|  |  |  |
| --- | --- | --- |
|  |  |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |

Институт Информационных технологий

Кафедра Инструментального и прикладного программного обеспечения

**ОТЧЕТ ПО ПРАКТИЧЕСКИМ РАБОТАМ**

**по дисциплине**

«Интерфейсы прикладного программирования»

|  |  |
| --- | --- |
| Выполнил студент группы ИКБО-01-21 | Маров Г.А |
| Принял ассистент | Зарипов Е.А. |

|  |  |  |
| --- | --- | --- |
| Практиечские работы выполнены | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2023 г.

**ОГЛАВЛЕНИЕ**

[ПРАКТИЧЕСКАЯ РАБОТА №1 3](#_Toc151461739)

[ПРАКТИЧЕСКАЯ РАБОТА №2 6](#_Toc151461740)

[ПРАКТИЧЕСКАЯ РАБОТА №3 11](#_Toc151461741)

[ПРАКТИЧЕСКАЯ РАБОТА №4 13](#_Toc151461742)

[ПРАКТИЧЕСКАЯ РАБОТА №5 19](#_Toc151461743)

[ПРАКТИЧЕСКАЯ РАБОТА №6 26](#_Toc151461744)

[ПРАКТИЧЕСКАЯ РАБОТА №7 32](#_Toc151461745)

[ПРАКТИЧЕСКАЯ РАБОТА №8 38](#_Toc151461746)

[ВЫВОД 48](#_Toc151461747)

# ПРАКТИЧЕСКАЯ РАБОТА №1

**Цель работы**

Целью данной работы является первичное знакомство с интерфейсами прикладного программирования и их применением на практике при разработке программных систем.

**Постановка задачи**

Используя любой публичный WEB API (большой список API, например, можно взять на сайте https://any-api.com/), создать приложение, демонстрирующее возможности выбранного API. Ограничений по используемым языкам, готовым библиотекам, упрощающим работу с API нет, но желательно использовать языки JAVA или Python.

**Ход работы**

На Рисунках 1 – 4 представлены результаты выполнения практической работы. Для выполнения работы было принято решение написать приложение с REST архитектурой на Java, с использованием Spring, а также с клиентской частью на React.



Рисунок 1 – Ендпоинт, принимающий запросы пользователя, запрашивающий данные у API Aviasales и возвращающий их пользователю

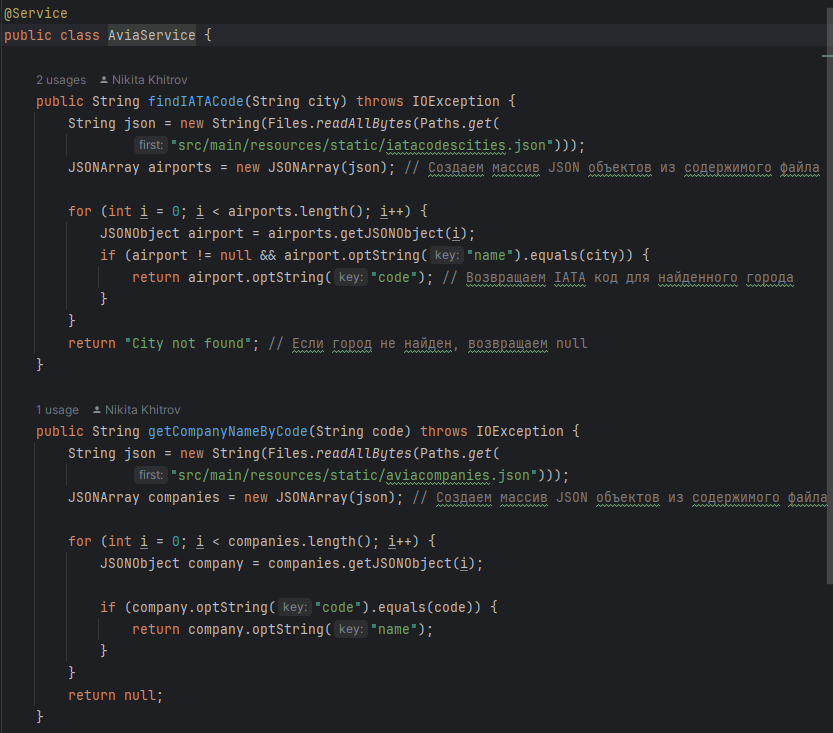


Рисунок 2 – Функции сервиса обработки «raw» данных, приходящих с API (часть 1)

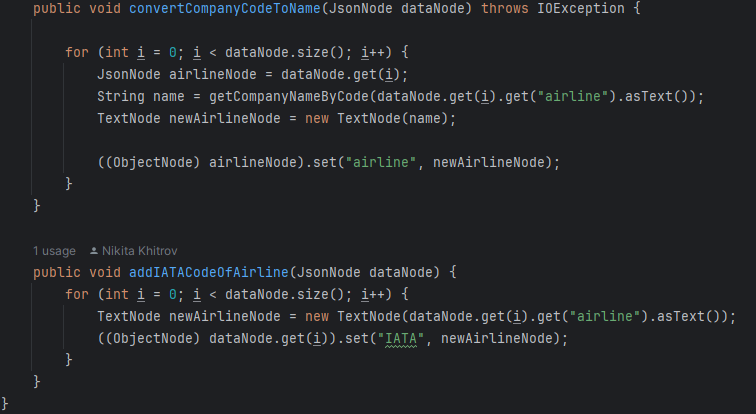


Рисунок 3 – Функции сервиса обработки «raw» данных, приходящих с API (часть 2)

Тестирование приложения представлено на Рисунке 4.

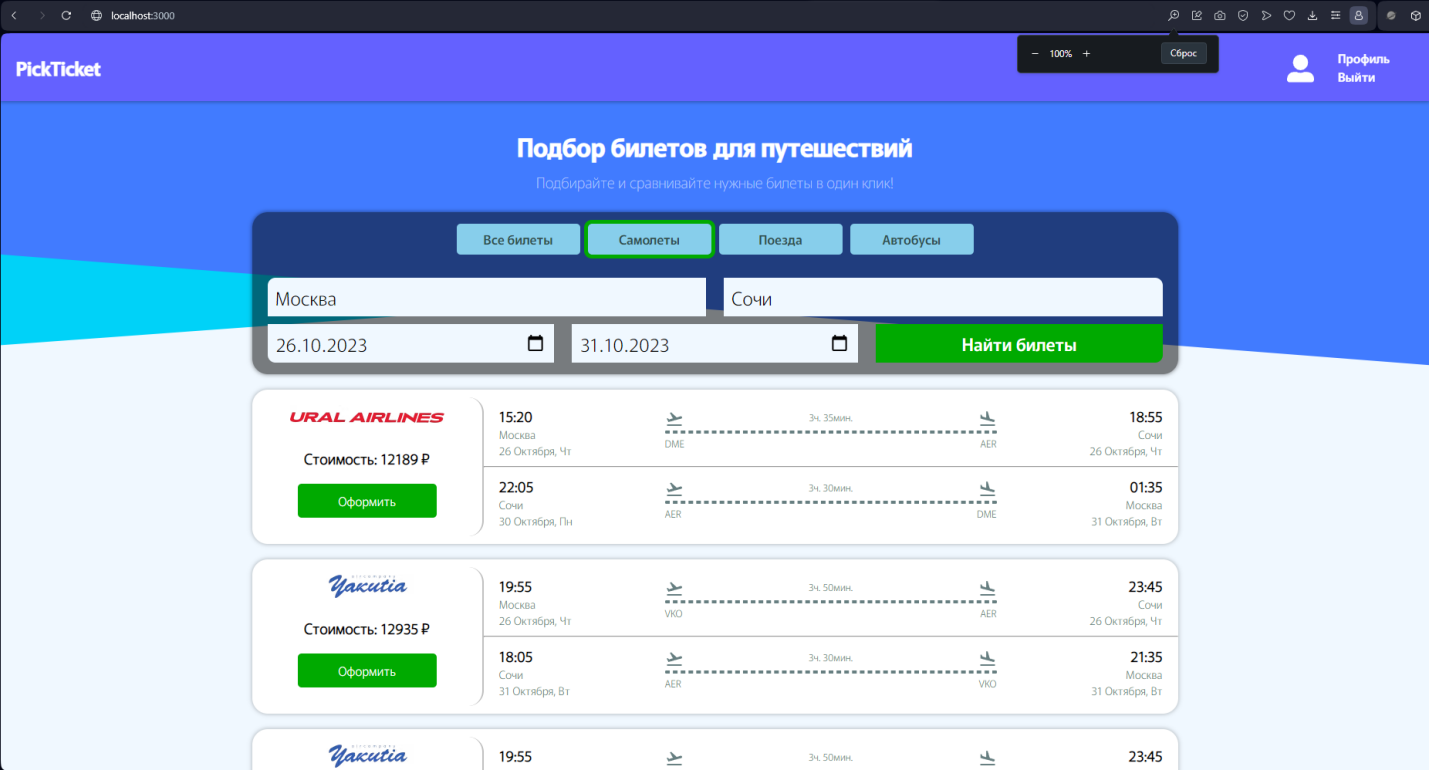


Рисунок 4 – Проверка работы сервиса на frontend части

**Вывод**

В ходе выполнения практического занятия я познакомился с интерфейсами прикладного программирования и применил их на практике. С помощью публичного API поиска билетов, я создал приложение получения информации о билетах.

# Практическая работа №2

**Цель работы**

Целью данной работы является закрепление на практике навыков работы с публичными API, в части тестирования URL-адресов конечных точек REST для получения доступа к API, а также первичное знакомство студентов с графическими интерфейсами для выполнения запросов и более подробным описанием RESTful webAPI.

**Постановка задачи**

1. Получить список поддерживаемых API Currency Layer валют.

2. Необходимо протестировать API Currency Layer, путём создания запроса об исторических курсах валют для 22 февраля 2018 года, для евро, фунтов стерлингов и иен с исходной валютой доллар США.

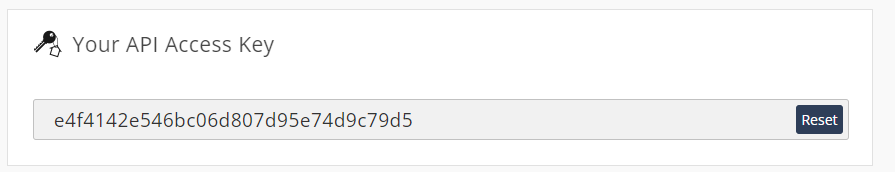
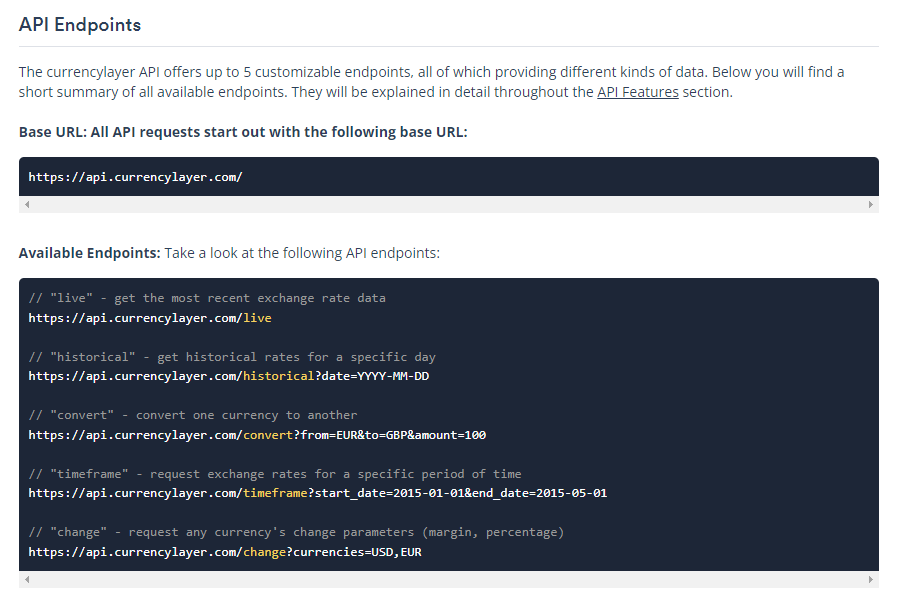
3. Получить исторические данные о курсе евро к доллару США, начиная с 25 февраля 2016 по 21 февраля 2017 года.

Дополнительным заданием требуется ознакомиться с API поставщика погоды openweathermap, получив текущие погодные данные для любого выбранного местоположения и протестировать запрос на 16 дневний прогноз для любого города (https://openweathermap.org/forecast16). Попробуйте протестировать API других поставщиков погоды (Gismeteo, Яндекс.Погоды и т. д.).

Ход работы

В данной работе необходимо приложение для тестирования API –Postman и API-ключ, которой можно получить после регистрации (рисунок 5). Для составления URL запросов необходимо использовать документацию используемого API (рисунок 6).

Создаем первый запрос при помощи документации, обращаясь к эндпоинту «/list» (рисунок 7), при этом необходимо выполнить авторизацию с помощью ключа, передаваемого в хэдере запроса. Используем запрос истории с указанием даты, валют конвертации и исходной валюты (рисунок 8). Запрос с промежутком дат показан на рисунке 9, запрос к Openweather API представлен на рисунке 10.

  
Рисунок 5 – Персональный ключ для обращения к API  
Рисунок 6 – Эндпоинты используемого API

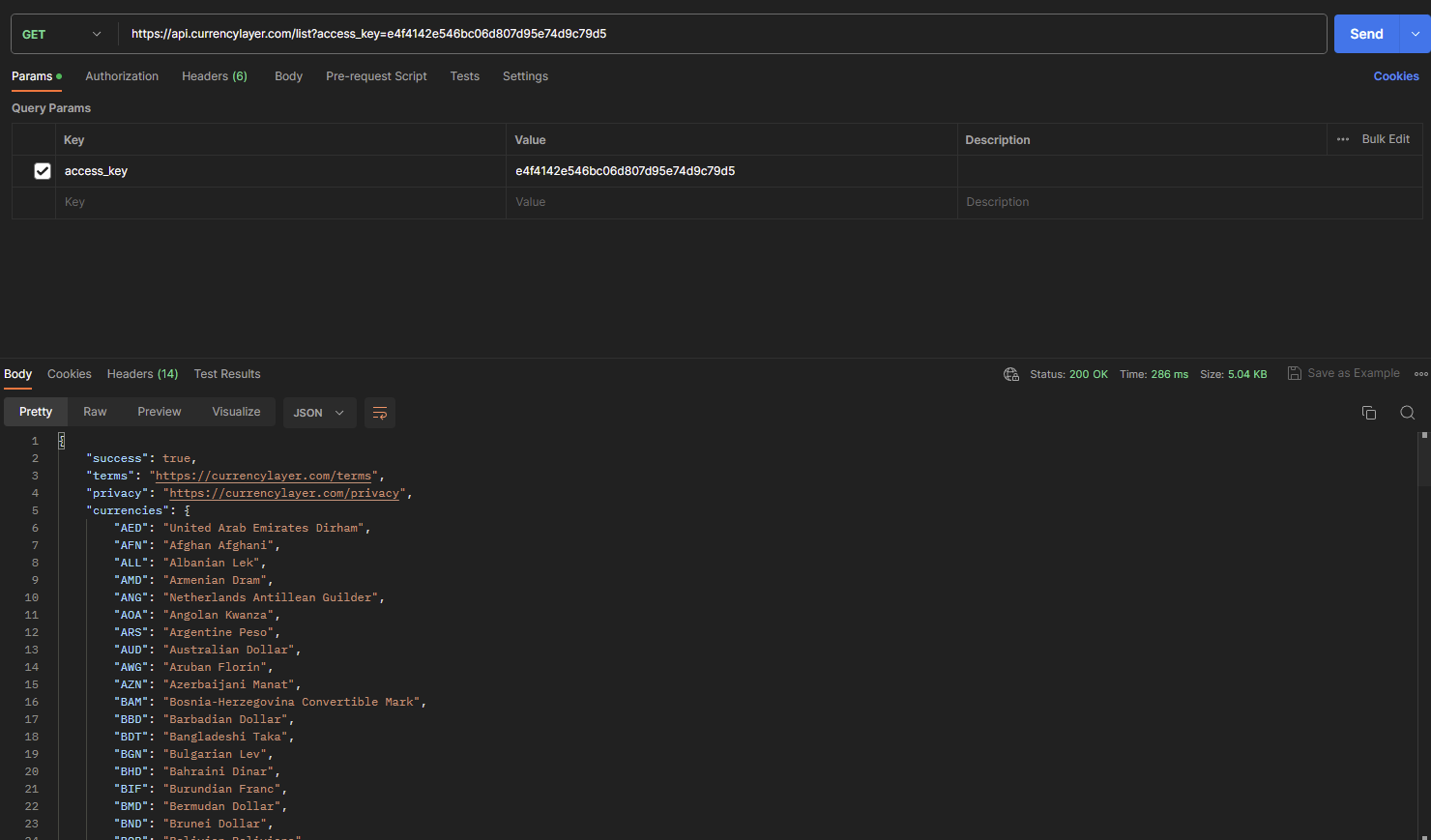
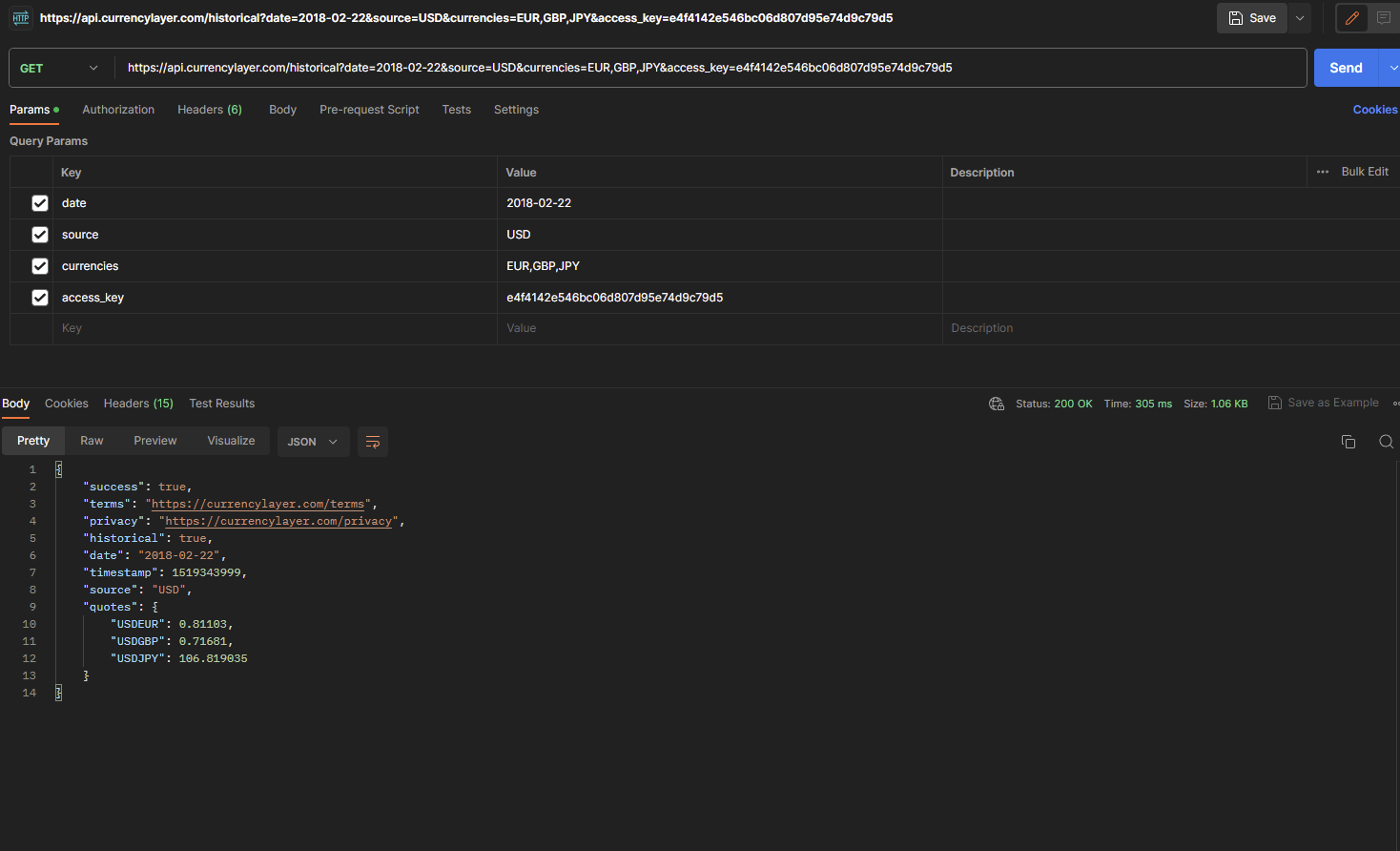


Рисунок 7 – Выполнение первого запроса

  
Рисунок 8 – Выполнение запроса об исторических курсах валют для 22 февраля 2018 года

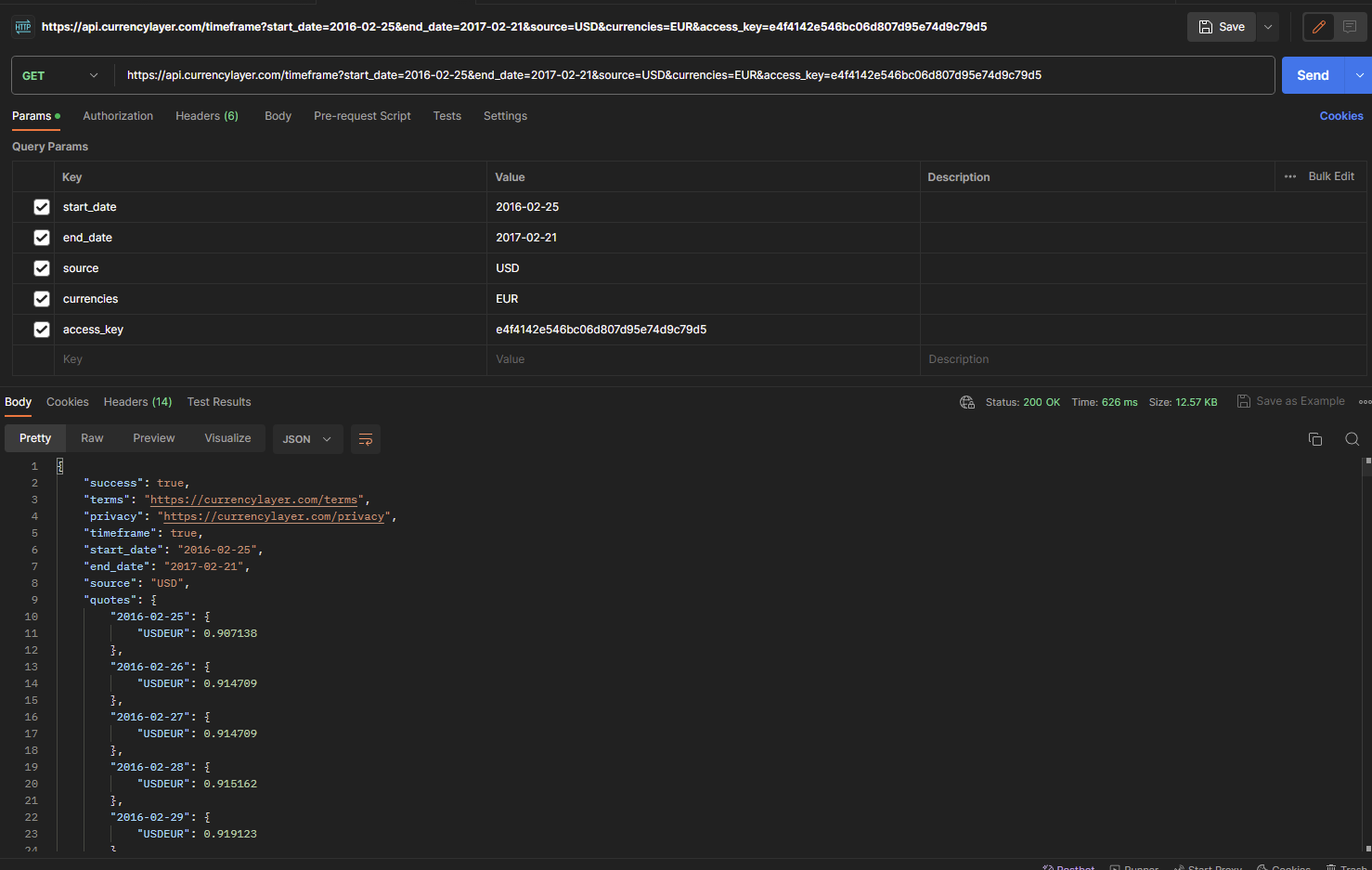


Рисунок 9 – Выполнение запроса с промежутком дат

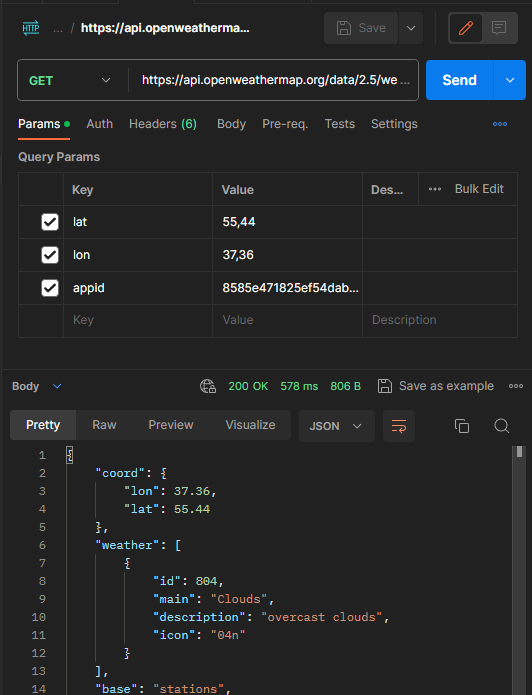


Рисунок 10 – Тестирование API OpenWeather

**Вывод**

В результате выполнения данной практической работы были закреплены навыки работы с публичными API, в части тестирования URL-адресов конечных точек REST для получения доступа к API, а также первичное знакомство студентов с графическими интерфейсами для выполнения запросов

# Практическая работа №3

**Цель работы**

Целью данной практической работы является первичное знакомство студентов с популярным инструментом командной строки для передачи файлов с синтаксисом URL – cURL.

**Постановка задачи**

Вам необходимо создать веб-страницу, на которой делается запрос к API при помощи Asynchronous Javascript And Xml (AJAX). В качестве примера рассмотрим создание веб-страницы с информацией о текущем прогнозе погоды, полученной от поставщика openweathermap.org.

**Ход работы**

На Рисунках 11-12 показаны код страницы, на которой делается запрос к API при помощи Asynchronous Javascript And Xml (AJAX) и результат отображения страницы в браузере.

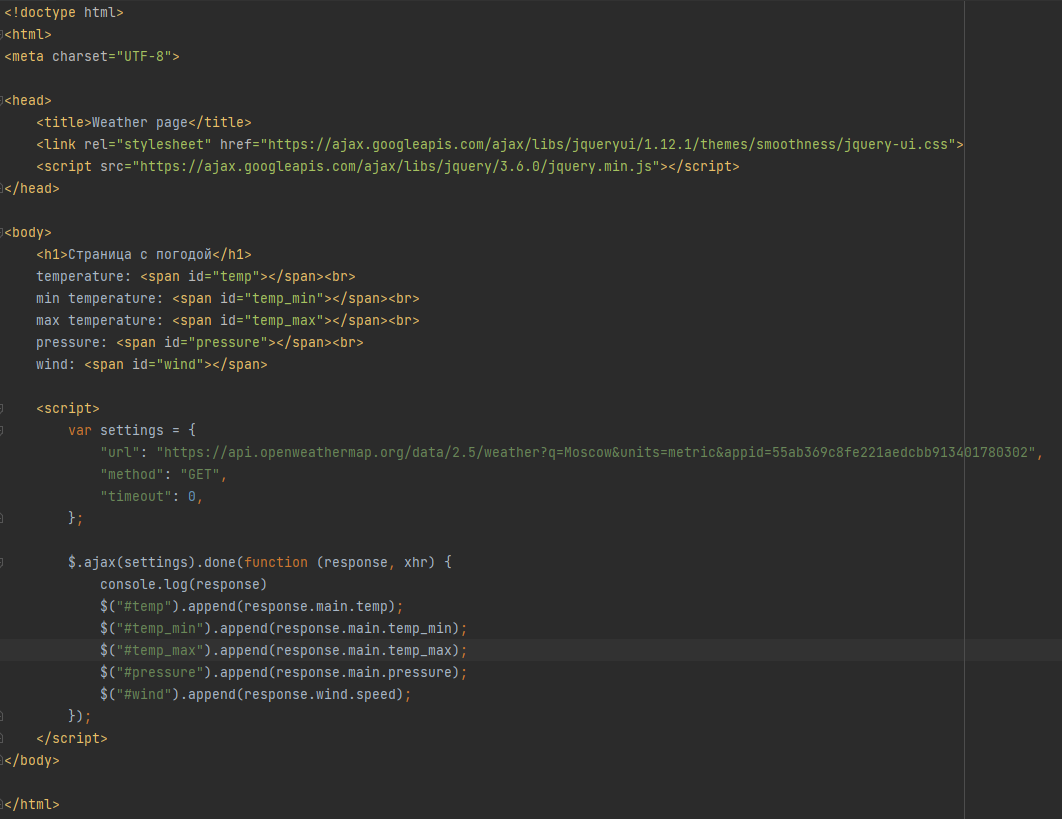


Рисунок 11 – Код страницы

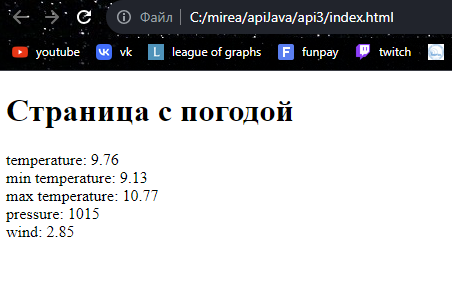


Рисунок 12 – Результат отображения страницы

**Вывод**

В ходе выполнения практического занятия я познакомился с популярным инструментом командной строки для передачи файлов с синтаксисом URL – cURL, а также создал веб-страницу, на которой делается запрос к API при помощи Asynchronous Javascript And Xml (AJAX).

# Практическая работа №4

**Цель работы**

Целью данной практической работы является знакомство с принципами REST и создание простейшего имитационного REST API.

**Постановка задачи**

1. Проверить, установлен ли NodeJS. В случае отсутствия установить с официального сайта разработчика.

2. По аналогии с вышеуказанным примером, необходимо, с помощью jsonserver, создать имитационный API для сервиса поиска книг в библиотеке. Для этого необходимо создать, как минимум три конечных точки: авторы, книги и библиотеки.

3. Протестировать работу с локальным сервисом, путём соответствующих запросов. Дать объяснение каждому запросу и возвращаемому ответу. Для этого желательно использовать curl или Postman.

Ход работы

Node.js уже был установлен данном компьютере, чтобы убедиться в этом, введём соответствующую команду в консоль (рисунок 13). Далее в соответствии с заданием создадим базу данных в виде json-файла (рисунок 14). После этого мы можем установить node.js сервер через консоль, что можно наблюдать на рисунке 15. Затем необходимо подключить созданную БД к нашему серверу, то как это было сделано представлено на рисунке 16.

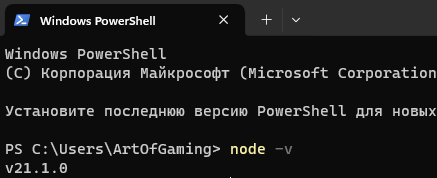


Рисунок 13 – Проверка версии установленного Node.js

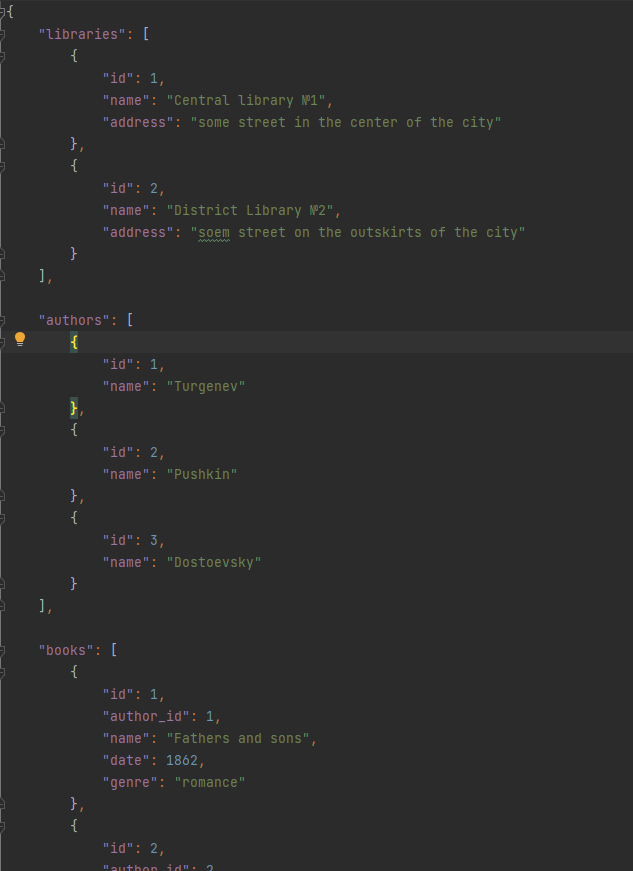


Рисунок 14 – Отрывок из созданной базы данных для будущего сервера

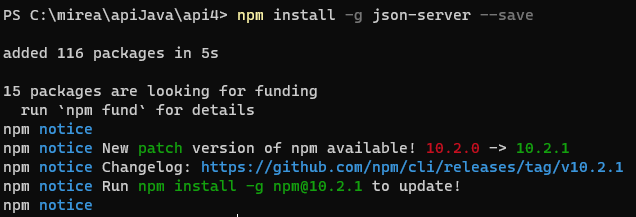


Рисунок 15 – Установка json-server

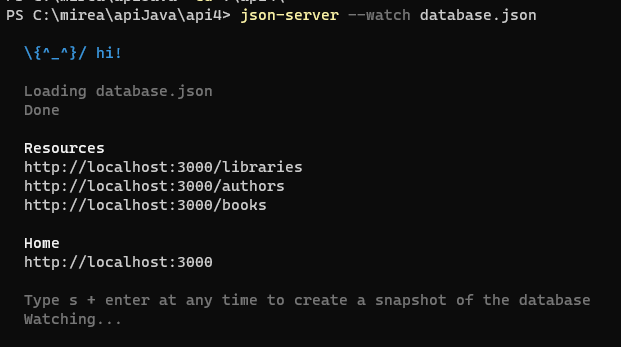


Рисунок 16 – Подключение базы данных к созданному json-server

После проведения всех вышеописанных манипуляций наш сервер полностью сконфигурирован и готов к работе. Давайте убедимся в этом и для этого выполним к нему ряд запросов через Postman (рисунки 17-21).

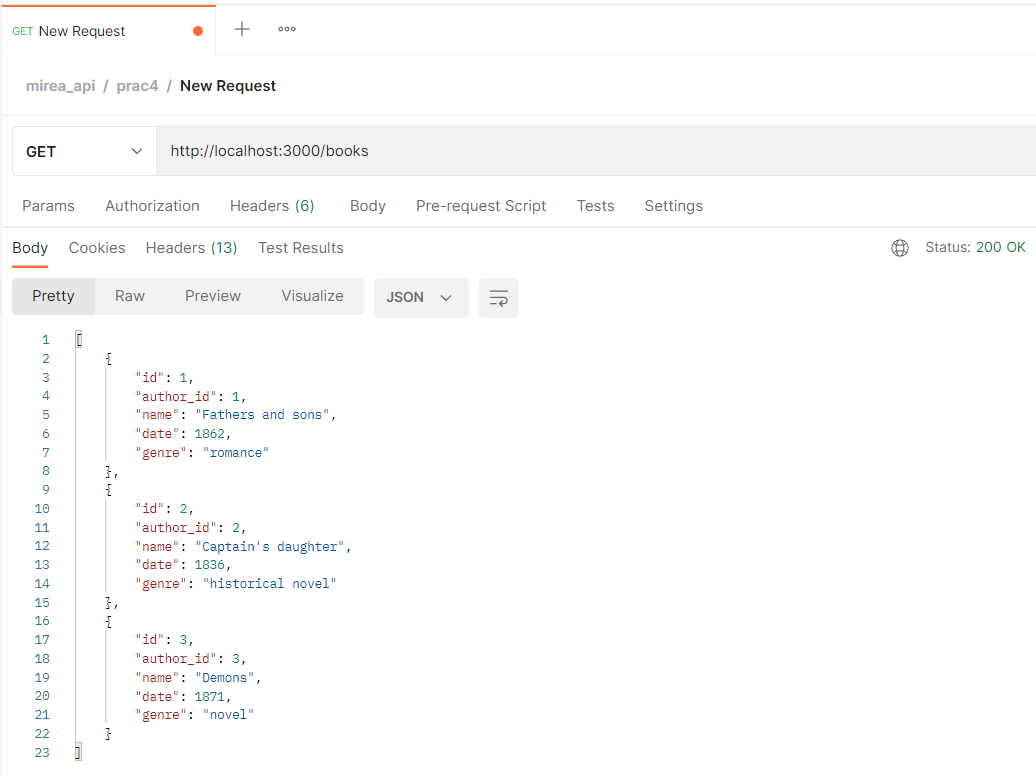


Рисунок 17 – Получение списка всех книг

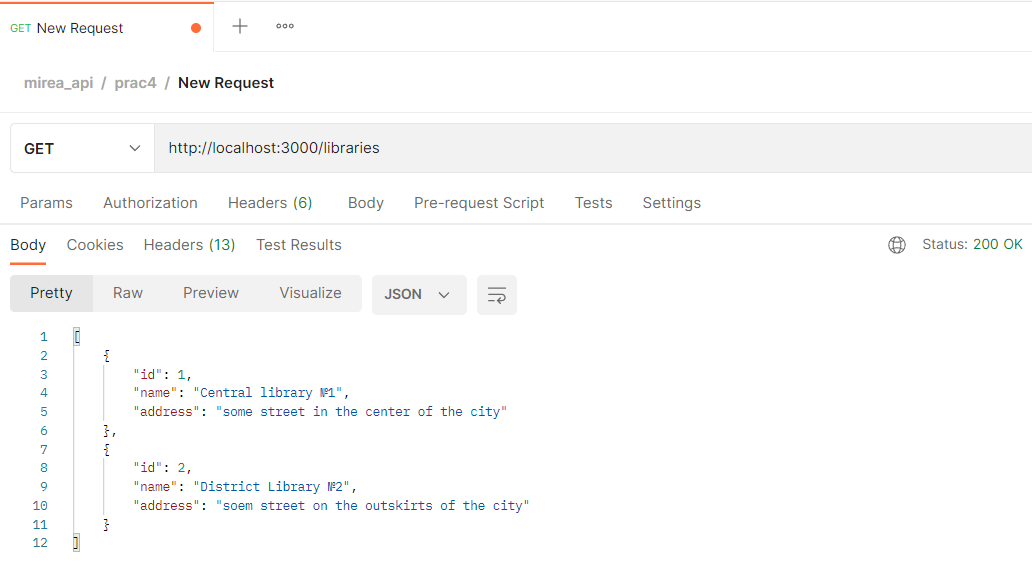


Рисунок 18 – Получение списка всех библиотек

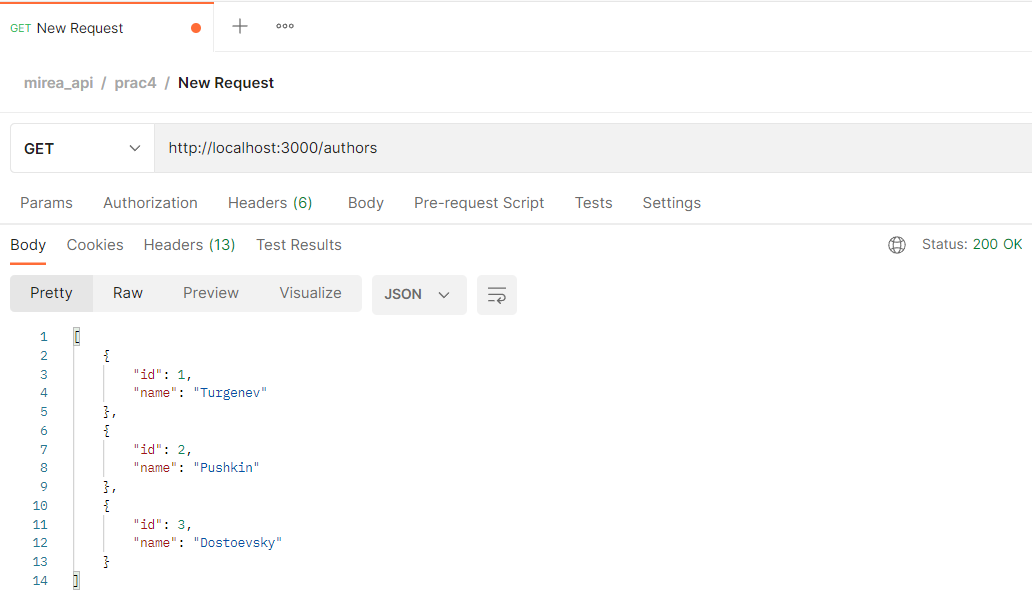


Рисунок 19 – Получение списка всех авторов

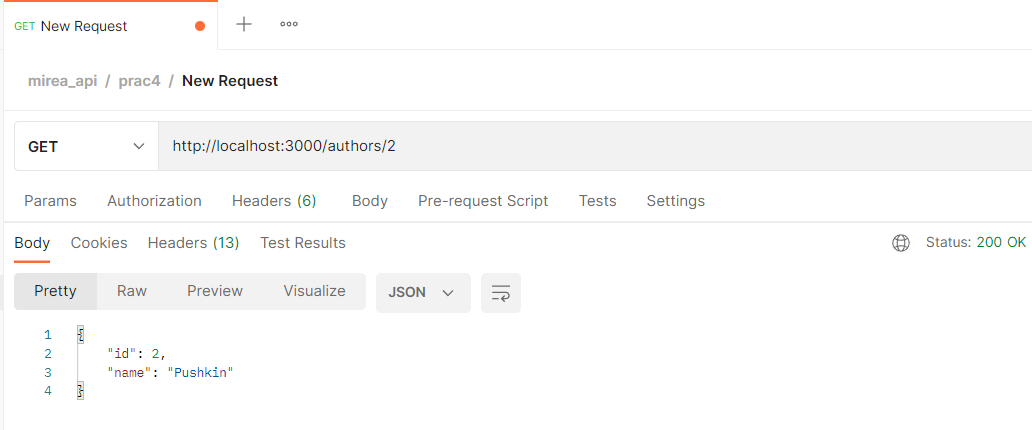


Рисунок 20 – Получение конкретного автора по id

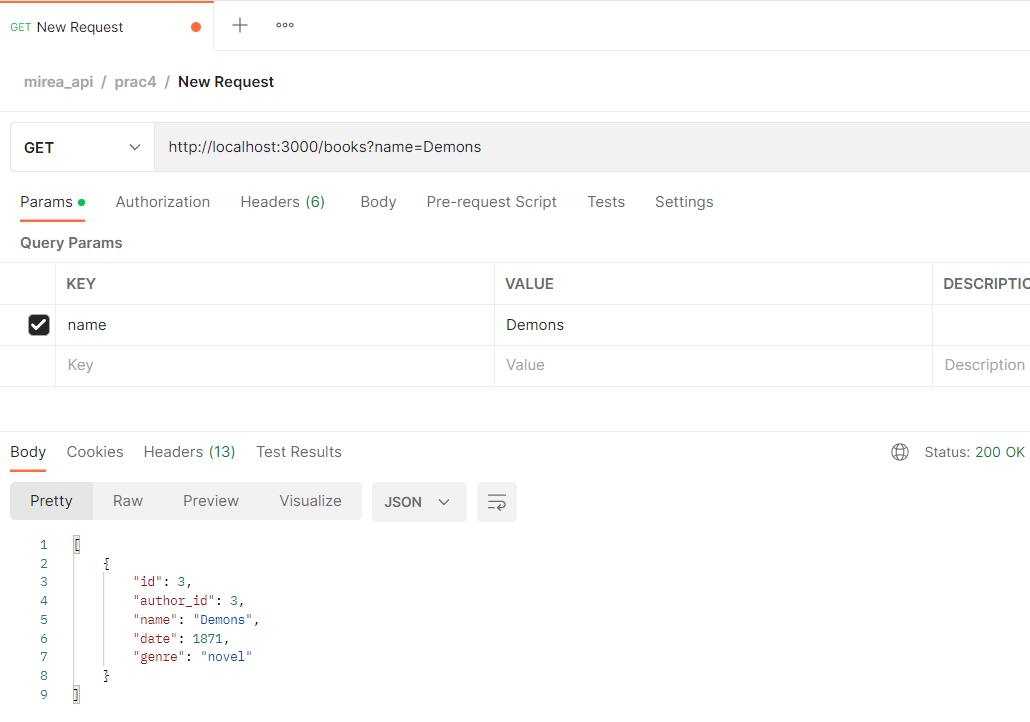


Рисунок 21 – Получение книги по названию

**Вывод**

В результате выполнения данной практической работы были получены навыки по установке и работе с Node.js. Изучен формат json, составлено несколько файлов согласно заданию, использованы вложение нескольких объектов друг в друга. Создано свое API по теме «библиотека», протестированы с помощью Postman запросы на выборку, изменение, обновление. Углублено понимание HTTP-запросов и REST-архитектуры.

# Практическая работа №5

**Цель работы**

Проектирование и реализация простейшего REST API.

**Постановка задачи**

Используя материалы практической работы и предложенный стек технологий, необходимо создать и развернуть на сервере Heroku приложение для управления списком телефонных контактов. Стоит обратить внимание на то, что используемый стек технологий очень активно развивается, а также сама облачная PaaS система Heroku может вводить определённые ограничения на поддержку используемых в данном проекте технологий, поэтому могут возникать определённые трудности со сборкой проекта, при попытке простого копирования исходников кода из данной практической работы. По согласованию с преподавателем, можно выбрать и другой стек технологий, а также и другую облачную систему.

Ход работы

На рисунке 22 показана стартовая страница cyclic.sh, на котором будет запущен сервер.

Так на листинге 1 представлен исходный код примерного приложения на node.js.

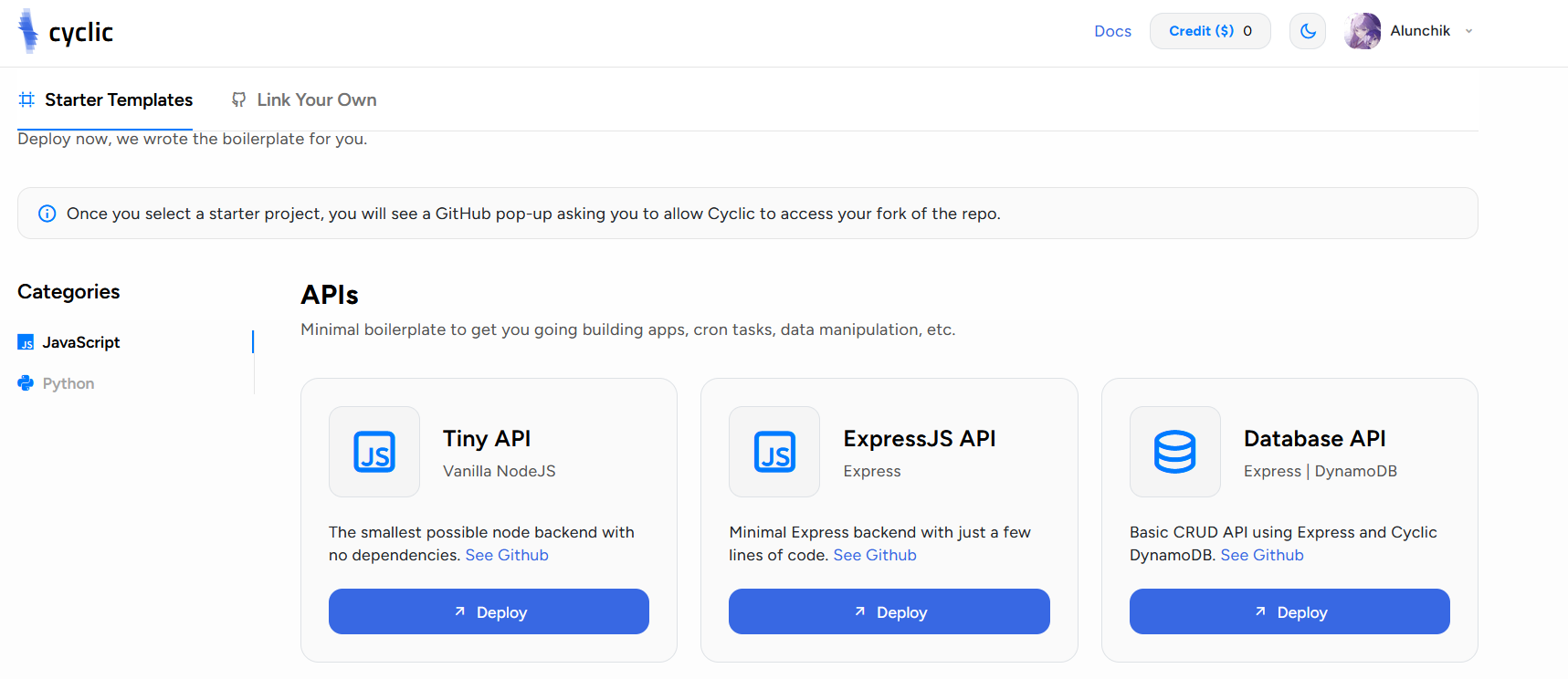


Рисунок 22 – Стартовая страница cyclic.sh

Листинг 1 – Исходный код приложения

const mongoose = require('mongoose');  
  
const contactSchema = new mongoose.Schema({  
 name: {  
 type: String,  
 required: true  
 },  
 phone: {  
 type: String,  
 required: true  
 },  
 // Добавьте другие поля, если необходимо  
});  
  
module.exports = mongoose.model('Contact', contactSchema);

const express = require('express');  
const router = express.Router();  
  
// Получение всех контактов  
router.get('/', async (req, res) => {  
 const pool = req.app.get('pool');  
 try {  
 const { rows } = await pool.query('SELECT \* FROM contacts');  
 res.json(rows);  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
// Добавление нового контакта  
router.post('/', async (req, res) => {  
 const pool = req.app.get('pool');  
 const { name, phone } = req.body;  
 try {  
 const { rows } = await pool.query('INSERT INTO contacts (name, phone) VALUES ($1, $2) RETURNING \*', [name, phone]);  
 res.status(201).json(rows[0]);  
 } catch (err) {  
 res.status(400).json({ message: err.message });  
 }  
});  
  
// Обновление контакта  
router.put('/:id', async (req, res) => {  
 const pool = req.app.get('pool');  
 const { id } = req.params;  
 const { name, phone } = req.body;  
 try {  
 const { rows } = await pool.query('UPDATE contacts SET name = $1, phone = $2 WHERE id = $3 RETURNING \*', [name, phone, id]);  
 res.json(rows[0]);  
 } catch (err) {  
 res.status(400).json({ message: err.message });  
 }  
});  
  
// Удаление контакта  
router.delete('/:id', async (req, res) => {  
 const pool = req.app.get('pool');  
 const { id } = req.params;  
 try {  
 await pool.query('DELETE FROM contacts WHERE id = $1', [id]);  
 res.json({ message: 'Контакт удален' });  
 } catch (err) {  
 res.status(500).json({ message: err.message });  
 }  
});  
  
module.exports = router;

const express = require('express');  
const { Pool } = require('pg');  
const cors = require('cors');  
const contactRoutes = require('./routes/contacts');  
  
require('dotenv').config();  
  
const app = express();  
const PORT = process.env.PORT || 5000;  
  
// Создание пула подключений к PostgreSQL  
const pool = new Pool({  
 connectionString: process.env.DATABASE\_URL  
});  
  
// Делаем pool доступным во всем приложении  
app.set('pool', pool);  
  
// Middleware  
app.use(cors());  
app.use(express.json()); // Для разбора JSON-запросов  
  
// Маршруты  
app.use('/api/contacts', contactRoutes);  
  
// Запуск сервера  
app.listen(PORT, () => {  
 console.log(`Сервер запущен на порту ${PORT}`);  
});

На данном этапе разработанное приложение уже развернуто на хостинге cyclic.sh и доступно всем пользователям сети Интернет. На рисунках 23-27 продемонстрирован его функционал.

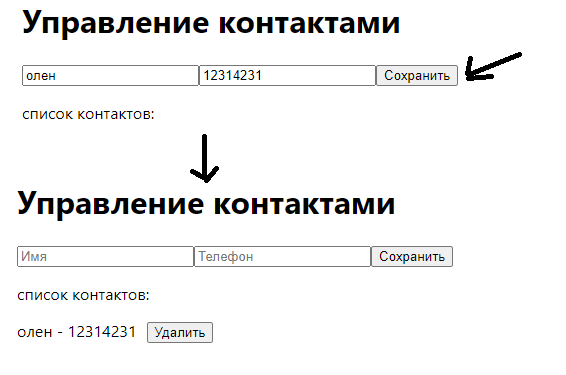


Рисунок 23 – Добавление нового контакта

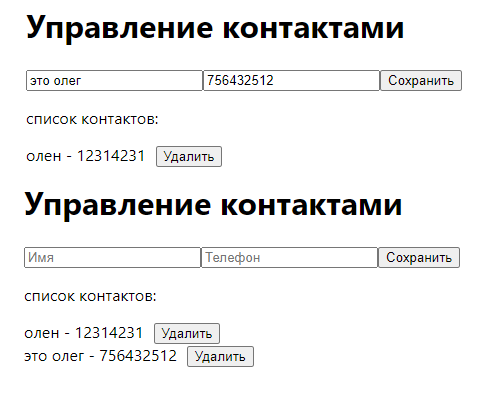


Рисунок 24 – Добавление второго контакта

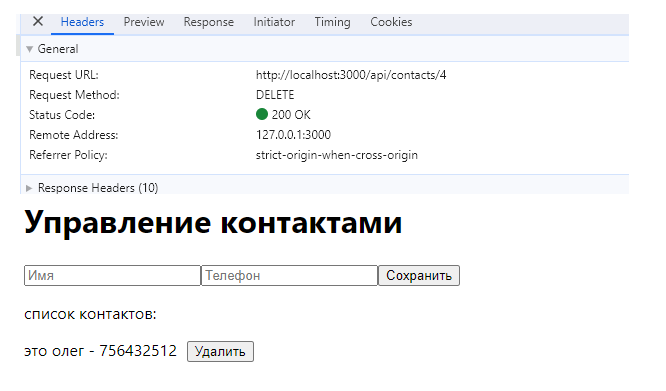


Рисунок 25 – Удаление контакта

Вывод

В результате выполнения данной практической работы были получены навыки по проектированию простейшего RESTful API, представляющее из себя базовое приложение для управления списком телефонных контактов, и его развертыванию на сервере cyclic.

# Практическая работа №6

**Цель работы**

Целью данной практической работы является знакомство студентов с основными подходами к построению API и инструментарием, используемый для ускорения процесса разработки API, в части документирования интерфейса прикладного программирования.

**Постановка задачи**

В данной практической работе необходимо ознакомиться с техникой разработки документации и создания программных интерфейсов на основе спецификации OpenAPI. В качестве задания необходимо выбрать любую прикладную задачу, например связанной с тематикой компьютерных игр, компьютерного интернет магазина и т. д. и создать интерфейс прикладного программирования для данной задачи, используя для этого OpenAPI, ReDOC, prism. Для создания спецификации OpenAPI на языке YAML можно воспользоваться Swagger Editor (https://editor.swagger.io/) Для проверки правильности работы решения необходимо отправить соответствующий запрос, например, в Postman и сверить запрос с ReDoc в правой части html-страницы.

Ход работы

В результате выполнения данной работы был создан файл swagger.yaml для API маркетплейса компьютеров. На рисунках 28-30 приведен код этого файла. На рисунке 31 показана сгенерированная ReDoc страница с описанием API. На рисунке 32 показан вывод Prism в консоль, а на рисунках 33-34 тестирование вызовов Prism в Postman.

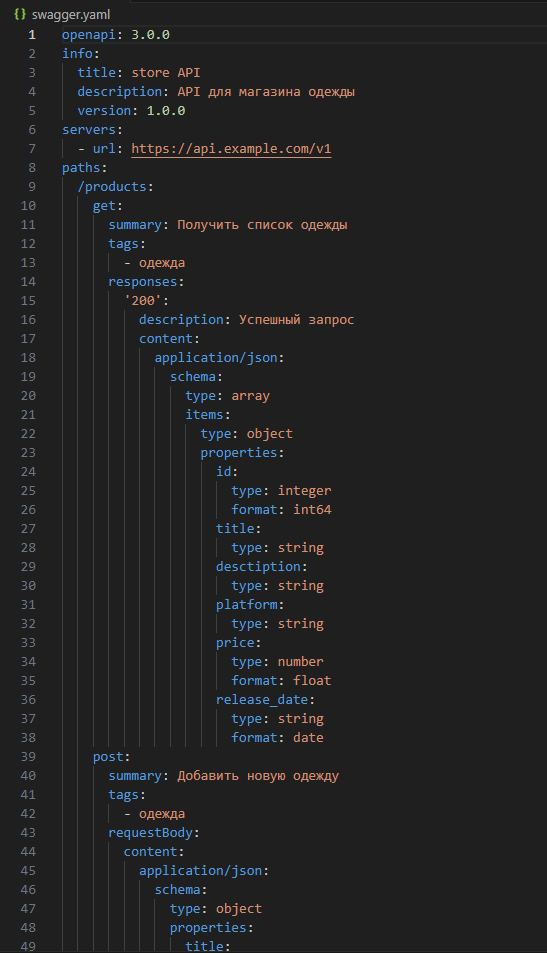


Рисунок 28 – Первая часть файла swagger.yaml

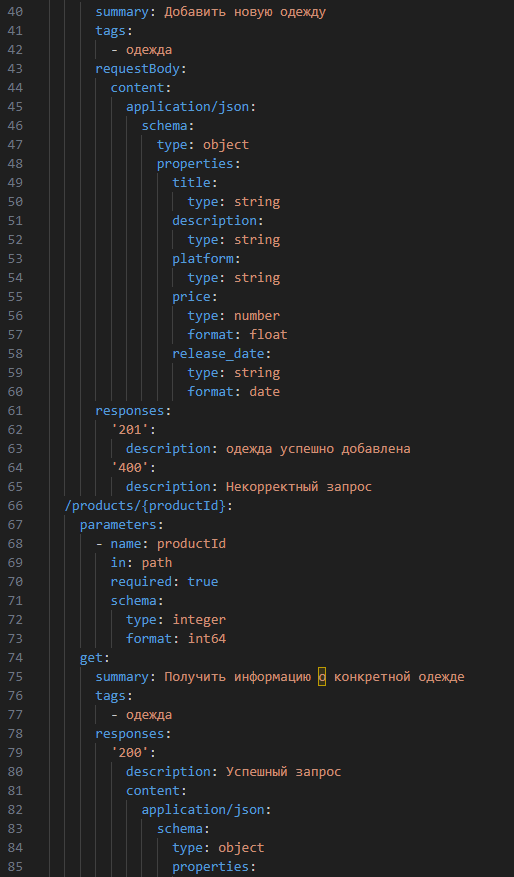


Рисунок 29 – Вторая часть файла swagger.yaml

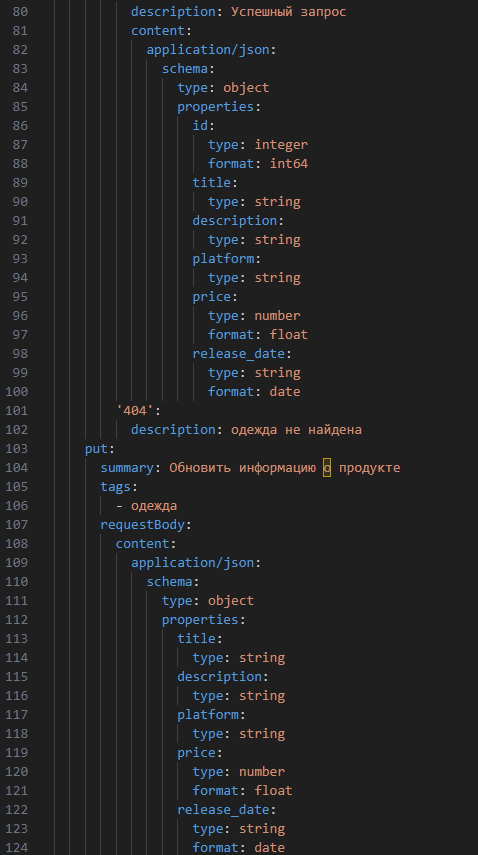


Рисунок 30 – Третья часть файла swagger.yaml

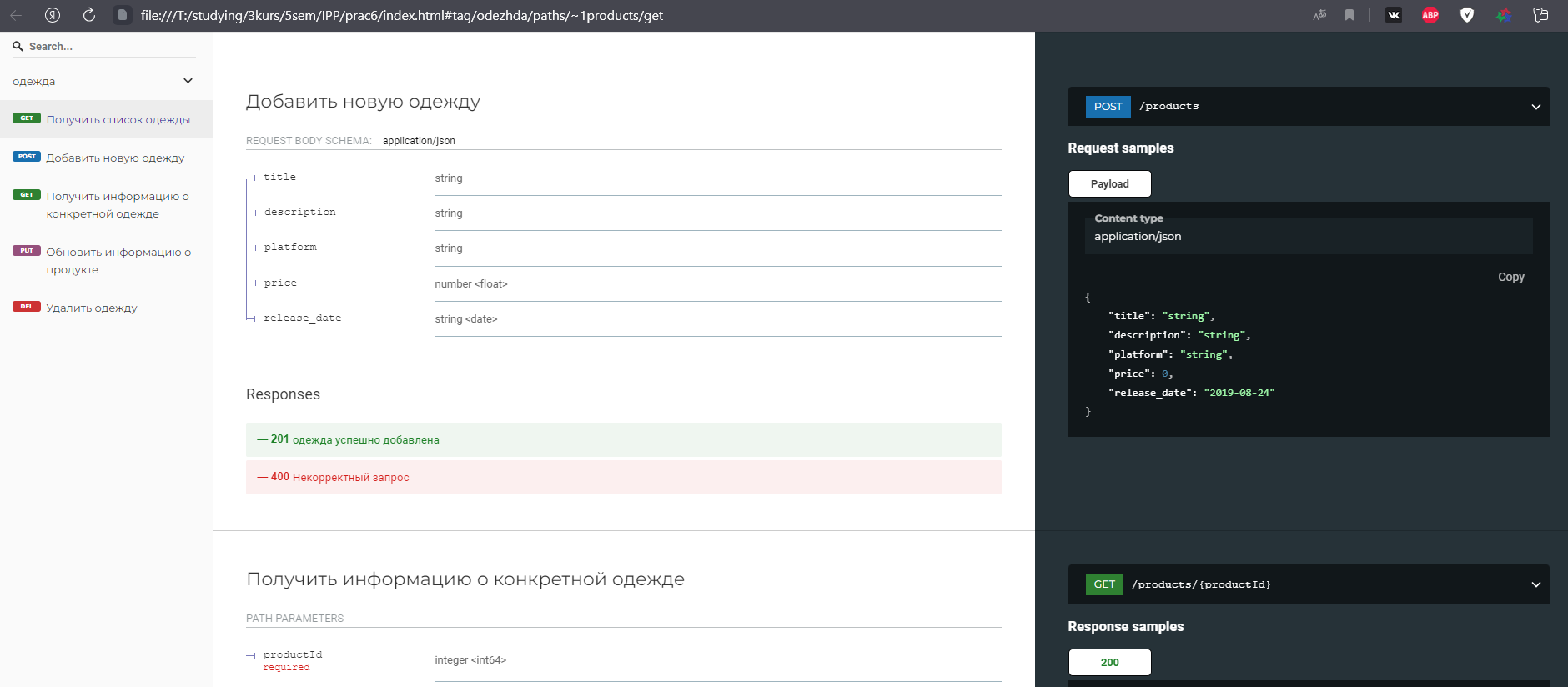


Рисунок 31 – Сгенерированная страница ReDoc

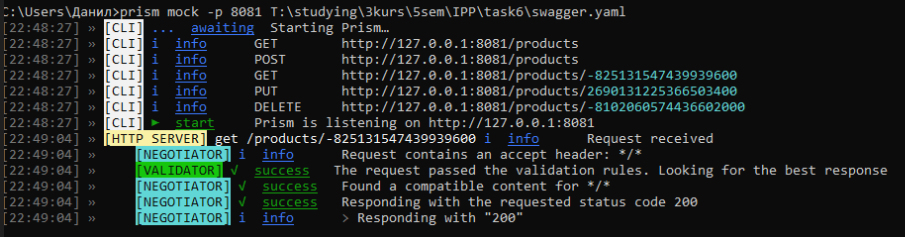


Рисунок 32 – Вывод prism в консоли

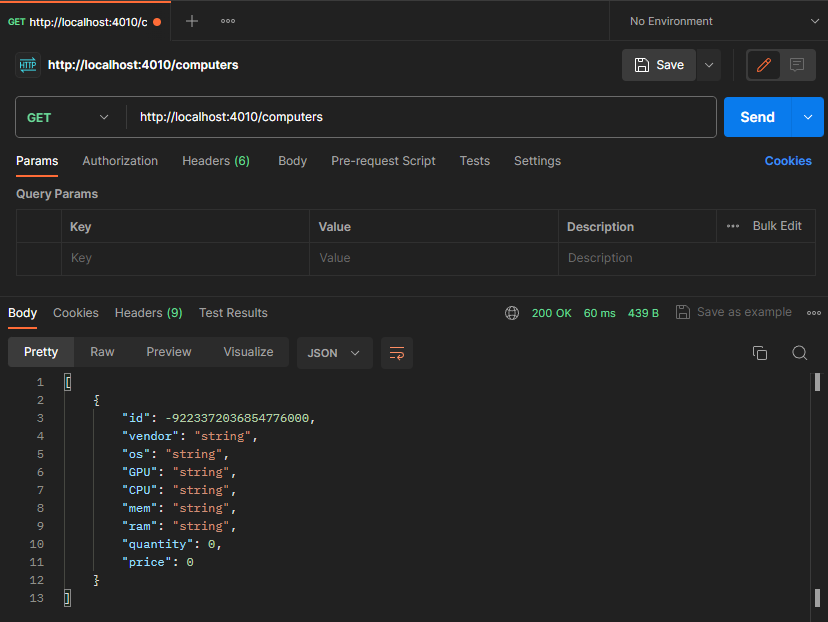


Рисунок 33 – Тестирование вызова к Prism

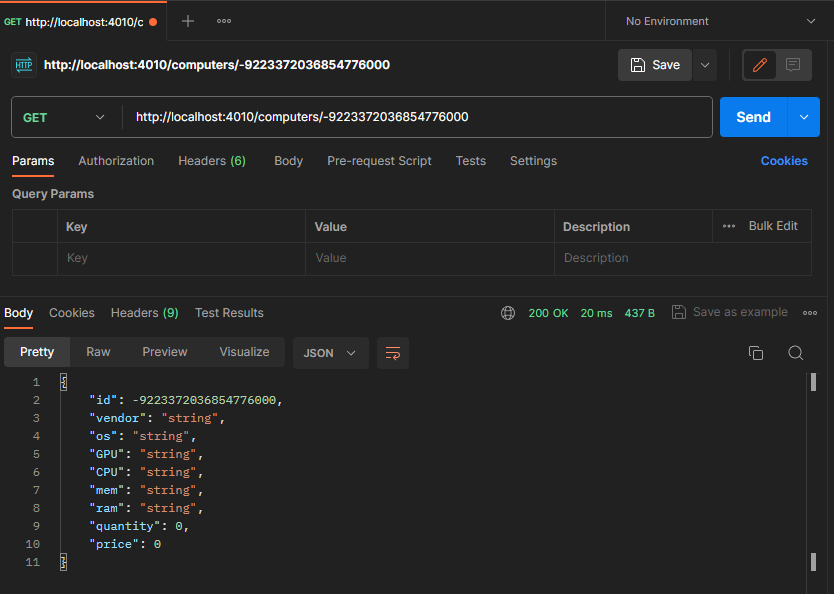


Рисунок 34 – Тестирование получения записи по id из Prism

**Вывод**

Ознакомился с основными подходами и использовал их на практике для построения API и инструментарием, используемым для ускорения процесса разработки API, в части документирования интерфейса прикладного программирования.

# Практическая работа №7

**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.

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 следующей бизнес-задачи:

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).

Ход работы

Для создания Node.js сервера необходимо провести конфигурацию системы. На рисунках 35-36 приведены команды, необходимые для настройки окружения для дальнейшей работы. Далее в файле index.js определяется конфигурация запускаемого сервера (листинг 2), и затем в файле schema.js (листинг 3) необходимо определить схему GraphQL в соответствие с индивидуальным вариантом.

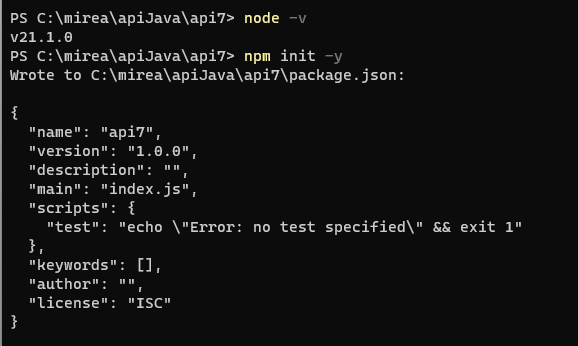


Рисунок 35 – Инициализация node.js проекта

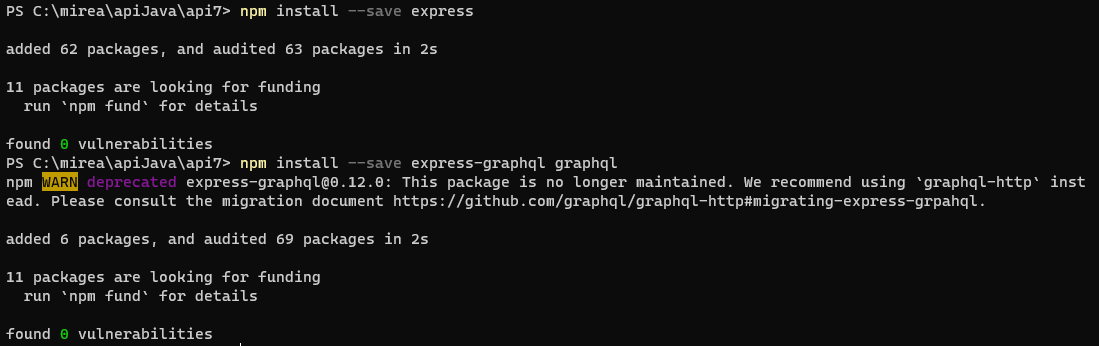


Рисунок 36 – Установка express и GraphQL

Листинг 2 – Код index.js

const express = require('express');

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

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

const app = express();

const port = 1234;

app.use('/graphql',

    graphqlHTTP({

        schema: schema,

        graphiql: true,

    }))

app.listen(port);

Листинг 3 – Код schema.js

const graphql = require('graphql');

const \_ = require("lodash");

const {

    GraphQLID,

    GraphQLString,

    GraphQLList,

    GraphQLInt,

    GraphQLNonNull,

    GraphQLObjectType,

    GraphQLSchema

} = graphql;

const cars = [

    {

        id: "1",

        title: "Kalina Restyling",

        brand: "LADA",

        price: 500000,

        age: 5

    },

    {

        id: "2",

        title: "C220",

        brand: "Mercedes-Benz",

        price: 5000000,

        age: 1

    },

    {

        id: "3",

        title: "Land Cruiser 200",

        brand: "Toyota",

        price: 3000000,

        age: 3

    }];

const CarInfoType = new GraphQLObjectType({

    name:'CarInfo',

    fields: () =>({

        id: {type: GraphQLID},

        title: {type: GraphQLString},

        brand: {type: GraphQLString},

        price: {type: GraphQLInt},

        age: {type: GraphQLInt}

    })

})

const RootQuery = new GraphQLObjectType({

name: 'RootQueryType',

Продолжение листинга 3

  fields: {

        info:{

            type: GraphQLString,

            resolve(parent, args){

                return "Server's running"

            }

        },

        car:{

            type: CarInfoType,

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

            resolve(parent, args) {

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

            }

        },

        cars:{

            type: new GraphQLList(CarInfoType),

            resolve(parent, args) {

                return cars;

            }

        }

    }

});

const Mutations = new GraphQLObjectType({

    name: 'Mutations',

    fields:{

        addcar: {

            type: CarInfoType,

            args: {

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

                title: {type: new GraphQLNonNull(GraphQLString)},

                brand: {type: GraphQLString},

                price: {type: new GraphQLNonNull(GraphQLInt)},

                age: {type: GraphQLInt}

            },

            resolve(parent, args) {

                const arrLength = cars.push(args);

                return cars [arrLength - 1];

            }

        }

    }

});

module.exports = new GraphQLSchema({

    query: RootQuery,

    mutation: Mutations

});

Командой node index.js выполним запуск сервера. Перейдя по URL http://localhost:1234/graphql мы попадаем на веб-интерфейс GrapiQL, позволяющий в удобной форме выполнять GraphQL запросы (рисунок 37).

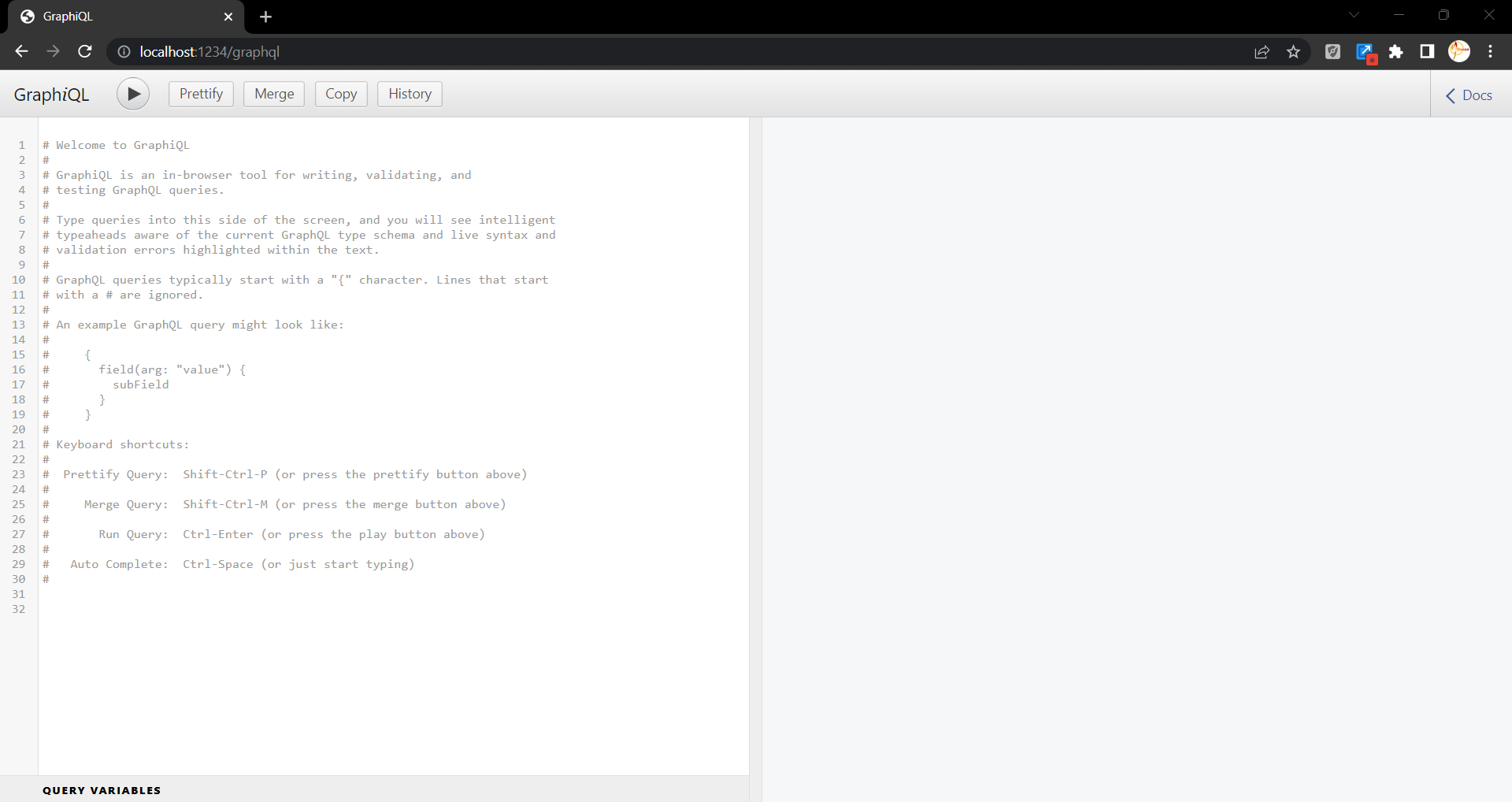


Рисунок 37 – Веб-интерфейс GraphiQL

Далее выполним проверку работы GraphQL сервера с помощью POST-запроса mutation (рисунок 38) и GET-запроса cars (рисунок 39), которые добавляют новый автомобиль и выводят список всех имеющихся машин соответственно.

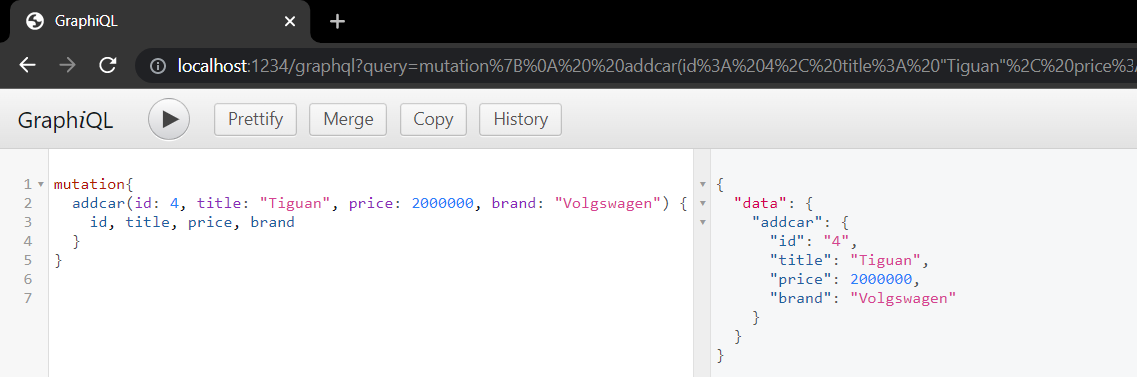


Рисунок 38 – Выполнение запроса mutation

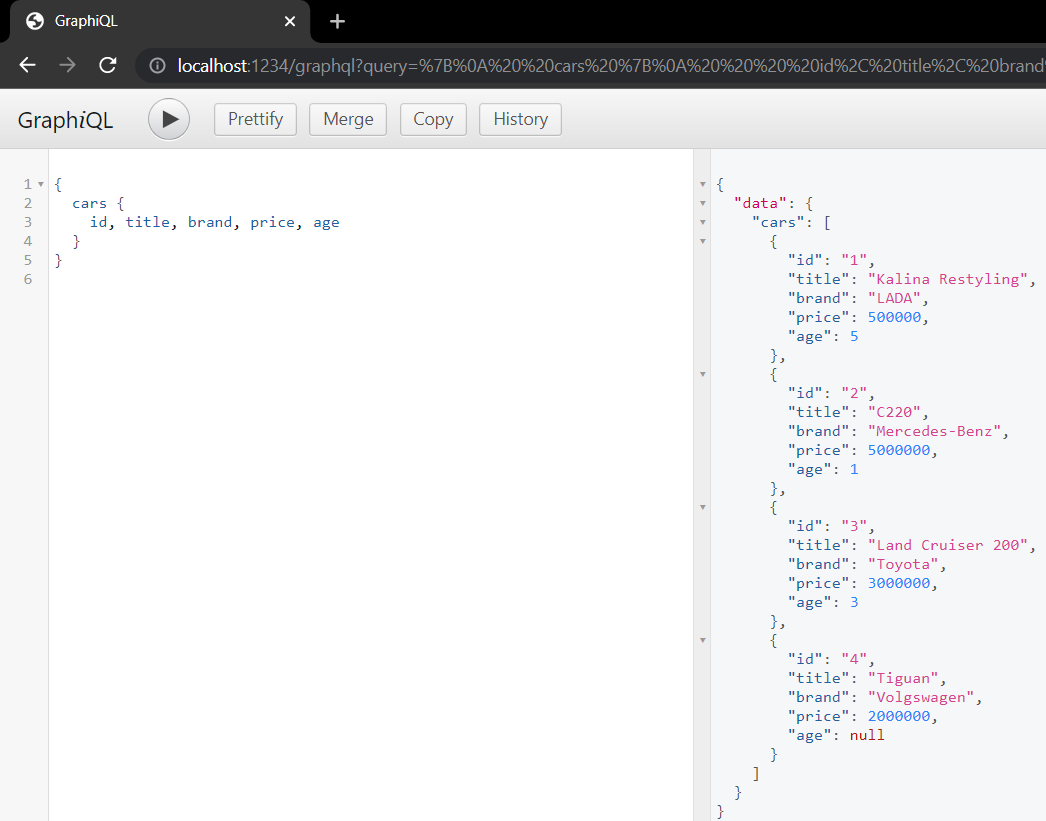


Рисунок 39 – Выполнение запроса cars

Вывод

Используя теоретические сведения из данной практической работы, открытые интернет-источники, официальную документацию по GraphQL с использованием SDL создал схему, реализовал сервер и клиента GraphQL для приложения для хранения информации об автомобилях.

# Практическая работа №8

**Цель работы**

Знакомство обучающихся с ещё одним архитектурным стилем, который может использоваться для создания программных интерфейсов приложений широкого назначения. Данный стиль основывается на использовании технологии удалённого вызова процедур (англ. remote procedure call) и её дальнейшей модернизации, осуществлённой в рамках технологии gRPC.

**Постановка задачи**

Используя теоретические сведения из данной практической работы, открытые интернет-источники, официальную документацию по gRPC необходимо реализовать минимально жизнеспособный продукт (MVP) клиент-серверное приложение с API и базовыми операциями create, read, update, delete (CRUD) для следующих бизнес-задач: сервис для формирования списка личных задач на день. Приложение должно позволять создавать задачу, редактировать её, просматривать список задач, удалять задачу из списка. Дополнительный функционал, которые вы хотели бы реализовать в данном приложении, необходимо обсудить и согласовать с преподавателем.

**Ход работы**

Инициализируем проект и устанавливаем библиотеки Express, gRPC, hbs и uuid (Рисунки 40-41).

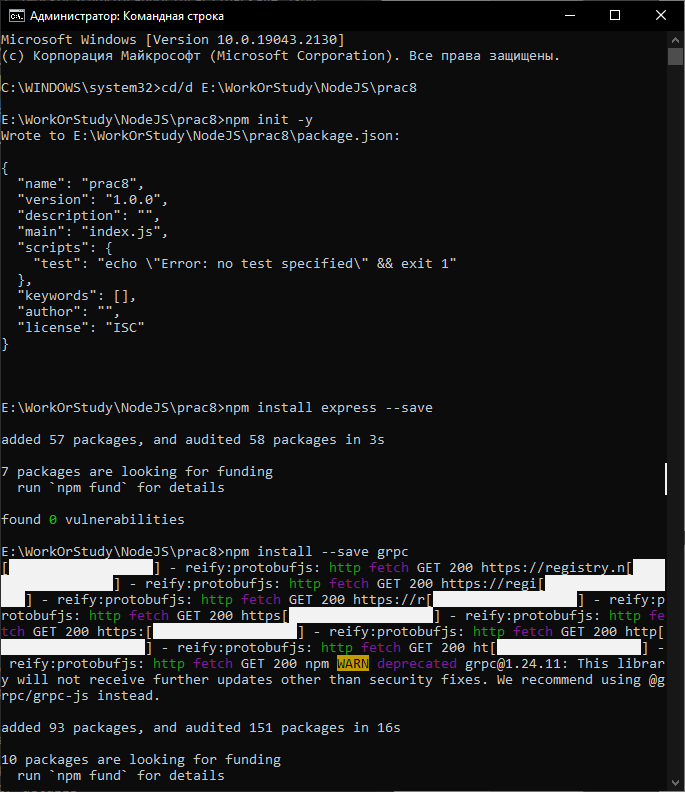


Рисунок 40 – Инициализация проекта

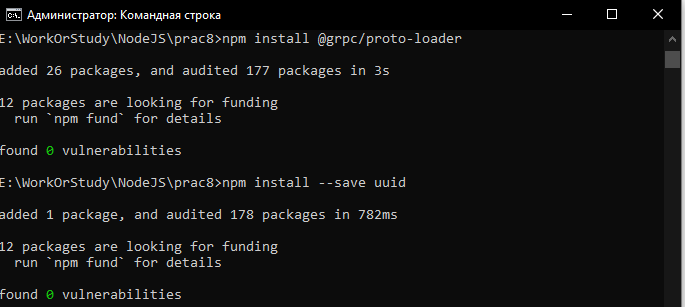


Рисунок 41 – Установка зависимостей

Далее необходимо сконфигурировать \*.proto файл (Рисунок 42) и подстроить server.js (Листинг 4) и index.js (Листинг 5) файлы для корректного удаленного вызова функций.

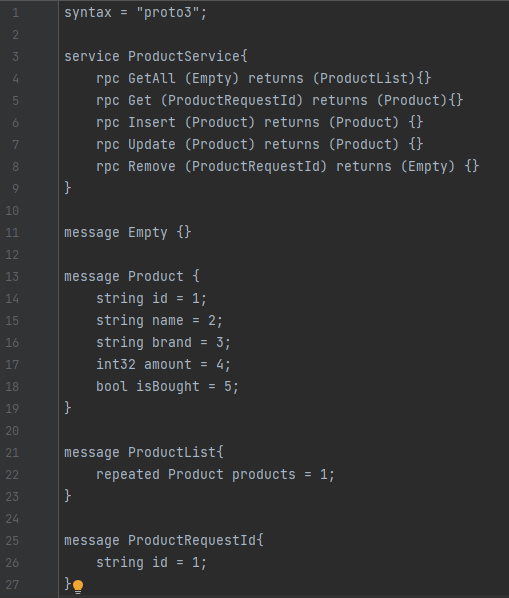


Рисунок 42 – Файл product.proto

Листинг 4 – Код server.js

const PROTO\_PATH = "./product.proto";

var grpc = require("grpc");

var protoLoader = require("@grpc/proto-loader");

var packageDefinition = protoLoader.loadSync(PROTO\_PATH, {

keepCase: true,

longs: String,

enums: String,

arrays: true

});

var productProto = grpc.loadPackageDefinition(packageDefinition);

const {v4: uuidv4} = require("uuid");

const server = new grpc.Server();

const products = [

{

Продолжение листинга 4

id: "1",

name: "Хлеб",

brand: "Белый",

amount: 1,

isBought: false

},

{

id: "2",

name: "Пиво",

brand: "Жигули барное",

amount: 20,

isBought: false

}

];

server.addService(productProto.ProductService.service, {

getAll: (\_, callback) => {

callback(null, {products: products});

},

get: (call, callback) => {

let product = products.find(n => n.id == call.request.id);

if (product) {

callback(null, product);

} else {

callback({

code: grpc.status.NOT\_FOUND,

details: "Не найдено"

});

}

},

insert: (call, callback) => {

let product = call.request;

product.id = uuidv4();

products.push(product);

callback(null, product);

},

update: (call, callback) => {

let existingProduct = products.find(n => n.id == call.request.id);

if (existingProduct) {

existingProduct.name = call.request.name;

existingProduct.brand = call.request.brand;

existingProduct.amount = call.request.amount;

existingProduct.isBought = call.request.isBought;

callback(null, existingProduct);

} else {

callback({

code: grpc.status.NOT\_FOUND,

details: "Не найдено"

});

Продолжение листинга 4

}

},

remove: (call, callback) => {

let existingProductIndex = products.findIndex(

n => n.id == call.request.id

);

if (existingProductIndex != -1) {

products.splice(existingProductIndex, 1);

callback(null, {});

}

else {

callback({

code: grpc.status.NOT\_FOUND,

details: "Не найдено"

});

}

}

});

server.bind("127.0.0.1:50051", grpc.ServerCredentials.createInsecure());

console.log("Сервер запущен по адресу http://127.0.0.1:50051");

server.start();

Листинг 5 – Код index.js

const client = require("./client");

const path = require("path");

const express = require("express");

const bodyParser = require("body-parser");

const app = express();

app.set("views", path.join(\_\_dirname, "views"));

app.set("view engine", "hbs");

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({extended: false}));

app.get("/", (req, res) => {

client.getAll(null, (err, data) => {

if (!err) {

res.render("products", {

results: data.products

});

}

});

});

app.post("/save", (req, res) => {

let newProduct = {

name: req.body.name,

brand: req.body.brand,

amount: req.body.amount,

isBought: req.body.isBought

Продолжение листинга 5

};

newProduct.isBought = newProduct.isBought == null ? false : true;

client.insert(newProduct, (err, data) => {

if (err) throw err;

console.log("Товар добавлен", data);

res.redirect("/");

});

});

app.post("/update", (req, res) => {

const updateProduct = {

id: req.body.id,

name: req.body.name,

brand: req.body.brand,

amount: req.body.amount,

isBought: req.body.isBought

};

updateProduct.isBought = updateProduct.isBought == null ? false : true;

client.update(updateProduct, (err, data) => {

if (err) throw err;

console.log("Товар успешно обновлён", data);

res.redirect("/");

});

});

app.post("/remove", (req, res) => {

client.remove({id: req.body.product\_id}, (err, \_) => {

if (err) throw err;

console.log("Товар удалён");

res.redirect("/");

});

});

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {

console.log("Сервер запущен на порту %d", PORT);

});

После необходимо создать фронтэнд-часть с помощью шаблонизатора hbs, (фрагмент представлен на рисунке 43).

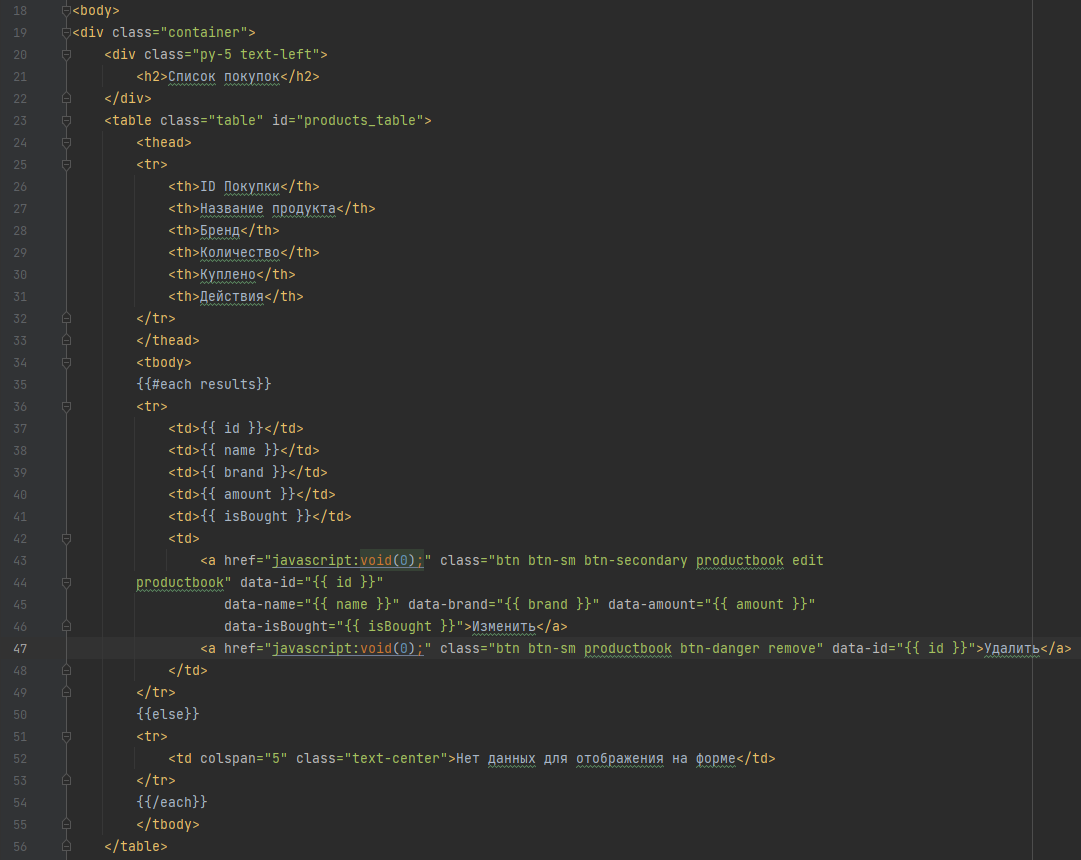


Рисунок 43 – Файл products.hbs

Переходим к тестированию, запускаем раздельно серверную и клиентскую часть (Рисунки 44-45).

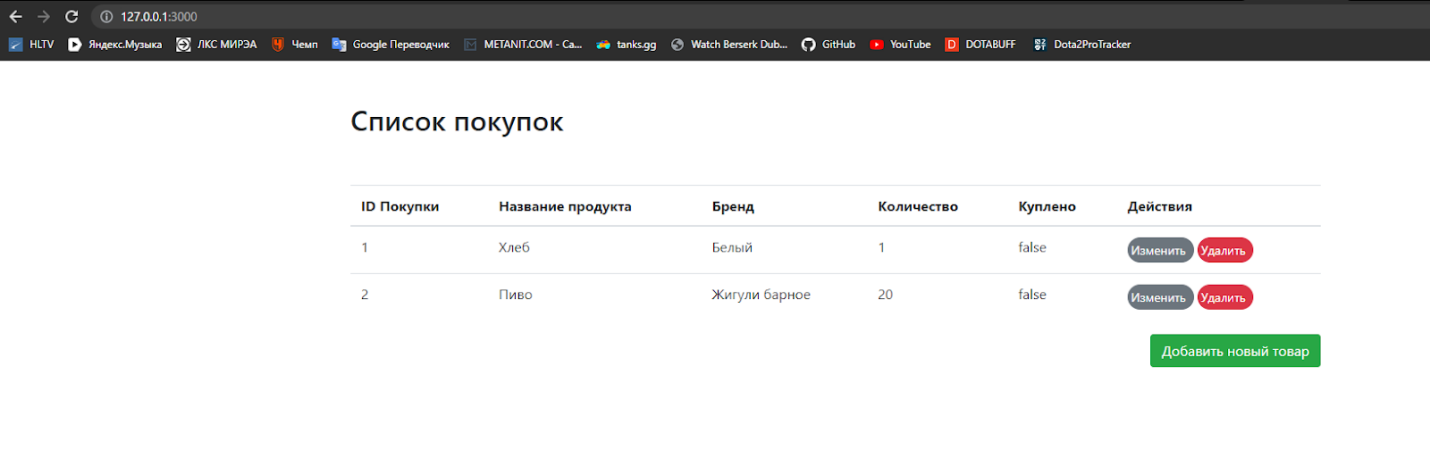


Рисунок 44 – Серверная часть



Рисунок 45 – Клиентская часть

Проведем тестирование всех методов (Create, Read, Update, Delete). Результаты представлены в виде логов в консоле клиентской части, а также в графическом интерфейсе пользователю (Рисунки 46-48).



Рисунки 46 – Стартовый вид при запуске

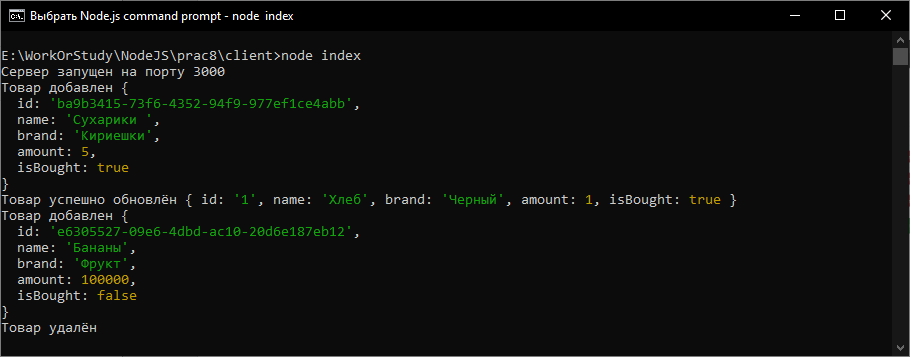


Рисунок 47 – Лог всех изменений

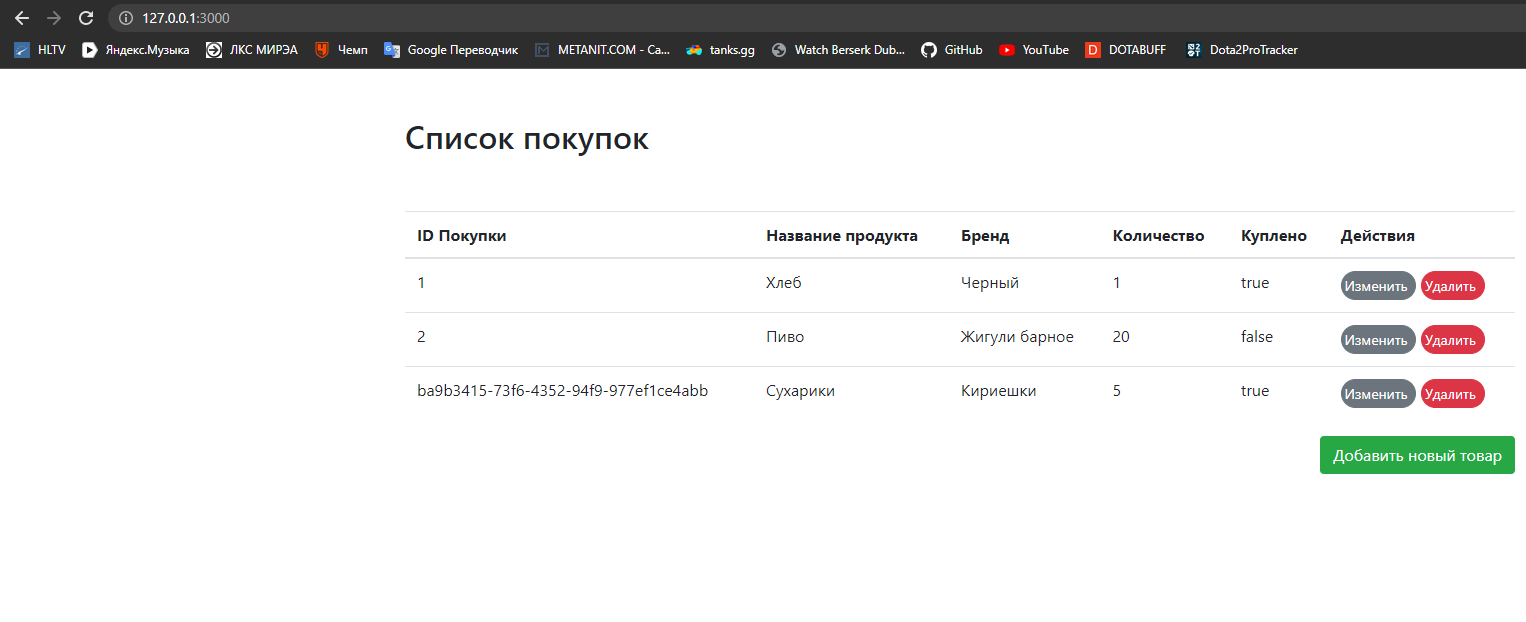


Рисунок 48 – Конечный вид фронт-части API

Вывод

Используя теоретические сведения из данной практической работы, открытые интернет-источники, официальную документацию по gRPC реализовал минимально жизнеспособный продукт (MVP) клиент-серверное приложение с API и базовыми операциями create, read, update, delete (CRUD) для cервиса.

# Вывод

В ходе выполнения восьми практических работ были освоены различные аспекты работы с публичными WEB API. Каждая из задач предоставила ценный опыт в создании приложений, демонстрирующих функциональность выбранных API. Были созданы веб-страницы с использованием AJAX для запросов к openweathermap.org, что позволило динамически обновлять данные о прогнозе погоды.

Далее, была проведена работа с jsonserver для имитации API сервиса поиска книг в библиотеке, созданы соответствующие конечные точки и протестированы запросы с использованием curl и Postman. Затем, с использованием предоставленного стека технологий, было успешно развернуто приложение для управления списком телефонных контактов на сервере.

Также был освоен процесс разработки документации и создания программных интерфейсов с использованием спецификации OpenAPI, ReDOC и prism. Этот этап позволил создать удобный интерфейс программирования приложения, а также проверить его работу с помощью Postman.

Наконец, с применением знаний о GraphQL и gRPC были реализованы клиент-серверные приложения с базовыми операциями CRUD, демонстрируя понимание принципов работы этих технологий. В результате, данные практические работы позволили овладеть важными навыками работы с различными API и их интеграцией в приложения.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Алпатов А.Н., «Практикум по проектированию и разработке программных интерфейсов приложений в WEB» [Электронный ресурс] – Режим доступа: https://online-edu.mirea.ru/pluginfile.php?file=%2F970821%2Fmod\_resource%2Fcontent%2F0%2F%D0%9F%D1%80%D0%B0%D0%BA%D1%82%D0%B8%D0%BA%D1%83%D0%BC\_%D0%BF%D0%BE\_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8E\_%D0%B8\_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B5\_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D1%85\_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D0%BE%D0%B2.pdf (дата обращения: 22.10.2023)
2. Справочная информация JavaRush [Электронный ресурс] – Режим доступа: https://javarush.com/groups/posts/1094-ajax-i-drugie-jazihki (дата обращения: 07.09.2023)
3. Статья Baeldung [Электронный ресурс] – Режим доступа: https://www.baeldung.com/angular-js-rest-api (дата обращения: 10.09.2023)
4. Статья Habr [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/590679/ (дата обращения: 12.09.2023)
5. Методические указания для выполнения правктических работ [Электронный ресурс] – Режим доступа: https://online-edu.mirea.ru/pluginfile.php?file=%2F970821%2Fmod\_resour ce%2Fcontent%2F0%2FПрактикум\_по\_проектированию\_и\_разработке\_программных\_интерфейсов.pdf (дата обращения: 28.09.2023)
6. Статья Habr [Электронный ресурс] – Режим доступа: https://habr.com/ru/companies/ruvds/articles/445268 (дата обращения: 10.10.2023)
7. Статья Medium [Электронный ресурс] – Режим доступа: https://medium.com/maddevs-io/introduction-to-grpc-6de0d9c0fe61 (дата обращения: 15.10.2023)