НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування та спеціалізованих комп’ютерних**

**систем**

**РГР**

з дисципліни

**«Бази даних і засоби управління»**

Тема: «Створення додатку бази даних, орієнтованого на взаємодію з СУБД PostgreSQL»

Виконав: студент III курсу

ФПМ групи КВ-13

Горбик Д.В.

Київ – 2023

*Метою роботи* є здобуття вмінь програмування прикладних додатків баз даних PostgreSQL.

*Загальне завдання* роботи полягає у наступному:

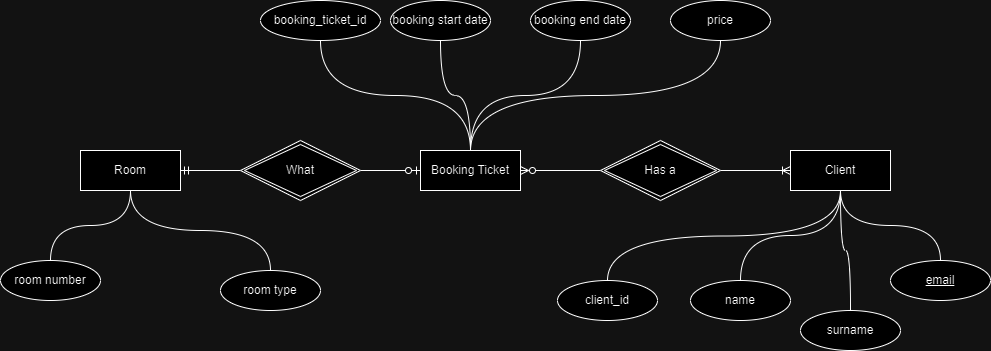
1. Реалізувати функції перегляду, внесення, редагування та вилучення даних у таблицях бази даних, створених у лабораторній роботі №1, засобами консольного інтерфейсу.
2. Передбачити автоматичне пакетне генерування «рандомізованих» даних у базі.
3. Забезпечити реалізацію пошуку за декількома атрибутами з двох 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).

*Деталізоване завдання*:

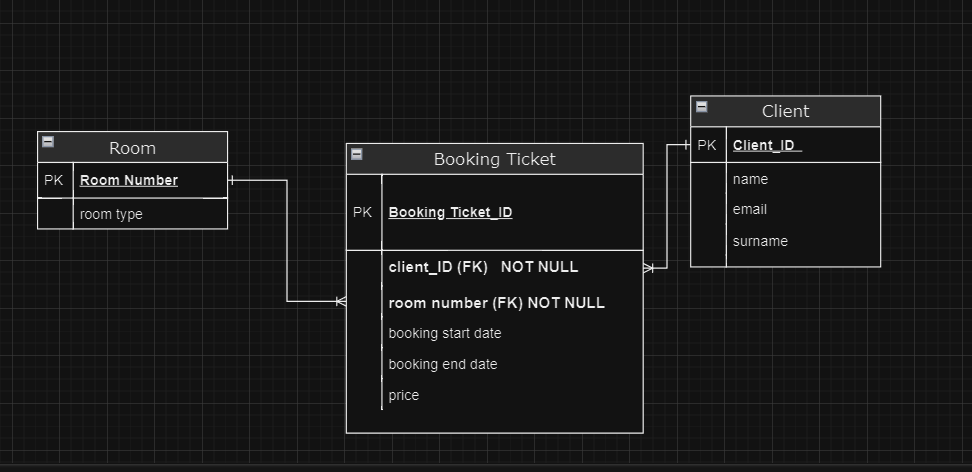
1. Забезпечити можливість уведення/редагування/вилучення даних у таблицях бази даних з можливістю контролю відповідності типів даних атрибутів таблиць (рядків, чисел, дати/часу). Для контролю пропонується два варіанти: контроль при введенні (валідація даних) та перехоплення помилок (try..except) від сервера PostgreSQL при виконанні відповідної команди SQL. Особливу увагу варто звернути на дані таблиць, що мають зв’язок 1:N. При цьому з боку батьківської таблиці необхідно контролювати **вилучення** рядків за умови наявності даних у підлеглій таблиці. З точки зору підлеглої таблиці варто контролювати наявність відповідного рядка у батьківській таблиці при виконанні **внесення** нових даних. Унеможливити виведення програмою системних помилок на екрані шляхом їх перехоплення і адекватної обробки. Внесення даних виконується користувачем у консольному вікні програми.
2. Забезпечити можливість автоматичної генерації великої кількості даних у таблицях за допомогою вбудованих у PostgreSQL функцій роботи з псевдовипадковими числами. Дані мають бути згенерованими **не мовою програмування, а відповідним SQL-запитом**!

**Інформація про модель та структуру бази даних**

Рис. 1 - Концептуальна модель предметної області “Готельний бронювальний портал”



Нижче (Рис. 2) наведено логічну модель бази даних:

Рис. 2 – Логічна модель бази даних

Зміни у порівнянні з першою лабораторною роботою відсутні.

**Середовище розробки та налаштування підключення до бази даних**

Для виконання лабораторної роботи використовувалась мова програмування Python .

Для підключення до серверу бази даних PostgreSQL використано модуль «psycopg2».

**Опис структури програми**

Програма містить 4 основні модулі: **Analytics, Booking, Client, Room,**

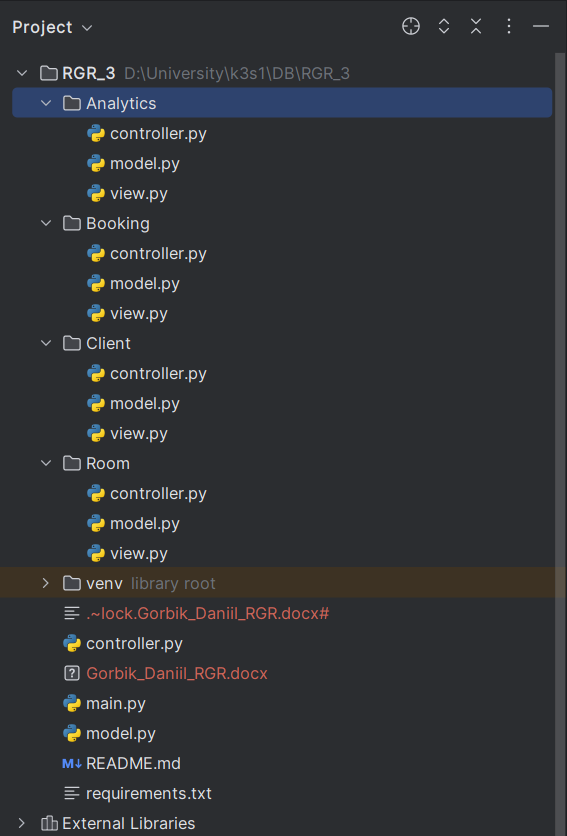
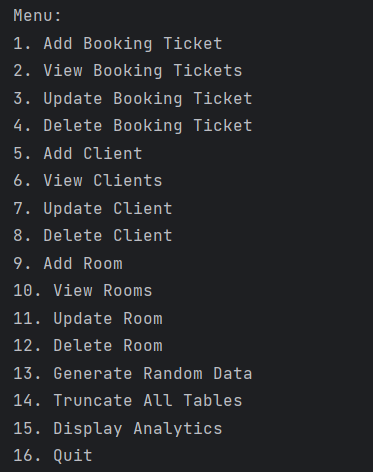


Рис. 3 – Структура програмного коду

**Структура меню програми**

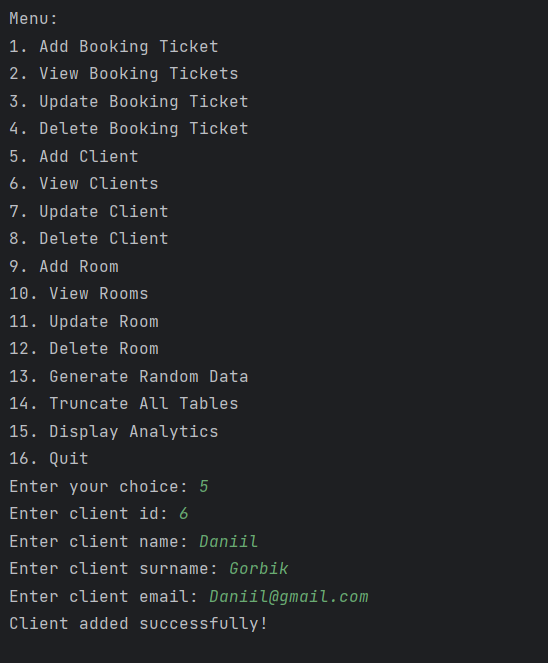


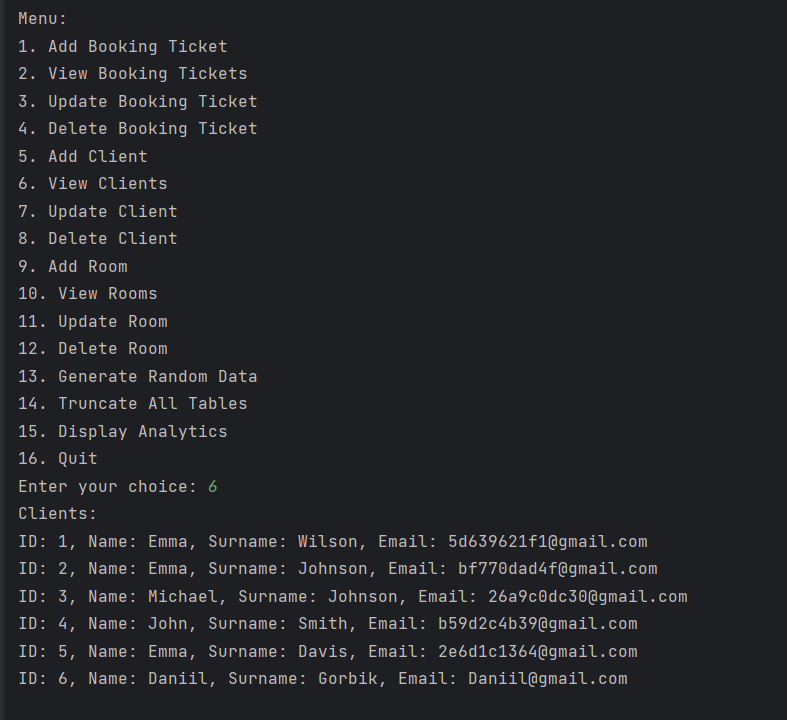
**Пункт 1**

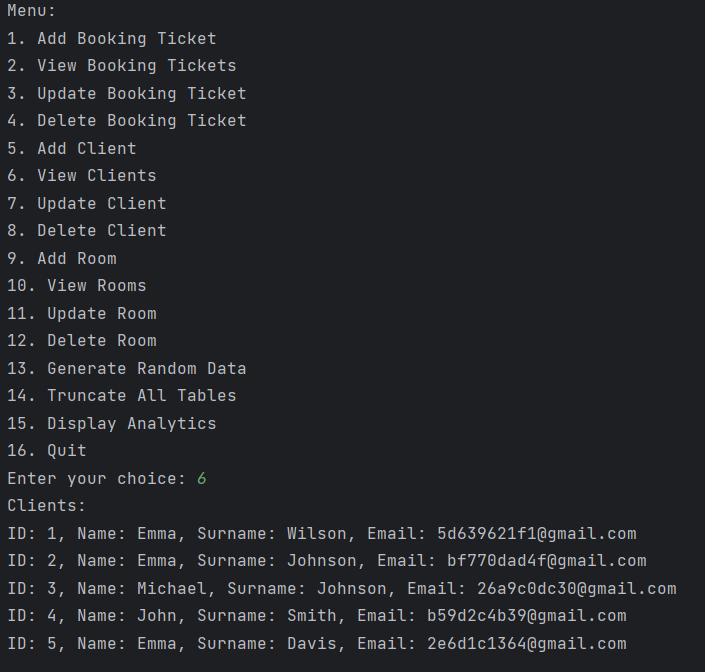
Реалізувати функції внесення, редагування та вилучення даних у таблицях бази даних, створених у лабораторній роботі №1, засобами консольного інтерфейсу.

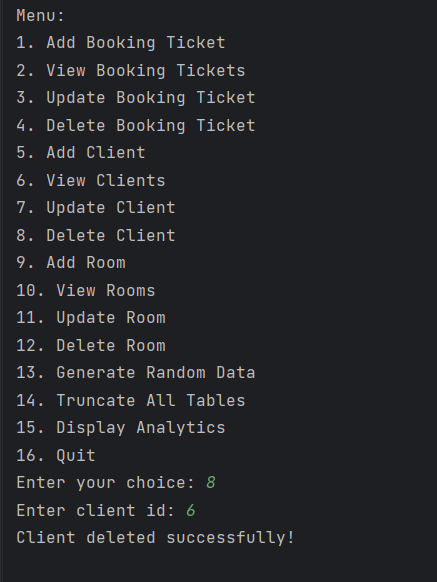
**Внесення даних**

Створення нового Client:

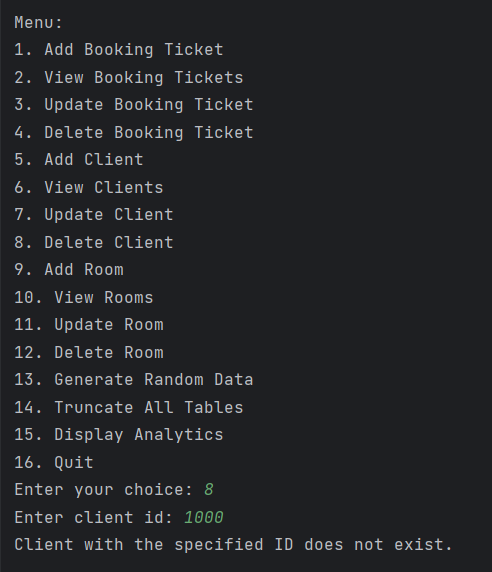




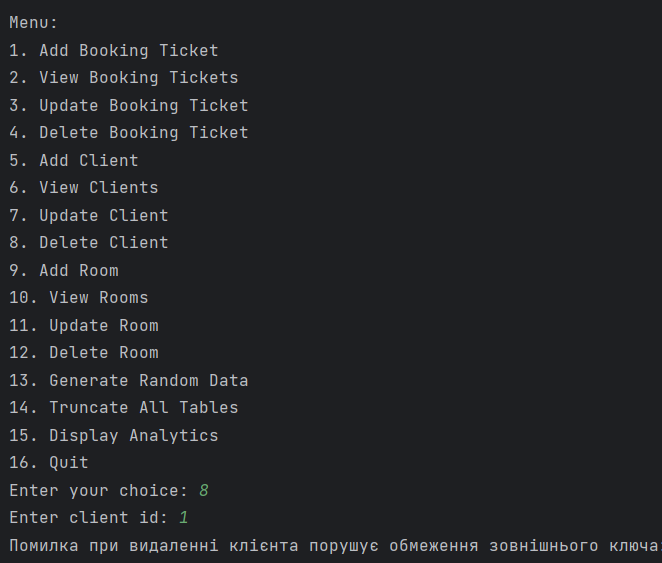
**Видалення даних**



Якщо уведено неіснуючий id :

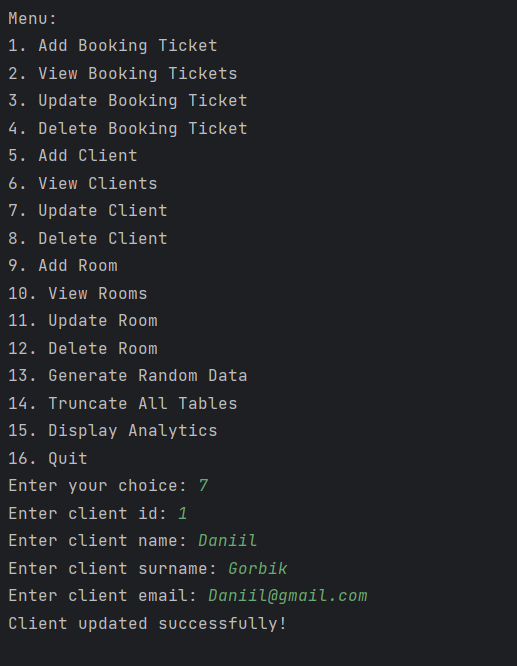


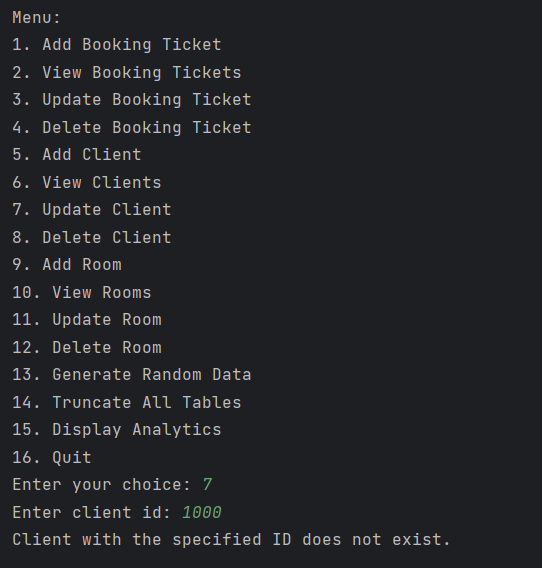
Порушує обмеження зовнішнього ключа:



Помилка при порушенні обмеження зовнішнього ключа виникає тоді, коли спроба вставити або змінити дані в таблиці, яка містить зовнішній ключ, порушує обмеження цього ключа. Обмеження зовнішнього ключа визначає, що значення в стовпці, який посилається на інший стовпець в іншій таблиці, повинно відповідати існуючим значенням в цій іншій таблиці.

Редагування даних



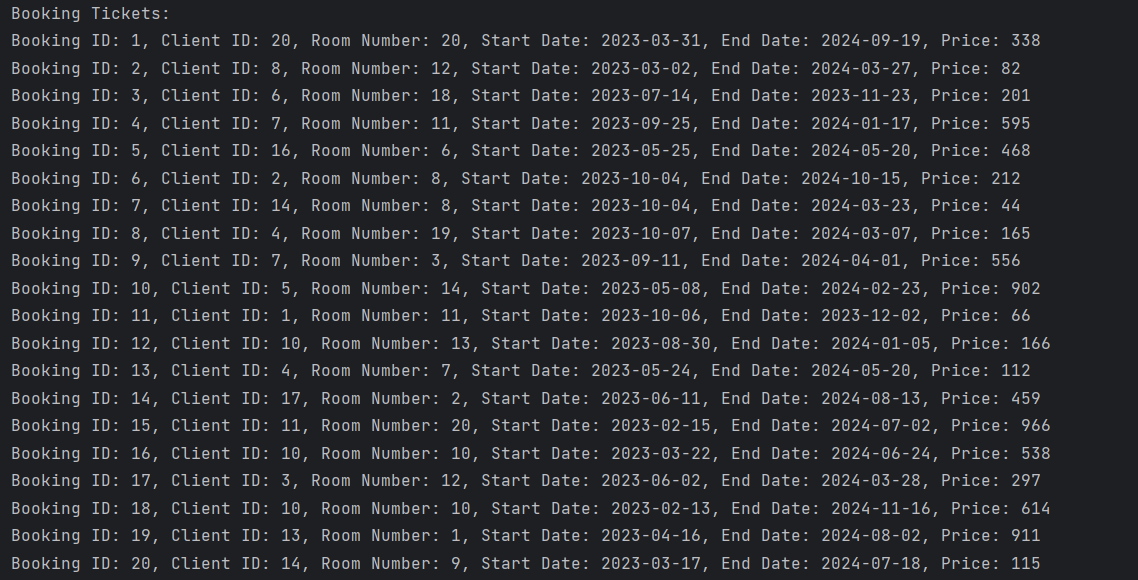


**Пункт 2**

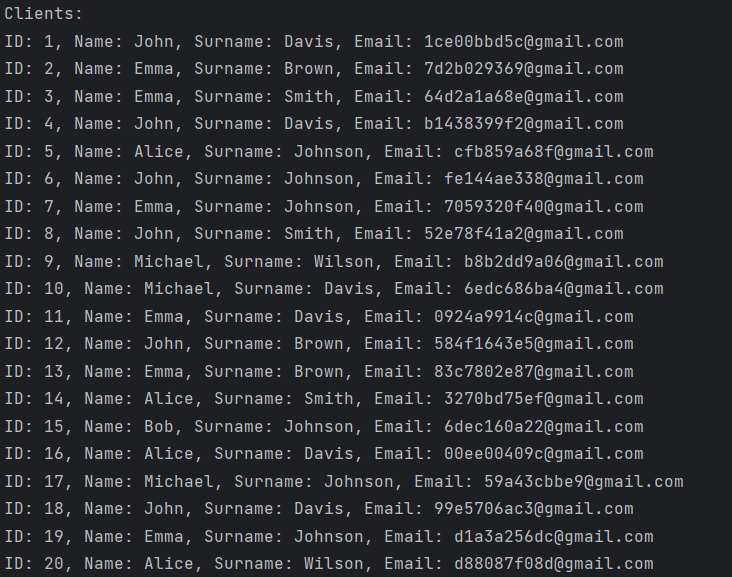
Забезпечити можливість автоматичної генерації великої кількості даних у таблицях за допомогою вбудованих у PostgreSQL функцій роботи з псевдовипадковими числами. Дані мають бути згенерованими **не мовою програмування, а відповідним SQL-запитом**!

Витяги деяких рандомізованих рядків з таблиць:

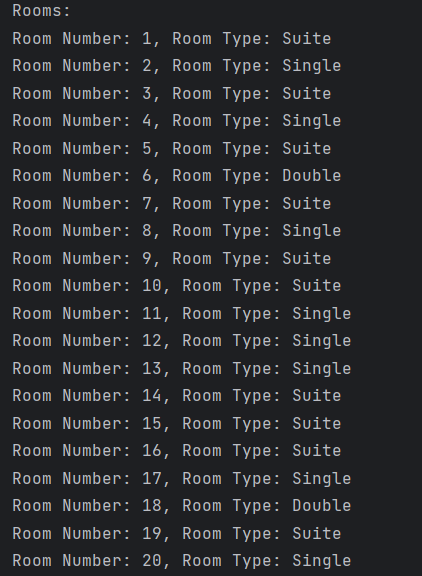
**Booking Ticket**

****

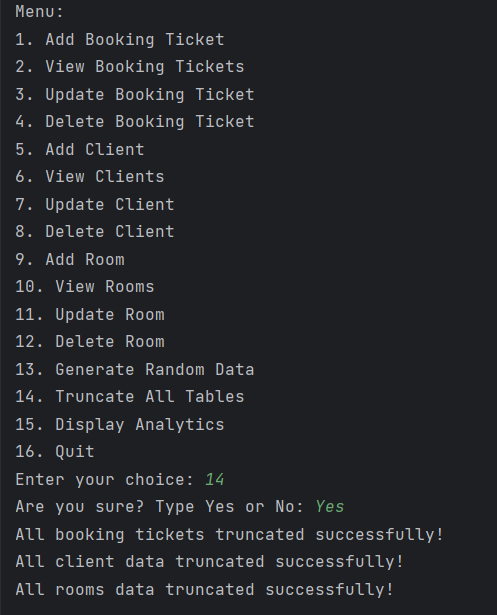
**Client**

****

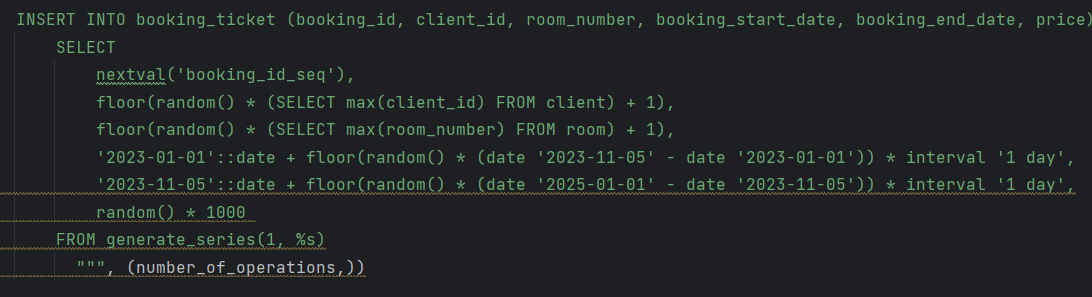
**Rooms**

****

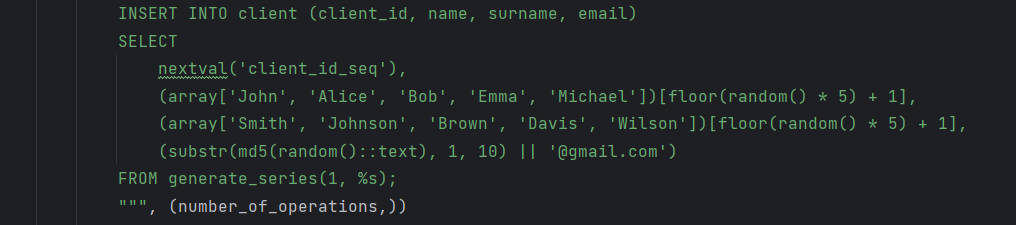
**Очищення всіх таблиць**

****

**SQL запити рандомізованого заповнення:**







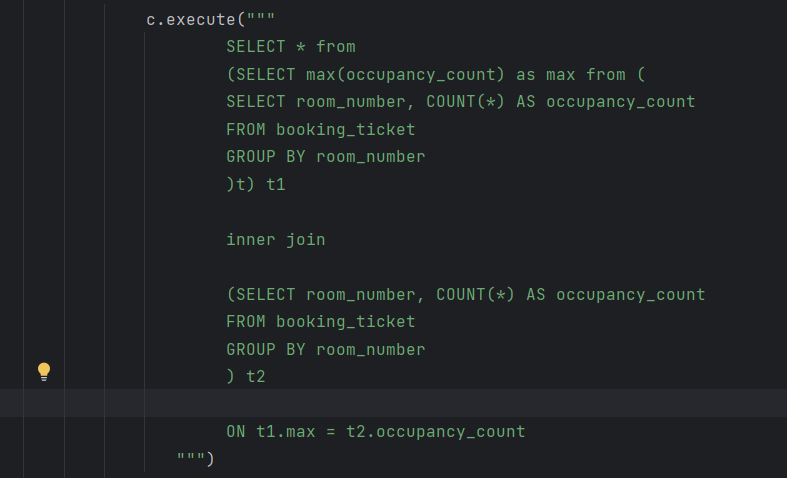
**Пункт 3**

Для реалізації пошуку необхідно підготувати 3 запити, що включають дані з декількох таблиць і фільтрують (WHERE) та групують (GROUP BY) рядки за 3-4 атрибутами цих таблиць. Забезпечити можливість уведення конкретних значень констант для фільтрації з клавіатури користувачем. Крім того, після виведення даних необхідно вивести час виконання запиту у мілісекундах. Перевірити швидкодію роботи запитів на попередньо згенерованих даних.

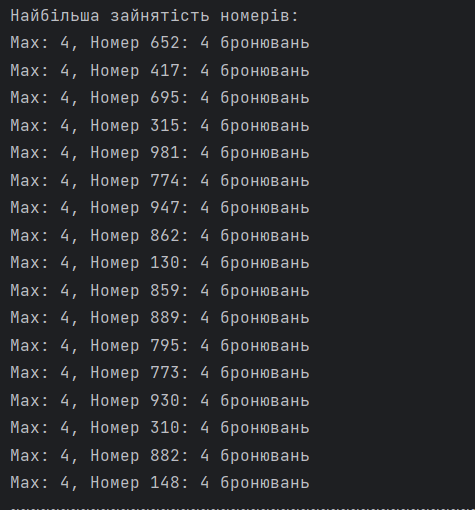
Було підготовлено три SQL запити:

* найбільша зайнятість номерів :
* кількість замовлень за останні два тижні
* аналітика найбільш активних клієнтів :

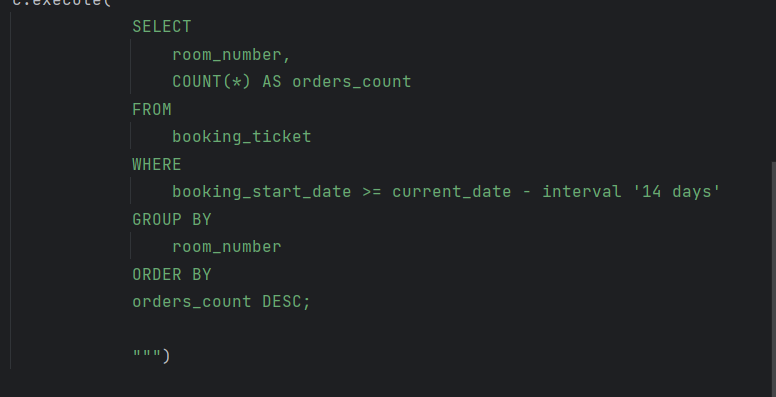
найбільша зайнятість номерів :



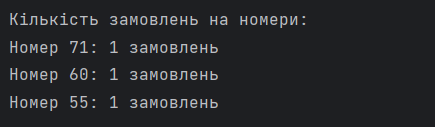
Результат:



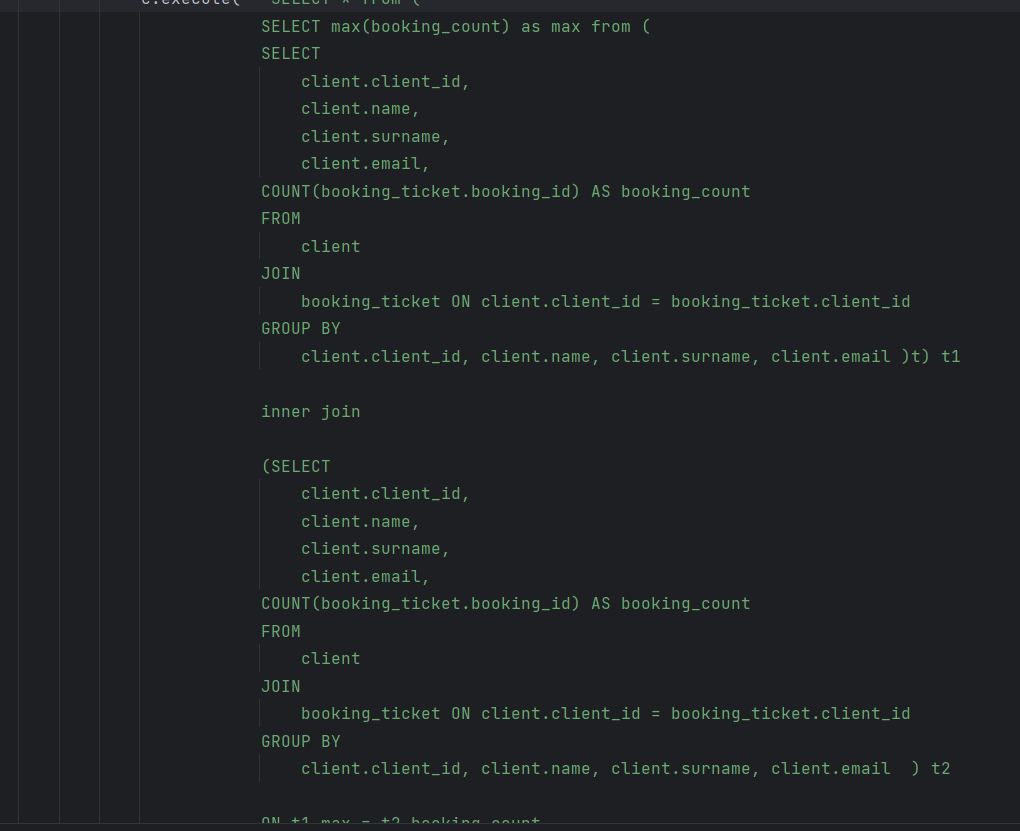
кількість замовлень за останні два тижні :



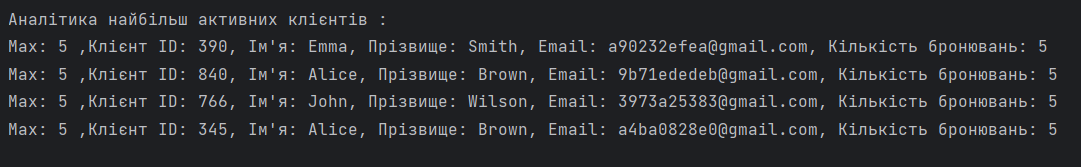
Результат:



аналітика найбільш активних клієнтів :



Результат:

**Код програмного модулю model**

**Analytics/model.py**

# ModelAnalytics  
#################################################################################  
class ModelAnalytics:  
 def \_\_init\_\_(self, db\_model):  
 self.conn = db\_model.conn  
  
 def room\_occupancy(self):  
 c = self.conn.cursor()  
 try:  
 c.execute("""  
 SELECT \* from  
 (SELECT max(occupancy\_count) as max from (  
 SELECT room\_number, COUNT(\*) AS occupancy\_count  
 FROM booking\_ticket  
 GROUP BY room\_number  
 )t) t1  
   
 inner join   
   
 (SELECT room\_number, COUNT(\*) AS occupancy\_count  
 FROM booking\_ticket  
 GROUP BY room\_number  
 ) t2  
   
 ON t1.max = t2.occupancy\_count   
 """)  
  
 room\_occupancy\_data = c.fetchall() # Get data from the query  
  
 self.conn.commit()  
 return room\_occupancy\_data  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error in room occupancy analytics: {str(e)}")  
 return None  
  
  
 def number\_of\_orders(self):  
 c = self.conn.cursor()  
 try:  
 c.execute("""  
 SELECT  
 room\_number,  
 COUNT(\*) AS orders\_count  
 FROM  
 booking\_ticket  
 WHERE  
 booking\_start\_date >= current\_date - interval '14 days'  
 GROUP BY  
 room\_number  
 ORDER BY  
 orders\_count DESC;  
  
 """)  
  
 number\_of\_orders\_data = c.fetchall() # Get data from the query  
  
 self.conn.commit()  
 return number\_of\_orders\_data  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error in analyzing the number of orders: {str(e)}")  
 return None  
  
 def client\_analytics(self):  
 c = self.conn.cursor()  
 try:  
 c.execute("""  
  
 WITH ClientBookingCounts AS (  
 SELECT   
 client.client\_id,   
 client.name,   
 client.surname,   
 client.email,   
 COUNT(booking\_ticket.booking\_id) AS booking\_count   
 FROM   
 client   
 JOIN   
 booking\_ticket ON client.client\_id = booking\_ticket.client\_id   
 GROUP BY   
 client.client\_id, client.name, client.surname, client.email  
 )  
   
   
 SELECT \*   
 FROM (  
 SELECT max(booking\_count) as max  
 FROM ClientBookingCounts  
 ) AS t1  
   
 INNER JOIN (  
 SELECT   
 client\_id,   
 name,   
 surname,   
 email,   
 booking\_count  
 FROM ClientBookingCounts  
 ) AS t2 ON t1.max = t2.booking\_count;  
   
 """)  
  
 number\_of\_orders\_data = c.fetchall() # Get data from the query  
  
 self.conn.commit()  
 return number\_of\_orders\_data  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error in customer analytics: {str(e)}")  
 return None

**Booking/model.py**

# ModelBookingTicket  
#################################################################################  
class ModelBookingTicket:  
 def \_\_init\_\_(self, db\_model):  
 self.conn = db\_model.conn  
  
 def add\_booking\_ticket(self, booking\_id, client\_id, room\_number, booking\_start\_date, booking\_end\_date, price):  
 c = self.conn.cursor()  
 try:  
 # Check if client\_id and room\_number match parent tables  
 c.execute('SELECT 1 FROM client WHERE client\_id = %s', (client\_id,))  
 client\_exists = c.fetchone()  
  
 c.execute('SELECT 1 FROM room WHERE room\_number = %s', (room\_number,))  
 room\_exists = c.fetchone()  
  
 if not client\_exists or not room\_exists:  
 # Return an exception notification and throw an error  
 return False # Or throw an exception to process it further  
 else:  
 # All checks have passed, insert into booking\_ticket  
 c.execute(  
 'INSERT INTO booking\_ticket (booking\_id, client\_id, room\_number, '  
 'booking\_start\_date, booking\_end\_date, price) VALUES (%s, %s, %s, %s, %s, %s)',  
 (booking\_id, client\_id, room\_number, booking\_start\_date, booking\_end\_date, price))  
 self.conn.commit()  
 return True  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error when adding a booking: {str(e)}")  
 return False  
  
 def get\_all\_booking\_tickets(self):  
 c = self.conn.cursor()  
 c.execute('SELECT \* FROM booking\_ticket')  
 return c.fetchall()  
  
 def update\_booking\_ticket(self, booking\_id, client\_id, room\_number, booking\_start\_date, booking\_end\_date, price):  
 c = self.conn.cursor()  
 try:  
 # Attempting to update a record  
 c.execute('UPDATE booking\_ticket SET client\_id=%s, room\_number=%s, booking\_start\_date=%s, '  
 'booking\_end\_date=%s, price=%s WHERE booking\_id=%s',  
 (client\_id, room\_number, booking\_start\_date, booking\_end\_date, price, booking\_id))  
 self.conn.commit()  
 return True # Returns True if the update was successful  
 except Exception as e:  
 # Handling an error if the update failed  
 self.conn.rollback()  
 print(f"Error when updating a reservation: {str(e)}")  
 return False # Returns False if insertion fails  
  
 def delete\_booking\_ticket(self, booking\_id):  
 c = self.conn.cursor()  
 try:  
 # Attempting to update a record  
 c.execute('DELETE FROM booking\_ticket WHERE booking\_id=%s', (booking\_id,))  
 self.conn.commit()  
 return True # Returns True if the update was successful  
 except Exception as e:  
 # Handling an error in case the deletion failed  
 self.conn.rollback()  
 print(f"Error when deleting a reservation: {str(e)}")  
 return False # Returns False if insertion fails  
  
 def check\_booking\_existence(self, booking\_id):  
 c = self.conn.cursor()  
 c.execute("SELECT 1 FROM booking\_ticket WHERE booking\_id = %s", (booking\_id,))  
 return bool(c.fetchone())  
  
 def create\_booking\_sequence(self):  
 c = self.conn.cursor()  
 c.execute("""  
 DO $$  
 BEGIN  
 IF NOT EXISTS (SELECT 1 FROM pg\_sequences WHERE schemaname = 'public' AND sequencename = 'booking\_id\_seq') THEN  
 CREATE SEQUENCE booking\_id\_seq;  
 ELSE  
 DROP SEQUENCE booking\_id\_seq;  
 CREATE SEQUENCE booking\_id\_seq;  
 END IF;  
 END $$;  
 """)  
 self.conn.commit()  
  
 def generate\_rand\_booking\_ticket\_data(self, number\_of\_operations):  
 c = self.conn.cursor()  
 try:  
 c.execute("""  
 INSERT INTO booking\_ticket (booking\_id, client\_id, room\_number, booking\_start\_date, booking\_end\_date, price)  
 SELECT  
 nextval('booking\_id\_seq'),  
 floor(random() \* (SELECT max(client\_id) FROM client) + 1),  
 floor(random() \* (SELECT max(room\_number) FROM room) + 1),  
 '2023-01-01'::date + floor(random() \* (date '2023-11-05' - date '2023-01-01')) \* interval '1 day',  
 '2023-11-05'::date + floor(random() \* (date '2025-01-01' - date '2023-11-05')) \* interval '1 day',  
 random() \* 1000   
 FROM generate\_series(1, %s)  
 """, (number\_of\_operations,))  
  
 self.conn.commit()  
 return True  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error while generating booking tickets: {str(e)}")  
 return False  
  
  
 def truncate\_booking\_table(self):  
 c = self.conn.cursor()  
 try:  
 # Insert data  
 c.execute("""DELETE FROM booking\_ticket""")  
 self.conn.commit()  
 return True # Returns True if the insertion was successful  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error when adding a client: {str(e)}")  
 return False # Returns False if insertion fails

**Client/model.py**

# ModelClient  
############################################################################  
class ModelClient:  
 def \_\_init\_\_(self, db\_model):  
 self.conn = db\_model.conn  
  
 def add\_client(self, client\_id, name, surname, email):  
 c = self.conn.cursor()  
 try:  
 c.execute('INSERT INTO client (client\_id, name, surname, email) VALUES (%s, %s, %s, %s)',  
 (client\_id, name, surname, email))  
 self.conn.commit()  
 return True # Returns True if the update was successful  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error when adding a client: {str(e)}")  
 return False # Returns False if insertion fails  
  
 def get\_all\_clients(self):  
 c = self.conn.cursor()  
 c.execute('SELECT \* FROM client')  
 return c.fetchall()  
  
 def update\_client(self, client\_id, name, surname, email):  
 c = self.conn.cursor()  
 try:  
 c.execute('UPDATE client SET name=%s, surname=%s, email=%s WHERE client\_id=%s',  
 (name, surname, email, client\_id))  
 self.conn.commit()  
 return True # Returns True if the update was successful  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error when updating the client: {str(e)}")  
 return False # Returns False if insertion fails  
  
 def delete\_client(self, client\_id):  
 c = self.conn.cursor()  
 try:  
 c.execute('DELETE FROM client WHERE client\_id=%s', (client\_id,))  
 self.conn.commit()  
 return True # Returns True if the update was successful  
 except Exception as e:  
 self.conn.rollback()  
 print(f"An error when deleting a client breaks the foreign key restriction: {str(e)}")  
 return False # Returns False if insertion fails  
  
 def check\_client\_existence(self, client\_id):  
 c = self.conn.cursor()  
 c.execute("SELECT 1 FROM client WHERE client\_id = %s", (client\_id,))  
 return bool(c.fetchone())  
  
 def create\_client\_sequence(self):  
 # Check for the existence of a sequence  
 c = self.conn.cursor()  
 c.execute("""  
 DO $$  
 BEGIN  
 IF NOT EXISTS (SELECT 1 FROM pg\_sequences WHERE schemaname = 'public' AND sequencename = 'client\_id\_seq') THEN  
 -- Якщо послідовності не існує, створюємо її  
 CREATE SEQUENCE client\_id\_seq;  
 ELSE  
 -- Якщо послідовність існує, видаляємо і створюємо нову  
 DROP SEQUENCE client\_id\_seq;  
 CREATE SEQUENCE client\_id\_seq;  
 END IF;  
 END $$;  
 """)  
 self.conn.commit()  
  
 def generate\_rand\_client\_data(self, number\_of\_operations):  
 c = self.conn.cursor()  
 try:  
 # Insert data  
 c.execute("""  
 INSERT INTO client (client\_id, name, surname, email)  
 SELECT  
 nextval('client\_id\_seq'),   
 (array['John', 'Alice', 'Bob', 'Emma', 'Michael'])[floor(random() \* 5) + 1],   
 (array['Smith', 'Johnson', 'Brown', 'Davis', 'Wilson'])[floor(random() \* 5) + 1],   
 (substr(md5(random()::text), 1, 10) || '@gmail.com')   
 FROM generate\_series(1, %s);  
 """, (number\_of\_operations,))  
 self.conn.commit()  
 return True # Returns True if the update was successful  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error when adding a client: {str(e)}")  
 return False # Returns False if insertion fails  
  
  
 def truncate\_client\_table(self):  
 c = self.conn.cursor()  
 try:  
 # Insert data  
 c.execute("""DELETE FROM client""")  
 self.conn.commit()  
 return True # Returns True if the update was successful  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error when adding a client: {str(e)}")  
 return False # Returns False if insertion fails

**Room/model.py**

# ModelRoom  
##############################################################################  
class ModelRoom:  
 def \_\_init\_\_(self, db\_model):  
 self.conn = db\_model.conn  
  
 def add\_room(self, room\_number, room\_type):  
 c = self.conn.cursor()  
 try:  
 c.execute('INSERT INTO room (room\_number ,room\_type) VALUES (%s, %s)', (room\_number, room\_type,))  
 self.conn.commit()  
 return True # Returns True if the update was successful  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error when adding a room: {str(e)}")  
 return False # Returns False if insertion fails  
  
 def get\_all\_rooms(self):  
 c = self.conn.cursor()  
 c.execute('SELECT \* FROM room')  
 return c.fetchall()  
  
 def update\_room(self, room\_number, room\_type):  
 c = self.conn.cursor()  
 try:  
 c.execute('UPDATE room SET room\_type=%s WHERE room\_number=%s', (room\_type, room\_number))  
 self.conn.commit()  
 return True # Returns True if the update was successful  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error when updating a room: {str(e)}")  
 return False # Returns False if insertion fails  
  
 def delete\_room(self, room\_number):  
 c = self.conn.cursor()  
 try:  
 c.execute('DELETE FROM room WHERE room\_number=%s', (room\_number,))  
 self.conn.commit()  
 return True # Returns True if the update was successful  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error when deleting a room: {str(e)}")  
 return False # Returns False if insertion fails  
  
 def check\_room\_existence(self, room\_number):  
 c = self.conn.cursor()  
 c.execute('SELECT 1 FROM room WHERE room\_number = %s', (room\_number,))  
 return c.fetchone() is not None  
  
 def create\_room\_sequence(self):  
 # Check for the existence of a sequence  
 c = self.conn.cursor()  
 c.execute("""  
 DO $$  
 BEGIN  
 IF NOT EXISTS (SELECT 1 FROM pg\_sequences WHERE schemaname = 'public' AND sequencename = 'room\_number\_seq') THEN  
 -- Якщо послідовності не існує, створюємо її  
 CREATE SEQUENCE room\_number\_seq;  
 ELSE  
 -- Якщо послідовність існує, видаляємо і створюємо нову  
 DROP SEQUENCE room\_number\_seq;  
 CREATE SEQUENCE room\_number\_seq;  
 END IF;  
 END $$;  
 """)  
 self.conn.commit()  
 def generate\_rand\_room\_data(self, number\_of\_operations):  
 c = self.conn.cursor()  
 try:  
 # Insert data  
 c.execute("""  
 INSERT INTO room (room\_number, room\_type)  
 SELECT  
 nextval('room\_number\_seq'),   
 (array['Single', 'Double', 'Suite'])[floor(random() \* 3) + 1]  
 FROM generate\_series(1, %s);  
 """, (number\_of\_operations,))  
 self.conn.commit()  
 return True # Returns True if the insertion was successful  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error when adding a room: {str(e)}")  
 return False # Returns False if insertion fails  
  
 def truncate\_room\_table(self):  
 c = self.conn.cursor()  
 try:  
 # Insert data  
 c.execute("""DELETE FROM room""")  
 self.conn.commit()  
 return True # Returns True if the update was successful  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error when adding a client: {str(e)}")  
 return False # Returns False if insertion fails

**model.py**

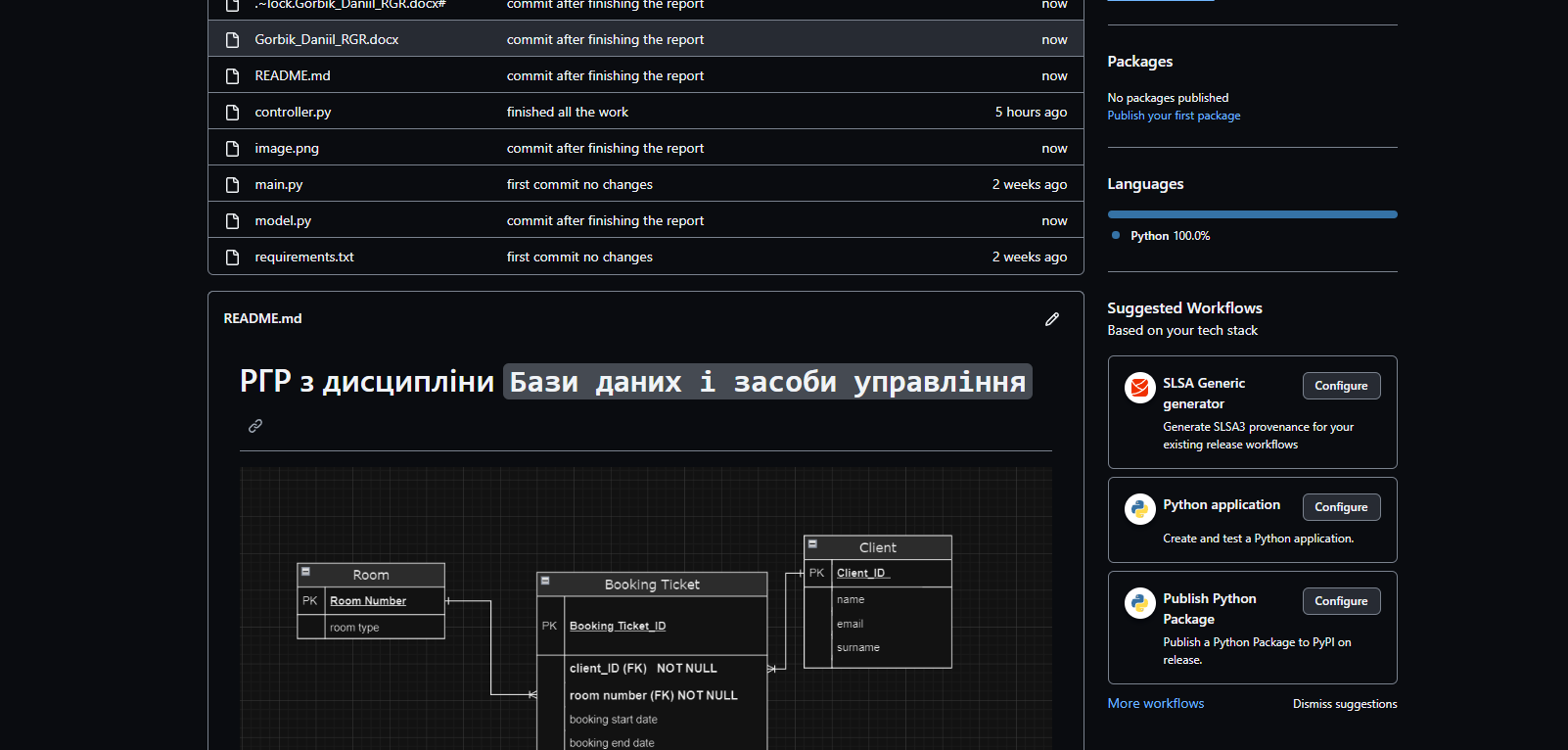
import psycopg2  
  
  
class Model:  
 def \_\_init\_\_(self):  
 self.conn = psycopg2.connect(  
 dbname='postgres',  
 user='postgres',  
 password='1',  
 host='localhost',  
 port=5432  
 )  
 self.create\_tables()  
  
 def create\_tables(self):  
 c = self.conn.cursor()  
 # Check for tables  
 c.execute("SELECT EXISTS (SELECT 1 FROM information\_schema.tables WHERE table\_name = 'booking\_ticket')")  
 booking\_ticket\_table\_exists = c.fetchone()[0]  
  
 c.execute("SELECT EXISTS (SELECT 1 FROM information\_schema.tables WHERE table\_name = 'client')")  
 client\_table\_exists = c.fetchone()[0]  
  
 c.execute("SELECT EXISTS (SELECT 1 FROM information\_schema.tables WHERE table\_name = 'room')")  
 room\_table\_exists = c.fetchone()[0]  
  
 if not booking\_ticket\_table\_exists:  
 c.execute('''  
 CREATE TABLE booking\_ticket (  
 booking\_id SERIAL PRIMARY KEY,  
 client\_id INTEGER NOT NULL,  
 room\_number INTEGER NOT NULL,  
 booking\_start\_date DATE NOT NULL,  
 booking\_end\_date DATE NOT NULL,  
 price DECIMAL(10, 2) NOT NULL  
 )  
 ''')  
 if not client\_table\_exists:  
 c.execute('''  
 CREATE TABLE client (  
 client\_id SERIAL PRIMARY KEY,  
 name TEXT NOT NULL,  
 surname TEXT NOT NULL,  
 email TEXT  
 )  
 ''')  
 if not room\_table\_exists:  
 c.execute('''  
 CREATE TABLE room (  
 room\_number SERIAL PRIMARY KEY,  
 room\_type TEXT NOT NULL  
 )  
 ''')  
  
 self.conn.commit()

**Короткий опис функцій**

Програма ділиться на 4 папки і головні файли model, controller, main.  
Файли:

1. Analytics – відповідає за 3 пункт РГР
2. Booking – відповідає за таблицю booking
3. Client – відповідає за таблицю client
4. Room – відповідає за таблицю room

**Ілюстрації програмного коду на Github**

  
  
Посилання на репозиторій:

https://github.com/Dan-live/c3s1\_RGR\_DB/tree/main