Министерство науки и высшего образования Российской Федерации

ФГАОУ ВО «Северо-Восточный федеральный университет имени М.К. Аммосова»

Институт математики и информатики

Кафедра «Информационные технологии»

**ОТЧЕТ**

**Учебной (эксплуатационной) практики**

ФИО студента: Никифоров Арсен Германович

Направление подготовки: 02.03.02 Фундаментальная информатика и информационные технологии, профиль: Программирование и информационные технологии

Курс обучения: 2 (второй), группа БА-ФИИТ-21

Вид практики: Учебная (эксплуатационная)

Сроки практики: 22.06.2023 - 05.07.2023 (2 недели)

Место прохождения практики: кафедра «Информационные технологии» (дистанционно)

Руководитель практики: **Саввинова Дина Петровна**, ст. преподаватель

Ф.И.О. должность, ученое звание руководителя

Якутск, 2023

**Ссылка на репозиторий учебной практики:**<https://github.com/Arsenshmid/Training_Practice>

**Введение**

В данном отчете представлен обзор и анализ трех электронных курсов, пройденных в рамках учебной практики. Целью практики являлось повышение профессиональных навыков и компетенций в области информационных технологий, анализа данных и экономики.

Курсы, рассмотренные в отчете, включают следующие темы:

1. **Курс на платформе Stepik «Интерактивный тренажер по SQL»** (ДВФУ, 35 ч.) - изучение основ реляционной модели данных, запросов SQL к связанным таблицам, работы с базами данных и пользовательскими запросами. Курс состоит из 22 уроков, 10 тестов и 172 интерактивных задач. [Ссылка на курс](https://stepik.org/course/63054/promo)
2. **Курс на платформе НОУ «ИНТУИТ» «Табличный процессор Excel в экономических и финансовых расчетах»** (72 ч.) - обучение работы с различными функциями и инструментами Microsoft Excel для проведения экономических и финансовых анализов. Курс включает 17 лекций, 8 тестов и экзамен. [Ссылка на курс](https://intuit.ru/studies/courses/3659/901/info)
3. **Курс на платформе Stepik «Проектный практикум»** - ознакомление с основами представления данных, протоколами SOAP и RESTful, а также работа с серверной и клиентской частями приложения. Курс состоит из 14 уроков и 3 тестов. [Ссылка на курс](https://stepik.org/course/52624/promo)

В ходе прохождения курсов были получены теоретические знания и практические навыки, которые могут быть применены в дальнейшей профессиональной деятельности. В данном отчете приведены подробные разъяснения и анализ материалов каждого курса, а также обобщение полученных знаний и опыта.

**Основная часть**

В течение учебной практики я прошел три курса, которые охватывают различные аспекты работы с данными и их анализа. Следующие курсы были пройдены на разных платформах:

1. "Интерактивный тренажер по SQL" на платформе Stepik;
2. "Табличный процессор Excel в экономических и финансовых расчетах" на платформе НОУ «ИНТУИТ»;
3. "Проектный практикум" на платформе Stepik.

**Курс "Интерактивный тренажер по SQL"**

На курсе "Интерактивный тренажер по SQL" я изучил основы реляционной модели данных и теории баз данных. В частности, я освоил запросы SQL к связанным таблицам, углубился в работу с базами данных и закрепил навыки написания SQL запросов. В ходе обучения я успешно выполнил 22 урока, 10 тестов и 172 интерактивных задачи, что подтверждает мою компетентность в данной области.

**Курс "Табличный процессор Excel в экономических и финансовых расчетах"**

Курс "Табличный процессор Excel в экономических и финансовых расчетах" позволил мне освоить широкий спектр возможностей программы Excel. В рамках курса я изучил создание и редактирование таблиц и диаграмм, научился использовать встроенные функции для проведения статистического анализа, работу с базами данных, структурирование таблиц, создание сводных таблиц, консолидацию данных, использование финансовых функций и анализ данных, прогнозирование, анализ временных рядов и применение метода скользящей средней. В результате обучения я прошел 17 лекций, 8 тестов и успешно сдал экзамен.

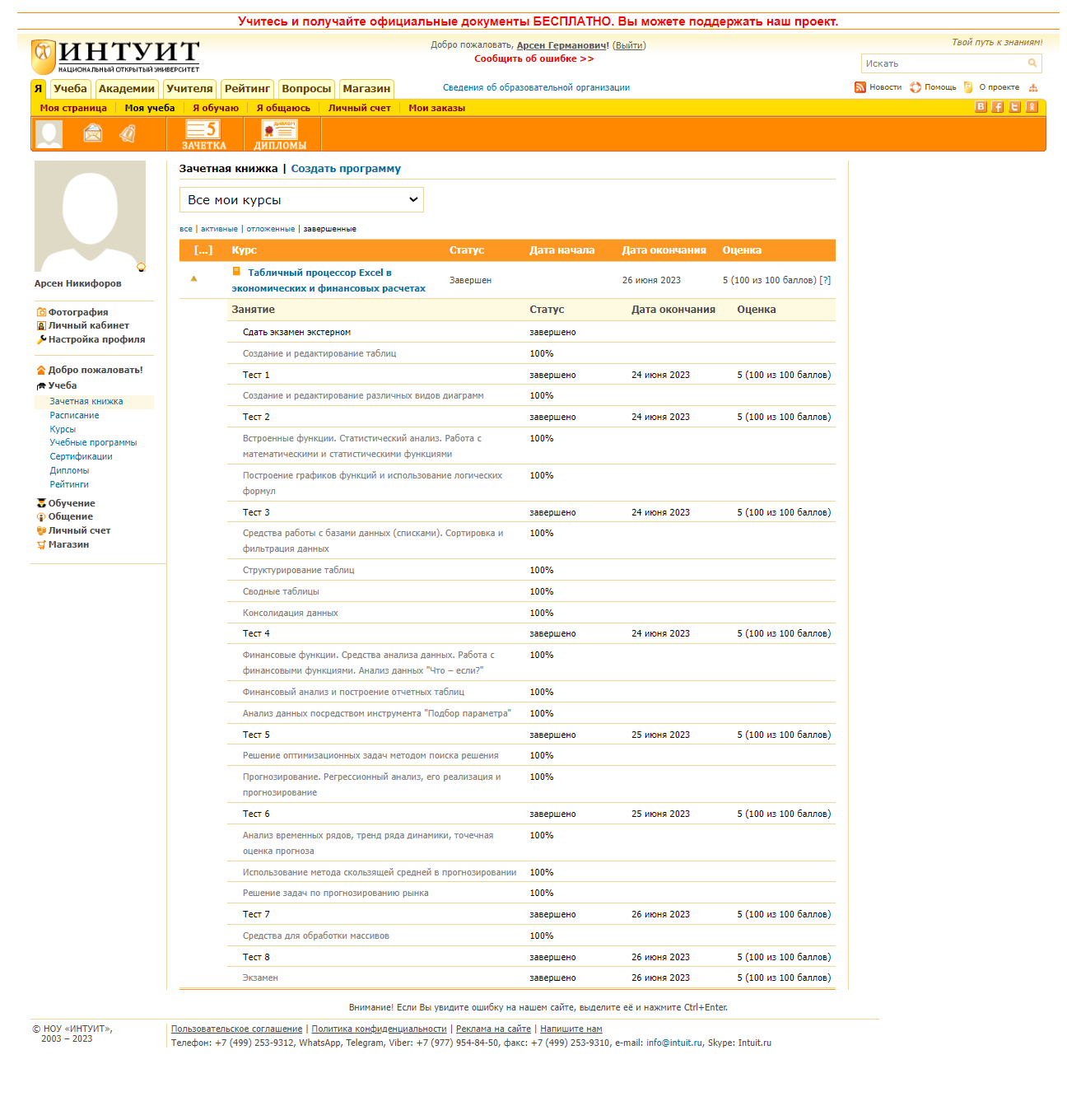
**Курс "Проектный практикум"**

В ходе прохождения курса "Проектный практикум" я изучил различные способы представления данных, такие как SOAP, API, RESTful, правила именования ресурсов в RESTful архитектуре, а также серверную и клиентскую части приложения. Завершив 14 уроков и 3 теста, я смог успешно применить полученные знания на практике.

**Проектная работа**

После завершения всех курсов я разработал проект по курсу "Проектный практикум". В проекте я использовал знания, полученные на курсах, для решения практических задач, связанных с анализом данных и созданием информационных систем. Разработанный проект содержит код программной разработки, который я приложил к этому отчету. Этот проект продемонстрировал мою способность к синтезу и применению полученных знаний в реальных условиях.

НОУ ИНТУИТ

100/100 баллов по всем тестам и экзамену  




Проектный практикум

*Я выбрал стек технологий:*

* Frontend: React
* Backend: Express Framework (Node.js)

Можно использовать React на фронтенде и Express Framework на бэкенде для создания клиент-серверного веб-приложения. В этом случае, React будет использоваться для построения интерфейса пользователя на стороне клиента, а Express Framework будет использоваться для создания сервера и обработки запросов от клиента. Для связи между клиентом и сервером буду использовать.

**Немного теории**

HTTP-методы - это способ взаимодействия с сервером и выполнения операций с данными. Вот некоторые методы и их примеры:

* GET - используется для получения данных с сервера, например, получение списка товаров: GET http://example.com/products.
* POST - используется для создания новых данных на сервере, например, создание нового пользователя: POST http://example.com/users.
* PUT - используется для обновления существующих данных на сервере, например, обновление информации о пользователе с идентификатором 123: PUT http://example.com/users/123.
* DELETE - используется для удаления данных на сервере, например, удаление пользователя с идентификатором 123: DELETE http://example.com/users/123.

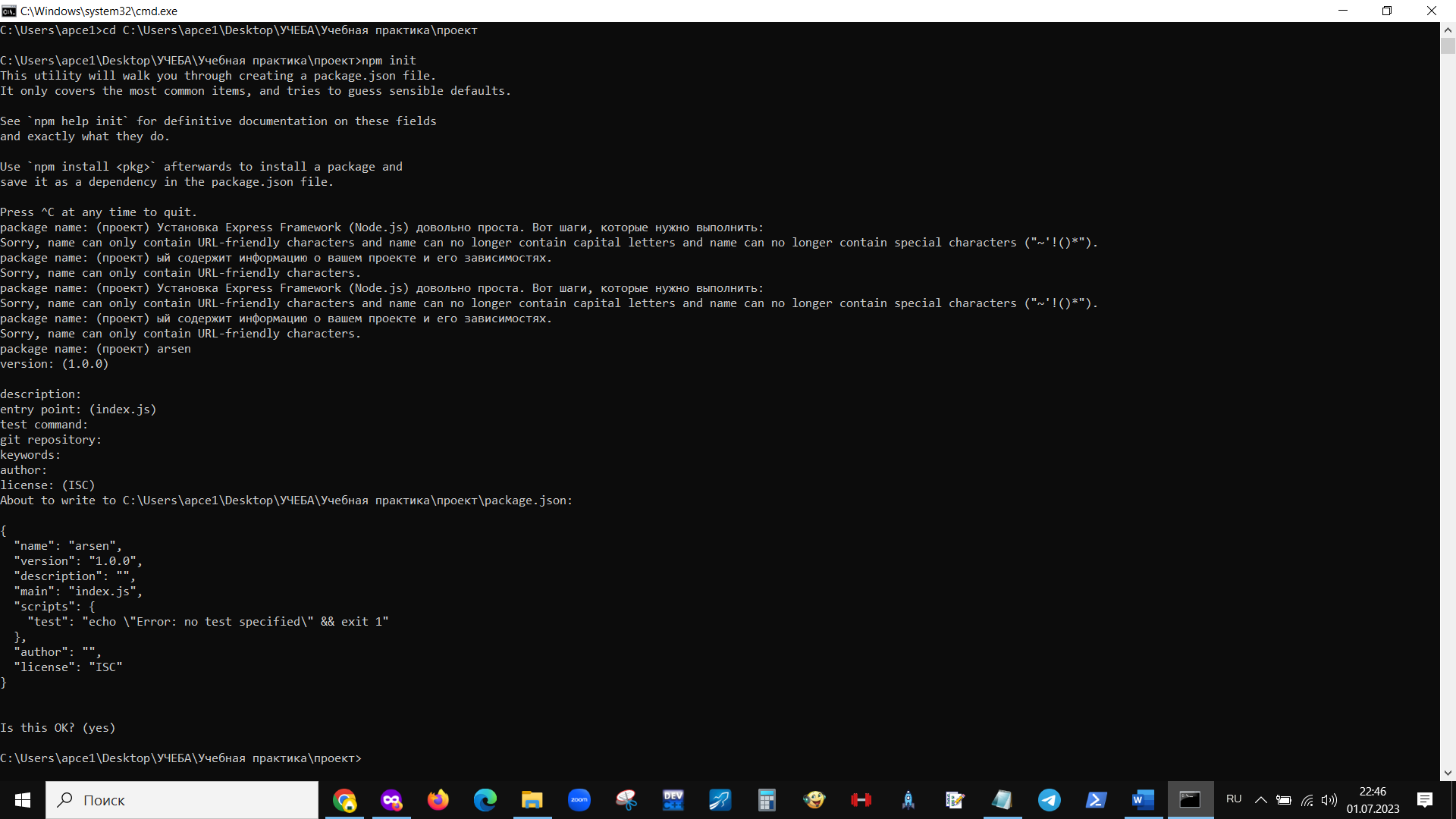
Когда вы отправляете запрос на сервер с определенным методом, сервер обрабатывает этот запрос и выполняет соответствующую операцию с данными. Например, если вы отправляете GET-запрос на http://example.com/products, сервер вернет список товаров в ответе.

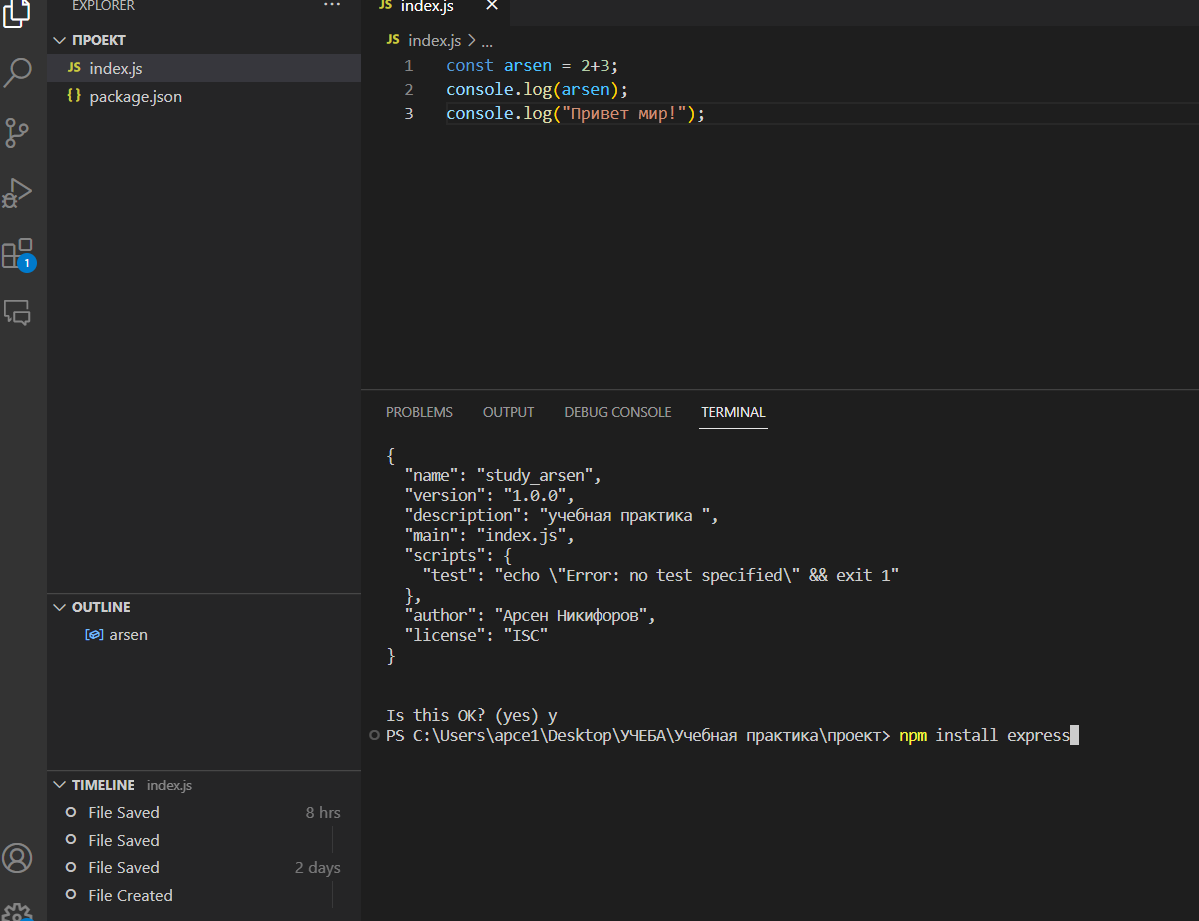
HTTP-методы используются при работе с веб-приложениями и веб-сервисами, включая REST API.

Установил node js с сайта и Express с помощью cmd.

* cd C:\Users\apce1\Desktop\УЧЕБА\Учебная практика\проект - переход в папку проекта.
* npm init - инициализация нового проекта Node.js и создание файла package.json.
* Задание имени проекта - arsen.
* Остальные поля package.json оставлены пустыми, кроме ключа "test", который содержит команду для запуска тестов.
* После запуска npm init созданный package.json выводится в командной строке и запрашивается подтверждение.
* После подтверждения файла package.json успешно создан.

В данном случае был создан новый проект Node.js с помощью npm init и создан файл package.json, который содержит информацию о проекте и его зависимостях.



Инициализировал свой проект, установил express и node js.   


**Лабораторная работа №1**

Разработать веб-службы, которые по запросу GET по адресу (URL) будет возвращать значение для следующих функций:

* Прописью возвращать значение числа, заданного в **URL**в качестве параметра пути (*Path Parameter*);
* Возвращает решение уравнения ***ax2+bx+c=0***, где ***a*, *b*** и ***c*** задаются в виде параметров запроса (*url?a=1&b=4&c=-1*);
* Возвращает день недели по заданной дате, которая передается как параметр запроса (*url?date=12.12.2028*);
* Возвращает число Фибоначчи, заданное своим индексом (*в виде параметра пути*);
* Возвращает название региона России по заданному номеру региона (*в виде параметра пути*). Например, при заданном *url/72*, должно возвращаться «Тюменская область».

Критерии выполнения лабораторной:

* Разработаны APIs из расчёта одна функция - одно API;
* Каждая функция отрабатывает корректно.

**Решение:**

**const** express = require('express'); *// Подключаем модуль Express для создания сервера*

**const** app = express(); *// Создаем экземпляр сервера*

app.**get**('/:number', (req, res) => {

**const** { number } = req.params; *// Извлекаем значение параметра number из запроса*

**const** numToWords = require('number-to-words'); *// Подключаем модуль number-to-words для преобразования числа в слова*

**const** result = numToWords.toWords(number); *// Преобразуем число в слова*

res.json({ result }); *// Отправляем JSON-ответ с преобразованными словами*

});

app.listen(3000, () => {

console.log('Server started on port 3000'); *// Выводим сообщение в консоль о том, что сервер запущен*

});

В данном коде мы создаём экземпляр приложения Express, который слушает запросы на порте 3000.

Затем мы используем метод app.get() для определения маршрута, который будет обрабатывать запросы GET с указанным параметром пути :number. Параметр пути извлекается из объекта req.params. Мы используем библиотеку number-to-words для преобразования числа в текстовый эквивалент. Результат отправляется в формате JSON с помощью метода res.json().

Запускаем приложение, используя метод app.listen().

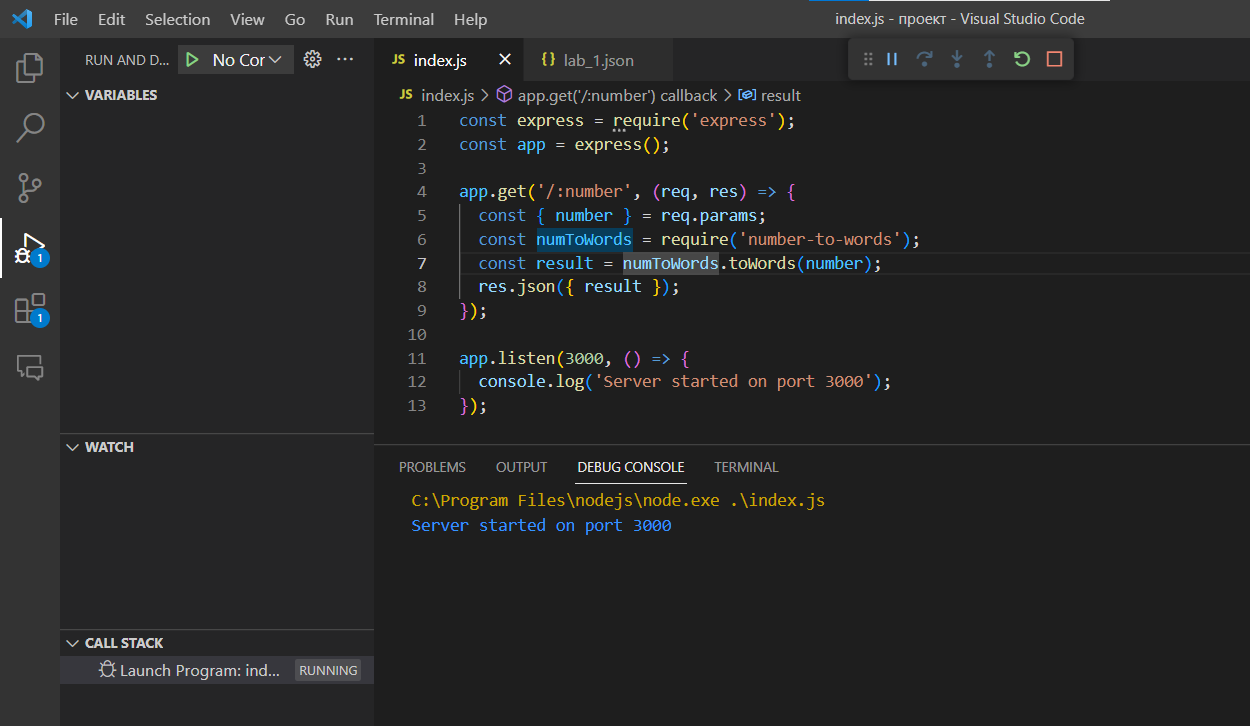
Теперь можно запустить приложение с помощью команды node index.js и проверить его работу, к примеру, введя в браузере URL http://localhost:3000/123, где 123 - число, которое нужно преобразовать в текстовый эквивалент. Если всё работает корректно, то в ответе должен быть JSON-объект вида:

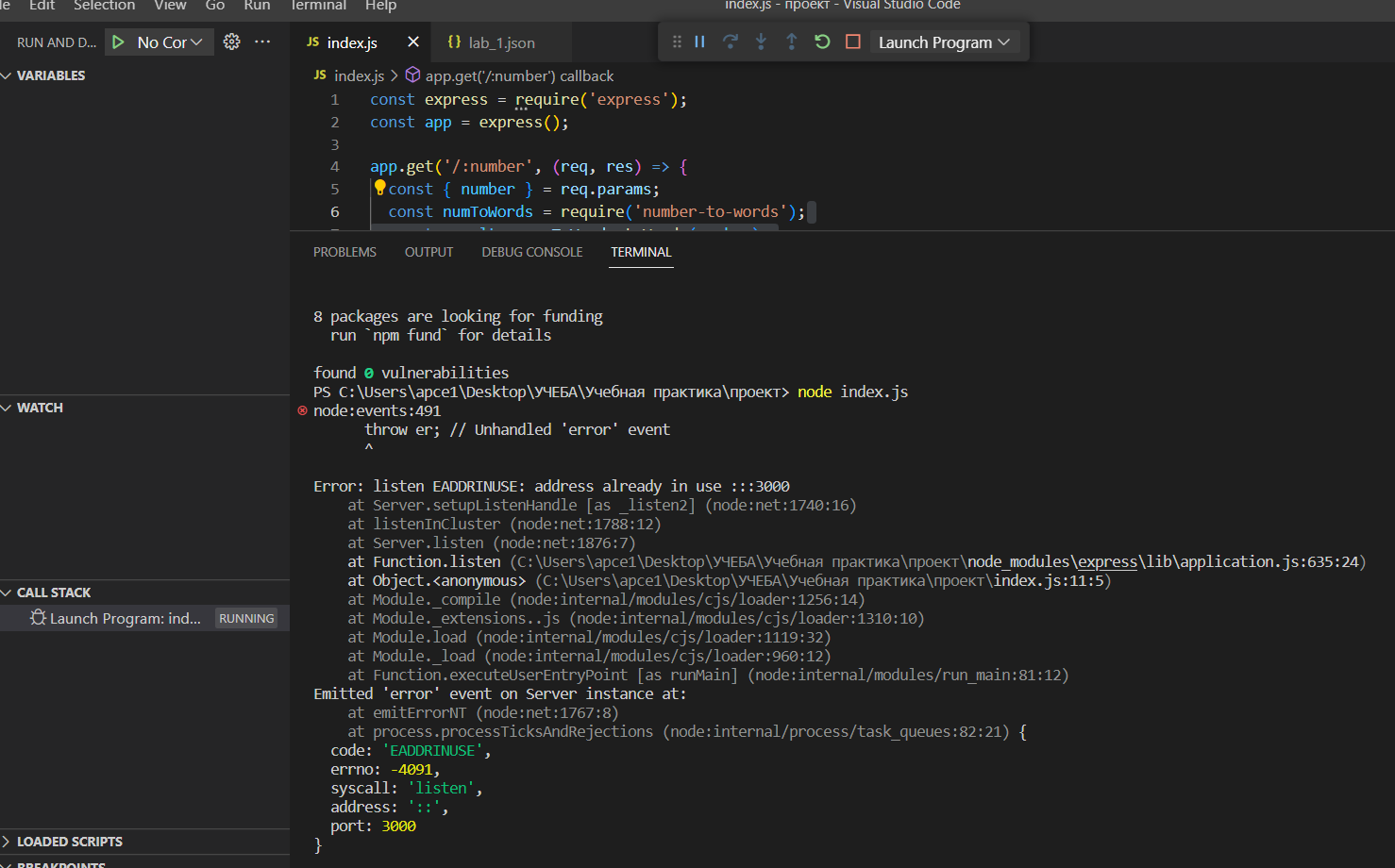
**Json:**

{

"result": "one hundred twenty-three"

}





Результат работы:

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

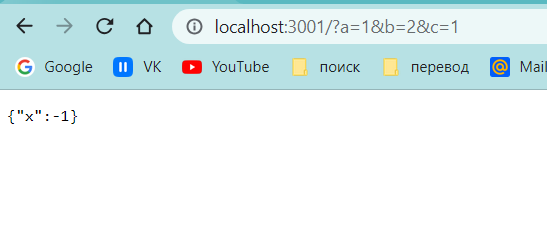
Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

После того, как вы убедились в корректной работе приложения, сохранил результаты работы API в файл согласно указанным критериям. Для этого создайл файл lab\_1.json и сохранил в него JSON-объект. После этого добавил файл в свой репозиторий на GitHub.

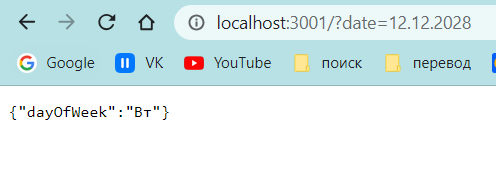
Так же и со всеми остальными работами.

* Возвращает решение уравнения ***ax2+bx+c=0***, где ***a*, *b*** и ***c*** задаются в виде параметров запроса (*url?a=1&b=4&c=-1*);
* const express = require('express');
* const app = express();
* const fs = require('fs');
* // Обработчик GET-запроса для решения уравнения
* app.get('/', (req, res) => {
* const a = parseFloat(req.query.a);
* const b = parseFloat(req.query.b);
* const c = parseFloat(req.query.c);
* // Решение уравнения
* const discriminant = b \* b - 4 \* a \* c;
* let result;
* if (discriminant > 0) {
* const x1 = (-b + Math.sqrt(discriminant)) / (2 \* a);
* const x2 = (-b - Math.sqrt(discriminant)) / (2 \* a);
* result = { x1, x2 };
* } else if (discriminant === 0) {
* const x = -b / (2 \* a);
* result = { x };
* } else {
* result = 'No real roots';
* }
* // Отправляем результат в виде JSON
* res.json(result);
* // Сохраняем результат в файл
* const data = JSON.stringify({ url: req.originalUrl, response: result, method: 'GET' }, null, 2);
* fs.writeFile('lab\_1.json', data, (err) => {
* if (err) {
* console.error('Ошибка при сохранении файла:', err);
* } else {
* console.log('Файл сохранен');
* }
* });
* });
* // Запуск сервера на порту 3001
* app.listen(3001, () => {
* console.log('Сервер запущен на порту 3001');
* });

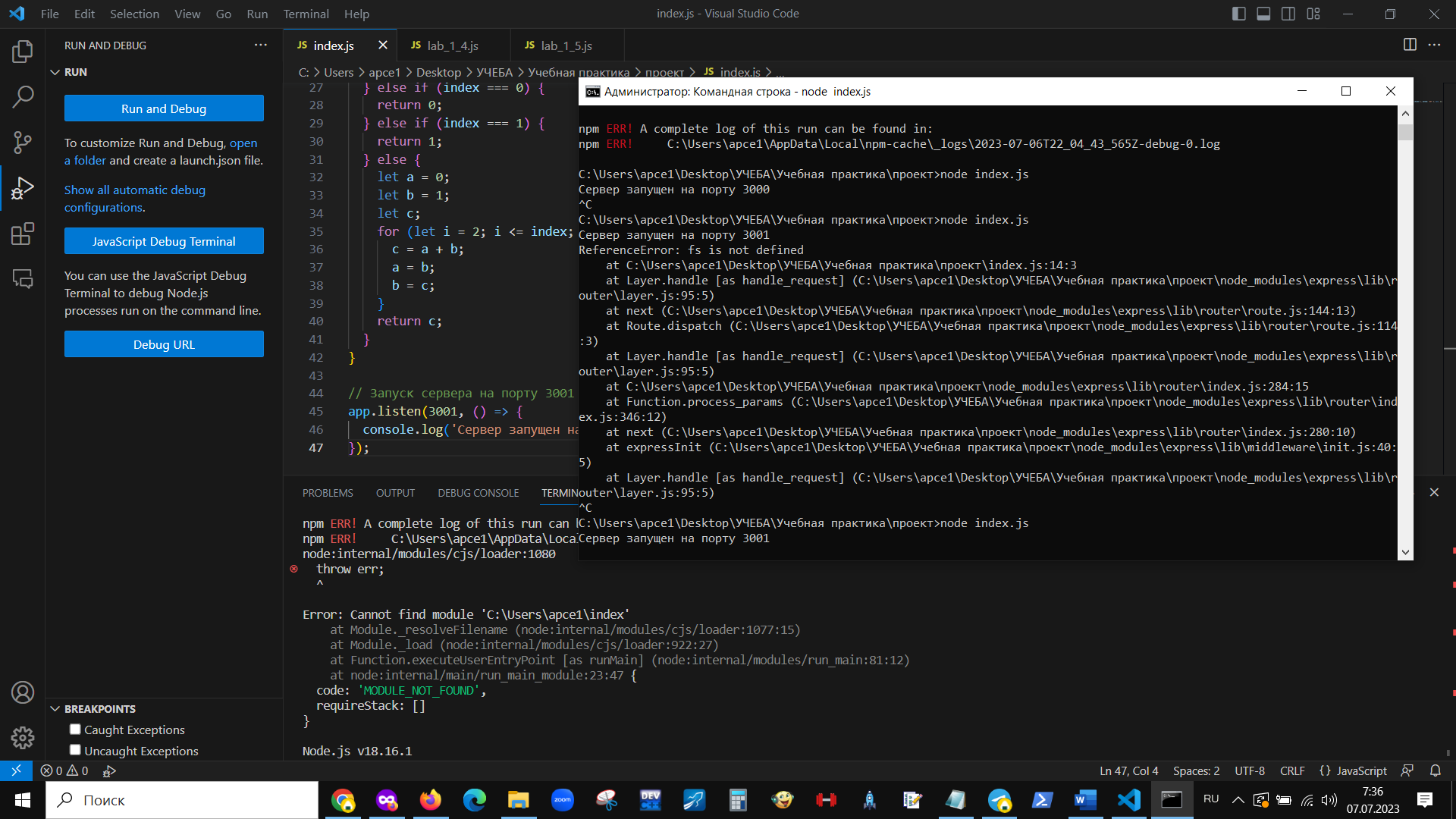
Результат  
<http://localhost:3001/?a=1&b=2&c=1>  


* Возвращает день недели по заданной дате, которая передается как параметр запроса (*url?date=12.12.2028*);
* const express = require('express');
* const app = express();
* // Обработчик GET-запроса для получения дня недели по заданной дате
* app.get('/', (req, res) => {
* const date = req.query.date;
* const dayOfWeek = getDayOfWeek(date);
* // Отправляем результат в виде JSON
* res.json({ dayOfWeek });
* // Сохраняем результат в файл
* const data = JSON.stringify({ url: req.originalUrl, response: { dayOfWeek }, method: 'GET' }, null, 2);
* fs.writeFile('lab\_1.json', data, (err) => {
* if (err) {
* console.error('Ошибка при сохранении файла:', err);
* } else {
* console.log('Файл сохранен');
* }
* });
* });
* // Функция для получения дня недели по заданной дате
* function getDayOfWeek(dateStr) {
* const date = new Date(dateStr);
* const daysOfWeek = ['Вс', 'Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб'];
* return daysOfWeek[date.getDay()];
* }
* // Запуск сервера на порту 3001
* app.listen(3001, () => {
* console.log('Сервер запущен на порту 3001');
* });

Результат:  
http://localhost:3001/?date=12.12.2028



* Возвращает число Фибоначчи, заданное своим индексом (*в виде параметра пути*);



const express = require('express');

const app = express();

// Обработчик GET-запроса для получения числа Фибоначчи по индексу

app.get('/:index', (req, res) => {

  const index = parseInt(req.params.index);

  const fibonacciNumber = getFibonacciNumber(index);

  // Отправляем результат в виде JSON

  res.json({ fibonacciNumber });

  // Сохраняем результат в файл

  const data = JSON.stringify({ url: req.originalUrl, response: { fibonacciNumber }, method: 'GET' }, null, 2);

  fs.writeFile('lab\_2.json', data, (err) => {

    if (err) {

      console.error('Ошибка при сохранении файла:', err);

    } else {

      console.log('Файл сохранен');

    }

  });

});

// Функция для получения числа Фибоначчи по индексу

function getFibonacciNumber(index) {

  if (index < 0) {

    return NaN;

  } else if (index === 0) {

    return 0;

  } else if (index === 1) {

    return 1;

  } else {

    let a = 0;

    let b = 1;

    let c;

    for (let i = 2; i <= index; i++) {

      c = a + b;

      a = b;

      b = c;

    }

    return c;

  }

}

// Запуск сервера на порту 3001

app.listen(3001, () => {

  console.log('Сервер запущен на порту 3001');

});

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

* Возвращает название региона России по заданному номеру региона (*в виде параметра пути*). Например, при заданном *url/72*, должно возвращаться «Тюменская область».

const express = require('express');

const app = express();

// Словарь номеров и названий регионов

const regions = {

  '01': 'Республика Адыгея',

  '02': 'Республика Башкортостан',

  '03': 'Республика Бурятия',

  '04': 'Республика Алтай',

  '05': 'Республика Дагестан',

  '06': 'Республика Ингушетия',

  '07': 'Кабардино-Балкарская Республика',

  '08': 'Республика Калмыкия',

  '09': 'Карачаево-Черкесская Республика',

  '10': 'Республика Карелия',

  '11': 'Республика Коми',

  '12': 'Республика Марий Эл',

  '13': 'Республика Мордовия',

  '14': 'Республика Саха (Якутия)',

  '15': 'Республика Северная Осетия - Алания',

  '16': 'Республика Татарстан',

  '17': 'Республика Тыва',

  '18': 'Удмуртская Республика',

  '19': 'Республика Хакасия',

  '20': 'Чувашская Республика',

  '21': 'Алтайский край',

  '22': 'Краснодарский край',

  '23': 'Красноярский край',

  '24': 'Приморский край',

  '25': 'Ставропольский край',

  '26': 'Хабаровский край',

  '27': 'Амурская область',

  '28': 'Архангельская область',

  '29': 'Астраханская область',

  '30': 'Белгородская область',

  '31': 'Брянская область',

  '32': 'Владимирская область',

  '72': 'Тюменская область'

};

// Обработчик GET-запроса для получения названия региона

app.get('/:regionNumber', (req, res) => {

  const regionNumber = req.params.regionNumber;

  const regionName = regions[regionNumber];

  if (regionName) {

    res.json({ region: regionName });

  } else {

    res.status(404).json({ error: 'Region not found' });

  }

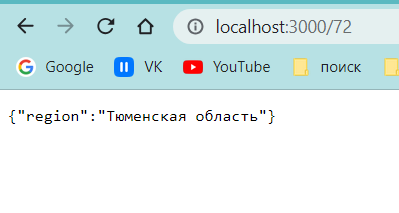
});

// Запуск сервера на порту 3000

app.listen(3000, () => {

  console.log('Сервер запущен на порту 3000');

});

 Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

**Лабораторная работа №2**

Разработать веб-службы для работы с базой данных.

Необходимо создать базу данных. В базе данных должны присутствовать следующие сущности: «**заявка**», «**клиент**» и «**услуга**». Заявка оформляется на одного клиента. Заявка может содержать несколько услуг.

Для выполнения задания, сначала создали базу данных и определили схемы для таблиц "заявка", "клиент" и "услуга".

Затем, создали RESTful API, используя Express.js для обработки CRUD-операций (создание, чтение, обновление и удаление) для каждой из трех таблиц.

**Шаг 1: Создание базы данных и таблиц**

1. Создали базу данных, используя СУБД, такую как PostgreSQL
2. Определили схемы для таблиц "заявка", "клиент" и "услуга". Ниже приведен пример схемы для каждой таблицы:

*-- Таблица "клиент"*

**CREATE** **TABLE** clients (

id SERIAL **PRIMARY** **KEY**,

name **VARCHAR**(255) **NOT** **NULL**,

email **VARCHAR**(255) **UNIQUE** **NOT** **NULL**

);

*-- Таблица "услуга"*

**CREATE** **TABLE** services (

id SERIAL **PRIMARY** **KEY**,

name **VARCHAR**(255) **NOT** **NULL**,

price **DECIMAL**(10, 2) **NOT** **NULL**

);

*-- Таблица "заявка"*

**CREATE** **TABLE** requests (

id SERIAL **PRIMARY** **KEY**,

client\_id **INT** **NOT** **NULL**,

service\_id **INT** **NOT** **NULL**,

created\_at **TIMESTAMP** **NOT** **NULL** **DEFAULT** NOW(),

**FOREIGN** **KEY** (client\_id) **REFERENCES** clients (id),

**FOREIGN** **KEY** (service\_id) **REFERENCES** services (id)

);

**Шаг 2: Создание API для работы с таблицами**

1. Установили express, body-parser, и cors с помощью npm

**npm install express body-parser cors**

1. Создали файл server.js и импортировали необходимые модули:

// Подключение необходимых модулей

const express = require('express'); // Модуль Express.js для создания сервера

const bodyParser = require('body-parser'); // Модуль для обработки данных запроса

const cors = require('cors'); // Модуль для обеспечения Cross-Origin Resource Sharing (CORS)

1. Создали объект приложения Express и настроили его:

// Создание экземпляра приложения Express

const app = express();

// Подключение middleware для обработки данных запроса в формате JSON

app.use(bodyParser.json());

// Подключение middleware для обеспечения Cross-Origin Resource Sharing (CORS)

app.use(cors());

1. Установил и настроил модуль для работы с базой данных, например pg для PostgreSQL или mysql2 для MySQL. Создал файл для определения моделей и запросов к базе данных - models.js.
2. Реализовал CRUD-операции для каждой из сущностей (клиент, услуга и заявка) в server.js. Вот пример реализации API для сущности "клиент":
3. // Добавить клиента
4. app.post('/api/clients', async (req, res) => {
5. // Вставка данных в таблицу клиентов
6. });
7. // Получить список клиентов
8. app.get('/api/clients', async (req, res) => {
9. // Получение списка клиентов
10. });
11. // Получить клиента по ID
12. app.get('/api/clients/:id', async (req, res) => {
13. // Получение клиента по ID
14. });
15. // Обновить клиента по ID
16. app.put('/api/clients/:id', async (req, res) => {
17. // Обновление данных клиента по ID
18. });
19. // Удалить клиента по ID
20. app.delete('/api/clients/:id', async (req, res) => {
21. // Удаление клиента по ID
22. });

6. Запустил сервер на определенном порту:

const PORT = process.env.PORT || 3001;

app.listen(PORT, () => {

    console.log(`Server is running on port ${PORT}`);

});

Для отправки HTTP-запросы на сервер, буду использовать HTTP-клиент Postman.

В PostgreSQL создал таблицы  
CREATE TABLE clients (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL

);

CREATE TABLE services (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL,

price NUMERIC NOT NULL

);

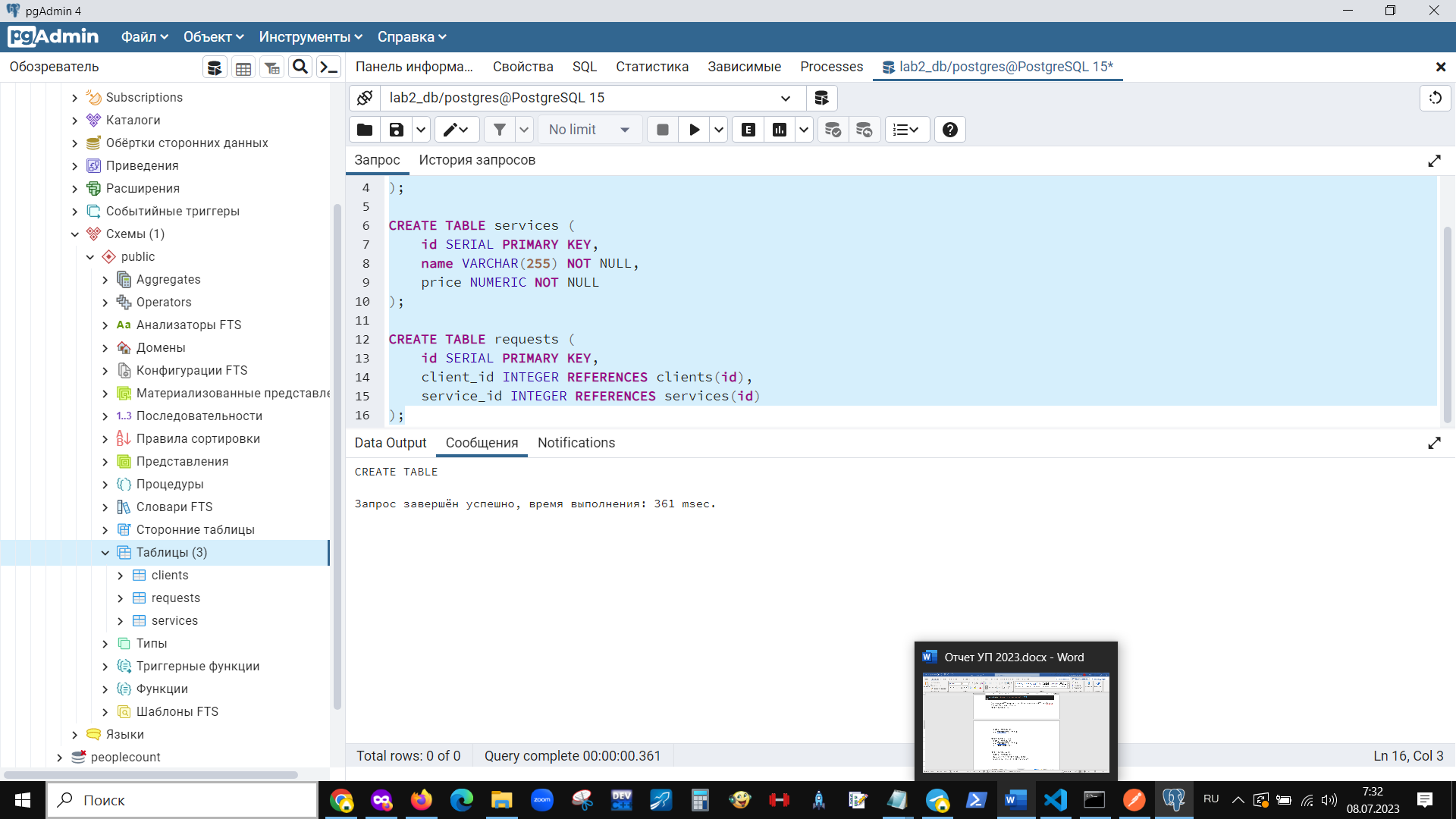
CREATE TABLE requests (

id SERIAL PRIMARY KEY,

client\_id INTEGER REFERENCES clients(id),

service\_id INTEGER REFERENCES services(id)

);



Server.js для создания сервера

const express = require('express');

const cors = require('cors');

const knex = require('knex');

const db = knex({

  client: 'pg',

  connection: {

    host: '127.0.0.1',

    user: 'postgres',

    password: '1',

    database: 'lab2\_db'

  }

});

const app = express();

app.use(cors());

app.use(express.json());

// Ваш код API будет здесь

// Получить список клиентов

app.get('/clients', async (req, res) => {

    try {

      const clients = await db.select('\*').from('clients');

      res.json(clients);

    } catch (err) {

      res.status(500).json({ error: 'Ошибка на сервере' });

    }

  });

  // Добавить нового клиента

  app.post('/clients', async (req, res) => {

    try {

      const { name } = req.body;

      const [id] = await db('clients').insert({ name }).returning('id');

      res.json({ id, name });

    } catch (err) {

      res.status(500).json({ error: 'Ошибка на сервере' });

    }

  });

  // Обновить клиента

  app.put('/clients/:id', async (req, res) => {

    try {

      const { id } = req.params;

      const { name } = req.body;

      await db('clients').where({ id }).update({ name });

      res.json({ id, name });

    } catch (err) {

      res.status(500).json({ error: 'Ошибка на сервере' });

    }

  });

  // Удалить клиента

  app.delete('/clients/:id', async (req, res) => {

    try {

      const { id } = req.params;

      await db('clients').where({ id }).del();

      res.json({id });

    } catch (err) {

      res.status(500).json({ error: 'Ошибка на сервере' });

    }

  });

app.listen(3000, () => {

  console.log('Server is running on port 3000');

});

Добавил код для создания API:

Index.js

const { Pool } = require('pg');

const express = require('express');

const app = express();

const pool = new Pool({

    user: 'your\_db\_user',

    host: 'localhost',

    database: 'your\_db\_name',

    password: 'your\_db\_password',

    port: 5432,

});

// Добавить клиента

app.post('/api/clients', async (req, res) => {

    try {

        const { name, email, phone } = req.body;

        const result = await pool.query('INSERT INTO clients (name, email, phone) VALUES ($1, $2, $3) RETURNING id', [name, email, phone]);

        res.status(201).json({ id: result.rows[0].id });

    } catch (err) {

        console.error(err);

        res.status(500).json({ error: 'An error occurred while adding a client' });

    }

});

// Получить список клиентов

app.get('/api/clients', async (req, res) => {

    try {

        const result = await pool.query('SELECT \* FROM clients');

        res.status(200).json(result.rows);

    } catch (err) {

        console.error(err);

        res.status(500).json({ error: 'An error occurred while fetching clients' });

    }

});

// Получить клиента по ID

app.get('/api/clients/:id', async (req, res) => {

    try {

        const { id } = req.params;

        const result = await pool.query('SELECT \* FROM clients WHERE id = $1', [id]);

        if (result.rowCount === 0) {

            res.status(404).json({ error: 'Client not found' });

            return;

        }

        res.status(200).json(result.rows[0]);

    } catch (err) {

        console.error(err);

        res.status(500).json({ error: 'An error occurred while fetching the client' });

    }

});

// Обновить клиента по ID

app.put('/api/clients/:id', async (req, res) => {

    try {

        const { id } = req.params;

        const { name, email, phone } = req.body;

        await pool.query('UPDATE clients SET name = $1, email = $2, phone = $3 WHERE id = $4', [name, email, phone, id]);

        res.status(200).json({ message: 'Client updated successfully' });

    } catch (err) {

        console.error(err);

        res.status(500).json({ error: 'An error occurred while updating the client' });

    }

});

// Удалить клиента по ID

app.delete('/api/clients/:id', async (req, res) => {

    try {

        const { id } = req.params;

        await pool.query('DELETE FROM clientsWHERE id = $1', [id]);

        res.status(200).json({ message: 'Client deleted successfully' });

    } catch (err) {

        console.error(err);

        res.status(500).json({ error: 'An error occurred while deleting the client' });

    }

});

**Создание frontend с использованием React**

npm install -g create-react-app

create-react-app lab2\_frontend

В итоге server.js и App.js выглядят следующим образом:

const express = require('express');

const cors = require('cors');

const knex = require('knex');

const db = knex({

  client: 'pg',

  connection: {

    host: '127.0.0.1',

    user: 'postgres',

    password: '1',

    database: 'lab2\_db'

  }

});

const app = express();

app.use(cors());

app.use(express.json());

app.get('/clients', async (req, res) => {

  try {

    const clients = await db.select('\*').from('clients');

    res.json(clients);

  } catch (err) {

    res.status(500).json({ error: 'Ошибка на сервере' });

  }

});

app.get('/services', async (req, res) => {

  try {

    const services = await db.select('\*').from('services');

    res.json(services);

  } catch (err) {

    res.status(500).json({ error: 'Ошибка на сервере' });

  }

});

app.get('/requests', async (req, res) => {

  try {

    const requests = await db.select('\*').from('requests');

    res.json(requests);

  } catch (err) {

    res.status(500).json({ error: 'Ошибка на сервере' });

  }

});

app.listen(3000, () => {

  console.log('Server is running on port 3000');

});

В итоге server и App выглядят следующим образом:  
  
import React, { useState, useEffect } from 'react';

import axios from 'axios';

function App() {

  const [clients, setClients] = useState([]);

  const [services, setServices] = useState([]);

  const [requests, setRequests] = useState([]);

  useEffect(() => {

    getClients();

    getServices();

    getRequests();

  }, []);

  const getClients = async () => {

    try {

      const res = await axios.get('http://localhost:3000/clients');

      setClients(res.data);

    } catch (err) {

      console.error(err);

    }

  };

  const getServices = async () => {

    try {

      const res = await axios.get('http://localhost:3000/services');

      setServices(res.data);

    } catch (err) {

      console.error(err);

    }

  };

  const getRequests = async () => {

    try {

      const res = await axios.get('http://localhost:3000/requests');

      setRequests(res.data);

    } catch (err) {

      console.error(err);

    }

  };

  return (

    <div>

      <h1>Клиенты</h1>

      <table>

        <thead>

          <tr>

            <th>ID</th>

            <th>Имя</th>

          </tr>

        </thead>

        <tbody>

          {clients.map((client) => (

            <tr key={client.id}>

              <td>{client.id}</td>

              <td>{client.name}</td>

            </tr>

          ))}

        </tbody>

      </table>

      <h1>Услуги</h1>

      <table>

        <thead>

          <tr>

            <th>ID</th>

            <th>Название</th>

            <th>Цена</th>

          </tr>

        </thead>

        <tbody>

          {services.map((service) => (

            <tr key={service.id}>

              <td>{service.id}</td>

              <td>{service.name}</td>

              <td>{service.price} руб.</td>

            </tr>

          ))}

        </tbody>

      </table>

      <h1>Заявки</h1>

      <table>

        <thead>

          <tr>

            <th>ID</th>

            <th>ID клиента</th>

            <th>ID услуги</th>

          </tr>

        </thead>

        <tbody>

          {requests.map((request) => (

            <tr key={request.id}>

              <td>{request.id}</td>

              <td>{request.client\_id}</td>

              <td>{request.service\_id}</td>

            </tr>

          ))}

        </tbody>

      </table>

    </div>

  );

}

export default App;

Результат работы веб-службы для работы с базой данных, при запуске localhost отображает следующее:  


**Лабораторная работа №3**

Разработать веб-службу, на основе фреймворка SOAP. Данная лабораторная выполняется на основе предыдущей лабораторной.

Критерии выполнения лабораторной:

* На основе фреймворка **SOAP**созданы операци на ***чтение/добавление/изменение/удаление*** записей для сущностей из лабораторной 2;
* В программном коде **отсутствуют** какие-либо SQL запросы.

Решение:

**Часть 1: Настройка окружения**

Для начала, был установлен JDK (Java Development Kit), набор утилит, необходимых для разработки программ на языке Java. Версия Java 11 или выше была скачана и установлена с официального сайта

Затем я скачал и установил IDE (Integrated Development Environment) - среду разработки. Для этих целей была выбрана IntelliJ IDEA, и она была загружена с официального сайта

Далее установка PostgreSQL - системы управления базами данных. Скачал установщик с официального сайта и завершил процесс установки.

**Часть 2: Создание базы данных**

После установки PostgreSQL, ввел учетные данные, которые были заданы при установке.

Для создания новой базы данных, я кликнул правой кнопкой мыши на "Databases" и выбрал "Create -> Database". Было присвоено имя "my\_database".

Затем, в созданной базе данных были выполнены SQL запросы, предоставленные для создания таблиц "clients", "services" и "requests".

**CREATE** **TABLE** clients (

id SERIAL **PRIMARY** **KEY**,

name **VARCHAR**(255) **NOT** **NULL**

);

**CREATE** **TABLE** services (

id SERIAL **PRIMARY** **KEY**,

name **VARCHAR**(255) **NOT** **NULL**,

price **NUMERIC** **NOT** **NULL**

);

**CREATE** **TABLE** requests (

id SERIAL **PRIMARY** **KEY**,

client\_id **INTEGER** **REFERENCES** clients(id),

service\_id **INTEGER** **REFERENCES** services(id)

);

Далее заполнил таблицы

*-- Заполнение таблицы "clients"*

**INSERT** **INTO** clients (name) **VALUES**

('John Doe'),

('Jane Smith'),

('Mike Johnson'),

('Emily Brown');

*-- Заполнение таблицы "services"*

**INSERT** **INTO** services (name, price) **VALUES**

('Web Development', 2500),

('Graphic Design', 1200),

('Data Analysis', 1800),

('Mobile App Development', 3000);

*-- Заполнение таблицы "requests"*

**INSERT** **INTO** requests (client\_id, service\_id) **VALUES**

(1, 3),

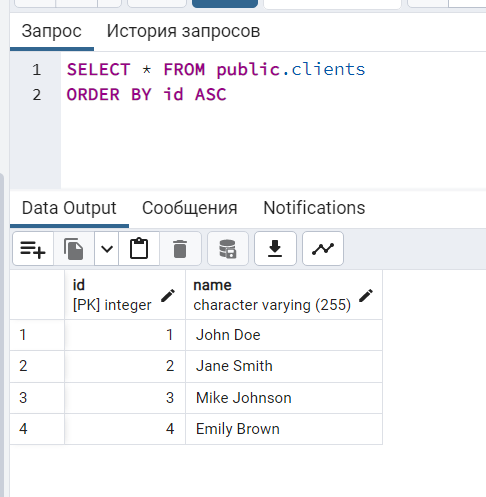
(2, 1),

(3, 2),

(4, 4),

(1, 2),

(3, 1);



**Часть 3: Создание Spring Boot проекта**

Я перешел на сайт Spring Initializr ↗.

Я выбрал следующие параметры:

Project: Maven Project

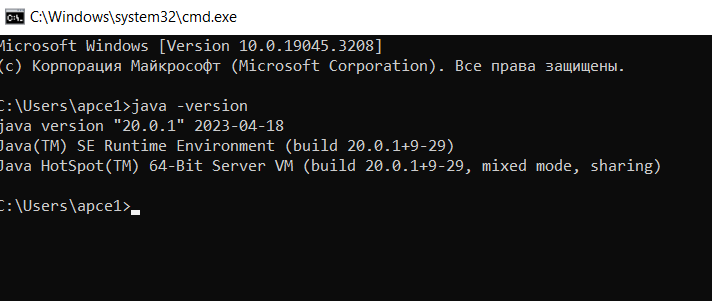
Language: Java

Spring Boot: [Latest Version]

Project Metadata: [Мои предпочтения]

Packaging: Jar

Java: 20

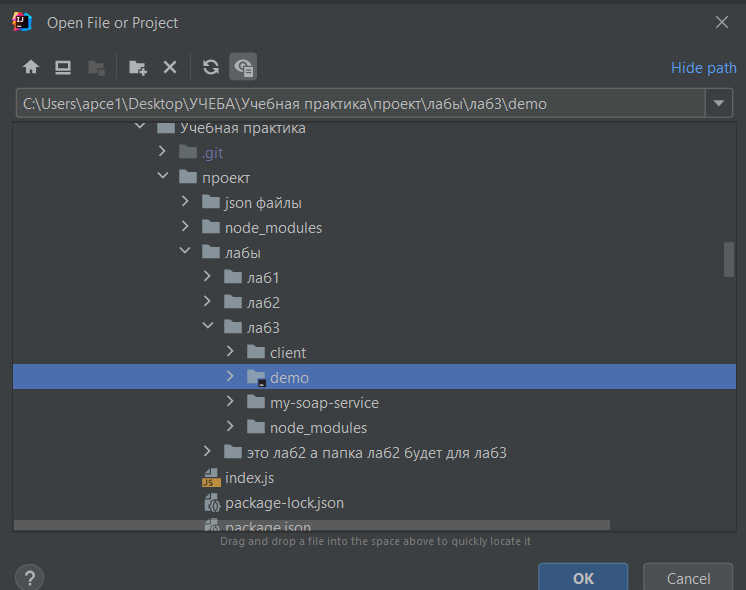


В разделе "Dependencies" я добавил следующие зависимости: "Spring Web", "Spring Data JPA", "PostgreSQL Driver", "Spring Web Services".

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Выбрал "Generate" и загрузил проект. Распаковал архив на моем компьютере и открыл проект в IntelliJ IDEA.



**Часть 4: Настройка подключения к базе данных**

В директории src/main/resources изменил файл application.properties:

spring.datasource.url=jdbc:postgresql://localhost:5432/arsenshmid  
spring.datasource.username=postgres  
spring.datasource.password=your\_1  
spring.jpa.hibernate.ddl-auto=update

**Часть 5: Разработка веб-службы**

Создание моделей данных В пакете "model" я создал классы для моделей данных "Client", "Service" и "Request". Эти классы представляют сущности таблиц базы данных.

Client.java

package com.example.demo.model;  
  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
  
@Entity  
public class Client {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 private String name;  
  
 // конструктор  
 public Client() {}  
  
 public Client(String name) {  
 this.name = name;  
 }  
  
 // геттеры и сеттеры  
 public Long getId() {  
 return id;  
 }  
  
 public void setId(Long id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

}

Service.java

package com.example.demo.model;  
  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
  
@Entity  
public class Service {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 private String name;  
 private double price;  
  
 // Constructors  
 public Service() {}  
  
 public Service(String name, double price) {  
 this.name = name;  
 this.price = price;  
 }  
  
  
 public Long getId() {  
 return id;  
 }  
  
 public void setId(Long id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 public void setPrice(double price) {  
 this.price = price;  
 }  
}

Request.java

package com.example.demo.model;  
  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
import javax.persistence.ManyToOne;  
  
@Entity  
public class Request {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
  
 @ManyToOne  
 private Client client;  
  
 @ManyToOne  
 private Service service;  
  
 // Constructors  
 public Request() {}  
  
 public Request(Client client, Service service) {  
 this.client = client;  
 this.service = service;  
 }  
  
  
 public Long getId() {  
 return id;  
 }  
  
 public void setId(Long id) {  
 this.id = id;  
 }  
  
 public Client getClient() {  
 return client;  
 }  
  
 public void setClient(Client client) {  
 this.client = client;  
 }  
  
 public Service getService() {  
 return service;  
 }  
  
 public void setService(Service service) {  
 this.service = service;  
 }  
}

Создание репозиториев В пакете "repository" я создал интерфейсы репозиториев для каждой модели данных. Репозитории обеспечивают взаимодействие с базой данных.

**"ClientRepository.java"**

**package** com.example.demo.repository;

**import** com.example.demo.model.Client;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** org.springframework.stereotype.Repository;

@Repository

**public** **interface** ClientRepository **extends** JpaRepository<Client, Long> {

}

**"ServiceRepository.java"**

**package** com.example.demo.repository;

**import** com.example.demo.model.Service;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** org.springframework.stereotype.Repository;

@Repository

**public** **interface** ServiceRepository **extends** JpaRepository<Service, Long> {

}

**"RequestRepository.java"**

**package** com.example.demo.repository;

**import** com.example.demo.model.Request;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** org.springframework.stereotype.Repository;

@Repository

**public** **interface** RequestRepository **extends** JpaRepository<Request, Long> {

}

Создание контроллера для веб-службы:

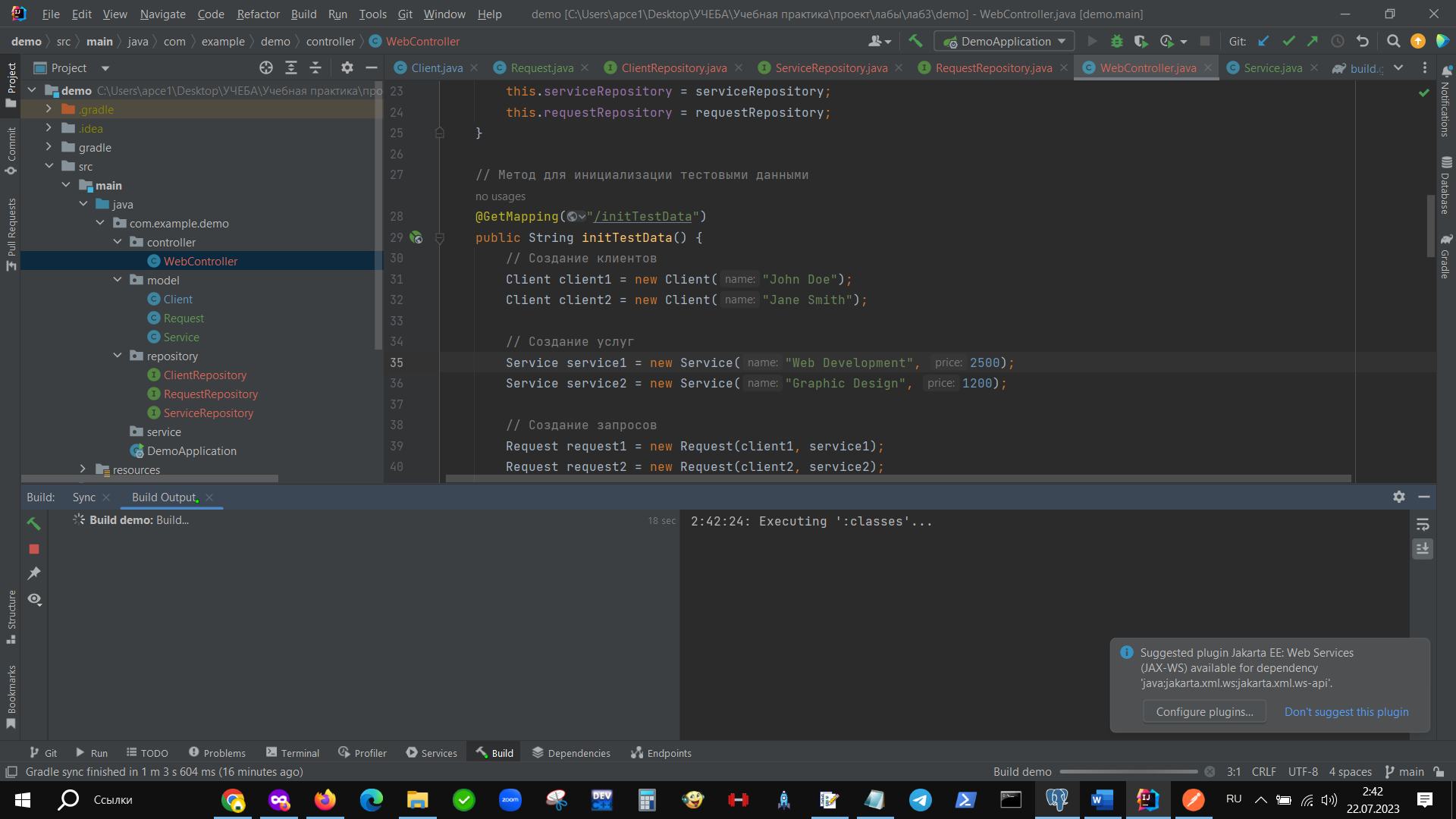
Внутри папки "src/main/java" я создал пакет "com.example.demo.controller". В этом пакете создал контроллер для обработки запросов, связанных с операциями чтения/добавления/изменения/удаления записей.

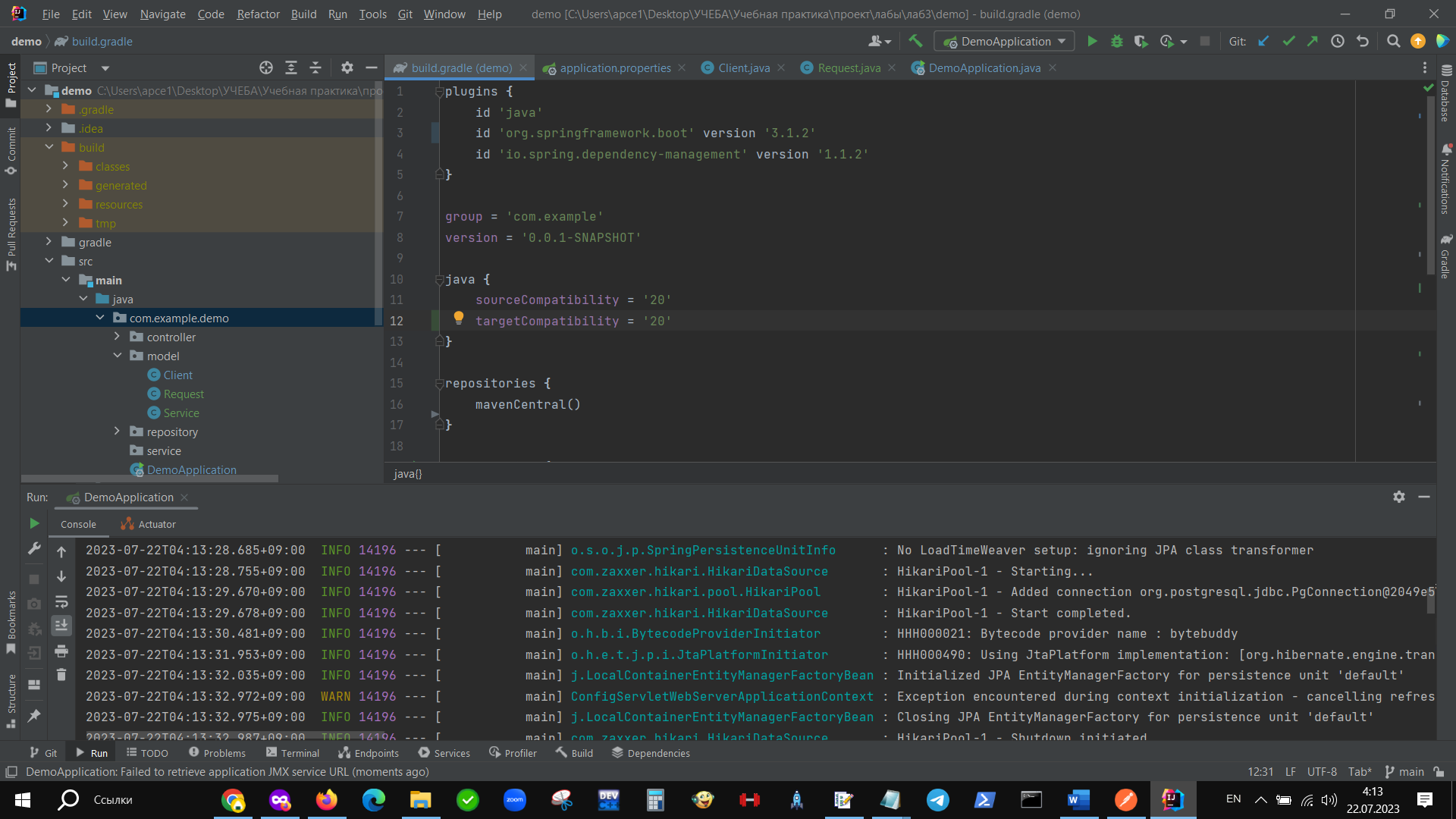
package com.example.demo.controller;  
  
import com.example.demo.model.Client;  
import com.example.demo.model.Request;  
import com.example.demo.model.Service;  
import com.example.demo.repository.ClientRepository;  
import com.example.demo.repository.RequestRepository;  
import com.example.demo.repository.ServiceRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
public class WebController {  
 private final ClientRepository clientRepository;  
 private final ServiceRepository serviceRepository;  
 private final RequestRepository requestRepository;  
  
 @Autowired  
 public WebController(ClientRepository clientRepository, ServiceRepository serviceRepository, RequestRepository requestRepository) {  
 this.clientRepository = clientRepository;  
 this.serviceRepository = serviceRepository;  
 this.requestRepository = requestRepository;  
 }  
  
 // Метод для инициализации тестовыми данными  
 @GetMapping("/initTestData")  
 public String initTestData() {  
 // Создание клиентов  
 Client client1 = new Client("John Doe");  
 Client client2 = new Client("Jane Smith");  
  
 // Создание услуг  
 Service service1 = new Service("Web Development", 2500);  
 Service service2 = new Service("Graphic Design", 1200);  
  
 // Создание запросов  
 Request request1 = new Request(client1, service1);  
 Request request2 = new Request(client2, service2);  
  
 // Сохранение данных в базу данных  
 clientRepository.save(client1);  
 clientRepository.save(client2);  
  
 serviceRepository.save(service1);  
 serviceRepository.save(service2);  
  
 requestRepository.save(request1);  
 requestRepository.save(request2);  
  
 return "Test data initialized successfully!";  
 }  
  
 // Примеры других методов для операций чтения/добавления/изменения/удаления записей:  
  
 // Получение всех клиентов  
 @GetMapping("/clients")  
 public List<Client> getAllClients() {  
 return clientRepository.findAll();  
 }  
  
 // Добавление нового клиента  
 @PostMapping("/clients")  
 public Client addNewClient(@RequestBody Client client) {  
 return clientRepository.save(client);  
 }  
  
 // Получение всех услуг  
 @GetMapping("/services")  
 public List<Service> getAllServices() {  
 return serviceRepository.findAll();  
 }  
  
 // Изменение информации об услуге по ID  
 @PutMapping("/services/{id}")  
 public Service updateService(@PathVariable Long id, @RequestBody Service updatedService) {  
 Service service = serviceRepository.findById(id).orElse(null);  
 if (service != null) {  
 service.setName(updatedService.getName());  
 service.setPrice(updatedService.getPrice());  
 return serviceRepository.save(service);  
 }  
 return null;  
 }  
  
 // Удаление услуги по ID  
 @DeleteMapping("/services/{id}")  
 public String deleteService(@PathVariable Long id) {  
 Service service = serviceRepository.findById(id).orElse(null);  
 if (service != null) {  
 serviceRepository.delete(service);  
 return "Service with ID " + id + " deleted successfully!";  
 }  
 return "Service with ID " + id + " not found!";  
 }  
}

**чтобы работало, добавил следующие зависимости в gradle:**

plugins **{** id 'java'  
 id 'org.springframework.boot' version '3.1.2'  
 id 'io.spring.dependency-management' version '1.1.2'  
**}**group = 'com.example'  
version = '0.0.1-SNAPSHOT'  
  
java **{** sourceCompatibility = '20'  
**}**repositories **{** mavenCentral()  
**}**dependencies **{** implementation 'javax.persistence:javax.persistence-api:2.2'  
 implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
 implementation 'org.springframework.boot:spring-boot-starter-web'  
 implementation 'org.springframework.boot:spring-boot-starter-web-services'  
 runtimeOnly 'org.postgresql:postgresql'  
 testImplementation 'org.springframework.boot:spring-boot-starter-test'  
**}**tasks.named('test') **{** useJUnitPlatform()  
**}**

**Часть 6: Тестирование веб-службы**





Чтобы протестировать работоспособность веб-службы, использовал Postman, для отправки HTTP-запросов и проверки ответов.

**Примеры тестовых запросов**:

* GET: http://localhost:8080/initTestData (для инициализации тестовыми данными)
* GET: http://localhost:8080/clients (получить всех клиентов)
* POST: http://localhost:8080/clients (добавить нового клиента)
* GET: http://localhost:8080/services (получить все услуги)
* PUT: http://localhost:8080/services/{id} (изменить информацию об услуге с указанным id)
* DELETE: http://localhost:8080/services/{id} (удалить услугу с указанным id)

**Часть 7. Создание SOAP веб-службы**

Для создания SOAP веб-службы, добавил зависимость "spring-ws-core" в файл build.gradle:

Gradle:

dependencies {

...

implementation 'org.springframework.boot:spring-boot-starter-web-services'

...

}

Затем создал XSD схему для определения структуры SOAP сообщений. В папке src/main/resources создадим файл "services.xsd":

**<xs:schema** xmlns:xs="http://www.w3.org/2001/XMLSchema"

xmlns:tns="http://www.example.com/services"

targetNamespace="http://www.example.com/services"

elementFormDefault="qualified"**>**

**<xs:element** name="GetServiceRequest"**>**

**<xs:complexType>**

**<xs:sequence>**

**<xs:element** name="id" type="xs:long"**/>**

**</xs:sequence>**

**</xs:complexType>**

**</xs:element>**

**<xs:element** name="GetServiceResponse"**>**

**<xs:complexType>**

**<xs:sequence>**

**<xs:element** name="Service" type="tns:Service"**/>**

**</xs:sequence>**

**</xs:complexType>**

**</xs:element>**

**<xs:complexType** name="Service"**>**

**<xs:sequence>**

**<xs:element** name="id" type="xs:long"**/>**

**<xs:element** name="name" type="xs:string"**/>**

**<xs:element** name="price" type="xs:double"**/>**

**</xs:sequence>**

**</xs:complexType>**

**</xs:schema>**

Далее, создадим классы, которые будут использоваться для обработки SOAP запросов и ответов. В пакете "com.example.demo.soap" создадим класс "ServiceEndpoint":

**package** com.example.demo.soap;

**import** com.example.demo.model.Service;

**import** com.example.demo.repository.ServiceRepository;

**import** com.example.services.GetServiceRequest;

**import** com.example.services.GetServiceResponse;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.ws.server.endpoint.annotation.Endpoint;

**import** org.springframework.ws.server.endpoint.annotation.PayloadRoot;

**import** org.springframework.ws.server.endpoint.annotation.RequestPayload;

**import** org.springframework.ws.server.endpoint.annotation.ResponsePayload;

@Endpoint

**public** **class** ServiceEndpoint {

**private** **static** **final** String NAMESPACE\_URI = "http://www.example.com/services";

**private** **final** ServiceRepository serviceRepository;

@Autowired

**public** ServiceEndpoint(ServiceRepository serviceRepository) {

**this**.serviceRepository = serviceRepository;

}

@PayloadRoot(namespace = NAMESPACE\_URI, localPart = "GetServiceRequest")

@ResponsePayload

**public** GetServiceResponse getService(@RequestPayload GetServiceRequest request) {

GetServiceResponse response = **new** GetServiceResponse();

Service service = serviceRepository.findById(request.getId()).orElse(**null**);

**if** (service != **null**) {

com.example.services.Service soapService = **new** com.example.services.Service();

soapService.setId(service.getId());

soapService.setName(service.getName());

soapService.setPrice(service.getPrice());

response.setService(soapService);

}

**return** response;

}

}

Теперь, SOAP веб-служба готова к использованию. Тестировал её с помощью SOAP UI. Пример SOAP запроса для проверки работы веб-службы:

**<soapenv:Envelope** xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ser="http://www.example.com/services"**>**

**<soapenv:Header/>**

**<soapenv:Body>**

**<ser:GetServiceRequest>**

**<ser:id>**1**</ser:id>**

**</ser:GetServiceRequest>**

**</soapenv:Body>**

**</soapenv:Envelope>**

В ответ на этот запрос, веб-служба вернула информацию об услуге с указанным id.

Еще примеры запросов, которые я протестировал после создания веб-службы:  
Пример 1: Запрос на получение всех клиентов

SOAP запрос:

xml

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ser="http://www.example.com/services">

<soapenv:Header/>

<soapenv:Body>

<ser:GetAllClientsRequest/>

</soapenv:Body>

</soapenv:Envelope>

SOAP ответ:

xml

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ser="http://www.example.com/services">

<soapenv:Header/>

<soapenv:Body>

<ser:GetAllClientsResponse>

<ser:Client>

<ser:id>1</ser:id>

<ser:name>John Doe</ser:name>

</ser:Client>

<ser:Client>

<ser:id>2</ser:id>

<ser:name>Jane Smith</ser:name>

</ser:Client>

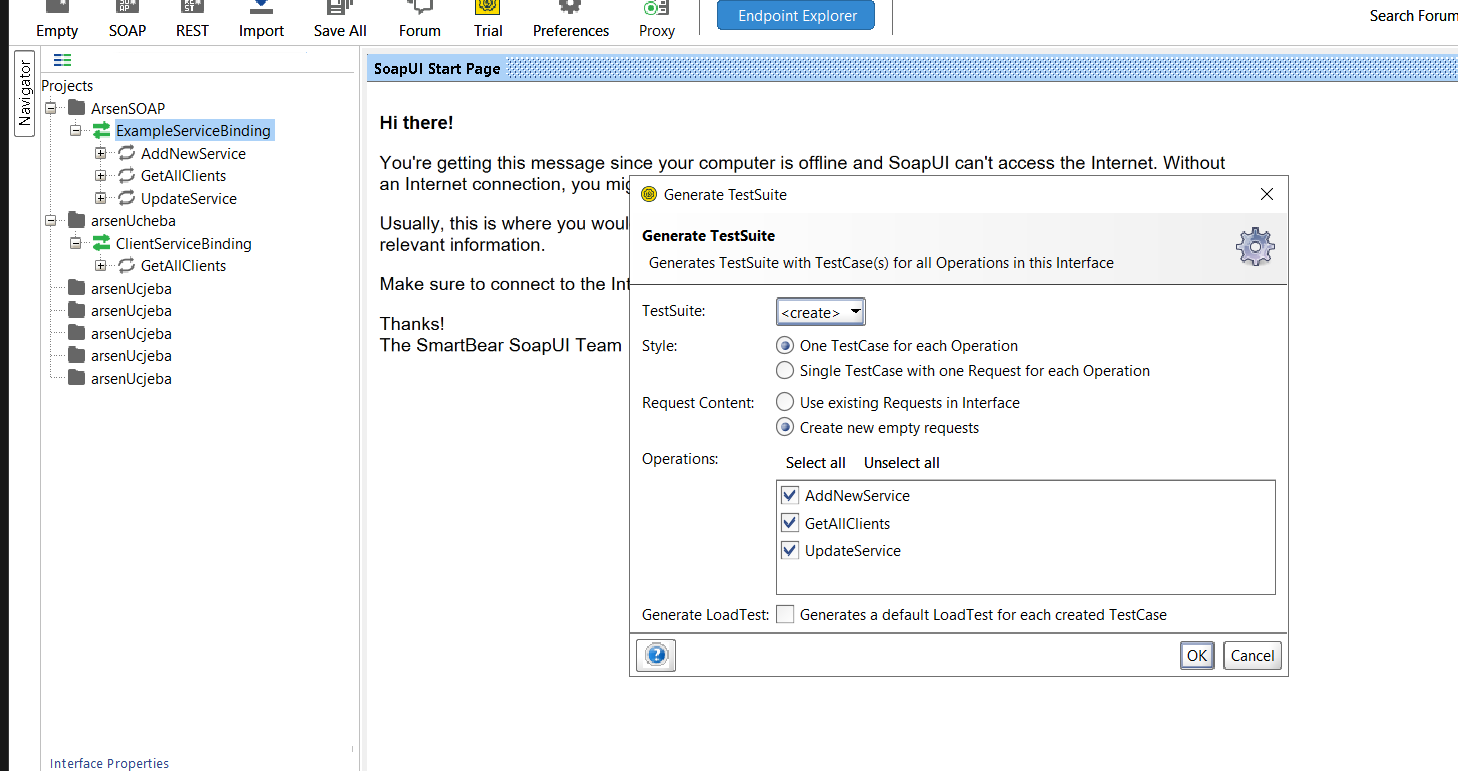
<!-- Другие клиенты -->

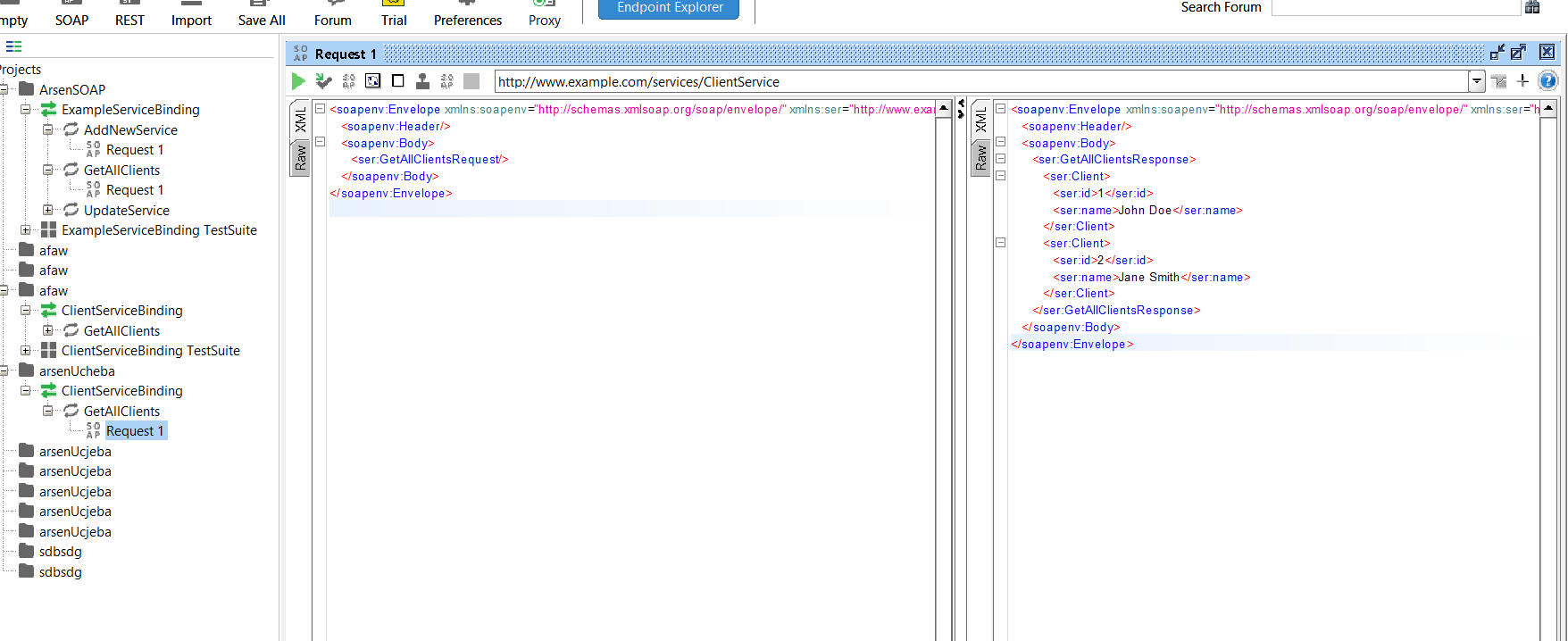
</ser:GetAllClientsResponse>

</soapenv:Body>

</soapenv:Envelope>

Для результата использовал SOAPUI, куда надо прописать путь к wsdl





Пример 2: Запрос на добавление новой услуги

SOAP запрос:

xml

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ser="http://www.example.com/services">

<soapenv:Header/>

<soapenv:Body>

<ser:AddNewServiceRequest>

<ser:name>Web Design</ser:name>

<ser:price>1800</ser:price>

</ser:AddNewServiceRequest>

</soapenv:Body>

</soapenv:Envelope>

SOAP ответ:

xml

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ser="http://www.example.com/services">

<soapenv:Header/>

<soapenv:Body>

<ser:AddNewServiceResponse>

<ser:serviceAdded>true</ser:serviceAdded>

</ser:AddNewServiceResponse>

</soapenv:Body>

</soapenv:Envelope>

Изображение выглядит как текст, снимок экрана, линия

Автоматически созданное описание

Пример 3: Запрос на изменение информации об услуге

SOAP запрос:

xml

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ser="http://www.example.com/services">

<soapenv:Header/>

<soapenv:Body>

<ser:UpdateServiceRequest>

<ser:id>3</ser:id>

<ser:name>Mobile App Development</ser:name>

<ser:price>3500</ser:price>

</ser:UpdateServiceRequest>

</soapenv:Body>

</soapenv:Envelope>

SOAP ответ:

xml

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ser="http://www.example.com/services">

<soapenv:Header/>

<soapenv:Body>

<ser:UpdateServiceResponse>

<ser:serviceUpdated>true</ser:serviceUpdated>

</ser:UpdateServiceResponse>

</soapenv:Body>

</soapenv:Envelope>

Изображение выглядит как текст, линия, снимок экрана

Автоматически созданное описание

**Лабораторная работа №4**

### Шаг 1: Настройка окружения

Для начала работы было необходимо убедиться, что на компьютере установлен Node.js и npm (Node Package Manager). Если Node.js отсутствовал, то я скачал и установил его с официального сайта: <https://nodejs.org/>

### Шаг 2: Создание проекта

1. Открыл командную строку и перешел в папку, где планировал создать проект:

bash

cd C:\Users\apce1\Desktop\УЧЕБА\Учебная практика\проект\лабы\лаб44

1. Создал новый проект React с помощью Create React App:

lua

npx create-react-app client

1. Затем перешел в папку проекта React:

bash

cd client

### Шаг 3: Настройка серверной части

1. Вернулся в корневую папку проекта и создал папку для серверной части:

bash

cd ..

mkdir server

cd server

1. Инициализировал проект Node.js в папке сервера:

csharp

npm init -y

1. Установил Express и axios для сервера:

npm install express axios

### Шаг 4: Настройка Express сервера

1. В папке сервера (server) создал файл server.js и добавил в него код для настройки Express сервера.
2. const express = require('express');
3. const axios = require('axios');
4. const app = express();
5. const port = 5000;
6. // Middleware для разрешения CORS (если клиент и сервер работают на разных доменах)
7. app.use((req, res, next) => {
8. res.header('Access-Control-Allow-Origin', '\*');
9. next();
10. });
11. // Роут для обработки запроса от клиента
12. app.get('/api/data', async (req, res) => {
13. try {
14. const url = 'http://www.mocky.io/v2/5c7db5e13100005a00375fda';
15. const response = await axios.get(url);
16. const modifiedData = {};
17. // Заменяем пробелы на нижнее подчёркивание в данных из внешнего API
18. for (const key in response.data) {
19. modifiedData[key] = response.data[key].replace(/ /g, '\_');
20. }
21. res.json(modifiedData);
22. } catch (error) {
23. console.error('Error fetching data:', error);
24. res.status(500).json({ error: 'Failed to fetch data from external API.' });
25. }
26. });
27. app.listen(port, () => {
28. console.log(`Server is running on http://localhost:${port}`);
29. });

### Шаг 5: Настройка клиентской части

1. Вернулся в папку клиента (client) и установил axios для клиента:

cd ../client

npm install axios

1. Отредактировал файл src/App.js в папке клиента (client/src) и заменил его содержимое на следующий код:
2. import React, { useState, useEffect } from 'react';
3. import axios from 'axios';
4. function App() {
5. const [data, setData] = useState({});
6. useEffect(() => {
7. // Вызываем серверную часть для получения данных и замены пробелов
8. axios
9. .get('http://localhost:5000/api/data')
10. .then((response) => setData(response.data))
11. .catch((error) => console.error('Error fetching data:', error));
12. }, []);
13. return (
14. <div>
15. <h1>Modified Data from External API</h1>
16. <pre>{JSON.stringify(data, null, 2)}</pre>
17. </div>
18. );
19. }
20. export default App;

### Шаг 6: Запуск проекта

1. В папке сервера (server) запустил Express сервер:

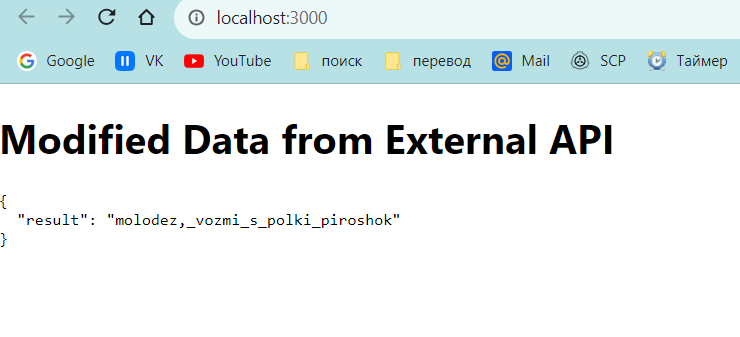
node server.js

1. В папке клиента (client) запустил React приложение:

npm start

Изображение выглядит как текст, программное обеспечение, Мультимедийное программное обеспечение, Шрифт

Автоматически созданное описание



# **Результат работы:**

{

"result": "molodez,\_vozmi\_s\_polki\_piroshok"

}

**Лабораторная работа №5**

Необходимо разработать веб-приложение, которое будет осуществлять функции, описанные в сценариях, представленных в ниже. В данная лабораторная работа продолжает **лабораторную 2**и **лаборатоную 4**.

Приложение должно включать серверную часть (back-end) и клиентскую часть (front-end). **Back-end** необходимо реализовать с использованием **REST**сервисов. **Front-end** приложение может быть разработано при помощи любой технологии. Клиентская часть должна вызывать

Критерии выполнения лабораторной:

* Создано клиентское приложение для сущностей из **лабораторной 2**;
* В клиентском приложении выполняется вызов **APIs**, которые были разработаны в **лабораторной 2**. Вызов **REST API**программно выполняется в **лабораторной 4.**

Отчет:

Программа разработки веб-приложения для лабораторной работы №5

Для выполнения лабораторной работы №5, я решил разработать веб-приложение с клиентской частью на React и серверной частью на Express Framework. Приложение должно взаимодействовать с базой данных, которая содержит сущности "заявка", "клиент" и "услуга". Кроме того, требуется использовать REST сервисы для обращения к базе данных и для вызова другого API, которое будет заменять пробелы на нижнее подчёркивание.

В рамках лабораторной работы №5 было разработано веб-приложение, которое осуществляет функции, описанные в сценариях из лабораторных работ №2 и №4. Проект выполнен с использованием следующих технологий:

* Front-end: React
* Back-end: Express Framework

**Шаги выполнения лабораторной работы:**

1. Создание проекта и структуры директорий: Проект был создан в папке C:\Users\apce1\Desktop\УЧЕБА\Учебная практика\проект\лабы\лаб5. В ней были созданы две подпапки - client для клиентской части и server для серверной части.
2. Реализация базы данных: Была создана база данных arsenshmid с тремя таблицами: clients, services, и requests. Таблицы были созданы согласно условиям лабораторной работы №2.
3. Создание серверной части (Back-end): В папке server был создан файл server.js, в котором реализован сервер с помощью Express Framework. При помощи библиотеки Knex, произведено подключение к базе данных PostgreSQL. Были созданы два API-маршрута:
   * GET /api/clients - для получения списка клиентов из базы данных;
   * POST /api/clients - для добавления нового клиента в базу данных.
4. Создание клиентской части (Front-end): В папке client был создан файл App.js, который содержит основную логику клиентской части приложения. С помощью React была реализована возможность добавления новых клиентов и получения списка всех клиентов из базы данных.
5. Проксирование запросов: В файле client/package.json был добавлен параметр proxy, который проксирует запросы клиента на серверную часть. Таким образом, клиентское приложение обращается к серверу по адресу http://localhost:3001.
6. Реализация отображения списка клиентов: Был создан компонент ClientList.js, который отображает список клиентов полученных с сервера. Этот компонент был использован в основном компоненте App.js для отображения списка клиентов.

**Результаты выполнения лабораторной работы:**

* Было разработано веб-приложение с использованием React и Express Framework, которое включает клиентскую и серверную части.
* Реализованы API-маршруты для работы с базой данных PostgreSQL, позволяющие получать список клиентов и добавлять новых клиентов.
* Клиентская часть успешно взаимодействует с сервером, получая список клиентов и добавляя новых.
* Применено проксирование запросов в клиентской части для обращения к серверу.

**Код программ:**

Sql запросы для базы данных lab2\_db

*-- Создание таблицы "заявка"*

**CREATE** **TABLE** заявка (

id **INT** **PRIMARY** **KEY** **AUTO\_INCREMENT**,

клиент\_id **INT** **NOT** **NULL**,

название **VARCHAR**(255) **NOT** **NULL**,

описание TEXT,

**FOREIGN** **KEY** (клиент\_id) **REFERENCES** клиент(id)

);

*-- Создание таблицы "клиент"*

**CREATE** **TABLE** клиент (

id **INT** **PRIMARY** **KEY** **AUTO\_INCREMENT**,

имя **VARCHAR**(100) **NOT** **NULL**,

телефон **VARCHAR**(20),

электронная\_почта **VARCHAR**(100)

);

*-- Создание таблицы "услуга"*

**CREATE** **TABLE** услуга (

id **INT** **PRIMARY** **KEY** **AUTO\_INCREMENT**,

название **VARCHAR**(255) **NOT** **NULL**,

описание TEXT

);

**App.js**

import React, { useState, useEffect } from 'react';

import axios from 'axios';

import ClientList from './components/ClientList';

function App() {

  const [newClientName, setNewClientName] = useState('');

  const [clients, setClients] = useState([]);

  const handleAddClient = () => {

    if (newClientName.trim() === '') return;

    axios.post('/api/clients', { name: newClientName })

      .then((response) => {

        setClients([...clients, response.data]);

        setNewClientName('');

      })

      .catch((error) => console.error('Error adding client', error));

  };

  const handleGetClients = () => {

    axios.get('/api/clients')

      .then((response) => setClients(response.data))

      .catch((error) => console.error('Error fetching clients', error));

  };

  useEffect(() => {

    handleGetClients();

  }, []); // Добавили пустой массив зависимостей, чтобы выполнить это только при первом рендеринге

  return (

    <div className="App">

      <h1>Client List</h1>

      <ClientList clients={clients} />

      <div>

        <input

          type="text"

          value={newClientName}

          onChange={(e) => setNewClientName(e.target.value)}

        />

        <button onClick={handleAddClient}>Add Client</button>

        <button onClick={handleGetClients}>Get Clients</button> {/\* Добавили кнопку \*/}

      </div>

    </div>

  );

}

export default App;

**server.js**

const express = require('express');

const knex = require('knex');

const app = express();

const port = 3001;

const db = knex({

  client: 'pg',

  connection: {

    host: '127.0.0.1',

    user: 'postgres',

    password: '1',

    database: 'lab2\_db'

  }

});

app.use(express.json());

app.get('/api/clients', async (req, res) => {

  try {

    const clients = await db.select().from('clients');

    res.json(clients);

  } catch (err) {

    console.error('Error executing query', err);

    res.status(500).json({ error: 'Internal server error' });

  }

});

app.post('/api/clients', async (req, res) => {

  const { name } = req.body;

  if (!name) {

    return res.status(400).json({ error: 'Name is required' });

  }

  try {

    const newClient = await db('clients').insert({ name }).returning('\*');

    res.status(201).json(newClient[0]);

  } catch (err) {

    console.error('Error executing query', err);

    res.status(500).json({ error: 'Internal server error' });

  }

});

app.listen(port, () => {

  console.log(`Server running on http://localhost:${port}`);

});

**ClientList.js**

import React from 'react';

function ClientList({ clients }) {

  return (

    <div>

      <h2>Client List</h2>

      <ul>

        {clients.map((client) => (

          <li key={client.id}>{client.name}</li>

        ))}

      </ul>

    </div>

  );

}

export default ClientList;

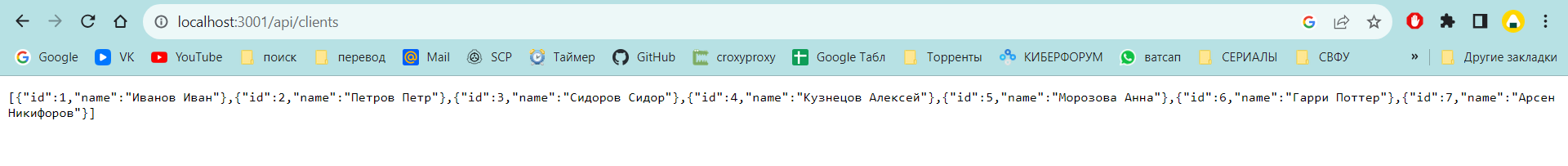
Можно проверить подключение следующим образом

http://localhost:3001/api/clients

Изображение выглядит как текст, снимок экрана, Шрифт, вода

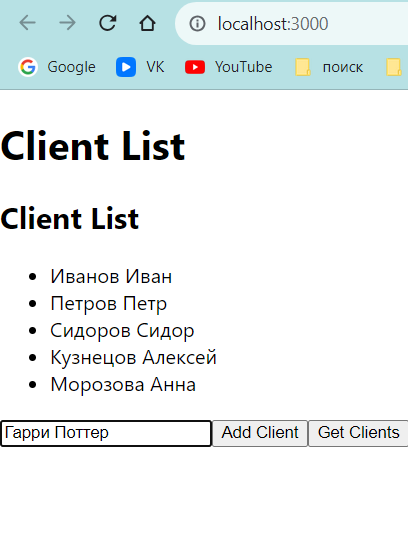
Автоматически созданное описание

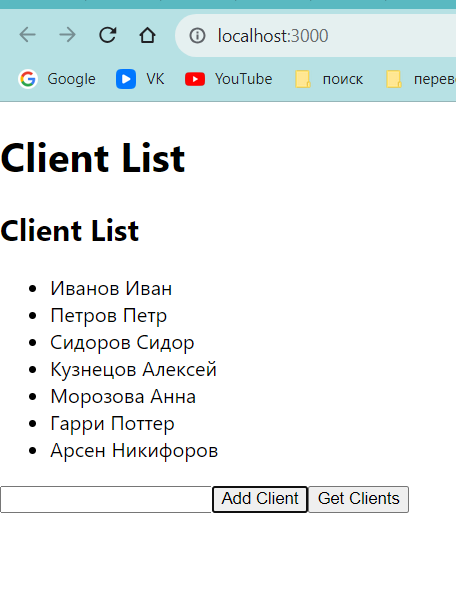
После добавления новых клиентов



[{"id":1,"name":"Иванов Иван"},{"id":2,"name":"Петров Петр"},{"id":3,"name":"Сидоров Сидор"},{"id":4,"name":"Кузнецов Алексей"},{"id":5,"name":"Морозова Анна"},{"id":6,"name":"Гарри Поттер"},{"id":7,"name":"Арсен Никифоров"}]

Запуск приложения





Добавление нового клиента

Новый пользователь добавлен в базу данных

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

App.js  
import React, { useState, useEffect } from 'react';

import axios from 'axios';

import ClientList from './components/ClientList';

function App() {

  const [newClientName, setNewClientName] = useState('');

  const [clients, setClients] = useState([]);

  const [mockData, setMockData] = useState(null);

  const [modifiedMockData, setModifiedMockData] = useState(null);

  const handleAddClient = () => {

    if (newClientName.trim() === '') return;

    axios.post('/api/clients', { name: newClientName })

      .then((response) => {

        setClients([...clients, response.data]);

        setNewClientName('');

      })

      .catch((error) => console.error('Error adding client', error));

  };

  const handleGetClients = () => {

    axios.get('/api/clients')

      .then((response) => setClients(response.data))

      .catch((error) => console.error('Error fetching clients', error));

  };

  const handleGetMockData = () => {

    axios.get('http://www.mocky.io/v2/5c7db5e13100005a00375fda')

      .then((response) => {

        setMockData(response.data);

        const modifiedData = response.data.replace(/ /g, '\_');

        setModifiedMockData(modifiedData);

      })

      .catch((error) => console.error('Error fetching mock data', error));

  };

  useEffect(() => {

    handleGetClients();

  }, []);

  return (

    <div className="App">

      <h1>Client List</h1>

      <ClientList clients={clients} />

      {mockData && (

        <div>

          <h2>Mock Data</h2>

          <p>{JSON.stringify(mockData)}</p>

          <h2>Modified Mock Data</h2>

          <p>{JSON.stringify(modifiedMockData)}</p>

        </div>

      )}

      <div>

        <input

          type="text"

          value={newClientName}

          onChange={(e) => setNewClientName(e.target.value)}

        />

        <button onClick={handleAddClient}>Add Client</button>

        <button onClick={handleGetClients}>Get Clients</button>

        <button onClick={handleGetMockData}>Get Mock Data</button>

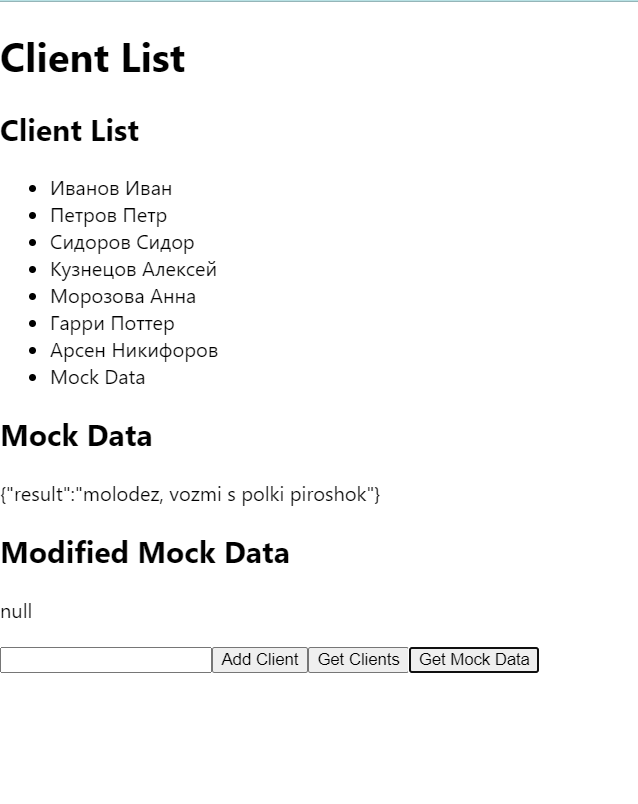
      </div>

    </div>

  );

}

export default App;

 Добавил отображение измененного mock  


Теперь при нажатии на кнопку "Get Mock Data" и успешном получении данных, они будут отображаться как исходный ответ, так и измененный с заменой пробелов на нижнее подчеркивание.

Итоговая лабораторная работа.

Необходимо реализовать серверное и клиентское приложения.

Сценарий работы системы тестирования:

**Добавление теста.**

1. Пользователь открывает главное окно программы.
2. Пользователь открывает окно добавления теста.
3. Пользователь вводит название теста.
4. Система открывает окно добавления вопросов к тесту.
5. Пользователь вводит вводит вопрос к тесту.
6. Система в том же окне показывает две пустые строки, в которые нужно ввести два варианта ответа к вопросу.
7. При вводе второго и последующих вариантов ответа, в окно добавляется новая строка для следующего варианта ответа.
8. После ввода всех вариантов ответа, пользователь напротив строки с правильным ответом отмечает CheckBox с правильным вариантом ответа.
9. Пользователь нажимает "**Готово**" и возвращается в окно ввода вопросов к тесту.
10. При необходимости пользователь нажимает на кнопку введения нового вопроса и открывается окно для ввода следующего вопроса.
11. Пользователь нажимает "**Сохранить**".
12. Все вопросы теста и варианты ответов теста сохраняются в БД.

**Альтернативный сценарий 1:**

     6a. Пользователь нажимает "**крестик**" напротив строки с ответом.

    7а. Вариант ответа, напротив которого был нажат "**крестик**" удаляется.

**Альтернативный сценарий 2:**

    11б. Пользователь нажимает кнопку "**Выйти**".

    12б. Открывается окно в котором пользователю предлагается сохранить тест.

    13б. Пользователь нажимает "**Сохранить**".

    14б. Все вопросы теста и варианты ответа к ним сохраняются в БД.

Отчет по выполнению "итоговой лабораторной работы":

Для выполнения данной лабораторной работы, мне было необходимо создать два приложения: серверное на базе Express и клиентское на базе React, которые обмениваются данными через REST API.

# Выполнение работы

## 1. Подготовка окружения

Перед началом работы я установил необходимые инструменты, включая Node.js, npm и PostgreSQL. Также я создал проектную директорию "C:\Users\apce1\Desktop\УЧЕБА\Учебная практика\проект\лабы\итогоаяЛаба" и инициализировал новый проект с помощью команды "npm init" для создания файла package.json.

## 2. Настройка базы данных

Для подключения к базе данных PostgreSQL, я использовал библиотеку knex. В файле "C:\Users\apce1\Desktop\УЧЕБА\Учебная практика\проект\лабы\итогоаяЛаба\server.js" я создал объект подключения к базе данных и настроил его параметры, такие как хост, пользователь, пароль и имя базы данных.

## 3. Создание таблиц в базе данных

Для хранения данных о тестах, вопросах и ответах я создал три таблицы в базе данных: "tests", "questions" и "answers". Каждая таблица была создана с использованием языка SQL с помощью миграций. В файлах миграций, например, "20230726000001\_create\_tests\_table.js", "20230726000002\_create\_questions\_table.js" и "20230726000003\_create\_answers\_table.js", я определил структуру таблиц, указал поля и связи между таблицами.

## 4. Создание клиентской части

Для создания клиентской части приложения, я использовал библиотеку React. В файле "C:\Users\apce1\Desktop\УЧЕБА\Учебная практика\проект\лабы\итогоаяЛаба\client\src\App.js" я создал компонент App, который представляет собой главное окно программы. В этом окне пользователь может добавлять тесты и вопросы.

Для добавления теста пользователь вводит его название, затем система открывает окно добавления вопросов. В окне добавления вопросов пользователь вводит текст вопроса и варианты ответов. При необходимости можно добавить еще вариантов ответов или удалить существующие.

## 5. Обработка сохранения теста в базе данных

При нажатии кнопки "Сохранить" вопросы теста и варианты ответов сохраняются в базе данных с помощью AJAX-запроса. В файле "C:\Users\apce1\Desktop\УЧЕБА\Учебная практика\проект\лабы\итогоаяЛаба\client\src\App.js" я реализовал функцию handleSaveTest, которая выполняет POST-запрос на сервер с данными о тесте и вопросах.

Таблицы

CREATE TABLE tests (

    id SERIAL PRIMARY KEY,

    title VARCHAR(255) NOT NULL

  );

  CREATE TABLE questions (

    id SERIAL PRIMARY KEY,

    test\_id INTEGER NOT NULL,

    question\_text TEXT NOT NULL,

    FOREIGN KEY (test\_id) REFERENCES tests (id) ON DELETE CASCADE

  );

  CREATE TABLE answers (

    id SERIAL PRIMARY KEY,

    question\_id INTEGER NOT NULL,

    answer\_text TEXT NOT NULL,

    is\_correct BOOLEAN NOT NULL DEFAULT FALSE,

    FOREIGN KEY (question\_id) REFERENCES questions (id) ON DELETE CASCADE

  );

Server.js

const express = require('express');

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

const knex = require('knex');

// Подключение к базе данных (используется PostgreSQL)

const db = knex({

  client: 'pg',

  connection: {

    host: '127.0.0.1',

    user: 'postgres',

    password: '1',

    database: 'arsenshmid'

  }

});

const app = express();

app.use(bodyParser.json());

// Создаем тест

app.post('/api/tests', (req, res) => {

  const { title, questions } = req.body;

  if (!title || !questions || !Array.isArray(questions) || questions.length === 0) {

    return res.status(400).json({ error: 'Некорректные данные для создания теста.' });

  }

  // Сохранение теста и вопросов в базу данных

  db.transaction(trx => {

    return db('tests')

      .insert({ title })

      .returning('id')

      .then(testId => {

        const questionsData = questions.map(question => ({

          test\_id: testId[0],

          question\_text: question.text

        }));

        return trx('questions')

          .insert(questionsData)

          .returning('id')

          .then(questionIds => {

            const answersData = questions.reduce((acc, question, index) => {

              question.answers.forEach((answer, ansIndex) => {

                acc.push({

                  question\_id: questionIds[index],

                  answer\_text: answer,

                  is\_correct: ansIndex === question.correctIndex

                });

              });

              return acc;

            }, []);

            return trx('answers').insert(answersData);

          });

      })

      .then(trx.commit)

      .catch(trx.rollback);

  })

  .then(() => {

    return res.status(201).json({ message: 'Тест успешно создан.' });

  })

  .catch(error => {

    return res.status(500).json({ error: 'Ошибка при создании теста.' });

  });

});

// Здесь можно добавить другие роуты и логику работы с БД

const PORT = 5000; // Вы можете использовать любой свободный порт на вашем компьютере

app.listen(PORT, () => {

  console.log(`Сервер запущен на порту ${PORT}.`);

});

App.js

import React, { useState } from 'react';

import axios from 'axios';

function App() {

  const [title, setTitle] = useState('');

  const [questions, setQuestions] = useState([{ text: '', answers: ['', ''] }]);

  const handleAddQuestion = () => {

    setQuestions([...questions, { text: '', answers: ['', ''] }]);

  };

  const handleRemoveQuestion = (index) => {

    const updatedQuestions = [...questions];

    updatedQuestions.splice(index, 1);

    setQuestions(updatedQuestions);

  };

  const handleAddAnswer = (questionIndex) => {

    const updatedQuestions = [...questions];

    updatedQuestions[questionIndex].answers.push('');

    setQuestions(updatedQuestions);

  };

  const handleRemoveAnswer = (questionIndex, answerIndex) => {

    const updatedQuestions = [...questions];

    updatedQuestions[questionIndex].answers.splice(answerIndex, 1);

    setQuestions(updatedQuestions);

  };

  const handleSaveTest = () => {

    if (!title || !questions.some(q => q.text)) {

      alert('Заполните все поля перед сохранением теста.');

      return;

    }

    // Выполните AJAX запрос для отправки данных на сервер

    const data = { title, questions };

    axios.post('/api/tests', data)

      .then(response => {

        alert(response.data.message);

        setTitle('');

        setQuestions([{ text: '', answers: ['', ''] }]);

      })

      .catch(error => {

        alert(error.response.data.error);

      });

  };

  return (

    <div>

      <h1>Система тестирования</h1>

      <div>

        <label>Название теста:</label>

        <input

          type="text"

          value={title}

          onChange={(e) => setTitle(e.target.value)}

        />

      </div>

      <div>

        {questions.map((question, index) => (

          <div key={index}>

            <label>Вопрос {index + 1}:</label>

            <input

              type="text"

              value={question.text}

              onChange={(e) => {

                const updatedQuestions = [...questions];

                updatedQuestions[index].text = e.target.value;

                setQuestions(updatedQuestions);

              }}

            />

            {question.answers.map((answer, ansIndex) => (

              <div key={ansIndex}>

                <input

                  type="text"

                  value={answer}

                  onChange={(e) => {

                    const updatedQuestions = [...questions];

                    updatedQuestions[index].answers[ansIndex] = e.target.value;

                    setQuestions(updatedQuestions);

                  }}

                />

                {ansIndex === question.answers.length - 1 && (

                  <button onClick={() => handleAddAnswer(index)}>Добавить ответ</button>

                )}

                {ansIndex > 1 && (

                  <button onClick={() => handleRemoveAnswer(index, ansIndex)}>Удалить ответ</button>

                )}

              </div>

            ))}

            <button onClick={handleAddQuestion}>Добавить вопрос</button>

            {index > 0 && (

              <button onClick={() => handleRemoveQuestion(index)}>Удалить вопрос</button>

            )}

          </div>

        ))}

      </div>

      <button onClick={handleSaveTest}>Сохранить</button>

    </div>

  );

}

export default App;

C:\Users\apce1\Desktop\УЧЕБА\Учебная практика\проект\лабы\итогоаяЛаба\knexfile.js

module.exports = {

  development: {

    client: 'pg',

    connection: {

      host: '127.0.0.1',

      user: 'postgres',

      password: '1',

      database: 'arsenshmid',

    },

    migrations: {

      directory: './migrations',

    },

    seeds: {

      directory: './seeds',

    },

  },

};

C:\Users\apce1\Desktop\УЧЕБА\Учебная практика\проект\лабы\итогоаяЛаба\server\migrations\20230726000001\_create\_tests\_table.js

exports.up = function (knex) {

    return knex.schema.createTable('tests', (table) => {

      table.increments('id').primary();

      table.string('title', 255).notNullable();

    });

  };

  exports.down = function (knex) {

    return knex.schema.dropTableIfExists('tests');

  };

C:\Users\apce1\Desktop\УЧЕБА\Учебная практика\проект\лабы\итогоаяЛаба\server\migrations\20230726000002\_create\_questions\_table.js

exports.up = function (knex) {

    return knex.schema.createTable('questions', (table) => {

      table.increments('id').primary();

      table.integer('test\_id').notNullable().references('id').inTable('tests').onDelete('CASCADE');

      table.text('question\_text').notNullable();

    });

  };

  exports.down = function (knex) {

    return knex.schema.dropTableIfExists('questions');

  };

C:\Users\apce1\Desktop\УЧЕБА\Учебная практика\проект\лабы\итогоаяЛаба\server\migrations\20230726000003\_create\_answers\_table.js

exports.up = function (knex) {

    return knex.schema.createTable('answers', (table) => {

      table.increments('id').primary();

      table.integer('question\_id').notNullable().references('id').inTable('questions').onDelete('CASCADE');

      table.text('answer\_text').notNullable();

      table.boolean('is\_correct').notNullable().defaultTo(false);

    });

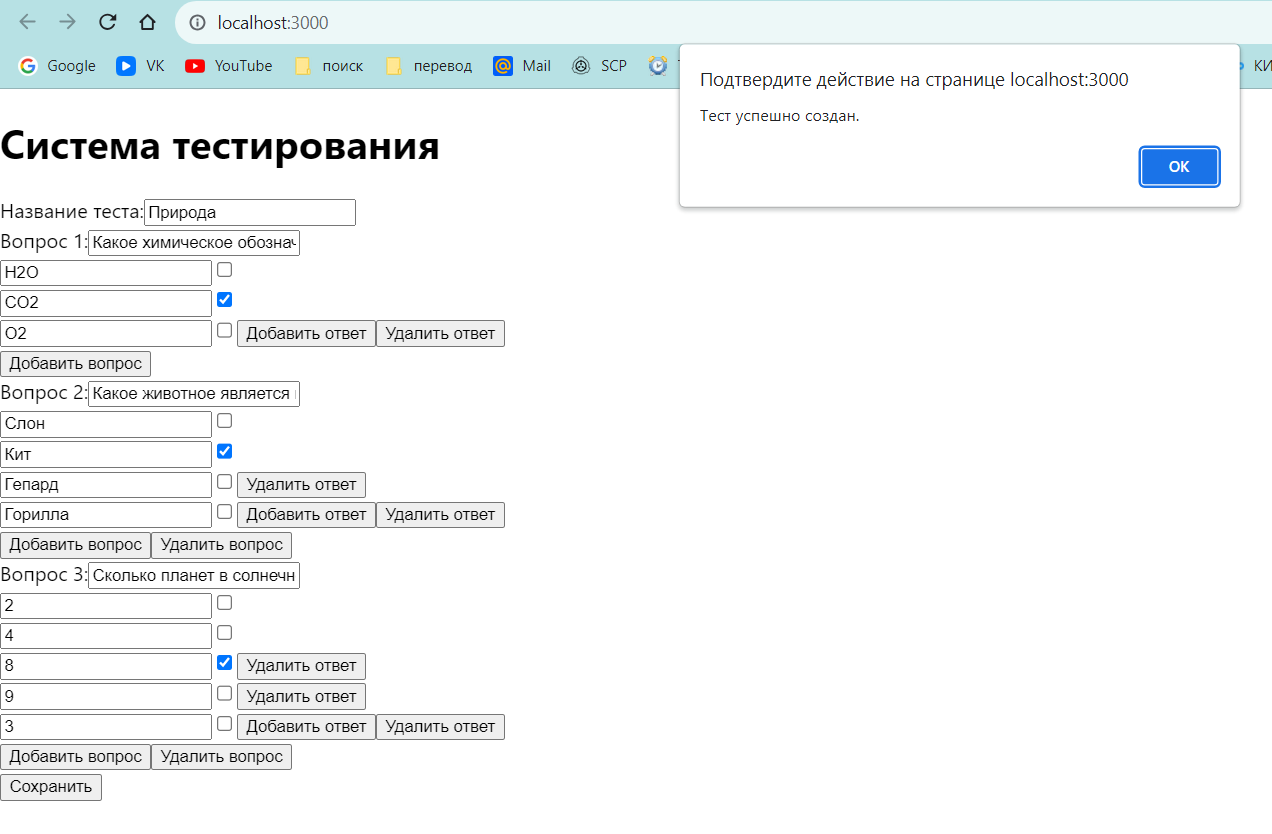
  };

  exports.down = function (knex) {

    return knex.schema.dropTableIfExists('answers');

  };

Основной сценарий:



База данных:

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание  
Изображение выглядит как текст, снимок экрана, дисплей, число

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание

**Альтернативный сценарий 1:**

     6a. Пользователь нажимает "**крестик**" напротив строки с ответом.

    7а. Вариант ответа, напротив которого был нажат "**крестик**" удаляется.

import React, { useState } from 'react';

import axios from 'axios';

function App() {

  const [title, setTitle] = useState('');

  const [questions, setQuestions] = useState([{ text: '', answers: [{ text: '', isCorrect: false }] }]);

  const handleAddQuestion = () => {

    setQuestions([...questions, { text: '', answers: [{ text: '', isCorrect: false }] }]);

  };

  const handleRemoveQuestion = (index) => {

    const updatedQuestions = [...questions];

    updatedQuestions.splice(index, 1);

    setQuestions(updatedQuestions);

  };

  const handleAddAnswer = (questionIndex) => {

    const updatedQuestions = [...questions];

    updatedQuestions[questionIndex].answers.push({ text: '', isCorrect: false });

    setQuestions(updatedQuestions);

  };

  const handleRemoveAnswer = (questionIndex, answerIndex) => {

    const updatedQuestions = [...questions];

    updatedQuestions[questionIndex].answers.splice(answerIndex, 1);

    setQuestions(updatedQuestions);

  };

  const handleCheckboxChange = (questionIndex, answerIndex) => {

    const updatedQuestions = [...questions];

    updatedQuestions[questionIndex].answers = updatedQuestions[questionIndex].answers.map((answer, index) => ({

      ...answer,

      isCorrect: index === answerIndex,

    }));

    setQuestions(updatedQuestions);

  };

  const handleSaveTest = () => {

    if (!title || !questions.some(q => q.text)) {

      alert('Заполните все поля перед сохранением теста.');

      return;

    }

    // Выполните AJAX запрос для отправки данных на сервер

    const data = { title, questions };

    axios.post('/api/tests', data)

      .then(response => {

        alert(response.data.message);

        setTitle('');

        setQuestions([{ text: '', answers: [{ text: '', isCorrect: false }] }]);

      })

      .catch(error => {

        alert(error.response.data.error);

      });

  };

  return (

    <div>

      <h1>Система тестирования</h1>

      <div>

        <label>Название теста:</label>

        <input

          type="text"

          value={title}

          onChange={(e) => setTitle(e.target.value)}

        />

      </div>

      <div>

        {questions.map((question, index) => (

          <div key={index}>

            <label>Вопрос {index + 1}:</label>

            <input

              type="text"

              value={question.text}

              onChange={(e) => {

                const updatedQuestions = [...questions];

                updatedQuestions[index].text = e.target.value;

                setQuestions(updatedQuestions);

              }}

            />

            {question.answers.map((answer, ansIndex) => (

              <div key={ansIndex}>

                <input

                  type="text"

                  value={answer.text}

                  onChange={(e) => {

                    const updatedQuestions = [...questions];

                    updatedQuestions[index].answers[ansIndex].text = e.target.value;

                    setQuestions(updatedQuestions);

                  }}

                />

                <input

                  type="checkbox"

                  checked={answer.isCorrect}

                  onChange={() => handleCheckboxChange(index, ansIndex)}

                />

                <span onClick={() => handleRemoveAnswer(index, ansIndex)}>❌</span>

                {ansIndex === question.answers.length - 1 && (

                  <button onClick={() => handleAddAnswer(index)}>Добавить ответ</button>

                )}

              </div>

            ))}

            <button onClick={handleAddQuestion}>Добавить вопрос</button>

            {index > 0 && (

              <button onClick={() => handleRemoveQuestion(index)}>Удалить вопрос</button>

            )}

          </div>

        ))}

      </div>

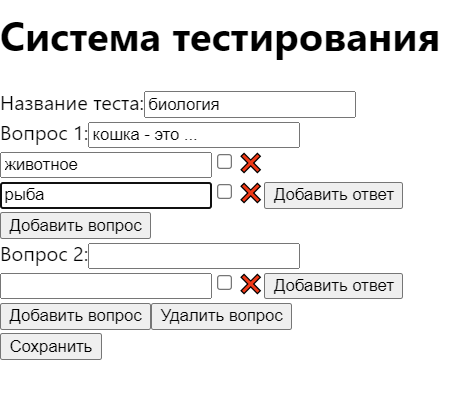
      <button onClick={handleSaveTest}>Сохранить</button>

    </div>

  );

}

export default App;



Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

**Альтернативный сценарий 2:**

    11б. Пользователь нажимает кнопку "**Выйти**".

    12б. Открывается окно в котором пользователю предлагается сохранить тест.

    13б. Пользователь нажимает "**Сохранить**".

    14б. Все вопросы теста и варианты ответа к ним сохраняются в БД.

App.js

import React, { useState } from 'react';

import axios from 'axios';

function App() {

  const [title, setTitle] = useState('');

  const [questions, setQuestions] = useState([{ text: '', answers: [{ text: '', isCorrect: false }] }]);

  const [isExitModalOpen, setIsExitModalOpen] = useState(false); // New state variable

  const handleAddQuestion = () => {

    setQuestions([...questions, { text: '', answers: [{ text: '', isCorrect: false }] }]);

  };

  const handleRemoveQuestion = (index) => {

    const updatedQuestions = [...questions];

    updatedQuestions.splice(index, 1);

    setQuestions(updatedQuestions);

  };

  const handleAddAnswer = (questionIndex) => {

    const updatedQuestions = [...questions];

    updatedQuestions[questionIndex].answers.push({ text: '', isCorrect: false });

    setQuestions(updatedQuestions);

  };

  const handleRemoveAnswer = (questionIndex, answerIndex) => {

    const updatedQuestions = [...questions];

    updatedQuestions[questionIndex].answers.splice(answerIndex, 1);

    setQuestions(updatedQuestions);

  };

  const handleCheckboxChange = (questionIndex, answerIndex) => {

    const updatedQuestions = [...questions];

    updatedQuestions[questionIndex].answers = updatedQuestions[questionIndex].answers.map((answer, index) => ({

      ...answer,

      isCorrect: index === answerIndex,

    }));

    setQuestions(updatedQuestions);

  };

  const handleSaveTest = () => {

    if (!title || !questions.some(q => q.text)) {

      alert('Заполните все поля перед сохранением теста.');

      return;

    }

    // Выполните AJAX запрос для отправки данных на сервер

    const data = { title, questions };

    axios.post('/api/tests', data)

      .then(response => {

        alert(response.data.message);

        setTitle('');

        setQuestions([{ text: '', answers: [{ text: '', isCorrect: false }] }]);

      })

      .catch(error => {

        alert(error.response.data.error);

      });

  };

  const handleSaveTestAndExit = () => {

    handleSaveTest();

    handleCloseExitModal();

  };

  const handleOpenExitModal = () => {

    setIsExitModalOpen(true);

  };

  const handleCloseExitModal = () => {

    setIsExitModalOpen(false);

  };

  return (

    <div>

      <h1>Система тестирования</h1>

      <div>

        <label>Название теста:</label>

        <input

          type="text"

          value={title}

          onChange={(e) => setTitle(e.target.value)}

        />

      </div>

      <div>

        {questions.map((question, index) => (

          <div key={index}>

            <label>Вопрос {index + 1}:</label>

            <input

              type="text"

              value={question.text}

              onChange={(e) => {

                const updatedQuestions = [...questions];

                updatedQuestions[index].text = e.target.value;

                setQuestions(updatedQuestions);

              }}

            />

            {question.answers.map((answer, ansIndex) => (

              <div key={ansIndex}>

                <input

                  type="text"

                  value={answer.text}

                  onChange={(e) => {

                    const updatedQuestions = [...questions];

                    updatedQuestions[index].answers[ansIndex].text = e.target.value;

                    setQuestions(updatedQuestions);

                  }}

                />

                <input

                  type="checkbox"

                  checked={answer.isCorrect}

                  onChange={() => handleCheckboxChange(index, ansIndex)}

                />

                {ansIndex === question.answers.length - 1 && (

                  <button onClick={() => handleAddAnswer(index)}>Добавить ответ</button>

                )}

                {ansIndex > 1 && (

                  <button onClick={() => handleRemoveAnswer(index, ansIndex)}>Удалить ответ</button>

                )}

              </div>

            ))}

            <button onClick={handleAddQuestion}>Добавить вопрос</button>

            {index > 0 && (

              <button onClick={() => handleRemoveQuestion(index)}>Удалить вопрос</button>

            )}

          </div>

        ))}

      </div>

      <button onClick={handleOpenExitModal}>Выйти</button>

      {/\* Exit Modal \*/}

      {isExitModalOpen && (

        <div>

          <p>Вы хотите сохранить тест перед выходом?</p>

          <button onClick={handleSaveTestAndExit}>Сохранить</button>

          <button onClick={handleCloseExitModal}>Не сохранять</button>

        </div>

      )}

      {/\* End of Exit Modal \*/}

      <button onClick={handleSaveTest}>Сохранить</button>

    </div>

  );

}

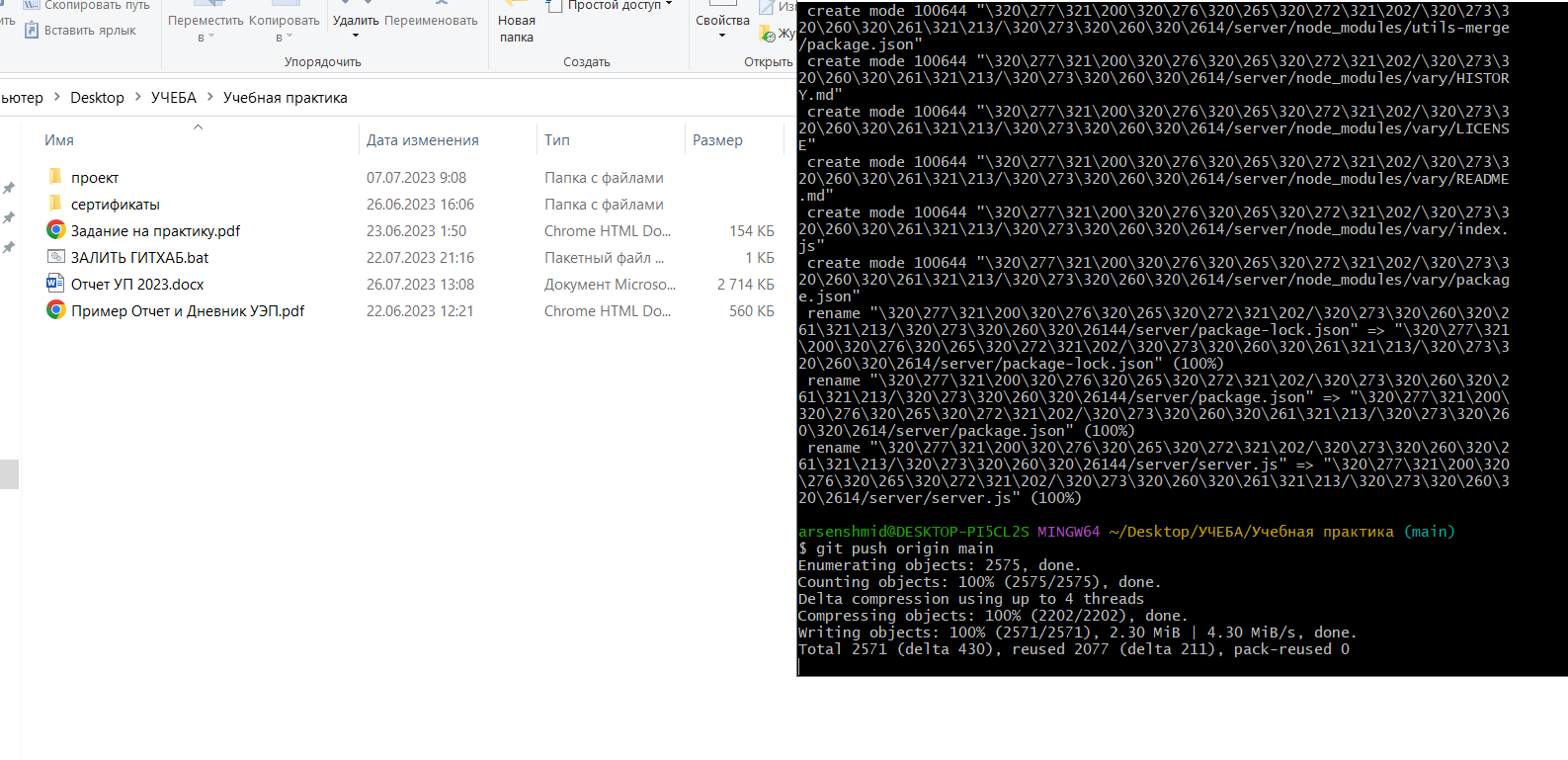
export default App;

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

После каждой лабы, загружал версию проекта на гитхаб.   
  


**Заключение**

В результате учебной практики я успешно прошел три курса, расширил свои знания в области SQL, работы с табличным процессором Excel и разработки проектов. Благодаря этому опыту я могу применять полученные знания на практике и эффективно решать задачи, связанные с работой с данными, анализом данных и разработкой проектов.

Я предоставил подробный отчет о своей учебной практике, который включает краткое содержание выполненных работ, дневник практики, заполненный на каждый день практики, описание разработки проекта по курсу "Проектный практикум", скриншоты и полученные сертификаты. Все задания были выполнены в соответствии с календарным планом