|  |
| --- |
|  |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования  **"МИРЭА - Российский технологический университет"**  **РТУ МИРЭА** |
| Институт информационных технологий (ИТ) |
| Кафедра инструментального и прикладного программного обеспечения |

|  |  |
| --- | --- |
| **ОТЧЕТ**  **ПО ПРАКТИЧЕСКОЙ РАБОТЕ №6** | |
| **по дисциплине** |  |
| **«Архитектура клиент-серверных приложений»**  **Тема: «Введение в GraphQL»** | |
| Выполнил студент группы: ИКБО-20-21 | Хитров Н.С. |
| Принял преподаватель кафедры ИиППО | Волков М.Ю. |

|  |  |  |
| --- | --- | --- |
| Практические работы выполнены | « » 2023 г. |  |
| «Зачтено» | « » 2023 г. |  |

Москва

2023

ОГЛАВЛЕНИЕ

[Цель работы 3](#_Toc148123595)

[Теоретическое введение 3](#_Toc148123596)

[Постановка задачи 3](#_Toc148123597)

[Программный код 7](#_Toc148123598)

[Результаты работы 12](#_Toc148123599)

[Вывод 13](#_Toc148123600)

# 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.

# Теоретическое введение

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 решение 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 определения с сайта <https://graphql.org/> 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 современых 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 архитектруным 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).

# Постановка задачи

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 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.

# Программный код

На листинге 1 представлена реализация операций получения данных с использованием GraphQL.

Листинг 1 – schema.js

const {  
 GraphQLObjectType,  
 GraphQLSchema,  
 ***GraphQLID***,  
 ***GraphQLString***,  
 ***GraphQLInt***,  
 ***GraphQLBoolean***,  
 GraphQLList,  
 GraphQLNonNull  
} = ***require***('graphql');  
const ***\_*** = ***require***('lodash');  
  
const PurchaseType = new GraphQLObjectType({  
 name: 'Purchase',  
 fields: () => ({  
 id: {type: ***GraphQLID***},  
 text: {type: ***GraphQLString***},  
 qty: {type: ***GraphQLInt***},  
 completed: {type: ***GraphQLBoolean***},  
 userId: {type: ***GraphQLID***}  
 })  
});  
  
const UserType = new GraphQLObjectType({  
 name: 'User',  
 fields: () => ({  
 id: {type: ***GraphQLID***},  
 name: {type: ***GraphQLString***},  
 email: {type: ***GraphQLString***},  
 password: {type: GraphQLString}  
 })  
});

const RootQuery = new GraphQLObjectType({ // методы GET всего, что есть  
 name: 'RootQueryType',  
 fields: {  
 getAllUsers: {  
 type: new GraphQLList(UserType),  
 resolve(parent, args) {  
 return users;  
 }  
 },  
 getAllPurchases: {  
 type: new GraphQLList(PurchaseType),  
 resolve(parent, args) {  
 return purchases;  
 }  
 }}}

На листинге 2 представлен пример операций добавления (мутации) с использованием GraphQL.

Листинг 2 – schema.js

const Mutation = new GraphQLObjectType({  
 name: 'Mutation',  
 fields: {  
 createPurchase: { // POST запрос создания покупки  
 type: PurchaseType,  
 args: {  
 text: {type: new GraphQLNonNull(***GraphQLString***)},  
 qty: {type: new GraphQLNonNull(***GraphQLInt***)},  
 completed: {type: new GraphQLNonNull(***GraphQLBoolean***)},  
 userId: {type: new GraphQLNonNull(***GraphQLID***)}  
 },  
 resolve(parent, args) {  
 purchases.push({id: purchases.length + 1, ...args});  
 purchases[purchases.length - 1].userId = parseInt(purchases[purchases.length - 1].userId);  
 return purchases[purchases.length - 1];  
 }  
 }  
 }  
});

На листинге 3 представлен главный файл проекта index.js, в котором мы определяем используемые инструменты.

Листинг 3 – index.js

const express = ***require***('express'); // работаем с экспрессом  
const { graphqlHTTP } = ***require***('express-graphql'); // графкуэл  
const app = express(); // создаем апи на экспрессе  
const cors = ***require***('cors');  
const port = 8080;  
const ***schema*** = ***require***('./schema/schema');  
  
app.use(cors());  
app.use( // добавляем графкуэл в приложение  
 '/graphql',  
 graphqlHTTP(  
 {  
 schema: ***schema***,  
 graphiql: true // тестовый УИ для графкуэл  
 }  
 ),  
);  
app.listen(port);

На листинге 4 представлен код файла разметки клиентской части приложения.

Листинг 4 – client.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Покупки</title>  
</head>  
<body>  
<h1>Приложение для хранения списков покупок</h1>  
  
<div>  
 <label for="purchaseId">ID покупки: </label>  
 <input type="text" id="purchaseId">  
</div>  
  
<div>  
 <label for="userId">ID пользователя: </label>  
 <input type="text" id="userId">  
</div>  
  
<div>  
 <label for="newPurchaseText">Текст новой покупки: </label>  
 <input type="text" id="newPurchaseText">  
</div>  
  
<div>  
 <label for="newPurchaseQty">Количество покупки: </label>  
 <input type="text" id="newPurchaseQty">  
</div>  
  
<div>  
 <label for="newPurchaseCompleted">Завершена ли покупка?</label>  
 <input type="checkbox" id="newPurchaseCompleted">  
</div>  
  
<button onclick="getPurchaseById()">Получить покупку по ID</button>  
<button onclick="getPurchasesByUserId()">Получить покупки по ID пользователя</button>  
<button onclick="getUserById()">Получить пользователя по ID</button>  
<button onclick="createPurchase()">Создать покупку</button>  
<button onclick="getAllUsers()">Получить всех пользователей</button>  
<button onclick="getAllPurchases()">Получить все покупки</button>  
  
<div id="output">  
  
</div>

На листинге 5 представлен код скрипта клиентской части приложения, определяющий запросы GraphQL.

Листинг 5 – client.html

<script>  
 const ***apiUrl*** = 'http://127.0.0.1:8080/graphql';  
  
 // в конце каждого метода вызываем, чтобы получить ответ и вставить его в элемент на страничке  
 function sendGraphQLRequest(query, variables = {}) {  
 fetch(***apiUrl***, {  
 method: 'POST',  
 headers: {  
 'Content-Type': 'application/json',  
 },  
 body: ***JSON***.stringify({  
 query,  
 variables,  
 }),  
 })  
 .then(response => response.json())  
 .then(data => {  
 ***document***.getElementById('output').innerHTML = ***JSON***.stringify(data, null, 2);  
 })  
 .catch(error => ***console***.error('Error:', error));  
 }  
  
 function getPurchaseById() {  
 const purchaseId = ***document***.getElementById('purchaseId').value;  
  
 // обращаемся к методам из graphql схемы (schema)  
 const query = `  
 query ($id: ID!) {  
 purchaseById(id: $id) {  
 id  
 text  
 qty  
 completed  
 userId  
 }  
 }  
 `;  
  
 const variables = {  
 "id": purchaseId  
 };  
  
 sendGraphQLRequest(query, variables);  
 }  
  
 function getPurchasesByUserId() {  
 const userId = ***document***.getElementById('userId').value;  
  
 const query = `  
 query ($userId: ID!) {  
 purchasesByUserId(userId: $userId) {  
 id  
 text  
 qty  
 completed  
 userId  
 }  
 }  
 `;  
  
 const variables = {  
 "userId": userId  
 };  
  
 sendGraphQLRequest(query, variables);  
 }  
  
 function getUserById() {  
 const userId = ***document***.getElementById('userId').value;  
  
 const query = `  
 query ($id: ID!) {  
 userById(id: $id) {  
 id  
 name  
 email  
 password  
 }  
 }  
 `;  
  
 const variables = {  
 "id": userId  
 };  
  
 sendGraphQLRequest(query, variables);  
 }  
  
 function createPurchase() {  
 const text = ***document***.getElementById('newPurchaseText').value; // достаем из инпутов  
 const qty = parseInt(***document***.getElementById('newPurchaseQty').value);  
 const completed = ***document***.getElementById('newPurchaseCompleted').checked;  
 const userId = ***document***.getElementById('userId').value;

# Результаты работы

Чтобы начать работу приложения необходимо сначала запустить сервер (рисунок 2), а пользователю перейти по ссылке http://127.0.0.1:8080.

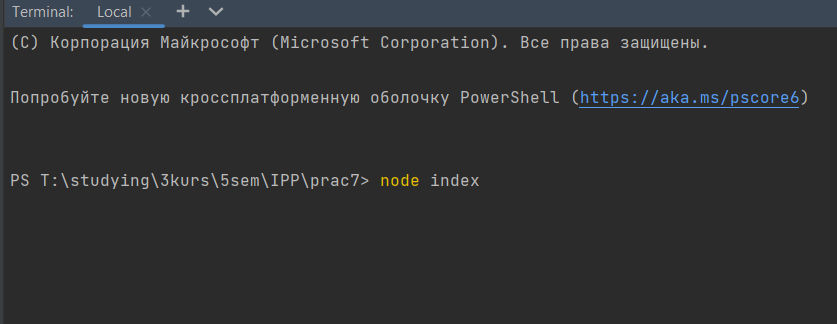


Рисунок 2 – Запуск сервера

Чтобы запустить клиентскую часть приложения, нужно открыть файл client.html.

На следующем рисунке (рисунок 3) представлен встроенный в GraphQL интерфейс воспроизведения запросов.

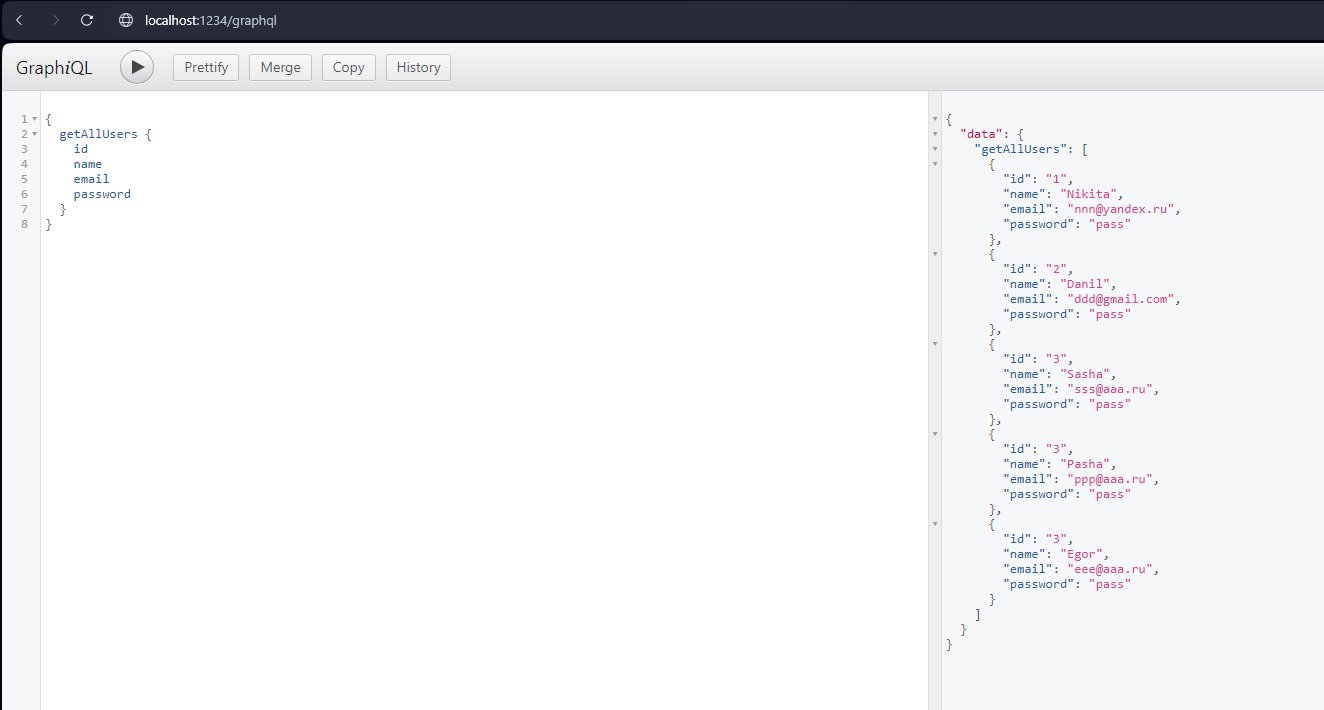


Рисунок 3 – Интерфейс GraphiQL

# Вывод

В результате выполнения данной практической работы было проведено ознакомление с набирающим популярность современным подходом к проектированию и реализации API на основе графовых моделей и с реализующей данный подход технологией на основе спецификации GraphQL.

**Список использованных источников**

1. Справочная информация Habr – URL: https://habr.com/ru/companies/ruvds/articles/414079/ (дата обращения: 16.10.2023) – текст: электронный.
2. Статья Baeldung – URL: https://www.baeldung.com/java-call-graphql-service (дата обращения: 17.10.2023) – текст: электронный.
3. Справочная информация Habr – URL: https://habr.com/ru/articles/513170/ (дата обращения: 20.10.2023) – текст: электронный.