**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**ІМЕНІ ТАРАСА ШЕВЧЕНКА**

Факультет комп’ютерних наук та кібернетики

Кафедра теорії та технології програмування

**Звіт**

до лабораторної роботи на тему

**Фрагментарна реалізація системи управління табличними базами даних**

Виконала студентка 4-го курсу

Сушко Діана Валентинівна

**Київ – 2024**

**Зміст**

Table of Contents

[Опис завдання 3](#_Toc177849640)

[Використані технології 4](#_Toc177849641)

[Етап 1. UML специфікація 5](#_Toc177849642)

[Етап 2. Розробка локальної (десктоп) версії СУБД 11](#_Toc177849643)

[Етап 3. Розробка клієнт-серверної (для локальних мереж) або монолітної (для веб) архітектури 22](#_Toc177849644)

[Етап 4. Використання веб-орієнтованої версії з використанням технологій на вибір (REST) 30](#_Toc177849645)

[Етап 5. Порівняльний аналіз розробки однієї і тієї ж вашої системи з використанням різних технологій та підходів 36](#_Toc177849646)

# **Опис завдання**

Основні вимоги щодо структури бази:  
∙ кількість таблиць принципово не обмежена (реляції між таблицями не враховувати);  
∙ кількість полів та кількість записів у кожній таблиці також принципово не обмежені.

У кожній роботі для всіх варіантів для полів у таблицях треба забезпечити підтримку таких типів:  
∙ integer;  
∙ real;  
∙ char;  
∙ string.

Також у кожній роботі треба реалізувати функціональну підтримку для:  
∙ створення бази;  
∙ створення (із валідацією даних) та знищення таблиці з бази;  
∙ створення, валідації, перегляду та редагування рядків таблиці;  
∙ збереження табличної бази на диску та зчитування її з диску.

Варіант: Money, MoneyInvl, Combine Tables

# **Використані технології**

* IntelliJ IDEA (IDE) інтегроване середовище розробки під Java платформу.
* Spring Boot фреймворк для розробки веб-сервісів
* SWING/AWT – бібліотеки для GUI застосунків під Java платформу
* Spring MVC – фреймворк для розробки застосунків під MVC-архітектуру
* app diagrams – середовище для створення UML діаграм

# **Етап 1. UML специфікація**

Для створення UML специфікації був використаний ресурс app diagrams для створення коректних діаграм під архітектуру системи.

Розглянемо діаграму активності для створення БД:

Розглянемо діаграму активності для перегляду БД:

A diagram of a flowchart

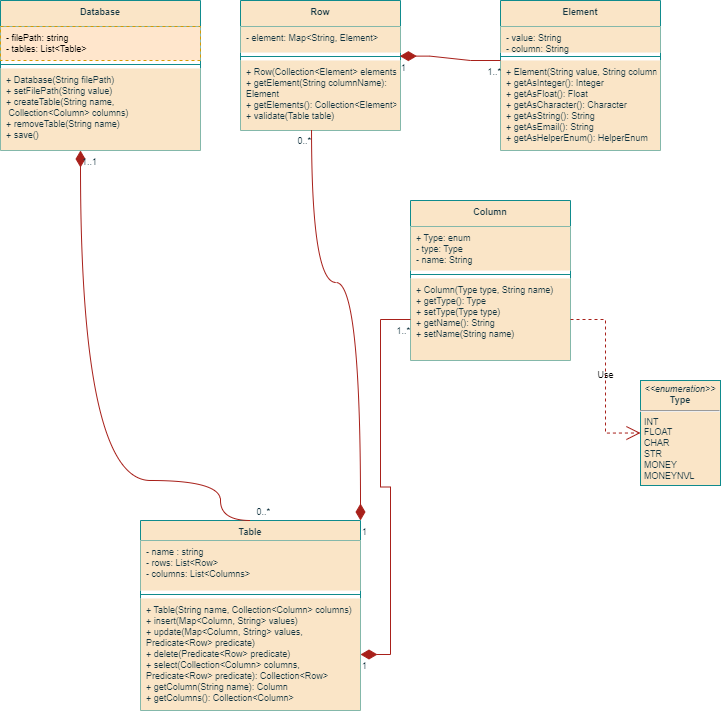
Description automatically generated

Тепер розглянемо діаграму взаємодії до БД:

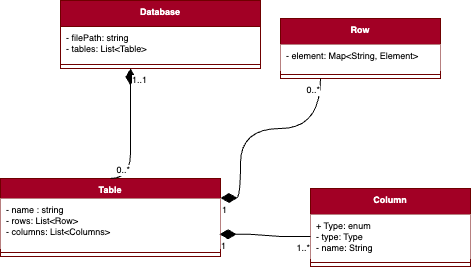
A diagram of a diagram

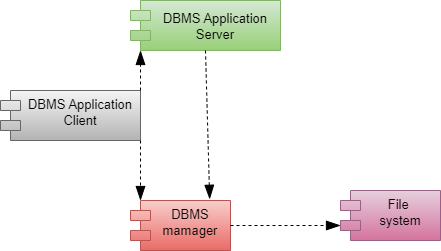
Description automatically generated

Тепер невеличка діаграма класів яка поєднує в собі Database, Table, Column, Type, Row, Element:



За варіантом необхідно зробити об’єднання/різницю таблиць, ось діаграма VOPC яка стосується класів які безпосередньо беруть в цьому участь:



Тепер діаграма компонентів але розподілена:  


Діаграма компонентів але для GUI:

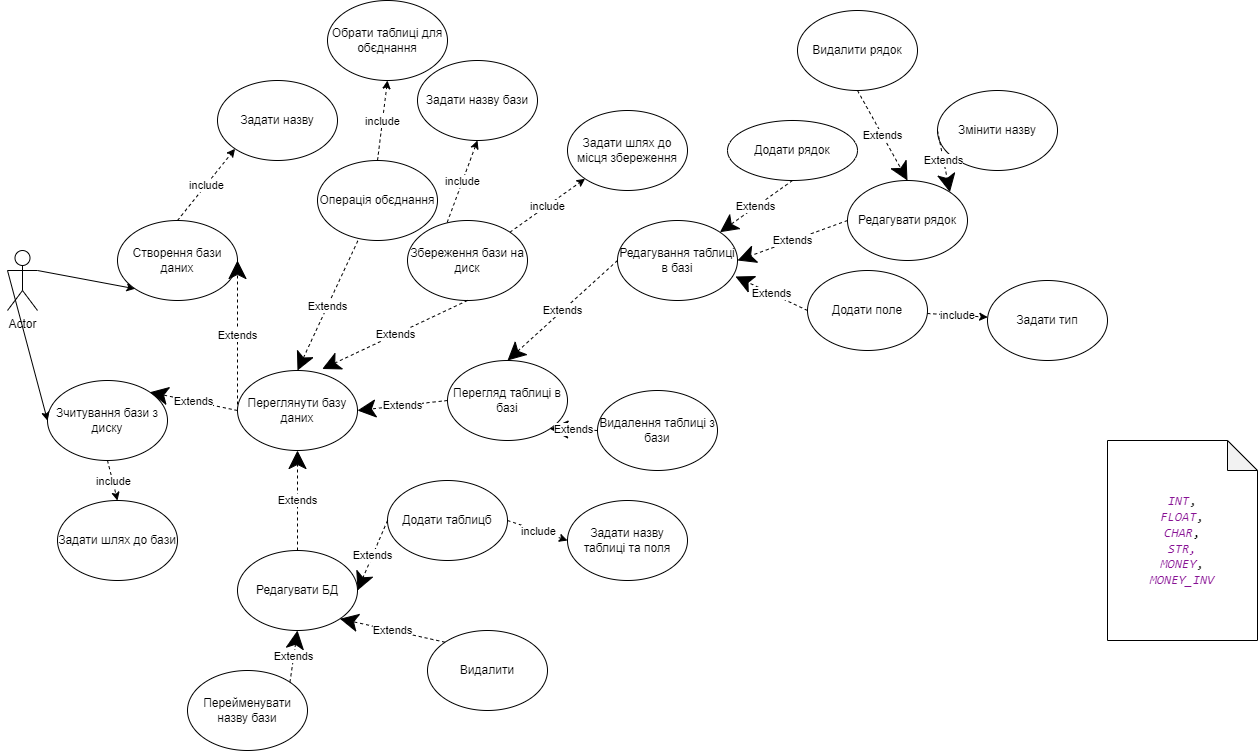


Тепер розглянемо діаграму розгортання розподілену:

A diagram of a computer network

Description automatically generated with medium confidence

Тепер розберемо діаграму прецедентів адже користувач має якось взаємодіяти з системою



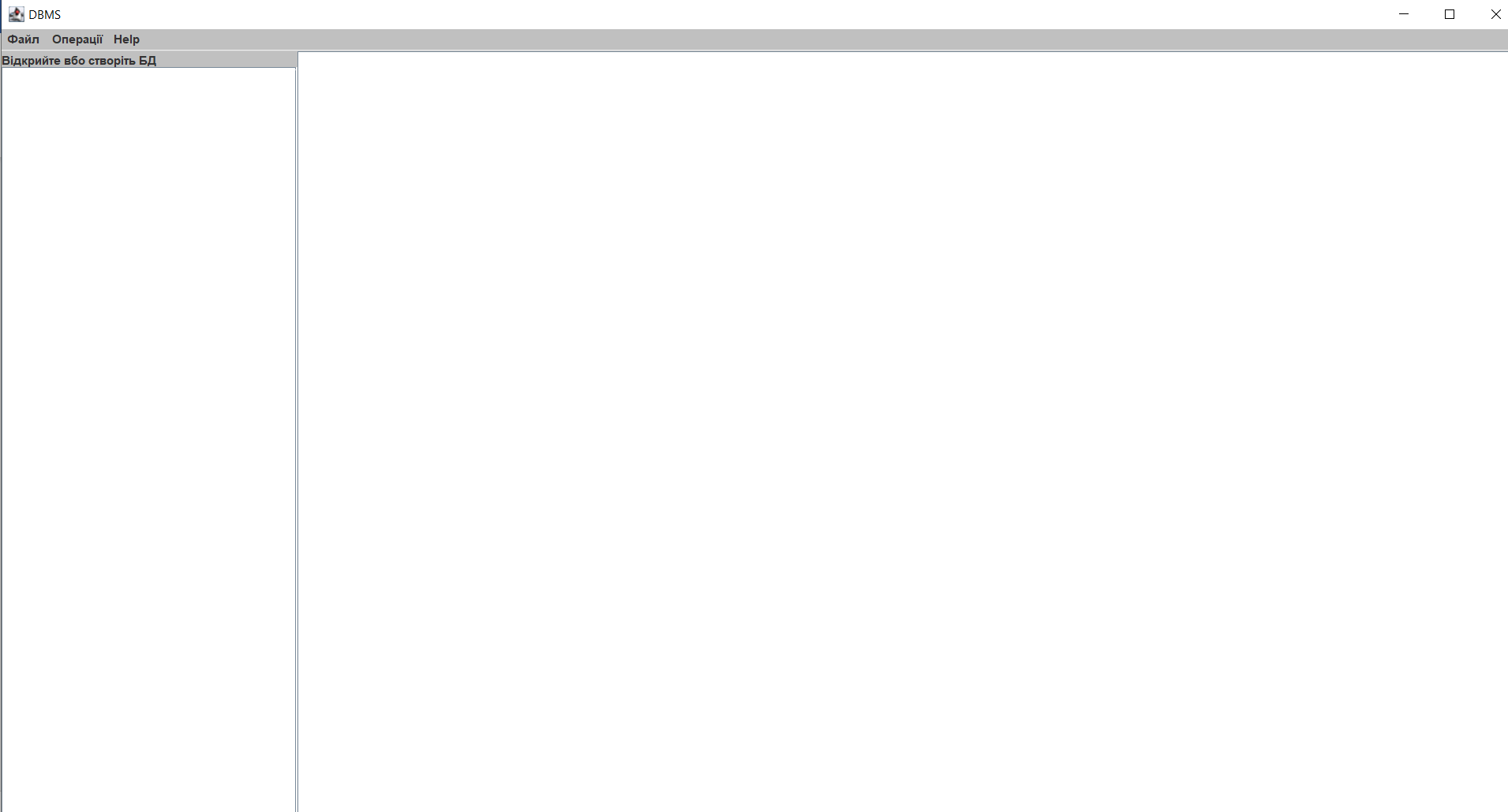
Ну і на останок система має певний автомат станів тому під нього також створимо діаграму:

A diagram of a company

Description automatically generated

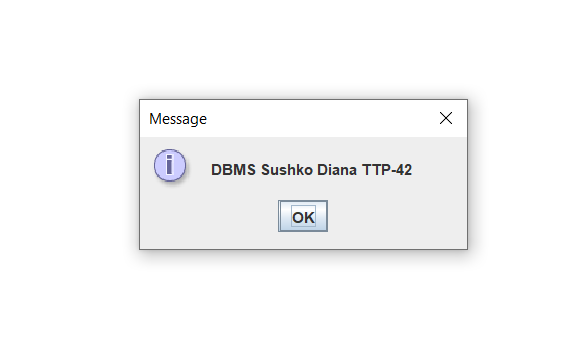
# **Етап 2. Розробка локальної (десктоп) версії СУБД**

Розробка локальної десктопної системи в першу чергу вимагає активне використання GUI бібліотек під платформу Java. Було обрано Swing. Розглянемо основні функції десктопної версії.



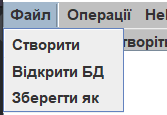
Користувач може як самостійно ввести запит внизу а також може скористатися додатковими кнопками.

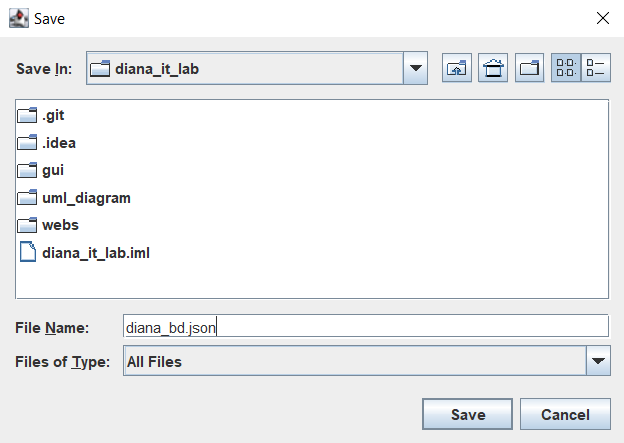
Це виводиться на Help -> Info.



Далі ситуація наступна користувач може як створити базу з нуля,

завантажити існуючу

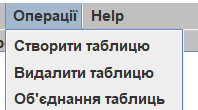


Створімо з нуля до прикладу вибираючи куди і з якою назвою створювати у файлову систему.  


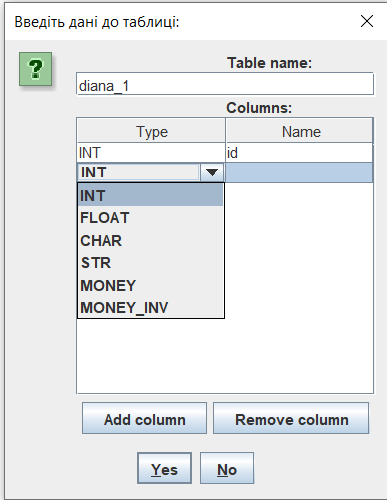
Результат створення



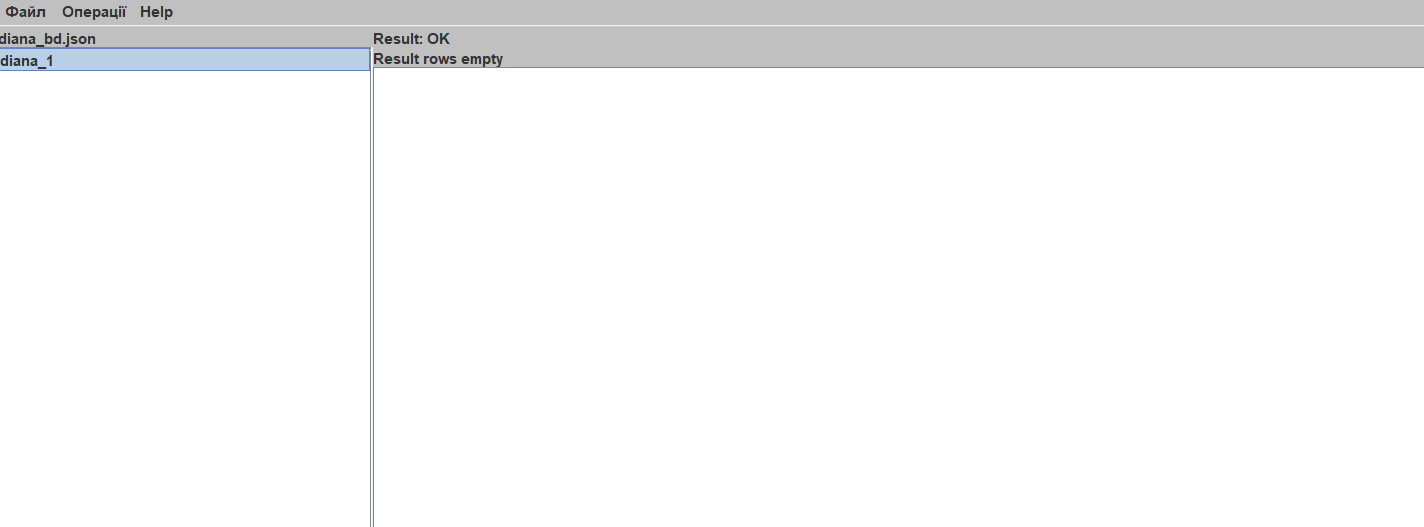
Основні операції з таблицями:



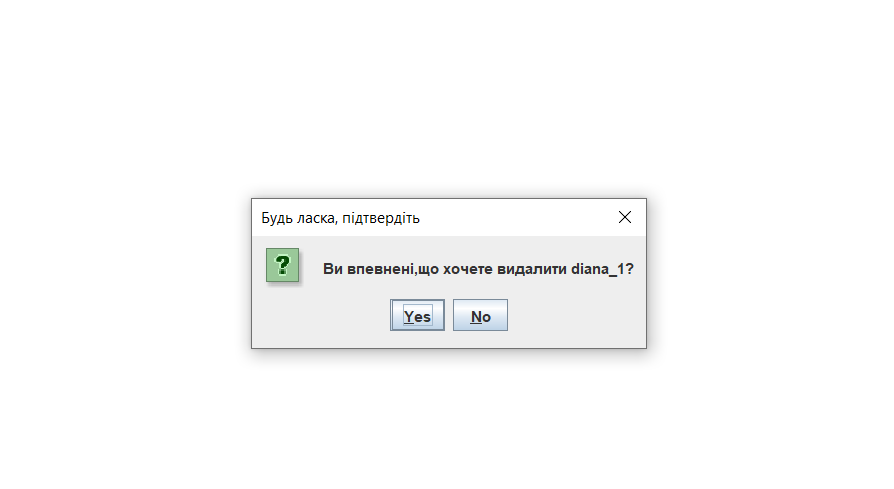
Створімо таблицю



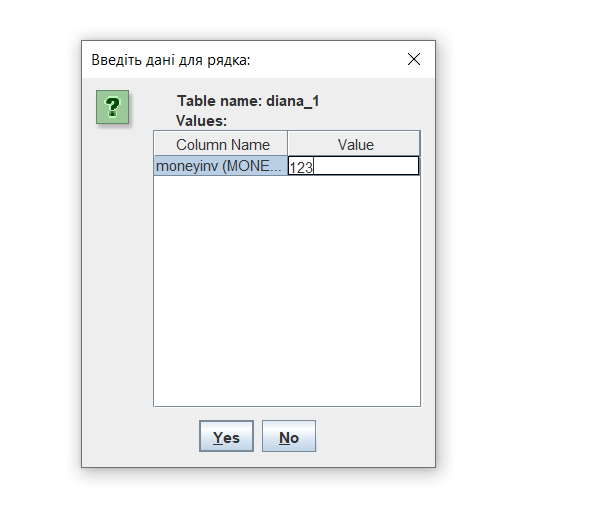
Інтерфейс для створення таблиці.



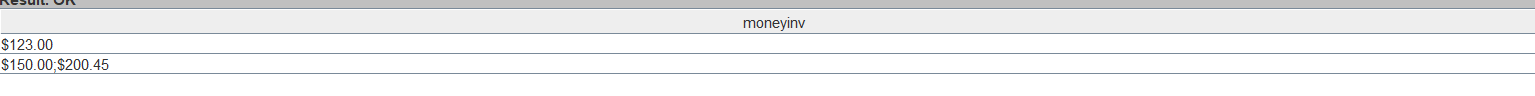
Тепер видалимо її



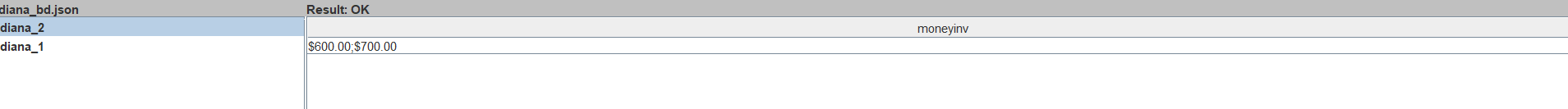
Додаємо рядок в таблицю:

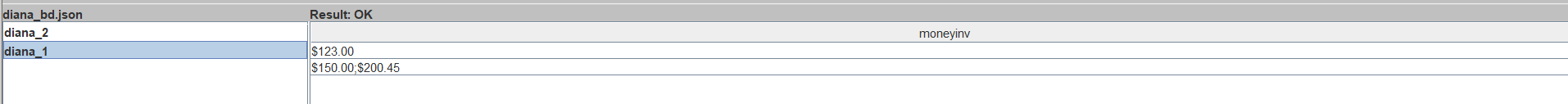


Отримаємо рядок

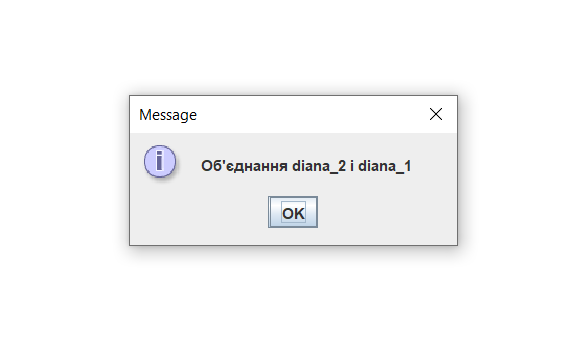


Тепер пробуємо об’єднати таблиці згідно варіанту.





Попереджувальне повідомлення:



Результат об’єднання



Також в рамка Етапу 2 було реалізовано Unit тестуваня основного SQL-подібного парсеру:

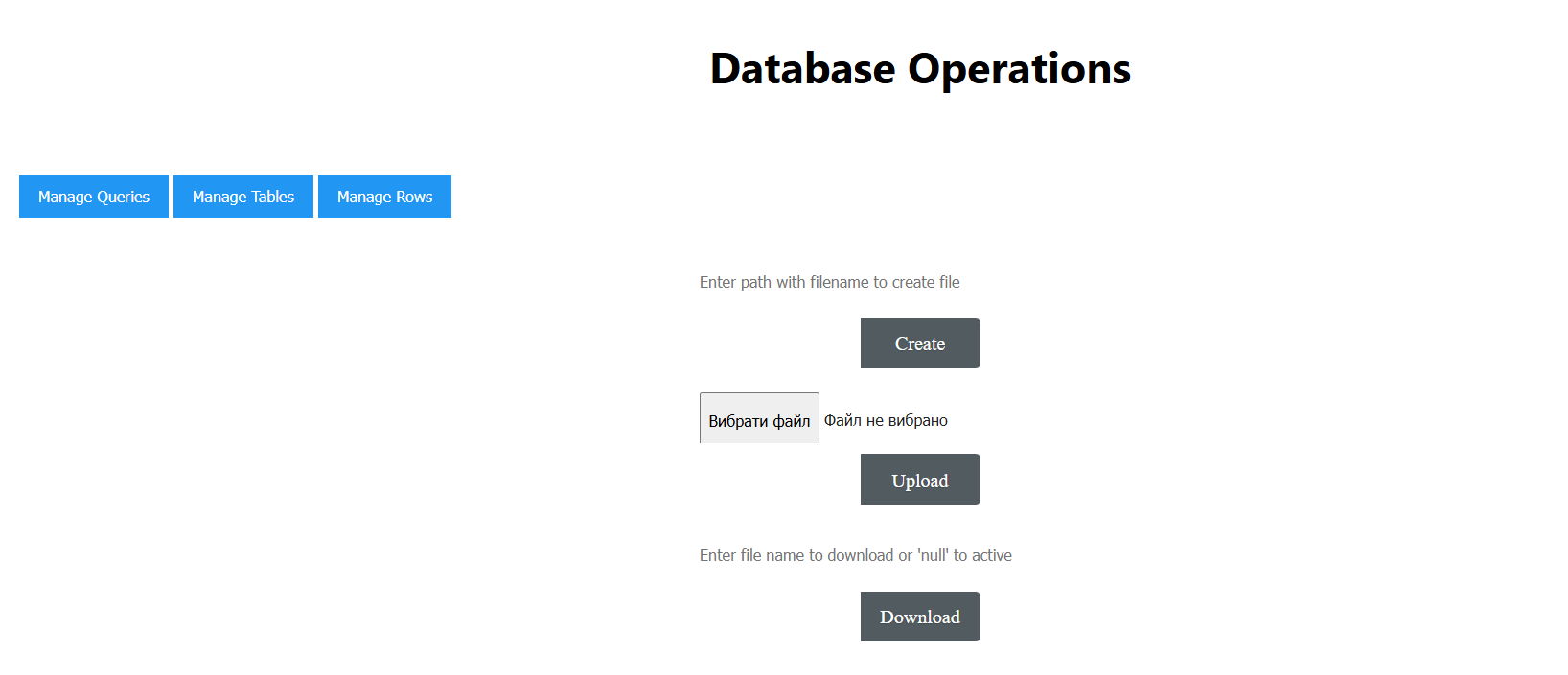
package org.example.gui;  
  
  
import org.example.gui.database.Database;  
import org.example.gui.database.Result;  
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.*assertSame*;  
  
class DatabaseTest {  
  
 @Test  
 void createTable() {  
 Database database = new Database(null);  
 *assertSame*(Result.Status.*OK*, database.query("create table test1 (INT id, STR name)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("create table test2 (INT id, STR name)").getStatus());  
  
 Result result = database.query("list tables");  
 *assertSame*(Result.Status.*OK*, result.getStatus());  
 *assertSame*(2, result.getRows().size());  
 }  
  
 @Test  
 void removeTable() {  
 Database database = new Database(null);  
 *assertSame*(Result.Status.*OK*, database.query("create table test1 (INT id, STR name)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("create table test2 (INT id, STR name)").getStatus());  
  
 Result resultBeforeRemoving = database.query("list tables");  
 *assertSame*(Result.Status.*OK*, resultBeforeRemoving.getStatus());  
 *assertSame*(2, resultBeforeRemoving.getRows().size());  
  
 *assertSame*(Result.Status.*OK*, database.query("remove table test1").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("remove table test2").getStatus());  
  
 Result result = database.query("list tables");  
 *assertSame*(Result.Status.*OK*, result.getStatus());  
 *assertSame*(0, result.getRows().size());  
 }  
  
 @Test  
 void deleteRows() {  
 Database database = new Database(null);  
 *assertSame*(Result.Status.*OK*, database.query("create table test1 (INT id, STR name)").getStatus());  
  
 *assertSame*(Result.Status.*OK*, database.query("insert into test1 (id, name) values(1, KOSTYAAAA697)").getStatus());  
 *assertSame*(1, database.query("select \* from test1").getRows().size());  
  
  
 *assertSame*(Result.Status.*OK*, database.query("delete from test1 where id=1").getStatus());  
 *assertSame*(0, database.query("select \* from test1").getRows().size());  
 }  
  
 @Test  
 void money() {  
 Database database = new Database(null);  
 *assertSame*(Result.Status.*OK*, database.query("create table test2 (MONEY money)").getStatus());  
  
 *assertSame*(Result.Status.*FAIL*, database.query("insert into test2 (money) values(100as000)").getStatus());  
 *assertSame*(Result.Status.*FAIL*, database.query("insert into test2 (money) values(assssa100000)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("insert into test2 (money) values(100000.00)").getStatus());  
 }  
  
 @Test  
 void moneyInv() {  
 Database database = new Database(null);  
 *assertSame*(Result.Status.*OK*, database.query("create table test3 (MONEY\_INV inv)").getStatus());  
  
 *assertSame*(Result.Status.*FAIL*, database.query("insert into test3 (inv) values($aaasas100000.00;$10,000.00)").getStatus());  
 *assertSame*(Result.Status.*FAIL*, database.query("insert into test3 (inv) values(100000.asas00;$asaa10,000.00)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("insert into test3 (inv) values(100000.00;10000.00)").getStatus());  
 }  
  
 @Test  
 void combineOperation() {  
 Database database = new Database(null);  
 *assertSame*(Result.Status.*OK*, database.query("create table white\_cats (INT id, STR name)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("create table black\_cats (INT id, STR name)").getStatus());  
  
 *assertSame*(Result.Status.*OK*, database.query("insert into white\_cats (id, name) values(1, cat1)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("insert into white\_cats (id, name) values(3, cat2)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("insert into white\_cats (id, name) values(4, cat4)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("insert into white\_cats (id, name) values(6, cat8)").getStatus());  
  
 *assertSame*(Result.Status.*OK*, database.query("insert into black\_cats (id, name) values(3, cat2)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("insert into black\_cats (id, name) values(4, cat4)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("insert into black\_cats (id, name) values(6, cat15)").getStatus());  
  
 Result result = database.query("COMBINE white\_cats WITH black\_cats");  
 *assertSame*(Result.Status.*OK*, result.getStatus());  
 *assertSame*(7, result.getRows().size());  
  
 *assertSame*(Result.Status.*OK*, database.query("create table white\_cats\_2 (INT id, STR name, MONEY price)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("create table black\_cats\_2 (INT id, STR name, MONEY\_INV price\_interval)").getStatus());  
  
 result = database.query("COMBINE white\_cats\_2 WITH black\_cats\_2");  
 *assertSame*(Result.Status.*OK*, result.getStatus());  
 *assertSame*(0, result.getRows().size());  
  
 *assertSame*(Result.Status.*OK*, database.query("create table white\_cats\_3 (INT id, STR name, MONEY price)").getStatus());  
 *assertSame*(Result.Status.*OK*, database.query("create table black\_cats\_3 (INT id, STR name)").getStatus());  
  
 result = database.query("COMBINE white\_cats\_3 WITH black\_cats\_3");  
 *assertSame*(Result.Status.*OK*, result.getStatus());  
 *assertSame*(0, result.getRows().size());  
 }  
}

}

# 

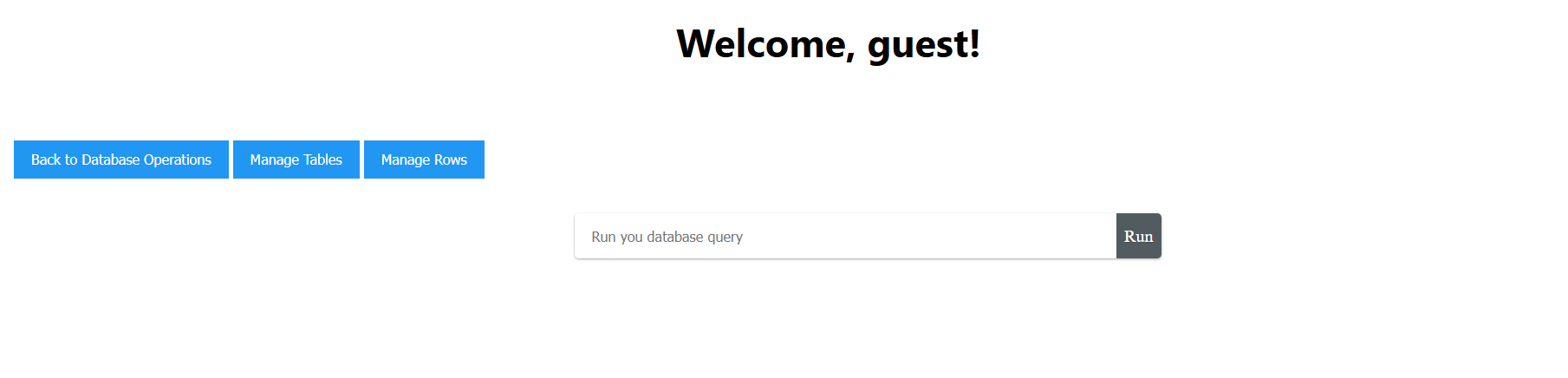
# **Етап 3. Розробка клієнт-серверної (для локальних мереж) або монолітної (для веб) архітектури**

Реалізація була зроблена за допомогою Spring MVC фреймворку та Thymeleaf для забезпечення монолітної веб архітектури. SQL парсер лишається той самий як і на етапі 2 але інтерфейс доступу до функціоналу це тепер веб MVC а не GUI застосунок.



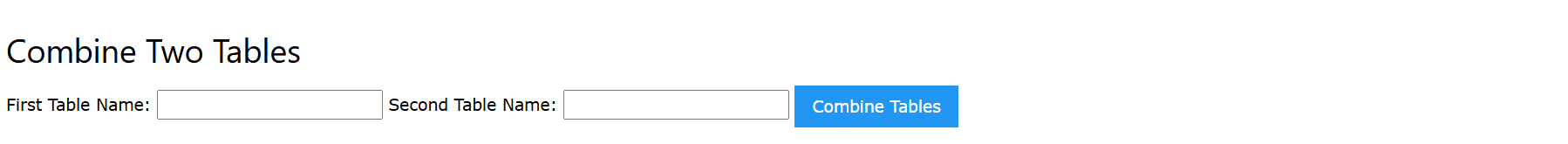
Даний інтерфейс як дозволяє створити з нуля, завантажити вже існуючу так і завантажити файл активної наразі БД.

Також дозволяється використання запитів до бази безпосередньо за допомогою SQL запитів див інтерфейс приклад:

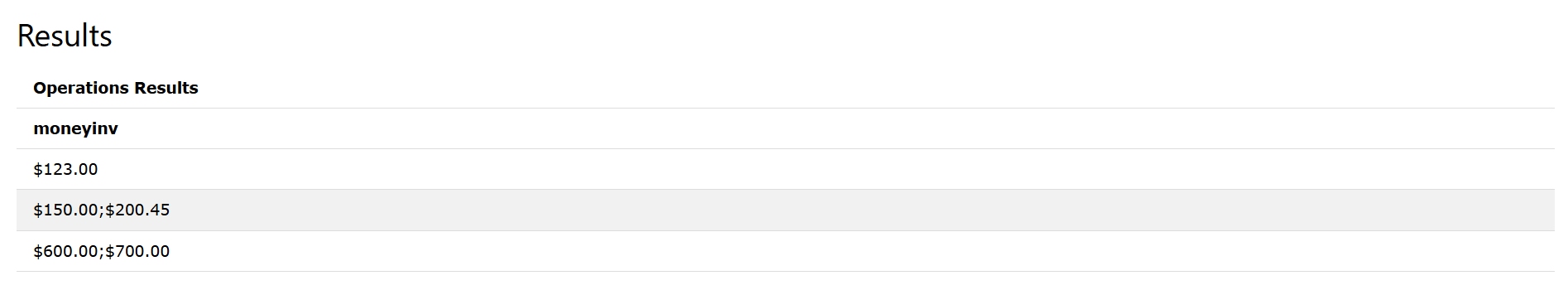


Також надається доступ до управління Таблиць

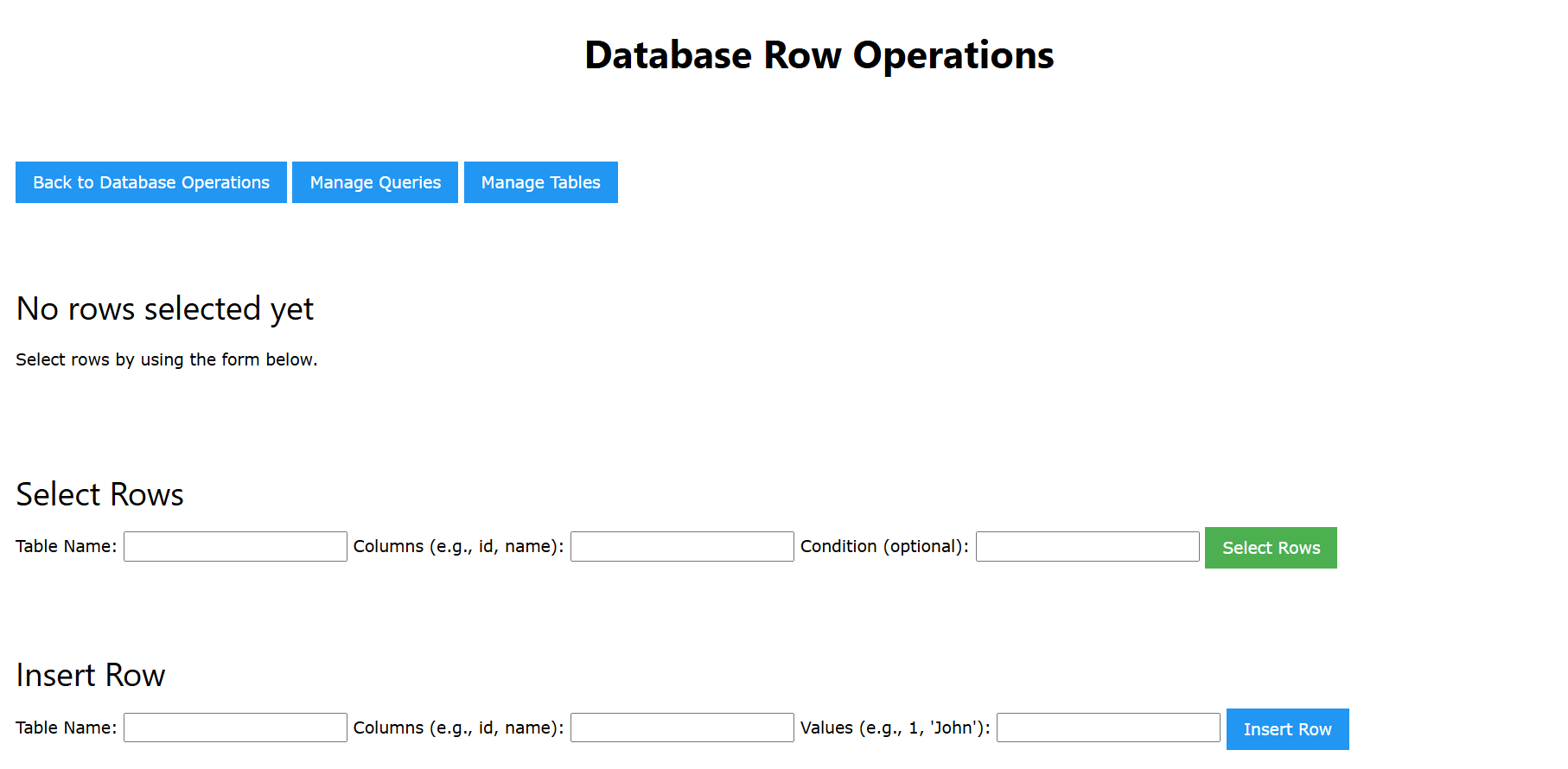


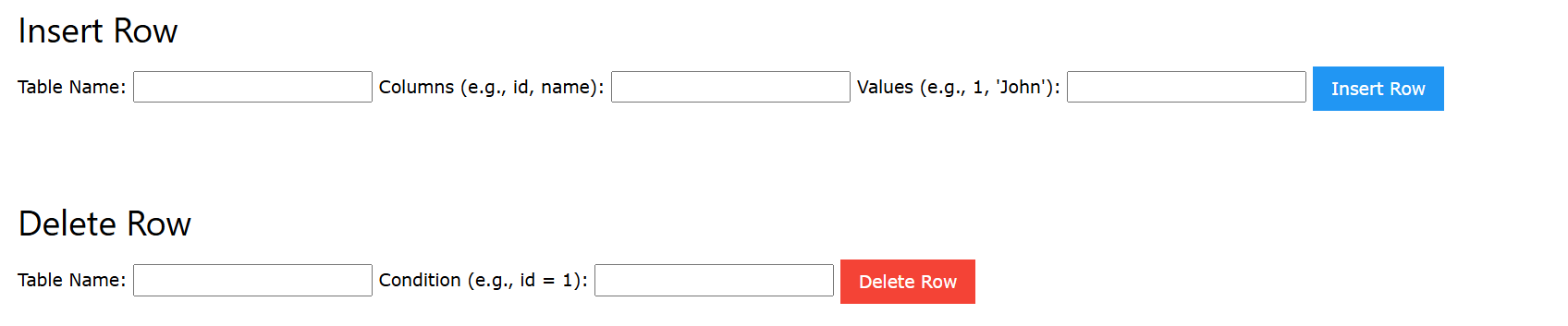


Розглянемо операцію combine:

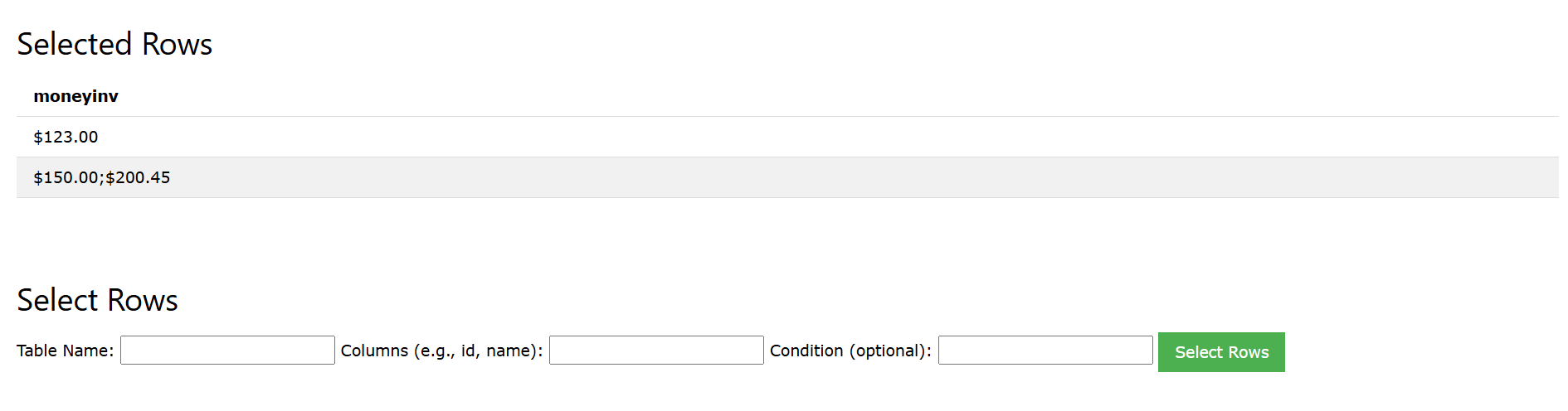


Окрім того наявна сторінка для управління рядками:





Наведемо використання



Введення нового рядку



Розглянемо повну структуру БД у файлі:

{  
 "filePath": "D:\\java\\diana\_it\_lab\\diana\_bd.json",  
 "tables": {  
 "diana\_2": {  
 "name": "diana\_2",  
 "rows": [  
 {  
 "elements": {  
 "moneyinv": {  
 "value": "$600.00;$700.00",  
 "column": "moneyinv"  
 }  
 }  
 }  
 ],  
 "columns": {  
 "moneyinv": {  
 "type": "MONEY\_INV",  
 "name": "moneyinv",  
 "nullAllowed": true  
 }  
 }  
 },  
 "diana\_1": {  
 "name": "diana\_1",  
 "rows": [  
 {  
 "elements": {  
 "moneyinv": {  
 "value": "$123.00",  
 "column": "moneyinv"  
 }  
 }  
 },  
 {  
 "elements": {  
 "moneyinv": {  
 "value": "$150.00;$200.45",  
 "column": "moneyinv"  
 }  
 }  
 },  
 {  
 "elements": {  
 "moneyinv": {  
 "value": "$123.00;$1,000.00",  
 "column": "moneyinv"  
 }  
 }  
 }  
 ],  
 "columns": {  
 "moneyinv": {  
 "type": "MONEY\_INV",  
 "name": "moneyinv",  
 "nullAllowed": true  
 }  
 }  
 }  
 }  
}

# 

# **Етап 4. Використання веб-орієнтованої версії з використанням технологій на вибір (REST)**

Для покращення візуалізації всіх REST API ендпоїнтів було використано SWAGGER UI а також відповідно було перевикористано весь функціонал взаємодії з БД як на етапі 2, 3.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

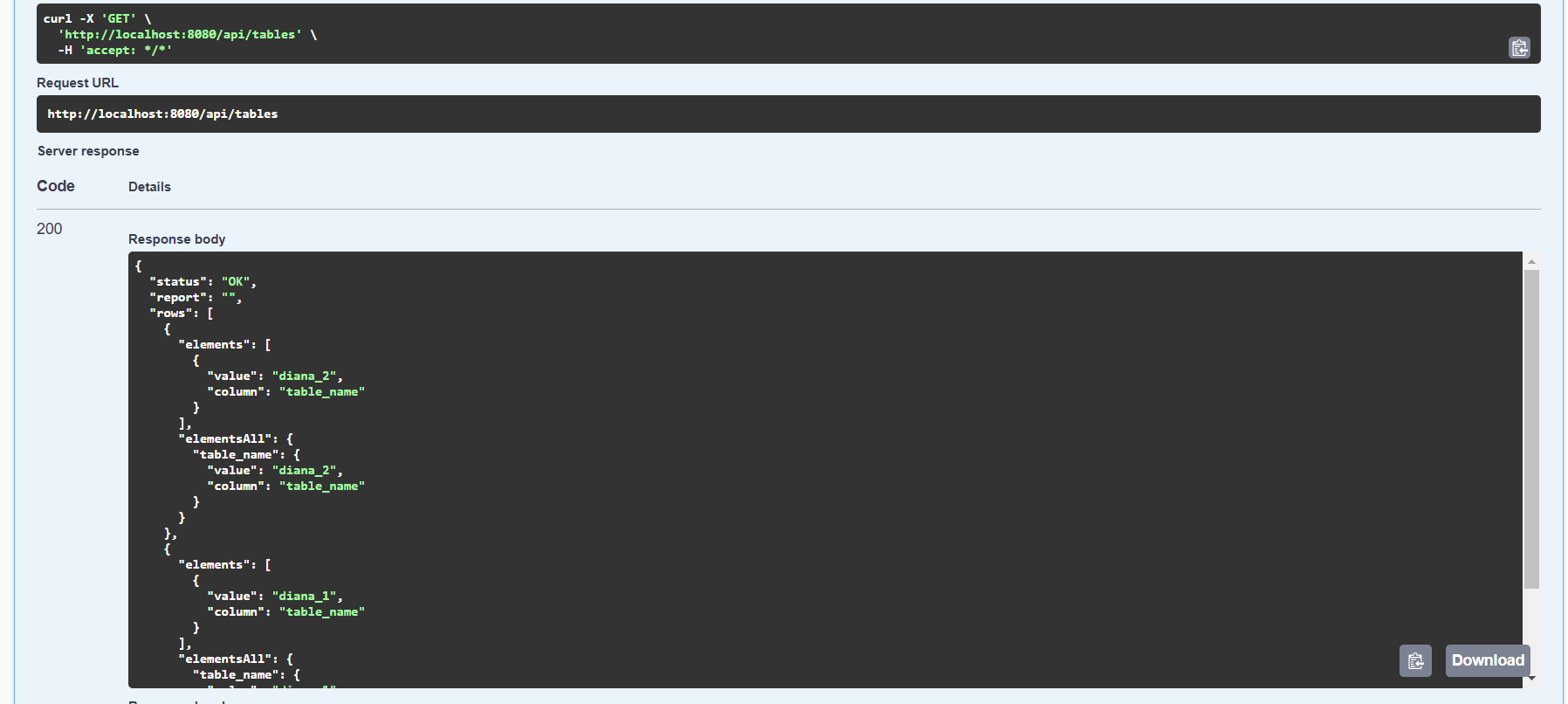
Переглянемо основні запити:

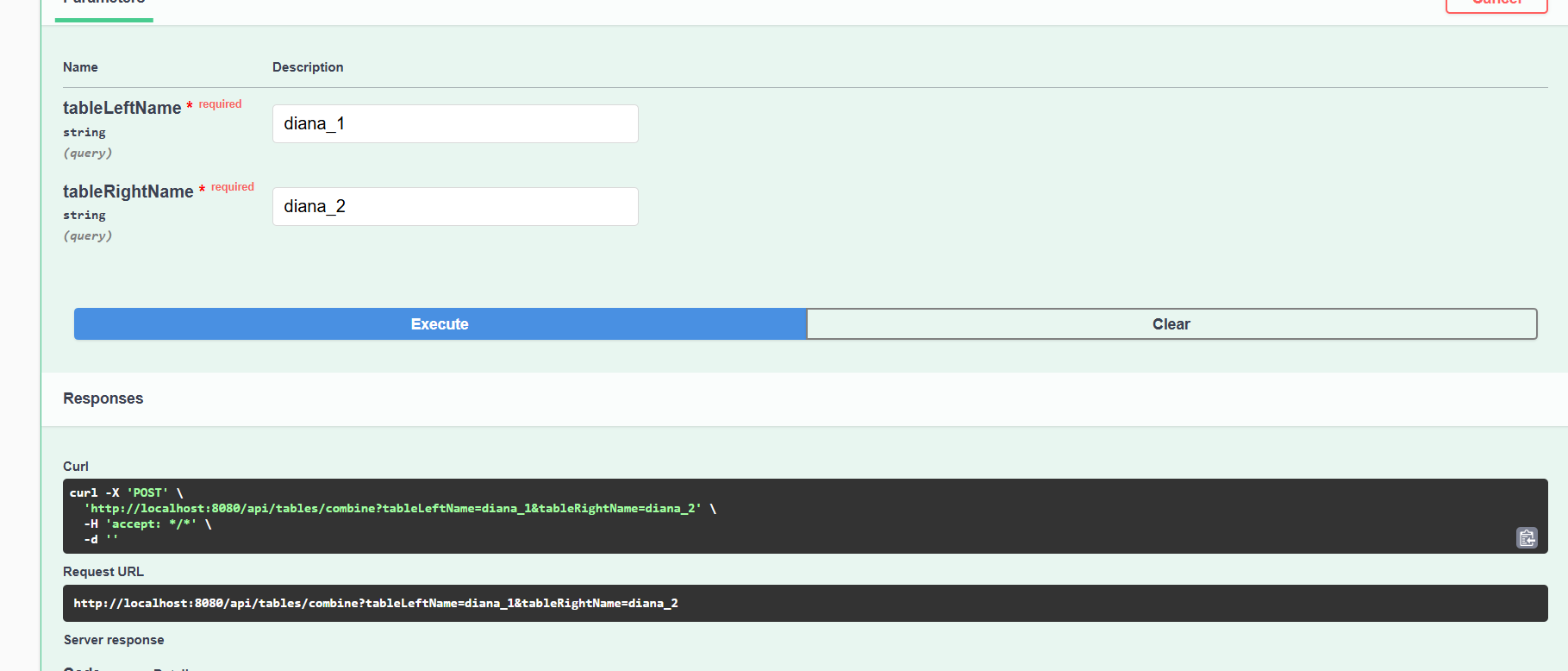
A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated





**{**

**"status": "OK",**

**"report": "",**

**"rows": [**

**{**

**"elements": [**

**{**

**"value": "$123.00",**

**"column": "moneyinv"**

**}**

**],**

**"elementsAll": {**

**"moneyinv": {**

**"value": "$123.00",**

**"column": "moneyinv"**

**}**

**}**

**},**

**{**

**"elements": [**

**{**

**"value": "$150.00;$200.45",**

**"column": "moneyinv"**

**}**

**],**

**"elementsAll": {**

**"moneyinv": {**

**"value": "$150.00;$200.45",**

**"column": "moneyinv"**

**}**

**}**

**},**

**{**

**"elements": [**

**{**

**"value": "$123.00;$1,000.00",**

**"column": "moneyinv"**

**}**

**],**

**"elementsAll": {**

**"moneyinv": {**

**"value": "$123.00;$1,000.00",**

**"column": "moneyinv"**

**}**

**}**

**},**

**{**

**"elements": [**

**{**

**"value": "$600.00;$700.00",**

**"column": "moneyinv"**

**}**

**],**

**"elementsAll": {**

**"moneyinv": {**

**"value": "$600.00;$700.00",**

**"column": "moneyinv"**

**}**

**}**

**}**

**]**

**}**

Тепер вантажимо БД активну.

A screen shot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

Результат завантаження.

{

"filePath": "D:\\java\\diana\_it\_lab\\diana\_bd.json",

"tables": {

"diana\_2": {

"name": "diana\_2",

"rows": [

{

"elements": {

"moneyinv": {

"value": "$600.00;$700.00",

"column": "moneyinv"

}

}

}

],

"columns": {

"moneyinv": {

"type": "MONEY\_INV",

"name": "moneyinv",

"nullAllowed": true

}

}

},

"diana\_1": {

"name": "diana\_1",

"rows": [

{

"elements": {

"moneyinv": {

"value": "$123.00",

"column": "moneyinv"

}

}

},

{

"elements": {

"moneyinv": {

"value": "$150.00;$200.45",

"column": "moneyinv"

}

}

},

{

"elements": {

"moneyinv": {

"value": "$123.00;$1,000.00",

"column": "moneyinv"

}

}

}

],

"columns": {

"moneyinv": {

"type": "MONEY\_INV",

"name": "moneyinv",

"nullAllowed": true

}

}

}

}

}

# 

# **Етап 5. Порівняльний аналіз розробки однієї і тієї ж вашої системи з використанням різних технологій та підходів**

**1. Спільні елементи**

* **Обробка даних:** У всіх трьох реалізаціях основна логіка роботи з даними залишається однаковою. Це стосується CRUD-операцій (створення, читання, оновлення, видалення), використання бази даних, обробки запитів.
* **Архітектура класів:** Концепція таких класів, як "Таблиця", "База даних", "Атрибут", є спільною для всіх реалізацій, навіть якщо вони реалізовані у різних технологіях.
* **Юніт-тестування:** Для забезпечення надійності системи, всі етапи повинні передбачати юніт-тести для основної логіки, що обробляє дані.

**2. Відмінності**

* **Технології:**
  + **Десктопна версія (Етап 2):** Використовується Java Swing для створення графічного інтерфейсу, локальна база даних (наприклад, SQLite або H2). Фокус на локальній роботі без потреби в мережі.
  + **Клієнт-серверна архітектура (Етап 3):** Використовуються технології JavaFX (якщо клієнт десктопний) або JSP/Thymeleaf (якщо вебінтерфейс) та Spring Boot для реалізації серверної частини. Дані зберігаються у віддаленій базі даних, а взаємодія здійснюється через REST API.
  + **Веб-орієнтована версія (Етап 4):** Використовуються Spring MVC, Thymeleaf для реалізації вебінтерфейсу. Запити обробляються через REST/GraphQL/gRPC, що дозволяє забезпечити доступ до системи через веббраузер.
* **Доступність:**
  + **Десктопна версія:** Працює лише на одному комп'ютері, обмежена локальними ресурсами. Недоступна з інших пристроїв.
  + **Клієнт-серверна архітектура:** Доступна в межах локальної мережі або через інтернет, залежно від конфігурації. Потрібен клієнтський додаток.
  + **Веб-орієнтована версія:** Доступна з будь-якого пристрою, що має доступ до мережі та веббраузер. Не вимагає встановлення додаткового програмного забезпечення.
* **Складність розробки та розгортання:**
  + **Десктопна версія:** Найпростіша для розробки, не потребує складної інфраструктури. Але кожен користувач повинен мати локальну копію програми.
  + **Клієнт-серверна архітектура:** Складність збільшується через необхідність налаштування сервера, створення API, синхронізацію даних між клієнтом та сервером.
  + **Веб-орієнтована версія:** Найбільша складність через необхідність створення багаторівневої архітектури, безпеки та масштабованості.
* **Продуктивність:**
  + **Десктопна версія:** Висока продуктивність для локальних задач, оскільки немає мережевої затримки.
  + **Клієнт-серверна архітектура:** Продуктивність залежить від якості мережі, але розділення навантаження між клієнтом і сервером покращує загальну ефективність.
  + **Веб-орієнтована версія:** Продуктивність залежить від серверної інфраструктури та оптимізації коду, але за правильної конфігурації може бути дуже високою.

**3. Переваги та недоліки**

* **Десктопна версія (Java Swing):**
  + **Переваги:**
    - Простота розробки та тестування.
    - Висока продуктивність для локальних задач.
    - Незалежність від мережі.
  + **Недоліки:**
    - Обмежена масштабованість.
    - Необхідність встановлення програми на кожен комп’ютер.
    - Відсутність спільного доступу до даних.
* **Клієнт-серверна архітектура:**
  + **Переваги:**
    - Баланс між складністю розробки та функціональністю.
    - Підтримка одночасної роботи кількох користувачів.
    - Легкість розширення та адаптації.
  + **Недоліки:**
    - Складність налаштування серверної інфраструктури.
    - Залежність від якості мережі.
    - Потреба у підтримці клієнтського додатку.
* **Веб-орієнтована версія:**
  + **Переваги:**
    - Доступність з будь-якого пристрою з інтернетом.
    - Масштабованість та легкість розширення функціональності.
    - Легкість оновлення (немає потреби в оновленні на кожному пристрої).
  + **Недоліки:**
    - Складність розробки та підтримки.
    - Вимоги до серверної інфраструктури.
    - Залежність від наявності інтернет-з'єднання.

**4. Обґрунтування оптимальної технології**

Для вибору оптимальної технології необхідно враховувати контекст використання системи:

* Якщо основна мета — забезпечення швидкої та надійної роботи на окремих комп'ютерах, наприклад, для малих офісів без інтернету, оптимальним вибором є **десктопна версія**.
* Для середніх компаній, де важлива спільна робота та збереження даних у центральній базі, найкраще підійде **клієнт-серверна архітектура**.
* Якщо пріоритетом є доступність та масштабованість, а також планується велика кількість користувачів, то оптимальним вибором буде **веб-орієнтована версія**.

Вибір залежить від конкретних умов та вимог проекту, але загалом, для сучасних систем з розрахунком на масштабованість та доступність, веб-орієнтована архітектура є найкращим рішенням.

**Основна таблиця з порівняннями:**

| **Параметр** | **Десктоп (Етап 2)** | **Клієнт-сервер (Етап 3)** | **Веб-орієнтована версія (Етап 4)** |
| --- | --- | --- | --- |
| **Технології** | Java Swing, SQLite | JavaFX/Swing, Spring Boot, REST | Spring Boot, Spring MVC, Thymeleaf, REST/GraphQL/gRPC |
| **Архітектура** | Локальна база, GUI | Сервер/клієнт, REST | Вебсервер, MVC |
| **Складність реалізації** | Низька | Середня | Висока |
| **Простота тестування** | Висока (малі системи) | Середня (клієнт-сервер взаємодія) | Низька (багато рівнів системи) |
| **Продуктивність** | Висока для локальних задач | Висока | Залежить від інфраструктури |
| **Масштабованість** | Низька | Середня | Висока |
| **Доступність** | Тільки на комп'ютері | Мережа | Веб |
| **Переваги** | Простота, незалежність | Баланс між складністю і гнучкістю | Доступність, масштабованість |
| **Недоліки** | Лімітованість, складність розгортання для різних користувачів | Потреба в мережевій інфраструктурі | Складність розробки та підтримки |