

# GraphQL

## Цель работы:

Целью данной практической работы является знакомство обучающихся с набирающим популярность современным подходом к проектированию и реализации API на основе графовых моделей и с реализующей данный подход технологией на основе спецификации GraphQL.

## Краткие теоретические сведения:

В предыдущих работах, нами было рассмотрен один из самых распространённых, на сегодняшний момент, подходов к проектированию и разработке программных интерфейсов в вебе, который основан на архитектуре REST. Используемый ресурсно-ориентированный подход предполагает организацию данных поверх веб-ресурсов, определяемых посредством URI, а для реализации базовых операций, таких как чтение/запись, удаление, обновления использовались базовые глаголы протокола HTTP. Однако, несмотря на все преимущества данного подхода, у REST подхода имеется ряд существенных недостатков, которые могут влиять, при определённых сценариях использования, в том числе, на производительность программного интерфейса.

Самый типичный сценарий взаимодействия с API, демонстрирующий ограничения REST по сравнению с архитектурой на основе графовой модели, является сложная выборка данных. Например, имеется веб-приложение, реализующую функционал социальной сети. Клиентская часть этого приложения (например, веб-клиент) должна отображать множество данных, таких как количество просмотров страницы, количество друзей, подписчиков и т.д. В случае использования REST API, для извлечения всех требуемых данных, обычно, необходимо обратиться к нескольким конечным точкам. Таким образом, количество запросов для отображения полноценной страницы пользователя социальной сети может потребовать несколько десятков, а то и сотен запросов к API. В случае большого количества числа единовременных пользователей такой системы, количество реальных запросов к API может достигать нескольких миллионов запросов в минуту, снижая общую производительность системы в целом. Для решения этой задачи, в рамках REST архитектуры, необходимо обеспечить, либо оптимизацию соответствующей структуры конечных точек, либо оптимизировать количество запросов на сервер API.

Вышеизложенный пример демонстрирует существенный недостаток базовой архитектуры REST, а именно недостаточность выборки данных. Недостаточная выборка, точно также как избыточная выборка данных являются существенными недостатками данного архитектурного стиля. Так, при недостаточной выборке конечная точка не может предоставить все данные, необходимые для реализации пользовательского сценария, что было и продемонстрировано в вышеизложенном примере. Это

обстоятельство приводит к необходимости повторных запросов к API. Для получения всей недостающей информации. Но, в то же время, избыточность выборки данных также является существенной проблемой, приводящей к лишним накладным расходам на передачу ненужных клиенту API данных.

В случае использования графовых моделей извлечение всех требуемых данных происходит одновременно, посредством обращения к единой точке входа API. Таким образом графовая модель позволяет оптимизировать процесс извлечения данных. Графовая модель предоставляет набор методов создания, извлечения данных, которые имплементированы в виде графа, а службы являются узлами этого графа. Данный граф также может включать в себя дополнительные элементы, такие как метаданные, элементы конфигурации, дополнительные службы и т.д. Структура и способы формирования данного графа определяется соответствующей спецификацией, поддерживающую соответствующую графовую модель.

Одним из самых распространённых и широко используемых примеров спецификации графовой модели является GraphQL. Согласно официальному определению с сайта <https://graphql.org/> [6] GraphQL - «это язык запросов для API и среда выполнения для выполнения этих запросов с вашими существующими данными». Многими разработчиками GraphQL позиционируется как технология разработки прикладных программных интерфейсов, ориентированная на реализацию, а не на описание. GraphQL был разработан в рамках работы над социальной сетью Facebook и в настоящее время представлен сообществу разработчиков, как отдельный шаблон взаимодействия в рамках фонда GraphQL Foundation.

В современных реалиях, GraphQL стоит рассматривать именно как технологию, включающую в себя, как спецификацию GraphQL Foundation, так и язык запросов с набором соответствующего инструментария.

Шаблон взаимодействия, по своей природе, описывает поведение сервера API, при этом сами данные и их структура, в отличии от подхода определённого архитектурным шаблоном REST, определяется не самим сервером, а непосредственно клиентом. Так в рамках полноценной REST архитектуры, требуется обеспечить принцип единственного унифицированного интерфейса, в соответствии с требованиями, изложенными в диссертационной работе Роя Филдинга. Унификация пользовательского интерфейса, отчасти, обеспечивается использованием в качестве механизма взаимодействия транспортного протокола HTTP. Современные, полноценные REST сервисы третьего уровня модели зрелости Ричардсона используют практически все возможности инфраструктуры HTTP. Использование 4 основных базовых глаголов HTTP, таких как GET, POST, PUT, и DELETE избавляет разработчика от необходимости придумывать свои методы для реализации базовых операций CRUD для каждого нового приложения, что также влияет на унификацию программного интерфейса и упрощает взаимодействие со множеством программных интерфейсов, реализованных в рамках

архитектуры REST. В случае REST архитектуры идентификация ресурса происходит посредством использования URL, что позволяет, с совместным использованием протокола HTTP, реализовать полноценные механизмы кеширования. В случае же GraphQL, таких соглашений нет и это может накладывать определённые сложности, в том числе и на кеширование (обойти это ограничение возможно с помощью внедрения пакетной технологии обработки и кеширования, например DataLoader, или документоориентированных СУБД, использующих NoSQL, таких как Redis).

В настоящее время многими технологическими компаниями, такими как Facebook, Coursera, PayPal, Twitter, GitHub используется GraphQL. Многие компании, с целью преодоления технологических недостатков архитектуры REST, разработали собственные, чем-то схожие, но в тоже время сильно различающиеся решения, такие как, например, falcor от Netflix[7]. Данный пример очень показателен тем, что в дальнейшем Netflix отказалась от использования falcor, отдав предпочтение GraphQL[8]. При этом, простота миграции с одной технологии на другую стала возможной, благодаря тому, что контракт между клиентом и сервером в рамках технологии GraphQL определяется, непосредственно, схемой SDL(англ. Schema Definition Language), которая и определяет соответствующие бизнес-процессы. Таким образом технология GraphQL можно использовать для унификации всех систем в единый API. На рисунке 6.1 показана возможная схема организации данного программного интерфейса, как слоя интеграции.

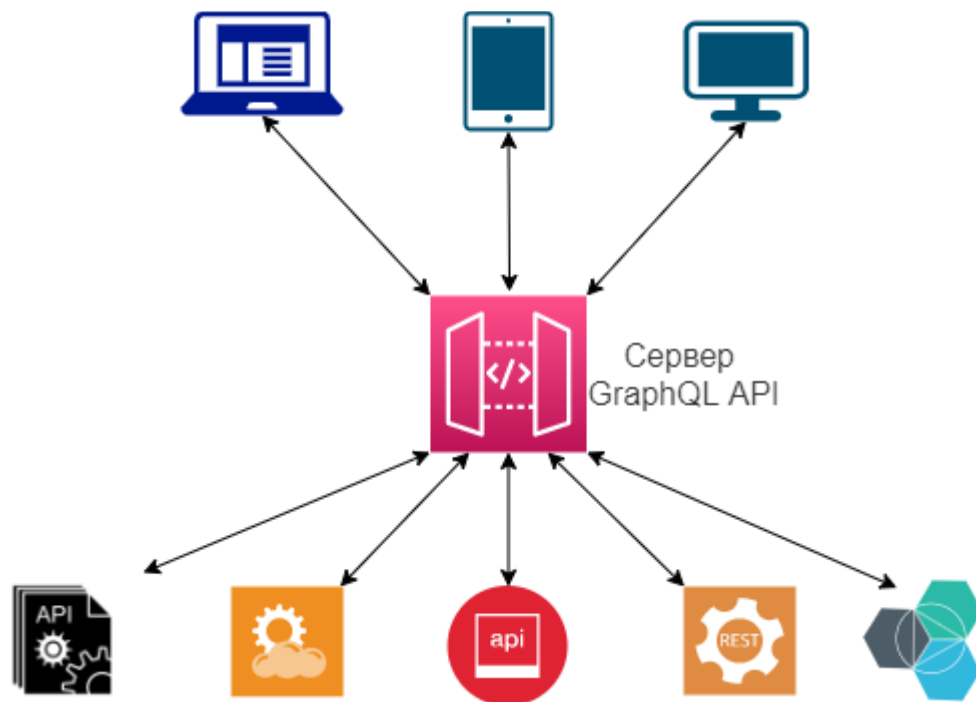


Рисунок 6.1 - Пример использования GraphQL как слоя интеграции разрозненных компонентов

Как видно из рисунка 6.1, сервер GraphQL может являться промежуточным слоем, который интегрирует различные сервисы в единый механизм, обеспечивая при этом единую точку входа, что в свою очередь позволяет унифицировать доступ клиентам. В таком сценарии использования, сервер GraphQL действует как шлюз, предоставляя собственную модифицированную реализацию классического паттерна API Gateway, использование которого, наиболее распространено в рамках микросервисных архитектур. На практике сценариев использования и реализации GraphQL довольно-таки большое количество, но в любом случае, независимо от сценария использования, GraphQL всегда будет обеспечивать единую точку входа, которая будет получать данные с запросом, в котором будут описаны, какие ресурсы запрашиваются. На рисунке 6.2 показан сценарий использования сервера GraphQL для взаимодействия с хранилищем данных.

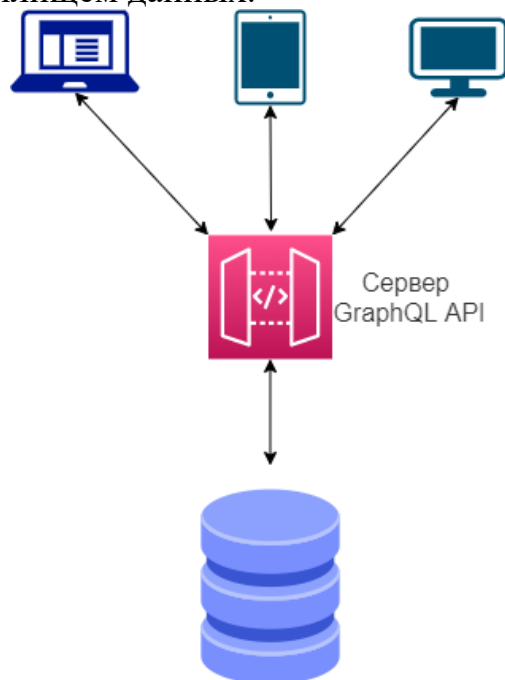


Рисунок 6.2 - Сценарий использования GraphQL в качестве промежуточного слоя для доступа к данным

Сценарий, показанный на рисунке 6.2, позволяет упростить взаимодействие с хранилищем данных, в условиях гетерогенности клиентов, в силу того, что GraphQL поддерживает *запросы* и *мутации*, с помощью которых можно реализовать базовые операции чтения и записи данных. В тоже время, нет никаких общепринятых положений, стандартов, которые бы требовали, чтобы на одном сервере GraphQL реализовывалась только одна единая точка. Возможны сценарии, когда взаимодействие с клиентом будет происходить в контексте запросов GraphQL, которые будут трансформироваться традиционные REST или наоборот. Для упрощения и автоматизации решения данной задачи разработано большое

количество программных инструментов, например `tyk.io`. Такой подход формирует отдельное направление архитектурных решений программных интерфейсов в вебе. Программные интерфейсы, обладающие такими свойствами реализуют *гибридный* подход к проектированию и реализации API. Такие API на практике и в технической литературе могут именоваться *гибридными*.

Далее рассмотрим основные подходы к . Как было уже сказано выше, в основе сервера GraphQL лежит схема, сформированная с использованием языка определения схемы - SDL, которая, в свою очередь, определяет модель доступа к данным, которые могут быть получены. Именно схема определяет, как можно получить соответствующие данные, определяет какие запросы возможны, какие типы данных используются и т.д. Базовым элементом схемы GraphQL является понятие «*тип*» (англ. *type*), с соответствующими полями, внутри которых, в том числе, определяются отношения с другими типами. Базовая структура типа представлена ниже.

```
type название типа {  
    Название поля №1: тип возвращаемых данных  
    Название поля №2: [список] # Комментарий  
}
```

В данном описании после ключевого слова `type` идёт именование типа. В фигурных скобках определяются поля типов, с указанием типа возвращаемых данных. В настоящее время поддерживаются в качестве возвращаемых значений объекты, скаляры, перечисления, интерфейс и объединения [9]. Количество полей определяется соответствующей структурой данных, описываемой с помощью схемы. Ниже представлен пример определения типа на SDL.

```
Type News {  
    numbers: Int #возвращает скалярный тип INT  
    author: String  
}
```

Определённый выше тип является логически объектом, а в графе тип определяет тип узла, с соответствующими свойствами. Именно через поля типа осуществляется указание, какие свойства будет иметь этот узел. В данном примере определён тип объекта `News`, при этом в схеме не определено, где данные будут храниться и откуда будут извлекаться, что делает схему независимой от конкретной реализации. Ориентируясь на описание схемы GraphQL клиент определяет те данные, которые он может получить от сервера API. Таким образом, схема является текстовой нотацией графа данных и определённых над данным графом операций. В настоящее время, выделяют 3 основные базовые операции:

- *запрос* (англ. *query*) - предназначены для извлечения данных

- *мутация* (англ. *mutation*) - позволяют данные изменять
- *подписка* (англ. *subscription*) - схожи с запросами, с той лишь разницей, что клиенты при таком способе взаимодействия с сервером клиент постоянно ожидает возникновения события на сервере и в случае возникновения события получает определённые данные без необходимости повторных запросов.

Рассмотрим пример описания схемы для простейшей бизнес задачи. Стоит понимать, что, с целью облегчения понимания на первоначальных этапах знакомства с данной технологией, структура спроектированной схемы будет на данном этапе значительно упрощена, а в дальнейшем, при её реализации в программном обеспечении, будет немного расширена. Итак, мы являемся разработчиками сервиса микроблогинга (некий аналог Twitter). Перед нами стоит задача разработать простейший функционал веб системы микроблогинга. При взаимодействии с данным приложением пользователь должен получать краткую информацию о всех постах в микроблоге данного автора. При нажатии на соответствующую запись в микроблоге читатель должен получить доступ к полному сообщению в микроблоге. Автору доступна возможность создания нового поста в микроблоге.

Для того, чтобы описать схему GraphQL для данной бизнес-задачи необходимо, определить данные, объекты и взаимоотношения между объектами. Любое сообщение в сообщении микроблога может иметь такие данные, как дата создания поста, автор сообщения, краткая аннотация сообщения и т.д. В свою очередь автор сообщения может иметь поле с фамилией, именем, отчеством, имя пользователя и т.д. Выделение автора в отдельную структуру данных может быть оправдано для расширения возможного функционала. Так, в дальнейшем, может потребоваться получать информацию о всех постах данного автора. Выделение автора, как отдельную сущность позволит решить эту задачу проще. Исходя из вышеизложенного, для данной бизнес-задачи объектами могут быть пост в микроблоге и автор поста в микроблоге. Далее необходимо определить отношения между объектами. Так каждый пост в микроблоге обязательно имеет одного автора, и в тоже время, один автор может иметь более одного сообщения в микроблоге. На рисунке 6.3 показан граф отношений между объектами.

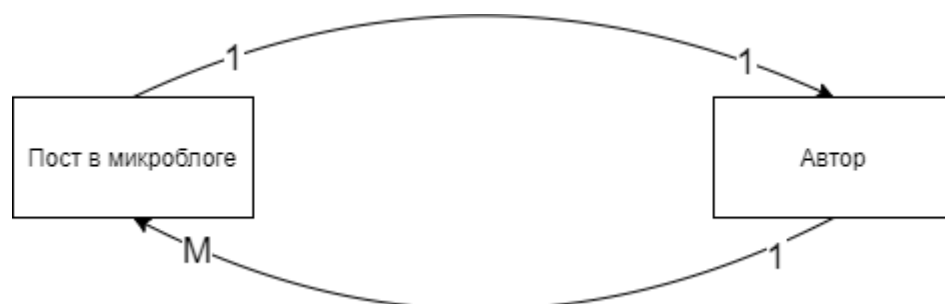


Рисунок 6.3 - Схема отношений между объектами «Автор» и «Пост в микроблоге»

На рисунке 6.3 объектами являются «Автор» и «Пост в микроблоге», которые являются узлами графа, а отношения между объектами определяют дуги графа. Узлы также содержат информацию об объектах, определяемых полями в схеме GraphQL. После того, как выделены объект и определены отношения между объектами, необходимо, используя язык SDL, трансформировать спроектированный граф в схему GraphQL. Стоит отметить, что при формировании схемы необходимо следовать соглашению о спецификации GraphQL, которая размещена на официальном портале [graphql.org](https://graphql.org).

Опишем объекты «Автор» и «Пост в микроблоге» как тип объекта BlogAuthor и Blog с соответствующими полями, а также отношения между объектами:

```
type Blog {  
  bid: ID!  
  annotation: String  
  author: BlogAuthor!  
}  
  
type BlogAuthor {  
  uid: ID!  
  username: String!  
  name: String  
  surname: String  
  blog: [Blog] #список постов автора  
}
```

Свойства объектного типа Blog определены полями bid, annotation и author. Свойства объектного типа BlogAuthor определены полями uid, username, name, surname, blog. Стоит обратить внимание на символ восклицательного знака в полях bid, author и uid, username. Наличие восклицательного знака определяет данное поле, в данном контексте, как обязательное, что означает, что данное поле не может быть неопределено (N/A) или определяться как NULL. Это связано с тем, что в соответствии с бизнес-задачей пост в микроблоге не может быть анонимным, в обязательном порядке он должен иметь уникальный идентификатор, а автор должен определяться, как минимум, его пользовательским именем (никнеймом), выбранным и указанным при регистрации. Поле uid также позволяет идентифицировать автора и в некоторых случаях идентификация с использованием uid (user ID) может быть более предпочтительна.

В выше указанном примере отношения между объектами BlogAuthor и Blog определены, посредством указания в поле объекта другого объекта. В данном примере, свойство **author** в типе Blog показывает, что данный тип связан с другим типом BlogAuthor, через данное поле. В терминологии графовой модели это означает, что узел, определяемый типом Blog, имеет отношение с другим узлом, определяемый типом BlogAuthor и данное отношение определено исходящим ребром с меткой **author**. Таким образом, поле **author** является одновременно и ребром в графовой модели. При этом данное ребро обязательно, так как тип возвращаемого значения помечен восклицательным знаком. На рисунке 6.4 показана графовая модель.

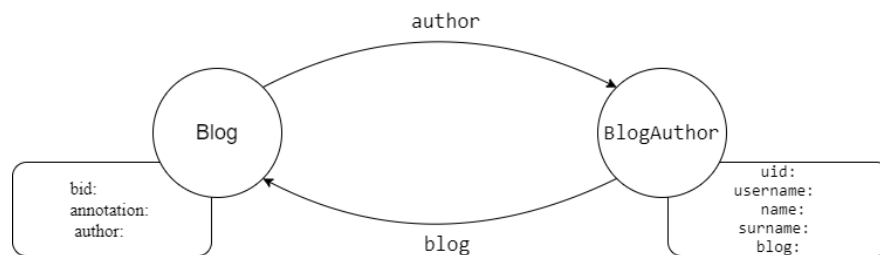


Рисунок 6.4 - Граф отношений объекта Blog и объекта BlogAuthor

После того как граф отношений между объектами типа сформирован, необходимо в схеме определить ещё особые типы объектов, которые будут определять операции над данными. Ранее было показано, что GraphQL определяет 3 базовые операции: запрос, мутация, подписка. *Query* операции можно отождествлять с операциями извлечения данных, а *mutation* операции - с операциями изменения данных. *Subscription* для данного примера использованы не будут. Определим *Query* операцию для извлечения всех постов в микроблоге и поста по конкретному bid:

```
type Query {  
  PostMessage(bid: ID!): Blog  
  ListBlogs(): [Blog]  
}  
  
type Blog {  
  bid: ID!  
  annotation: String  
  author: BlogAuthor!  
}  
  
type BlogAuthor {  
  uid: ID!  
  username: String!
```



```

    name: String
    surname: String
    blog: [Blog] #список постов автора
}

```

В данном примере в типе Query определены два поля `PostMessage` и `ListBlogs`. Поле `PostMessage` возвращает соответствующий тип `Blog`, а поле `ListBlogs` возвращает список соответствующего типа.

Для изменения данных необходимо использовать тип `Mutation`. В данном примере type `Mutation` может быть определён следующим образом:

```

type Query {
    PostMessage(bid: ID!): Blog
    ListBlogs(): [Blog]
}

type Mutation {
    addBlog(annotation: String, author: String, authorID: ID!): Blog!
}

type Blog {
    bid: ID!
    annotation: String
    author: BlogAuthor!
}

type BlogAuthor {
    uid: ID!
    username: String!
    name: String
    surname: String
    blog: [Blog] #список постов автора
}

```

Стоит обратить внимание на именование query операций. Рекомендуемой практикой именования операций извлечения данных является использование приставок *get* к названию поля (классические гетторы в объектно-ориентированном программировании), а для операций мутаций - *set* + название поля. Таким образом, название поля `PostMessage` в операции query, может быть заменено на `getPostMessage`, а поле `ListBlogs` может быть заменено на `getListBlogs`. Для упрощения проектирования схем

на языке SDL можно воспользоваться различными инструментами. Среди них можно отметить среды проектирования GraphQL[10] и GraphQL IDE [11], а также средство визуального проектирования GraphQL Editor [12]. Данное программное обеспечение позволяет упростить процесс написания схем и визуализировать спроектированные схемы, что значительно упрощает понимание сложных GraphQL схем. На рисунке 6.5 показана визуализация спроектированной схемы в GraphQL Editor.

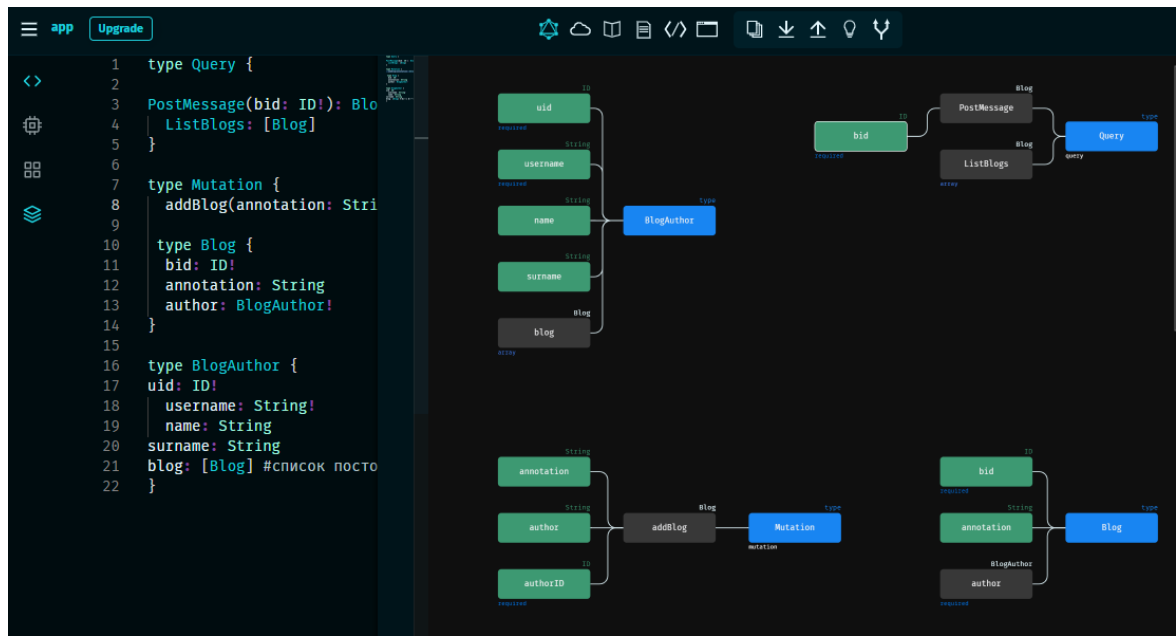


Рисунок 6.4 - Визуализация разработанной схемы

Отличительной особенностью сервиса GraphQL Editor является то, что он позволяет создать имитационный сервер GraphQL для тестирования спроектированной схемы (вкладка «GraphQL Cloud» верхнего меню, далее вкладка «faker», в новом слое «Choose endpoint» поле «Fake backend»). Перейдя по указанной системой ссылке (конечной точке) можно попасть на имитационный, созданный системой сервер GraphQL, в среду клиента GraphQL. На рисунке 6.5 показано окно клиента GraphQL.

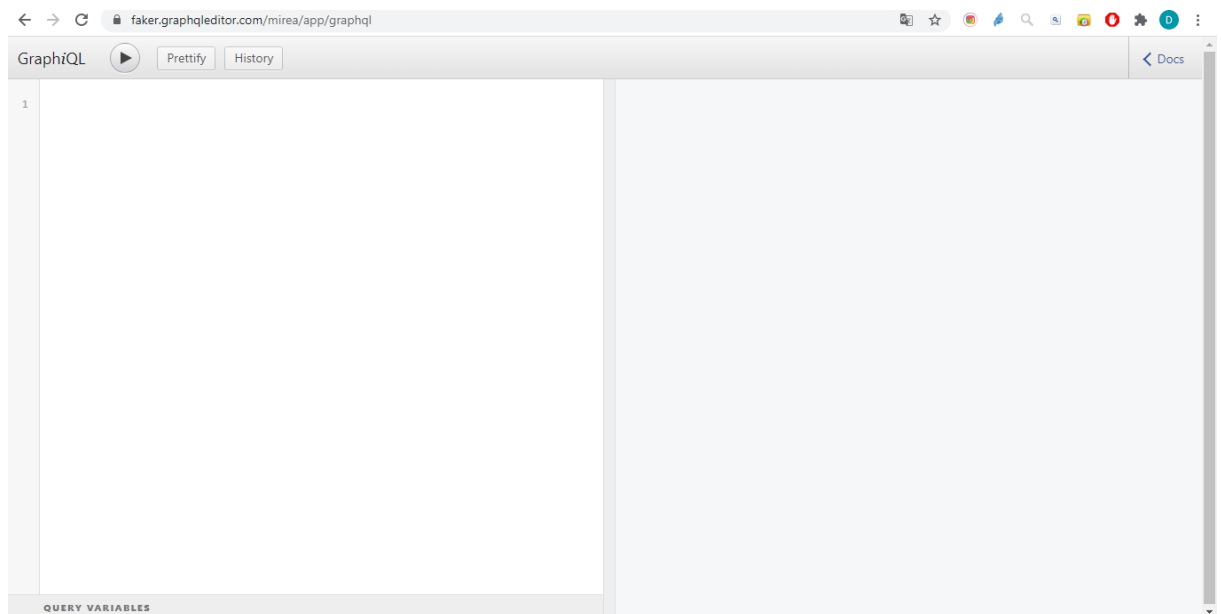


Рисунок 6.5 - Окно клиента GraphiQL

В данном окне можно делать запросы к имитационному серверу. Сделаем запрос на извлечение данных (операция *Query*):

```
query MyQuery {  
  ListBlogs {  
    author {  
      name  
      surname  
    }  
    bid  
  }  
}
```

Результат операции *Query* показан на рисунке 6.5.



Рисунок 6.5 - Результат операции *Query*

Как видно из рисунка 6.5, клиент GraphQL отправил запрос на извлечение списка всех блогов, с необходимыми полями (имя, фамилия и номер сообщения в блоге). GraphQL сервер вернул сгенерированные данные, так как сервер является имитационным и данные ранее в систему не вносились. Подобное программное обеспечение удобно использовать именно для тестирования спроектированной схемы и проверки её корректности, в случаях, если, например, серверная часть ещё не готова.

Процесс выполнения запроса GraphQL стоит рассмотреть более подробно. Как было уже сказано выше, GraphQL это не только язык запросов, но и целая технология со своей архитектурой. В рассмотренном выше примере, простейшего API по трёхуровневой схеме клиент-сервер взаимодействия с отдельным хранилищем данных. Сервер GraphQL, в данном сценарии, одновременно являлся и шлюзом, что и позволило направить все запросы в единую конечную точку сервера API. Однако на практике, для повышения производительности, обеспечения кеширования операций и решения других задач, может вводиться ещё один слой, представляющий собой прокси-сервер. На рисунке 6.6 показана архитектура такого решения.

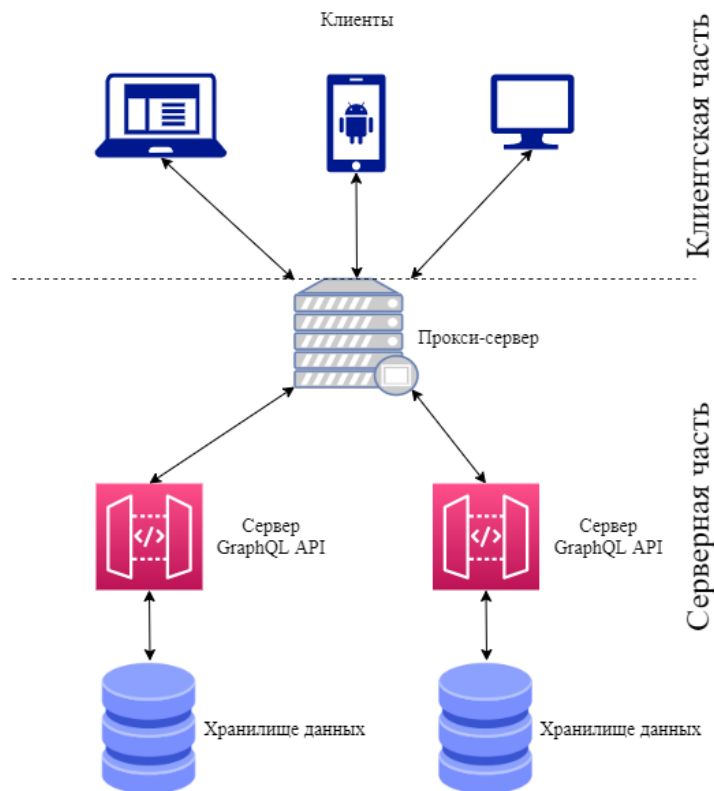


Рисунок 6.6 - Архитектура GraphQL API с прокси-сервером

При таком сценарии, прокси-сервер выполняет роль шлюза API, что может приводить к повышению производительности сервера API. На

данной схеме показано, что данный шлюз позволяет интегрировать разные GraphQL серверы в единую систему.

Одной лишь спроектированной схемы не достаточно для полноценной реализации GraphQL сервера API. Для того, чтобы сделать программный интерфейс работоспособным нужно иметь запущенный сервер GraphQL, клиент, который будет осуществлять запросы к прокси-серверу, который будет осуществлять кеширование запросов к серверу GraphQL, соответствующий программный код на стороне сервера, реализующую основную логику, а также базу данных. В такой схеме клиентская часть отправляет запрос на серверную часть, который на серверной стороне проверяется и, в случае успешной проверки, выполняется сервером GraphQL. Схема исполнения Query операции показана на рисунке 6.7.

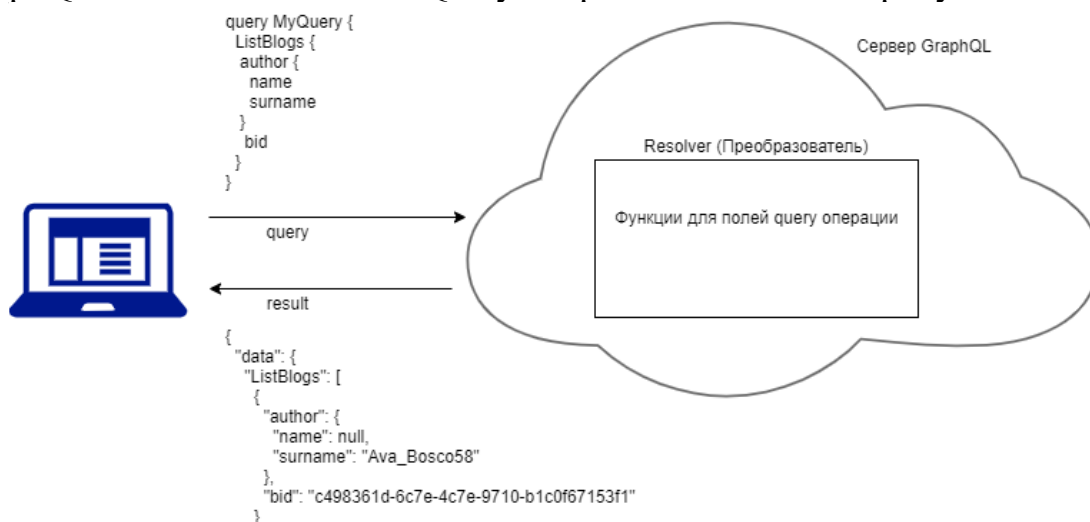


Рисунок 6.7 - Схема исполнения query операции

Схема выполнения запросов предполагает, выполнения запроса всегда начинается с корневого поля и выполняется в ширину. Резольверы по своей природе аналогичны REST контроллерам, которые присутствуют во многих фреймворках, позволяющих реализовать архитектуру REST, например, Spring и т.д. Основной задачей резольвера является обработка данных и возвращение требуемых результатов, так как только резольвер содержит и хранит соответствующие доступные методы для разрешения типа и предоставления связей с другими типами. Другими словами, преобразователь является функцией вызываемой для каждого поля операции, которая возвращает данные для требуемого поля типа, учитывая при этом связь данного типа с другими типами. Именно сервер GraphQL разрешает запросы, вызывая соответствующие резольверы каждого поля операции, ожидая окончания выполнения всех резольверов и извлекая все данные из базы данных, после чего, возвращает клиенту ответ в соответствующем формате. Необходимо обратить внимание не то, что некоторые реализации серверов GraphQL могут создавать резольверы по умолчанию, что, в некоторых случаях, избавляет разработчика от указания резольвера для всех полей. Что представляют из себя функции

резольверов? Внутри функций резольверов может быть любой программный код, в котором и заключена логика по связи полей запросов с механизмами хранения и извлечения данных. Например, для примера с приложением для микроблога запрос на извлечение содержимого блога может выглядеть следующим образом:

```
query MyQuery {
  PostMessage(bid: 10)
  { annotation
    author {
      name
      uid
    }
  }
}
```

Возможный ответ имитационного сервера на данный запрос показан на рисунке 6.8.

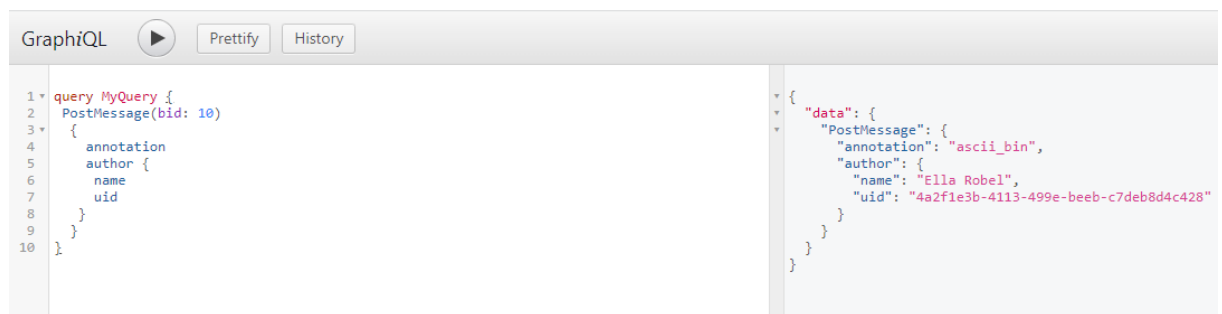


Рисунок 6.8 - Ответ сервера на query операцию

Сервер GraphQL, скрывая всю сложность от клиента, вернул требуемые ему данные, при этом, в тексте запроса явно не указывалось ни формат требуемых данных, ни конкретные механизмы получения этих данных, ни конкретные хранилища данных, откуда эти данные извлекаются. Такой подход упрощает взаимодействие клиента с программным интерфейсом, так как именно разработчик программного интерфейса обеспечивает все требуемые механизмы извлечения данных на стороне сервера. Рассмотрим структуру URL адреса конечной точки имитационного сервера GraphQL API

<https://faker.graphqleditor.com/mirea/app/graphql?query=query%20MyQuery%20%7B%0A%20PostMessage...>

В данном URL можно явно выделить две составляющие, разделённые вопросительным знаком. Часть до знака разделителя «?» определяет непосредственно конечную точку, а оставшаяся часть, после данного знака разделителя, является передаваемым параметром, а в данном случае через эти параметры передаются операции над данными. При таком подходе конечная точка будет одной (она же точка входа), а вся сложность конечной точки скрыта за передаваемыми параметрами, что является

хорошим подходом к проектированию дизайна структуры конечных точек, в рамках разработки программных интерфейсов в вебе.

Остаётся открытым вопрос с реализацией извлечения данных из хранилищ. Так как внутри преобразователя возможна реализация программного кода для доступа к данным. Для вышеприведённого примера преобразователи могут выглядеть следующим образом:

```
PostMessage(_, args){  
  return sql.raw('SELECT * FROM message WHERE bid = %s', args.id);  
} ListBlogs(author){  
  return request(`https://microblogtest.net/v1/writers/${author.bid}`);  
}
```

Однако такой подход имеет ряд недостатков, поэтому на практике SQL запросы, URL адреса и т.д. лучше вынести в отдельный программный код. Отдельно хочется отметить про программное обеспечение, упрощающее процесс интеграции и создания масштабируемых GraphQL API. Так консорциум GraphQL разработана собственная система управления графовой базой данных - Dgraph. Так согласно официальной документации, Dgraph представляет собой систему управления распределёнными базами данных с открытым исходным кодом, которая позволяет быстро развернуть базу данных по схеме GraphQL (по своей природе Dgraph может выполнять роль бэкенда). Это может быть сделано разными способами, например, путём определения структуры данных в формате документа JSON, либо путём передачи схемы GraphQL, с определёнными в ней типами, в сервер Dgraph. Dgraph, на основании переданной ему схемы, подготавливает графовую базу данных для хранения данных с их отношениями и запускает сервер GraphQL API. Такой подход частично избавляет программиста от написания серверного кода.

В качестве примера рассмотрим процесс создания простого клиента для API микроблога с использованием сервера Node.js, который использовался в предыдущих практических работах.

Прежде всего необходимо проверить установлен ли Node.js на локальной машине, выполнив в Node.js Command Prompt команду *node -v*. В случае, если установлен в терминал вернётся информация о номере версии установленного Node.js. Далее необходимо создать директорию для своего проекта и инициализировать проект JS, выполнив последовательно команды, как показано на рисунке 6.9.



```
Node.js command prompt
Your environment has been set up for using Node.js 12.18.3 (x64) and npm.

C:\Users\L>cd C:\

C:\>mkdir projectapp

C:\>cd C:\projectapp

C:\projectapp>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

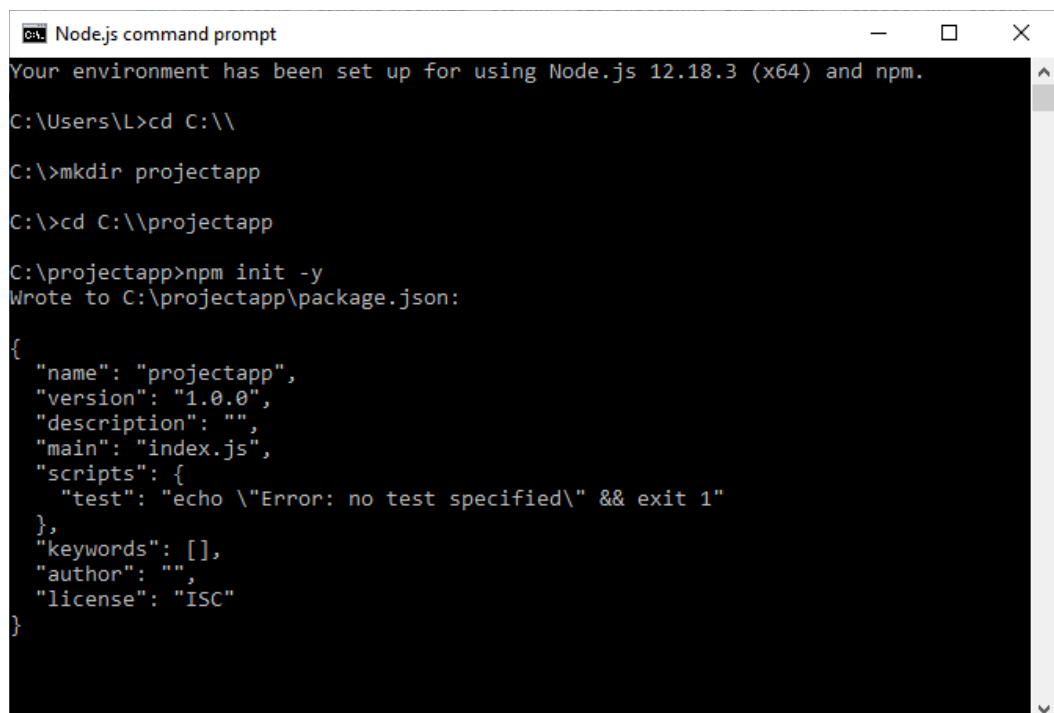
Press ^C at any time to quit.
package name: (projectapp)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\projectapp\package.json:

{
  "name": "projectapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Рисунок 6.9 - Создание директории проекта и инициализация проекта

На рисунке 6.9 создана директория *projectapp* в корне диска C:\ , а также сконфигурирован и инициализирован данный проект командой *init*.

Для того чтобы использовать конфигурацию по умолчанию и не настраивать соответствующие поля, можно выполнить *npm init* с ключом *-y*, как показано на рисунке 6.10.



```
Node.js command prompt
Your environment has been set up for using Node.js 12.18.3 (x64) and npm.

C:\Users\L>cd C:\

C:\>mkdir projectapp

C:\>cd C:\projectapp

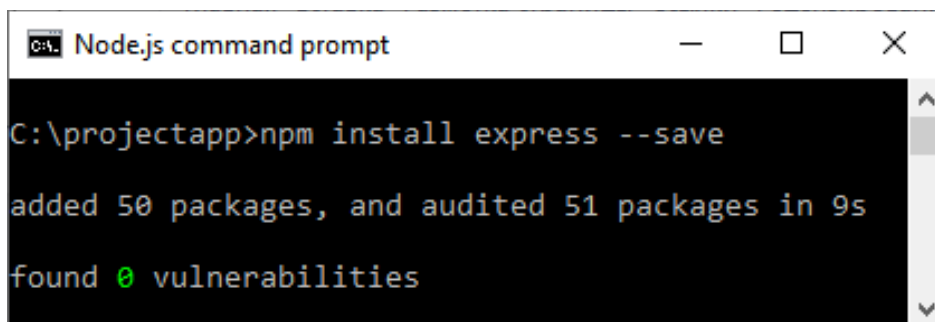
C:\projectapp>npm init -y
Wrote to C:\projectapp\package.json:

{
  "name": "projectapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Рисунок 6.9 - Создание директории проекта и инициализация проекта конфигурацией по умолчанию



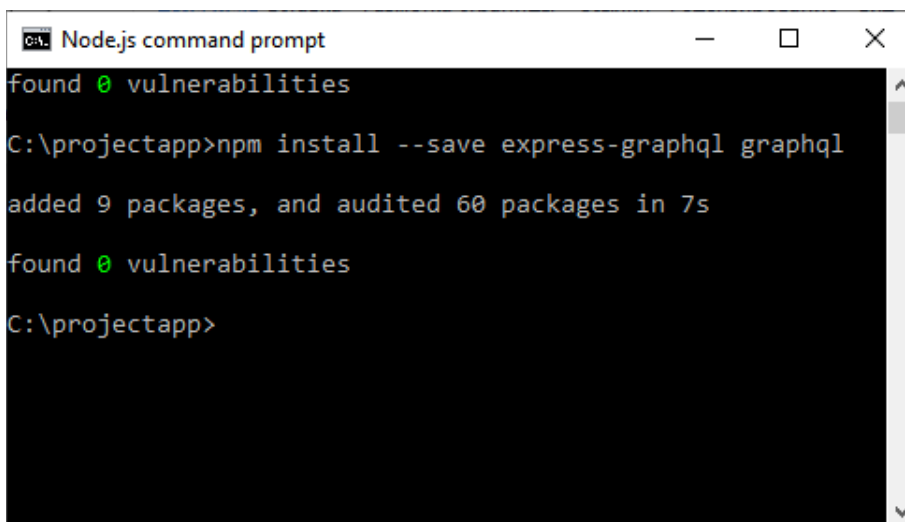
Далее, находясь в директории проекта, установим веб фреймворк `express`, командой `npm install express`. Результат выполнения данной команды показан на рисунке 6.10.



```
Node.js command prompt
C:\projectapp>npm install express --save
added 50 packages, and audited 51 packages in 9s
found 0 vulnerabilities
```

Рисунок 6.10 - Установка Express

Для того, чтобы избежать многочисленных предупреждений от `npm` необходимо при инициализации проекта полностью его сконфигурировать, заполнив соответствующие поля или использовать конфигурацию по умолчанию. Установим ещё два пакета - `express-graphql` и `graphql`, с помощью команды `npm install --save express-graphql graphql`. `GraphQL` библиотека, которая предназначена для исполнения JavaScript в среде GraphQL, `express-graphql` является библиотекой для интеграции Express со средой GraphQL. Результат выполнения данной команды показан на рисунке 6.11.



```
Node.js command prompt
found 0 vulnerabilities
C:\projectapp>npm install --save express-graphql graphql
added 9 packages, and audited 60 packages in 7s
found 0 vulnerabilities
C:\projectapp>
```

Рисунок 6.10 - Установка express-graphql и graphql

Далее в директории проекта создадим файл `index.js` (по умолчанию в конфигурации указан именно данный файл) следующего содержания:

```
const express = require('express');
const app = express(); // инициализация объекта приложения
const port = 1234; // номер порта
```

```
//выполнение серий функций req,res при совпадении
корневого пути
app. use('/',(req,res) => {
    res.send("Сервер GraphQL запущен! Единая входная
точка сервера GraphQL /graphql")
})
app.listen(port); // прослушиваем порт 1234
```

Структура проекта проекта и содержимое файла показано на рисунке 6.11.

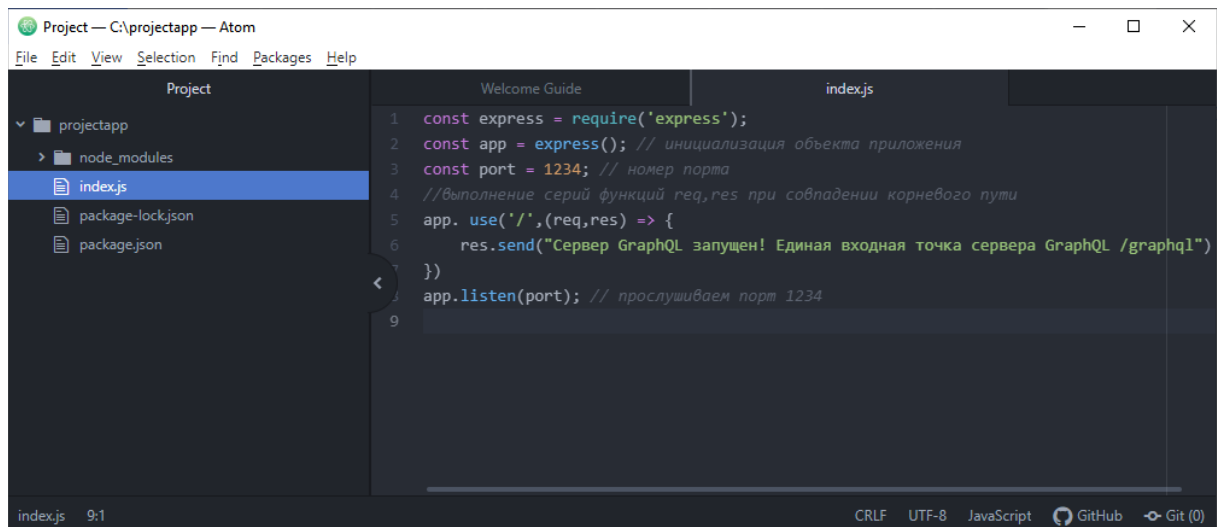


Рисунок 6.11 - Структура проекта и содержимое файла index.js

Данное приложение Express можно запустить путём выполнения в Node.js command prompt команды `node index.js`, находясь в директории проекта. В случае успешного запуска по адресу `http://localhost:port`, где `port` соответствует номеру порта, указанного в `const port` приложения, будет виден следующий результат, как показано на рисунке 6.12.

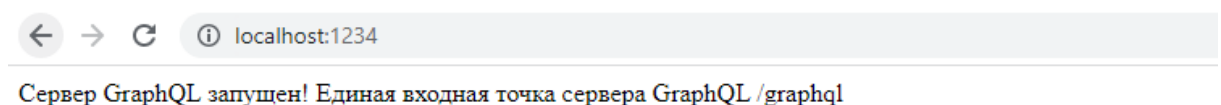


Рисунок 6.12 - Результат работы запущенного приложения Express

Дальнейшим шагом является определение конечной точки сервера для создания маршрута и определении схемы. В данном случае конечной точкой будет `/graphql`. Для этого модернизируем содержимое файла

*index.js*, который является точкой входа для нашего приложения, как показано на рисунке 6.13.

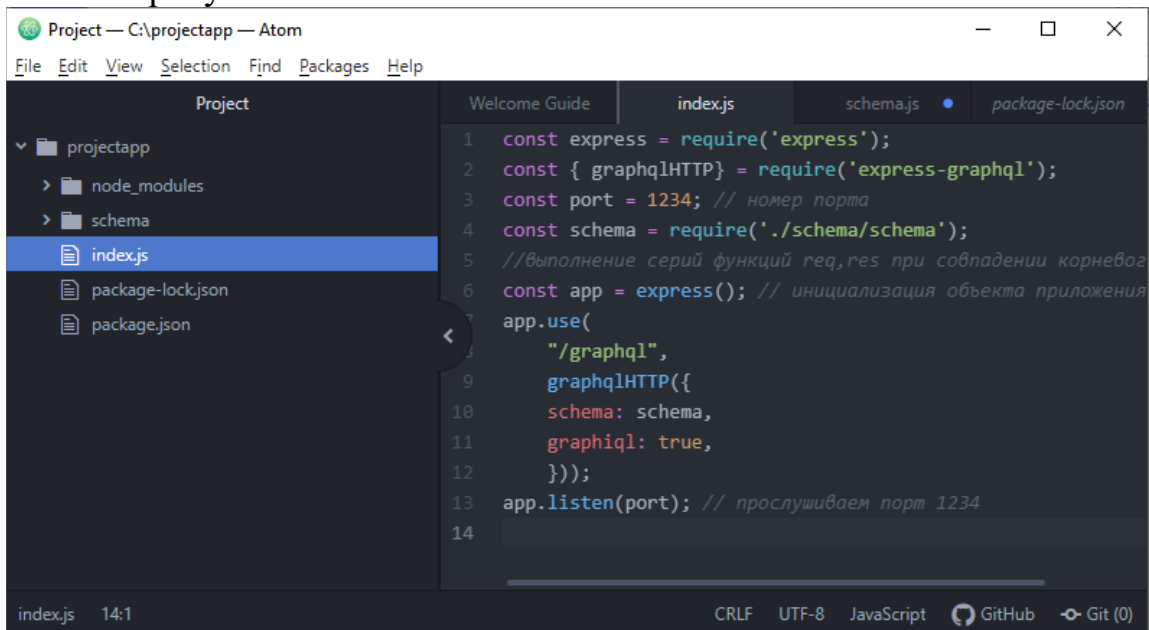


Рисунок 6.13 - Создание конечной точки и использование промежуточного ПО Express

После внесения изменения и перезапуска сервера, можно обратиться к конечной точки <http://localhost:1234/graphql>. Результат должен быть таким, как показано на рисунке 6.14.

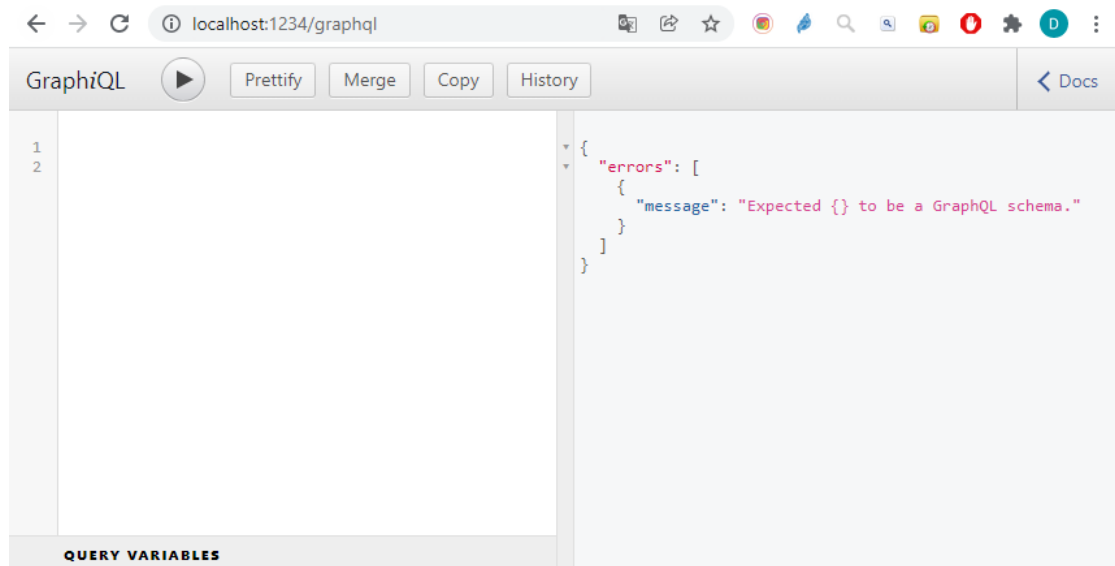


Рисунок 6.14 - Результат обращения к конечной точке

В отличие от REST, где требовалось создание нескольких конечных точек, в нашем сценарии использования, GraphQL будет использовать только одну единственную конечную точку */graphql*. Именование данной конечной точки, в зависимости от задач и проекта, можно выбрать любое, например */api*. Стоит также обратить внимание на сообщение об ошибке, которую генерирует *graphiql*. Так как ранее была создана новая директория

schema, где будет размещаться schema.js файл, но так она ещё пустая, поэтому генерируется сообщение об ошибке.

Далее необходимо привязать схему GraphQL для данного проекта. Прежде чем сервер сможет отдавать требуемые клиентом данные, необходимо описать с какими данными и каким образом сервер будет работать с данными. Для этого и необходима схема GraphQL. Для этого ранее была определена новая директория *schema* в структуре нашего проекта, где будем размещать файлы, связанные со схемой GraphQL. В данной директории создадим файл с расширением \*.js и подключим graphql к данному проекту, импортируя *graphql* и соответствующие типы *GraphQL*, как показано на рисунке 6.14.

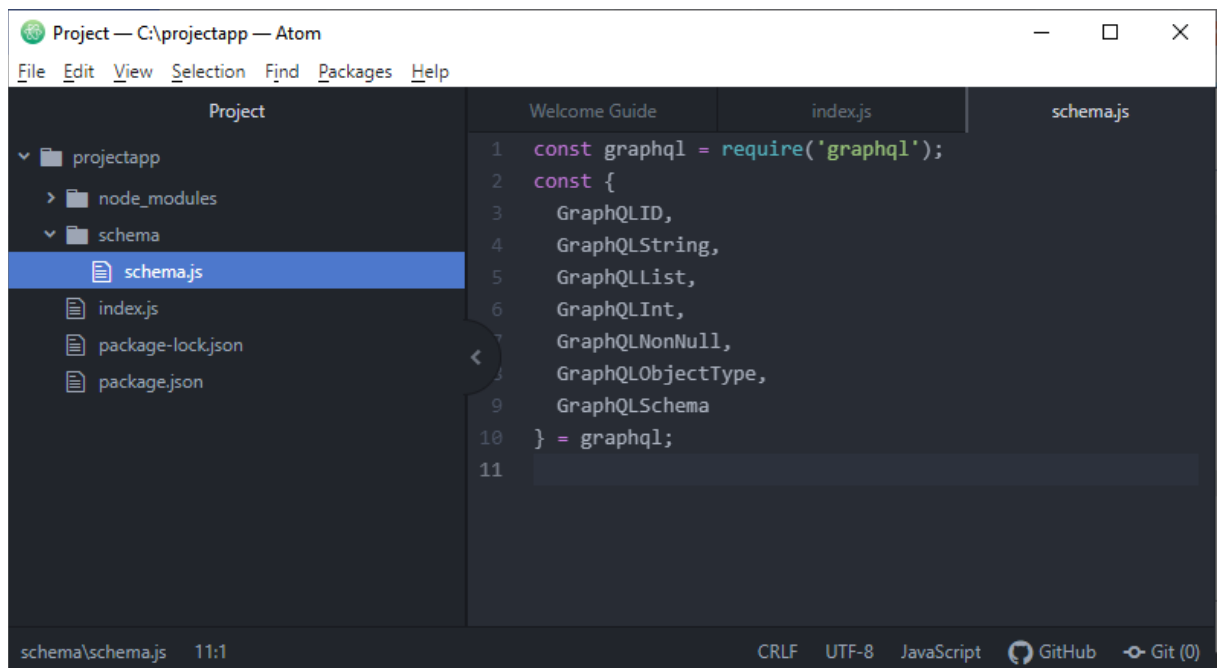


Рисунок 6.14 - Разбивка сложной структуры объекта graphql

Далее необходимо определить схему и корневой запрос для сервера. Нужно помнить, что каждый GraphQL сервер должен иметь корневую операцию, при этом из трёх операций *query*, *mutation* и *subscription*, каждая из которых имеет один связанный с ней тип, обязателен только *query* тип и этот тип должен быть объектным типом. Схема должна определять и содержать начальный тип корневой операции. Именно корневой *query* тип задаёт структуру полей ответа на основе выбранных полей объекта. Корневая операция, которая находится на самом верху иерархии, также задаёт ту самую точку входа сервера GraphQL. Реализуем данную корневую операцию, которая, на данный момент, будет просто выдавать текст с информацией о том, что сервер запущен, если *query* запрос будет содержать поле *info* и экспортируем схему. Это даст гарантию, что сервер обрабатывает поступающие запросы. Пример исходного кода приведён на рисунке 6.15.

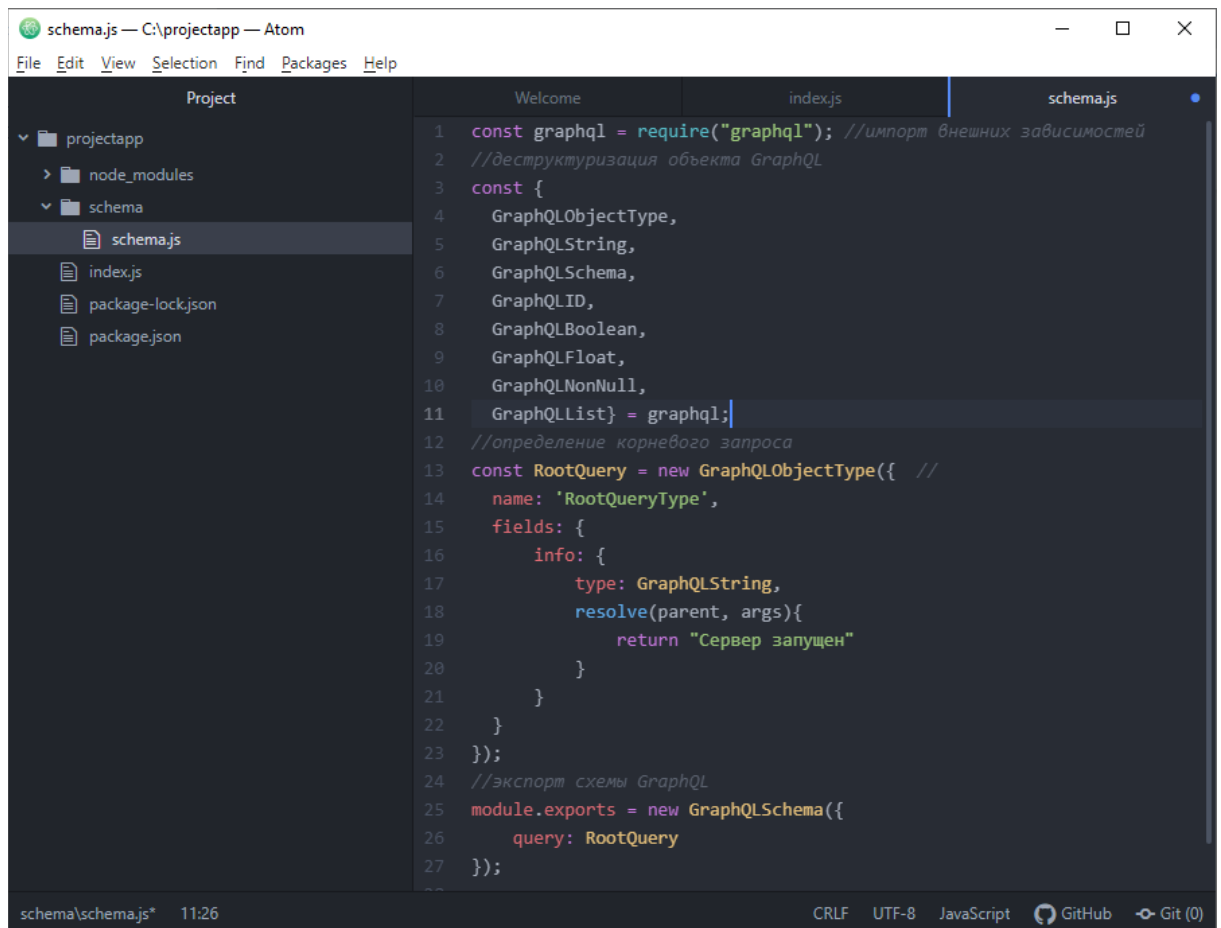


Рисунок 6.15 - Реализация корневого объекта RootQuery и экспорт экземпляра GraphQLSchema

Для того, чтобы в ручную не перезапускать сервер после каждого изменения в исходных файлах приложения, можно воспользоваться инструментом nodemon, установив его через npm, выполнив команду *npm install nodemon --save -dev* и добавив в содержимое файла package.json строку "start": "./node\_modules/nodemon/bin/nodemon.js ./src/index.js".

Для того чтобы проверить работоспособность конечной точки можно в GraphQL сделать запрос следующего содержания, как показано на рисунке 6.16(поле graphql: должно определяться true, иначе графического интерфейса GraphQL у конечной точки не будет и все запросы нужно будет определять через параметры в строке браузера).

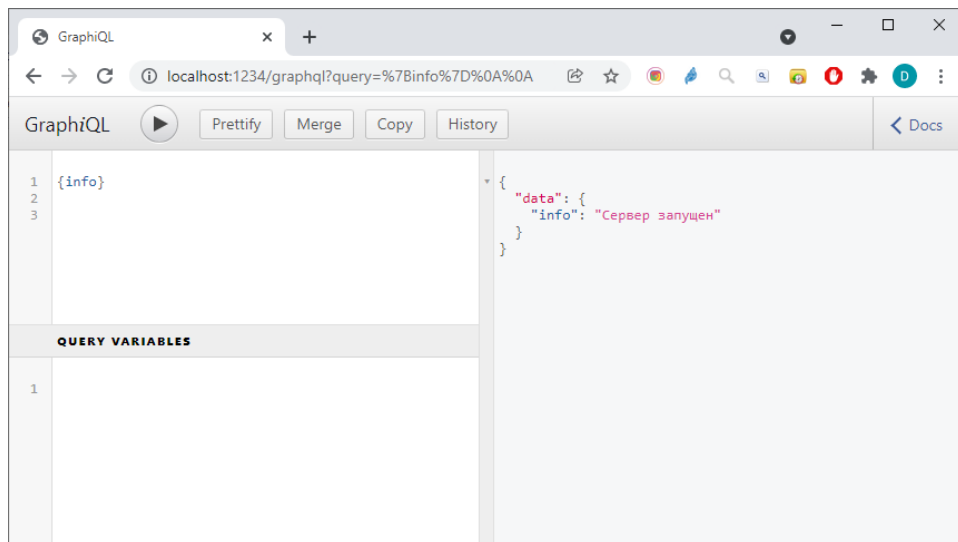


Рисунок 6.16 - Результат обработки запроса о текущем состоянии сервера

Стоит обратить внимание, что при нажатии на меню «Docs» в GraphQL, откроется выпадающее меню, в котором будет определена схема с корневой точкой, как показано на рисунке 6.17.

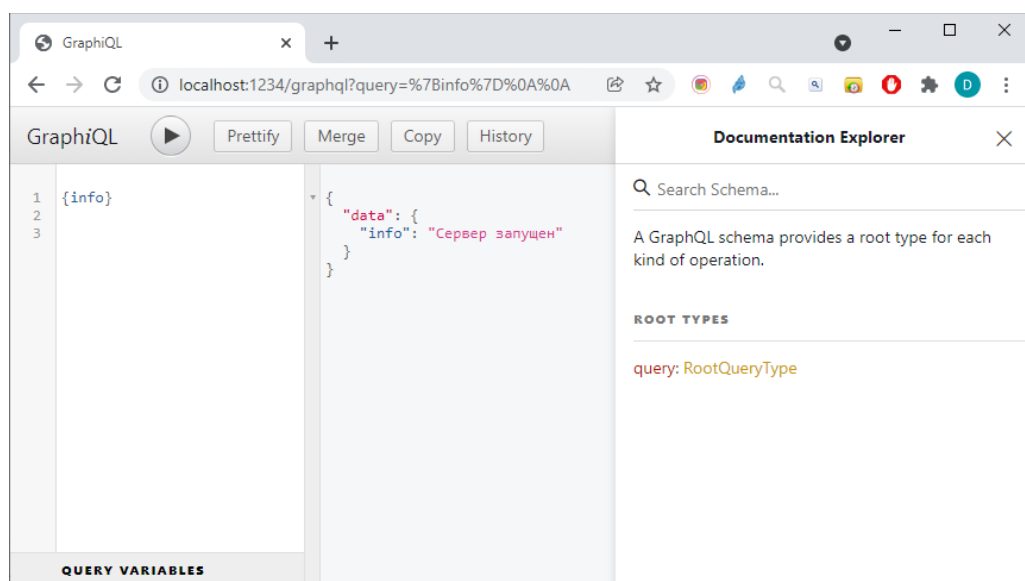


Рисунок 6.17 - Проводник схемы с корневой точкой RootQueryType

Далее создадим типы для хранения сообщений в микроблоге и авторах блога и определим отношения между типами, в соответствии с ранее предложенной схемой, которую **немного изменим**, с целью упрощения интеграции к данному приложению базы данных.

```
const BlogType = new GraphQLObjectType({
  name: "Blog",
  fields: () => ({
    id: { type: GraphQLID },
    bid: { type: GraphQLID },
```

```

        annotation: { type: GraphQLString },
        blogauthor: {
          type: BlogAuthorType,
          resolve(parent, args) {
            return _.find(authors, { id: parent.authorId });
          }
        }
      })
    });

const BlogAuthorType = new GraphQLObjectType({
  name: "BlogAuthor",
  fields: () => ({
    id: { type: GraphQLID },
    username: { type: GraphQLString },
    name: { type: GraphQLString },
    surname: { type: GraphQLString },
    blog: {
      type: BlogType,
      resolve(parent, args) {
        return _.find(authors,
{authorId:parent.id});
      }
    }
  })
});

```

В данном коде были объявлены GraphQL Object Types «*BlogType*» и «*BlogAuthorType*», являющиеся функциями, которые принимают объект `name:` и `fields:`. В данной схеме определены отношения между данными объектами. Так в поле `blogauthor:` объекта *BlogType* было объявлено, что это свойство определено как *BlogAuthorType*, реализованное ниже. А объект *BlogAuthorType* определяет поле `blog:` как *BlogType*. В случае, если потребуется получить данные об авторе соответствующего блога, GraphQL, благодаря соответствующему преобразователю, сможет найти пользователя, идентификатор (`authorId:`) которого будет совпадать со значением идентификатора пользователя.

В объекте *RootQuery* необходимо добавить код, который будет определять запрос для объектов внутри корневой точки, в полном соответствии с бизнес-задачей.

```

blog: {
  type: BlogType,

```

```

    args: { id: { type: GraphQLID } },
    resolve(parent, args) {
      return _.find(blogs, { id: args.id });
    }
  },
  blogs: {
    type: new GraphQLList(BlogType),
    resolve(parent, args) {
      return blogs;
    }
  },
  authors:{
    type: BlogAuthorType,
    args: { id: { type: GraphQLID } },
    resolve(parent, args) {
      return _.find(authors, { id: args.id });
    }
  }
}

```

В продемонстрированном выше коде поля `blog:`, `blogs:` и `authors` не определены как базовые скалярные типы (`var GraphQLInt : GraphQLScalarType`; `var GraphQLFloat : GraphQLScalarType`; `var GraphQLString : GraphQLScalarType` ; и т.д.), а определены как *type*, которые ссылаются на ранее определённые «*BlogType*» и «*BlogAuthorType*». Каждое из полей содержит резольвер, который возвращает соответствующий объект, разрешая значение для поля *type*. Так резольвер поля *blogs* возвращает объект *blogs*, а для нашей задачи это означает возврат всех записей в микроблогах. Соответственно, это позволяет разрешить такой запрос, как

```

{
  blogs{
    id
    annotation
  }
}

```

В резольвере поля *authors*: извлекается конкретное значение `args` аргумента `id` из поля запроса к конечной точке. Извлечённое значение `id` используется для поиска по коллекции и в случае, если значения совпадает необходимо вернуть содержимое данного элемента коллекции. При этом используется метод `_.find()` библиотеки `lodash`. Использование, в данном случае, библиотеки `lodash` оправдано тем, что данная библиотека упрощает работу с коллекциями, так как для демонстрации



работоспособности будут использоваться подготовленные тестовые данные. Соответственно, необходимо установить библиотеку `lodash` и добавить в `schema.js` строку `const _ = require("lodash");`. Общая структура кода объекта `RootQuery` показана на рисунке 6.20.

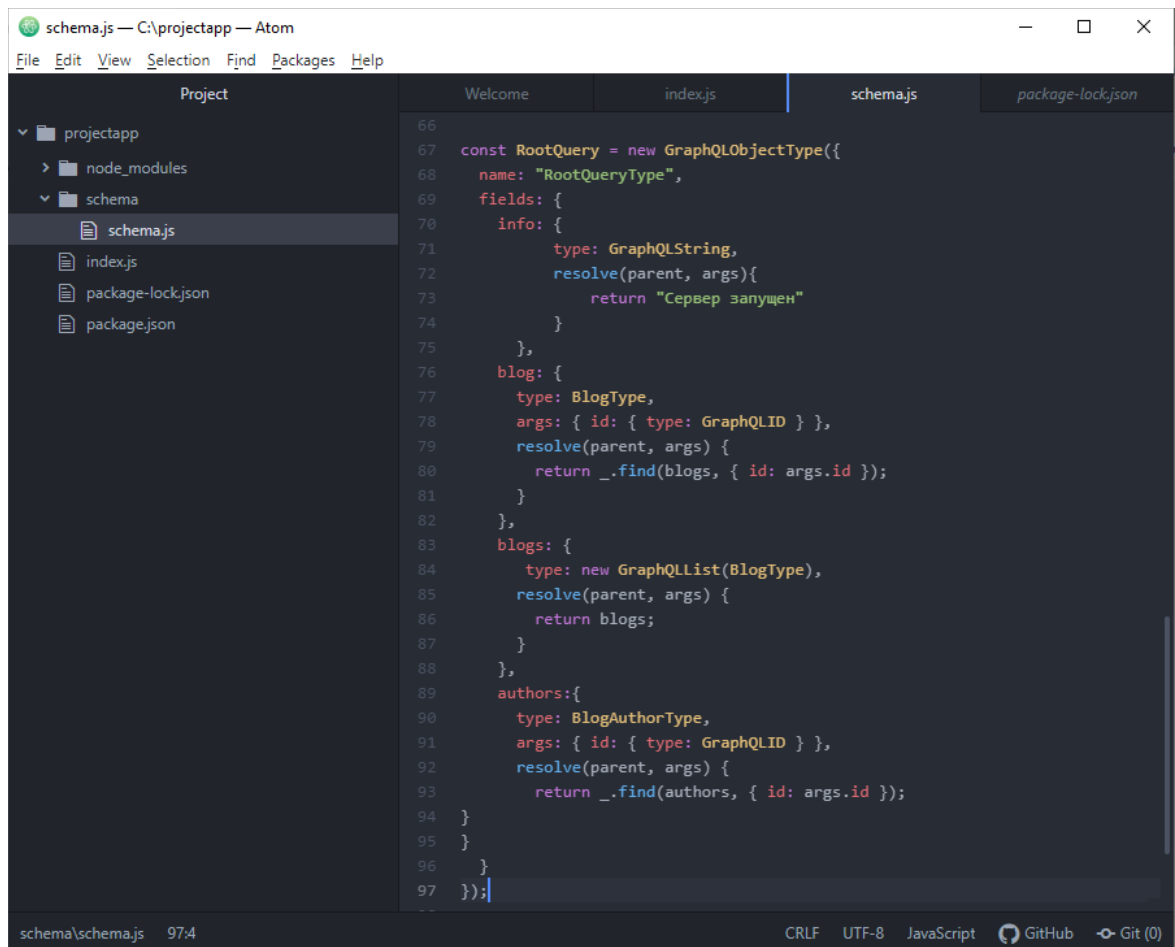


Рисунок 6.20- Общая структура кода объекта `RootQuery`

Для демонстрации работоспособности сервера GraphQL и демонстрации правильности реализации схемы, в требуемом соответствии с бизнес-задачей, необходимо подготовить набор тестовых данных. Как было сказано выше, отличительной особенностью технологии GraphQL является независимость от источника данных. Исходя из этого, в ручную, создадим и проинициализируем два массива, один для хранения записей в блогах, другой - для хранения информации об авторах микроблогов. Код представлен ниже.

```
const blogs = [
  { id: "1", bid: "73898966-e68f-46bc-be55-a4f28bc646ef", annotation: "Первая запись в микроблоге", authorId:"1"},
  { id: "2", bid: "7c97f5ee-4dc8-4538-bf69-63ee30bf0e74", annotation: "Сегодня очень тёплая погода", authorId:"2"},
]
```

```

    { id: "3", bid: "064b71d2-5807-4e59-a961-09b1366427e9", annotation: "Привет!", authorId:"b0124e22-6811-409a-90fb-6491f0a200db"}]];

```

```

const authors = [
{
  id: "1",
  username: "Ivan1",
  name: "Иван",
  surname: "Сергеев"
},
{
  id: "2",
  username: "Jax",
  name: "Андрей",
  surname: "Иванов"
},
{
  id: "b0124e22-6811-409a-90fb-6491f0a200db",
  username: "Qwerty",
  name: "Виктор",
  surname: "Андреев"
}]];

```

Изменённая структура файла schema.js показана на рисунке 6.21.

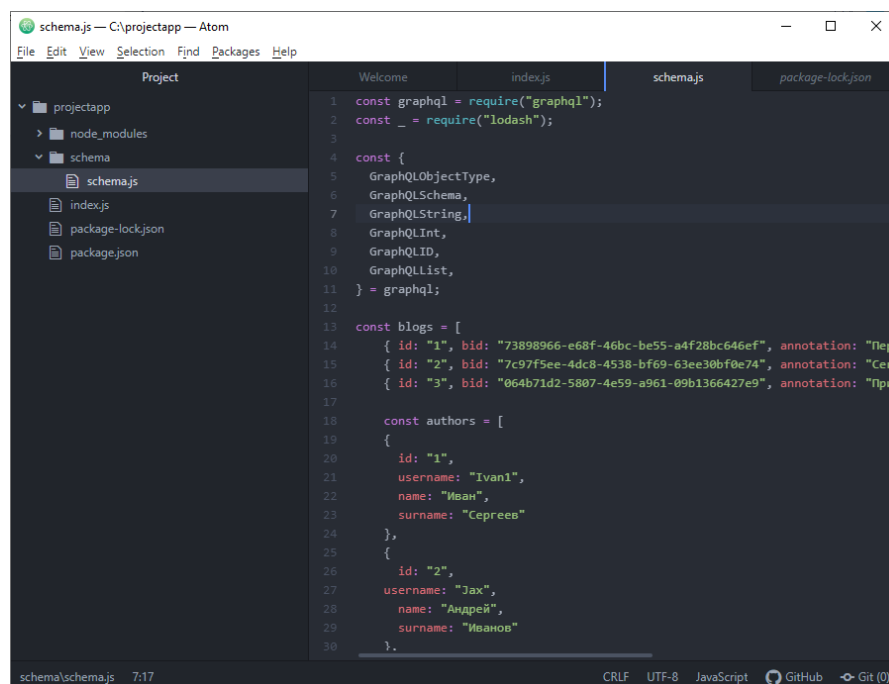


Рисунок 6.21- Структура файла schema.js после добавления соответствующих массивов

После сохранения всех изменений в файлах необходимо запустить/перезапустить сервер GraphQL API, выполнив команду *node index.js*, находясь в директории проекта. Если ошибок в программном коде нет, сервер запустится. Далее, перейдя по адресу <http://localhost:1234/graphql>, можно попасть в веб-интерфейс GraphiQL, в котором можно делать запросы. Протестируем работоспособность API сервера путём формирования запроса на извлечение всех записей во всех микроблогах, как показано на рисунке 6.22.

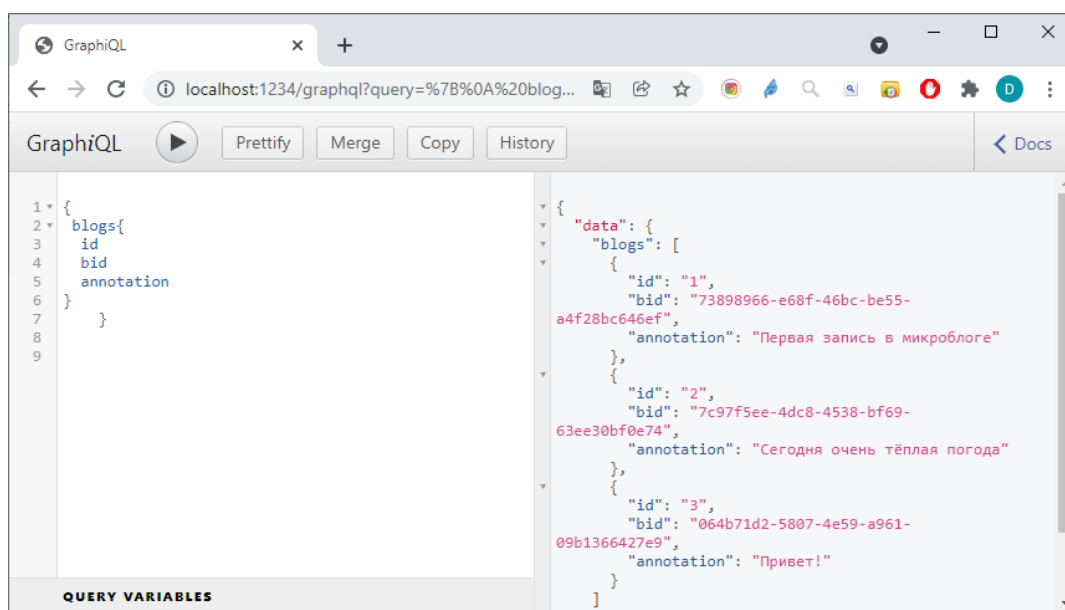


Рисунок 6.22 - Пример обработки запроса на извлечение всех записей из микроблогов

Для извлечения данных об авторе с идентификатором `b0124e22-6811-409a-90fb-6491f0a200db` необходимо выполнить запрос, показанный на рисунке 6.23.

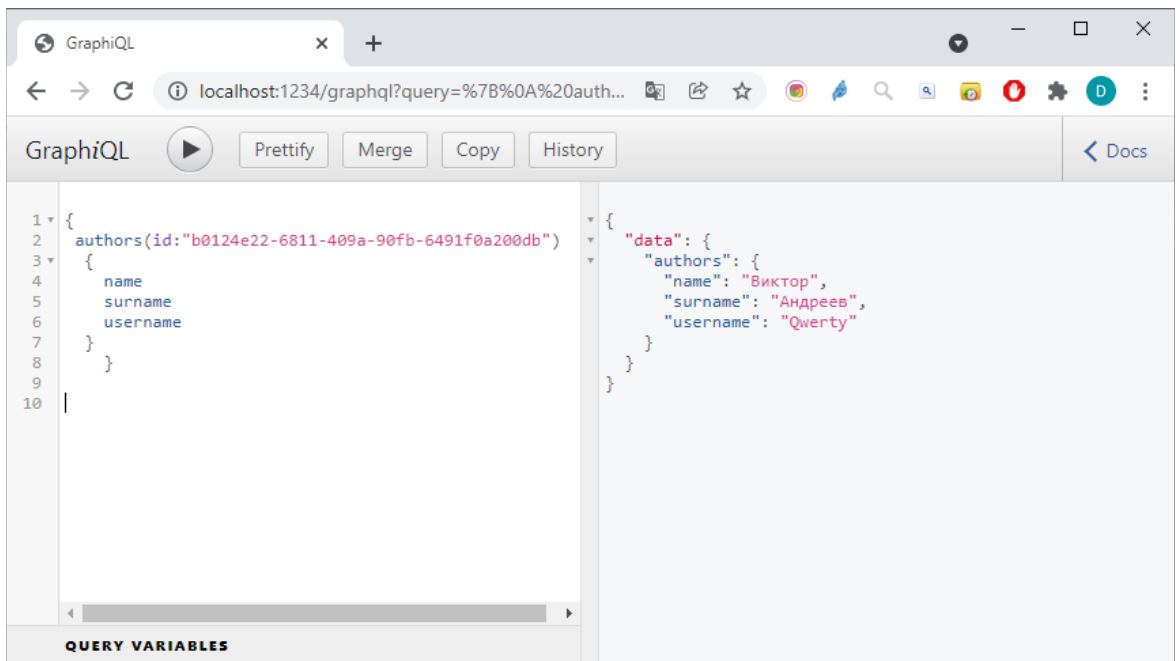


Рисунок 6.23 - Извлечение данных об авторе с уникальным идентификатором b0124e22-6811-409a-90fb-6491f0a200db

В силу того, что в схеме определена связь между объектами, запросы могут быть сложнее. Например, для того, чтобы извлечь информацию об авторе конкретной записи в микроблоге с известным идентификатором записи, необходимо сделать запрос, как показано на рисунке 6.24.

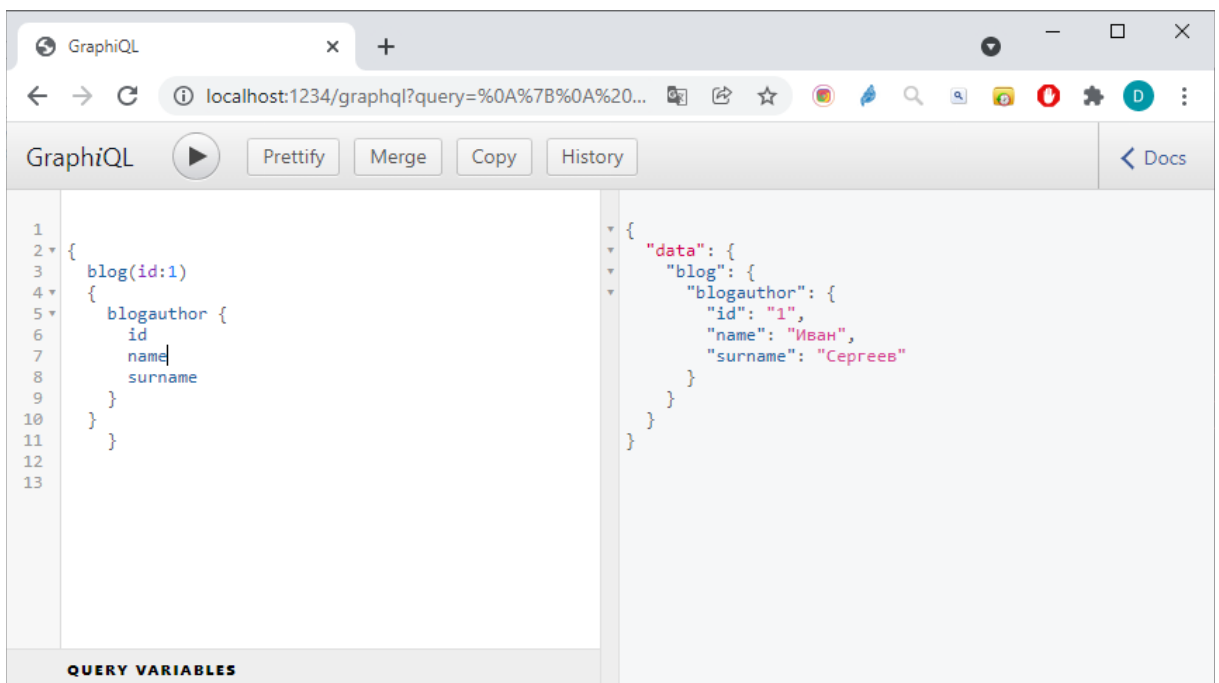


Рисунок 6.24 - Извлечение данных об авторе микроблога по известному ID записи в микроблоге

Стоит обратить внимание на особенности формирования уникальных идентификаторов, в соответствующих массивах данных. Так в данном примере, для демонстрации, уникальные идентификаторы формировались,

как в виде нумерованных последовательных записей, то есть каждый ID новой записи в микроблогах всей системы получался путём прибавления к предыдущему значению единицы, так и с использованием стандарта GUID (возможно использование UUID и похожих стандартов), например, `id: "b0124e22-6811-409a-90fb-6491f0a200db"`. Использование второго подхода в реальных системах более предпочтительно, так как это позволяет обезопасить от компрометации данных путём перебора значений ID в рамках запроса. Помимо этого, многие современные базы данных, такие как, например, MongoDB, позволяют генерировать случайным образом UUID объекты, что упрощает процесс создания уникальных идентификаторов для их использования.

Клиентом разработанного сервера GraphQL может любая сущность. В том числе, ранее рассмотренный в предыдущих практических работах Postman. Для того, чтобы осуществить query запрос на конечную точку с использованием Postman, необходимо в строке адреса конечной точки указать адрес конечной точки <http://127.0.0.1:1234/graphql>, а в нижележащем меню выбрать пункт «Body», где установить check box пункта «GraphQL» и в появившемся поле QUERY написать требуемый запрос к конечной точке, после чего нажать кнопку «SEND». На рисунке 6.25 показан пример запроса к серверу API в среде Postman.

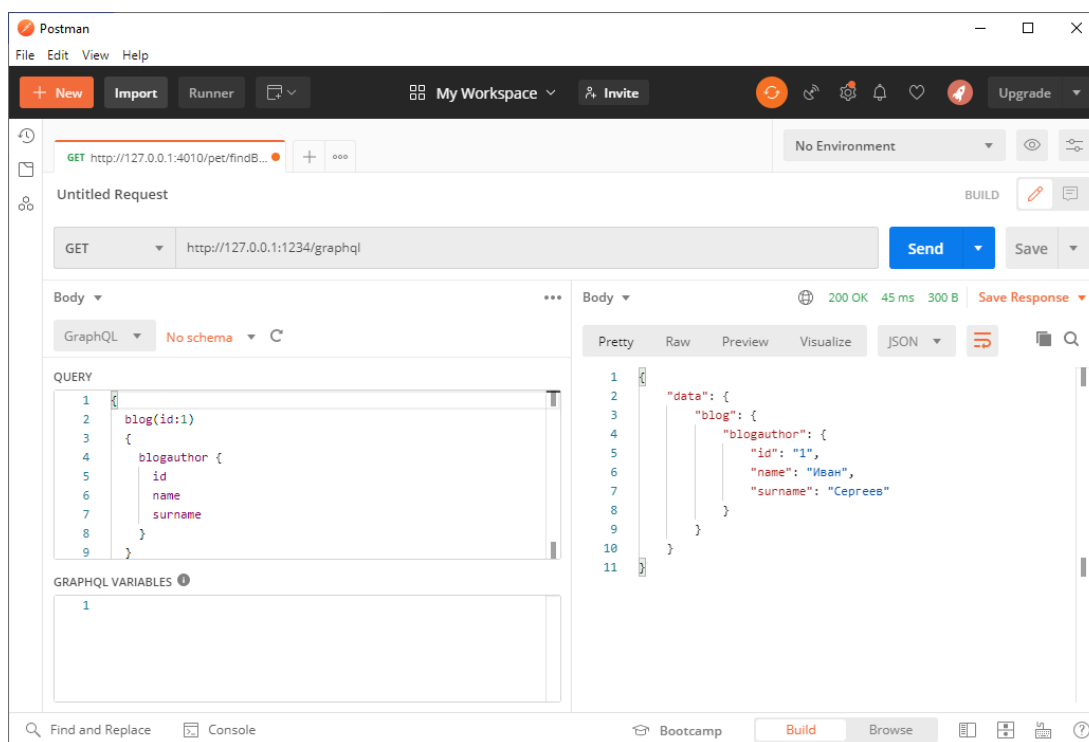


Рисунок 6.25 - Пример query операции к серверу API в среде Postman

Для полноценного сервера API явно недостаточно того, что сервер может только отдавать данные. Необходимо расширить возможности спроектированного и созданного API создавать и обновлять данные. Для этого необходимо реализовать механизм обработки mutation операций.

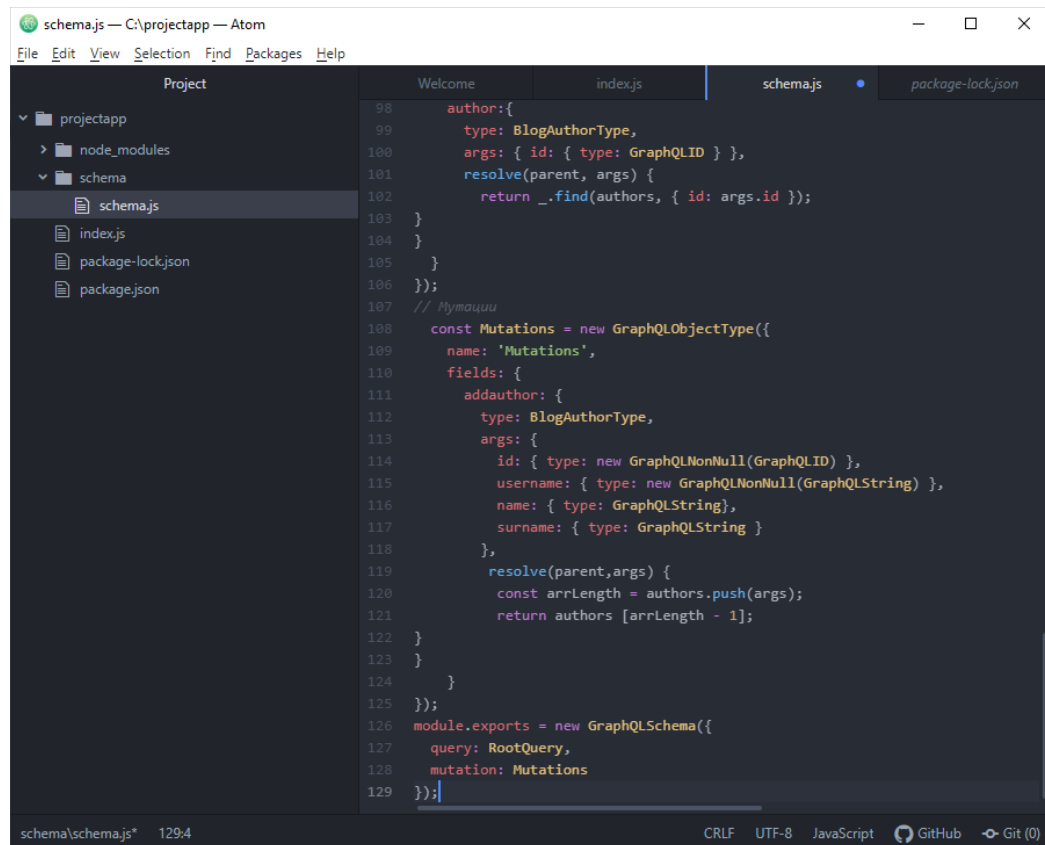
Ранее было отмечено, что GraphQL это прежде всего технология создания декларативных программных интерфейсов, которая включает в себя язык запросов и среду исполнения, которую можно использовать для изменения данных. Для реализации механизма создания, изменения, обновления данных (операции create, update, delete) используют именно мутации. По аналогии с query, необходимо создать объект GraphQLObjectType и экспортировать его в рамках экспорта схемы. Рассмотрим пример мутаций для создания нового пользователя микроблога. Следующий код демонстрирует добавление нового пользователя в массив пользователей:

```
const Mutations = new GraphQLObjectType({
  name: 'Mutations',
  fields: {
    addauthor: {
      type: BlogAuthorType,
      args: {
        id: { type: new GraphQLNonNull(GraphQLID)
},
        username: { type: new
GraphQLNonNull(GraphQLString) },
        name: { type: GraphQLString},
        surname: { type: GraphQLString }
      },
      resolve(parent,args) {
        const arrLength = authors.push(args);
        return authors [arrLength - 1];
      }
    }
  }
});
```

Стоит обратить внимание, на определение полей *id:* и *username:* . Согласно ранее определённой схеме поля уникального идентификатора зарегистрированного пользователя и его уникальный никнейм обязательны, соответственно, данные поля не могут быть *null*. Для этого они определяются через GraphQLNonNull. Это даёт гарантию, что в нашем массиве (или в базе данных ит.д.) при создании пользователя это поле будет обязательно определено, что важно для обеспечения функционала, для которых эти данные критичны.

Так как мутации являются частью GraphQL схемы, необходимо изменить программный код импорта схемы, добавив *mutation: Mutations*.

Структура данной части программного кода (файл schema.js) показана на рисунке 6.26.



```
98   author: {
99     type: BlogAuthorType,
100    args: { id: { type: GraphQLID } },
101    resolve(parent, args) {
102      return _.find(authors, { id: args.id });
103    }
104  }
105 }
106 });
107 // Мутации
108 const Mutations = new GraphQLObjectType({
109   name: 'Mutations',
110   fields: {
111     addauthor: {
112       type: BlogAuthorType,
113       args: {
114         id: { type: new GraphQLNonNull(GraphQLID) },
115         username: { type: new GraphQLNonNull(GraphQLString) },
116         name: { type: GraphQLString },
117         surname: { type: GraphQLString }
118       },
119       resolve(parent, args) {
120         const arrLength = authors.push(args);
121         return authors[arrLength - 1];
122       }
123     }
124   }
125 });
126 module.exports = new GraphQLSchema({
127   query: RootQuery,
128   mutation: Mutations
129 });
```

Рисунок 6.26 - Структура кода мутации и экспорта схемы

После сохранения изменения в файле schema.js и перезапуска/запуска сервера API, можно протестировать mutation операцию для добавления нового пользователя, как показано на рисунке 6.27.

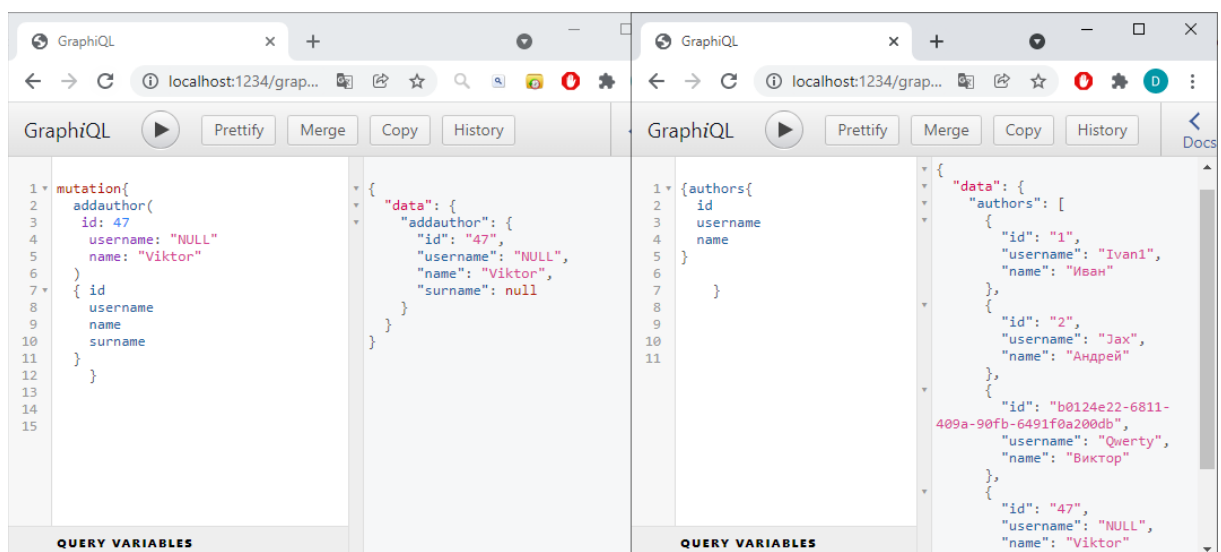


Рисунок 6.27 - Запрос на создание нового пользователя и query запрос на извлечение всех пользователей

Стоит обратить внимание, что поле `username` добавленного поля определена как `String` и `NULL`, в данном контексте, является именно строкой, а не значением `NULL`. В качестве эксперимента попробуйте создать запрос на добавление нового пользователя без указания полей `id` или `username` в теле `mutation` запроса и посмотрите на поведение сервера API и о том, какую информацию он вернет клиенту GraphQL.

### **Задание на самостоятельную работу**

Используя теоретические сведения из данной практической работы, открытые интернет-источники, официальную документацию по GraphQL необходимо, с использованием SDL создать схему, реализовать сервер и клиента GraphQL для следующих бизнес-задач (по выбору):

**1. Создание приложения для хранения списка книг в библиотеке.** Схема должна определять требуемые типы данных с полями, возвращающими определённые данные. Должны быть определены поля: `name` (название книги), `genre` (жанр книги), `id` (уникальный идентификатор книги), `author` (ФИО автора книги). Дополнительные поля и соответствующие типы, если они будут нужны для решения данной задачи, определяются самостоятельно.

**2. Создание приложения для хранения информации об автомобилях.** Схема должна реализовывать возможность хранения краткой информации об автомобиле (поле `title`), уникальном идентификаторе автомобиля (поле `id`), информации о бренде автомобиля (поле `brand`), текущей цены автомобиля (поле `price`), возраста автомобиля (поле `age`).

**3. Создание приложения для хранения списка покупок.** Схема должна описывать следующий полный набор данных: уникальный идентификатор товара для покупки (поле `id`), краткое именование товара (поле `text`), количество требуемого товара (поле `qty`), поле флага покупки, определяемое через `true/false` (поле `completed`), а также уникальный идентификатор пользователя. Если требуются дополнительные поля, то они определяются обучающимися самостоятельно.

Клиентом API может быть любая сущность, например, веб-клиент (приложение для браузера), мобильный клиент (приложение для android), десктоп приложение для операционной системы Windows и т.д. Используемый стек технологий не ограничен, но, желательно, использовать продемонстрируемую выше связку.