



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування та спеціалізованих комп'ютерних
систем**

Розрахунково-графічна робота

з дисципліни **Бази даних і засоби управління**

*на тему: “Створення додатку бази даних, орієнтованого на взаємодію з СУБД
PostgreSQL”*

Виконав:

студент III курсу

групи КВ-22

Ковкін В. В.

Перевірів:

Павловський В. І.

Мета: здобуття вмінь програмування прикладних додатків баз даних PostgreSQL.

Виконання роботи

Нижче наведені сутності предметної області «Відвідуваність занять студентами» та зв'язки між ними.

Сутності предметної області

Для побудови бази даних обраної області, були виділені такі сутності:

1. Громадянин (Citizen)

Атрибути: ідентифікатор громадянина, ім'я, номер телефону, адреса.
Призначення: збереження даних щодо громадян, які потребують вакцинації.

2. Вакцинація (Vaccination)

Атрибути: ідентифікатор вакцинації, ідентифікатор громадянина, ідентифікатор лікаря, ідентифікатор вакцини, ідентифікатор медичного закладу, дата вакцинації.
Призначення: збереження даних про кожну здійснену вакцинацію.

3. Вакцина (Vaccine)

Атрибути: ідентифікатор вакцини, кількість необхідних доз.
Призначення: збереження інформації про вакцини.

4. Лікар (Doctor)

Атрибути: ідентифікатор лікаря, ім'я, номер телефону.
Призначення: збереження даних щодо лікарів, які здійснюють вакцинацію.

5. Медичний заклад (Clinic)

Атрибути: ідентифікатор медичного закладу, адреса.
Призначення: збереження даних про медичні заклади, в яких працюють лікарі.

6. Лікарня Лікар (Doctor_Clinic)

Атрибути: ідентифікатор відповідності, ідентифікатор лікаря, ідентифікатор медичного закладу.

Призначення: збереження інформації про відповідність лікаря та медичного закладу.

7. Вакцина-Медичний заклад (Vaccine_Clinic)

Атрибути: ідентифікатор таблиці, ідентифікатор вакцини, ідентифікатор медичного закладу.

Призначення: збереження інформації про наявність вакцин у медичних закладах.

Зв'язки між сутностями предметної області

Один медичний заклад може мати багато лікарів і один лікар може працювати в різних медичних закладах. Тому між сутностями Медичний заклад і Лікар існує зв'язок М:Н.

В одному медичному закладі може бути доступно багато вакцин і також вакцини можуть зберігатися у різних медичних закладах. Тому між сутностями Медичний заклад і Вакцина існує зв'язок М:Н.

Для вакцинації кожен раз використовується один громадянин, одна вакцина, один лікар і один медичний заклад. Для цього зв'язку додана сутність **Вакцинація**. **Вакцинація** це сутність-зв'язок 4-х сутностей **Громадянин** – **Вакцина** – **Лікар** – **Лікарня**.

Графічне подання концептуальної моделі зображено на рисунку 1.

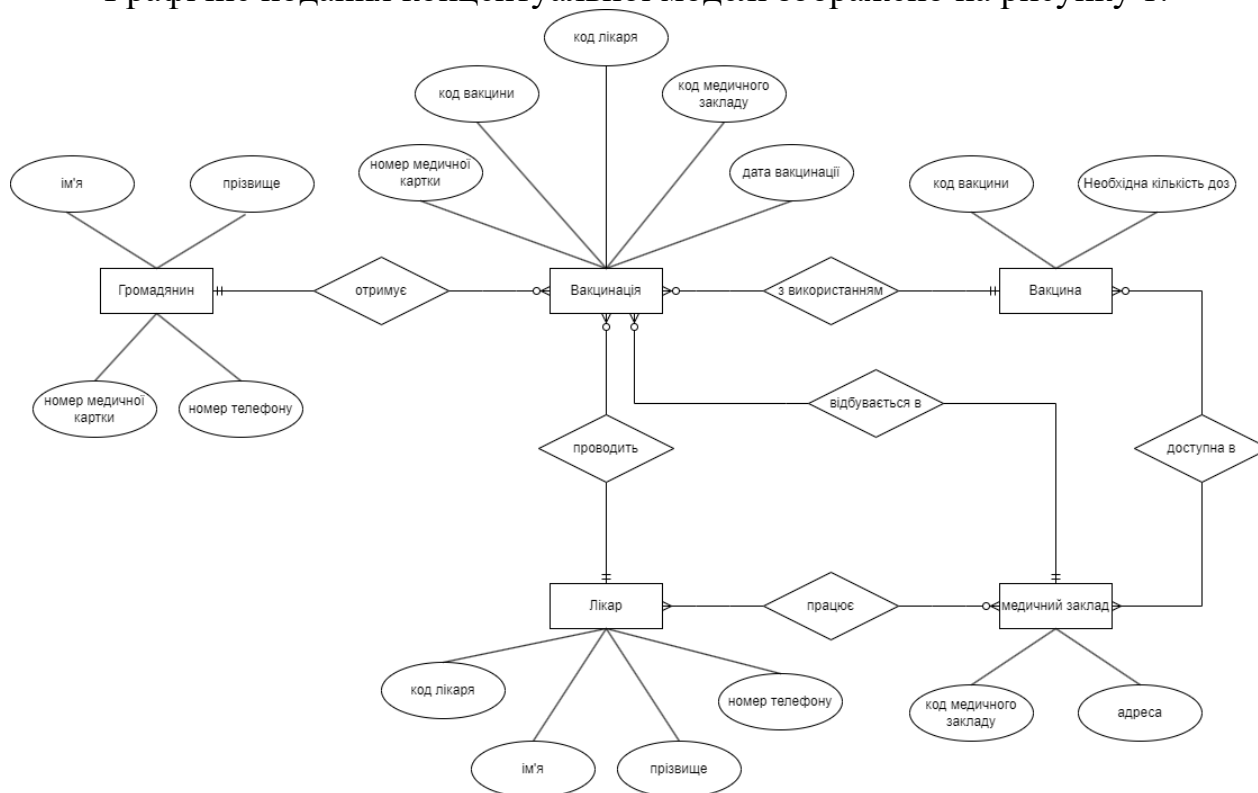


Рисунок 1 – ER-діаграма, побудована за нотацією Чена

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

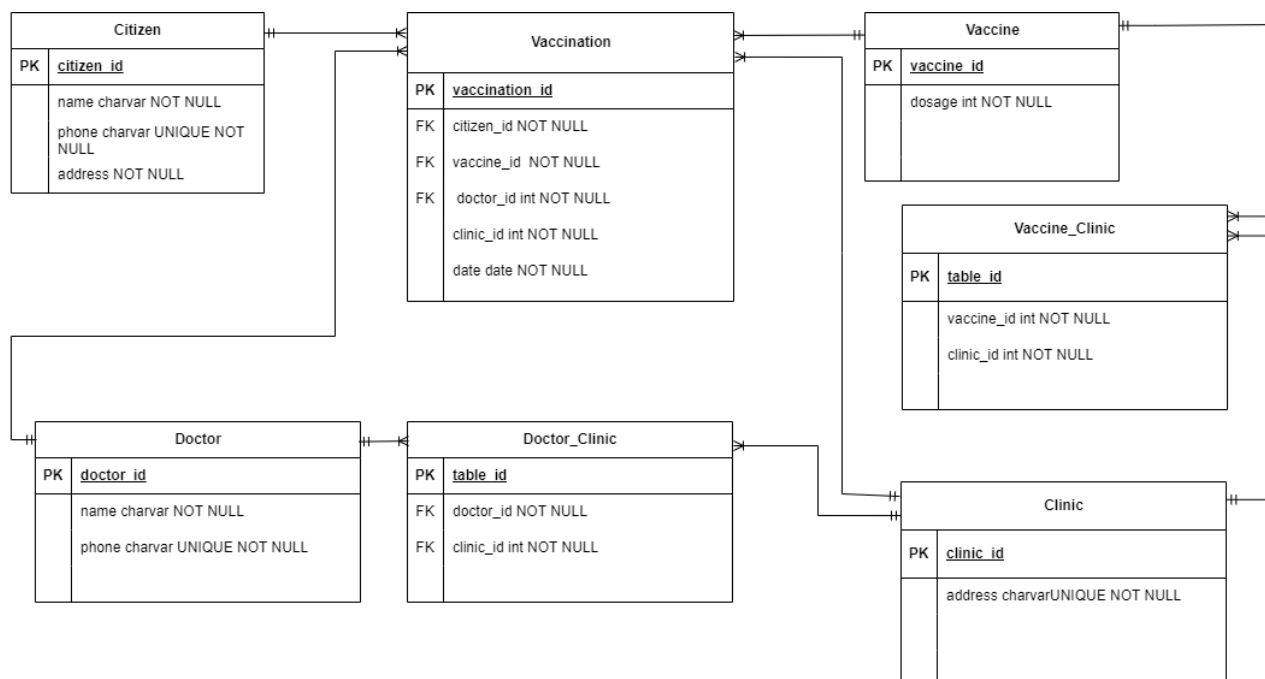


Рисунок 2 – Логічна модель

Середовище та компоненти розробки

У процесі розробки була використана мова програмування Python, інтегроване середовище розробки PyCharm, а також була використана бібліотека `psycopg2`, яка надає API для взаємодії з базою даних PostgreSQL.

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 та її опис

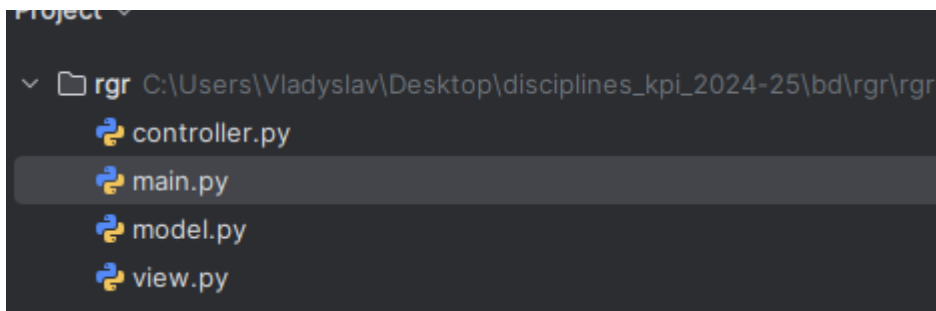


Рисунок 3 – Структура програми

З файлу `main.py` відбувається виклик контролера та передача йому управління.

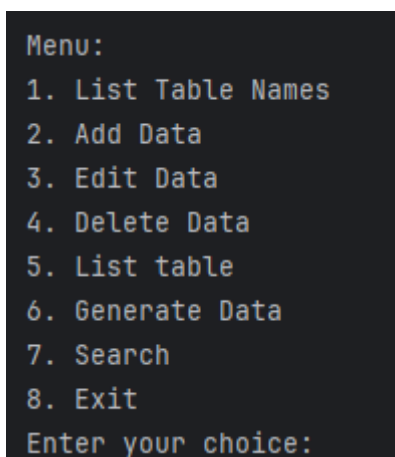
У файлі `model.py` описаний клас моделі, який відповідає за управління підключенням до бази даних і виконанням низькорівневих запитів до неї.

У файлі `controller.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.py` 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 MVC.

Схему меню користувача з описом функціональності кожного пункту



List Table Names

Опис: Виводить список усіх таблиць, доступних у базі даних. Цей пункт дозволяє переглядати назви таблиць, що містяться у базі.

Add Data

Опис: Дозволяє користувачу додавати нові записи в таблицю. Користувач вводить назву таблиці, стовпці та значення, і програма виконує вставку.

Edit Data

Опис: Дозволяє редагувати наявні записи в таблиці. Користувач вказує таблицю, ідентифікатор рядка, стовпці, які потрібно змінити, і нові значення.

Delete Data

Опис: Дає можливість видалити записи з таблиці. Користувач вводить назву таблиці та ідентифікатор рядка, який потрібно видалити.

List Table

Опис: Виводить кілька перших рядків з обраної таблиці. Користувач вводить назву таблиці та кількість рядків, які потрібно показати.

Generate Data

Опис: Генерує випадкові дані для вибраної таблиці. Користувач вводить кількість рядків для генерації, і програма автоматично додає ці рядки у таблицю.

Search

Опис: Дозволяє виконувати пошук у базі даних за різними критеріями. Пошук можна здійснювати за датами, іменами, діапазонами чисел і т.д.

Exit

Опис: Завершує роботу програми.

Фрагмент коду (файл controller.py), в якому наведено головний цикл роботи програми

```
def run(self):
    while True:
        choice = self.show_menu()
        if choice == '1':
            self.list_tables_names()
        elif choice == '2':
            self.add_data()
```

```

elif choice == '3':
    self.edit_data()
elif choice == '4':
    self.delete_data()
elif choice == '5':
    self.list_table()
elif choice == '6':
    self.generate_data()
elif choice == '7':
    self.search_menu()
elif choice == '8':
    break

```

Фрагмент коду (файл model.h), в якому наведено функції перегляду, внесення, редагування, вилучення та генерації у базі даних:

Функція перегляду даних:

```

def get_listed_table(self, table_name, n_rows):
    try:
        query = f"SELECT * FROM {table_name} LIMIT %s"
        cursor = self.conn.cursor()
        cursor.execute(query, (n_rows,))

        column_names = [desc[0] for desc in cursor.description]
        rows = cursor.fetchall()

        result = [column_names] + rows
        return result, 0
    except psycopg2.Error as e:
        return f"Error: {e.pgerror}", 1
    finally:
        cursor.close()

```

Функція `get_listed_table` отримує список перших `n_rows` записів з таблиці. Вона виконує SQL-запит `SELECT *` і обмежує кількість рядків за допомогою `LIMIT`. Функція повертає назви стовпців і вибрані рядки. У разі помилки повертається повідомлення про помилку.

Параметри:

- **table_name** (str): Назва таблиці.
- **n_rows** (int): Кількість рядків, що потрібно вибрати з таблиці.

Функція внесення даних:

```

def add_data(self, table, columns, values):
    try:
        cursor = self.conn.cursor()
        columns_str = ", ".join(columns)
        placeholders = ", ".join(["%s"] * len(values))
        query = f"INSERT INTO {table} ({columns_str}) VALUES ({placeholders})"
        cursor.execute(query, values)
        self.conn.commit()
        return "Data added successfully!"
    except psycopg2.Error as e:
        self.conn.rollback()
        return e.pgerror

```

```
finally:
    cursor.close()
```

Функція `add_data` додає новий запис до таблиці в базі даних. Вона приймає назву таблиці, список стовпців і відповідні значення для вставки. Функція генерує SQL-запит для вставки даних і виконує його. Після успішної вставки даних змінення зберігаються в базі даних. У разі помилки транзакція відкочується, і повертається повідомлення про помилку.

Параметри:

- **table** (str): Назва таблиці.
- **columns** (list of str): Список стовпців.
- **values** (list): Список значень для вставки.

Функція оновлення даних:

```
def edit_data(self, table, id, columns, new_values):
    try:
        cursor = self.conn.cursor()
        set_clause = ", ".join([f"{col} = %s" for col in columns])
        primary_keys = self.get_primary_key_columns(table)
        query = f"UPDATE {table} SET {set_clause} WHERE {primary_keys[0]} = %s"
        values = new_values + [id]
        cursor.execute(query, values)
        self.conn.commit()
        return "Data updated successfully!"
    except psycopg2.Error as e:
        self.conn.rollback()
        return e.pgerror
    finally:
        cursor.close()
```

Функція `edit_data` оновлює існуючі дані в таблиці бази даних на основі заданих стовпців та значень. Вона приймає назву таблиці, ідентифікатор запису, список стовпців та нові значення, після чого формує SQL-запит для оновлення та виконує його. Якщо запит успішно виконано, зміни зберігаються в базі даних. У разі помилки транзакція відкочується, і повертається повідомлення про помилку.

Параметри:

- **table** (str): Назва таблиці.
- **id** (int): Ідентифікатор запису, що потребує редагування.
- **columns** (list of str): Список стовпців для оновлення.
- **new_values** (list): Список нових значень для стовпців.

Функція видалення значень:

```
def delete_data(self, table, id):
    try:
        cursor = self.conn.cursor()
        primary_keys = self.get_primary_key_columns(table)
        query = f"DELETE FROM {table} WHERE {primary_keys[0]} = %s"
        cursor.execute(query, (id,))
        self.conn.commit()
```



```

        return "Data deleted successfully!"
    except psycopg2.Error as e:
        self.conn.rollback()
        return e.pgerror
    finally:
        cursor.close()

```

Функція `delete_data` видаляє запис із зазначеної таблиці на основі його ідентифікатора. Вона формує SQL-запит для видалення даних та виконує його. Якщо операція видалення успішна, зміни фіксуються в базі даних. У разі виникнення помилки транзакція відкочується, і повертається повідомлення про помилку.

Параметри:

- **table** (str): Назва таблиці.
- **id** (int): Ідентифікатор запису, який потрібно видалити.

Функції генерування даних:

```

def generate_data(self, table_name, n_rows):
    if table_name not in self.get_tables():
        return f"Error: Table '{table_name}' does not exist!"
    {self.get_tables()}
    try:
        primary_key = self.get_primary_key_columns(table_name)[0]
        with self.conn.cursor() as cursor:
            cursor.execute(f"SELECT COALESCE(MAX({primary_key}), 0) FROM {table_name}")
            last_key = cursor.fetchone()[0]

        table_generators = {
            "vaccine": self.generate_vaccine_data,
            "doctor": self.generate_doctor_data,
            "clinic": self.generate_clinic_data,
            "vaccination": self.generate_vaccination_data,

```

Функція `generate_data` автоматично генерує дані для таблиці, вибираючи відповідний генератор для кожної таблиці. Вона також перевіряє наявність таблиці в базі даних і генерує нові записи, починаючи з максимального значення поточного первинного ключа.

Параметри:

- **table_name** (str): Назва таблиці для генерації даних.
- **n_rows** (int): Кількість рядків для вставки.

```

def generate_vaccine_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO vaccine ({primary_key}, dosage)
                SELECT i, floor(random() * 10 + 1)::integer
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} vaccine records starting from ID

```

```
{last_key + 1}."
    except psycopg2.Error as e:
        self.conn.rollback()
    return f"Error generating vaccine data: {e.pgerror}"
```

Функція `generate_vaccine_data` генерує дані для таблиці `vaccine`, включаючи випадкові значення доз вакцин.

Параметри:

- **last_key** (int): Останнє значення первинного ключа в таблиці.
- **n_rows** (int): Кількість рядків для генерації.
- **primary_key** (str): Назва первинного ключа для таблиці.

```
def generate_doctor_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO doctor ({primary_key}, name, phone)
                SELECT i, 'Doctor_' || i,
                       '+380' || LPAD((floor(random() * 1000000000)::bigint)::text,
2, '0')
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} doctor records starting from ID {last_key
+ 1}."
    except psycopg2.Error as e:
        self.conn.rollback()
    return f"Error generating doctor data: {e.pgerror}"
```

Функція `generate_doctor_data` генерує дані для таблиці `doctor`, створюючи випадкові імена лікарів і їхні телефони.

Параметри:

- **last_key** (int): Останнє значення первинного ключа в таблиці.
- **n_rows** (int): Кількість рядків для генерації.
- **primary_key** (str): Назва первинного ключа для таблиці.

```
def generate_clinic_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO clinic ({primary_key}, address)
                SELECT i, 'Clinic Address ' || i
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} clinic records starting from ID {last_key
+ 1}."
    except psycopg2.Error as e:
        self.conn.rollback()
    return f"Error generating clinic data: {e.pgerror}"
```

Функція `generate_clinic_data` генерує дані для таблиці `clinic`, створюючи випадкові адреси клінік.

Параметри:

- **last_key** (int): Останнє значення первинного ключа в таблиці.

- **n_rows** (int): Кількість рядків для генерації.
- **primary_key** (str): Назва первинного ключа для таблиці.

```
def generate_citizen_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO citizen ({primary_key}, name, address, phone)
                SELECT i, 'Citizen_' || i, 'City, Street ' || i,
                '380' || LPAD((floor(random() * 1000000000)::bigint)::text,
9, '0')
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} citizen records starting from ID
{last_key + 1}."
        except psycopg2.Error as e:
            self.conn.rollback()
            return f"Error generating citizen data: {e.pgerror}"
```

Функція `generate_citizen_data` генерує дані для таблиці `citizen`, створюючи випадкові імена громадян, адреси та телефони.

Параметри:

- **last_key** (int): Останнє значення первинного ключа в таблиці.
- **n_rows** (int): Кількість рядків для генерації.
- **primary_key** (str): Назва первинного ключа для таблиці.

```
def generate_doctor_clinic_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO doctor_clinic ({primary_key}, doctor_id, clinic_id)
                SELECT i,
                (SELECT doctor_id FROM doctor ORDER BY random() LIMIT 1),
                (SELECT clinic_id FROM clinic ORDER BY random() LIMIT 1)
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} doctor_clinic records starting from ID
{last_key + 1}."
        except psycopg2.Error as e:
            self.conn.rollback()
            return f"Error generating doctor_clinic data: {e.pgerror}"
```

Функція `generate_doctor_clinic_data` генерує зв'язки між лікарями та клініками для таблиці `doctor_clinic`.

Параметри:

- **last_key** (int): Останнє значення первинного ключа в таблиці.
- **n_rows** (int): Кількість рядків для генерації.
- **primary_key** (str): Назва первинного ключа для таблиці.

```
def generate_vaccination_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO vaccination ({primary_key}, citizen_id, doctor_id,
vaccine_id, clinic_id, date)
                SELECT i,
                    (SELECT citizen_id FROM citizen ORDER BY random() LIMIT 1),
                    (SELECT doctor_id FROM doctor ORDER BY random() LIMIT 1),
                    (SELECT vaccine_id FROM vaccine ORDER BY random() LIMIT 1),
                    (SELECT clinic_id FROM clinic ORDER BY random() LIMIT 1),
                    NOW() - (floor(random() * 365)::int * INTERVAL '1 day')
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} vaccination records starting from ID
{last_key + 1}."
        except psycopg2.Error as e:
            self.conn.rollback()
            return f"Error generating vaccination data: {e.pgerror}"
```

Функція `generate_vaccination_data` генерує дані для таблиці `vaccination`, зв'язуючи громадян, лікарів, вакцини та клініки.

Параметри:

- **last_key** (int): Останнє значення первинного ключа в таблиці.
- **n_rows** (int): Кількість рядків для генерації.
- **primary_key** (str): Назва первинного ключа для таблиці.

```
def generate_vaccine_clinic_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO vaccine_clinic ({primary_key}, vaccine_id, clinic_id)
                SELECT i,
                    (SELECT vaccine_id FROM vaccine ORDER BY random() LIMIT 1),
                    (SELECT clinic_id FROM clinic ORDER BY random() LIMIT 1)
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} vaccine_clinic records starting from ID {last_key
+ 1}."
        except psycopg2.Error as e:
            self.conn.rollback()
            return f"Error generating vaccine_clinic data: {e.pgerror}"
```

Функція `generate_vaccine_clinic_data` генерує зв'язки між вакцинами та клініками для таблиці `vaccine_clinic`.

Параметри:

- **last_key** (int): Останнє значення первинного ключа в таблиці.
- **n_rows** (int): Кількість рядків для генерації.
- **primary_key** (str): Назва первинного ключа для таблиці.

Повний код програми

Файл main.py:

```
from controller import Controller

if __name__ == "__main__":
    controller = Controller()
    controller.run()
```

Файл model.py:

```
import psycopg2
from time import time

class Model:
    def __init__(self):
        self.conn = psycopg2.connect(
            dbname = "vaccination_control",
            user = "postgres",
            password = "2534",
            host = "localhost",
            port = 5432
        )

    def get_tables(self):
        try:
            cursor = self.conn.cursor()
            cursor.execute("""SELECT table_name FROM information_schema.tables
                               WHERE table_schema = 'public'""")
            return [table[0] for table in cursor.fetchall()]
        except psycopg2.Error as e:
            return f"Error fetching tables: {e.pgerror}"

    def edit_data(self, table, id, columns, new_values):
        try:
            cursor = self.conn.cursor()
            set_clause = ", ".join([f"{col} = %s" for col in columns])
            primary_keys = self.get_primary_key_columns(table)
            query = f"UPDATE {table} SET {set_clause} WHERE {primary_keys[0]} = %s"
            values = new_values + [id]
            cursor.execute(query, values)
            self.conn.commit()
            return "Data updated successfully!"
        except psycopg2.Error as e:
            self.conn.rollback()
            return e.pgerror
        finally:
            cursor.close()

    def add_data(self, table, columns, values):
        try:
            cursor = self.conn.cursor()
            columns_str = ", ".join(columns)
            placeholders = ", ".join(["%s"] * len(values))
            query = f"INSERT INTO {table} ({columns_str}) VALUES ({placeholders})"
            cursor.execute(query, values)
            self.conn.commit()
            return "Data added successfully!"
        except psycopg2.Error as e:
            self.conn.rollback()
            return e.pgerror
        finally:
            cursor.close()
```

```

def delete_data(self, table, id):
    try:
        cursor = self.conn.cursor()
        primary_keys = self.get_primary_key_columns(table)
        query = f"DELETE FROM {table} WHERE {primary_keys[0]} = %s"
        cursor.execute(query, (id,))
        self.conn.commit()
        return "Data deleted successfully!"
    except psycopg2.Error as e:
        self.conn.rollback()
        return e.pgerror
    finally:
        cursor.close()

def get_columns(self, table_name):
    query = """
    SELECT column_name
    FROM information_schema.columns
    WHERE table_schema = 'public' AND table_name = %s;
    """
    try:
        with self.conn.cursor() as cursor:
            cursor.execute(query, (table_name,))
            columns = [row[0] for row in cursor.fetchall()]
            return columns
    except Exception as e:
        self.conn.rollback()
        print(f"Error retrieving columns for table {table_name}: {e}")
        return []
    finally:
        cursor.close()

def get_primary_key_columns(self, table_name):
    try:
        query = """
        SELECT a.attname
        FROM pg_index i
        JOIN pg_attribute a ON a.attrelid = i.indrelid AND a.attnum =
ANY(i.indkey)
        WHERE i.indrelid = %s::regclass AND i.indisprimary;
        """
        cursor = self.conn.cursor()
        cursor.execute(query, (table_name,))
        result = cursor.fetchall()
        return [row[0] for row in result]
    except psycopg2.Error as e:
        self.conn.rollback()
        return f"Error: {e.pgerror}"
    finally:
        cursor.close()

def get_listed_table(self, table_name, n_rows):
    try:
        query = f"SELECT * FROM {table_name} LIMIT %s"
        cursor = self.conn.cursor()
        cursor.execute(query, (n_rows,))

        column_names = [desc[0] for desc in cursor.description]
        rows = cursor.fetchall()

        result = [column_names] + rows
        return result, 0
    except psycopg2.Error as e:
        return f"Error: {e.pgerror}", 1
    finally:
        cursor.close()

def generate_data(self, table_name, n_rows):
    if table_name not in self.get_tables():

```

```

        return f"Error: Table '{table_name}' does not exist! {self.get_tables()}"
    try:
        primary_key = self.get_primary_key_columns(table_name)[0]
        with self.conn.cursor() as cursor:
            cursor.execute(f"SELECT COALESCE(MAX({primary_key}), 0) FROM
{table_name}")
            last_key = cursor.fetchone()[0]

        table_generators = {
            "vaccine": self.generate_vaccine_data,
            "doctor": self.generate_doctor_data,
            "clinic": self.generate_clinic_data,
            "vaccination": self.generate_vaccination_data,
            "doctor_clinic": self.generate_doctor_clinic_data,
            "citizen": self.generate_citizen_data
        }

        if table_name not in table_generators:
            return f"Error: No data generator defined for table '{table_name}'"

        return table_generators[table_name](int(last_key), int(n_rows), primary_key)

    except psycopg2.Error as e:
        self.conn.rollback()
        return f"Database error: {e.pgerror}"

    except Exception as e:
        return f"Unexpected error: {str(e)}"

def generate_vaccine_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO vaccine ({primary_key}, dosage)
                SELECT i, floor(random() * 10 + 1)::integer
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} vaccine records starting from ID {last_key +
1}."

    except psycopg2.Error as e:
        self.conn.rollback()
        return f"Error generating vaccine data: {e.pgerror}"

def generate_doctor_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO doctor ({primary_key}, name, phone)
                SELECT i, 'Doctor_' || i,
                '+380' || LPAD((floor(random() * 1000000000)::bigint)::text, 9,
'0')
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} doctor records starting from ID {last_key +
1}."

    except psycopg2.Error as e:
        self.conn.rollback()
        return f"Error generating doctor data: {e.pgerror}"

def generate_clinic_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO clinic ({primary_key}, address)
                SELECT i, 'Clinic Address ' || i
                FROM generate_series(%s, %s) AS i
            """

```

```

        cursor.execute(query, (last_key + 1, last_key + n_rows))
        self.conn.commit()
        return f"Inserted {n_rows} clinic records starting from ID {last_key +
1}."
    except psycopg2.Error as e:
        self.conn.rollback()
        return f"Error generating clinic data: {e.pgerror}"

def generate_citizen_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO citizen ({primary_key}, name, address, phone)
                SELECT i, 'Citizen_' || i, 'City, Street ' || i,
                    '+380' || LPAD((floor(random() * 1000000000)::bigint)::text, 9,
'0')
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} citizen records starting from ID {last_key +
1}."
    except psycopg2.Error as e:
        self.conn.rollback()
        return f"Error generating citizen data: {e.pgerror}"

def generate_doctor_clinic_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO doctor_clinic ({primary_key}, doctor_id, clinic_id)
                SELECT i,
                    (SELECT doctor_id FROM doctor ORDER BY random() LIMIT 1),
                    (SELECT clinic_id FROM clinic ORDER BY random() LIMIT 1)
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} doctor_clinic records starting from ID
{last_key + 1}."
    except psycopg2.Error as e:
        self.conn.rollback()
        return f"Error generating doctor_clinic data: {e.pgerror}"

def generate_vaccination_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO vaccination ({primary_key}, citizen_id, doctor_id,
vaccine_id, clinic_id, date)
                SELECT i,
                    (SELECT citizen_id FROM citizen ORDER BY random() LIMIT 1),
                    (SELECT doctor_id FROM doctor ORDER BY random() LIMIT 1),
                    (SELECT vaccine_id FROM vaccine ORDER BY random() LIMIT 1),
                    (SELECT clinic_id FROM clinic ORDER BY random() LIMIT 1),
                    NOW() - (floor(random() * 365)::int * INTERVAL '1 day')
                FROM generate_series(%s, %s) AS i
            """
            cursor.execute(query, (last_key + 1, last_key + n_rows))
            self.conn.commit()
            return f"Inserted {n_rows} vaccination records starting from ID {last_key
+ 1}."
    except psycopg2.Error as e:
        self.conn.rollback()
        return f"Error generating vaccination data: {e.pgerror}"

def generate_vaccine_clinic_data(self, last_key, n_rows, primary_key):
    try:
        with self.conn.cursor() as cursor:
            query = f"""
                INSERT INTO vaccine_clinic ({primary_key}, vaccine_id, clinic_id)

```



```

        SELECT i,
            (SELECT vaccine_id FROM vaccine ORDER BY random() LIMIT 1),
            (SELECT clinic_id FROM clinic ORDER BY random() LIMIT 1)
        FROM generate_series(%s, %s) AS i
    """
    cursor.execute(query, (last_key + 1, last_key + n_rows))
    self.conn.commit()
    return f"Inserted {n_rows} vaccine_clinic records starting from ID
{last_key + 1}."
except psycopg2.Error as e:
    self.conn.rollback()
    return f"Error generating vaccine_clinic data: {e.pgerror}"

def search_query_1(self, date_start, date_end, citizen_name_pattern):
    """
    Пошук вакцинацій у вказаному діапазоні дат, імені громадянина за шаблоном.
    """
    query = """
    SELECT vaccination.vaccination_id, citizen.name, doctor.name, vaccination.date
    FROM vaccination
    JOIN citizen ON vaccination.citizen_id = citizen.citizen_id
    JOIN doctor ON vaccination.doctor_id = doctor.doctor_id
    WHERE vaccination.date BETWEEN %s AND %s
    AND citizen.name LIKE %s
    GROUP BY vaccination.vaccination_id, citizen.name, doctor.name,
vaccination.date
    ORDER BY vaccination.date;
    """
    try:
        with self.conn.cursor() as cursor:
            start_time = time()
            cursor.execute(query, (date_start, date_end,
f"%{citizen_name_pattern}%"))
            column_names = [desc[0] for desc in cursor.description]
            rows = cursor.fetchall()
            results = [column_names] + rows
            execution_time = (time() - start_time) * 1000
            return results, execution_time
    except psycopg2.Error as e:
        return f"Database error: {e.pgerror}", 0

def search_query_2(self, dosage_range, clinic_address_pattern):
    """
    Пошук клінік з вакциною у зазначеному діапазоні дозувань та адресою за шаблоном.
    """
    query = """
    SELECT clinic.address, vaccine.dosage, COUNT(vaccine_clinic.vaccine_id)
    FROM clinic
    JOIN vaccine_clinic ON clinic.clinic_id = vaccine_clinic.clinic_id
    JOIN vaccine ON vaccine_clinic.vaccine_id = vaccine.vaccine_id
    WHERE vaccine.dosage BETWEEN %s AND %s
    AND clinic.address LIKE %s
    GROUP BY clinic.address, vaccine.dosage
    ORDER BY COUNT(vaccine_clinic.vaccine_id) DESC;
    """
    try:
        with self.conn.cursor() as cursor:
            start_time = time()
            cursor.execute(query, (dosage_range[0], dosage_range[1],
f"%{clinic_address_pattern}%"))
            column_names = [desc[0] for desc in cursor.description]
            rows = cursor.fetchall()
            results = [column_names] + rows
            execution_time = (time() - start_time) * 1000
            return results, execution_time
    except psycopg2.Error as e:
        return f"Database error: {e.pgerror}", 0

def search_query_3(self, doctor_name_pattern, max_vaccines):
    """
    Пошук лікарів, які проводили менше зазначеної кількості вакцинацій, і їх клінік.

```

```

"""
query = """
    SELECT doctor.name, clinic.address, COUNT(vaccination.vaccination_id)
    FROM doctor
    JOIN doctor_clinic ON doctor.doctor_id = doctor_clinic.doctor_id
    JOIN clinic ON doctor_clinic.clinic_id = clinic.clinic_id
    LEFT JOIN vaccination ON doctor.doctor_id = vaccination.doctor_id
    WHERE doctor.name LIKE %s
    GROUP BY doctor.name, clinic.address
    HAVING COUNT(vaccination.vaccination_id) < %s
    ORDER BY COUNT(vaccination.vaccination_id) ASC;
"""

try:
    with self.conn.cursor() as cursor:
        start_time = time()
        cursor.execute(query, (f"%{doctor_name_pattern}%", max_vaccines))
        column_names = [desc[0] for desc in cursor.description]
        rows = cursor.fetchall()
        results = [column_names] + rows
        execution_time = (time() - start_time) * 1000
        return results, execution_time
except psycopg2.Error as e:
    return f"Database error: {e.pgerror}", 0

```

view.py

```

class View:
    def show_message(self, message):
        print(message)

    def list_names(self, tables):
        print("Tables names:")
        for table in tables:
            print(table)

    def get_list_table_input(self):
        table_name = input("Enter table name: ")
        n_rows = int(input("Enter number of rows: "))
        return table_name, n_rows

    def get_data_input(self):
        try:
            table = input("Enter table name: ")
            columns = input("Enter column names separated by space: ").split()
            val = input("Enter values separated by space: ").split()
            if len(columns) != len(val):
                raise ValueError("The number of columns should be equal to the
number of values")
        except ValueError as e:
            print((f"ERROR: {e}"))
        return table, columns, val

    def get_update_input(self):
        while True:
            try:
                table = input("Enter Table Name: ")
                id = int(input("Enter ID of row that needs to be edited: "))
                columns = input("Enter column names separated by space:
").split()
                new_values = input("Input new values separated by space:
").split()
                return table, id, columns, new_values
            except ValueError as e:
                print(f"ERROR: {e}")

    def get_delete_input(self):
        try:

```

```

        table = input("Enter Table Name: ")
        id = int(input("Enter ID of row that needs to be deleted: "))
        return table, id
    except ValueError as e:
        print(f"ERROR: {e}")

    def get_generate_data_input(self):
        try:
            table = input("Enter table name: ")
            n_rows = input("Enter number of generated rows: ")
            return table, n_rows
        except ValueError as e:
            print(f"ERROR: {e}")

    def get_search_query_1_input(self):
        print("Search Query 1: Find vaccinations within a specific date range
and filter by citizen name.")
        range_start = input("Enter the start date (YYYY-MM-DD): ")
        range_end = input("Enter the end date (YYYY-MM-DD): ")
        citizen_name = input("Enter part of the citizen's name (for pattern
matching): ")
        return range_start, range_end, citizen_name

    def get_search_query_2_input(self):
        print("Search Query 2: Find clinics with vaccines in a specific dosage
range and filter by clinic address.")
        dosage_start = input("Enter the minimum dosage: ")
        dosage_end = input("Enter the maximum dosage: ")
        clinic_address = input("Enter part of the clinic's address (for pattern
matching): ")
        return (int(dosage_start), int(dosage_end)), clinic_address

    def get_search_query_3_input(self):
        print("Search Query 3: Find doctors with fewer vaccinations than a given
limit and filter by doctor name.")
        doctor_name = input("Enter part of the doctor's name (for pattern
matching): ")
        max_vaccinations = input("Enter the maximum number of vaccinations: ")
        return doctor_name, int(max_vaccinations)

    def show_results(self, results):
        if not results:
            print("No results found.")
            return

        if isinstance(results[0], (tuple, list)):
            column_count = len(results[0])
            column_headers = [f"Column {i + 1}" for i in range(column_count)]
        else:
            column_headers = ["Result"]
        results = [[row] for row in results]
        header_row = " | ".join(column_headers)
        print(header_row)
        print("-" * len(header_row))
        for row in results:
            print(" | ".join(map(str, row)))

```

controller.py

```

from model import Model
from view import View

import sys

class Controller:

```

```

def __init__(self):
    self.view = View()
    try:
        self.model = Model()
        self.view.show_message("Connected successfully")
    except Exception as e:
        self.view.show_message(f"Error occurred: {e}")
        sys.exit(1)

def run(self):
    while True:
        choice = self.show_menu()
        if choice == '1':
            self.list_tables_names()
        elif choice == '2':
            self.add_data()
        elif choice == '3':
            self.edit_data()
        elif choice == '4':
            self.delete_data()
        elif choice == '5':
            self.list_table()
        elif choice == '6':
            self.generate_data()
        elif choice == '7':
            self.search_menu()
        elif choice == '8':
            break

def show_menu(self):
    self.view.show_message("\nMenu:")
    self.view.show_message("1. List Table Names")
    self.view.show_message("2. Add Data")
    self.view.show_message("3. Edit Data")
    self.view.show_message("4. Delete Data")
    self.view.show_message("5. List table")
    self.view.show_message("6. Generate Data")
    self.view.show_message("7. Search")
    self.view.show_message("8. Exit")
    return input("Enter your choice: ")

def list_tables_names(self):
    tables = self.model.get_tables()
    self.view.list_names(tables)

def add_data(self):
    table, columns, val = self.view.get_data_input()
    self.view.show_message(self.model.add_data(table, columns, val))

def edit_data(self):
    table, id, columns, new_value = self.view.get_update_input()
    self.view.show_message(self.model.edit_data(table, id, columns,
new_value))

def delete_data(self):
    table, id = self.view.get_delete_input()
    self.view.show_message(self.model.delete_data(table, id))

def list_table(self):
    table_name, n_rows = self.view.get_list_table_input()
    listed_table, error = self.model.get_listed_table(table_name, n_rows)
    if error == 1:

```

```

        self.view.show_message(listed_table)
    else:
        for row in listed_table:
            self.view.show_message(row)

    def generate_data(self):
        self.view.show_message("\nGenerate Data Menu:")
        table_name = input("Enter Table Name:")
        n_rows = input("Enter number of generated rows:")
        self.view.show_message(self.model.generate_data(table_name, n_rows))

    def search_menu(self):
        self.view.show_message("Search Menu:")
        self.view.show_message("1: Query 1 - Find vaccinations by date range and citizen name.")
        self.view.show_message("2: Query 2 - Find clinics by dosage range and address.")
        self.view.show_message("3: Query 3 - Find doctors by name and vaccination count.")

        choice = input("Enter your choice (1-3): ")
        choice_query = {
            '1': (self.view.get_search_query_1_input,
self.model.search_query_1),
            '2': (self.view.get_search_query_2_input,
self.model.search_query_2),
            '3': (self.view.get_search_query_3_input,
self.model.search_query_3),
        }
        if choice not in choice_query:
            self.view.show_message("Invalid choice. Please enter a valid option.")

        return

        input_func, query_func = choice_query[choice]
        inputs = input_func()
        results, execution_time = query_func(*inputs)

        self.view.show_results(results)
        self.view.show_message(f"Query executed in {execution_time:.2f} ms.")

```

Результати виконання програми

Вставка даних:

До вставки даних:

	citizen_id [PK] integer	name character varying	address character varying	phone character varying
1	1	Jeff	Kyiv, Random Street, 1	+380333333333

```

Enter your choice: 2
Enter table name: citizen
Enter column names separated by space: citizen_id name address phone
Enter values separated by space: 2 Carl street1 +380924823482
Data added successfully!

```

Після вставки даних

	citizen_id [PK] integer	name character varying	address character varying	phone character varying
1	1	Jeff	Kyiv, Random Street, 1	+380333333333
2	2	Carl	street1	+380924823482

Редагування даних:

	clinic_id [PK] integer	address character varying
1	1	Kyiv, Random Clinic str. 1
2	2	Kyiv, Random Clinic str.2

```

Menu:
1. List Table Names
2. Add Data
3. Edit Data
4. Delete Data
5. List table
6. Generate Data
7. Search
8. Exit
Enter your choice: 3
Enter Table Name: clinic
Enter ID of row that needs to be edited: 2
Enter column names separated by space: address
Input new values separated by space: newAddress
Data updated successfully!

```

Data Output			Messages	Notifications
	clinic_id [PK] integer		address character varying	
1	1		Kyiv, Random Clinic str. 1	
2	2		newAddress	










Видалення даних:

Data Output			Messages	Notifications
	doctor_id [PK] integer		name character varying	phone character varying
1	1		Bob	+380123456789
2	2		Jeff	+380111111111
3	4		Brad	+380000000001










```

Menu:
1. List Table Names
2. Add Data
3. Edit Data
4. Delete Data
5. List table
6. Generate Data
7. Search
8. Exit
Enter your choice: 4
Enter Table Name: doctor
Enter ID of row that needs to be deleted: 4
Data deleted successfully!

```

Data Output Messages Notifications			
        			
	doctor_id [PK] integer	name character varying	phone character varying
1	1	Bob	+380123456789
2	2	Jeff	+380111111111

Генерування даних:

Data Output Messages Notifications			
        			
	doctor_id [PK] integer	name character varying	phone character varying
1	1	Bob	+380123456789
2	2	Jeff	+380111111111


```
Menu:
1. List Table Names
2. Add Data
3. Edit Data
4. Delete Data
5. List table
6. Generate Data
7. Search
8. Exit
Enter your choice: 6

Generate Data Menu:
Enter Table Name:doctor
Enter number of generated rows:100000
Inserted 100000 doctor records starting from ID 3.

Menu:
```

Data Output Messages Notifications			
<div> </div>			
	doctor_id [PK] integer	name character varying	phone character varying
1	1	Bob	+380123456789
2	2	Jeff	+380111111111
3	3	Doctor_3	+380590158788
4	4	Doctor_4	+380131780062
5	5	Doctor_5	+380909373230
6	6	Doctor_6	+380649460433
7	7	Doctor_7	+380871098191
8	8	Doctor_8	+380896441694
9	9	Doctor_9	+380942392430
10	10	Doctor_10	+380940809966
11	11	Doctor_11	+380990700996
12	12	Doctor_12	+380511423421
13	13	Doctor_13	+380879396606
14	14	Doctor_14	+380430110497
15	15	Doctor_15	+380238537851
16	16	Doctor_16	+380605482990
17	17	Doctor_17	+380364616295
18	18	Doctor_18	+380359274085
19	19	Doctor_19	+380727554729
20	20	Doctor_20	+380543168424
21	21	Doctor_21	+380486734906
22	22	Doctor_22	+380245228562
23	23	Doctor_23	+380175886629
24	24	Doctor_24	+380036959588
25	25	Doctor_25	+380197673071
26	26	Doctor_26	+380070468715
27	27	Doctor_27	+380559715682

Data OutputMessagesNotifications

	vaccination_id [PK] integer	citizen_id integer	doctor_id integer	vaccine_id integer	clinic_id integer	date date
1	1	1	1	1	1	2024-10-16
2	2	1	2	2	2	2024-10-16

```
Menu:
1. List Table Names
2. Add Data
3. Edit Data
4. Delete Data
5. List table
6. Generate Data
7. Search
8. Exit
Enter your choice: 6

Generate Data Menu:
Enter Table Name:vaccination
Enter number of generated rows:100000
Inserted 100000 vaccination records starting from ID 3.
```

SQL

	vaccination_id [PK] integer	citizen_id integer	doctor_id integer	vaccine_id integer	clinic_id integer	date date	
(3)	1	1	4	15538	4875	2	2024-01-28
ts	2	2	4	15538	4875	2	2024-09-04
ies	3	3	4	15538	4875	2	2024-04-09
	4	4	4	15538	4875	2	2024-02-07
	5	5	4	15538	4875	2	2024-10-19
c	6	6	4	15538	4875	2	2024-03-08
(3)	7	7	4	15538	4875	2	2024-08-21
	8	8	4	15538	4875	2	2024-04-14
id	9	9	4	15538	4875	2	2023-11-26
d	10	10	4	15538	4875	2	2024-03-24
ts	11	11	4	15538	4875	2	2024-05-25
	12	12	4	15538	4875	2	2023-12-15
ies	13	13	4	15538	4875	2	2024-09-05
	14	14	4	15538	4875	2	2024-01-08
	15	15	4	15538	4875	2	2023-12-01
	16	16	4	15538	4875	2	2024-01-29
ic	17	17	4	15538	4875	2	2023-12-18
ns	18	18	4	15538	4875	2	2024-11-20
	19	19	4	15538	4875	2	2024-04-08
	20	20	4	15538	4875	2	2024-07-12
	21	21	4	15538	4875	2	2024-03-14

Перегляд даних:

	citizen_id [PK] integer	name character varying	address character varying	phone character varying
1	1	Jeff	Kyiv, Random Street, 1	+380333333333
2	2	Carl	street1	+380924823482
3	3	Citizen_3	City, Street 3	+380217359698
4	4	Citizen_4	City, Street 4	+380002338777
5	5	Citizen_5	City, Street 5	+380182058616
6	6	Citizen_6	City, Street 6	+380267790043
7	7	Citizen_7	City, Street 7	+380113250227
8	8	Citizen_8	City, Street 8	+380435721380
9	9	Citizen_9	City, Street 9	+380380173954
10	10	Citizen_10	City, Street 10	+380895900548
11	11	Citizen_11	City, Street 11	+380040450059
12	12	Citizen_12	City, Street 12	+380300242612

```

4. Delete Data
5. List table
6. Generate Data
7. Search
8. Exit
Enter your choice: 5
Enter table name: citizen
Enter number of rows: 10
['citizen_id', 'name', 'address', 'phone']
(1, 'Jeff', 'Kyiv, Random Street, 1', '+380333333333')
(2, 'Carl', 'street1', '+380924823482')
(3, 'Citizen_3', 'City, Street 3', '+380217359698')
(4, 'Citizen_4', 'City, Street 4', '+380002338777')
(5, 'Citizen_5', 'City, Street 5', '+380182058616')
(6, 'Citizen_6', 'City, Street 6', '+380267790043')
(7, 'Citizen_7', 'City, Street 7', '+380113250227')
(8, 'Citizen_8', 'City, Street 8', '+380435721380')
(9, 'Citizen_9', 'City, Street 9', '+380380173954')
(10, 'Citizen_10', 'City, Street 10', '+380895900548')

```

Пошуковий запит 1:

```

8. Exit
Enter your choice: 7
Search Menu:
1: Query 1 - Find vaccinations by date range and citizen name.
2: Query 2 - Find clinics by dosage range and address.
3: Query 3 - Find doctors by name and vaccination count.
Enter your choice (1-3): 1
Search Query 1: Find vaccinations within a specific date range and filter by citizen name.
Enter the start date (YYYY-MM-DD): 2023-01-01
Enter the end date (YYYY-MM-DD): 2024-01-01
Enter part of the citizen's name (for pattern matching): Citizen_4
vaccination_id | name | name | date
29485 | Citizen_4 | Doctor_15538 | 2023-11-25
40885 | Citizen_4 | Doctor_15538 | 2023-11-25
11413 | Citizen_4 | Doctor_15538 | 2023-11-25
40273 | Citizen_4 | Doctor_15538 | 2023-11-25
56165 | Citizen_4 | Doctor_15538 | 2023-11-25
73987 | Citizen_4 | Doctor_15538 | 2023-11-25
54170 | Citizen_4 | Doctor_15538 | 2023-11-25
8171 | Citizen_4 | Doctor_15538 | 2023-11-25
16257 | Citizen_4 | Doctor_15538 | 2023-11-25

```

Пошуковий запит 2:

```

Enter your choice: 7
Search Menu:
1: Query 1 - Find vaccinations by date range and citizen name.
2: Query 2 - Find clinics by dosage range and address.
3: Query 3 - Find doctors by name and vaccination count.
Enter your choice (1-3): 2
Search Query 2: Find clinics with vaccines in a specific dosage range and filter by clinic address.
Enter the minimum dosage: 2
Enter the maximum dosage: 5
Enter part of the clinic's address (for pattern matching): str
address | dosage | count
Kyiv, Random Clinic str. 1 | 5 | 1
Query executed in 3.00 ms.

```

Пошуковий запит 3:

```

2: Query 2 - Find clinics by dosage range and address.
3: Query 3 - Find doctors by name and vaccination count.
Enter your choice (1-3): 3
Search Query 3: Find doctors with fewer vaccinations than a given limit and filter by doctor name.
Enter part of the doctor's name (for pattern matching): Doctor
Enter the maximum number of vaccinations: 5
name | address | count
Doctor_2 | Kyiv, Random Clinic str. 1 | 0
Query executed in 2.00 ms.

```

Репозиторій Github: https://github.com/kovkinvladyslav/bd_labs/tree/main/rgr