МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики
(повне найменування назва факультету)

Кафедра дискретного аналізу та інтелектуальних систем
(повна назва кафедри)

Дипломна робота

РОЗРОБКА SAAS WEB-API З ВИКОРИСТАННЯМ ГРАФОВОЇ МОДЕЛІ ДАНИХ ТА ПРОТОКОЛУ АВТОРИЗАЦІЇ OAUTH 2.0.

Виконав: студент IV курсу, групи ПМі-43с
напряму підготовки (спеціальності)
122 «Комп'ютерні науки»
(шифр і назва спеціальності)
Бондар А. С.
(підпис) (прізвище та ініціали)
Керівник _____
(підпис) (прізвище та ініціали)
Рецензент _____
(підпис) (прізвище та ініціали)

2023

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет Прикладної математики та інформатики

Кафедра Дискретного аналізу та інтелектуальних систем

Спеціальність 122 «Комп'ютерні науки»
(шифр і назва)

«ЗАТВЕРДЖУЮ»

Завідувач кафедри _____

"**31**" **серпня** **2022** **року**

ЗАВДАННЯ

НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Бондару Артуру Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка SaaS Web-API з використанням графової моделі даних та протоколу авторизації OAuth 2.0.

керівник роботи асистент кафедри дискретного аналізу та інтелектуальних систем Васильків Тетяна Ігорівна
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені Вченою радою факультету від "13" вересня 2022 року № 15

2. Строк подання студентом роботи 13.06.2023р.

3. Вихідні дані до роботи Специфікація RFC, інтернет-ресурс APIs You Won't Hate та інші, середовище виконання NodeJS, фреймворк React, база даних Neo4j, мова HTTP-запитів GraphQL та пакет для взаємодії з нею Apollo, документація NodeJS, React, Neo4j, GraphQL, Apollo відповідно.

4. Зміст дипломної роботи (перелік питань, які потрібно розробити) Вибір підходів та аналіз вимог до розробки застосунку за заданими вимогами, проектування архітектури та підбір технологій для реалізації, демонстрація роботи програми.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Схема взаємодії сервісів у протоколі авторизації OAuth 2.0., схеми архітектури застосунку, блок-схеми користувацької взаємодії із застосунком, зображення для порівняння графових структур даних з реляційними, знімки екрану, що демонструють роботу програми.

6. Консультанти розділів роботи

Розділ	**Прізвище, ініціали та посада консультанта**	**Підпис, дата**	
		завдання видав	**завдання прийняв**

7. Дата видачі завдання 31 серпня 2022 р.

Керівник роботи _____ **Васильків Т. І.**
(підпис) (прізвище та ініціали)

№ з/п	**Назва етапів дипломної роботи**	**Строк виконання етапів роботи**	**Примітка**
1.	Дослідження існуючих графових БД та підходів	02.10.2022	
	до розробки API.		
2.	Дослідження теоретичних відомостей про хмарні	15.10.2022	
	обчислення.		
3.	Дослідження теоретичних відомостей про	27.10.2022	
	розробку API.		
4.	Дослідження протоколу авторизації OAuth 2.0.	24.11.2022	
5.	Побудова архітектурних схем програми. Підбір	18.12.2022	
	технологій для розробки застосунку.		
6.	Створення основного функціоналу UI- та	07.03.2023	
	Configuration-сервісів.		
7.	Створення основного функціоналу	10.04.2023	
	Dataproxy-сервіса.		
8.	Імплементація протоколу Oauth 2.0. для	12.05.2023	
	графового API.		
9.	Порівняння графової моделі даних з реляційною	20.05.2023	
	на конкретних прикладах.		

КАЛЕНДАРНИЙ ПЛАН

Суть дипломної роботи

Робота складається із вступу, трьох розділів, висновку та джерел. Обсяг дипломної роботи: 42 сторінки та 37 рисунків. Список використаних джерел складається з 7 найменувань.

Мета даної роботи – дослідження перспектив та розробка безпечного SaaS Web-API застосунку з використанням графової моделі даних. Ця робота є вагомою, оскільки використання графових систем у веб-розробці є малодослідженим. Якщо використання графів на рівні API досягло певного поширення завдяки розвитку GraphQL, то їх використання на рівні даних все ще є доволі рідкісним. Графові бази даних суттєво молодші за реляційні, проте пропонують виняткову гнучкість та інтуїтивність завдяки тому, що граф як абстракція представлення даних набагато ближчий до різноманітних структур із нашого світу. У більшості випадків створити схему взаємодії між сутностями набагато простіше використовуючи граф, аніж таблицю. Також дослідження включає в себе застосування протоколу авторизації OAuth 2.0. для забезпечення захищеного доступу до даних (раніше не реалізованого відкрито у графових системах) та використання SaaS як моделі розповсюдження користувацьких API.

Перший розділ описує теоретичні відомості та досліджує вимоги і варіації розробки застосунку. Другий розділ описує проектування прототипу застосунку. Третій розділ демонструє взаємодію із розробленою програмою.

Новизна роботи полягає у використанні повністю графової моделі даних для побудови та дистрибуції API та реалізації протоколу авторизації OAuth 2.0. для графової системи.

Зміст

Вступ

Важливість інформації в сучасному світі зростає дуже стрімко: це можна помітити як на побутовому рівні, так і звернувшись до наукових досліджень. До прикладу, дослідження наукових працівників Університету Берклі у 2003 році [1] показало, що у попередньому 2002 році людством було вироблено інформації об'ємом у $18 \cdot 10^{18}$ байт (18 ексабайт), а за п'ять минулих років людством було вироблено інформації більше, ніж за всю попередню історію. Крім того, дослідники зафіксували темп приросту, рівний 30% на рік. Хоч і, на жаль, схожих досліджень останнім часом проведено не було, можна із деякою упевненістю вважати, що ця динаміка якщо не зросла, то, як мінімум, не спала.

Основним джерелом інформації у наш час є Інтернет, і веб-застосунки як засіб для роботи з даними потрібні всюди та дуже швидко, умови ведення розробки в Інтернеті не дозволяють витрачати багато часу на їх виробництво. Проміжною ланкою між клієнтом (кінцевим веб-застосунком) та даними зазвичай виступає API - Application Programming Interface.

Потреба в API існує у кожному веб-сервісі, а крім цього вони можуть використовуватись у машинному навчанні, інтернеті речей, тощо. Закономірно, для полегшення розробки тривіальних рішень були створені інструменти в області хмарних обчислень, класифіковані як SaaS (Software as a Service), які вже успішно використовуються в невибагливих системах, як от Content Management Systems, проте їм, зазвичай, бракує гнучкості.

Паралельно з цим у світі інформаційних технологій зросла важливість *зв'язків* (в широкому значенні цього слова). Ця тенденція проявляється на всіх рівнях: розбиття "монолітних" застосунків на зв'язні мікросервіси, використання сторонніх сервісів, пов'язаних з основною аплікацією за певним строго визначеним протоколом (зокрема стандарти авторизації через "третю сторону", як от OpenID Connect та OAuth). Чи не найяскравіше проблема проявилась у сфері даних: найбільш поширена, заснована на строгих правилах реляційної

алгебри реляційна модель не завжди дозволяє знайти просте, ефективне та “елегантне” рішення для зберігання та пошуку інформації. В результаті логічного проектування з’являється множина “таблиць”, яка може ускладнити розуміння системи та затруднити пошук строгими правилами. Відповідно до цього почали набирати популярність альтернативні моделі репрезентації даних, серед них документно-орієнтована, яка, хоч і забезпечує підвищену швидкість доступу до інформації, залишається абсолютно куцою при роботі зі зв’язками.

Однією із найбільш розповсюджених структур даних є графова, яка найкраще підходить для репрезентації даних, у яких інформаційне навантаження *зв’язків між об’єктами* важить не менше, ніж інформація, що зберігається у самих об’єктах. Крім того, будова графа набагато більш наближена до багатьох природних структур (як графи зручно уявляти молекули, екосистеми, ієрархії, тощо), що може значно спростити взаємодію та поліпшити ефективність у роботі з графовою системою.

Саме тому в якості дипломної роботи я вирішив розробити SaaS-застосунок з використанням графової моделі даних та підсилений протоколом авторизації OAuth 2.0, щоб створити інфраструктуру, яка відповідає викликам сучасності у сфері нестандартної роботи з даними та задовольняє вимоги з безпеки користувача.

Постановка задачі

Розробити APIaaS (API as a Service) на базі графової моделі даних та протоколу авторизації OAuth 2.0.

Під час виконання курсової роботи необхідно виконати такі завдання:

1. Ознайомитись з теоретичною базою та вимогами до розробки API та програмного забезпечення у сфері хмарних обчислень, закрема SaaS, дослідити специфікацію стандарту OAuth 2.0.
2. Спроектувати веб-сервіс: підібрати технології, визначити архітектуру, спроектувати базу даних.
3. Розробити веб-сервіс згідно опрацьованих вимог та заданої архітектури.
4. Протестувати розроблений APIaaS.

Розділ 1. Теоретичні відомості

1.1. Розробка програмного забезпечення у галузі хмарних обчислень

1.1.1. Хмарні обчислення

Згідно Національного Інституту Стандартів та Технологій США, хмарні обчислення — це модель для забезпечення повсюдного, зручного мережевого доступу “на вимогу” до спільного пулу обчислювальних ресурсів, що підлягають налаштуванню (наприклад, до комунікаційних мереж, серверів, засобів збереження даних, прикладних програм та сервісів), і які можуть бути оперативно надані та звільнені з мінімальними управлінськими витратами та зверненнями до провайдера [2].

При використанні хмарних обчислень програмне забезпечення надається користувачеві як Інтернет-сервіс. Користувач має доступ до власних даних, але не може управляти і не повинен піклуватися про інфраструктуру, операційну систему і програмне забезпечення, з яким він працює. «Хмарою» метафорично називають інтернет, який приховує всі технічні деталі. Згідно з документом IEEE, опублікованим у 2008 році, «хмарні обчислення — це парадигма, в рамках якої інформація постійно зберігається на серверах у мережі інтернет і тимчасово кешується на клієнтській стороні, наприклад на персональних комп'ютерах, ігрових приставках, ноутбуках, смартфонах тощо».

Хмарні обчислення — це не єдина технологія, а швидше галузь, яка інкорпорує різні моделі надання послуг. Ось основні із них:

- Програмне забезпечення як послуга (англ. *Software-as-a-Service, SaaS*) це модель розповсюдження програмного забезпечення, в якій постачальник розміщує програми та робить їх доступними для кінцевих користувачів через Інтернет.
- Інфраструктура як послуга (англ. Infrastructure-as-a-Service, IaaS) це модель обслуговування, в межах якої споживачу надається можливість керувати

засобами обробки та збереження, комунікаційними мережами, та іншими фундаментальними обчислювальними ресурсами, на базі яких споживач може розгортати та виконувати довільне програмне забезпечення. Провайдер підтримує віртуалізацію, серверне обладнання, сховище, мережі.

- Платформа як послуга (англ. Platform-as-a-Service, PaaS) вважається найскладнішим із трьох моделей хмарних обчислень. PaaS має певну схожість із SaaS, основна відмінність полягає в тому, що замість доставки програмного забезпечення в Інтернеті, це, насправді, платформа для створення програмного забезпечення, яке постачається через Інтернет. Провайдер підтримує час виконання, інтеграцію SOA, бази даних, серверне програмне забезпечення. Прикладами таких сервісів є Salesforce та Heroku.

1.1.2. SaaS (Програмне забезпечення як сервіс)

Програмне забезпечення як послуга (SaaS) — це програмне забезпечення, яке належить, постачається та керується віддалено одним або кількома постачальниками. Постачальник постачає програмне забезпечення на основі одного набору загальних визначень коду та даних, який споживається в моделі “один-до-багатьох” усіма замовниками в будь-який час.

Це означає, що програма працюватиме в єдиній версії та конфігурації для всіх клієнтів або орендарів. Хоча різні клієнти, які підписалися, працюватимуть в одному екземплярі хмари зі спільною інфраструктурою та платформою, дані кожного із клієнтів все одно будуть розділені.

Типова “багатоорендна” (англ. *multi-tenant*) архітектура програм SaaS означає, що постачальник хмарних послуг може обслуговувати, керувати оновленнями та виправленням помилок швидше, простіше та ефективніше. Замість того, щоб впроваджувати зміни в кількох екземплярах, інженери можуть

вносити необхідні зміни для всіх клієнтів, підтримуючи один спільний екземпляр.

1.1.2.1. Вимоги до розробки SaaS

- **Аудиторський контроль безпеки даних.** Це означає, що постачальник SaaS повинен регулярно проводити аудит додатків сторонніх розробників та бути готовим засвідчити їх безпечність клієнту. Існує безліч стандартів, які регулюють аудит безпеки, але одним із найпоширеніших SAS-70. Іншими популярними стандартами є SysTrust, WebTrust та ISO 27001.
- **Висока доступність системи.** Навіть якщо програма чи мережа на короткий час недоступні (така вже природа Інтернету), у розробника SaaS немає виправдання для втрати даних користувача. Для цього часто використовується підхід із використанням реплікації.
- **Розподілені та стійкі до стихійних лих центри обробки даних.** Розподіленість центрів обробки даних є одним із методів, які використовуються розробниками SaaS для досягнення високої доступності. Також, знаходження даних в кількох географічно розкиданих дата-центрах є важливою, оскільки вона захищає користувачів від різноманітних регіональних катастроф - як техногенних, так і природних, - а також тимчасових перевантажень Інтернету, які можуть вплинути на час відповіді аплікації.
- **Інтеграція з іншими сервісами.** SaaS - це, у першу чергу, гнучкий сервіс, який має бути однаково легким як для інтеграції з системою клієнта, так і для внутрішньої інтеграції зі сторонніми сервісами.

1.1.3. APIaaS (API як сервіс)

APIaaS — це підкатегорія SaaS, яка дозволяє створювати та розміщувати API (інтерфейси програмного забезпечення, або ж прикладні програмні

інтерфейси - надалі використовуватиму другий переклад). Для того, щоб краще розібратись із цим терміном, визначимо термін API.

1.2. API (Application Programming Interface)

1.2.1. Визначення

Прикладний програмний інтерфейс (API) — це зв’язок між комп’ютерами або між комп’ютерними програмами [3]. Це тип програмного інтерфейсу, який пропонує послуги іншим частинам програмного забезпечення.

На відміну від інтерфейсу користувача, який з'єднує комп'ютер з людиною, інтерфейс прикладного програмування з'єднує комп'ютери або частини програмного забезпечення один з одним. Він не призначений для безпосереднього використання особою (кінцевим користувачем), крім програміста, який вбудовує його в програмне забезпечення. API часто складається з різних частин, які діють як інструменти або послуги, доступні програмісту. Вважається, що програма або програміст, який використовує одну з цих частин, *викликає* цю частину API. Виклики, які складають API, також називають методами, запитами або, найчастіше, кінцевими точками (англ. *endpoint*). Ці кінцеві точки визначаються та документуються специфікацією API.

Однією з цілей API є приховати внутрішні деталі роботи системи, відкриваючи лише ті частини, які будуть корисні для користувача-програміста, і узгоджено підтримувати їх, згідно специфікації, навіть якщо внутрішні деталі функціоналу з часом зміняться.

API може бути спеціально створений для конкретної “екосистеми” програмного забезпечення, або ж це може бути загальний стандарт, який забезпечує взаємодію між багатьма системами.

Термін API найчастіше використовується для позначення веб-API, які дозволяють спілкуватися між комп’ютерами через Інтернет. Останні розробки у сфері застосування API призвели до зростання популярності мікросервісів, які в

кінцевому підсумку є слабо пов’язаними службами, доступ до яких здійснюється через публічні API.

1.2.2. Вимоги до розробки веб-API

Дизайн прикладного програмного інтерфейсу значним чином залежить від вибраного підходу: розділяють RPC, REST, SOAP, GraphQL та інші [4]. Варто зазначити, що не існує конкретних протоколів веб-API, кожен з підходів - це всього лиш уособлений набір практик та характеристик, і кожен із них має свою власну специфікацію, проте все ще можна виділити деякі ключові принципи:

- **Приховування інформації.** Користувач не має знати деталей реалізації внутрішніх модулів.
- **Безпека.** Поширеними загрозами для публічних API є, зокрема, SQL injection, DoS атаки, порушення автентифікації та розкриття конфіденційних даних.
- **Здатність до еволюції.** Веб-API повинен мати можливість розвиватися та додавати функціональні можливості незалежно від клієнтських програм. У міру розвитку API існуючі клієнтські програми повинні продовжувати функціонувати без змін.
- **Підтримка CRUD-операцій.** CRUD (Create, Read, Update, Delete) - це набір з 4 основних функцій для управління даними.

Таким чином, APIaaS є SaaS, в якому сервісом для розміщення виступає API. Усі вимоги до розробки SaaS застосовні до APIaaS, і крім цього до них додаються:

- Можливість створювати, тестувати та розгортати власні служби API.
- Можливість підключення сторонніх API.

1.2.3. GraphQL як мова запитів та підхід до розробки API

GraphQL (Graph Query Language) - це мова запитів для API та середовище для виконання цих запитів із наявними даними. Формулювання запиту з

GraphQL пропонує виняткову гнучкість та конкретність, адже GraphQL як середовище надає доступ до повноцінної ієрархічної строго типізованої структури усього API. Описуючи більш детально, основними перевагами GraphQL є:

- Гнучкість: GraphQL дозволяє клієнту отримувати саме ті дані, які йому потрібні, тому що запит на сервер може містити тільки ті поля, які потрібні клієнту. Це дозволяє зменшити кількість передаваних даних та покращити швидкість їх передачі.
- Масштабованість: GraphQL дозволяє підключати різні джерела даних до одного запиту. Наприклад, клієнт може отримувати дані з бази даних та з зовнішніх API в одному запиті. Це дозволяє зменшити кількість запитів до сервера та покращити швидкість отримання даних.
- Типізація: GraphQL має вбудовану систему типів, що дозволяє перевірити коректність запиту та отриманих даних ще до їх передачі клієнту. Це дозволяє зменшити кількість помилок та підвищити якість коду.
- Кешування: GraphQL має вбудовану систему кешування, що дозволяє зберігати результати запитів та повторно використовувати їх, коли клієнт знову запитує ті ж самі дані. Це дозволяє зменшити кількість запитів до сервера та покращити швидкість відгуку сервера.
- Покращена комунікація: GraphQL дозволяє побудувати більш ефективну комунікацію між клієнтом та сервером, оскільки клієнт може точно вказати, які дані йому потрібні та в якому форматі він хоче їх отримати. Це дозволяє зменшити кількість непотрібних даних та зробити комунікацію між клієнтом та сервером більш ефективною та прозорою.

З усього вищезазначеного не зовсім зрозуміло, чому GraphQL, окрім мови запитів та середовища їх виконання, вважається підходом до розробки API. Справді, API побудоване з використанням GraphQL істотно є лише RPC (Remote Procedure Call) застосунком з багатьма хорошими ідеями запозиченими у REST/HTTP [5]. Тим не менш, це одна з найбільш швидко зростаючих екосистем API, здебільшого через певну плутанину, описану вище, яка зокрема виникла тому, що GQL був створена компанією Facebook, яка до того конструювала в основному REST-подібні API.

Компанія була в пошуку пригожого підходу для побудови веб-програмного інтерфейсу, що зумів би ефективно ділитись даними з усіма різноманітними сервісами Facebook. Зрештою, вони розробили цю систему мови запитів у стилі RPC, щоб ігнорувати більшу частину транспортного рівня, тобто вони мали повний контроль над концепціями веб-API. Ендпоінти зникли, ресурси, які оголошували про власну можливість кешування, зникли, потреба єдиного інтерфейсу (uniform interface у специфікації REST) стерта, і усі вищеперелічені аспекти роблять GraphQL настільки неймовірно простим, що його можна вписати в AMQP або будь-який інший транспортний протокол.

Основна перевага GraphQL полягає в тому, що за замовчуванням він забезпечує найменшу відповідь від API, оскільки ви запитуєте лише певні біти даних, які вам потрібні, що мінімізує частину завантаження вмісту HTTP-запиту.

Це також зменшує кількість HTTP-запитів, необхідних для отримання даних для кількох ресурсів, відомих як “проблема HTTP N+1”.

Враховуючи вищеперелічене та незважаючи на деякі присутні в GraphQL недоліки, як от складнощі з представленням стану (побудовою скінченного автомата) а також проблеми з HTTP-кешуванням, можна зробити висновок, що GraphQL найкраще підходить для відкритих API з великою кількістю клієнтів та для “легких” програм, для яких важлива швидкість та простота розробки, адже рушій GraphQL сам відповідає за валідацію схеми запиту, надає базовий користувацький інтерфейс та є простим для опанування. З цього зокрема

випливає, що він є підходящим вибором для побудови гнучкого та зручного для інтеграції API, що є однією з вимог до побудови APIaaS. Крім того, GraphQL створює та розглядає структуру даних сервера як граф, що робить його хорошим вибором при використанні графової БД, забезпечуючи повністю графову модель даних.

1.3. Забезпечення безпечного доступу до даних в SaaS

Зазвичай, на практиці навіть найпростіші відкриті API потребують авторизації для захисту даних, а для більш комплексних систем з великою кількістю сутностей цей процес буде лише ускладнюватись.

Також, як було описано вище, SaaS застосунок має бути однаково легким як для інтеграції з системою клієнта, так і для внутрішньої інтеграції зі сторонніми сервісами, і чи не найважливішою такою інтеграцією зазвичай є інтеграція з сервісами автентифікації та авторизації. Потреби в цих процесах можуть різнитись від користувача до користувача, тому реалізувати одну “авторську” опцію не є рішенням, що відповідає цим вимогам.

При створенні таких інтеграцій, однією із найскладніших частин є робота з нюансами програм сторонніх розробників. Для імплементації подібних рішень зазвичай звертаються до вже описаних протоколів, і у випадку з авторизацією для SaaS-застосунків найпоширенішим рішенням є OAuth 2.0.

1.3.1. Протокол авторизації OAuth 2.0.

OAuth 2.0 - це протокол авторизації, який забезпечує зручний та безпечний доступ до захищених ресурсів користувача через веб- та мобільні додатки. Він описує механізми, які дозволяють клієнтським додаткам отримувати доступ до захищених ресурсів, таких як профілі користувачів, без необхідності передавати їхні логіни та паролі.

Згідно зі стандартом RFC 6749 [6], “платформа авторизації OAuth 2.0 дає змогу стороннім додаткам отримувати обмежений доступ до HTTP-сервера, або

від імені власника ресурсу, організовуючи взаємодію затвердження (approval interaction) між власником ресурсу та HTTP-сервером, або дозволяючи програмі “третьої сторони” отримати доступ від свого власного імені. Специфікація OAuth 2.0. замінює та застаріває протокол OAuth 1.0, описаний у RFC 5849”.

У традиційній моделі автентифікації клієнт-сервер клієнт запитує ресурс з обмеженим доступом (захищений ресурс) на сервері шляхом автентифікації на цьому сервері за допомогою облікових даних власника ресурсу. Щоб надати стороннім програмам доступ до обмежених ресурсів, власник ресурсу ділиться своїми обліковими даними з “третьою стороною”. Це створює кілька проблем і обмежень:

- Сторонні програми повинні зберігати облікові дані власника ресурсу для подальшого використання, як правило, пароль у відкритому вигляді.
- Сервери повинні підтримувати автентифікацію паролів, незважаючи на недоліки безпеки, властиві паролям.
- Програми сторонніх розробників отримують надто широкий доступ до захищених ресурсів власника ресурсу, залишаючи власників ресурсів без можливості обмежувати тривалість або доступ до обмеженої підмножини ресурсів.
- Власники ресурсів не можуть скасувати доступ до окремої “третьої сторони”, не скасувавши доступ для всіх третіх сторін, і повинні зробити це, змінивши пароль “третьої сторони”.
- Компрометація будь-якої сторонньої програми призводить до компрометації пароля кінцевого користувача та всіх даних, захищених цим паролем.

OAuth вирішує ці проблеми, запроваджуючи рівень авторизації та відокремлюючи роль клієнта від ролі власника ресурсу. У OAuth клієнт (будь-який веб-застосунок) запитує доступ до ресурсів, які контролюються власником ресурсу та розміщені на сервері ресурсів, і отримує інший набір облікових даних,

ніж у власника ресурсу. Замість того, щоб використовувати облікові дані власника ресурсу для доступу до захищених ресурсів, клієнт отримує токен доступу (access token) — рядок, що позначає конкретну область, час життя та інші атрибути доступу. Токени доступу видаються стороннім клієнтам сервером авторизації зі схвалення власника ресурсу. Клієнт використовує токен доступу для доступу до захищених ресурсів, розміщених на сервері ресурсів. Наприклад, кінцевий користувач (власник ресурсу, resource owner) може надати службі друку (клієнту, client) доступ до своїх захищених фотографій, які зберігаються в службі обміну фотографіями (сервер ресурсу, resource server), не повідомляючи своє ім’я користувача та пароль службі друку. Замість цього він автентифікується безпосередньо на сервері авторизації, якому довіряє служба обміну фотографіями (auth server), який видає облікові дані для делегування служби друку (токен доступу).

Оскільки OAuth 2.0. це стандарт, що з часом доповнюється та розвивається, у його специфікації є декілька “потоків” (flows). Вищерозглянутий приклад відповідає потоку Authorization Code Grant і є застосовним у більшості випадків, проте варто зазначити, що окремі ситуації потребують відмінних підходів: зокрема, для авторизації девайсів, як от ігрових консолей чи смарт-телевізорів, використовується Device Authorization Grant Flow, для мобільних застосунків може використовуватись Resource Owner Password Credentials Grant Flow, тощо.

Схему авторизації за специфікацією Authorization Code Grant Flow зображено на рисунку 1.1:

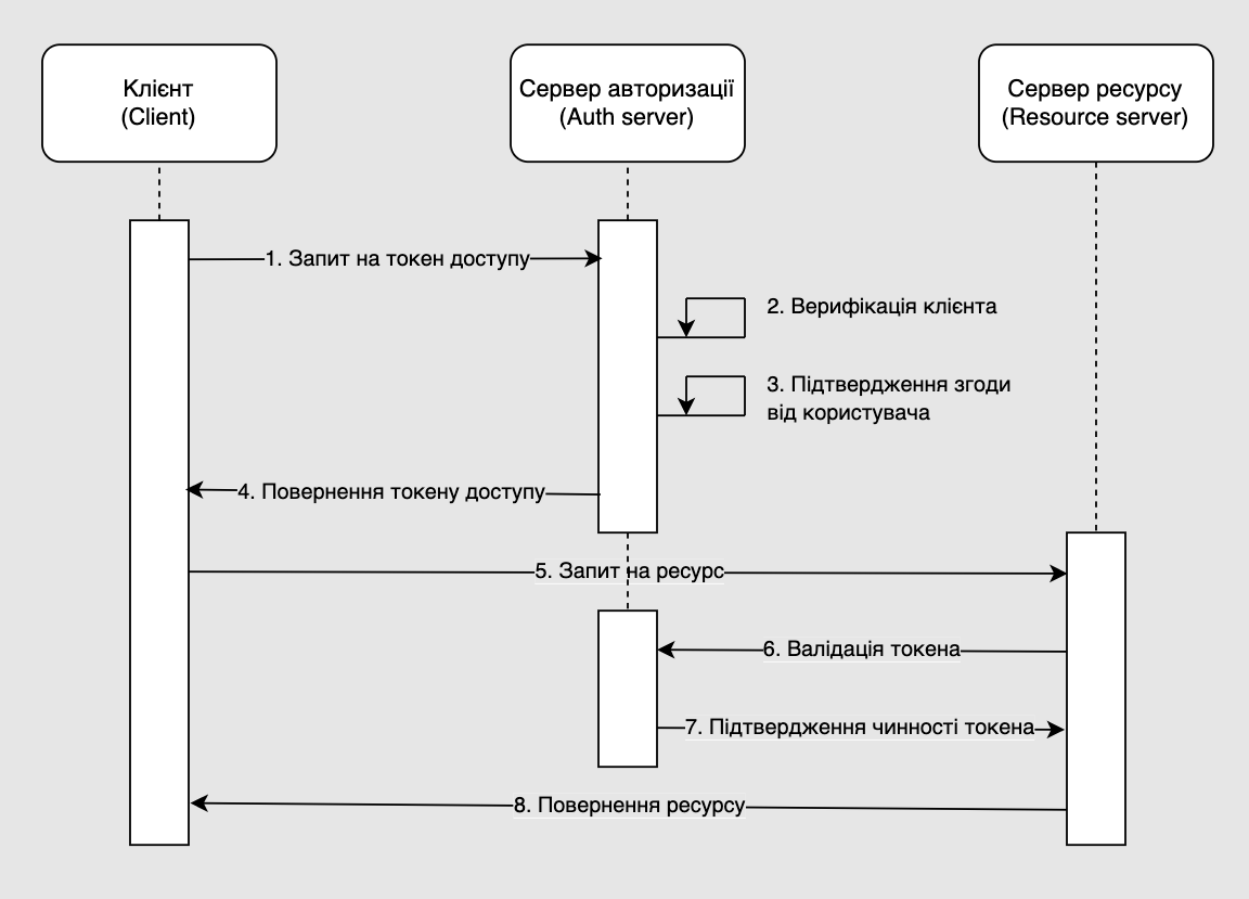

Рисунок 1.1 – OAuth 2.0. Authorization Code Grant Flow

Розділ 2. Проектування SaaS Web-API з використанням графової моделі даних

Ознайомившись із загальноприйнятими вимоги до розробки хмарних сервісів та API, можна перейти до проектування конкретного прикладу.

2.1. Загальний опис

Як вже було зазначено у вступі, головною ціллю цієї роботи було створення гнучкого SaaS Web-API, і гнучкість може забезпечити графова модель даних. Застосунок-прототип має надавати інтерфейс для створення та розповсюдження графових API, а надалі безпечний доступ до користувацьких API, забезпечувати їх безпеку та консистентність.

2.2. Структура

Оскільки SaaS використовує *multi-tenant* архітектуру, в структурі будь-якого SaaS-застосунку має прослідковуватись відношення “один-до-багатьох” (один постачальник, від якого залежить багато веб-сервісів-клієнтів). Окреслимо дизайн майбутнього застосунку таким чином:

- **Configuration service**: сервіс-постачальник. Шляхом взаємодії з цим сервісом користувач може завідувати своїм клієнтським сервісом: модифікувати його схему, змінювати конфігурацію, надавати доступ іншим користувачам. База даних цього сервісу-постачальника зберігатиме інформацію про користувачів та конфігурацію користувацьких сервісів.
- **UI service**: користувацький інтерфейс для сервісу-постачальника. Надає графічний веб-інтерфейс для всіх функцій Configuration service.
- **Auth service**: сервер авторизації у схемі OAuth 2.0. Authorization Code Grant Flow. Цей сервіс відповідає за перевірку особи

користувача та надання облікових даних для автентифікації, які можна використовувати для доступу до захищених ресурсів на інших серверах (у цьому конкретному випадку лише configuration service, проте уявимо, що в проект необхідно додати ще один сервіс - наприклад, сервіс імпорту даних із файлів. Тоді цей сервіс теж буде користуватись сервером авторизації). Коли користувач намагається отримати доступ до захищеного ресурсу, система або служба може зробити запит на автентифікацію від auth-сервісу, який потім перевіряє особу користувача та повертає токен доступу.

- **Dataproxy service**: користувацький сервіс. Безпосередньо сервіс клієнта, який створюється, модифікується, та проходить аудит зі сторони сервісу-постачальника. В базі даних Dataproxy зберігається користувацька інформація, права доступу до якої задаються Configuration-схемою, проте до самої інформації Configuration сервіс достукатись ніяк не може (зберігається принцип безпеки даних).
- **Dataproxy authentication service**: користувацький auth-сервіс. Може використовуватись користувачем як сервер авторизації у власній екосистемі, може бути замінений на будь-яке інше готове рішення. Шлях до інтеграції вільний.

Загальна структура прототипу проекту та блок-схеми взаємодії з сервісами зображені на рисунках 2.1 - 2.4.

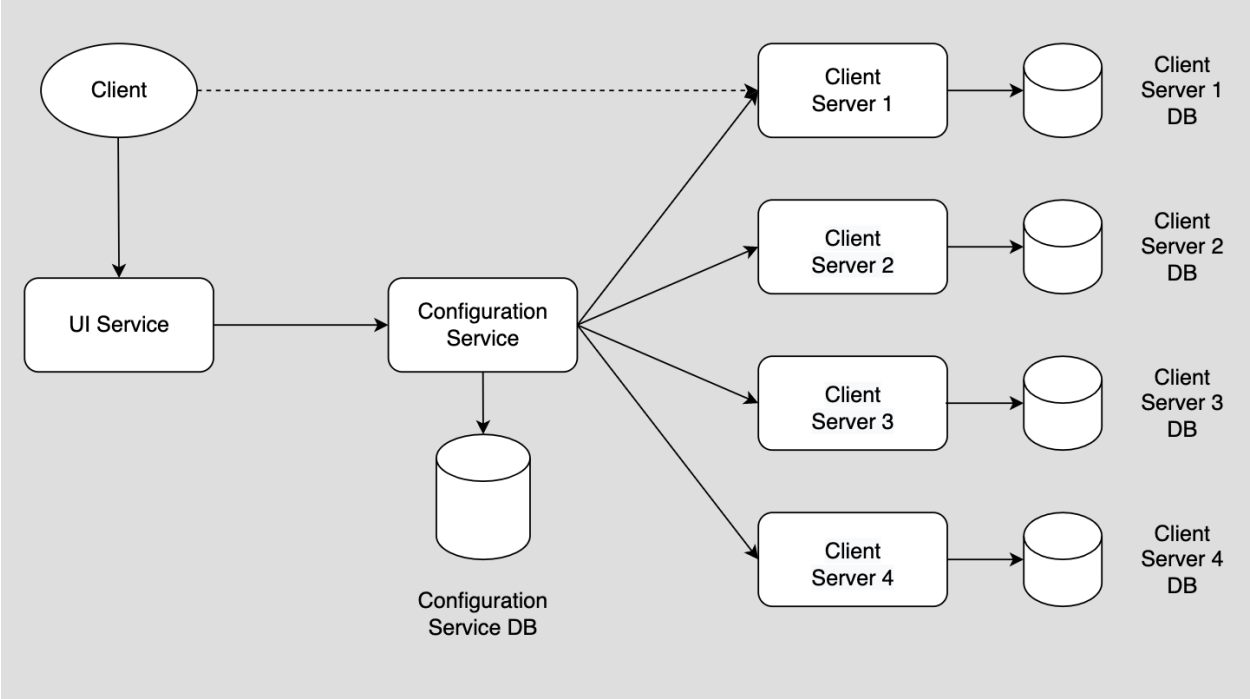

Рисунок 2.1 – Структура SaaS-застосунку в (без авторизаційної складової)

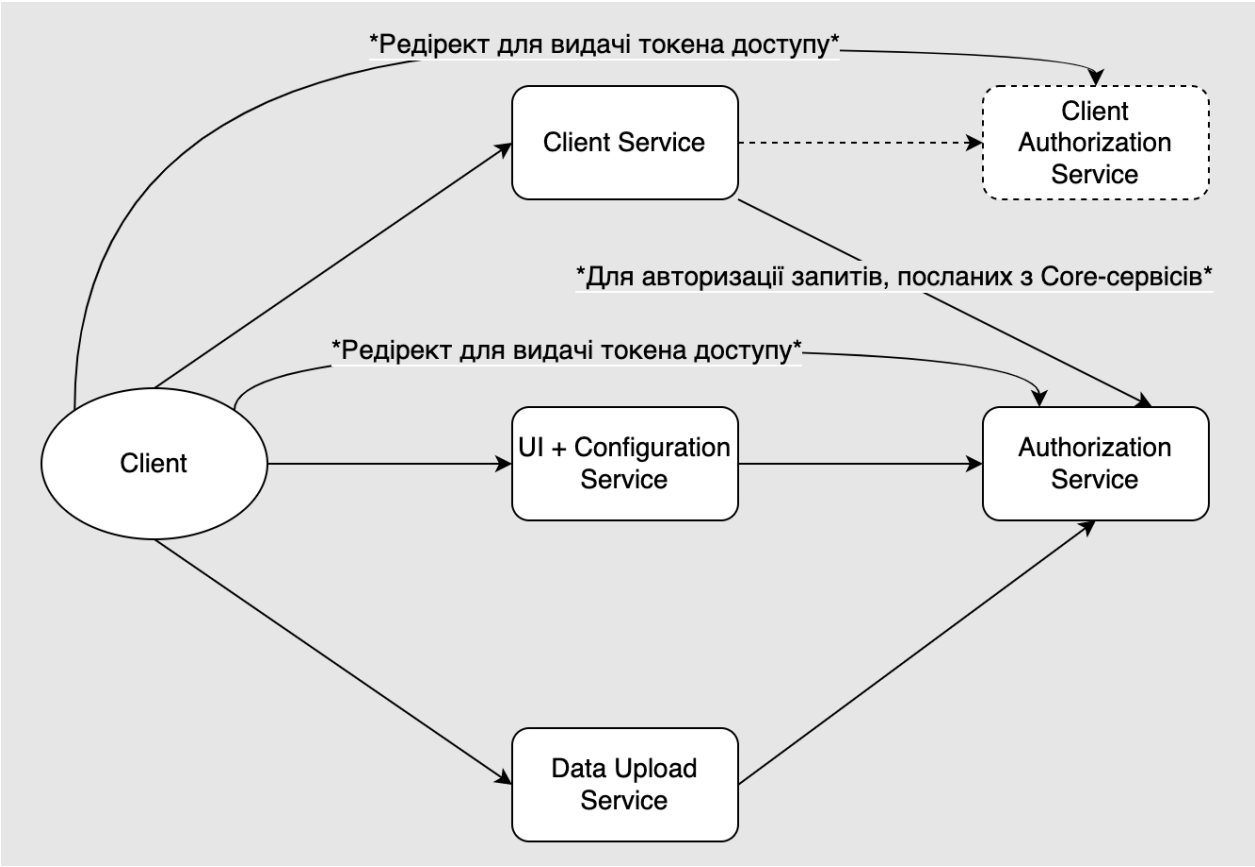

Рисунок 2.2 – Структура застосунку в області авторизації

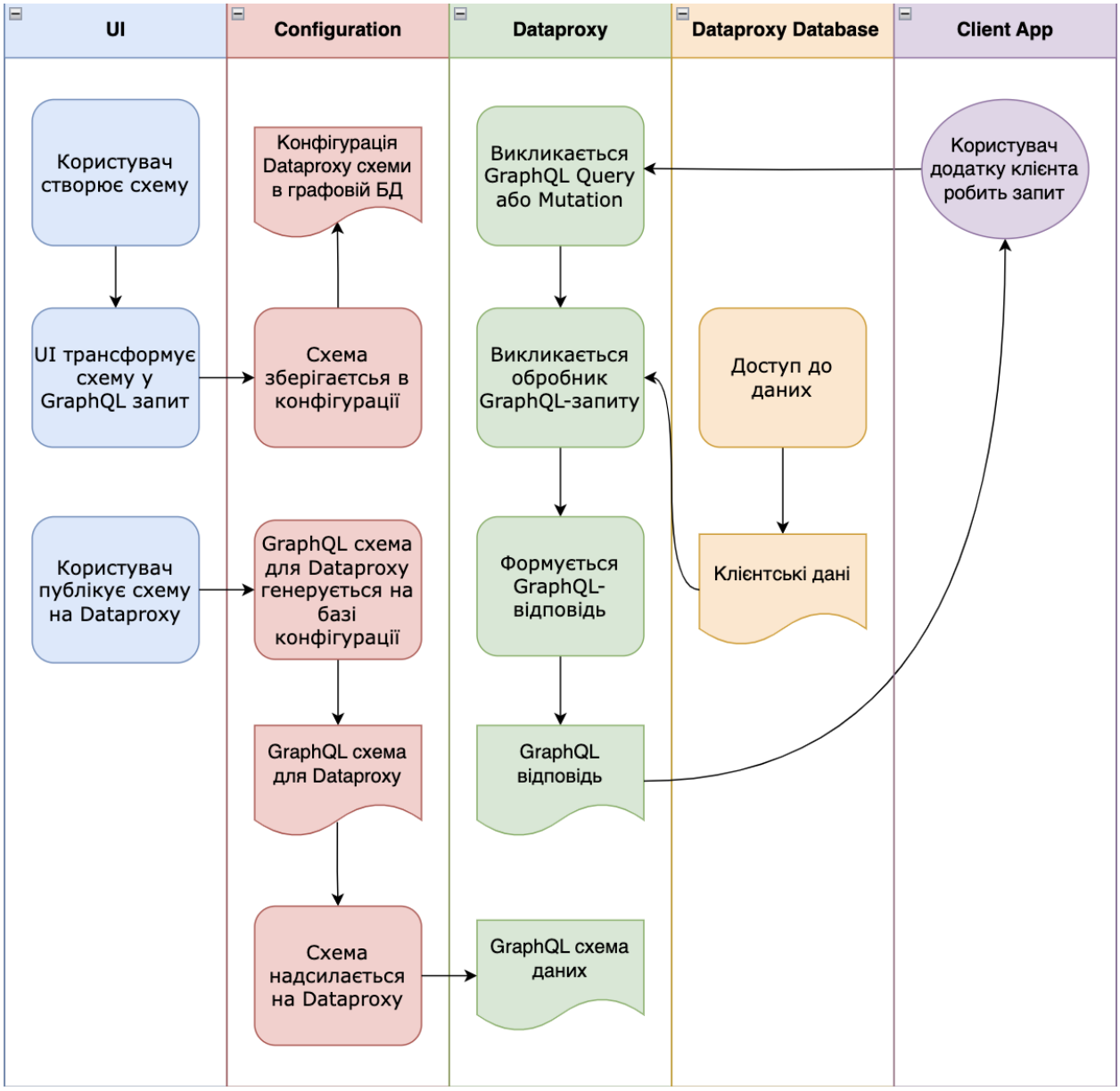

Рисунок 2.3 – Блок-схема, яка відображає взаємодію користувача із застосунком в області даних

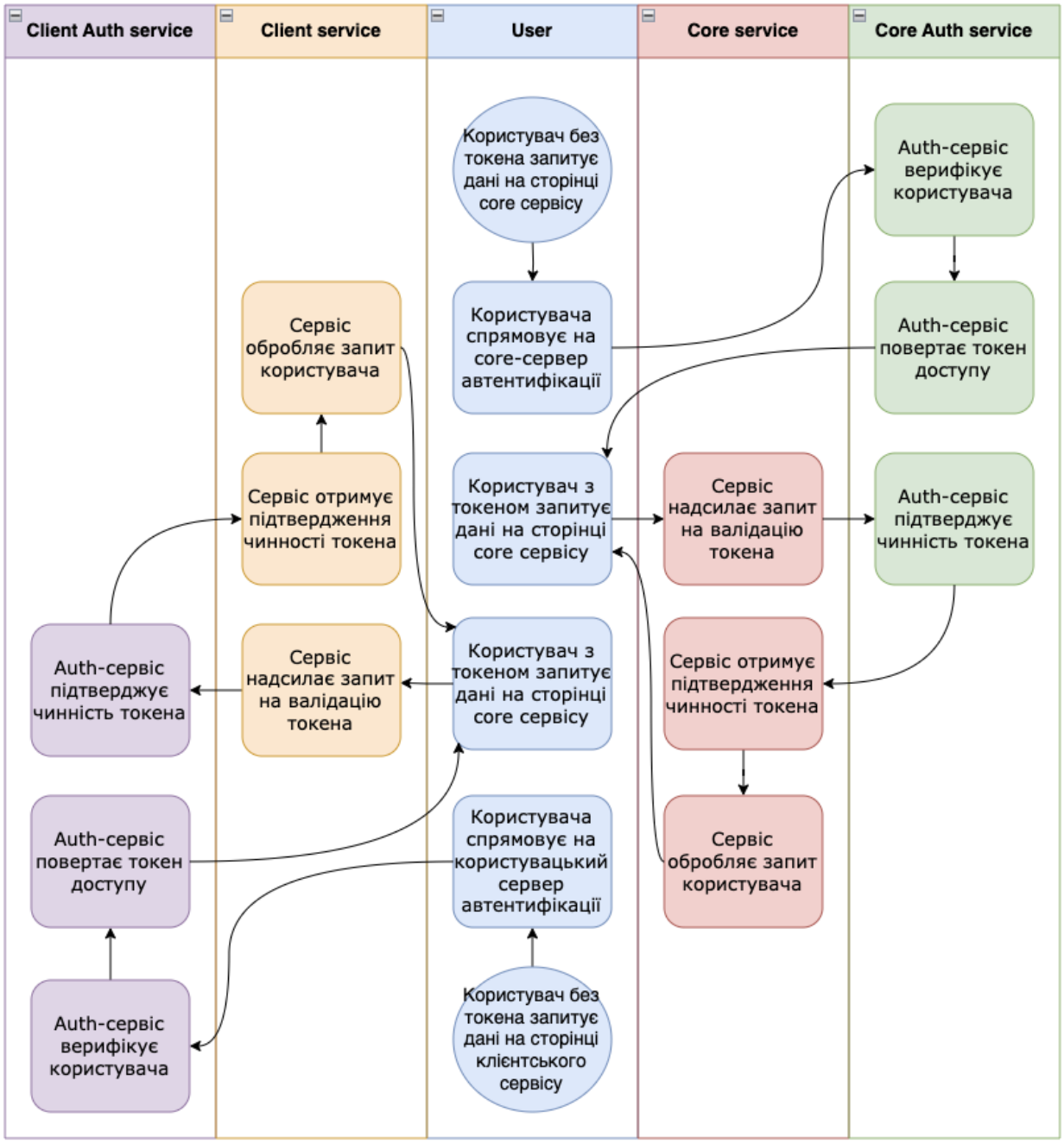

Рисунок 2.4 – Блок-схема, яка відображає взаємодію користувача із застосунком в області авторизації: інтеракція з користувацьким сервером авторизації мало в чому відрізняється від інтеракції з основним

2.3. Технологічний стек

2.3.1. Підхід до комунікації

Як було зазначено вище, за використання графової моделі, спілкування між клієнтською та серверною стороною зазвичай реалізується за допомогою GraphQL. GraphQL дозволяє змоделювати всю схему комунікацій як граф, а також, закономірно, є більш гнучкою ніж найближчі аналоги.

GraphQL запити поділяються на Query (не змінюють стан - аналог методу GET в HTTP) та Mutation (змінюють стан - аналог POST/PUT/PATCH методів).

2.3.2. База даних

Сучасні умови розробки пропонують широке різноманіття графових баз даних, як от TigerGraph, ArangoDB, Neo4j та Amazon Neptune [7]. Їх порівняння може бути темою для окремої обширної курсової, адже:

- **TigerGraph** має найвищу ефективність, проте підтримує дуже мало мов програмування;
- **ArangoDB** є найкращою в питаннях масштабування бд;
- **Neo4j** є найстарішою та найбільш розповсюдженою графовою базою, тож виграє, коли справа доходить до підтримки БД;
- **Amazon Neptune** працює на інфраструктурі Amazon Web Services, тому може вважатись найбільш довговічною.

Для конкретно цього прототипу я вибрав Neo4j, адже вона є найбільш доступною.

2.3.3. Мова програмування

Я прийняв рішення розробляти застосунок на TypeScript, оскільки це моя основна мова програмування. Крім того, розробка на TS має свої плюси, серед яких:

- Можливість використовувати одну мову на фронтенді та на бекенді

- Швидкість освоєння та відносна швидкість розробки
- Наявність великої кількості допоміжних ресурсів та бібліотек (наприклад, Apollo для роботи з GraphQL)

2.3.3.1. Frontend-технології

Клієнтська частина застосунку створена з використанням JavaScript-фреймворку React. React дозволяє створювати односторінкові застосунки (англ. *SPA, Single Page Application*), у яких весь необхідний код завантажується разом зі сторінкою, або динамічно довантажується за потребою, зазвичай у відповідь на дії користувача. Сторінка не оновлюється і не перенаправляє користувача до іншої сторінки у процесі роботи з нею. Для створення графів було використано бібліотеку D3, для GraphQL-комунікації з бекендом - ApolloClient.

Розділ 3. Демонстрація розробленого прототипу

Рисунок 3.1 демонструє початкову сторінку застосунку - сторінку входу. Ця сторінка надається auth-сервісом, а не безпосередньо UI-сервісом.

На рисунку 3.2 зображено контроль-панель користувача, на якій знаходяться його організації та Dataproxy-схеми. Для подальшої демонстрації я створив одну таку схему.

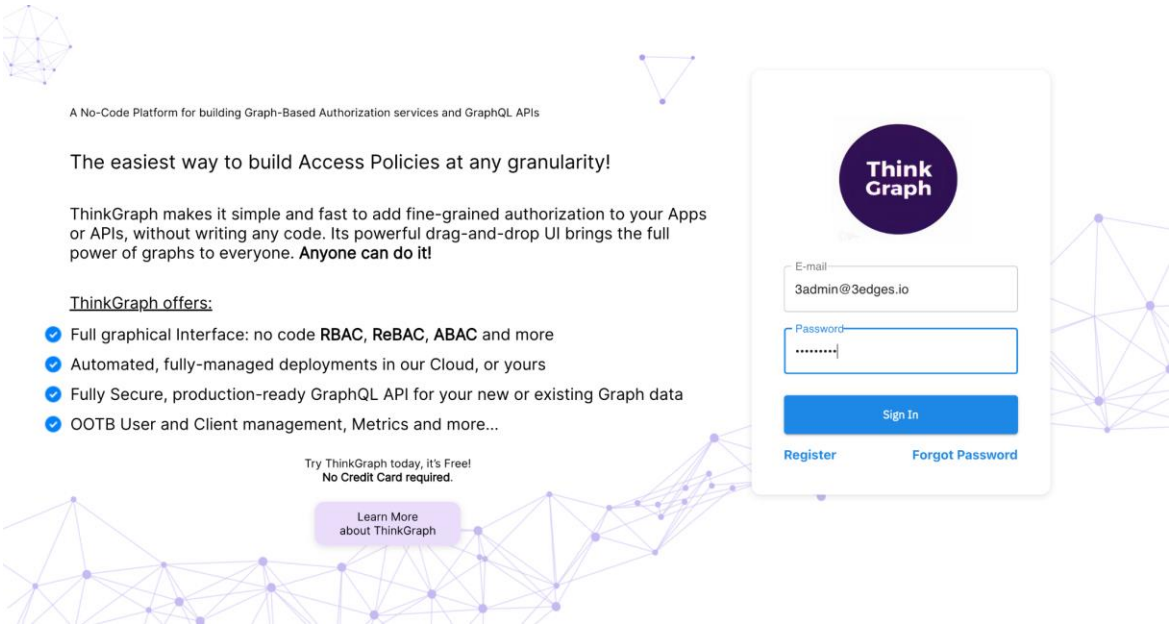

Рисунок 3.1 – Початкова сторінка веб-сайту (сторінка автентифікації)

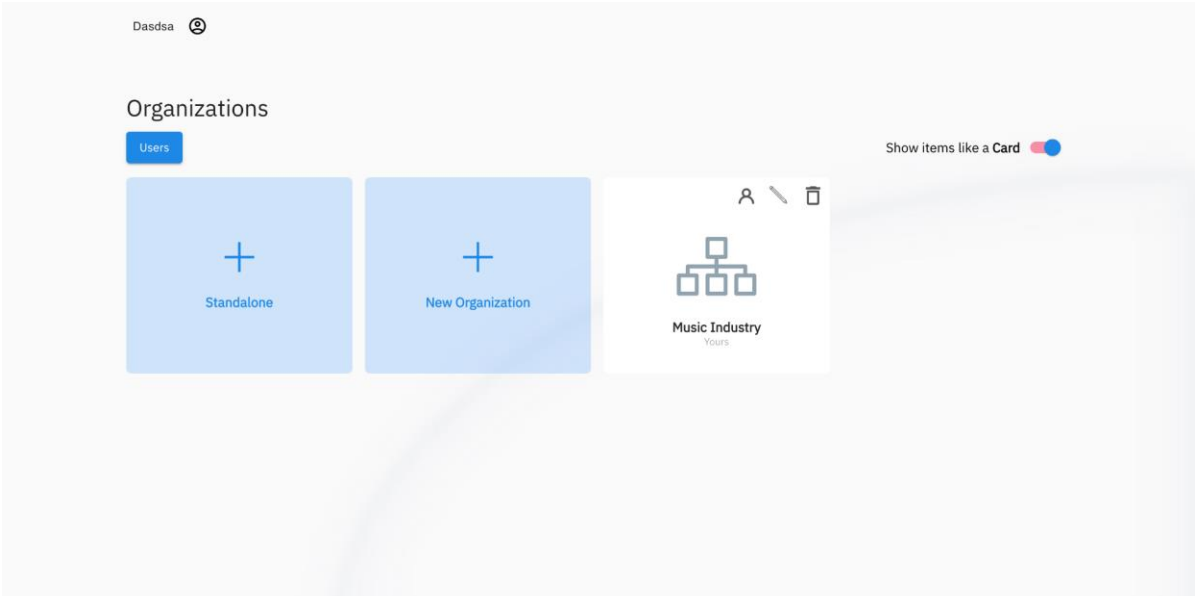

Рисунок 3.2 – Контроль-панель користувача

На рисунку 3.3 зображене діалогове вікно створення/зміни конфігурації користувацького Dataproxy-сервера. Через те, що прототип розгорнуто локально, його Connection URL знаходиться на localhost, проте цілком можливо розгорнути користувацький сервіс і на віддаленому комп’ютері.

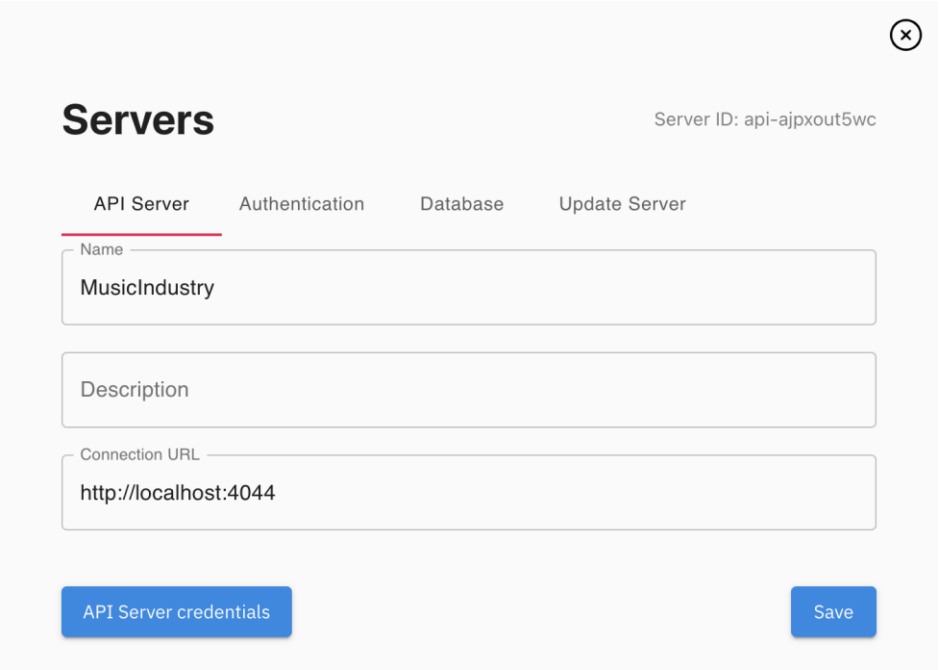

Рисунок 3.3 – Діалогове вікно конфігурації користувацького Dataproxy-сервера

Також у вікні конфігурації можна налаштувати базу даних.

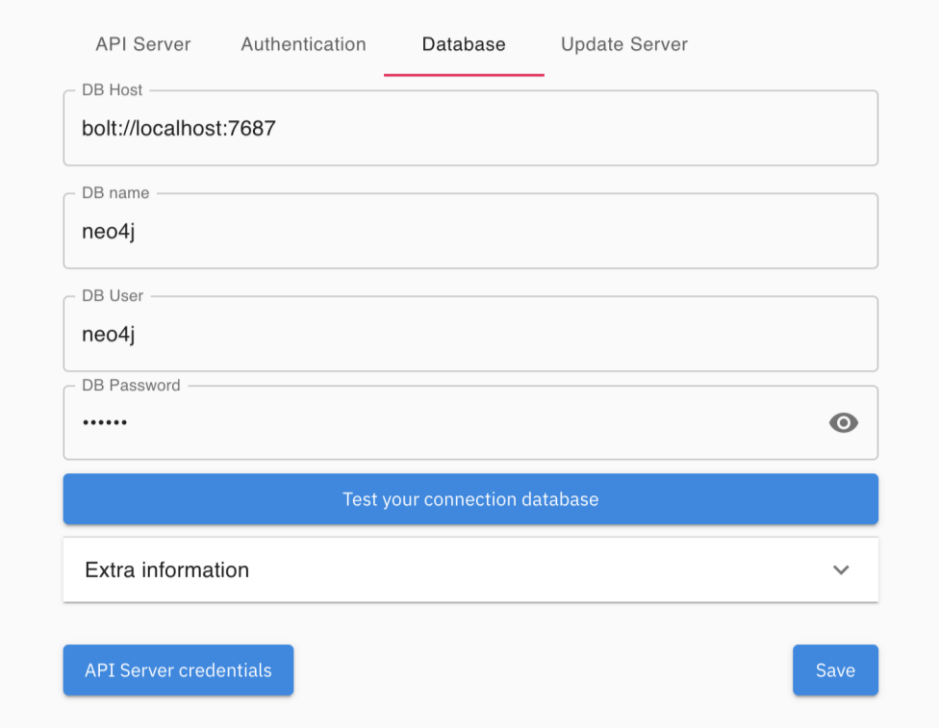

Рисунок 3.4 – Діалогове вікно конфігурації бази даних користувацького dataproxy-сервера

На рисунку 3.5 зображено інтерфейс сторінки конфігурації схеми API користувацького сервера. Зліва знаходиться меню, яке дозволяє додавати нові об’єкти, відношення та інші сутності до схеми. На рисунку 3.6 – меню конкретного об’єкта, в якому можна побачити та редагувати його властивості.

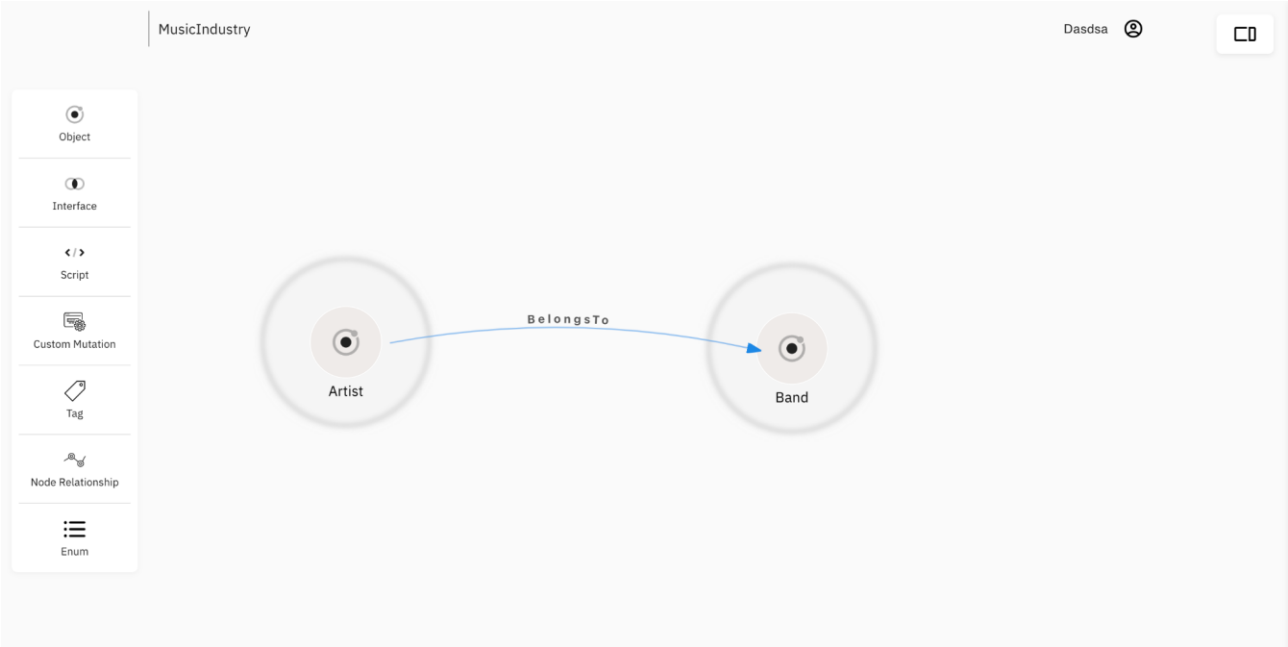

Рисунок 3.5 – Інтерфейс сторінки конфігурації схеми користувацького API

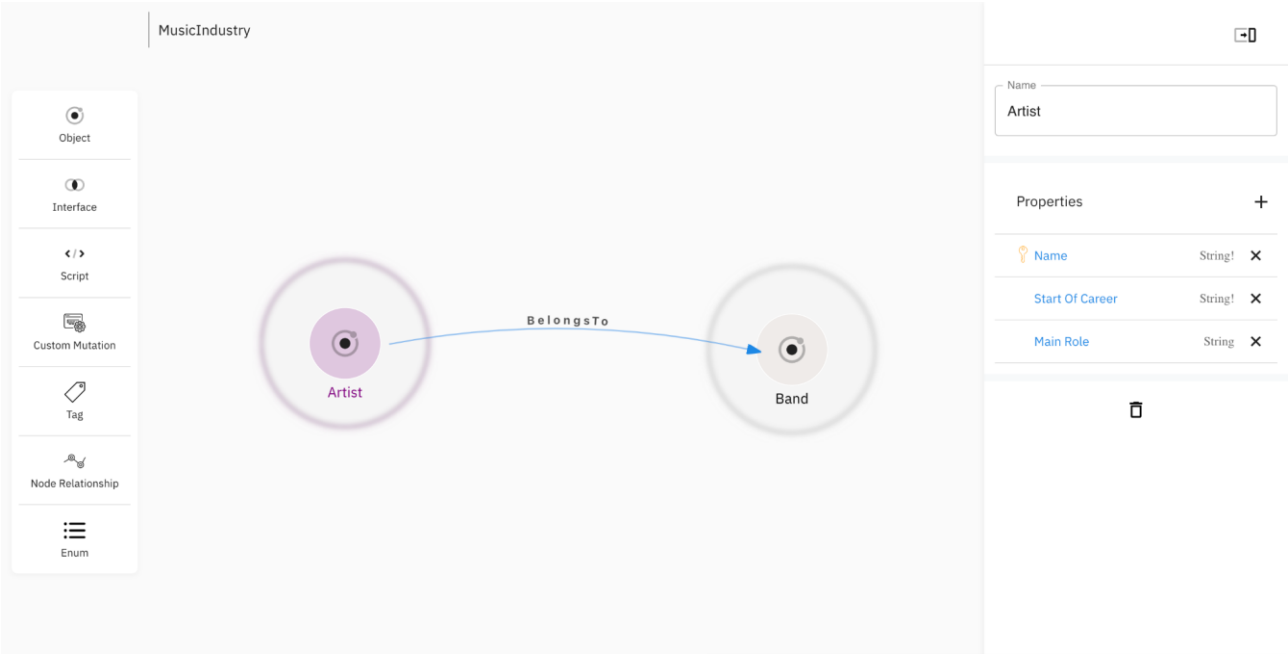

Рисунок 3.6 – Меню конкретного об’єкта зі схеми користувацького API

Для демонстрації я створив схему (рисунок 3.7) яка відображає відношення сутностей у галузі музичної індустрії: цей приклад було підібрано спеціально для порівняння графової моделі із реляційною (та ж схема даних, проте з використанням реляційної моделі зображена на рисунку 3.8).

Оцінивши обидві схеми, можна помітити, наскільки реляційна більш ускладнена: для кожного відношення типу “багато до багатьох” потрібно створювати проміжну таблицю, проте справжні труднощі з проектуванням починаються, коли у двох різних сутностей є спільне відношення до третьої (наприклад, альбом може випустити як незалежний артист, так і група). Тоді приходиться дублювати однакові по своїй суті таблиці (Artist_Album і Band_Album), або створювати додаткову узагальнювальну таблицю зі спільними атрибутами Artist і Band. Схожі приклади часто роблять реляційну схему не інтуїтивно-зрозумілою, а також можуть призвести до проблем із масштабуванням.

Очевидно, що в такому випадку в реляційній базі даних проблем зі складністю та часом виконання запитів буде набагато більше, ніж у графовій. На рисунках 3.9-3.10 зображені Cypher- та SQL-запити, які отримують інформацію про всі альбоми, до випуску яких причетний конкретний артист (в рамках вище запропонованої схеми).

Крім того, у графових системах зв’язки теж можуть зберігати інформацію: наприклад, **Artist {} - Belongs To { Join Date: “01.01.1970”} - Band {}**). Схожу концепцію в реляційній БД можна реалізувати через додавання додаткових колонок в проміжні таблиці, проте це далеко не так інтуїтивно.

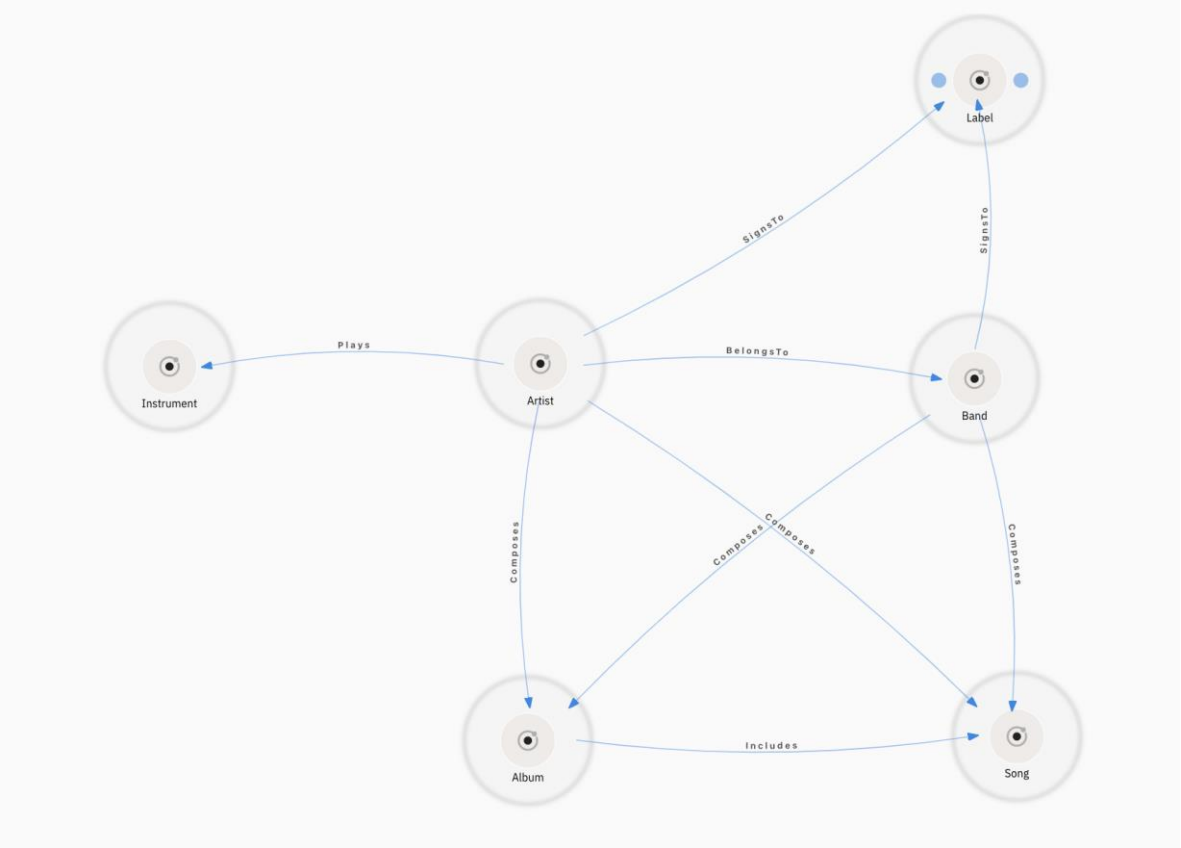

Рисунок 3.7 – Демонстраційна схема з використанням графової моделі

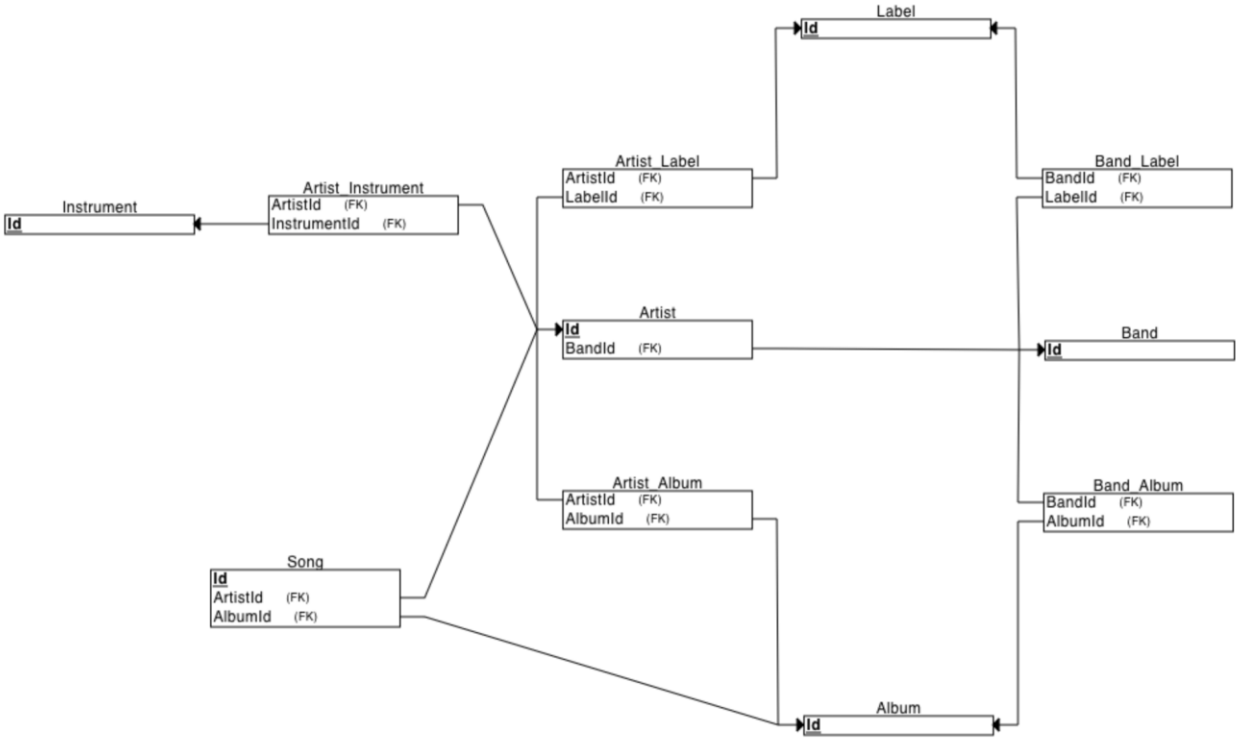

Рисунок 3.8 – Реляційна репрезентація вище представленої схеми

```
MATCH (artist:Artist {name: "David Gilmour"})
    -[:BelongsTo|Releases*1..2]→(album:Album)
RETURN artist, album;
```

Рисунок 3.9 – Приклад графового запиту на мові Cypher (використовується у Neo4j)

```
SELECT al.*
FROM artists a
INNER JOIN artists_albums aa ON a.id = aa.artist_id
INNER JOIN albums al ON aa.album_id = al.id
WHERE a.name = "David Gilmour"
UNION
SELECT al.*
FROM artists a
LEFT JOIN bands b ON b.id = a.band_id
LEFT JOIN bands_albums ba ON b.id = ba.band_id
INNER JOIN albums al ON aa.album_id = al.id
WHERE a.name = "David Gilmour";
```

Рисунок 3.10 – Запит з рисунку 8, переписаний на SQL

Процес оновлення схеми користувацького сервера зображено на рисунках 3.11 та 3.12.

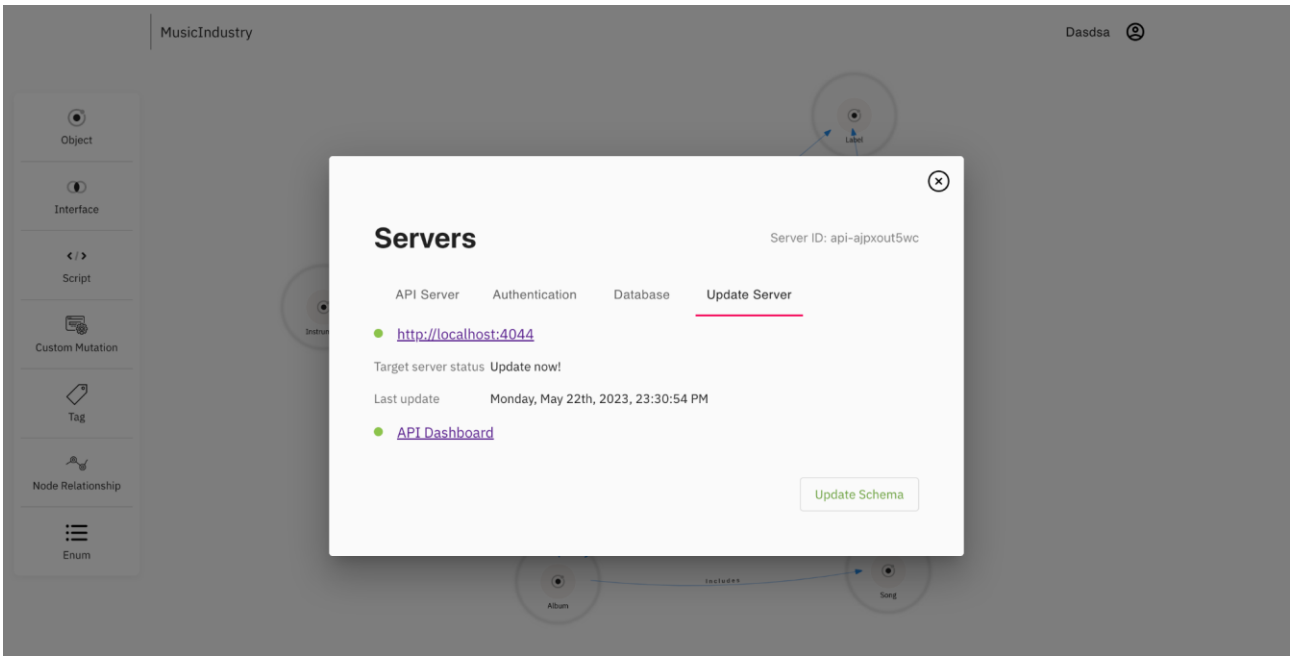

Рисунок 3.11 – Вікно оновлення користувацького сервера: до оновлення

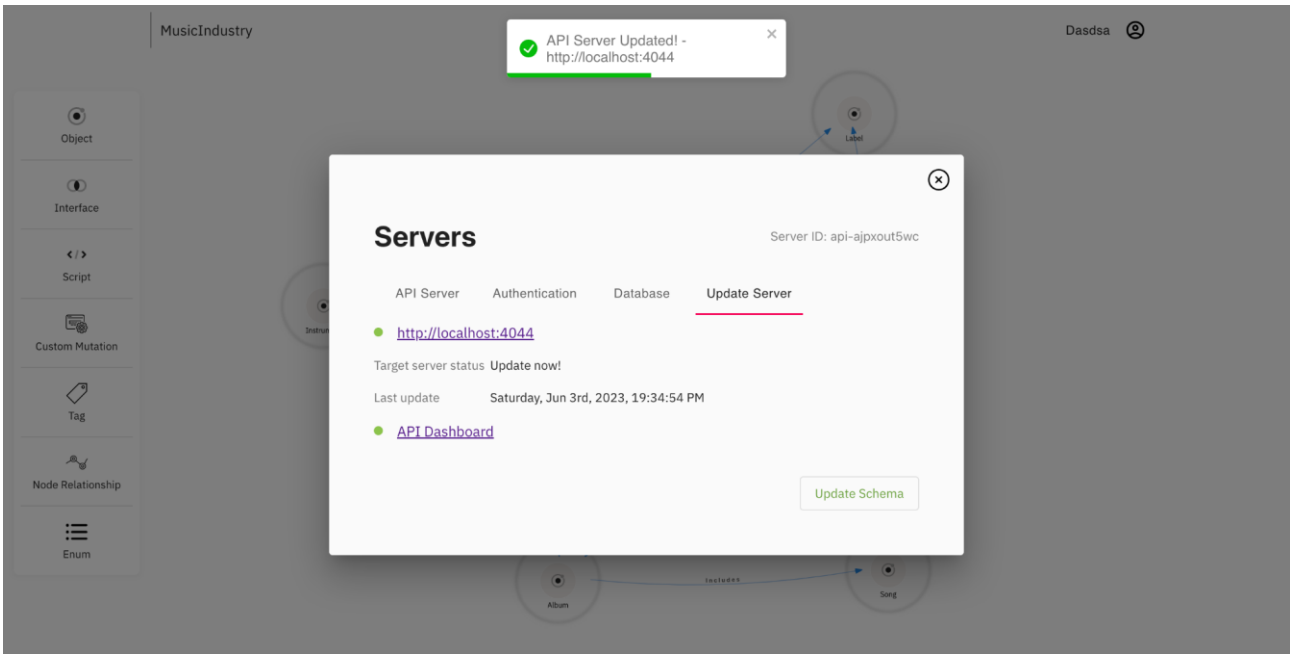

Рисунок 3.12 – Вікно оновлення користувацького сервера: після оновлення

Тепер детальніше розглянемо налаштування авторизації для Dataproxy-сервера. В меню Authentication можна помітити довгий список опцій, пов’язаних з протоколом OAuth 2.0. Як було зазначено вище, програма дає можливість користувачу самостійно вибирати сервер авторизації, тому у випадаючому списку є дві опції: ThinkGraph (користувацький auth-сервіс, розроблений як готовий приклад, що працює “з коробки”) та External (OAuth 2.0. auth-сервіс, наданий будь-яким іншим постачальником хмарних рішень, наприклад, Auth0).

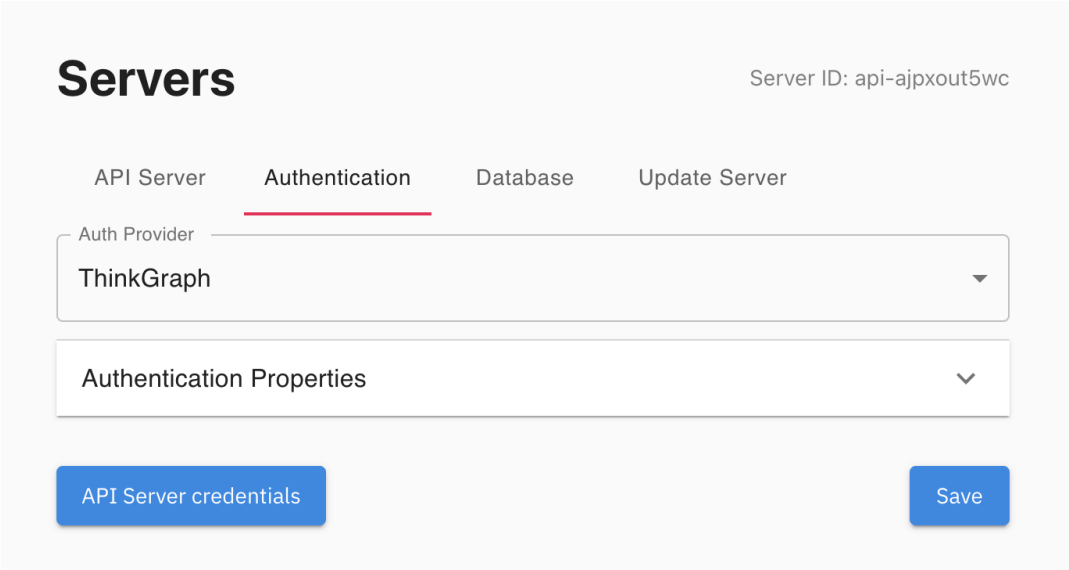

Рисунок 3.13 – Діалогове вікно конфігурації користувацького auth-сервіса

Розглянемо детальніше конфігурацію ThinkGraph auth-сервіса (рисунок 3.14):

- **OAuth service URL**: URL користувацького auth-сервіса.
- **Subject Type**: сутність на конфігураційній схемі, яка виступатиме суб’єктом авторизації.
- **Subject Naming Property**: атрибут суб’єктної сутності, за яким відбуватиметься авторизація (приклад: username, email, phone number).
- **User Claims**: Список атрибутів, які додатки третьої сторони можуть отримати як інформацію про користувача.
- **Authentication Query**: Cypher-запит в користувацьку базу даних, який буде використовуватись для знаходження підходящого

користувача при авторизації. Змінні з префіксом config взяті з цього меню.

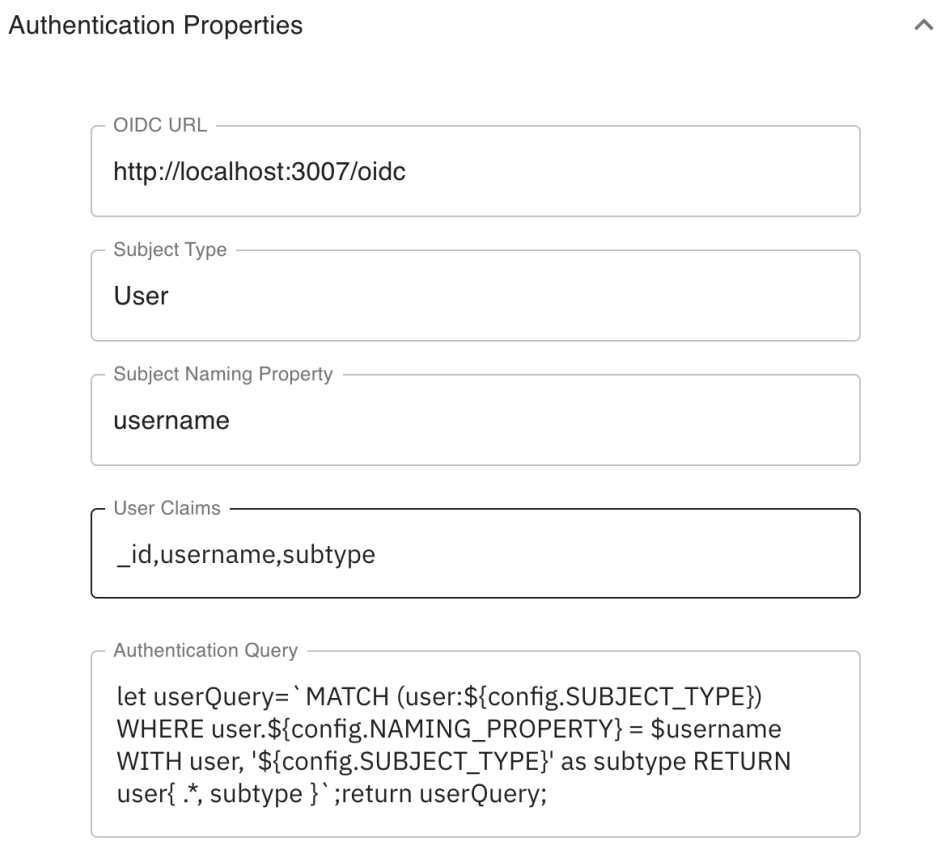

Рисунок 3.14 – Опції налаштування ThinkGraph auth-сервіса

Тепер розглянемо налаштування стороннього auth-сервіса (на прикладі Auth0):

- **Provider URL**: URL auth-сервіса. Надається провайдером.
- **Token Endpoint**: HTTP endpoint (кінцева точка), за якою провайдер надає токен доступу або refresh-токен.
- **Token Introspection Endpoint**: кінцева точка, запит на яку повертає властивості токена: статус його активності, сфери застосування, час, коли токен стане неактивним, id OAuth-клієнта, якому був виданий токен, тощо.
- **Authorize Endpoint**: кінцева точка, яка використовується для взаємодії з власником ресурсу та отримання коду авторизації, який

потім можна використати для доступу до Token Endpoint та Token Introspection Endpoint.

- **Logout Endpoint URL**: кінцева точка, яка дозволяє відкликати токен.
- **Access Token Type**: тип токена доступу, можливі опції - opaque та JWT. Opaque (непрозорий) токен - випадковий рядок або ідентифікатор, який сам по собі не має ніякого значення. Часто це ідентифікатор або посилання на токен, що зберігається на стороні сервера авторизації. JWT (JSON Web Token) - самодостатній токен, представлений у вигляді компактного рядка та містить дані у форматі JSON. Він складається з трьох частин: заголовка (header), корисного навантаження (payload) і підпису (signature). Корисне навантаження містить всю інформацію про токен, наприклад його видавця, термін дії та додаткові дані.
- **JWKS URI**: кінцева точка, яка повертає об’єкт JSON, який представляє набір JWK. Об’єкт JSON містить атрибут keys, який є масивом JWK. JWK (JSON Web Key) - це формат, який визначає стандартне представлення криптографічних ключів (приватних і публічних), які використовуються у JSON веб-токенах та інших криптографічних операціях у веб-додатках.
- **External Provider Name**: назва стороннього провайдера.
- **Token Claims Mapping**: задає перехід від назв атрибутів в специфікації стороннього провайдера до назв, розпізнаваних сервісами всередині системи
- **Client ID**: ідентифікатор користувача в сторонньому сервісі.
- **Client Password**: пароль користувача в сторонньому сервісі.

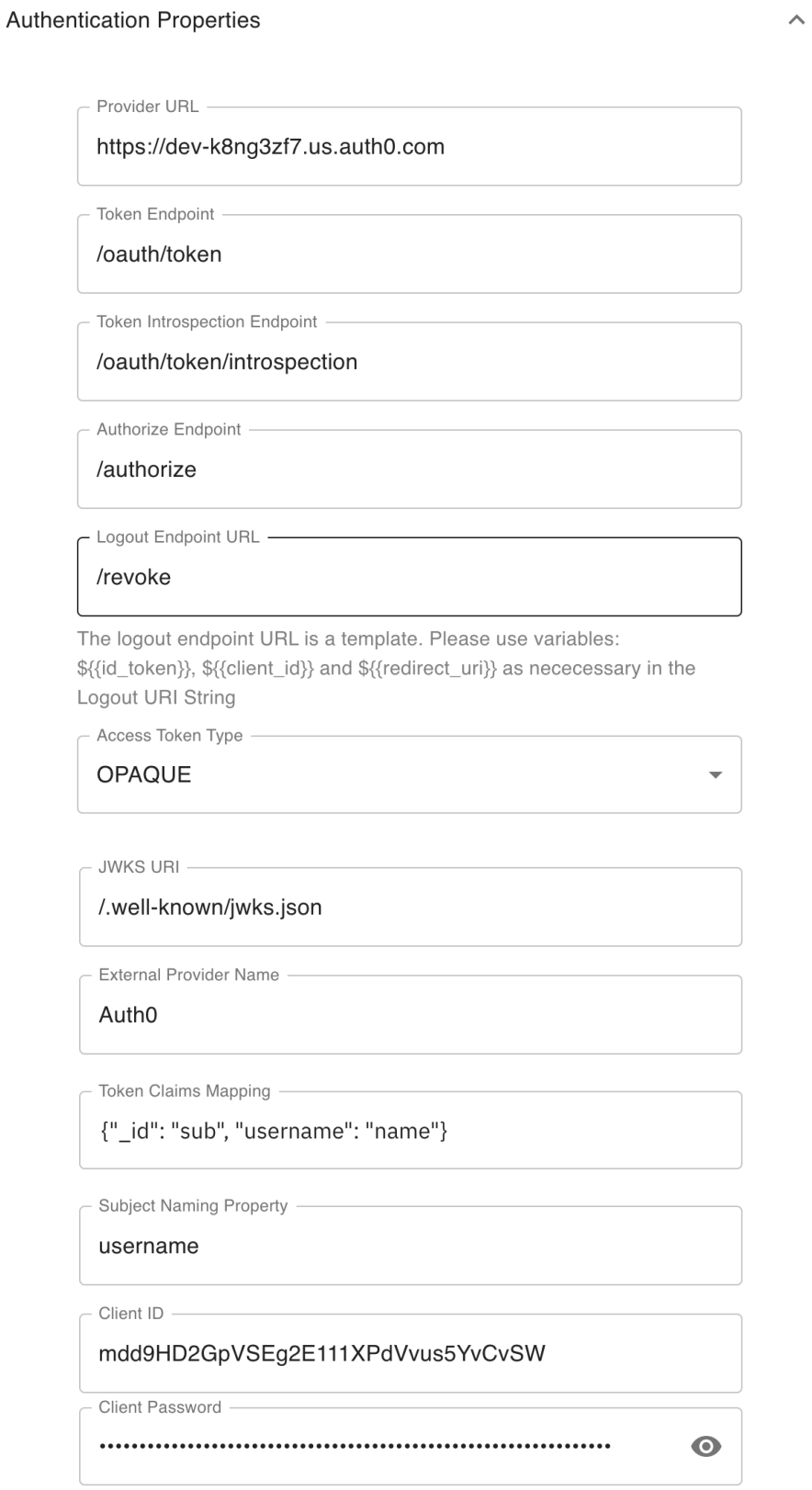

Рисунок 3.15 – Опції налаштування для стороннього auth-сервіса

Нижче зображено базову контроль-панель користувацького сервера: до та після оновлення. На рисунку 3.16 можемо бачити специфікацію користувацького GraphQL API, яка була створена базуючись на схемі, заданій на Configuration-сервісі.

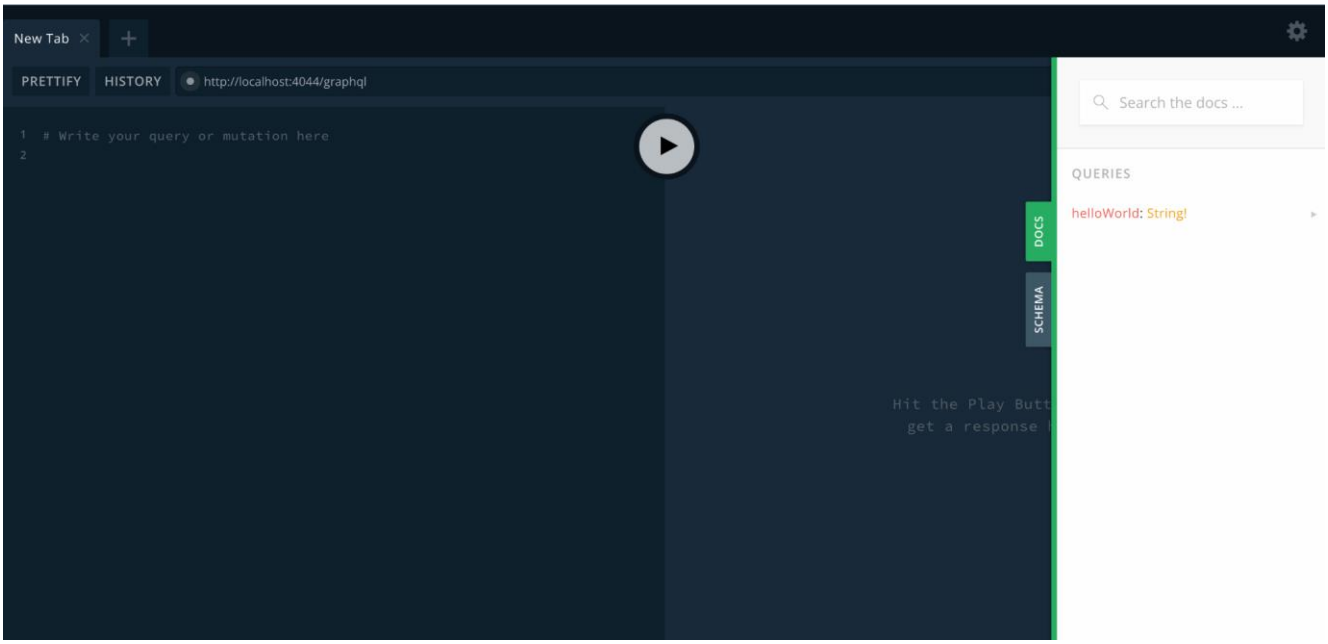

Рисунок 3.16 – Контроль-панель користувацького сервера, до оновлення схеми

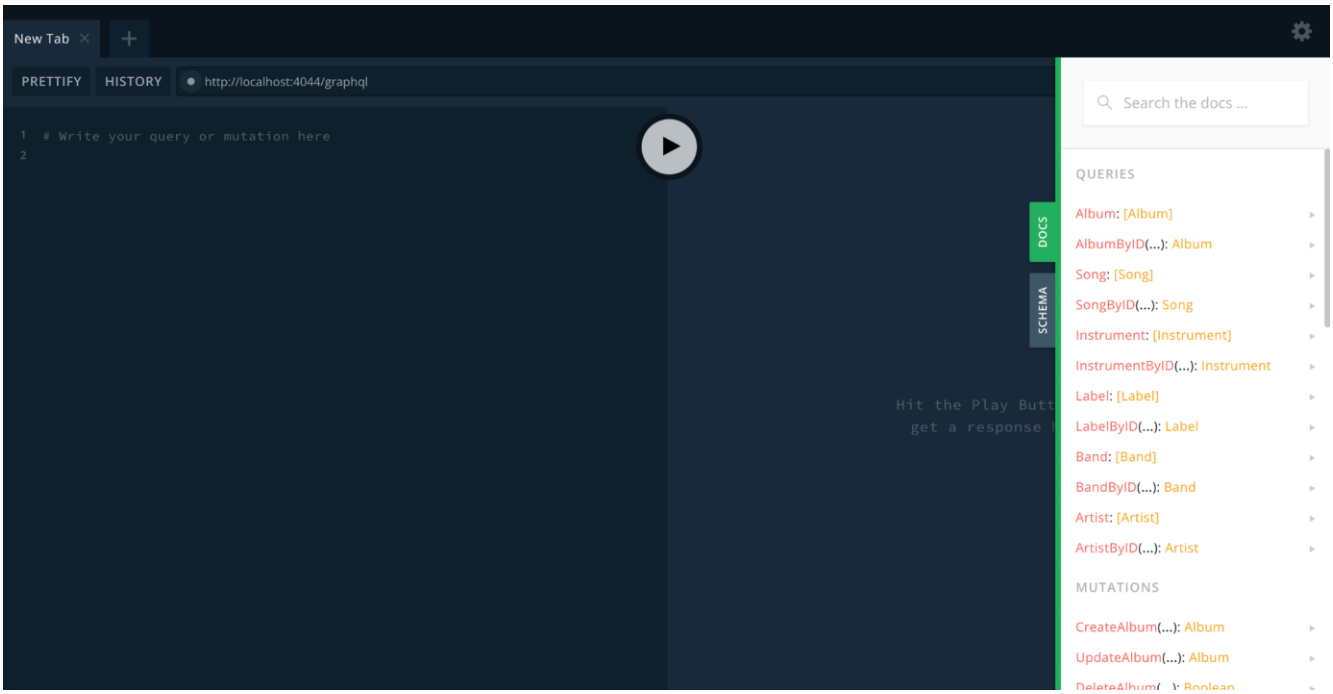

Рисунок 3.17 – Контроль-панель користувацького сервера, після оновлення (схема зліва)

Для того, щоб отримати доступ до даних з користувацького сервера, необхідно авторизуватись через auth-сервіс. Запит на отримання доступу з використанням найпримітивнішого grant-типу та відповідь на нього зображено на рисунку 3.18.

Рисунок 3.18 – Запит на користувацький auth-сервіс, виконаний за допомогою curl

В якості базової авторизації на **Token Endpoint** були використані base64-закодовані **Client ID** та **Client Password**. Також варто зазначити, що grant_type=password вважається застарілим та небезпечним у стандарті OAuth 2.0., проте авторизація з використанням іншого grant-типу потребувала б створення клієнтської веб-частини, тому в якості прикладу було використано саме його.

Далі помістимо отриманий токен доступу у значення HTTP-гедера “Authorization” (рисунок 3.19):

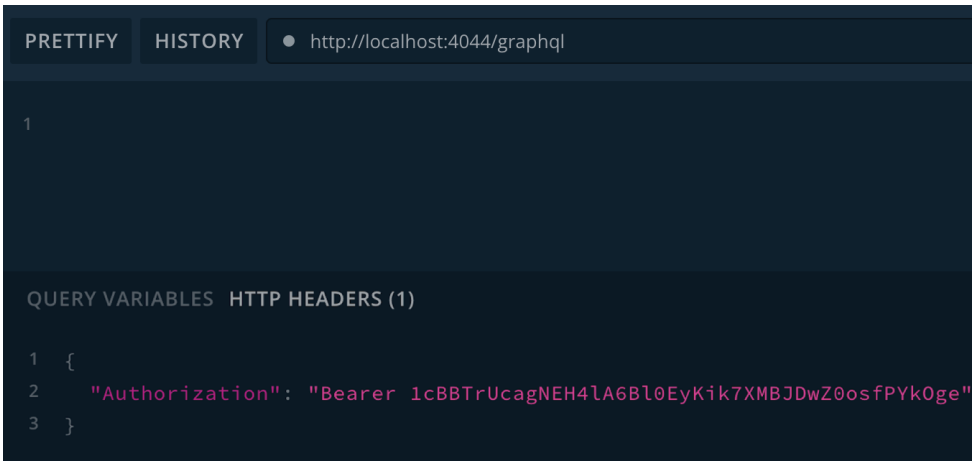

Рисунок 3.19 – Отриманий токен доступу як Authorization-гедер, заданий через користувацьку контроль-панель

На рисунках 3.20 - 3.21 – взаємодія з користувацьким API. Гнучкість GraphQL дозволяє нам створювати декілька різних сутностей за раз (рис. 15), та кастомізувати запити на отримання даних за бажанням (конкретно на рис. 16 - отримуємо список артистів, групи, до яких вони належать та всю супровідну інформацію).

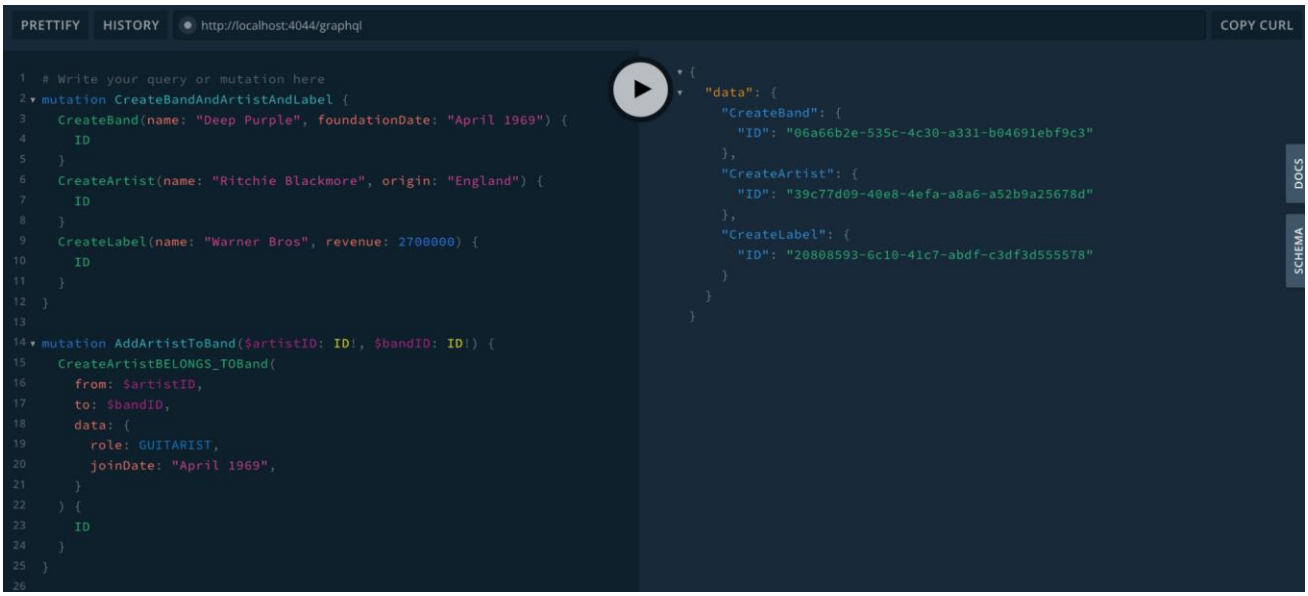

Рисунок 3.20 – Взаємодія з користувацьким сервером - додавання даних

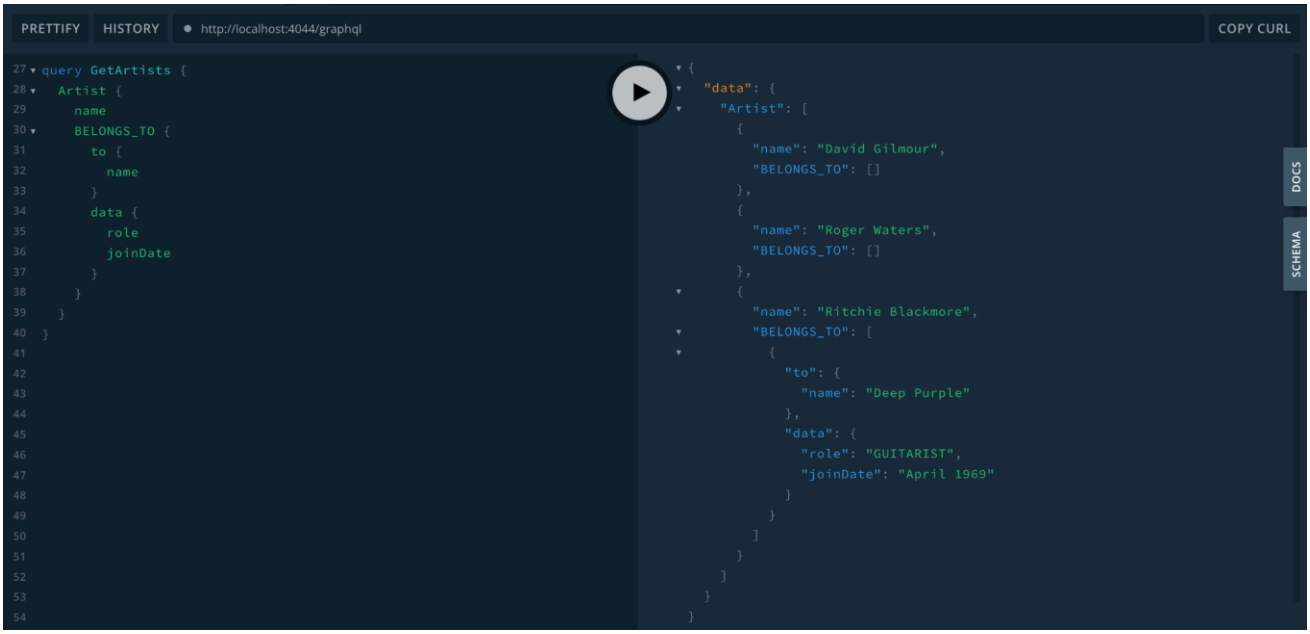

Рисунок 3.21 – Взаємодія з користувацьким сервером - отримання даних

Висновки

Результатом виконання курсової роботи став веб-застосунок, функціонал якого дозволяє розмножувати API-сервери з власноруч заданою конфігурацією. Створені API-сервери проектуються за допомогою та використовують графову модель даних. Застосунок наділений зручним користувацьким інтерфейсом. Досягнено високого рівня гнучкості та безпеки на різних рівнях взаємодії із застосунком.

Програма виконує всі постаставлені перед нею задачі та є піддатливою до розширення та редагування.

Список використаних джерел

1. *Lyman P., Varian H.R.* How much information. - Release of the University of California. Oct.27, 2003.
2. *Peter Mell, Timothy Grance.* The NIST Definition of Cloud Computing. - National Institute of Standards and Technology Special Publication 800-145. September 2011.
3. *Martin Reddy*. API Design for C++. - Elsevier. Mar.14, 2011.
4. *Phil Sturgeon*. Build APIs You Won't Hate. 12 August 2015.
5. *Phil Sturgeon.* GraphQL vs REST: Overview. 24 January 2017.
6. *Aaron Parecki*. The Little Book of OAuth 2.0 RFCs. 1 February 2020.
7. *Sam Bell.* Comparing Graph Databases. 13 September 2019.