

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«Модуль робити GraphQL для .NET»

Виконав студент групи ІПЗ-21-1
БЕКЕР Михайло Михайлович

Керівник роботи:
САВІЦЬКИЙ Роман Святославович

Рецензент:
ЄФРЕМОВ Юрій Миколайович

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ КАФЕДРА
ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. Кафедри інженерії програ-
много забезпечення

Тетяна Вакалюк

«26» лютого 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **БЕККЕР Михайло Михайлович**

Керівник роботи: **САВІЦЬКИЙ Роман Святославович**

Тема роботи: **«Модуль роботи GraphQL для .NET»,**

Затверджена Наказом закладом вищої освіти «8» серпня 2025р., №61/с

Вихідні дані роботи: методи та засоби, необхідні для автоматизації
роботи з GraphQL для .NET розробників.

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх сто-
суються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Савіцький Р.С.	02.08.2025р.	02.08.2025р.
2	Савіцький Р.С.	02.08.2025р	02.08.2025р
3	Савіцький Р.С.	02.08.2025р	02.08.2025р.

РЕФЕРАТ

Випускна робота бакалавра складається з модулю роботи Graphql для .NET та пояснювальної записки. Текстова частина викладена на 86 сторінках друкованого тексту. Пояснювальна записка має 7 сторінок додатків. Список використаних джерел містить 30 пунктів та займає 3 сторінки. В роботі наведено 99 ілюстрацій та 23 таблиці. Загальний обсяг роботи 91 сторінка.

Метою роботи є розробка модулю Graphql під .NET для автоматизації роботи розробників.

В розділах пояснювальної записки було розглянуто постановку задачі та сформульовано цілі дослідження, що визначають напрями подальшої розробки. Проведено аналіз існуючих аналогів програмних продуктів, який дозволив виявити їхні переваги та недоліки й сформулювати вимоги до власної системи. Окрему увагу приділено обґрунтуванню вибору інструментальних засобів реалізації.

Також було описано вимоги до програмного забезпечення та побудована архітектура додатку. Наведено моделювання взаємодії між компонентами, що дозволяє відобразити логіку роботи та забезпечити узгодженість між окремими модулями. Розглянуто етапи проєктування та реалізації алгоритмів, необхідних для виконання основних функцій програми.

В останньому розділі було описано інтерфейс розробленого додатку та порядок його використання. Описано процедуру інсталяції та налаштування системи, а також наведено приклади взаємодії користувача з консольним інтерфейсом. Розглянуто основні команди, які реалізують функціональні можливості програмного забезпечення, та описано логіку їхнього виконання.

КЛЮЧОВІ СЛОВА: .NET, GRAPHQL, АВТОМАТИЗАЦІЯ, CLI, ТОКЕНІЗАЦІЯ, БАЗА ДАНИХ, ORM.

					ІПЗ.КР.Б – 121 – 25 – ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	Модуль роботи Graphql для .NET Пояснювальна записка	Літ.	Арк.	Акрушів
Розроб.		Беккер М.М.					4	92
Керівник.		Савіцький Р.С.				ФІКТ, гр. ІПЗ-21-1		
Реценз.								
Н. Контр.		О. Тичина						
Зав. каф.		Т. Вакалюк						

ABSTRACT

The bachelor's thesis consists of a Graphql for .NET work module and an explanatory note. The text part is presented on 86 pages of printed text. The explanatory note has 7 pages of appendices. The list of references includes 30 items and occupies 3 pages. The paper contains 99 illustrations and 23 tables. The total volume of the work is 91 pages.

The aim of the work is to develop a Graphql module for .NET to automate the work of developers.

The sections of the explanatory note considered the formulation of the problem and formulated the objectives of the study, which determine the directions for further development. An analysis of existing analogues of software products was carried out, which made it possible to identify their advantages and disadvantages and to formulate requirements for our own system. Particular attention was paid to justifying the choice of implementation tools.

The requirements for the software were also described and the architecture of the application was built. Modelling of the interaction between components was provided, which allows the logic of operation to be reflected and ensures consistency between individual modules. The stages of designing and implementing the algorithms necessary to perform the main functions of the programme were considered.

The last section describes the interface of the developed application and how to use it. The procedure for installing and configuring the system is described, and examples of user interaction with the console interface are given. The main commands that implement the functional capabilities of the software are considered, and the logic of their execution is described.

KEYWORDS: .NET, GRAPHQL, AUTOMATION, CLI, TOKENIZATION, DATABASE, ORM.

		Беккер М.М.			ІІЗ.КР.Б - 121 - 25 - ІІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		5

Зміст

РЕФЕРАТ	1
ABSTRACT	5
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	7
ВСТУП.....	8
РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМИ ТА ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ	11
1.1 Постановка задачі	11
1.2 Аналіз аналогів програмного продукту.....	15
1.3 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення	28
Висновки до першого розділу	30
РОЗДІЛ 2. ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	31
2.1 Опис вимог	31
2.2 Моделювання взаємодії компонентів системи.....	33
2.3 Проєктування та реалізація алгоритмів роботи програми	39
2.4 Реалізація консольного додатку	42
Висновки до другого розділу.....	57
РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З КОНСОЛЬНИМ ДОДАТКОМ	59
3.1 Порядок встановлення та налаштування системи	59
3.2 Структура інтерфейсу та порядок роботи з консольним додатком	60
3.3 Тестування роботи системи.....	76
Висновки до третього розділу	81
ВИСНОВКИ.....	82
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	83

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – прикладний програмний інтерфейс

AST – Abstract Syntax Tree

CLI – command-line interface

MVC – Model View Controller

ORM – Object Relational Mapper

REST – Representational State Transfer

ПЗ – програмне забезпечення

SQL – Structured Query Language

SDL – Schema Definition Language

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		7

ВСТУП

Розробка програмного забезпечення поєднує в собі складні процеси, що потребують пильної уваги, відповідальності та узгодження між всіма залученими сторонами. Проте, водночас, також існують й відносно прості, рутинні процеси, або задачі, які, зазвичай, автоматизуються за допомогою спеціалізованих інструментів. Одним із найпоширеніших прикладів є генерація однотипного коду, що особливо актуальна у сфері веброзробки, де значна частина архітектурних рішень і підходів має типовий характер, що, в свою чергу, зумовлює потребу у створенні засобів, здатних спростити та прискорити процес розробки. Одним з типових підходів для організації взаємодії клієнтських і серверних застосунків, що дозволяє ефективно маніпулювати ними є мова запитів GraphQL. У екосистемі мови програмування JavaScript вже сформувано широкий набір інструментів, що дозволяють автоматизувати процес побудови вебдодатків, використовуючи GraphQL. Водночас, екосистема .NET, навіть при наявності таких багатофункціональних бібліотек для роботи з GraphQL як «GraphQL .NET» та «Hotchocolate», не має ніяких автоматизаційних інструментів, що могли б допомогти пришвидшити розробку. Саме тому тема розробки атоматизаційних інструментів для роботи з GraphQL в .NET є актуальною та має значну практичну цінність.

Відповідно мета дослідження, полягає в розробці інструменту автоматизації рутинних задач розробників, що працюють з бібліотеками «GraphQL .NET» та «Hotchocolate» за допомогою платформи .NET.

Для досягнення поставленої мети, необхідно виконати низку послідовних завдань, що полягають у:

- аналізу наявних інструментів, що, зазвичай, використовують генерацію коду для автоматизації роботи розробників, який дозволить виявити типові підходи, функціональні можливості та недоліки існуючих рішень;
- проектуванні архітектури власної системи, яка передбачає можливість розширення за рахунок впровадження системи плагінів, що забезпечує

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		8

гнучкість, масштабованість та простоту інтеграції нового функціоналу у майбутньому;

- реалізації функціоналу порівняння вже існуючого шаблонного коду та генерації на основі нього нового. Такий підхід дозволить мати змогу автоматизувати роботу з кодом не тільки в рамках існуючих бібліотек або існуючих підходів, а й в рамках власних імплементацій та підходів, що можуть бути специфічними для кожного проєкту;
- виконанні тестування створеного функціоналу, що надасть змогу переконатися в стабільності його роботи, відповідності поставленим вимогам, а також його готовності до практичного застосування.

Об'єктом дослідження є процес автоматизації рутинних задач розробники, за допомогою генерації коду у рамках розробки GraphQL додатків.

Предметом дослідження є методи, інструменти та підходи до створення систем генерації коду з підтримкою GraphQL у середовищі .NET.

У процесі виконання дослідження було застосовано комплекс методів та підходів, що сприяли ефективному досягненню поставленої мети та забезпечили високу якість реалізації розробленого програмного забезпечення.

На початковому етапі були використані методи аналізу та порівняння існуючих аналогів систем генерації коду, що дозволило здійснити огляд поточних рішень, визначити їх функціональні можливості, обмеження та недоліки, що дало змогу чітко сформулювати вимоги до власної системи та уникнути типових проблем, притаманних існуючим підходам.

Процес реалізації програмного забезпечення базувався на принципах об'єктно-орієнтованого програмування, що забезпечило логічну структуру коду, високий рівень модульності, повторного використання компонентів, а також зручність масштабування та супроводу системи.

Важливим архітектурним рішенням стала реалізація плагінної моделі, яка дозволила забезпечити гнучкість та розширюваність функціоналу без необхідності внесення змін до основного ядра системи. Такий підхід значно підвищив

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		9

адаптивність програмного продукту до змін вимог і спростив процес інтеграції нових функцій.

З метою перевірки коректності реалізованого функціоналу та забезпечення стабільності роботи системи використовувалося юніт-тестування на основі фреймворку xUnit. Це дозволило на ранніх етапах виявляти помилки, контролювати поведінку компонентів і гарантувати відповідність очікуваному результату.

Окрему роль у технічній реалізації відіграло використання інструментарію Roslyn API, який надає доступ до синтаксичних і семантичних структур C# коду на рівні компілятора. Завдяки цьому стало можливим виконання детального аналізу та трансформації коду, що є необхідним для реалізації механізмів порівняння й часткової генерації.

Практична значущість виконаної роботи полягає у створенні реального інструменту, який може бути інтегрований у процес розробки GraphQL API в програмних проектах на платформі .NET. Теоретична значущість полягає в узагальненні сучасних підходів до генерації коду, дослідженні особливостей використання Roslyn для аналізу синтаксичних дерев, а також у формалізації принципів побудови плагінних архітектур.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		10

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМИ ТА ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ

1.1 Постановка задачі

До типових задач кожного вебдодатку, незалежно від підходу для обміну даними, відносяться: створення, оновлення, отримання, видалення, фільтрація, розділення по сторінкам та сортування даних, а також визначення типових компонентів (класів, атрибутів, конфігурацій). Такі типові задачі, в кожному проєкті, можуть реалізовуватись різними способами, проте всі вони, зазвичай, реалізуються всього один раз, й в подальшому, перевикористовуються під час розробки, шляхом копіювання попередньо імплементованої логіки з невеликими змінами. Проблема полягає в тому, що розробники витрачають на такі рутинні задачі той час, який вони могли б приділити іншим, більш важливим речам. Більше того, нові розробники, можуть помилково реалізувати вже готовий функціонал, тільки іншим способом, не знаючи, що існує стандартизований підхід для вирішення задачі, на якою вони працюють.

Окрім традиційних REST-орієнтованих підходів щодо реалізації вищеписаних програмних типових компонентів, дедалі ширшого поширення набуває мова запитів GraphQL.

Graphql – це мова запитів, основна перевага якої, полягає у гнучкому обміні даними між клієнтом і сервером за рахунок можливості вказувати та отримувати тільки ті дані, які дійсно потрібні [1]. Такий підхід надає змогу гнучко керувати потоком даних у сучасних додатках, водночас вирішуючи проблему надмірного або недостатнього отримання даних.

У межах екосистеми .NET підтримку GraphQL забезпечують, зокрема, бібліотеки HotChocolate та GraphQL.NET, що також містять в собі проблеми вищезазначені загальні [2]. Більше того, дані бібліотеки мають також і свої власні, унікальні проблеми, на які також варто звернути увагу.

Розглянемо детальніше «Graphql .NET», де однією із основних проблем, є дублювання типів даних. Мова запитів Graphql має свою систему типів, тому класи, визначені за допомогою C#, повинні бути конвертовані у graphql-сумісні типи.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		11

Система типів – це логічна система, що складається з набору правил, які присвоюють кожному терміну властивість, яка називається типом. Система типів graphql складається з наступних типів, таких як: скалярних типів (scalar types), об'єктних типів (object types), перелічень (enum), вхідних типів (input), інтерфейсів (interface) та об'єднань (union). Доступні операції також описуються за допомогою типів.

Для конвертації C# класів у graphql-сумісні типи, бібліотека «GraphQL.NET» використовує code-first підхід, який полягає у ручному визначенню graphql типів, на основі вже існуючих C# класів (рисунк 1.1).

```
public class Droid
{
    public Guid Id { get; set; }
    public string Name { get; set; } = string.Empty;
    public string Kind { get; set; } = string.Empty;
}

public class DroidGraphType: ObjectGraphType<Droid>
{
    public DroidGraphType()
    {
        Field(x => x.Id);
        Field(x => x.Name);
        Field(x => x.Kind);
    }
}
```

Рис. 1.1 – Дублювання типів

Сама бібліотека, як вирішення цієї проблеми, надає можливість використовувати авто-реєстрацію типів, базуючись на існуючих класах, але у багатьох випадках розробники надають перевагу саме ручному визначенню типів, тому що такий підхід надає змогу покращити продуктивність додатку, а саме підвищити час запуску. У випадку з автогенерацією типів, потрібен аналіз існуючих класів у run-time, що може бути довготривалою дією, особливо, якщо класів в додатку велика кількість. Ручне визначення типів надає змогу бібліотеці завчасно знати та розуміти які типи потрібно створити, тобто часозатратний аналіз класів не потрібен.

Оскільки GraphQL є лише мовою запитів, вона не містить вбудованих знань про джерело даних і не накладає обмежень на спосіб їх отримання. Таким чином, джерелом інформації може виступати будь-яка система: реляційна або документо-орієнтована база даних, зовнішній API, файлова система чи навіть вручну

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				12
Змн.	Арк.	№ докум.	Підпис	Дата		

сформовані об'єкти. Бібліотека GraphQL NET, в свою чергу, дотримується того ж самого підходу, і не надає готових рішень щодо роботи з даними. На це варто звернути увагу, оскільки в сучасних вебдодатках в .NET екосистемі, джерелом даних, в основному, виступає база даних, а для роботи з базами даних використовуються дві основні бібліотеки: Dapper та EntityFramework.

Dapper відноситься до виду інструментів, що називаються micro-orm, основна задача яких, полягає у зіставленні об'єктів (object mapping) [3]. Приклад використання наведено на рисунку 1.2.

```
var sql = "SELECT * FROM Product WHERE CategoryId = @categoryId";
var products = connection.Query<Product>(sql, new { categoryId = 1 }).ToList();
```

Рис 1.2 – Приклад отримання даних за допомогою Dapper

EntityFramework – це повноцінний фреймворк для роботи з базою даних, він дозволяє абстрагуватися від написання sql запитів, шляхом їх автогенерації [4]. Дана бібліотека є досить обширною та вирішує такі задачі як: кешування результатів, відслідковування змін, ліневе завантаження (lazy loading), міграції баз даних, керування ідентифікацією (identity management), керування зв'язками (association management). На рисунку 1.3 продемонстровано приклад використання фреймворку для управління даними (читання, запис).

```
using var db = new ApplicationDbContext();

var blog = await db.Blogs.OrderBy(b => b.BlogId).FirstAsync();
var blogsWithPosts = await db.Blogs.Include(b => b.Posts).ToListAsync();

var newBlog = new Blog { Title = "Blog No 1", Url = "https://example.com" };
db.Blogs.Add(newBlog);
await db.SaveChangesAsync();
```

Рис 1.3 – Приклад управління даним за допомогою EntityFramework

Проблема полягає у тому, що GraphQL .NET не інтегрована під наведені бібліотеки, тому існують зовнішні рішення, що допомагають вирішити цю задачу, але, навіть не зважаючи на це, розробники можуть імплементувати інтеграцію інструментів, що вони використовують, власними силами.

У процесі розвитку програмних проєктів, незалежно від того, чи використовуються вже наявні рішення, чи реалізуються власні – з'являються типові підходи

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				13
Змн.	Арк.	№ докум.	Підпис	Дата		

та стандартизовані патерни, що призводять до дублювання коду з невеликими змінами, схоже до того, як було продемонстровано з оголошенням graphql типів, або з імплементацією типових задач.

Водночас HotChocolate має вбудовану інтеграцію з EntityFramework та імплементацію пагінації, сортування, фільтрації, що спрощує розробку, у випадку використання саме цієї ORM [5]. Проте така тісна залежність від EntityFramework обумовлює обмежену підтримку альтернативних засобів доступу до даних, наприклад, таких як Dapper, інтеграція з яким потребує додаткових зусиль з боку розробника. Тобто, маючи гарну автоматизацію рутинних задач, бібліотека Hotcholate бракує гнучкості. Більше того, загальні підходи, які імплементує бібліотека, не завжди відповідають потребам проєкту, а змінювати їх важко, або взагалі неможливо.

Отже, незалежно від підходів та інструментів розробки, в кожному проєкті, що використовує GraphQL разом з .NET, розробники стикаються з проблемами дублювання коду, як на рівні реалізації типових задач, так і на рівні використання інструментів для роботи з даними.

На основі попереднього розділу, метою дослідження є розробка інструменту для автоматизації рутинних процесів, пов'язаних з побудовою GraphQL API у .NET середовищі. Основна перевага рішення, полягає у зменшенні обсягу роботи розробників, що надає змогу приділяти більше уваги бізнес логіці, аніж написанню типових компонентів проєкту.

Для досягнення поставленої мети, необхідно, шляхом детального аналізу бібліотек для роботи з GraphQL у .NET, виявити ключові елементи, які є повторюваними, а також сформулювати вимоги до функціональності інструменту для автоматизації відповідних процесів. На основі сформованих вимог, розробити архітектуру інструменту автоматизації, і, в подальшому, імплементувати необхідний функціонал, який повинен бути гнучким та зручним у використанні. Завершальним етапом є тестування працездатності розробленого рішення та оцінка його ефективності в реальних умовах.

У багатьох екосистемах розробки програмного забезпечення активно використовуються інструменти командного рядка, призначені для автоматизації

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				14
Змн.	Арк.	№ докум.	Підпис	Дата		

типових та повторюваних дій, до того ж, у багатьох екосистемах CLI інструменти вже стали стандартом.

Оскільки такі інструменти демонструють ефективність у прискоренні розробки та роблять її простішою, у даному дослідженні, засобом для автоматизації виступає саме CLI.

CLI – це тип інтерфейсу взаємодії користувача з програмним забезпеченням, що передбачає використання текстових команд у спеціальному середовищі (терміналі, оболонці або консолі).

Отже, основна задача полягає у реалізації CLI інструменту (модулю) для автоматизації роботи з GraphQL у .NET.

1.2 Аналіз аналогів програмного продукту

Екосистеми для розробки програмного забезпечення включають в себе обширну кількість допоміжних CLI інструментів. Наприклад, такі php backend-фреймворки як Symfony [6] та Laravel [7], використовують CLI інструменти Symfony Console + Maker Bundle, та, Artisan відповідно. Javascript frontend-фреймворк Angular [8] використовує Angular CLI, а backend framework, тієї ж мови, NestJs [9] використовує NestJs CLI.

Наведені CLI інструменти мають обширний функціонал не тільки по генерації типових компонентів, але й по управлінню проєкту в цілому. Оскільки для управління проєктами в екосистемі .NET вже існує dotnet CLI, уваги цьому приділяти не варто, більше того, управління проєктами не входить в перелік задач, що повинні бути виконаними в процесі дослідження.

Для роботи з graphql, в екосистемі javascript, існує багатофункціональна бібліотека graphql-codegen [10], частиною якої є також і CLI, що має назву graphql-codegen/cli. Розглядати згадані CLI інструменти окремо недоцільно, тому що всі вони вирішують одну й ту саму проблему, але для різних мов програмування, або фреймворків. Більше того, тільки NestJs CLI надає можливість працювати у цьому напрямку з GraphQL, всі інші інструменти, в основному, автоматизують роботу по створенню REST API, ORM-моделей, MVC компонентів, repository-pattern класів, тощо.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		15

REST – це архітектурний стиль для побудови розподілених систем, зокрема веб-сервісів, який ґрунтується на використанні простих інтерфейсів, ідентифікованих ресурсів та взаємодій між клієнтом і сервером без стану(stateless).

ORM – це технологія програмування, яка надає змогу налаштовувати взаємодію між об'єктами предметної області з реляційні структурами даних.

MVC – це архітектурний шаблон проєктування програмного забезпечення, який забезпечує розділення системи на три логічні компоненти, що взаємодіють між собою, а саме: Model, View та Controller.

Repository-pattern – це шаблон проєктування, який забезпечує абстракцію доступу до даних, ізолюючи бізнес-логіку від деталей взаємодії з джерелом даних.

Атоматизаційні CLI інструменти для таких php фреймворків як Symfony та Laravel зосереджені саме для створення вищеописаних компонентів. Наприклад, за допомогою команди «php artisan make:controller UserController», що використовується у фреймворку Laravel, можна згенерувати типовий код контролеру саме для цього фреймворку. На рисунку 1.4 наведено приклад згенерованого класу.

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class UserController extends Controller
{
    //
}
```

Рис 1.4 – Приклад згенерованого контролеру

Згенерований код виглядає доволі простим, але, насправді, надає змогу зекономити велику кількість часу, оскільки не потрібно вручну імпортувати та визначати простори імен, не потрібно знати базовий клас, від якого потрібно наслідуватись, а також не потрібно вручну описувати й сам клас.

Схожі команди доступні й для інших компонентів, але щоб сконцентруватися більше на GraphQL, розглянемо екосистему javascript, яка характеризується обширним набіром інструментів для роботи з нею. Такі бібліотеки як graphql-codegen, graphql-codegen/cli, typegraphql-prisma [11], apollo-tools [12], graphql-cli [13], graphql

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		16

nexus [14] – надають велику кількість функціоналу для роботи з graphql, проте слід зазначити, що не всі вони націлені саме на генерацію коду. До тих бібліотек, які працюють безпосередньо з кодогенерацією, відносяться: NestJs, typegraphql-prisma, graphql-codegen/cli. Тому саме ці бібліотеки й варто розглядати у даному контексті.

Спочатку розглянемо NestJs, щоб зрозуміти базові підходи та концепції для роботи з graphql, а потім перейдемо до більш складних реалізацій.

Встановимо фреймворк за допомогою офіційної документації, яка рекомендує починати зі встановлення саме CLI за допомогою node package manager (рисунк 1.5).

```
$ npm i -g @nestjs/cli
```

Рис 1.5 – Встановлення фреймворку

Наступним кроком створимо тестовий проєкт за допомогою команди «nest new test-proj» (рис 1.6).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\Playg> nest new test-proj
✦ We will scaffold your app in a few seconds..

✔ Which package manager would you ❤️ to use? npm
CREATE test-proj/.prettierrc (54 bytes)
CREATE test-proj/eslint.config.mjs (869 bytes)
CREATE test-proj/nest-cli.json (179 bytes)
CREATE test-proj/package.json (2108 bytes)
CREATE test-proj/README.md (5126 bytes)
CREATE test-proj/tsconfig.build.json (101 bytes)
CREATE test-proj/tsconfig.json (565 bytes)
CREATE test-proj/src/app.controller.ts (286 bytes)
CREATE test-proj/src/app.module.ts (259 bytes)
CREATE test-proj/src/app.service.ts (150 bytes)
CREATE test-proj/src/main.ts (236 bytes)
CREATE test-proj/src/app.controller.spec.ts (639 bytes)
CREATE test-proj/test/jest-e2e.json (192 bytes)
CREATE test-proj/test/app.e2e-spec.ts (699 bytes)

✔ Installation in progress... 🔄
🔥 Successfully created project test-proj
```

Рис 1.6 – Створення проєкту

Відразу, за допомогою команди «npm i @nestjs/graphql @nestjs/apollo @apollo/server graphql» встановимо всі потрібні бібліотеки для роботи з graphql.

Детально розглядати встановлені бібліотеки немає потреби, оскільки всі вони, незалежно від мов програмування, фреймворків, екосистем, працюють за допомогою двох основних підходів: schema-first та code-first.

Schema-first підхід передбачає наявність завчасно визначеної graphql-схеми у файлі з розширенням graphql. Перевагою такого підходу є можливість розділити схему на менші частини. До того ж, з таким підходом, маємо єдине джерело істини,

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				17
Змн.	Арк.	№ докум.	Підпис	Дата		

тобто визначену заздалегідь graphql схему, на основі якої можна генерувати типи як для backend так і для frontend. Проте без автогенерації типів на основі визначеної схеми, такий підхід має одні лиш недоліки, тому що потрібно синхронізувати серверні та клієнтні типи відповідно до схеми. У випадках, коли архітектура додатку складна, автогенерація не завжди може бути доцільною, а інколи, й шкідливою.

Code-first підхід передбачає визначення типів через код і їх послідовну генерацію у graphql типи, що потім переносяться або генеруються у graphql схему.

Graphql схема – це компонент, що описує доступні типи даних, дозволені операції і взагалі структуру самого API [15].

Перевагами code-first підходу є: абстрагування від graphql типів, можливість не поглиблюватись у GraphQL SDL, автогенерація схеми, синхронізація між кодом та схемою (тому що схема завжди генерується на основі коду).

Розглянемо як NestJs пропонує працювати з schema-first підходом. На рисунку 1.7 продемонстровано основний підхід, який NestJs пропонує відразу використовувати, а саме: автогенерацію типів за допомогою конфігурації, що визначена у основному модулі додатку.

```
@Module({
  imports: [
    DroidsModule,
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      playground: true
    }),
  ],
  controllers: [AppController],
  providers: [AppService]
});
export class AppModule { }
```

Рис 1.7 – Приклад конфігурації для schema-first підходу

Конфігурація, представлена на рисунку, описує шляхи для пошуку GraphQL схем за допомогою typePaths властивості, а також описує, що потрібно згенерувати типи на основі цих схем у файл graphql.ts, що знаходиться у директорії src; за це відповідає властивість definitions.

Відразу використаємо більш просунутий тип визначення graphql схем, що надає змогу розділяти схему на окремі файли.

Визначимо два файли з описами типів, основний – main.graphql, а також додатковий additional.graphql.

Для полегшення аналізу, визначимо по два типи в кожному файлі і, відповідно, по два запити в кореневому об'єкті Query.

Запит – це операція на отримання певних даних від сервера GraphQL. Клієнти визначають структуру відповіді, що забезпечує точний та ефективний пошук даних.

Операція – це окремий запит, мутація або підписка, яку клієнти надсилають на сервер GraphQL для маніпуляції даними.

Мутація – це операція яка змінює дані на сервері. Вона дозволяє клієнтам виконувати операції створення, оновлення або видалення, змінюючи дані.

Підписка – це довгоживуча операція GraphQL в режимі реального часу, яка забезпечує зв'язок у реальному часі, дозволяючи клієнтам отримувати оновлення даних з сервера, коли відбуваються певні події або зміни.

На рисунку 1.8 визначені запити, а також типи за допомогою GraphQL SDL.

```
type Query{
  humans(): [Human!]
  animals(): [Animal!]
}

type Human{
  id: ID!
  name: String!
  email: String!
  animals: [Animal!]
}

type Animal{
  id: ID!
  name: String!
}

extend type Query{
  droids(): [Droid!]
  warriors(): [Warrior!]
}

type Droid{
  id: ID!
  name: String!
  model: String!
}

type Warrior{
  id: ID!
  armor: Float!
  weapons: [String!]
}
```

Рис 1.8 – Типи, визначені у файлах

В результаті, після запуску додатку будуть згенеровані всі необхідні типи у файлі graphql.ts. Але цього недостатньо для того, щоб побачити будь-який результат, оскільки робота з даними ще не описана (рисунок 1.9).

```

export abstract class IQuery{
  abstract droids(): Nullable<Droid[]> | Promise<Nullable<Droid[]>>;
  abstract warriors(): Nullable<Warrior[]> | Promise<Nullable<Warrior[]>>;
  abstract humans(): Nullable<Human[]> | Promise<Nullable<Human[]>>;
  abstract animals(): Nullable<Animal[]> | Promise<Nullable<Animal[]>>;
}

export class Droid{
  id: string;
  name: string;
  model: string;
}

export type Warrior{
  id: string;
  armor: number;
  weapons?: Nullable<string[]>
}

```

Рис 1.9 – Приклад згенерованих типів

Для того, щоб у відповідь на graphql-запит отримати будь-які дані, використовуються резолвери (resolvers). Щоб визначити resolver в nestjs cli є команда **nest -g resolver <modulename>**, в результаті, у вказаному модулі, буде згенеровано наступний шаблонний код (рисунок 1.10).

```

import { Resolver } from '@nestjs/graphql';

@Resolver()
export class DroidsResolver {}

```

Рис 1.10 – Резолвер

Розширмо цей клас відповідно до схеми, повернувши з функції droids() список відповідних об'єктів (рисунок 1.11).

```

@Query(() => [Droid])
droids(): Droid[] {
  return [
    {
      id: '123',
      name: 'droid_1',
      model: '05-45'
    },
    {
      id: '12345',
      name: 'droid_2',
      model: '05-55'
    },
    {
      id: '1645',
      name: 'droid_3',
      model: '06-66'
    },
  ]
}

```

Рис. 1.11 – Функція

Відкривмо графічний інтерфейс для тестування GraphQL під назвою Altair та виконаємо запит, щоб впевнитись, що все працює (рисунок 1.12)

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				20
Змн.	Арк.	№ докум.	Підпис	Дата		

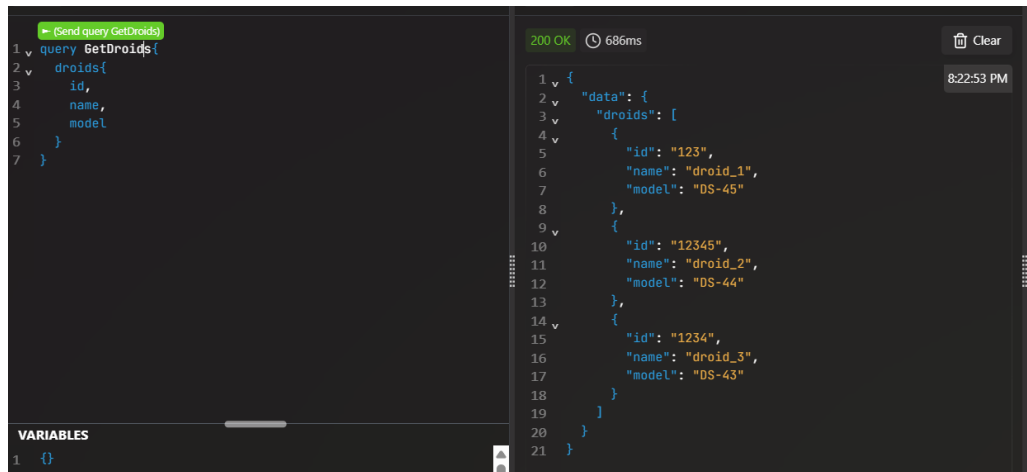


Рис 1.12 – Отримані дані

На даному етапі було описано всі можливості фреймворку Nest та Nest CLI відповідно до schema-first підходу.

Розглянемо code-first підхід. Адаптуємо конфігурацію під даний підхід (рисунк 1.13).

```
@Module({
  imports: [
    DroidsModule,
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      playground: true,
      autoSchemaFile: join(process.cwd(), 'src/schema.gql')
    })
  ],
  controllers: [AppController],
  providers: [AppService]
});
export class AppModule {}
```

Рис 1.13 – Нова конфігурація

В рамках code-first підходу, достатньо визначити лише одну властивість – `autoSchemaFile`, яка призначена для ідентифікації файлу, в який потрібно згенерувати типи. Наступним кроком потрібно визначити типи за допомогою javascript і typescript, на основі яких буде згенеровано graphql схему (рисунок 1.14).

```
import { Field, ID, ObjectType } from '@nestjs/graphql';

@ObjectType({ description: 'recipe' })
export class Droid {
  @Field(() => ID)
  id: string;

  @Field(() => String)
  name: string;

  @Field(() => String)
  model: string;
}
```

Рис 1.14 – Приклад оголошення типу

В даному випадку, типи генеруються за допомогою декораторів (все, що починається з `@`), що схоже на підхід, який вже був продемонстрований раніше,

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		21

тобто, як можна помітити, підходи однакові, різниця лише в синтаксисі. Після того, як типи визначені, можна запустити додаток та переглянути згенеровану схему (рис. 1.15).

```
# -----
# THIS FILE WAS AUTOMATICALLY GENERATED (DO NOT MODIFY)
# -----

type Droid {
  id: ID!
  name: String!
  model: String!
}

type Query {
  droids: [Droid!]!
}
```

Рис 1.15 – Згенеровані типи

При спробі виконати запит, який було продемонстровано при schema-first підході, отримаємо той самий результат.

Аналіз підходу NestJS щодо роботи з GraphQL показав, що CLI-інструментарій обмежується лише базовою генерацією структури резолверів, тоді як генерація схеми, типів, об'єктів – здійснюється за допомогою конфігураційних параметрів модулю. Такий підхід щодо автоматизації, потенційно можна застосувати й у реалізації CLI.

Розглянемо typegraphql-prisma бібліотеку, що представляє собою інтеграцію type-graphql бібліотеки з prisma orm. В основі даної бібліотеки лежить концепція code-first підходу, що є ідентичною до NestJs. Більше того, type-graphql використовує підхід для оголошення класів з декораторами, схожим до реалізації NestJs.

Основна перевага цієї бібліотеки, полягає у можливості генерувати graphql - CRUD операції, звичайні типи (graphql object types) та вхідні типи (graphql input types).

Встановимо та налаштуємо prisma, будемо використовувати SQLite в якості бази даних, а як генератор вкажемо typegraphql. Приклад конфігурації наведено на рисунку 1.16.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				22
Змн.	Арк.	№ докум.	Підпис	Дата		

```

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = "file:./dev.db"
}

generator typegraphql {
  provider = "typegraphql-prisma"
  output   = "../src/generated/type-graphql"
}

```

Рис 1.16 – Конфігурація prisma-схеми

Оголосимо дві моделі та запустимо міграцію за допомогою команди «prisma migrate dev –name init» (рисунок 1.17).

```

model Droid {
  id      Int      @id @default(autoincrement())
  name    String
  email   String   @unique
  weapons Weapon[]
}

model Weapon {
  id      Int      @id @default(autoincrement())
  title   String
  droid   Droid    @relation(fields: [ownerId], references: [id])
  ownerId Int
}

```

Рис 1.17 – Конфігурація prisma-схеми (моделі)

В результаті виконання команди буде створено файл бази даних, а також згенеровані всі необхідні типи та CRUD операції, включаючи фільтрацію, сортування та пагінацію (рисунок 1.18, 1.19, 1.20).

Create Table Create Index Modify Table Delete Table Print Refresh		
Name	Type	Schema
Tables (4)		
Droid		CREATE TABLE "Droid" ("id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT, "name" TEXT NOT NULL, "email" TEXT NOT NULL)
id	INTEGER	"id" INTEGER NOT NULL
name	TEXT	"name" TEXT NOT NULL
email	TEXT	"email" TEXT NOT NULL
Weapon		CREATE TABLE "Weapon" ("id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT, "title" TEXT NOT NULL, "ownerId" INTEGER NOT NULL)
id	INTEGER	"id" INTEGER NOT NULL
title	TEXT	"title" TEXT NOT NULL
ownerId	INTEGER	"ownerId" INTEGER NOT NULL
_prisma_migrations		CREATE TABLE "_prisma_migrations" ("id" TEXT PRIMARY KEY NOT NULL, "checksum" TEXT NOT NULL, "finished_at" DATETIME, "migration_name" TEXT NOT NULL)
sqlite_sequence		CREATE TABLE sqlite_sequence(name,seq)
Indices (1)		

Рис 1.18 – Згенерована база даних

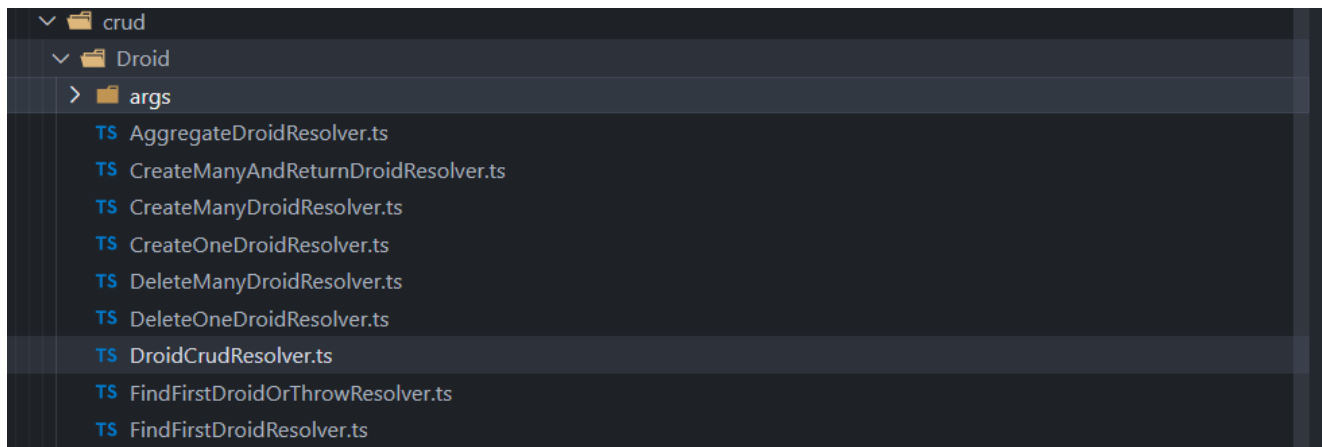


Рис 1.19 – Згенеровані типи та основні операції

```
@TypeGraphQL.Resolver(_of => Droid)
export class DroidCrudResolver {
  @TypeGraphQL.Query(_returns => AggregateDroid, {
    nullable: false,
  })
  async aggregateDroid(@TypeGraphQL.Ctx() ctx: any, @TypeGraphQL.Info() info: GraphQLResolveInfo, @TypeGraphQL.Args() args: AggregateDroidArgs): Promise<AggregateDroid> {
    return getPrismaFromContext(ctx).droid.aggregate({
      ...args,
      ...transformInfoIntoPrismaArgs(info),
    });
  }

  @TypeGraphQL.Mutation(_returns => AffectedRowsOutput, {
    nullable: false
  })
  async createManyDroid(@TypeGraphQL.Ctx() ctx: any, @TypeGraphQL.Info() info: GraphQLResolveInfo, @TypeGraphQL.Args() args: CreateManyDroidArgs): Promise<AffectedRowsOutput> {
    const { _count } = transformInfoIntoPrismaArgs(info);
    return getPrismaFromContext(ctx).droid.createMany({
      ...args,
      ...(_count && transformCountFieldIntoSelectRelationsCount(_count)),
    });
  }
}
```

Рис 1.20 – Приклад CRUD резолвера

Отже, маємо повну автоматизацію основних операцій за допомогою лише однієї команди: «`prisma migrate dev --name init`», що також працює на основі конфігураційного файлу.

Наступним кроком розглянемо `graphql-codegen` та `graphql-codegen/cli`. Ця бібліотека, в основному, використовується на frontend стороні й використовує `schema-first` підхід. Основна перевага бібліотеки полягає у інтеграції з багатьма frontend бібліотеками, такими як `react` (`tanstack/react-query`, `urql`), `Vue Apollo`, `Angular Apollo`, `Svelte Apollo`. Для всіх перерахованих бібліотек доступна генерація типів, запитів, мутацій, підписок, генерація функцій. Приклад конфігураційного файлу наведено на рисунку 1.21.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		24

```
import type { CodegenConfig } from '@graphql-codegen/cli'

const config: CodegenConfig = {
  schema: 'https://localhost:4000/graphql',
  documents: ['src/**/*.tsx'],
  generates: {
    './src/gql/': {
      preset: 'client',
    },
  },
}

export default config;
```

Рис 1.21 – Конфігураційний файл

Конфігураційний файл не є обов'язковим, досягти того ж самого результату можна за допомогою CLI команди (рисунк 1.22).

```
graphql-codegen \
  --schema http://localhost:400/graphql \
  --documents "src/**/*.graphql" \
  --generates src/generated/graphql.ts \
  --plugins typescript typescript-operations
```

Рис 1.22 – Приклад CLI команди

Бібліотека широко використовує концепцію плагінів, що надає гнучкості інструменту, оскільки розробники можуть його легко розширювати власним функціоналом. У таблиці 1.1 продемонстровані основні плагіни, що є найбільш застосованими у розробці.

Таблиця 1.1

Найпоширеніші плагіни

Плагін	Призначення
typescript	Генерація базових TypeScript-типів
typescript-operations	Генерація типів для GraphQL-запитів
typescript-react-apollo	React hooks («useQuery», «useMutation»)
typescript-resolvers	Генерація resolvers для бекенду
fragment-matcher	Генерація інфо про union/interfaces
introspection	Завантаження схеми через introspection

Щодо backend частини, бібліотека не надає настільки обширної автоматизації як для frontend й обмежується тільки генерацією типів та порожніх резолверів. Також, варто зазначити, що використовується schema-first підхід, тобто backend

розробники обов'язково мають визначати схему. Наступний приклад демонструє спосіб застосування (рисунок 1.23).

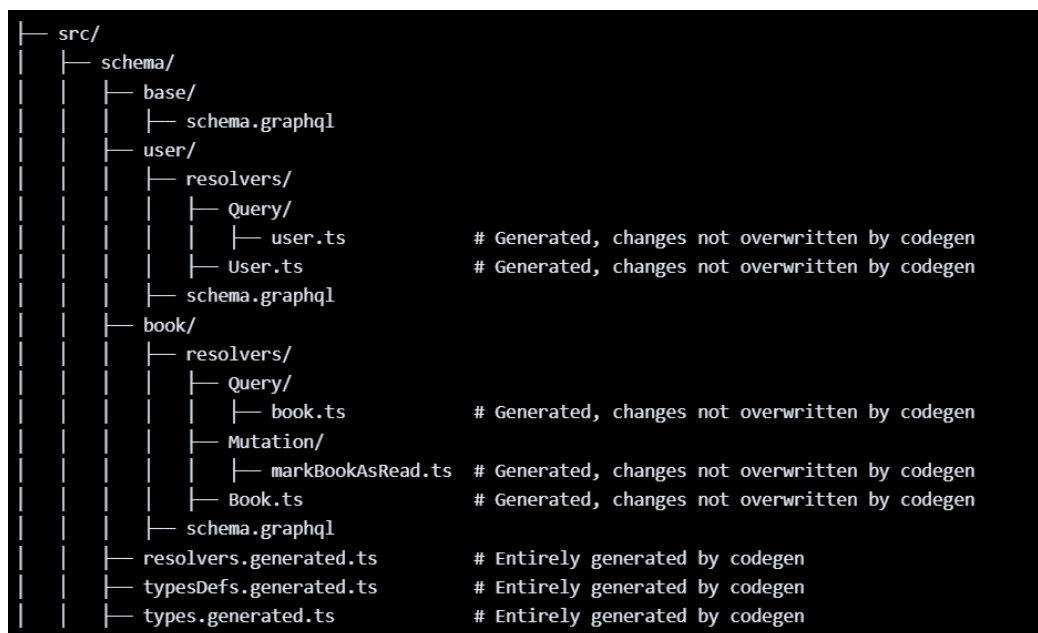


Рис 1.23 – Приклад використання

Згенерований резолвер наведено на рисунку 1.24.

```

import type { QueryResolvers } from '../types.generated';

export const user: NonNullable<QueryResolver['user']> = async = (_parent, _arg, _ctx) => {
  /* Implement Query.user resolver logic here */
}

```

Рис 1.24 – Резолвер

Отже, бібліотека реалізує широкий функціонал під найвідоміші frontend бібліотеки, дещо обмеженіший функціонал також реалізований й для backend бібліотек (Apollo, GraphQL Yoga). Можна зробити висновок, що бібліотека, хоча й тільки серед найпопулярніших рішень, є майже framework agnostic.

У результаті аналізу інструментів, виявилось, що в реалізації кожного з них широко застосовується підхід з використанням конфігураційних файлів. Незважаючи на це, деякі рішення надають можливість не використовувати конфігураційні файли, шляхом заміни їх на CLI команди.

Огляд інструментів, також дозволив виявити наступні їх недоліки та переваги. Перевагами бібліотеки **Graphql Code Generator (graphql-codegen)** є наявність плагінної архітектури, яка надає інструменту гнучкості у використанні; незалежність від конкретного backend або frontend фреймворку, шляхом імплементції

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		26

автоматизації для найпопулярніших рішень; обширний функціонал та зручність розробки, наприклад підтримка watch режиму.

Недоліками бібліотеки є обмежена модульність: усі згенеровані компоненти знаходяться у одному файлі, що ускладнює підтримку; підтримка лише schema-first підходу, що переважає написання схеми вручну; відсутність генерації CRUD-операцій, фільтрації, пагінації, сортування.

Перевагами бібліотеки **Typegraphql-prisma** є автоматична генерація CRUD-операцій, включно з фільтрацією, сортуванням і пагінацією; модульна структура згенерованих файлів, що полегшує масштабування та підтримку.

Недоліками бібліотеки є тісний зв'язок з prisma-orm; генерація базується виключно на конфігураційному файлі, немає CLI інтерфейсу; відсутність контролю згенерованих файлів, наприклад через CLI, тому генерується велика кількість файлів, можливо, навіть непотрібних.

Переваги бібліотеки **NestJS** полягають у підтримці code-first та schema-first підходів та наявності CLI.

До недоліків бібліотеки відносяться висока залежність від NestJS фреймворку; попри наявність CLI, відсутній обширний набір команд; відсутність генерації CRUD операцій; відсутність модульності, що призводить до генерації компонентів в один файл.

У таблиці 1.2 наведено підсумок для розглянутих інструментів:

Таблиця 1.2

Порівняльна таблиця інструментів

Критерій / Інструмент	Graphql codegen	Typegraphql-prisma	NesJS
Підхід	Schema-first	Підтримка code-first	Code-first & schema-first
Генерація CRUD-операцій	Відсутня	Наявна	Відсутня
Модульність згенерованого коду	Відсутня, генерація в один файл	Наявна	Відсутня, генерація в один файл
Гнучкість налаштувань	Наявна, через використання плагінів	Тільки обмежений конфігураційний файл	Тільки обмежений конфігураційний файл
Гнучкість налаштувань	Наявна, через використання плагінів	Тільки обмежений конфігураційний файл	Тільки обмежений конфігураційний файл

Режим спостереження (watch)	Наявний	Відсутній	Відсутній
Залежність від фреймворків	Залежність відсутня частково	Залежний від Prisma ORM	Залежний від NestJS
Підтримка фільтрації, пагінації, сортування	Відсутня	Наявна	Відсутня
Можливість використання без конфіг. Файлу	Є можливість	Немає можливості	Немає можливості

Варто зазначити, що жоден з проаналізованих інструментів, не забезпечує автоматичної генерації шаблонного коду, який враховує індивідуальні архітектурні рішення розробників. Існуючі інструменти, зосереджуються переважно на типових схемах або фреймворк-орієнтованих патернах, тобто генерація базується на уніфікованих підходах, що не враховують внутрішню структуру проєкту, або модульну організацію, що формується безпосередньо командою розробки.

Це відкриває можливості для виділення кращих практик з існуючих рішень і адаптацію їх у .NET, тому що у екосистемі .NET такі автомазичні інструменти не розвинені. Більше того, є можливість реалізації алгоритму порівняння, що дозволить автоматично виявляти загальні, повторювані фрагменти логіки та генерувати на основі їх нові.

1.3 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Автоматизаційні інструменти, зокрема генератори коду або сервіси для полегшення розробки, можуть бути реалізованими з використанням будь якої мови програмування, кожна з яких має свої недоліки та переваги, але на даному етапі буде розглянено такі мови як: GO, RUST, C#, Python, C++. Такий вибір обумовлений тим, що наведені мови програмування активно використовуються у сфері розробки інструментів, сервісів, бібліотек; мають обширні екосистеми та підтримку спільноти; дозволяють створювати продуктивні, кросплатформні та розширювальні рішення; забезпечують доступ до API компілятора, AST-аналізу або відповідних бібліотек, що є важливим для генерації коду.

У таблиці 1.4 наведена порівняльна характеристика наведених мов програмування.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		28

Таблиця 1.3

Порівняння мов програмування

Характеристика	C# (.NET)	Go	Rust	C++	Python
Типізація	Статична	Статична	Статична	Статична	Динамічна
Продуктивність	Висока (JIT, близька до нативної)	Висока (компіляція у нативний код)	Дуже висока, без runtime	Дуже висока	Середня / низька
Кривизна навчання	Середня	Низька	Висока (особливо через ownership)	Висока	Низька
Кривизна навчання	Переважає Microsoft-екосистема	Відсутність дженериків (до недавнього часу)	Складна для новачків	Складна підтримка, складна компіляція	Повільна, проблеми зі скейлінгом
Екосистема	Потужна, розвинена (.NET, NuGet)	Менша, але швидко зростає	Активно розвивається, але ще молода	Велика, але фрагментована	Дуже велика (pip, PyPI, open-source)
Застосування	Web, Desktop, Cloud, Enterprise, Tools	Мікросервіси, CLI, сервери	Системне ПО, безпечні аплікації	Ігри, системне ПО, драйвери	Скрипти, Data Science, ML, Web
Парадигми програмування	ООП, функціональне	Імперативне	Функціональне, імперативне	ООП, імперативне, шаблонне	ООП, імперативне, функціональне
Розмір виконуваного файлу	Середній	Малий	Малий	Малий	Великий (інтерпретатор + залежності)
Можливість розширення плагінами	Висока (через .NET, MEF, DI тощо)	Можлива, але менш усталена	Можлива, через crates	Можлива, але не стандартна	Дуже гнучка через модулі та пакети
Сильні сторони	Інтеграція з .NET, UI, Enterprise	Простота, concurrency (goroutines)	Безпечна робота з пам'яттю	Контроль над ресурсами, performance	Простота, скрипти, ML/AI

Базуючись на наведеній таблиці, можна обрати найбільш оптимальний варіант для розробки, проте, насправді, ключовим фактором при виборі технології для розробки такого роду інструментів є зв'язок мови з тим середовищем, для якого інструмент призначений. У даному випадку – це .NET та бібліотеки для роботи GraphQL, що використовують C#, тому доцільно розробляти інструмент саме цією мовою програмування. Такий підхід полегшує підтримувальність додатку, через залучення розробників, для яких реалізується автоматизація, вони можуть

аналізувати код, покращувати його, розширювати. Гарним прикладом, з попереднього розділу, є бібліотека graphql-codegen та її підхід з плагінами, що дозволяє гнучко розширювати функціонал, оскільки кодова база одна. Саме тому вибір .NET та C# є обґрунтованим і стратегічно доцільним.

Висновки до першого розділу

У даному розділі було окреслено основну проблематику, шляхом аналізу не тільки основних бібліотек для роботи з GraphQL, а й бібліотек для роботи з даними, оскільки вони завжди взаємопов'язані між собою. Також було здійснено детальний аналіз аналогічних рішень із демонстрацією принципів їхньої роботи, що дозволило оцінити їхні можливості не лише на концептуальному, але й на практичному рівні.

Додатково приділено увагу аналізу CLI-додатків в цілому, з метою виявлення особливостей їхнього дизайну та формування вимог до побудови власного інструменту. У результаті було визначено ключові переваги та недоліки існуючих рішень, на основі яких сформульовано висновки щодо застосування найкращих практик у майбутньому додатку. Крім того, було обґрунтовано вибір інструментів і мови програмування шляхом порівняльного аналізу.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		30

РОЗДІЛ 2. ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Опис вимог

У таблиці 2.1 визначено опис функціональних вимог додатку.

Таблиця 2.1

Опис функціональних вимог

Вимога	Деталізація
Генерація GraphQL-типів на основі моделі (code-first підхід)	Можливість приймати шлях до C# класу як вхідний параметр
	Можливість аналізувати C# класи
	Формувати відповідні типи, орієнтуючись на цільову бібліотеку, що вказується в параметрах
Підключення та виконання користувацьких плагінів	Можливість зчитувати користувацькі плагіни
	Можливість виконання зчитаного коду плагіну, відповідно до API
	Взаємодіяти з плагіном через основний код, а також через CLI (можливість передавати параметри, тощо)
Генерація C# типів на основі GraphQL-схеми (schema-first підхід)	Можливість зчитувати та аналізувати файл, у якому визначена схема
	Підтримувати модульний підхід щодо генерації типів
	Підтримувати параметри, які дозволяють керувати модульністю генерації
Порівняння файлів	Виявляти зміни між файлами, щоб виявити повторювальні місця та генерувати на основі них нові файли
Генерація на основі результатів порівняння	Можливість генерувати новий файл, базуючись на виявлених змінах

Також до функціональних вимог відносяться типові вимоги щодо CLI додатків: підтримувати парадачу аргументів, валідація аргументів, надання документації при неправильному використанні команд, або при запиті на документацію.

Нефункціональні вимоги, також визначені у таблиці 2.2.

Таблиця 2.2

Опис нефункціональних вимог

Вимога	Деталізація
Кросплатформеність	Інструмент має працювати у середовищах Windows, Linux та macOS
Розширюваність	Архітектура має дозволяти легко додавати нові плагіни або шаблони без зміни основного коду

Description	Опис команди	Використовується для надання інформації про ціль виконання команди
Execute	Метод для виконання команди	Метод, що виконує основну логіку команди
ParseArgs	Метод, для трансформації команди в об'єкт	Метод, що створює типізовані аргументи

Основним користувачем, й, відповідно, актором є backend розробник, що використовує C# бібліотеки для роботи з graphql.

Першим, і найбільш типовим сценарієм є використання code-first підходу і генерація класу, що описує graphql тип, на основі уже наявної моделі або сутності. У цьому випадку, користувач викликає інструмент генерації за допомогою командного рядка, передаючи всі необхідні для генерації параметри: тобто шлях до файлу, де міститься клас, на основі якого потрібно згенерувати інший, а також назву бібліотеки, для якої це потрібно зробити.

Другим варіантом використання є написання власного плагіну для генерації будь-якого коду, а потім передачі у CLI параметрів, які визначають, де знаходиться імплементація плагіну, на основі чого CLI інструмент виконає код плагіну відповідно до вказівок.

Третім варіантом використання, є створення graphql схеми, виклик CLI інструменту з параметрами, що визначають шлях до graphql схеми, на основі якої, будуть згенеровані відповідні типи. Передбачено використання різноманітних параметрів для керуванням генерації для забезпечення модульності програми, якщо це потрібно. Базовий сценарій передбачає генерацію типів в окремі файли, що будуть знаходитися в одній й тій самій директорії.

Четвертий сценарій охоплює можливість виявлення змін, базуючись на двох, вже існуючих, файлах, й генерація такого ж третього, але з відповідними змінами, які будуть вказані як параметри при виконання CLI-команди.

Останній варіант використання, полягає у визначенні у конфігураційному файлі частин коду, які є унікальними, опис на що їх потрібно замінювати при генерації нового, схожого функціоналу, що надасть гнучкості рішенням.

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		32

Таким чином, запропоновані сценарії демонструють, що інструмент охоплює як базові випадку генерації, так і більш гнучкі варіанти використання, що дозволяють налаштовувати логіку роботи інструменту під конкретні вимоги проєкту, або потреби розробника.

Візуалізуємо вищесказане за допомогою діаграми використання, оскільки вона дозволяє описати взаємодію між акторами та системою та ідентифікувати основні функціональні можливості, а також сценарії використання системи (рисунк 2.1).

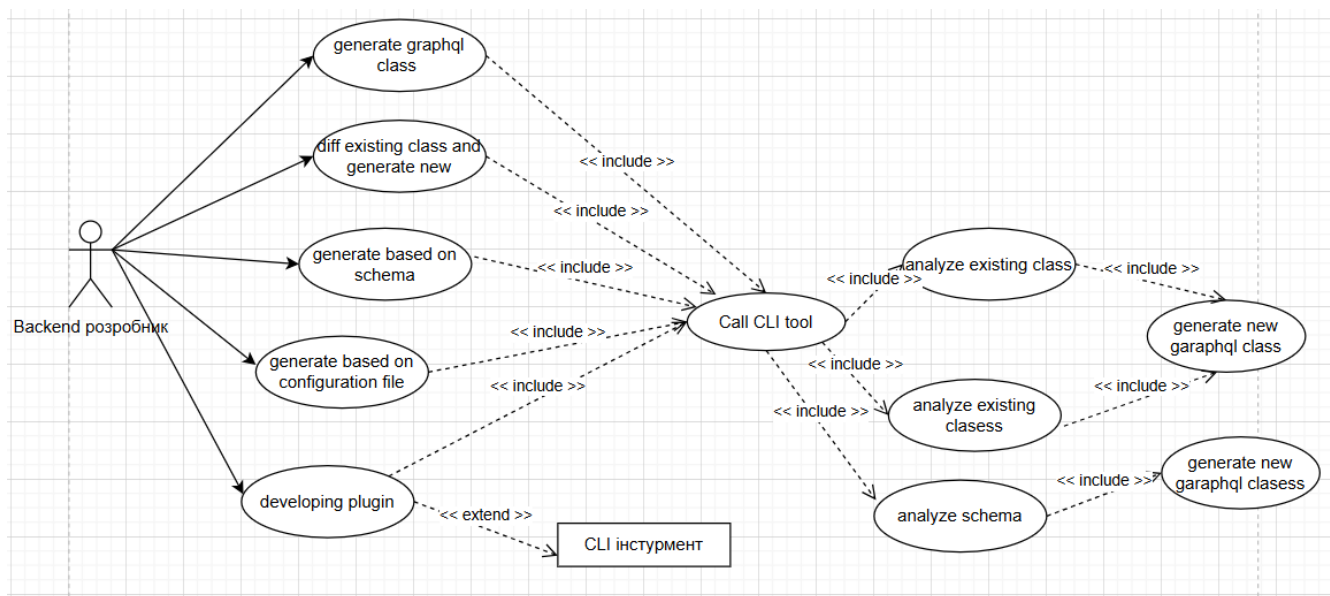


Рис 2.1 – Діаграма використання

2.2 Моделювання взаємодії компонентів системи

З метою формалізації логіки роботи інструменту, доцільно представити основний процес роботи програми у вигляді діаграми активності. Такий підхід дозволяє чітко відобразити послідовність дій, умови переходів та альтернативні гілки виконання, а також продемонструвати загальну структуру виконання команд та зрозуміти як користувач взаємодіє із системою на кожному етапі.

Діаграма на рисунку 2.2. описує основний процес роботи програми.

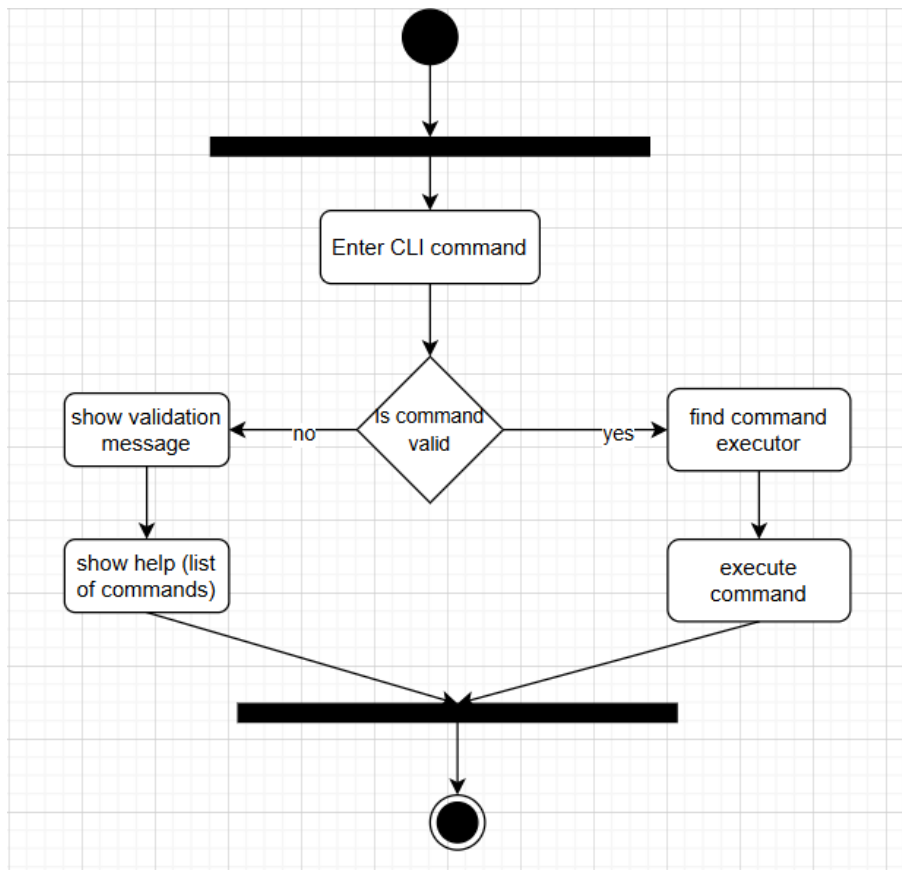


Рис 2.2 – Діаграма активностей для основного процесу роботи програми

Діаграма активностей, що зображена на рисунку 2.3, демонструє процес генерації типів на основі існуючого класу, де очікується передача параметрів, а саме шлях до файлу, що містить клас. При виконанні команди користувачем, система перевіряє існування команди. Якщо наданої команди не існує, виводиться повідомлення про помилку, а також демонструється допоміжна інформація у вигляді списку доступних команд. У разі, якщо команда є валідною, викликається генератор на основі класів, який продовжує подальшу обробку. Наступним кроком здійснюється перевірка наявності відповідного файлу. Якщо файл відсутній, система виводить повідомлення про помилку. Якщо ж файл існує, відбувається додаткова перевірка на вміст класу, а саме: чи містить він публічні властивості. У випадку, коли таких властивостей немає, користувач також отримує повідомлення про помилку. Якщо ж усі умови виконано, команда успішно виконується, а система генерує клас для роботи з GraphQL на основі отриманого опису.

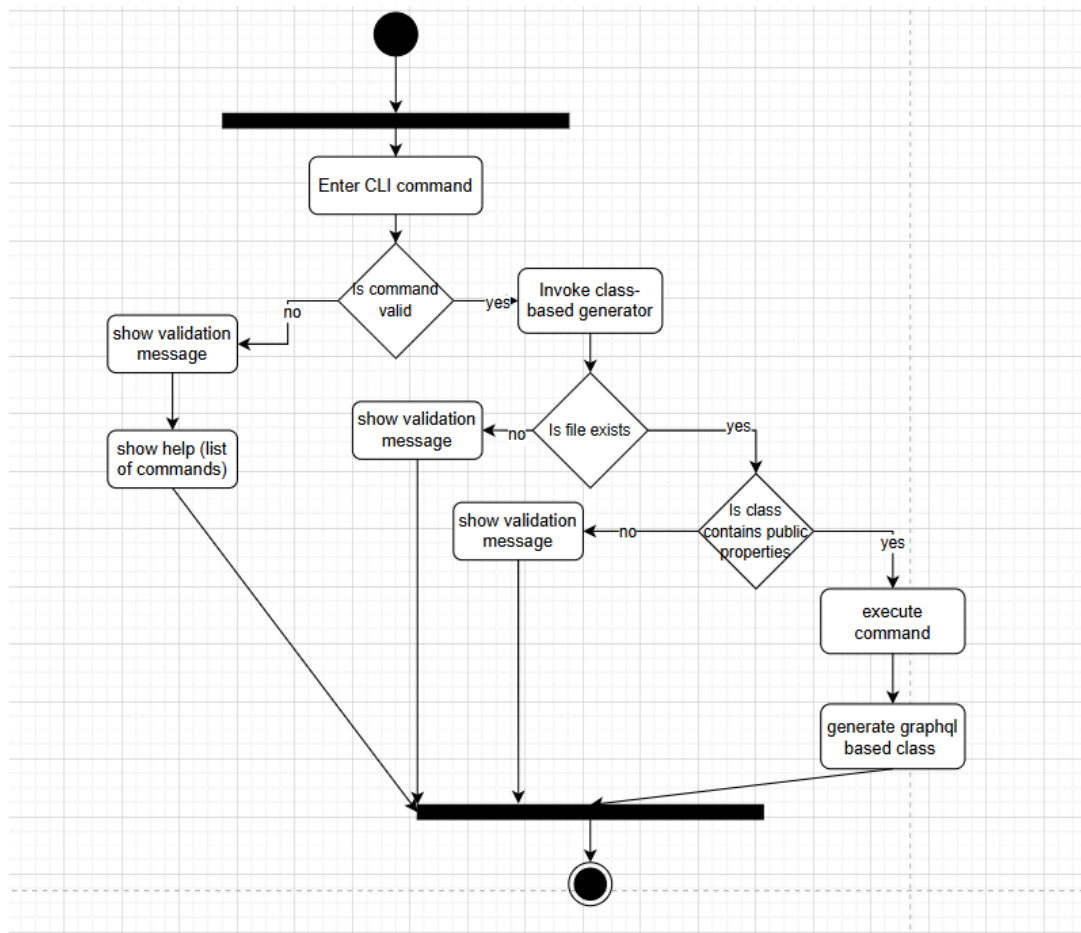


Рис 2.3 – Діаграма активностей для генерації graphql класу

Діаграма на рисунку 2.4 по структурі схожа на попередню, оскільки на ній також описаний процес початкової валідації команди. Єдине, чим дана команда відрізняється від попередньої – це валідація graphql схеми. Валідна GraphQL схема повинна відповідати специфікації та забезпечувати узгодженість усіх типів і операцій. Вона має містити кореневі типи (Query, за потреби Mutation і Subscription), які визначають допустимі точки входу. Усі ідентифікатори (типи, поля, аргументи) повинні бути унікальними, а кожне поле – мати чітко визначений і коректно оголошений тип. Для обов’язкових значень використовується модифікатор «!».

Також схема має містити правильно визначені скалярні, об’єктні, інтерфейсні, enum- та входні типи. Всі посилання на типи повинні бути валідними, без циклічних залежностей. Таким чином, валідна схема гарантує повноту, однозначність і можливість безпомилкової побудови запитів.

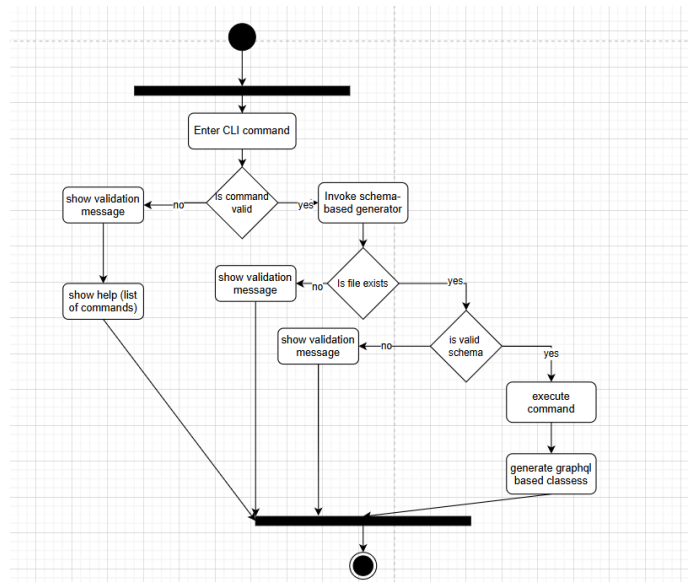


Рис 2.4 – Діаграма активностей для генерації коду на основі схеми

На рисунку 2.5, продемонстровано процес виконання команди, що визначає різницю між вмістом двох файлів та генерує на основі цих змін третій файл, з поміткою місць, які відрізняються, й відповідно повинні бути змінені. Основною відмінністю даної команди від попередньої – це також специфічна валідація, яка перевіряє чи у файлах дійсно є якась різниця. Також важливим кроком є виклик іншого сервісу, що відповідає за генерацію та аналіз вмісту файлів вцілому. Ця команда має недолік у вигляді великої кількості вхідних параметрів, які обов’язкові для її коректного виконання.

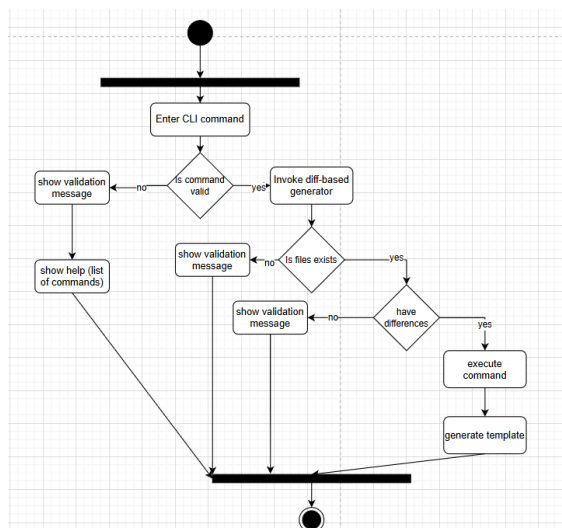


Рис 2.5 – Діаграма активностей для генерації коду на основі класів

Діаграма, що зображена на рисунку 2.6, демонструє процес роботи з плагінами. Процес запуску команди типовий, проте її логіка та кількість кроків

виконання відрзняється значно. Після успішної валідації команди, перевіряється існуючість плагіну, шлях до якого передається через параметри. У випадку, якщо плагін відсутній – виводиться повідомлення про помилку. Якщо ж плагін існує, потрібно перевірити його вміст на наявність класів, що імплементують інтерфейс, що відає за опис команди. Якщо такі класи є, вони спочатку проходять процес підготовки до запуску, а потім, безпосередньо запускаються. При цьому система відповідає з всі помилки, які можуть виникнути під час виконання й реагує на них відповідно.

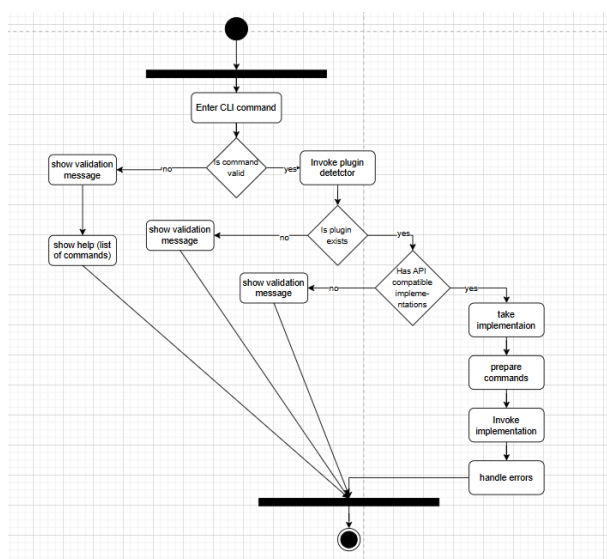


Рис 2.6 – Діаграма активностей для генерації коду на основі класів

Для кращого розуміння внутрішньої логіки роботи інструменту доцільно змодельовати послідовність взаємодії між його основними компонентами. Одним із найбільш наочних способів подання таких взаємодій є діаграма послідовності [16] (рисунок 2.7).

У контексті даної роботи, діаграма демонструє як розробник взаємодіє з інструментом через інтерфейс командного рядка, те, як обробляються вхідні параметри, викликаються механізми парсингу або генерації. Такий підхід дозволяє не лише зрозуміти загальну архітектуру взаємодії, а й виявити ключові точки розширення.

Логіка роботи інструменту буде виконуватись через взаємодію таких основних компонентів як: CLI модуль, FileSystemApi, Roslyn, HotChocolate.Language, ArgumentsParser.

У таблиці 2.3 продемонстровано цілі використання наведених компонентів.

Таблиця 2.3

Опис основних компонентів

Компонент	Опис
CLI модуль	Оркестрування усієї програми
FileSystemApi	Модуль роботи з файловою системою
Roslyn	Модуль, який буде відповідати за аналіз та генерацію C# коду
HotChocolate.Language	Модуль, що буде відповідати за аналіз graphql типів, схем, тощо
ArgumentsParser	Модуль, що відповідає за роботу з параметрами, що передаються до CLI

Найоб'ємнішим та найскладнішим компонентом для реалізації є модуль оркестрування програмою, оскільки всі інші модулі або використовуються вже готові реалізації, або є простими та прамолінійними у реалізації. Також оркестрування програмою включає в себе такі важливі й складні аспекти як: коректна обробка помилок, логування, стабільність роботи у випадку, коли програма завершується неочікувано та непередбачено.

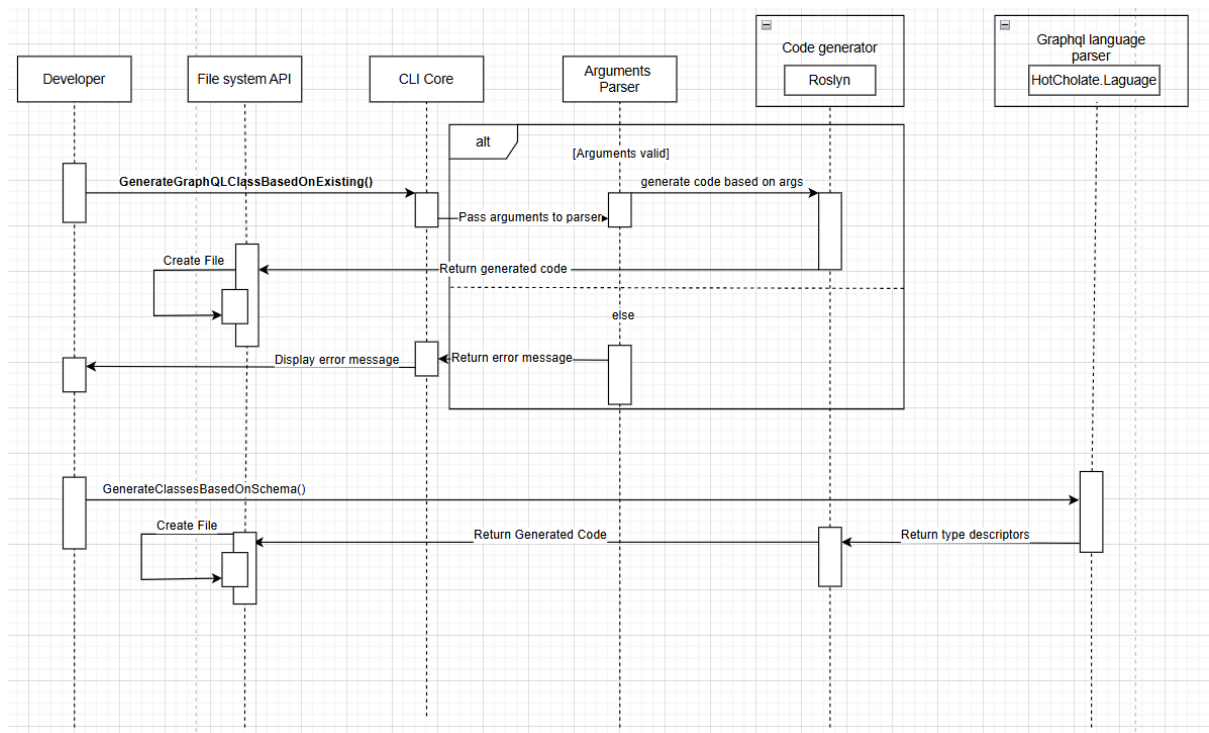


Рис 2.7 – Діаграма послідовності

Звісно, справжня реалізація буде відрізнятися кількістю модулів, що будуть взаємодіяти між собою, але зображати їх на діаграмі недоцільно, оскільки вона

буде перевантаженою та її буде важко читати і розуміти. Тому на діаграмі представлені загальні компоненти програми.

Отже, на даному етапі, було продемонстровано взаємодію між основними компонентами програми, яку ініціює користувач (розробник).

2.3 Проектування та реалізація алгоритмів роботи програми

У даному розділі представлена реалізація програмного засобу для генерації коду на основі вхідних даних. Програму реалізовано як консольний застосунок, який є можливість розширяти за допомогою системи плагінів, що забезпечує модульність і простоту підтримки.

Оскільки взаємодія з програмними компонентами відбувається через консольний інтерфейс, а саме через набір команд з відповідними параметрами, було реалізовано набір абстракцій та основних класів, що допомагають імплементувати команди будь якої складності. На рисунку 2.8 продемонстрована UML діаграма, на якій описані ці основні компоненти:

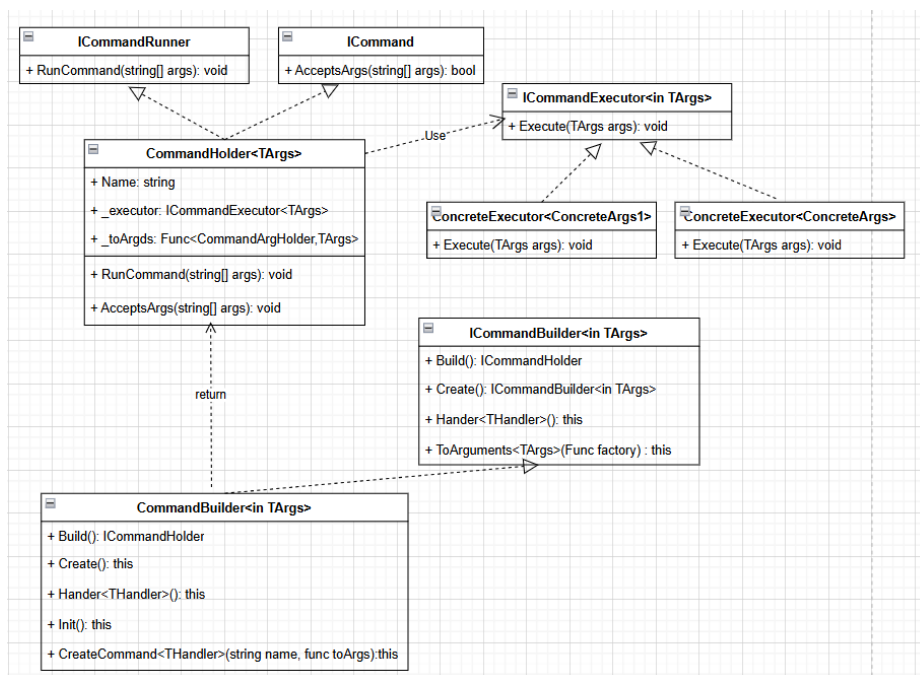


Рис 2.8 – UML діаграма основних класів

Для побудови консольних команд було використано шаблон проектування «Builder» [17], що дозволяє гнучко створювати будь які об'єкти, шляхом застосування «fluent interface». На рисунку 2.9 представлена діаграма, що демонструє загальну структуру шаблону.

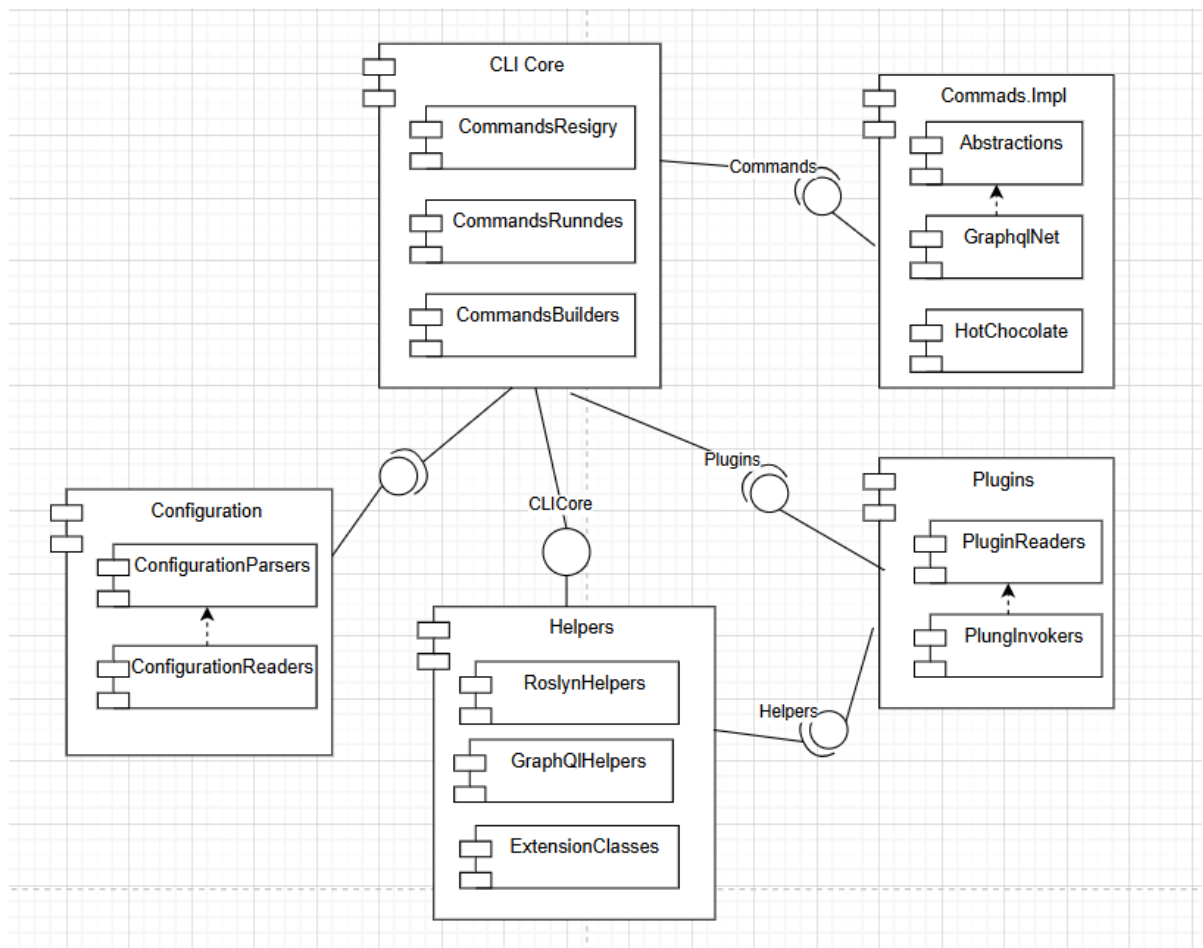


Рис 2.10 – Діаграма компонентів

Діаграма розгортання є типом структурної діаграми, яка використовується для візуалізації фізичного розташування компонентів системи та залежностей між ними на різних вузлах апаратного забезпечення. У даному випадку, поширення CLI-додатку відбувається шляхом його публікації як Nuget-паketу, тому на діаграмі відображено три основних компоненти: машина розробника, сервер Nuget та середовище кінцевого користувача. Діаграма на рисунку 2.12 демонструє, як після компіляції та пакування додатку у формат «.nupkg», відбувається його завантаження на саму платформу Nuget.org, з якої користувачі мають змогу встановити пакет за допомогою dotnet CLI. За допомогою такого підходу, забезпечується централізований механізм розгортання, полегшує оновлення додатку та сприяє його повторному використанню у різних середовищах.

Nuget – це офіційний пакетний менеджер для платформи .NET, який забезпечує механізм створення, поширення та використання повторно використовуваних бібліотек або інструментів у вигляді NuGet-пакетів. NuGet значно спрощує

управління залежностями в проєктах, дозволяючи розробникам легко підключати сторонні або внутрішні бібліотеки без необхідності ручного налаштування або копіювання коду.

Кожен NuGet-пакет являє собою архів із метаданими (файл «.nuspec»), збірками (наприклад, .dll [18]), файлами конфігурацій та, за потреби, додатковими скриптами. Ці пакети можуть публікуватися у відкритому репозиторії NuGet.org або на приватних серверах компаній. Після публікації пакет стає доступним для встановлення через стандартні засоби командного рядка, зокрема dotnet CLI або NuGet CLI.

Таким чином, NuGet виступає ключовим інструментом в екосистемі .NET, сприяючи модульності, повторному використанню коду, ефективному управлінню версіями залежностей та автоматизації процесів розгортання програмного забезпечення (рисунок 2.11).

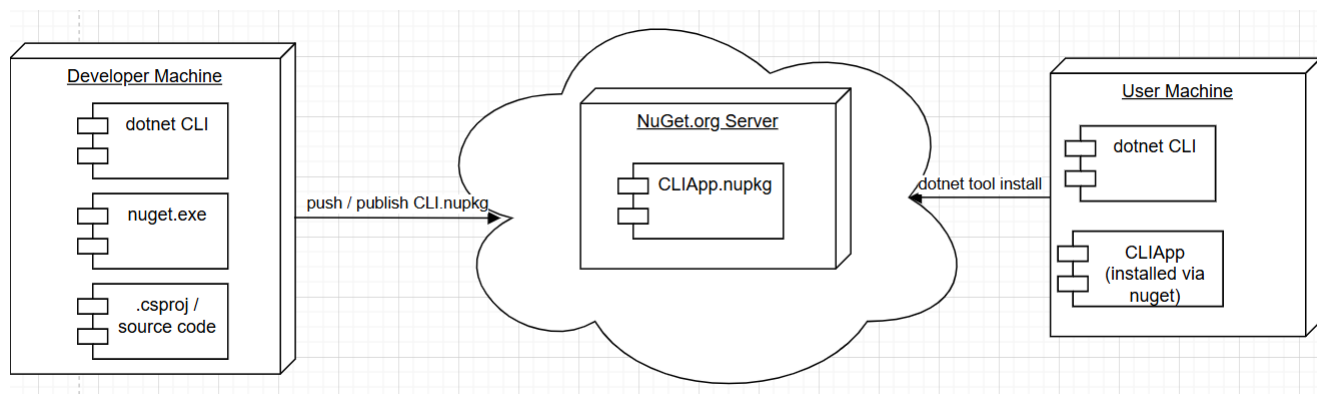


Рис 2.11 – Діаграма розгортання, використовуючи Nuget

2.4 Реалізація консольного додатку

У попередньому розділі було представлено основні компоненти програми на концептуальному рівні. Відповідно, цей розділ зосереджується на конкретних аспектах реалізації програмного забезпечення.

Ядром програми є клас CliCore, що містить в собі лише чотири методи: один публічний метод Run та три приватних TryRun, RegisterCommand, InitComands.

У таблиці 2.4 описано відповідальності цих методів.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		42

Опис основних методів

Метод	Опис
Run	Запускає додаток
TryRun	Намагається виконати команду, яка відповідає переданим параметрам
RegisterCommand	Реєструє будь яку команду
InitComands	Викликає метод реєстрації команд

Приклад використання методу реєстрації зазначено на рисунку 2.12.

```
RegisterCommand<HelpCommand, HelpCommandArguments>(  
    name: "help",  
    p => new HelpCommandArguments(_commands),  
    description: "Help command"  
);  
  
RegisterCommand<GenerateGraphQLClassCommand, GenerateGraphQLClassArgs>(  
    name: "netgql",  
    reader => new GenerateGraphQLClassArgs(reader.GetOrEmpty("f1"), reader.GetOrEmpty("f2"), reader.GetOrEmpty("c1")),  
    description: "Generates class that describes graphql type"  
);
```

Рис 2.12 – Приклад реєстрації команд

RegisterCommand метод використовує CommandBuilder для побудови команди. CommandBuilder, в свою чергу, створює об'єкт CommandHolder<Targs>, що містить в собі всю необхідну інформацію про команду, що імплементує такі інтерфейси як ICommand, ICommandRunner. Основною задачею CommandBuilder є побудова команди для запуску. У таблиці 2.5 наведено відповідальності цих абстракцій:

Таблиця 2.5

Опис абстракцій

Назва	Опис
CommandBuilder	Відповідає за побудову команди: ім'я, опис, аргументи, виконувач
Icommand	Описує сутність, що може бути командою
CommandHolder<Targs>	Абстракція, яка інкапсулює в собі команду. Є командою сама по собі, оскільки може запускатися.

Приклад використання CommandBuilder наведено на рисунку 2.13.

```
public static CommandHolder<TArgs> CreateCommand<THandler>(  
    string name,  
    string description,  
    Func<CommandArgumentsHolder, TArgs> factory  
    ) where THandler : ICommandExecutor<TArgs>, new()  
{  
    return CommandBuilder<TArgs>  
        .Init()  
        .Define(name)  
        .Description(description)  
        .Handler<THandler>()  
        .ToArguments(factory)  
        .Build();  
}
```

Рис 2.13 – Приклад використання CommandBuilder

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		43

Даний метод є однією з найважливіших частин програми, оскільки він відповідає за створення команди. У аблиці 2.6 описано параметри цього методу, які дозволяють зрозуміти основну структуру не тільки цього класу, а й команд загалом.

Таблиця 2.6

Опис методів класу

Назва	Опис
Name	Назва команди
Description	Опис команди
Handler	Абстракція, що виконує команду
Factory	Метод, що конвертує консольні параметри у типізований об'єкт

Такі параметри як name та description є зрозумілими, оскільки це звичайні рядки, тоді як такі параметри як Handler та Factory потребують додаткових пояснень, оскільки вони також є важливо й невід'ємною частиною програми. З сигнатури методу, що представлений на рисунку, зрозуміло, що Handler – це об'єкт, який імплементує інтерфейс ICommandExecutor<Targs>. Цей інтерфейс описує лише один метод Execute, що приймає типізовані аргументи, й, відповідно, імплементує логіку будь якої команди (рисунок 2.14).

```
public void Execute(GenerateAppSchemaArgs rawArgs)
{
    if (string.IsNullOrEmpty(rawArgs.ProjectName) && string.IsNullOrEmpty(rawArgs.Namespace))
    {
        return;
    }

    if (string.IsNullOrEmpty(rawArgs.OutDirPath))
    {
        return;
    }

    var args = NormalizeArgs(rawArgs);
    var schemaClass = ConstructSchemaClass(args.SchemaName);
    var registerTypeMappingsMethod = MakeRegisterMethod(args.RegisterMethodName);
    var constructor = MakeEmptyConstructor(args.SchemaName);
    var classWithMembers = schemaClass.AddMembers(constructor, registerTypeMappingsMethod);
    var namespaceDecl = FileScopedNamespaceDeclaration(IdentifierName(args.Namespace)).AddMembers(classWithMembers);
    var compilationUnit = MakeCompilationUnit(namespaceDecl);
    var unit = compilationUnit.NormalizeWhitespace().ToFullString();
    File.WriteAllText($"{args.OutDirPath}{Path.DirectorySeparatorChar}{args.SchemaName}.cs", unit);
}
```

Рис 2.14 – Приклад реалізації інтерфейсу

Factory параметр – це делегат, що приймає на вхід допоміжний клас для парсингу параметрів та повертає на виході типізовані аргументи команди. Будь який аргумент команди зобов'язаний реалізувати інтерфейс Iargument та публічний метод Match, що надає змогу ідентифікувати, що команда приймає ті аргументи, що передані через консольний інтерфейс (приклад наведено на рисунку 2.15).

```

reader => new RegisterGraphTypeArgs()
{
    GraphTypeNamespace = reader.GetOrEmpty("gn"),
    SchemaClassName = reader.GetOrEmpty("sn"),
    ShcemaPath = reader.GetOrEmpty("path"),
}

```

Рис 2.15 – Приклад визначення лямбда функції, що відповідає делегату

За допомогою вищенаведених абстракцій і класів будуються всі інші компоненти програми. Наступний крок – це, безпосередньо, написання логіки роботи команд та реєстрація їх у основному класі програми CliCore.

У таблиці 2.7 наведено набір вже реалізованих команд для бібліотеки «GraphQL.NET» та мету їх використання.

Таблиця 2.7

Опис команд для бібліотеки «GraphQL.NET»

Команда	Мета
netgql class	Згенерувати новий клас, що описує graphql тип, на основі наявної моделі або сутності.
Netgql schema	Згенерувати клас, що описує graphql схему
netgql register	Реєстрація відповідностей типів: відповідність класу до класу, що описує graphql тип
netgql setup	Автоматично налаштувати всі необхідні компоненти для використання бібліотеки.
Netgql root-query	Створити клас для головного об'єкту, що описує запити
netgql root-mutation	Створити клас для головного об'єкту, що описує мутації
netgql register-roots	Зареєструвати запити та мутації.

Набір команд для бібліотеки «Hotchocolate» представлені у таблиці 2.8.

Таблиця 2.8

Опис команд для бібліотеки «Hotchocolate»

Команда	Мета
hot full-resolver	Згенерувати клас, в якому буде описано резолвер з пагінацією, фільтрацією, сортуванням, використовуючи EntityFrameworkCore.
Hot schema	Згенерувати клас, що описує graphql схему
hot setup	Реєстрація відповідностей типів: відповідність класу до класу, що описує graphql тип
hot add type	Реєструє тип у конфігурації
netgql root-query	Створити клас для головного об'єкту, що описує запити
netgql root-mutation	Створити клас для головного об'єкту, що описує мутації

Як було продемонстровано раніше, кожна реалізована команда імплементує інтерфейс `IcommandExecutor<Targs>`, у якому міститься основний метод `Execute(Targs args)`, реалізація якого, в кожній команді, може мати будь яку складність.

На рисунку 2.16 зображено набір файлів, що імплементують логіку команди для генерації класу, що описує `graphql` тип.

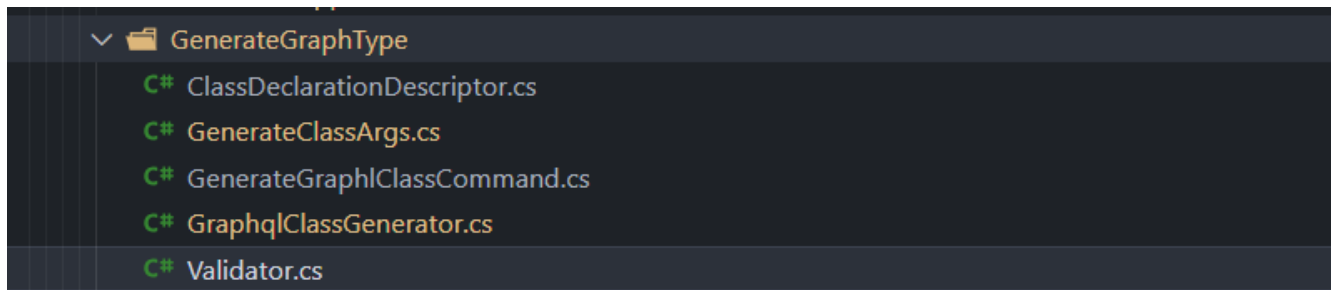


Рис 2.16 – Структура команди

Реалізації методу `Execute` цієї команди наведена на рисунку 2.17.

```
public void Execute(GenerateGraphQLClassArgs args)
{
    if (!_validator.IsValid(args, out var formatted))
        return;

    var code = File.ReadAllText(formatted.ExistingClassPath);
    if (string.IsNullOrEmpty(code))
    {
        Console.WriteLine("Cannot generate code based on empty file");
        return;
    }

    var classCompilationUnit = _generator.CreateCompilationUnit(code, args);
    if (classCompilationUnit == null)
    {
        Console.WriteLine("Failed to generate class, check your input file");
        return;
    }

    File.WriteAllText($"{Path.DirectorySeparatorChar}{args.OutPath}.cs", classCompilationUnit.ToFullString());
}
```

Рис 2.17 – Приклад реалізації методу

В даному випадку, валідуються параметри, зчитується та аналізується код вхідного файлу, а потім, генерується новий клас, та записується в вихідний файл. Метод використовує клас `GraphQLClassGenerator`, що відповідає за основну логіку для побудови класу. Даний клас використовує Roslyn API для генерації C# коду. Наступний код демонструє приклад використання Roslyn API для побудови класу з generic параметрами (рисунок 2.18).

```

public GenericNameSyntax MakeGenericType(string ofType)
{
    return GenericName(Identifier(GraphqlNetBaseClassName))
        .WithTypeArgumentList(
            TypeArgumentList(
                SingletonSeparatedList<TypeSyntax>(
                    IdentifierName(ofType)
                )
            )
        );
}

```

Рис 2.18 – Приклад побудови класу

У таблиці 2.9 зазначено необхідні методи, що містить клас GraphQLClassGenerator для побудови не тільки класу, а й всіх необхідних компонентів, що зазвичай, містить файл з розширенням «CS», включаючи простори імен та «usings».

Таблиця 2.9

Опис методів класу для конструювання класу

Назва методу	Опис
CreateCompilationUnit	Створює синтаксичне дерево CompilationUnitSyntax на основі вхідного коду та аргументів, якщо можлива генерація коректного класу
ConstructClass	Конструює клас, включаючи простори імен, конструктор класу з необхідними викликами методів
InheritGenericClass	Додає до згенерованого класу наслідування від базового класу, що використовуються для опису graphql типів
ConstructGraphQLClass	Створює клас, що описує graphql тип
GetNamespace	Надає простори імен визначеного класу
TryGetNamespace	Допоміжний метод для знаходження просторів імен
MakeFields	Створює виклики методів, що описують graphql поля (Field(x => x.Name))

Попри те, що кожна з реалізованих команд має власну, унікальну логіку, всі вони, як і поточний приклад, використовують Roslyn API як основу, тому можна зробити висновок, що кожна команда – це надбудова над Roslyn API для бібліотек

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		47

«GraphQL .NET» та «Hotchocolate». З огляду на це, було наведено лише одну реалізацію команди, яка демонструє загальний підхід до побудови функціональності.

Roslyn – це платформа від Microsoft для мов програмування C# і Visual Basic, що є частиною екосистеми .NET [19]. Вона надає повноцінний API для аналізу, генерації, трансформації та компіляції коду, відкриваючи доступ до синтаксичного дерева, семантичної інформації та процесу компіляції. Roslyn реалізує компілятори як сервіси, що дозволяє інтегрувати інструменти аналізу коду, автодоповнення, рефакторингу та діагностики безпосередньо у програми.

Наступним кроком було реалізовано систему плагінів, що працює за допомогою рефлексії та Roslyn.

Рефлексія (англ. Reflection) в C# - це механізм платформи .NET, який дозволяє програмі досліджувати свою власну структуру під час виконання [20]. Зокрема, рефлексія дає змогу отримувати інформацію про типи, їхні члени (поля, властивості, методи, події), атрибути, а також створювати екземпляри типів, викликати методи або отримувати та встановлювати значення полів і властивостей динамічно, без прямого звернення до типів у коді. У даному контексті рефлексія використовується для аналізу бібліотек класів, знаходження імплементованих команд та їх подальше виконання.

Якщо плагін визначений в одному лише файлі з розширенням «cs», використовується Roslyn для компіляції коду у пам'яті, надання збірки, яка в подальшому аналізується за допомогою рефлексії, що надає змогу виконувати імплементовані методи (рисуюнок 2.19).

```
public CSharpCompilation CreateInMemoryUnit(SyntaxTree syntaxTree)
{
    return CSharpCompilation.Create(
        "InMemoryAssembly",
        new[] { syntaxTree },
        new[]
        {
            MetadataReference.CreateFromFile(typeof(object).Assembly.Location),
            MetadataReference.CreateFromFile(typeof(Console).Assembly.Location),
        },
        new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary));
}
```

Рис 2.19 – Приклад створення компіляційного коду

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				48
Змн.	Арк.	№ докум.	Підпис	Дата		

Наступний приклад коду демонструє використання рефлексії для знаходження імплементацій інтерфейсу ICommandInvoker<Targs>, виклику його методів, передавання в них необхідних параметрів (рисуюнок 2.20).

```
foreach (var type in assembly.GetTypes())
{
    if (!type.IsClass || type.IsAbstract) continue;
    var executorInterface = type.GetInterfaces()
        .FirstOrDefault(i =>
            i.IsGenericType &&
            i.GetGenericTypeDefinition() == openGenericInterface);

    if (executorInterface == null)
        continue;

    Type genericArgumentsType = executorInterface.GetGenericArguments()[0];
    MethodInfo? parseMethod = type.GetMethod(
        "ParseArgs",
        BindingFlags.Public | BindingFlags.Instance | BindingFlags.FlattenHierarchy
    );
    if (parseMethod == null ||
        parseMethod.GetParameters().Length != 1 ||
        parseMethod.GetParameters()[0].ParameterType != typeof(ArgsReader))
    {
        Console.WriteLine($"Type {type.Name} does not contain definition of ParseArgs(ArgsReader)");
        continue;
    }

    object? executorInstance = Activator.CreateInstance(type);
    object? tArgsInstance = parseMethod.Invoke(executorInstance, [reader]);
    MethodInfo? executeMethod = executorInterface.GetMethod("Execute");
    if (executeMethod == null)
    {
        Console.WriteLine($"Method Execute not found: {type.Name}");
        continue;
    }

    executeMethod.Invoke(executorInstance, [tArgsInstance]);
}
```

Рис 2.20 – Приклад аналізу коду за допомогою рефлексії

За допомогою вищенаведеного коду, також відбувається сканування бібліотек класів, що містять користувацькі плагіни.

Далі, було реалізовано алгоритм порівняння двох файлів, знаходження різниці між ними, й генерація третього файлу, на основі змін у вже проаналізованих файлах. Одним з способів порівняння файлів, є використання Roslyn API для аналізу AST.

Абстрактне синтаксичне дерево – це ієрархічне дерево, яке представляє синтаксичну структуру вихідного коду у вигляді абстрагованих вузлів, що відповідають конструкціям мови програмування [21]. Кожен вузол у дереві відповідає певному елементу коду: наприклад, оператору, виразу, декларації змінної або методу.

AST не включає всі лексичні деталі (такі як коментарі або форматування), зосереджуючись на логічній, структурній сутності програми, що робить його придатним для компілятивного аналізу, трансформацій, рефакторингу та статичного аналізу коду.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		49

Проте у даному випадку, використовувати аналіз різниці абстрактних синтаксичних дерев не є доцільним, оскільки такий підхід передбачає виконання операцій, що потребують рекурсивних алгоритмів, що не є оптимальним підходом. Більше того, порівняння синтаксичних дерев полягає у порівнянні складних об'єктів та структур, що є більш затратним, аніж порівняння, наприклад, рядків. Також синтаксичні дерева надають структурований опис коду, детальну інформацію про його синтаксичні конструкції, яка, в даному випадку, не потрібна.

Альтернативним підходом, є визначення різниці файлів, використовуючи їх порівняння по токенам.

Токен (англ. token) – це найменша значуща одиниця вихідного коду, яка розпізнається на етапі лексичного аналізу [22]. Токени є результатом розбиття тексту програми на логічні елементи, кожен з яких має тип (наприклад, ключове слово, ідентифікатор, оператор, літерал) і, зазвичай, позицію в коді.

Токени створюються у результаті процесу, що має назву токенизація.

Токенизація (англ. tokenization) – це перший етап синтаксичного аналізу, під час якого вихідний текст програми розбивається на послідовність токенів, тобто найменших значущих одиниць мови. Цей процес виконується лексичним аналізатором (лексером) і слугує основою для подальшого побудови синтаксичного дерева [23].

У Таблиці 2.10 описано основні переваги використання порівняння файлів по токенам.

Таблиця 2.10

Перваги використання порівнянн по токенам

Назва	Опис
Послідовність, як список	Токени можна легко зберігати у List<SyntaxToken>, і тоді вони поведуться як звичайні масиви – ідеально для алгоритмів типу LCS (Longest Common Subsequence), або навіть примітивного for-цикла з індексами
Пряме порівняння	Порівняння токенів через SyntaxFactory.AreEqual(t1, t2) чи просто t1.ValueText == t2.ValueText. – швидко і точно.
Проста заміна	Токени можна замінити за індексом, потім перетворити список назад у текст за потреби

Точне відображення змісту	На відміну від вузлів AST, токени включають Trivia (пробіли, коментарі), що дозволяє зберігати оригінальний вигляд коду
---------------------------	---

Розробка власного токенизатора для мови C# не є доцільною з практичного та технічного погляду, оскільки така задача потребує врахування великої кількості граматичних особливостей мови, постійної підтримки відповідності специфікації та забезпечення коректної обробки всіх мовних конструкцій [24]. У свою чергу, платформа Roslyn, вже надає стабільний, перевірений та повнофункціональний механізм лексичного аналізу, який гарантує точну токенизацію відповідно до офіційної граматики C# [25].

На рисунку 2.21 наведено приклад отримання токенів програми за допомогою Roslyn.

```
public List<SyntaxToken> GetTokens(string filePath)
{
    var tree = CsharpSyntaxTree.ParseText(File.ReadAllText(filePath));
    return tree.GetRoot().DescendantTokens().ToList();
}
```

Рис 2.21 – Приклад отримання токенів програми або файлу

Також, основна перевага такого підходу, полягає у способі, за допомогою якого Roslyn надає інформацію про токени. Зазвичай, токен можна трактувати як символ, або рядок, але Roslyn формує токени в об'єкти, що надають інформацію про них, за допомогою якої можна виявляти, наприклад, позицію токена у файлі [26].

У таблиці 2.11 продемонстровані деякі доступні властивості та методи, що надає об'єкт, який описує токен.

Таблиця 2.11

Опис корисних методів

Властивість / Метод	Тип	Опис
Kind()	SyntaxKind	Тип токена (ключове слово, оператор, ідентифікатор тощо).
ValueText	string	Чисте текстове значення токена без обгортки (наприклад, без лапок).
Span	TextSpan	Діапазон позицій токена у вихідному коді (без trivia).
SpanStart	int	Початкова позиція токена в коді.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		51

FullSpan	TextSpan	Діапазон, включаючи trivia (коментарі, пробіли).
LeadingTrivia	SyntaxTriviaList	Trivia (коментарі, пробіли) до токена.
HasTrailingTrivia	bool	Чи має токен trivia після себе.
GetLocation()	Location	Інформація про позицію у файлі (рядок/стовпець).

Зважаючи на це, основний алгоритм полягає у порівнянні токенів за допомогою ValueText властивості, запам'ятовуючи їх позицію, а потім, на основі параметрів, що будуть передані через консольний інтерфейс, замінити токен на відповідне значення параметру [27]. Приклад коду, що порівнює токени з двох файлів наведено на рисунку 2.22.

```
public List<(int index, string token1, string token2)> CompareTokens(
    List<SyntaxToken> tokens1,
    List<SyntaxToken> tokens2
)
{
    var result = new List<(int, string, string)>();
    int max = Math.Max(tokens1.Count, tokens2.Count);

    for (int i = 0; i < max; i++)
    {
        string t1 = i < tokens1.Count ? tokens1[i].ValueText : "";
        string t2 = i < tokens2.Count ? tokens2[i].ValueText : "";

        if (t1 != t2)
        {
            result.Add((i, t1, t2));
        }
    }

    return result;
}
```

Рис 2.22 – Метод порівняння токенів, отриманих з файлів

Крок з заміною токенів за індексами, де є відмінності, наведено на рисунку 2.23.

```
public List<SyntaxToken> ReplaceTokens(
    List<SyntaxToken> originalTokens,
    List<TokenDifference> differences,
    string replaceTo
)
{
    var resultTokens = new List<SyntaxToken>(originalTokens);

    foreach (var diff in differences)
    {
        if (diff.Index >= 0 && diff.Index < resultTokens.Count)
        {
            var original = resultTokens[diff.Index];
            var replaced = Identifier(
                original.LeadingTrivia,
                replaceTo,
                original.TrailingTrivia
            );

            resultTokens[diff.Index] = replaced;
        }
    }

    return resultTokens;
}
```

Рис 2.23 – Метод для заміни токенів за індексами, де є відмінності

На даному етапі, реалізована більша частина функціональності додатку, проте, під час розробки, виявилось, що деякі команди потребують приймати багато параметрів для коректної роботи, а саме параметри пов'язані з просторами імен. Основна проблема полягає у тому, що простори імен не завжди відповідають структурі папок у проєкту, а деякі файли потребують використання специфічних директив підключення простору імен (ключове слово `using`). До того ж, деякі параметри команд є досить типовими, що вимагає у користувача постійно вказувати одні й ті самі параметри. Поточне рішення відповідає основній архітектурі додатку, що полягає у можливості мати незалежні команди, що не потребують ніяких зовнішніх конфігурацій, але такий підхід виявляється не гнучким у рамках використання інструменту. Тому, було прийнято рішення реалізувати підхід, який широко застосовується подібними інструментами – робота через конфігураційний файл [28]. Основною метою такого підходу є визначення основних метаданих додатку, в якому використовується інструмент автоматизації. У даному випадку, використовується файл у форматі `json`, в якому описуються основні параметри додатку. На даному етапі конфігураційний файл дозволяє описувати шаблони простору імен, наприклад, що шлях до файлу `Folder1/Folder2/GraphQL/Types` має власний визначений простір імен `MainApp.GraphTypes`. Також конфігураційний файл використовується для гнучкішої роботи зі `schema-first` підходом, де можна вказати директорії, в які потрібно генерувати вихідні файли.

На рівні додатку реалізовано статичний клас `ConfigurationHolder`, при наявності конфігураційного файлу, зчитує його й надає доступ до конфігураційних параметрів всім іншим компонентам програми. Приклад використання наведено на наступному рисунку 2.24.

```
private string NormalizeNamespace(GenerateAppSchemaArgs args)
{
    if (!string.IsNullOrEmpty(args.Namespace))
    {
        return args.Namespace;
    }

    var schemaPath = args.OutDirPath.Replace("/", ".");
    return $"{ConfigurationHolder.TryGetValue("ProjectName")}{schemaPath}";
}
```

Рис 2.24 – Використання конфігураційного файлу для подубови простору імен

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		53

Оскільки пакування, розповсюдження та публікація програмного продукту є невід’ємними етапами повного циклу розробки, у цьому розділі також розглядається процес створення NuGet-пакету та його підготовки до публікації. Такий підхід відповідає принципам завершеної розробки програмного забезпечення, де фінальна форма продукту передбачає зручне встановлення кінцевим користувачем.

Для того, щоб опублікувати додаток як інструмент потрібно спочатку необхідні зміни у файл проєкту.

Файл проєкту C# (.csproj) – це XML-документ, що визначає конфігурацію, структуру та залежності C#-проєкту в екосистемі .NET. Він містить інформацію про тип проєкту (наприклад, консольний застосунок, бібліотека), цільову платформу, використовувані пакети, вихідні файли, умови збірки та інші параметри, необхідні для компіляції та публікації застосунку.

В даному випадку, пакет публікується як інструмент, що й потрібно вказати у файлі. У таблиці 2.12 наведено всі необхідні параметри (метадані), що потрібно вказати для правильної публікації пакету.

Таблиця 2.12

Опис необхідних метаданих

Параметр	Опис
<PackAsTool>	Визначає, що проєкт буде паковано як глобальний або локальний dotnet tool.
<ToolCommandName>	Ім’я команди, яку користувач буде запускати з CLI (наприклад, dotnet sparkql).
<PackageId>	Унікальний ідентифікатор NuGet-пакета. Використовується під час публікації та встановлення.
<Version>	Номер версії пакета. Дотримується формату SemVer.
<Authors>	Автор(и) пакета. Відображається на NuGet.org.
<RepositoryUrl>	Посилання на репозиторій проєкту (GitHub, GitLab тощо).
Продовження таблиці 2.12	Продовження таблиці 2.12

Файл проєкту розробленого додатку наведено на рисунку 2.25.

```

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <PackAsTool>true</PackAsTool>
    <ToolCommandName>sparkql</ToolCommandName>
    <PackageId>SparkQL</PackageId>
    <Version>1.0.11</Version>
    <Authors>Bekker Mykhailo</Authors>
    <RepositoryUrl>https://github.com/Ajoke32/diploma</RepositoryUrl>
    <Description>GraphQL automation (*_*)</Description>
    <PackageLicenseExpression>MIT</PackageLicenseExpression>
    <PackageReadmeFile>README.md</PackageReadmeFile>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\Sparkql.Codegen\Sparkql.Codegen.csproj" />
  </ItemGroup>

  <ItemGroup>
    <None Include="LICENSE.MD" Pack="true" PackagePath="" />
    <None Include="README.md" Pack="true" PackagePath="" />
  </ItemGroup>
</Project>

```

Рис 2.25 – Файл проєкту реалізованого додатку

Також, як видно на рисунку, важливими компонентами додатку є ліцензія та readme файли.

LICENSE.md файл визначає правові умови використання програмного забезпечення. У ньому зазначається, яку ліцензію застосовано (наприклад, MIT, Apache 2.0, GPL тощо), що дозволено користувачеві (використання, модифікація, розповсюдження) та які обмеження або умови застосовуються. Наявність ліцензії є обов’язковою умовою для публікації відкритого коду на більшості платформ (зокрема GitHub, NuGet). У даному випадку використовується ліцензія MIT.

README.md файл виконує роль технічної документації початкового рівня, в якій подається загальна інформація про призначення проєкту, інструкції щодо встановлення, приклади використання, вимоги до середовища тощо.

Без вищеописаних файлів, пакет не пройде валідацію на nuget.org.

Після того, як всі конфігураційні дії виконано, можна збирати пакет за допомогою команди «dotnet pack» (рисунок 2.26).

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		55

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL> dotnet pack Sparkql.Codegen\Sparkql.Codegen.csproj --configuration Release
Restore complete (0.6s)
Sparkql.Codegen net9.0 succeeded (0.5s) → Sparkql.Codegen\bin\Release\net9.0\Sparkql.Codegen.dll
Sparkql.Codegen succeeded with 4 warning(s) (3.4s) → Sparkql.Codegen\bin\Release\net9.0\Sparkql.Codegen.dll
C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.Codegen\Commands\CommandBuidler.cs(7,18): warning CS8618: No
n-nullable field '_name' must contain a non-null value when exiting constructor. Consider adding the 'required' modifier
or declaring the field as nullable.
C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.Codegen\Commands\CommandBuidler.cs(9,35): warning CS8618: No
n-nullable field '_executor' must contain a non-null value when exiting constructor. Consider adding the 'required' modi
fier or declaring the field as nullable.
C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.Codegen\Commands\CommandBuidler.cs(11,32): warning CS8618: N
on-nullable field '_command' must contain a non-null value when exiting constructor. Consider adding the 'required' modi
fier or declaring the field as nullable.
C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.Codegen\Commands\CommandBuidler.cs(13,47): warning CS8618: N
on-nullable field '_toArgs' must contain a non-null value when exiting constructor. Consider adding the 'required' modif
ier or declaring the field as nullable.
Build succeeded with 4 warning(s) in 6.2s
```

Рис 2.26 – Результат виконання команди

Після чого, останнім кроком, завантажити пакет на nuget.or: за допомогою CLI або через вебсайт (рисунки 2.27, 2.28, 2.29).

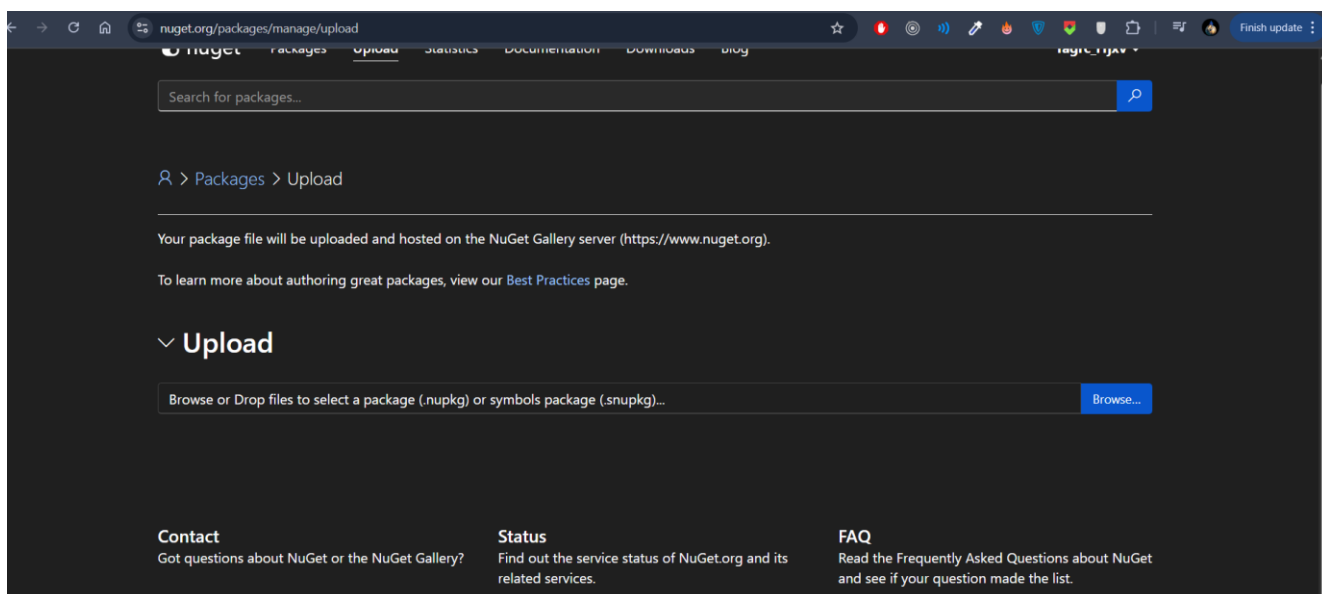


Рис 2.27 – Приклад завантаження пакету через сайт

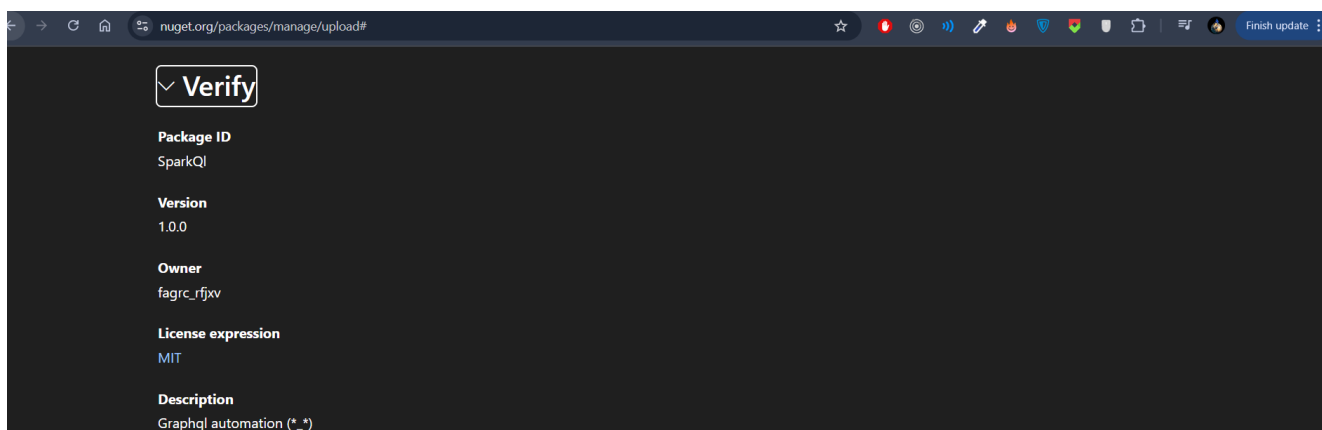


Рис 2.28 – Інформація про пакет

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		56

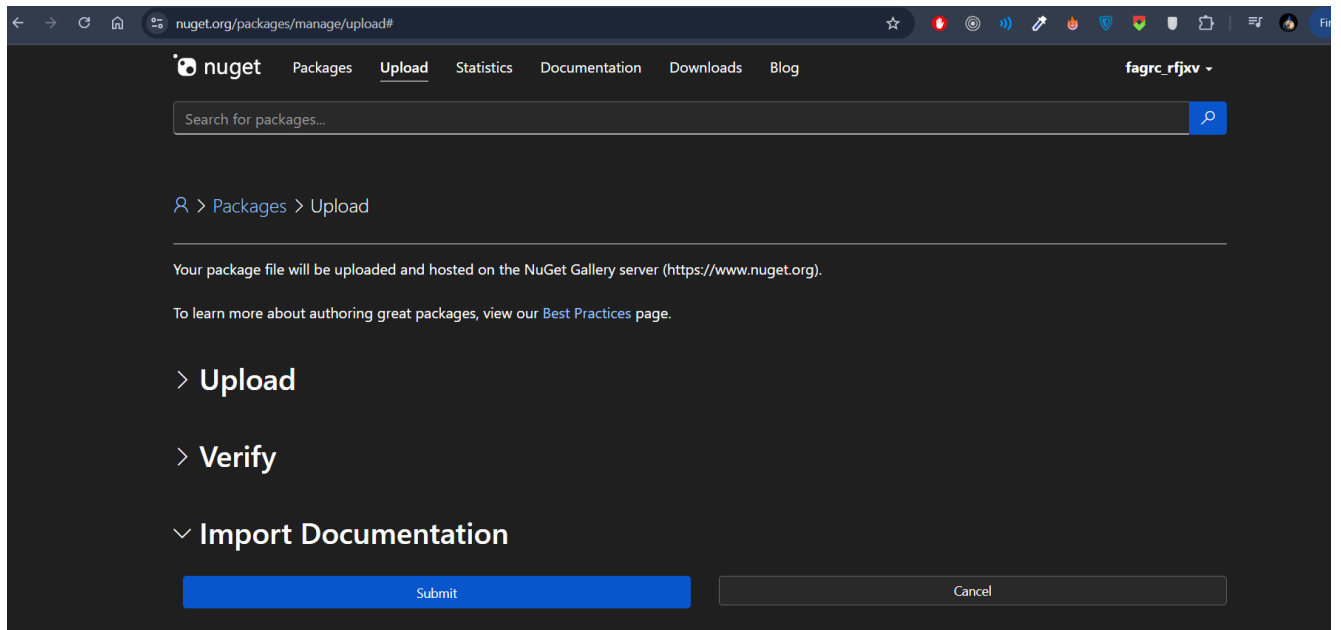


Рис 2.29 – Інформація про пакет

Після відправки пакету потрібно почекати деякий час, поки він пройде додаткову валідацію та стане публічно доступний для завантаження (рисунок 2.30).

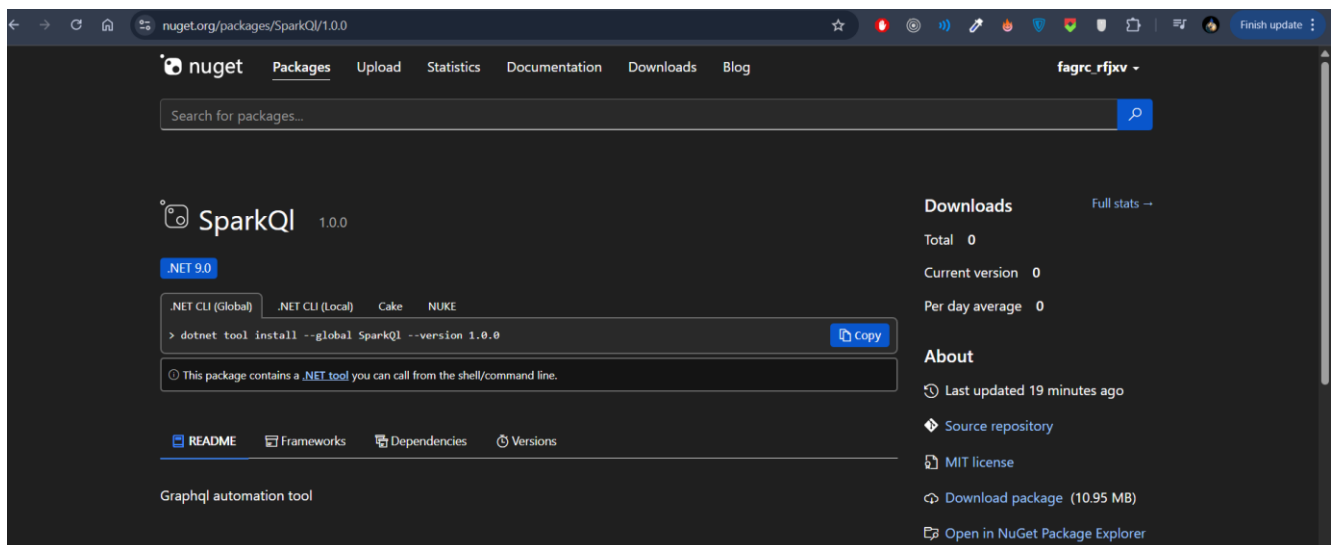


Рис 2.30 – Публічно доступний для завантаження пакет

Висновки до другого розділу

У цьому розділі було сформульовано функціональні та нефункціональні вимоги до системи, а також визначено основних акторів, що дозволило окреслити очікувану поведінку програмного забезпечення, його ключові характеристики та взаємодію з користувачами й зовнішніми системами. На основі отриманих вимог змодельовано взаємодію компонентів системи, спроектовано та реалізовано ключові алгоритми роботи програми. Окрему увагу приділено опису архітектурних рішень

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		
						57

і підходів до реалізації окремих функціональних можливостей. Крім того, здійснено пакування застосунку та його публікацію у вигляді пакета NuGet, що забезпечує можливість подальшого поширення та інтеграції.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		58

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З КОНСОЛЬНИМ ДОДАТКОМ

3.1 Порядок встановлення та налаштування системи

Модуль для роботи з GraphQL для .NET – це консольний додаток, реалізований за допомогою мови програмування C# та бібліотек класів. Такий підхід забезпечує модульність та повторне використання коду, що є важливою вимогою при розробці сучасних програмних засобів. Після завершення розробки програмний продукт було упаковано у формат NuGet-пакету, що є стандартним способом поширення бібліотек у середовищі .NET. Надалі пакет було опубліковано в централізованому репозиторії NuGet.org, який виступає єдиним офіційним джерелом для отримання та оновлення сторонніх бібліотек, що гарантує їхню автентичність та підтримку.

NuGet.org надає стандартизований механізм розповсюдження програмних компонентів для .NET, завдяки чому користувачі можуть встановлювати бібліотеку безпосередньо з офіційного репозиторію. Для цього використовується .NET CLI – інструмент командного рядка, який дозволяє швидко додавати, видаляти та оновлювати залежності. Встановлення пакету виконується однією командою, що зображено на рисунку 3.1.

```
dotnet tool install --global SparkQL
```

Рис. 3.1 – Команда для встановлення

Якщо ж є потреба інтегрувати інструмент безпосередньо в окремий проєкт, можна використати іншу команду

```
dotnet add package SparkQL
```

Рис. 3.2 – Команда для встановлення у проєкт

Мінімальні вимоги, яким повинен відповідати пристрій для використання додатку, наведені у таблиці 3.1. Крім того, для належної роботи рекомендується встановити останню стабільну версію .NET SDK та мати доступ до мережі Інтернет для отримання пакету з NuGet.org.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		59

Мінімальні вимоги

Назва	Вимога
Операційна система	Windows 10 / Linux (Ubuntu 20.04+) / macOS 10.15 або новіші
Процесор	1,6 ГГц або швидший, з підтримкою архітектури x64
Оперативна пам'ять	2 ГБ або більше
Місце на диску	100 МБ вільного простору для встановлення додатку та залежностей

3.2 Структура інтерфейсу та порядок роботи з консольним додатком

Як було зазначено в попередньому розділі, консольний додаток доступний у реєстрі nuget.org (рисунок 3.3), тому першим кроком є встановлення пакету у вигляді глобального інструменту (dotnet tool). Встановити пакет можна двома способами: через середовище розробки, або, більш універсальним способом – через dotnet CLI.

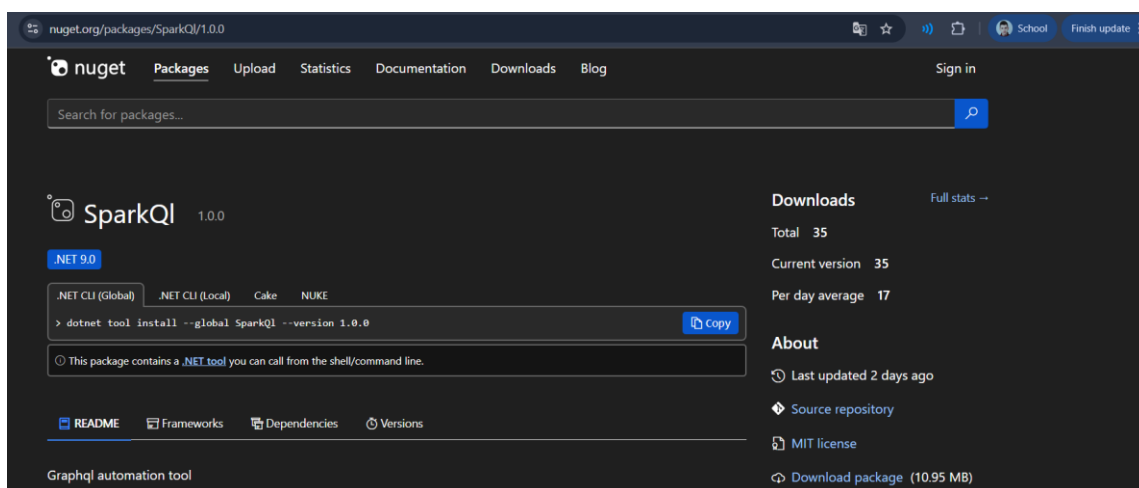


Рис 3.3 – Додаток, розміщений на Nuget.org

Для встановлення пакету через dotnet CLI, достатньо виконати команду: «dotnet tool install --global Sparkql» (рисунок 3.2) після чого, інструмент буде доступний для використання через команду «sparkql».

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL> dotnet tool install --global SparkQL --version 1.0.0
You can invoke the tool using the following command: sparkql
Tool 'sparkql' (version '1.0.0') was successfully installed.
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL>
```

Рис 3.4 – Приклад встановлення через команду

Для перевірки роботи додатку, можна виконати команду help, щоб побачити доступні команди (рисунок 3.5).

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				60
Змн.	Арк.	№ докум.	Підпис	Дата		

```

PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql help
Executing command help
Command: netgql setup
Arguments: p (startup file)
Description: setups GraphQL .NET library

Command: netgql class
Arguments: f1 (source file) f2 (output file) e (entity)
Description: generates class to describe graphql type

Command: netgql schema
Arguments: f1 (schema file) o (output file)
Description: generates classes based on schema

Command: netgql register
Arguments: e (entity) o (output file) n (namespace) clr (clr type) graph (graph type)
Description: registers type mapping

```

Рис 3.5 – Приклад виконання команди «help»

Упевнившись, що додаток працює, можна починати роботу з ним. Для цього потрібно створити будь-який додаток, що використовує бібліотеку «GraphQL .NET» або «HotChocolate».

Потім, для коректної роботи додатку, потрібно визначити конфігураційний файл у форматі json з базовими параметрами: rootDir, projectNamespace. Ці параметри потрібні для правильної ідентифікації параметрів, а саме шляхів до файлів, що передаються в консольний додаток (рисунк 3.6).

```

{
  "rootDir": '.',
  "projectNamespace": "GraphQLNetDemo"
}

```

Рис 3.6 – Приклад конфігураційного файлу

В складних додатках конфігурація може потребувати додаткових параметрів, оскільки при генерації будь яких класів, потрібно описувати й також простори імен та взаємозв'язки між ними, що можуть бути визначеними нестандартним шляхом.

Порядок роботи з бібліотекою «GraphQL .NET»

Будь який graphql додаток починається зі схеми, оскільки саме схема є основним компонентом, який описує всі інші. Для генерації схеми (класу) доступна команда «sparkql netgql schema», що приймає такі параметри як: шлях до директорії в яку потрібно згенерувати схему, простір імен (опціонально), назву схеми (опціонально), назву методу для реєстрації відповідностей типів (опціонально):

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql netgql schema o=/Schema
-----Info-----
Generating schema class to: /Schema...
Schema class AppSchema has been succesfully generated!
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> |
```

Рис 3.7 – Приклад виконання netgql schema команди

В результаті буде згенеровано файл з наступним вмістом, що наведено на рисунку 3.8.

```
using GraphQL.Types;

namespace GraphQLNetDemo;

public class AppSchema : Schema
{
    public AppSchema(IServiceProvider provider) : base(provider)
    {
    }

    public void RegisterTypeMappings()
    {
    }
}
```

Рис 3.8 – Приклад згенерованого класу

На даному етапі схема не описує жодного типу, запиту, мутації або підписки, оскільки вони не були згенеровані. Для генерації запиту, мутації або підписки доступна команда: `sparkql netgql <rootObjectName>`, де `rootObjectName` – це назва компоненту, який потрібно згенерувати. Команда приймає такі параметри як: назва об'єкту, шлях до директорії в яку потрібно згенерувати схему, помітку про спосіб оголошення класу (`partial`, `common`), а також опціонально простір імен та простори імен, що потрібно підключити (рисунок 3.9).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql netgql root-query o= ./ n=RootQuery t=common
-----Info-----
Generating root query class to: ./...
RootQuery class has been succesfully generated!
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> |
```

Рис 3.9 – Приклад виконання команди `sparkql netgql root-query`

В результаті буде згенеровано клас з вмістом, наведеним на рисунку 3.10.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		62

```
using GraphQL.Types;

namespace GraphQLNetDemo;

public class RootQuery : ObjectGraphType
{
    public RootQuery(){}
}
```

Рис 3.10 – Приклад згенерованого запиту

Наступним кроком, виконавши команду `sparkql netgql inject root`, можна зареєструвати клас до схеми, що приведе до зміни попередньо згенерованої схеми. Команда приймає такі параметри як: шлях до файлу зі схемою, шлях до файлу з визначени класом, а також опціонально простір імен (рисунок 3.11).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql netgql inject root s=./Schema root=./RootQuery
-----Info-----
Injecting RootQuery class to AppScheme...
RootQuery class has been succesfully injected!
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> |
```

Рис 3.11 – Приклад виконання команди `sparkql inject root`

В результаті оновлена схема буде мати вигляд, що продемонстровано на рисунку 3.12. Тобто кореневий запит буде призначено у властивість `Query`, що знаходиться у базовону класі схеми й ініціалізується, у даному випадку, через конструктор класу.

```
public class AppSchema : Schema
{
    public AppSchema(IServiceProvider provider) : base(provider)
    {
        Query = provider.GetRequiredService<RootQuery>();
    }

    public void RegisterTypeMappings(){}
}
```

Рис 3.12 – Приклад згенерованої схеми

Аналогічні кроки потрібно застосувати для таких компонентів як: мутація та підписка. Альтернативним рішенням є запуск команди `sparkql netgql roots`, що згенерує основні об'єкти, а також схему, в якій ці об'єкти будуть автоматично зареєстровані. Команда приймає такі параметри як: шлях до директорії в яку потрібно

згенерувати файли, а також, опціонально, список об'єктів, які не потрібно генерувати та реєструвати. Наприклад, запустивши команду наступним чином: «sparkql netgql roots d=Folder exclude=subscription, буде згенеровано схему з кореневими запитом та мутацією (рисунк 3.13, 3.14).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql netgql roots d=Folder exclude=subscription
-----Info-----
Injecting root objects into AppSchema...
Root objects have been succesfully injected!
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> |
```

Рис 3.13 – Приклад виконання команди sparkql netgql roots

```
public class AppSchema : Schema
{
    public AppSchema(IServiceProvider provider) : base(provider)
    {
        Query = provider.GetRequiredService<RootQuery>();
        Mutation = provider.GetRequiredService<RootMutation>();
    }

    public void RegisterTypeMappings(){}
}
```

Рис 3.14 – Результат виконання: згенерована схема з кореневими об'єктами

На даному етапі кореневі об'єкти не описують жодного типу або запиту. Для генерації класу, що описує graphql ти доступна команда sparkql netgql class, що приймає такі входні параметри як: шлях до файлу з існуючим класом, шлях до директорії в яку потрібно згенерувати новий клас, а також опціонально простори імен для використання та визначення. Припустимо, що у додатку уже існує клас User, який розміщено у папці Models, що зображено на рисунку 3.15.

```
public class User
{
    public string Name { get; set; } = string.Empty;
    public string Surname { get; set; } = string.Empty;
    public string Email { get; set; } = string.Empty;
}
```

Рис 3.15 – Приклад існуючого класу

Запустивши команду «sparkql netgql class s=Models/User.cs o=Types/UserGraphType.cs», буде згенеровано клас, що описує graphql тип (рисунк 3.16, 3.17).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql netgql class s=Models/User.cs o=Types/UserGraphType.cs
-----Info-----
Generating UserGraphType class based on User model...
UserGraphType class has been succesfully generated! Out directory: ./Types
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> |
```

Рис 3.16 – Приклад виконання команди sparkql netgql class

```
using GraphQLNetDemo.Models;
using GraphQL.Types;

public class UserGraphType : ObjectGraphType<User>
{
    public UserGraphType()
    {
        Field(x => x.Name);
        Field(x => x.Surname);
        Field(x => x.Email);
    }
}
```

Рис 3.17 – Результат виконання: згенерований клас для опису graphql типу

Тепер, описаний graphql тип, можна використовувати у запитах, мутаціях або підписах, але мати тільки один типу – недостатньо, потрібно також описати сам запит, мутацію або підписку, яка буде відповідати за маніпуляцію над згенерованим типом. Для цього доступна команда «sparkql netgql query», що приймає такі параметри як: директорія в яку потрібно згенерувати файл, назву запиту, мутації або підписки, а також, опціонально, залежності (рисунок 3.18).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql netgql query d=./Queries n=UserQuery
-----Info-----
Generating UserQuery class...
UserQuery query has been succesfully generated! Out directory: ./Queries
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> |
```

Рис 3.18 – Приклад виконання команди sparkql netgql query

В результаті виконання команди буде згенеровано клас, що наведено на рисунку 3.19.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		65

```
using GraphQL.Types;

public sealed class UserQuery : ObjectGraphType
{
    public UserQuery(){}
}
```

Рис 3.19 – Результат виконання: згенерований клас для опису запиту

На даний момент клас порожній та не описує жодного запиту(поля). Щоб додати запит для отримання користувачів, потрібно виконати команду «sparkql netgql query resolver», що приймає такі параметри як: шлях до запиту, назву запиту, шлях до файлу з описом типу, а також опціональну помітку, що ідентифікує чи повинен запит бути асинхронним. В результаті виконання команди, до попередньо згенерованого запиту, буде додано наступне поле, що надає доступ до операцій з користувачами (рисунок 3.20, 3.21).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\SparkQL.App> sparkql netgql query resolver q=./Queries/UserQuery t=Types/UserGraphType n=users
-----Info-----
Injecting Resolver into UsersQuery object...
UserQuery has been succesfully updated with a new resolver!
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\SparkQL.App> |
```

Рис 3.20 – Приклад виконання команди sparkql netgql query resolver

```
public sealed class UserQuery : ObjectGraphType
{
    public UserQuery()
    {
        Field<ListGraphType<UserGraphType>>("users")
        .Resolve(context => new List<User>());
    }
}
```

Рис 3.21 – Результат виконання: згенерований резолвер

Наступним кроком потрібно додати запит для отримання користувачів до кореневого запиту за допомогою команди «sparkql netgql rootQuery add», що приймає такі параметри як: шлях до файлу запиту, який потрібно додати, шлях до файлу з кореневим запитом, назву запиту. В результаті виконання команди, до кореневого запиту буде додано запит для отримання користувачів (рис. 3.22, 3.23).

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		66

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQl\Sparkql.App> sparkql netgql rootQuery add q=./Queries/UserQuery root=./RootQuery n=UserQuery
-----Info-----
Injecting UserQuery class into the root query object...
UserQuery has been succesfully injected!
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQl\Sparkql.App> |
```

Рис 3.22 – Приклад виконання команди sparkql netgql rootQuery add

```
public sealed class RootQuery : ObjectGraphType
{
    public RootQuery()
    {
        Field<UserQuery>("userQuery")
        .Resolve(_ => new {});
    }
}
```

Рис 3.23 – Результат виконання

Всі вищеописані команди можна запускати більше ніж один раз – вміст файлу буде збережено, а нові сутності будуть додані до вже існуючих.

На даному етапі згенеровано мінімальний набір компонентів для тестування graphql операцій, тому наступним кроком є конфігурація бібліотеки у вебдодатках, де вона, в основному, й використовується. У контексті .NET для побудови вебдодатків використовується фреймворк ASP.NET, з яким бібліотека «GraphQL.NET» найчастіше інтегрується, використовуючи HTTP протокол, тому інтеграція бібліотеки полягає у реєстрації відповідних компонентів у контейнері залежностей, а також визначення кінцевої точки для доступу до GraphQL API. Для виконання зазначених кроків, доступна команда «sparkql netgql setup», що приймає такі параметри як: шлях до файлу зі схемою (рисунок 3.24).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQl\Sparkql.App> sparkql netgql setup s=./AppSchema
-----Info-----
Updatiing Program.cs class with basic configuration...
Program.cs has been succesfully updated! /graphql endpoint is available
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQl\Sparkql.App> |
```

Рис 3.24 – Приклад виконання команди sparkql netgql setup

В результаті виконання команди, до файлу Program.cs буде додано конфігурацію, що зображена на рисунку 3.25. Згенерований код описує вхідну точку для доступу до graphql API за допомогою Minimal API, який надається фреймворком ASP.NET. Також додається схема, конфігурується провайдер помилок, таким чином, що вони було якомога детальнішими й зрозумілими.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		67

Команду можна запускати декілька раз та для файлу з будь яким вмістом, згенерований код ніколи не перепише сам файл, а тільки додасть відповідний код в потрібні місця

```
builder.Services.AddSingleton<RootQuery>();
builder.Services.AddSingleton<RootMutation>();
builder.Services.AddGraphQL(builder =>
{
    builder.AddSchema<AppSchema>()
        .AddSystemTextJson()
        .AddErrorInfoProvider(opt =>
        {
            opt.ExposeExceptionDetails = true;
            opt.ExposeExceptionDetailsMode = GraphQL.Execution.ExposeExceptionDetailsMode.Message;
        }).AddGraphTypes();
});

var app = builder.Build();
app.MapPost("/graphql/api", async (HttpContext context, IDocumentExecuter<ISchema> documentExecuter, IGraphQLSerializer serializer) =>
{
    var request = await serializer.ReadAsync<GraphQLRequest>(context.Request.Body, context.RequestAborted);

    var opts = new ExecutionOptions
    {
        Query = request?.Query,
        DocumentId = request?.DocumentId,
        Variables = request?.Variables,
        Extensions = request?.Extensions,
        CancellationToken = context.RequestAborted,
        RequestServices = context.RequestServices,
        User = context.User,
    };

    return new ExecutionResult<HttpResult>(await documentExecuter.ExecuteAsync(opts));
});
```

Рис 3.25 – Результат виконання: згенерована конфігурація

В результаті можна запусити додаток та протестувати запит (рисунок 3.26).

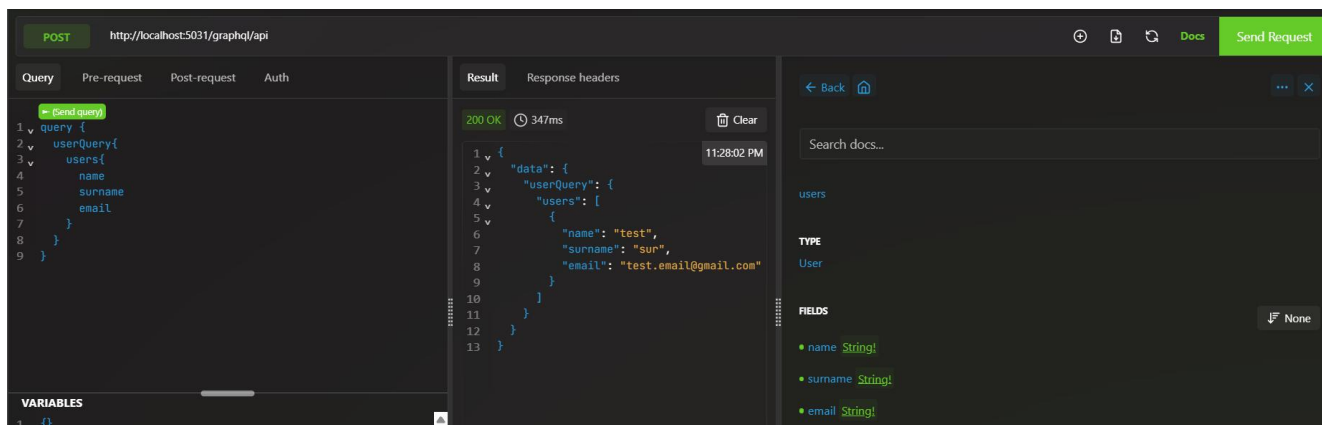


Рис 3.26 – Працюючий запит

Для бібліотеки GraphQL .NET також доступна команда генерації типів, базуючись на вже визначеній схемі. Для цього доступна команда «sparkql netgql generate types», що згенерує типи, основууючись на схемі, або команда «sparkql netgql generate s=config», що згенерує набір компонентів, що визначені у конфігураційному файлі.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		68

На рисунку 3.27 наведено приклад конфігураційного файлу, що описує основні компоненти, що потрібні для коректної генерації всього необхідного коду.

```
{
  "rootDir": ".",
  "projectNamespace": "GraphQLNetDemo",
  "generation": {
    "schemaPath": "./schema.graphql",
    "typesFolder": "GraphQL/Types",
    "queriesFolder": "GraphQL/Queries",
    "mutationsFolder": "GraphQL/Mutations",
    "rootQuery": "GraphQL/RootQuery",
    "rootMutation": "GraphQL/rootMutation",
    "appSchema": "GraphQL/AppSchema.cs"
  }
}
```

Рис 3.27 – Приклад конфігураційного файлу

Для визначеної схеми, зображеної на рисунку 3.28, після запуску команди (рисунк 3.29), буде згенеровано структура папок та файлів (рисунк 3.30)

```
appSchema.graphql
1 type Query {
2   books: [Book!]
3   authors: [Author!]
4 }
5
6 type Book {
7   title: String!
8   author: Author
9 }
10
11 type Author {
12   id: ID!
13   name: String!
14   books: [Book]
15 }
```

Рис 3.28 – Приклад graphql схеми

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql netgql generate s=config
-----Info-----
Schema found in ./schema
Generating AppSchema class
Generating RooQuery class
Generating RootMutation class

Generating models for entity Author
Generating queries for entity Author
Generating graph class definition for entity Author
Injecting Author too root objects

Generating models for entity Book
Generating queries for entity Book
Generating graph class definition for entity Book
Injecting Book too root objects
Components have been succesfully generated!
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> |
```

Рис 3.29 – Приклад виконання команди sparkql netgql generate s=config

Варто також зазначити, що структуру папок необов'язково задавати через конфігураційний файл, тоді будуть використовуватись стандартні значення.

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				69
Змн.	Арк.	№ докум.	Підпис	Дата		

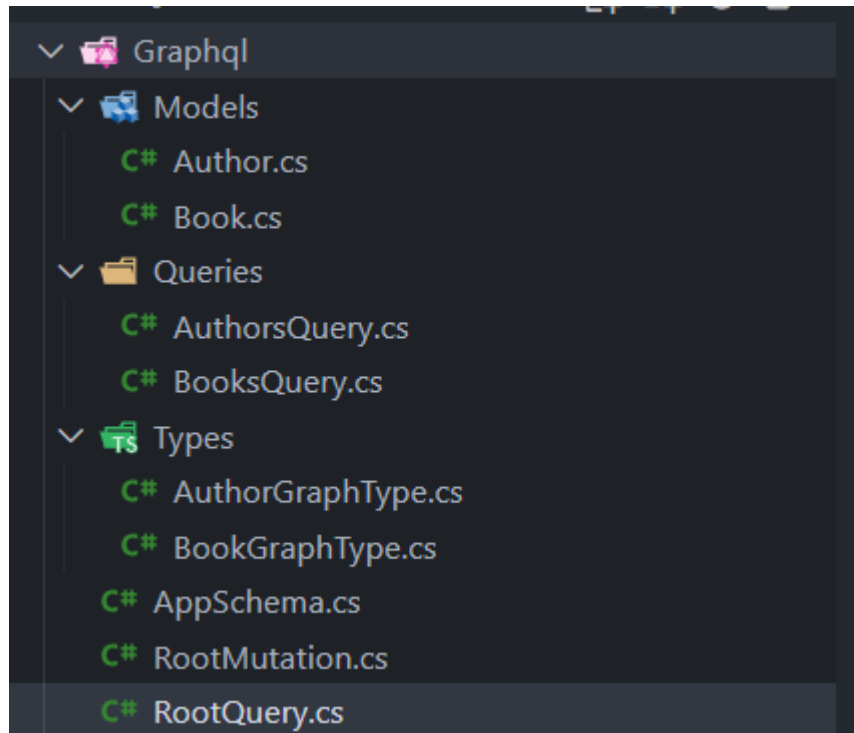


Рис 3.30 – Результат виконання: згенеровані папки та файли

Порядок роботи з бібліотекою «Hotchocolate»

Бібліотека HotChocolate є набагато простішою для використання, тому команд для цієї бібліотеки надається не таку велику кількість, як у випадку з «GraphQL.NET». Аналогічно до попереднього прикладу використання, першим кроком потрібно визначити конфігураційний, щоб CLI додаток мав змогу працювати з відносними шляхами, просторами імен та іншими важливими компонентами.

На відміну від «GraphQL.NET», HotChocolate не потребує визначення схеми, вона генерується автоматично на основі кореневого об'єкту. Тому першим кроком потрібно визначити цей об'єкт. Для цього доступна команда «sparkql hot root=query», що приймає два параметри: назву кореневого об'єкта, а також шлях до директорії, у яку потрібно згенерувати клас (рисунок 3.31).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql hot root=query n=RootQuery o=./Hot
-----Info-----
Generating RootQuery class
RootQuery has been succesfully generated!
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> |
```

Рис 3.31 – Приклад виконання команди sparkql hot root=query

В результаті виконання команди, буде згенеровано порожній клас, що зображено на рисунку 3.32

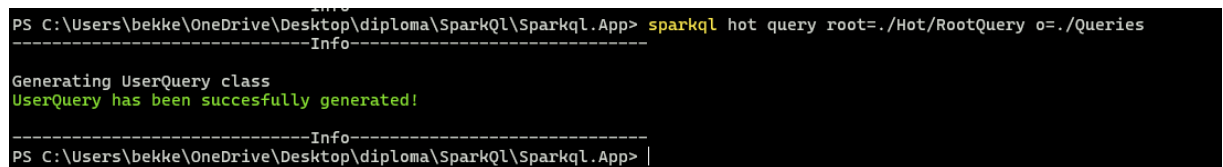
		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				70
Змн.	Арк.	№ докум.	Підпис	Дата		

```
public class RootQuery
{

}
```

Рис 3.32 – Результат виконання: згенерований клас

Всі запити можна визначати безпосередньо в згенерованому класі, але, якщо моделей або сутностей в проєкті велика кількість, то клас буде важко аналізувати, розширювати та розуміти. Тому бібліотека надає можливість розділяти запити шляхом розширення кореневого запиту, використовуючи атрибути. Для генерації такого запиту з розширенням, доступна команда «sparkql hot query», що приймає такі параметри як: шлях до кореневого об'єкту або його назву, простори імен (опціонально), директорія, в яку потрібно згенерувати клас (рисунок 3.33).



```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql hot query root=./Hot/RootQuery o=./Queries
Info-----
Generating UserQuery class
UserQuery has been succesfully generated!
Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> |
```

Рис 3.33 – Приклад виконання команди sparkql hot query

В результаті виконання команди, буде згенеровано клас, наведений на рисунку 3.34.

```
namespace HotNetDemo.Queries;

[ExtendObjectType(typeof(RootQuery))]
public class UsersQuery
{

}
```

Рис 3.34 – Результат виконання: згенерований клас з розширенням

Наступним кроком, як і у випадку з «Graphql .NET», потрібно провести конфігураційні дії. Для базової конфігурації доступна команда «sparkql hot setup» (рисунок 3.35), що приймає шлях до файлу, який містить конфігурацію додатку (опціонально, зазвичай це Program.cs).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql hot setup
-----Info-----
Updatiing Program.cs class with basic configuration...
Program.cs has been succesfully updated! /graphql endpoint is available
-----Info-----
```

Рис 3.35 – Приклад виконання команди sparkql hot setup

В результаті виконання комнади, буде згенеровано код, що зображено на рисунку 3.36.

```
using HotNetDemo;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddGraphQLServer()
    .AddQueryType<RootQuery>();

var app = builder.Build();

app.MapGraphQL();
app.Run();
```

Рис 3.36 – Результат виконання: згенерована базова конфігурація

На даному етапі згенеровано всі базові компоненти, тому можна протестувати їх роботу через вбудований клієнт Graphql (риунок 3.37).

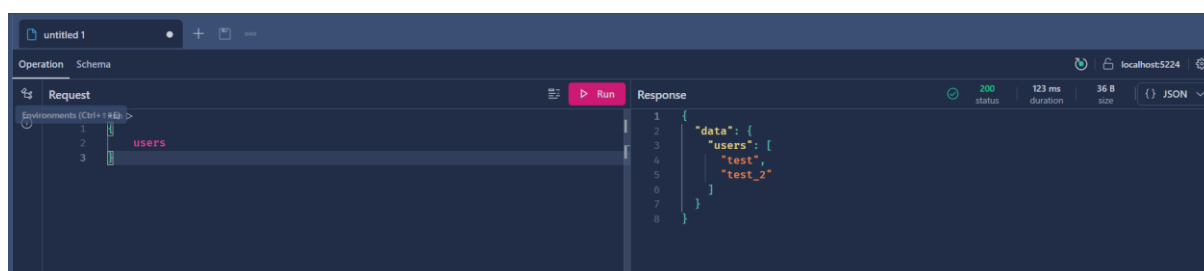


Рис 3.37 – Результати тестування

Порядок роботи з плагінами

CLI інструмент, надає змогу визначати власні плагіни, що будуть запускатися як команди. Створені плагіни можна вказати в конфігураційному файлі, або запустивши команду: «sparkql plugin», що приймає такі параметри як: шлях до реалізованого плагіну, а також всі необхідні параметри для запуску плагіну.

Для визначення власної команди, потрібно імплементувати інтерфейс `ICommandInvoker`, що вимагає реалізувати методи, що продемонстровані у таблиці 3.2
Опис методів та властивостей:

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		72

Опис методів та властивостей

Метод	Опис	Призначення
Name	Назва команди	Використовується для надання інформації про команди плагіну
Description	Опис команди	Використовується для надання інформації про ціль виконання команди
Execute	Метод для виконання команди	Метод, що виконує основну логіку команди
ParseArgs	Метод, для трансформації команди в об'єкт	Метод, що створює типізовані аргументи

Приклад реалізованого плагіну наведено на рисунку 3.38.

```
public class HelloWorldCommand : CliContext, ICommandInvoker<HelloWorldArgs>
{
    public string Name => "hello";

    public string Description => "Prints hello world";

    public void Execute(HelloWorldArgs args)
    {
        Console.WriteLine("Hello, world from {args.From}, {args.Message}");
    }

    public HelloWorldArgs ParseArgs(ArgsReader reader) => new(reader.GetOrDefault("from"), reader.GetOrDefault("msg"));
}
```

Рис 3.38 – Приклад реалізації

Команди можуть бути визначені у бібліотеці класів, або у файлі з розширенням .cs. Допускається реалізація будь якої кількості команд, основним критерієм є імплементація інтерфейсу ICommandInvoker.

Якщо конфігураційний файл містить шляхи до реалізованих плагінів, можна виконати команду «sparkql <назва команди> <параметри>», після чого команду буде автоматично знайдено та виконано. В результаті виконання команди, у будь якому вигляді, отримуємо результат, що наведено на рисунку 3.39.

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql plugin path=../Plugins/Impl from=California msg=", today here is too hot"
-----Info-----
Plugin found in ../Plugins/Impl
Scanning library for implemented commands
Implementation found
Running command: hello
Plugin says:
Hello, world from California, today here too hot
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App>
```

Рис 3.39 – Приклад виконання команди sparkql plugin

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				73
Змн.	Арк.	№ докум.	Підпис	Дата		

Також, щоб отримати доступ до контексту додатку, в якому розміщені корисні методи, а також інформація про контекст виконання команд, потрібно наслідуватись від класу CliContext. Після чого, у класі, через наслідування, з'явиться доступ до поля context (рисунок 3.40).

```
public class HelloWorldCommand : CliContext, ICommandInvoker<HelloWorldArgs>
{
    public string Name => "hello";

    public string Description => "Prints hello world";

    public void Execute(HelloWorldArgs args)
    {
        _context.Success($"Hello, world from {args.From}, {args.Message}");
        _context.Warn($"Hello, world from {args.From}, {args.Message}");
    }

    public HelloWorldArgs ParseArgs(ArgsReader reader) => new(reader.GetOrDefault("from"), reader.GetOrDefault("msg"));
}
```

Рис 3.40 – Приклад використання контексту

Порядок роботи з алгоритмом порівняння

CLI додаток надає змогу генерувати новий код базуючись на вже існуючому за допомогою порівняння файлів, використовуючи команду «sparkql save compare», що, як аргумент, приймає шлях до файлів, які потрібно порівняти та знайти різницю. Знайдена різниця буде записана у тимчасову директорію у вигляді метаданих, щоб при наступному запуску, вже нової команди «sparkql generate diff», згенерувати новий файл, основуючись на попередньому аналізі. Такий підхід забезпечує гнучкість інструменту у роботі з будь-якими архітектурами додатку (рисунок 3.41).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQl\Sparkql.App> sparkql save compare f1=./Queries/UserQuery f2=./Queries/CarsQuery
-----Info-----
Comparing files ./Queries/UserQuery with ./Queries/CarsQuery
Difference has been identified, please run command sparkql diff to generate new component
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQl\Sparkql.App> |
```

Рис 3.41 – Приклад виконання команди sparkql save compare

На рисунку 3.42 зображено, типовий код для роботи з даними, включаючи пагінацію, сортування для бібліотеки Hotchocolate.

Бібліотека Hotchocolate широко застосовує підхід з атрибубувати в мові програмування C#.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				74
Змн.	Арк.	№ докум.	Підпис	Дата		

Атрибути у C# – це спеціальна конструкція, яка використовується для зберігання додаткової інформації про програмні елементи (класи, методи, властивості, параметри тощо).

```
[ExtendObjectType(typeof(RootQuery))]
public class UsersQuery
{
    [UsePaging(IncludeTotalCount = true, DefaultPageSize = 10)]
    [UseFiltering]
    [UseSorting]
    [UseProjection]
    public IQueryable<User> GetUsers(AppDbContext context)
    {
        return context.Users;
    }

    [UsePaging(IncludeTotalCount = true, DefaultPageSize = 10)]
    [UseFiltering]
    [UseSorting]
    [UseProjection]
    public User? GetUserByIdAsync(Guid id, AppDbContext context)
    {
        var user = context.Users.FirstOrDefault(u => u.Id == id);
        if (user == null)
        {
            return null;
        }
        return user;
    }
}
```

Рис 3.42 – Приклад типового коду

У новому, такому ж класі, зміни будуть полягати тільки у назві сутності, тому достатньо запустити команду «sparkql generate diff», передавши такі параметри як: назва сутності, директорія для генерації нового класу, щоб отримати аналогічний код у новому файлі (рисунок 3.43).

```
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> sparkql generate diff entity=Company o=../Queries/CompanyQuery
-----Info-----
Trying to replace differences to Company
All changes have been applied, see file: ../Queries/CompanyQuery.cs
-----Info-----
PS C:\Users\bekke\OneDrive\Desktop\diploma\SparkQL\Sparkql.App> |
```

Рис 3.43 – Приклад виконання команди sparkql generate diff

На рисунку 3.44 зображено згенерований новий клас з відповідними замінами, що були передані за допомогою параметру entity. Алгоритм ідентифікує зміни, аналізує слова для заміни, визначає їх форму (множина, однина) й замінює їх на відповідник. У випадку з просторами імен, або з їх підключеннями – використовуються ті, що були зазначені в обох файлах при аналізі різниці.

```

[ExtendObjectType(typeof(RootQuery))]
public class CompanyQuery
{
    [UsePaging(IncludeTotalCount = true, DefaultPageSize = 10)]
    [UseFiltering]
    [UseSorting]
    [UseProjection]
    public IQueryable<Company> GetCompanies(AppDbContext context)
    {
        return context.Companies;
    }

    [UsePaging(IncludeTotalCount = true, DefaultPageSize = 10)]
    [UseFiltering]
    [UseSorting]
    [UseProjection]
    public Company? GetCompayByIdAsync(Guid id, AppDbContext context)
    {
        var company = context.Companies.FirstOrDefault(u => u.Id == id);
        if (company == null)
        {
            return null;
        }

        return company;
    }
}

```

Рис 3.44 – Результат виконання: згенерований код

Недоліком підходу є вимога до наявності як мінімум двох файлів з однаковим кодом, для виявлення різниці. Але основна перевага полягає у можливості генерації коду на основі будь-якого іншого, навіть виходячи за рамки graphql.

Отже, CLI інструмент надає змогу генерувати всі необхідні компоненти для роботи з graphql для двох цільових бібліотек «Hotchocolate» та «GraphQL .NET», а також підтримує систему плагінів та генерацію нового коду на основі вже існуючого.

3.3 Тестування роботи системи

Протягом розробки, кожен окремий компонент або логічно завершена частина функціоналу супроводжувалась unit-тестуванням [29], що забезпечувало раннє виявлення помилок, спрощувало рефакторинг і підвищувало загальну стабільність розробленого рішення [30]. Було протестовано лише основні компоненти програми та деякі команди, що є важливими для додатку.

Загальна кількість юніт тестів дорівнює 44 (рисунки 3.45).

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		76

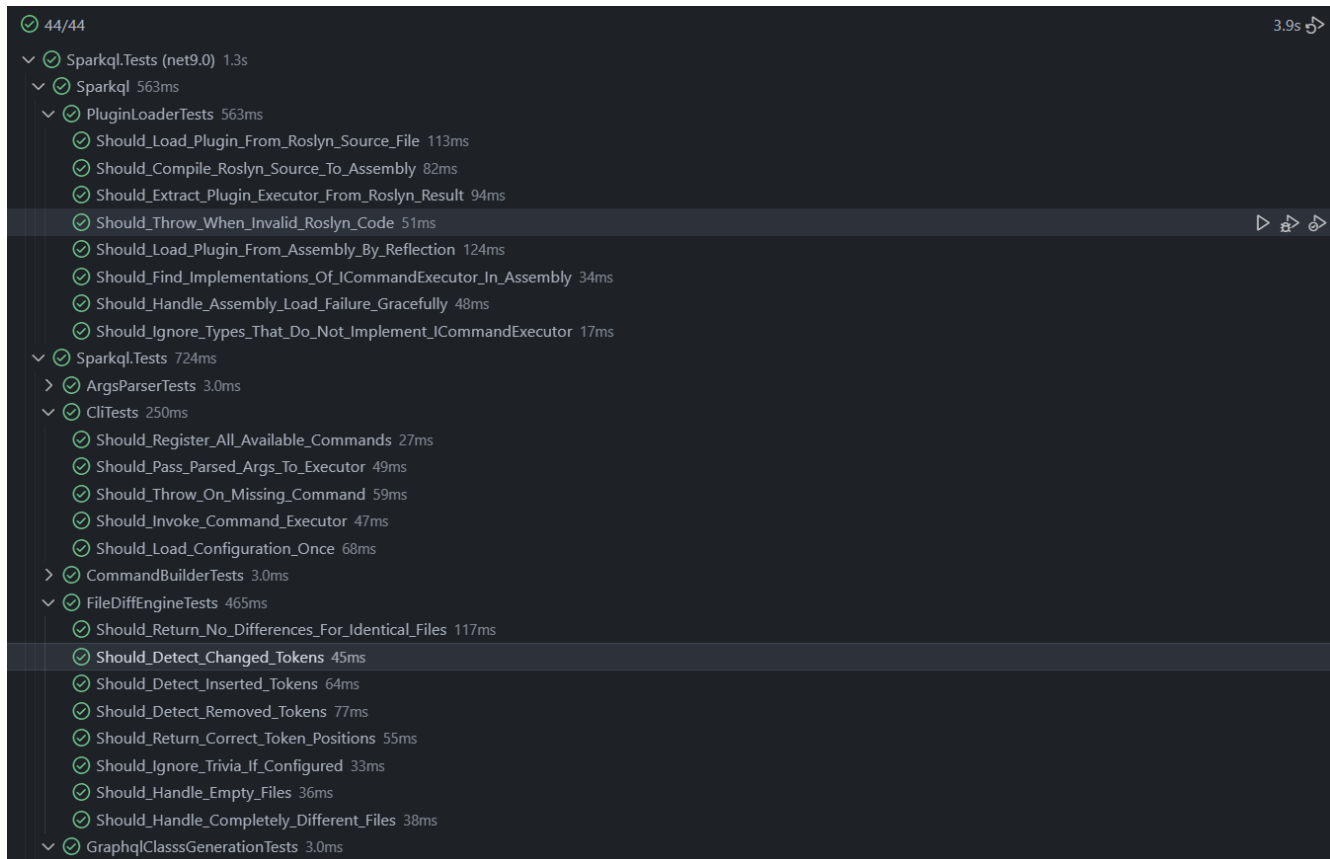


Рис 3.45 – Unit тести

У таблиці 3.3. описано тести та їх призначення для класу CliCore.

Таблиця 3.3

Опис реалізованих тестів для класу CliCore

Тест	Опис	Крайні випадки
Should_Register_All_Available_Commands	Перевіряє, чи всі доступні команди були зареєстровані в контейнері/реєстрі.	Дублікат команд
Should_Pass_Parsed_Args_To_Executor	Перевіряє, чи парсинг аргументів передається конкретному виконувачу команди.	-Аргументи у неправильному форматі - Аргументи не відповідають типу TArgs
Should_Throw_On_Missing_Command	Генерується виняток, при відсутності команди.	Невідома команда
Should_Invoke_Command_Executor	Перевіряє, що знайдену команду викликає через відповідний метод	Метод Execute не реалізовано
Should_Invoke_Command_Executor	Перевіряє, що знайдену команду викликає через відповідний метод	Метод Execute не реалізовано
Should_Load_Configuration_Once	Перевіряє, що конфігурація завантажується лише один раз при старті програми.	- Відсутній файл конфігурації -Пошкоджений формат

Продовження таблиці 3.3

Should_Load_Configuration_Once	Перевіряє, що конфігурація завантажується лише один раз при старті програми.	- Відсутній файл конфігурації - Пошкоджений формат
--------------------------------	--	---

У таблиці 3.4. описано тести та їх призначення для класу PluginLoader.

Таблиця 3.4

Опис реалізованих тестів для класу PluginLoader

Тест	Опис	Крайні випадки
Should_Load_Plugin_From_Roslyn_Source_File	Перевіряє, чи завантажується плагін з вихідного cs-файлу за допомогою Roslyn	Файл не існує
Should_Compile_Roslyn_Source_To_Assembly	Перевіряє, що вихідний код успішно компілюється в збірку (in-memory)	Синтаксичні помилки у коді
Should_Compile_Roslyn_Source_To_Assembly	Перевіряє, що вихідний код успішно компілюється в збірку (in-memory)	Синтаксичні помилки у коді
Should_Extract_Plugin_Executor_From_Roslyn_Result	Перевіряє, чи можливо отримати екземпляр класу, що реалізує ICommandExecutor, із збірки	Клас не реалізує інтерфейс
Should_Throw_When_Invalid_Roslyn_Code	Перевіряє, що при некоректному C#-коді виникає виняток або відповідна помилка	Ключові слова переплутані
Should_Load_Plugin_From_Assembly_By_Reflection	Перевіряє завантаження плагіна зі скомпільованої DLL-збірки через рефлексію.	Збірка пошкоджена
Should_Find_Implementations_Of_ICommandExecutor_In_Assembly	Перевіряє, чи правильно знаходяться всі реалізації інтерфейсу ICommandExecutor<T>	Абстрактні класи
Should_Handle_Assembly_Load_Failure_Gracefully	Перевіряє, що програма не аварійно завершується при помилці завантаження збірки.	Неправильний шлях до DLL
Should_Ignore_Types_That_Do_Not_Implement_ICommandExecutor	Перевіряє, що типи, які не реалізують потрібний інтерфейс, не потрапляють до реєстру.	Інші публічні класи в збірці

У таблиці 3.5. описано тести та їх призначення для класу ArgumentParser.

Таблиця 3.5

Опис реалізованих тестів для класу ArgumentParser

Тест	Опис	Крайні випадки
Should_Throw_If_Args_Are_Empty	Перевіряє, що виняток генерується при порожніх аргументах.	Порожній масив

Продовження таблиці 3.5

Should_Parse_CommandName _From_Single_Arg	Виділяє командне ім'я, якщо аргумент без '='.	Аргумент із зайвими пробілами
Should_Parse_One_KeyValue_Argument	Перевіряє розбір одного параметра у форматі key=value.	Порожній ключ або значення
Should_Parse_Multiple_KeyValue_Arguments	Перевіряє розбір кількох аргументів key=value.	Повторювані ключі
Should_Ignore_Arguments _With_More_Than_One_Equals	Перевіряє ігнорування аргументів з кількома «=».	Коректні значення зі спецсимволами
Should_Not_Throw_For_Valid_Args	Перевіряє, що валідні аргументи не спричиняють помилок.	Нетипові символи
Should_Handle_Args_With_Empty_Value	Перевіряє підтримку параметрів із порожнім значенням.	- key= (відсутнє значення)

У таблиці 3.5. описано тести та їх призначення для класу GraphQLClassGenerator.

Таблиця 3.6

Опис реалізованих тестів для класу GraphQLClassGenerator

Тест	Опис	Крайні випадки
Should_Generate_ObjectGraphType _For_Given_Model	Перевіряє, що клас GraphQL створюється на основі моделі.	Модель з наслідуванням
Should_Generate_Field _For_Each_Public_Property	Перевіряє створення полів	Властивості без get
Should_Use_Correct_Field_Types	Перевіряє відповідність типів GraphQL і C#.	Nullable-типи
Should_Skip_Private _Or_Ignored_Properties	Перевіряє, що приватні або ігноровані поля не враховуються.	[JsonIgnore] або custom attribute
Should_Handle_Collections _As_Lists_Of_Types	Перевіряє обробку колекцій як списків GraphQL-типів.	Масиви та IEnumerable
Should_Generate_Valid_CSharp_Syntax	Перевіряє, що результат компілюється як коректний C#.	Спеціальні символи у назвах
Should_Generate_Expected _Type_Name_Suffix	Перевіряє додавання суфіксу (наприклад, GraphType).	Конфлікт імен

У таблиці 3.7. описано тести та їх призначення для класу CommandBuilder.

Таблиця 3.7

Опис реалізованих тестів для класу `CommandBuilder`

Назва тесту	Опис основного сценарію	Крайні випадки
<code>Should_Create_Command_With_Expected_Values</code>	Перевіряє створення команди з усіма параметрами.	Відсутній опис або executor
<code>Should_Assign_CommandExecutor_Instance</code>	Перевіряє правильне створення інстансу виконавця команди.	Executor без конструктора
<code>Should_Fail_If_ArgumentsFactory_Not_Set</code>	Перевіряє, що без фабрики аргументів виникає помилка.	Null-значення factory
<code>Should_Allow_Method_Chaining</code>	Перевіряє, що методи повертають <code>this</code> (Fluent API).	Порушення ланцюга викликів
<code>Should_Set_Name_Correctly</code>	Перевіряє встановлення імені команди.	Порожній або null рядок
<code>Should_Set_Description_Correctly</code>	Перевіряє встановлення опису команди.	Надто довгий опис
<code>Should_Create_Command_Using_Static_Factory</code>	Перевіряє побудову команди через <code>CreateCommand</code> .	Один із параметрів не задано
<code>Should_Throw_If_Build_Called_Before_Handler</code>	Перевіряє поведінку <code>Build()</code> без <code>Handler</code> .	Виняток або null у виконавці

У таблиці 3.8. описано тести та їх призначення для класу `CodeComparer`.

Таблиця 3.8

Опис реалізованих тестів для класу `CodeComparer`

Назва тесту	Опис основного сценарію	Крайні випадки
<code>Should_Return_No_Differences_For_Identical_Files</code>	Перевіряє, що при однакових файлах немає різниць.	Різниця лише в пробілах чи коментарях
<code>Should_Detect_Changed_Tokens</code>	Фіксує змінені токени у файлах.	Зміна назви змінної
<code>Should_Detect_Inserted_Tokens</code>	Визначає додані нові токени у другому файлі.	Вставка нового методу
<code>Should_Detect_Removed_Tokens</code>	Визначає видалені токени.	Видалення ключового слова
<code>Should_Return_Correct-Token_Positions</code>	Перевіряє правильне позиціонування змінених токенів.	Зміни в межах одного рядка
<code>Should_Ignore_Trivia_If_Configured</code>	Ігнорує пробіли, табуляції та коментарі.	Ввімкнено опцію <code>ignore trivia</code>
<code>Should_Handle_Empty_Files</code>	Перевіряє поведінку при порожніх вхідних файлах.	Один або обидва файли порожні
<code>Should_Handle_Completely_Different_Files</code>	Перевіряє реакцію на повністю різні файли.	Повна відсутність спільних токенів

Висновки до третього розділу

У даному розділі було представлено порядок інсталяції та налаштування системи, наведено приклади типових сценаріїв використання, а також визначено мінімальні вимоги до її використання. Розглянуто структуру інтерфейсу та описано послідовність роботи з консольним застосунком із демонстрацією основних команд. Крім того, проведено тестування розробленого додатку за допомогою unit-тестів, що дозволило перевірити коректність реалізованого функціоналу та підтвердити відповідність програмного продукту визначеним вимогам.

У процесі тестування було проаналізовано різні сценарії роботи системи, зокрема обробку як коректних, так і некоректних вхідних даних. Це дало змогу оцінити надійність та стійкість додатку до можливих помилок користувача. Результати перевірки підтвердили, що система здатна стабільно функціонувати в умовах практичної експлуатації, а також може бути масштабована та вдосконалена у разі подальшого розвитку.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		81

ВИСНОВКИ

В рамках кваліфікаційної роботи бакалавра, було спроектовано та реалізовано модуль для роботи GraphQL під платформу .NET у вигляді CLI-додатку, задля чого було проведено ряд досліджень.

В першому розділі роботи було визначено основні поняття, окреслена проблематика, проаналізовані аналоги програмного продукту, що дозволило виявити основні переваги й недоліки щодо реалізації додатку. Також під час аналізу аналогів системи за тематикою кваліфікаційної роботи, було окреслено критерії успішної реалізації додатку, додатково було визначено переваги системи, розробленої в рамках кваліфікаційної роботи: гнучкість за допомогою системи плагінів, наявність команд для найпопулярніших бібліотек, алгоритм порівняння та генерації файлів, можливість використовувати конфігураційні файли.

В другому розділі було спроектовано загальний алгоритм роботи програми, визначено об'єктно-орієнтовану модель системи. Під час проектування консольного додатку було досліджено різноманітні підходи до розробки схожих систем, визначено алгоритми, що будуть основою функціоналу. На завершальному етапі було реалізовано та опубліковано консольний додаток як інструмент для платформи .NET

В третьому розділі було детально та покроково описано роботу з реалізованим додатком, розглянуто основні особливості та можливості. Також в цьому розділі було проведено тестування за допомогою юніт тестів, виявлення та виправлення помилок й відлагодження усіх компонентів системи. Тестування показало, що додаток успішно виконує свої функції, є надійним та високоякісним.

В цілому, отримані результати підтверджують практичну цінність розробленого інструменту та його потенціал для подальшого розвитку.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		82

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Офіційний сайт GraphQL [Електронний ресурс] – Режим доступу до ресурсу: <https://graphql.org/learn/>.
2. GraphQL .NET Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://graphql-dotnet.github.io>.
3. Dapper Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://www.learndapper.com/>.
4. Entity Framework Core Documentation [Електронний ресурс] – <https://learn.microsoft.com/en-us/ef/>.
5. Hotchocolate Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://chillicream.com/docs/>.
6. Офіційний сайт Symfony [Електронний ресурс] - Режим доступу до ресурсу: <https://symfony.com/>.
7. Офіційний сайт Laravel [Електронний ресурс] - Режим доступу до ресурсу: <https://laravel.com/>.
8. Офіційний сайт Angular [Електронний ресурс] - Режим доступу до ресурсу: <https://angular.dev/>.
9. Офіційний сайт NestJS [Електронний ресурс] - Режим доступу до ресурсу: <https://angular.dev/https://nestjs.com/>.
10. GraphQL Codegen Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://the-guild.dev/graphql/codegen/docs/>.
11. Prisma Typegraphql Documentation [Електронний ресурс] - Режим доступу до ресурсу: <https://prisma.typegraphql.com/docs>.
12. Apollo Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://www.apollographql.com/docs/>.
13. GraphQL-cli documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://github.com/Urigo/graphql-cli/tree/master/docs>.
14. GraphQL – nexus [Електронний ресурс] – Режим доступу до ресурсу: <https://nexusjs.org/>.

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савицький Р.С.				83
Змн.	Арк.	№ докум.	Підпис	Дата		

15. GraphQL Specification [Електронний ресурс] – Режим доступу до ресурсу: <https://spec.graphql.org/October2021/>.
16. Robert C. Martin, Joseph Albahari. Functional Design: Principles, Patterns, and Practices, 2023
17. Refactoring Guru [Електронний ресурс] – Режим доступу до ресурсу: <https://refactoring.guru/>.
18. Joseph Albahari. C# 9.0 in a Nutshell, 2021.
19. Roslyn API Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/>.
20. Microsoft Docs. Reflection in .NET [Електронний ресурс] – Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection>.
21. Santosh Singh. Create Your Own Compiler in C# Using ANTLR: A Practical Guide, 2021.
22. Fundamental Data Structures and Algorithms in C# [Електронний ресурс] – Режим доступу до ресурсу: <https://dev.to/adavidoaiei/fundamental-datastructures-and-algorithms-in-c-4ocf>.
23. Pearson, Aho, Lam, Sethi, Ullman. Compilers: Principles, Techniques, and Tools, 2023.
24. Grune, D., Jacobs, C. Parsing Techniques: A Practical Guide. Springer, 2008
25. ECMA International. ECMA-334: C# Language Specification [Електронний ресурс] – Режим доступу до ресурсу: <https://www.ecma-international.org/publications-and-standards/standards/ecma-334/>.
26. Working with Files & Directories in C# [Електронний ресурс] - Режим доступу до ресурсу: <https://www.tutorialsteacher.com/csharp/csharp-file>.
27. Harold Abelson, Gerald Jay Sussman, Julie Sussman. Structure and Interpretation of Computer Programs, 2022.
28. Robert C. Martin. Clean Craftsmanship: Disciplines, Standards, and Ethics, 2021.
29. xUnit Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://xunit.net/docs/>.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		84

30. Unit testing best practices [Електронний ресурс] – Режим доступу до ресурсу:
<https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices>.

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		85

ДОДАТКИ

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		86

Лістинг реалізації команди для генерації класу що описує graphql тип на основі вже існуючого класу:

```
public class GraphQLClassGenerator
{
    public const string GraphQLTypePostfix = "GraphType";

    private readonly GraphQLNetBuilder _builder;

    public GraphQLClassGenerator()
    {
        _builder = new GraphQLNetBuilder();
    }

    public CompilationUnitSyntax? CreateCompilationUnit(string code, GenerateGraphQLClassArgs args)
    {
        {
            var tree = CSharpSyntaxTree.ParseText(code);
            var descriptor = ConstructClass(tree.GetCompilationUnitRoot(), args);
            if (descriptor.IsEmpty)
                return null;

            if (descriptor.ClassDeclaration == null)
                return null;

            return CompilationUnit()
                .AddUsings(_builder.GetUsings(descriptor).ToArray())
                .AddMembers(descriptor.ClassDeclaration)
                .NormalizeWhitespace();
        }

        private ClassDeclarationDescriptor ConstructClass(
            CompilationUnitSyntax? root,
            GenerateGraphQLClassArgs args
        )
        {
            {
                if (root == null)
                    return ClassDeclarationDescriptor.Empty;

                foreach (var classDecl in root.DescendantNodes().OfType<ClassDeclarationSyntax>())
                {
                    if (!string.IsNullOrEmpty(args.ClassName) && classDecl.Identifier.Text != args.ClassName)
                        continue;

                    return CreateClassDescriptor(classDecl);
                }

                return ClassDeclarationDescriptor.Empty;
            }
        }
    }
}
```

		Беккер М.М.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		87

```
private ClassDeclarationDescriptor CreateClassDescriptor(ClassDeclarationSyntax classDecl)
{
    var declaredClassName = classDecl.Identifier.Text;
    var graphqlClassName = $"{declaredClassName}{GraphQLTypePostfix}";
    return new ClassDeclarationDescriptor(
        InheritGenericClass(
            declaredClassName,
            ConstructGraphQLClass(
                graphqlClassName,
                _builder.MakeConstructor(graphqlClassName, MakeFields(classDecl).ToArray())
            )
        ),
        GetNamespace(classDecl)
    );
}
```

```
private ClassDeclarationSyntax InheritGenericClass(
    string genericParameter,
    ClassDeclarationSyntax classDeclaration
)
{
    return classDeclaration.WithBaseList(
        BaseList(
            SingletonSeparatedList<BaseTypeSyntax>(
                SimpleBaseType(_builder.MakeGenericType(genericParameter))
            )
        )
    );
}
```

```
private ClassDeclarationSyntax ConstructGraphQLClass(
    string className,
    ConstructorDeclarationSyntax constructor
)
{
    return ClassDeclaration(className)
        .AddModifiers(Token(SyntaxKind.PublicKeyword))
        .AddMembers(constructor);
}
```

```
private string GetNamespace(ClassDeclarationSyntax classDecl)
{
    var fileScopedNamespace = TryGetNamespace<BaseNamespaceDeclarationSyntax>(
        classDecl,
        (syntax) => syntax != null ? syntax.Name.ToString() : string.Empty
    );

    if (fileScopedNamespace != string.Empty)
        return fileScopedNamespace;

    return TryGetNamespace<NamespaceDeclarationSyntax>(
```

		Беккер М.М.			ІІЗ.КР.Б – 121 – 25 – ІІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		88

```

        classDecl,
        (syntax) => syntax != null ? syntax.Name.ToString() : string.Empty
    );
}

private string TryGetNamespace<TDeclarationSyntax>(
    ClassDeclarationSyntax classDecl,
    Func<TDeclarationSyntax?, string> getString
)
{
    var exists = classDecl.Ancestors().OfType<TDeclarationSyntax>().FirstOrDefault();
    if (exists != null)
        return getString(exists);

    return string.Empty;
}

private IEnumerable<ExpressionStatementSyntax> MakeFields(ClassDeclarationSyntax classDecl)
{
    foreach (var prop in classDecl.Members.OfType<PropertyDeclarationSyntax>())
    {
        yield return _builder.MakeField(_builder.MakeLambda(prop.Identifier.Text));
    }
}
}

```

Лісінг реалізації команди для реєстрації типу в схему:

```

public class RegisterGraphTypeCommand : ICommandExecutor<RegisterGraphTypeArgs>
{
    public const string GraphqlNetRegisterTypeMapping = "RegisterTypeMapping";

    public void Execute(RegisterGraphTypeArgs args)
    {
        if (!File.Exists(args.SchemaPath))
        {
            Console.WriteLine("Schema file does not exists");
            return;
        }
        var code = File.ReadAllText(args.SchemaPath);
        var tree = CSharpSyntaxTree.ParseText(code);
        var root = tree.GetCompilationUnitRoot();
        var updatedClass = GetUpdatedClass(root, args);
        if (updatedClass == null)
        {
            Console.WriteLine("Cannot update classes properly, check validity of the provided file");
            return;
        }

        var withUsings = AddUsingsIfNecessary(root, updatedClass, args);
        File.WriteAllText(args.SchemaPath, withUsings.NormalizeWhitespace().ToFullString());
    }
}

```

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ІІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		89

```

private CompilationUnitSyntax AddUsingsIfNecessary(
    CompilationUnitSyntax root,
    CompilationUnitSyntax updatedClass,
    RegisterGraphTypeArgs args
)
{
    if (HasNecessaryUsings(root, args))
        return updatedClass;

    return updatedClass.AddUsings(
        UsingDirective(IdentifierName(args.ClrTypeNamespace)),
        UsingDirective(IdentifierName(args.GraphTypeNamespace))
    );
}

private CompilationUnitSyntax? GetUpdatedClass(
    CompilationUnitSyntax root,
    RegisterGraphTypeArgs args
)
{
    var classDeclaration = GetClassDeclaration(root, args);
    if (classDeclaration == null)
    {
        Console.WriteLine($"Cannot find {args.SchemaClassName} declaration");
        return null;
    }

    var registerContainer = FindRegisterContainer(classDeclaration, args);
    if (registerContainer == null)
    {
        Console.WriteLine($"Cannot find register method: {args.RegisterMethod}");
        return null;
    }

    var memberAccess = MakeMathodAccess(MakeGeneriMethod(args));
    var registerMethodInvocation = MakeRegsisterMethodInvokation(memberAccess);
    var index = GetLastRegisterInvokationIndex(registerContainer);
    if (index == -1)
    {
        Console.WriteLine("Not found any registered types");
        return null;
    }

    var updatedBody = GetUpdatedBody(registerContainer, index, registerMethodInvocation);
    var updatedMethod = registerContainer.WithBody(updatedBody);
    var updatedClass = classDeclaration.ReplaceNode(registerContainer, updatedMethod);
    return root.ReplaceNode(classDeclaration, updatedClass);
}

```

```

private ClassDeclarationSyntax? GetClassDeclaration(CompilationUnitSyntax root,
RegisterGraphTypeArgs args)
{
    if (string.IsNullOrEmpty(args.SchemaClassName))
    {
        return root.DescendantNodes()
            .OfType<ClassDeclarationSyntax>()
            .FirstOrDefault();
    }

    return root.DescendantNodes()
        .OfType<ClassDeclarationSyntax>()
        .FirstOrDefault(c => c.Identifier.Text == args.SchemaClassName);
}

private MethodDeclarationSyntax? FindRegisterContainer(ClassDeclarationSyntax classDeclaration,
RegisterGraphTypeArgs args)
{
    return classDeclaration.Members
        .OfType<MethodDeclarationSyntax>()
        .FirstOrDefault(m => m.Identifier.Text == args.RegisterMethod);
}

private MemberAccessExpressionSyntax MakeMathodAccess(GenericNameSyntax genericName)
{
    return MemberAccessExpression(
        SyntaxKind.SimpleMemberAccessExpression,
        ThisExpression(),
        genericName
    );
}

private GenericNameSyntax MakeGeneriMethod(RegisterGraphTypeArgs args)
{
    return GenericName(Identifier(GraphqlNetRegisterTypeMapping))
        .WithTypeArgumentList(
            TypeArgumentList(SeparatedList<TypeSyntax>(new SyntaxNodeOrToken[]
            {
                IdentifierName(args.ClrTypeName),
                Token(SyntaxKind.CommaToken),
                IdentifierName(GetGraphTypeName(args))
            })))
        );
}

static string GetGraphTypeName(RegisterGraphTypeArgs args)
{
    return string.IsNullOrEmpty(args.GraphTypeName)
        ? $"{{args.ClrTypeName}}GraphType"
        : args.GraphTypeName;
}

```

		Беккер М.М.			ІІЗ.КР.Б - 121 - 25 - ІІЗ	Арк.
		Савіцький Р.С.				91
Змн.	Арк.	№ докум.	Підпис	Дата		

```

private ExpressionStatementSyntax
MakeRegisterMethodInvocation(MemberAccessExpressionSyntax memberAccess)
{
    return ExpressionStatement(
        InvocationExpression(memberAccess)
            .WithArgumentList(ArgumentList())
    );
}

private int GetLastRegisterInvocationIndex(MethodDeclarationSyntax method)
{
    if (method.Body == null)
        return -1;

    var invocations = method.Body
        .DescendantNodes()
        .OfType<InvocationExpressionSyntax>()
        .ToArray();

    for (int i = invocations.Length - 1; i >= 0; i--)
    {
        var invocation = invocations[i];
        if (invocation.Expression is MemberAccessExpressionSyntax memberAccess &&
            memberAccess.Expression is ThisExpressionSyntax &&
            memberAccess.Name is GenericNameSyntax genericName &&
            genericName.Identifier.Text == GraphQLNetRegisterTypeMapping)
        {
            return i + 1;
        }
    }

    return 0;
}

private BlockSyntax? GetUpdatedBody(
    MethodDeclarationSyntax registerContainer,
    int index,
    ExpressionStatementSyntax registerMethodInvocation
)
{
    if (registerContainer.Body == null)
        return null;

    return registerContainer.Body.WithStatements(
        registerContainer.Body.Statements.Insert(index, registerMethodInvocation)
    );
}

private bool HasNecessaryUsings(CompilationUnitSyntax root, RegisterGraphTypeArgs args)
{
    return root.DescendantNodes()
        .OfType<UsingDirectiveSyntax>()

```

		Беккер М.М.			ІІЗ.КР.Б - 121 - 25 - ІІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		92

```

        .Any(u => u.ToFullString().Contains(args.ClrTypeNamespace) ||
            u.ToFullString().Contains(args.GraphTypeNamespace)
        );
    }
}

```

		Беккер М.М.			ІПЗ.КР.Б - 121 - 25 - ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		93