Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОННИКИ
Факультет компьютерных систем и сетей
Кафедра электронных вычислительных машин
Дисциплина: Базы данных

Тема «Автосервис»
Лабораторная работа №6
Создание приложения для базы данных

Студент: А.Ю. Вадецкий
Преподаватель: Д.В. Куприянова

МИНСК 2025

ВВЕДЕНИЕ

Современные информационные системы немыслимы без использования баз данных (БД), которые обеспечивают структурированное хранение, эффективное управление и быстрый доступ к большим объемам данных. Приложения для работы с базами данных играют ключевую роль в автоматизации бизнес-процессов, аналитике и поддержке принятия решений. Одной из наиболее популярных систем управления базами данных (СУБД) является PostgreSQL, благодаря своей надежности, открытому исходному коду и поддержке расширенных функций, таких как транзакции, триггеры и полнотекстовый поиск.

Целью данной работы является разработка прикладного приложения, обеспечивающего взаимодействие пользователя с базой данных через интуитивно понятный интерфейс, без необходимости написания SQL-запросов вручную. Такой подход минимизирует ошибки, связанные с человеческим фактором, и делает работу с данными доступной даже для пользователей без глубоких знаний SQL.

В рамках задачи предполагается реализовать базовые операции управления данными: создание и удаление таблиц, редактирование их структуры, резервное копирование и восстановление информации. Особое внимание уделяется механизмам экспорта данных в форматы, удобные для анализа (например, Excel), а также сохранению и повторному использованию пользовательских запросов. Это позволяет повысить гибкость работы с информацией и адаптировать приложение под конкретные сценарии использования.

Разрабатываемое приложение демонстрирует интеграцию PostgreSQL с выбранным языком программирования, подчеркивая возможности взаимодействия между клиентской частью и СУБД. Важным аспектом является обеспечение отказоустойчивости через резервное копирование и реализацию транзакций для сохранения целостности данных.

1 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

1.1 Системные требования

Для запуска программы необходимо:

- React (сборка через Create React App, Vite или аналоги);
- Java 11+ (OpenJDK или Oracle JDK);
- Spring Boot 2.7+ (или другой Java-фреймворк);
- Maven 3.6+ или Gradle 7.x (для сборки бэкенда);
- PostgreSQL 13+ / MySQL 8+ (или другая СУБД, если требуется);
- веб-сервер (Nginx, Apache) или встроенный сервер (Tomcat, Netty);
- 2+ ядра CPU (4+ рекомендуется для продакшена);
- 4 ГБ ОЗУ (8+ ГБ рекомендуется);
- 10 ГБ свободного места (SSD предпочтительно);
- современный браузер (Chrome, Firefox, Edge) для клиентской части;
- Git 2.30+ (для контроля версий);
- Docker (опционально, для контейнеризации);
- порты: 80/443 (HTTP/HTTPS) для фронтенда, 8080/8443 для API;
- доступ в интернет (для загрузки зависимостей);
- поддерживаемые ОС: Windows 10+, macOS 12+, Linux (Ubuntu 20.04+, CentOS 7+).

1.2 Раздел Queries

Главное окно приложения представляет собой интерфейс для работы с базой данных. Верхние кнопки:

1. Queries – переключает на раздел запросов (готовые и ручные SQL).
2. Tables – открывает управление таблицами (создание, изменение, удаление).
3. Backup – переход к резервному копированию (экспорт/импорт данных).

Нижний экран разделен на две основные области:

1. Слева - интерфейс с выбором кнопок управления.
2. Справа – консоль с выводом результата.

1.2.1 Вкладка Predefenited Queries

Изначально выбран раздел «Predefenited Queries» с готовыми запросами в базу данных.

Начальное окно приложения представлено на рисунке 1.1.

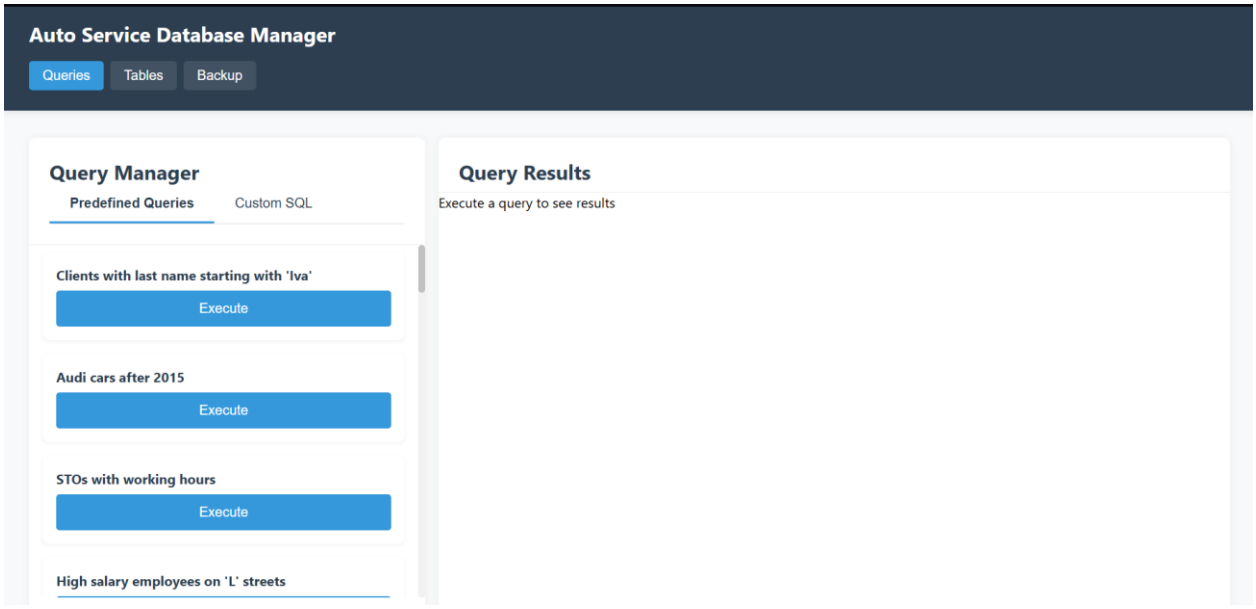

Рисунок 1.1 – Начальное окно приложения

1.2.2 Вкладка Custom SQL

При выборе вкладки Custom SQL. Появляется меню:

1. Текстовое поле для SQL-запросов:
   - позволяет вручную вводить SQL-запросы (SELECT, INSERT, UPDATE, DELETE и др.);
   - поддерживает многострочный ввод;
   - можно вставлять заранее подготовленные запросы.
2. Кнопки управления:
   - execute – выполняет введённый SQL-запрос и выводит результат в правой панели;
   - если запрос некорректный – появится ошибка в уведомлении;
   - если запрос возвращает данные – они отобразятся в виде таблицы или текста;
   - save – сохраняет текущий запрос в базу данных (появится запрос на ввод названия). После сохранения запрос добавляется в список Saved Queries.
3. Блок Saved Queries (сохранённые запросы):
   - список ранее сохранённых SQL-запросов (название + сам запрос). Для каждого запроса есть две кнопки:
   - execute – сразу выполняет запрос и показывает результат справа;
   - delete – удаляет запрос из базы (после подтверждения).

1.3 Раздел Tables

Раздел Tables предоставляет полный контроль над структурой и содержимым таблиц БД.

1.3.1 Вкладка Browse

Элементы управления:

1. Выпадающий список таблиц:
   - за что отвечает: выбор таблицы для просмотра/редактирования;
   - особенности: автоматически загружает список всех таблиц при открытии раздела.
2. Таблица данных (правая панель):
   Отображает: все записи выбранной таблицы

Вкладка Browse представлена на рисунке 1.3.
Редактирование данные представлено на рисунке 1.4.

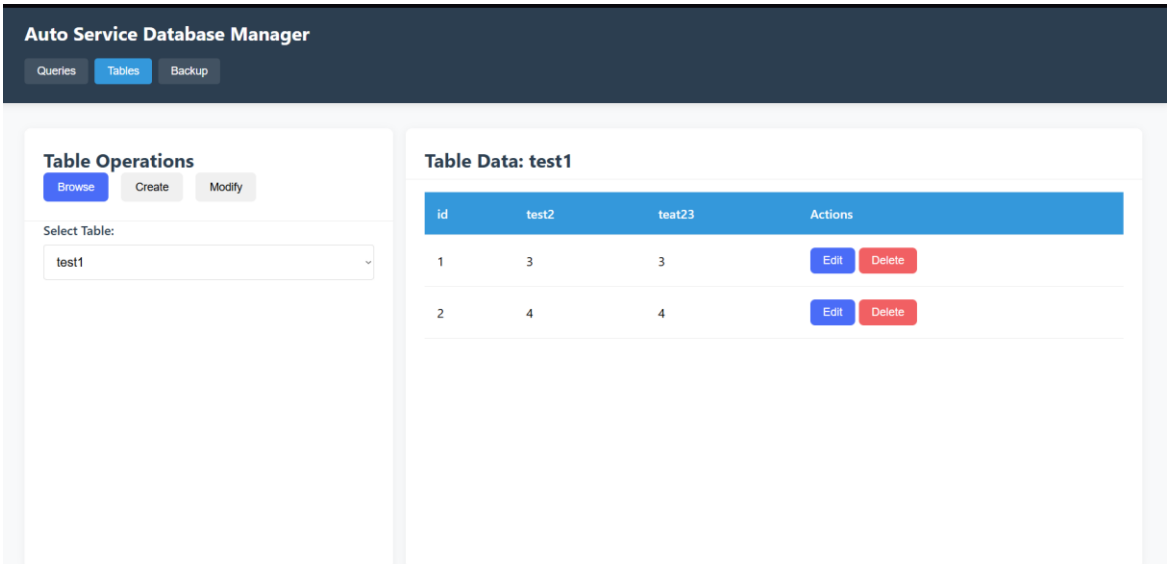

Рисунок 1.3 – Вкладка Browse

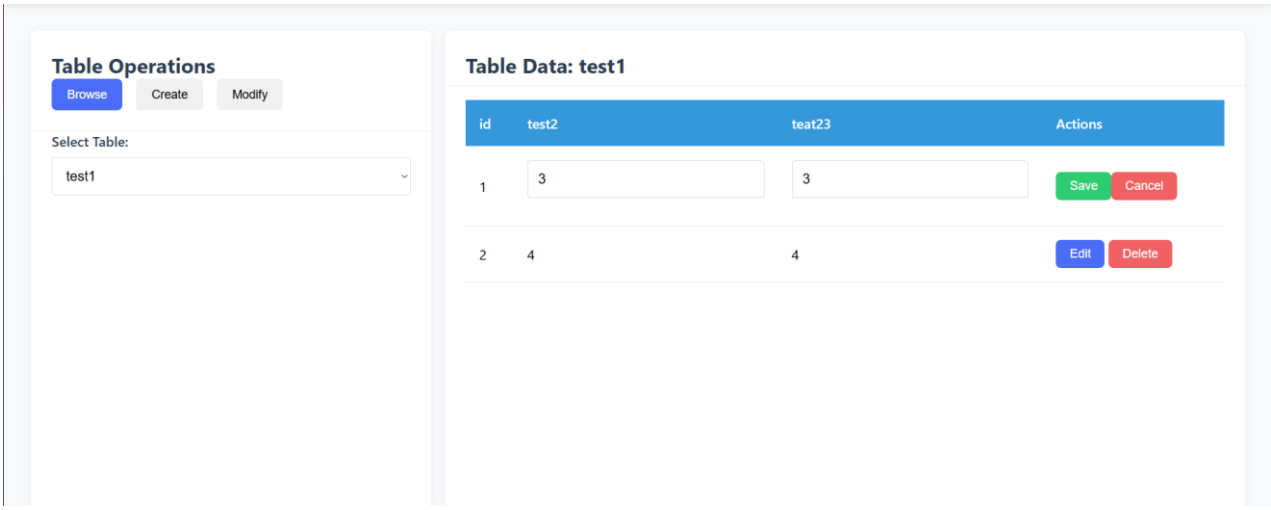

Рисунок 1.4 – Редактирование данных

1.3.2 Вкладка Create

Служит для создания новой таблицы.

Элементы управления:

1. Поле Table Name:
   - назначение: ввод имени новой таблицы;
   - ограничения: только латинские буквы, цифры и _.
2. Раздел Columns:
   - назначение: определение структуры новой таблицы;
   - текстовое поле – указываем название столбцов;
   - выпадающий список – выбираем формат столбца.
3. Кнопка Create Table:
   - действие: создает новую таблицу с указанными параметрами;
   - результат: таблица появляется в выпадающем списке.

Пример создания таблицы представлен на рисунке 1.5

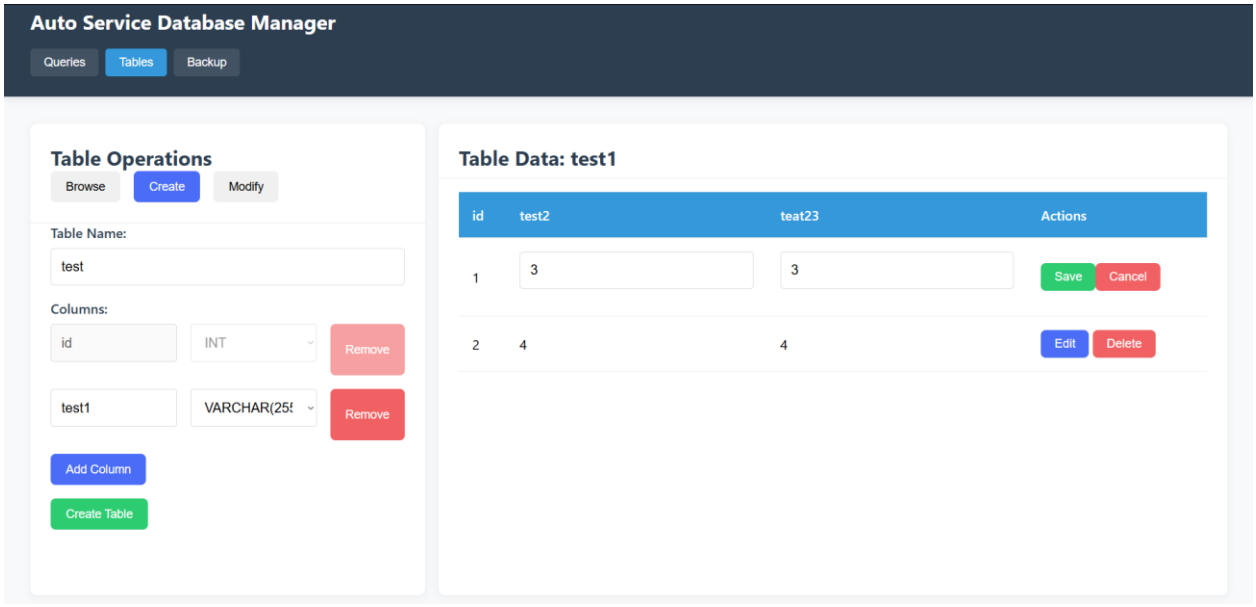

Рисунок 1.5 – Пример создания таблицы

1.3.3 Вкладка Modify

Для работы с данным разделом необходимо выбрать таблицу во вкладке Browse.

1.3.3.1 Изменение структуры

Добавление столбца:

1. Поле "Column Name" - имя нового столбца.
2. Выпадающий список "Column Type" - выбор типа данных.
   - VARCHAR(255);
   - INT;
   - FLOAT;
   - DATE;

- BOOLEAN;
- INTERVAL.

3. Кнопка "Add Column" - подтверждает добавление.

1.3.3.2 Удаление столбца

1. Поле "Column Name" - имя удаляемого столбца.
2. Кнопка "Drop Column" - удаляет столбец (требует подтверждения).
3. Работа с данными.

1.3.4.1 Обновление данных

1. Поле "Column to Update" - имя изменяемого столбца.
2. Поле "New Value" - новое значение.
3. Поле "WHERE Condition" - условие выбора строк (напр. id = 5).
4. Кнопка "Update Data" - применяет изменения.

1.3.4.1 Добавление строки

1. Текстовое поле Insert Row (JSON).
2. Формат: {"column1":"value1", "column2":value2}.
3. Кнопка "Insert Row" - добавляет новую запись.

1.3.4.2 Удаление таблицы

1. Кнопка Drop Table:
   - действие: полное удаление текущей таблицы;
   - особенности: запрашивает подтверждение.

На рисунке 1.6 представлен раздел Modify.

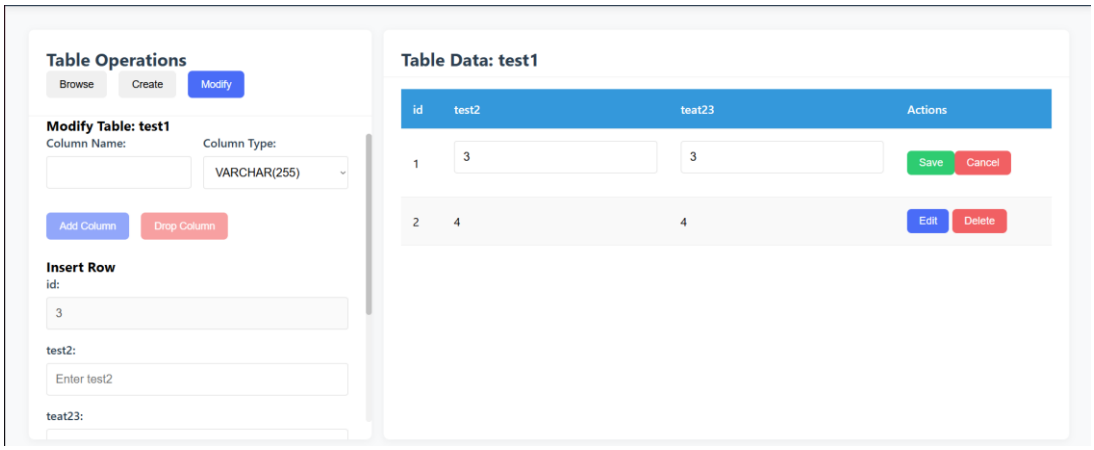

Рисунок 1.6 – Раздел Modify

1.4 Раздел Bakcup

Раздел Backup предоставляет инструменты для экспорта данных в различные форматы.

1.4.1 Вкладка Export

Элементы управления:

1. Выпадающий список Table Name:
   - назначение: выбор таблицы для экспорта;
   - особенности: автоматически загружает список таблиц.
2. Поле Filename:
   - назначение: указание имени файла (без расширения);
   - автодополнение: показывает полное имя файла (пример: data.xlsx).
3. Кнопка Export to Excel:
   - действие: создает XLSX-файл с данными выбранной таблицы;
   - результат: файл скачивается автоматически;
   - уведомление: «Exported to [filename].xlsx successfully».

На рисунке 1.7 представлена вкладка Export.

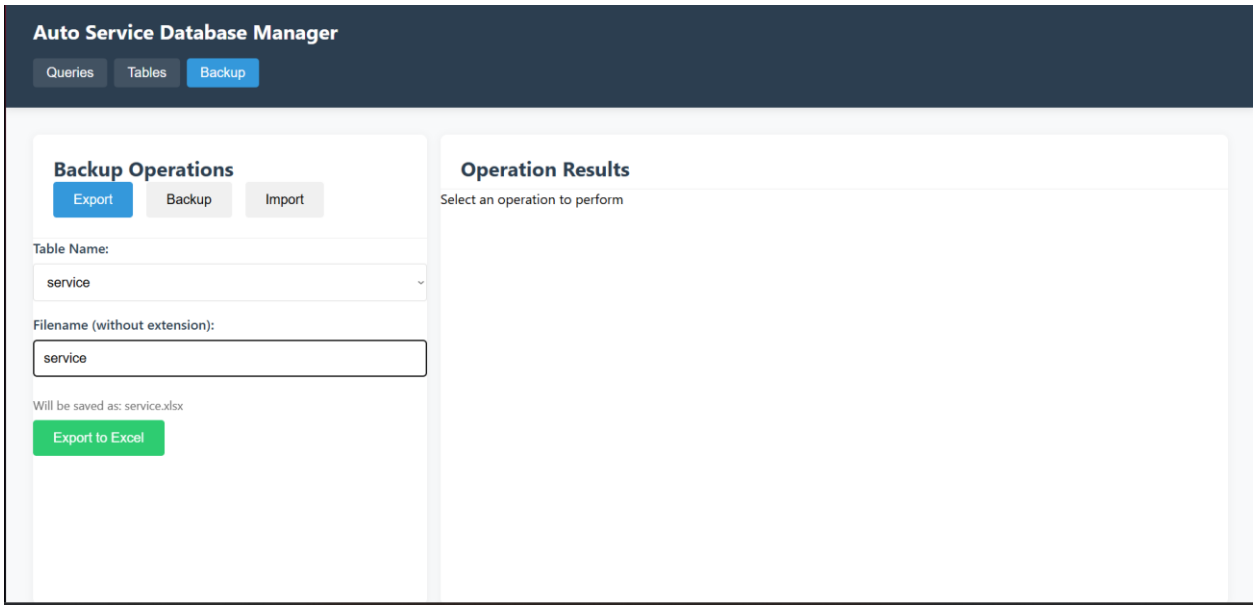

Рисунок 1.7 – Вкладка Export

1.4.2 Вкладка Backup

Элементы управления:

1. Table Backup (Копирование таблицы);
   - выпадающий список Table Name – выбор таблицы;
   - поле Filename – имя резервной копии.

2. Кнопка Backup Table:
   - создает SQL-файл с дампом таблицы;
   - формат: [filename].sql.
3. Full Database Backup (Полная копия БД):
   - поле Filename – имя резервной копии;
   - кнопка Full Backup:
     - создает полный дамп всей базы данных;
     - включает все таблицы, связи и данные;
     - формат: [filename].sql.

Вкладка Backup представлена на рисунке 1.9.

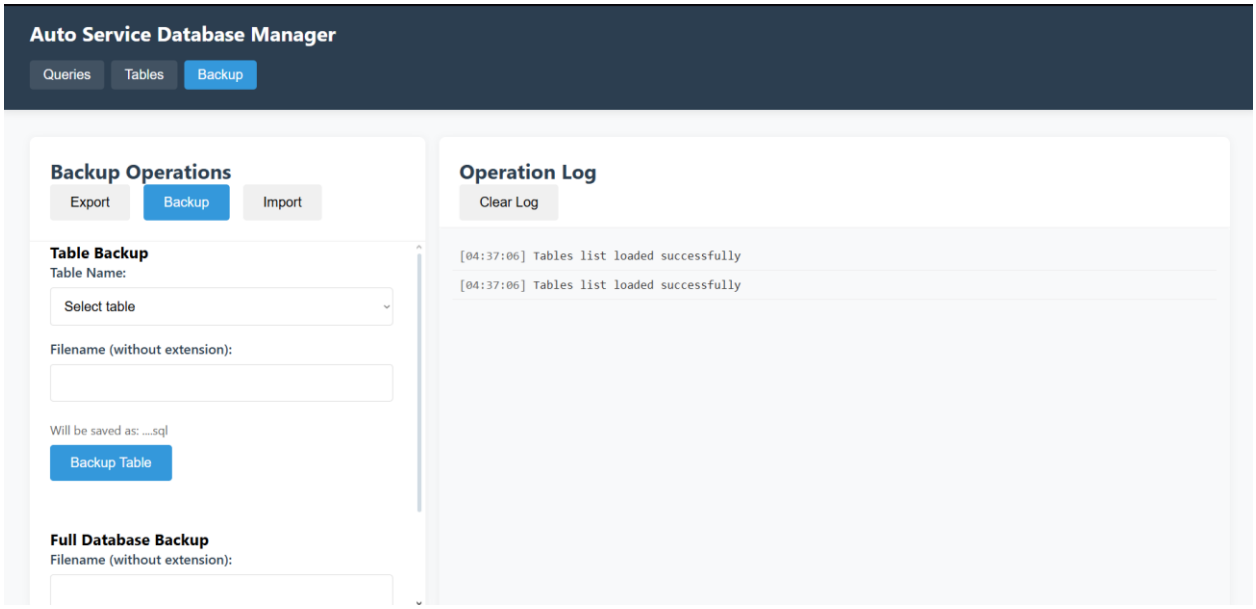

Рисунок 1.9 – Вкладка Backup

1.4.3 Вкладка Import

Элементы управления:

1. Кнопка Select SQL File:
   - открывает диалог выбора файла;
   - поддерживает только .sql файлы;
   - после выбора показывает имя файла.
2. Кнопка Import Backup:
   - загружает и выполняет SQL-скрипт;
   - восстанавливает таблицы и данные.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы было разработано настольное приложение для взаимодействия с базой данных, написанное на языке программирования Java с использованием фреймворка Spring Boot. Основное назначение программы — обеспечение удобного пользовательского интерфейса для выполнения операций с базой данных и визуализации результатов заданных запросов.

Реализована возможность создания, редактирования и удаления таблиц, а также отдельных записей. Приложение поддерживает добавление новых полей, резервное копирование и восстановление как отдельных таблиц, так и всей базы данных. Это обеспечивает стабильную и безопасную работу с данными, минимизируя риск их потери.

Также внедрен модуль управления SQL-запросами, в котором предусмотрен механизм отображения заранее определенных запросов (в соответствии с лабораторными работами №4 и №5), а также возможность добавления новых пользовательских запросов и их сохранения. Результаты запросов можно экспортировать в файл для дальнейшего использования, в том числе в формате, совместимом с Excel.

Таким образом, приложение обеспечивает все необходимые функции для эффективной работы с базой данных в рамках учебного проекта, а также обладает интуитивно понятным интерфейсом, пригодным для дальнейшего расширения и модернизации.

2. ЛИСТИНГ КОДА

QuerryController.java:

```
package com.example.database_backend.controller;

import com.example.database_backend.service.QueryService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Map;

@RestController
@RequestMapping("/queries")
@CrossOrigin(origins = " http://localhost:3000")
public class QueryController {

    @Autowired
    private QueryService queryService;


    @GetMapping("/clients_with_last_name_Iva")
    public List<Map<String, Object>> getClientsWithLastNameIva()
{
        return queryService.getClientsWithLastNameIva();
    }


    @GetMapping("/cars_audi_after_2015")
    public List<Map<String, Object>> getCarsAudiAfter2015() {
        return queryService.getCarsAudiAfter2015();
    }


    @GetMapping("/stos_working_hours_and_address")
    public List<Map<String, Object>>
getStosWorkingHoursAndAddress() {
        return queryService.getStosWorkingHoursAndAddress();
    }


    @GetMapping("/employees_high_salary_street_L")
    public List<Map<String, Object>>
getEmployeesHighSalaryStreetL() {
        return queryService.getEmployeesHighSalaryStreetL();
    }


    @GetMapping("/completed_orders_with_street_address")
    public List<Map<String, Object>>
getCompletedOrdersWithStreetAddress() {
```

```
        return
queryService.getCompletedOrdersWithStreetAddress();
    }


    @GetMapping("/spare_parts_in_expensive_orders")
    public List<Map<String, Object>>
getSparePartsInExpensiveOrders() {
        return queryService.getSparePartsInExpensiveOrders();
    }


    @GetMapping("/employees_low_salary_high_order_amount")
    public List<Map<String, Object>>
getEmployeesLowSalaryHighOrderAmount() {
        return
queryService.getEmployeesLowSalaryHighOrderAmount();
    }


    @GetMapping("/orders_with_bosch_parts")
    public List<Map<String, Object>> getOrdersWithBoschParts() {
        return queryService.getOrdersWithBoschParts();
    }


    @GetMapping("/stos_high_salary_employees_high_order_amount")
    public List<Map<String, Object>>
getStosHighSalaryEmployeesHighOrderAmount() {
        return
queryService.getStosHighSalaryEmployeesHighOrderAmount();
    }


    @GetMapping("/employees_in_stos_with_high_order_amount")
    public List<Map<String, Object>>
getEmployeesInStosWithHighOrderAmount() {
        return
queryService.getEmployeesInStosWithHighOrderAmount();
    }


    @GetMapping("/stos_with_high_salary_employees_and_high_order
_amount")
    public List<Map<String, Object>>
getStosWithHighSalaryEmployeesAndHighOrderAmount() {
        return
queryService.getStosWithHighSalaryEmployeesAndHighOrderAmount();
    }
```

```
    @GetMapping("/stos_with_employees_high_salary_and_high_order
_amount")
    public List<Map<String, Object>>
getStosWithEmployeesHighSalaryAndHighOrderAmount() {
        return
queryService.getStosWithEmployeesHighSalaryAndHighOrderAmount();
    }


    @GetMapping("/total_completed_orders_sum")
    public Double getTotalCompletedOrdersSum() {
        return queryService.getTotalCompletedOrdersSum();
    }


    @GetMapping("/average_service_cost_by_category")
    public List<Map<String, Object>>
getAverageServiceCostByCategory() {
        return queryService.getAverageServiceCostByCategory();
    }


    @GetMapping("/young_clients_count")
    public Integer getYoungClientsCount() {
        return queryService.getYoungClientsCount();
    }


    @GetMapping("/most_expensive_order")
    public Map<String, Object> getMostExpensiveOrder() {
        return queryService.getMostExpensiveOrder();
    }


    @GetMapping("/car_count_by_brand")
    public List<Map<String, Object>> getCarCountByBrand() {
        return queryService.getCarCountByBrand();
    }


    @GetMapping("/orders_with_expensive_services")
    public List<Map<String, Object>>
getOrdersWithExpensiveServices() {
        return queryService.getOrdersWithExpensiveServices();
    }


    @GetMapping("/employees_with_above_average_salary")
    public List<Map<String, Object>>
getEmployeesWithAboveAverageSalary() {
        return
queryService.getEmployeesWithAboveAverageSalary();
```

```
    }


    @GetMapping("/spare_parts_with_low_stock")
    public List<Map<String, Object>> getSparePartsWithLowStock()
{
        return queryService.getSparePartsWithLowStock();
    }


    @GetMapping("/top3_stos_by_employee_count")
    public List<Map<String, Object>>
getTop3StosByEmployeeCount() {
        return queryService.getTop3StosByEmployeeCount();
    }


    @GetMapping("/incomplete_orders")
    public List<Map<String, Object>> getIncompleteOrders() {
        return queryService.getIncompleteOrders();
    }


    @GetMapping("/top5_services")
    public List<Map<String, Object>> getTop5Services() {
        return queryService.getTop5Services();
    }


    @GetMapping("/clients_with_cars")
    public List<Map<String, Object>> getClientsWithCars() {
        return queryService.getClientsWithCars();
    }


    @GetMapping("/total_inventory_value")
    public Double getTotalInventoryValue() {
        return queryService.getTotalInventoryValue();
    }
    @PostMapping("/executeCustom")
    public List<Map<String, Object>>
executeCustomQuery(@RequestParam String query) {
        return queryService.executeCustomQuery(query);
    }


    @PostMapping("/saveCustom")
    public String saveCustomQuery(@RequestParam String
queryName, @RequestParam String query) {
        queryService.saveCustomQuery(queryName, query);
        return "Custom query saved successfully!";
    }
```

```
    @GetMapping("/getCustom")
    public String getSavedCustomQuery(@RequestParam String
queryName) {
        return queryService.getSavedCustomQuery(queryName);
    }

}
```

TabbleController:

```
package com.example.database_backend.controller;

import com.example.database_backend.service.TableService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.Map;

@RestController
@RequestMapping("/tables")
@CrossOrigin(origins = "http://localhost:3000")
public class TableController {

    @Autowired
    private TableService tableService;

    @PostMapping("/create")
     public String createTable(@RequestParam String tableName,
@RequestParam String columns) {
        tableService.createTable(tableName, columns);
        return "Table created successfully!";
    }

    @PutMapping("/update")
    public String updateTable(
            @RequestParam String tableName,
            @RequestParam String columnName,
            @RequestParam String newValue,
            @RequestParam String condition) {
        tableService.updateTable(tableName, columnName, newValue,
condition);
        return "Update successful!";
    }
    @PostMapping("/insert")
    public String insertRow(
            @RequestParam String tableName,
            @RequestBody Map<String, Object> columnValues) {
```

```
        tableService.insertRow(tableName, columnValues);
        return "Insert successful!";
    }


    @DeleteMapping("/drop")
    public String dropTable(@RequestParam String tableName) {
        tableService.dropTable(tableName);
        return "Table dropped successfully!";
    }


    @PostMapping("/addColumn")
      public String addColumn(@RequestParam String tableName,
@RequestParam String columnName, @RequestParam String columnType)
{
       tableService.addColumn(tableName, columnName, columnType);
        return "Column added successfully!";
    }


    @DeleteMapping("/dropColumn")
      public String dropColumn(@RequestParam String tableName,
@RequestParam String columnName) {
        tableService.dropColumn(tableName, columnName);
        return "Column dropped successfully!";
    }


    @GetMapping("/export")
     public String exportToExcel(@RequestParam String tableName,
@RequestParam String filename) {
        try {
            tableService.exportToExcel(tableName, filename);
            return "Export successful!";
        } catch (IOException e) {
            return "Error during export: " + e.getMessage();
        }
    }

    @GetMapping("/backup")
   public String backupTableToSql(@RequestParam String tableName,
@RequestParam String filename) {
        try {
            tableService.backupTableToSql(tableName, filename);
            return "Backup successful!";
        } catch (IOException e) {
            return "Error during backup: " + e.getMessage();
        }
    }
    @GetMapping("/fullBackup")
```

```
       public String fullBackupDatabase(@RequestParam String
filename) {
        try {
            tableService.fullBackupDatabase(filename);
            return "Full backup successful!";
        } catch (IOException e) {
            return "Error during full backup: " + e.getMessage();
        }
    }
    @PostMapping("/loadBackup")
    public String loadBackup(@RequestParam("file") MultipartFile
file) {
        try {

           File tempFile = File.createTempFile("backup", ".sql");
            file.transferTo(tempFile);


            tableService.loadBackup(tempFile.getAbsolutePath());

            return "Backup loaded successfully!";
        } catch (IOException e) {
            return "Error during backup load: " + e.getMessage();
        }
    }
    @GetMapping("/getAllTables")
    public List<String> getAllTables() {
        return tableService.getAllTables();
    }
}

QueryService:
package                    com.example.database_backend.service;

import    org.springframework.beans.factory.annotation.Autowired;
import               org.springframework.jdbc.core.JdbcTemplate;
import                   org.springframework.stereotype.Service;

import                                          java.util.List;
import                                           java.util.Map;

@Service
public             class             QueryService             {

    @Autowired
    private              JdbcTemplate               jdbcTemplate;


    public List<Map<String, Object>> executeQuery(String query) {
        return                  jdbcTemplate.queryForList(query);
    }
```

```
    public List<Map<String, Object>> getClientsWithLastNameIva()
{
        String query = "SELECT last_name, birth_date FROM client
WHERE            last_name            LIKE            '%Ива%'";
        return                                  executeQuery(query);
    }


    public  List<Map<String,  Object>>  getCarsAudiAfter2015()  {
        String query = "SELECT brand, model, manufacture_year FROM
car  WHERE  manufacture_year  >  2015  AND  brand  =  'Audi'";
        return                                  executeQuery(query);
    }


    public                List<Map<String,                Object>>
getStosWorkingHoursAndAddress()                                    {
        String  query  =  "SELECT  name,  address,  phone,  email,
working_hours FROM sto WHERE working_hours LIKE '08:00-20:00' AND
address   LIKE   'ул.   Л%'   AND   phone   LIKE   '+79161%'";
        return                                  executeQuery(query);
    }


    public                List<Map<String,                Object>>
getEmployeesHighSalaryStreetL()                                    {
        String  query  =  "SELECT  e.first_name,  e.last_name,
e.position, e.salary, a.name AS sto_name, a.address FROM employee
e JOIN sto a ON e.sto_id = a.id WHERE a.address LIKE 'ул. Л%' AND
e.salary      >      50000      ORDER      BY      e.last_name";
        return                                  executeQuery(query);
    }


    public                List<Map<String,                Object>>
getCompletedOrdersWithStreetAddress()                              {
        String query = "SELECT o.id, c.first_name, c.last_name,
o.total_amount, o.order_datetime FROM ordertable o JOIN client c
ON o.client_id = c.id JOIN car car ON o.car_id = car.id JOIN sto
sto ON car.id = sto.id WHERE sto.address LIKE '%ул.%' AND o.status
=       'Выполнено'       ORDER       BY       o.order_datetime";
        return                                  executeQuery(query);
    }


    public                List<Map<String,                Object>>
getSparePartsInExpensiveOrders()                                   {
        String query = "SELECT DISTINCT sp.name, sp.manufacturer,
sp.price FROM sparepart sp JOIN ordertable o ON sp.id = o.car_id
WHERE  o.total_amount  >  10000  AND  sp.stock_quantity  <  10";
        return                                  executeQuery(query);
```

```
    }

    public List<Map<String, Object>> getEmployeesLowSalaryHighOrderAmount() {
        String query = "SELECT e.first_name, e.last_name, e.position, e.salary FROM employee e JOIN (SELECT DISTINCT car_id AS sto_id FROM ordertable WHERE total_amount > 5000) o ON e.sto_id = o.sto_id WHERE e.salary < (SELECT AVG(salary) FROM employee) ORDER BY e.salary DESC";
        return executeQuery(query);
    }

    public List<Map<String, Object>> getOrdersWithBoschParts() {
        String query = "SELECT o.id, sp.name, o.total_amount FROM ordertable o JOIN sparepart sp ON o.car_id = sp.id WHERE sp.manufacturer = 'Bosch' AND o.status = 'Выполнено' ORDER BY o.total_amount DESC";
        return executeQuery(query);
    }

    public List<Map<String, Object>> getStosHighSalaryEmployeesHighOrderAmount() {
        String query = "SELECT a.name, a.address, COUNT(e.id) AS high_salary_employees FROM sto a JOIN employee e ON a.id = e.sto_id JOIN ordertable o ON a.id = o.car_id WHERE e.salary > (SELECT AVG(salary) FROM employee) AND o.total_amount > 10000 GROUP BY a.id ORDER BY high_salary_employees DESC";
        return executeQuery(query);
    }

    public List<Map<String, Object>> getEmployeesInStosWithHighOrderAmount() {
        String query = "SELECT e.first_name, e.last_name, e.position, e.salary, a.name AS sto_name FROM employee e JOIN sto a ON e.sto_id = a.id WHERE a.id IN (SELECT DISTINCT car_id FROM ordertable WHERE total_amount > 5000) ORDER BY e.salary DESC";
        return executeQuery(query);
    }

    public List<Map<String, Object>> getStosWithHighSalaryEmployeesAndHighOrderAmount() {
        String query = "SELECT a.name, a.address, COUNT(e.id) AS high_salary_employees FROM sto a JOIN employee e ON a.id = e.sto_id JOIN ordertable o ON a.id = o.car_id WHERE e.salary > 55000 AND o.total_amount > 10000 GROUP BY a.id ORDER BY high_salary_employees DESC";
        return executeQuery(query);
```

```
    }

    public List<Map<String, Object>>
getStosWithEmployeesHighSalaryAndHighOrderAmount() {
        String query = "SELECT a.name, a.address, COUNT(e.id) AS
high_salary_employees FROM sto a JOIN employee e ON a.id = e.sto_id
JOIN ordertable o ON a.id = o.car_id WHERE e.salary > 10000 AND
o.total_amount > 15000 GROUP BY a.id ORDER BY
high_salary_employees DESC";
        return executeQuery(query);
    }

    public Double getTotalCompletedOrdersSum() {
        String query = "SELECT SUM(total_amount) AS
total_completed_orders_sum FROM ordertable WHERE status =
'Выполнено'";
        return jdbcTemplate.queryForObject(query, Double.class);
    }

    public List<Map<String, Object>>
getAverageServiceCostByCategory() {
        String query = "SELECT category, AVG(cost) AS
avg_service_cost FROM service GROUP BY category ORDER BY
avg_service_cost DESC";
        return executeQuery(query);
    }

    public Integer getYoungClientsCount() {
        String query = "SELECT COUNT(*) AS young_clients_count
FROM client WHERE birth_date > '1990-12-31'";
        return jdbcTemplate.queryForObject(query, Integer.class);
    }

    public Map<String, Object> getMostExpensiveOrder() {
        String query = "SELECT o.id, o.total_amount, c.first_name,
c.last_name FROM ordertable o JOIN client c ON o.client_id = c.id
ORDER BY o.total_amount DESC LIMIT 1";
        return jdbcTemplate.queryForMap(query);
    }

    public List<Map<String, Object>> getCarCountByBrand() {
        String query = "SELECT brand, COUNT(*) AS cars_count FROM
car GROUP BY brand ORDER BY cars_count DESC";
        return executeQuery(query);
    }
```

```
    public List<Map<String, Object>>
getOrdersWithExpensiveServices() {
        String query = "SELECT o.id, s.name, s.cost FROM
ordertable o JOIN carservice cs ON o.car_id = cs.car_id JOIN
service s ON cs.service_id = s.id WHERE s.cost > 5000";
        return executeQuery(query);
    }


    public List<Map<String, Object>>
getEmployeesWithAboveAverageSalary() {
        String query = "SELECT first_name, last_name, position,
salary FROM employee WHERE salary > (SELECT AVG(salary) FROM
employee) ORDER BY salary DESC";
        return executeQuery(query);
    }


    public List<Map<String, Object>> getSparePartsWithLowStock()
{
        String query = "SELECT name, manufacturer, stock_quantity
FROM sparepart WHERE stock_quantity < 20 ORDER BY stock_quantity";
        return executeQuery(query);
    }


    public List<Map<String, Object>> getTop3StosByEmployeeCount()
{
        String query = "SELECT s.name, COUNT(e.id) AS
employees_count FROM sto s JOIN employee e ON s.id = e.sto_id GROUP
BY s.name ORDER BY employees_count DESC LIMIT 3";
        return executeQuery(query);
    }


    public List<Map<String, Object>> getIncompleteOrders() {
        String query = "SELECT o.id, c.brand, c.model,
o.total_amount, o.order_datetime FROM ordertable o JOIN car c ON
o.car_id = c.id WHERE o.status = 'В процессе' ORDER BY
o.order_datetime";
        return executeQuery(query);
    }


    public List<Map<String, Object>> getTop5Services() {
        String query = "SELECT s.name, COUNT(cs.id) AS
service_count FROM service s JOIN carservice cs ON s.id =
cs.service_id GROUP BY s.name ORDER BY service_count DESC LIMIT
5";
        return executeQuery(query);
    }
```

```
    public List<Map<String, Object>> getClientsWithCars() {
        String query = "SELECT c.first_name, c.last_name,
car.brand, car.model, car.license_plate FROM client c JOIN
ordertable o ON c.id = o.client_id JOIN car ON o.car_id = car.id
GROUP BY c.id, car.id";
        return executeQuery(query);
    }


    public Double getTotalInventoryValue() {
        String query = "SELECT SUM(price * stock_quantity) AS
total_inventory_value FROM sparepart";
        return jdbcTemplate.queryForObject(query, Double.class);
    }

    public List<Map<String, Object>> executeCustomQuery(String
query) {
        return jdbcTemplate.queryForList(query);
    }


    public void saveCustomQuery(String queryName, String query) {
        String sql = "INSERT INTO saved_queries (name, query)
VALUES (?, ?)";
        jdbcTemplate.update(sql, queryName, query);
    }


    public String getSavedCustomQuery(String queryName) {
        String sql = "SELECT query FROM saved_queries WHERE name
= ?";
        return jdbcTemplate.queryForObject(sql, String.class,
queryName);
    }
}
```

TableService:

```
package com.example.database_backend.service;

import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;

import java.io.*;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map;
```

```
import java.util.stream.Collectors;

@Service
public class TableService {

    @Autowired
    private JdbcTemplate jdbcTemplate;


    public void createTable(String tableName, String columns) {
        String sql = "CREATE TABLE " + tableName + " (" + columns
+ ")";
        jdbcTemplate.execute(sql);
    }


    public void dropTable(String tableName) {
         String sql = "DROP TABLE IF EXISTS " + tableName + "
CASCADE";
        jdbcTemplate.execute(sql);
    }


     public void addColumn(String tableName, String columnName,
String columnType) {
        String sql = "ALTER TABLE " + tableName + " ADD COLUMN "
+ columnName + " " + columnType;
        jdbcTemplate.execute(sql);
    }


    public void dropColumn(String tableName, String columnName) {
        String sql = "ALTER TABLE " + tableName + " DROP COLUMN "
+ columnName;
        jdbcTemplate.execute(sql);
    }


    public void exportToExcel(String tableName, String filename)
throws IOException {
        String sql = "SELECT * FROM " + tableName;
                        List<Map<String,    Object>>    rows    =
jdbcTemplate.queryForList(sql);

        Workbook workbook = new XSSFWorkbook();
        Sheet sheet = workbook.createSheet(tableName);


        Row headerRow = sheet.createRow(0);
        int cellNum = 0;
        for (String columnName : rows.get(0).keySet()) {
```

```
             headerRow.createCell(cellNum++).setCellValue(columnN
ame);
        }


        int rowNum = 1;
        for (Map<String, Object> rowData : rows) {
            Row row = sheet.createRow(rowNum++);
            cellNum = 0;
            for (Object value : rowData.values()) {
                 row.createCell(cellNum++).setCellValue(value.toS
tring());
            }
        }

                  try    (FileOutputStream    fileOut    =    new
FileOutputStream(filename)) {
            workbook.write(fileOut);
        }
    }


        public   void   backupTable(String   tableName,   String
backupTableName) {
         String sql = "CREATE TABLE " + backupTableName + " AS
SELECT * FROM " + tableName;
        jdbcTemplate.execute(sql);
    }


    public void backupDatabase(String filename) throws IOException
{
        List<String> tableNames = jdbcTemplate.queryForList("SHOW
TABLES", String.class);

        Workbook workbook = new XSSFWorkbook();

        for (String tableName : tableNames) {
            String sql = "SELECT * FROM " + tableName;
                          List<Map<String,   Object>>   rows   =
jdbcTemplate.queryForList(sql);

            Sheet sheet = workbook.createSheet(tableName);


            Row headerRow = sheet.createRow(0);
            int cellNum = 0;
            if (!rows.isEmpty()) {
                for (String columnName : rows.get(0).keySet()) {
                     headerRow.createCell(cellNum++).setCellValue
(columnName);
                }
```

```
            }


            int rowNum = 1;
            for (Map<String, Object> rowData : rows) {
                Row row = sheet.createRow(rowNum++);
                cellNum = 0;
                for (Object value : rowData.values()) {
                     row.createCell(cellNum++).setCellValue(value
.toString());
                }
            }
        }

                    try    (FileOutputStream    fileOut    =    new
FileOutputStream(filename)) {
            workbook.write(fileOut);
        }
    }

       public  void  backupTableToSql(String  tableName,  String
filename) throws IOException {
        String sql = "SELECT * FROM " + tableName;
                        List<Map<String,     Object>>     rows     =
jdbcTemplate.queryForList(sql);

        try (FileWriter fileWriter = new FileWriter(filename)) {

            fileWriter.write("DROP TABLE IF EXISTS `" + tableName
+ "` CASCADE;\n");
             fileWriter.write("CREATE TABLE `" + tableName + "`
(\n");


                         List<Map<String,   Object>>   columns   =
jdbcTemplate.queryForList(
                              "SELECT  COLUMN_NAME,  DATA_TYPE  FROM
INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = ?",
                    tableName);

            for (Map<String, Object> column : columns) {
                                String   columnName   =   (String)
column.get("COLUMN_NAME");
                                String   columnType   =   (String)
column.get("DATA_TYPE");
                  fileWriter.write("    `" + columnName + "` " +
columnType + ",\n");
            }
            fileWriter.write("    PRIMARY KEY (`id`)\n");
            fileWriter.write(");\n\n");
```

```
            for (Map<String, Object> row : rows) {
                fileWriter.write("INSERT INTO `" + tableName + "`
VALUES (");
                for (Object value : row.values()) {
                    if (value instanceof String) {
                        fileWriter.write("'" + value + "', ");
                    } else {
                        fileWriter.write(value + ", ");
                    }
                }
                fileWriter.write(");\n");
            }
        }
    }

    public void updateTable(String tableName, String columnName,
String newValue, String condition) {
        String sql = "UPDATE " + tableName + " SET " + columnName
+ " = ? WHERE " + condition;
        jdbcTemplate.update(sql, newValue);
    }

     public void insertRow(String tableName, Map<String, Object>
columnValues) {
        if (!columnValues.containsKey("id")) {
              Integer maxId = jdbcTemplate.queryForObject("SELECT
MAX(id) FROM " + tableName, Integer.class);
            int newId = (maxId != null) ? maxId + 1 : 1;
            columnValues.put("id", newId);
        }

           StringBuilder sql = new StringBuilder("INSERT INTO
").append(tableName).append(" (");
        StringBuilder values = new StringBuilder(" VALUES (");

        columnValues.forEach((column, value) -> {
            sql.append(column).append(", ");
            values.append("?, ");
        });

        sql.setLength(sql.length() - 2);
        values.setLength(values.length() - 2);
        values.append(")");

        sql.append(") ").append(values.toString());

                                jdbcTemplate.update(sql.toString(),
columnValues.values().toArray());
    }

      public void fullBackupDatabase(String filename) throws
IOException {
```

```
        try (FileWriter writer = new FileWriter(filename)) {

            List<String> tables = getTablesInDependencyOrder();


            writer.write("BEGIN;\n\n");


            writer.write("-- Удаление существующих таблиц\n");
            for (int i = tables.size() - 1; i >= 0; i--) {
                writer.write(String.format("DROP TABLE IF EXISTS
\"%s\" CASCADE;\n", tables.get(i)));
            }
            writer.write("\n");


            writer.write("-- Создание структуры таблиц\n");
            for (String table : tables) {
               writer.write(getTableCreationDDL(table) + "\n\n");
            }


            writer.write("-- Вставка данных\n");
            for (String table : tables) {
                            List<Map<String,  Object>>  rows  =
jdbcTemplate.queryForList("SELECT * FROM \"" + table + "\"");
                if (!rows.isEmpty()) {
                                List<String>  columns  =  new
ArrayList<>(rows.get(0).keySet());

                    writer.write("-- Данные таблицы: " + table +
"\n");
                    for (Map<String, Object> row : rows) {
                        writer.write(buildInsertStatement(table,
columns, row) + "\n");
                    }
                    writer.write("\n");
                }
            }


            writer.write("-- Восстановление внешних ключей\n");
            for (String table : tables) {
                String fkScript = getForeignKeysDDL(table);
                if (!fkScript.isEmpty()) {
                    writer.write(fkScript);
                }
            }


            writer.write("\nCOMMIT;\n");
            writer.write("-- Бэкап успешно завершен\n");
```

```
        } catch (Exception e) {
            throw new IOException("Ошибка при создании бэкапа: "
+ e.getMessage(), e);
        }
    }

    private List<String> getTablesInDependencyOrder() {

        List<String> allTables = jdbcTemplate.queryForList(
                "SELECT table_name FROM information_schema.tables
" +
                                "WHERE table_schema = 'public' AND
table_type = 'BASE TABLE'",
                String.class);


                  Map<String,   List<String>>   dependencies   =
jdbcTemplate.queryForList(
                         "SELECT tc.table_name, ccu.table_name AS
referenced_table " +
                                                                "FROM
information_schema.table_constraints tc " +
                                                                "JOIN
information_schema.constraint_column_usage ccu " +
                                            "ON tc.constraint_name =
ccu.constraint_name " +
                                "WHERE tc.constraint_type = 'FOREIGN
KEY' AND tc.table_schema = 'public'")
                .stream()
                .collect(Collectors.groupingBy(
                        row -> (String) row.get("table_name"),
                        Collectors.mapping(
                                                row  ->  (String)
row.get("referenced_table"),
                                    Collectors.toList()
                        )
                ));


        List<String> sortedTables = new ArrayList<>();
        List<String> remainingTables = new ArrayList<>(allTables);

        while (!remainingTables.isEmpty()) {
            boolean changed = false;
            for (int i = 0; i < remainingTables.size(); i++) {
                String table = remainingTables.get(i);
                                            List<String>    deps    =
dependencies.getOrDefault(table, Collections.emptyList());

                if (sortedTables.containsAll(deps)) {
                    sortedTables.add(table);
```

```
                    remainingTables.remove(i);
                    changed = true;
                    break;
                }
            }

            if (!changed) {

                sortedTables.addAll(remainingTables);
                remainingTables.clear();
            }
        }

        return sortedTables;
    }

    private String getTableCreationDDL(String tableName) {

                    List<Map<String,    Object>>    columns    =
jdbcTemplate.queryForList(
                   "SELECT  column_name,  data_type,  is_nullable,
column_default " +
                        "FROM information_schema.columns " +
                             "WHERE  table_schema  =  'public'  AND
table_name = ? " +
                        "ORDER BY ordinal_position",
                tableName);


        List<String> primaryKeys = jdbcTemplate.queryForList(
                                    "SELECT    column_name    FROM
information_schema.key_column_usage " +
                                "WHERE  table_schema  =  'public'  AND
table_name = ? AND constraint_name = ?",
                String.class,
                tableName,
                tableName + "_pkey");

        StringBuilder ddl = new StringBuilder("CREATE TABLE \"" +
tableName + "\" (\n");


        List<String> serialColumns = new ArrayList<>();


        for (Map<String, Object> column : columns) {
           String columnName = (String) column.get("column_name");
            String dataType = (String) column.get("data_type");
           String isNullable = (String) column.get("is_nullable");
                           String   columnDefault   =   (String)
column.get("column_default");
```

```
                boolean isSerial = columnDefault != null &&
columnDefault.startsWith("nextval(");

            if (isSerial) {

                if (dataType.equals("integer")) {
                     ddl.append("  \"").append(columnName).append
("\" SERIAL");
                    serialColumns.add(columnName);
                } else if (dataType.equals("bigint")) {
                     ddl.append("  \"").append(columnName).append
("\" BIGSERIAL");
                    serialColumns.add(columnName);
                } else {

                     ddl.append("  \"").append(columnName).append
("\" ").append(dataType)
                                                        .append("  DEFAULT
").append(columnDefault);
                }
            } else {

                 ddl.append("  \"").append(columnName).append("\"
").append(dataType);

                if ("NO".equals(isNullable)) {
                    ddl.append(" NOT NULL");
                }

                                if (columnDefault != null &&
!columnDefault.isEmpty()) {
                   ddl.append(" DEFAULT ").append(columnDefault);
                }
            }

            ddl.append(",\n");
        }


        if (!primaryKeys.isEmpty()) {
            ddl.append("  PRIMARY KEY (");
            ddl.append(primaryKeys.stream()
                    .map(key -> "\"" + key + "\"")
                    .collect(Collectors.joining(", ")));
            ddl.append("),\n");
        }


        if (ddl.charAt(ddl.length()-2) == ',') {
            ddl.delete(ddl.length()-2, ddl.length());
        }
        ddl.append("\n);");
```

```
        return ddl.toString();
    }

    private String getForeignKeysDDL(String tableName) {
        StringBuilder fkDdl = new StringBuilder();


                    List<Map<String,    Object>>    foreignKeys    =
jdbcTemplate.queryForList(
                "SELECT tc.constraint_name, kcu.column_name, " +
                        "ccu.table_name AS foreign_table_name, "
+
                        "ccu.column_name AS foreign_column_name,
" +
                       "rc.update_rule, rc.delete_rule " +
                      "FROM information_schema.table_constraints
tc " +
                       "JOIN information_schema.key_column_usage
kcu " +
                    "ON tc.constraint_name = kcu.constraint_name
" +
                                                                "JOIN
information_schema.constraint_column_usage ccu " +
                    "ON ccu.constraint_name = tc.constraint_name
" +
                                                                "JOIN
information_schema.referential_constraints rc " +
                     "ON rc.constraint_name = tc.constraint_name
" +
                       "WHERE tc.constraint_type = 'FOREIGN KEY'
" +
                       "AND tc.table_schema = 'public' " +
                       "AND tc.table_name = ?",
                tableName);

        for (Map<String, Object> fk : foreignKeys) {
                         String   constraintName   =   (String)
fk.get("constraint_name");
            String columnName = (String) fk.get("column_name");
                          String   foreignTable   =   (String)
fk.get("foreign_table_name");
                         String   foreignColumn   =   (String)
fk.get("foreign_column_name");
            String updateRule = (String) fk.get("update_rule");
            String deleteRule = (String) fk.get("delete_rule");

                                       fkDdl.append("ALTER     TABLE
\"").append(tableName).append("\" ")
                                            .append("ADD   CONSTRAINT
\"").append(constraintName).append("\" ")
```

```
                                                .append("FOREIGN   KEY
(\"").append(columnName).append("\") ")
                                                  .append("REFERENCES
\"").append(foreignTable).append("\"
(\"").append(foreignColumn).append("\") ")
                                                .append("ON   UPDATE
").append(updateRule).append(" ")
                                                .append("ON   DELETE
").append(deleteRule).append(";\n");
        }

        return fkDdl.toString();
    }

       private   String   buildInsertStatement(String   tableName,
List<String> columns, Map<String, Object> row) {
         StringBuilder sb = new StringBuilder("INSERT INTO \"" +
tableName + "\" (");


        sb.append(columns.stream()
                .map(col -> "\"" + col + "\"")
                .collect(Collectors.joining(", ")));
        sb.append(") VALUES (");


        List<String> values = new ArrayList<>();
        for (String column : columns) {
            Object value = row.get(column);
            values.add(formatValueForSql(column, value));
        }
        sb.append(String.join(", ", values));
        sb.append(");");

        return sb.toString();
    }

    private String formatValueForSql(String column, Object value)
{
        if (value == null) {
            return "NULL";
        }

        if (value instanceof String) {
              return "'" + value.toString().replace("'", "''") +
"'";
        }

        if (value instanceof java.util.Date) {
                                      return     "'"     +     new
java.sql.Timestamp(((java.util.Date)value).getTime()) + "'";
        }
```

```
        if (value instanceof Boolean) {
            return (Boolean)value ? "TRUE" : "FALSE";
        }

                        if     ("duration".equals(column)     ||
"warranty_period".equals(column)) {
            return formatInterval(value.toString());
        }

        return value.toString();
    }

    private String formatInterval(String intervalStr) {
        String[] parts = intervalStr.split(" ");
        StringBuilder result = new StringBuilder("'");

        try {
            int years = Integer.parseInt(parts[0]);
            int months = Integer.parseInt(parts[2]);
            int days = Integer.parseInt(parts[4]);
            int hours = Integer.parseInt(parts[6]);
            int minutes = Integer.parseInt(parts[8]);
            double seconds = Double.parseDouble(parts[10]);

           if (years > 0) result.append(years).append(" years ");
            if (months > 0) result.append(months).append(" months
");
            if (days > 0) result.append(days).append(" days ");
           if (hours > 0) result.append(hours).append(" hours ");
               if  (minutes  >  0)  result.append(minutes).append("
minutes ");
               if  (seconds  >  0)  result.append(seconds).append("
seconds ");

            if (result.charAt(result.length()-1) == ' ') {
                result.deleteCharAt(result.length()-1);
            }
        } catch (Exception e) {
            return "'" + intervalStr + "'::interval";
        }

        result.append("'::interval");
        return result.toString();
    }
    public void loadBackup(String filename) throws IOException {
          try  (BufferedReader  reader  =  new  BufferedReader(new
FileReader(filename))) {
            String line;
            StringBuilder sql = new StringBuilder();

            while ((line = reader.readLine()) != null) {
```

```
                                    if   (line.trim().isEmpty()   ||
line.trim().startsWith("--")) {
                        continue;
                    }

                    sql.append(line);

                    if (line.trim().endsWith(";")) {
                        try {
                            jdbcTemplate.execute(sql.toString());
                        } catch (Exception e) {
                             System.err.println("Error executing SQL:
" + sql.toString());
                            throw e;
                        }
                        sql.setLength(0);
                    }
                }
            }
        }
        public List<String> getAllTables() {
            return jdbcTemplate.queryForList(
                    "SELECT table_name FROM information_schema.tables
" +
                                    "WHERE table_schema = 'public' AND
table_type = 'BASE TABLE'",
                    String.class);
        }
}
```

BackupView.js:

```
import React, { useState, useEffect } from 'react';
import api from '../api';

const BackupView = ({ showNotification }) => {
  const [activeTab, setActiveTab] = useState('export');
  const [formData, setFormData] = useState({
    tableName: '',
    filename: ''
  });
  const [file, setFile] = useState(null);
  const [tables, setTables] = useState([]);
  const [operationLog, setOperationLog] = useState([]);

  useEffect(() => {
    const fetchTables = async () => {
      try {
        const response = await api.getAllTables();
        setTables(response.data);
        addToLog('Tables list loaded successfully');
      } catch (err) {
```

```
                showNotification(err.response?.data?.message   ||
err.message, 'error');
        addToLog(`Error loading tables: ${err.message}`);
      }
    };
    fetchTables();
  }, []);

  const addToLog = (message) => {
    setOperationLog(prev => [
      { timestamp: new Date().toLocaleTimeString(), message },
      ...prev.slice(0, 50)
    ]);
  };

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value.replace(/\./g, '')
    });
  };

  const handleFileChange = (e) => {
    setFile(e.target.files[0]);
    addToLog(`Selected file: ${e.target.files[0]?.name}`);
  };

  const handleExport = async () => {
     addToLog(`Starting export of table ${formData.tableName} to
Excel`);
    try {
                  await     api.exportToExcel(formData.tableName,
formData.filename);
        showNotification(`Exported  to  ${formData.filename}.xlsx
successfully`);
                      addToLog(`Successfully      exported      to
${formData.filename}.xlsx`);
    } catch (err) {
      showNotification(err.response?.data?.message || err.message,
'error');
      addToLog(`Export failed: ${err.message}`);
    }
  };

  const handleBackup = async () => {
    addToLog(`Starting backup of table ${formData.tableName}`);
    try {
                    await     api.backupTable(formData.tableName,
formData.filename);
       showNotification(`Backup ${formData.filename}.sql created
successfully`);
      addToLog(`Table backup created: ${formData.filename}.sql`);
```

```
    } catch (err) {
      showNotification(err.response?.data?.message || err.message,
'error');
      addToLog(`Backup failed: ${err.message}`);
    }
  };

  const handleFullBackup = async () => {
    addToLog('Starting full database backup');
    try {
      await api.fullBackup(formData.filename);
         showNotification(`Full  backup  ${formData.filename}.sql
created successfully`);
      addToLog(`Full backup created: ${formData.filename}.sql`);
    } catch (err) {
      showNotification(err.response?.data?.message || err.message,
'error');
      addToLog(`Full backup failed: ${err.message}`);
    }
  };

  const handleImport = async () => {
    if (!file) {
      showNotification('Please select a file first', 'error');
      addToLog('Import attempted without file selection');
      return;
    }

    addToLog(`Starting import from file: ${file.name}`);
    try {
      await api.loadBackup(file);
      showNotification('Backup imported successfully');
      addToLog('Backup imported successfully');

      const response = await api.getAllTables();
      setTables(response.data);
    } catch (err) {
      showNotification(err.response?.data?.message || err.message,
'error');
      addToLog(`Import failed: ${err.message}`);
    }
  };

  return (
    <div className="main-container">
      <div className="panel backup-operations-panel">
        <div className="panel-header">
          <h2>Backup Operations</h2>
          <div className="form-row">
            <button
              onClick={() => setActiveTab('export')}
```

```
                  className={`button ${activeTab === 'export' ?
'button-primary' : ''}`}
              >
                Export
              </button>
              <button
                onClick={() => setActiveTab('backup')}
                    className={`button ${activeTab === 'backup' ?
'button-primary' : ''}`}
              >
                Backup
              </button>
              <button
                onClick={() => setActiveTab('import')}
                    className={`button ${activeTab === 'import' ?
'button-primary' : ''}`}
              >
                Import
              </button>
            </div>
          </div>

          <div className="backup-content">
            {activeTab === 'export' && (
              <div className="form-group">
                <label>Table Name:</label>
                <select
                  name="tableName"
                  value={formData.tableName}
                  onChange={handleChange}
                  className="form-select"
                >
                  <option value="">Select table</option>
                  {tables.map(table => (
                                                <option    key={table}
value={table}>{table}</option>
                  ))}
                </select>

                <label>Filename (without extension):</label>
                <input
                  type="text"
                  name="filename"
                  value={formData.filename}
                  onChange={handleChange}
                />
                   <p  className="filename-hint">Will  be  saved  as:
{formData.filename || '...'}.xlsx</p>

                <button
                  onClick={handleExport}
                  className="button button-success"
```

```
                                disabled={!formData.tableName   ||
!formData.filename}
              >
                Export to Excel
              </button>
            </div>
          )}

          {activeTab === 'backup' && (
            <div className="backup-tabs-container">
              <div className="form-group">
                <h3>Table Backup</h3>
                <label>Table Name:</label>
                <select
                  name="tableName"
                  value={formData.tableName}
                  onChange={handleChange}
                  className="form-select"
                >
                  <option value="">Select table</option>
                  {tables.map(table => (
                                          <option   key={table}
value={table}>{table}</option>
                  ))}
                </select>

                <label>Filename (without extension):</label>
                <input
                  type="text"
                  name="filename"
                  value={formData.filename}
                  onChange={handleChange}
                />
                  <p className="filename-hint">Will be saved as:
{formData.filename || '...'}.sql</p>

                <button
                  onClick={handleBackup}
                  className="button button-primary"
                               disabled={!formData.tableName   ||
!formData.filename}
                >
                  Backup Table
                </button>
              </div>

              <div className="form-group">
                <h3>Full Database Backup</h3>
                <label>Filename (without extension):</label>
                <input
                  type="text"
                  name="filename"
```

```
                  value={formData.filename}
                  onChange={handleChange}
                />
                  <p className="filename-hint">Will be saved as:
{formData.filename || '...'}.sql</p>

                <button
                  onClick={handleFullBackup}
                  className="button button-primary"
                  disabled={!formData.filename}
                >
                  Full Backup
                </button>
              </div>
            </div>
          )}

          {activeTab === 'import' && (
            <div className="form-group">
              <input
                type="file"
                id="backupFile"
                className="file-input"
                onChange={handleFileChange}
                accept=".sql"
              />
              <label htmlFor="backupFile" className="button">
                Select SQL File
              </label>
              {file && <p>Selected: {file.name}</p>}

              <button
                onClick={handleImport}
                className="button button-primary"
                disabled={!file}
              >
                Import Backup
              </button>
            </div>
          )}
        </div>
      </div>

      <div className="panel results-panel">
        <div className="panel-header">
          <h2>Operation Log</h2>
          <button
            className="button button-small"
            onClick={() => setOperationLog([])}
          >
            Clear Log
          </button>
```

```
        </div>
        <div className="log-container">
          {operationLog.length > 0 ? (
            <div className="log-content">
              {operationLog.map((entry, index) => (
                <div key={index} className="log-entry">
                                        <span    className="log-
time">[{entry.timestamp}]</span>
                                        <span    className="log-
message">{entry.message}</span>
                </div>
              ))}
            </div>
          ) : (
            <p>No operations performed yet</p>
          )}
        </div>
      </div>
    </div>
  );
};

export default BackupView;

QueryView.js:
import React, { useState, useEffect } from 'react';
import api from '../api';

const QueryView = ({ showNotification }) => {
  const [queryResults, setQueryResults] = useState(null);
  const [loading, setLoading] = useState(false);
  const [customQuery, setCustomQuery] = useState('');
  const [savedQueries, setSavedQueries] = useState([]);
  const [columns, setColumns] = useState([]);
       const      [activeQueryTab,      setActiveQueryTab]      =
useState('predefined');
  const [resultType, setResultType] = useState(null);


  useEffect(() => {
    const fetchSavedQueries = async () => {
      try {
         const response = await api.executeCustomQuery('SELECT *
FROM saved_queries');
        setSavedQueries(response.data);
      } catch (err) {
                showNotification(err.response?.data?.message   ||
err.message, 'error');
      }
    };
    fetchSavedQueries();
  }, [showNotification]);
```

```
  const formatInterval = (value) => {
    if (typeof value === 'object' && value !== null) {
      if (value.hours !== undefined || value.minutes !== undefined
|| value.seconds !== undefined) {
        const hours = value.hours || 0;
        const minutes = value.minutes || 0;
        const seconds = value.seconds || 0;
        return `${hours}h ${minutes}m ${seconds}s`;
      }
      return JSON.stringify(value);
    }
    return String(value);
  };


  const executePredefinedQuery = async (endpoint, name) => {
    setLoading(true);
    try {
      const response = await api.executePredefinedQuery(endpoint);
      processQueryResults(response.data, name);
    } catch (err) {
      showNotification(err.response?.data?.message || err.message,
'error');
    } finally {
      setLoading(false);
    }
  };


  const executeCustomQueryHandler = async () => {
    if (!customQuery.trim()) {
      showNotification('Please enter a query', 'error');
      return;
    }

    setLoading(true);
    try {
      const response = await api.executeCustomQuery(customQuery);
      processQueryResults(response.data, 'Custom Query');
    } catch (err) {
      showNotification(err.response?.data?.message || err.message,
'error');
    } finally {
      setLoading(false);
    }
  };

  const processQueryResults = (data, queryName) => {
```

```
    if (typeof data === 'string' || typeof data === 'number' ||
data === null) {
        setQueryResults(String(data));
        setResultType('scalar');
              showNotification(`Query   "${queryName}"   executed
successfully`);
        return;
    }


    if (typeof data === 'object' && !Array.isArray(data)) {
        const processedData = [data].map(row => {
            const processedRow = {};
            for (const [key, value] of Object.entries(row)) {
                 processedRow[key] = typeof value === 'object' ?
JSON.stringify(value) : String(value);
            }
            return processedRow;
        });

        setQueryResults(processedData);
                    setColumns(processedData.length    >    0    ?
Object.keys(processedData[0]) : []);
        setResultType('table');
              showNotification(`Query   "${queryName}"   executed
successfully`);
        return;
    }


    if (Array.isArray(data)) {
        const processedData = data.map(row => {
            const processedRow = {};
            for (const [key, value] of Object.entries(row)) {
                 processedRow[key] = typeof value === 'object' ?
JSON.stringify(value) : String(value);
            }
            return processedRow;
        });

        setQueryResults(processedData);
                    setColumns(processedData.length    >    0    ?
Object.keys(processedData[0]) : []);
        setResultType('table');
              showNotification(`Query   "${queryName}"   executed
successfully`);
        return;
    }


    setQueryResults(null);
    setResultType(null);
```

```
     showNotification('Unexpected response format from server',
'error');
};


  const saveQuery = async () => {
    if (!customQuery.trim()) {
      showNotification('Please enter a query to save', 'error');
      return;
    }

    try {
      const queryName = prompt('Enter a name for this query:') ||
`Custom Query ${savedQueries.length + 1}`;


      await api.executeCustomQuery(
            `INSERT INTO saved_queries (name, query) VALUES
('${queryName}', '${customQuery}')`
      );


      const response = await api.executeCustomQuery('SELECT * FROM
saved_queries');
      setSavedQueries(response.data);

      showNotification('Query saved successfully');
      setCustomQuery('');
    } catch (err) {
     showNotification(err.response?.data?.message || err.message,
'error');
    }
  };


  const deleteSavedQuery = async (id) => {
    try {
       await api.executeCustomQuery(`DELETE FROM saved_queries
WHERE id = ${id}`);
      const response = await api.executeCustomQuery('SELECT * FROM
saved_queries');
      setSavedQueries(response.data);
      showNotification('Query deleted successfully');
    } catch (err) {
     showNotification(err.response?.data?.message || err.message,
'error');
    }
  };


  const predefinedQueries = [
```

```
    { name: "Clients with last name starting with 'Iva'", endpoint:
"clients_with_last_name_Iva" },
         { name: "Audi cars after 2015", endpoint:
"cars_audi_after_2015" },
        { name: "STOs with working hours", endpoint:
"stos_working_hours_and_address" },
     { name: "High salary employees on 'L' streets", endpoint:
"employees_high_salary_street_L" },
      { name: "Completed orders with addresses", endpoint:
"completed_orders_with_street_address" },
       { name: "Parts in expensive orders", endpoint:
"spare_parts_in_expensive_orders" },
    { name: "Low salary employees in expensive orders", endpoint:
"employees_low_salary_high_order_amount" },
        { name: "Orders with Bosch parts", endpoint:
"orders_with_bosch_parts" },
      { name: "STOs with high salary employees", endpoint:
"stos_high_salary_employees_high_order_amount" },
    { name: "Employees in STOs with expensive orders", endpoint:
"employees_in_stos_with_high_order_amount" },
    { name: "STOs with high salary employees and expensive orders",
endpoint: "stos_with_high_salary_employees_and_high_order_amount"
},
     { name: "STOs with employees earning >10,000 and expensive
orders",                                                endpoint:
"stos_with_employees_high_salary_and_high_order_amount" },
      { name: "Total sum of completed orders", endpoint:
"total_completed_orders_sum" },
      { name: "Average service cost by category", endpoint:
"average_service_cost_by_category" },
        { name: "Clients born after 1990", endpoint:
"young_clients_count" },
         { name: "Most expensive order", endpoint:
"most_expensive_order" },
    { name: "Car count by brand", endpoint: "car_count_by_brand"
},
      { name: "Orders with services >5000 rub", endpoint:
"orders_with_expensive_services" },
     { name: "Employees with above average salary", endpoint:
"employees_with_above_average_salary" },
        { name: "Parts with stock <20", endpoint:
"spare_parts_with_low_stock" },
      { name: "Top 3 STOs by employee count", endpoint:
"top3_stos_by_employee_count" },
    { name: "Incomplete orders", endpoint: "incomplete_orders" },
      { name: "Top 5 most requested services", endpoint:
"top5_services" },
        { name: "Clients with their cars", endpoint:
"clients_with_cars" },
         { name: "Total inventory value", endpoint:
"total_inventory_value" }
  ];
```

```
  return (
    <div className="main-container">
      <div className="panel">
        <div className="panel-header">
          <h2>Query Manager</h2>
          <div className="query-tabs">
            <button
                    className={`tab-button  ${activeQueryTab  ===
'predefined' ? 'active' : ''}`}
              onClick={() => setActiveQueryTab('predefined')}
            >
              Predefined Queries
            </button>
            <button
             className={`tab-button ${activeQueryTab === 'custom'
? 'active' : ''}`}
              onClick={() => setActiveQueryTab('custom')}
            >
              Custom SQL
            </button>
          </div>
        </div>

        {activeQueryTab === 'predefined' ? (
          <div className="queries-container">
            <div className="query-grid">
              {predefinedQueries.map((query, index) => (
                <div key={index} className="query-card">
                  <h3>{query.name}</h3>
                  <button
                                                  onClick={()    =>
executePredefinedQuery(query.endpoint, query.name)}
                    className="button button-primary"
                    disabled={loading}
                  >
                    Execute
                  </button>
                </div>
              ))}
            </div>
          </div>
        ) : (
          <>
            <div className="form-group">
              <textarea
                value={customQuery}
                onChange={(e) => setCustomQuery(e.target.value)}
                placeholder="Enter your SQL query here"
                rows={4}
              />
              <div className="form-row">
```

```
                <button
                  onClick={executeCustomQueryHandler}
                  className="button button-primary"
                  disabled={loading || !customQuery.trim()}
                >
                  Execute
                </button>
                <button
                  onClick={saveQuery}
                  className="button button-success"
                  disabled={loading || !customQuery.trim()}
                >
                  Save
                </button>
              </div>
            </div>

            <div className="queries-container">
              <h3>Saved Queries</h3>
              <div className="query-grid">
                {savedQueries.map((query) => (
                  <div key={query.id} className="query-card">
                    <h3>{query.name}</h3>
                    <p className="query-text">{query.query}</p>
                    <div className="query-actions">
                      <button
                        onClick={() => {
                          setCustomQuery(query.query);
                          executeCustomQueryHandler();
                        }}
                        className="button button-primary"
                        disabled={loading}
                      >
                        Execute
                      </button>
                      <button
                        onClick={() => {
                            if (window.confirm('Delete this saved
query?')) {
                             deleteSavedQuery(query.id);
                          }
                        }}
                        className="button button-danger"
                      >
                        Delete
                      </button>
                    </div>
                  </div>
                ))}
              </div>
            </div>
          </>
```

```
        )}
      </div>

      <div className="panel results-panel">
        <div className="panel-header">
          <h2>Query Results</h2>
        </div>

        {loading ? (
          <div className="loading">Loading data...</div>
        ) : (
          <div className="results-container">
            {resultType === 'table' && (
              <div className="table-container">
                <table>
                  <thead>
                    <tr>
                      {columns.map((key) => (
                        <th key={key}>{key}</th>
                      ))}
                    </tr>
                  </thead>
                  <tbody>
                    {queryResults && queryResults.length > 0 ? (
                      queryResults.map((row, index) => (
                        <tr key={index}>
                          {columns.map((key) => (
                            <td key={`${index}-${key}`}>
                                  {key === 'duration' || key ===
'warranty_period'
                                ? formatInterval(row[key])
                                : String(row[key])}
                            </td>
                          ))}
                        </tr>
                      ))
                    ) : (
                      <tr>
                              <td colSpan={columns.length || 1}
className="no-data">
                          No data returned
                        </td>
                      </tr>
                    )}
                  </tbody>
                </table>
              </div>
            )}

            {resultType === 'scalar' && (
              <div className="scalar-result">
                <div className="scalar-result-header">
```

```
                  <h3>Result</h3>
                </div>
                <div className="scalar-result-value">
                  {queryResults}
                </div>
              </div>
            )}

            {resultType === null && (
              <div className="no-results">
                Execute a query to see results
              </div>
            )}
          </div>
        )}
      </div>
    </div>
  );
};

export default QueryView;

TableManagement.js:
import React, { useState, useEffect } from 'react';
import api from '../api';

const TableManagement = ({ showNotification }) => {
  const [activeTab, setActiveTab] = useState('browse');
  const [tables, setTables] = useState([]);
  const [tableData, setTableData] = useState([]);
  const [selectedTable, setSelectedTable] = useState('');
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);
  const [formData, setFormData] = useState({
    tableName: '',
    columns: '',
    columnName: '',
    columnType: 'VARCHAR(255)',
    condition: '',
    newValue: '',
    rowData: '{}'
  });
  const [editingRow, setEditingRow] = useState(null);
  const [editFormData, setEditFormData] = useState({});
  const [columns, setColumns] = useState([]);

  const formatInterval = (value) => {
    if (typeof value === 'object' && value !== null) {
      if (value.hours !== undefined || value.minutes !== undefined
|| value.seconds !== undefined) {
        const hours = value.hours || 0;
        const minutes = value.minutes || 0;
```

```
        const seconds = value.seconds || 0;
        return `${hours}h ${minutes}m ${seconds}s`;
      }
      return JSON.stringify(value);
    }
    return String(value);
  };

  useEffect(() => {
    const fetchTables = async () => {
      try {
        setIsLoading(true);
        const response = await api.getAllTables();

        if (response.data && Array.isArray(response.data)) {
          setTables(response.data);
        } else {
          throw new Error('Invalid tables data format');
        }
      } catch (err) {
        console.error('Failed to load tables:', err);
        setError('Failed to load tables list');
        showNotification('Failed to load tables list', 'error');
      } finally {
        setIsLoading(false);
      }
    };
    fetchTables();
  }, []);

  const loadTableData = async (tableName) => {
    if (!tableName) return;

    try {
      setIsLoading(true);
      setError(null);
      const response = await api.getTableData(tableName);

      if (!response.data) {
        throw new Error('No data received');
      }

      const data = Array.isArray(response.data) ? response.data :
[response.data];
      setTableData(data);
      setSelectedTable(tableName);

      const cols = data.length > 0
        ? Object.keys(data[0])
        : (response.columns || []);
      setColumns(cols);
```

```
      showNotification(`Table ${tableName} loaded successfully`);
    } catch (err) {
      console.error('Failed to load table data:', err);
      const errorMsg = err.response?.data?.message || err.message;
      setError(`Failed to load table: ${errorMsg}`);
        showNotification(`Failed to load table: ${errorMsg}`,
'error');
      setTableData([]);
      setColumns([]);
    } finally {
      setIsLoading(false);
    }
  };

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value
    });
  };

  const handleEditChange = (e) => {
    setEditFormData({
      ...editFormData,
      [e.target.name]: e.target.value
    });
  };

  const handleCreateTable = async () => {
    if (!formData.tableName || !formData.columns) {
       showNotification('Table name and columns are required',
'error');
      return;
    }

    try {
      setIsLoading(true);
     await api.createTable(formData.tableName, formData.columns);
         showNotification(`Table ${formData.tableName} created
successfully`);
      setTables([...tables, formData.tableName]);
      loadTableData(formData.tableName);
    } catch (err) {
      console.error('Failed to create table:', err);
              showNotification(`Failed to create table:
${err.response?.data?.message || err.message}`, 'error');
    } finally {
      setIsLoading(false);
    }
  };

  const handleUpdateTable = async () => {
```

```
        if (!selectedTable || !formData.columnName ||
!formData.newValue || !formData.condition) {
       showNotification('All fields are required for update',
'error');
      return;
    }

    try {
      setIsLoading(true);
      await api.updateTable(
        selectedTable,
        formData.columnName,
        formData.newValue,
        formData.condition
      );
      showNotification('Table updated successfully');
      loadTableData(selectedTable);
    } catch (err) {
      console.error('Failed to update table:', err);
               showNotification(`Failed to update table:
${err.response?.data?.message || err.message}`, 'error');
    } finally {
      setIsLoading(false);
    }
  };

  const handleInsertRow = async () => {
    if (!selectedTable || !formData.rowData) {
       showNotification('Table must be selected and row data
provided', 'error');
      return;
    }

    try {
      setIsLoading(true);
      const data = JSON.parse(formData.rowData);
      await api.insertRow(selectedTable, data);
      showNotification('Row inserted successfully');
      loadTableData(selectedTable);
    } catch (err) {
      console.error('Failed to insert row:', err);
                showNotification(`Failed to insert row:
${err.response?.data?.message || err.message}`, 'error');
    } finally {
      setIsLoading(false);
    }
  };

  const handleDropTable = async () => {
    if (!selectedTable) {
      showNotification('No table selected', 'error');
      return;
```

```
    }

    try {
      setIsLoading(true);
      await api.dropTable(selectedTable);
            showNotification(`Table ${selectedTable} dropped
successfully`);
      setTables(tables.filter(table => table !== selectedTable));
      setSelectedTable('');
      setTableData([]);
      setColumns([]);
    } catch (err) {
      console.error('Failed to drop table:', err);
              showNotification(`Failed to drop table:
${err.response?.data?.message || err.message}`, 'error');
    } finally {
      setIsLoading(false);
    }
  };

  const handleAddColumn = async () => {
        if (!selectedTable || !formData.columnName ||
!formData.columnType) {
        showNotification('Column name and type are required',
'error');
      return;
    }

    try {
      setIsLoading(true);
        await api.addColumn(selectedTable, formData.columnName,
formData.columnType);
      showNotification('Column added successfully');
      loadTableData(selectedTable);
    } catch (err) {
      console.error('Failed to add column:', err);
              showNotification(`Failed to add column:
${err.response?.data?.message || err.message}`, 'error');
    } finally {
      setIsLoading(false);
    }
  };

  const handleDropColumn = async () => {
    if (!selectedTable || !formData.columnName) {
        showNotification('Table and column name are required',
'error');
      return;
    }

    try {
      setIsLoading(true);
```

```
      await api.dropColumn(selectedTable, formData.columnName);
      showNotification('Column dropped successfully');
      loadTableData(selectedTable);
    } catch (err) {
      console.error('Failed to drop column:', err);
                showNotification(`Failed    to    drop    column:
${err.response?.data?.message || err.message}`, 'error');
    } finally {
      setIsLoading(false);
    }
  };

  const startEditing = (row) => {
    setEditingRow(row);
    const editableFields = {...row};
    delete editableFields.id;
    setEditFormData(editableFields);
  };

  const saveEditedRow = async () => {
    try {
      const condition = `id = ${editingRow.id}`;

      for (const [column, value] of Object.entries(editFormData))
{
        if (editingRow[column] !== value) {
          await api.updateTable(
            selectedTable,
            column,
            typeof value === 'string' ? `'${value}'` : value,
            condition
          );
        }
      }

      showNotification('Row updated successfully');
      setEditingRow(null);
      loadTableData(selectedTable);
    } catch (err) {
      console.error('Failed to update row:', err);
                showNotification(`Failed    to    update    row:
${err.response?.data?.message || err.message}`, 'error');
    }
  };

  const cancelEditing = () => {
    setEditingRow(null);
  };

  return (
    <div className="main-container">
      <div className="panel">
```

```
        <div className="panel-header">
          <h2>Table Operations</h2>
          <div className="form-row tabs">
            <button
              onClick={() => setActiveTab('browse')}
                  className={`button ${activeTab === 'browse' ?
'button-primary' : ''}`}
            >
              Browse
            </button>
            <button
              onClick={() => setActiveTab('create')}
                  className={`button ${activeTab === 'create' ?
'button-primary' : ''}`}
            >
              Create
            </button>
            <button
              onClick={() => setActiveTab('modify')}
                  className={`button ${activeTab === 'modify' ?
'button-primary' : ''}`}
            >
              Modify
            </button>
          </div>
        </div>

        <div className="panel-content">
          {isLoading && <div className="loading">Loading...</div>}
          {error && <div className="error-message">{error}</div>}

          {activeTab === 'browse' && (
            <div className="form-group">
              <label>Select Table:</label>
              <select
                value={selectedTable}
                onChange={(e) => loadTableData(e.target.value)}
                className="form-select"
                disabled={isLoading}
              >
                <option value="">Select table</option>
                {tables.map(table => (
                                          <option    key={table}
value={table}>{table}</option>
                ))}
              </select>
            </div>
          )}

          {activeTab === 'create' && (
            <div className="form-group">
              <label>Table Name:</label>
```

```
              <input
                type="text"
                name="tableName"
                value={formData.tableName}
                onChange={handleChange}
                disabled={isLoading}
              />

                    <label>Columns (format: "col1 TYPE, col2
TYPE"):</label>
              <textarea
                name="columns"
                value={formData.columns}
                onChange={handleChange}
               placeholder="id INT PRIMARY KEY, name VARCHAR(100)"
                disabled={isLoading}
              />

              <button
                onClick={handleCreateTable}
                className="button button-success"
                disabled={isLoading}
              >
                {isLoading ? 'Creating...' : 'Create Table'}
              </button>
            </div>
          )}

          {activeTab === 'modify' && selectedTable && (
            <>
              <div className="form-group">
                <h3>Modify Table: {selectedTable}</h3>

                <div className="form-row">
                  <div className="form-control">
                    <label>Column Name:</label>
                    <input
                      type="text"
                      name="columnName"
                      value={formData.columnName}
                      onChange={handleChange}
                      disabled={isLoading}
                    />
                  </div>

                  <div className="form-control">
                    <label>Column Type:</label>
                    <select
                      name="columnType"
                      value={formData.columnType}
                      onChange={handleChange}
                      disabled={isLoading}
```

```
                    >
                                                        <option
value="VARCHAR(255)">VARCHAR(255)</option>
                      <option value="INT">INT</option>
                      <option value="FLOAT">FLOAT</option>
                      <option value="DATE">DATE</option>
                      <option value="BOOLEAN">BOOLEAN</option>
                      <option value="INTERVAL">INTERVAL</option>
                    </select>
                  </div>
                </div>

                <div className="form-row">
                  <button
                    onClick={handleAddColumn}
                    className="button button-primary"
                    disabled={isLoading}
                  >
                    {isLoading ? 'Adding...' : 'Add Column'}
                  </button>
                  <button
                    onClick={handleDropColumn}
                    className="button button-danger"
                    disabled={isLoading}
                  >
                    {isLoading ? 'Dropping...' : 'Drop Column'}
                  </button>
                </div>
              </div>

              <div className="form-group">
                <h3>Table Data Operations</h3>

                <div className="form-row">
                  <div className="form-control">
                    <label>Column to Update:</label>
                    <input
                      type="text"
                      name="columnName"
                      value={formData.columnName}
                      onChange={handleChange}
                      disabled={isLoading}
                    />
                  </div>

                  <div className="form-control">
                    <label>New Value:</label>
                    <input
                      type="text"
                      name="newValue"
                      value={formData.newValue}
                      onChange={handleChange}
```

```
                disabled={isLoading}
              />
            </div>
          </div>

          <div className="form-control">
            <label>WHERE Condition:</label>
            <input
              type="text"
              name="condition"
              value={formData.condition}
              onChange={handleChange}
              placeholder="id = 1"
              disabled={isLoading}
            />
          </div>

          <button
            onClick={handleUpdateTable}
            className="button button-primary"
            disabled={isLoading}
          >
            {isLoading ? 'Updating...' : 'Update Data'}
          </button>

          <div className="form-group">
            <label>Insert Row (JSON):</label>
            <textarea
              name="rowData"
              value={formData.rowData}
              onChange={handleChange}
              placeholder='{"column1": "value1", "column2":
"value2"}'
              disabled={isLoading}
            />

            <button
              onClick={handleInsertRow}
              className="button button-primary"
              disabled={isLoading}
            >
              {isLoading ? 'Inserting...' : 'Insert Row'}
            </button>
          </div>

          <div className="form-group">
            <button
              onClick={handleDropTable}
              className="button button-danger"
              disabled={isLoading}
            >
              {isLoading ? 'Dropping...' : 'Drop Table'}
```

```
                  </button>
                </div>
              </div>
            </>
          )}
        </div>
      </div>

      <div className="panel results-panel">
        <div className="panel-header">
          <h2>Table Data: {selectedTable || 'none'}</h2>
        </div>

        <div className="panel-content">
          {isLoading ? (
            <div className="loading">Loading table data...</div>
          ) : error ? (
            <div className="error-message">{error}</div>
          ) : (
            <div className="table-container">
              <table>
                <thead>
                  <tr>
                    {columns.map((key) => (
                      <th key={key}>{key}</th>
                    ))}
                    {columns.length > 0 && <th>Actions</th>}
                  </tr>
                </thead>
                <tbody>
                  {tableData.length > 0 ? (
                    tableData.map((row, index) => (
                      <tr key={index}>
                        {columns.map((key) => (
                          <td key={key}>
                            {editingRow && editingRow.id === row.id
&& key !== 'id' ? (
                              <input
                                type="text"
                                name={key}
                                value={editFormData[key] || ''}
                                onChange={handleEditChange}
                              />
                            ) : (
                                key === 'duration' || key ===
'warranty_period'
                                ? formatInterval(row[key])
                                : String(row[key])
                            )}
                          </td>
                        ))}
                        <td>
```

```
                            {editingRow && editingRow.id === row.id
? (
                              <>
                                <button
                                  onClick={saveEditedRow}
                                   className="button button-success
small"
                                >
                                  Save
                                </button>
                                <button
                                  onClick={cancelEditing}
                                    className="button button-danger
small"
                                >
                                  Cancel
                                </button>
                              </>
                            ) : (
                              <button
                                onClick={() => startEditing(row)}
                                   className="button button-primary
small"
                              >
                                Edit
                              </button>
                            )}
                          </td>
                        </tr>
                      ))
                    ) : (
                      <tr>
                                 <td  colSpan={columns.length  +  1}
className="no-data">
                            {selectedTable ? 'No data available' :
'Select a table to view data'}
                        </td>
                      </tr>
                    )}
                  </tbody>
                </table>
              </div>
            )}
          </div>
        </div>
      </div>
    );
};

export default TableManagement;
```

api.js:

```
import axios from 'axios';

const api = axios.create({
  baseURL: 'http://localhost:3000',
  headers: {
    'Content-Type': 'application/json'
  }
});

const sanitizeFilename = (filename) => {
  return filename.replace(/\./g, '') + '.sql';
};

const apiService = {



  executePredefinedQuery(endpoint) {
    return api.get(`/queries/${endpoint}`);
  },


  executeCustomQuery(query) {
     return api.post('/queries/executeCustom', null, { params: {
query } });
  },

  saveCustomQuery(queryName, query) {
      return api.post('/queries/saveCustom', null, { params: {
queryName, query } });
  },

  getSavedQuery(queryName) {
    return api.get('/queries/getCustom', { params: { queryName }
});
  },




  getTableData(tableName) {
    return this.executeCustomQuery(`SELECT * FROM ${tableName}`);
  },

  getAllTables() {
    return api.get('/tables/getAllTables');
  },


  getTablesList() {
    return this.executeCustomQuery('SHOW TABLES');
```

```
  },


  createTable(tableName, columns) {
    return api.post('/tables/create', null, { params: { tableName,
columns } });
  },

  updateTable(tableName, columnName, newValue, condition) {
    return api.put('/tables/update', null, {
      params: { tableName, columnName, newValue, condition }
    });
  },

  insertRow(tableName, data) {
    return api.post('/tables/insert', data, {
      params: { tableName }
    });
  },

  dropTable(tableName) {
    return api.delete('/tables/drop', { params: { tableName } });
  },

  addColumn(tableName, columnName, columnType) {
    return api.post('/tables/addColumn', null, {
      params: { tableName, columnName, columnType }
    });
  },

  dropColumn(tableName, columnName) {
    return api.delete('/tables/dropColumn', {
      params: { tableName, columnName }
    });
  },



  exportToExcel(tableName, filename) {
    return api.get('/tables/export', {
      params: { tableName, filename: sanitizeFilename(filename) }
    });
  },

  backupTable(tableName, filename) {
    return api.get('/tables/backup', {
      params: { tableName, filename: sanitizeFilename(filename) }
    });
  },

  fullBackup(filename) {
    return api.get('/tables/fullBackup', {
```

```
      params: { filename: sanitizeFilename(filename) }
    });
  },

  loadBackup(file) {
    const formData = new FormData();
    formData.append('file', file);
    return api.post('/tables/loadBackup', formData, {
      headers: {
        'Content-Type': 'multipart/form-data'
      }
    });
  },




  getClientsWithLastNameIva() {
                                                          return
this.executePredefinedQuery('clients_with_last_name_Iva');
  },

  getClientsWithCars() {
    return this.executePredefinedQuery('clients_with_cars');
  },

  getYoungClientsCount() {
    return this.executePredefinedQuery('young_clients_count');
  },


  getCarsAudiAfter2015() {
    return this.executePredefinedQuery('cars_audi_after_2015');
  },

  getCarCountByBrand() {
    return this.executePredefinedQuery('car_count_by_brand');
  },


  getStosWorkingHoursAndAddress() {
                                                          return
this.executePredefinedQuery('stos_working_hours_and_address');
  },

  getTop3StosByEmployeeCount() {
                                                          return
this.executePredefinedQuery('top3_stos_by_employee_count');
  },


  getEmployeesHighSalaryStreetL() {
```

```
                                                                    return
this.executePredefinedQuery('employees_high_salary_street_L');
  },

  getEmployeesLowSalaryHighOrderAmount() {
                                                                    return
this.executePredefinedQuery('employees_low_salary_high_order_amo
unt');
  },

  getEmployeesWithAboveAverageSalary() {
                                                                    return
this.executePredefinedQuery('employees_with_above_average_salary
');
  },


  getCompletedOrdersWithStreetAddress() {
                                                                    return
this.executePredefinedQuery('completed_orders_with_street_addres
s');
  },

  getIncompleteOrders() {
    return this.executePredefinedQuery('incomplete_orders');
  },

  getMostExpensiveOrder() {
    return this.executePredefinedQuery('most_expensive_order');
  },

  getTotalCompletedOrdersSum() {
                                                                    return
this.executePredefinedQuery('total_completed_orders_sum');
  },


  getSparePartsInExpensiveOrders() {
                                                                    return
this.executePredefinedQuery('spare_parts_in_expensive_orders');
  },

  getSparePartsWithLowStock() {
                                                                    return
this.executePredefinedQuery('spare_parts_with_low_stock');
  },

  getTotalInventoryValue() {
    return this.executePredefinedQuery('total_inventory_value');
  },
```

```
  getAverageServiceCostByCategory() {
                                                          return
this.executePredefinedQuery('average_service_cost_by_category');
  },

  getTop5Services() {
    return this.executePredefinedQuery('top5_services');
  },


  getOrdersWithBoschParts() {
                                                          return
this.executePredefinedQuery('orders_with_bosch_parts');
  },

  getOrdersWithExpensiveServices() {
                                                          return
this.executePredefinedQuery('orders_with_expensive_services');
  },

  getStosHighSalaryEmployeesHighOrderAmount() {
                                                          return
this.executePredefinedQuery('stos_high_salary_employees_high_ord
er_amount');
  },

  getEmployeesInStosWithHighOrderAmount() {
                                                          return
this.executePredefinedQuery('employees_in_stos_with_high_order_a
mount');
  },

  getStosWithHighSalaryEmployeesAndHighOrderAmount() {
                                                          return
this.executePredefinedQuery('stos_with_high_salary_employees_and
_high_order_amount');
  },

  getStosWithEmployeesHighSalaryAndHighOrderAmount() {
                                                          return
this.executePredefinedQuery('stos_with_employees_high_salary_and
_high_order_amount');
  }
};

export default apiService;

App.js:
import React, { Suspense, lazy, useState } from 'react';
import './App.css';

const QueryView = lazy(() => import('./components/QueryView'));
```

```
const TableManagement = lazy(() => import('./components/TableManagement'));
const BackupView = lazy(() => import('./components/BackupView'));

function App() {
  const [activeView, setActiveView] = useState('queries');
  const [notification, setNotification] = useState(null);

  const showNotification = (message, type = 'success') => {
    setNotification({ message, type });
    setTimeout(() => setNotification(null), 5000);
  };

  const views = {
    queries: {
      name: 'Queries',
      component: <QueryView showNotification={showNotification} />
    },
    tables: {
      name: 'Tables',
      component: <TableManagement showNotification={showNotification} />
    },
    backup: {
      name: 'Backup',
      component: <BackupView showNotification={showNotification} />
    }
  };

  return (
    <div className="App">
      {notification && (
        <div className={`notification ${notification.type}`}>
          {notification.message}
        </div>
      )}

      <div className="header">
        <h1>Auto Service Database Manager</h1>
        <div className="nav-buttons">
          {Object.entries(views).map(([key, view]) => (
            <button
              key={key}
              onClick={() => setActiveView(key)}
              className={`button ${activeView === key ? 'active' : ''}`}
            >
              {view.name}
            </button>
          ))}
        </div>
```

```
      </div>

      <div className="main-container">
           <Suspense  fallback={<div  className="loading">Loading
module...</div>}>
          {views[activeView].component}
        </Suspense>
      </div>
    </div>
  );
}

export default App;
```