Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих

комп’ютерних систем

ЛАБОРАТОРНА РОБОТА №2

З ДИСЦИПЛІНИ “БАЗИ ДАНИХ ТА ЗАСОБИ УПРАВЛІННЯ”

ТЕМА: “Засоби оптимізації роботи СУБД PostgreSQL ”

Виконав: Зорєв М.А

Група: КВ-11

Оцінка

Київ - 2023

Завдання:

1. Перетворити модуль “Модель” з шаблону MVC РГР у вигляд об’єктно-реляційної проекції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL.
3. Розробити тригер бази даних PostgreSQL.
4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

Завдання 1

Коли мова йде про розробку Java-застосунків, що взаємодіють з базами даних, справи можуть ускладнюватися. Управління даними та взаємодія з базами даних може бути складним завданням. Тут на допомогу приходить техніка Об'єктно-Реляційного Відображення (ORM).

ORM — це корисний підхід, що з'єднує світ програмування на Java з базами даних. Він спрощує процес, відображаючи таблиці бази даних на класи Java, щоб розробники могли працювати з даними у більш зрозумілий, об'єктно-орієнтований спосіб.

Java Persistence API (JPA) — це набір правил, які визначають, як використовувати ORM у Java-застосунках. Він надає стандартний спосіб роботи з базами даних, незалежно від конкретного ORM-фреймворку, який використовується.

Серед популярних ORM-фреймворків для Java виділяється Hibernate. Hibernate є відкритим фреймворком, який дотримується правил JPA та пропонує додаткові можливості та поліпшення.

За допомогою Hibernate розробники можуть зосередитися на написанні коду Java, не займаючись складними запитаннями до бази даних. Hibernate вирішує завдання створення, оновлення та отримання даних, спрощуючи роботу з базами даних.

Hibernate також пропонує розширені можливості, такі як кешування, відкладене завантаження та управління транзакціями. Ці можливості покращують продуктивність та масштабованість Java-застосунків. Hibernate також надає потужну мову запитів, відому як Hibernate Query Language (HQL), що спрощує написання запитань до бази даних.

Отже, для виконання 1 завдання, було використано Spring Data JPA, пропоную розглянути реалізацію:

1. Спершу було створено класи-сутності для об’єктів-сутностей, що представлені в БД:

Таблиця для представлення об’єкту Dish:

```java
14 usages
@Entity
@NoArgsConstructor
@AllArgsConstructor
public class Dish {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long dishId;
    private String dishName;
    private int weight;
    private int price;
    private int servingsAmount;

    @ManyToOne
    @JoinColumn(name = "order_id", referencedColumnName = "orderId")
    private CustomerOrder order;
```

Таблиця для представлення об’єкту Client:

```java
@Entity
@AllArgsConstructor
@NoArgsConstructor
public class Client {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long clientId;
    private String clientEmailAddress;
    private String clientName;
    private String clientPhoneNumber;
```

Таблиця для представлення об’єкту Courier:

```java
@Entity
@AllArgsConstructor
@NoArgsConstructor
public class Courier {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long courierId;
    private String courierPhoneNumber;
    private String courierName;
    private String transportKind;
    private double courierRating;
```

Таблиця для представлення об’єкту CustomerOrder:

```java
18 usages
@Entity
@NoArgsConstructor
@AllArgsConstructor
public class CustomerOrder  {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long orderId;
    private int orderNumber;
    private String deliveryAddress;
    private Timestamp deliveryDateTime;

    @ManyToOne
    @JoinColumn(name = "client_id", referencedColumnName = "clientId")
    private Client client;

    @ManyToOne
    @JoinColumn(name = "courier_id", referencedColumnName = "courierId")
    private Courier courier;
```

Для реалізації зовнішніх зв’язків були використані поля, які мають анотації @ManyToOne та @JoinColumn. Вони дозволили прописати по якій колонці ми будемо з’єднувати таблиці, а також який тип зв’язку в нас буде.

2. Далі створюється репозиторії для цих сутностей, щоб мати можливість взаємодіяти з базою даних. Ці репозиторії наслідують JpaRepository, який приховує готові реалізації, нам потрібно всього лише написати для якої

моделі ми хочемо застосувати цей репозиторій. Ось їхні реалізації:

3. Репозиторій для Client:

```
4 usages
public interface ClientRepository extends JpaRepository<Client, Long> {
    1 usage
    @Query("SELECT c FROM Client c JOIN CustomerOrder o ON c.clientId = o.client.clientId GROUP BY c ORDER BY COUNT(o) DESC")
    List<Client> getClientsWithMostOrders();
}
```

Репозиторій для Courier:

```
3 usages
public interface CourierRepository extends JpaRepository<Courier, Long> {
}
```

Репозиторій для CustomerOrder:

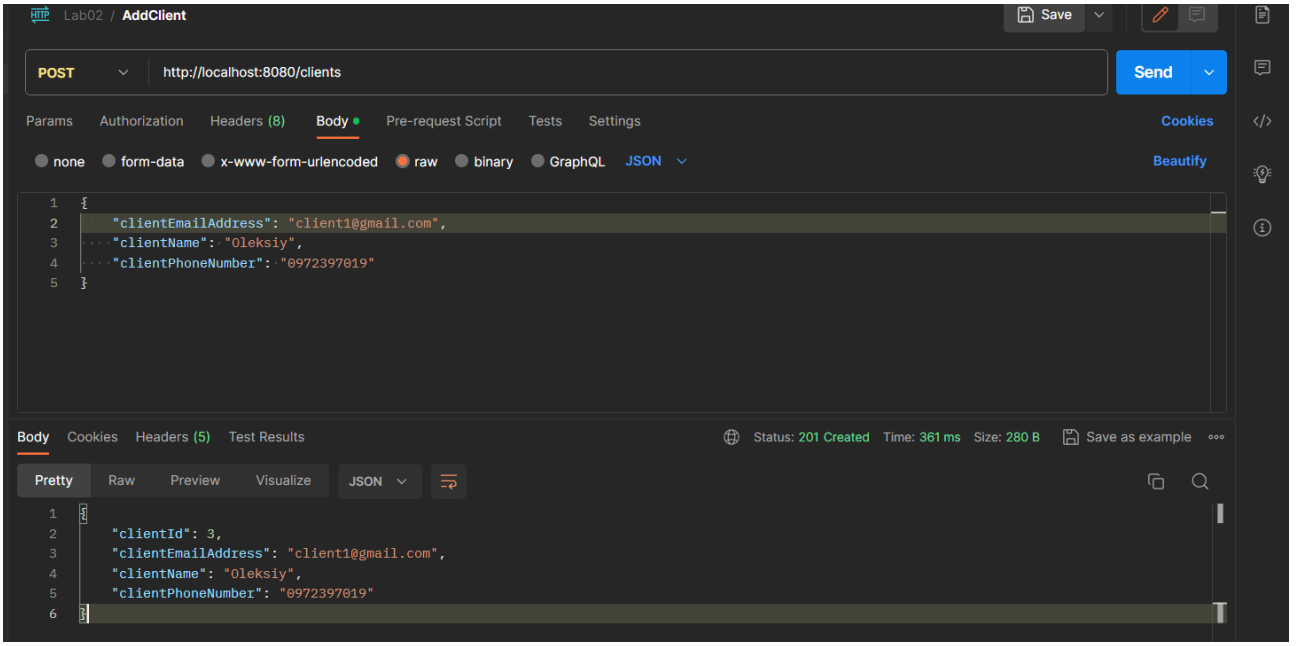
```
3 usages
public interface CustomerOrderRepository extends JpaRepository<CustomerOrder, Long> {

}
```

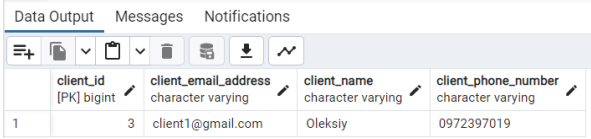
Репозиторій для Dish:

```
3 usages
public interface DishRepository extends JpaRepository<Dish, Long> {
}
```

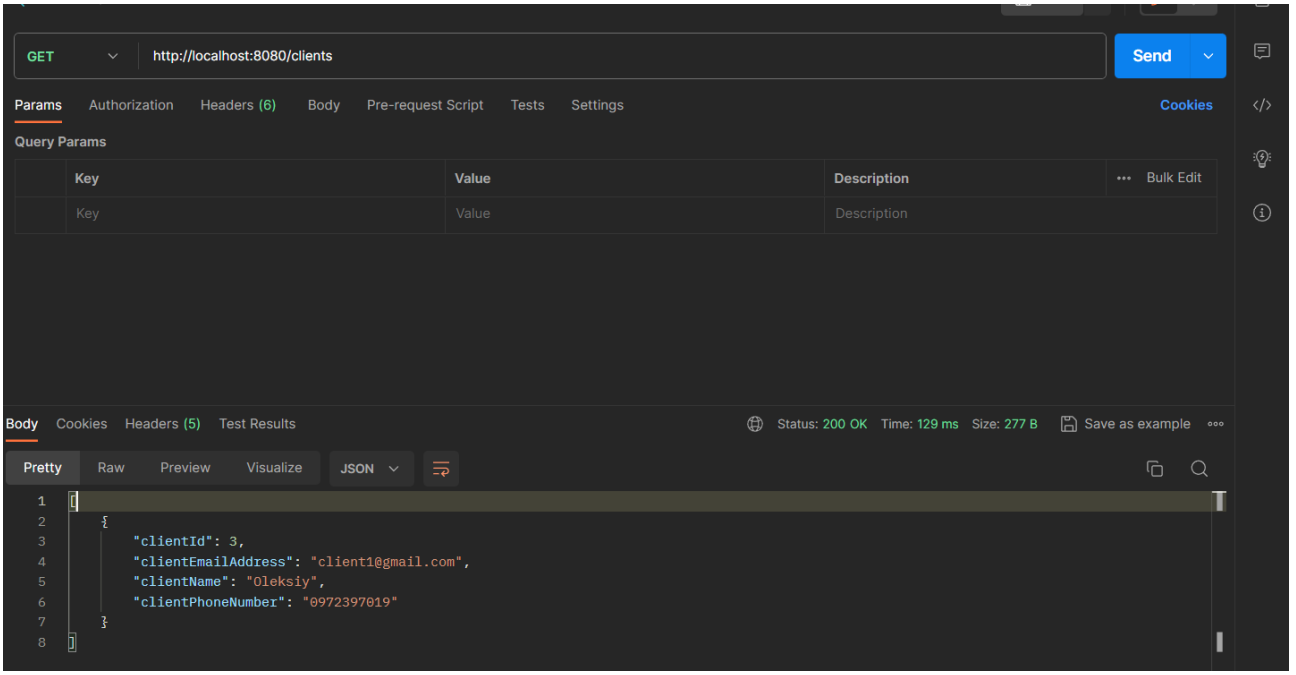
4. Для демонстрації коректної роботи програми було використавно Postman, в якому виконав різноманітні операції над класами, які вимагались в умові.
   Результати роботи:

5.
   Створення клієнта:

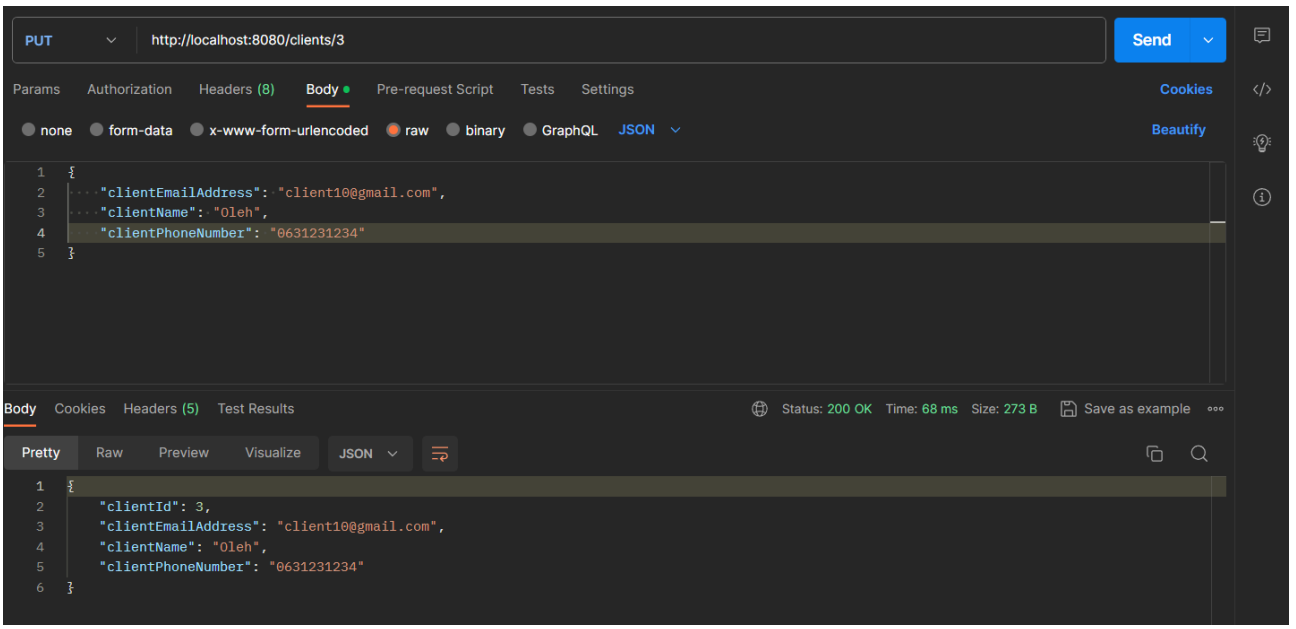


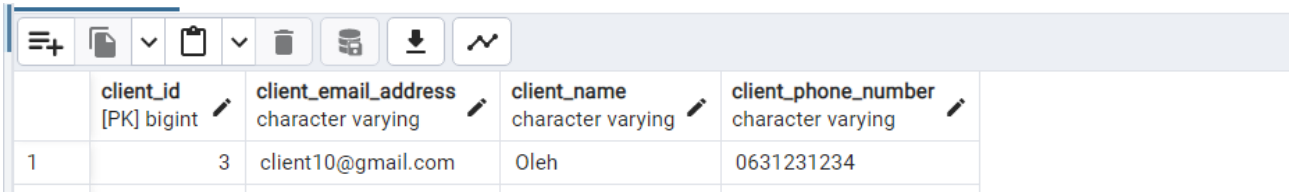


Отримання інформації про всіх клієнтів:



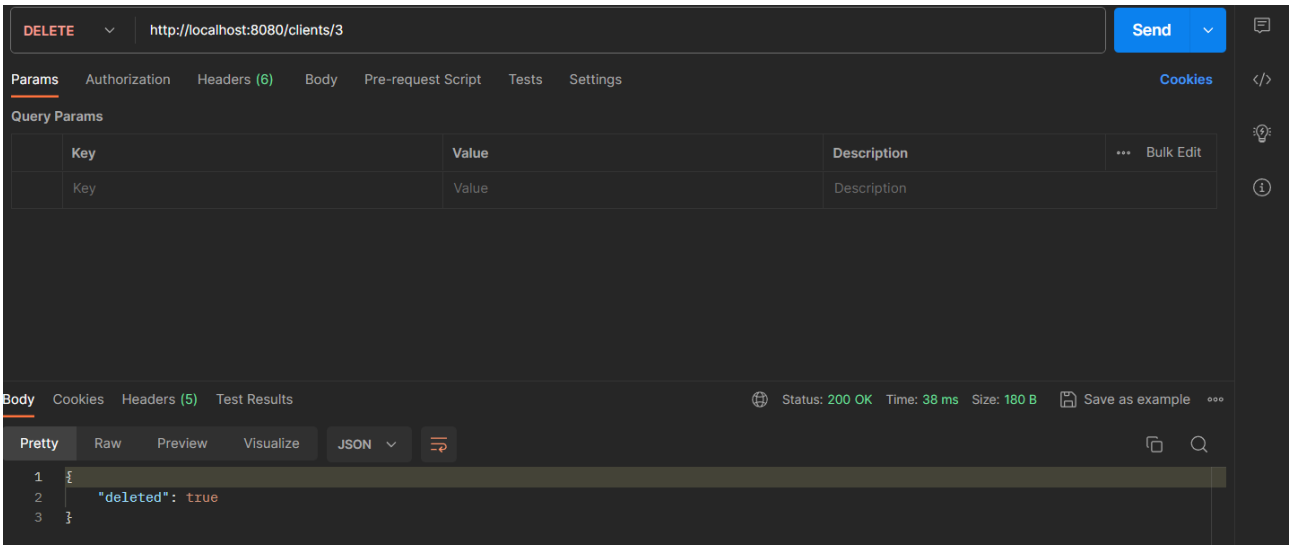
Оновлення клієнта:

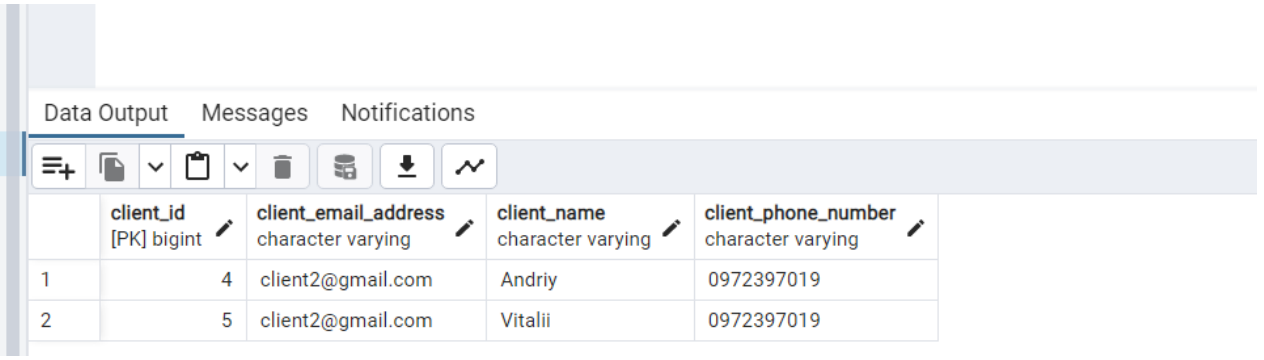




	client_id [PK] bigint	client_email_address character varying	client_name character varying	client_phone_number character varying
1	3	client10@gmail.com	Oleh	0631231234

Видалення клієнта:

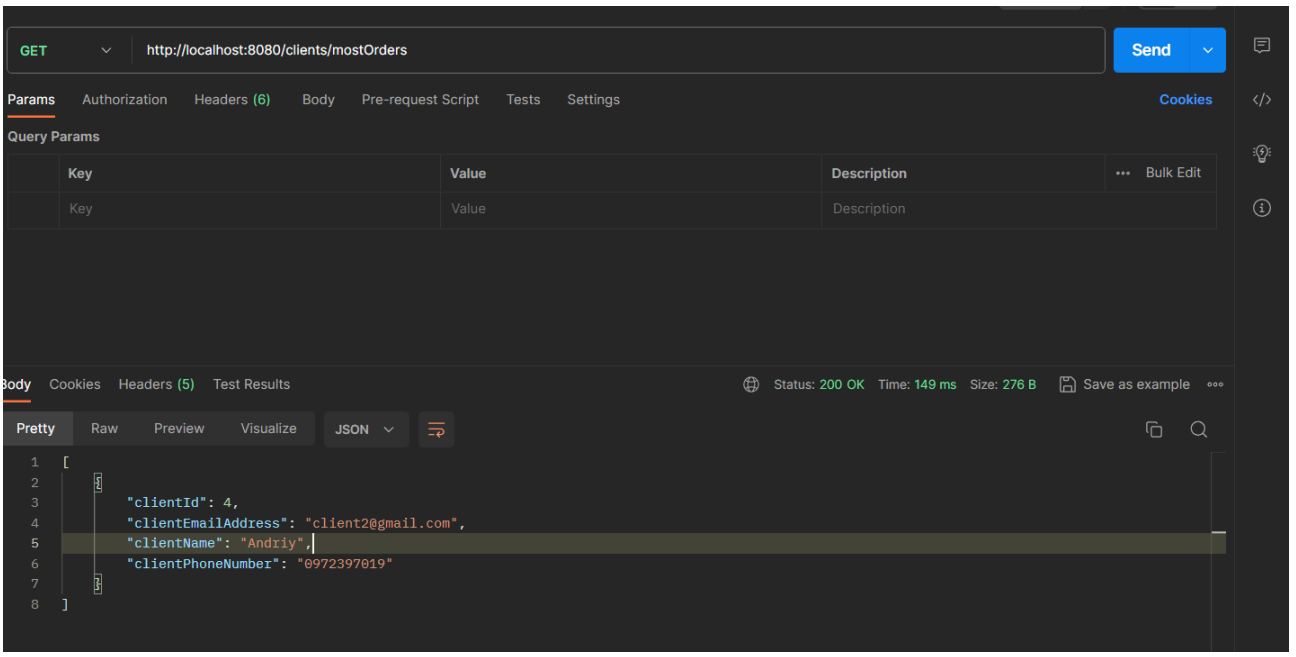




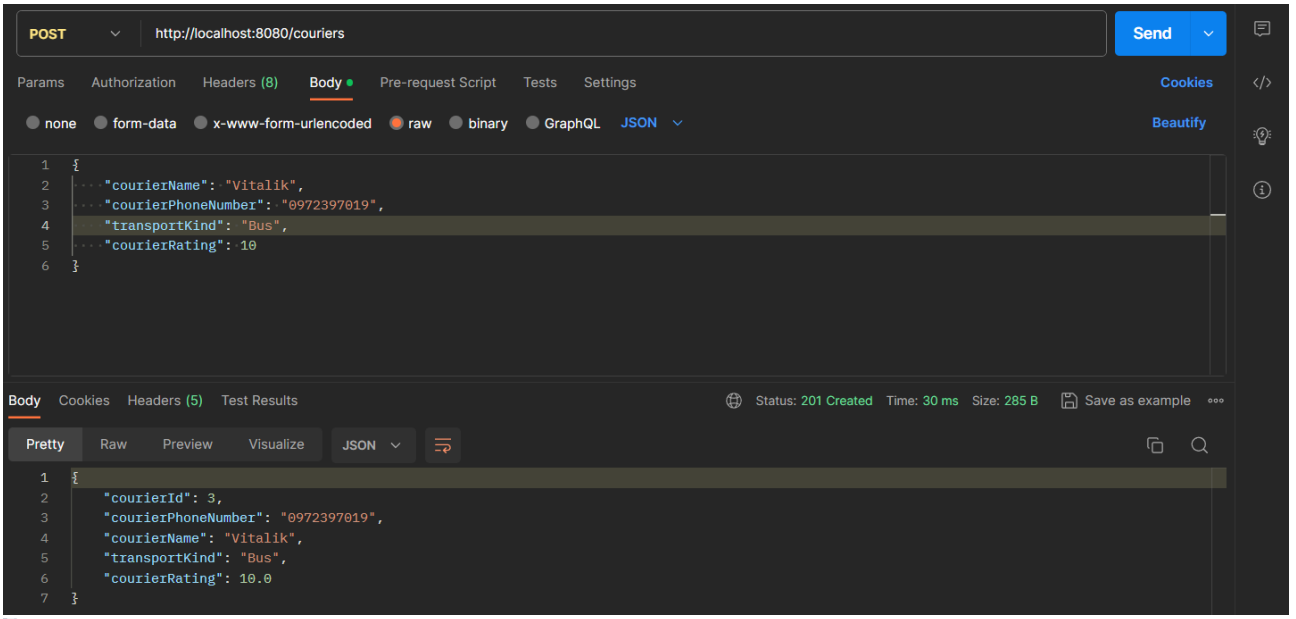
Data Output Messages Notifications

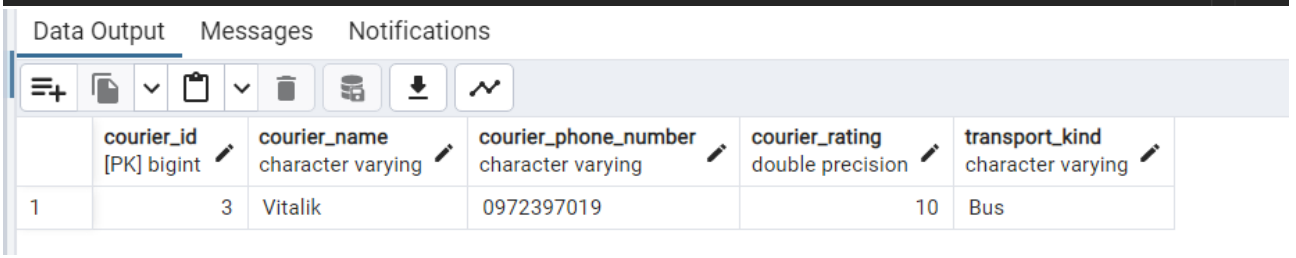
	client_id [PK] bigint	client_email_address character varying	client_name character varying	client_phone_number character varying
1	4	client2@gmail.com	Andriy	0972397019
2	5	client2@gmail.com	Vitalii	0972397019

Список клієнтів з найбільшою кількістю замовлень:

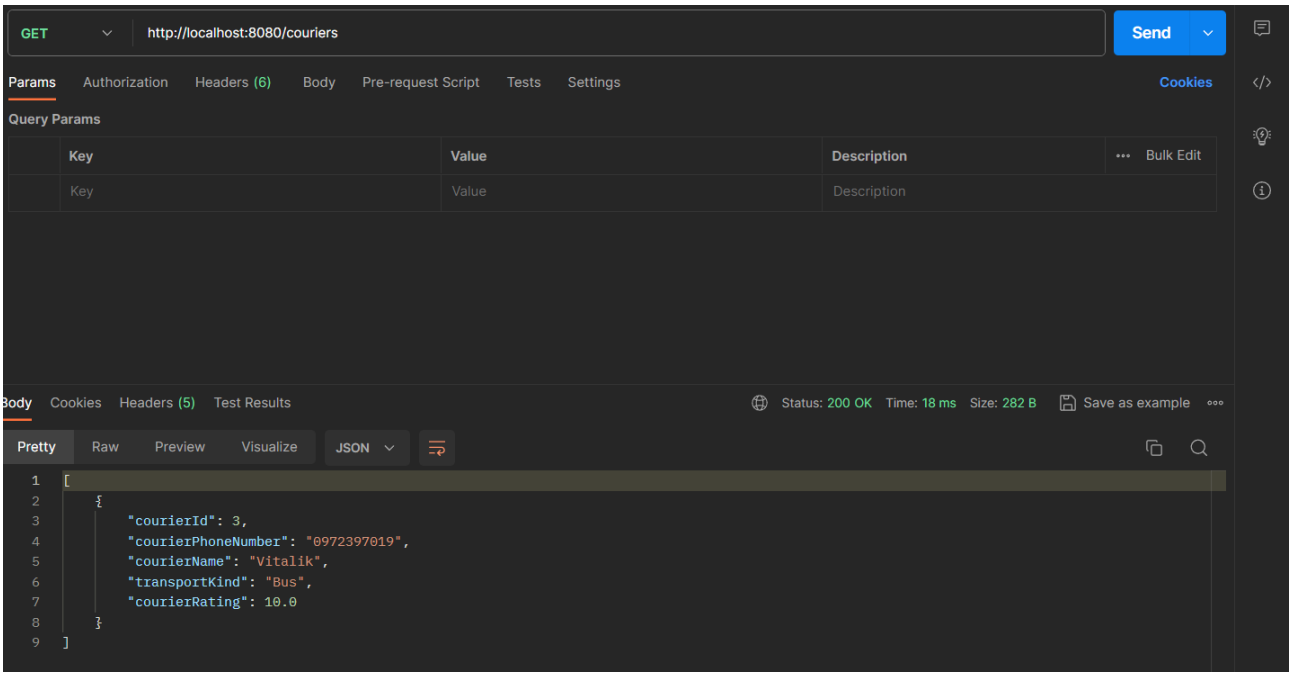


Створення кур’єра:

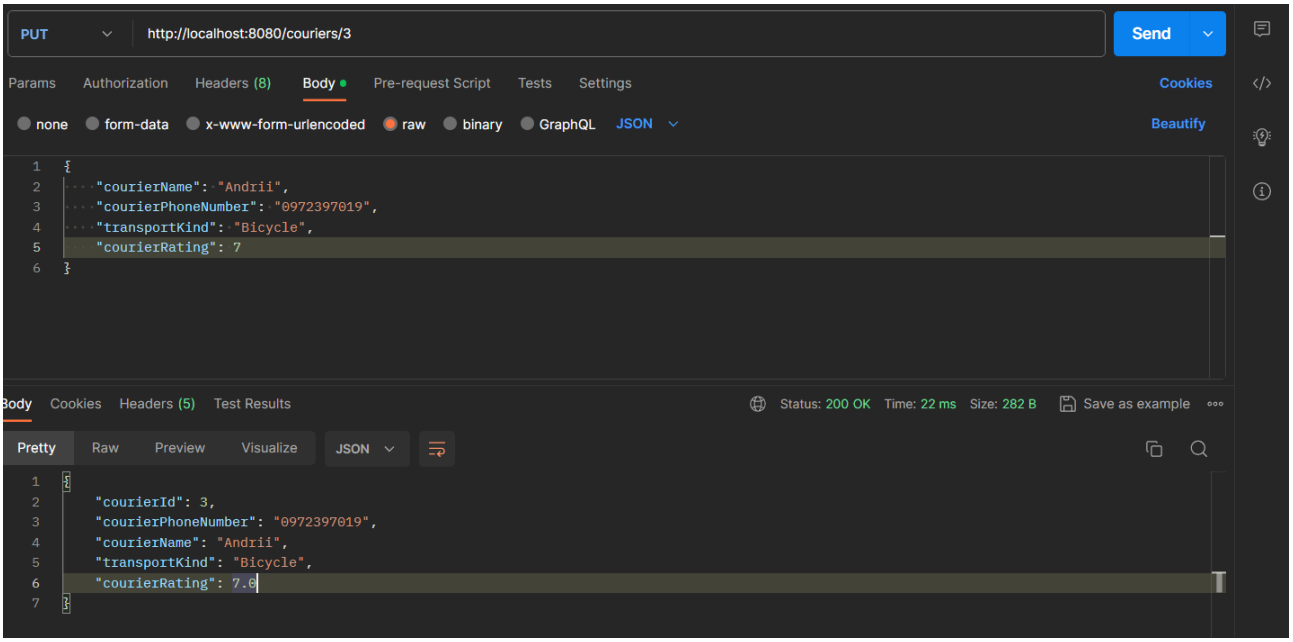




Отримання інформації про всіх кур’єрів:



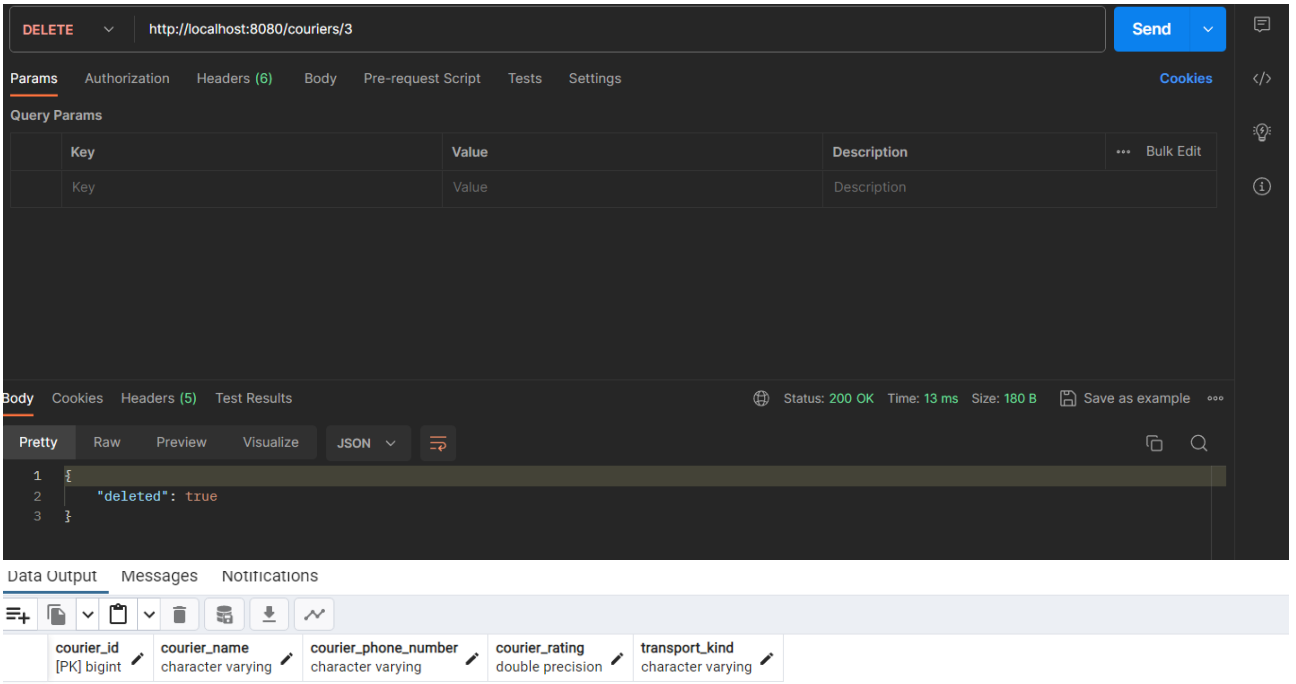
Оновлення кур’єра:



Data Output | Messages | Notifications

	courier_id [PK] bigint	courier_name character varying	courier_phone_number character varying	courier_rating double precision	transport_kind character varying
1	3	Andrii	0972397019	7	Bicycle

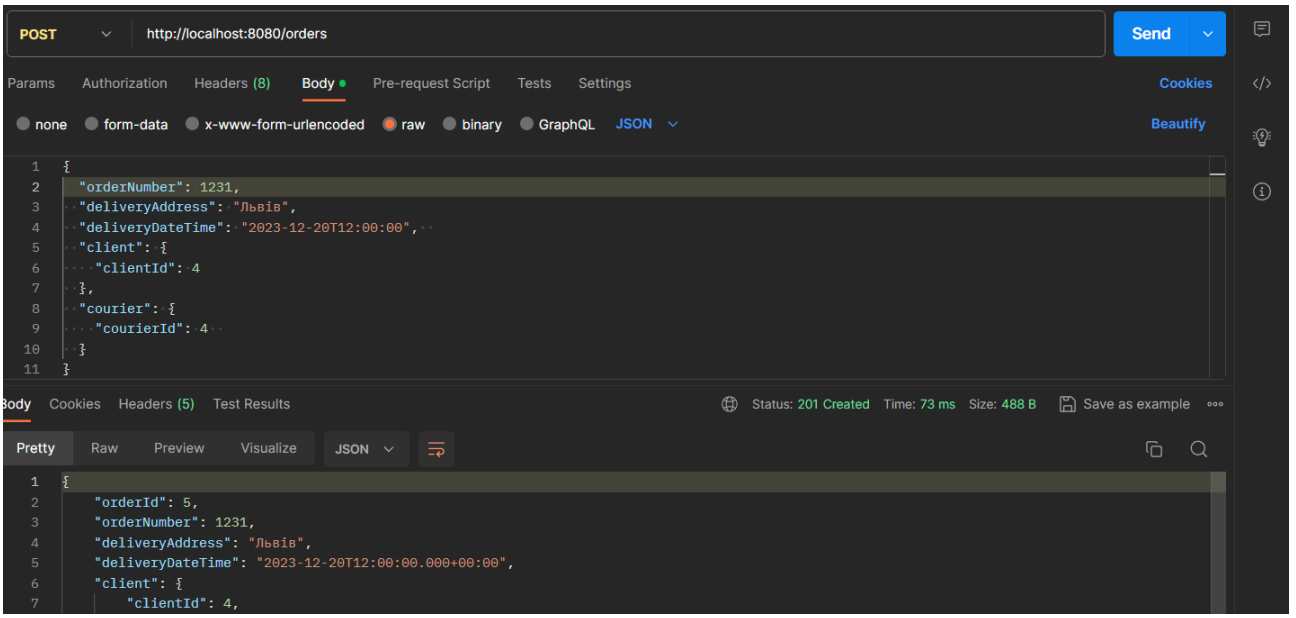
Видалення кур’єра:



Data Output | Messages | Notifications

courier_id [PK] bigint	courier_name character varying	courier_phone_number character varying	courier_rating double precision	transport_kind character varying

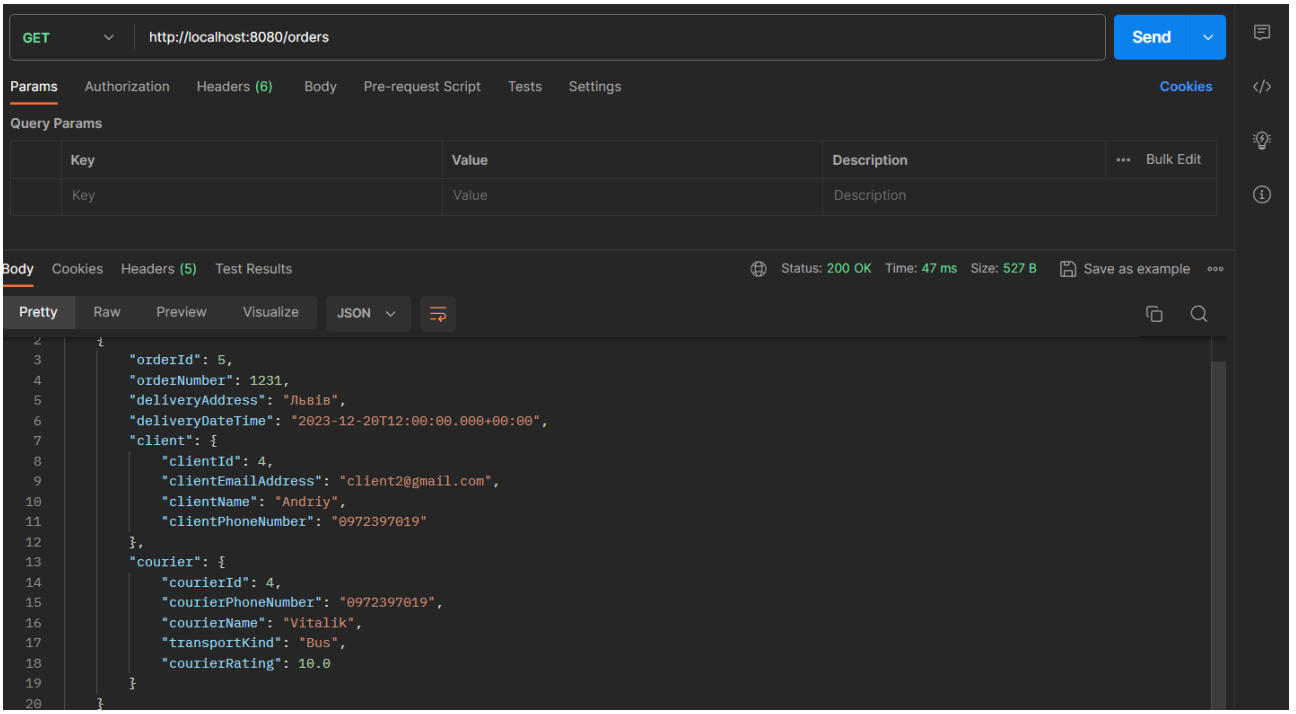
Створення замовлення:



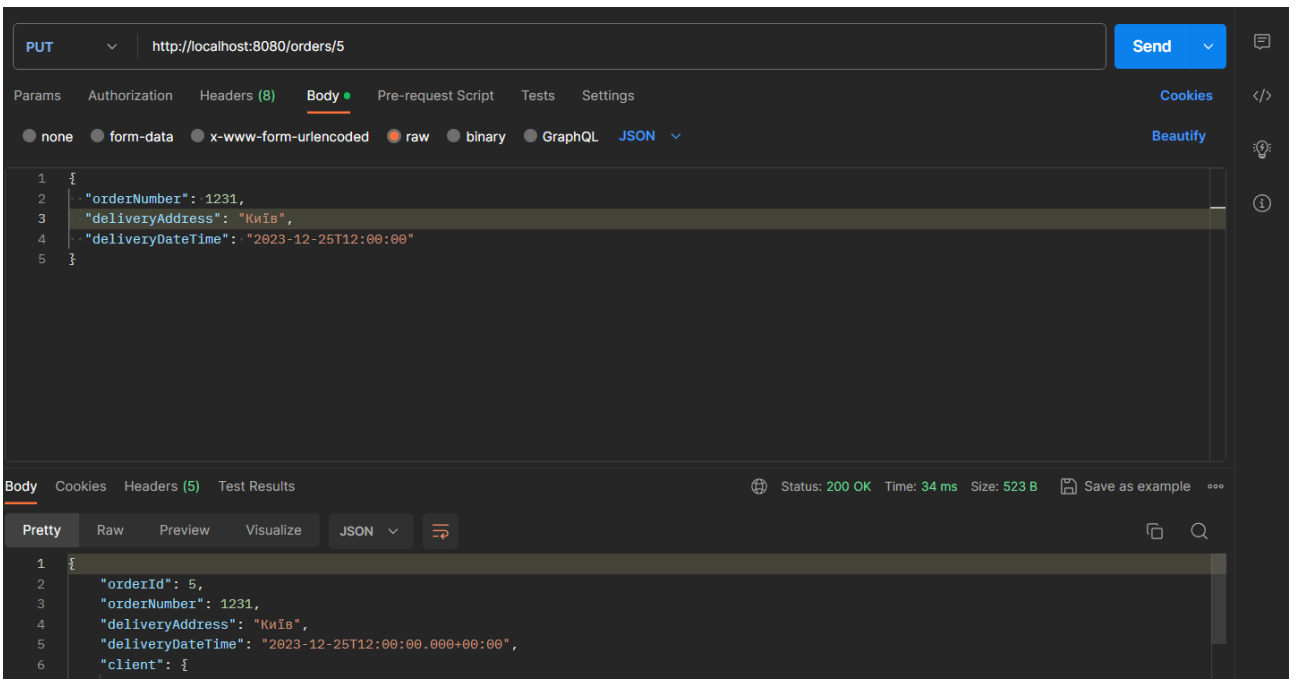
Data Output | Messages | Notifications

	order_id [PK] bigint	delivery_address character varying	delivery_date_time timestamp without time zone (6)	order_number integer	client_id bigint	courier_id bigint
1	5	Львів	2023-12-20 14:00:00	1231	4	4

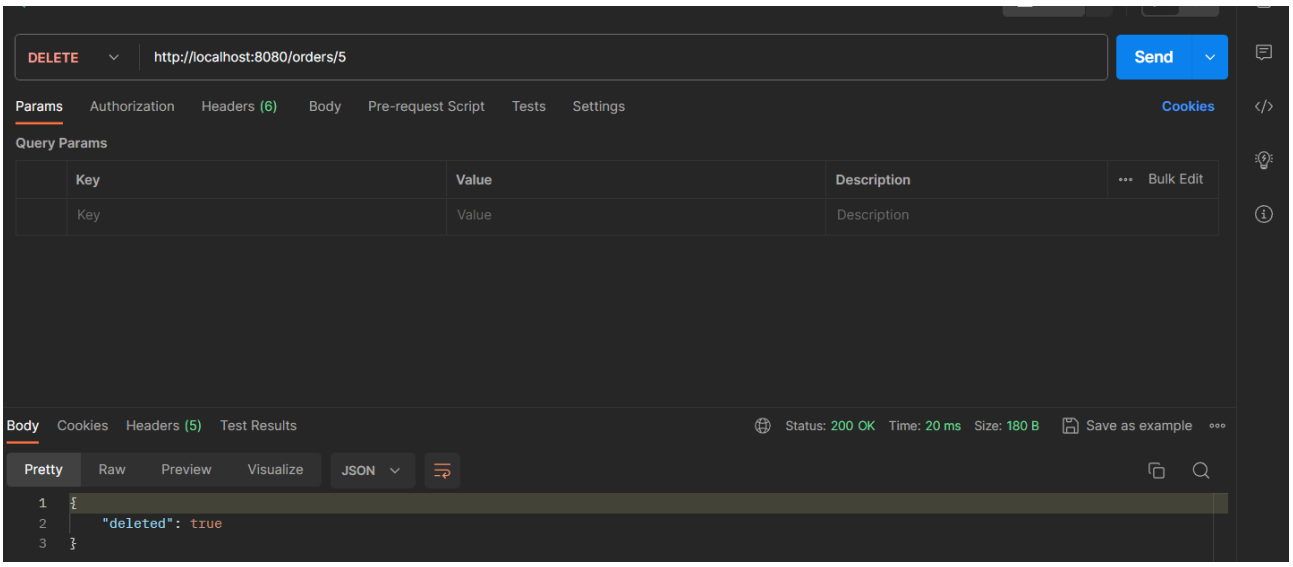
Отримання інформації про всі замовлення:



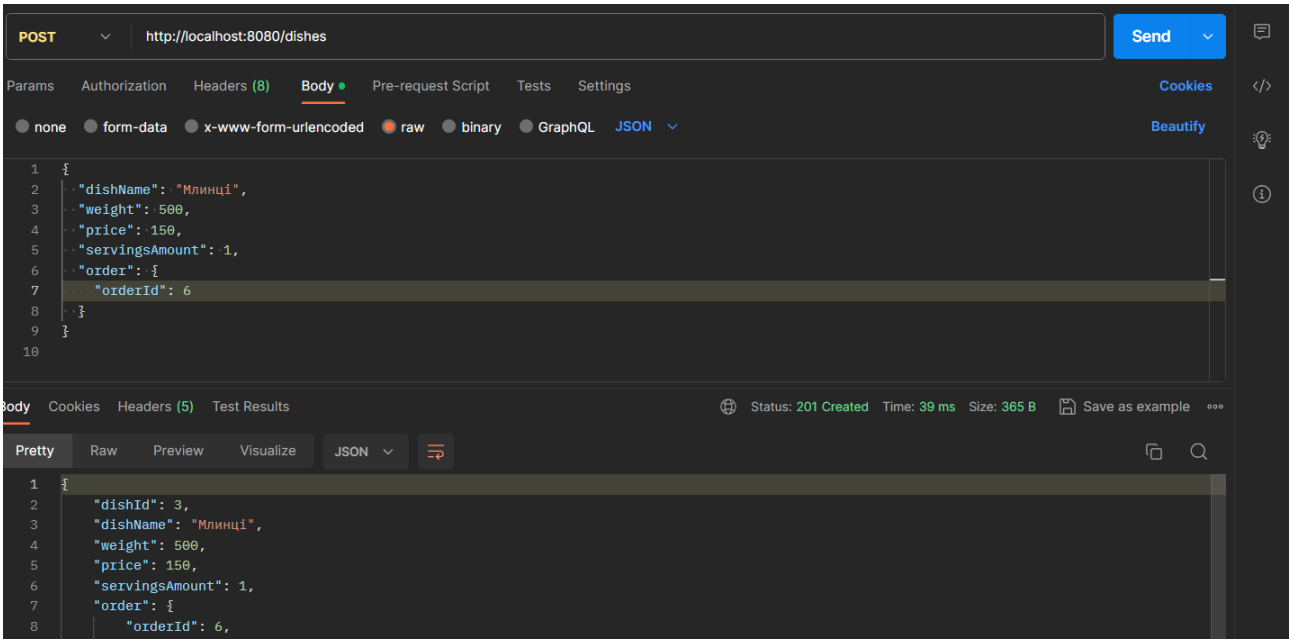
Оновлення замовлення:

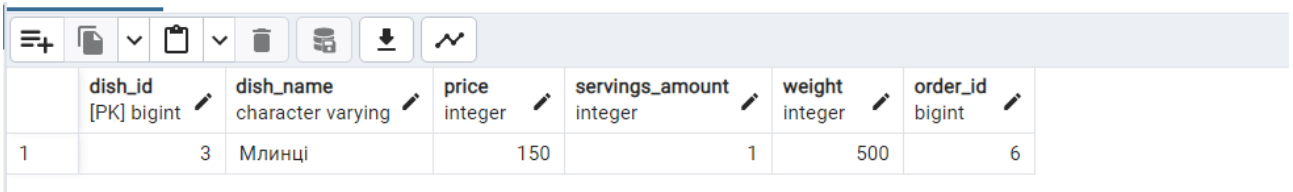


Видалення замовлення:



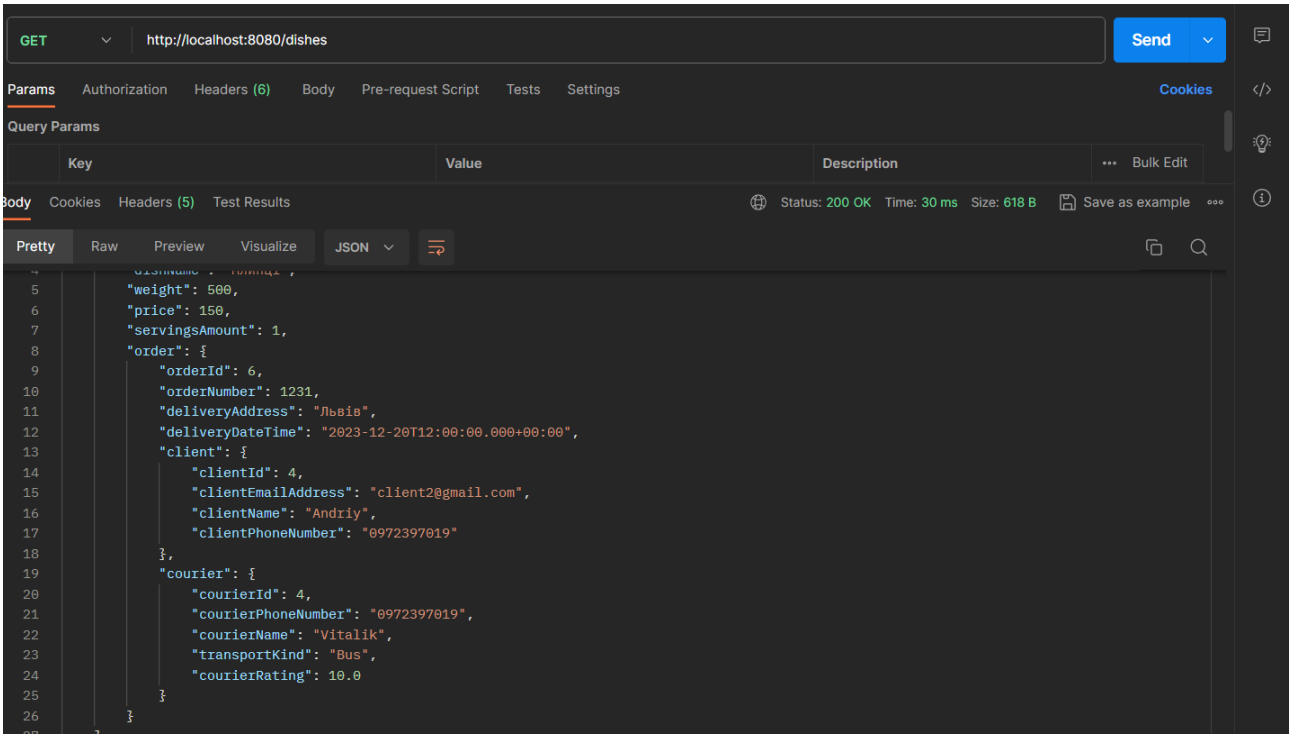
Створення страви:



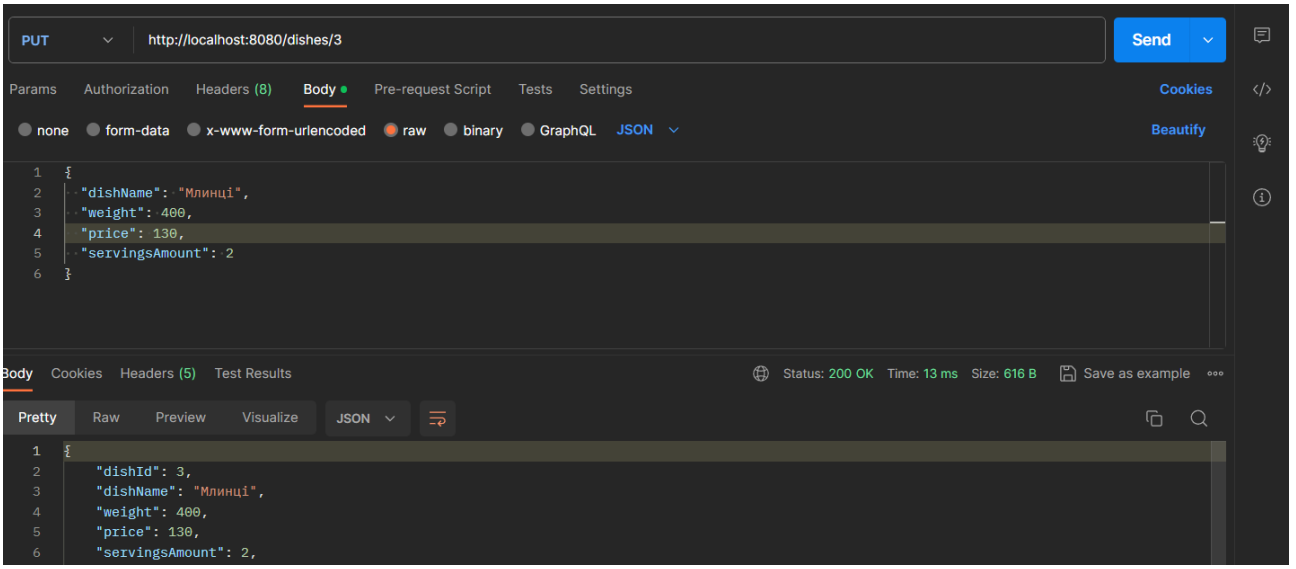


	dish_id [PK] bigint	dish_name character varying	price integer	servings_amount integer	weight integer	order_id bigint
1	3	Млинці	150	1	500	6

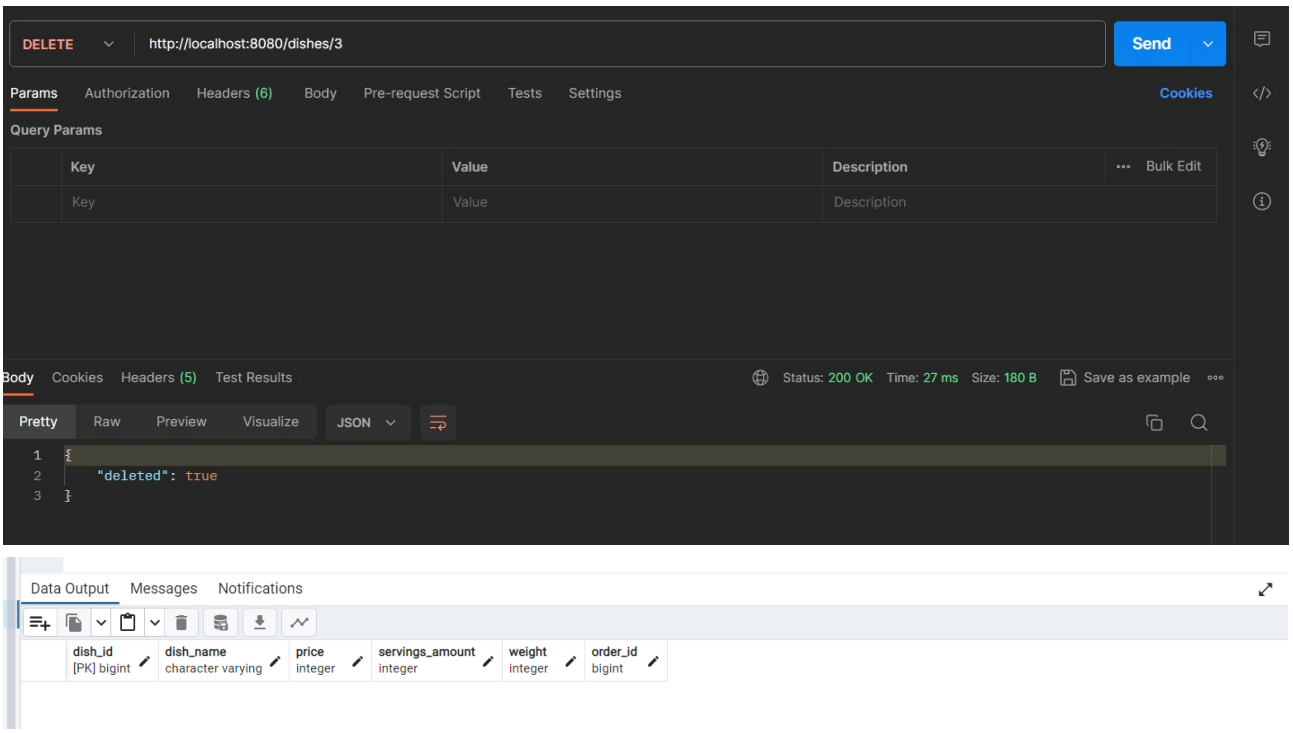
Отримання інформації про всі страви:



Оновлення страви:



Видалення страви:



Завдання 2

BRIN

Основна ідея Brin-індексу полягає в групуванні блоків даних у проміжки (range) і побудові індексу, який містить інформацію про кожен проміжок. Кожен проміжок представляє собою діапазон значень у відсортованих блоках даних. Для числових або датових даних, Brin-індекс може ефективно управляти діапазонами, що дозволяє прискорити виконання запитів, які використовують умови діапазонного пошуку.

Основні переваги Brin-індексу включають:

- Ефективність для великих обсягів даних: Brin-індекси особливо ефективні для таблиць з великою кількістю даних, коли інші типи індексів можуть виявитися менш ефективними.
- Економія місця: Цей тип індексу зазвичай вимагає менше місця в порівнянні з традиційними B-дерев'яними індексами, оскільки Brin-індекс не зберігає інформацію про кожен окремий рядок даних.
- Підтримка зжаття: Brin-індекс підтримує зжаття, що може додатково зменшити вимоги до місця.

GIN

GIN-індекси призначені для ефективної роботи з даними, що мають складні структури, такі як масиви чи текстові документи. Основна ідея GIN-індексу полягає в побудові інвертованого індексу для кожного елемента в масиві або кожного терміну в текстовому документі.

Основні властивості та переваги GIN-індексів:

- Масиви та текстові поля: GIN-індекси найчастіше використовуються для індексації масивів та текстових полів, де значення може містити багато елементів або термінів.
- Повний текстовий пошук: GIN-індекси є потужним інструментом для повного текстового пошуку в PostgreSQL. Вони дозволяють ефективно виконувати пошук за словами у текстових документах.
- Множинні значення: GIN-індекси підтримують множинні значення для одного рядка, дозволяючи ефективно індексувати та шукати дані, що мають багато значень.
- Пошук в масивах: Якщо у вас є стовпець, який містить масиви, GIN-індекс дозволить ефективно виконувати пошук за елементами цих масивів.

Далі перейдемо до прикладів створення та використання індексів, аналізу їх роботи:

1. **BRIN-індекс для стовпця courier_rating у таблиці courier:**

```
CREATE INDEX idx_courier_rating_brin ON public.courier USING BRIN(courier_rating);
```

Цей індекс може бути ефективним, якщо часто виконуються запити, які фільтрують кур'єрів за їхнім рейтингом.

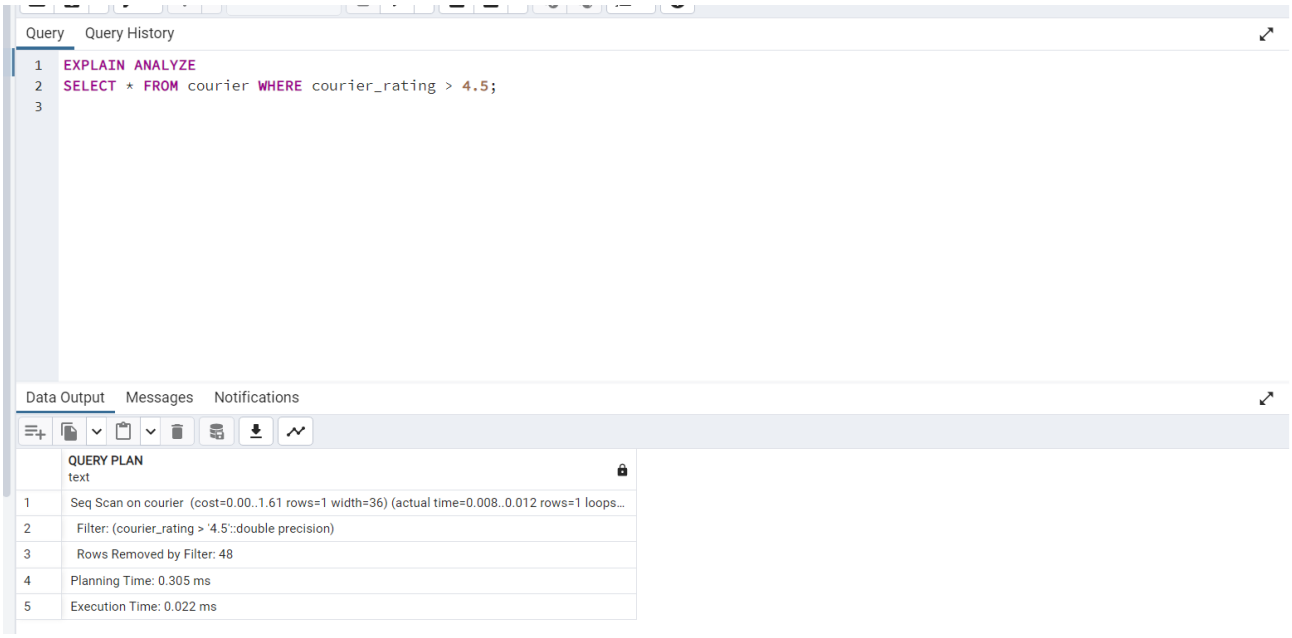
Ось результати роботи з та без індексу:

Query Query History

```
1  EXPLAIN ANALYZE
2  SELECT * FROM courier WHERE courier_rating > 4.5;
3
```

Data Output Messages Notifications

	QUERY PLAN text
1	Seq Scan on courier (cost=0.00..1.61 rows=1 width=36) (actual time=0.016..0.021 rows=1 loops...
2	Filter: (courier_rating > '4.5'::double precision)
3	Rows Removed by Filter: 48
4	Planning Time: 0.521 ms
5	Execution Time: 0.035 ms



Як бачимо, час з використанням індексу зменшився, отже він буде ефктивним в даній ситуації.

2. **BRIN-індекс для стовпця weight у таблиці dish:**

CREATE INDEX idx_dish_weight_brin ON public.dish USING BRIN(weight);

Цей індекс може бути корисним, якщо часто виконуються запити, які фільтрують страви за їхньою вагою.

```
EXPLAIN ANALYZE
SELECT AVG(weight) AS avg_weight, MAX(weight) AS max_weight
FROM dish
WHERE weight > 500
GROUP BY weight
ORDER BY weight DESC;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	GroupAggregate (cost=1.61..1.63 rows=1 width=40) (actual time=0.056..0.057 rows=0 loops=1)
2	Group Key: weight
3	-> Sort (cost=1.61..1.61 rows=1 width=4) (actual time=0.055..0.056 rows=0 loops=1)
4	Sort Key: weight DESC
5	Sort Method: quicksort Memory: 25kB
6	-> Seq Scan on dish (cost=0.00..1.60 rows=1 width=4) (actual time=0.030..0.031 rows=0 loops...
7	Filter: (weight > 500)
8	Rows Removed by Filter: 48
9	Planning Time: 0.592 ms
10	Execution Time: 0.100 ms

Query | Query History

```
-- Запит з сортуванням та агрегацією
EXPLAIN ANALYZE
SELECT AVG(weight) AS avg_weight, MAX(weight) AS max_weight
FROM dish
WHERE weight > 500
GROUP BY weight
ORDER BY weight DESC;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	GroupAggregate (cost=1.61..1.63 rows=1 width=40) (actual time=0.035..0.036 rows=0 loops=1)
2	Group Key: weight
3	-> Sort (cost=1.61..1.61 rows=1 width=4) (actual time=0.034..0.034 rows=0 loops=1)
4	Sort Key: weight DESC
5	Sort Method: quicksort Memory: 25kB
6	-> Seq Scan on dish (cost=0.00..1.60 rows=1 width=4) (actual time=0.021..0.022 rows=0 loops...
7	Filter: (weight > 500)
8	Rows Removed by Filter: 48
9	Planning Time: 0.424 ms
10	Execution Time: 0.070 ms

Знову можемо спостерігати, що наявність індекса пришвидшує пошук з агрегуючими функціями.

3. **GIN-індекс для текстового вектора transport_kind в таблиці courier:**

CREATE INDEX idx_courier_transport_kind_gin ON public.courier USING GIN (to_tsvector('english', transport_kind));

Цей приклад створює GIN-індекс для повного текстового вектора transport_kind у таблиці courier з використанням функції to_tsvector для індексації тексту.

Query Query History

```
1  EXPLAIN ANALYZE
2  SELECT * FROM courier
3  WHERE to_tsvector('english', transport_kind) @@ to_tsquery('english', 'your_search_term');
4
```

Data Output Messages Notifications

	QUERY PLAN text
1	Seq Scan on courier (cost=0.00..13.86 rows=1 width=36) (actual time=0.085..0.086 rows=…
2	Filter: (to_tsvector('english'::regconfig, (transport_kind)::text) @@ "'search" <-> "term'"::tsqu…
3	Rows Removed by Filter: 49
4	Planning Time: 0.412 ms
5	Execution Time: 0.097 ms

```
1  EXPLAIN ANALYZE
2  SELECT * FROM courier
3  WHERE to_tsvector('english', transport_kind) @@ to_tsquery('english', 'your_search_term');
4
```

Data Output Messages Notifications

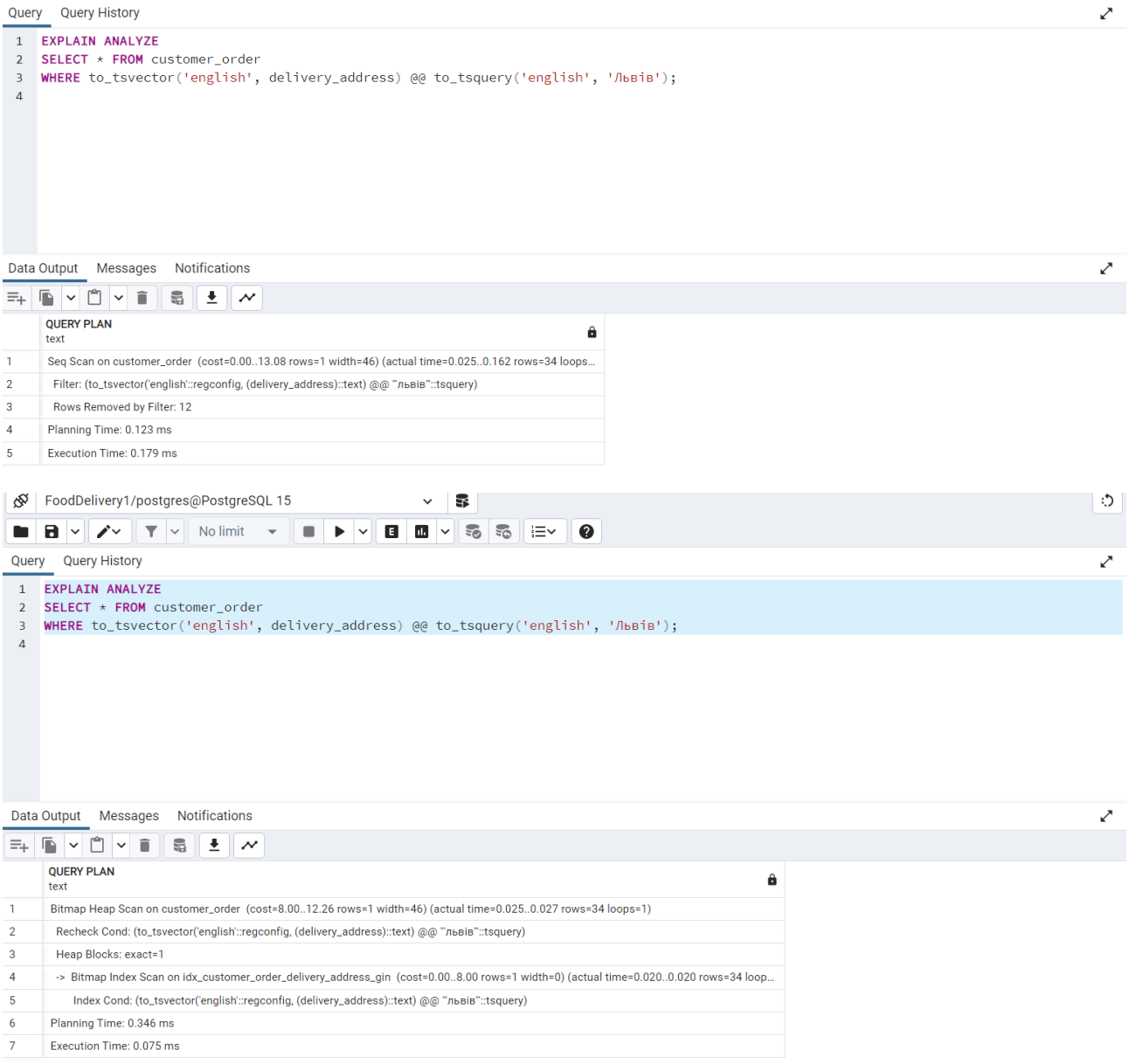
	QUERY PLAN text
1	Seq Scan on courier (cost=0.00..13.86 rows=1 width=36) (actual time=0.068..0.068 rows=…
2	Filter: (to_tsvector('english'::regconfig, (transport_kind)::text) @@ "'search" <-> "term'"::tsqu…
3	Rows Removed by Filter: 49
4	Planning Time: 0.420 ms
5	Execution Time: 0.080 ms

Можемо спостерігати, що наявність індекса не суттєво, але пришвидшує пошук з агрегуючими функціями.

4. **GIN-індекс для стовпця delivery_address у таблиці customer_order:**

CREATE INDEX idx_customer_order_delivery_address_gin ON public.customer_order USING GIN(to_tsvector('english', delivery_address));

У цьому прикладі ми використовуємо GIN-індекс для індексації текстових даних у стовпці delivery_address. Функція to_tsvector конвертує текст у вектор, який можна індексувати.



В даному випадку ми можемо спостерігати, що індекс суттєво пришвидшив виконання запиту.

Завдання 3

Ознайомився з інформацією про тригери і згідно до варіанту створив тригер, який має умови для тригера: before insert, delete.

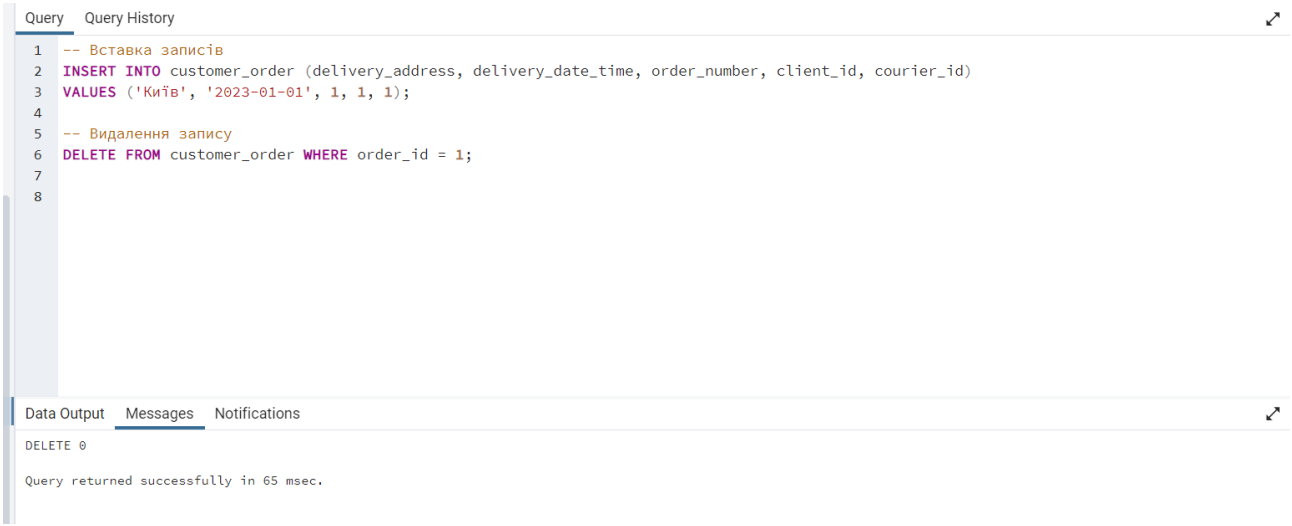
У цьому прикладі ми створюємо тригер order_changes_trigger, який викликає функцію log_order_changes перед вставкою чи видаленням

запису в таблиці customer_order. Функція log_order_changes логує зміни до таблиці order_changes_log.

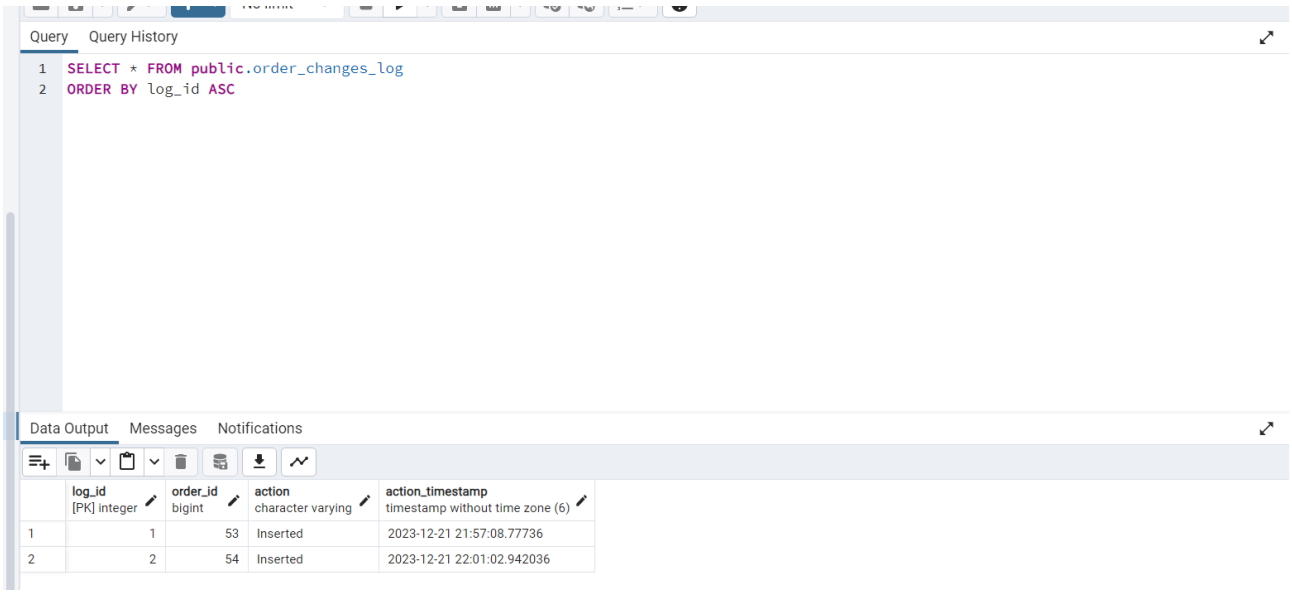
```
-- Створення таблиці order_changes_log
CREATE TABLE IF NOT EXISTS public.order_changes_log
(
    log_id SERIAL PRIMARY KEY,
    order_id bigint,
    action VARCHAR(50),
    action_timestamp timestamp(6) without time zone
);
-- Створення тригера
CREATE OR REPLACE FUNCTION log_order_changes()
RETURNS TRIGGER AS $$
DECLARE
    action_text VARCHAR(50);
BEGIN
    -- Визначення типу дії (вставка або видалення)
    IF TG_OP = 'INSERT' THEN
        action_text := 'Inserted';
    ELSIF TG_OP = 'DELETE' THEN
        action_text := 'Deleted';
    ELSE
        action_text := 'Unknown';
    END IF;
    -- Логування змін
    INSERT INTO order_changes_log (order_id, action, action_timestamp)
    VALUES (NEW.order_id, action_text, CURRENT_TIMESTAMP);
    RETURN NULL; -- Повертаємо NULL, оскільки це тригер "before"
END;
$$ LANGUAGE plpgsql;
```

```
-- Створення тригера "before insert, delete"
CREATE TRIGGER order_changes_trigger
BEFORE INSERT OR DELETE ON customer_order
FOR EACH ROW EXECUTE FUNCTION log_order_changes();
```

Виконання вставки записів, а також видалення



Зміни зафіксувались в журналі логувань:



Спроба вставити декілька записів:

Query Query History

```
-- Вставка декількох записів
INSERT INTO customer_order (delivery_address, delivery_date_time, order_number, client_id, courier_id)
VALUES
    ('Київ', '2023-01-01', 1, 1, 1),
    ('Львів', '2023-02-01', 2, 2, 2),
    ('Одеса', '2023-03-01', 3, 3, 3);

-- Видалення запису
DELETE FROM customer_order WHERE order_id IN (1, 2, 3);
```

Data Output Messages Notifications

```
DELETE 0

Query returned successfully in 114 msec.
```

Журнал логувань в свою чергу залогував всю інформацію:

Query Query History

```
SELECT * FROM public.order_changes_log
ORDER BY log_id ASC
```

Data Output Messages Notifications

	log_id [PK] integer	order_id bigint	action character varying	action_timestamp timestamp without time zone (6)
1	1	53	Inserted	2023-12-21 21:57:08.77736
2	2	54	Inserted	2023-12-21 22:01:02.942036
3	3	55	Inserted	2023-12-21 22:07:20.567309
4	4	56	Inserted	2023-12-21 22:07:20.567309
5	5	57	Inserted	2023-12-21 22:07:20.567309

Завдання 4

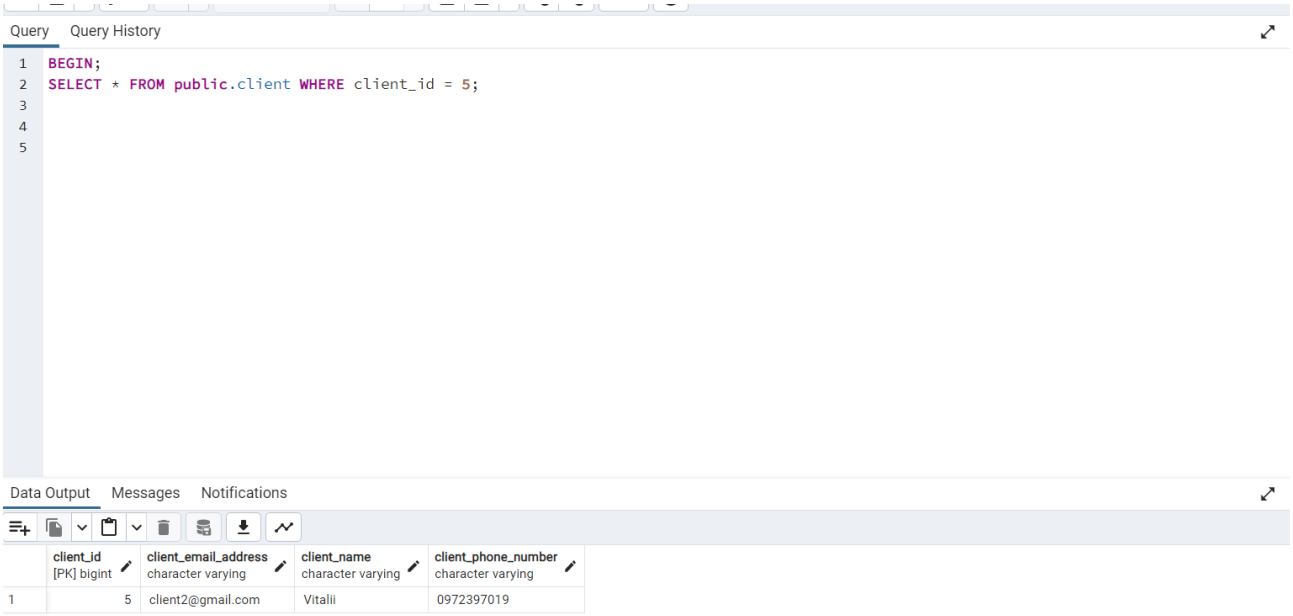
Рівень ізоляції READ COMMITTED

В середовищі PostgreSQL цей рівень стоїть по дефолту, тому не допускається брудне читання, і його неможливо продемонструвати. Ось приклад, як рівень ізоляції READ COMMITTED бореться з брудним читанням:

Сесія 1 (Читання Даних):

BEGIN;

SELECT * FROM public.client WHERE client_id = 1; -- Запит 1

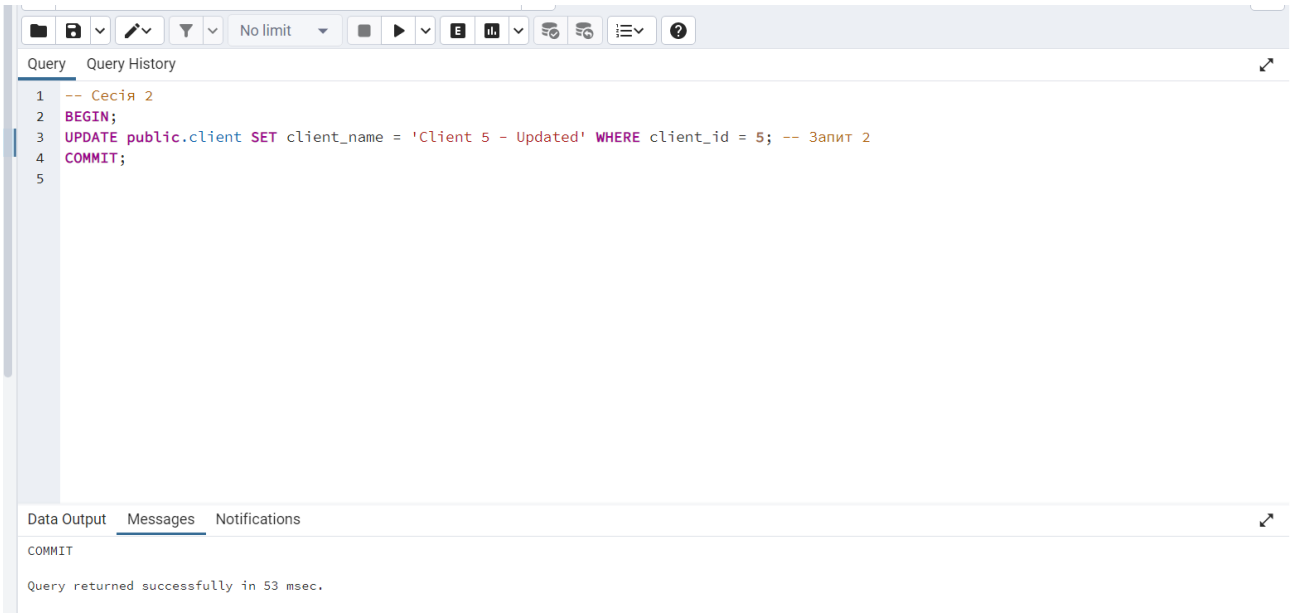


Сесія 2 (Модифікація Даних):

BEGIN;

UPDATE public.client SET client_name = 'Client 1 - Updated' WHERE client_id = 1; -- Запит 2

COMMIT;



Сесія 1 (Продовження Читання Даних):

SELECT * FROM public.client WHERE client_id = 1; -- Запит 3

COMMIT;

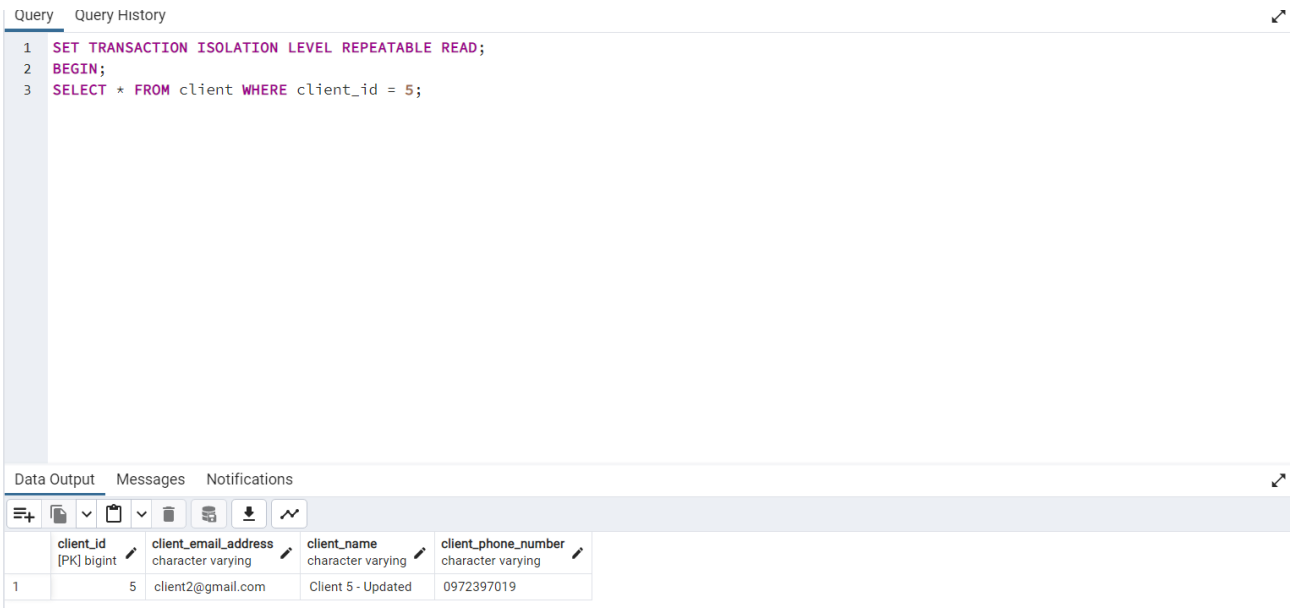
Запит 1 в сесії 1 відобразить старий запис, оскільки транзакція модифікації (Запит 2) в сесії 2 не закінчилася. Запит 3 в сесії 1, який викликається після COMMIT в сесії 2, відобразить оновлений запис.

Це показує, як рівень ізоляції "read committed" допомагає уникнути брудного читання, оскільки читання в сесії 1 побачить тільки закомічені зміни в сесії 2.

Рівень ізоляції REPEATABLE READ

При рівні ізоляції «Repeatable Read», транзакція бачить всі зміни в базі даних, які були фіксовані іншими транзакціями до її початку. Це означає, що під час виконання транзакції всі зміни, внесені іншими транзакціями до її початку, будуть видимі, і вони не будуть змінюватися під час виконання транзакції. Після фіксації транзакції, всі її зміни стають видимими для інших транзакцій.

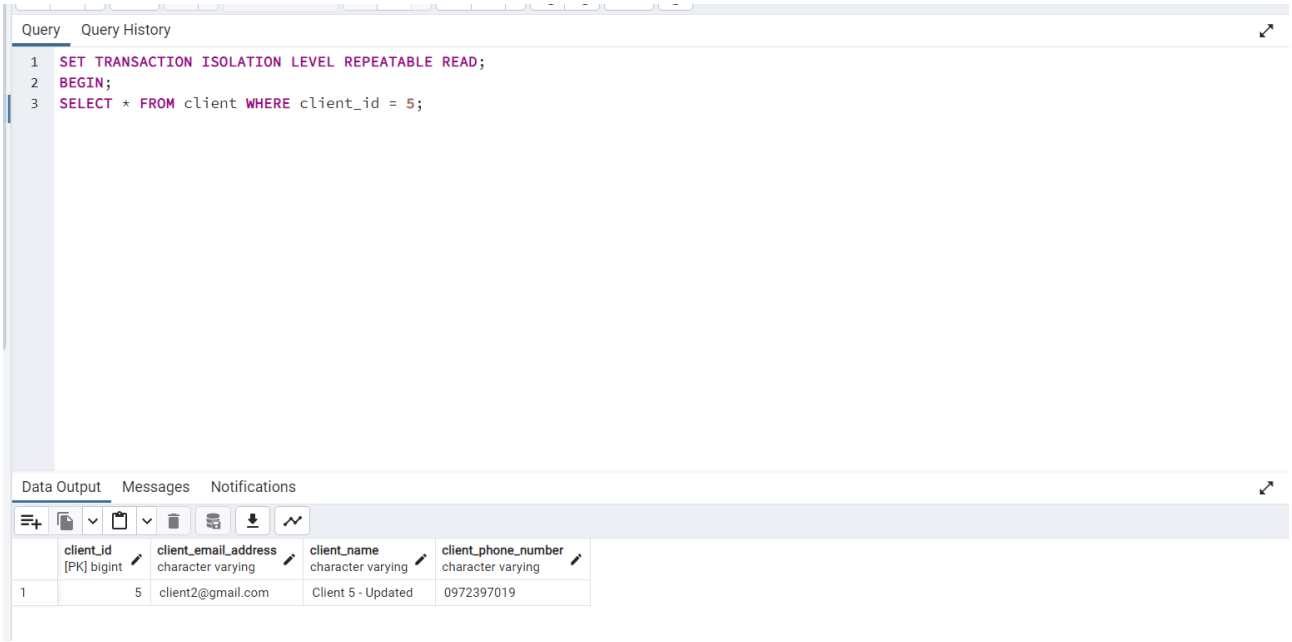
Застосуємо SELECT з client_id = 5, попередньо зафіксувавши рівень ізоляції repeatable read:



Спробуємо оновити ім’я клієнта з client_id = 5 на Oleh у іншій транзакції:

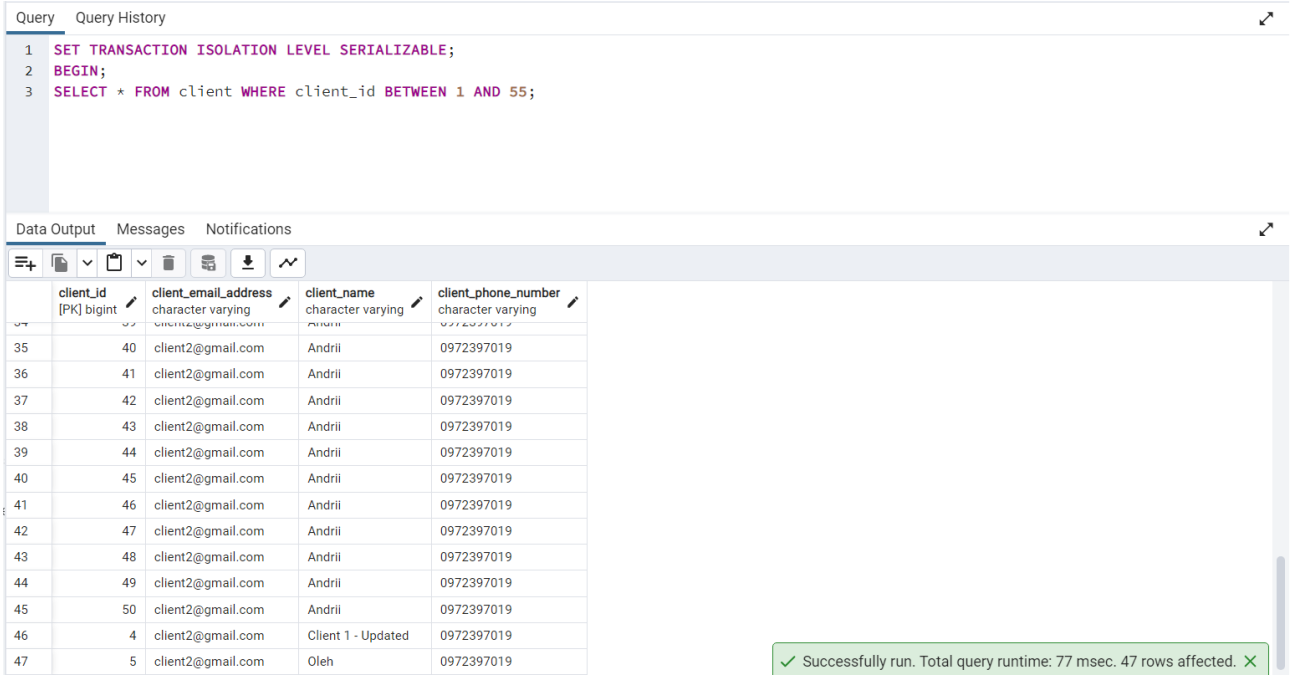
Повторна вибірка першою транзакцією:

Після фіксації другої транзакції, перша транзакція відображатиме дані з самого початку її виконання (заморожені дані). Отже, отримуємо старе ім’я клієнта оскільки дані для першої транзакції були заморожені до початку другої транзакції

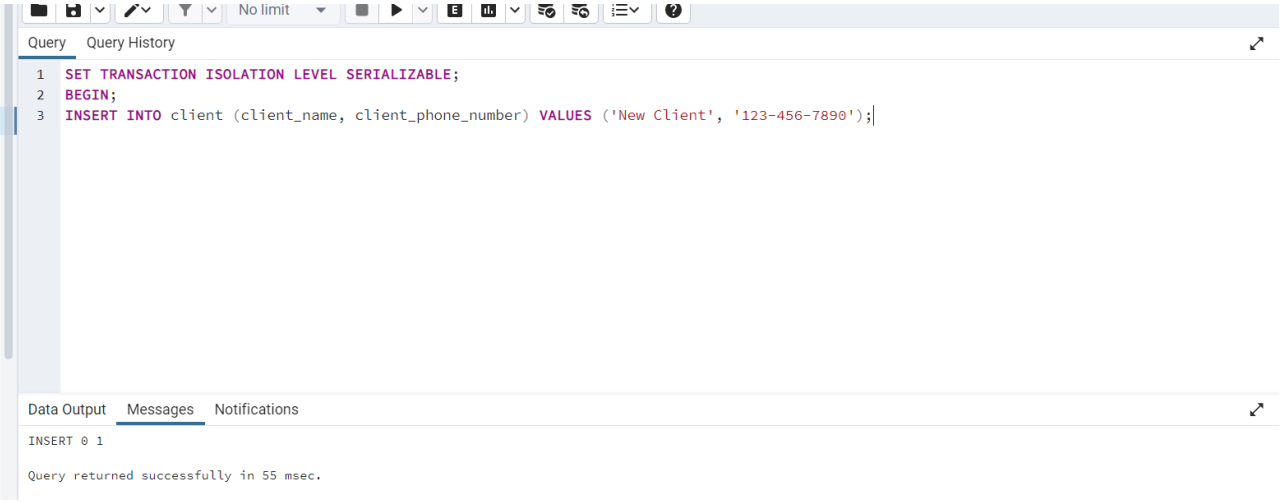


Рівень ізоляції SERIALIZABLE

Вибираємо всіх клієнтів з айді в діапазоні 1-55:



Вставляємо нові дані в цей проміжок в паралельній транзакції:



В 1 транзакції дані не змінились, отже рівень ізоляції SERIALIZABLE захистив нас від фантомного читання: