|  |
| --- |
|  |
| 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 многослойные клиент-серверные архитектуры**»** | |
| Выполнил студент группы **ИКБО-20-21** | Сидоров С.Д. |
| Принял преподаватель кафедры ИиППО | Волков М.Ю. |

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

Москва 2023

**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/ [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 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).

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 Netfliх[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.

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 будет описаны, какие ресурсы запрашиваются. На рисунке 6.2 показан сценарий использования сервера GraphQL для взаимодействия с хранилищем данных.

Сценарий, показанный на рисунке 6.2, позволяет упростить взаимодействие с хранилищем данных, в условиях геторогенности 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 решаения 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.

**Программный код**

В листинге 1 продемонстрирован код файла schema.js описывающий работу GraphQL. В листинге 2 продемонстрирован код отвечающий за взаимодействие с пользователем.

Листинг 1 – schema.js

const graphql = require('graphql');

const \_ = require("lodash");

const {

    GraphQLObjectType,

    GraphQLID,

    GraphQLString,

    GraphQLNonNull,

    GraphQLList,

    GraphQLSchema

  } = require('graphql');

const books = [

    {

        id: "1",

        name: "Harry Potter",

        genre: "Fantasy",

        authorId:"1"

    },

    {

        id: "2",

        name: "Приключение желтого чемоданчика",

        genre: "Fantasy",

        authorId:"2"

    },

    {

        id: "3",

        name: "Война и мир",

        genre: "Роман-эпопея",

        authorId:"3"

    }

];

const authors = [

    {

        id: "1",

        name: "Джоан Роулинг"

    },

    {

        id: "2",

        name: "Прокофьев Умка"

    },

    {

        id: "3",

        name: "Лев Николаевич Толстой"

    }

];

const BookType = new GraphQLObjectType({

    name: "Book",

    fields: () => ({

        id: { type: GraphQLID },

        name: { type: GraphQLString },

        genre: { type: GraphQLString },

        author: { type: AuthorType, resolve(parent, args) {

            return \_.find(authors, { id: parent.authorId });

        }}

    })

});

const AuthorType = new GraphQLObjectType({

    name: "Author",

    fields: () => ({

        id: { type: GraphQLID },

        name: { type: GraphQLString },

    })

});

const RootQuery = new GraphQLObjectType({

    name: 'RootQueryType',

    fields: {

        book: {

            type: BookType,

            args: { id: { type: GraphQLID } },

            resolve(parent, args) {

Продолжение листинга 1

return \_.find(books, { id: args.id });

            }

        },

        books: {

            type : new graphql.GraphQLList(BookType),

            resolve(parent, args){

                return books;

            }

        },

        author:{

            type: AuthorType,

            args: { id: { type: GraphQLID } },

            resolve(parent, args) {

                return \_.find(authors, { id: args.id });

            }

        },

        authors:{

            type : new graphql.GraphQLList(AuthorType),

            resolve(parent, args){

                return authors;

            }

        }

    }

});

const Mutations = new GraphQLObjectType({

    name: 'Mutations',

    fields: {

        addauthor: {

        type: AuthorType,

        args: {

            id: { type: new GraphQLNonNull(GraphQLID) },

            name: { type: GraphQLString}

        },

        resolve(parent,args) {

                const arrLength = authors.push(args);

                return authors [arrLength - 1];

            }

        },

        addbook: {

            type: BookType,

            args: {

                id: { type: new GraphQLNonNull(GraphQLID) },

                name: { type: GraphQLString},

                genre: {type: GraphQLString},

                authorId: {type: new GraphQLNonNull(GraphQLID)}

            },

            resolve(parent,args) {

                    const arrLength = books.push(args);

                    return books[arrLength - 1];

                }

            },

        }

    }

);

module.exports = new GraphQLSchema({

    query: RootQuery,

    mutation: Mutations

});

Листинг 2 – index.js

const express = require('express');

const { graphqlHTTP } = require('express-graphql');

const app = express();

const port = 1234;

const schema = require('./schema/schema');

app.use(

    "/graphql",

    graphqlHTTP({

        schema: schema,

        graphiql: true,

    })

);

app.listen(port);

**Вывод программы**

На рисунках 1 - 3 показано тестирование созданного сервиса при помощи graphiql.

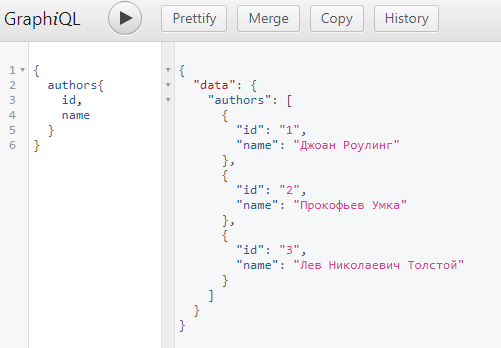


Рисунок 1– Результат получения всех авторов



Рисунок 2 – Результат получения списка книг с их авторами



Рисунок 3 – Результат создания новой покупки

**Вывод**

В результате выполнения данной практической работы было проведено ознакомление с многослойными клиент-серверными архитектурами, посмотреть разницу между ними, выявить плюсы и минусы.