**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторних робіт № 3,4 з дисципліни

«Компоненті програмної інженерії 3.»

**«**Підвищення продуктивності системи, мікросервісна архітектура, GraphQL**»**

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-92 Бераудо Еліза*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2021

Зміст

[1 Мета лабораторної роботи 3](#_Toc89011751)

[2 Завдання 4](#_Toc89011752)

[3 Виконання 21](#_Toc89011753)

[3.1 Загальна архітектура застосунку 21](#_Toc89011754)

[3.2 Вихідний код застосунку 21](#_Toc89011755)

[3.3 Приклади роботи застосунку 21](#_Toc89011756)

[Критерії оцінювання 22](#_Toc89011757)

# Мета лабораторної роботи

Мета роботи – підвищення продуктивності системи, з наявними зовнішніми обмеженнями. Навчитись реалізовувати на практиці мікросервісну архітектуру. Розробка системи гнучкої до вимог клієнта.

# Завдання

**Завдання лабораторної роботи 3**

Виконати доопрацювання розробленої у лабораторній роботі №2 системи з метою підвищення продуктивності системи, з наявними зовнішніми обмеженнями. Застосувати «Caching», «Limit Response Time», «Introduce Concurrency».

Для цього виконати наступне:

* згенерувати 100 тис записів товарів у власній базі;
* перший постачальник (сторонній сервіс) має низьку продуктивність; земулювати час відповіді на пошук товарів, до 20-30 секунд;
* згенерувати в БД 2-ого постачальника (стороннього сервісу) 50 тис записів. (фільтр відсутній дані повертаються по 5 тис записів на сторінку);
* припускаємо, що сервіси змінюють ціну не частіше ніж 1 раз на день (на початку робочого дня);
* запит на пошук має тривати ~0.3 секунди, максимальний час 1 секунда.

У звіті

«Caching» застосовувалось як створення таблиці в базі даних яка заповнюється на початку робочого дня та використовується користувачем як основна. При запитах POST, PUT та DELETE обробці підлягають як записи в Cache таблиці так і записи в базі основного застосунку

Код реалізації:

import datetime  
import threading  
import multiprocessing as mp  
  
from flask\_restful import reqparse  
  
from PersistanceLayer.SingletonDataBase import Singleton  
*#from ProductBuilder import Director, Service1ProductBuilder, Service2ProductBuilder, OwnProductBuilder*from BusinessLayer.ProductBuilder import Director, OwnProductBuilder, Service1ProductBuilder, Service2ProductBuilder  
from psycopg2.extras import execute\_values  
  
class SingletonCache(type):  
 \_instances = {}  
 def \_\_call\_\_(cls, \*args, \*\*kwargs):  
 if cls not in cls.\_instances:  
 instance = super().\_\_call\_\_(\*args, \*\*kwargs)  
 cls.\_instances[cls] = instance  
 return cls.\_instances[cls]  
class CacheProduct(metaclass=SingletonCache):  
 def \_\_init\_\_(self):  
 self.own\_cache = []  
 self.service\_1\_cache = []  
 self.service\_2\_cache = []  
 def time\_to\_update(self):  
 dt = datetime.datetime.now()  
 tomorrow = dt + datetime.timedelta(days=1)  
 return (datetime.datetime.combine(tomorrow, datetime.time.min) - dt).seconds  
 def own\_prod(self, q):  
 director = Director()  
 builder = OwnProductBuilder()  
 director.builder = builder  
 director.build\_all\_product()  
 own = builder.product  
 *# print(len(own.products))* q.put(own.products)  
  
 def serv1\_prod(self, q):  
 director = Director()  
 builder = Service1ProductBuilder()  
 director.builder = builder  
 director.build\_all\_product()  
 serv1 = builder.product  
 *# print(len(serv1.products))* q.put(serv1.products)  
 def serv2\_prod(self, q):  
 director = Director()  
 builder = Service2ProductBuilder()  
 director.builder = builder  
 director.build\_all\_product()  
 serv2 = builder.product  
 *# print(len(serv2.products))* q.put(serv2.products)  
 def update(self):  
 conn = Singleton().conn  
 q1 = mp.Queue()  
 p1 = mp.Process(target=self.own\_prod, args=(q1,))  
  
 q2 = mp.Queue()  
 p2 = mp.Process(target=self.serv1\_prod, args=(q2,))  
  
 q3 = mp.Queue()  
 p3 = mp.Process(target=self.serv2\_prod, args=(q3,))  
 p1.start()  
 p2.start()  
 p3.start()  
 self.own\_cache = q1.get()  
 self.service\_1\_cache = q2.get()  
 self.service\_2\_cache = q3.get()  
 with conn.cursor() as cursor:  
 cursor.execute(**'TRUNCATE CacheProduct'**)  
 execute\_values(cursor,**'''INSERT INTO CacheProduct ("productName", "descriptionN", "price") VALUES %s'''**, [(args[**"productName"**], args[**"descriptionN"**], args[**"price"**]) for args in  
 self.own\_cache + self.service\_1\_cache + self.service\_2\_cache])  
 conn.commit()  
 p1.join()  
 p2.join()  
 p3.join()  
  
 director = Director()  
 builder = OwnProductBuilder()  
 director.builder = builder  
 director.build\_all\_product()  
 own = builder.product  
 self.own\_cache = own.products  
 *# print(len(self.own\_cache))* builder = Service1ProductBuilder()  
 director.builder = builder  
 director.build\_all\_product()  
 service1 = builder.product  
  
 self.service\_1\_cache = service1.products  
 *# print(len(self.service\_1\_cache))* builder = Service2ProductBuilder()  
 director.builder = builder  
 director.build\_all\_product()  
 service2 = builder.product  
 self.service\_2\_cache = service2.products  
 *# print(len(self.service\_2\_cache))  
 #print(self.time\_to\_update())* timer = threading.Timer(self.time\_to\_update(), self.update)  
 timer.start()  
 def get\_cache(self):  
 parser = reqparse.RequestParser()  
 *# parser.add\_argument("sale\_type")* parser.add\_argument(**"product\_name"**)  
 parser.add\_argument(**"min\_price"**)  
 parser.add\_argument(**"max\_price"**)  
 args = parser.parse\_args()  
 parse\_str = **'''SELECT \* FROM CacheProduct '''** filt\_opt = []  
 *# if args['sale\_type']:  
 # filt\_opt.append(['"sale\_type"=', args['sale\_type']])* if args[**'product\_name'**]:  
 filt\_opt.append([**'"product\_name"='**, args[**'product\_name'**]])  
 if args[**'min\_price'**]:  
 filt\_opt.append([**'"price">'**, args[**'min\_price'**]])  
 if args[**'max\_price'**]:  
 filt\_opt.append([**'"price"<'**, args[**'max\_price'**]])  
 if len(filt\_opt)>0:  
 parse\_str+=**'WHERE '** for i in range(len(filt\_opt)):  
 parse\_str += filt\_opt[i][0]+**"'"**+ filt\_opt[i][1] + **"'"** if i+1 < len(filt\_opt):  
 parse\_str+=**' AND '** conn = Singleton().conn  
 with conn.cursor() as cursor:  
 cursor.execute(parse\_str)  
 rows = cursor.fetchall()  
 result = []  
 for row in rows:  
 a = {**"productId"**: row[0], **"productName"**: (row[1]), **"descriptionN"**: (row[2]), **"price"**: row[3]}  
  
 result.append(a)  
 return result

«Limit Response Time» було використано до першого сервісу

Код реалізації:

time.sleep(random.randint(20, 30))

**Завдання лабораторної роботи 4**

Перетворити застосунок розроблений у лабораторних роботах 2,3 у застосунок з мікросервісною архітектурою.

Для цього виконати наступне:

* додати сервіс реєстрації;
* додати сервіс аутентифікації;
* додати сервіс пошуку (фільтрації);
* додати сервіс бронювання (покупки, замовлень, розподілу, тощо).
* реалізувати взаємозв’язок між сторонніми постачальниками (сервісами) та основним застосунком, має виконуватись:
  1. метод виконання бронювання (покупок, замовлень, тощо);
  2. список бронювань (покупок, замовлень, тощо).
* для зв’язку застосувати Grpc;
* забезпечити коректну роботу сервісів бронювання та пошуку (фільтрації), якщо відмовив один з сторонніх постачальників (сервісів), або обидва;
  1. реалізувати Fallback;
  2. додати retry.
* зобразити схему архітектури;

**Альтернативне завдання.**

* Реалізувати GraphQL сервер для зчитування 3-х сутностей: бронювання/товар/користувач (в залежності від предметної області сутності змінюються, наприклад покупка/товар/клієнт для інтернет магазину) (товари мають братись як з власної БД так і з даних сторонніх постачальників (сервісів))
* Реалізувати GraphQL Mutations для створення бронювання, додавання, редагування та видалення товару.

У звіті до лабораторних робіт відобразити втілення архітектурного паттерну для вашої предметної області та приклади роботи застосунку.

Окремо вставити структурований вихідний код.

**Варіанти предметних областей.**

**2. Товари в кредит.** Меблевий магазин надає послуги з придбання товарів у кредит. Для оформлення кредиту покупець повинен надати банківську виписку. Рішення про видачу кредиту здійснюється шляхом зіставлення ціни товару і доходів покупця: визначається термін погашення кредиту, відсоткова ставка кредиту. На підставі цих даних формується графік погашення кредиту, що визначає дату і суму чергового внеску. Сформований графік передається адміністрації магазину.

Якщо клієнт не виплачує внесок у встановлений час, то нараховується пеня за кожний день прострочки. Поки пеня не погашена, виплата внесків за наступні місяці не приймається.

Оформлення договору на кредит відбувається за участю певного банку, за що банк отримує фіксований відсоток комісії. Вибір банку виконує адміністрація меблевого магазину.

# Виконання

## Загальна архітектура застосунку

Втілення архітектурного паттерну для предметної області, можна побачити на рисунку 3.1.

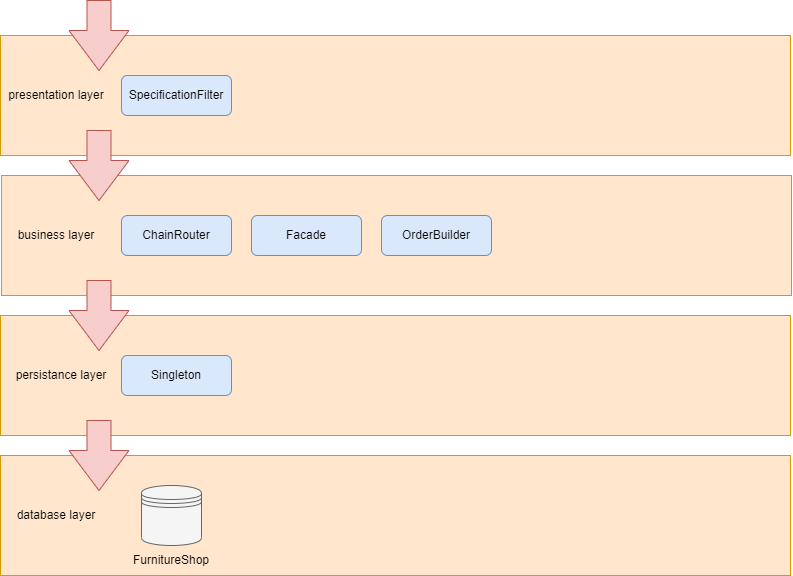


Рисунок 3.1 – Загальна архітектура застосунку

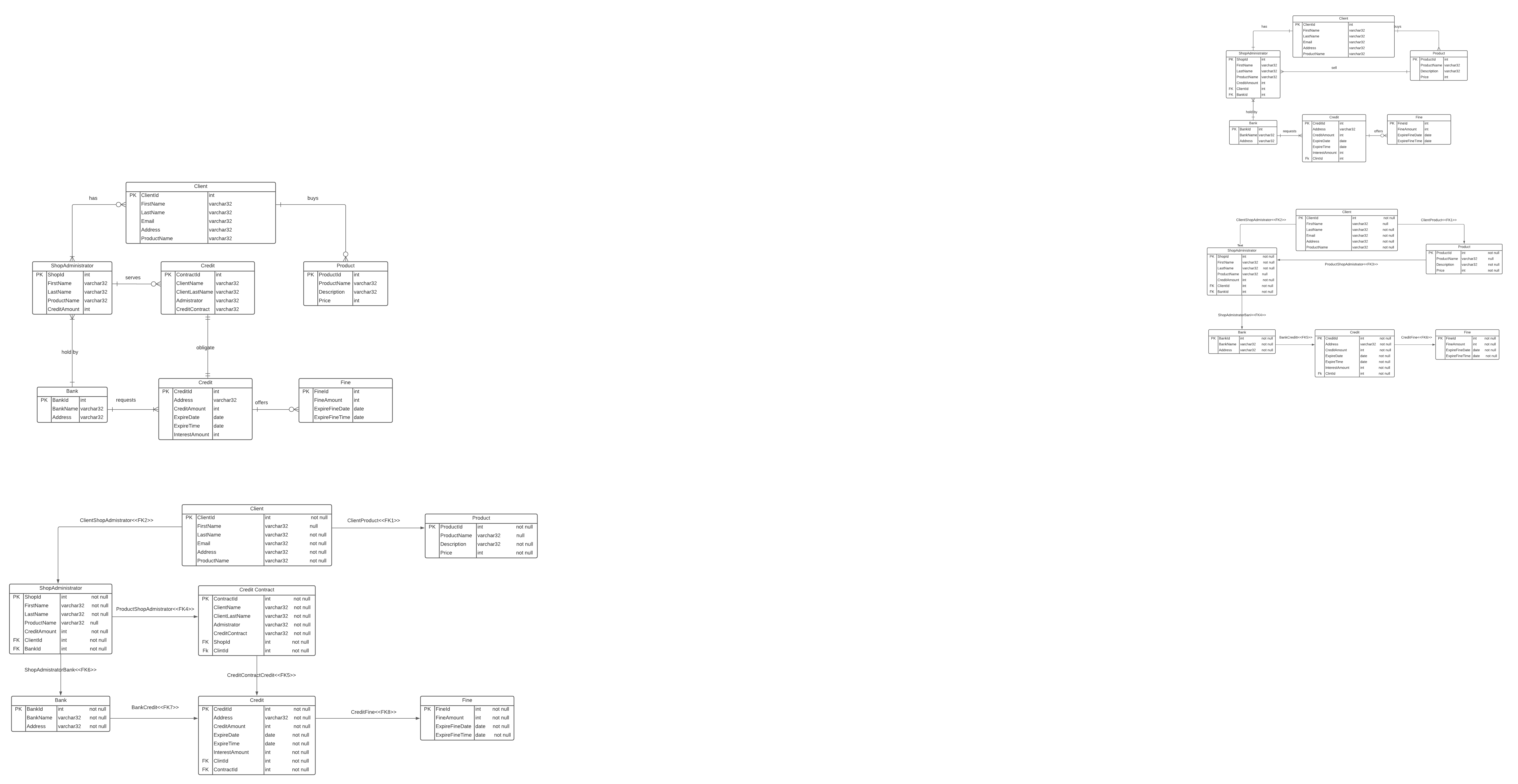
На рисунку 3.2 наведено діаграму класів застосунку з урахуванням використаних паттернів проектування.



Рисунок 3.2 – Діаграма класів застосунку

## Фізична модель БД

На рисунку 3.2 наведено фізичну модель БД, яка використовується у застосунку.



## Вихідний код застосунку

Init.py

from flask import Flask  
from flask\_cors import CORS  
from flask\_sqlalchemy import SQLAlchemy  
import psycopg2  
  
app = Flask(\_\_name\_\_)  
CORS(app)  
  
app.config[**"SQLALCHEMY\_DATABASE\_URI"**] = **"postgresql://fzmnncct:iJ9JicP4J4xgMABPXP43CxS\_cFpXyQ9N@abul.db.elephantsql.com/fzmnncct"**app.config[**"SQLALCHEMY\_TRACK\_MODIFICATIONS"**] = False  
db = SQLAlchemy(app)  
  
@app.route(**'/'**)  
def hello():  
 return **'My First API !!'**

app.py

from api import app, db  
from ariadne import load\_schema\_from\_path, make\_executable\_schema, \  
 graphql\_sync, snake\_case\_fallback\_resolvers, ObjectType  
from ariadne.constants import PLAYGROUND\_HTML  
from flask import request, jsonify  
from api.queries import \*  
from api.mutations import \*  
  
query = ObjectType(**"Query"**)  
mutation = ObjectType(**"Mutation"**)  
query.set\_field(**"listProducts"**, listProducts\_resolver)  
query.set\_field(**"getProduct"**, getProduct\_resolver)  
query.set\_field(**"listCustomers"**, listCustomers\_resolver)  
query.set\_field(**"getCustomer"**, getCustomer\_resolver)  
query.set\_field(**"listOrders"**, listOrders\_resolver)  
query.set\_field(**"getOrder"**, getOrder\_resolver)  
  
mutation.set\_field(**"createProduct"**, create\_product\_resolver)  
mutation.set\_field(**"updateProduct"**, update\_product\_resolver)  
mutation.set\_field(**"deleteProduct"**, delete\_product\_resolver)  
mutation.set\_field(**"createCustomer"**, create\_customer\_resolver)  
mutation.set\_field(**"updateCustomer"**, update\_customer\_resolver)  
mutation.set\_field(**"deleteCustomer"**, delete\_customer\_resolver)  
mutation.set\_field(**"createOrder"**, create\_order\_resolver)  
mutation.set\_field(**"updateOrder"**, update\_order\_resolver)  
mutation.set\_field(**"deleteOrder"**, delete\_order\_resolver)  
  
type\_defs = load\_schema\_from\_path(**"schema.graphql"**)  
schema = make\_executable\_schema(  
 type\_defs, query, mutation, snake\_case\_fallback\_resolvers  
)  
  
@app.route(**"/graphql"**, methods=[**"GET"**])  
def graphql\_playground():  
 return PLAYGROUND\_HTML, 200  
  
@app.route(**"/graphql"**, methods=[**"POST"**])  
def graphql\_server():  
 data = request.get\_json()  
 success, result = graphql\_sync(  
 schema,  
 data,  
 context\_value=request,  
 debug=app.debug  
 )  
 status\_code = 200 if success else 400  
 return jsonify(result), status\_code

schema.graphql

schema{  
 query: Query  
 mutation: Mutation  
}  
type Mutation{  
 createProduct(name: String!, description: String!, price: Int!):ProductResult!  
 updateProduct(productid:ID!, name: String, description: String, price: Int): ProductResult!  
 deleteProduct(productid:ID):ProductResult!  
 createCustomer(name: String!, surname: String, email: String, address: String):CustomerResult!  
 updateCustomer(customerid:ID!, name: String, surname: String, email: String, address: String):CustomerResult!  
 deleteCustomer(customerid:ID!):CustomerResult!  
 createOrder(customerid: Int!, productid: Int!): OrderResult!  
 updateOrder(orderid:ID!, customerid: Int!, productid: Int!): OrderResult!  
 deleteOrder(orderid:ID!): OrderResult!  
}  
  
type product{  
 productid: ID!  
 name: String!  
 description: String!  
 price: Int!  
}  
  
type ProductResult {  
 success: Boolean!  
 errors: [String]  
 product: product  
}  
  
type ProductsResult{  
 success: Boolean!  
 errors: [String]  
 products: [product]  
}  
  
type customer{  
 customerid: ID!  
 name: String!  
 surname: String!  
 email: String!  
 address: String!  
}  
type CustomerResult{  
 success: Boolean!  
 errors: [String]  
 customer: customer  
}  
type CustomersResult{  
 success: Boolean!  
 errors: [String]  
 customers: [customer]  
}  
  
type order{  
 orderid: ID!  
 customerid: Int!  
 productid: Int!  
}  
type OrderResult{  
 success: Boolean!  
 errors: [String]  
 order: order  
}  
type OrdersResult{  
 success: Boolean!  
 errors: [String]  
 orders: [order]  
}  
  
type Query{  
 listProducts:ProductsResult!  
 getProduct(id: ID!): ProductResult!  
 listCustomers:CustomersResult!  
 getCustomer(id: ID!): CustomerResult!  
 listOrders:OrdersResult!  
 getOrder(id: ID!): OrderResult!  
}

models.py

from app import db  
  
class Product(db.Model):  
 productid = db.Column(db.Integer, primary\_key=True)  
 name = db.Column(db.String)  
 description = db.Column(db.String)  
 price = db.Column(db.Integer)  
  
 def to\_dict(self):  
 return {  
 **"productid"**: self.productid,  
 **"name"**: self.name,  
 **"description"**: self.description,  
 **"price"**: self.price  
 }  
  
class Customer(db.Model):  
 customerid = db.Column(db.Integer, primary\_key=True)  
 name = db.Column(db.String)  
 surname = db.Column(db.String)  
 email = db.Column(db.String)  
 address = db.Column(db.String)  
  
 def to\_dict(self):  
 return {  
 **"customerid"**: self.customerid,  
 **"name"**: self.name,  
 **"surname"**: self.surname,  
 **"email"**: self.email,  
 **"address"**: self.address  
 }  
  
  
class Order(db.Model):  
 orderid = db.Column(db.Integer, primary\_key=True)  
 customerid = db.Column(db.Integer)  
 productid = db.Column(db.Integer)  
  
 def to\_dict(self):  
 return {  
 **"orderid"**: self.orderid,  
 **"customerid"**: self.customerid,  
 **"productid"**: self.productid  
 }

queries.py

from .models import Product, Customer, Order  
from ariadne import convert\_kwargs\_to\_snake\_case  
  
def listProducts\_resolver(obj, info):  
 try:  
 products = [product.to\_dict() for product in Product.query.limit(5000)]  
  
 print(products)  
 payload = {  
 **"success"**: True,  
 **"products"**: products  
 }  
 except Exception as error:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [str(error)]  
 }  
 return payload  
  
  
@convert\_kwargs\_to\_snake\_case  
def getProduct\_resolver(obj, info, id):  
 try:  
 product = Product.query.get(id)  
 print(product.to\_dict())  
 payload = {  
 **"success"**: True,  
 **"product"**: product.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Product item matching {id} not found"**]  
 }  
 return payload  
  
def listCustomers\_resolver(obj, info):  
 try:  
 customers = [customer.to\_dict() for customer in Customer.query.all()]  
  
 print(customers)  
 payload = {  
 **"success"**: True,  
 **"customers"**: customers  
 }  
 except Exception as error:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [str(error)]  
 }  
 return payload  
  
@convert\_kwargs\_to\_snake\_case  
def getCustomer\_resolver(obj, info, id):  
 try:  
 customer = Customer.query.get(id)  
 print(customer.to\_dict())  
 payload = {  
 **"success"**: True,  
 **"customer"**: customer.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Customer item matching {id} not found"**]  
 }  
 return payload  
  
  
def listOrders\_resolver(obj, info):  
 try:  
 orders = [order.to\_dict() for order in Order.query.all()]  
 payload = {  
 **"success"**: True,  
 **"orders"**: orders  
 }  
 except Exception as error:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [str(error)]  
 }  
 return payload  
  
  
@convert\_kwargs\_to\_snake\_case  
def getOrder\_resolver(obj, info, id):  
 try:  
 order = Order.query.get(id)  
  
 payload = {  
 **"success"**: True,  
 **"order"**: order.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Order item matching {id} not found"**]  
 }  
 return payload

mutations.py

from ariadne import convert\_kwargs\_to\_snake\_case  
from api import db  
from api.models import Product, Customer, Order  
  
def create\_product\_resolver(obj, info, name, description, price ):  
 try:  
 product = Product(name=name, description=description, price=price)  
 db.session.add(product)  
 db.session.commit()  
 payload = {  
 **"success"**: True,  
 **"product"**: product.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Product item attribute is not correct"**]  
 }  
 return payload  
  
@convert\_kwargs\_to\_snake\_case  
def update\_product\_resolver(obj, info, productid, name, description, price):  
 try:  
 product = Product.query.get(productid)  
 if product:  
 product.name = name  
 product.description = description  
 product.price = price  
 db.session.add(product)  
 db.session.commit()  
 payload = {  
 **"success"**: True,  
 **"product"**: product.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Not Found"**]  
 }  
 return payload  
  
@convert\_kwargs\_to\_snake\_case  
def delete\_product\_resolver(obj, info, productid):  
 try:  
 product = Product.query.get(productid)  
 db.session.delete(product)  
 db.session.commit()  
 payload = {  
 **"success"**: True,  
 **"product"**: product.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Not Found"**]  
 }  
 return payload  
  
def create\_customer\_resolver(obj, info, name, surname, email, address):  
 try:  
 customer = Customer(name=name,surname=surname, email=email, address=address)  
 db.session.add(customer)  
 db.session.commit()  
 payload = {  
 **"success"**: True,  
 **"customer"**: customer.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Customer item attribute is not correct"**]  
 }  
 return payload  
  
@convert\_kwargs\_to\_snake\_case  
def update\_customer\_resolver(obj, info, customerid, name, surname, email, address):  
 try:  
 customer = Customer.query.get(customerid)  
 if customer:  
 customer.name = name  
 customer.surname = surname  
 customer.email = email  
 customer.address = address  
 db.session.add(customer)  
 db.session.commit()  
 payload = {  
 **"success"**: True,  
 **"customer"**: customer.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Not Found"**]  
 }  
 return payload  
  
@convert\_kwargs\_to\_snake\_case  
def delete\_customer\_resolver(obj, info, customerid):  
 try:  
 customer = Customer.query.get(customerid)  
 db.session.delete(customer)  
 db.session.commit()  
 payload = {  
 **"success"**: True,  
 **"customer"**: customer.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Not Found"**]  
 }  
 return payload  
  
def create\_order\_resolver(obj, info, customerid, productid):  
 try:  
 order = Order(customerid=customerid, productid=productid)  
 db.session.add(order)  
 db.session.commit()  
 payload = {  
 **"success"**: True,  
 **"order"**: order.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Order item attribute is not correct"**]  
 }  
 return payload  
  
@convert\_kwargs\_to\_snake\_case  
def update\_order\_resolver(obj, info, orderid, customerid, productid):  
 try:  
 order = Order.query.get(orderid)  
 if order:  
 order.customerid = customerid  
 order.productid = productid  
 db.session.add(order)  
 db.session.commit()  
 payload = {  
 **"success"**: True,  
 **"order"**: order.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Not Found"**]  
 }  
 return payload  
  
@convert\_kwargs\_to\_snake\_case  
def delete\_order\_resolver(obj, info, orderid):  
 try:  
 order = Order.query.get(orderid)  
 db.session.delete(order)  
 db.session.commit()  
 payload = {  
 **"success"**: True,  
 **"order"**: order.to\_dict()  
 }  
 except AttributeError:  
 payload = {  
 **"success"**: False,  
 **"errors"**: [**"Not Found"**]  
 }  
 return payload

## Приклади роботи застосунку

На рисунках 3.3 - 3.6 показані приклади роботи застосунку.

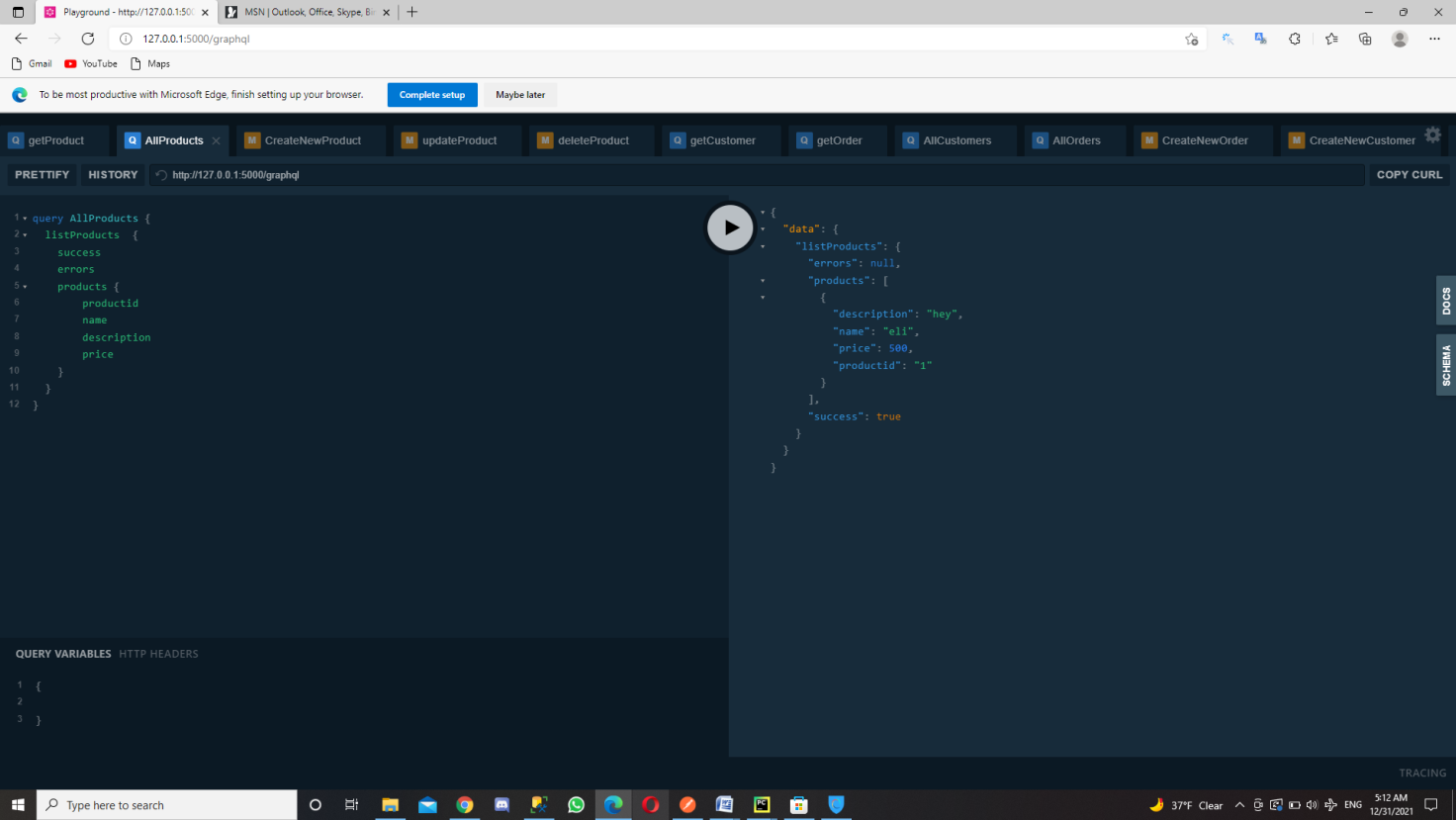


Рисунок 3.3 – Приклад виконання запиту allProducts

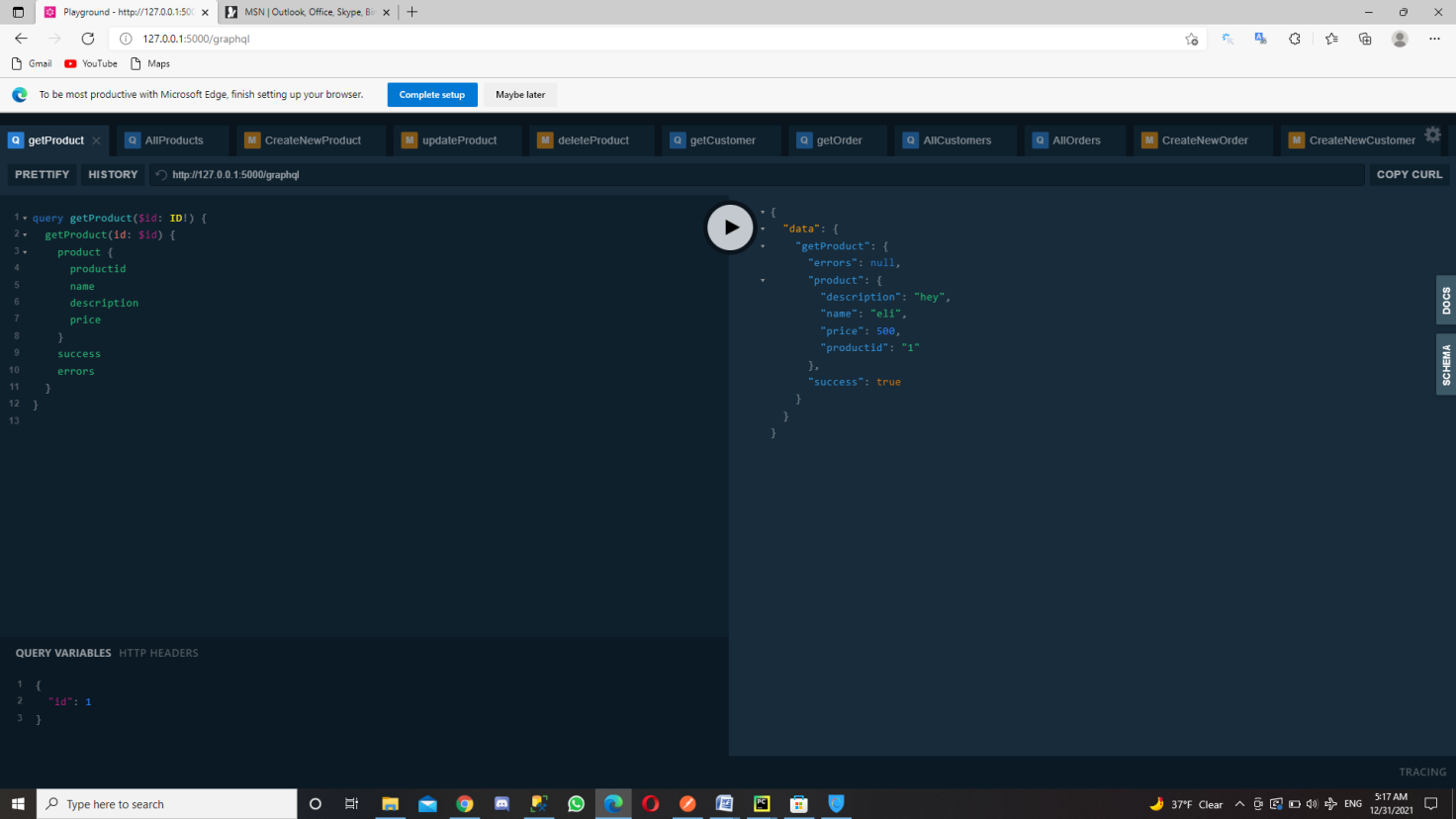


Рисунок 3.4 – Приклад виконання запиту getProducts

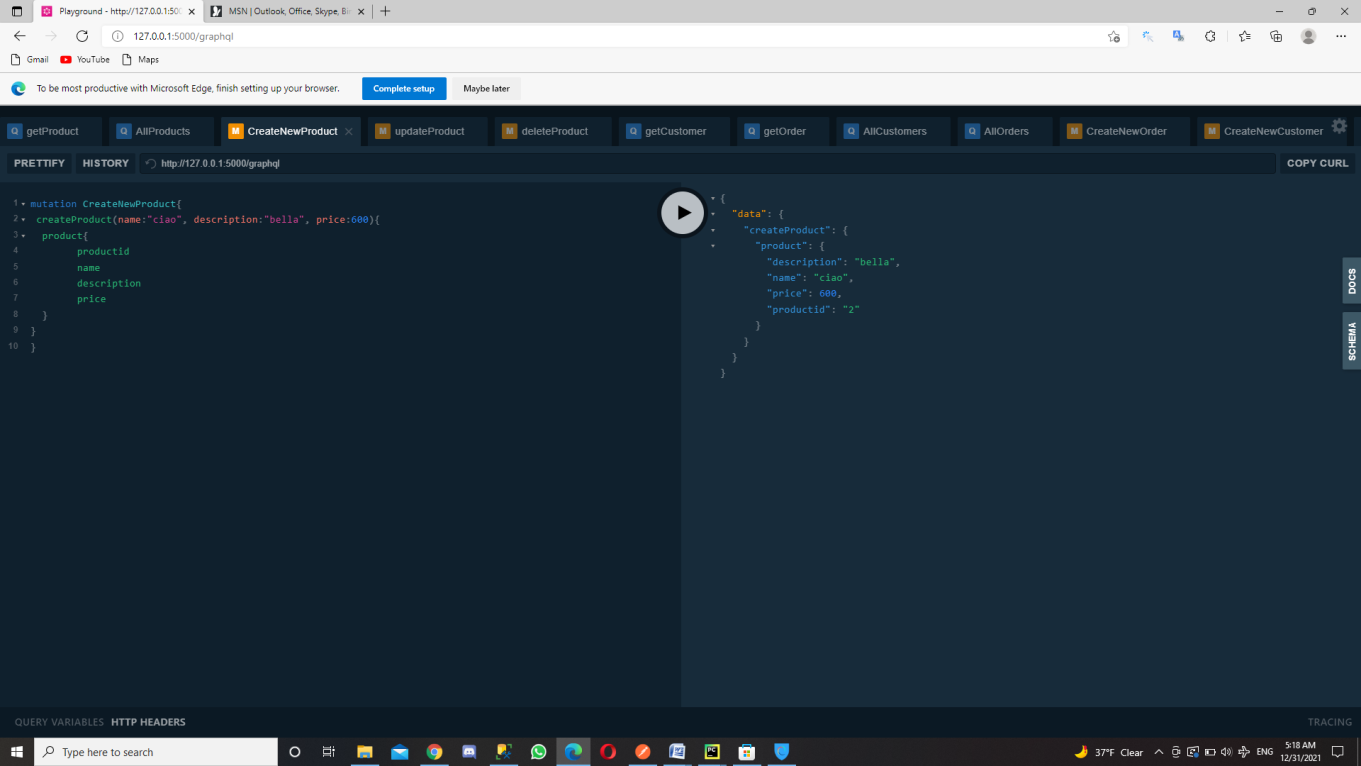


Рисунок 3.5 – Приклад виконання запиту createNewProduct

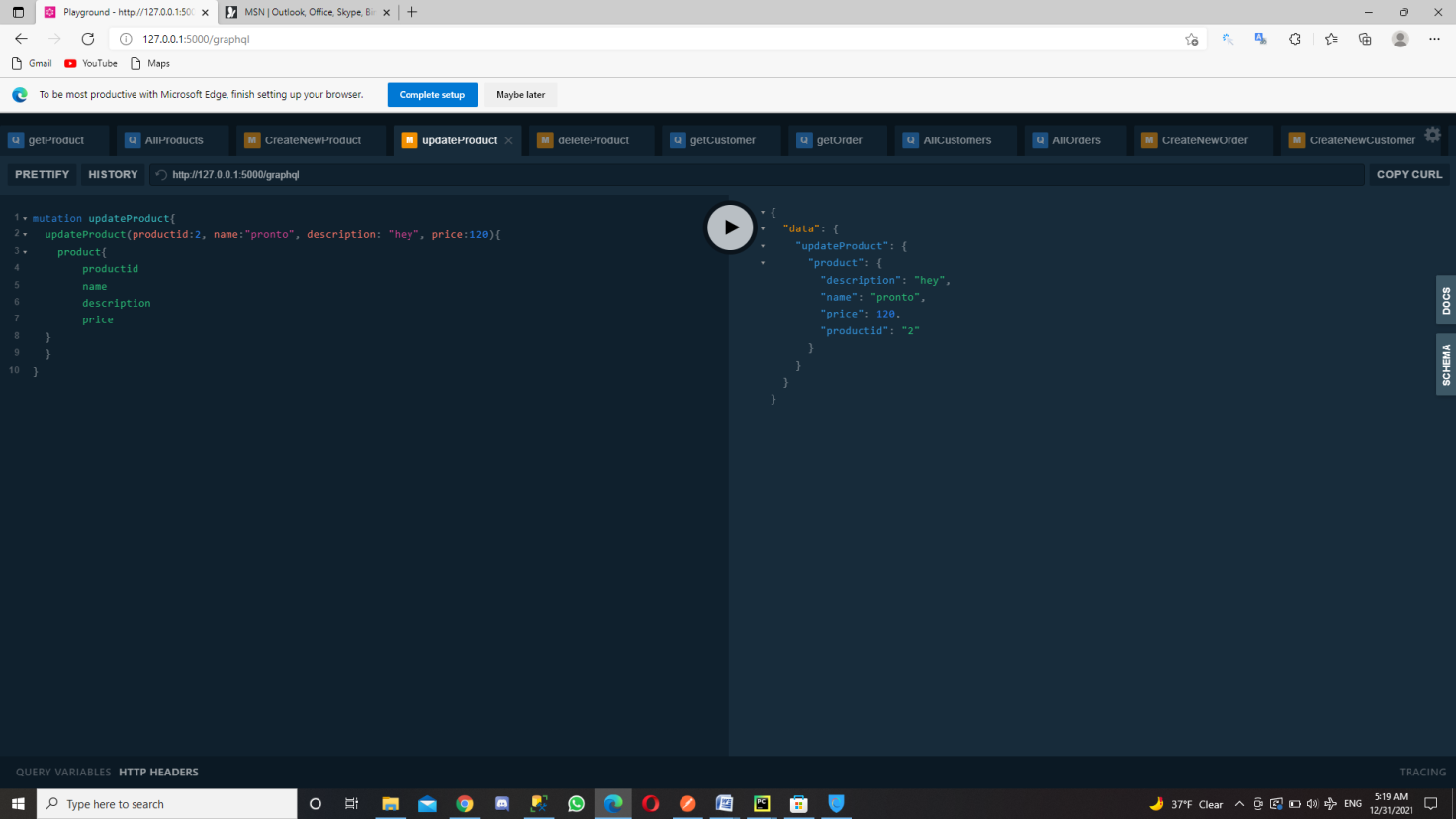


Рисунок 3.6 – Приклад виконання запиту updateProduct

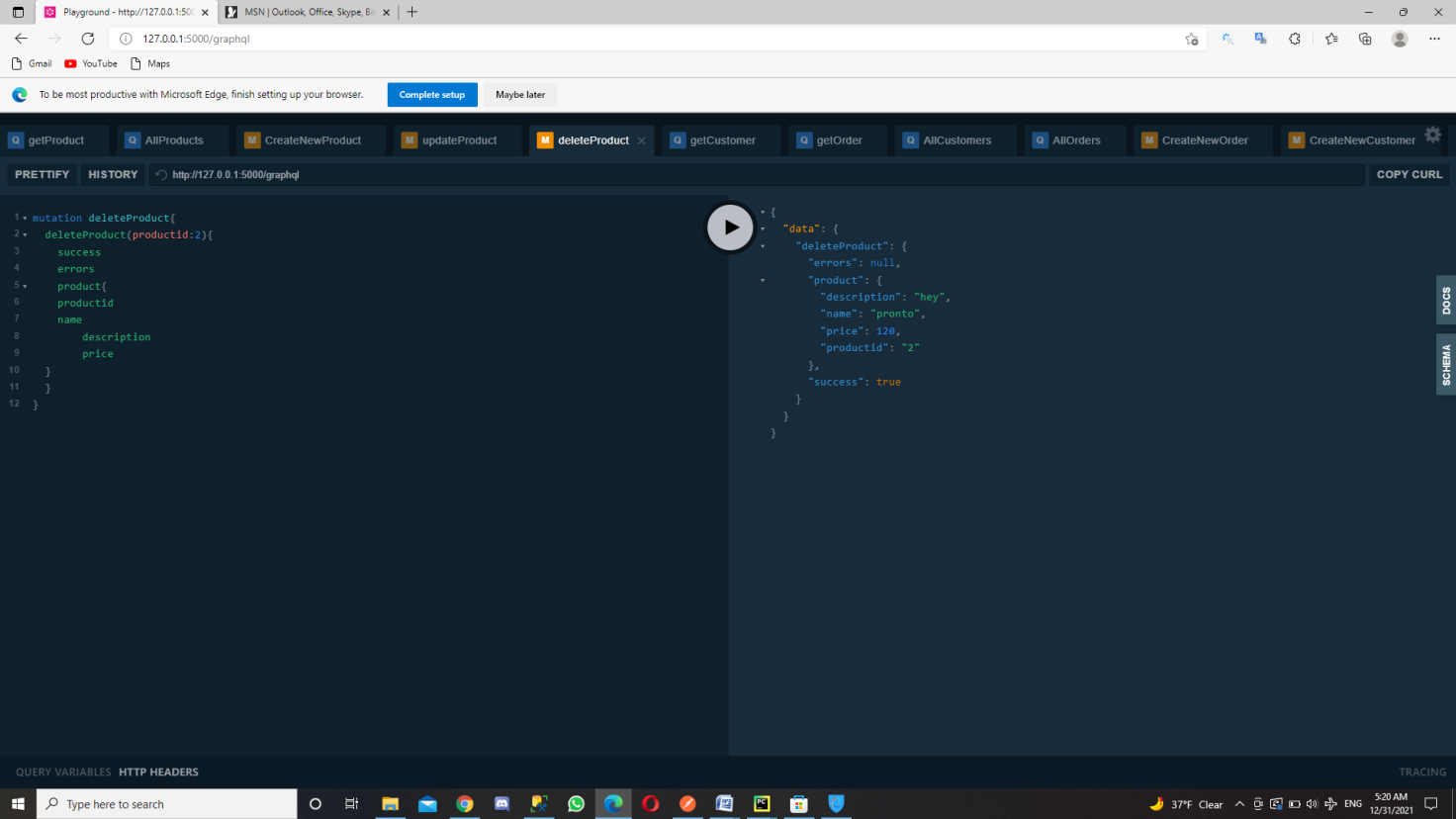


Рисунок 3.7 – Приклад виконання запиту deleteProduct

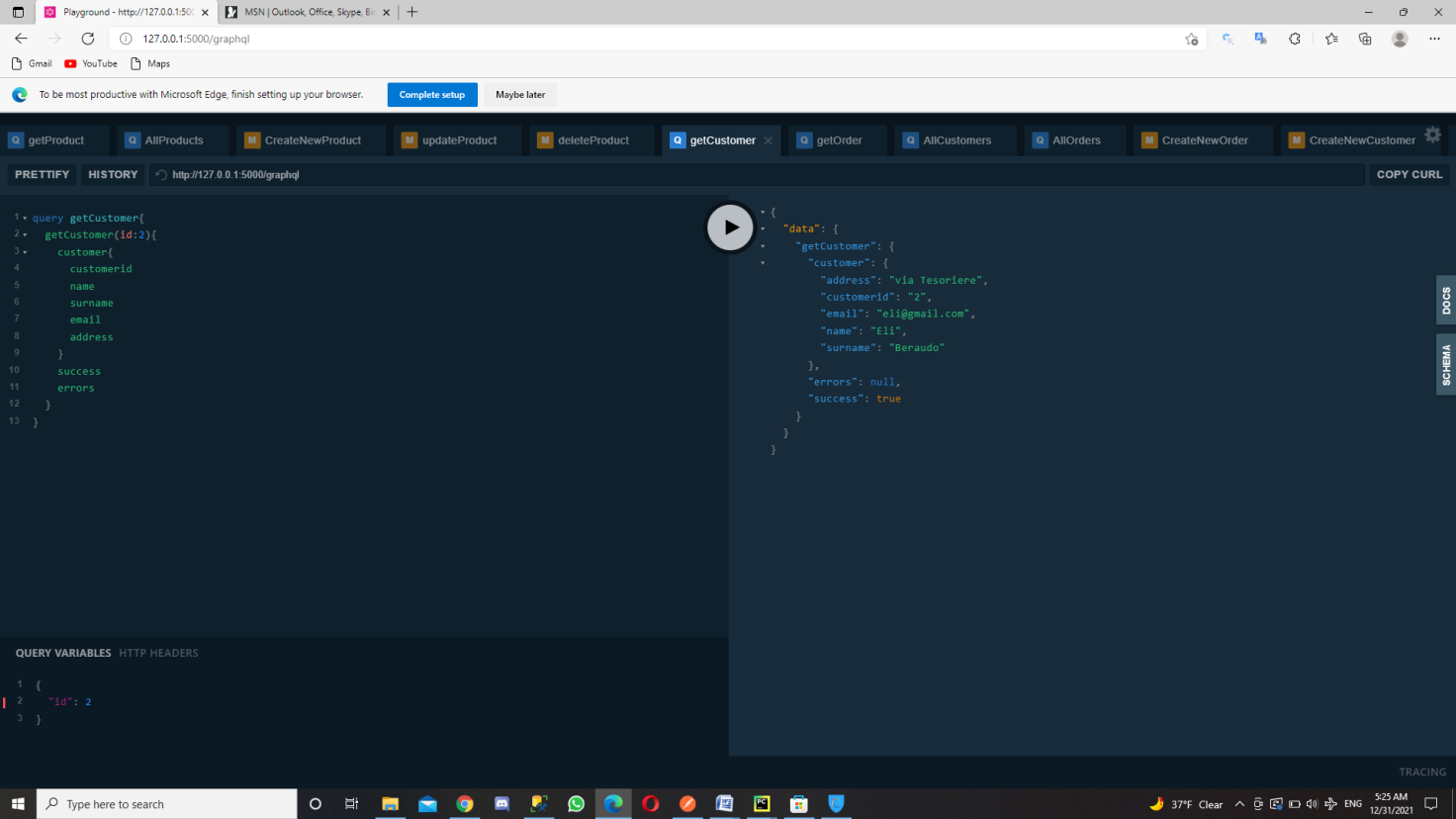


Рисунок 3.8 – Приклад виконання запиту getCustomer

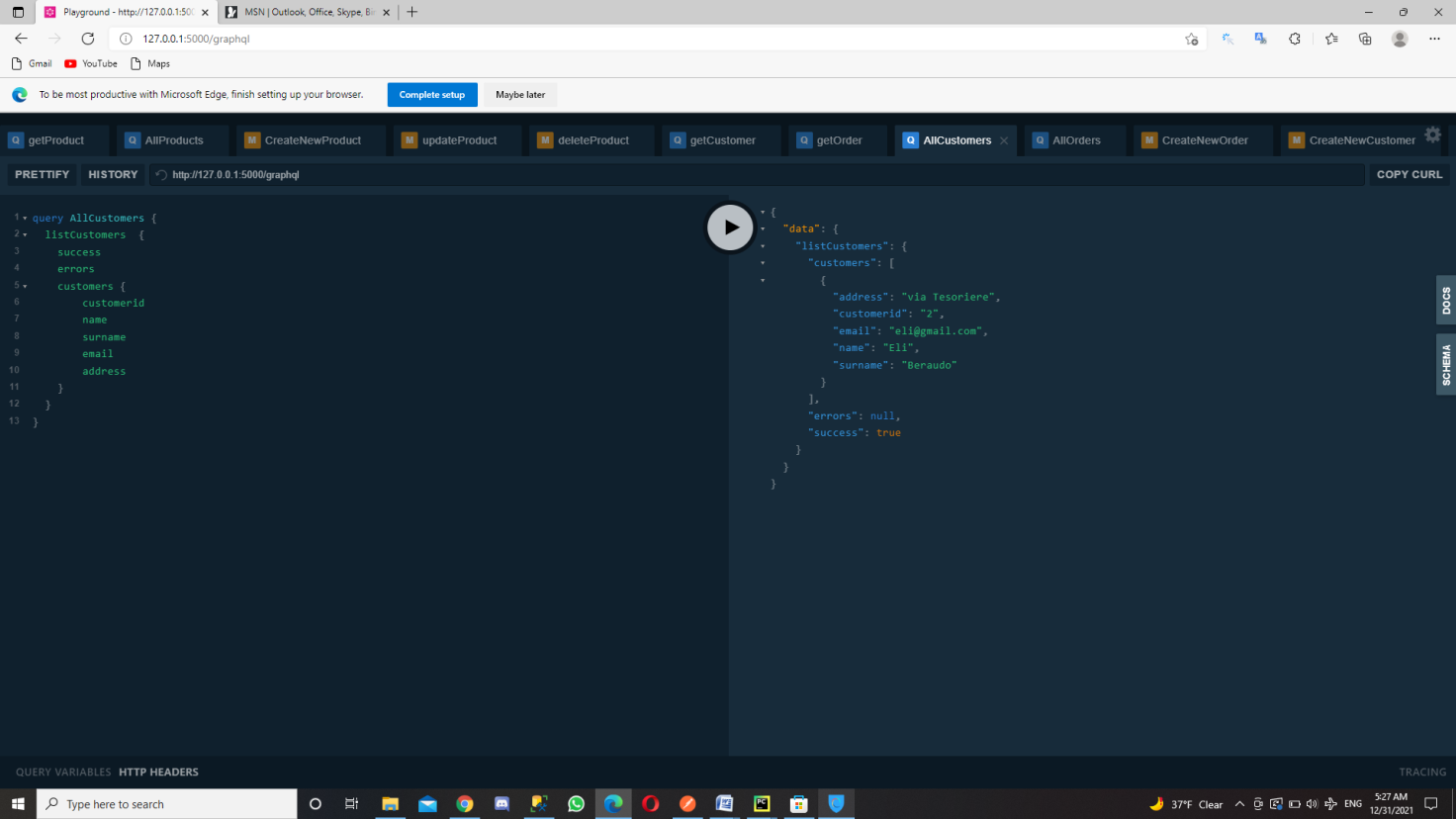


Рисунок 3.9 – Приклад виконання запиту AllCustomer

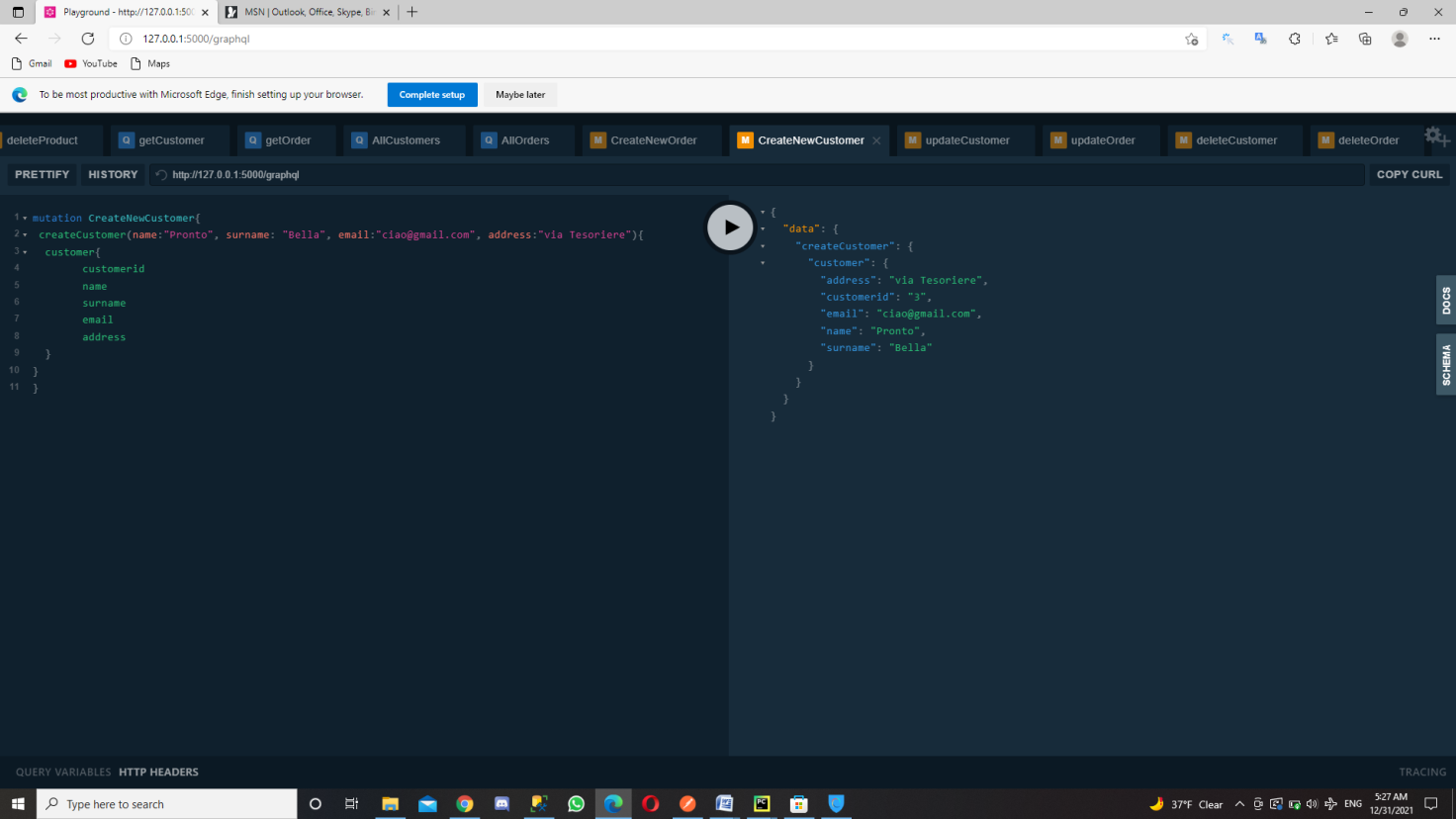


Рисунок 3.10 – Приклад виконання запиту createNewCustomer

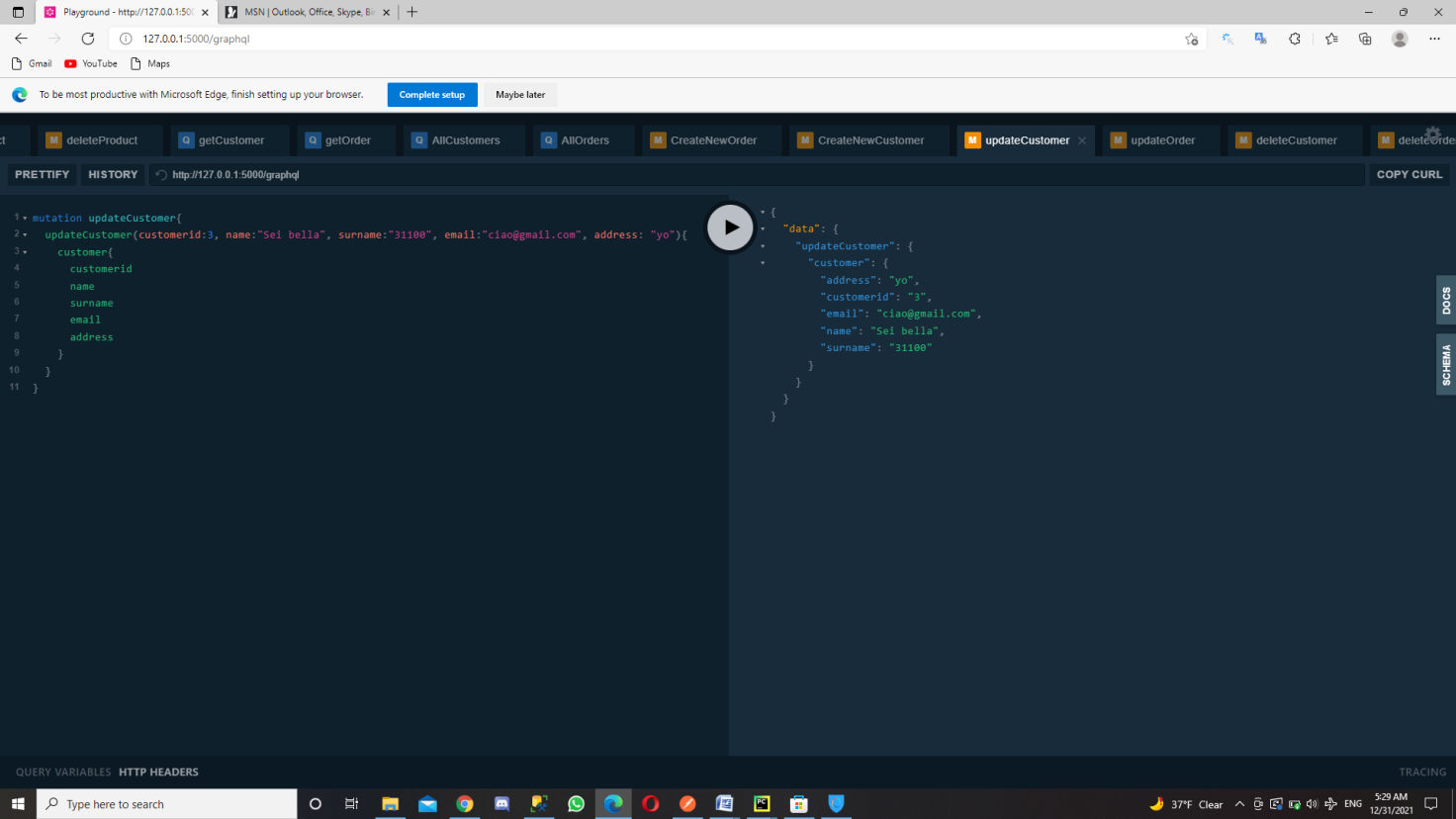


Рисунок 3.11 – Приклад виконання запиту updateCustomer

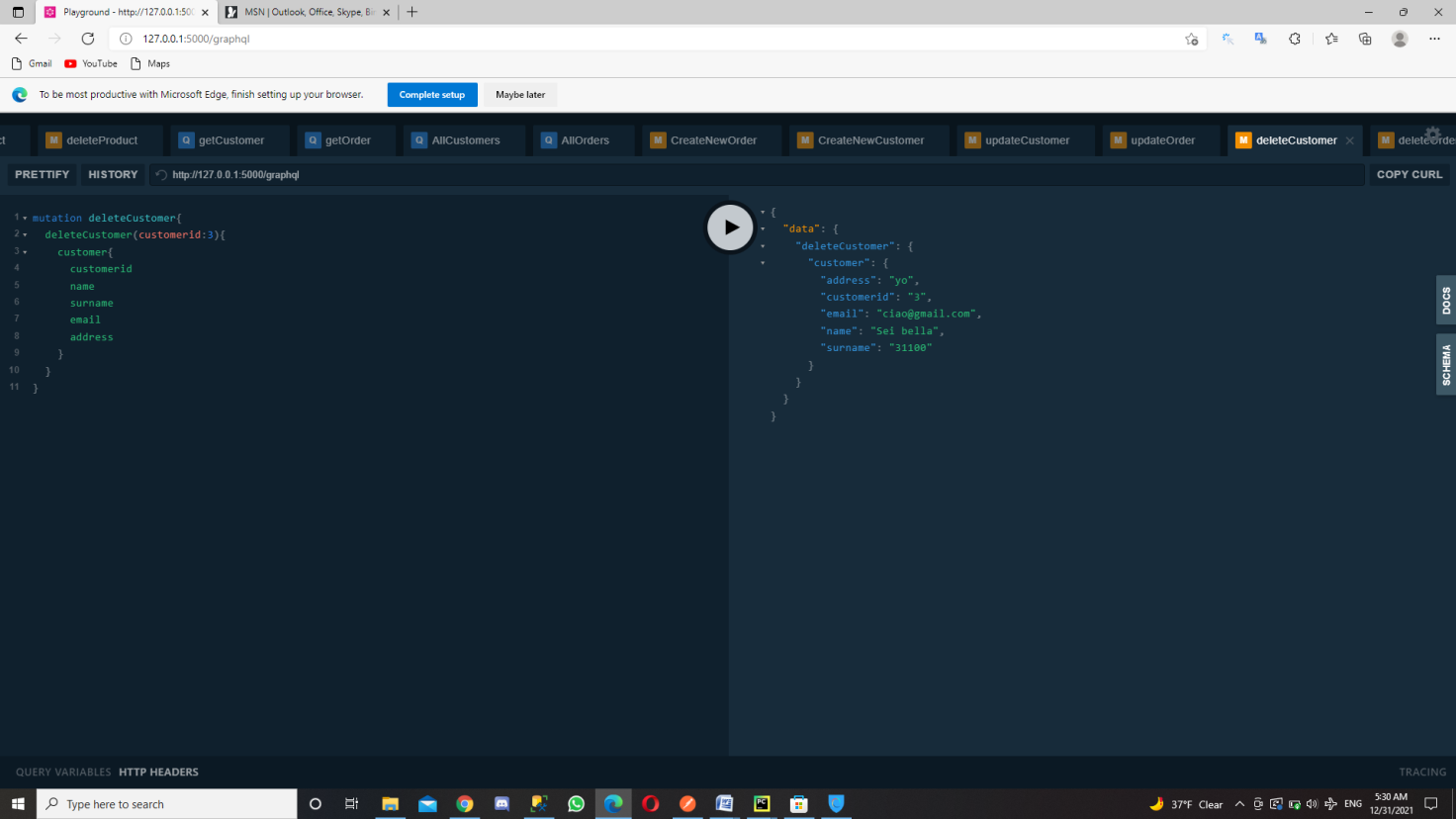


Рисунок 3.12 – Приклад виконання запиту deleteCustomer

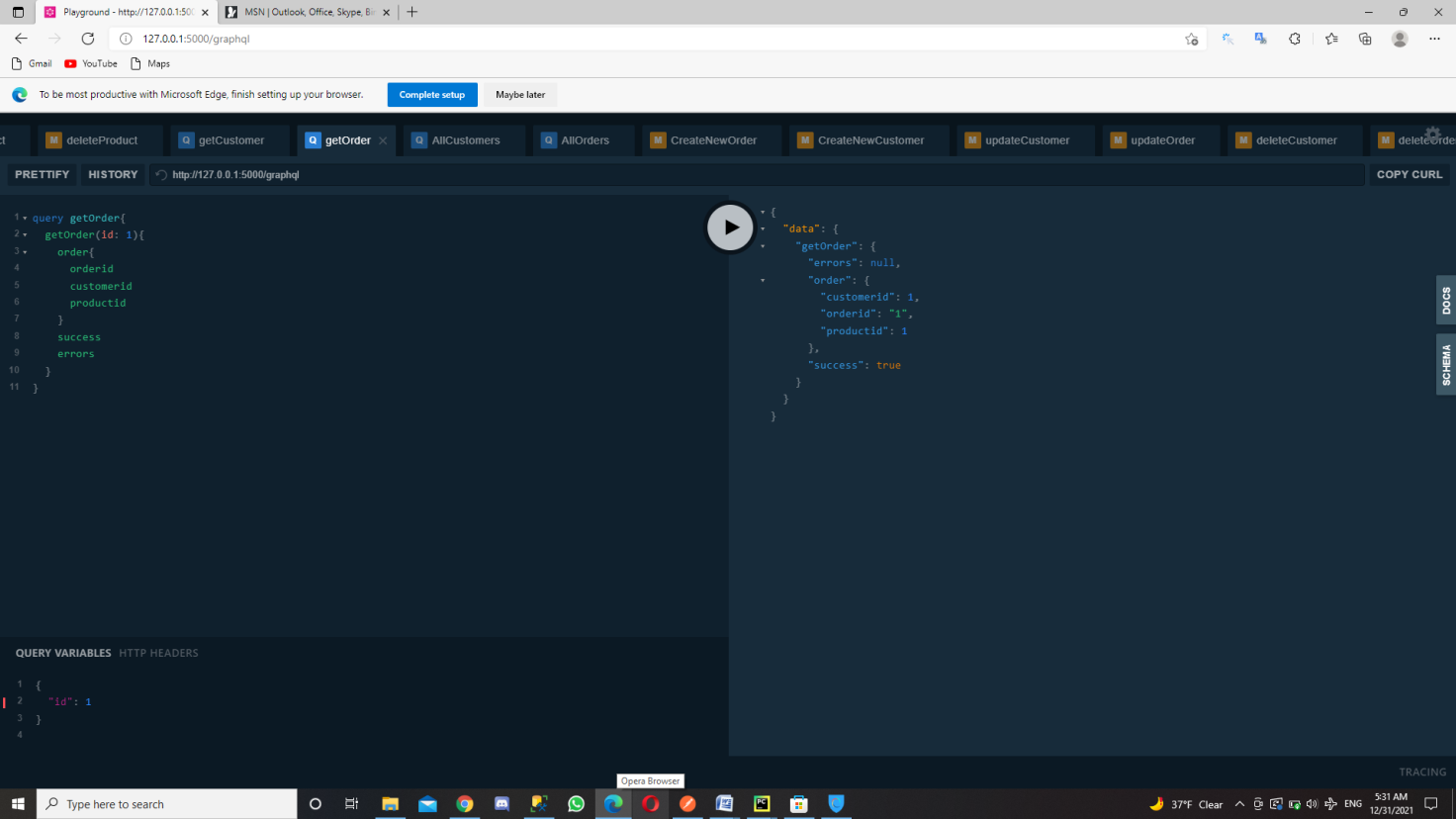


Рисунок 3.13 – Приклад виконання запиту getOrder

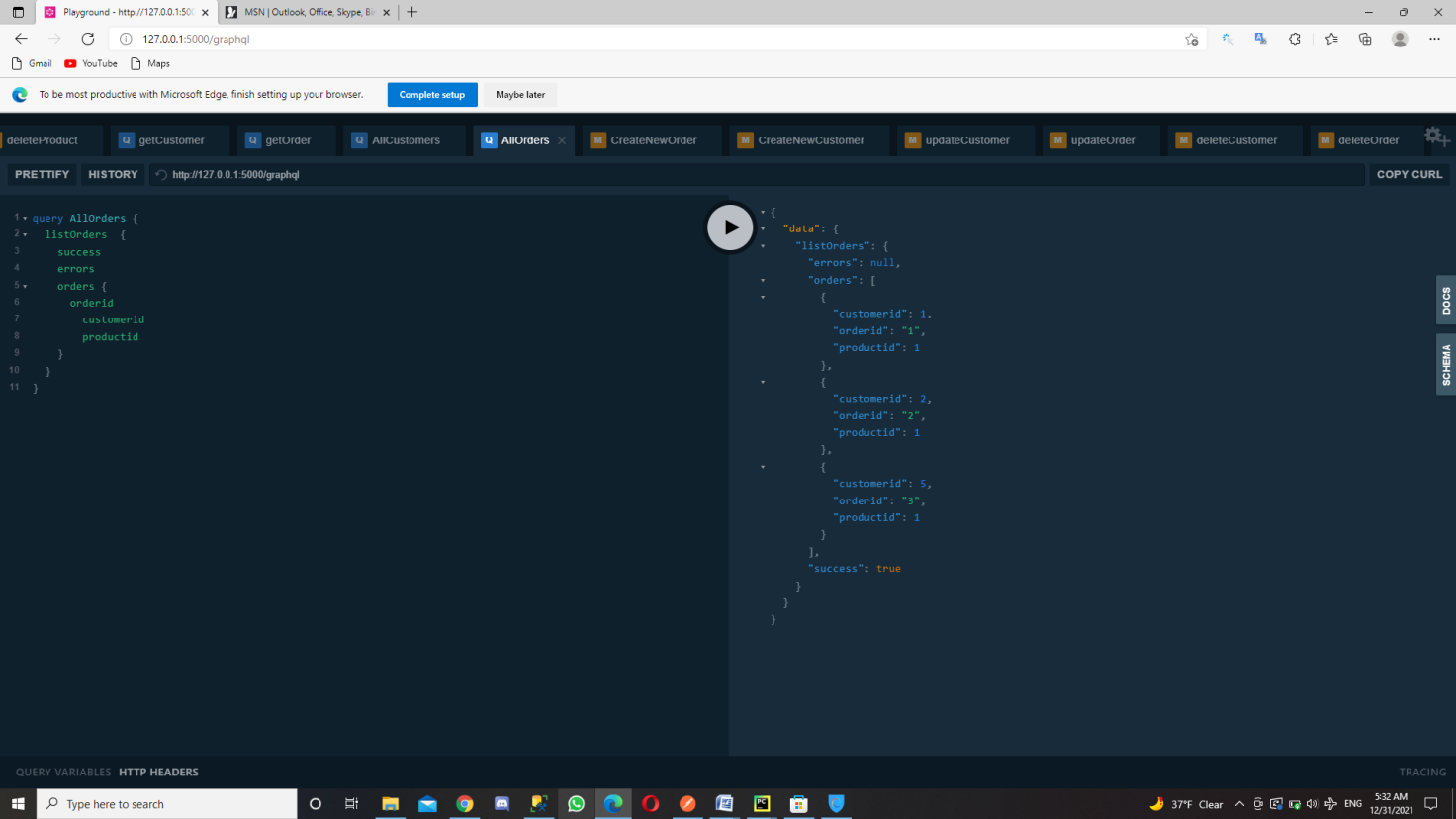


Рисунок 3.14 – Приклад виконання запиту AllOrders

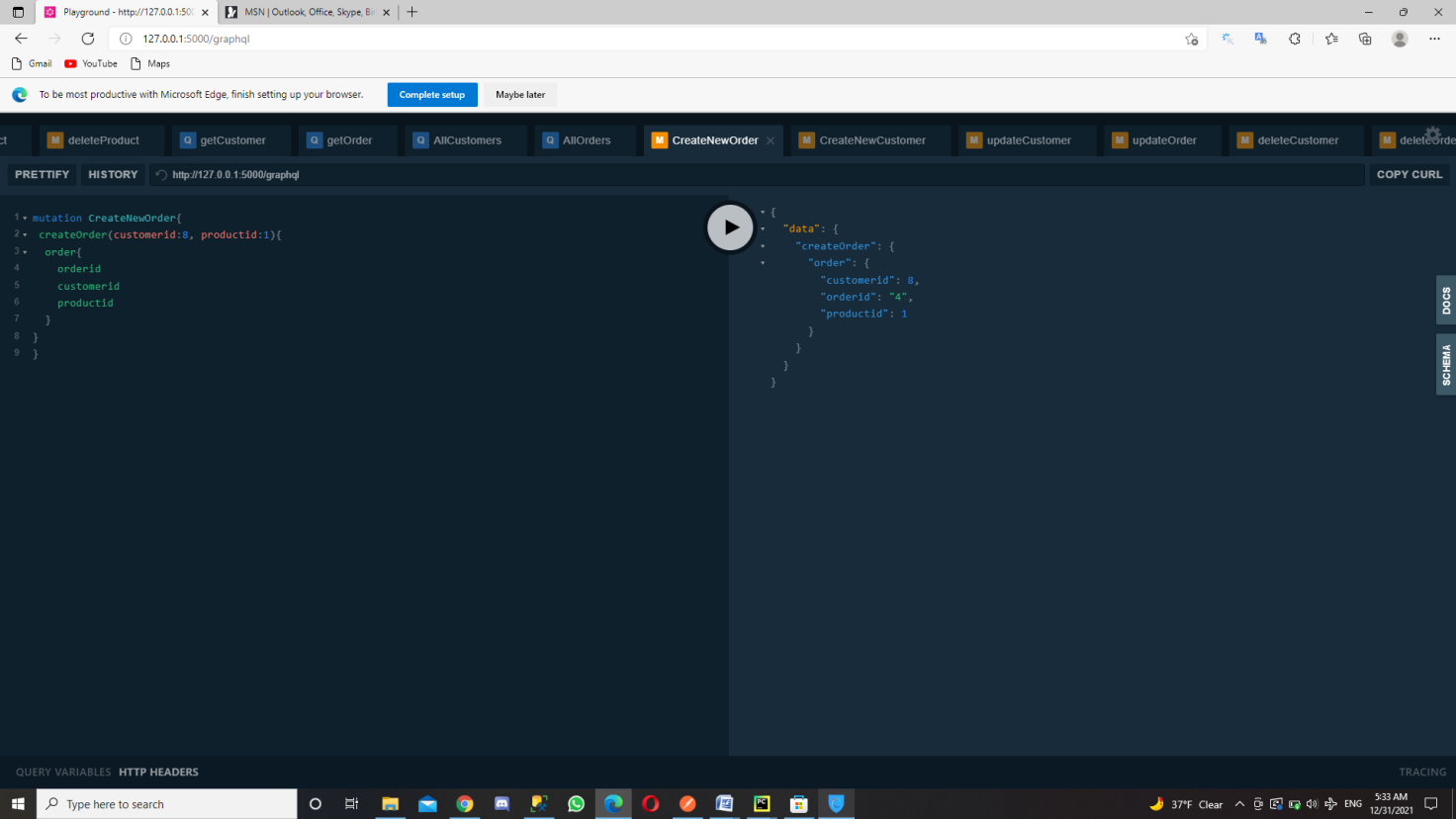


Рисунок 3.15 – Приклад виконання запиту createNewOrder

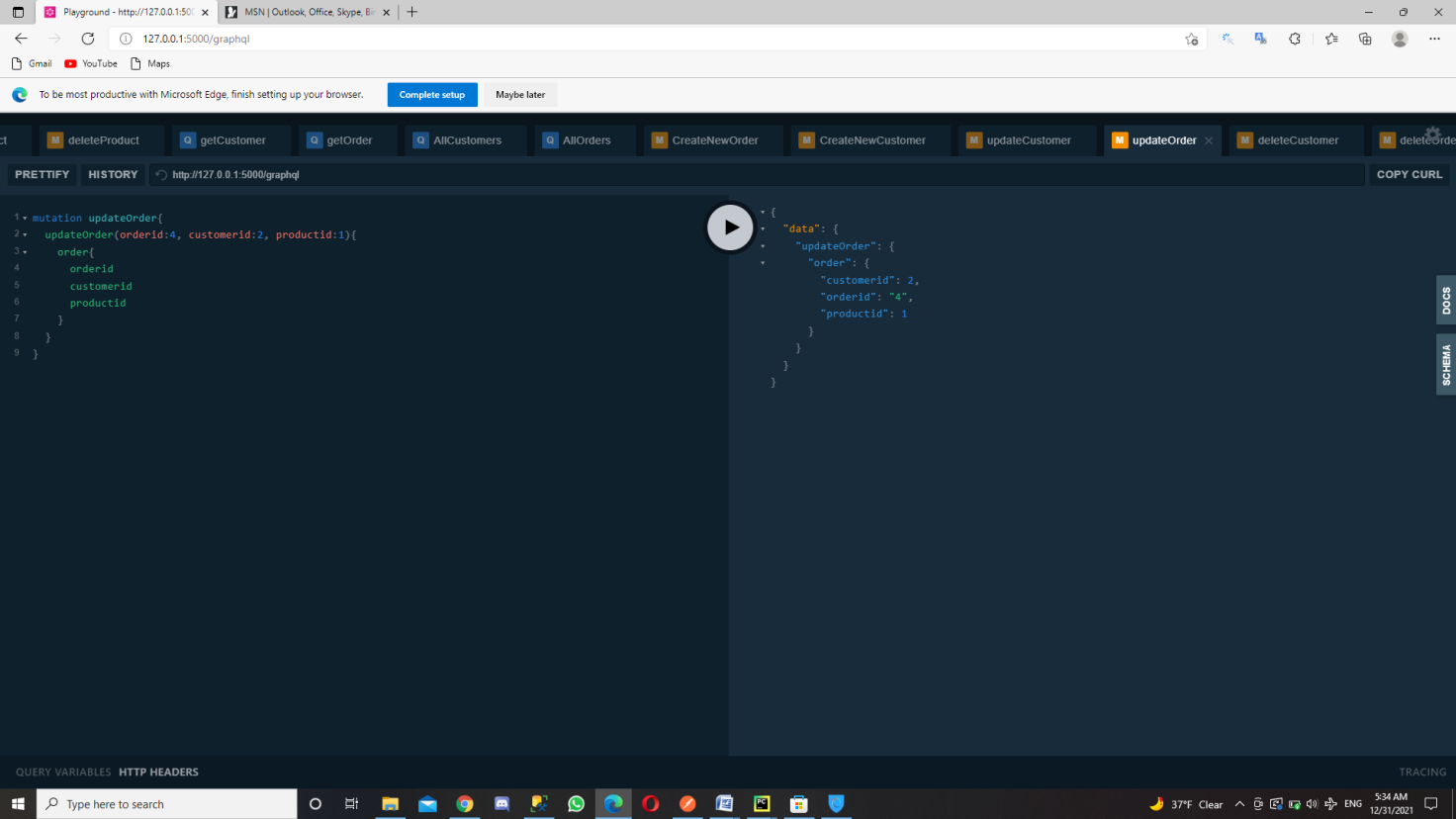


Рисунок 3.16 – Приклад виконання запиту updateOrder

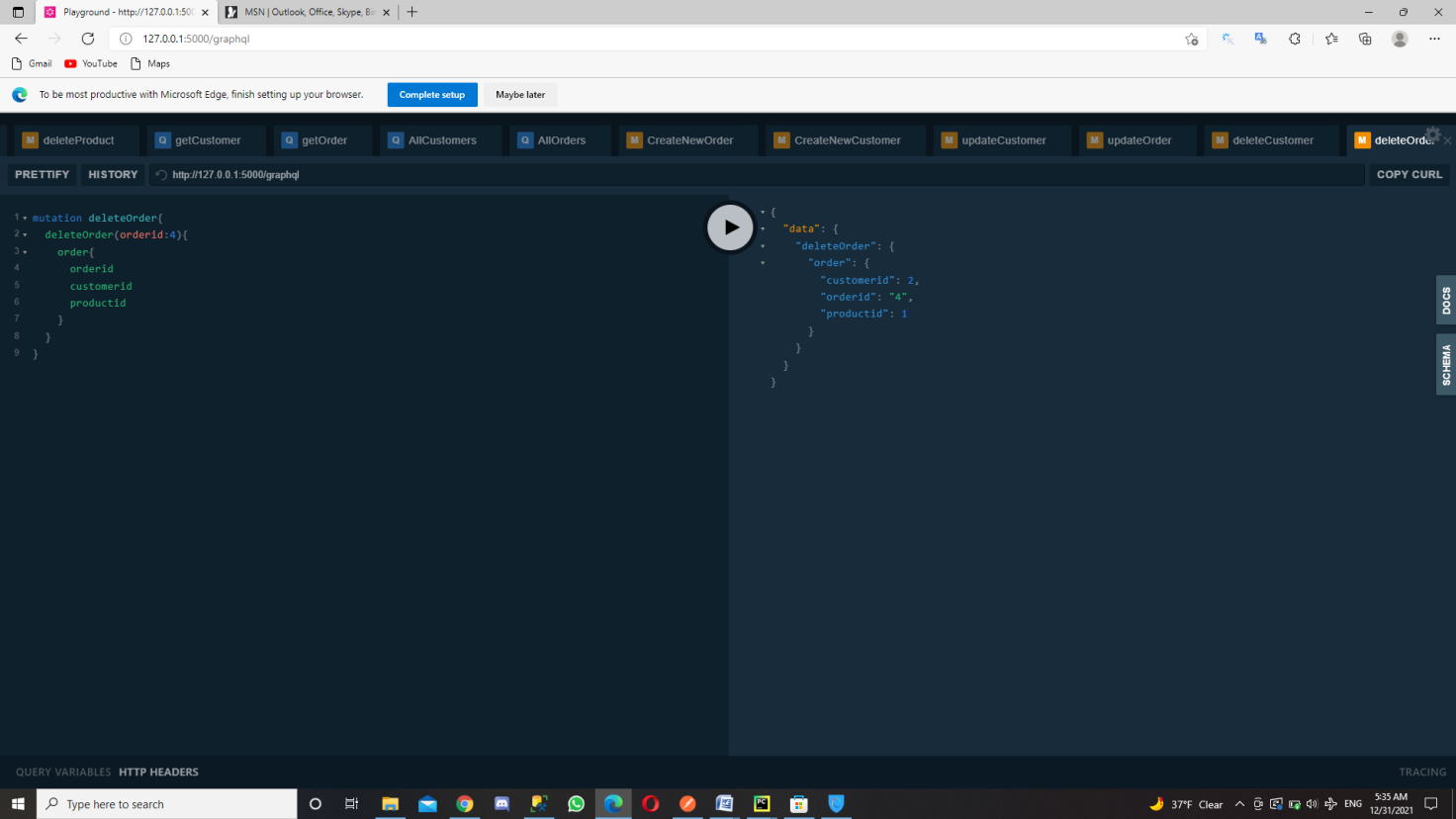


Рисунок 3.17 – Приклад виконання запиту deleteOrder

Критерії оцінювання

За умови здачі лабораторних робіт до 26.12.2021 включно максимальний бал дорівнює – 10. Після 26.12.2021 максимальний бал дорівнює – 4.

**Критерії оцінювання у відсотках від максимального балу (лабораторна робота №3):**

* заповнення БД застосунку (власної) 10%;
* робота першого постачальника 20%;
* робота другого постачальника 20%;
* запит на пошук 50%.

**Критерії оцінювання у відсотках від максимального балу (лабораторна робота №4):**

* основні сервіси 30%;
* додати сервіс бронювання 20%;
* реалізувати взаємозв’язок між сторонніми постачальниками (сервісами) та основним застосунком, має виконуватись 20%;
* забезпечити коректну роботу сервісів бронювання та пошуку 20%;
* зобразити схему архітектури 10%.

**Альтернативне завдання.**

* реалізувати GraphQL сервер 60%;
* реалізувати GraphQL Mutations 40%;