**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

A logo of a triangle

Description automatically generated

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ЛИПЕЦКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

|  |  |
| --- | --- |
| Институт | компьютерных наук |
| Кафедра | автоматизированных систем управления |

ИНДИВИДУАЛЬНОЕ ДОМАШНЕЕ ЗАДАНИЕ

По дисциплине: "Базы данных"

На тему: "Разработка прототипа прикладного приложения для БД"

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | ПИ-22-1 | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  подпись, дата | Насонов Н.С. |
| Руководитель | |  |  |  |
| к.т.н., доцент  ученая степень, ученое звание | |  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  подпись, дата | Алексеев В.А. |

Липецк, 2024 г.

**Цель работы**

Получение первичных навыков разработки прикладных приложений для БД, освоение фреймворков для работы с БД.

**Задание**

Реализовать прикладное приложение, обеспечивающее просмотр и редактирование содержимого спроектированной в ходе лабораторного практикума БД. Прототип должен реализовывать функции для поддержки одного бизнес-процесса информационной системы и содержать следующие элементы интерфейса:

1) Меню для навигации по функциям (левое или верхнее). Для мобильных приложений могут использоваться карточки для навигации по функциям.

2) Табличное представление данных с фильтром и пагинацией (для веб-приложений) https://www.uprock.ru/articles/dizayn-tablic-ot-putanicy-k-intuitivno-ponyatnomu-opytu В таблице не должны отображаться искусственные идентификаторы, они должны быть заменены содержательными значениями полей.

3) Редактирование записей в таблице «на месте» - при небольшом числе атрибутов, если горизонтальная прокрутка отсутствует.

4) Редактирование записей в режиме «модального окна» или «боковой панели» (паттерн «контекстный оверлей»). Не должны отображаться искусственные идентификаторы, они должны быть заменены содержательными значениями полей.

5) Выпадающие списки или чек-листы для выбора связанных значений в другой таблице.

6) Календари, переключатели, чекбоксы – там, где это применимо.

7) Отчет с выбором периода и других объектов (например, отчет по определенной категории товаров за год и т.п.).

Прикладное приложение должно содержать минимум 3 табличных представления и минимум 2 формы редактирования в режиме «модального окна» или «боковой панели». Дополнительным плюсом является наличие в отчете графика/диаграммы или возможности экспорта в формат PDF.

**Описание предметной области.**

Предметная область связана с автоматизацией процесса начисления и распределения стипендий среди студентов образовательного учреждения. Основные проблемы, которые решает система: сложность обработки большого объема данных вручную, высокая вероятность ошибок при расчете стипендий, задержки с выплатами. Также существует необходимость учета различных видов стипендий, которые назначаются по различным критериям (академические успехи, социальное положение, спортивные достижения и т.д.).

**Основные бизнес-процессы.**

1. Внесение данных о студентах — деканат вводит или обновляет данные о студентах, включая их личную информацию и данные об успеваемости. Студент может отправить запрос на изменение данных при неточностях
2. Предоставление отчетов — система формирует отчеты для бухгалтерии (сводные данные о бюджете и распределении средств), деканата (успеваемость студентов) и студентов (информация о назначенных стипендиях и их изменениях).
3. Внесение изменений в документы – студенты и деканат могут корректировать документы (например, исправления в успеваемости), а система обновляет расчеты.

**Категории пользователей.**

1. Бухгалтерия

Бухгалтерия – просмотр общего бюджета, а также автоматическое распределение стипендий между студентами-бюджетниками

1. Студенты

Студенты – просмотр назначенных стипендий, внесение информации о достижениях, влияющих на нее, а также подача документов на повышенную стипендию в установленной форме. Возможность просмотра информации об успеваемости и достижениях, для запросов об исправлении.

1. Деканат

Деканат – внесение успеваемости студентов и их личной информации, и их личной информации, исправление ошибок в документах.

**Требования к отчетам.**

* Отчет для деканата: Отчеты по успеваемости студентов, которые помогают отслеживать их учебные достижения и контролировать соответствие начисления стипендий.

**Физическая модель**

Диаграмма физической модели с отображением name, разработанной в среде Sybase PowerDesigner представлена на рисунке 1.



Рисунок – Диаграмма физической модели с отображением name

Диаграмма физической модели с отображением code, разработанной в среде Sybase PowerDesigner представлена на рисунке 2.



Рисунок – Диаграмма физической модели с отображением code

**Разработка прикладного приложения**

**Платформа разработки**

1. Аппаратные средства
   1. Операционная система для разработки: Windows 11 с подсистемой WSL2.
   2. Оборудование: ПК/ноутбук с процессором Ryzen 7, 32 ГБ оперативной памяти.

**Программные средства разработки**

1. **Операционная система для разработки**.

Проект разрабатывался на ОС Windows 11 и тестировался в Linux-средах с использованием Docker.

1. **Контейнеризация и оркестрация**

Использовался Docker для контейнеризации всех сервисов (бэкенд, фронтенд, база данных). Docker Compose применялся для управления многосервисной архитектурой.

1. **Система управления базами данных (СУБД).**
   1. PostgreSQL версии 16, развёрнутая в контейнере Docker.
   2. Для инициализации базы данных использовался скрипт clone\_db.sql, содержащий структуру и тестовые данные.
   3. Для ранней разработки использовался
2. **Языки программирования**
   1. Go (Golang): Использовался для разработки бэкенда. Основной запускаемый модуль программы размещён в папке backend/main, а внутренняя логика организована в виде пакетов внутри backend/internal/.
   2. Python**:** Использовался для модуля backend содержащий генерацию PDF.
   3. HTML, CSS, JavaScript**:** Применялись для разработки фронтенда.
3. **Фреймворки и библиотеки**
   1. Для бэкенда на Go
      1. Gin: Фреймворк для создания HTTP API.
      2. PostgreSQL driver (pgx): Для работы с базой данных.
   2. Для фронтенда
      1. NodeJS для запуска фронтенд части.
   3. Для бэкенда на Python
      1. Веб фреймворк Flask для связи с фронтендом и бэкендом на go
      2. FPDF для генерации PDF файлов
4. **Инструменты разработки и интеграции**
   1. IDE: Visual Studio Code с установленными плагинами для Go, Python и Docker.
   2. Система контроля версий: Git, с настройкой .gitignore для исключения временных файлов и артефактов Docker.
   3. Файлы конфигурации:

docker-compose.yml для управления сервисами и контейнерами.

go.mod и go.sum для управления зависимостями в Go.

requirements.txt для зависимостей Python.

**Схема взаимодействия с БД**

1. **Связь приложения с БД**

Для соединения с базой данных используется библиотека pgx (PostgreSQL driver). Подключение настраивается через строку подключения, параметры которой берутся из переменных окружения (docker-compose.yml). Используется пул соединений для повышения производительности.

//Парсим для подключения к БД

    dbHost := getEnv("DB\_HOST", "localhost")

    dbPort := getEnv("DB\_PORT", "5432")

    dbUser := getEnv("DB\_USER", "postgres")

    dbPassword := getEnv("DB\_PASSWORD", "0000")

    dbName := getEnv("DB\_NAME", "University\_DB")

    //Подключаемся к БД

    connStr := fmt.Sprintf("postgres://%s:%s@%s:%s/%s?sslmode=disable",

        dbUser, dbPassword, dbHost, dbPort, dbName)

//Создаем логгер для вывода ошибок

logger := log.New()

    logger.SetLevel(log.InfoLevel)

    logger.SetFormatter(&log.TextFormatter{})

//Передаем параметры для подключения

    config, err := pgxpool.ParseConfig(connStr)

    if err != nil {

        logger.WithError(err).Fatalf("can't parse pgxpool config")

    }

    config.ConnConfig.Tracer = &tracelog.TraceLog{

        Logger:   pgxLogrus.NewLogger(logger),

        LogLevel: tracelog.LogLevelDebug,

    }

//Создаем пул запросов

    pool, err := pgxpool.NewWithConfig(context.Background(), config)

    if err != nil {

        logger.WithError(err).Fatalf("can't create new pool")

    }

    defer pool.Close()

    repo := repository.NewRepository(pool, logger)

1. **Получение данных из БД**

Для получения данных используется модуль repository. SQL-запросы вынесены в отдельный файл (queries) для удобства поддержки. Функция включает в себя фильтрацию, поиск, номер страницы и количество строк (для получения лишь нужного количества для отображения используется offset и limit)

func (q \*Queries) GetAllStudent(ctx context.Context, filters map[string]string, rowCount, page int, search string) ([]\*domain.Student, int, error) {

    getAllStudent := `SELECT \* FROM student WHERE 1=1`

    countQuery := `SELECT COUNT(\*) FROM student WHERE 1=1`

//Добавляем поиск по подстроке в запрос к БД

if search != "" {

sqlSearch := ` AND (name\_group ILIKE '%` + search + `%' OR email\_student ILIKE '%` + search + `%' OR second\_name\_student ILIKE '%` + search + `%' OR first\_name\_student ILIKE '%` + search + `%' OR surname\_student ILIKE '%` + search + `%')`

        getAllStudent += sqlSearch

        countQuery += sqlSearch

}

    var args []interface{}

//Получаем готовые запросы после применения фильтров

    getAllStudent, countQuery, args = UnpackFilter(ctx, getAllStudent, countQuery, filters, rowCount, page)

    rows, err := q.pool.Query(ctx, getAllStudent, args...)

    if err != nil {

        return nil, 0, fmt.Errorf("can't query students: %w", err)

    }

    defer rows.Close()//Отложенное закрытие соединения

//Распаковываем полученные данные в структуру описывающую эти данные

    students := []\*domain.Student{}

    for rows.Next() {

        student := &domain.Student{}

        err := rows.Scan(

            &student.Id\_num\_student,

            &student.Name\_group,

            &student.Email\_student,

            &student.Second\_name\_student,

            &student.First\_name\_student,

            &student.Surname\_student)

        if err != nil {

            return nil, 0, fmt.Errorf("can't scan students: %w", err)

        }

        students = append(students, student)

    }

    if err = rows.Err(); err != nil {

        return nil, 0, err

    }

    // Получаем общее количество строк без пагинации

    var count int

    err = q.pool.QueryRow(ctx, countQuery, args...).Scan(&count)

    if err != nil {

        return nil, 0, fmt.Errorf("can't count students: %w", err)

    }

    return students, count, nil

}

**Функция фильтрации**

func UnpackFilter(ctx context.Context, getAll, countQuery string, filters map[string]string, rowCount, page int) (string, string, []any) {

    var args []interface{}

    for key, value := range filters {

        getAll += fmt.Sprintf(" AND %s = $%d", key, len(args)+1)

        countQuery += fmt.Sprintf(" AND %s = $%d", key, len(args)+1)

        args = append(args, value)

    }

    offset := (page - 1) \* rowCount

    getAll += fmt.Sprintf(" LIMIT %d OFFSET %d", rowCount, offset)

    return getAll, countQuery, args

}

1. **Модификация данных в БД**

Для изменения/добавления/удаления данных используется аналогичная схема с подготовленным SQL запросом.

**Добавление записи**

func (q \*Queries) CreateStudent(ctx context.Context, id\_num\_student uint64, name\_group, email\_student, second\_name\_student,

    first\_name\_student, surname\_student string) (\*domain.Student, error) {

    sqlStatement := `INSERT INTO student (id\_num\_student,name\_group, email\_student, second\_name\_student, first\_name\_student, surname\_student) VALUES ($1, $2, $3, $4, $5, $6) RETURNING id\_num\_student`

    err := q.pool.QueryRow(ctx, sqlStatement, id\_num\_student, name\_group, email\_student, second\_name\_student, first\_name\_student, surname\_student).Scan(&id\_num\_student)

    if err != nil {

        return nil, fmt.Errorf("can't create student: %w", err)

    }

    student, err := q.FindStudentByID(ctx, id\_num\_student)

    if err != nil {

        return nil, fmt.Errorf("can't find student: %w", err)

    }

    return student, err

}

**Изменение записи**

func (q \*Queries) UpdateStudentbyID(ctx context.Context, id\_num\_student uint64, name\_group, email\_student, second\_name\_student,

    first\_name\_student, surname\_student string) (\*domain.Student, error) {

    sqlStatement := `UPDATE student SET name\_group=$1, email\_student=$2, second\_name\_student=$3, first\_name\_student=$4, surname\_student=$5 WHERE id\_num\_student=$6 RETURNING id\_num\_student`

    err := q.pool.QueryRow(ctx, sqlStatement, name\_group, email\_student, second\_name\_student, first\_name\_student, surname\_student, id\_num\_student).Scan(&id\_num\_student)

    if err != nil {

        return nil, fmt.Errorf("can't update student: %w", err)

    }

    student, err := q.FindStudentByID(ctx, id\_num\_student)

    if err != nil {

        return nil, fmt.Errorf("can't find student: %w", err)

    }

    return student, err

}

**Удаление записи**

func (q \*Queries) DeleteStudentbyID(ctx context.Context, id\_num\_student uint64) error {

    sqlStatement := `DELETE FROM student WHERE id\_num\_student=$1`

    \_, err := q.pool.Exec(ctx, sqlStatement, id\_num\_student)

    if err != nil {

        return fmt.Errorf("can't delete student: %w", err)

    }

    return nil

}

1. **Вызов хранимой процедуры**

В проекте возможна поддержка хранимых процедур. К примеру, для автоматического расчета стипендии по среднему баллу студента.

func (r \*Repository) CalculateScholarship(name\_semester, scholarship\_type string) error {

const callProcedure =`CALL assign\_scholarships($1,$2)`

\_, err := r.db.Exec(callProcedure, name\_semester, scholarship\_type)

return err

}

**Разработка пользовательского интерфейса**

1. Был создан основной HTML файл, который хранит основные моменты для динамической загрузки таблиц, фильтров, названия таблиц. А также основных кнопок
2. Минималистичная стилизация интерфейса в CSS для удобства использования пользовательского интерфейса и правильного позиционирования элементов.

На рисунках 3, 4, 5 представлен разработанный пользовательский интерфейс.

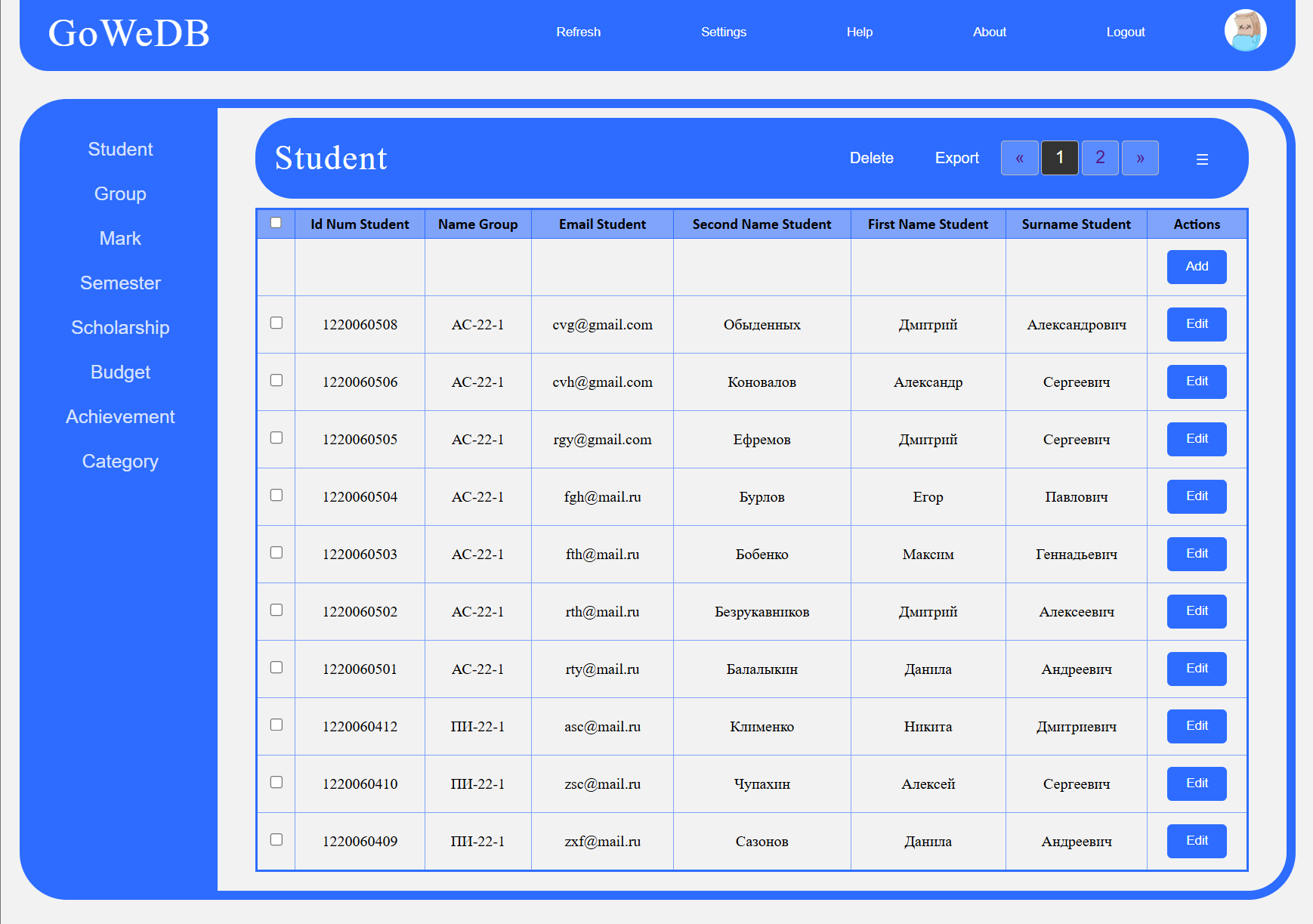


Рисунок 3 – Общий пользовательский интерфейс

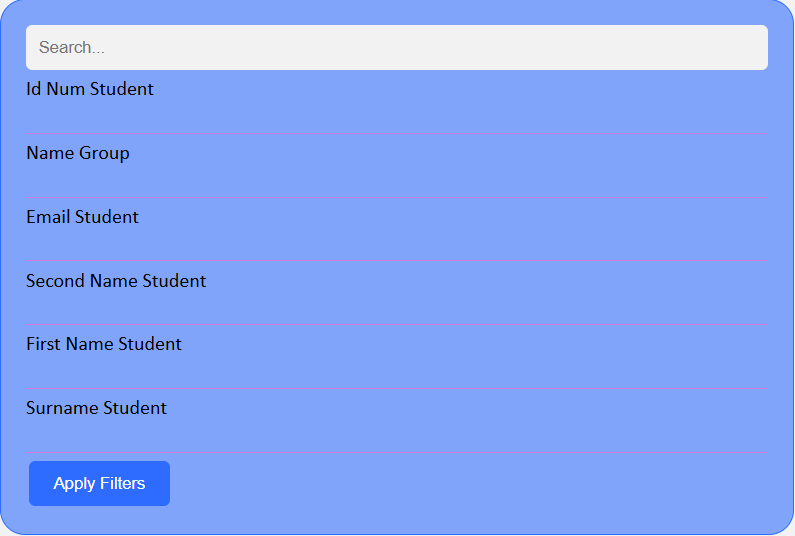


Рисунок 4 – Интерфейс фильтрации

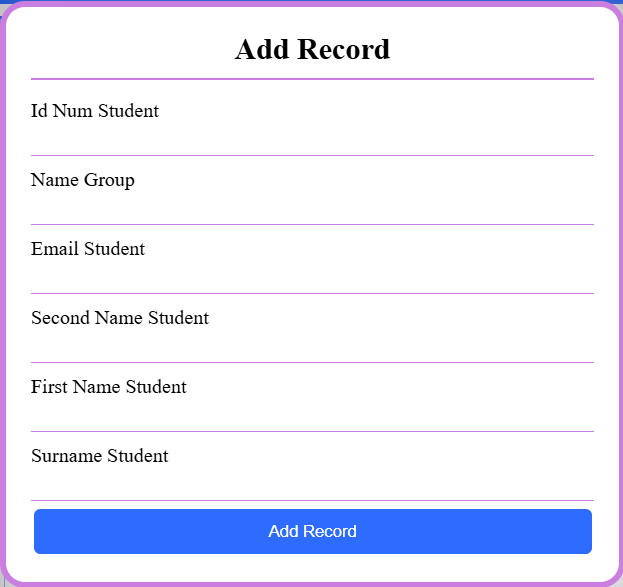


Рисунок 5 – Интерфейс модального окна

1. Написание JS для динамического создания фильтров по meta информации, полученной с сервера, а также переключателей таблиц. Функций для получения таблиц по фильтрам, поиску и номеру страниц.

**Функция загрузки таблиц**

async function loadTableData(tableName,page=1,filters = {}) {

try {

const filterParams = new URLSearchParams(filters).toString(); // Преобразуем фильтры в строку параметров

try{

search=document.getElementById("search-input").value; //Получаем данные из строки

} catch{

search='';

}

const url = `${options}/api/v1/${tableName}/?page=${page}${filterParams ? '&' + filterParams : ''}&search=${search}`;

currentURL = url;

currentTable = tableName;

console.log(url)

const [dataResponse, metadataResponse] = await Promise.all([

fetch(url),

fetch(`${options}/api/v1/${tableName}/metadata`)

]);

const data = await dataResponse.json();

const metadata = await metadataResponse.json();

const tableData = document.getElementById("table-data");

const tableNameElement = document.getElementById("table-name");

tableNameElement.textContent = capitalizeWords(tableName);

tableData.innerHTML = "";

// Создаем фильтры на основе метаданных

if (Object.keys(filters).length == 0) {

createFilters(metadata.data); // Добавляем фильтры

}

if (data.data.length > 0) {

createTableHeader(tableData, metadata.data);

createTableBody(tableData, data.data, metadata.data, tableName,page,filters);

paginationLoad(tableName, page, data.pages,filters);

}

} catch (error) {

console.error("Error loading table data:", error);

}

}

**Функция для создания и применения фильтров**

Фильтры содержат стандартную проверку на типы.

// Функция для динамического создания фильтров

async function createFilters(metadata) {

const filterContainer = document.getElementById("filters-form");

filterContainer.innerHTML = `<input type="text" id="search-input" placeholder="Search...">`; // Очищаем старые фильтры

const filters = {}; // Это объект для хранения фильтров

metadata.forEach(column => {

if (column.filterable !== false) {

const filterDiv = document.createElement("div");

const label = document.createElement("label");

label.textContent = capitalizeWords(column.name.replace(/\_/g, " "));

const input = document.createElement("input");

input.type = column.type || "text"; // Используем текстовый input по умолчанию

input.id = `filter-${column.name}`;

// if (column.required) {

// input.required = true; // Добавляем валидацию для обязательных полей

// }

// Сохраняем фильтры в объект

input.addEventListener("input", () => {

filters[column.name] = input.value;

});

if (column.id){

filterDiv.style.display ="none";

}

filterDiv.appendChild(label);

filterDiv.appendChild(input);

filterContainer.appendChild(filterDiv);

}

});

// Кнопка применения фильтров

const applyButton = document.createElement("button");

applyButton.textContent = "Apply Filters";

applyButton.addEventListener("click", async (event) => applyFilters(filters,event));

filterContainer.appendChild(applyButton);

}

function applyFilters(filters,event) {

event.preventDefault();

const tableName = document.getElementById("table-name").textContent.toLowerCase();

loadTableData(tableName, 1, filters); // Перезагружаем таблицу с фильтрами

}

**Текст программы**

Ранее была рассмотрена работа с БД на уровне драйвера с использованием библиотеки pgx. Сейчас рассмотрим передачу данных, возвращаемых функциями, работающими с БД. Рассмотрим обработку запроса на получение данных с фильтрацией, поиском и номером страницы. При успешном запросе возвращаем код 200 и json с данными и количеством страниц. Иначе возвращаем код 400/403 и ошибку.

func getAllStudent(c \*gin.Context, a \*app.App) {

//Получаем номер страницы и переводим в числовое значение из url

    param := c.Query("page")

    page, err := strconv.Atoi(param)

    if err != nil {

        c.JSON(http.StatusBadRequest, ErrorResponse(err))

        return

    }

//Получаем параметр поиска для поиска по под строке из url

    search := c.Query("search")

    // Собираем фильтры (все параметры, кроме page и search)

    filters := make(map[string]string)

    for key, value := range c.Request.URL.Query() {

        if key != "page" && key != "search" {

            if len(value[0]) > 0 {

                filters[key] = value[0] // берем значение фильтра

            }

        }

    }

// Отправляем данные на универсальный обработчик (в данном случае он простой и содержит один репозиторий для работы), который в последствии вызывает функцию для получения данных с БД

    students, count, err := a.GetAllStudent(c, filters, rowCount, page, search)

    if err != nil {

        c.JSON(http.StatusForbidden, ErrorResponse(err))

        return

    }

    c.JSON(http.StatusOK, AllStudentSuccessResponse(students, rowCount, count, page))

}

**Тестирование прикладного приложения**

На стороне backend были написаны интеграционные и юнит тесты бизнес-логики. Проведем системное тестирование прикладного приложения.

Протестируем поиск и фильтрацию при помощи UI интерфейса. На рисунке 6 представлен результат поиска.

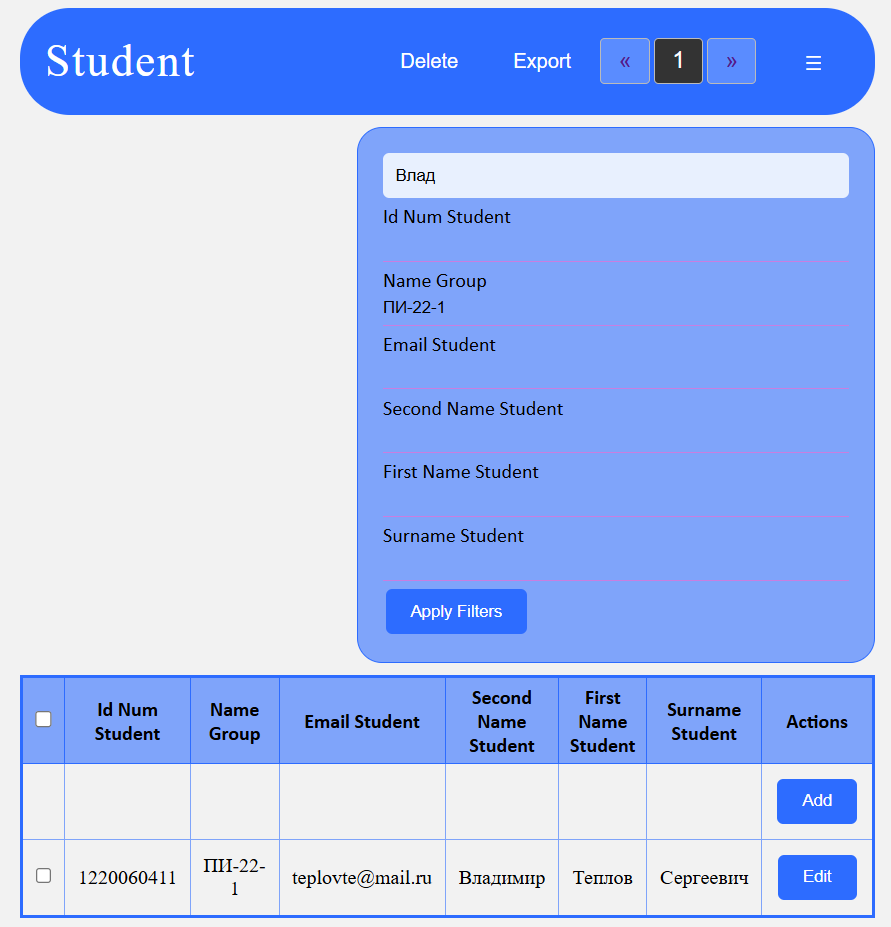


Рисунок 6 – Тестирование фильтрации с поиском

Отредактируем данные найденные при помощи фильтров. На рисунке 7 представлено редактирование в строке. На рисунке 8 результат выполнения редактирования.

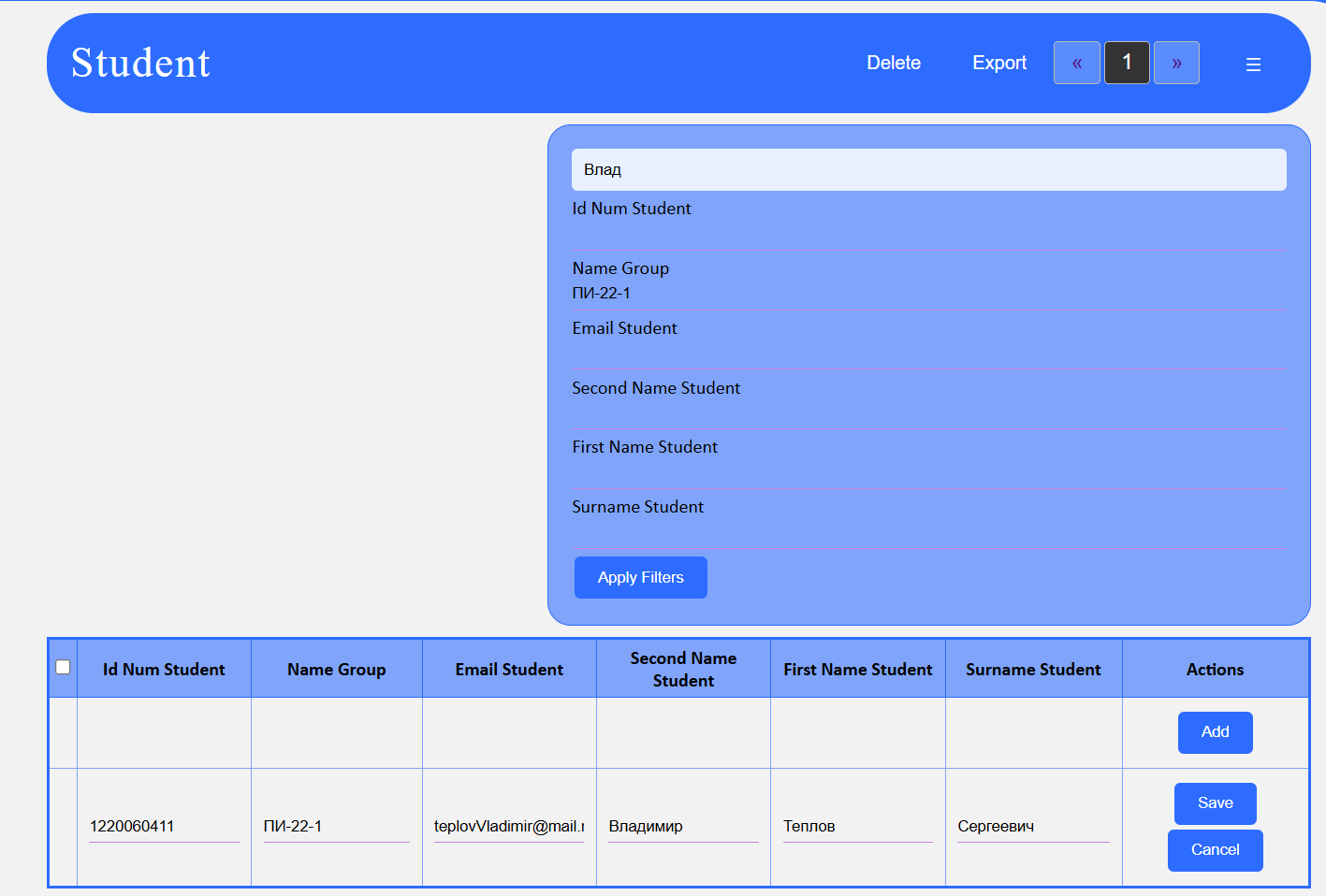


Рисунок 7 – Редактирование данных в строке

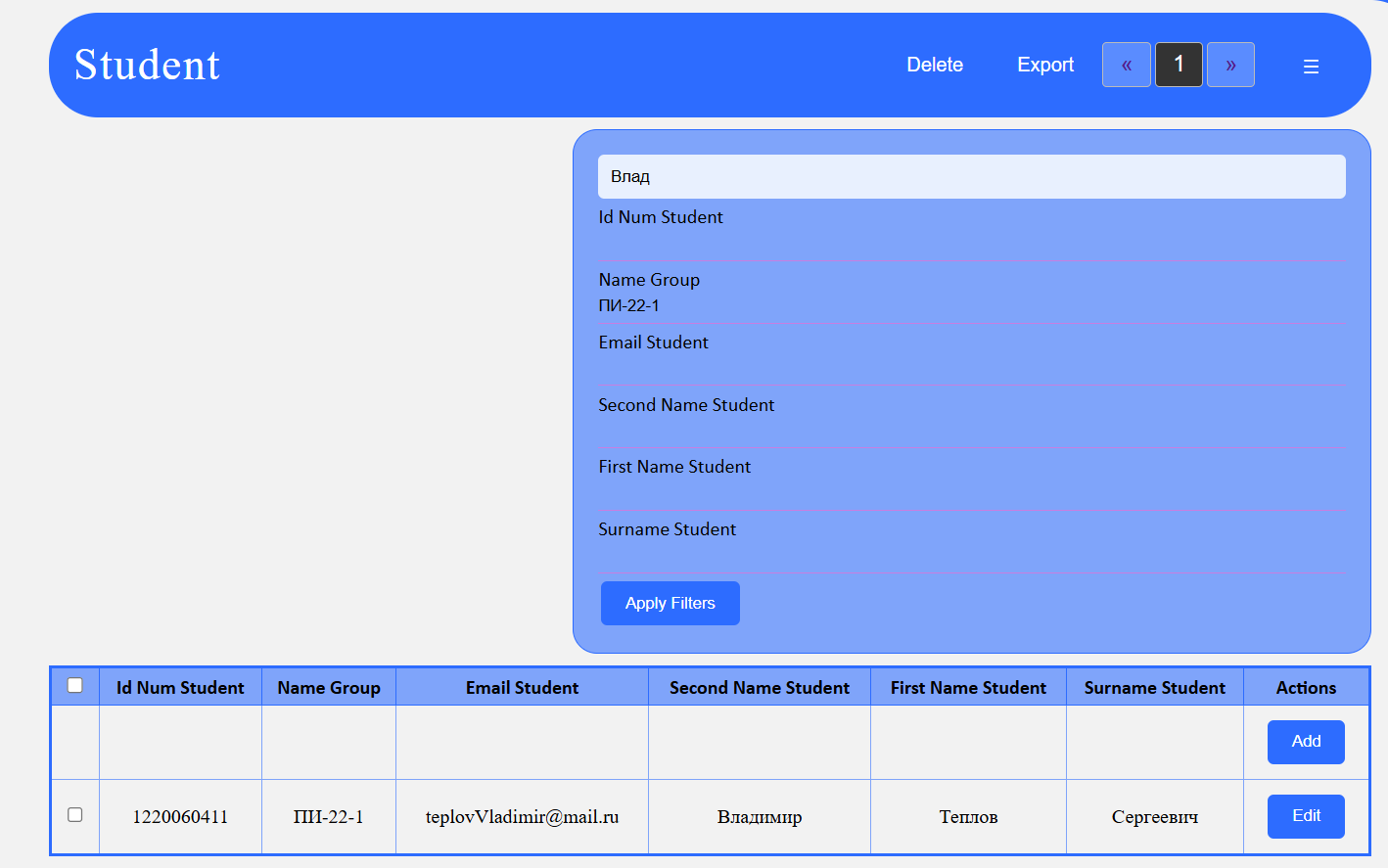


Рисунок 8 – Результат редактирования

Протестируем внесение новых данных в таблицу на этом же примере. Для этого нажмем Add и откроется модальное окно. На рисунке 9 представлено добавление, на рисунке 10 добавленная запись.

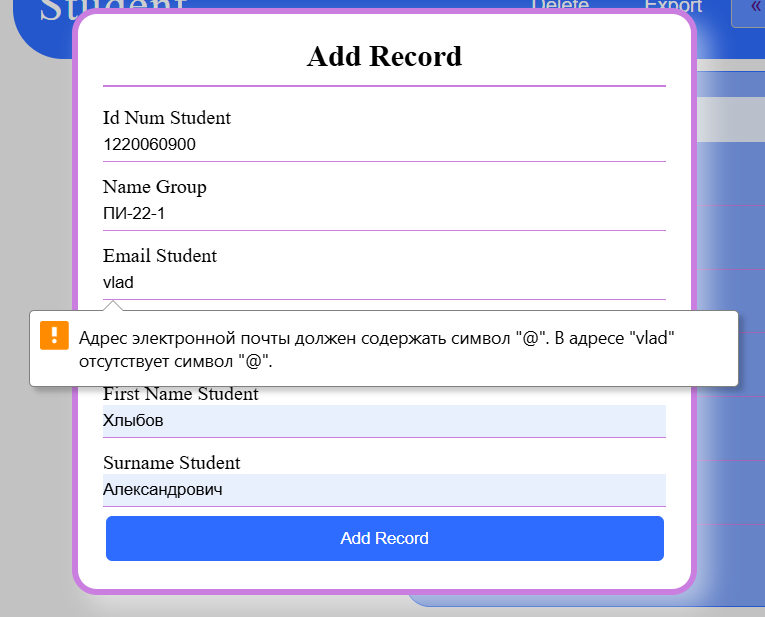


Рисунок 9 – модальное окно для добавления записи

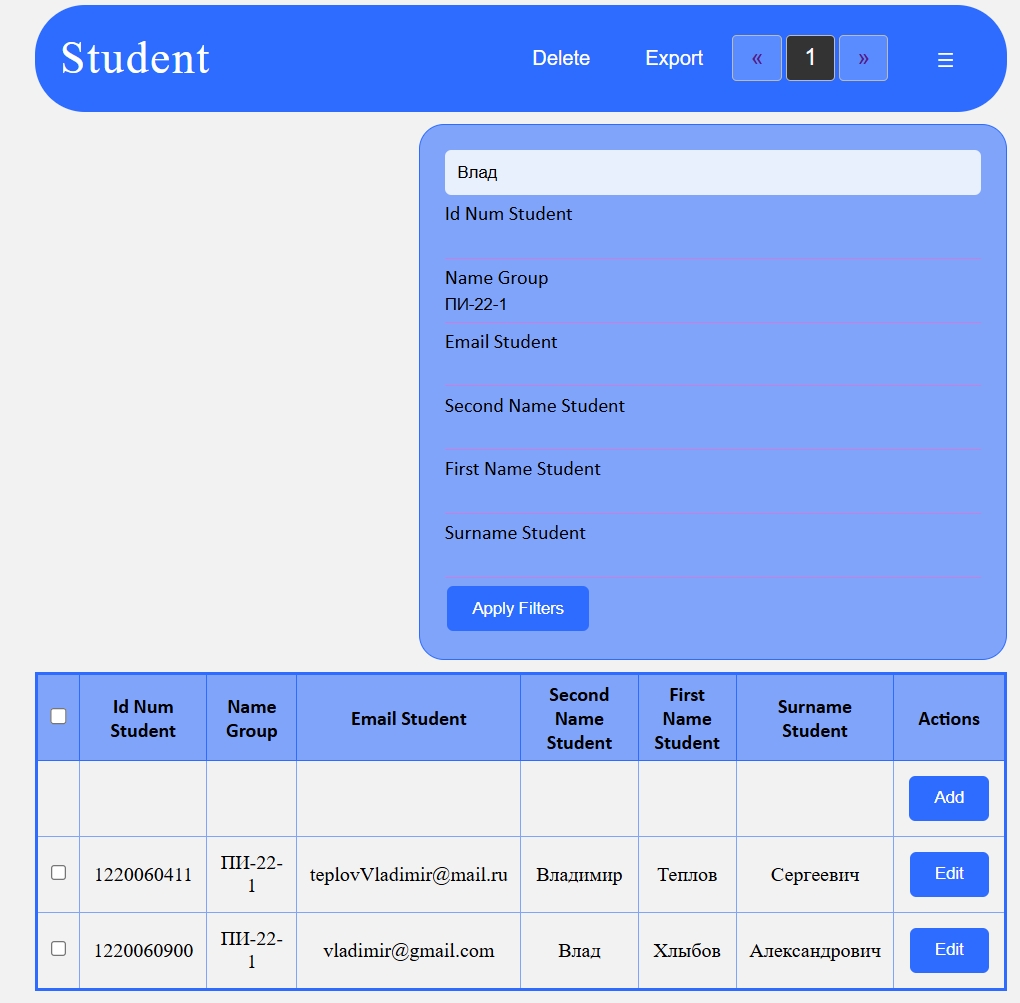


Рисунок 10 – Результат добавления записи

Протестируем создание отчетов при помощи фильтрации. К примеру, выведем всех студентов группы ПИ-22-1. На рисунке 11 представлен пример отчета.

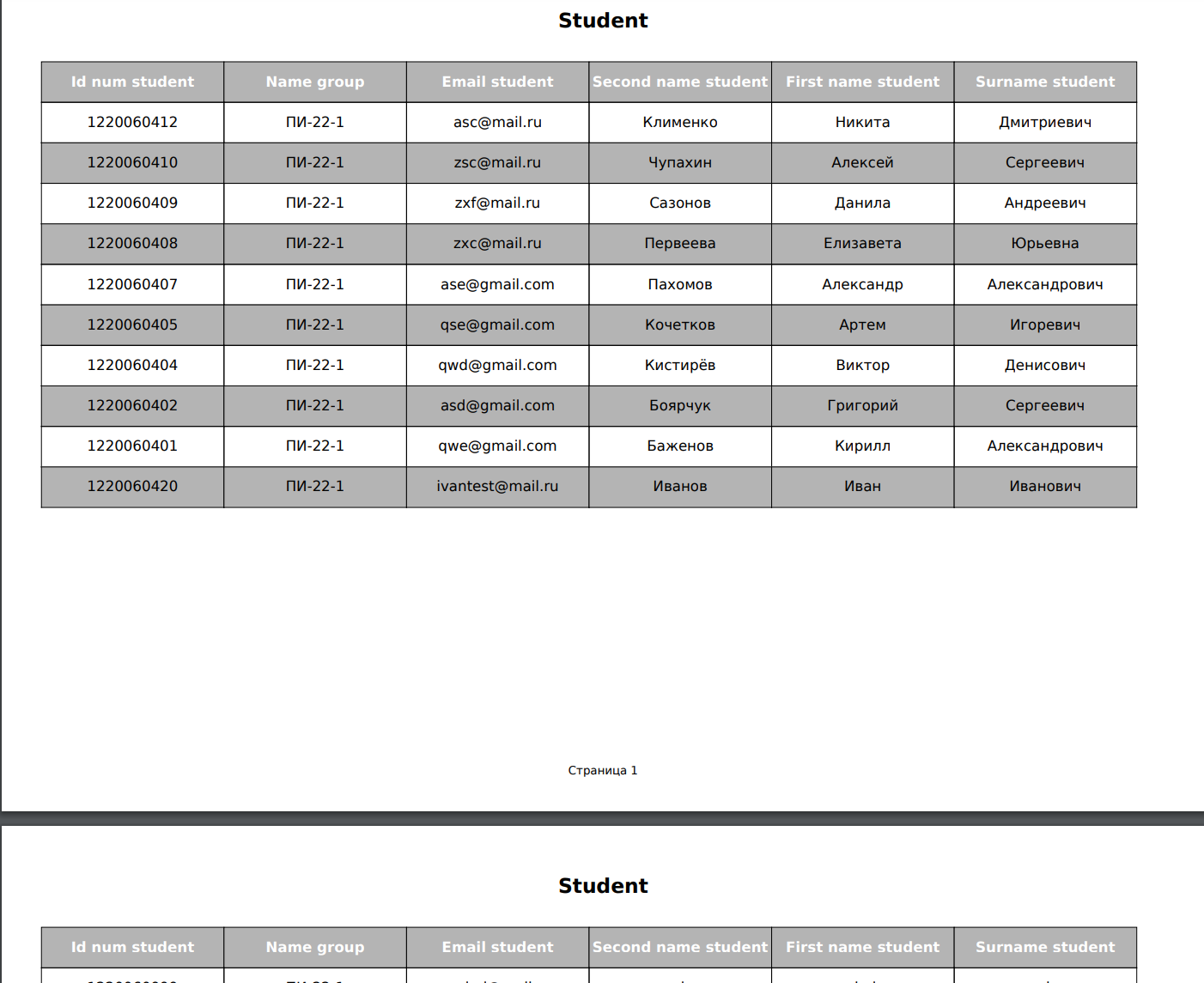


Рисунок 11 – Пример отчета

**Вывод**

В ходе данной работы был разработан прототип Web-приложения для работы с базой данных. Приложение использует БД PostgreSQL, backend для работы с БД реализован на GO с фреймворком GIN и библиотекой pgx для работы с БД на уровне драйвера. Также были написаны интеграционные тесты для проверки работы API. Отчеты были сгенерированы при помощи Python с использованием Flask и FPDF. Использовался Docker для контейнеризации всех сервисов (бэкенд, фронтенд, база данных). Фронтенд разработан с использованием чистого JS.