A black and white drawing of a building

Description automatically generated

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського» ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

# Кафедра системного програмування та спеціалізованих комп’ютерних систем

**Розрахункова Графічна робота**

з дисципліни **Бази даних і засоби управління**

*на тему: “Сторення додатку бази данних орієнтованого на взаємодіюд з СУБД PostgreSQL”*

Виконав:

студент ІІI курсу

групи КВ-22

Савельєв О.С

Перевірив:

Павловский В. І.

Київ – 2024

**Мета:** здобуття вмінь проектування бази даних та практичних навичок створення реляційних баз даних за допомогою PostgreSQL.

**Виконання роботи**

**Опис предметної області**

Предметна область – система обліку медичних препаратів, представлена на наведених діаграмах, охоплює сукупність лікарських засобів, їх характеристик, виробників, категорій та хвороб, які ці препарати лікують або для яких вони протипоказані. Основна мета системи – створити структуру для зберігання та управління інформацією про медичні препарати, що дозволить ефективно організувати дані про їх виробництво, класифікацію, використання та протипоказання.

1. Ліки (Medicine)

Атрибути: medicine\_id, назва ліків (medicine\_name).

Призначення: збереження даних про ліки.

2. Категорія (Category)

Атрибути: category\_id, назва категорії (category\_name).

Призначення: збереження даних про категорії ліків.

3. Специфікації (Specs)

Атрибути: specs\_id, назва ліцензії (license\_name), умови видачі (terms\_of\_issue), форма випуску (medicine\_form).

Призначення: збереження специфікацій ліків.

4. Виробник (Manufacturer)

Атрибути: manufacturer\_id, назва виробника (manufacturer\_name), місце розташування виробничих потужностей (location\_of\_facilities).

Призначення: збереження інформації про виробників ліків.

5. Хвороба (Disease)

Атрибути: disease\_id, назва хвороби (disease\_name).

Призначення: збереження інформації про хвороби, які можуть бути лікувані або протипоказані для певних ліків.

**Опис зв’язків між сутностями:**

1. Зв'язок «Ліки» - «Категорія»:

Тип зв’язку: N:1

Опис: Одні ліки можуть належати до однієї категорії (category\_id), але одна категорія може мати багато ліків.

2. Зв'язок «Ліки» - «Специфікації»:

Тип зв’язку: 1:1.

Опис: Кожен препарат має конкретні специфікації (specs\_id).

3. Зв'язок «Ліки» - «Виробник»:

Тип зв’язку: N:1.

Опис: Одні ліки можуть бути виготовлені лише одним виробником (manufacturer\_id), але один виробник може виготовляти багато ліків.

4. Зв'язок «Ліки» - «Хвороба (Лікування)» *Призначення*:

Тип зв’язку: N:M.

Опис: Одні ліки можуть назначатись для декількох, і кожна хвороба може лікуватися багатьма ліками (treats).

5. Зв'язок «Ліки» - «Хвороба (Протипоказання)»: *Протипоказання*

Тип зв’язку: N:M.

Опис: Одні ліки можуть мати протипоказання для багатьох хвороб, і кожна хвороба може мати протипоказання лише для певних ліків (contradicts).

Изображение выглядит как текст, диаграмма, рисунок, снимок экрана

Автоматически созданное описание

Рисунок 1 – ER-діаграма, побудована за нотацією Чена

**Перетворення концептуальної моделі у логічну модель та схему бази даних**

Сутність Medicine перетворено в таблицю Medicine з первинним ключем Medicine\_id та атрибутами Medicine\_name (назва препарату) та зовнішніми ключами Specs\_id, Category\_id, Teats\_id (ідентифікатори характеристик, категорій та захворювань, які лікуються).

Сутність Category перетворено в таблицю Category з первинним ключем Category\_id та атрибутами Category\_name (назва категорії).

Сутність Specs перетворено в таблицю Specs з первинним ключем Specs\_id та атрибутами License\_name (дані про ліцензію), Terms\_of\_issue (умови видачі), Medicine\_form (форма препарату).

Сутність Manufacturer перетворено в таблицю Manufacturer з первинним ключем Manufacturer\_id та атрибутами Manufacturer\_name (назва виробника) та Location\_of\_facilities (місцезнаходження виробничих потужностей).

Сутність Disease перетворено в таблицю Disease з первинним ключем Disease\_id та атрибутами Disease\_name (назва захворювання).

Оскільки в логічній моделі безпосередній зв’язок N:M існує між сутностями Medicine та Disease (ліки, які лікують конкретні захворювання), для його реалізації була створена таблиця Treats з первинним ключем tab\_id, що містить зовнішні ключі Medicine\_id та Disease\_id, що пов’язують ліки з відповідними захворюваннями.

Оскільки також існує зв’язок N:M між сутностями Medicine та Manufacturer (ліки, які виробляються конкретними виробниками), для його реалізації була створена таблиця Made\_by з первинним ключем tab\_id, що містить зовнішні ключі Medicine\_id та Manufacturer\_id, що пов’язують ліки з виробниками.

Графічне подання логічної моделі «Сутність-зв’язок» зображено на рисунку 2.

Изображение выглядит как текст, снимок экрана, меню, диаграмма

Автоматически созданное описание

Рисунок 2 – Схема бази даних

**Серидовище та компоненти розробки**

У процесы роботи було вибрано користуватись мовою С++ та бібліотекою pqxx

**Шаблон проектування**

Модель-представлення-контролер (MVC) – це шаблон проектування, що використовується у програмі. Кожен компонент відповідає за певну функціональну частину:

1.Модель (Model) це клас, що відображає логіку роботи з даними,

обробляє всі операції з даними, такі як додавання, оновлення, вилучення.

2. Представлення (View) - це клас, через який користувач взаємодіє з програмою. У даному випадку, консольний інтерфейс, який відображає дані для користувача та зчитує їх з екрану.

3. Контролер (Controller) це клас, який відповідає за зв'язок між користувачем і системою. Він приймає введені користувачем дані та обробляє їх. В залежності від результатів, викликає відповідні дії з Model або View.

Даний підхід дозволяє розділити логіку програми на логічні компоненти, що полегшує розробку, тестування і підтримку продукту.

**Структура програми**

*Изображение выглядит как снимок экрана, текст, Мультимедийное программное обеспечение, программное обеспечение

Автоматически созданное описание*

*Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание*

3 файлу RGRBD.cpp відбувається виклик контролера та передача йому управління.

У файлі MODL.cpp описаний клас моделі, який відповідає за управління підключенням до бази даних і виконанням низькорівневих запитів до неї.

У файлі CONTRL.py реалізовано інтерфейс взаємодії з користувачем, включаючи обробку запитів користувача, виконання пошуку, а також інші дії, необхідні для взаємодії з моделлю та представленням.

У файлі VIEW.cpp описаний клас, який відображає результати виконання різних дій користувача на екрані консолі. Цей компонент відповідає за представлення даних користувачу в зручному для сприйняття вигляді.

Отже, структура програми відповідає патерну МѴС.

**Код програми**

**MODL.cpp**

#pragma once

#include <iostream>

#include "VIEW.cpp"

#include <locale>

#include <pqxx/pqxx> // Library for PostgreSQL operations

#include <string>

#include <cstdlib>

#include <ctime>

class Model {

protected:

pqxx::connection\* conn;

pqxx::connection\* connectToDatabase() {

try {

// Establish connection to the database

conn = new pqxx::connection("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

if (!conn->is\_open()) {

std::cerr << "Error: Unable to open database connection." << std::endl;

delete conn;

conn = nullptr;

}

return conn;

}

catch (const std::exception& e) {

std::cerr << "Database connection error: " << e.what() << std::endl;

return nullptr;

}

}

public:

Model(pqxx::connection\* connection) : conn(connection) {}

~Model() = default;

int findNextFreeId(pqxx::work& txn, const std::string& table\_name) {

// Query to dynamically get the primary key column name for the specified table

std::string pk\_query = "SELECT column\_name FROM information\_schema.key\_column\_usage "

"WHERE table\_name = " + txn.quote(table\_name) + " AND constraint\_name = "

"(SELECT constraint\_name FROM information\_schema.table\_constraints "

"WHERE table\_name = " + txn.quote(table\_name) + " AND constraint\_type = 'PRIMARY KEY');";

pqxx::result pk\_result = txn.exec(pk\_query);

if (pk\_result.empty()) {

throw std::runtime\_error("No primary key found for table: " + table\_name);

}

std::string pk\_column = pk\_result[0][0].c\_str();

// Query to find the next available ID based on the primary key column

std::string id\_query = "SELECT COALESCE(MAX(" + pk\_column + "), 0) + 1 FROM public.\"" + table\_name + "\";";

pqxx::result id\_result = txn.exec(id\_query);

return id\_result[0][0].as<int>();

}

void createMedicine(int medicine\_id, int manufacturer\_id, int specs\_id, int category\_id, int treats\_id, int contradicts\_id, const std::string& medicine\_name) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

// Insert into Medicines table

std::string query\_medicine =

"INSERT INTO public.\"Medicines\" (\"Medicine\_id\", \"Manufacturer\_id\", \"Specs\_id\", \"Category\_id\", \"Treats\_id\", \"Contradicts\_id\", \"Medicine\_name\") VALUES (" +

txn.quote(medicine\_id) + ", " + txn.quote(manufacturer\_id) + ", " + txn.quote(specs\_id) + ", " + txn.quote(category\_id) + ", " + txn.quote(treats\_id) + ", " + txn.quote(contradicts\_id) + ", " + txn.quote(medicine\_name) + ");";

txn.exec(query\_medicine);

// Insert into Medicine\_Treats\_Diseases

int next\_treats\_id = findNextFreeId(txn, "Medicine\_Treats\_Diseases");

std::string query\_treats =

"INSERT INTO public.\"Medicine\_Treats\_Diseases\" (tab\_id, \"Medicine\_id\", \"Disease\_id\") VALUES (" +

txn.quote(next\_treats\_id) + ", " + txn.quote(medicine\_id) + ", " + txn.quote(treats\_id) + ");";

txn.exec(query\_treats);

// Insert into Medicine\_Contradicts\_Diseases

int next\_contradicts\_id = findNextFreeId(txn, "Medicine\_Contradicts\_Diseases");

std::string query\_contradicts =

"INSERT INTO public.\"Medicine\_Contradicts\_Diseases\" (tab\_id, \"Medicine\_id\", \"Disease\_id\") VALUES (" +

txn.quote(next\_contradicts\_id) + ", " + txn.quote(medicine\_id) + ", " + txn.quote(contradicts\_id) + ");";

txn.exec(query\_contradicts);

// Insert into man\_med\_tab

int next\_man\_med\_id = findNextFreeId(txn, "man\_med\_tab");

std::string query\_man\_med =

"INSERT INTO public.man\_med\_tab (tab\_id, \"Medicine\_id\", \"Manufacturer\_id\") VALUES (" +

txn.quote(next\_man\_med\_id) + ", " + txn.quote(medicine\_id) + ", " + txn.quote(manufacturer\_id) + ");";

txn.exec(query\_man\_med);

txn.commit();

std::cout << "Medicine and related records created successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl;

}

}

void readMedicine(int medicine\_id) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

if (!conn.is\_open()) {

std::cerr << "Error: Unable to connect to the database." << std::endl;

return;

}

pqxx::work txn(conn);

std::string query = "SELECT \* FROM public.\"Medicines\" WHERE \"Medicine\_id\" = " + txn.quote(medicine\_id) + ";";

std::cout << "Executing query: " << query << std::endl;

pqxx::result result = txn.exec(query);

if (result.empty()) {

std::cout << "No medicine found with Medicine\_id: " << medicine\_id << std::endl;

return;

}

// Access columns by index in the result set

for (const auto& row : result) {

std::cout << "Medicine ID: " << row[0].as<int>() << std::endl; // "Medicine\_id"

std::cout << "Manufacturer ID: " << row[1].as<int>() << std::endl; // "Manufacturer\_id"

std::cout << "Specs ID: " << row[2].as<int>() << std::endl; // "Specs\_id"

std::cout << "Category ID: " << row[3].as<int>() << std::endl; // "Category\_id"

std::cout << "Treats ID: " << row[4].as<int>() << std::endl; // "Treats\_id"

std::cout << "Contradicts ID: " << row[5].as<int>() << std::endl; // "Contradicts\_id"

std::cout << "Medicine Name: " << row[6].c\_str() << std::endl; // "Medicine\_name"

}

}

catch (const std::exception& e) {

std::cerr << "Error reading medicine: " << e.what() << std::endl;

}

}

void updateMedicine(int medicine\_id, const std::string& column\_name, const std::string& new\_value) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

// Update the Medicines table

std::string query = "UPDATE public.\"Medicines\" SET \"" + column\_name + "\" = " + txn.quote(new\_value) +

" WHERE \"Medicine\_id\" = " + txn.quote(medicine\_id) + ";";

txn.exec(query);

// If the column is Treats\_id, update Medicine\_Treats\_Diseases

if (column\_name == "Treats\_id") {

std::string query\_update\_treats =

"UPDATE public.\"Medicine\_Treats\_Diseases\" SET \"Disease\_id\" = " + txn.quote(new\_value) +

" WHERE \"Medicine\_id\" = " + txn.quote(medicine\_id) + ";";

txn.exec(query\_update\_treats);

}

// If the column is Contradicts\_id, update Medicine\_Contradicts\_Diseases

if (column\_name == "Contradicts\_id") {

std::string query\_update\_contradicts =

"UPDATE public.\"Medicine\_Contradicts\_Diseases\" SET \"Disease\_id\" = " + txn.quote(new\_value) +

" WHERE \"Medicine\_id\" = " + txn.quote(medicine\_id) + ";";

txn.exec(query\_update\_contradicts);

}

// If the column is Manufacturer\_id, update man\_med\_tab

if (column\_name == "Manufacturer\_id") {

std::string query\_update\_man\_med =

"UPDATE public.man\_med\_tab SET \"Manufacturer\_id\" = " + txn.quote(new\_value) +

" WHERE \"Medicine\_id\" = " + txn.quote(medicine\_id) + ";";

txn.exec(query\_update\_man\_med);

}

txn.commit();

std::cout << "Medicine and related records updated successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl;

}

}

void deleteMedicine(int medicine\_id) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

// Delete from Medicine\_Treats\_Diseases

std::string query\_delete\_treats = "DELETE FROM public.\"Medicine\_Treats\_Diseases\" WHERE \"Medicine\_id\" = " + txn.quote(medicine\_id) + ";";

txn.exec(query\_delete\_treats);

// Delete from Medicine\_Contradicts\_Diseases

std::string query\_delete\_contradicts = "DELETE FROM public.\"Medicine\_Contradicts\_Diseases\" WHERE \"Medicine\_id\" = " + txn.quote(medicine\_id) + ";";

txn.exec(query\_delete\_contradicts);

// Delete from man\_med\_tab

std::string query\_delete\_man\_med = "DELETE FROM public.man\_med\_tab WHERE \"Medicine\_id\" = " + txn.quote(medicine\_id) + ";";

txn.exec(query\_delete\_man\_med);

// Delete from Medicines table

std::string query\_delete\_medicine = "DELETE FROM public.\"Medicines\" WHERE \"Medicine\_id\" = " + txn.quote(medicine\_id) + ";";

txn.exec(query\_delete\_medicine);

txn.commit();

std::cout << "Medicine and related records deleted successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl;

}

}

//SPECS

void createSpec(int specs\_id, const std::string& license\_name, bool term\_of\_issue, const std::string& medicine\_form) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

std::string query = "INSERT INTO public.\"Specs\" (\"specs\_id\", \"Lisence\_name\", \"Term\_of\_issue\", \"Medicine\_form\") VALUES (" +

txn.quote(specs\_id) + ", " + txn.quote(license\_name) + ", " + txn.quote(term\_of\_issue) + ", " + txn.quote(medicine\_form) + ");";

txn.exec(query);

txn.commit();

std::cout << "Spec created successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl;

}

}

void readSpec(int specs\_id) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

if (!conn.is\_open()) {

std::cerr << "Error: Unable to connect to the database." << std::endl;

return;

}

pqxx::work txn(conn);

std::string query = "SELECT \* FROM public.\"Specs\" WHERE \"specs\_id\" = " + txn.quote(specs\_id) + ";";

std::cout << "Executing query: " << query << std::endl;

pqxx::result result = txn.exec(query);

if (result.empty()) {

std::cout << "No spec found with Specs\_id: " << specs\_id << std::endl;

return;

}

// Access columns by index if exact names cause issues

for (const auto& row : result) {

std::cout << "Specs ID: " << row[0].as<int>() << std::endl;

std::cout << "License Name: " << row[1].c\_str() << std::endl;

std::cout << "Term of Issue: " << (row[2].as<bool>() ? "True" : "False") << std::endl;

std::cout << "Medicine Form: " << row[3].c\_str() << std::endl;

}

}

catch (const std::exception& e) {

std::cerr << "Error reading spec: " << e.what() << std::endl;

}

}

void updateSpec(int specs\_id, const std::string& column\_name, const std::string& new\_value) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

std::string query = "UPDATE public.\"Specs\" SET \"" + column\_name + "\" = " + txn.quote(new\_value) +

" WHERE \"specs\_id\" = " + txn.quote(specs\_id) + ";";

txn.exec(query);

txn.commit();

std::cout << "Spec updated successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl;

}

}

void createCategory(int category\_id, const std::string& category\_name) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

std::string query = "INSERT INTO public.\"Categories\" (\"Category\_ID\", \"Category\_Name\") VALUES (" +

txn.quote(category\_id) + ", " + txn.quote(category\_name) + ");";

txn.exec(query);

txn.commit();

std::cout << "Category created successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl;

}

}

void readCategory(int category\_id) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

std::string query = "SELECT \* FROM public.\"Categories\" WHERE \"Category\_ID\" = " + txn.quote(category\_id) + ";";

pqxx::result result = txn.exec(query);

if (result.empty()) {

std::cout << "No category found with Category\_ID: " << category\_id << std::endl;

return;

}

for (const auto& row : result) {

std::cout << "Category ID: " << row[0].as<int>() << std::endl;

std::cout << "Category Name: " << row[1].c\_str() << std::endl;

}

}

catch (const std::exception& e) {

std::cerr << "Error reading category: " << e.what() << std::endl;

}

}

void deleteSpec(int specs\_id) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

std::string query = "DELETE FROM public.\"Specs\" WHERE \"specs\_id\" = " + txn.quote(specs\_id) + ";";

txn.exec(query);

txn.commit();

std::cout << "Spec deleted successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error deleting spec: " << e.what() << std::endl;

}

}

void updateCategory(int category\_id, const std::string& new\_name) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

std::string query = "UPDATE public.\"Categories\" SET \"Category\_Name\" = " + txn.quote(new\_name) +

" WHERE \"Category\_ID\" = " + txn.quote(category\_id) + ";";

txn.exec(query);

txn.commit();

std::cout << "Category updated successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl;

}

}

void deleteCategory(int category\_id) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

std::string query = "DELETE FROM public.\"Categories\" WHERE \"Category\_ID\" = " + txn.quote(category\_id) + ";";

txn.exec(query);

txn.commit();

std::cout << "Category deleted successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl;

}

}

void createDisease(int disease\_id, const std::string& disease\_name) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

std::string query = "INSERT INTO public.\"Diseases\" (\"Disease\_id\", \"Disease\_name\") VALUES (" +

txn.quote(disease\_id) + ", " + txn.quote(disease\_name) + ");";

txn.exec(query);

txn.commit();

std::cout << "Disease created successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl;

}

}

void readDisease(int disease\_id) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

std::string query = "SELECT \* FROM public.\"Diseases\" WHERE \"Disease\_id\" = " + txn.quote(disease\_id) + ";";

pqxx::result result = txn.exec(query);

if (result.empty()) {

std::cout << "No disease found with Disease\_id: " << disease\_id << std::endl;

return;

}

for (const auto& row : result) {

std::cout << "Disease ID: " << row[0].as<int>() << std::endl;

std::cout << "Disease Name: " << row[1].c\_str() << std::endl;

}

}

catch (const std::exception& e) {

std::cerr << "Error reading disease: " << e.what() << std::endl;

}

}

void updateDisease(int disease\_id, const std::string& new\_name) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

std::string query = "UPDATE public.\"Diseases\" SET \"Disease\_name\" = " + txn.quote(new\_name) +

" WHERE \"Disease\_id\" = " + txn.quote(disease\_id) + ";";

txn.exec(query);

txn.commit();

std::cout << "Disease updated successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl;

}

}

void deleteDisease(int disease\_id) {

try {

pqxx::connection conn("dbname=postgres user=postgres password=1 host=127.0.0.1 port=5432");

pqxx::work txn(conn);

std::string query = "DELETE FROM public.\"Diseases\" WHERE \"Disease\_id\" = " + txn.quote(disease\_id) + ";";

txn.exec(query);

txn.commit();

std::cout << "Disease deleted successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl;

}

}

void createManufacturer(int manufacturer\_id, const std::string& manufacturer\_name, const std::string& location) {

if (!connectToDatabase()) return;

try {

pqxx::work txn(\*conn);

std::string query = "INSERT INTO public.\"Manufactures\" (\"Manufacturer\_id\", \"Manufacturer\_name\", \"Location of facilities\") "

"VALUES (" + txn.quote(manufacturer\_id) + ", " + txn.quote(manufacturer\_name) + ", " + txn.quote(location) + ");";

txn.exec(query);

txn.commit();

std::cout << "Manufacturer created successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error creating manufacturer: " << e.what() << std::endl;

}

}

// Read a manufacturer along with its related medicines

void readManufacturer(int manufacturer\_id) {

if (!connectToDatabase()) return;

try {

pqxx::work txn(\*conn);

// Corrected query to use the "Manufactures" table

std::string query = "SELECT \* FROM public.\"Manufactures\" WHERE \"Manufacturer\_id\" = " + txn.quote(manufacturer\_id) + ";";

pqxx::result result = txn.exec(query);

if (result.empty()) {

std::cout << "No manufacturer found with Manufacturer\_id: " << manufacturer\_id << std::endl;

return;

}

// Display manufacturer information

for (const auto& row : result) {

std::cout << "Manufacturer ID: " << row[0].as<int>() << "\n"

<< "Manufacturer Name: " << row[1].c\_str() << "\n"

<< "Location of facilities: " << row[2].c\_str() << "\n"

<< "-----------------------------------" << std::endl;

}

}

catch (const std::exception& e) {

std::cerr << "Error reading manufacturer: " << e.what() << std::endl;

}

}

void updateManufacturer(int manufacturer\_id, const std::string& column\_name, const std::string& new\_value) {

if (!connectToDatabase()) return;

try {

pqxx::work txn(\*conn);

std::string query = "UPDATE public.\"Manufactures\" SET \"" + column\_name + "\" = " + txn.quote(new\_value) +

" WHERE \"Manufacturer\_id\" = " + txn.quote(manufacturer\_id) + ";";

txn.exec(query);

txn.commit();

std::cout << "Manufacturer updated successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error updating manufacturer: " << e.what() << std::endl;

}

}

void deleteManufacturer(int manufacturer\_id) {

if (!connectToDatabase()) return;

try {

pqxx::work txn(\*conn);

// Delete related entries from the junction table first

std::string delete\_junction = "DELETE FROM public.man\_med\_tab WHERE \"Manufacturer\_id\" = " + txn.quote(manufacturer\_id) + ";";

txn.exec(delete\_junction);

// Delete from Manufacturers table

std::string delete\_manufacturer = "DELETE FROM public.\"Manufactures\" WHERE \"Manufacturer\_id\" = " + txn.quote(manufacturer\_id) + ";";

txn.exec(delete\_manufacturer);

txn.commit();

std::cout << "Manufacturer and related records deleted successfully." << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error deleting manufacturer: " << e.what() << std::endl;

}

}

void generateData(int count, int choice) {

if (choice == 1) {

try {

pqxx::work txn(\*conn);

srand(static\_cast<unsigned>(time(0))); // Seed for random number generation

for (int i = 0; i < count; ++i) {

// Generate the next available Medicine ID

int medicine\_id = txn.exec("SELECT COALESCE(MAX(\"Medicine\_id\"), 0) + 1 FROM public.\"Medicines\";")[0][0].as<int>();

// Generate unique Specs data and insert into the Specs table

int specs\_id = txn.exec("SELECT COALESCE(MAX(\"specs\_id\"), 0) + 1 FROM public.\"Specs\";")[0][0].as<int>();

std::string license\_name = "License\_" + std::to\_string(rand() % 1000);

bool term\_of\_issue = rand() % 2 == 0;

std::string medicine\_form = (rand() % 2 == 0) ? "Tablet" : "Capsule";

// Insert the new Specs entry

std::string insert\_specs\_query =

"INSERT INTO public.\"Specs\" (\"specs\_id\", \"Lisence\_name\", \"Term\_of\_issue\", \"Medicine\_form\") VALUES (" +

txn.quote(specs\_id) + ", " + txn.quote(license\_name) + ", " + (term\_of\_issue ? "TRUE" : "FALSE") + ", " + txn.quote(medicine\_form) + ");";

txn.exec(insert\_specs\_query);

// Randomly select existing Manufacturer, Category, and Disease IDs to use as foreign keys

int manufacturer\_id = txn.exec("SELECT \"Manufacturer\_id\" FROM public.\"Manufactures\" ORDER BY RANDOM() LIMIT 1;")[0][0].as<int>();

int category\_id = txn.exec("SELECT \"Category\_ID\" FROM public.\"Categories\" ORDER BY RANDOM() LIMIT 1;")[0][0].as<int>();

int treats\_id = txn.exec("SELECT \"Disease\_id\" FROM public.\"Diseases\" ORDER BY RANDOM() LIMIT 1;")[0][0].as<int>();

int contradicts\_id = txn.exec("SELECT \"Disease\_id\" FROM public.\"Diseases\" ORDER BY RANDOM() LIMIT 1;")[0][0].as<int>();

// Generate a random name for the medicine

std::string medicine\_name = "Medicine\_" + std::to\_string(rand() % 1000);

// Insert into Medicines table

std::string insert\_medicine\_query =

"INSERT INTO public.\"Medicines\" (\"Medicine\_id\", \"Manufacturer\_id\", \"Specs\_id\", \"Category\_id\", \"Treats\_id\", \"Contradicts\_id\", \"Medicine\_name\") VALUES (" +

txn.quote(medicine\_id) + ", " + txn.quote(manufacturer\_id) + ", " + txn.quote(specs\_id) + ", " +

txn.quote(category\_id) + ", " + txn.quote(treats\_id) + ", " + txn.quote(contradicts\_id) + ", " + txn.quote(medicine\_name) + ");";

txn.exec(insert\_medicine\_query);

// Insert into Medicine\_Contradicts\_Diseases junction table

int contradicts\_tab\_id = txn.exec("SELECT COALESCE(MAX(tab\_id), 0) + 1 FROM public.\"Medicine\_Contradicts\_Diseases\";")[0][0].as<int>();

std::string insert\_contradicts\_query =

"INSERT INTO public.\"Medicine\_Contradicts\_Diseases\" (tab\_id, \"Medicine\_id\", \"Disease\_id\") VALUES (" +

txn.quote(contradicts\_tab\_id) + ", " + txn.quote(medicine\_id) + ", " + txn.quote(contradicts\_id) + ");";

txn.exec(insert\_contradicts\_query);

// Insert into Medicine\_Treats\_Diseases junction table

int treats\_tab\_id = txn.exec("SELECT COALESCE(MAX(tab\_id), 0) + 1 FROM public.\"Medicine\_Treats\_Diseases\";")[0][0].as<int>();

std::string insert\_treats\_query =

"INSERT INTO public.\"Medicine\_Treats\_Diseases\" (tab\_id, \"Medicine\_id\", \"Disease\_id\") VALUES (" +

txn.quote(treats\_tab\_id) + ", " + txn.quote(medicine\_id) + ", " + txn.quote(treats\_id) + ");";

txn.exec(insert\_treats\_query);

// Insert into man\_med\_tab junction table

int man\_med\_tab\_id = txn.exec("SELECT COALESCE(MAX(tab\_id), 0) + 1 FROM public.\"man\_med\_tab\";")[0][0].as<int>();

std::string insert\_man\_med\_query =

"INSERT INTO public.\"man\_med\_tab\" (tab\_id, \"Medicine\_id\", \"Manufacturer\_id\") VALUES (" +

txn.quote(man\_med\_tab\_id) + ", " + txn.quote(medicine\_id) + ", " + txn.quote(manufacturer\_id) + ");";

txn.exec(insert\_man\_med\_query);

}

// Commit the transaction after inserting all rows

txn.commit();

std::cout << "Random data generated successfully for the Medicines table and associated junction tables." << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error generating random data for Medicines: " << e.what() << std::endl;

}

}

else if (choice == 2) {

try {

pqxx::work txn(\*conn);

srand(static\_cast<unsigned>(time(0))); // Seed for random number generation

for (int i = 0; i < count; ++i) {

// Generate random values for each column

int specs\_id = txn.exec("SELECT COALESCE(MAX(\"specs\_id\"), 0) + 1 FROM public.\"Specs\";")[0][0].as<int>(); // Get the next available ID

std::string license\_name = "License\_" + std::to\_string(rand() % 1000); // Random license name

bool term\_of\_issue = rand() % 2 == 0; // Random boolean

std::string medicine\_form = (rand() % 2 == 0) ? "Tablet" : "Capsule"; // Random medicine form

// Construct the SQL insert query

std::string insert\_query =

"INSERT INTO public.\"Specs\" (\"specs\_id\", \"Lisence\_name\", \"Term\_of\_issue\", \"Medicine\_form\") VALUES (" +

txn.quote(specs\_id) + ", " + txn.quote(license\_name) + ", " + (term\_of\_issue ? "TRUE" : "FALSE") + ", " + txn.quote(medicine\_form) + ");";

// Execute the insert query

txn.exec(insert\_query);

}

// Commit the transaction after inserting all rows

txn.commit();

std::cout << "Random data generated successfully for the Specs table." << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error generating random data for Specs: " << e.what() << std::endl;

}

}

else if (choice == 3) {

try {

pqxx::work txn(\*conn);

srand(static\_cast<unsigned>(time(0))); // Seed for random number generation

for (int i = 0; i < count; ++i) {

// Generate random values for each column

int category\_id = txn.exec("SELECT COALESCE(MAX(\"Category\_ID\"), 0) + 1 FROM public.\"Categories\";")[0][0].as<int>(); // Get the next available ID

std::string category\_name = "Category\_" + std::to\_string(rand() % 1000); // Random category name

// Construct the SQL insert query

std::string insert\_query =

"INSERT INTO public.\"Categories\" (\"Category\_ID\", \"Category\_Name\") VALUES (" +

txn.quote(category\_id) + ", " + txn.quote(category\_name) + ");";

// Execute the insert query

txn.exec(insert\_query);

}

// Commit the transaction after inserting all rows

txn.commit();

std::cout << "Random data generated successfully for the Categories table." << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error generating random data for Categories: " << e.what() << std::endl;

}

}

else if (choice == 4) {

try {

pqxx::work txn(\*conn);

srand(static\_cast<unsigned>(time(0))); // Seed for random number generation

for (int i = 0; i < count; ++i) {

// Generate random values for each column

int disease\_id = txn.exec("SELECT COALESCE(MAX(\"Disease\_id\"), 0) + 1 FROM public.\"Diseases\";")[0][0].as<int>(); // Get the next available ID

std::string disease\_name = "Disease\_" + std::to\_string(rand() % 1000); // Random disease name

// Construct the SQL insert query

std::string insert\_query =

"INSERT INTO public.\"Diseases\" (\"Disease\_id\", \"Disease\_name\") VALUES (" +

txn.quote(disease\_id) + ", " + txn.quote(disease\_name) + ");";

// Execute the insert query

txn.exec(insert\_query);

}

// Commit the transaction after inserting all rows

txn.commit();

std::cout << "Random data generated successfully for the Diseases table." << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error generating random data for Diseases: " << e.what() << std::endl;

}

}

else if (choice == 5) {

try {

pqxx::work txn(\*conn);

srand(static\_cast<unsigned>(time(0))); // Seed for random number generation

for (int i = 0; i < count; ++i) {

// Generate random values for each column

int manufacturer\_id = txn.exec("SELECT COALESCE(MAX(\"Manufacturer\_id\"), 0) + 1 FROM public.\"Manufactures\";")[0][0].as<int>(); // Get the next available ID

std::string manufacturer\_name = "Manufacturer\_" + std::to\_string(rand() % 1000); // Random manufacturer name

std::string location = "Location\_" + std::to\_string(rand() % 100); // Random location name

// Construct the SQL insert query

std::string insert\_query =

"INSERT INTO public.\"Manufactures\" (\"Manufacturer\_id\", \"Manufacturer\_name\", \"Location of facilities\") VALUES (" +

txn.quote(manufacturer\_id) + ", " + txn.quote(manufacturer\_name) + ", " + txn.quote(location) + ");";

// Execute the insert query

txn.exec(insert\_query);

}

// Commit the transaction after inserting all rows

txn.commit();

std::cout << "Random data generated successfully for the Manufacturers table." << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error generating random data for Manufacturers: " << e.what() << std::endl;

}

}

}

void outputAllTableNames() {

try {

pqxx::work txn(\*conn);

// Query to get all table names in the 'public' schema

std::string query = "SELECT table\_name FROM information\_schema.tables WHERE table\_schema = 'public';";

pqxx::result result = txn.exec(query);

std::cout << "Tables in the database:" << std::endl;

for (const auto& row : result) {

std::string table\_name = row["table\_name"].c\_str();

std::cout << "- " << table\_name << std::endl;

}

}

catch (const std::exception& e) {

std::cerr << "Error retrieving table names: " << e.what() << std::endl;

}

}

void outputTableColumns(const std::string& table\_name) {

try {

pqxx::work txn(\*conn);

// Query to get all column names for the specified table in the 'public' schema

std::string query = "SELECT column\_name FROM information\_schema.columns WHERE table\_schema = 'public' AND table\_name = " + txn.quote(table\_name) + ";";

pqxx::result result = txn.exec(query);

// Check if the table has any columns

if (result.empty()) {

std::cout << "No columns found for table: " << table\_name << " (or the table does not exist)." << std::endl;

return;

}

std::cout << "Columns in table '" << table\_name << "':" << std::endl;

for (const auto& row : result) {

std::string column\_name = row["column\_name"].c\_str();

std::cout << "- " << column\_name << std::endl;

}

}

catch (const std::exception& e) {

std::cerr << "Error retrieving columns for table '" << table\_name << "': " << e.what() << std::endl;

}

}

void outputMostCommonManufacturer() {

if (!connectToDatabase()) return; // Ensure the database connection is established

try {

pqxx::work txn(\*conn);

// Query to find the most common Manufacturer\_id in the man\_med\_tab table

std::string query =

"SELECT \"Manufacturer\_id\", COUNT(\*) AS frequency "

"FROM public.\"man\_med\_tab\" "

"GROUP BY \"Manufacturer\_id\" "

"ORDER BY frequency DESC "

"LIMIT 1;";

pqxx::result result = txn.exec(query);

if (result.empty()) {

std::cout << "No manufacturers found in the man\_med\_tab table." << std::endl;

return;

}

// Extract the most common Manufacturer\_id and its frequency

// Extract the most common Manufacturer\_id and its frequency

int manufacturer\_id = result[0][0].as<int>(); // Assuming Manufacturer\_id is in the first column

int frequency = result[0][1].as<int>(); // Assuming frequency is in the second column

std::cout << "Most common manufacturer in the table:" << std::endl;

std::cout << "- Manufacturer ID: " << manufacturer\_id << std::endl;

std::cout << "- Occurrences: " << frequency << std::endl;

// Now, use the manufacturer\_id to fetch detailed information

std::string manufacturerQuery =

"SELECT \* FROM public.\"Manufactures\" WHERE \"Manufacturer\_id\" = " + txn.quote(manufacturer\_id) + ";";

pqxx::result manufacturerResult = txn.exec(manufacturerQuery);

if (manufacturerResult.empty()) {

std::cout << "No details found for Manufacturer\_id: " << manufacturer\_id << std::endl;

return;

}

// Display detailed manufacturer information

const auto& row = manufacturerResult[0];

std::cout << "- Manufacturer Name: " << row[1].c\_str() << std::endl;

std::cout << "- Location: " << row[2].c\_str() << std::endl;

}

catch (const pqxx::sql\_error& e) {

std::cerr << "SQL error: " << e.what() << std::endl;

std::cerr << "Query was: " << e.query() << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error retrieving most common manufacturer: " << e.what() << std::endl;

}

}

void outputMostCommonTreat() {

try {

pqxx::work txn(\*conn);

// Query to find the most common Disease\_id in the Medicine\_Treats\_Diseases table

std::string query =

"SELECT d.\"Disease\_id\", dis.\"Disease\_name\", COUNT(\*) AS frequency "

"FROM public.\"Medicine\_Treats\_Diseases\" d "

"JOIN public.\"Diseases\" dis ON d.\"Disease\_id\" = dis.\"Disease\_id\" "

"GROUP BY d.\"Disease\_id\", dis.\"Disease\_name\" "

"ORDER BY frequency DESC "

"LIMIT 1;";

pqxx::result result = txn.exec(query);

if (result.empty()) {

std::cout << "No diseases found in the Medicine\_Treats\_Diseases table." << std::endl;

return;

}

// Extract the most common Disease\_id and its frequency

int disease\_id = result[0][0].as<int>(); // Disease\_id

std::string disease\_name = result[0][1].c\_str(); // Disease\_name

int frequency = result[0][2].as<int>(); // frequency

// Output the most common disease

std::cout << "Most common Treated disease:" << std::endl;

std::cout << "- Disease ID: " << disease\_id << std::endl;

std::cout << "- Disease Name: " << disease\_name << std::endl;

std::cout << "- Occurrences: " << frequency << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error retrieving most common disease: " << e.what() << std::endl;

}

}

void outputMostCommonContradicted() {

try {

pqxx::work txn(\*conn);

// Query to find the most common Disease\_id in the Medicine\_Contradicts\_Diseases table

std::string query =

"SELECT c.\"Disease\_id\", dis.\"Disease\_name\", COUNT(\*) AS frequency "

"FROM public.\"Medicine\_Contradicts\_Diseases\" c "

"JOIN public.\"Diseases\" dis ON c.\"Disease\_id\" = dis.\"Disease\_id\" "

"GROUP BY c.\"Disease\_id\", dis.\"Disease\_name\" "

"ORDER BY frequency DESC "

"LIMIT 1;";

pqxx::result result = txn.exec(query);

if (result.empty()) {

std::cout << "No diseases found in the Medicine\_Contradicts\_Diseases table." << std::endl;

return;

}

// Extract the most common Disease\_id and its frequency

int disease\_id = result[0][0].as<int>(); // Disease\_id

std::string disease\_name = result[0][1].c\_str(); // Disease\_name

int frequency = result[0][2].as<int>(); // frequency

// Output the most common contradicted disease

std::cout << "Most common contradicted disease:" << std::endl;

std::cout << "- Disease ID: " << disease\_id << std::endl;

std::cout << "- Disease Name: " << disease\_name << std::endl;

std::cout << "- Occurrences: " << frequency << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error retrieving most common contradicted disease: " << e.what() << std::endl;

}

}

};

**VIEW.cpp**

#include <iostream>

#include <locale>

#include <pqxx/pqxx> // Library for PostgreSQL operations

#include <string>

#include <cstdlib>

#include <ctime>

#pragma once

class View {

public:

View() {}

~View() = default;

void showMessage(const std::string& message) {

std::cout << message << std::endl;

}

void showError(const std::string& errorMessage) {

std::cerr << "Error: " << errorMessage << std::endl;

}

void showMenu()

{

std::cerr << "\nChoose opitions:\n " << std::endl;

std::cerr << "1 Add to tables " << std::endl;

std::cerr << "2 Update tables " << std::endl;

std::cerr << "3 Delete from tables " << std::endl;

std::cerr << "4 Generate Data " << std::endl;

std::cerr << "5 Exit program " << std::endl;

std::cerr << "6 Out All tables names " << std::endl;

std::cerr << "7 Out table col names " << std::endl;

std::cerr << "8 Out most common man " << std::endl;

std::cerr << "9 Out most Treated desiease " << std::endl;

std::cerr << "10 Out most Contradicted desease desiease " << std::endl;

}

void askTableName()

{

std::cerr << "\nEnter valid table name: " << std::endl;

}

void askColName(const std::string& message)

{

std::cerr << "\nEnter valid coulumn name in table " << message << ":" << std::endl;

}

};

**CONTRL.cpp**

#pragma once

#include <iostream>

#include "MODL.cpp"

#include "VIEW.cpp"

#include <locale>

#include <pqxx/pqxx> // Library for PostgreSQL operations

#include <string>

#include <cstdlib>

#include <ctime>

class Controller {

private:

Model\* model;

View\* view;

bool isRunning;

void addData() {

int choice;

view->showMessage("Choose table to add data:");

view->showMessage("1 - Medicine, 2 - Spec, 3 - Category, 4 - Disease, 5 - Manufacturer");

std::cin >> choice;

if (choice == 1) {

int medicine\_id, manufacturer\_id, specs\_id, category\_id, treats\_id, contradicts\_id;

std::string medicine\_name;

view->showMessage("Enter Medicine ID:");

std::cin >> medicine\_id;

view->showMessage("Enter Manufacturer ID:");

std::cin >> manufacturer\_id;

view->showMessage("Enter Specs ID:");

std::cin >> specs\_id;

view->showMessage("Enter Category ID:");

std::cin >> category\_id;

view->showMessage("Enter Treats ID:");

std::cin >> treats\_id;

view->showMessage("Enter Contradicts ID:");

std::cin >> contradicts\_id;

view->showMessage("Enter Medicine Name:");

std::cin >> medicine\_name;

model->createMedicine(medicine\_id, manufacturer\_id, specs\_id, category\_id, treats\_id, contradicts\_id, medicine\_name);

}

else if (choice == 2) {

int specs\_id;

std::string license\_name, medicine\_form;

bool term\_of\_issue;

view->showMessage("Enter Specs ID:");

std::cin >> specs\_id;

view->showMessage("Enter License Name:");

std::cin >> license\_name;

view->showMessage("Enter Term of Issue (1 for true, 0 for false):");

std::cin >> term\_of\_issue;

view->showMessage("Enter Medicine Form:");

std::cin >> medicine\_form;

model->createSpec(specs\_id, license\_name, term\_of\_issue, medicine\_form);

}

else if (choice == 3) {

int category\_id;

std::string category\_name;

view->showMessage("Enter Category ID:");

std::cin >> category\_id;

view->showMessage("Enter Category Name:");

std::cin >> category\_name;

model->createCategory(category\_id, category\_name);

}

else if (choice == 4) {

int disease\_id;

std::string disease\_name;

view->showMessage("Enter Disease ID:");

std::cin >> disease\_id;

view->showMessage("Enter Disease Name:");

std::cin >> disease\_name;

model->createDisease(disease\_id, disease\_name);

}

else if (choice == 5) {

int manufacturer\_id;

std::string manufacturer\_name, location;

view->showMessage("Enter Manufacturer ID:");

std::cin >> manufacturer\_id;

view->showMessage("Enter Manufacturer Name:");

std::cin >> manufacturer\_name;

view->showMessage("Enter Location of Facilities:");

std::cin >> location;

model->createManufacturer(manufacturer\_id, manufacturer\_name, location);

}

else {

view->showError("Invalid option. Please try again.");

}

}

void updateData() {

int choice;

view->showMessage("Choose table to update data:");

view->showMessage("1 - Medicine, 2 - Spec, 3 - Category, 4 - Disease, 5 - Manufacturer");

std::cin >> choice;

if (choice == 1) {

int medicine\_id;

std::string column\_name, new\_value;

view->showMessage("Enter Medicine ID:");

std::cin >> medicine\_id;

view->showMessage("Enter column name to update (e.g., Manufacturer\_id, Specs\_id, Category\_id, Treats\_id, Contradicts\_id, Medicine\_name):");

std::cin >> column\_name;

view->showMessage("Enter new value:");

std::cin >> new\_value;

model->updateMedicine(medicine\_id, column\_name, new\_value);

}

else if (choice == 2) {

int specs\_id;

std::string column\_name, new\_value;

view->showMessage("Enter Specs ID:");

std::cin >> specs\_id;

view->showMessage("Enter column name to update (e.g., License\_name, Term\_of\_issue, Medicine\_form):");

std::cin >> column\_name;

view->showMessage("Enter new value:");

std::cin >> new\_value;

model->updateSpec(specs\_id, column\_name, new\_value);

}

else if (choice == 3) {

int category\_id;

std::string new\_value;

view->showMessage("Enter Category ID:");

std::cin >> category\_id;

view->showMessage("Enter new Category Name:");

std::cin >> new\_value;

model->updateCategory(category\_id, new\_value);

}

else if (choice == 4) {

int disease\_id;

std::string new\_value;

view->showMessage("Enter Disease ID:");

std::cin >> disease\_id;

view->showMessage("Enter new Disease Name:");

std::cin >> new\_value;

model->updateDisease(disease\_id, new\_value);

}

else if (choice == 5) {

int manufacturer\_id;

std::string column\_name, new\_value;

view->showMessage("Enter Manufacturer ID:");

std::cin >> manufacturer\_id;

view->showMessage("Enter column name to update (e.g., Manufacturer\_name, Location of facilities):");

std::cin >> column\_name;

view->showMessage("Enter new value:");

std::cin >> new\_value;

model->updateManufacturer(manufacturer\_id, column\_name, new\_value);

}

else {

view->showError("Invalid option. Please try again.");

}

}

void deleteData() {

int choice;

view->showMessage("Choose table to delete data:");

view->showMessage("1 - Medicine, 2 - Spec, 3 - Category, 4 - Disease, 5 - Manufacturer");

std::cin >> choice;

if (choice == 1) {

int medicine\_id;

view->showMessage("Enter Medicine ID to delete:");

std::cin >> medicine\_id;

model->deleteMedicine(medicine\_id);

}

else if (choice == 2) {

int specs\_id;

view->showMessage("Enter Specs ID to delete:");

std::cin >> specs\_id;

model->deleteSpec(specs\_id);

}

else if (choice == 3) {

int category\_id;

view->showMessage("Enter Category ID to delete:");

std::cin >> category\_id;

model->deleteCategory(category\_id);

}

else if (choice == 4) {

int disease\_id;

view->showMessage("Enter Disease ID to delete:");

std::cin >> disease\_id;

model->deleteDisease(disease\_id);

}

else if (choice == 5) {

int manufacturer\_id;

view->showMessage("Enter Manufacturer ID to delete:");

std::cin >> manufacturer\_id;

model->deleteManufacturer(manufacturer\_id);

}

else {

view->showError("Invalid option. Please try again.");

}

}

void generateData() {

view->showMessage("Choose table to generate data:");

view->showMessage("1 - Medicine, 2 - Spec, 3 - Category, 4 - Disease, 5 - Manufacturer");

int choice;

std::cin >> choice;

view->showMessage("Choosе number of writes:");

int count;

std::cin >> count;

model->generateData(count, choice);

}

void outTableNames() {

view->showMessage("Names of all tables:");

model->outputAllTableNames();

}

void outTableColNames() {

view->showMessage("Select table:");

std::string table\_name;

std::cin >> table\_name;

model->outputTableColumns(table\_name);

}

void findMostPopularMan() {

view->showMessage("Most common manufacurer:");

model->outputMostCommonManufacturer();

}

void findMostTreated() {

model->outputMostCommonTreat();

}

void findMostContradicted()

{

model->outputMostCommonContradicted();

}

public:

Controller(Model\* m, View\* v) : model(m), view(v), isRunning(true) {};

void run() {

while (isRunning) {

view->showMenu();

int choice;

std::cin >> choice;

switch (choice) {

case 1:

addData();

break;

case 2:

updateData();

break;

case 3:

deleteData();

break;

case 4:

generateData();

break;

case 5:

isRunning = false;

break;

case 6:

outTableNames();

break;

case 7:

outTableColNames();

break;

case 8:

findMostPopularMan();

break;

case 9:

findMostContradicted();

break;

case 10:

findMostTreated();

break;

default:

view->showError("Invalid option. Please try again.");

break;

}

}

}

};

**Результати роботи**

Структура меню

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Кожен з пунктів виконує зазначену в ньому дію

**Вставка даних:**

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, линия, Шрифт, число

Автоматически созданное описаниеТакож важливо продемострувати правильність додавання в джанкшн таблиці Изображение выглядит как снимок экрана, текст, линия, Шрифт

Автоматически созданное описание Изображение выглядит как текст, линия, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание

Редагування данних

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описаниеТакож демострую зміни у відповідній джанкш табличці Изображение выглядит как снимок экрана, линия, Прямоугольник, Параллельный

Автоматически созданное описание

Видалення данних

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание  
Видалимо Specs з ID 84

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание Изображение выглядит как текст, Шрифт, число, линия

Автоматически созданное описание

У випадку якщо ми спробуємо видалити звязані данні сервер поверне помилку та скаже юзеру що він зробив не так

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Генерування даних:

Найцікавіший пункт усі незалежні данні генеруються шляхом псевдо випадкового генрування запитами сікюель , а от залежна від усіх табличка Medicines генеруеться на основі вже наявних в підлеглих їй таблицях данних

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт

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

Автоматически созданное описание

Изображение выглядит как текст, число, Шрифт, программное обеспечение

Автоматически созданное описаниеИзображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание

Пошук данних та виведення інформації про БД

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание