать не нужно
  - если фронтендер в браузере хоть как-то сохранит токен, то привет XSS ☠️
- `Если мобильное приложение` — то в "AsyncStorage"
  - передаем токен через HTTP-заголовки с каждым запросом

☝️ Пссс, Бэкендер! ☝️

Ты должен:

- уметь принимать и ставить токен через httpOnly куки
- а если там пусто, то посмотреть в HTTP-заголовках
- иначе Индетификация не прошла и перед тобой аноним 👻

3. Авторизация — Прикручиваем ACL

Ее можно и нужно настраивать на следующих уровнях:

- на уровне сервера (apollo, express, koa и пр.)
- на уровне GraphQL-схемы (глобально на первых полях схемы)
- на уровне полей (в resolve методах)
- на уровне связей между типами (в resolve методах)

3.1. Авторизация на уровне сервера (apollo, express, koa и пр.)

- считать токен из кук, либо заголовков
- провалидировать токен и произвести индетификацию пользователя
- передать пользователя в `context` graphql
- либо завернуть запрос, если токен невалиден или пользователь забанен

Пишем функцию помогайку получения пользователя из реквеста:

```
async function getUserFromReq(req: $Request) {
  const token = req?.cookies?.token || req?.headers?.authorization;
  if (token) {
    const payload = jwt.verify(token, JWT_SECRET_KEY);
    const userId = payload?.sub;
    if (userId) {
      const user = await users.find(userId);
      if (user) {
        if (user.isBanned) throw new Error('Looser!');
        return user;
      }
    }
  }
  return null;
}
```

Пишем функцию помогайку получения пользователя из реквеста:

```
async function getUserFromReq(req: $Request) {
  const token = req?.cookies?.token || req?.headers?.authorization;
  if (token) {
    const payload = jwt.verify(token, JWT_SECRET_KEY);
    const userId = payload?.sub;
    if (userId) {
      const user = await users.find(userId);
      if (user) {
        if (user.isBanned) throw new Error('Looser!');
        return user;
      }
    }
  }
  return null;
}
```

Пишем функцию помогайку получения пользователя из реквеста:

```
async function getUserFromReq(req: $Request) {
  const token = req?.cookies?.token || req?.headers?.authorization;
  if (token) {
    const payload = jwt.verify(token, JWT_SECRET_KEY);
    const userId = payload?.sub;
    if (userId) {
      const user = await users.find(userId);
      if (user) {
        if (user.isBanned) throw new Error('Looser!');
        return user;
      }
    }
  }
  return null;
}
```

Пишем функцию помогайку получения пользователя из реквеста:

```
async function getUserFromReq(req: $Request) {
  const token = req?.cookies?.token || req?.headers?.authorization;
  if (token) {
    const payload = jwt.verify(token, JWT_SECRET_KEY);
    const userId = payload?.sub;
    if (userId) {
      const user = await users.find(userId);
      if (user) {
        if (user.isBanned) throw new Error('Looser!');
        return user;
      }
    }
  }
  return null;
}
```

Пишем функцию помогайку получения пользователя из реквеста:

```
async function getUserFromReq(req: $Request) {
  const token = req?.cookies?.token || req?.headers?.authorization;
  if (token) {
    const payload = jwt.verify(token, JWT_SECRET_KEY);
    const userId = payload?.sub;
    if (userId) {
      const user = await users.find(userId);
      if (user) {
        if (user.isBanned) throw new Error('Looser!');
        return user;
      }
    }
  }
  return null;
}
```

Ну и на сервере формируем GraphQL-контекст

```
const server = new ApolloServer({
  schema,
  context: async ({ req }) => {
    let user;
    try {
      user = await getUserFromReq(req);
    } catch (e) {
      throw new AuthenticationError('You provide incorrect token!');
    }
    const hasRole = (role) => { // Примитивный RBAC
      if (user && Array.isArray(user.roles)) return user.roles.includes(role);
      return false;
    }
    return { req, user, hasRole };
  },
```

Ну и на сервере формируем GraphQL-контекст

```
const server = new ApolloServer({
  schema,
  context: async ({ req }) => {
    let user;
    try {
      user = await getUserFromReq(req);
    } catch (e) {
      throw new AuthenticationError('You provide incorrect token!');
    }
    const hasRole = (role) => { // Примитивный RBAC
      if (user && Array.isArray(user.roles)) return user.roles.includes(role);
      return false;
    }
    return { req, user, hasRole };
  },
```

Ну и на сервере формируем GraphQL-контекст

```
const server = new ApolloServer({
  schema,
  context: async ({ req }) => {
    let user;
    try {
      user = await getUserFromReq(req);
    } catch (e) {
      throw new AuthenticationError('You provide incorrect token!');
    }
    const hasRole = (role) => { // Примитивный RBAC
      if (user && Array.isArray(user.roles)) return user.roles.includes(role);
      return false;
    }
    return { req, user, hasRole };
  },
```

Ну и на сервере формируем GraphQL-контекст

```
const server = new ApolloServer({
  schema,
  context: async ({ req }) => {
    let user;
    try {
      user = await getUserFromReq(req);
    } catch (e) {
      throw new AuthenticationError('You provide incorrect token!');
    }
    const hasRole = (role) => { // Примитивный RBAC
      if (user && Array.isArray(user.roles)) return user.roles.includes(role);
      return false;
    }
    return { req, user, hasRole };
  },
```

Ну и на сервере формируем GraphQL-контекст

```
const server = new ApolloServer({
  schema,
  context: async ({ req }) => {
    let user;
    try {
      user = await getUserFromReq(req);
    } catch (e) {
      throw new AuthenticationError('You provide incorrect token!');
    }
    const hasRole = (role) => { // Примитивный RBAC
      if (user && Array.isArray(user.roles)) return user.roles.includes(role);
      return false;
    }
    return { req, user, hasRole };
  },
```

Ну и на сервере формируем GraphQL-контекст

```
const server = new ApolloServer({
  schema,
  context: async ({ req }) => {
    let user;
    try {
      user = await getUserFromReq(req);
    } catch (e) {
      throw new AuthenticationError('You provide incorrect token!');
    }
    const hasRole = (role) => { // Примитивный RBAC
      if (user && Array.isArray(user.roles)) return user.roles.includes(role);
      return false;
    }
    return { req, user, hasRole };
  },
```

3.2. Авторизация на уровне GraphQL-схемы

(глобально на верхних полях схемы)

Это когда на первом уровне в `Query` размещаются так называемые namespace-типы.

Еще их можно назвать поля-роли `viewer`, `me`, `admin` и пр.

```
query {
  viewer { # любые пользователи имееют доступ к получению данных
    getNews
    getAds
  }
  me { # здесь отображаются данные только для текущего пользователя
    nickname
    photo
  }
  admin { # а здесь методы, которые доступны только админам
    shutdown
    exposePersonalData
  }
}
```

Тоже самое можно применить к мутациям.

Как это работает?

Если resolve-метод для поля вернул null, undefined или выбросил ошибку, то обработка вложенных полей не происходит.

```
const AdminNamespace = new GraphQLObjectType({
  name: 'AdminNamespace',
  fields: () => ({
    shutdown: { ... },
    exposePersonalData: { ... },
  }),
});

const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: { ... },
    me: { ... },
    admin: {
      type: AdminNamespace,
```

```
const AdminNamespace = new GraphQLObjectType({
  name: 'AdminNamespace',
  fields: () => ({
    shutdown: { ... },
    exposePersonalData: { ... },
  }),
});

const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: { ... },
    me: { ... },
    admin: {
      type: AdminNamespace,
```

```
const AdminNamespace = new GraphQLObjectType({
  name: 'AdminNamespace',
  fields: () => ({
    shutdown: { ... },
    exposePersonalData: { ... },
  }),
});

const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: { ... },
    me: { ... },
    admin: {
      type: AdminNamespace,
```

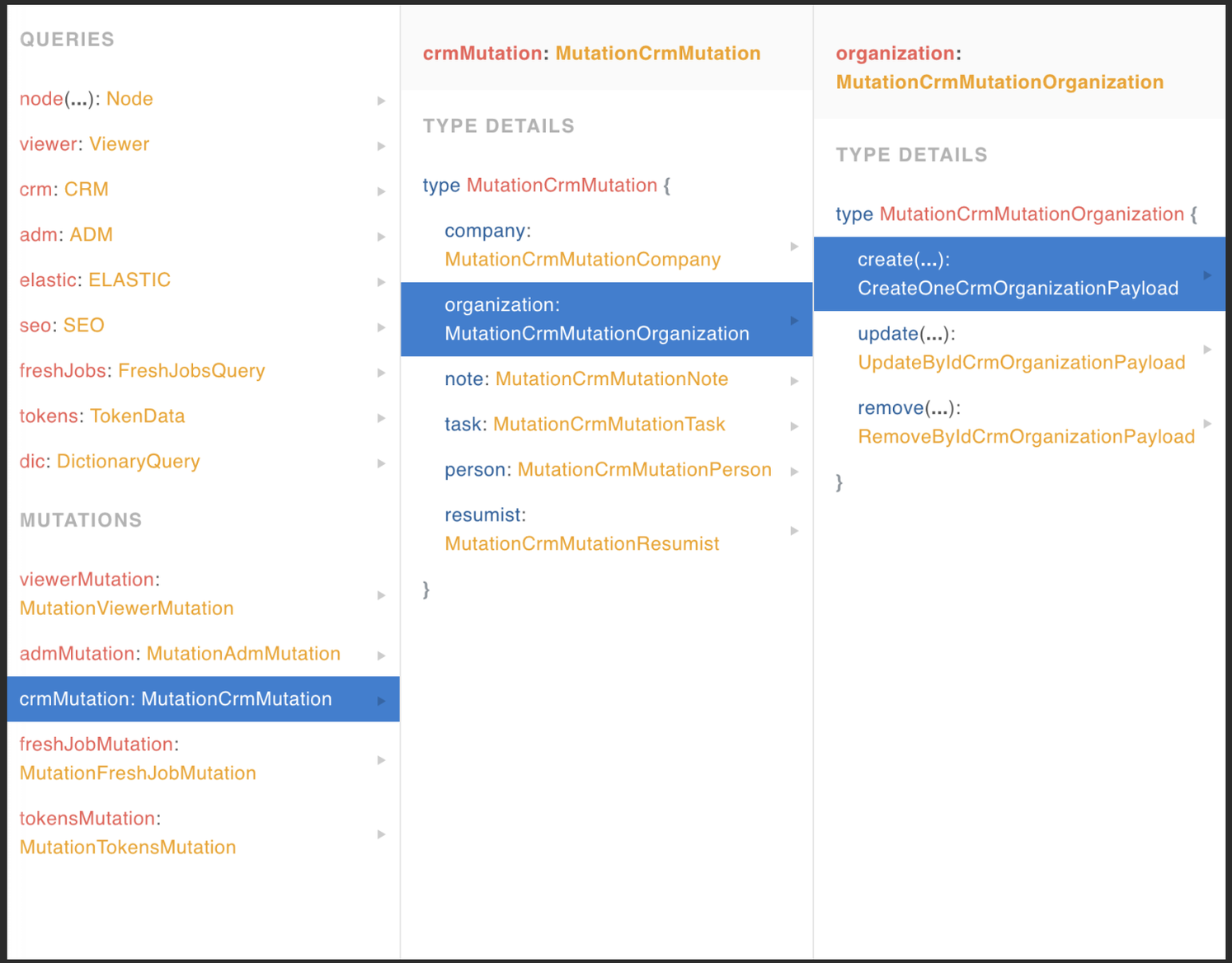
```
const AdminNamespace = new GraphQLObjectType({
  name: 'AdminNamespace',
  fields: () => ({
    shutdown: { ... },
    exposePersonalData: { ... },
  }),
});

const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: { ... },
    me: { ... },
    admin: {
      type: AdminNamespace,
```

```
const AdminNamespace = new GraphQLObjectType({
  name: 'AdminNamespace',
  fields: () => ({
    shutdown: { ... },
    exposePersonalData: { ... },
  }),
});

const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: { ... },
    me: { ... },
    admin: {
      type: AdminNamespace,
```

```
const AdminNamespace = new GraphQLObjectType({
  name: 'AdminNamespace',
  fields: () => ({
    shutdown: { ... },
    exposePersonalData: { ... },
  }),
});

const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: { ... },
    me: { ... },
    admin: {
      type: AdminNamespace,
```

```
const AdminNamespace = new GraphQLObjectType({
  name: 'AdminNamespace',
  fields: () => ({
    shutdown: { ... },
    exposePersonalData: { ... },
  }),
});

const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: { ... },
    me: { ... },
    admin: {
      type: AdminNamespace,
```

Неймспейсы еще хороши тем, что позволяют красиво нарезать ваше API и не делать из него помойку (как например это делает Prisma).


QUERIES
node(...): Node
viewer: Viewer
crm: CRM
adm: ADM
elastic: ELASTIC
seo: SEO
freshJobs: FreshJobsQuery
tokens: TokenData
dic: DictionaryQuery
MUTATIONS
viewerMutation: MutationViewerMutation
admMutation: MutationAdmMutation
crmMutation: MutationCrmMutation
freshJobMutation: MutationFreshJobMutation
tokensMutation: MutationTokensMutation
crmMutation: MutationCrmMutation
TYPE DETAILS
type MutationCrmMutation {
company: MutationCrmMutationCompany
organization: MutationCrmMutationOrganization
note: MutationCrmMutationNote
task: MutationCrmMutationTask
person: MutationCrmMutationPerson
resumist: MutationCrmMutationResumist
}
organization: MutationCrmMutationOrganization
TYPE DETAILS
type MutationCrmMutationOrganization {
create(...): CreateOneCrmOrganizationPayload
update(...): UpdateByIdCrmOrganizationPayload
remove(...): RemoveByIdCrmOrganizationPayload
}

3.3. Авторизация на уровне полей (в resolve-методах)

Когда у вас в `context` есть информация о текущем пользователе и его роли, то можно настроить авторизацию для конкретных полей.

Дано:

У нас есть Пользователь и мы храним
его последний IP-адрес в поле `lastIp`.

Задача:

Разрешить отображение ip-адреса
только админу и самому пользователю.

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    lastIp: {
      type: GraphQLString,
      resolve: (source, _, context) => {
        const { id, lastIp } = source;

        // return IP for ADMIN
        if (context.hasRole('ADMIN')) return lastIp;

        // return IP for current user
```

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    lastIp: {
      type: GraphQLString,
      resolve: (source, _, context) => {
        const { id, lastIp } = source;

        // return IP for ADMIN
        if (context.hasRole('ADMIN')) return lastIp;

        // return IP for current user
```

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    lastIp: {
      type: GraphQLString,
      resolve: (source, _, context) => {
        const { id, lastIp } = source;

        // return IP for ADMIN
        if (context.hasRole('ADMIN')) return lastIp;

        // return IP for current user
```

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    lastIp: {
      type: GraphQLString,
      resolve: (source, _, context) => {
        const { id, lastIp } = source;

        // return IP for ADMIN
        if (context.hasRole('ADMIN')) return lastIp;

        // return IP for current user
```

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    lastIp: {
      type: GraphQLString,
      resolve: (source, _, context) => {
        const { id, lastIp } = source;

        // return IP for ADMIN
        if (context.hasRole('ADMIN')) return lastIp;

        // return IP for current user
```

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    lastIp: {
      type: GraphQLString,
      resolve: (source, _, context) => {
        const { id, lastIp } = source;

        // return IP for ADMIN
        if (context.hasRole('ADMIN')) return lastIp;

        // return IP for current user
```

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    lastIp: {
      type: GraphQLString,
      resolve: (source, _, context) => {
        const { id, lastIp } = source;

        // return IP for ADMIN
        if (context.hasRole('ADMIN')) return lastIp;

        // return IP for current user
```

3.4. Авторизация на уровне связей между типами (в resolve-методах)

Это практически тоже самое что и авторизация на уровне полей.

Но есть нюанс ☝️

Вы должны проверить не просто возможность получения связанных объектов,

но и сами полученные объекты, на право отображания.

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    metaList: {
      type: new GraphQLList(MetaDataType),
      resolve: async (source, _, context) => {
        const { id } = source;

        // тут если надо проверяем есть ли доступ (как в пункте 3)

        // если доступ есть, то получаем данные
        let metaList = await Meta.find(o => o.userId === id);
```

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    metaList: {
      type: new GraphQLList(MetaDataType),
      resolve: async (source, _, context) => {
        const { id } = source;

        // тут если надо проверяем есть ли доступ (как в пункте 3)

        // если доступ есть, то получаем данные
        let metaList = await Meta.find(o => o.userId === id);
```

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    metaList: {
      type: new GraphQLList(MetaDataType),
      resolve: async (source, _, context) => {
        const { id } = source;

        // тут если надо проверяем есть ли доступ (как в пункте 3)

        // если доступ есть, то получаем данные
        let metaList = await Meta.find(o => o.userId === id);
```

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    metaList: {
      type: new GraphQLList(MetaDataType),
      resolve: async (source, _, context) => {
        const { id } = source;

        // тут если надо проверяем есть ли доступ (как в пункте 3)

        // если доступ есть, то получаем данные
        let metaList = await Meta.find(o => o.userId === id);
```

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    metaList: {
      type: new GraphQLList(MetaDataType),
      resolve: async (source, _, context) => {
        const { id } = source;

        // тут если надо проверяем есть ли доступ (как в пункте 3)

        // если доступ есть, то получаем данные
        let metaList = await Meta.find(o => o.userId === id);
```

```
const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    name: {
      type: new GraphQLNonNull(GraphQLString),
    },
    metaList: {
      type: new GraphQLList(MetaDataType),
      resolve: async (source, _, context) => {
        const { id } = source;

        // тут если надо проверяем есть ли доступ (как в пункте 3)

        // если доступ есть, то получаем данные
        let metaList = await Meta.find(o => o.userId === id);
```

Авторизация на уровне связей между типами делает проверку

- до получения данных
- и после получения данных

Иначе можно отдать данные, которые отдавать не стоит 😉

RECAP: Мы разобрали с вами авторизацию

- на уровне сервера (apollo, express, koa и пр.)
- на уровне GraphQL-схемы (глобально на первых полях схемы)
- на уровне полей (в resolve-методах)
- на уровне связей между типами (в resolve-методах)

Что можно ещё накрутить с авторизацией?

А что если ограничивать доступ по путям полей в запросе?

```
mutation {
  login { ... } # GUEST
  logout { ... } # USER
}

query {
  articles { ... } # USER
  me {
    debugInfo { ... } # only for ADMIN
    profile { ... } # USER
  }
}
```

```
mutation {
  login { ... } # GUEST
  logout { ... } # USER
}

query {
  articles { ... } # USER
  me {
    debugInfo { ... } # only for ADMIN
    profile { ... } # USER
  }
}
```

Можно описать такой политикой

```
ADMIN: # имеет доступ ко всему
  *
USER: # имеет доступ только к следующим путям графа
  articles.*
  me.profile.*
  logout.*
GUEST: # может вызвать только login
  login.*
```

Текущий путь поля в запросе можно получить из `info`:

```
resolve: (source, args, context, info) => {

  const path = getPathFromInfo(info);

  if (!context.hasAccess(path)) return null;

  // ...
},
```

Текущий путь поля в запросе можно получить из `info`:

```
resolve: (source, args, context, info) => {

  const path = getPathFromInfo(info);

  if (!context.hasAccess(path)) return null;

  // ...
},
```

Текущий путь поля в запросе можно получить из `info`:

```
resolve: (source, args, context, info) => {

  const path = getPathFromInfo(info);

  if (!context.hasAccess(path)) return null;

  // ...
},
```

Реализация `getPathFromInfo`

```
function getPathFromInfo(info: GraphQLResolveInfo): Array<string | number> | fa
  if (!info || !info.path) return false;
  const res = [];
  let curPath = info.path;
  while (curPath) {
    if (curPath.key) {
      res.unshift(curPath.key);
      if (curPath.prev) curPath = curPath.prev;
      else break;
    } else break;
  }
  return res;
}
```

```
// На входе в `info.path`
{ prev: { prev: { prev: undefined, key: 'articles' }, key: 0 }, key: 'author' }

// На выходе:
['articles', 0, 'author']
```

А как реализовать `context.hasAccess(path)`?

А как реализовать `context.hasAccess(path)`?

Ну это дело техники 😜

А как реализовать `context.hasAccess(path)`?

Ну это дело техники 😜

Объявляется он как `hasRole` из раздела 3.1. Авторизация на уровне сервера

Самое главное идею передать!

Подробнее про Авторизацию читаем тут

Авторизация Appendix:

Почему я использую три токена

`user`, `account`, `admin`

- `user` — чтобы индетифицировать текущего пользователя
- `account` — чтобы индетифицировать доступ к данным
- `admin` — чтобы индетифицировать админа

`user`

Никакие данные я обычно с id-шником пользователя не храню.

Все данные я вяжу к `account`

Юзеры это ненадежный материал

Например: какой-то менеджер регистрирует личный кабинет, через полгода увольняется; на его место приходит новый менеджер, и теперь везде надо перебивать старое мыло пользователя на новое.

`account`

Всё что создается —
статья, документ, заказ, заявка и пр.

Всё привязывается к аккаунту.

Аккаунт хранит в себе id пользователей.

Самый кайф в том, что:

- к одному аккаунту могут иметь доступ несколько пользователей
- а один пользователь может иметь доступ к нескольким аккаунтам (только дайте ему выпадайку, чтоб удобно было переключать аккаунты).

`admin`

Чтобы индетифицировать админа системы

И давать ему доступ в закрытую админскую часть

А еще фишка в том, что админ может получить токен `user` и `account` и ...

А еще фишка в том, что админ может получить токен `user` и `account` и ... используя стандартные интерфейсы зайти в любой личкаб и посмотреть глазами пользователя, что у него происходит

Три токена

`user` `account` `admin`

Берите и пользуйтесь на здоровье 😉

Section #4
Паттерны построения схем

В этой секции речь пойдет:

- как работать с ошибками в GraphQL
- i18n — интернационализация в GraphQL
- расширенное получение данных после мутации

Как работать с ошибками в GraphQL

В GraphQL я бы выделил 4 группы ошибок

- Фатальные ошибки
- Ошибки валидации
- Runtime ошибки
- Пользовательские ошибки

Когда клиент отправляет GraphQL-запрос по HTTP, то первым его встречает код вашего сервера.

1. Фатальные ошибки

- кончилась память
- забыли установить пакет
- грубая синтаксическая ошибка в коде
- ошибки сервера

В таком случае вы получаете

500 Internal Server Error

Ваш запрос даже не дошел до пакета `graphql`

Если фатальных ошибок нет,
то GraphQL-запрос передается в пакет `graphql`.

Пакет `graphql` возвращает ответ по спецификации

```
{
  // для возврата данных
  data: {},

  // для возврата ошибок, массив между прочим 😳
  errors: [...],

  // объект для пользовательских данных, сюда пихайте что хотите
  extensions: {}
}
```

GraphQL возвращает массив ошибок,
ведь в одном запросе может быть много ошибок 😈

GraphQL-ответ через HTTP всегда возвращает `код 200` ☝️

- GraphQL может вернуть много разных ошибок
- GraphQL не привязан к HTTP-протоколу

Забудьте про пачку HTTP-кодов, если вы пришли из мира RESTfull API

В мире GraphQL с HTTP всего один код `200`.

Ну и код `500`, если сервер помер.

Что происходит когда прилетает GraphQL-запрос на сервер?

Первым делом пакет `graphql`, проводит парсинг и валидацию GraphQL-запроса.

2. Ошибки валидации

- ошибка невалидного GraphQL-запроса
- не передали переменную
- проверяет на соответствие с вашей GraphQL-схемой
  - запросили несуществующее поле
  - не передали обязательный аргумент
  - передали неверный тип для аргумента

Ошибка при запросе несуществующего поля

```
{
  errors: [
    {
      message: 'Cannot query field "wrong" on type "Query".',
      locations: [{ line: 3, column: 11 }],
    },
  ],
}
```

При этом сервер возвращает код 200 и это нормально.

Забыли передать переменную

```
{
  errors: [
    {
      message: 'Variable "$q" of required type "String!" was not provided.',
      locations: [{ line: 2, column: 16 }],
    },
  ],
}
```

Если нет ошибок парсинга и валидации, то пакет `graphql` начинает выполнять запрос вызывая resolve-методы.

3. Runtime ошибки в resolve-методах

- throw new Error("")
- undefined is not a function (юзайте Flowtype или TypeScript)
- ошибка невалидного значения в return

Если возникла ошибка, то

- обработка ветки графа приостанавливается (вложенные resolve-методы вызываться не будут)
- на месте элемента, где произошла ошибка возвращается `null`
- ошибка добавляется в массив `errors`

☝ **НО при этом соседние ветки продолжают работать**

Пример: throw new Error("")

```
const searchResolver = (_, args) => {
  if (!args.q) throw new Error('missing q');
  return { text: args.q };
};
```

```
query {
  s1: search(q: "ok") { text }
  s2: search { text }
  s3: search(q: "good") { text }
}
```

```
{
  errors: [{
    message: 'missing q',
    locations: [{ line: 4, column: 11 }],
    path: ['s2']
  }],
  data: {
    s1: { text: 'ok' },
    s2: null,
    s3: { text: 'good' }
  },
}
```

Пример: throw new Error("")

```
const searchResolver = (_, args) => {
  if (!args.q) throw new Error('missing q');
  return { text: args.q };
};
```

```
query {
  s1: search(q: "ok") { text }
  s2: search { text }
  s3: search(q: "good") { text }
}
```

```
{
  errors: [{
    message: 'missing q',
    locations: [{ line: 4, column: 11 }],
    path: ['s2']
  }],
  data: {
    s1: { text: 'ok' },
    s2: null,
    s3: { text: 'good' }
  },
}
```

Пример: throw new Error("")

```
const searchResolver = (_, args) => {
  if (!args.q) throw new Error('missing q');
  return { text: args.q };
};
```

```
query {
  s1: search(q: "ok") { text }
  s2: search { text }
  s3: search(q: "good") { text }
}
```

```
{
  errors: [{
    message: 'missing q',
    locations: [{ line: 4, column: 11 }],
    path: ['s2']
  }],
  data: {
    s1: { text: 'ok' },
    s2: null,
    s3: { text: 'good' }
  },
}
```

Пример: throw new Error("")

```
const searchResolver = (_, args) => {
  if (!args.q) throw new Error('missing q');
  return { text: args.q };
};
```

```
query {
  s1: search(q: "ok") { text }
  s2: search { text }
  s3: search(q: "good") { text }
}
```

```
{
  errors: [{
    message: 'missing q',
    locations: [{ line: 4, column: 11 }],
    path: ['s2']
  }],
  data: {
    s1: { text: 'ok' },
    s2: null,
    s3: { text: 'good' }
  },
}
```

Пример: ошибка невалидного значения в return

```
const ooops = {
  type: new GraphQLList(GraphQLString),
  resolve: () => ['ok', { hey: 666 }],
};
```

```
{
  errors: [
    {
      message: 'String cannot represent value: { hey: 666 }',
      locations: [{ line: 3, column: 11 }],
      path: ['ooops', 1],
    },
  ],
  data: {
    ooops: [
      'ok',
      null
    ]
  },
}
```

Пример: ошибка невалидного значения в return

```
const ooops = {
  type: new GraphQLList(GraphQLString),
  resolve: () => ['ok', { hey: 666 }],
};
```

```
{
  errors: [
    {
      message: 'String cannot represent value: { hey: 666 }',
      locations: [{ line: 3, column: 11 }],
      path: ['ooops', 1],
    },
  ],
  data: {
    ooops: [
      'ok',
      null
    ]
  },
}
```

Пример: ошибка невалидного значения в return

```
const ooops = {
  type: new GraphQLList(GraphQLString),
  resolve: () => ['ok', { hey: 666 }],
};
```

```
{
  errors: [
    {
      message: 'String cannot represent value: { hey: 666 }',
      locations: [{ line: 3, column: 11 }],
      path: ['ooops', 1],
    },
  ],
  data: {
    ooops: [
      'ok',
      null
    ]
  },
}
```

Пример: ошибка невалидного значения в return

```
const ooops = {
  type: new GraphQLList(GraphQLString),
  resolve: () => ['ok', { hey: 666 }],
};
```

```
{
  errors: [
    {
      message: 'String cannot represent value: { hey: 666 }',
      locations: [{ line: 3, column: 11 }],
      path: ['ooops', 1],
    },
  ],
  data: {
    ooops: [
      'ok',
      null
    ]
  },
}
```

Продолжим пример и сделаем так:

```
const ooops = {
-  type: new GraphQLList(GraphQLString),
+  type: new GraphQLList(new GraphQLNonNull(GraphQLString)),
  resolve: () => ['ok', { hey: 666 }],
};
```

GraphQL чуть-чуть зверeет, т.к. теперь не может вернуть массив с null значением

```
{
  errors: [
    {
      message: 'String cannot represent value: { hey: 666 }',
      locations: [{ line: 3, column: 11 }],
      path: ['ooops', 1],
    },
  ],
  data: {
-    ooops: [ 'ok', null ]
+    ooops: null
  },
}
```

А еще есть `extensions` в ошибках, чтобы передать клиентам дополнительные данные об ошибке

```
const searchResolver = () => {
  const e = new Error('WoW');
  e.extensions = { a: 'Additional error data for client' };
  throw e;
};
```

```
{
  errors: [
    {
      message: 'WoW',
      locations: [{ line: 1, column: 9 }],
      path: ['search'],
      extensions: { a: 'Additional error data for client' }, // <-- ☝
    },
  ],
  data: { search: null },
  extensions: {} // <-- это экстеншн на глобальном уровне
}
```

А еще есть `extensions` в ошибках, чтобы передать клиентам дополнительные данные об ошибке

```
const searchResolver = () => {
  const e = new Error('WoW');
  e.extensions = { a: 'Additional error data for client' };
  throw e;
};
```

```
{
  errors: [
    {
      message: 'WoW',
      locations: [{ line: 1, column: 9 }],
      path: ['search'],
      extensions: { a: 'Additional error data for client' }, // <-- ☝
    },
  ],
  data: { search: null },
  extensions: {} // <-- это экстеншн на глобальном уровне
}
```

А еще есть `extensions` в ошибках, чтобы передать клиентам дополнительные данные об ошибке

```
const searchResolver = () => {
  const e = new Error('WoW');
  e.extensions = { a: 'Additional error data for client' };
  throw e;
};
```

```
{
  errors: [
    {
      message: 'WoW',
      locations: [{ line: 1, column: 9 }],
      path: ['search'],
      extensions: { a: 'Additional error data for client' }, // <-- ☝
    },
  ],
  data: { search: null },
  extensions: {} // <-- это экстеншн на глобальном уровне
}
```

А еще есть `extensions` в ошибках, чтобы передать клиентам дополнительные данные об ошибке

```
const searchResolver = () => {
  const e = new Error('WoW');
  e.extensions = { a: 'Additional error data for client' };
  throw e;
};
```

```
{
  errors: [
    {
      message: 'WoW',
      locations: [{ line: 1, column: 9 }],
      path: ['search'],
      extensions: { a: 'Additional error data for client' }, // <-- ☝
    },
  ],
  data: { search: null },
  extensions: {} // <-- это экстеншн на глобальном уровне
}
```

Как работать с такими ошибками на клиентской стороне?

Как работать с такими ошибками на клиентской стороне?

Показать одну ошибку в модалке не проблема!

А если ошибок две?

А если ошибок две?

А если ошибки надо показывать в разных частях приложения?

А если ошибок две?

А если ошибки надо показывать в разных частях приложения?

Как разобрать массив ошибок и нужные передать по дереву компонент?

Куралесим велосипед на клиенте?

Куралесим велосипед на клиенте?

😈

✋ Не спешим!

💃 Есть красивое решение!

Помните формат GraphQL-ответа?

```
{
  // для возврата данных
  data: {},

  // для возврата массива ошибок
  errors: [...],

  // объект для пользовательских данных, сюда пихайте что хотите
  extensions: {}
}
```

Помните формат GraphQL-ответа?

```
{
  // для возврата данных
  data: {},

  // для возврата массива ошибок
  errors: [...],

  // объект для пользовательских данных, сюда пихайте что хотите
  extensions: {}
}
```

А что если передавать ошибки не только в `errors`,
но и в `data` 🤔

Передавать в `data` не все ошибки подряд, а только...

4. Пользовательские ошибки

- недостаточно прав для просмотра
- недоступно в вашем регионе
- пользователь забанен
- необходимо оплатить контент
- и т.п.

У пользовательских ошибок есть одно свойство — их надо показать пользователю, чтоб он дальше что-то сделал.

Для передачи ошибок в `data` надо просто создать Типы ошибок:

```
type VideoInProgressProblem {
  estimatedTime: Int
}

type VideoNeedBuyProblem {
  price: Int
}

type VideoApproveAgeProblem {
  minAge: Int
}
```

И клиенту возвращать

- либо запись с данными `Video`
- либо запись с пользовательской ошибкой `Video*Problem`

Для этого необходимо использовать Union-тип

```
const VideoResultType = new GraphQLUnionType({
  // Даем имя типу.
  // Здорово если если вы выработаете конвенцию в своей команде
  // и к таким Union-типам будите добавлять суффикс Result
  name: 'VideoResult',

  // как хорошие бекендеры добавляем какое-нибудь описание
  description: 'List of video or problems',

  // объявляем типы через массив, которые могут быть возвращены
  types: () => [
    VideoType,
    VideoInProgressProblemType,
    VideoNeedBuyProblemType,
    VideoApproveAgeProblemType,
```

Для этого необходимо использовать Union-тип

```
const VideoResultType = new GraphQLUnionType({
  // Даем имя типу.
  // Здорово если если вы выработаете конвенцию в своей команде
  // и к таким Union-типам будите добавлять суффикс Result
  name: 'VideoResult',

  // как хорошие бекендеры добавляем какое-нибудь описание
  description: 'List of video or problems',

  // объявляем типы через массив, которые могут быть возвращены
  types: () => [
    VideoType,
    VideoInProgressProblemType,
    VideoNeedBuyProblemType,
    VideoApproveAgeProblemType,
```

Для этого необходимо использовать Union-тип

```
const VideoResultType = new GraphQLUnionType({
  // Даем имя типу.
  // Здорово если если вы выработаете конвенцию в своей команде
  // и к таким Union-типам будите добавлять суффикс Result
  name: 'VideoResult',

  // как хорошие бекендеры добавляем какое-нибудь описание
  description: 'List of video or problems',

  // объявляем типы через массив, которые могут быть возвращены
  types: () => [
    VideoType,
    VideoInProgressProblemType,
    VideoNeedBuyProblemType,
    VideoApproveAgeProblemType,
```

Для этого необходимо использовать Union-тип

```
const VideoResultType = new GraphQLUnionType({
  // Даем имя типу.
  // Здорово если если вы выработаете конвенцию в своей команде
  // и к таким Union-типам будите добавлять суффикс Result
  name: 'VideoResult',

  // как хорошие бекендеры добавляем какое-нибудь описание
  description: 'List of video or problems',

  // объявляем типы через массив, которые могут быть возвращены
  types: () => [
    VideoType,
    VideoInProgressProblemType,
    VideoNeedBuyProblemType,
    VideoApproveAgeProblemType,
```

Ну а дальше возвращать либо запись, либо проблему (ошибку)

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      list: {
        type: new GraphQLList(VideoResultType),
        resolve: () => {
          return [
            new Video({ title: 'DOM2 in the HELL', url: 'https://url' }),
            new VideoApproveAgeProblem({ minAge: 21 }),
            new VideoNeedBuyProblem({ price: 10 }),
            new VideoInProgressProblem({ estimatedTime: 220 }),
          ];
        },
      },
```

Ну а дальше возвращать либо запись, либо проблему (ошибку)

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      list: {
        type: new GraphQLList(VideoResultType),
        resolve: () => {
          return [
            new Video({ title: 'DOM2 in the HELL', url: 'https://url' }),
            new VideoApproveAgeProblem({ minAge: 21 }),
            new VideoNeedBuyProblem({ price: 10 }),
            new VideoInProgressProblem({ estimatedTime: 220 }),
          ];
        },
      },
```

Ну а дальше возвращать либо запись, либо проблему (ошибку)

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      list: {
        type: new GraphQLList(VideoResultType),
        resolve: () => {
          return [
            new Video({ title: 'DOM2 in the HELL', url: 'https://url' }),
            new VideoApproveAgeProblem({ minAge: 21 }),
            new VideoNeedBuyProblem({ price: 10 }),
            new VideoInProgressProblem({ estimatedTime: 220 }),
          ];
        },
      },
```

Тогда фронтендеры смогут писать такие запросы:

```
query {
  list {
    ...on Video {
      title
      url
    }
    ...on VideoInProgressProblem {
      estimatedTime
    }
    ...on VideoNeedBuyProblem {
      price
    }
    ...on VideoApproveAgeProblem {
      minAge
    }
```

Тогда фронтендеры смогут писать такие запросы:

```
query {
  list {
    ...on Video {
      title
      url
    }
    ...on VideoInProgressProblem {
      estimatedTime
    }
    ...on VideoNeedBuyProblem {
      price
    }
    ...on VideoApproveAgeProblem {
      minAge
    }
```

Тогда фронтендеры смогут писать такие запросы:

```
query {
  list {
    ...on Video {
      title
      url
    }
    ...on VideoInProgressProblem {
      estimatedTime
    }
    ...on VideoNeedBuyProblem {
      price
    }
    ...on VideoApproveAgeProblem {
      minAge
    }
```

Тогда фронтендеры смогут писать такие запросы:

```
query {
  list {
    ...on Video {
      title
      url
    }
    ...on VideoInProgressProblem {
      estimatedTime
    }
    ...on VideoNeedBuyProblem {
      price
    }
    ...on VideoApproveAgeProblem {
      minAge
    }
```

Тогда фронтендеры смогут писать такие запросы:

```
query {
  list {
    ...on Video {
      title
      url
    }
    ...on VideoInProgressProblem {
      estimatedTime
    }
    ...on VideoNeedBuyProblem {
      price
    }
    ...on VideoApproveAgeProblem {
      minAge
    }
```

Ответ от сервера будет таким

```
{
  data: {
    list: [
      { __typename: 'Video', title: 'DOM2 in the HELL', url: 'https://url' },
      { __typename: 'VideoApproveAgeProblem', minAge: 21 },
      { __typename: 'VideoNeedBuyProblem', price: 10 },
      { __typename: 'VideoInProgressProblem', estimatedTime: 220 },
    ],
  },
}
```

В зависимости от `__typename` можно рендерить ту или иную компоненту.

Профит от таких пользовательских ошибок:

Профит от таких пользовательских ошибок:

- фронтендеры точно знают какие ошибки могут быть

Профит от таких пользовательских ошибок:

- фронтендеры точно знают какие ошибки могут быть
- расписаны поля ошибок (статический анализ)

Профит от таких пользовательских ошибок:

- фронтендеры точно знают какие ошибки могут быть
- расписаны поля ошибок (статический анализ)
- получать ошибки сразу на нужном уровне

Профит от таких пользовательских ошибок:

- фронтендеры точно знают какие ошибки могут быть
- расписаны поля ошибок (статический анализ)
- получать ошибки сразу на нужном уровне
- легко понять какая именно ошибка вернулась

Профит от таких пользовательских ошибок:

- фронтендеры точно знают какие ошибки могут быть
- расписаны поля ошибок (статический анализ)
- получать ошибки сразу на нужном уровне
- легко понять какая именно ошибка вернулась
- в результате чище, проще и безопаснее код

Подробнее про работу с ошибками в GraphQL читаем тут

i18n — интернационализация в GraphQL

Для API очень важно удобно предоставлять данные доступные на разных языках.

— Капитан Очевидность

Как можно указать необходимый язык

Как можно указать необходимый язык

- через HTTP-заголовки

Как можно указать необходимый язык

- через HTTP-заголовки
- через cookie или из профиля пользователя

Как можно указать необходимый язык

- через HTTP-заголовки
- через cookie или из профиля пользователя
- через аргумент корневого элемента

Как можно указать необходимый язык

- через HTTP-заголовки
- через cookie или из профиля пользователя
- через аргумент корневого элемента
- через аргумент поля

Как можно указать необходимый язык

- через HTTP-заголовки
- через cookie или из профиля пользователя
- через аргумент корневого элемента
- через аргумент поля
- через разные поля с суффиксом (лютейшая какафония)

i18n через HTTP-заголовки

```
Accept-Language: en-US,en;q=0.9,ru-RU;q=0.8,ru;q=0.7,kk;q=0.6
```

- отдается браузерами (обычно адекват)
- для мобильных приложений тоже вариант

Прелесть этого подхода в том, что никак не нужно модифицировать GraphQL-запрос.

Просто меняете http-заголовок и на тот же запрос получаете данные на нужном языке.

У фронтендера радость:

- язык задается на уровне NetworkLayer'a
  - тот кто юзает Relay и Apollo поймут меня
- на уровне компонентов ничего указывать не нужно
  - должны понять все

Для бэкендера раз плюнуть:

- пробросить `req` объект http-запроса в `context` GraphQL-сервера
- в методах `resolve(source, args, context, info)` считываем третий аргумент `context` и оттуда получаем язык

i18n через cookie

Это шайба в шайбу такой же метод, как описано выше про заголовки `Accept-Language`.

Только вы считываете не из заголовков а из cookie и пишете опять таки в `context`.

i18n из профиля клиента

Из кук, токена или еще как-то получаете id пользователя, подтягиваете его региональные настройки из БД и записываете их в `context`

i18n через аргумент корневого элемента

На верхнем уровне GraphQL-запроса фронтендер передает язык:

```
query {
  viewer(lang: "ru") {
    article(id: 10) { ... }
    someOtherData
  }
}
```

Как в таком случае пробросить язык до resolver'ов `article`, `someOtherData` и глубже?

```
query {
  viewer(lang: "ru") {
    article(id: 10) { ... }
    someOtherData
  }
}
```

Как в таком случае пробросить язык до resolver'ов `article`, `someOtherData` и глубже?

```
query {
  viewer(lang: "ru") {
    article(id: 10) { ... }
    someOtherData
  }
}
```

Правильно, опять нагадить в `context`. Кто сказал, что в `resolve`-методе нельзя что-то записывать в контекст?!

В коде это выглядит так

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: {
      type: ViewerType,
      args: {
        lang: {
          type: GraphQLString, // ну либо Enum, кому как удобнее
        },
      },
      resolve: (_, args, context) => {
        // раз и записали язык из аргумента в контекст
        context.lang = args.lang || 'ru';

        // LIFEHACK: возвращаем пустой объект
```

В коде это выглядит так

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: {
      type: ViewerType,
      args: {
        lang: {
          type: GraphQLString, // ну либо Enum, кому как удобнее
        },
      },
      resolve: (_, args, context) => {
        // раз и записали язык из аргумента в контекст
        context.lang = args.lang || 'ru';

        // LIFEHACK: возвращаем пустой объект
```

В коде это выглядит так

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: {
      type: ViewerType,
      args: {
        lang: {
          type: GraphQLString, // ну либо Enum, кому как удобнее
        },
      },
      resolve: (_, args, context) => {
        // раз и записали язык из аргумента в контекст
        context.lang = args.lang || 'ru';

        // LIFEHACK: возвращаем пустой объект
```

В коде это выглядит так

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: {
      type: ViewerType,
      args: {
        lang: {
          type: GraphQLString, // ну либо Enum, кому как удобнее
        },
      },
      resolve: (_, args, context) => {
        // раз и записали язык из аргумента в контекст
        context.lang = args.lang || 'ru';

        // LIFEHACK: возвращаем пустой объект
```

У фронтендеров опять-таки кайф

Не надо засорять компоненты на нижнем уровне знанием выбранного языка.

i18n через аргумент поля

В запросе передаем язык много раз

```
query {
  article(lang: "ru") {
    title
    reviews(lang: "ru", limit: 10) {
      text
    }
  }
  message {
    text(lang: "ru")
  }
}
```

В запросе передаем язык много раз

```
query {
  article(lang: "ru") {
    title
    reviews(lang: "ru", limit: 10) {
      text
    }
  }
  message {
    text(lang: "ru")
  }
}
```

Продвинутые фронтендеры заюзают GraphQL-переменную

```
query ($lang: String) {
  article(lang: $lang) {
    title
    reviews(lang: $lang, limit: 10) {
      text
    }
  }
  message {
    text(lang: $lang)
  }
}
```

Продвинутые фронтендеры заюзают GraphQL-переменную

```
query ($lang: String) {
  article(lang: $lang) {
    title
    reviews(lang: $lang, limit: 10) {
      text
    }
  }
  message {
    text(lang: $lang)
  }
}
```

Продвинутые фронтендеры заюзают GraphQL-переменную

```
query ($lang: String) {
  article(lang: $lang) {
    title
    reviews(lang: $lang, limit: 10) {
      text
    }
  }
  message {
    text(lang: $lang)
  }
}
```

Передавать язык через аргумент полей не самый приятный и удобный способ для фронтендеров!

Но иногда без этого метода никак!

```
query {
  article(id: 10) {
    textRU: text(lang: "ru")
    textEN: text(lang: "en")
    textES: text(lang: "es")
  }
}
```

Юзаются алиасы, чтоб фронтендеры смогли выпилить админку для управления переводами.

i18n через разные поля с суффиксом (лютейшая какафония)

Подход через поля с суффиксом для бэкендра выглядит достаточно компактно и красиво:

```
query {
  article(id: 10) {
    textRU
    textEN
    textES
  }
}
```

Но у фронтендера просто ад с таким подходом на фронте.

Но у фронтендера просто ад с таким подходом на фронте.

- при добавлении нового языка ему надо куралесить новые GraphQL-запросы

Но у фронтендера просто ад с таким подходом на фронте.

- при добавлении нового языка ему надо куралесить новые GraphQL-запросы
- однозначно допиливать компоненты

Но у фронтендера просто ад с таким подходом на фронте.

- при добавлении нового языка ему надо куралесить новые GraphQL-запросы
- однозначно допиливать компоненты
- может убить всю оптимизацию для прекомпилированных запросов в Relay (надо будет в коде пачку таких запросов хранить, на каждый случай языка)

Но у фронтендера просто ад с таким подходом на фронте.

- при добавлении нового языка ему надо куралесить новые GraphQL-запросы
- однозначно допиливать компоненты
- может убить всю оптимизацию для прекомпилированных запросов в Relay (надо будет в коде пачку таких запросов хранить, на каждый случай языка)
- тупо много кода с душком

Если фронту приспичит, он возьмет предыдущий подход с алиасами:

```
query {
  article(id: 10) {
    textRU: text(lang: "ru")
    textEN: text(lang: "en")
    textES: text(lang: "es")
  }
}
```

Резюме. Какой подход выбрать?

Берите сразу все, кроме последней какафонии.

На уровне сервера пишите в `context.lang`

На уровне сервера пишите в `context.lang`

- сперва из http-заголовка Accept-Language

На уровне сервера пишите в `context.lang`

- сперва из http-заголовка Accept-Language
- перезаписывайте, если есть язык в куках

На уровне сервера пишите в `context.lang`

- сперва из http-заголовка Accept-Language
- перезаписывайте, если есть язык в куках
- опять перезаписывайте, если есть профиль пользователя

На уровне сервера пишите в `context.lang`

- сперва из http-заголовка Accept-Language
- перезаписывайте, если есть язык в куках
- опять перезаписывайте, если есть профиль пользователя
- и еще раз перезапишите, если фронтендер указал язык через корневое поле в запросе

Затем пишем функцию помогайку для resolve-методов:

```
function getLang(args, context) {
  if (args.lang) return args.lang;
  else if (context.lang) return context.lang;
  else return 'ru';
}
```

И таскайте её по всем полям, где есть i18n.

Затем пишем функцию помогайку для resolve-методов:

```
function getLang(args, context) {
  if (args.lang) return args.lang;
  else if (context.lang) return context.lang;
  else return 'ru';
}
```

И таскайте её по всем полям, где есть i18n.

Затем пишем функцию помогайку для resolve-методов:

```
function getLang(args, context) {
  if (args.lang) return args.lang;
  else if (context.lang) return context.lang;
  else return 'ru';
}
```

И таскайте её по всем полям, где есть i18n.

Затем пишем функцию помогайку для resolve-методов:

```
function getLang(args, context) {
  if (args.lang) return args.lang;
  else if (context.lang) return context.lang;
  else return 'ru';
}
```

И таскайте её по всем полям, где есть i18n.

Бэкендер!

В твоих силах спасти фронтендера от каки в его коде!

Что сказал Капитан Очевидность в самом начале?

Что сказал Капитан Очевидность в самом начале?

Для API

Что сказал Капитан Очевидность в самом начале?

Для API

очень важно

Что сказал Капитан Очевидность в самом начале?

Для API

очень важно

удобно

Что сказал Капитан Очевидность в самом начале?

Для API

очень важно

удобно

предоставлять данные

Что сказал Капитан Очевидность в самом начале?

Для API

очень важно

удобно

предоставлять данные

доступные на разных языках.

Подробнее про интернационализация в GraphQL читайте тут

Section #5
Производительность

DataLoader

(avoiding N+1 problem)

DataLoader — это утилита сокращающая кол-во обращений в базу через batching запросов.

Пример: N+1 problem

запрашиваем список статей с именем автора

```
{
  articles {
    title
    author {
      name
    }
  }
}
```

Стандартная ситуация в мире GraphQL

для списка из 15 статей сделать 16 запросов в базу 😬

```
Run Article query: findMany()
Run Author query: findById(1)
Run Author query: findById(7)
Run Author query: findById(6)
Run Author query: findById(3)
Run Author query: findById(4)
Run Author query: findById(5)
Run Author query: findById(6)
Run Author query: findById(7)
Run Author query: findById(3)
Run Author query: findById(2)
Run Author query: findById(5)
Run Author query: findById(4)
Run Author query: findById(2)
Run Author query: findById(1)
```

Стандартная ситуация в мире GraphQL

для списка из 15 статей сделать 16 запросов в базу 😬

```
Run Article query: findMany()
Run Author query: findById(1)
Run Author query: findById(7)
Run Author query: findById(6)
Run Author query: findById(3)
Run Author query: findById(4)
Run Author query: findById(5)
Run Author query: findById(6)
Run Author query: findById(7)
Run Author query: findById(3)
Run Author query: findById(2)
Run Author query: findById(5)
Run Author query: findById(4)
Run Author query: findById(2)
Run Author query: findById(1)
```

Стандартная ситуация в мире GraphQL

для списка из 15 статей сделать 16 запросов в базу 😬

```
Run Article query: findMany()
Run Author query: findById(1)
Run Author query: findById(7)
Run Author query: findById(6)
Run Author query: findById(3)
Run Author query: findById(4)
Run Author query: findById(5)
Run Author query: findById(6)
Run Author query: findById(7)
Run Author query: findById(3)
Run Author query: findById(2)
Run Author query: findById(5)
Run Author query: findById(4)
Run Author query: findById(2)
Run Author query: findById(1)
```

Кто-то скажет:

— Фууу, какая отвратительная производительность!

— GraphQL лажа полная!

Но я скажу:

— На самом деле,
в GraphQL скормили
лажовый код
в resolve-методах

Что происходит под капотом:

```
{
  articles {
    title
    author {
      name
    }
  }
}
```

Что происходит под капотом:

```
{
  articles {
    title
    author {
      name
    }
  }
}
```

`Query.articles.resolve()` — 1 запрос для получения массива из 15 статей

Что происходит под капотом:

```
{
  articles {
    title
    author {
      name
    }
  }
}
```

`Query.articles.resolve()` — 1 запрос для получения массива из 15 статей

`Article.author.resolve()` — 15 запросов, чтоб для каждой статьи получить автора по Id.

Не нужно сразу тянуть автора по id!

Не нужно сразу тянуть автора по id!

Сперва собираем все айдишники авторов

Не нужно сразу тянуть автора по id!

Сперва собираем все айдишники авторов

Потом одним запросом тянем из БД

Вот так должно работать у хороших ребят:

```
1 Run Article query: findMany()
2 Run Author query: findByIds(1, 7, 6, 3, 4, 5, 2)
```

2 запроса вместо 16-ти

Для этой задачи нам и нужен `DataLoader`

DataLoader — это batcher и cacher в одном флаконе.

Механика получения данных через DataLoader

1. Инициализируем DataLoader

```
const authorDataLoader = new DataLoader(async batchLoad(ids) => { ... });
```

2. Запрашиваем Авторов по id где нужно

```
const authorPromise1 = authorDataLoader.load(1);
const authorPromise2 = authorDataLoader.load(2);
const authorPromise1_copy = authorDataLoader.load(1); // <--- CACHE detected 👍
// authorPromise1 === authorPromise1_copy
const authorPromise4 = authorDataLoader.load(4);
```

3. Автоматом на `nextTick()` выполнится batch-запрос, после чего все промисы из п.2 разрезолвятся.

Нюанс объявления `batchLoad` в DataLoader

```
// Создаем объект DataLoader и сразу в конструктор передаем
// функцию batch-загрузки по ids
const authorDataLoader = new DataLoader(
  async batchLoad(ids) => {
    // получили массив айдишников, дергаем записи одним запросом из базы
    const rows = await authorModel.findByIds(ids);
    // ВАЖНО: полученные записи мы ДОЛЖНЫ вернуть в том порядке
    //        в котором получили ids на входе
    // Если запись по id не будет найдена, то вернется undefined
    const sortedInIdsOrder = ids.map(id => rows.find(x => x.id === id));
    return sortedInIdsOrder;
  }
);
```

Иначе получим бардак в данных.

Нюанс объявления `batchLoad` в DataLoader

```
// Создаем объект DataLoader и сразу в конструктор передаем
// функцию batch-загрузки по ids
const authorDataLoader = new DataLoader(
  async batchLoad(ids) => {
    // получили массив айдишников, дергаем записи одним запросом из базы
    const rows = await authorModel.findByIds(ids);
    // ВАЖНО: полученные записи мы ДОЛЖНЫ вернуть в том порядке
    //        в котором получили ids на входе
    // Если запись по id не будет найдена, то вернется undefined
    const sortedInIdsOrder = ids.map(id => rows.find(x => x.id === id));
    return sortedInIdsOrder;
  }
);
```

Иначе получим бардак в данных.

Нюанс объявления `batchLoad` в DataLoader

```
// Создаем объект DataLoader и сразу в конструктор передаем
// функцию batch-загрузки по ids
const authorDataLoader = new DataLoader(
  async batchLoad(ids) => {
    // получили массив айдишников, дергаем записи одним запросом из базы
    const rows = await authorModel.findByIds(ids);
    // ВАЖНО: полученные записи мы ДОЛЖНЫ вернуть в том порядке
    //        в котором получили ids на входе
    // Если запись по id не будет найдена, то вернется undefined
    const sortedInIdsOrder = ids.map(id => rows.find(x => x.id === id));
    return sortedInIdsOrder;
  }
);
```

Иначе получим бардак в данных.

Нюанс объявления `batchLoad` в DataLoader

```
// Создаем объект DataLoader и сразу в конструктор передаем
// функцию batch-загрузки по ids
const authorDataLoader = new DataLoader(
  async batchLoad(ids) => {
    // получили массив айдишников, дергаем записи одним запросом из базы
    const rows = await authorModel.findByIds(ids);
    // ВАЖНО: полученные записи мы ДОЛЖНЫ вернуть в том порядке
    //        в котором получили ids на входе
    // Если запись по id не будет найдена, то вернется undefined
    const sortedInIdsOrder = ids.map(id => rows.find(x => x.id === id));
    return sortedInIdsOrder;
  }
);
```

Иначе получим бардак в данных.

Нюанс объявления `batchLoad` в DataLoader

```
// Создаем объект DataLoader и сразу в конструктор передаем
// функцию batch-загрузки по ids
const authorDataLoader = new DataLoader(
  async batchLoad(ids) => {
    // получили массив айдишников, дергаем записи одним запросом из базы
    const rows = await authorModel.findByIds(ids);
    // ВАЖНО: полученные записи мы ДОЛЖНЫ вернуть в том порядке
    //        в котором получили ids на входе
    // Если запись по id не будет найдена, то вернется undefined
    const sortedInIdsOrder = ids.map(id => rows.find(x => x.id === id));
    return sortedInIdsOrder;
  }
);
```

Иначе получим бардак в данных.

Вроде все просто,

пока не начинают женить DataLoader с GraphQL

Я думал, что всё просто с GraphQL + DataLoader:

- пока в чате по GraphQL не прочитал кучу негатива
- пока заново не перечитал документашку DataLoader'a

Это фиаско, котаны!

- нет толкового примера
- изобилие текста и методов уводит народ непонятно куда
  - воспринимается как замена для Redis'a
- сложно уловить что он должен быть в `context`е GraphQL
- **начинает криво использоваться**

Правило 1: DataLoader должен использоваться в рамках одного запроса.

- 👌 При получение запроса создайте DataLoader в `context`
  - Выполните запрос
  - Затем смело удаляйте контекст с даталоадером
- 👌 Для новых запросов вы создаете новые DataLoader'ы!

Правило 2: Для каждого резолвера свой DataLoader.

- 🤮 общий (глобальный) дата-лоадер ведет
  - к грязному коду, т.к. вы на уровне сервера объявляете batch-функцию
  - должен получить из БД полную запись, т.к. разные резолверы могут запросить разные поля
- 👌 DataLoader необходимо объявлять внутри resolve-метода
  - тут вы знаете какие поля нужны юзеру, их и тянем из БД

Ну что, пришло время примера

На уровне сервера в контексте создаем `WeakMap` для будущих дата-лоадеров

```
const server = new ApolloServer({
  schema,
  context: ({ req }) => ({
    req,
    dataloaders: new WeakMap(),
  }),
});
```

На уровне сервера в контексте создаем `WeakMap` для будущих дата-лоадеров

```
const server = new ApolloServer({
  schema,
  context: ({ req }) => ({
    req,
    dataloaders: new WeakMap(),
  }),
});
```

На уровне сервера в контексте создаем `WeakMap` для будущих дата-лоадеров

```
const server = new ApolloServer({
  schema,
  context: ({ req }) => ({
    req,
    dataloaders: new WeakMap(),
  }),
});
```

Никакого убогого кода с моделями на уровне сервера ☝️

На уровне сервера в контексте создаем `WeakMap` для будущих дата-лоадеров

```
const server = new ApolloServer({
  schema,
  context: ({ req }) => ({
    req,
    dataloaders: new WeakMap(),
  }),
});
```

Никакого убогого кода с моделями на уровне сервера ☝

Это позволит находить существующие дата-лоадары, либо ложить туда новый инстанс дата-лоадера для повторного использования. На уровне резолвера! ☝

На уровне resolve-метода

```
const ArticleType = new GraphQLObjectType({
  name: 'Article',
  fields: () => ({
    title: { type: GraphQLString },
    authorId: { type: GraphQLString },
    author: {
      type: AuthorType,
      resolve: (source, args, context, info) => {
        // context.dataloaders был создан на уровне сервера
        const { dataloaders } = context;

        // единожы инициализируем DataLoader для получения авторов по ids
        let dl = dataloaders.get(info.fieldNodes);
        if (!dl) {
          dl = new DataLoader(async (ids: any) => {
```

На уровне resolve-метода

```
const ArticleType = new GraphQLObjectType({
  name: 'Article',
  fields: () => ({
    title: { type: GraphQLString },
    authorId: { type: GraphQLString },
    author: {
      type: AuthorType,
      resolve: (source, args, context, info) => {
        // context.dataloaders был создан на уровне сервера
        const { dataloaders } = context;

        // единожы инициализируем DataLoader для получения авторов по ids
        let dl = dataloaders.get(info.fieldNodes);
        if (!dl) {
          dl = new DataLoader(async (ids: any) => {
```

На уровне resolve-метода

```
const ArticleType = new GraphQLObjectType({
  name: 'Article',
  fields: () => ({
    title: { type: GraphQLString },
    authorId: { type: GraphQLString },
    author: {
      type: AuthorType,
      resolve: (source, args, context, info) => {
        // context.dataloaders был создан на уровне сервера
        const { dataloaders } = context;

        // единожы инициализируем DataLoader для получения авторов по ids
        let dl = dataloaders.get(info.fieldNodes);
        if (!dl) {
          dl = new DataLoader(async (ids: any) => {
```

На уровне resolve-метода

```
const ArticleType = new GraphQLObjectType({
  name: 'Article',
  fields: () => ({
    title: { type: GraphQLString },
    authorId: { type: GraphQLString },
    author: {
      type: AuthorType,
      resolve: (source, args, context, info) => {
        // context.dataloaders был создан на уровне сервера
        const { dataloaders } = context;

        // единожы инициализируем DataLoader для получения авторов по ids
        let dl = dataloaders.get(info.fieldNodes);
        if (!dl) {
          dl = new DataLoader(async (ids: any) => {
```

На уровне resolve-метода

```
const ArticleType = new GraphQLObjectType({
  name: 'Article',
  fields: () => ({
    title: { type: GraphQLString },
    authorId: { type: GraphQLString },
    author: {
      type: AuthorType,
      resolve: (source, args, context, info) => {
        // context.dataloaders был создан на уровне сервера
        const { dataloaders } = context;

        // единожы инициализируем DataLoader для получения авторов по ids
        let dl = dataloaders.get(info.fieldNodes);
        if (!dl) {
          dl = new DataLoader(async (ids: any) => {
```

На уровне resolve-метода

```
const ArticleType = new GraphQLObjectType({
  name: 'Article',
  fields: () => ({
    title: { type: GraphQLString },
    authorId: { type: GraphQLString },
    author: {
      type: AuthorType,
      resolve: (source, args, context, info) => {
        // context.dataloaders был создан на уровне сервера
        const { dataloaders } = context;

        // единожы инициализируем DataLoader для получения авторов по ids
        let dl = dataloaders.get(info.fieldNodes);
        if (!dl) {
          dl = new DataLoader(async (ids: any) => {
```

Почему используется WeakMap и info.fieldNodes?

- `info.fieldNodes` — это объект, и он один и тот же для одинаковых путей запроса, eg. Query.articles.author.resolve()
- `WeakMap` — чтоб можно было использовать fieldNodes в качестве ключа для поиска уже созданных дата-лоадеров

Еще одна магия info.fieldNodes

```
{
  article(id: 5) {
    author { nickname } # DataLoader1
  }
  articles {
    title
    author { name email } # DataLoader2
  }
  lastComments {
    article {
      author  { name } # DataLoader3
    }
  }
}
```

По разным путям создаются разные дата-лоадеры.

Это позволяет создавать в одном и том же резолвере разные DataLoader в зависимости от GraphQL-запроса.

```
{
  article(id: 5) {
    author { nickname } # DataLoader1
  }
  articles {
    title
    author { name email } # DataLoader2
  }
  lastComments {
    article {
      author  { name } # DataLoader3
    }
  }
}
```

И каждый DataLoader может тянуть из базы, только те поля которые запросил пользователь.

В сухом остатке

DataLoader клевая утилита, если ее использовать разумно.

Если к ней относиться как к простому группировщику запросов.

В сухом остатке

- Не пытайтесь из DataLoader'а смастерить кэш
- Новые DataLoader'ы для каждого GraphQL-запроса
- Создаайте DataLoader в resolve-методе
- Ничего страшного если для одного резолвера у вас создасться несколько DataLoader'ов

Чистого кода и меньше бесполезных запросов в ваш ~~дом~~ бэкенд!

Подробнее про DataLoader с примерами можно почитать тут

Загрузка файлов в GraphQL

Слайдов пока нет

НО

Про загрузку файлов в GraphQL можно почитать тут

Section #6
БЕЗОПАСНОСТЬ

- прикручиваем QueryCost (Denial of Service attacks)
- готовим схему для PRODUCTION

Section #7
ГЕНЕРАЦИЯ СХЕМ

- генерация схем из моделей
- генерим две схемы для админов и для клиентов
- graphql-compose

Section #8
РАЗНОЕ

- тестирование схем
- документация
- версионирование
- циклические резолверы
- транзакции для нескольких мутаций
- фронетнедеры писающие кипятком

Послесловие

Как заставить бэкендеров запилить GraphQL API?

Как заставить бэкендеров запилить GraphQL API?

Никак!

Как заставить бэкендеров запилить GraphQL API?

Никак!

Себе дороже!

Как заставить бэкендеров запилить GraphQL API?

Никак!

Себе дороже!

Ещё понапилят вам всякого 💩💩💩

Надо искать фулстекера — фронтендера и бэкендера в одном флаконе, который будет вести разработку GraphQL API.

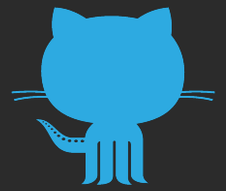
Надо искать фулстекера — фронтендера и бэкендера в одном флаконе, который будет вести разработку GraphQL API.

А он уже подтянет каких надо бэкендеров в свою команду и заставит сделать как надо.

Поставьте лайк за доклад! 👍

Спасибо за ваше внимание!

Вопросы к Паше?

 nodkz 

Ссылка на эту презентацию: http://bit.ly/holy-graphql