Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи і засоби автогенерації та аналізу вихідного коду програмного забезпечення

(тема)

Виконав:

студент II курсу, групи СПм-20-2

Носов В.С.
(прізвище, ініціали)

Спеціальність

123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма

Системне програмування
(повна назва освітньої програми)

Керівник: доц. Федорченко В.М.
(посада, прізвище, ініціали)

Допускається до захисту

В.о. зав. кафедри ЕОМ _____ Волк М.О.
(підпис) (прізвище, ініціали)

2022 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Носову Владиславу Сергійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Методи і засоби автогенерації та аналізу вихідного коду програмного забезпечення

затверджена наказом по університету від “ 24 ” березня 2022 р. № 413 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 18 травня 2022 р.

3. Вхідні дані до роботи
   1) літературні джерела
   2) матеріали практики
   3) платформа .NET Core

4. Перелік питань, що потрібно опрацювати у роботі
   1) аналіз предметної галузі та актуалізація рішень
   2) вибір інструментальних засобів розробки
   3) архітектура та проєктування програмного забезпечення
   4) опис прийнятих рішень

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Слайд-презентація – 16 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі, постановка задачі	28.03.2022-07.04.2022	
2	Огляд сучасних технологій генерації коду	07.04.2022-15.04.2022	
3	Проектування програмного продукту	15.04.2022-30.04.2022	
4	Перевірка чернетки кваліфікаційної роботи		
	та внесення змін до неї керівником	30.04.2022-06.05.2022	
5	Оформлення кваліфікаційної роботи	06.05.2022-13.05.2022	
6	Підготовка презентації та доповіді	13.05.2022-15.05.2022	
7	Рецензування роботи	15.05.2022-18.05.2022	

Дата видачі завдання 28 березня 2022 р.

Студент _____
(підпис)

Керівник роботи _____ доц. Федорченко В.М.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 75 с., 16 рисунок, 2 дод., 30 джерел.

C#, ГЕНЕРАЦІЯ, ПАРСИНГ, .NET, ROSLYN, CODEDOM, T4 TEMPLATES, BOILERPLATE CODE, GRAPHQL.

Метою кваліфікаційної роботи є аналіз та порівняння існуючих засобів для генерації вихідного коду, а також створення власного генератора вихідного коду. Методи розробки базуються на інструментах розробки консольних, десктопних та веб-застосувань на платформі .NET.

У ході виконання кваліфікаційної роботи проведено огляд предметної області, а також проведено аналіз можливостей існуючих інструментів для генерації коду. Визначено вимоги до основних функціональних можливостей, які має забезпечити розроблений генератор коду.

Основним завданням, яке вирішується є створення власного допоміжного інструменту для спрощення розробки веб-додатку, системи керування вмістом, вихідний код якого містив би багато шаблонного коду.

В результаті було проаналізовано перспективи застосування інструментів генерації вихідного коду: CodeDOM, IL injecting, T4 templates, snippets, Roslyn based source generators. Створено допоміжний інструмент для автоматичної генерації коду у вигляді консольного застосування. Генератор коду було реалізовано в середовищі Visual Studio 2022 за допомогою мови програмування C#.

ABSTRACT

Master’s thesis: 75 pages, 16 figures, 2 appendices, 30 sources.

C#, ГЕНЕРАЦІЯ, ПАРСИНГ, .NET, ROSLYN, CODEDOM, T4 TEMPLATES, BOILERPLATE CODE, GRAPHQL.

The major goal of this thesis is the analysis and comparison of existing tools for generating source code, as well as creating own source code generator. Development methods are based on tools for developing console, desktop and web applications on the .NET platform.

In the course of the attestation work, a review of the subject area was conducted, as well as an analysis of the possibilities of existing tools for code generation. The requirements to the basic functionalities which the developed code generator should provide are defined.

The main task to be solved is to create your own auxiliary tool to simplify the development of the web application, the source code of which would contain a lot of template code.

As a result, the prospects of using source code generation tools were analyzed: CodeDOM, IL injecting, T4 templates, snippets, Roslyn based source generators. A console application has been created as an auxiliary tool for automatic code generation. The code generator was implemented in Visual Studio 2022 using the C # programming language.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AST – абстрактне синтаксичне дерево (англ., Abstract Syntax Tree)

API – програмний інтерфейс додатків (англ., Application Programming Interface)

CMS – система керування вмістом (англ., content management system)

IL – проміжна мова (англ., Intermediate Language)

AOP – аспектно-орієнтоване програмування (англ., Aspect-oriented programming)

IDE – інтегроване середовище розробки (англ., Integrated Development Environment)

SDK – комплект для розробки програмного забезпечення (англ., software development kit)

GAC – глобальний кеш збірок (англ., Global Assembly Cache)

UML – уніфікована мова моделювання (англ., Unified Modeling Language)

DSL – мова, специфічна для предметної області (англ., domain-specific language)

T4 – Text Template Transformation Toolkit

XML – розширювана мова розмітки (англ., eXtensible Markup Language)

AOT – компілятор перед виконанням (англ. ahead-of-time compilation)

ВСТУП

Генерація коду не нова концепція, вона завжди була доступна в .NET. Генератор вихідного коду, за визначенням Microsoft – це «частина коду, яка виконується під час компіляції та може перевірити програму для створення додаткових файлів, які компілюються разом із рештою коду». Читання вмісту поточної компіляції означає пошук у синтаксичних деревах, для знаходження вузлів, маркерів та символів для певних умов. Якщо вони існують, створюється новий код C#, який включається як частина процесу компіляції.

Генерація коду дуже потужна і добре вивчена техніка, але багато розробників все ще неохоче її використовують. Нещодавно з'явилася можливість генерувати код C#, використовуючи сам компілятор – це робиться за допомогою API бібліотек Roslyn/CodeAnalisys, що надають, крім усього іншого, ще й парсинг, обхід і генерацію вихідного коду. Парсинг або синтаксичний розбір – це акт читання тексту і перетворення його в більш корисний формат в пам'яті [1].

При генерації коду генератор пишеться один раз, і використовується стільки разів, скільки потрібно. Надання певних вхідних даних для генератора та його виклик значно швидше, ніж написання коду вручну, тому генерація коду дозволяє заощадити час. За допомогою генерацією коду завжди отримується очікуваний код. Такий код буде завжди працювати так, як очікується, звичайно, за винятком помилок у генераторі.

Метою даної роботи є дослідження існуючих засобів для автоматичної генерації коду, знаходження їх сильних та слабких сторін, та розробка власного інструмента генерації, який би пришвидшив розробку веб-додатку, що являє собою систему керування вмістом з API у вигляді GraphQL.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА АКТУАЛІЗАЦІЯ РІШЕНЬ

1.1 Аналіз предметної галузі

Написання коду включає кілька (в основному складних) кроків – розбиття процесу або явища на чіткі і послідовні інструкції, управління вхідними і вихідними буферами, управління пам'яттю і так далі. З розвитком індустрії процес написання коду став простішим – були створені бібліотеки та фреймворки для загальних завдань (робота з файлами певних форматів, веб-сервери, математичні розрахунки), об'єктно-орієнтовані мови програмування для полегшення моделювання і т.д.

Генерація коду або кодогенерація – частина процесу компіляції, коли спеціальна частина компілятора, кодогенератор, конвертує синтаксично коректну програму в послідовність інструкцій, які можуть виконуватися на машині. При цьому можуть застосовуватися різні, в першу чергу машинно-залежні оптимізації. Часто кодогенератор є спільною частиною для багатьох компіляторів. Кожен з них генерує проміжний код, який подається на вхід кодогенератору [2].

Зазвичай, на вхід генератора коду подається дерево розбору або абстрактне синтаксичне дерево. Дерево перетвориться в лінійну послідовність інструкцій проміжної мови [2].

Складні компілятори, як правило, роблять кілька проходів через різні проміжні форми коду. Цей багатокроковий процес використовується тому, що багато алгоритмів оптимізації коду простіше реалізувати окремо, або ж тому, що якийсь крок оптимізації залежить від результату обробки іншого кроку. Окрім того, при такій організації легко створити один компілятор, який буде створювати код для кількох платформ, так як достатньо замінити останній крок генерації коду [2].

Подальші етапи компіляції можуть і не належати до «генерації коду», в

залежності від того, наскільки значними будуть зміни, що вносяться ними. Так, локальна оптимізація навряд чи може називатися «генерацією коду», проте сам генератор коду може включати в себе етап локальної оптимізації.

В додачу до основного завдання – перетворення коду з проміжного представлення в машинні інструкції – генератор коду зазвичай намагається оптимізувати код, створений тими чи іншими способами. Наприклад, він може використовувати більш швидкі інструкції, використовувати менше інструкцій, використовувати наявні регістри і запобігати надлишковим обчисленням.

Існують завдання, які, зазвичай, вирішують складні генератори коду:

- вибір інструкцій: які саме інструкції використати;
- планування інструкцій: у якому порядку розміщувати ці інструкції. Планування – це оптимізація, котра може значно впливати на швидкість виконання програми на конвеєрних процесорах;
- розміщення у регістрах: розміщення змінних програми у регістрах процесора;
- дані про налагодження покоління, якщо потрібно, так що код може бути налагоджений.

Вибір інструкцій зазвичай виконується рекурсивним обходом абстрактного синтаксичного дерева. В цьому випадку порівнюються частини конфігурацій дерева з шаблонами. Наприклад, деяке дерево може бути перетворене в лінійну послідовність інструкцій рекурсивну генерації послідовностей, а потім в іншу інструкцію.

Абстрактне синтаксичне дерево, або AST, – це деревовидне представлення вихідного коду комп’ютерної програми, яке передає структуру вихідного коду. Кожен вузол дерева представляє конструкцію, що зустрічається у вихідному коді [3].

В компіляторах, які використовують проміжну мову, може бути дві стадії вибору інструкцій – одна для перетворення дерева розбору в проміжний код, а друга (виконується значно пізніше) – для перетворення

проміжного коду в інструкції цільової системи команд. Друга стадія не вимагає обходу дерева: вона може виконуватися послідовно і зазвичай складається з простої заміни операцій проміжної мови відповідними їм кодами операцій. Насправді, якщо компілятор фактично є транслятором (наприклад, один переводить Eiffel в C), то друга стадія генерації коду може включати побудову дерева з лінійного проміжного коду.

Автоматична генерація коду – це зовсім не означає правильне та безпечне створення програмного забезпечення. Можна розцінювати її як додатковий етап кастомної команди для компіляції, але у цій функції можна створювати додаткові класи, інтерфейси вже існуючих бібліотек [4].

Так як досить складно просто уявити, навіщо це взагалі можна використовувати, самі розробники з Microsoft рекомендують працювати з цим виходячи з наступних правил [4].

Кодогенератори гарний інструмент для:

- прискорення створення веб-додатків;
- роботи із серіалізацією;
- роботи з командним рядком;
- роботи з інструментами враже SWIG.

Кодогенератори поганий інструмент для зміни будь-якого вже створеного коду, а це означає:

- змін недосконалостей мови програмування;
- оптимізацій;
- додавань механізмів логування;
- модифікацій проміжного коду IL;
- модифікацій алгоримту програми, у тому числі зміни угоди про виклики функцій.

1.2 Визначення об'єкта дослідження

Headless система керування вмістом (CMS) – це CMS, яка є лише

серверною частиною та не має жодного рівня презентації.

У традиційній CMS у є тіло (backend) і голова (fronend), які залежать один від одного. Тіло – це бекенд, де обробляється весь код веб-сайту, а вміст створюється та зберігається. Голова – це шар презентації, де можна побачити, як контент виглядатиме для користувачів. Вони з'єднані разом, тому, коли створюється або редагується вміст у в бек-офісі – тілі вашої CMS –можна побачити, як він виглядає в інтерфейсі – голові CMS. Традиційна монолітна CMS була створена, щоб зробити цей процес простим і легким як для розробників, так і для редакторів [5].

Але в сучасному світі веб-вміст споживається через безліч різних екранів і пристроїв. І щоб обробити весь цей вміст, поєднання бекенда та інтерфейсу може насправді принести більше шкоди, ніж користі. Тому була розроблена headless архітектура, щоб надати більше гнучкості в складному світі нових «голов». В headless CMS немає зв'язку між тілом і головою. Фронтенд частина була відрізана, щоб дати лише бекенд, де можна створити весь свій вміст і подати його будь-якому фронтенду [5].

Підключення фронтенд частини здійснюється через API RESTful або за допомогою GraphQL, де вміст стає доступним із Headless CMS. Щоб отримати доступ до вмісту, потрібно викликати API, що дозволяє відображати вміст на даному пристрої. Це означає, що створений вміст неактивний, доки не будуть здійснені будь-які виклики API, і він не буде активно розповсюджуватися [5].

Мова запитів GraphQL – це вибір полів на об'єктах, за якими потрібно зробити запит. А щоб він працював на будь-якій серверній платформі та мові програмування, він повинен мати власні правила щодо обробки запитів і даних, які можна повернути. Запит GraphQL дуже точно відповідає результату і що стає досить легко передбачити, що запит поверне, навіть якщо користувач мало знає про сам сервер GraphQL [6].

Щоб усе було структурованим, служба GraphQL, яка обробляє запити, матиме фіксований набір типів, які описують набір можливих даних,

доступних для запиту. Таким чином, щоразу, коли робиться запит, він перевіряється за цією схемою і виконується, лише якщо це дійсний запит. Хоча дані, які можна запитувати, залежатимуть від сервісу GraphQL, який використовується, вони завжди будуть передбачуваними та відповідати цій структурі об’єктів і полів [6].

Роботу з GraphQL підтримує зараз велика кількість платформ (рисунок 1.1): веб, Android, iOS та багато інших. GraphQL-клієнт відправляє запит на отримання даних або їх зміну, складений відповідно до схеми, на GraphQL-сервер. GraphQL-сервер, своєю чергою, є HTTP-сервер, з яким пов'язана схема GraphQL. Тобто мається на увазі, що через цю схему «пропускаються» всі запити, отримані від клієнта, та відповіді, що повертаються [7].

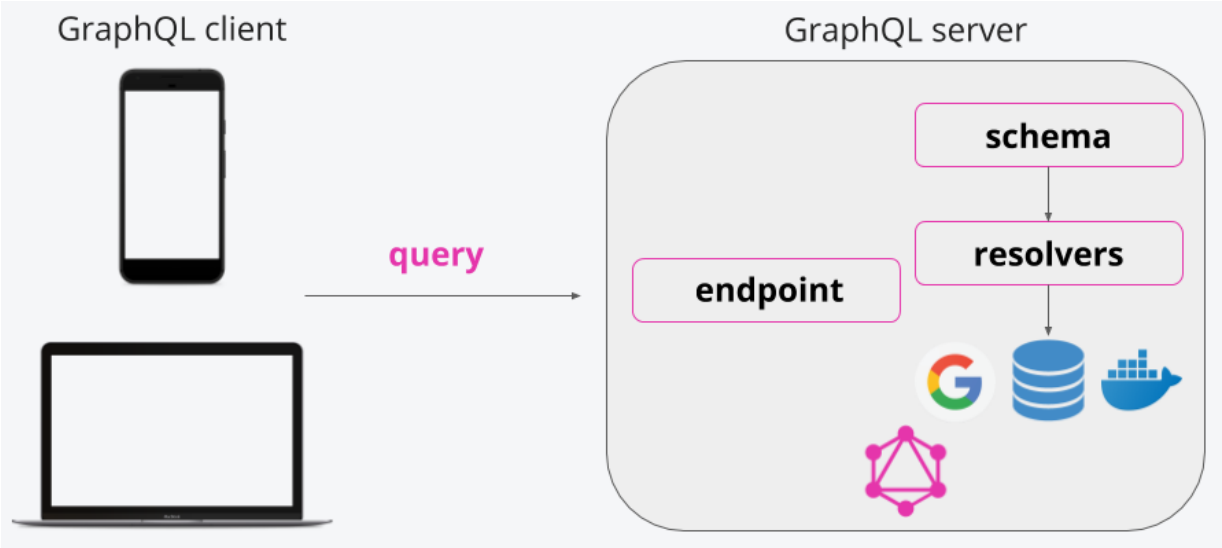


Рисунок 1.1 – Схема взаємодії клієнтів і сервера GraphQL

Сервер GraphQL не може знати, що робити із запитом, якщо йому не «пояснити» це за допомогою спеціальних функцій. Завдяки їм GraphQL розуміє, як отримати дані для полів, що запитуються. Ці функції пов'язані з відповідними полями і називаються розпізнавателями або резолверами (resolvers). Після цього клієнту повертається відповідь, яка відображає структуру даних, що запитується з клієнта, зазвичай у форматі JSON [7].

1.3 Формування вимог до програмної системи

Під час розробки системи керування вмістом Umbraco з headless архітектурою виникає потреба в створення API для отримання контенту. Таким API може виступити, наприклад, GraphQL, який надасть можливість вибору полів на об’єктах, за якими потрібно зробити запит.

Існує готове рішення Umbraco Heartcore, яке може виступити в якості системи керування вмістом (CMS) з GraphQL API. Таке рішення надає можливість автоматичної генерації схеми для API але є платним.

Отримання даних з системи керування вмістом, відбувається відповідно до схеми GraphQL. Тому основною проблемою стає створення вручну великої кількості класів для описання схеми API. Незважаючи на те, що код буде відрізнятися від класу до класу, він досить стереотипний за структурою, тому його краще генерувати автоматично, ніж писати вручну

Отже, основним завданням стає створення власного генератора класів C#, який буде здатний генерувати шаблонний код схеми GraphQL API.

1.3.1 Загальні вимоги

Для проектування програмної системи необхідно сформулювати певні вимоги такі як:

- перелік функціоналу, що повинна містити система;
- архітектура програмної системи;
- частини, з яких вона буде складатися;
- технології, які слід застосувати, для досягнення найкращого результату.

Загальні вимоги будуть наступні. Генератор коду, у вигляді консольного додатку, повинен бути здатний автоматично оновлювати схему API для існуючої headless системи керування вмістом, представленої у вигляді веб-додатку. Такий консольний додаток є допоміжним інструментом,

який використовується під час розробки. Для досягнення найкращого результату слід використати платформу .NET Core та сучасну технологію генерації коду Roslyn.

1.3.2 Функціональні вимоги

Створення GraphQL API для Umbraco CMS залежить від структури сторінок в CMS. Джерелом такої інформації може виступити спеціальний пакет uSync. USync приймає інформацію з Umbraco CMS, яка зберігається в базі даних, і переміщує її на диск у вигляді XML файлів.

Початкова версія генератора має буде реалізована у вигляді консольного додатку, який очікує XML файли у якості вхідного параметра і на основі них генерує класи мови C# . Для реалізації даної задачі необхідно виконати наступне:

а) описати XML файл, який має розуміти автоматичний генератор коду;

в) створити парсер, який міг би розбивати XML файл на окремі блоки;

г) створити генератор коду мови програмування C#.

1.3.3 Нефункціональні вимоги

Окрім перелічених вище функціональних вимог, слід розглянути нефункціональні вимоги, а саме вимоги до мов програмування, фреймворків та інструментів:

а) автоматичний генератор коду буде розроблено із використанням мови програмування C# та платформи .NET;

б) клієнт матиме вигляд консольного застосування, який буде надавати можливість генерувати безпосередньо програмний код;

в) також до нефункціональних вимог можна віднести те, що генератор має працювати швидко та мати можливість повідомляти про етапи процесу обробки XML файлів.

2 ВИБІР ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ РОЗРОБКИ

2.1 Огляд інструментів кодогенерації

Генерація коду була доступна в .NET з самого початку. Першим інструментом став CodeDom, який був доступний з версії 1.0. Він має менший набір функціональних можливостей, ніж сучасні генератори такі, як Roslyn. Крім того, CodeDom намагається абстрагувати всі мови (C#/Vb.NET), і через це не можливо користуватися перевагами певної мови [8].

Інструмент шаблонів T4 дозволяє генерувати код, маючи шаблон, написаний у нотації T4, цей шаблон використовується двигуном з деякими вхідними даними (файл або база даних тощо) для створення коду. Ця технологія раніше широко використовувалася, але Microsoft повільно відмовляється від цієї техніки в нових продуктах. Слід зазначити, що генератори джерел трохи відрізняються від простого механізму шаблонів [8].

Існують два основних інструменти, які використовують технологію ін'єкції IL – Postsharp і відкритий проект Fody. Обидва інструменти використовують техніку під назвою code weaving для введення IL коду у процесі збірки. Ця техніка дозволяла виконувати аспектно-орієнтоване програмування (AOP) протягом багатьох років, але вона має основний недолік – згенерований код є свого роду чорним ящиком, щось вводиться в код, але IDE і компілятор не знають про цей код. Через такий підхід багато людей не хочуть впроваджувати ці методи у свої проекти, тому що отриманий результат не буде чистим кодом [8].

Аспектно-орієнтоване програмування – це парадигма програмування, яка спрямована на збільшення модульності, дозволяючи розділити наскрізні проблеми. Вона робить це, додаючи додаткову поведінку до існуючого коду, не змінюючи сам код [9].

Генератори вихідного коду Roslyn, які постачаються разом із пакетом

SDK .NET 5, ймовірно, є одним з найкращих інструментів за останні кілька років. Завдяки великим можливостям Roslyn, розробник отримує дерево компіляції як вхідні дані, і може щось додати до цієї компіляції. Генератори дозволяють покращити написання коду, генеруючи його на льоту під час розробки замість того, щоб передавати набір компонентів, помічників і базових класів, які значною мірою покладаються на рефлексію [8].

2.1.1 Аналіз технології CodeDOM

CodeDOM – це досить складний набір класів, які можна знайти у просторах імен System.CodeDom та System.CodeDom.Compiler. Більшість типів у цих просторах імен можна знайти у збірках mscorlib.dll та System.dll у Global Assembly Cache (GAC). З цього зрозуміло, наскільки важливим є CodeDOM у .NET Framework Class Library [10].

Одна з причин того, що CodeDOM знаходиться на такому видному місці, полягає в тому, що він був у .NET Framework із самого початку. CodeDOM було випущено з оригінальною версією .NET 1.0 у лютому 2002 року. Він не дуже еволюціонував у наступних двох релізах фреймворку: в основному для підтримки покращеної моделі делегатів та додавання дженериків [10].

Однак, з 2005 року CodeDOM фактично не змінювався, частково тому, що це вже досить багатий інструмент, щоб підтримувати багато видів сценаріїв метапрограмування. Метапрограмування – це процес написання комп’ютерних програм, які, у свою чергу, пишуть інші програми. Процес метапрограмування надає більшу гнучкість програмістам, оскільки робота, яка зазвичай виконується під час виконання, розподіляється на період компіляції [11].

CodeDOM використовує так звані графи коду, щоб виразити код у вигляді даних. Граф коду – це ієрархічна структура даних, що представляє логіку та структуру алгоритму. Він може бути розібраний з тексту вихідного

коду або збудований покроково, як написання рядків коду вашою улюбленою мовою.

Графи коду можуть бути перетворені на виконуваний IL, коли приходить час їх використовувати. Дерева виразів, введені у версії .NET 3.0, можуть представляти код як дані. Вони також можуть бути побудовані програмно або розібрані синтаксично з вихідного коду і перетворені на IL для виконання. Графи коду та дерева виразів здаються однаковими на перший погляд. За лаштунками, проте, реалізація цих двох систем метапрограмування є досить різнорідною [10].

Простіри імен CodeDom у .NET Framework забезпечують такі переваги:

- CodeDom заснований на єдиній моделі відтворення вихідного коду. Таким чином, вихідний код може бути згенерований для будь-якої мови, яка підтримує специфікацію CodeDom;

- CodeDom дозволяє динамічно створювати, компілювати та виконувати програми під час виконання;

- CodeDom забезпечує незалежну від мови об'єктну модель для представлення структури вихідного коду в пам'яті;

- майбутні випуски CodeDom могли б перекладати файли вихідного коду між мовами так само, як програми-перетворювачі графічних файлів сьогодні. Наприклад, програма VB.NET може бути представлена у вигляді CodeDom, а потім перекладена на C# для іншого розробника.

Як і для будь-якої нової технології, її функціональність має обмеження – технологія CodeDom не є винятком, вона також має певні недоліки. Простіри імен CodeDom містять класи для концептуального представлення більшості програмних конструкцій. Приклади таких: оголошення, оператори, ітерації, масиви, приведення, коментарі, обробка помилок тощо. Проте існують обмеження щодо поточної реалізації CodeDom, які залишаться до тих пір, поки Microsoft не оновить простір імен CodeDom. Також слід зазначити, що технологія CodeDOM давно застаріла оскільки на заміну .Net framework вже давно прийшов .Net core [12].

Технологія CodeDom дозволяє генерувати код кількома мовами. Для створення коду необхідно створити дерево об'єктів з простору імен System.Codedom, що надає класи для відображення в мовно-незалежному стилі більшості програмних конструкцій. Ці об'єкти є синтаксичними конструкціями мов програмування, на кшталт оголошення класів і змінних, циклів, умов, привласнень, викликів методів тощо. Стверджується, що його застосування дозволяє не відволікатися на нюанси розробки кожної мови програмування та бути впевненим, що CodeDom все правильно обробить. Стверджується також, що така абстракція дозволяє краще структурувати код, дозволяючи при необхідності внесення змін не перейматися вже створеним кодом [13].

Переваги:

- технологія CodeDom надає єдину модель, що не залежить від мови реалізації, для представлення структури програмного коду;

- підтримка генерації кількома мовами. Оскільки для представлення програмного коду застосовується одна модель, виведення згенерованого коду можна здійснювати для багатьох мов, що підтримують технологію CodeDom. Це також означає, що генератор можна впроваджувати і для ще не створених мов, які з'являться в майбутньому. Впровадження підтримки CodeDom у майбутніх мовах буде достатнім для генерації [13];

- динамічна компіляція та запуск програмного коду. У просторі імен CodeDom є класи, що дозволяють динамічно компілювати програмний код у файли або бібліотеки, що виконуються. Можна також динамічно запускати скомпіловані програми.

- у міру розвитку технології CodeDom є можливість появи можливості конвертувати вихідний код програм з однієї мови в іншу. Наприклад, з C# Visual Basic і навпаки [13].

Недоліки:

- трудомісткість створення генератора. Для генерації одного рядка коду може знадобитися написати близько десятка рядків досить складного

коду. Але цей недолік можна зменшити, групуючи набори стандартних операцій створення коду в функції і процедури [13];

- не всі характеристики коду підтримуються, крім тих, які є спільними всім мов. Наприклад, немає вкладених просторів імен, немає підтримки списків змінних в оголошенні змінних, немає підтримки модифікатора unsafe C#. Хоча ці обмеження можна обійти за допомогою класу CodeSnippetExpression, але такий прийом сильно зменшує гнучкість генерації для різних мов [13];

- немає гарантії, що згенерований код компілюватиметься. Після генерації потрібно переконатися, що код коректний;

- поведінка згенерованого коду різними мовами може бути різною, аж до того, що однією мовою код компілюватиметься, а іншою - ні. І навіть якщо всіма мовами код скомпілюється успішно, нюанси поведінки згенерованого коду різними мовами можуть відрізнятися [13];

- програми із застосуванням CodeDom не мають гарної наочності шаблонів. Виконання пошуку потрібної ділянки шаблонного коду є складним заняттям;

- згенерувати код можна буде лише у тому стилі та форматі, який вбудовано за замовчуванням у CodeDom. Якщо ж потрібно створити код, стиль якого відрізняється від стандартного, то доведеться самостійно писати розширення для CodeDom [13].

Таким чином можна зробити висновки що до цієї технології. Якщо є необхідність генерувати один і той же код відразу кількома мовами, то CodeDom може виявитися дуже корисним інструментом. Ця технологія активно використовується компанією Microsoft. Особливо часто вона застосовується у Visual Studio, де під час виконання будь-яких дій залежно від вибраної мови автоматично створюються об'єкти та програмний код. Теоретично є можливість генерувати код для будь-якої мови, для якої існує провайдер CodeDom. Провайдер також включає підтримку компіляції згенерованого коду, причому набір об'єктів CodeDom конвертується

безпосередньо в MSIL. CodeDom підтримує будь-яку мову, для якої є провайдер. Розробники також можуть додати нову мову, створивши для неї власний провайдер. Однак, якщо потрібно генерувати код тільки на одній мові, застосування цієї технології може не виправдати себе. Набагато простіше може бути застосування шаблонів чи маніпуляція рядками тексту.

CodeDom абстрактний настільки, що дозволяє виконувати генерацію для кількох мов. Однак і складний настільки, що якщо не потрібно генерувати код одночасно кількома мовами, то його застосування може бути невигідним, витрати часу можуть виявитися занадто великими, тому слід розглянути інші інструменти кодогенерації.

2.1.2 Аналіз технології Reflection

Рефлексія під час виконання – це потужна технологія, яка була додана в .NET дуже давно. Існує незліченна кількість сценаріїв її використання. Дуже поширеним сценарієм є виконання певного аналізу коду користувача під час запуску програми та використання цих даних для створення чогось [14].

Багато фреймворків і бібліотек активно використовують елементи рефлексії, такі як System.Text.Json, System.Text.RegularExpressions, і такі фреймворки, як ASP.NET Core і WPF, які виявляють і/або видають типи з коду користувача під час виконання [14].

Багато з найпопулярніших пакетів NuGet активно використовують рефлексію для виявлення типів під час виконання. Інтеграція цих пакетів є важливою для більшості додатків .NET, тому «зв’язуваність» і здатність коду використовувати оптимізацію компілятора AOT можуть сильно постраждати [14]. Якщо в ієрархії класів зустрічається ланцюг віртуальних методів (з допомогою слів virtual, override), то компілятор будує так зване пізнє зв’язування. При пізньому зв’язуванні виклик методу відбувається на основі типу об’єкту, а не типу посилання на базовий клас. Пізнє зв’язування

використовується, коли потрібно реалізувати поліморфізм [15].

Наприклад, ASP.NET Core використовує рефлексію під час першого запуску веб-служби, щоб виявити визначені конструкції, щоб вона могла «з'єднати» такі речі, як контролери та сторінки. Хоча це дає змогу писати простий код із потужними абстракціями, це супроводжується зниженням продуктивності під час виконання: коли веб-служба або програма вперше запускається, вони не можуть приймати жодних запитів, доки не буде завершено весь код рефлексії під час виконання, який виявляє інформацію про код. Хоча це зниження продуктивності не є величезним, це певна фіксована вартість, яку програміст не можете покращити самостійно у власній програмі [14].

Коли створюється програма .NET, створюються збірки – наприклад, файли *.dll. Ці збірки містять модулі, які містять деякі типи. Типи містять члени. Рефлексія дозволяє прочитати інформацію про них. Таким чином, можна динамічно завантажувати нові файли *.dll і викликати їх методи або події без редагування коду. Динамічно означає, що це відбувається під час виконання. Іншими словами, програма яка компілюється, не знає, які типи потрібно використовувати, поки вона не запуститься. Таким чином можна створити клієнт, який може динамічно виконувати методи в інших збірках на основі правил. Якщо оновлюються класи в інших збірках, дотримуючись правил, не потрібно оновлювати код клієнта [16].

Це надзвичайно корисно, коли створюються програми типу плагінів. Спочатку створюються інтерфейси і викликаються методи від клієнта шляхом рефлексії. Потім можна створити плагіни, що відповідають інтерфейсу клієнта, які можна динамічно завантажувати як файли *.dll і виконувати [16].

Інший сценарій – розвиток фреймворку. Розробник фреймворка, може не знати, які реалізації створять користувачі, тому можна використовувати лише рефлексію для створення цих екземплярів. Один приклад – у деяких фреймворках MVVM, якщо створюються класи, дотримуючись конвенцій,

наприклад xxxViewModel, фреймворк може знайти всі ViewModels та автоматично завантажити їх за допомогою рефлексії [16].

Основною проблемою використання рефлексії може стати продуктивність. Оскільки вона працює під час виконання, теоретично вона працює трохи повільніше, ніж звичайна програма. Але вона гнучка для багатьох сценаріїв, особливо під час розробки фреймворку. Якщо прийнятно витрачати кілька секунд (або лише сотні мілісекунд) на завантаження збірок, можна використовувати рефлексію [16].

Що до використання цієї технології у вирішення завдання створення власного кодогенератора, слід зазначити, що дана технологія не підходить для вирішення завдання. Як було зазначено вище, для рефлексії поширеним сценарієм є виконання певного аналізу коду користувача під час запуску програми та використання цих даних для створення чогось, а у завданні яке подтрібно вирішити, треба створювати код з нуля, проводячи аналіз XML файлів.

2.1.3 Аналіз технології IL weaving

Майже в усіх програмах .NET зустрічається багато шаблонного коду, який хоча і має досить стандартну реалізацію, але є досить важливим з точки зору реалізації. Деякі поширені приклади є заміною методу ToString, Equals для класів моделі. Або імплементація IDisposable або INotifyPropertyChanged. У всіх цих прикладах більшість коду є досить стандартною [17].

Одним із способів автоматизації цього коду є автоматичне введення його у згенерований MSIL. Цей процес введення коду після компіляції безпосередньо в створену проміжну мову відомий як .NET Assembly weaving або IL weaving (подібно для переплетення байтового коду в Java) [17].

Замість того, щоб маніпулювати вихідним кодом, можна маніпулювати будівельними блоками .NET [18].

Якщо це робиться перед компіляцією, тобто додаються рядки

вихідного коду перед компіляцією, це відомо як source code weaving. IL Weaving – це процес, який змінює команди IL збірки після компіляції [17].

Fody – це бібліотека для збірки .NET, яка написана Саймоном Кроппом. Вона додає завдання після збірки в пайплайн MS, щоб маніпулювати згенерованим IL. Зазвичай для цього потрібне багато технічного коду, який надає Fody. Fody надає розширювану модель надбудов, де будь-хто може використовувати основний пакет Fody (який надає базовий код для додавання завдання після збірки та маніпулювання IL) і створювати власні спеціальні надбудови. напр. Equals. Fody генерує реалізацію методів Equals і GetHashCode, a ToString [17].

На рисунку (рисунок 2.1) показано, як проходить подібний процес.

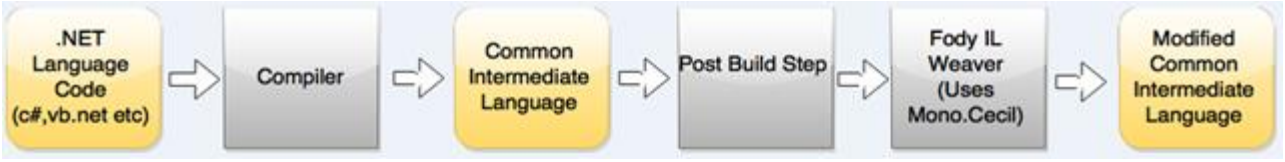


Рисунок 2.1 – Процес IL weaving

Можна зробити висновок, що технологія IL weaving також не підходить для вирішення поставленого завдання. Це гарний інструмент для того щоб модифікувати існуючий код, а не створювати його з нуля.

2.1.4 Аналіз технології T4

Текстові шаблони T4 під час розробки дозволяють створювати програмний код та інші файли у проекті Visual Studio. Як правило, шаблони створюються таким чином, що вони змінюють код, що створюється відповідно до даних моделі. Модель – це джерело даних, що описує певний аспект програми. Вона може бути представлена в будь-якій формі, як файл або база даних. Вона не повинна відповідати певній формі, наприклад, як

модель UML або модель DSL. Типові моделі створюються у вигляді таблиць або XML-файлів [19].

Наприклад, можна створити модель, яка визначає робочий процес як таблицю чи схему. На основі моделі можна створити програму, яка виконує робочий процес. У разі зміни вимог користувачів можна легко обговорити новий робочий процес із користувачами. Повторне створення коду на основі робочого процесу надійніше ніж оновлення коду вручну [19].

Текстовий шаблон містить поєднання тексту, який потрібно створити, та програмного коду, що створює змінні частини тексту (рисунок 2.2). Програмний код дозволяє повторювати чи умовно опускати частини згенерованого тексту. Створений текст може бути програмним кодом, який формує частину програми [20].

Можна розрізняти файли програм залежно від моделі. Модель є джерелом вхідних даних, таким як база даних, файл конфігурації, модель UML, модель DSL або інше джерело. Зазвичай, у одній моделі створюється кілька програмних файлів [19].

Як правило, рекомендовано розділяти код шаблонів на дві частини.

- конфігурація або частина, що відповідає за збирання даних, яка визначає значення змінних, але не містить текстові блоки. Її іноді називають частиною «моделі», оскільки вона створює збережену модель і зазвичай зчитує файл моделі.

- частина, що створює текст (наприклад foreach(...){...}), яка використовує значення змінних.

Цей поділ необов'язковий, однак він спрощує читання шаблону, роблячи менш складною частину, яка включає текст.

Існує два типи шаблонів T4: часу виконання і часу розробки. Різниця полягає в тому, що шаблон часу виконання T4 виконується в додатку для створення текстових рядків. Він створює клас *.cs, який містить метод TransformText(). Потім можна викликати цей метод для створення рядків, навіть якщо на цільовій машині не встановлено Visual Studio [19].

Навпаки, шаблон Т4 часу розробки генерує вихідний код або текстові файли, коли програміст зберігає шаблон у Visual Studio. Для використання шаблону часу виконання Т4, потрібно встановити властивість користувацького інструмента файлу як TextTemplatingFilePreprocessor. Для шаблону часу розробки Т4 властивість Custom Tool має бути встановлено на TextTemplatingFileGenerator [19].

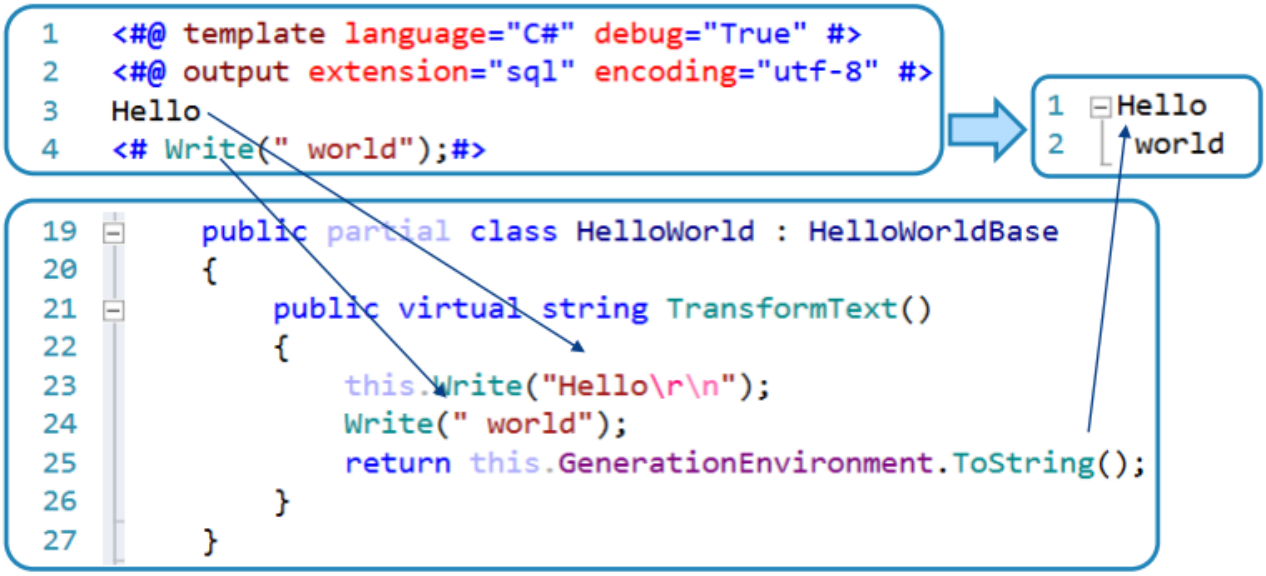


Рисунок 2.2 – Алгоритм роботи Т4 шаблону

Що до використання технології Т4, слід зазначити, що дана технологія хоч і має привабливий вигляд, але з її використанням з’являється низка проблем, здебільшого пов'язаних із підвантаженням системних бібліотек, які можуть вирішуватися зі змінним успіхом. Під час обробки збірки, у шаблона Т4 часто виникають проблеми з її знаходженням та завантаженням.

2.1.5 Аналіз технології Source generators

Механізм Source Generators – це розширення до компілятора, які дозволяють в момент компіляції, «на льоту», додати до коду якісь свої

фрагменти. Головною особливістю є те, що генератор має можливість отримати доступ до поточного контексту компіляції (тобто до синтаксичного дерева, всіх вихідних файлів, а також загальної семантичної моделі: інформації про типи та їх вміст як для компілюваного коду, так і всіх залежних збірок) , а також потрібним додатковим файлам у проекті [21].

Основні особливості генераторів:

- генератори виробляють один або більше рядків коду на C#, які додаються до компілюваного коду;
- працюють виключно на додавання. Генератори можуть додати новий код компіляції, але не можуть змінити існуючий;
- можуть генерувати діагностичну інформацію. Якщо вихідний код не може бути згенерований, генератор проінформує про цю проблему;
- можуть отримати доступ до додаткових файлів, які не містять текст на C#;
- запускаються не впорядковано. Кожен генератор отримує однаковий контекст компіляції на вхід і не має доступу до коду, згенерованого іншими генераторами;
- користувач підключає генератор до проекту, як звичайну збірку.

Генератор вихідного коду – це новий тип компонента, який можуть написати розробники C#, що дозволяє робити дві основні речі:

- отримати об'єкт компіляції, який представляє весь код користувача, який компілюється. Цей об’єкт можна перевіряти, а також можна написати код, який працює з синтаксисом і семантичними моделями для коду, що компілюється.
- створення вихідних файлів C#, які можна додати до об'єкта Compilation під час компіляції. Іншими словами, можна надати додатковий вихідний код як вхідні дані під час компіляції коду.

У поєднанні ці дві речі роблять генератори вихідного коду такими корисними. Можна перевірити код користувача з усіма метаданими, які компілятор нарощує під час компіляції, а потім випустити код C# назад у ту

саму компіляцію, яка базується на даних, які було проаналізовано [13].

Генератори вихідних кодів виконуються як етап компіляції (рисунок 2.3).

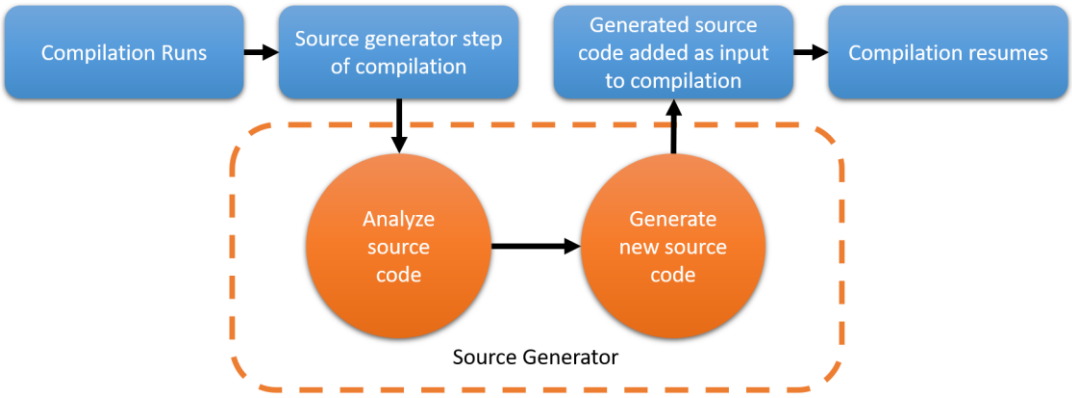


Рисунок 2.3 – Робота генераторів під час компіляції

Генератор вихідного коду – це збірка .NET Standard 2.0, яка завантажується компілятором разом із будь-якими аналізаторами. Його можна використовувати в середовищах, де можна завантажувати та запускати компоненти .NET Standart [13].

Сьогодні існує три загальних підходи до перевірки коду та генерації інформації або коду на основі цього аналізу, які використовуються сучасними технологіями: рефлексія під час виконання, IL weaving та жонглювання завданнями MSBuild [13]. Механізм MSBuild, надає схему XML для файлу проекту, який контролює, як платформа збирання обробляє та створює програмне забезпечення [22]. Генератори джерел можуть бути вдосконаленням у порівнянні з кожним підходом.

З генераторами вихідного коду фаза виявлення контролера під час запуску може відбуватися під час компіляції шляхом аналізу вихідного коду та видачі коду, необхідного для «підключення» програми. Це може призвести до швидшого запуску, оскільки дія, що відбувається під час виконання, може бути перенесена на час компіляції [13].

Генератори вихідного коду можуть підвищити продуктивність способами, які не обмежуються рефлексією під час виконання, щоб також виявляти типи. Деякі сценарії включають виклик завдання MSBuild C# кілька разів, щоб вони могли перевірити дані з компіляції. Виклик компілятора кілька разів впливає на загальний час, необхідний для створення програми [13].

Іншою можливістю, яку можуть запропонувати генератори вихідного коду, є уникнення використання деяких API із «строковим типом», наприклад, як працює маршрутизація ASP.NET Core між контролерами та розрізними сторінками. За допомогою генератора вихідних кодів маршрутизація може бути чітко введена з необхідними рядками, які генеруються як частина під час компіляції. Це зменшить кількість разів, коли неправильно введений рядковий литерал призводить до того, що запит не потрапляє на правильний контролер [13].

Іншою характеристикою генераторів вихідних кодів є те, що вони можуть допомогти усунути основні бар'єри для оптимізації компіляції на основі компоновщика та AOT [13].

Компілятор перед виконанням (англ. ahead-of-time compilation (AOT)) – це вид транслятора, який використовує метод компіляції перед виконанням. Використовується як для компіляції високорівневних мов програмування, так і для компіляції так званих «проміжкових» мов. В багатьох реалізаціях мов програмування використовується компіляція під час виконання, яка дозволяє компілювати проміжковий код напряму в бінарний під час його виконання, що дозволяє значно збільшити швидкість виконання, але ця стратегія потребує виділення додаткової пам'яті. Стратегія AOT не потребує виділення додаткової пам'яті, а також вона проходить з мінімальним навантаженням на систему. Процес компіляції повністю виконується перед виконанням програми. Компіляція перед виконанням дозволяє компілювати класи в машинний код для подальшого виконання однієї і тієї самої програми. Компілятор AOT працює з платформою обміну даними класу [13].

Найпоширенішим підходом до спільного використання коду/функціональності є надання базових/допоміжних класів або компонентів для повторного використання в інших проектах. Їх API та поведінка є відносно статичними, тобто компоненти не можуть реагувати на потреби розробників, як від додавання/вилучення функції чи перейменування властивості на льоту. Щоб зробити їх більш динамічними, бібліотеки та фреймворки зазвичай покладаються на рефлексію, яка має ряд обмежень і деякі обмеження продуктивності [23].

Звичайно, генератори вихідного коду також мають обмеження, але їх можливості розширюються далеко за межі рефлексії. На перший погляд може здатися, що генератори вихідного коду є заміною рефлексії, але це правда лише частково. Є варіанти використання, які більше підходять для генераторів вихідного коду та інші, які легше реалізувати за допомогою рефлексії. Якщо можливо реалізувати функцію за допомогою будь-якого із інструментів, треба використовувати такий, який найкраще відповідає (скоріше технічним) вимогам. Ось кілька плюсів і мінусів, на основі яких можна прийняти рішення.

Генератори вихідного коду Roslyn, які постачаються разом із пакетом SDK .NET 5, ймовірно, є одним з найкращих інструментів за останні кілька років. Завдяки великим можливостям Roslyn, розробник отримує дерево компіляції як вхідні дані, і може щось додати до цієї компіляції. Генератори дозволяють покращити написання коду, генеруючи його на льоту під час розробки замість того, щоб передавати набір компонентів, помічників і базових класів, які значною мірою покладаються на рефлексію [24].

Створення коду може бути не обов’язково під час компіляції, але те, що Roslyn дозволяє легко працювати з деревом компіляції робить його найпривабливішим інструментом.

За результатами порівняння засобів генерації можна визначити, що генератори вихідного коду є одним з найпопулярніших інструментів, які зібрали у собі можливості інших засобів. Генератори коду Roslyn дозволяють

легко створювати шаблонний код, та вирішують типові проблеми метапрограмування (навігація, налагодження, тестове покриття для процесу генерації коду, простота у використанні). Такі генератори не мають проблем з продуктивністю, що надає можливість проведення швидкої збірки. Підтримка IntelliSense та можливість проводити налагодження з діагностикою створюють більш комфортні умови під час розробки. Але слід зазначити, що у такого інструмента відсутня можливість редагування існуючого коду. Це означає, що неможливо змінити тіло методу або видалити властивості з класу, можна додати лише новий код [24].

Як компілятор, Roslyn канонічно знає все, що стосується синтаксису та семантики коду. Можна зібрати цілий клас, створивши такі частини, як імпорт простору імен, поля, властивості, конструктори, а потім створивши з цих частин єдиний блок компіляції. Це можна використовувати для створення шаблонних класів. Створення власного шаблону може значно збільшити швидкість кодування та підвищити узгодженість кодової бази. Платформа Roslyn може бути потужним інструментом, який допомагає під час розробки програмного забезпечення. Використовуючи Roslyn можна отримати більш послідовну базу коду та прискорити час розробки.

Якщо звернути увагу на інші інструменти, то T4 templates, на данний час, застарів і зник. Хоч він створює нові файли та може читати дані з інших файлів, а також надає можливість генерування статичного коду та має власний синтаксис, такий інструмент все одно вже не є актуальним. Головним його недоліком є застарілість платформи та відсутність прямої підтримки IDE. IL weaving стає нішевим інструментом. Більшість модифікацій можна легко реалізувати за допомогою генераторів. Різниця в реалізації – IL weaving важко налагодити та перевірити, а. деталі реалізації не будуть відображатися під час компіляції [24].

2.2 Інші засоби генерації

Існують засоби пов’язані з генерацією але повноцінними генераторами коду їх назвати не можна. На такі спеціальні засоби також можна звернути увагу.

2.2.1 Сніппети

Сніппети – це невеликі блоки багаторазового коду, які можна вставити в файли коду за допомогою комбінації гарячих клавіш. Наприклад, якщо ввести prop, а потім натиснути Tab у Visual Studio, VS автоматично згенерує властивість у класі та надасть можливість легко замінити назву властивості. VS вже надає багато вбудованих фрагментів коду, таких як prop, if, while, for, try тощо [16].

Перевага сніппетів полягає в тому, що можна замінити параметри. Наприклад, при використанні шаблону MVVM для програм UWP/Xamarin/WPF, часто потрібно створити властивості в класі, який реалізує інтерфейс INotifyPropertyChanged [16].

Сніппети придатні для повторного використання для вставки цілих класів, методів чи властивостей. Це корисно, коли створюються нові файли/класи/методи. Але для оновлення згенерованного коду після його завершення доведеться видалити наявний код, а потім створити його заново. Це заощаджує час від нудного копіювання/вставки, але його можна використовувати лише один раз [16].

На додаток до основних можливостей керування, функції керування сніппетами можна класифікувати відповідно до сфери взаємодії між сніппетами та текстовим редактором або програмою, у якій вони розміщені [16].

Сніппети можна поділити на групи:

- звичайний текст або «статичні» сніппети;

- інтерактивні або «динамічні» сніппети;
- скриптові сніппети.

Сніппети статичного типу в основному складаються з фіксованого тексту, який користувач може вибрати для вставки в поточний документ. Користувач не може вказати нічого іншого, крім, можливо, положення курсору щодо щойно вставленого тексту. Статичні сніппети подібні до простих макросів [16].

Сніппети динамічного типу містять фіксований текст у поєднанні з динамічними елементами (заповнювачами), які можуть бути змінені редактором або користувачем. Користувач може вказати як вміст динамічних елементів, так і їх положення щодо фіксованого тексту, як частину вибору того, що вставити в поточний документ. Прикладами динамічних елементів можуть бути такі змінні, як поточна дата або системний час, або введення від користувача, що надається через графічний інтерфейс, або введення з іншої програми [16].

Скриптові сніппети складаються з виконуваних сегментів коду на мові макросів або мові сценаріїв. Сніппети сценарію забезпечують найбільшу ступінь гнучкості для користувача, хоча це дещо залежить від мов програмування, які підтримують текстовий редактор, а також від того, чи є мова програмування добре відомою чи специфічною та унікальною для цього конкретного редактора [16].

2.2.2 Інструменти генерації в IDE

Також засоби генерації існують в IDE як додатковий інструмент.

Наприклад, JetBrains Rider надає різноманітні способи створення шаблонного коду. Можна використовувати неоголошені символи коду й автоматично генерувати ці символи при використанні, генерувати члени типу тощо [25].

Якщо в поточному файлі ввімкнено перевірку коду під час

проектування, JetBrains Rider виявляє відсутні елементи та пропонує відповідні швидкі виправлення для реалізації відсутніх членів [25].

Створення нового коду можна контролювати двома основними способами:

- після налаштування різних аспектів стилю коду (наприклад, стиль імен, правила форматування) JetBrains Rider буде виконувати свої вимоги під час генерації коду;

- залежно від налаштувань користувача, заглушки для нових методів можна генерувати різними способами. Наприклад, вони можуть створити новий NotImplementedException(), повернути значення за замовчуванням або включити код, який не буде компілюватися. Можна налаштувати ці та інші параметри в редакторі.

Існує безліч способів, за допомогою яких Visual Studio може допомогти створити, виправити та реорганізувати код.

- можна використовувати сніппети, щоб вставити шаблон, наприклад блок switch або оголошення enum. У Visual Studio є два типи сніппетів: сніппети розширення, які додаються у визначеній точці вставки і можуть замінити фрагмента, і сніппети об’ємного коду (лише C# і C++), які додаються навколо вибраного блоку коду [26].

- можна використовувати швидкі дії для створення коду, наприклад класів і властивостей, або для введення локальної змінної. Також можна використовувати швидкі дії для покращення коду, наприклад, для видалення непотрібних перетворень і невикористаних змінних або для додавання перевірок на нуль перед доступом до змінних.

- можна реорганізувати код, щоб перейменувати змінну, змінити порядок параметрів методу або синхронізувати тип з його іменем файлу.

3 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Проектування у вигляді діаграм

Абревіатура UML розшифровується як «уніфікована мова моделювання», а UML-схеми застосовуються, щоб наочно зобразити пристрій системи та те, як з нею взаємодіють інші системи та користувачі. Під поняттям «система» може бути на увазі веб або консольний додаток, робочий процес і так далі.

Для проектування програмної системи було використано засіб проектування draw.io та спроєктовано діаграму послідовності. Також було отримано діаграми залежностей типів за допомогою JetBrains ReSharper.

Діаграми послідовності – це діаграми взаємодії, які детально описують, як виконуються операції. Вони фіксують взаємодію між об’єктами в контексті співпраці. Діаграми послідовності – це час, і вони візуально показують порядок взаємодії, використовуючи вертикальну вісь діаграми, щоб відобразити час, які повідомлення надсилаються і коли [27].

Діаграми послідовності фіксують:

- взаємодію, яка відбувається у співпраці, яка реалізує варіант використання або операцію (діаграми екземплярів або загальні діаграми);
- взаємодію високого рівня між користувачем системи та системою, між системою та іншими системами або між підсистемами.

Можна описати процес роботи автоматичного генератора коду. Користувач буде створювати об'єкти в системі керування вмістом та зберігати XML опис цих об'єктів, ці файли будуть передаватися до генератора, після чого генератор буде робити аналіз та створювати словники з інформацію: які сторінки існують та який контент в них може бути розміщено, як пов'язані між собою класи, а також вигляд ієрархії файлів. Після цього отримані словники будудь передані далі генератору, який, у

свою чергу, поверне код у форматі класів мови C#. Діаграма послідовностей наведена нижче (рисунок 3.1).

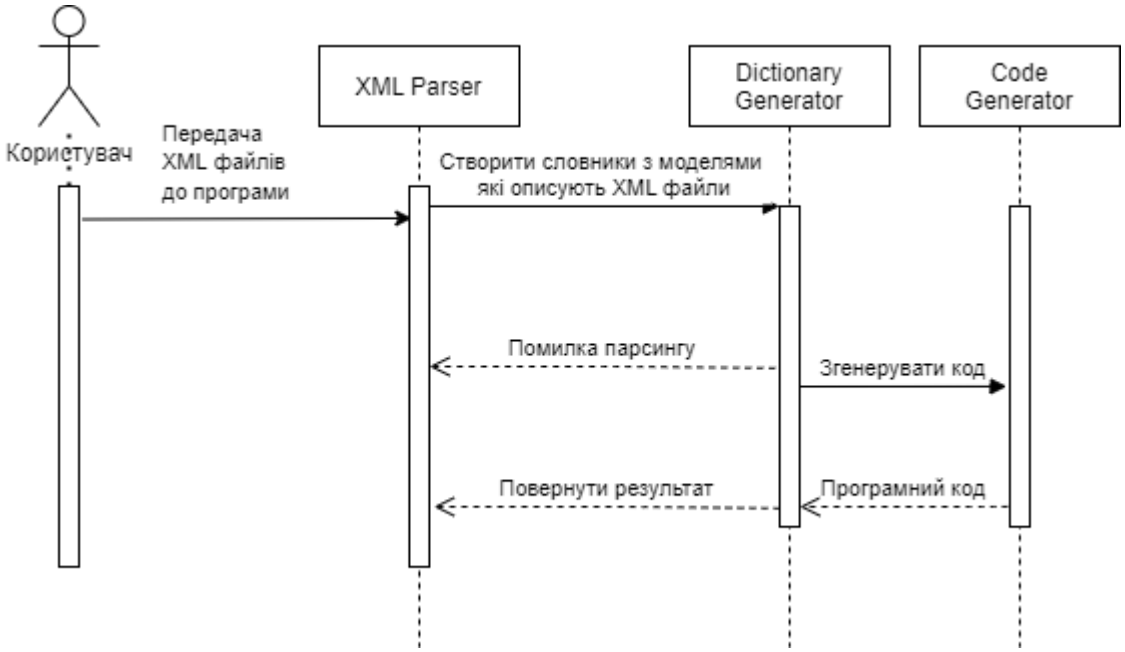


Рисунок 3.1 – Діаграма послідовностей

Основний клас генератор GraphqlGenerator є фасадом, що об'єднує в собі кілька допоміжних класів генераторів (рисунок 3.2). Шаблон «фасад» включає в себе один клас, який надає спрощені методи, необхідні клієнту, і делегує виклики методів існуючим системним класам [28]. FileService – це сервіс, що відповідає за збереження та видалення файлів. Цей сервіс використовують:

- генератор класу запитів graphql – BaseQueryGeneratorService;
- генератори GraphQL класів сторінок та їх полів – DocumentTypeGeneratorService, GroupGeneratorService, які відносяться до GraphQL запиту;
- генератор GraphQL класів, що являють собою класи, успадковані від типу UnionGraphType – UnionsGeneratorService.

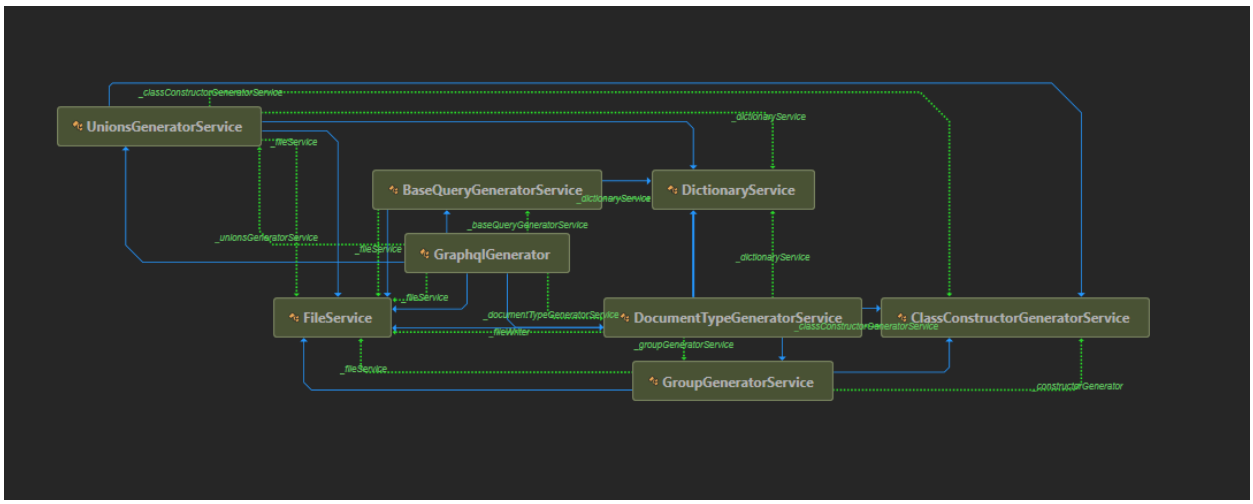


Рисунок 3.2 – Діаграма залежностей типів частин генератора

Генератори використовують сервіс-словник DictionaryService (рисунок 3.3), який збирає інформацію з XML файлів за допомогою сервісів-парсерів та надає опис того, які сторінки в CMS створено та який контент можна з них отримати.

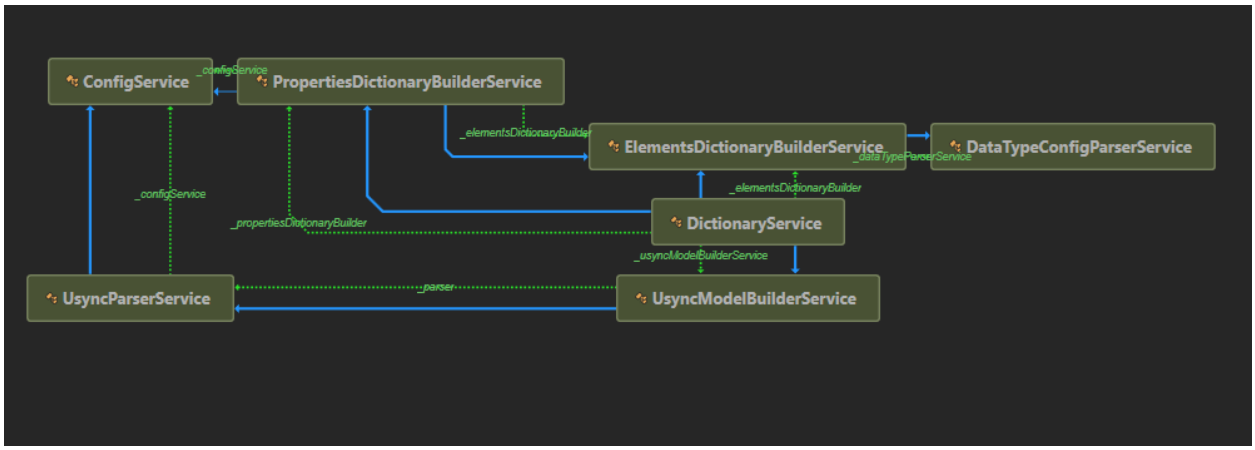


Рисунок 3.3 – Діаграма залежностей типів для DictionaryService

3.2 Проектування архітектури ПЗ

На базі вищезазначених вимог було побудовано схему архітектури системи. Програму реалізовано як консольний додаток, який виступає у

зв'язці з веб-додатком, який являє собою headless CMS з GraphQL API. Головна перевага даної архітектури – можливість швидкого оновлення схеми GraphQL API під час створення нових сторінок в CMS. Схему архітектури системи наведено нижче (рисунок 3.4).

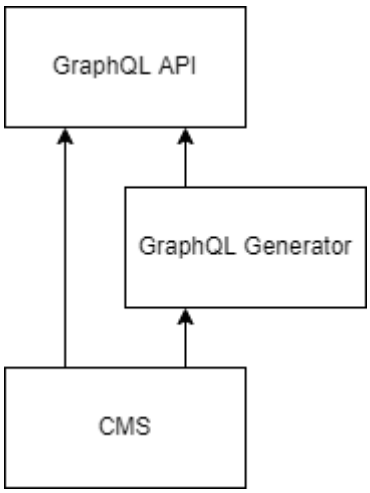


Рисунок 3.4 – Архітектура системи

Бізнес-логіка даної системи розподілена між сервісами, оскільки це дозволяє розподілити функціонал на незалежні блоки, які мають можливість працювати між собою.

4 ОПИС ПРИЙНЯТИХ РІШЕНЬ

4.1 Аналіз XML файлів

Для розробки генератора вихідного коду C# GraphQL API було використано платформу .NET Core.

Щоб створити GraphQL API для Umbraco CMS треба визначитись з тим, як отримувати інформацію про контент в цій системі керування вмістом. По перше, слід з'ясувати, які існують типи даних та як повз'язати ці типи з звичайними типами мови C#.

Гарним джерелом інформації може виступити спеціальний допоміжний NuGet пакет uSync. USync приймає інформацію з Umbraco CMS, яка зберігаються в базі даних, і переміщує її на диск (рисунок 4.1) для того, щоб її можна було копіювати та переміщувати між комп’ютерами та серверами. Цей пакет зберігає всю інформацію щодо сторінок, їх полів та типів полів в XML файлах. Такі файли розташовуються ієрархічно на диску, в такій самій структурі як і в CMS.

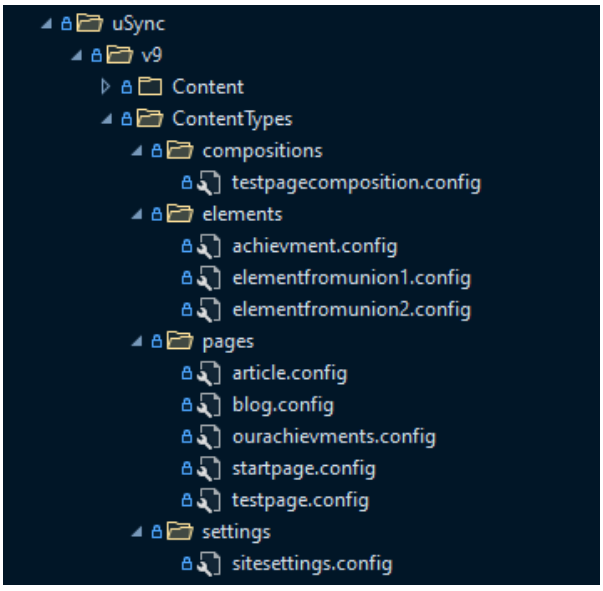


Рисунок 4.1 – Ієрархічна структура XML файлів uSync

Структура сторінки у вигляді XML файлу в Umbraco CMS представлена наступним чином (рисунок 4.2). В XML файлі присутня інформація щодо полів сторінки та їх типів, частин-композицій, з яких може складатися сторінка, директорії в якій збережена сторінка.

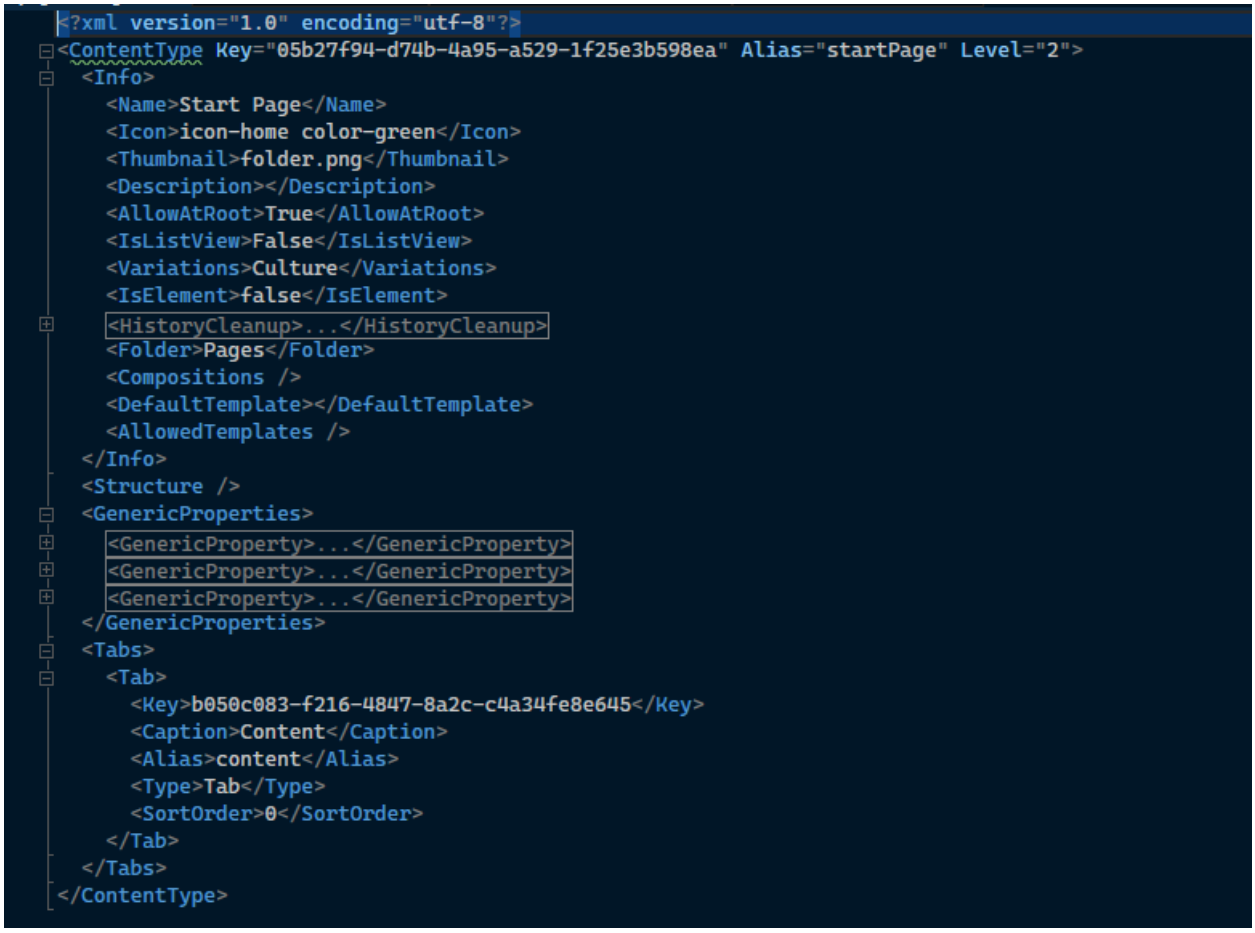


Рисунок 4.2 – XML структура сторінки Start Page

З цього XML файлу можна отримати інформацію про унікальну назву «alias» сторінки (startPage – Start Page) та її розташування – папку Pages. Також в цьому XML файлі представлено опис полів сторінки та типів (лістинг 4.1), які знаходяться в вузлах GenericProperty.

Перед початком написання парсеру XML файлів було переглянуто всі файли сторінок, їх поля та створено таблицю залежностей типів полей до типів C#:

Лістинг 4.1 – Поле Heading сторінки Start Page

```
<GenericProperty>
      <Key>6a253054-88e5-406f-8865-88d805ed011e</Key>
      <Name>Heading</Name>
      <Alias>heading</Alias>
      <Definition>0cc0eba1-9960-42c9-bf9b-
60e150b429ae</Definition>
      <Type>Umbraco.TextBox</Type>
      <Mandatory>false</Mandatory>
      <Validation></Validation>
      <Description><![CDATA[]]></Description>
      <SortOrder>0</SortOrder>
      <Tab Alias="content">Content</Tab>
      <Variations>Culture</Variations>
      <MandatoryMessage></MandatoryMessage>
      <ValidationRegExpMessage></ValidationRegExpMessage>
      <LabelOnTop>false</LabelOnTop>
</GenericProperty>
```

Типи елементів було поділено на дві групи – прості типи та кастомні типи.

До простих типів можна віднести всі типи, які напряму співвідносяться з типами C# GraphQL:

- Umbraco.TextBox – StringGraphType;
- Umbraco.MultipleTextstring – ListGraphType<StringGraphType>;
- Umbraco.TrueFalse – BooleanGraphType;
- Umbraco.TextArea – StringGraphType;
- Umbraco.Integer – IntGraphType;
- Umbraco.DateTime – DateTimeGraphType;
- Umbraco.ColorPicker – StringGraphType;
- Umbraco.CheckBoxList – ListGraphType<StringGraphType>;
- Umbraco.RadioButtonList – StringGraphType;
- Umbraco.ColorPicker.EyeDropper – StringGraphType.

Наприклад, розглянемо тип Umbraco.TextArea (рисунок 4.3).

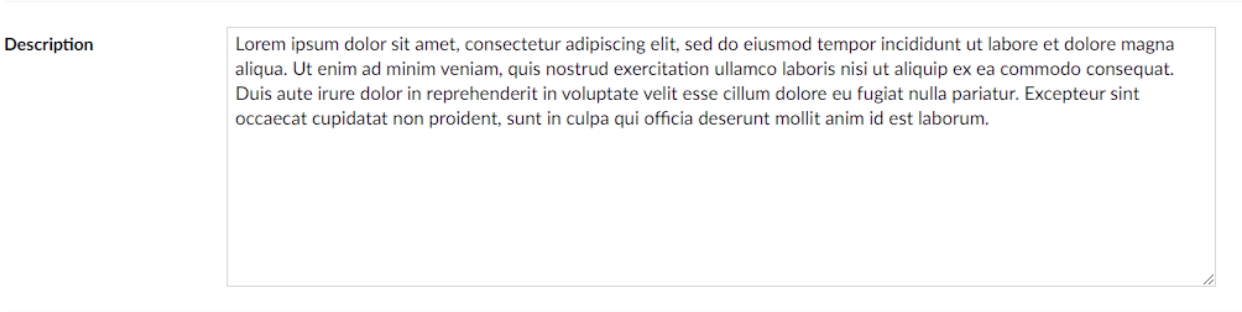


Рисунок 4.3 – Вигляд типу TextArea в CMS

Стає зрозуміло, що такий тип, область з текстом, легко можна перевести у тип string мови C# або у тип StringGraphType пакету GraphQL для створення API.

Кастомним типом є таке поле сторінки, яке не можливо напряму віднести до типу мови C#. Такий тип може містити у собі поля з простими або кастомними типами (рисунок 4.4). На кожен такий тип потрібно створити власний клас GraphQL.

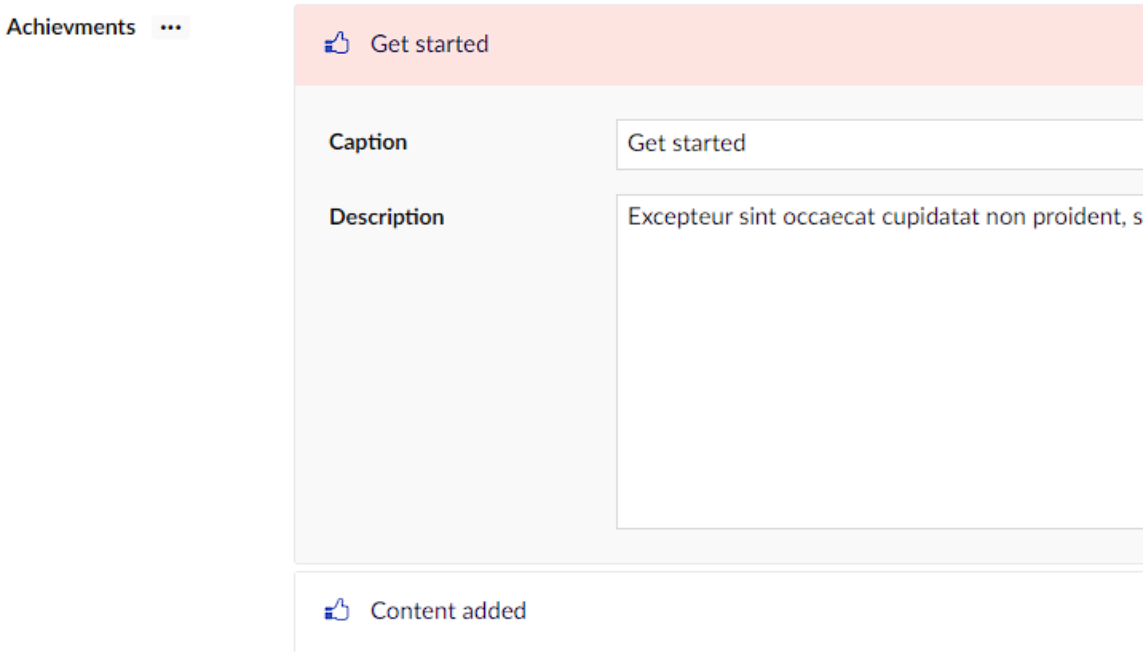


Рисунок 4.4 – Кастомний тип (поле з полями)

XML файл сторінки містить у собі опис того, яке поле на ній представлено. Наприклад, <Definition>93615251-9d13-4ab8-a188-4e125db31f77</Definition>, цей GUID є посиланням на унікальний ключ «key» кастомного типу даних (рисунок 4.5).

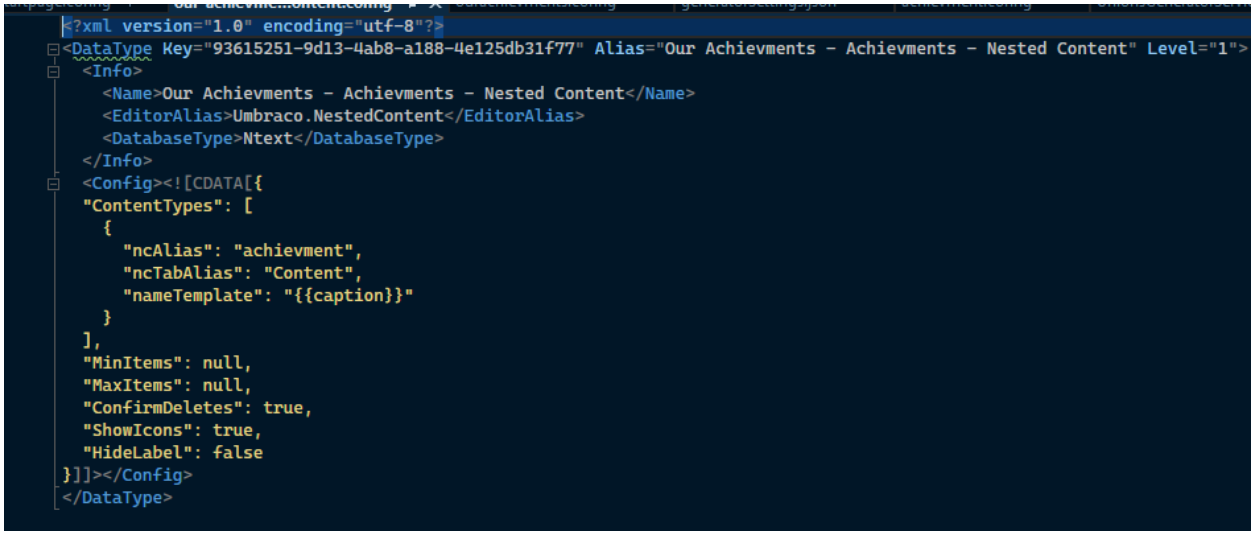


Рисунок 4.5 – XML файл кастомного типу даних

Можна побачити, що цей тип містить у собі посилання на DocumentType Umbraco CMS за допомогою його унікального імені (alias) – «achievment». XML файл «achievment» має таку ж саму структуру, як і звичайна сторінка (рисунок 4.2), тому його поля можна можна співвідности з типами C# GraphQL або власними створеними класами.

Таким чином було створено допоміжний сервіс DictionaryService, який за допомогою сервісів-парсерів DataTypeConfigParserService та UsyncParserService збирає інформацію про кожну сторінку з CMS та про кожне поле, яке присутнє на цій сторінці.

Така інформація представлена у вигляді звичайних DTO класів, які використовуються для передачі даних між підсистемами програми.

Нижче наведено клас-модель, який описує DocumentType з CMS (лістинг 4.2). У такому вигляді може бути представлено сторінку с полями, або кастомне поле зі своїм набором полей. Для того, щоб їх розрізнити

використовується поле IsElement.

Лістинг 4.2 – DTO клас сторінки

```
    public class GeneratorDocumentTypeModel
    {
        public string Alias { get; set; }
        public string Name { get; set; }
        public string Folder { get; set; }
        public bool IsElement { get; set; }
        public IEnumerable<GeneratorGroupModel> GroupsAliases {
get; set; }
    }
```

Всі поля зберігаються в класі-групі GeneratorGroupModel. Поля на сторінці представлено у вигляді наступної DTO моделі (лістинг 4.3):

Лістинг 4.3 – DTO клас поля сторінки

```
    public class GeneratorPropertyModel
    {
        public string Alias { get; set; }
        public string ClassName { get; set; }
        public PropertyTypeEnum PropertyType { get; set; }
    }
```

Тип PropertyTypeEnum представляє собою C# тип Enum у якому перечислено всі можливі види полів сторінки.

Логіка парсингу була написана за допомогою використання типу XmlDocument. Клас XmlDocument є представленням XML-документа в пам'яті. Він реалізує W3C XML Document Object Model (DOM) Level 1 Core і Core DOM Level 2. DOM означає об'єктну модель документа. Можна завантажити XML у DOM за допомогою класу XmlDocument, а потім програмно прочитати XML в документі [29].

4.2 Створення генератора

Після того, як усі XML файли пройшли процес парсингу та з них була отримана інформація про сторінки сайту, можна перейти до процесу створення генератора коду. В якості інструменту для створення автогенерації коду було обрано Roslyn. Генерація коду за допомогою Roslyn може бути досить багатослівною (особливо в порівнянні з CodeDom), але це не буде великою проблемою.

Схема GraphQL формується за допомогою класу, який успадковує клас Schema пакету GraphQL (лістинг 4.4).

Лістинг 4.4 – Базовий клас схема

```
    public class PageSchema : Schema
    {
        public PageSchema(IServiceProvider provider, BaseQuery
query) : base(provider)
        {
            Query = query;
        }
    }
```

Схема містить у собі поле Query, яке потрібно ініціалізувати за допомогою класу, який реалізує інтерфейс IObjectGraphType. ObjectGraphType представляє базовий клас за замовчуванням для всіх типів вихідних об’єктів графу (тобто тих, які мають власні поля).

Було прийнято рішення створити ієрархію класів з інтерфейсом IObjectGraphType. За допомогою Roslyn буде генеруватися файл класу BaseGeneratedQuery, який успадковує клас GraphQL ObjectGraphType, а його в свою чергу буде успадковувати клас BaseQuery, який буде написано власноруч. Такий клас потрібен для того, щоб додати до схеми GraphQL API деякі поля, які не можливо або дуже складно сгенерувати за допомогою автоматичного генератора коду.

Особливим випадком також є деякі класи, які описують поля сторінки

CMS. Таким полем є, наприклад, картинка, яка представлена типом Umbraco.MediaPicker, складність цього поля полягає в тому, що картинка містить у собі два поля – посилання на цю картинку та альтернативний текст. Рішенням для цієї проблеми виступає створення власноруч класу (лістинг 4.5), який би описав поля картинки, а потім, підключення цього класу за допомогою директиви using до класів, які будуть генеруватися автоматично.

Лістинг 4.5 – Клас для описання картинки

```
public class ImageGraphType: ObjectGraphType<IPublishedContent>
{
    public ImageGraphType()
    {
        Name = "image";

        Field<StringGraphType>("src",
            resolve: context => context.Source?.Url(mode:
UrlMode.Absolute));

        Field<StringGraphType>("alt",
            resolve: context =>
context.Source?.Value<string>("alternativeText"));
    }
}
```

Клас генератор було представлено у вигляді фасаду, який об’єднує в собі спеціальні класи генератори, які відповідають за генерацію своєї частини API. Наприклад, сервіс BaseQueryGeneratorService відповідає за генерацію класу BaseGeneratedQuery, у якому записані основні поля схеми API для отримання контенту зі сторінок CMS.

Головним методом цього класу є метод GenerateBaseQuery (лістинг 4.6) в який передаються моделі отримані від сервісу-словника.

Лістинг 4.6 – Метод генерації Query класу

```
    private string
GenerateBaseQuery(IEnumerable<GeneratorPageModel> pages)
    {
        string className = "BaseGeneratedQuery";

        var @namespace =
SyntaxFactory.NamespaceDeclaration(SyntaxFactory.ParseName(Const
ants.Generator.BasePageQueryNamespace));
        var classDeclaration =
SyntaxFactory.ClassDeclaration(className)

.AddModifiers(SyntaxFactory.Token(SyntaxKind.PublicKeyword))

.AddBaseListTypes(SyntaxFactory.SimpleBaseType(SyntaxFactory.Par
seTypeName("ObjectGraphType")));

        var variables = GetBasePageQueryVariables();
        foreach (var variable in variables)
        {
            classDeclaration =
classDeclaration.AddMembers(variable);
        }
        var ctor = GetBasePageQueryConstructor(className);
        classDeclaration = classDeclaration.AddMembers(ctor);

        var methods = GetBasePageQueryMethods(pages);
        foreach (var method in methods)
        {
            classDeclaration =
classDeclaration.AddMembers(method);
        }
        @namespace = @namespace.AddMembers(classDeclaration);
        string usings = string.Join("\r\n", pages.Select(x =>
x.Namespace));
        var code =
$"{Constants.BaseQueryUsings}\r\n{usings}\r\n\r\n" + @namespace
            .NormalizeWhitespace()
            .ToFullString();
        return code;
    }
```

Клас SyntaxFactory містить у собі дуже корисні для генерації методи, якими легко керувати, щоб отримати готовий код на виході. Наприклад, виклик метода NamespaceDeclaration відповідає за створення декларації простору імен.

Викликаючи метод ClassDeclaration можна створити декларацію класу, використовуючи допоміжні методи AddModifiers та AddBaseListTypes можна додати до декларації класу модифікатори доступу, наприклад, public, та додати успадкування від іншого класу. За допомогою виклику AddMembers до декларації класу можна додати поля або методи.

Для додавання декларації конструктора використовується виклик метода ConstructorDeclaration (лістинг 4.8), в який передається його назва. Так само, як і з декларацією класа, можна додати модификатори доступу за допомогою метода AddModifiers. Для додавання тіла конструктора використовується метод WithBody. Створення декларації конструктора виглядає наступним чином

Лістинг 4.7 – Декларація конструктора

```
        var classCtor =
SyntaxFactory.ConstructorDeclaration(className)
.AddModifiers(SyntaxFactory.Token(SyntaxKind.PublicKeyword))
.AddParameterListParameters(parameterList.ToArray())
.WithBody(SyntaxFactory.Block(ctorBodyParts));
```

Для додавання полів та методів відповідно використовуються VariableDeclaration та MethodDeclaration.

Створивши все необхідне, за допомогою метода AddMembers можна додати декларацію полів та методів до класу, а декларацію класу до простіру імен.

Наступним кроком для попередньо отриманого результату є використання методів .NormalizeWhitespace().ToFullString(), виклик яких дозволить отримати повний код класу у вигляді звичайної строки string.

Для того, щоб зберегти отриманий код до файлової системи було розроблено допоміжний сервіс FileService.

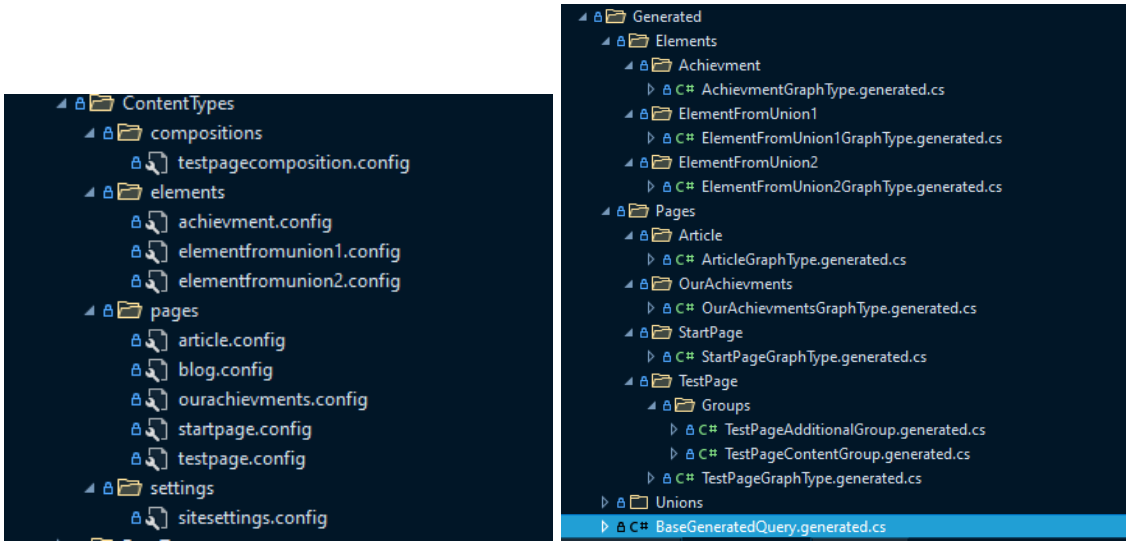
Виклик методу SaveGraphqlData (лістинг 4.8) цього сервісу дозволяє

зберегти отриману декларацію до файлової системи (рисунок 4.6).

Лістинг 4.8 – Метод збереження коду класа до файлової системи

```
    public void SaveGraphqlData(string outputFileName, string
folder, string content)
    {        string dataDirectory =
Path.Combine(Directory.GetParent(Directory.GetCurrentDirectory()
).Parent.Parent.Parent.FullName,
Constants.GraphqlProjectDirectory);
        string newDirectoryName = Path.Combine(dataDirectory,
Constants.GeneratedFolderName, folder);
        if (!Directory.Exists(newDirectoryName))
        {
            Directory.CreateDirectory(newDirectoryName);
        }
        string outputFilePath = Path.Combine(newDirectoryName,
outputFileName);
        File.WriteAllText(outputFilePath, content);
    }
```

Схожим чином створюються та зберігаються до файлової системи класи сторінок та класи кастомних полів.



а) б)

Рисунок 4.6 – Файлова структура: а) XML файли; б) сгенеровані файли

4.3 Використання отриманих результатів

Створений генератор коду гарантує те, що вся наявна інформація в XML файлах матиме своє відображення в коді, тобто майже вся інформація з системи керування контентом буде відображена в класах C#.

З іншого боку, створений автоматичний генератор коду не має можливості генерувати аблюлютно всю реалізацію класів GraphQL, відповідних до конфігурацій, зазначених в XML файлах.

Тим не менш, існуючий вариант, де програміст створює частину коду власноруч, а частину коду отримує за допомогою кодогенерації є надійним.

На практиці даний розроблений інструмент допомагає чітко отримати більшість необхідних класів GraphQL та оновити схему API. Більш того, отриману схему можна доповнити власноруч, додати спецефічні запити до API. Як показує досвід, в деяких випадках дивні та неочевидні конструкції програмного коду можуть бути зумовлені зайвою складністю або неточністю конфігурації в XML файлі.

Перевірити працездатність сгенерованого API можна за допомогою програми Postman. Потрібно туди передати схему API і Postman згенерує запити, які можна використовувати (рисунок 4.7) для отримання контенту з CMS.

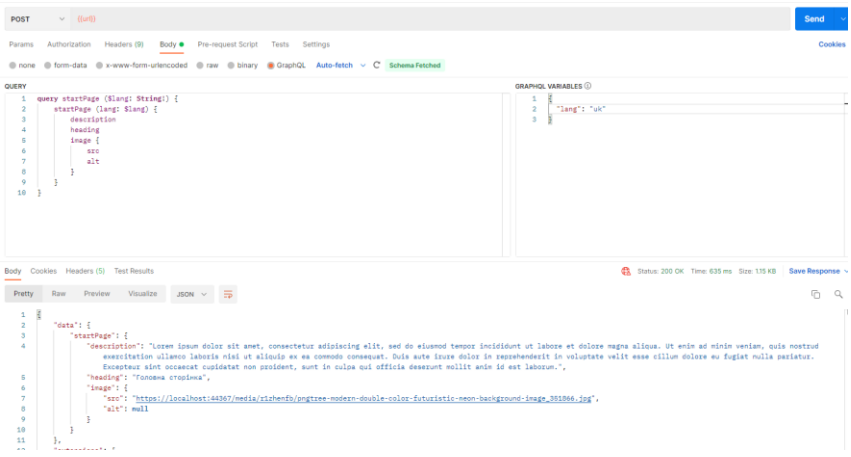


Рисунок 4.7 – Перевірка API за допомогою Postman

Цей контент представлено в системі керування вмістом наступним чином (рисунок 4.8):

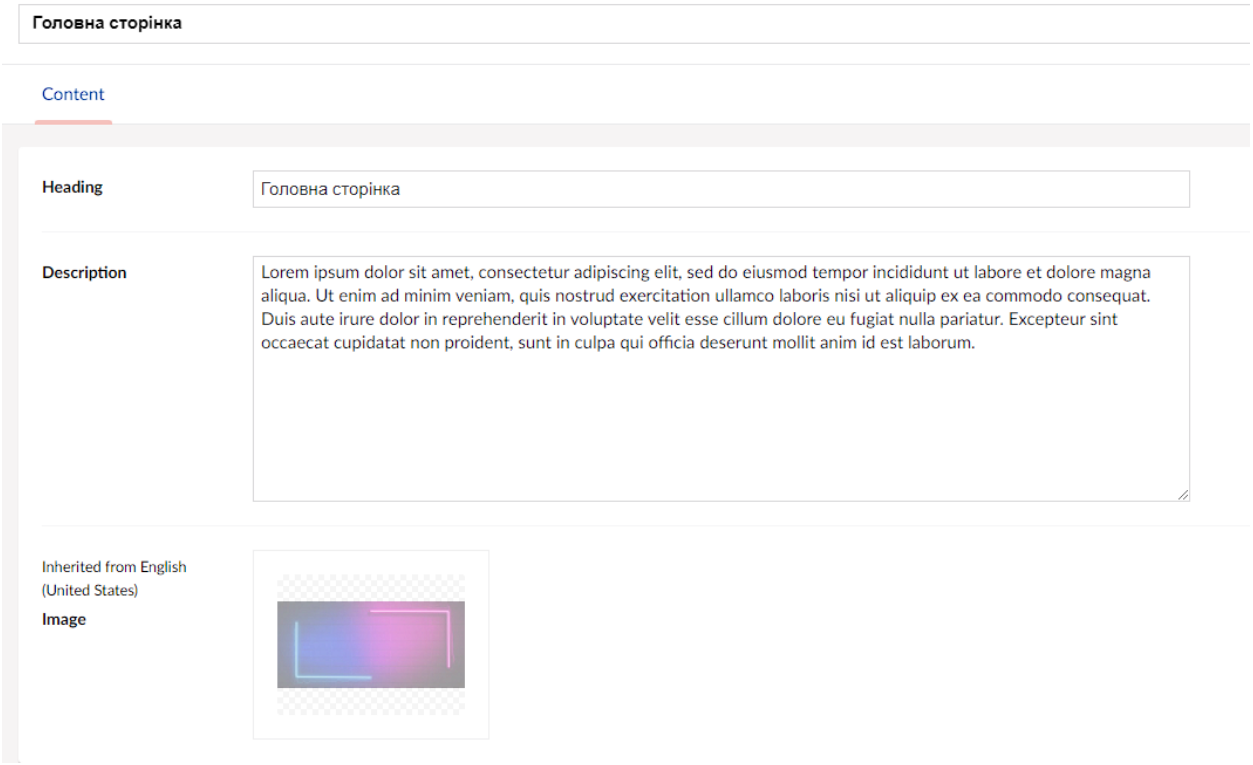


Рисунок 4.8 – Сторінка з контентом в CMS

У комп’ютерному програмуванні шаблонний код відноситься до розділів коду, які мають бути включені в багатьох місцях з незначними змінами або без них. Він часто використовується, коли йдеться про мови, які вважаються багатослівними, тобто програміст повинен написати багато коду, щоб виконувати мінімальну роботу [30].

Незважаючи на те, що код буде відрізнятися від класу до класу, він досить стереотипний за структурою, тому його краще генерувати автоматично, ніж писати вручну [30].

Слід зазначити, що розробка програмного забезпечення є доволі творчим процесом, і він не завжди підлягає чітким правилам. При використанні автоматичного генератора коду, можливо, втрачається індивідуальність, але з іншого боку – це надає можливість учасникам

процесу розробки програмного забезпечення не витрачати час на написання шаблонного коду, і витратити цей час на процес основної розробки.

На даному етапі цей інструмент здатний генерувати шаблонний код для простих сторінок в системі керування контентом, саме в тому вигляді, як представлені поля на самій сторінці, об'єднуючи їх у групи полів.

Генератор коду дозволяє створювати код саме для частини GraphQL API, але інфраструктурна частина коду все ще залишається відповідальністю розробника, бо неможливо чітко описати повну схему API.

В наукових дослідженнях даний інструмент можна використовувати в ті моменти, коли фахівець хоче зосередитися на дослідженні та структуризації певних процесів та сутноей, і після цього швидко створити шаблонний код, який потім буде доповнюватися.

Крім того, даний інструмент навчає гарній звичці, яка полегшить життя розробнику прорамного забезпечення – спочатку думати, а вже потім писати програмний код. Узагальнюючи, можна сказати, що автоматизація потрібна перш за все там, де чітко проглядаються ті чи інші шаблонні сутності та дії. В інших випадках – власноручна розробка виглядає більш привабливо.

ВИСНОВКИ

За результатами роботи було розроблено допоміжний інструмент, який можна використати для генерації шаблонного коду, а також для спрощення та прискорення розробки системи керування вмістом.

В роботі було проведено аналіз перспектив застосування засобів генерації вихідного коду: CodeDOM, IL injecting, T4 templates, snippets, Roslyn based source generators. В якості платформи для розробки власного генератора коду була обрана платформа .NET Core на мові C# в середовищі Visual Studio 2022. Як засіб генерації коду було обрано платформу Roslyn, тому що пов’язані з платформою пакети надають інструменти для створення спеціального шаблонного коду. Roslyn, як компілятор, канонічно знає все, що стосується синтаксису та семантики коду.

За результатами проведеного аналізу та досліджень було створено спеціалізований програмний продукт, інструмент, що призначений для автоматичної генерації коду, а саме C# класів, які у спільному використанні з власноруч написаними базовими класами можуть використовуватися для дуже швидкого створення та оновлення схеми GraphQL API у веб-додатку, який являє собою систему керування вмістом Umbraco CMS.

Наукова новизна результатів роботи полягає в створенні власного генератора коду за рахунок використання спеціалізованого інструмента компілятора .NET, також відомого під кодовою назвою Roslyn, який представляє собою набір компіляторів з відкритим вихідним кодом і API аналізу коду для мов C# і Visual Basic (VB.NET) від Microsoft.

Практичне значення отриманих результатів полягає в запропонованих технологічних рішеннях створення файлів коду C# зі звичайних файлів формату XML, які можуть бути використані розробниками програмного забезпечення при розробці власних рішень подібного спрямування.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. What is parsing? [Електронний ресурс] – Режим доступу до ресурсу: https://stackoverflow.com/questions/1788796/what-is-parsing/.

2. Машинно-залежна оптимізація в компіляторах [Електронний ресурс] – Режим доступу до ресурсу: https://core.ac.uk/download/pdf/55297225.pdf.

3. Abstract Syntax Tree [Електронний ресурс] – Режим доступу до ресурсу: https://deepsource.io/glossary/ast/.

4. Кодогенерация в Visual Studio [Електронний ресурс] – Режим доступу до ресурсу: https://itnan.ru/post.php?c=1&p=645267.

5. What is a Headless CMS? [Електронний ресурс] – Режим доступу до ресурсу: https://umbraco.com/knowledge-base/headless-cms/.

6. What is GraphQL API? [Електронний ресурс] – Режим доступу до ресурсу: https://umbraco.com/knowledge-base/graphql-api/.

7. Введение в GraphQL: что это за язык и как использовать его под Android [Електронний ресурс] – Режим доступу до ресурсу: https://dou.ua/lenta/articles/working-with-qraphql/.

8. Source Generators - real world example [Електронний ресурс] – Режим доступу до ресурсу: https://dominikjeske.github.io/source-generators/.

9. Introduction to Spring AOP [Електронний ресурс] – Режим доступу до ресурсу: https://www.baeldung.com/spring-aop/.

10. Создание кода при помощи CodeDOM [Електронний ресурс] – Режим доступу до ресурсу: https://www.smarly.net/metaprogramming-in-net/techniques-for-generating-code/generating-code-with-the-codedom/.

11. What is Metaprogramming? [Електронний ресурс] – Режим доступу до ресурсу: https://www.easytechjunkie.com/what-is-metaprogramming.htm/.

12. Microsoft .NET CodeDom Technology [Електронний ресурс] – Режим доступу до ресурсу: https://www.codeguru.com/visual-basic/microsoft-net-

codedom-technology/.

13. Лекция 7: Генерация объектно-ориентированного кода. Технология CodeDom [Електронний ресурс] – Режим доступу до ресурсу: https://intuit.ru/studies/courses/3733/975/lecture/14627.

14. Introducing C# Source Generators [Електронний ресурс] – Режим доступу до ресурсу: https://devblogs.microsoft.com/dotnet/introducing-c-source-generators/.

15. Пізнє та раннє зв’язування [Електронний ресурс] – Режим доступу до ресурсу: https://www.bestprog.net/uk/2020/04/10/c-late-and-early-binding-polymorphism-basic-concepts-examples-passing-a-reference-to-a-base-class-in-a-method-ua/#q01.

16. 4 ways to generate code in C# — Including Source Generators in .NET 5 [Електронний ресурс] – Режим доступу до ресурсу: https://levelup.gitconnected.com/four-ways-to-generate-code-in-c-including-source-generators-in-net-5-9e6817db425?gi=fbc7e939b350.

17. Code Weaving in .NET using Fody [Електронний ресурс] – Режим доступу до ресурсу: https://codingcanvas.com/code-weaving-using-fody/.

18. IL weaving [Електронний ресурс] – Режим доступу до ресурсу: https://michielsioen.be/2017-10-21-il-weaving/.

19. Design-Time Code Generation by using T4 Text Templates [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/ru-ru/visualstudio/modeling/design-time-code-generation-by-using-t4-text-templates?view=vs-2022.

20. Code generation/scaffolding with Visual Studio T4 templates [Електронний ресурс] – Режим доступу до ресурсу: https://www.tritac.com/nl/blog/code-generationscaffolding-with-visual-studio-t4-templates/.

21. Net Code Generation. Part 6. C# Source Generators [Електронний ресурс] – Режим доступу до ресурсу: https://mihailromanov.wordpress.com/2021/01/31/net-code-generation-part-6-c-

source-generators/.

22. MSBuild [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild?view=vs-2022.

23. Incremental Roslyn Source Generators In .NET 6: Code Sharing Of The Future - Part 1 [Електронний ресурс] – Режим доступу до ресурсу: https://www.thinktecture.com/en/net/roslyn-source-generators-introduction/.

24. Федорченко В.М., Носов В.С. Порівняльний аналіз засобів генерації вихідного коду програмного забезпечення в .NET : зб. тез доп. XV Всеукраїнська науково практична WEB конференція аспірантів, студентів та молодих вчених, м. Кривий Ріг, 22-24 березня 2022 р. Кривий Ріг, 2022. С. 73-75.

25. Code generation [Електронний ресурс] – Режим доступу до ресурсу: https://www.jetbrains.com/help/rider/Code_Generation__Index.html.

26. Code snippets [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/visualstudio/ide/code-snippets?view=vs-2022.

27. What is Sequence Diagram? [Електронний ресурс] – Режим доступу до ресурсу: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/.

28. Design Patterns - Facade Pattern [Електронний ресурс] – Режим доступу до ресурсу: https://www.tutorialspoint.com/design_pattern/facade_pattern.htm.

29. XmlDocument Class [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmldocument?view=net-6.0.

30. What is boilerplate and why do we use it? [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/free-code-camp/whats-boilerplate-and-why-do-we-use-it-let-s-check-out-the-coding-style-guide-ac2b6c814ee7.