

## Схема GraphQL (SDL)

Схема GraphQL, которая находится на каждом сервере этого типа, представляет собой своего рода "инструкцию", которая позволяет клиентам понять, какие операции сервер может выполнить и какие типы данных он может обрабатывать. Это своего рода дорожная карта, определяющая, какие типы данных можно запросить и как их можно запросить.

Давайте приведем пример: предположим, у вас есть схема GraphQL, которая определяет тип данных "User" и включает поля для имени и адреса электронной почты - "name" и "email". Если клиент попытается запросить номер телефона пользователя, допустим, "phone", то сервер отклонит этот запрос, потому что в схеме нет определения для номера телефона. Таким же образом, если клиент ошибочно напишет поле "name" как "nmae", сервер также отклонит запрос, потому что это не соответствует определению в схеме.

Схемы создаются с использованием языка определения схем GraphQL, также известного как SDL.

Что включает в себя схема?

- Описания типов объектов и полей, которые представляют данные, получаемые через API
- Базовые или корневые типы, определяющие группу операций, которые API может обрабатывать

Другими словами, схема включает в себя описание структуры данных, доступных через API, и допустимых операций, которые могут быть выполнены через этот API. Мы уже знаем три типа возможных операций и корневой тип query должен быть указан обязательно, необязательные корневые типы – mutation, subscription.

Разберем создание схемы GraphQL.

```
type Book {  
  id: Int!  
  title: String!  
  pages: Int  
  chapters: Int  
}  
  
type Query {
```

```
books: [Book!]
```

```
book(id: Int!): Book
```

```
}
```

Приведенный выше код определяет схему GraphQL, которая включает два типа: "Книга" ("Book") и "Запрос" ("Query").

- Тип "Книга" - не корневой, базовый тип
  - Включает четыре поля: "id", "title", "pages" и "chapters"
  - Поля "id" и "title" являются обязательными и должны иметь тип "Int" и "String", а поля "pages" и "chapters" являются необязательными и могут иметь тип "Int" или null
  - Восклицательный знак означает, что поле или аргумент не могут принимать значения null.
- Тип "Запрос" это один из возможных корневых типов, обязательный
  - Включает два поля: "books" и "book"
  - Поле "books" возвращает список объектов "Book", а поле "book" возвращает один объект "Book" на основе заданного параметра "id"
  - Опять же, восклицательный знак означает, что поле или аргумент не могут принимать значения null

В итоге, эта схема позволяет клиенту запрашивать список книг и получать одну книгу по ее ID. Схема гарантирует, что клиент может запрашивать только допустимые поля и параметры, а сервер возвращает данные в правильном формате.

Добавим, каждое поле в типе GraphQL может иметь ноль или более аргументов.

Разберем еще один пример схемы GraphQL.

```
type Book {
```

```
  id: Int!
```

```
    title: String!
    pages: Int
    chapters: Int
  }

  type Query {
    books: [Book!]
    book(id: Int!): Book
  }

  type Mutation {
    createBook(title: String!, pages: Int, chapters: Int): Book!
    updateBook(id: Int!, title: String, pages: Int, chapters: Int): Book!
    deleteBook(id: Int!): Boolean!
  }
```

В этой обновленной схеме мы добавили необязательный корневой тип "Mutation"

- Новый тип включает три поля: "createBook", "updateBook" и "deleteBook"
- Поле "createBook" позволяет клиенту создать новую книгу, указав название и, по желанию, количество страниц и глав. Оно возвращает только что созданный объект книги
- Поле "updateBook" позволяет клиенту обновить существующую книгу, указав ID и поля для обновления (название, страницы и/или главы). Возвращается обновленный объект книги
- Поле "deleteBook" позволяет клиенту удалить существующую книгу, указав ID. Оно возвращает булево значение, указывающее, было ли удаление успешным или нет

После добавления "Mutation" в схему клиент теперь может не только запрашивать книги, но и создавать, обновлять и удалять их.

Далее мы пропустим раздел про документирование, так как схема GraphQL является достаточным описанием сервиса и ее просто можно отправить клиентам вашего API, чтобы они знали как подключаться и какие методы и как использовать.

Остается один вопрос - как система понимает, откуда брать данные по тому или иному полю?

Для этого используются резолверы. Resolver или распознаватели — функция, которая возвращает данные для определённого поля. Эту функцию пишут разработчики и она извлекает данные из различных источников, включая REST API, базы данных и другие источники.

Представьте, что у нас есть схема GraphQL с типом "Клиент", который включает поля "имя" и "возраст". Когда клиент отправляет запрос на объект "Клиент", серверу GraphQL необходимо знать, как получить данные для полей "имя" и "возраст". Здесь на помощь приходят резолверы. Каждое поле в схеме имеет связанную с ним функцию resolver, которая отвечает за получение данных для этого поля. Например, мы можем определить функцию resolver для поля "имя", которая получает имя человека из базы данных, а "возраст" будем получать по API. Подробнее останавливаться на этом не будем, просто договоримся, что благодаря резолверам можно гибко получать данные для запрашиваемых полей из разных API или разных БД или других источников.