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

Кафедра системного програмування та спеціалізованих комп'ютерних систем

Лабораторна робота №1

з дисципліни

«Бази даних і засоби управління»

Тема: «***Проектування бази даних та ознайомлення з базовими операціями СУБД PostgreSQL***»

Виконав: студент 3 курсу
ФПМ групи КВ-83
Хаустович Олександр
Перевірив: Павловський В.І.

Київ – 2020

Ознайомлення з базовими операціями СУБД PostgreSQL

Метою роботи є здобуття вмінь програмування прикладних додатків баз даних PostgreSQL.
Загальне завдання роботи полягає у наступному:

1. Реалізувати функції внесення, редагування та вилучення даних у таблицях бази даних, створених у лабораторній роботі №1, засобами консольного інтерфейсу.
2. Передбачити автоматичне пакетне генерування «рандомізованих» даних у базі.
3. Забезпечити реалізацію пошуку за декількома атрибутами з 2-х та більше сутностей одночасно: для числових атрибутів – у рамках діапазону, для рядкових – як шаблон функції LIKE оператора SELECT SQL, для логічного типу – значення True/False, для дат – у рамках діапазону дат.
4. Програмний код виконати згідно шаблону MVC (модель-подання-контролер).

Вимоги до інтерфейсу користувача

1. Використовувати консольний інтерфейс користувача.

Варіант (предметна область): база даних для музичного стрімінгового сервісу.

Звіт

Нормалізована логічна модель даних БД

Нижче наведено схему нормалізованої бази даних спроектованої в Лабораторній роботі №1.

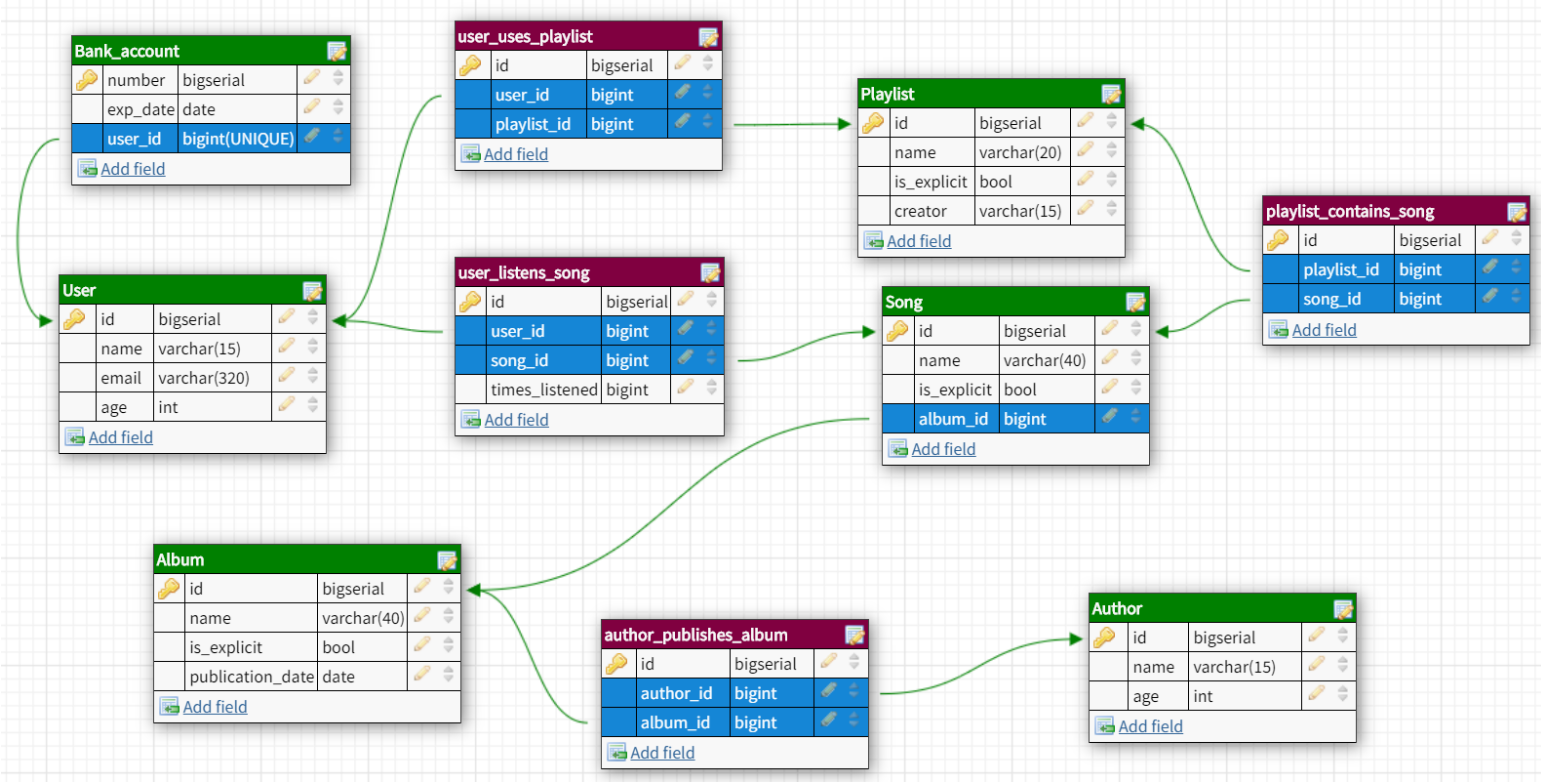


Рисунок 1.1 — Схема нормалізованої бази даних PostgreSQL на основі ER-моделі предметної області "Музичний стрімінговий сервіс".

Примітка. При побудові схеми БД використано сервіс Dbdesigner

Середовище розробки – Visual Studio Code. Мова програмування – Python 3.8. Бібліотека роботи з БД – psychopg2

Опис програми

Програма створена для управління базою даних за допомогою базових операцій СУБД PostgreSQL та реалізовує функціональні вимоги, що наведені у завданні. Програма складається з 3 модулів:

1. main.py — точка входу програми. Створення головного меню та підменю, яке являє собою відповідний контролер БД;
2. model.py — модуль містить опис класу Model, в якому виконується більшість бізнес-логіки за допомогою SQL-запитів;

3. view.py — модуль містить опис класу View, який відповідає за обробку та виведення даних, отриманих в результаті запитів до БД.

Структура меню програми

Нижче наведено структуру меню програми.

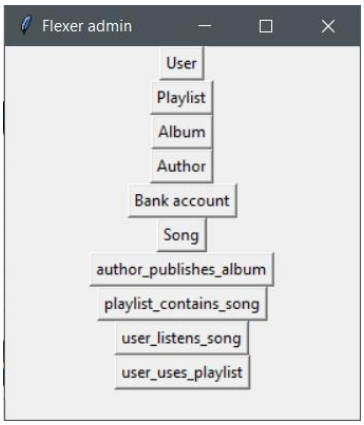


Рисунок 2.1 — Головне меню програми. Вибір таблиці для подальшої роботи

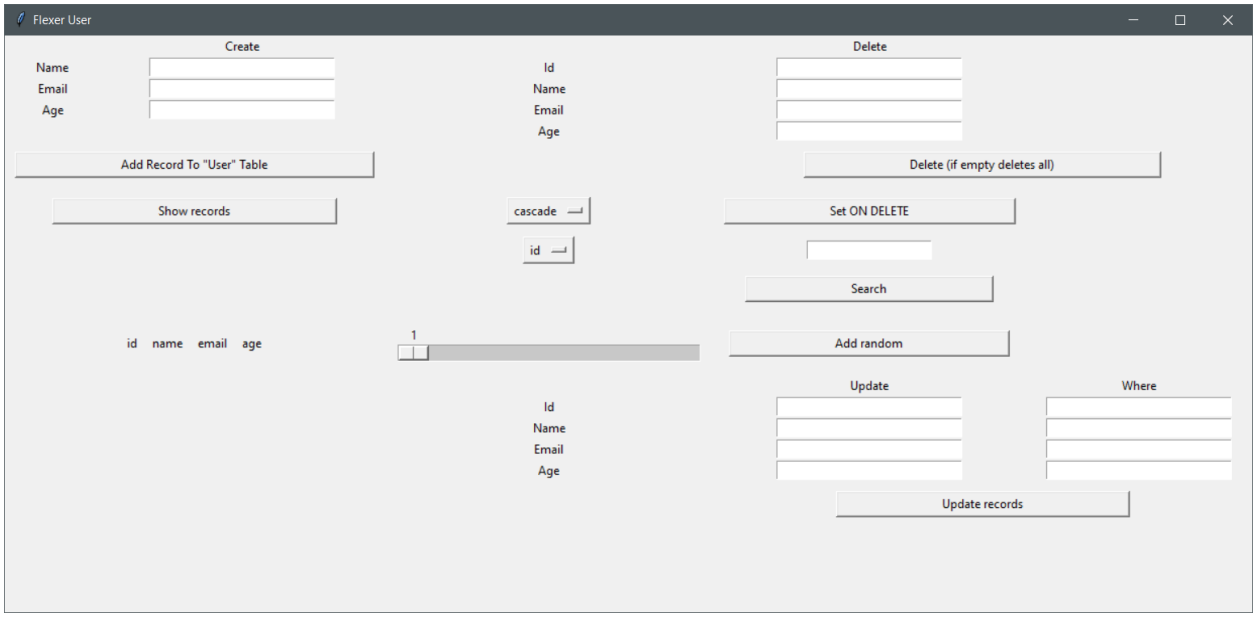


Рисунок 2.2 — Підменю програми (на прикладі таблиці User)

Інтерфейс програми та функції роботи з БД були спроектовані так, щоб динамічно підлаштовуватися під обрану для роботи таблицю, тому підменю для роботи з іншими таблицями не буде радикально відрізнятися від наведеного прикладу на рисунку 2.2. Також завдяки цьому програма

автоматично адаптується до бази даних (вилучення/додавання стовпця/таблиці та навіть повна зміна бази даних)

Інтерфейс програми

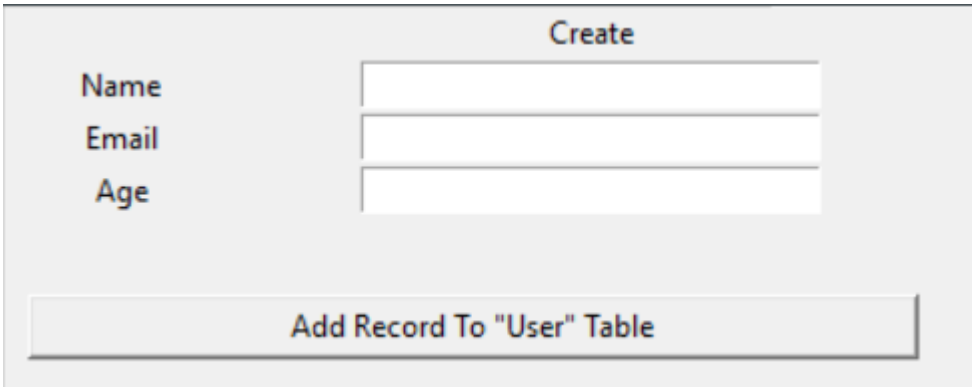


Рисунок 3.1 — Форма “Create” дозволяє додати запис в таблицю. Дані зразу з’являються в інтерфейсі

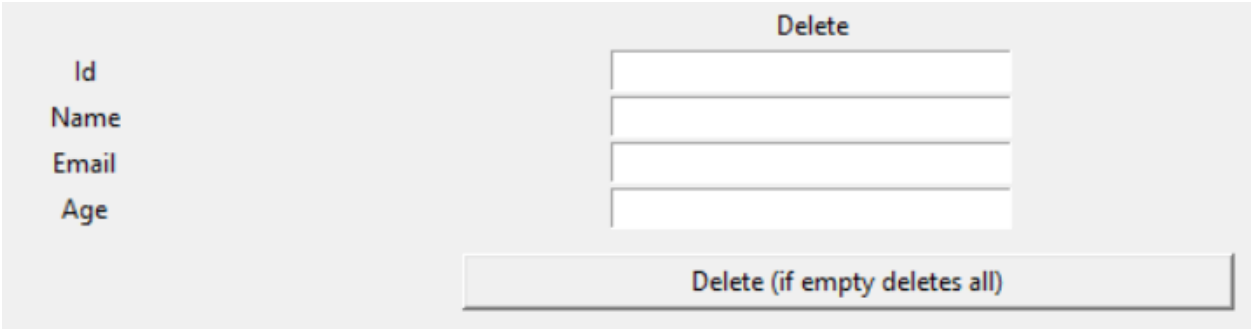


Рисунок 3.2 — Форма “Delete” дозволяє видаляти запис(-и) за одним чи декількома критеріями. Якщо відповідні поля не заповнені – видаляє всі записи з таблиці

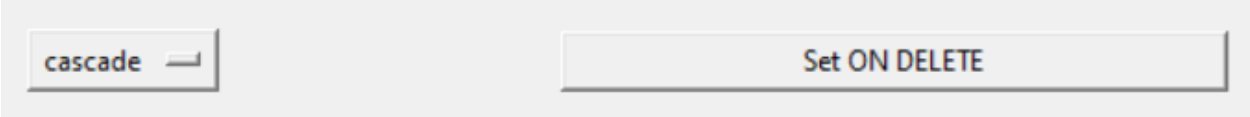


Рисунок 3.3 —Форма “Set ON Delete” дозволяє змінити поведінку видалення таблиці

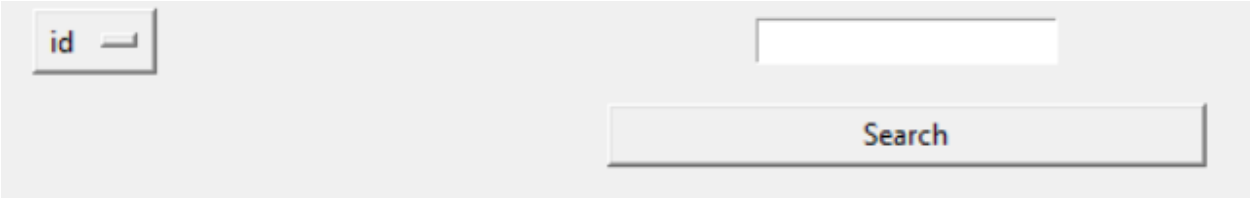


Рисунок 3.4 —Форма “Search” дозволяє пошук в таблиці

Рисунок 3.5 —Форма “Add random” дозволяє додати обрану кількість рандомізованих даних(засобами PostgreSQL)

Рисунок 3.4 —Форма “Show records” дозволяє переглянути записи в таблиці

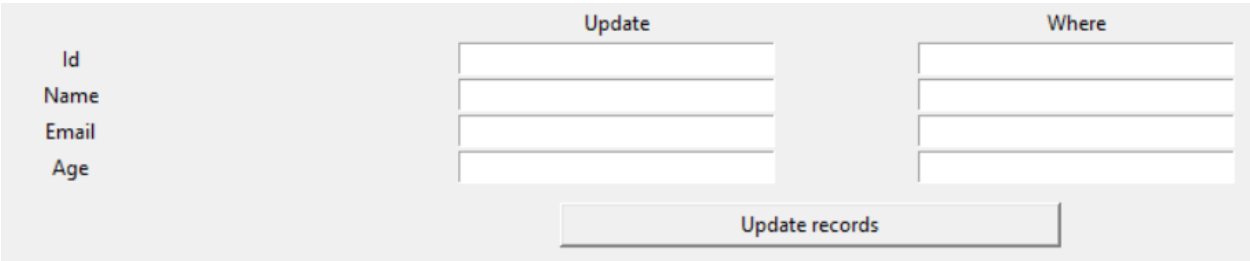


Рисунок 3.4 —Форма “Update records” дозволяє оновити записи в таблиці

Підключення до БД за допомогою бібліотек psycopg2 та contextlib

Нижче наведено програмну реалізацію підключення до БД. Завдяки використанню функції closing з бібліотеки contextlib відключення від БД відбувається автоматично, при виході з відповідного блоку програми.

```
def open_table(self, table_name):
    def open_tb():
        try:
            with closing(pg.connect(dbname='Flexer', user='postgres',
                password='1234', host='localhost')) as conn:
                with conn.cursor(cursor_factory=DictCursor) as cursor:
                    table_root = tk.Tk()
                    TableGui(table_root, table_name, cursor, conn)
                    table_root.mainloop()
        except pg.OperationalError:
            print('An error occured when connecting to database')
    return open_tb
```

Рисунок 4 — Підключення до БД

Код функціональної моделі таблиці

```
'''File model.py has Model class'''
from random import randint
from dateutil import parser
import psycopg2 as pg
import numpy as np
from psycopg2.sql import SQL, Identifier
from psycopg2.extras import DictCursor

class Model():
    '''Database CRUD model'''
    def __init__(self, cursor: DictCursor, table_name, columns_data):
        self.table_name = table_name
        self.cursor = cursor
        self.columns_data = np.array(columns_data)
        self.columns_data_but_id = columns_data[columns_data[:,0]!='id']
        param = '"' + '", "'.join(self.columns_data_but_id[:,0]) + '"'
        self.insert_request  = 'insert into {} ' + f'({param}) values ({("%s, "*len(self.columns_data_but_id))[:-2]})'
        self.insert_random_f_request = 'insert into {} ' + f'({param}) select'
        #Creating format template for random insert request
        for type_elem in self.columns_data_but_id[:,1]:
            if type_elem == 'character varying':
                for i in range(randint(1, 10)):
                    self.insert_random_f_request += ' chr(trunc(65+random()*25)::int) ||'
                self.insert_random_f_request = self.insert_random_f_request[:-3] + ','
            elif type_elem in ('bigint', 'integer'):
                self.insert_random_f_request += ' trunc(random()*100000)::int ,'
            elif type_elem == 'boolean':
                self.insert_random_f_request += '(trunc(random()*2)::int)::boolean,'
            elif type_elem == 'timestamp without time zone':
                self.insert_random_f_request += " timestamp '1950-01-01 00:00:00' + random()*(timestamp '1950-01-01 00:00:00' - current_timestamp),"

        self.insert_random_f_request = self.insert_random_f_request[:-1]
```

Рисунок 5.1 — Ініціалізація моделі та підлаштування під таблицю

```
    def create_record(self, data):
        '''Adds reccord to the table'''
        request = SQL(self.insert_request).format(Identifier(self.table_name))
        self.cursor.execute(request, data)
```

Рисунок 5.2 — Метод додавання запису до таблиці

```
def create_random(self, amount):
    ins_rnd_req = self.insert_random_f_request + f'from generate_series(1,{amount});'
    request = SQL(ins_rnd_req).format(Identifier(self.table_name))
    self.cursor.execute(request)
```

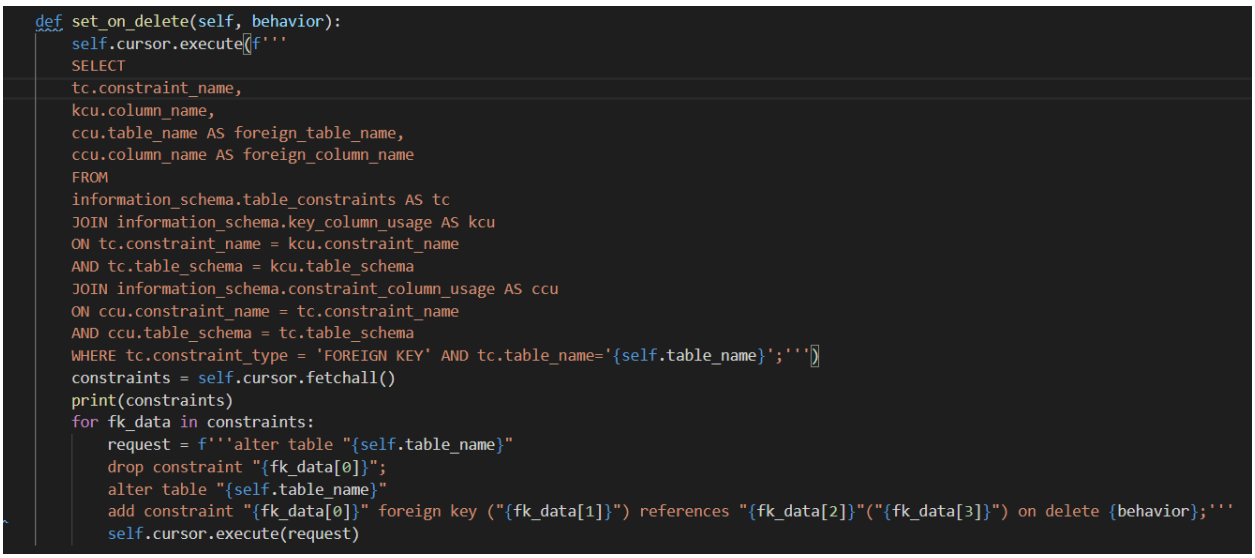
Рисунок 5.3 — Метод додавання випадкових записів до таблиці

```
def read_records(self):
    request = SQL('select * from {}').format(Identifier(self.table_name))
    self.cursor.execute(request)
    return self.cursor.fetchmany(10)
```

Рисунок 5.4 — Зчитування записів з таблиці

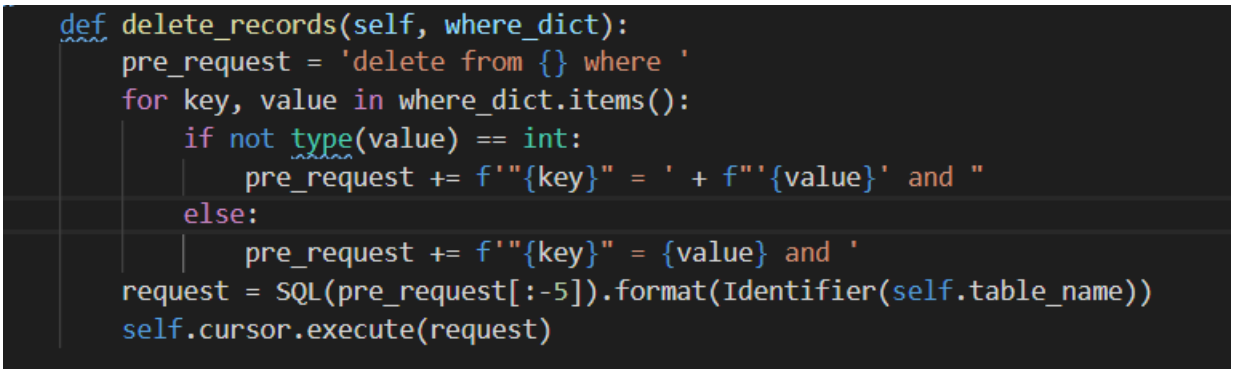
```
    def update_record(self, data_dict, where_dict):
        set_values = ''
        where_values = ''
        for key, value in data_dict.items():
            if not type(value) == int:
                set_values += f'"{key}" = ' + f"'{value}', "
            else:
                set_values += f'"{key}" = {value}, '
        for key, value in where_dict.items():
            if not type(value) == int:
                where_values += f'"{key}" = ' + f"'{value}' and "
            else:
                where_values += f'"{key}" = {value} and '
        pre_request = 'update {}' +  f' set {set_values[:-2]} where {where_values[:-5]};'
        #print(pre_request)
        request = SQL(pre_request).format(Identifier(self.table_name))
        self.cursor.execute(request)
```

Рисунок 5.5 — Метод оновлення запису



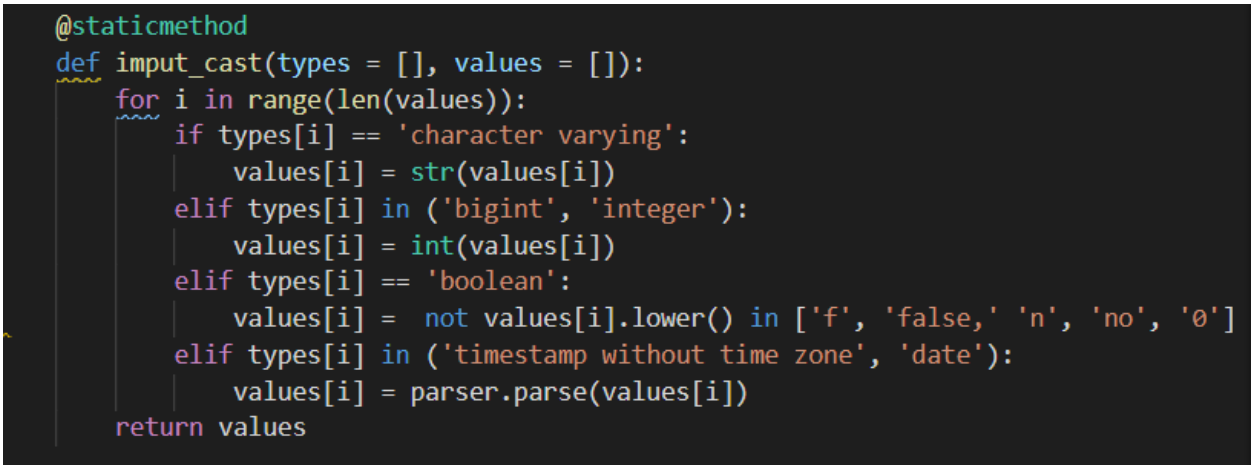
```
def set_on_delete(self, behavior):
    self.cursor.execute(f'''
    SELECT
    tc.constraint_name,
    kcu.column_name,
    ccu.table_name AS foreign_table_name,
    ccu.column_name AS foreign_column_name
    FROM
    information_schema.table_constraints AS tc
    JOIN information_schema.key_column_usage AS kcu
    ON tc.constraint_name = kcu.constraint_name
    AND tc.table_schema = kcu.table_schema
    JOIN information_schema.constraint_column_usage AS ccu
    ON ccu.constraint_name = tc.constraint_name
    AND ccu.table_schema = tc.table_schema
    WHERE tc.constraint_type = 'FOREIGN KEY' AND tc.table_name='{self.table_name}';''')
    constraints = self.cursor.fetchall()
    print(constraints)
    for fk_data in constraints:
        request = f'''alter table "{self.table_name}"
        drop constraint "{fk_data[0]}";
        alter table "{self.table_name}"
        add constraint "{fk_data[0]}" foreign key ("{fk_data[1]}") references "{fk_data[2]}"("{fk_data[3]}") on delete {behavior};'''
        self.cursor.execute(request)
```

Рисунок 5.6 — Метод зміни поведінки ON DELETE



```
def delete_records(self, where_dict):
    pre_request = 'delete from {} where '
    for key, value in where_dict.items():
        if not type(value) == int:
            pre_request += f'"{key}" = ' + f"'{value}' and "
        else:
            pre_request += f'"{key}" = {value} and '
    request = SQL(pre_request[:-5]).format(Identifier(self.table_name))
    self.cursor.execute(request)
```

Рисунок 5.7 — Метод видалення записів



```
@staticmethod
def imput_cast(types = [], values = []):
    for i in range(len(values)):
        if types[i] == 'character varying':
            values[i] = str(values[i])
        elif types[i] in ('bigint', 'integer'):
            values[i] = int(values[i])
        elif types[i] == 'boolean':
            values[i] =  not values[i].lower() in ['f', 'false,' 'n', 'no', '0']
        elif types[i] in ('timestamp without time zone', 'date'):
            values[i] = parser.parse(values[i])
    return values
```

Рисунок 5.8 — Метод перетворення типів даних

Вибірка елементів з таблиці

Форма “Search” дозволяє пошук в таблиці за значенням обраного поля. Отримані записи відображаються в формі “Show records”

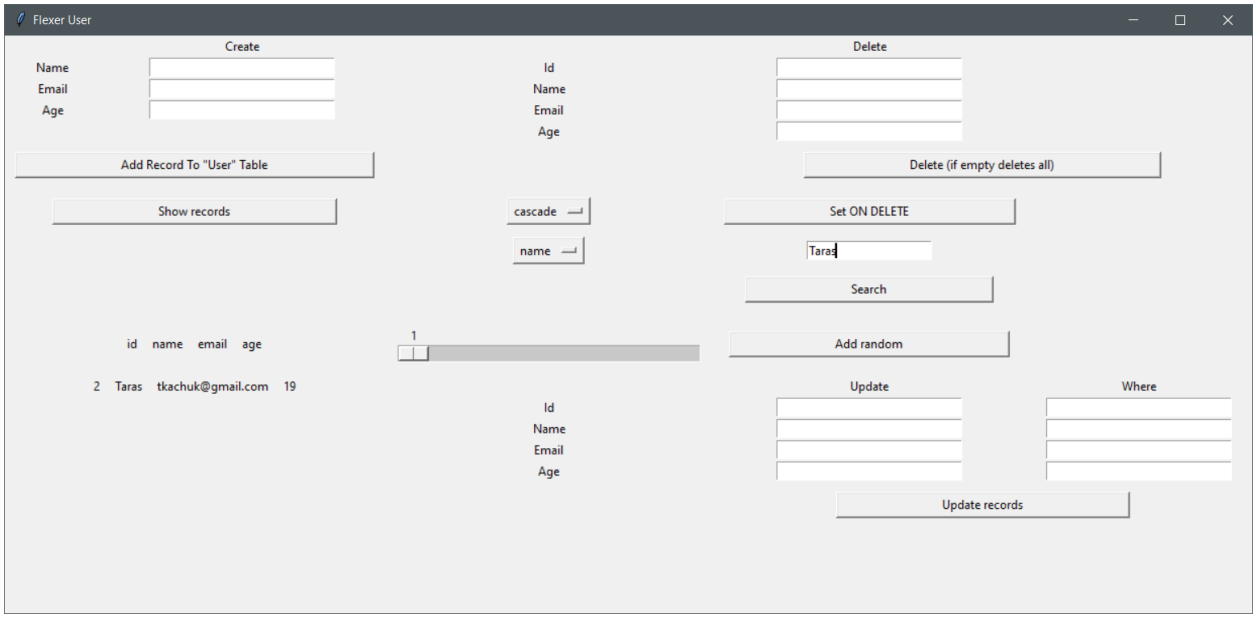


Рисунок 6 — Приклад пошуку в таблиці User за іменем

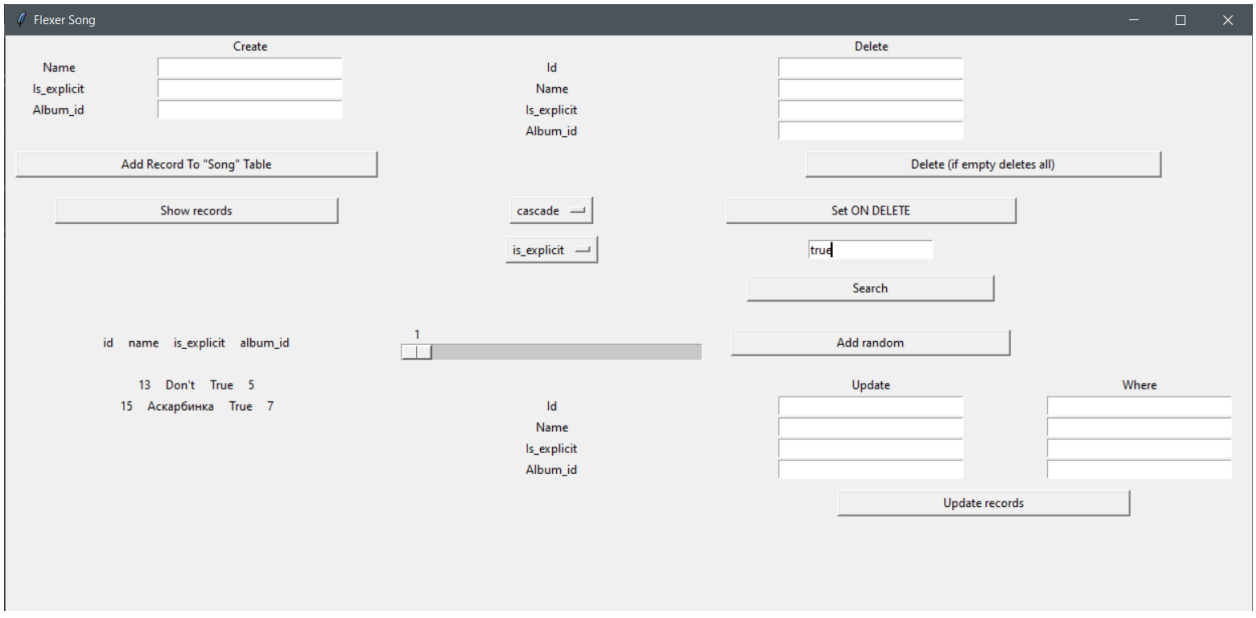


Рисунок 6.1 — Приклад пошуку в таблиці Song за полем is_explicit

Видалення зв’язаних між собою даних

Для зміни поведінки таблиці при видаленні записів потрібно скористатися формою “Set On Delete”, що викликає метод зміни поведінки (дивіться Рисунок 4.6)

PostgreSQL також дає можливість вибрати кілька режимів видалення:

1. **“ON DELETE SET NULL”** - всі Foreign key будуть мати значення Null, а за неможливості цього зробити буде повідомлення про помилку.

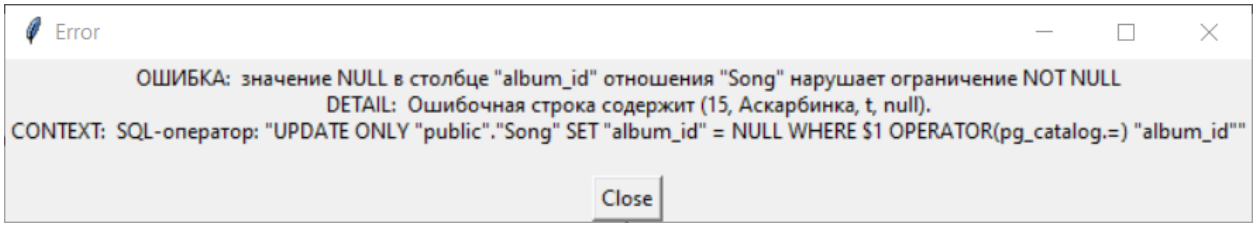


Рисунок 7.1 — Опрацювання помилки при поведінці видалення **SET NULL**

2. **“ON DELETE RESTRICT”** – не дає можливості видалити батьківський рядок, якщо в нього є дочірні. Поведінка дуже схожа на NO ACTION

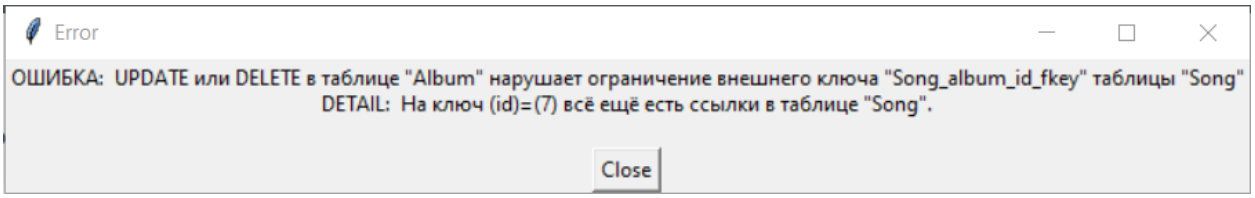


Рисунок 7.2 — Опрацювання помилки при поведінці видалення **RESTRICT**

Структура програми

Нижче наведено структуру програми та взаємодію окремих модулів.

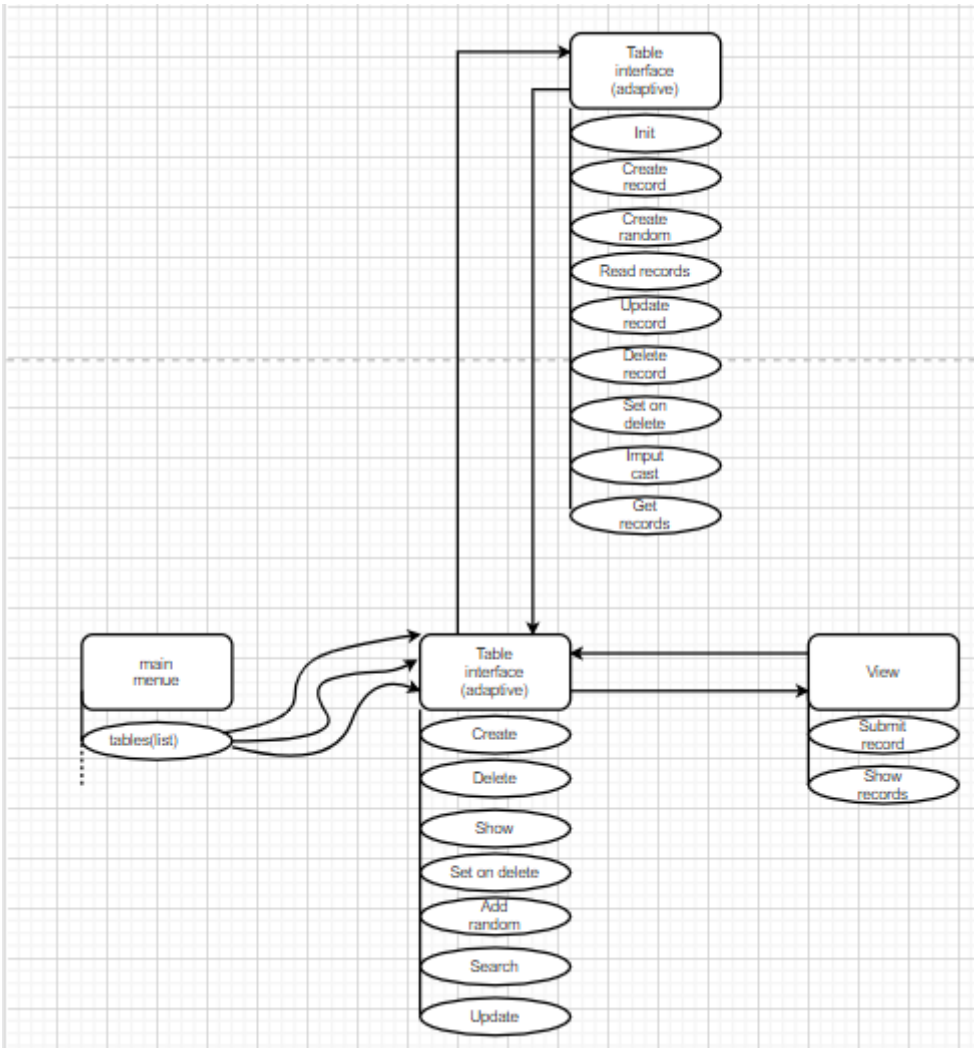


Рисунок 8 — Діаграма зв’язків модулів програми

Модуль main

Нижче наведено програмну реалізацію функцій та методів модуля main,

```
def submit(self):
        '''Submitting text and clearing textboxes'''
        try:
            values = [entry.get() for entry in self.column_entries.values()]
            types = tuple(self.columns_data[1:, 0])
            values = Model.imput_cast(types, values)
            self.model.create_record(values)
            View.clear(self.column_entries.values())

        except pg.DataError as e:
            self.message(e, 'Error')
        except ValueError  as e:
            self.message(e, 'Error')
        finally:
            self.conn.commit()
            self.show_items()
```

Рисунок 8.1 — Метод, що додає один запис

```
    def add(self):
        try:
            self.model.create_random(self.slider.get())
        except pg.IntegrityError:
            self.message('Random add is not available due to FK or constraints',
'Error')
        finally:
            self.show_items()
```

Рисунок 8.2 — Метод, що додає випадкові записи

```
    def set_on_delete(self):
        self.model.set_on_delete(self.variable.get())
        self.conn.commit()
```

Рисунок 8.3 — Метод, що змінює поведінку видалення

```
    def update(self):
        try:
            values_where = []
            types_where = []
            columns_where = []
            values_new = [entry.get() for entry in self.update_entries_new.values
()]
            for value, column, column_type in zip([entry.get() for entry in self.
update_entries_where.values()],
            self.columns_data[:,0], self.columns_data[:,1]):
                if not value == '':
                    values_where.append(value)
                    types_where.append(column_type)
                    columns_where.append(column)
            values_where = Model.imput_cast(types_where, values_where)

            values_new = []
            types_new = []
            columns_new = []

            for value, column, column_type in zip([entry.get() for entry in self.
update_entries_new.values()],
            self.columns_data[:,0], self.columns_data[:,1]):
                if not value == '':
                    values_new.append(value)
                    types_new.append(column_type)
                    columns_new.append(column)
            values_new = Model.imput_cast(types_new, values_new)

            self.model.update_record(dict(zip(columns_new, values_new)), dict(zip
(columns_where, values_where)))
            View.clear(self.update_entries_new.values())
            View.clear(self.update_entries_where.values())
        except pg.DataError as e:
            self.message(e, 'Error')
            self.conn.rollback()
        except ValueError  as e:
            self.message(e, 'Error')
            self.conn.rollback()
        except pg.IntegrityError:
            self.message('Update is not available due to FK or constraints', 'Err
or')
            self.conn.rollback()
        finally:
            self.conn.commit()
            self.show_items()
```

Рисунок 8.4 — Метод, що оновлює запис

```
    def show_items(self):
        try:
            query = self.model.read_records()
            View.show_records(query, self.query_labels[1:])
        except pg.OperationalError:
            self.message('Error', 'Error')
```

Рисунок 8.5 — Метод, що повертає записи таблиці

```
    def message(self, message, name):
        error_window = tk.Tk()
        error_window.title(name)
        error_label = tk.Label(error_window, text=message)
        error_label.pack()
        close_buttom = tk.Button(error_window, text='Close', command=error_window
.destroy)
        close_buttom.pack()
        self.conn.rollback()
        error_window.mainloop()
```

Рисунок 8.6 — Метод, виводить повідомлення про помилку

```
    def delete(self):
        try:
            types = []
            values = []
            columns = []
            for value, column, column_type in zip([entry.get() for entry in self.
delete_entries.values()],
            self.columns_data[:,0], self.columns_data[:,1]):
                if not value == '':
                    values.append(value)
                    types.append(column_type)
                    columns.append(column)
            values = Model.imput_cast(types, values)
            criteria = dict(zip(columns, values))
            self.model.delete_records(criteria)
        except pg.DataError as e:
            self.message(e, 'Error')
        except ValueError  as e:
            self.message(e, 'Error')
        except AttributeError:
            self.message("Field is empty or data types don't match", 'Error')
        finally:
            self.conn.commit()
            self.show_items()
```

Рисунок 8.7 — Модуль, що видаляє запис з таблиці

```
    def search(self):
        try:
            typee = self.columns_data[:,1][list(self.columns_data[:,0]).index(sel
f.search_var.get())]
            data = Model.imput_cast([typee], [self.search_entry.get()])
            column = self.search_var.get()
            query = self.model.get_records(column, data[0])
            View.show_records(query, self.query_labels[1:])
        except ValueError:
            self.message("Field is empty or data types don't match", 'Error')
        except TypeError:
            self.message("Field is empty or data types don't match", 'Error')
        except AttributeError:
            self.message("Field is empty or data types don't match", 'Error')
        finally:
            pass
```

Рисунок 8.8 — Метод, що виводить записи за певними критеріями

Модуль view

```
import tkinter as tk

class View():
    @staticmethod
    def show_records(records = [], labels = []):
        for label in labels:
            label['text'] = ''
        for i in range(0, min([len(records), len(labels)])):
            labels[i]['text'] = '     '.join([str(j) for j in records[i]])
```

Рисунок 9.1 — Метод, що виводить записи на екран

```
    @staticmethod
    def submit_record(entries):
        for entry in entries:
            entry.delete(0, tk.END)
```

Рисунок 9.2 — Метод, що очищує поля для введення

Приклади роботи програми

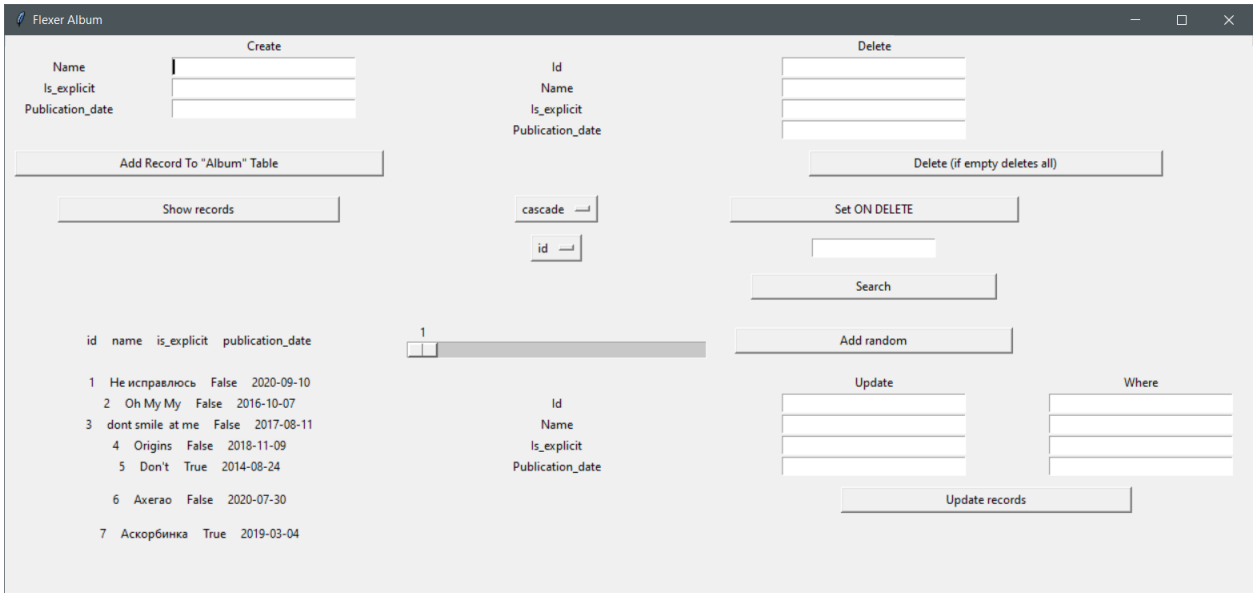


Рисунок 10.1 — Виведення записів з таблиці

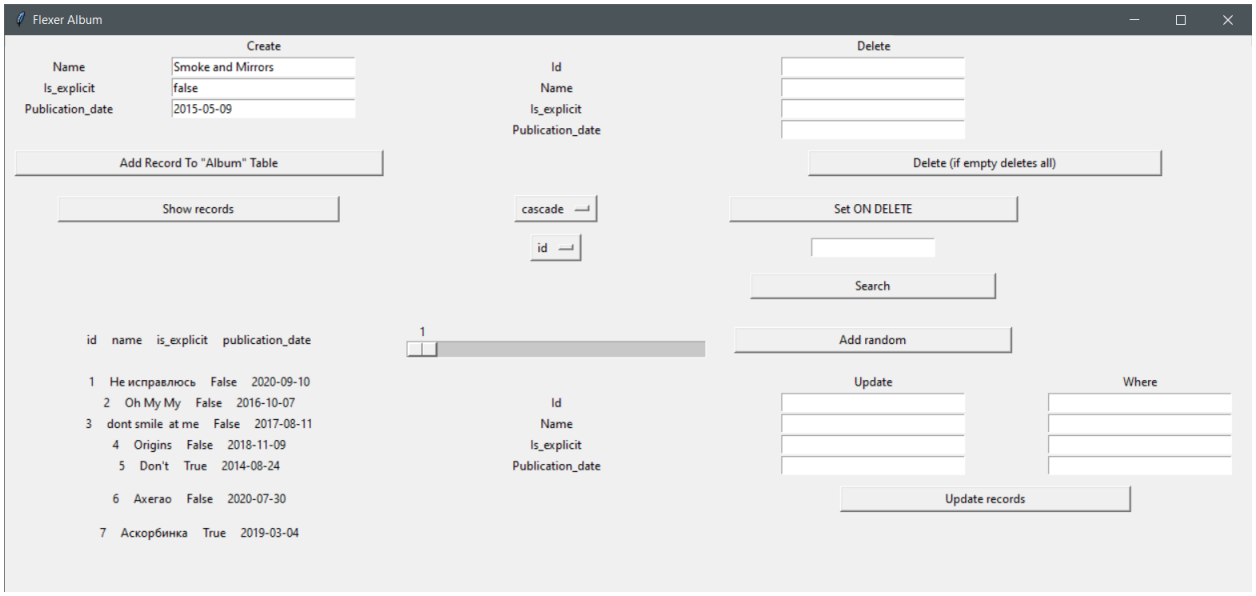


Рисунок 10.2 — Додавання запису в таблицю

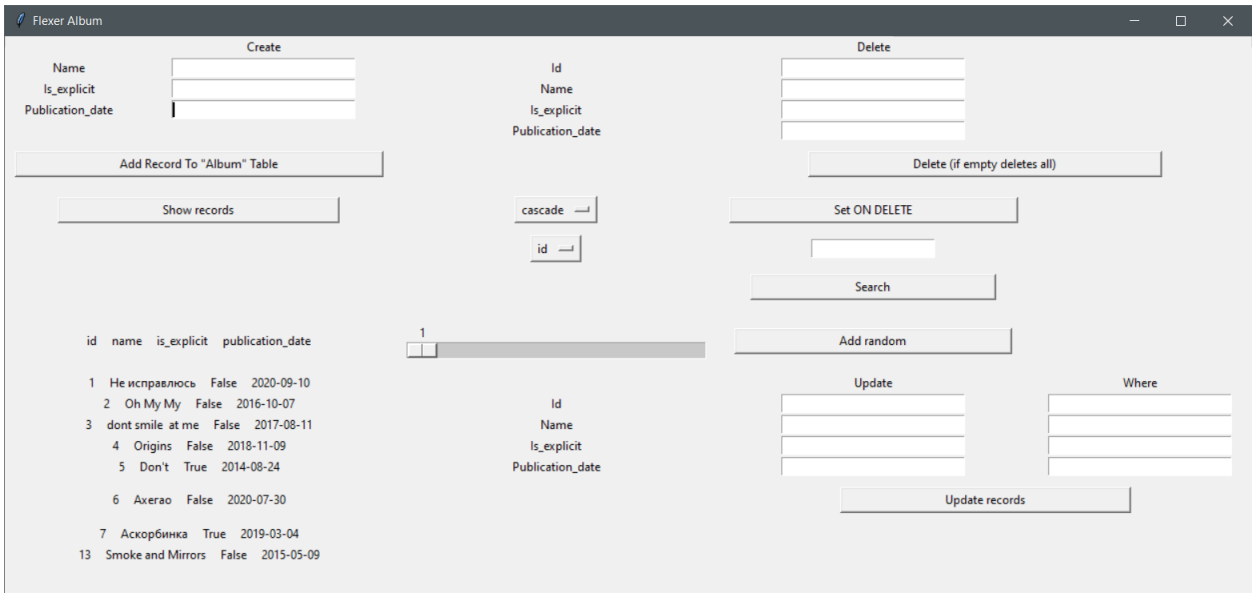


Рисунок 10.3 — Результат додавання запису в таблицю

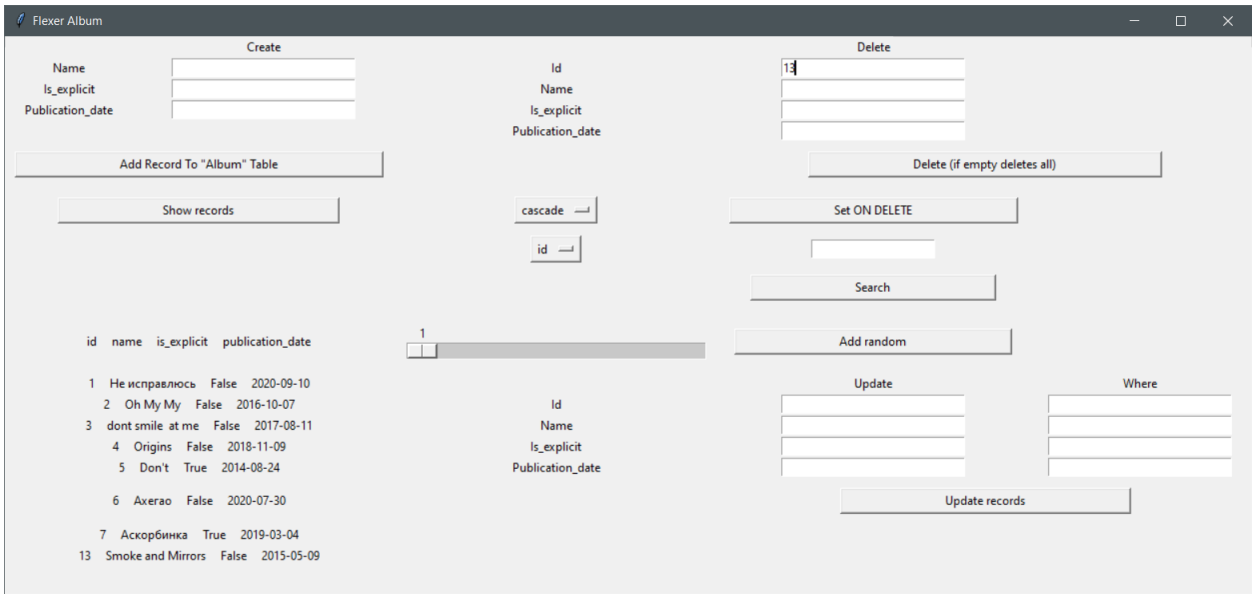


Рисунок 10.4 — Видалення запису з таблиці

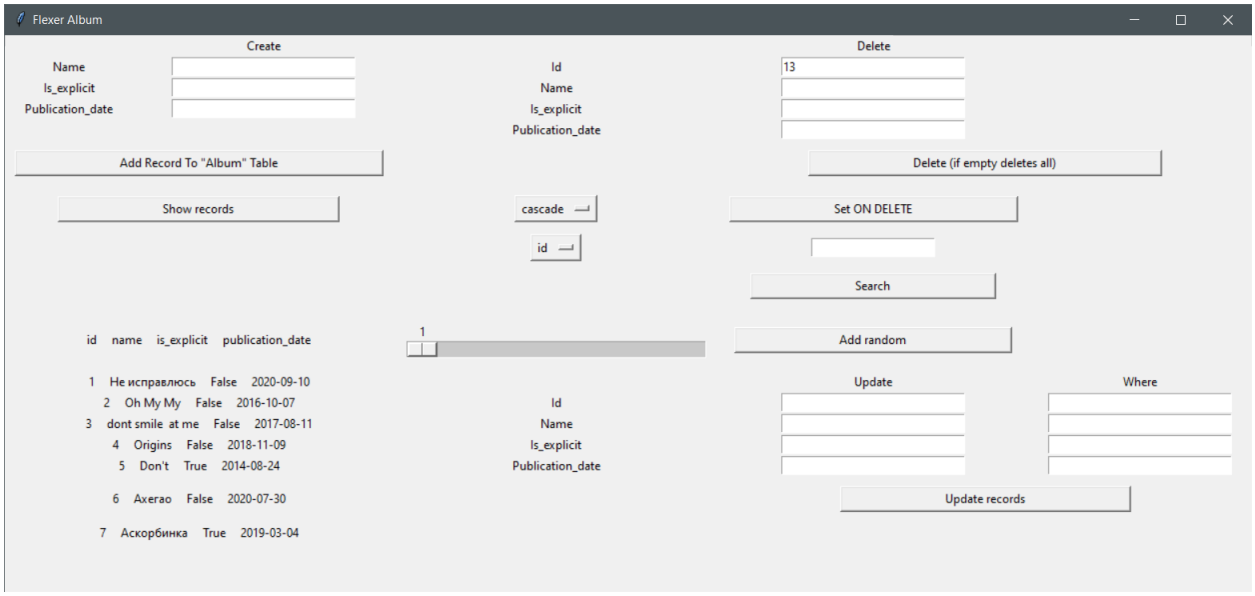


Рисунок 10.5 — Результат видалення запису з таблиці

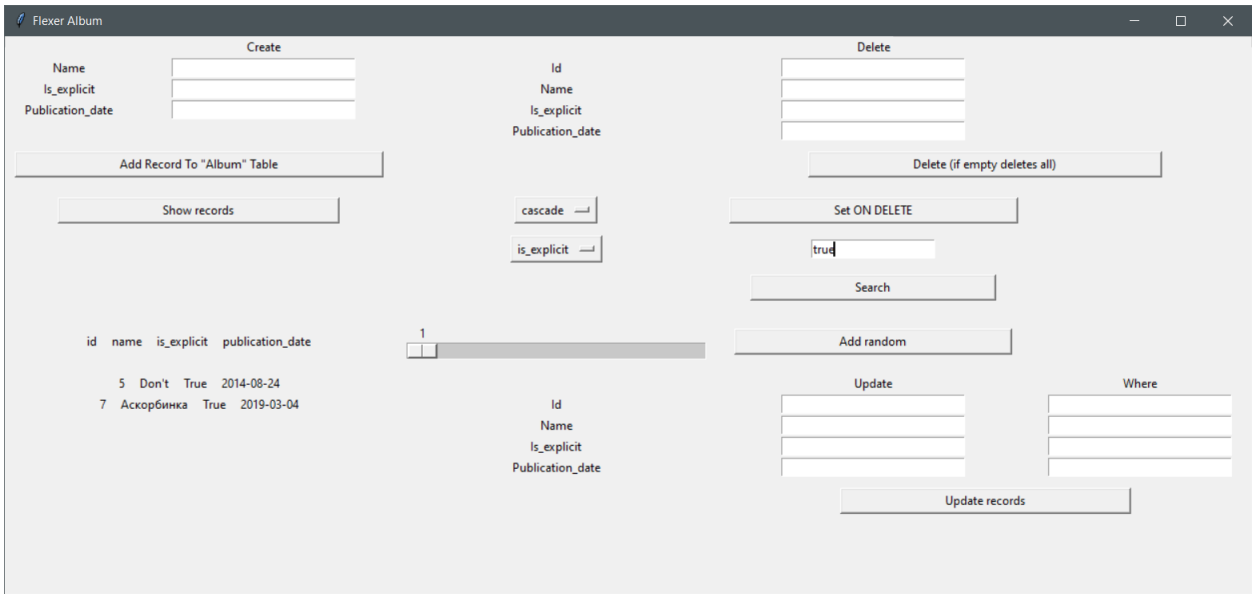


Рисунок 10.6 — Пошук за параметром в таблиці

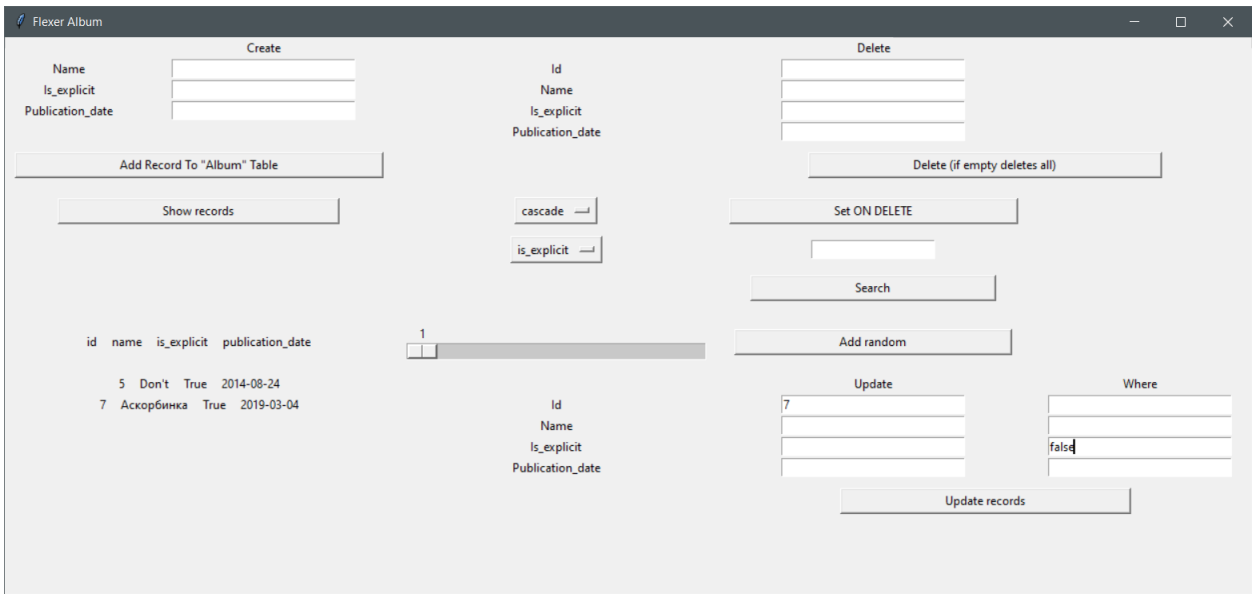


Рисунок 10.7 — Оновлення запису в таблиці

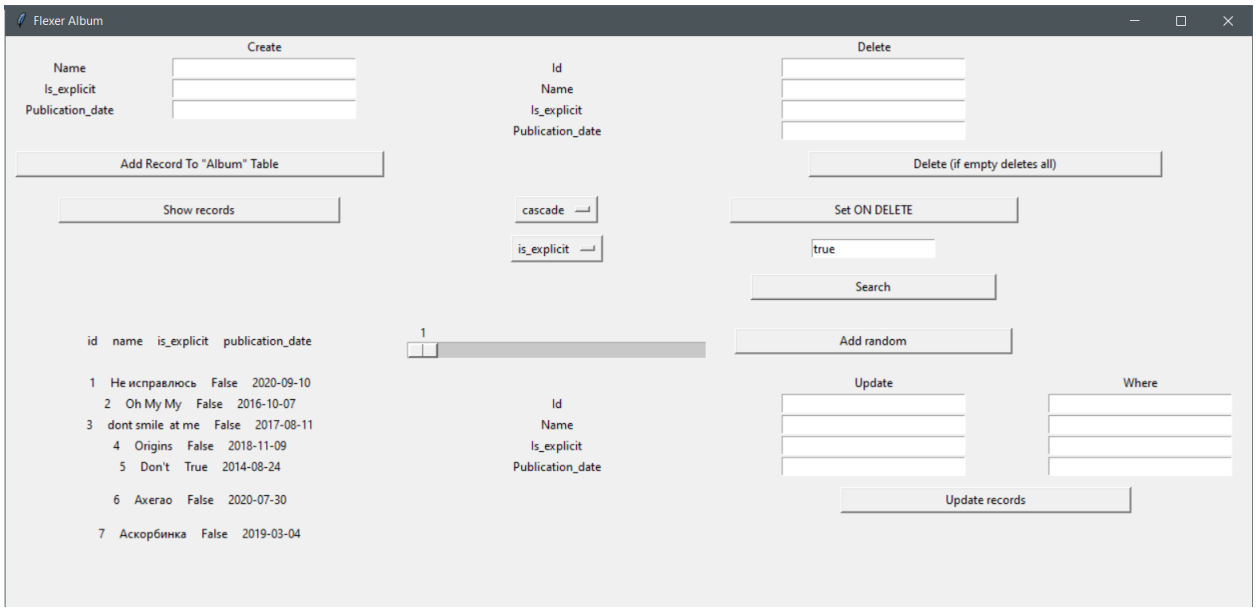


Рисунок 10.8 — Результат оновлення запису в таблиці