Вінницький національний технічний університет

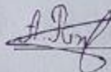
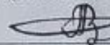
Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Пояснювальна записка

до бакалаврської дипломної роботи на тему:

Клієнт-серверна інформаційна система підрозділу навчального закладу з можливістю розгортання в хмарному середовищі. Частина 2. «Розробка розподіленого слабкозв'язаного серверного додатку мовою С#»

Виконав: студент 2 курсу, групи 1КІ-20мс
спеціальності 123 — Комп'ютерна інженерія
(шифр і назва напряму підготовки, спеціальності)
 Рижков А.К.
(прізвище та ініціали)
Керівник к.т.н., доцент каф. ОТ
 Войцеховська О.В.
(прізвище та ініціали)
«13» 06 2022 р.
Рецензент к.т.н., ст.викладач каф. ЗІ
Лукічов В.В.
(прізвище та ініціали)
«14» 06 2022 р.

Допущено до захисту
Зав. кафедри д.т.н., проф. О.Д. Азаров
«15» 06 2022 р.

Вінниця ВНТУ — 2022 рік

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Освітньо-кваліфікаційний рівень перший (бакалаврський)

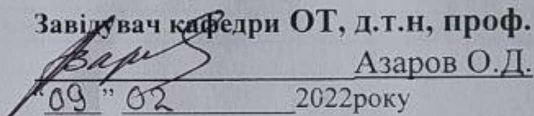
Галузь знань — 12 — Інформаційні технології
(шифр і назва)

Спеціальність — 123-«Комп'ютерна інженерія»
(шифр і назва)

Освітньо — професійна програма — Комп'ютерна інженерія
(Назва освітньо – професійної програми)

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ, д.т.н, проф.



Азаров О.Д.

"09" 02 2022року

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Рижкову Андрію Костянтиновичу

1 Тема проекту (роботи) Клієнт-серверна інформаційна система підрозділу навчального закладу з можливістю розгортання в хмарному середовищі. Частина 2. «Розробка розподіленого слабкозв'язаного серверного додатку мовою C#» керівник роботи Войцеховська Олена Валеріївна, затверджені наказом вищого навчального закладу від 24.03.2022 № 66

2 Термін подання студентом роботи 14.06.22

3 Вихідні дані до роботи: контент для наповнення бази даних інформаційної системи структурного підрозділу навчального закладу, технології реалізації серверної частини

4 Зміст текстової частини: аналіз сучасних технологій для розробки серверної частини інформаційної системи. Розробка серверної частини інформаційної системи структурного підрозділу навчального закладу.

Проектування високорівневої архітектури та реалізація існтрументарію серверної частини інформаційної системи

5 Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): Структурна схема високорівневої архітектури клієнт-серверної інформаційної системи, діаграма класів розподіленого слабкозв'язаного серверного додатку, схема взаємодії запиту GraphQl із моделями даних, структура бази даних слабкозв'язаного серверного додатку, мапа веб-сайту

6 Консультанти розділів роботи приведено в таблиці 1.

Таблиця 1 — Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада гонсультанта	Підпис, дата	
		видав	прийняв
1-3	Войцеховська О. В., к.т.н., доцент каф. ОТ	[signature]	[signature]

7 Дата видачі завдання 08.02.2022.

8 Календарний план виконання МКР приведений в таблиці 2.

Таблиця 2 — Календарний план

№ з/п	Назва етапів дипломного проекту (роботи)	Термін виконання		Примітка
		початок	закінчення	
1	Аналіз завдання	08.02.22		Виконав
2	Огляд архітектурних підходів проектування інформаційних систем	09.02.22	12.02.22	Виконав
3	Аналіз та вибір технологій для роботи із базою даних та вибір СУБД	12.02.22	20.02.22	Виконав
4	Огляд технологій вибірки даних, документації коду та вибір хмарного середовища	20.02.22	28.02.22	Виконав
5	Проектування високорівневої архітектури серверної частини веб-додатку	29.02.22	14.03.22	Виконав
6	Проектування бази даних серверного додатку	15.03.22	21.03.22	Виконав
7	Розробка та програмна реалізація рівня доступу до даних серверної частини	22.03.22	20.04.22	Виконав
8	Розробка та програмна реалізація рівня бізнес-логіки серверної частини	21.04.22	15.05.22	Виконав
9	Перевірка працездатності серверної частини веб-додатку	16.05.22	30.05.22	Виконав
10	Оформлення пояснювальної записки та ілюстративного матеріалу	01.06.22	13.06.22	Виконав
11	Перевірка якості виконання бакалаврської роботи та усунення недоліків	14.06.22		Виконав

Студент ____________ Рижков А. К.
Керівник роботи ____________ Войцеховська О. В.

АНОТАЦІЯ

Комплексна бакалаврська дипломна робота складається з 93 сторінки формату А4, на яких є 30 рисунків, перелік джерел посилання містить 13 найменувань.

В бакалаврській дипломній роботі проведено аналіз технологій для роботи з базою даних безпосередньо через середовище .NET, документування коду, технологій вибірки даних та обрано хмарне сховище.

Розроблено слабкозв’язаний серверний додаток для клієнт-серверної інформаційної системи підрозділу навчального закладу мовою C# з допомогою сучасних технологій розробки програмного забезпечення. Використано архітектурний підхід, при якому роботу над різними рівнями інформаційної системи можна розділити на декількох розробників.

Описано високорівневу архітектуру розроблюваного проекту та інструментарій серверної частини для комунікації рівня представлення та рівня бази даних в межах цієї архітектури.

Ключові слова: .NET, C#, Entity Framework, GraphQl, Swagger.

ABSTRACT

The complex bachelor's thesis consists of 72 A4 pages, which contain 30 figures, the list of reference sources contains 13 titles. In the bachelor's thesis the analysis of technologies for work with a database directly through the .NET environment, documentation of the code, technologies of data sampling is carried out and cloud storage is chosen.

A loosely coupled server application has been developed for the client-server information system of the educational unit of the educational institution in C # with the help of modern software development technologies. An architectural approach is used, in which work on different levels of the information system can be divided into several developers.

The high-level architecture of the developed project and the tools of the server part for communication of the presentation level and the database level within this architecture are described.

Keywords: .NET, C#, Entity Framework, GraphQl, Swagger.

ЗМІСТ

ВСТУП 8

1 АНАЛІЗ СУЧАСНИХ ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СЕРВЕРНОЇ ЧАСТИНИ ІНФОРМАЦІЙНОЇ СИСТЕМИ 10

1.1 Аналіз використання багаторівневої архітектури при побудові інформаційної системи 10

1.2 Аналіз сучасних ORM технологій доступу до даних 12

1.2.1 Огляд та особливості налаштування технології Entity Framework Core 12

1.2.2 Аналіз технології ADO.NET 14

1.3 Архітектурний підхід REST для покращеного використання протоколу HTTP 15

1.4 Огляд технологій для документування програмного коду 18

1.4.1 Огляд технології Swagger 18

1.4.2 Огляд технології Postman 19

1.5 Огляд технології для вибірки даних GraphQL 20

1.6 Обгрунтування вибору хмарного сховища 21

1.6 Вибір системи управління базами даних 22

2 РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ ІНФОРМАЦІЙНОЇ СИСТЕМИ СТРУКТУРНОГО ПІДРОЗДІЛУ НАВЧАЛЬНОГО ЗАКЛАДУ 25

2.1 Аналіз вимог до клієнт-серверної інформаційної системи та розробка мапи веб-додатку 25

2.2 Проектування інформаційної системи з використанням технології Web-API 26

2.2.1 HTTP коди статусів при проектуванні серверної частини 27

2.2.2 Моделі даних у ASP.NET Web API 29

2.3 Проектування бази даних у інформаційній системі 31

2.3.1 Опис моделі даних Entity Framework 33

2.3.2 Опис контексту даних Entity Framework для зв’язку із базою даних. 35

2.3.3 Основні операції з даними у Entity Framework 36
2.3.4 Використання шаблону «Repository» для маніпуляцій над даними 38
2.4 Використання технології Swagger для опису API у веб-додатку 39
2.5 Реалізація роботи з Google Cloud через платформу .Net 42
2.6 Реалізація технології вибірки даних GrapQL у веб-додатку 48
2.6.1 Опис моделі запиту GraphQL 49
2.6.2 Опис типів GraphQL 51
2.6.3 Конфігурація GraphQL 52
3 ПРОЕКТУВАННЯ ВИСОКОРІВНЕВОЇ АРХІТЕКТУРИ ТА РЕАЛІЗАЦІЯ ІНСТРУМЕНТАРІЮ СЕРВЕРНОЇ ЧАСТИНИ ІНФОРМАЦІЙНОЇ СИСТЕМИ 55
3.1 Проектування високорівневої архітектури серверної частини веб-додатку 55
3.2 Відправлення запитів на серверну частину веб додатку 59
ВИСНОВКИ 65
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 66
ДОДАТОК А Технічне завдання 68
ДОДАТОК Б Структурна схема високорівневої архітектури клієнт-серверної інформаційної системи 72
ДОДАТОК В Діаграма класів розподіленого слабкозв’язаного серверного додатку 73
ДОДАТОК Г Схема взаємодії запиту GraphQl із моделями даних 74
ДОДАТОК Д Структура бази даних слабкозв’язаного серверного додатку . 75
ДОДАТОК Е Мапа веб-сайту 76
ДОДАТОК Ж Лістинг інтерфейсу NewsController 77
ДОДАТОК К Лістинг інтерфейсу TeacherController 82
ДОДАТОК Л Лістинг інтерфейсу NewsService 87
ДОДАТОК М Лістинг інтерфейсу TeacherService 90
ДОДАТОК Н Протокол перевірки кваліфікаційної роботи на наявність текстових запозичень 93

						Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		

ВСТУП

Зараз, коли важко уявити світ без інтернету, університети борються за увагу абітурієнта саме у глобальній мережі. Перш за все, потрібно привернути увагу користувача до сайту, щоб він залишився на ньому та вивчав корисну інформацію. Окрім цього, якщо швидкодія веб-додатку буде низькою, користувач просто покине сайт, та не буде чекати.

Саме за це відповідає серверна частина веб-додатку. Тому, важливо дотримуватись певних правил, при написанні серверної частини веб-додатку. Перш за все — потрібно використовувати оптимальні технології, які можуть виконати поставлені задачі, при цьому не навантаживши сервер, та зможуть швидко надати відповідь на запит. Окрім цього, серверна частина повинна бути легкою для розширення, адже зазвичай функціональні можливості інформаційної системи часто змінюються. Також важливо, щоб серверна частина була зрозуміла для розробників рівня представлення та адміністративної панелі, адже саме вони будуть взаємодіяти із функціональними можливостями, описаними у API.

Тому актуальним є використання сучасних технологій, які здатні забезпечити гнучкість, високу швидкодію та чистоту коду при реалізації серверної частини веб-додатку.

Метою роботи є розробка серверної частини веб-додатку інформаційної системи структурного підрозділу навчального закладу, яка буде складатись із бази даних для зберігання інформації, серверної частини для обробки інформації та зберігання файлів у хмарному середовищі Google Cloud.

Для досягнення поставленої мети потрібно виконати такі **завдання**:

— проаналізувати архітектурні підходи до написання розподілених веб-додатків;

— провести аналіз існуючих технологій розробки серверної частини веб-додатків та визначити необхідний стек технологій для розробки;

— розробити мапу веб-додатку структурного підрозділу навчального закладу;

— розробити структуру бази даних у середовищі .NET, визначити структуру об’єктів та основні сутності;

— описати логіку функціонування серверної частини веб-додатку;

— провести тестування роботи розробленої серверної частини.

Об’єктом дослідження є процес створення серверної частини веб-додатку із використанням нових перспективних технологій, які зменшать навантаження на сервер.

Предметом дослідження є сучасні технології проектування серверної частини інформаційних систем з можливістю розгортання в хмарному середовищі.

Методами дослідження є використання архітектурного підходу REST, та принципів об’єктно орієнтованого програмування.

Апробація результатів бакалаврської роботи: зроблено доповіді на LI науково-технічній конференції підрозділів ВНТУ та на Всеукраїнській науково-практичній інтернет-конференції «Молодь в науці: дослідження, проблеми, перспективи» [1, 2].

Публікації за темою роботи: Аналіз архітектури розподіленого слабкозв’язного серверного додатку інформаційної системи / Войцеховська О. В., Рижков А. К. //Матеріали LI наукової-технічної конференції підрозділів ВНТУ, 2022 р. Режим доступу: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2022/paper/view/15103/12731

Використання патерну «Repository» при проектуванні розподіленого слабкозв’язного серверного додатку // Войцеховська О. В., Рижков А. К. // Матеріали Всеукраїнської науково-практичної інтернет-конференції «Молодь в науці: дослідження, проблеми, перспективи", 2022 р. Режим доступу: https://conferences.vntu.edu.ua/index.php/mn/mn2022/author/submission/16266

1 АНАЛІЗ СУЧАСНИХ ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СЕРВЕРНОЇ ЧАСТИНИ ІНФОРМАЦІЙНОЇ СИСТЕМИ

При проектуванні великої інформаційної системи важливим є оптимально визначений стек технологій, які будуть використовуватись. Окрім цього, сучасні технології та архітектурні підходи потрібні для полегшення написання коду та для можливості зробити систему більш гнучкою і здатною до подальшого розширення.

Отже, необхідно провести аналіз існуючих технологій, для виконання завдання, порівняти їх та обрати найбільш оптимальні.

1.1 Аналіз використання багаторівневої архітектури при побудові інформаційної системи

Інформаційна система — це великий проект, який потрібно використовувати та підтримувати тривалий час. Тому інформаційні системи прийнято ділити на рівні, при чому крайні рівні не можуть взаємодіяти між собою (наприклад рівень представлення не може напряму звертатися до бази даних). Це дозволяє зменшити зв’язаність, що дозволить змінювати частини проекту незалежно одна від одної. Внаслідок цього покращується масштабованість проекту. Також це збільшує можливості для тестування, що дозволяє зменшити кількість помилок та покращити стабільність роботи додатку. Зазвичай багаторівнева архітектура включає в себе:

- рівень доступу до даних — на ньому відбувається робота з базою даних, зокрема CRUD операції, вибірка даних;
- рівень бізнес-логіки — на ньому агрегуються дані, обробляються та перевіряються на відповідність обмеженням та переформатовуються в об’єкти передачі даних, що містять лише необхідні для певної операції дані;
- рівень представлення — клієнтські додатки, що забезпечують зручну роботу з системою для користувачів.

Ця структура може змінюватись залежно від потреб користувачів. Приклад багаторівневої архітектури приведено на рисунку 1.1.

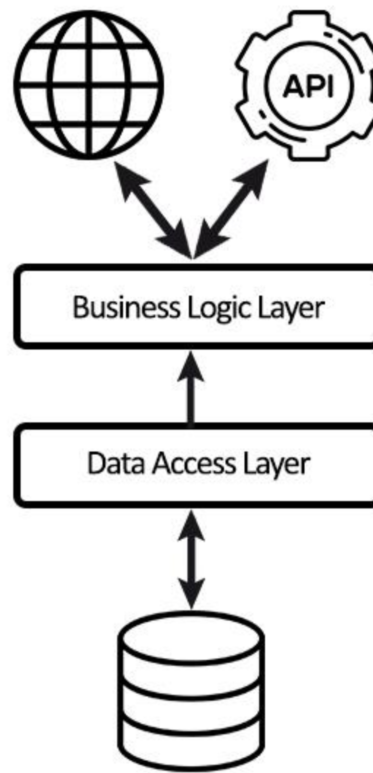


Рисунок 1.1 — Схема багаторівневої архітектура

При розробці невеликого проекту, на якому працювала б одна людина, можна було не звертати увагу на архітектуру додатку. Але у нашому випадку — це відносно великий проект, до розробки якого залучено чотири людини.

Застосування такої архітектури, в першу чергу, допомагає вирішити проблему взаємодії розробників під час написання коду. Окрім того, із таким підходом проект стає більш гнучкий та легший до розширення. Якщо, наприклад, потрібно внести додаткову логіку для обробки даних, зміни зачеплять лише рівень бізнес логіки, а весь інший код залишиться без змін [1].

Також з використанням багаторівневої архітектури розробники можуть працювати паралельно. Наприклад, під час написання основної логіки

серверної частини, можна створити копію даних (mocker) основних даних, і віддати його розробникам, які працюють над рівнем представлення. Розробники рівня представленя можуть працювати із шаблоном даних, які в майбутньому будуть надходити із серверної частини, поки бекенд знаходиться у розробці.

1.2 Аналіз сучасних ORM технологій доступу до даних

1.2.1 Огляд та особливості налаштування технології Entity Framework Core

Технологія Entity Framework Core (EF Core) являє собою об’єктно орієнтовану технологію від компанії Microsoft для доступу до даних. EF Core — це ORM-інструмент (object-relational mapping — відображення даних на реальні об’єкти). Тобто EF Core дозволяє працювати з базами даних (БД), але являє собою більший рівень абстрактності, зокрема можна абстрагуватись від самої БД і її таблиць і працювати з даними незалежно від типу сховища. Якщо на фізичному рівні робота відбувається із таблицями, індексами та ключами то на концептуальному рівні, який надає EF Core робота відбувається вже з об’єктами.

Entity Framework Core підтримує різні системи баз даних. Таким чином через EF можна працювати з будь-якою СУБД, якщо для неї є потрібний провайдер. За замовчуванням на даний момент Microsoft надають ряд вбудованих провайдерів для роботи із PostgreeSQL, SQLite та MS SQL Server [3].

Перевагою Entity Framework Core є те, що він являє собою універсальний API для роботи з даними. Якщо, наприклад, є потреба змінити СУБД, то основні зміни у проекті будуть відноситись в першу чергу до конфігурації та налаштування підключення до потрібних провайдерів. А код, який безпосередньо працює з базою даних, отримує данні, додає їх у БД і т.п. залишиться незмінним.

Як технологія доступу до даних, EF Core може використовувати різні платформи стеку .NET Core. Це і стандартні платформи типу Windows Forms, консольні додатки, WPF (Windows Presentation Foundation — підсистема для побудови графічних інтерфейсів), UWP (Universal Windows Platform — платформа для створення та запуску додатків у Windows 10) та ASP.NET Core. При цьому кросплатформеність технології EF Core дозволяє задіяти її не тільки для Windows, але і для Linux та Mac OS.

Центральною концепцією Entity Framework є поняття сутності, яка визначає набір даних, пов'язаних з певним об'єктом. Тому дана технологія пропонує роботу не з таблицями, а з об'єктами та їх колекціями.

Однією із переваг використання технології Entity Framework Core є її зручність при розробці гнучких інформаційних систем. Підхід коли розробник може працювати безпосередньо із об'єктами, замість того щоб описувати запити мовою SQL, є доволі зручним.

В першу чергу, при роботі з цією технологією необхідно описати модель об'єкту. В такому вигляді вона буде створена у базі даних, а типи даних, описані у моделі, будуть співставленні із релевантними в базі даних. Окрім цього, програма повинна містити контекст даних, який буде зберігати у собі усі таблиці. Наприклад, якщо створити колекцію DbSet типу User, це буде відповідати таблиці User у базі даних, яка буде обрана розробником [4].

Також, важливим елементом є стрічка підключення до бази даних. Саме вона містить основну інформацію, таку як назва бази даних та який тип бази даних використовується. Окрім цього, є низка додаткових параметрів, які можна налаштувати за необхідності, такі як надійне підключення та облікові данні для захисту бази даних. Загальний вигляд стрічки підключення до бази даних подано в лістингу 1.1

Лістинг 1.1 — Загальний вигляд стрічки підключення

```
Server=(localdb)\\mssqllocaldb;Database=helloappdb;Trusted_C
onnection=True;
```

З лістингу 1.1 видно, що якщо використовувати конкретно цю стрічку підключення, то програмі передається інформація про використання бази даних MS SQL, з назвою «helloappdb», а також додаткового параметру, який означає що буде використовуватись безпечне підключення [5].

Особливістю використання технології Entity Framework Core є те, що у будь який момент, коли розробник вирішить, що використання будь-якої іншої бази даних є більш доцільним, він може всього лише замінити стрічку підключення, в яку включить інший тип бази даних, а весь код, який відповідає за створення та основну обробку даних залишиться незмінним. Технологія зробить усю роботу по створенню нової бази даних, та нових таблиць автоматично.

1.2.2 Аналіз технології ADO.NET

ADO.NET — це набір класів, розроблений компанією Microsoft, які надають послуги доступу до даних. ADO.NET підтримує різноманітні потреби у розробці, включаючи створення клієнтських баз даних і бізнес об’єктів.

Основною перевагою ADO.NET є швидкість отримання даних із бази даних. Це важливо для великих систем, у базах даних яких зберігається багато складної інформації, яка пов’язана між собою. Тому наприклад, при вибірці даних, при написанні великого запиту, який проходить через декілька сутностей технологія ADO.NET поверне данні швидше порівняно з описаною вище технологією Entity Framework Core. Але окрім цього, ця технологія має два суттєвих недоліки. [6].

Першим недоліком є відсутність кросплатформеності для технології ADO.NET, тому сервер, на якому буде розгортатись інформаційна система має чіткі обмеження, щодо операційної системи та системних вимог. Другим недоліком використання технології ADO.NET є неможливість працювати із сутностями, такими як б’єкти у мові програмування C#. Для роботи із ADO.NET необхідно прямо у коді C# описувати запити на мові SQL, та ініціювати їх виконання. В той час як відмінною рисою використання саме

технології Entity Framework, як технології ORM, є використання запитів LINQ для вибірки даних із бази даних. За допомогою LINQ, яка існує у C# за замовчуванням, можна створювати різні запити на вибірку об'єктів, у тому числі і пов'язаних із різними асоціативними зв'язками. В свою чергу технологія Entity Framework автоматично транслює запит, описаний через мову програмування C# у вирази, які зрозумілі конкретній СУБД, і це як правило SQL.

1.3 Архітектурний підхід REST для покращеного використання протоколу HTTP

Архітектурний підхід REST — це архітектурний стиль взаємодії компонентів розподіленого додатку у мережі. Ця абривіатура розшифровується як Representational State Transfer, тобто передача стану представлення. Даний стиль взаємодії являє собою узгоджений набір обмежень, що враховуються при проектуванні розподіленної системи. У певних випадках це приводить до підвищення продуктивності та покращення архітектури. Можна сказати, що компоненти у REST взаємодіють між собою, як взаємодіють клієнти з серверами.

Такий підхід дає можливість отримати:

— масштабованість взаємодії компонентів системи;
— спільність інтерфейсів;
— незалежне впровадження компонентів;
— проміжні компоненти, які знижують затримку і посилюють безпеку.

Використовувати такий підхід необхідно коли:

— є обмеження на пропускну здатність з'єднання;
— є необхідність кешувати запити;
— система передбачає значне масштабування;
— є сервіси, що використовують AJAX.

За рахунок використання цієї технології відпадає необхідність у додаткових прошарках, що дозволяє передавати дані у такому вигляді, як вони зберігаються у сховищі. Тобто, дані не потрібно обертати у XML та не потрібно використовувати AMF(Action Message Format). Відбувається просто передача інформації.

Керування інформацією ресурса базується на протоколі передачі даних HTTP (HyperText Transfer Protocol). Для HTTP дія над даними задається з допомогою методів: GET, PUT, POST, DELETE. Маніпуляції CRUD(Create Read Update Delete) можуть виконуватись як лише з допомогою двох методів GET та POST, так і як з усіма чотирма методами [7].

Щоб розподілена система вважалася сконструйованою за REST архітектурою, тобто була Restful, вона повинна задовольняти таким критеріям:

— Client-Server (клієнт-серверна архітектура), тобто система повинна бути розділена на клієнтську та серверну частини;

— Stateless (безстановість), тобто серверна частина не повинна зберігати будь-яку інформацію про клієнтську частину, при цьому у запиті має зберігатися вся необхідна інформація для її обробки, і, якщо необхідно, дані для ідентифікації користувача;

— Cache (кеш), тобто кожна відповідь повинна бути відміченою, кешується вона чи ні для запобігання повторного використання клієнтом застарілих або некоректних даних, у відповідь на подальші запити;

— Uniform Interface (уніфікований інтерфейс), тобто уніфікований інтерфейс визначає інтерфейс між клієнтською та серверною частинами, а саме спрощує і відділяє архітектуру, що дозволяє розвиватися кожній частині окремо.

— Layered System (система, розділена на шари), тобто в REST допускається розділяти систему на ієрархію шарів, але за умови, що кожен компонент може бачити компоненти тільки безпосередньо наступного шару;

— Code-On-Demand (код на вимогу) — у парадигмі REST дозволяється завантаження і виконання коду або програми на стороні клієнту.

Сервери можуть тимчасово розширювати або персоналізувати функціональні можливості клієнтської частини, передаючи йому логіку, яку він може виконати, але цей пункт не є обов'язковим.

Розділення інтерфейсів означає, що клієнти не зв'язані зі зберіганням даних, які залишаються всередені кожного сервера, а отже мобільність коду клієнтського додатку покращується. Сервери не зв'язані з інтерфейсом користувача або станом, тому сервери можуть бути більш простими та масштабованими. Серверна та клієнтська частини розробляються незалежно і можуть бути легко замінені, якщо інтерфейс залишається незмінним.

Принцип єдиного інтерфейсу об'єднує чотири основні принципи:

— Identification of Resources — в REST ресурсом є все, що може бути поіменовано;

— Manipulation of Resources — представлення в REST використовується для виконання дій над ресурсами;

— Self-Descriptive Messages — запит і відповідь повинні зберігати в собі всю необхідну інформацію для обробки даних;

— Hypermedia as the Engine of Application State — статус ресурса передається через його тіло, заголовки, або як параметри рядка запиту.

Властивість RESTful API при неодноразовому виконанні певного запиту призводити до одного і того ж результату на сервері називається ідемпотентністю. Тобто створення великої кількості ідентичних запитів має такий же ефект, як і один запит. Слід відзначити, що при тому, що ідемпотентні операції створюють однаковий результат на сервері, сама відповідь може відрізнятися, адже наприклад стан ресурсу може змінитися між запитами.

Методи PUT та DELETE за визначенням є ідемпотентними. Не дивлячись на це, запит DELETE, при успішному його виконанні поверне статус 200 (OK) або 204 (No Content), але для подальших запитів буде повертатися 404 (Not Found). Тобто стан на сервері після кожного виклику

DELETE є однаковим, але відповіді відрізняється, оскільки після першого успішного виконання, ресурс вже видалено.

Методи GET, HEAD, OPTIONS визначені як безпечні, оскільки вони призначені лише для отриманні інформації і не повинні змінювати стан серверу. Також вони не повинні мати побічних ефектів, за виключенням нешкідливих, наприклад логування або кешування.

За визначенням, безпечні операції — ідемпотентні, так як призводять до одного і того ж результату на сервері. Вони реалізовані як операції тільки для читання. Однак безпечність не означає, що серверна частина повинен щоразу надсилати одну і ту ж відповідь.

1.4 Огляд технологій для документування програмного коду

1.4.1 Огляд технології Swagger

Технологія Swagger — це набір інструментів з відкритим кодом для написання API на основі REST. Це спрощує процес написання API за допомогою документації коду, вказуючи стандарти та надаючи інструменти, необхідні для написання зручних, безпечних, продуктивних і масштабованих API.

У сучасній сфері програмного забезпечення не існує жодної системи, яка працює в Інтернеті без надання API. Тобто відбувся перехід від монолітних систем до мікросервісів. І весь дизайн мікросервісів покладений на REST API.

У проекті великої інформаційної системи не повинно бути жодних лазівок або збоїв у функціональності API, адже весь проект базується на них.

Серед усіх функціональних можливостей, які доступні в технології Swagger, розглянемо Swagger Editor.

Swagger Editor — це інструмент, який допомагає розробнику перевіряти описані функціональні можливості API в режимі реального часу. Він перевіряє ці можливості на відповідність специфікації OAS Open API і забезпечує миттєвий візуальний зворотний зв’язок.

Інструмент редактора запускається, локально або в Інтернеті та надає миттєвий зворотній зв'язок щодо можливостей API, вказує, чи помилки обробляються належним чином або є якісь проблеми з синтаксисом. [8].

Він також має інтелектуальні функції автозаповнення, які дозволяють писати код швидше. Окрім того, що редактор легкий у налаштуванні, він також дозволяє розробникам створювати заглушки сервера для API для швидшої розробки. Завдяки таким інструментам, як Swagger Editor, розробники мають уявлення в реальному часі про те, як розвивається API. Це також допомагає проаналізувати, як сторонній розробник буде взаємодіяти з API.

Це корисно, при роботі у команді із декількох чоловік, які відповідають за різні компоненти системи. Так, наприклад, розробник, який відповідає за рівень представлення повинен розуміти, які методи описані у API, а також, у якому вигляді потрібно передавати аргументи у метод для коректної роботи. Так, за допомогою використання Swagger Editor, фронт-енд розробник може перейти на сторінку із задокументованим API, де перед ним будуть усі коментарі, які відносяться до методів. Розробник може спробувати виконати метод, для чого є зручне меню, у якому користувач може ввести данні, що є об'єктом. Після виконання буде виведено статус-код, і розробник зможе побачити як веде себе API, при передаванні різної інформації, що допоможе йому описувати свою частину, спираючись на задокументоване API.

1.4.2 Огляд технології Postman

Аналогів у такому вигляді, як Swagger Editor немає. Схожий функціонал має Postman. З його допомогою можна також протестувати як працює API, але якщо порівнювати його безпосередньо із Swagger Editor можна помітити багато недоліків. Наприклад — не зовсім зрозумілий інтерфейс для нового користувача. Окрім цього, для тестування методу, розробник, який займається рівнем представлення вже повинен знати тип цього методу (GET, POST, PUT, DELETE), а також структуру об'єкту, який потрібно передати серверу, щоб

отримати результат. Для цього потрібно буде забрати час у розробника, який займається рівнем доступу до даних. Він повинен буде пояснювати структуру об'єкту, який кожен із методів приймає, а це суттєво вплине на продуктивність обох розробників.

1.5 Огляд технології для вибірки даних GraphQL

GraphQL — це мова запитів для API, а також середовище для виконання запитів із існуючими даними. Вона являє собою повний і зрозумілий опис даних в API, дає клієнтам можливість отримувати саме ті дані, які їм потрібні у конкретний момент, та нічого більше. Це дуже спрощує масштабування API з часом, та представляє потужні інструменти для розробки.

Переваги використання технології GraphQL:

— можливість отримувати лише ту інформацію, яка необхідна у конкретний момент;

— можливість отримувати багато ресурсів в одному запиті;

— можливість описувати різні типи отримуваної інформації;

— можливість використання кешування для покращення швидкодії.

GraphQL був розроблений у великому проекті Facebook, але навіть набагато менші проекти можуть стикатися з обмеженнями традиційних REST API інтерфейсів.

Виникають такі ситуації, коли під час розробки, виникають потреби у зменшенні отримуваних даних від серверної частини. Наприклад, це може стати у нагоді для пришвидшення роботи веб-додатку.

Facebook придумали просте рішення: замість того щоб розробник описував багато endpoint, які повертають різну інформацію для різних потреб, можна описати всього один «розумний» endpoint, який за допомогою складних запитів буде повертати дані саме такої форми, яка потрібна клієнту.

Користуватись моделлю REST — це наче перевантажуватись зайвою роботою. Наприклад для отримання трьох різних послуг, необхідно зробити три різних запити (рисунок 1.2).

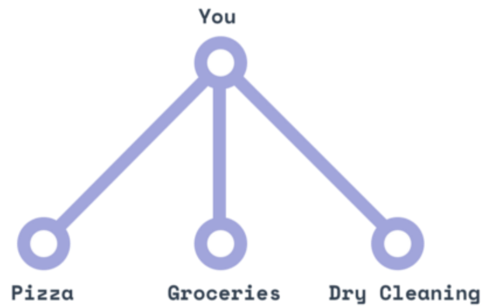


Рисунок 1.2 — Приклад використання REST

В свою чергу GraphQL схожий на особистого помічника, якому можна делегувати обов’язки, пояснивши деяку технічну інформацію, а потім просто робити запит на те, що потрібно. (рисунок 1.3) [9].

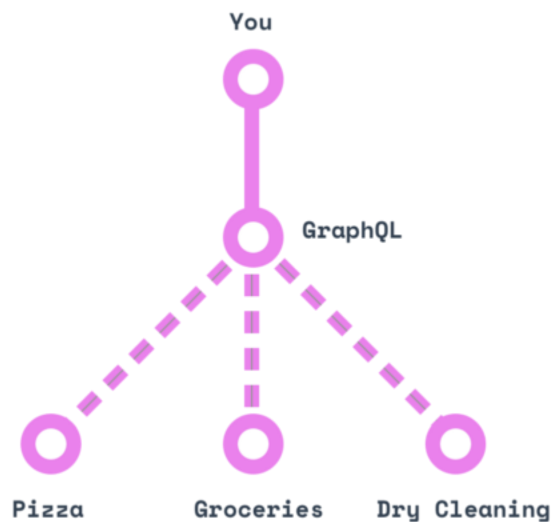


Рисунок 1.3 — Приклад використання GraphQL

Тому GraphQl доцільно використовувати у проекту як технологію вибірки інформації із бази даних.

1.6 Обгрунтування вибору хмарного сховища

При проектуванні додатку з’явилась проблема із збереженням фотографій, які будуть відображатись у новинах, а також фотографій викладачів. Так як актуальний сервер дає певні обмеження на збереження

зображень прямо у файловій системі серверу, а збереження фотографій безпосередньо всередині бази даних вплине на швидкодію, та значно збільшить об’єм бази даних, необхідно шукати альтернативні вирішення проблеми.

Однією з таких альтернатив є використання хмарного сховища Google Cloud Storage. У ньому можна зберігати великі об’єми інформації та швидко до них доступатись, а також його використання полегшить роботу розробникам, які відповідають за рівень представлення.

Google Cloud Storage — це хмарна технологія розроблена компанією Google. Вона дозволяє платити лише за той об’єм даних, який використовується. Тобто не потрібно одразу оплачувати сховище на певний об’єм. По мірі збільшення інформації, яку зберігає сховище воно буде автоматично розширятись. Проаналізувавши об’єм інформації, яку необхідно буде зберігати у сховищі, можна зробити висновок що це буде самим оптимальним варіантом, який поєднає у собі швидкодію, зручність користування та низьку ціну.

За допомогою спеціального SDK, у коді сервера описано логіку для збереження файлів, отриманих від користувачів, безпосередньо у сховище. В свою чергу, у базу даних буде зберігатись лише посилання на файл у хмарі, яке необхідне на рівні представлення, для відображення зображення. Також у базі даних зберігається ім’я файлу, для можливості його видалення безпосередньо через код серверу.

1.6 Вибір системи управління базами даних

Вибір бази даних грає важливу роль при створенні будь-якого веб-додатку, так як навіть самі маленькі інформаційні системи мають потребу у збереженні інформації на сервері. Тому важливо обрати оптимальну систему управління базами даних (СУБД).

Найпопулярнішими з існуючих СУБД є:

— MySQL;

— MS SQL Server;

— PostgreSQL;

— Oracle.

MySQL та MS SQL Server схожі між собою, але кожна з них має свої переваги та недоліки. Вони обидві виконують однакову функцію, хоча мають відмінності у використанні. Обидві СУБД базуються на SQL (Structured Query Language) [10].

Основні відмінності між MySQL та MS SQL Server:

— середовище розробки — SQL сервер краще працює із платформою .NET, в той час, як MySQL може бути використаний з майже будь-якою іншою мовою програмування, при цьому найчастіше його використовують при інтеграції з PHP;

— синтаксис;

— SQL Server більше ніж просто СУБД — Microsoft прив’язали до нього багато інструментів, які допомагають у розробці.

Проаналізувавши основні відмінності між MySQL та MS SQL Server, можна зробити висновок, що для розробки інформаційної системи, з використанням ASP.NET Web-API краще використовувати саме MS SQL Server, адже ця СУБД краще інтегрується із мовою програмування C#, та надає більше інструментів для роботи із даними.

Для проведення будь-яких базових маніпуляцій з даними у інформаційній системі використовується серверна частина. Важливо, щоб ці маніпуляції відбувались швидко, а розробникам було зручно працювати із кодом. Для цього було проведено аналіз технологій, які потрібні для роботи із базою даних, документування коду, вибірки інформації та збереження файлів.

Окрім цього, дані із серверної частини повинні швидко досягати рівня представлення, щоб відображатись користувачу. Тому було прийнято рішення використовувати GraphQL у поєднанні з REST API, для того щоб поєднати переваги кожного із цих підходів, за рахунок чого можна пришвидшити процес розробки та оптимізувати серверну частину.

Про розробці серверної частини інформаційної системи підрозділу навчального закладу будуть використовуватися такі технології:

— Entity Framework Core, для комунікації із базою даних із середовища .NET;

— GraphQL, для вибірки даних, які потрібні у конкретний момент часу;

— Swagger, для документації коду;

— Google Cloud, для збереження файлів у хмарному сховищі та зменшення навантаження на сервер.

2 РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ ІНФОРМАЦІЙНОЇ СИСТЕМИ СТРУКТУРНОГО ПІДРОЗДІЛУ НАВЧАЛЬНОГО ЗАКЛАДУ

2.1 Аналіз вимог до клієнт-серверної інформаційної системи та розробка мапи веб-додатку

Проаналізовано інформацію, яку буде містити розроблювана інформаційна система, що дозволило сформувати вимоги до веб-додатку. Так, було вирішено, що інформаційна система повинна містити інформацію про:

— історію кафедри;

— новини кафедри;

— викладачів, їхні досягнення, біографію, нагороди та дисципліни, які вони викладають;

— наукову роботу, яка проводиться на кафедрі;

— методичну роботу, яка проводиться на кафедру;

— навчальна інформація, для студентів та абітурієнтів, що буде містити в собі інформацію про освітні програми, дисципліни, які викладаються на кафедрі.

Головна сторінка, розроблюваного веб-додатку відкривається через посилання на додаток. Вона зацікавить користувача, та буде містити посилання на інші сторінки інформаційної системи.

Основні вимоги, що висуваються до контенту:

— динамічні сторінки (новини, викладачі, наукова та методична робота, тощо), маніпуляції над якими відбуваються через серверну частину, повинні заповнюватись через адміністративну панель, для чого розробник адміністративної панелі використовує описане в серверній частині API;

— статичні сторінки (історія кафедри, головна сторінка) заповнюються розробником рівня представлення і не мають зв’язку із серверною частиною.

Проаналізувавши вимоги, було складено мапу веб-додатку (рисунок 2.1), в якій представлено основні сторінки, що будуть відображатись у веб-клієнті розроблюваної клієнт-серверної інформаційної системи.

Деякі з них є статичними і просто представляють загальну інформацію. Таку інформацію не потрібно зберігати у базі даних, адже вона не буде змінюватись із часом, тому вона описується тільки на рівні представлення.

Інша інформація, яка представлена на динамічних сторінках, буде змінюватись адміністратором ресурсу, тому потрібно надати інструментарій у вигляді API для зручної маніпуляції над даними.

Рисунок 2.1 — Мапа веб-додатку інформаційної системи

2.2 Проектування інформаційної системи з використанням технології Web-API

Для розробки серверної частини було прийнято рішення використовувати технологію Web-API, яка працює за принципом веб-служби, яка розгортається на сервері, та відправляє відповіді на запити із клієнтською частини [11]. Її відмінність від підходу MVC, в тому, що метод буде повертати статус код замість представлення. У нашому випадку це робить розробку більш гнучкою, так як веб-розробник може використовувати будь-які

технології на рівні представлення, і лише викликати API у потрібні моменти. В свою чергу API буде повертати статус код, який визначає успішність виконання запиту, який надійшов до серверу (рисунок 2.2).

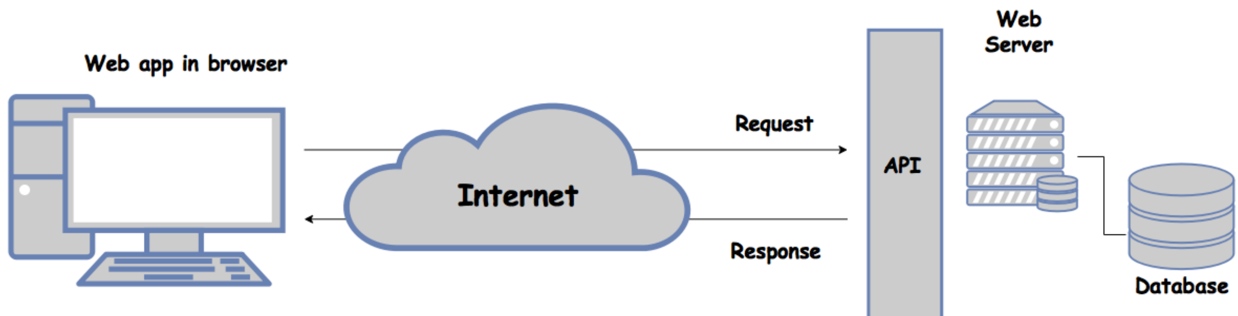


Рисунок 2.2 — Приклад роботи з WEB-API

Комунікація між рівнем представлення та API відбувається за допомогою класів, які називають контролери. Контролери проводять маніпуляції над даними, які зберігаються в моделях та повертають оброблені дані на рівень представлення, разом із статус кодом.

2.2.1 HTTP коди статусів при проектуванні серверної частини

Код статусу HTTP — це відповідь сервера на запит браузера. Коли користувач потрапляє на веб-сайт, браузер надсилає запит на серверну частину веб-додатку, після чого отримує відповідь на свій запит у вигляді тризначного кодом: код статусу HTTP [12].

Ці коди стану є Інтернет-еквівалентом розмови між вашим браузером і сервером. Вони повідомляють, чи все в порядку між ними, чи виникла помилка. Розуміння кодів стану та способів їх використання допоможе швидко діагностувати помилки сайту, щоб мінімізувати час простою на сайті. Можна навіть використовувати деякі з цих кодів статусу, щоб допомогти пошуковим системам і людям отримати доступ до веб-додатку; наприклад, переспрямування 301 повідомить ботам і людям, що сторінка назавжди перемістилася в інше місце.

Коди статусу поділяються на п’ять категорій:

— 100-199 — інформаційні;

— 200-299 — успішні;

— 300-399 — перенаправлення;

— 400-499 — клієнтські помилки;

— 500-599 — серверні помилки.

API, створене для інформаційної системи, повертає різні коди статусів залежно від конкретної ситуації. Для прикладу розберем метод Put, який відповідає за оновлення інформації у базі даних (лістинг 2.1).

Лістинг 2.1 — Метод Put

```
[HttpPut("EditedNews")]
        public ActionResult Put(NewsRequest newsModel){
            if (newsModel == null)
                return BadRequest();
            if (!_newsRepository.GetList().Any(n => n.Id ==
newsModel.Id))
                return NotFound();
            var news = _newsService.CreateNews(newsModel);
            _newsRepository.Update(news);
            return Ok(news);}
```

У прикладі, що описаний в лістингу 2.1, за різних умов рівень представлення отримає різні коди статусів. Так, наприклад, якщо модель, яку потрібно оновити, прийшла пустою, серверна частина поверне код 400 — Bad Request (Поганий запит). Тому що у тілі запиту прийшло не валідне значення.

Якщо прийшла модель, з неіснуючим id, серверна частина поверне код 404 — Not Found (Не знайдено), що означає, що новини з таким ідентифікаційним номером не має у базі даних, тому її не можливо оновити. Статус 404 не вказує, чи ресурс відсутній назавжди чи лише тимчасово.

Якщо ж проблем під час виконання не виникало, серверна частина поверне статус 200 — OK, що означає що данні успішно оновились. Це код статусу для нормальної, повсякденної, правильно функціонуючої сторінки. Відвідувачі та боти проходять через посилання успішно.

Окрім цього, при створенні рекомендовано використовувати код статусу 201 — Created (Створений), а при виникненні помилок під час виконання програмного коду повертати 500 — Internal Server Error (Внутрішня помилка сервера).

2.2.2 Моделі даних у ASP.NET Web API

Для кожної сутності, яка використовується при розробці серверної частини, було створено по дві моделі, а саме Request та Response (запит та відповідь). Перш за все, це використовується через те, що запити, які приходять від рівня представлення, та відповіді, які йому відправляються, мають трохи змінену структуру, ніж у самої сутності. Окрім того, використання безпосередньо об’єкту сутності для роботи із рівнем представлення вважається поганою практикою.

Порівняємо сутність News, (рисунок 2.3 а), із моделлю NewsRequest, яка приходить з рівня представлення (рисунок 2.3 б).

```
public class News
{
    Ссылок: 4
    public int Id { get; set; }
    Ссылок: 4
    public string Title { get; set; }
    Ссылок: 4
    public string Description { get; set; }
    Ссылок: 4
    public string Content { get; set; }
    Ссылок: 2
    public DateTime DateTime { get; private set; }
    Ссылок: 4
    public string ImageStorageUrl { get; set; }
    Ссылок: 2
    public string ImageName { get; set; }
```

а)

```
ublic class NewsRequest
{
    ссылка: 1
    public int Id { get; set; }
    ссылка: 1
    public string Title { get; set; }
    ссылка: 1
    public string Description { get; set; }
    ссылка: 1
    public string Content { get; set; }
    Ссылок: 2
    public IFormFile ImageFile { get; set; }
}
```

б)

Рисунок 2.3 — Опис моделі сутності (а) та моделі запиту (б)

Запит не містить в собі поле для дати. Це обумовлено тим, що дату не потрібно присвоювати вручну, а вона присвоюється автоматично після створення об’єкту новини (лістинг 2.2).

Також, замість полів, які зберігають посилання на зображення та його ім’я, які описані у сутності, в моделі описано поле, яке зберігає в собі файл. Це

обумовлено тим, що у запиті приходить файл, який зберагіється у сховищі Google Cloud, а тому у базу даних буде записано лише посилання на це зображення, щоб не засмічувати її зайвою інформацією.

Лістинг 2.2 — Присвоєння даті значення, при створенні новини

```
public News()
        {
            DateTime = DateTime.Now;
        }
```

При порівнянні класу сутності News (рисунок 2.4 а) із моделлю NewsResponse (рисунок 2.4 б), багатьох змін не видно.

Єдина відмінність — відсутність імені зображення, у моделі відповіді. Це зроблено лише тому що, вона не потрібна на рівні представлення. Вона зберігається у базі даних лише для того, щоб можна було видалити зображення із хмари при видаленні відповідної новини, де використовується це зображення.

```
public class News
{
    Ссылок: 4
    public int Id { get; set; }
    Ссылок: 4
    public string Title { get; set; }
    Ссылок: 4
    public string Description { get; set; }
    Ссылок: 4
    public string Content { get; set; }
    Ссылок: 2
    public DateTime DateTime { get; private set; }
    Ссылок: 4
    public string ImageStorageUrl { get; set; }
    Ссылок: 2
    public string ImageName { get; set; }
```

а)

```
public class NewsResponse
{
    ссылка: 1
    public int Id { get; set; }
    ссылка: 1
    public string Title { get; set; }
    ссылка: 1
    public string Description { get; set; }
    ссылка: 1
    public string Content { get; set; }
    ссылка: 1
    public DateTime DateTime { get; set; }
    ссылка: 1
    public string ImageStorageUrl { get; set; }
```

б)

Рисунок 2.4 — Опис моделі сутності (а) та моделі відповіді (б)

Модель відповіді містить в собі посилання на зображення, для того, щоб на рівні представлення можна було відобразити зображення за посиланням.

Код, який відповідає за приведення об’єкту моделі до об’єкту сутності приведено в лістингу 2.3

Лістинг 2.3 — Приведення об’єкту моделі до об’єкту сутності

```
public News CreateNews(NewsRequest newsEditingModel)
        {
            var news = new News();
            string fileName;

            news.Title = newsEditingModel.Title;
            news.Description = newsEditingModel.Description;
            news.Content = newsEditingModel.Content;

            if (newsEditingModel.ImageFile != null)
            {
                news.ImageStorageUrl = 
_imagesService.SaveImages(newsEditingModel.ImageFile, out 
fileName);
                news.ImageName = fileName;
            }
            return news;
        }
```

В лістингу 2.3 усі поля співставляються, а зображення зберігається у хмарі, після чого в об’єкті сутності зберігається посилання на зображення у хмарному сховищі.

2.3 Проектування бази даних у інформаційній системі

У будь-якій великій інформаційній системі передбачено використання бази даних, для зберігання певної інформації. При розробці інформаційної системи було вирішено використовувати реляційну базу даних MS SQL Server, так як вона має такі основні переваги:

— легка у використанні;
— має можливості кодування та компресії інформації;
— має хорошу захищеність бази даних.

У нашому випадку для бази даних описано дві таблиці, які не пов’язані між собою. Для майбутнього відображення на сайті було описано таблиці для зберігання інформації про новини та викладачів.

Після аналізу вимог, для збереження новин було обрано опис таких основних полів:

— Id, для ідентифікації кожної новини унікальним числовим значенням;

— Title, для збереження заголовку новини;

— Description, для збереження стислого опису статті, який може зацікавити користувача;

— Content, для збереження основного контенту новини;

— DateTime, для збереження дати створення новини;

— ImageStorageUrl, для збереження посилання на зображення новини, яка зберігається у хмарному сховищі Google Cloud;

— ImageName, для збереження назви зображення.

Ці поля містять усю необхідну інформацію, для коректного відображення новини на рівні представлення. Окрім цього, за рахунок того, що було використано хмарне сховище, це дозволить не засмічувати базу даних зайвою інформацією, такою як масив байтів, що являє собою зображення. Так як зображення зберігається у хмарі, достатньо зберігати посилання на це зображення, використовуючи яке на рівні представлення можна без проблем отримати доступ до рисунку і показати його на сторінці із новиною.

Поле DateTime ініціалізується автоматично. Одразу після створення новини, система фіксує поточний час і присвоює його як значення цього поля.

Також була описана таблиця для зберігання інформації про викладачів, яка містить такі поля:

— Id, для ідентифікації кожного запису про викладача унікальним числовим значенням;

— FirstName, для зберігання імені викладача;

— SecondName, для зберігання прізвища викладача;

— Position, для зберігання посади, яку займає викладач;

— Merits, для зберігання досягнень викладача;

— Biography, для зберігання стислої інформації про викладача;

— Awards, для зберігання нагород, які має викладач;

— ImageStorageUrl, для збереження посилання на фотографію викладача, яка зберігається у хмарному сховищі Google Cloud;

— ImageName, для збереження назви зображення.

Виходячи з описаних вимог, таблиця із такими полями буде містити вичерпну інформацію про кожного викладача, яка буде корисна для користувачів інформаційного ресурсу.

Для зберігання фотографій викладачів, також буде використовуватись хмарне сховище Google Cloud, що дозволить повністю розгрузити базу даних від зайвої інформації, та підвищити її швидкодію.

Але створювати базу даних вручну немає сенсу, так як є потреба в тому щоб підтримувати її програмно. Для цього у проект була підключена технологія Entity Framework

2.3.1 Опис моделі даних Entity Framework

Перейдемо до створення моделі, яка буде описувати сутність. Наприклад, опишем модель новини, яка буде містити ідентифікатор, текстові поле для назви, опису, контенту, посилання на картинку у Google Cloud, ім’я картинки та поле дати для збереження дати створення новини (лістинг 2.4).

На вигляд це звичайний клас, який містить у собі декілька автоматичних властивостей та конструктор. Але кожна така властивість буде співставлятися з окремим стовпчиком у таблиці з бази даних.

Слід відзначити, що Entity Framework потребує визначення ключа елемента, для створення первинного ключа в таблиці в БД. За замовчуванням, при генерації БД, технологія буде розглядати властивості з іменами Id або [ім’я_класу]Id (в цьому випадку було б NewsId). Якщо ж потрібно створити складений ключ, або просто призначити ключем поле із іншою назвою, потрібно буде використовувати конфігурацію.

Лістинг 2.4 — Модель News

```
public class News
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public string Content { get; set; }
        public DateTime DateTime { get; private set; }
        public string ImageStorageUrl { get; set; }
        public string ImageName { get; set; }

        public News()
        {
            DateTime = DateTime.Now;
        }
    }
```

Після створення бази даних, таблиця News буде створена та описана, та буде мати вигляд, який зображено в лістингу 2.5

Лістинг 2.5 — Створення таблиці News у базі даних

```
CREATE TABLE [dbo].[News](
     [Id] [int] IDENTITY(1,1) NOT NULL,
     [Title] [nvarchar](max) NULL,
     [Description] [nvarchar](max) NULL,
     [Content] [nvarchar](max) NULL,
     [DateTime] [datetime] NOT NULL,
     [ImageStorageUrl] [nvarchar](max) NULL,
     [ImageName] [nvarchar](max) NULL,
```

Технологія Entity Framework здатна автоматично створити таблицю, поля у якій будуть співставленні властивостям, описаним у класі. Це дає розробнику можливість проводити маніпуляції із об'єктом, використовуючи платформу .NET, та за допомогою спеціальних функцій, застосувати зміни до відповідного запису у таблиці бази даних.

Аналогічним образом було створено таблицю, для зберігання інформації про викладачів.

2.3.2 Опис контексту даних Entity Framework для зв’язку із базою даних.

Взаємодія з базою даних у Entity Framework Core відбувається за допомогою спеціального класу — контексту даних. Основну функціональність Entity Framework Core для роботи з базами даних у MS SQL Server складають класи, які розміщуються у просторі імен Microsoft.EntityFrameworkCore.

Серед усього набору класів цього простору імен потрібно виділити:

— DbContext, який визначає контекст даних, що використовується для взаємодії з базою даних;

— DbSet, який являє собою набір об’єктів, що зберігаються у базі даних — представляють собою таблиці на стороні БД.

В будь-якому додатку, що працює із базою даних через Entity Framework, потрібен контекст — клас, який буде унаслідуватись від DbContext, наприклад Context (лістинг 2.6).

Цей клас унаслідується від базового класу DbContext, який описаний у просторі імен System.Data.Entity.

У цьому класі також визначені автоматичні властивості News та Teachers, які буде зберігати набір даних типу News та Teacher. Через ці колекції буде відбуватись комунікація із таблицями, які створені у базі даних за допомогою технології.

Лістинг 2.6 — Клас Context

```
public class Context: DbContext{
        private const string DATABASE_NAME =
"VNTU_ComputingTechnologiesDepartmeent";
        public Context(): base(DATABASE_NAME)
        {
            Database.CreateIfNotExists();
        }

        public DbSet<News> News { get; set; }
        public DbSet<Teacher> Teachers { get; set; }}
```

Слід зазначити, що для зв’язку із базою даних необхідно передати стрічку підключення.

У лістингу 2.6, видно конструктор класу DbContext передається назва поля, у конфігураційному файлі, де зберігається стрічка підключення до бази даних (лістинг 2.7). Базовий клас автоматично вичитає інформацію, про стрічку підключення з конфігураційного файлу, і буде використовувати її для зв’язку із базою даних.

Лістинг 2.7 — Стрічка підключення, що зберігається у конфігураційному файлі

```
"ConnectionStrings": {
    "VNTU_ComputingTechnologiesDepartmeent": "Data
Source=(localdb)\\MSSQLLocalDB;Initial
Catalog=VNTU_ComputingTechnologiesDepartmeent;Integrated
Security=True;Connect
Timeout=30;Encrypt=False;TrustServerCertificate=False;Applic
ationIntent=ReadWrite;MultiSubnetFailover=False"
       },
```

Також варто відмітити, що за замовчуванням бази даних не існує. Тому у конструкторі контексту викликається метод Database.CreateIfNotExist(), який при створенні контексту автоматично перевірить наявність бази даних і, якщо вона відсутня, створить її.

Тепер, коли логіка для створення таблиць у базі даних описана, потрібно описати логіку основних операцій над даними. Для того, щоб залишати код «чистим», та гнучким можна використати шаблон проектування «Repository».

2.3.3 Основні операції з даними у Entity Framework

Для додавання об’єкта у базу даних використовується метод Add, визначений у класі DbSet, в який передається об’єкт що додається (лістинг 2.8). Цей метод встановлює статус Added як стан нового об’єкту. Тому метод _context.SaveChanges() генерує вираз INSERT для вставки моделі у таблицю.

Лістинг 2.8 — Додання нового об’єкту у базу даних

```
_context.News.Add(item);
context.SaveChanges()
```

Видалення відбувається за допомогою методу Remove, який приймає об’єкт, що повинен видалитись із бази даних (лістинг 2.9). Даний метод встановлює статус об’єкту в Deleted, завдяки чому Entity Framework при виконанні методу _context.SaveChanges() згенерується SQL-вираз DELETE.

Лістинг 2.9 — Видалення об’єкту з бази даних

```
_context.News.Remove(item);
context.SaveChanges();
```

При зміненні об’єкту EF (лістинг 2.10) сам відслідковує всі зміни і коли викликається метод SaveChanges(), буде сформовано SQL-вираз UPDATE для обраного об’єкту, який обновить об’єкт у базі даних.

Лістинг 2.10 — Редагування об’єкту у базі даних

```
if (news != null)
    {
          news.Title = item.Title;
           news.Description = item.Description;
           news.Content = item.Content;
           news.ImageStorageUrl = item.ImageStorageUrl;
           _context.SaveChanges();
}
```

Аналогічним чином була описана логіка для обробки інформації про викладачів. Уся ця логіка була описана всередині класів «репозиторіїв».

2.3.4 Використання шаблону «Repository» для маніпуляцій над даними

Одним із найбільш часто використовуваних патернів при роботі з даними є патерн «Repository». Репозиторій дозволить абстрагуватись від конкретних підключень до джерел даних з якими працює програма і являється поміжною ланкою між класами, що призначені для роботи з даними і з рештою програми.

Для того, щоб не засмічувати безпосередньо контролери логікою для маніпуляцій з рівнем бази даних, уся ця логіка була винесена у клас «репозиторій», у якому викликаються потрібні методи за потреби.

Так, наприклад, у класі контролеру можна залишити лише логіку, для повернення статус кодів, а всю логіку для додавання інформації у базу даних винести в репозиторій (лістинг 2.11).

Лістинг 2.11 — Приклад методу контролеру

```
[HttpGet("NewsId")]
        public ActionResult Get(int id)
        {
            var news = _newsRepository.GetById(id);
            if (news == null)
                return NotFound();
            var newsViewModel =
_newsService.CreateNewsViewModel(news);
            return Ok(newsViewModel);}
```

У лістинг 2.11, видно, що об’єкт типу News отримується з бази даних безпосередньо у репозиторії (лістинг 2.12) та передається у контролер, який в свою чергу може передавати цю інформацію на рівень представлення.

Лістинг 2.12 — Метод «GetById», описаний, у репозиторії

```
public News GetById(int id) => _context.News.Find(id);
```

У репозиторії робота відбувається безпосередньо з контекстом даних, який містить в собі інформацію, що зберігається в базі даних. За допомогою

методу Find, можна знайти об’єкт у базі даних, використовуючи ключове значення. У даному випадку, поле id.

2.4 Використання технології Swagger для опису API у веб-додатку

Специфікація OpenAPI 3.0 визначає стандартизований, незалежний від мови інтерфейс до RESTful API, що дозволяє розуміти можливості сервісу без доступу до вихідного коду, документації чи аналізу трафіку. Визначення OpenAPI можуть використовуватися засобами генерації документації.

В нашому випадку для автоматичної генерації документації на основі XML коментарів до коду та атрибутів був використаний пакет Swashbuckle. Згенерована документація є інтерактивною, тобто виконувати запити до API можна через сторінку з документацією, що є зручним для розробників клієнтських додатків.

Налаштування виконується в проекті ASP.NET WebAPI, в класі Startup. В методі ConfigureServices відбувається реєстрація Swagger генератору, при цьому вказавши конфігурацію. В даному випадку вказується версію API, додається його назву та опис. Також включається використання XML коментарів для генерації документу OpenAPI, вказується шлях до XML файла (лістинг 2.13). Також в методі Configure, якщо середовище налаштоване для розробки то Swagger включається в конвеєр обробки запиту та налаштовується відображення SwaggerUI (лістинг 2.14).

Після вдалого налаштування, перевірити працездатність можна запустивши проект, та додавши до посилання «/swagger». Приклад відображення користувацького інтерфейсу зображено на рисунку 2.5

Сторінка з інтерактивною документацією містить список вибору версії та короткі дані про нього, також відображено кінцеві точки API (endpoints), згруповані за тегами, в якості яких за замовчуванням виступають назви контролерів.

Лістинг 2.13 — Налаштування Swagger в методі ConfigureServices

```
services.AddSwaggerGen(c =>
      {
 c.SwaggerDoc("v1", new OpenApiInfo {
              Title = "ot_api",
              Version = "v1",
              Description = "Документація для інформаційної системи кафедри обчислювальної техніки ВНТУ" });
var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
c.IncludeXmlComments(xmlPath);
            });
```

Лістинг 2.14 — Додання middleware та налаштування SwaggerUI

```
if (env.IsDevelopment()){
         app.UseDeveloperExceptionPage();
         app.UseSwagger();
         app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "ot_api v1"));
}
```

Також можна додати документацію до метода, для отримання інформації через користувацький інтерфейс Swagger (лістинг 2.15).

Лістинг 2.15 — Приклад документування метода

```
        /// <summary>
        /// Get all list of news from db
        /// </summary>
        /// <returns>List of news</returns>
        [HttpGet("News")]
        public ActionResult Get()
```

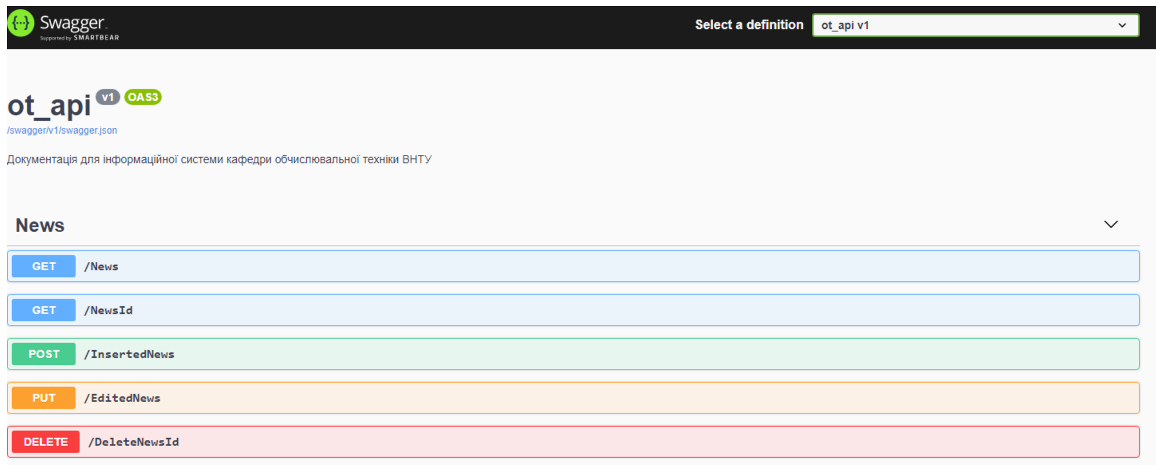


Рисунок 2.5 — Згенерована сторінка з описом API за допомогою Swagger

Різні HTTP методи при цьому виділено окремими кольорами. Після натискання на кінцеву точку відображаються необхідні параметри, або приклад даних, що відправляються в тілі запиту (з можливістю вибрати формат відображення), а також статус-коди, що можуть бути отримані у відповідь. Доступна кнопка «спробувати», яка дозволяє виконати запит (рисунок 2.6). У результаті виконання запиту, можна побачити статус-код, дані відправлені у відповідь, заголовки запиту, а також URL запиту та код, за допомогою якого можна виконати запит за допомогою утиліти curl (рисунок 2.7).

Parameters
Try it out
Name
Description
Title
string
(query)
Title
Description
string
(query)
Description
Content
string
(query)
Content

Рисунок 2.6 — Приклад згенерованого методу у Swagger

Curl

```
curl -X POST "https://localhost:44331/InsertedNews?Title=Title&Description=Description&Content=Content" -H "accept: */*" -H "Content-Type: multipart/form-data" -d ""
```

Request URL

```
https://localhost:44331/InsertedNews?Title=Title&Description=Description&Content=Content
```

Server response

Code	Details
200	Response body


```
{
  "id": 14,
  "title": "Title",
  "description": "Description",
  "content": "Content",
  "dateTime": "2022-06-05T01:19:15.1502674+03:00",
  "imageStorageUrl": null,
  "imageName": null
}
```


Response headers

```
access-control-allow-origin: *
content-type: application/json; charset=utf-8
date: Sat04 Jun 2022 22:19:15 GMT
server: Microsoft-IIS/10.0
x-powered-by: ASP.NET
```

Рисунок 2.7 — Відповідь серверної частини на запит

Після успішно налаштованого Swagger Editor, та документації коду, згенеровану сторінку може використовувати розробник, який працює над рівнем представлення, для того щоб розуміти, які методи описані в API, та об’єкти якої структури вони приймають.

2.5 Реалізація роботи з Google Cloud через платформу .Net

Оскільки як сховище для збереження зображень, які завантажуються у інформаційній системі, було обрано хмарне сховище Google Cloud, необхідно додати реалізацію вивантаження файлів прямо через середовище розробки .Net. Але в першу чергу, для використання Google Cloud, необхідно створити там сховище з іменем «Bucket».

Для цього потрібно мати акаунт Google, який прив’язаний до Google Console, яка дає можливість використовувати усі технології Google. Після створення акаунту, необхідно налаштувати доступність до сховища, щоб не було проблем із тим, щоб отримувати зображення на рівні представлення. Тому, через спеціальний пункт «APIs & Services» ми виставляєм можливість переглядати файли усім [13].

Окрім цього, для коректної роботи програми безпосередньо із сховищем необхідно створити файл із обліковими даними у форматі json, щоб програма

могла вносити зміни у сховище від імені користувача. Після генерації такого файлу засобами Google Console, його потрібно скопіювати у папку із програмним рішенням, для того, щоб інформацію з нього можна було вичитати та зайти під користувацьким обліковим засобом.

Тепер, коли сховище створено, а файл знаходиться за правильним шляхом, потрібно модифікувати файл конфігурації, в який потрібно включити шлях до файлу із обліковими даними, та назву самого bucket, у який ми будем додавати файли. Стрічки, що було додано до конфігураційного файлу приведені в лістингу 2.16

Лістинг 2.16 — Стрічки для роботи із Google Cloud у конфігураційному файлі

```
  "GoogleCredentialFile": "JsonServiceAccount\\test-vntu-
cloud-6e66512bfe07.json",
  "GoogleCloudStorageBucket": "vntu-test-images",
```

Bucket має назву «vntu-test-images», а файл із обліковими даними зберігається у папці JsonServiceAccount.

Тепер потрібно описати код, який буде працювати із Google Cloud Storage. Для цього був описаний інтерфейс, у якому описані основні методи, які повинен реалізовувати клас, що працює із хмарним сховищем (лістинг 2.17).

У інтерфейсі описані два основних методи, які потрібні для роботи із сховищем: завантаження файлу та видалення файлу зі сховища.

Лістинг 2.17 — Інтерфейс ICloudStorage

```
public interface ICloudStorage
    {
        string UploadFile(IFormFile imageFile, string
fileNameForStorage);
        void DeleteFile(string fileNameForStorage);}
```

Також потрібно описати клас, що реалізує цей інтерфейс і визначить конкретну логіку для додавання файлів та їх видалення. Конструктор цього класу приймає конфігурацію, щоб із неї можна було отримати інформацію про облікові дані та назву bucket. Окрім цього, в конструкторі описана логіка для створення клієнту, за допомогою якого можна проводити маніпуляції з сховищем. Опис конструктора подано в лістингу 2.18

Після ініціалізації клієнта для роботи зі сховищем, потрібно описати методи для вивантаження файлів та їх видалення.

Щоб завантажити файл, потрібно відкрити потік, у який потрібно завантажити файл, який приходить із рівня представлення. Після чого викликати метод UploadObject, який завантажує файл у конкретний bucket.

Лістинг 2.18 — Реалізація конструктора GoogleCloudStorage

```
public GoogleCloudStorage(IConfiguration configuration){
            googleCredential =
GoogleCredential.FromFile(configuration.GetValue<string>("GoogleCredentialFile"));
            storageClient =
StorageClient.Create(googleCredential);
            bucketName =
configuration.GetValue<string>("GoogleCloudStorageBucket");
        }
```

Цей метод приймає як параметр назву bucket, у який файл буде завантажений, ім’я файлу, та потік, у якому зараз знаходиться зображення. Також цей метод повертає об’єкт, який був вивантажений у хмару, у якого є поле MediaLink, яке буде зберігатись у базі даних, для надання його у рівень представлення. Приклад реалізації методу вивантаження файлу приведено в лістингу 2.19

Реалізуємо метод для видалення зображення із сховища. Потрібно просто викликати метод DeleteObject у змінної клієнту сховища.

Лістинг 2.19 — Метод UploadFile

```
public string UploadFile(IFormFile imageFile, string fileNameForStorage)
        {
            using (var memoryStream = new MemoryStream())
            {
                imageFile.CopyToAsync(memoryStream);
                var dataObject = storageClient.UploadObject(bucketName, fileNameForStorage, null, memoryStream);
                return dataObject.MediaLink;
            }
        }
```

Цей метод приймає ім'я bucket, із якого буде видалено файл, та ім'я самого файлу, який потрібно видалити. Приклад реалізації методу видалення файлу приведено в лістингу 2.20

Лістинг 2.20 — Метод DeleteFile

```
public void DeleteFile(string fileNameForStorage)
        {
            storageClient.DeleteObject(bucketName, fileNameForStorage);
        }
```

Для роботи із сховищем було описано ще один клас, який працює як посередник між контролером, та безпосередньо класом для роботи із хмарним сховищем. Окрім цього у цьому класі описана додаткова логіка для зміни назви перед збереженням (лістинг 2.21).

В свою чергу метод DeleteImage, який приймає imageName як параметр, просто викликає функцію для видалення файлу із хмарного середовища.

У лістингу 2.21 видно, що змінюється назва файлу. Для неї генерується нове унікальне значення типу GUID, для унікальної назви кожного зображення. Це зроблено для того, щоб уникнути видалення файлів з однаковими іменами, які будуть завантажуватись у хмарне середовище. Цей метод повертає посилання на зображення у хмарі, а також ім'я файлу, для того, щоб записати цю інформацію у базу даних.

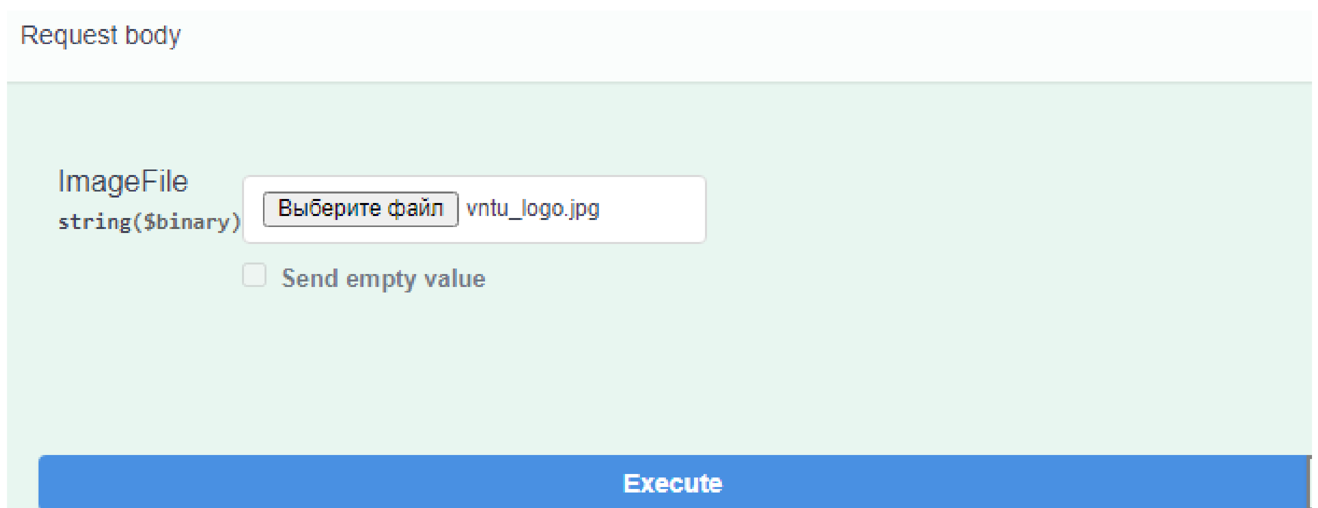
Лістинг 2.21 — Реалізація методу SaveImages

```
public string SaveImages(IFormFile imageFile, out string
fileName)
        {
            string filePath = null;
            string cloudLink = null;
            if (imageFile != null)
            {
                Guid id = Guid.NewGuid();
                string ext =
Path.GetExtension(imageFile.FileName);
                filePath = id + ext;

                cloudLink =
cloudStorage.UploadFile(imageFile, filePath);
            }
            fileName = filePath;
            return cloudLink;
        }
```

Для перевірки працездатності було створено новий об’єкт типу News, у тіло запиту якого було додано зображення (рисунок 2.8 а).

У відповіді з серверної частини приходить посилання на зображення та його назва (рисунок 2.8 б). Ця інформація потрапляє у базу даних.



а)

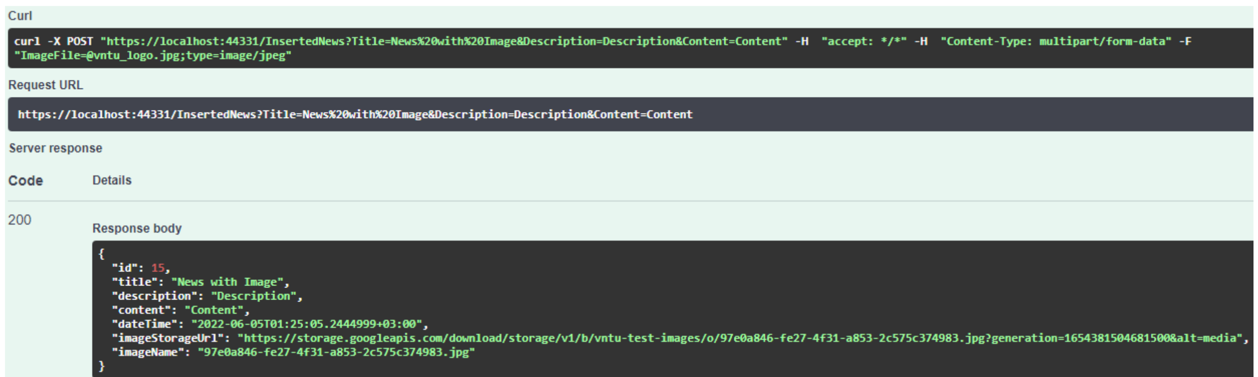


Рисунок 2.7 — Завантажене зображення у тілі запиту (а) та відповідь серверної частини (б)

Після завантаження нового об’єкту із зображенням потрібно переглянути базу даних, для того щоб перевірити чи об’єкт додався коректно (рисунок 2.9). З рисунку 2.9 видно, що усі поля заповнились саме тою інформацією, яка була отримана від клієнтської частини, під час створення об’єкту. Окрім того створилось посилання на файл у Google Cloud, а також його ім’я.

Id	Title	Description	Content	DateTime	ImageStorageUrl	ImageName
1	Title	Description	Content	2022-05-19 13:54:49.843	https://storage.googleapis.com/download/storage/...	e742a5d4-6672-478c-8a1e-09dec436cf00.png

Рисунок 2.9 — Запис у базі даних

Якщо перейти у Google Cloud Console, та відкрити сховище, можна побачити, що файл завантажився у bucket успішно (рисунок 2.10). Тут записана його назва, його розмір, та дата завантаження.

Filter by name prefix only ▾ | Filter | Filter objects and folders

☐	Name	Size	Type	Created	Storage class	Last modified	Public access	
☐	e742a5d4-6672-478c-8a1e-09dec43...	21.1 KB		May 19, 2022, ...	Standard	May 19, 2022, ...	Public to internet	Copy URL

Рисунок 2.10 — Запис про зображення у Google Cloud

Останнім етапом перевірки працездатності було створення простої веб-сторінки, на якій розміщено зображення, використовуючи посилання, яке повертається із бази даних при запиті. Результат зображено на рисунку 2.11

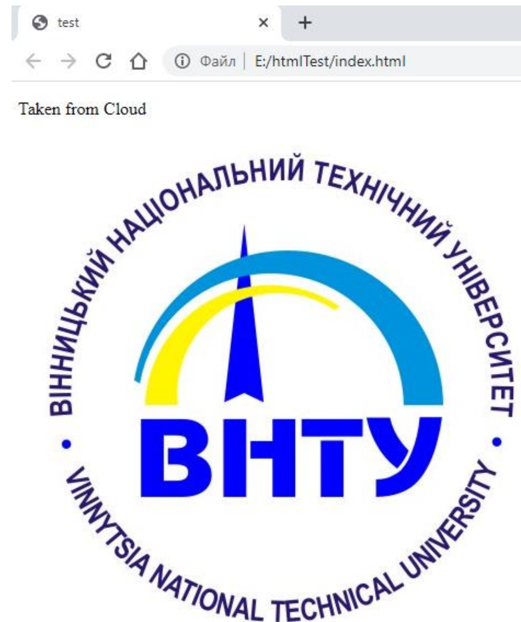


Рисунок 2.11 — Вигляд сторінки, яка відображує зображення з Google Cloud

Налаштування Google Cloud для використання у програмному проекті працює успішно. Файл завантажується у хмарне сховище, у базу даних зберігається посилання на зображення, яке успішно відображується на веб-сторінці.

2.6 Реалізація технології вибірки даних GrapQL у веб-додатку

Для реалізації запитів за допомогою GraphQL було прийнято рішення використовувати бібліотеку HotChocolate, за допомогою якої можна зручно працювати із технологією GraphQL прямо у середовищі розробки .NET [14].

Ця технологія дозволить вибирати на рівні представлення лише ту інформацію, яка потрібна в конкретний момент часу, що дозволить запобігти «over fetching», тобто перевантаження. Так, при використанні підходу REST у повній мірі, потрібно було б створювати декілька endpoint для сутності News,

один з яких повертав би інформацію лише про назву та заголовок та опис новини, для відображення картки із новиною на головній сторінці, а інший повертав би усю інформацію про новину, щоб користувач міг її прочитати. А якщо потреби із часом зміняться, це могло б збільшити кількість кінцевих точок.

Схема взаємодії GraphQl із додатком зображена на рисунку 2.12.

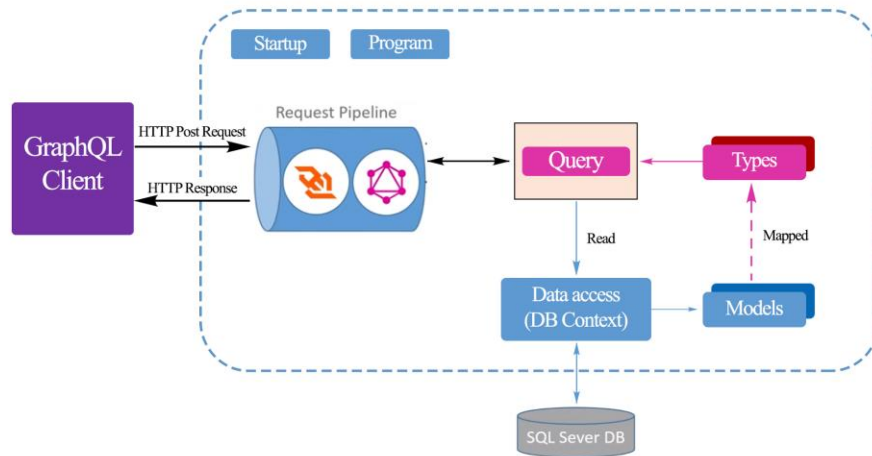


Рисунок 2.12 — Взаємодія GraphQL із серверною частиною

Приходить запит від клієнту GraphQL, який обробляється за допомогою класу Query. В свою чергу всередині класу Query описані методи, для читання даних із бази даних. База даних отримує інформацію, у вигляді моделі сутності, яку потрібно привести до типу, який буде відображатись на рівні представлення, тому відбувається «mapping» до типів, об’єкти яких вже повертаються у вигляді відповіді на запит.

2.6.1 Опис моделі запиту GraphQL

Розглянемо клас Query, який у собі буде містити основні методи для вибірки інформації. В лістингу 2.22 зображено приклад методу, який повертає дані про новини у вигляді IQueryable<News>, отримуючи їх із бази даних. Для цього, як параметр цей метод приймає контекст даних, для роботи безпосередньо із базою даних.

Лістинг 2.22 — Метод GetNews

```
[UseDbContext(typeof(Context.Context))]
        [UseFiltering]
        [UseSorting]
        public IQueryable<Entities.News>
GetNews([ScopedService] Context.Context context)
        {
            return context.News;
        }
```

Цей метод викликається безпосередньо з клієнту GraphQL, який не зможе передати параметр типу Context, у метод, для того щоб він продовжив роботу із базою даних. Тому, об’єкт контексту потрібно передати, використовуючи ін’єкцію залежностей (лістинг 2.23).

Ін’єкція залежностей, дає можливість впроваджувати об’єкти класів в конструктори, та методи. Це потрібно для того, щоб усі класи, описані у додатку Web-API використовували один і той самий об’єкт.

Лістинг 2.23 — Ін’єкція контексту у методи та конструктори

```
services.AddPooledDbContextFactory<Context.Context>(provider
=>
provider.UseSqlServer(Configuration.GetConnectionString("VNT
U_ComputingTechnologiesDepartmeent")));
            services.AddScoped<Context.Context>(p =>
p.GetRequiredService<IDbContextFactory<Context.Context>>().C
reateDbContext());
```

Використання PooledDbContextFactory потрібно для паралельного виконання запитів, які додаток отримує від клієнту. Якщо зареєструвати звичайний контекст, при спробі виконання декількох методів одночасно, виникне помилка, і лише один із клієнтів, який робив запит, отримає результат.

Використовується метод UseSqlServer, для зазначення бази даних, яка буде використовувати, і у параметри цього методу повинна передаватись стрічка підключення. Тому було описано логіку для отримання стрічки підключення із конфігураційного файлу.

Аналогічні операції було проведено, для отримання об'єктів викладачів із бази даних. Окрім цього, у об'єктах сутностей присутні поля, які не потрібні на рівні представлення, а для сутності викладачів необхідно описати додаткову логіку для отримання поля «Merits». Тому потрібно описати типи сутностей, які буде отримувати клієнт, після виконання запиту.

2.6.2 Опис типів GraphQL

Для того, щоб зазначити, що клас є типом сутності, необхідно реалізувати наслідування від базового об'єкту типу «ObjectType<T>». Щоб створити правила, для типу сутності, який буде використовуватись на рівні представлення, потрібно перевизначити метод Configure. Приклад перевизначеного методу Configure для сутності Teacher приведено в лістингу 2.24.

За допомогою IObjectTypeDescriptor, який метод отримує як параметр, можна створювати деякі додаткові правила.

Лістинг 2.24 — Перевизначення методу Configure класу TeacherType

```
protected override void
Configure(IObjectTypeDescriptor<Teacher> descriptor)
        {
            descriptor.Description("Represents teacher model
for computing technology web-site");

            descriptor.Field(f => f.ImageName).Ignore();

            descriptor.Field(f => f.Merits)
                .Type<ListType<StringType>>()
                .Resolve(context =>
                {
                    var teacher = context.Parent<Teacher>();
                    return
teacher.Merits.Split(';').ToList();
                });
        }
```

За допомогою методу Field, можна отримати поле, для якого будуть описуватись правила. Так, наприклад, поле ImageName буде ігноруватись, так

як це поле не потрібне на рівні представлення. Воно зберігається у базі даних лише для ситуації, коли потрібно видалити зображення із сховища Google Cloud, тому користувачам не обов’язково знати про нього.

Окрім цього, для поля «Merits» описана додаткова логіка, перед тим як показувати його на веб сторінці. Через те, що база даних MS SQL Server не підтримує тип колекції, було прийнято рішення зберігати у базі даних інформацію з поля «Merits» у текстовому полі, використовуючи розділовий знак, щоб відділити кожен об’єкт списку. Тому, перед тим як відправляти інформацію на рівень представлення, потрібно перетворити поле типу стрічка у поле колекції.

У лістингу 2.24, видно, що для поля Merits присвоюється новий тип «ListType<StringType>», який буде зрозумілий для технології GraphQl.

Але для того, щоб використовувати ці типи їх потрібно зареєструвати у конфігурації.

2.6.3 Конфігурація GraphQL

Окрім вище зазначеної конфігурації ін’єкції контексту, потрібно описати ще додаткову логіку, для використання серверу GraphQL, а також реєстрації типів, які будуть використовуватись (лістинг 2.25).

Лістинг 2.25 — Приклад конфігурації сервісу на використання GraphQL

```
services
                .AddGraphQLServer()
                .AddType<NewsType>()
                .AddType<TeacherType>()
                .AddQueryType<Query>()
                .AddFiltering()
                .AddSorting();
```

У прикладі вище реєструється сервер GraphQL, зазначається використання типів NewsType та TeacherType, а як тип запиту використовується описаний клас Query.

Окрім цього описується можливість фільтрації та сортування об’єктів.

Останнім кроком у налаштуванні GraphQL, є реєстрація endpoint, на якому буде знаходитись середовище для роботи із технологією. Для цього у методі Configure, у секції, у якій реєструються кінцеві точки потрібно додати MapGraphQL (лістинг 2.26);

Лістинг 2.26 — Приклад додавання endpoint GraphQL

```
app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
                endpoints.MapGraphQL();
            });
```

За замовчування endpoint, для використання GraphQL, можна знайти, якщо додати до адреси стрічку «/graphql»

Для перевірки працездатності можна описати перший запит, який повинен повернути інформацію із бази даних (рисунок 2.13).

В запиті описано відбір ім’я та прізвища викладача, а також поля «merits».

Рисунок 2.13 — Приклад виконання запиту GraphQL

Користувач отримує лише ту інформацію, яка йому потрібна у конкретний момент часу, і не засмітив тіло відповіді не потрібними даними. Окрім цього поле «Merits» відображається саме так, як його очікують на рівні представлення, а при спробі включити у тіло запиту поле «ImageName» користувач отримає помилку, що свідчить про успішну реєстрацію моделей для роботи із GraphQL.

Схема взаємодії запитів із моделями приведена на рисунку 2.14.

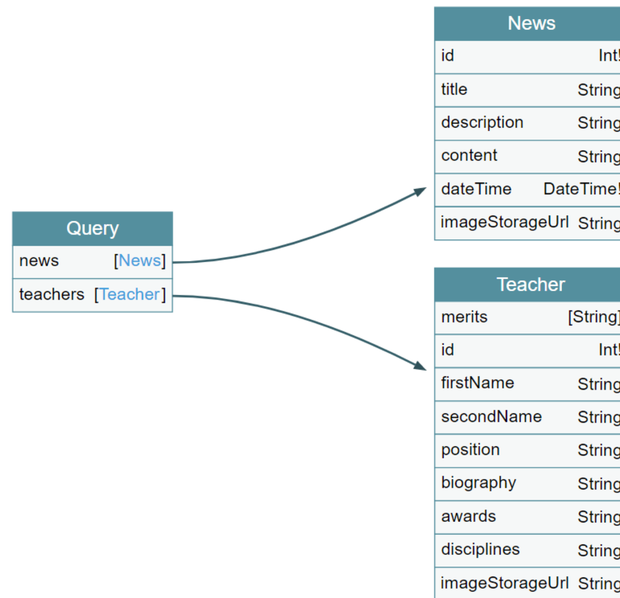


Рисунок 2.14 — Схема взаємодії запиту із моделями

У схемі зазначається взаємодія із сутностями, через клас Query, та описується яка інформація доступна для запиту із серверу.

У проекті вдалось поєднати підходи REST та GraphQl, при вдалому використанні яких, вони нівелюють недоліки один одного. Тому розробники рівня представлення можуть отримувати як усю інформацію із бази даних, про конкретні об’єкті, так і вибіркову інформацію, яка потрібна у конкретний момент.

Окрім цього, за допомогою технології Entity Framework Core вдалось налаштувати комунікацію проекту із базою даних, тому вся інформація буде зберігатись у СУБД, яку вибере користувач.

За допомогою технології Swagger було впроваджено документацію коду, та зручне середовище для тестування API. Завдяки цьому, розробники рівня представлення мають краще розуміння, як працювати із серверною частиною додатку.

3 ПРОЕКТУВАННЯ ВИСОКОРІВНЕВОЇ АРХІТЕКТУРИ ТА РЕАЛІЗАЦІЯ ІНСТРУМЕНТАРІЮ СЕРВЕРНОЇ ЧАСТИНИ ІНФОРМАЦІЙНОЇ СИСТЕМИ

В розроблюваній інформаційні системі для взаємодії між клієнтською та серверною частинами було розроблено API, яке поєднує REST та GraphQl. Отже, для усіх динамічних сторінок потрібно описати функціональні можливості або інструментарій API, щоб адміністратор міг динамічно додавати інформацію.

3.1 Проектування високорівневої архітектури серверної частини веб-додатку

Так як серверна частина була описана за допомогою мови програмування C#, що є об'єктно орієнтованою мовою програмування, у програмному рішенні описано багато класів, які взаємодіють між собою. Це зроблено для того, щоб проект був гнучким та легко розширювався.

Загальна діаграма класів розподіленого слабкозв'язаного серверного додатку зображена на додатку В. На ній подано загальну архітектуру серверної частини — як контролери взаємодіють із репозиторіями, репозиторії в свою чергу взаємодіють із базою даних, а за допомогою класів сервісів, об'єкти із бази даних приводяться до виду моделей відповіді. Після чого клас контролеру відправляє інформацію у відповідь на запит клієнтської частини.

Точкою взаємодії між серверною та клієнтською частиною веб-додатку є контролери, які повертають усю запитану інформацію. Контролери напряму не працюють із базою даних. Вони працюють із репозиторіями (рисунок 3.1) для отримання інформації із бази даних та із сервісами для обробки інформації (рисунок 3.2).

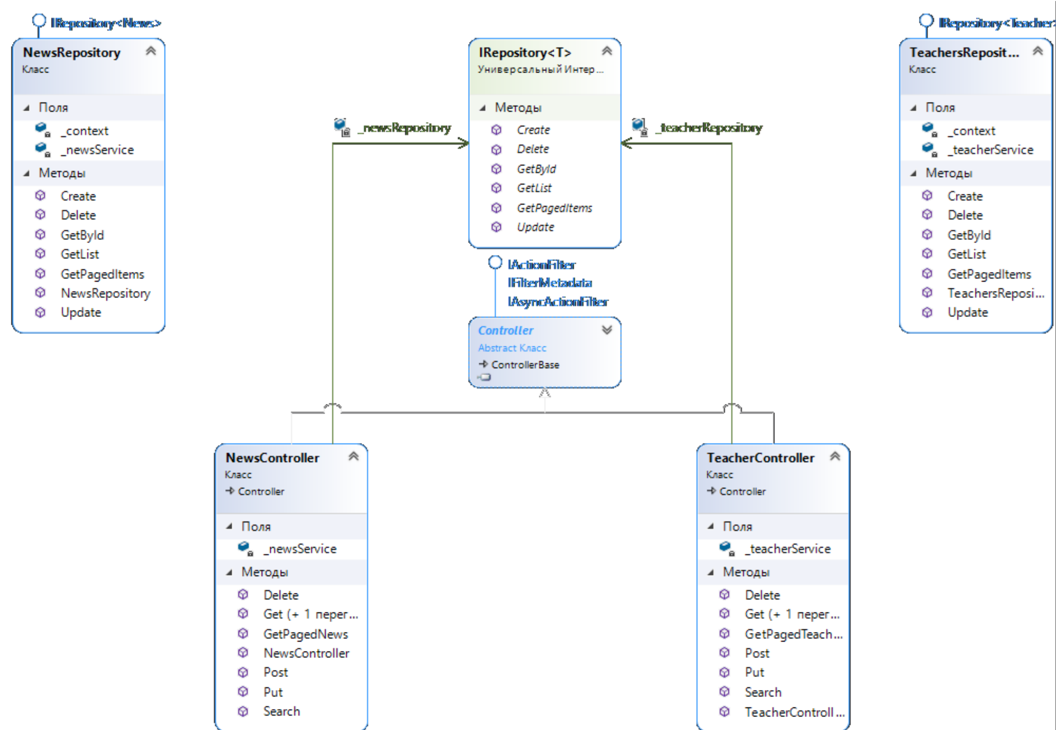


Рисунок 3.1 — Схема зв’язків між класами контролерів та репозиторіїв

При розробці інформаційної системи, яку потрібно підтримувати, важливо писати «чистий» та легко розширюваний код. Для цього існують принципи SOLID. Принципи SOLID — це акронім, від перших букв п’яти принципів об’єктно орієнтованого програмування:

— S — single responsibility Principle — тобто принцип єдиної відповідальності, коли клас відповідальний лише за одну функцію;

— O — open-closed Principle — тобто принцип відкритості закритості, коли клас повинен бути відкритим для розширення але закритим для модифікації;

— L — Liskov substitution Principle — тобто принцип підстановки Лісков, коли клас-наслідник може бути підставлений замість базового класу;

— I — interface segregation Principle — тобто принцип розділення інтерфейсу, коли один інтерфейс повинен бути відповідальним лише за одну реалізації;

— D — dependency inversion Principle — тобто принцип інверсії залежностей, коли сутності повинні залежати від абстракцій, а не від конкретних реалізацій.

Для дотримання принципу інверсії залежностей SOLID реалізація відділена від абстракції. Тому контролери реалізують інтерфейс типу IRepository, та працюють з ним. Конкретна реалізація цього інтерфейсу ініціалізується за допомогою dependency injection.

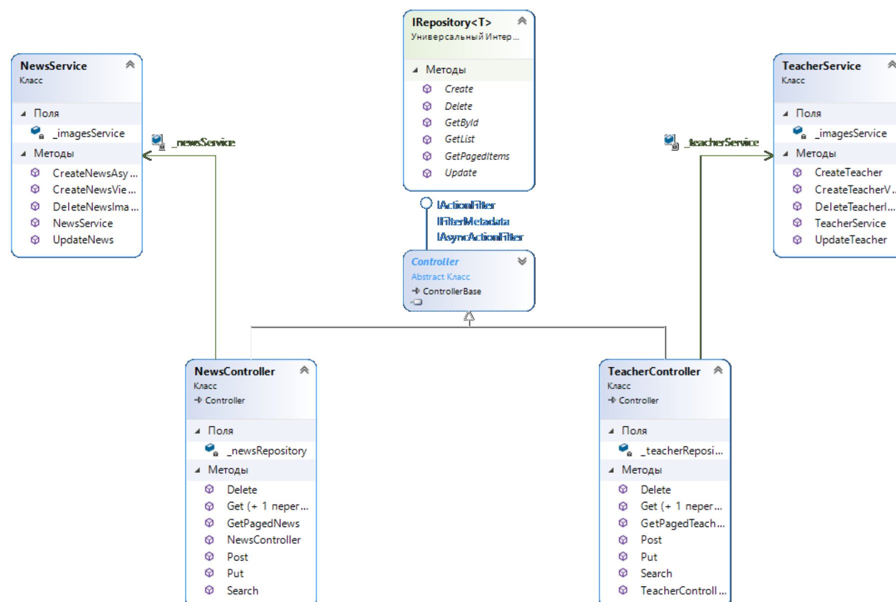


Рисунок 3.2 — Схема зв’язків між класами контролерів та сервісів

Репозиторії в свою чергу працюють із контекстом даних (рисунок 3.3), який зв’язаний з базою даних. З його допомогою відбувається робота безпосередньо із базою даних.

Класи сервісів конкретної сутності взаємодіють із сервісом, який працює із зображеннями (рисунок 3.4). У ньому описані функціональні можливості збереження та видалення зображення у хмарній платформі Google Cloud.

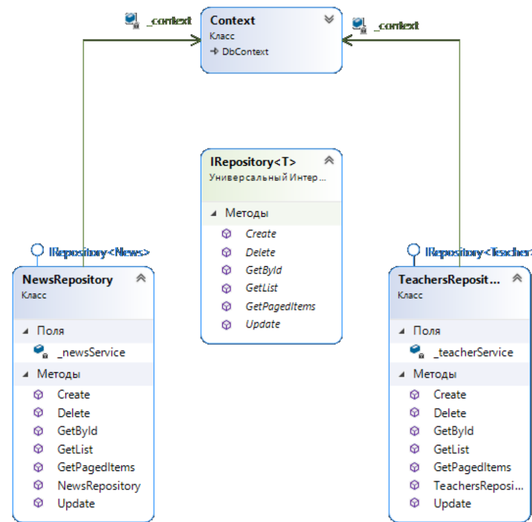


Рисунок 3.3 — Схема зв’язків між класами репозиторіїв із контекстом даних

Таке розмежування, коли кожен сервіс відповідає лише за одну функцію, відповідає першому принципу SOLID, а саме Single Responsibility (єдиної відповідальності). Це потрібно для більшої гнучкості коду. Так, кожен сервіс, який описаний для конкретної сутності може використовувати загальний клас сервісу, який відповідає за роботу із зображеннями, і розробнику не потрібно описувати окремо класи які будуть зберігати зображення викладачів та новин.

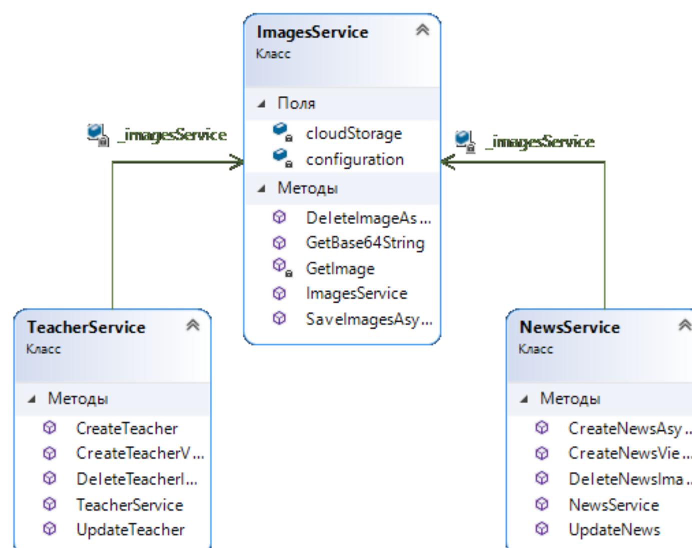


Рисунок 3.4 — Схема зв’язків між класами сервісів сутностей із сервісом зображень

Окрім цього класи конкретної сутності працюють із моделями даних (рисунок 3.5). Вони конвертують модель запиту, який приходить з рівня представлення у модель сутності, яку зберігають в базі даних, а також у модель відповіді, яку серверна частина відправляє у відповідь на запит клієнтської частини.

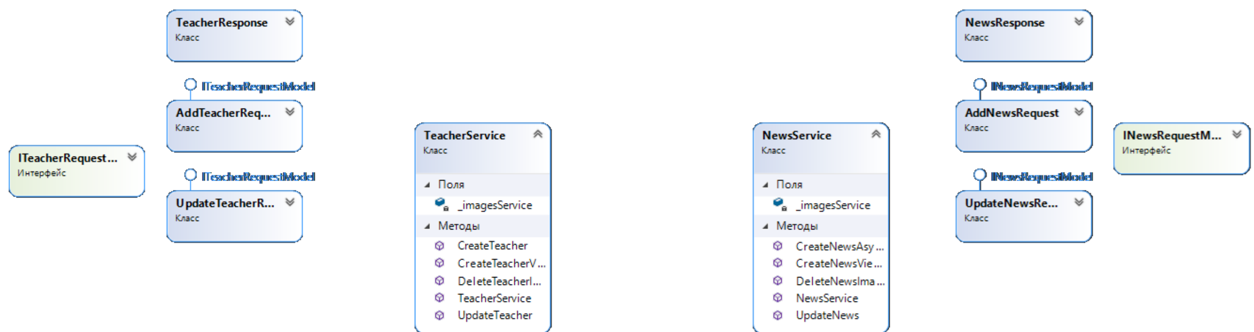


Рисунок 3.5 — Схема зв’язків між класами сервісів сутностей із моделями

Використання класів-сервісів допомагає уникнути повторюваності коду, адже конвертація моделі запиту у модель сутності відбувається у багатьох функціях. І замість того, щоб у кожній такій функції описувати логіку для конвертації моделей, можна викликати метод сервісу.

3.2 Відправлення запитів на серверну частину веб додатку

У серверній частині описані функції для отримування, редагування, видалення та додавання інформації. Окрім цього є можливість отримання інформації за сторінками та пошуку по сторінці.

У веб-додатку описано роутинг — це шлях, на який потрібно відправляти запит, для отримання відповіді у вигляді даних та коду статусу.

Ці функції викликаються розробником адміністративної панелі. Він описує візуальний інтерфейс для адміністратора, який використовує кнопки на веб-сайті для відправлення запитів серверній частині. Окрім цього, якщо функція потребує вказання певних параметрів, користувач вводить їх у текстові поля, представлені у інтерфейсі веб-сайту.

Для отримання новин, що зберігаються у базі даних потрібно запустити додаток із описаною серверною частиною. Під час тестування інформація буде знаходитись на локальному хості комп'ютера на відповідному порті. Тому для отримання усіх новин, які зараз знаходяться у базі даних, потрібно відправити запит за шляхом «https://localhost:44331/News». Результат виконання запиту приведено на рисунку 3.7.

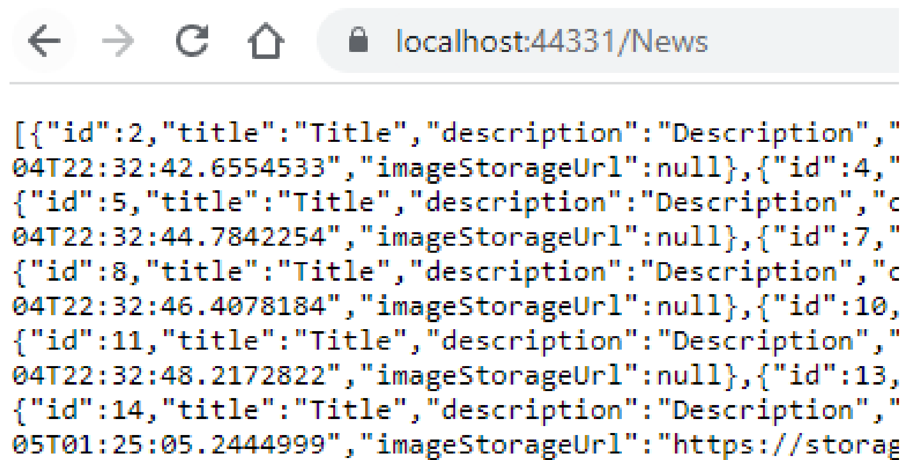


Рисунок 3.7 — Приклад виконання запиту отримання усіх новин

Також, можна отримати новину за певним id. Для цього, до шляху «https://localhost:44331/NewsId» потрібно додати параметр із конкретним значенням id того об'єкту, який шукає користувач (рисунок 3.8).

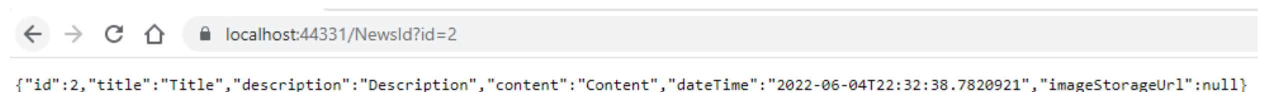


Рисунок 3.8 — Приклад виконання запиту отримання новини за id

Для того щоб додати новину до бази даних потрібно використовувати адресу «https://localhost:44331/InsertedNews», а як параметр передати інформацію про заголовок новини, її опис та контент. Для тестування цього запиту використовується інструмент для тестування Postman, адже у ньому можна перевірити запит написавши адресу (рисунок 3.9). Параметри

розділяються знаком «&». У випадку, якщо користувач хоче додати зображення до новини, його потрібно додати у тіло запиту. У випадку, якщо об’єкт новини додався успішно, серверна частина разом із кодом статусу 200 (OK) повертає новину, яка була додана у базу даних, одразу із посиланням на зображення, якщо воно є.

POST https://localhost:44331/InsertedNews?title=title&description=desc&content=content

Send Save

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
title	title	
description	desc	
content	content	
Key	Value	Description

Body Cookies Headers (5) Test Results Status: 200 OK Time: 4.21 s Size: 328 B Save Response

Pretty Raw Preview Visualize JSON

```
{
    "id": 16,
    "title": "title",
    "description": "desc",
    "content": "content",
    "dateTime": "2022-06-06T00:49:00.3356428+03:00",
    "imageStorageUrl": null,
    "imageName": null
}
```

Рисунок 3.9 — Приклад виконання запиту на додавання новини

Аналогічно із функцією для додавання новини у базу даних використовується функція для редагування даних, але потрібно додати параметр id, у який потрібно присвоїти id того елементу, що буде редагуватись.

Функція видалення приймає id як параметр, і видаляє об’єкт із бази даних.

Окрім цього, описана функція для пошуку новини за заголовком та описом. Ця функція шукає новини, у яких в описі чи заголовку є стрічка, яка передається у вигляді параметру, і повертає усі новини де ця стрічка була

знайдена. У прикладі, описаному на рисунку 3.10, зображено пошук новин, у яких в заголовку або описі зустрічається стрічка «title».

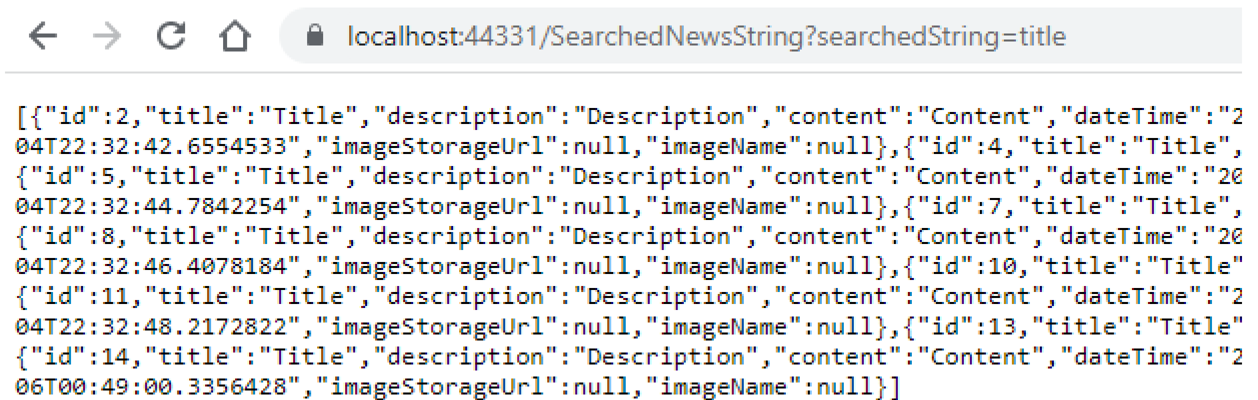


Рисунок 3.10 — Приклад функції пошуку за стрічкою

Остання функція, описана у REST-API, потрібна для отримання новин, які будуть відображатись на певній сторінці, а також інформація про загальну кількість сторінок, та чи наступна сторінка із новинами не буде пустою (рисунок 3.11).

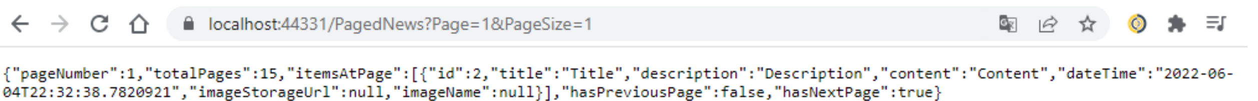


Рисунок 3.11 — Приклад функції отримання новин за сторінкою

Ця функція як параметр приймає номер сторінки, для якої потрібно отримати новини, а також розмір сторінки — тобто кількість новин, які будуть відображатись на одній сторінці. Аналогічні функції також були описані для інформації про викладачів.

Для того, щоб отримувати лише потрібні поля про новини із бази даних, до проекту була підключена технологія GraphQl. Для того щоб її використовувати, потрібно до адреси, на якій хоститься серверна частина додати «/graphql». На цю адресу потрібно присилати запити, але їх зовнішній вигляд відрізняється від тих, які відсилаються на адресу REST-API. У цих

запитах потрібно описати лише ті поля, які потрібно отримувати від серверної частини. Приклад запиту, який потрібно відправляти на адресу graphql зображено в лістингу 3.1

Лістинг 3.1 — Приклад запиту GraphQl

```
query{
  news {
    title
    description
    imageStorageUrl}
}
```

В лістингу 3.1 зображено запит, який від серверної частини отримує не усі новини, а лише інформацію про заголовок, опис та посилання на зображення. Цей запит можна використати на головній сторінці веб-сайту, де потрібно вказати останні новини, але не відображати їх контентз метою економії місця.

Також можна додати фільтрацію за певним параметром. Приклад для отримання заголовку, опису та посилання на зображення для новини, у якої id дорівнює 5 приведено в лістингу 3.2

Лістинг 3.2 — Приклад запиту GraphQl із фільтрацією

```
query{
  news(where: {id: {eq: 5}}) {
    title
    description
    imageStorageUrl}
}
```

І остання функція, яка описана у GraphQl — це можливість отримувати дані, відсортовані за певним параметром. Приклад, для отримання інформації відсортованої за датою приведено в лістингу 3.3

Цей запит може використовуватись для відображення новин на головній сторінці, де спочатку новини потрібно показувати від нових до старих.

Лістинг 3.3 — Приклад запиту GraphQl із сортуванням за датою

```
query{
  news(order: {dateTime:DESC}) {
    title
    description
    imageStorageUrl
    dateTime}
}
```

Отже, для додавання інформації у базу даних використовується метод POST. Розробник адміністративної панелі, після заповнення усіх полів через веб-інтерфейс, надсилає запит на серверну частину, виконуючи метод POST, у який передається усі необхідна інформація. Аналогічно з цим, для представлення усіх новин, які зберігаються у базі даних, розробник адміністративної панелі виконує метод GET, а усю отриману інформацію розміщує на сайті.

Для відображення новин на головній сторінці, розробник рівня представлення може використовувати GraphQl, і отримати тільки ті поля з об’єкту новини, які йому потрібні для відображення на веб-сторінці.

ВИСНОВКИ

В комплексній бакалаврській дипломній роботі було спроектовано та реалізовано розподілений слабкозв'язаний серверний додаток мовою C# для клієнт-серверної інформаційної системи підрозділу навчального закладу.

Проаналізовано найпопулярніші технології для роботи із базою даних, використовуючи середовище розробки .NET, документації коду, вибірки інформації із бази даних, збереження зображень у хмарному середовищі, системи управління базами даних. Також проведено аналіз архітектурних підходів розробки для покращення роботи у команді.

Визначено оптимальний стек технологій для розробки слабкозв'язаного серверного додатку для клієнт-серверної інформаційної системи підрозділу навчального закладу. Для роботи із базою даних через середовище розробки .NET використана ORM система Entity Framework. Для вибірки інформації з бази даних використано технологію GraphQl. З метою економії ресурсів файлової системи серверу файли зображень зберігаються в хмарному середовищі Google Cloud.

Також в роботі спроектовано високорівневу архітектуру серверної частини веб-додатку та описано інструмантарій і функціональні можливості API для додавання, редагування, видалення та отримання інформації із бази даних. Для демонстрації функціональниих можливостей розробленого серверного додатку проведене тестування відправлення запитів на сервер, яке показало корректність роботи додатку. Важливим моментом при написанні проекту є його гнучкість, адже веб-додаток буде доповнюватись новими функціональними можливостями. Програмний код написаний із дотриманням принципів об'єктно-орієнтованого програмування SOLID для забезпечення можливості гнучкої та адаптивної підтримки при подальшому розвитку розробленої інформаційної системи підрозділу навчального закладу.

Результати комплексної бакалаврської дипломної роботи впроваджено мережі ВНТУ як сайт кафедри обчислювальної техніки (додаток Р)

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Аналіз архітектури розподіленого слабкозв'язного серверного додатку інформаційної системи [Електронний ресурс] / О.В. Войцеховська, А.К. Рижков — 2022. — conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2022/author/submission/15103
2. Використання патерну «Repository» при проектуванні розподіленого слабкозв'язного серверного додатку [Електронний ресурс] / Войцеховська О. В., Рижков А. К. — 2022 р. Режим доступу: https://conferences.vntu.edu.ua/index.php/mn/mn2022/author/submission/16266
3. What is Entity Framework Core [Електронний ресурс]. Режим доступу: https://www.entityframeworktutorial.net/what-is-entityframework.aspx.
4. Creating Entity Framework Data Model. [Електронний ресурс]. Режим доступу:https://www.entityframeworktutorial.net/entityframework6/create-entity-data-model.aspx.
5. Getting Started with Entity Framework 6. [Електронний ресурс]. Режим доступу: https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application.
6. ADO.NET Overview. [Електронний ресурс]. Режим доступу: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview.
7. What is REST. [Електронний ресурс]. Режим доступу: https://restfulapi.net/.
8. API Documentation & Design Tools with Swagger. [Електронний ресурс]. Режим доступу: https://swagger.io/
9. What is GraphQL [Електронний ресурс]. Режим доступу: https://habr.com/ru/post/326986/
10. Differences between MySQL and SQL Server [Електронний ресурс]. Режим доступу: https://www.geeksforgeeks.org/difference-between-mysql-and-ms-sql-server/

11. What is WEB-API [Електронний ресурс]. Режим доступу: https://www.tutorialsteacher.com/webapi/what-is-web-api

12. What are HTTP Status Code [Електронний ресурс]. Режим доступу: https://umbraco.com/knowledge-base/http-status-codes/

13. Create A Storage Bucket In Google Cloud Platform [Електронний ресурс]. Режим доступу: https://www.c-sharpcorner.com/article/create-a-storage-bucket-in-google-cloud-platform/

14. Introduction to GrahpQL [Електронний ресурс]. Режим доступу: https://graphql.org/learn/

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп’ютерної інженерії
Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
_____ проф., д.т.н. О.Д. Азаров
«__» ________ 2022 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання бакалаврської дипломної роботи
Клієнт-серверна інформаційна система підрозділу навчального закладу з можливістю розгортання в хмарному середовищі. Частина 2. «Розробка розподіленого слабкозв’язаного серверного додатку мовою C#»
08-23.КБДР.046.00.000 ТЗ

Керівник роботи: к.т.н.,доц. каф. ОТ.
_______________Войцеховська О.В.

Виконав: студент гр. 1КІ-20мс
____________________Рижков А. К.

2022

1 Підстава для виконання дипломної роботи (ДР):

— актуальність розробки полягає у необідності забезпечення структурного підрозділу навчального закладу клієнт серверною інформаційною системою шляхом розробки серверного додатку для неї з використанням сучасних архітектурних підходів та технологій розробки серверної частини;

— наказ про затвердження теми бакалаврської дипломної роботи.

2 Мета і призначення ДР:

— метою роботи є розробка серверної частини веб-додатку інформаційної системи структурного підрозділу навчального закладу, яка буде складатись із бази даних для зберігання інформації, серверної частини для обробки інформації та хмарного середовища Google Cloud для зберігання файлів;

— призначення розробки — виконання комплексної бакалаврської дипломної роботи із подальшою пітримкою та розвитком.

3 Джерела розробки:

— контент для наповнення бази даних інформаційної системи;

— інформація по технологіям розробки серверної частини.

4 Технічні вимоги до виконання ДР:

— динамічні сторінки, маніпуляції над якими відбуваються через серверну частину, повинні заповнюватись через адміністративну панель, для чого розробник адміністративної панелі використовує описане в серверній частині API;

— статичні сторінки заповнюються розробником рівня представлення і не мають зв’язку із серверною частиною;

— серверна частина повинна коректно відповідати на запити клієнтської частини, обробляти помилки, при отриманні не валідної

інформації від клієнтської частини;

— серверна частина повинна повертати відповідний статусний код із повідомленням про помилку.

5 Етапи ДР та очікувані результати

Етапи роботи та очікуванні результати приведено в Таблиці А.1.

Таблиця А.1 — Етапи МКР

№ з/п	Назва етапів дипломного проекту (роботи)	Термін виконання		Очікувані результати
		початок	закінчення	
1	Аналіз завдання	08.02.22		Задачі дослідження
2	Огляд архітектурних підходів проектування інформаційних систем	09.02.22	12.02.22	Аналітичний огляд джерел
3	Аналіз та вибір технологій для роботи із базою даних та вибір СУБД	12.02.22	20.02.22	Розділ 1
4	Огляд технологій вибірки даних, документації коду та вибір хмарного середовища	20.02.22	28.02.22	Розділ 1
5	Проектування високорівневої архітектури серверної частини веб-додатку	29.02.22	14.03.22	Розділ 2
6	Проектування бази даних серверного додатку	15.03.22	21.03.22	Розділ 2
7	Розробка та програмна реалізація рівня доступу до даних серверної частини	22.03.22	20.04.22	Розділ 2
8	Розробка та програмна реалізація рівня бізнес-логіки серверної частини	21.04.22	15.05.22	Розділ 2
9	Перевірка працездатності серверної частини веб-додатку	16.05.22	30.05.22	Розділ 3
10	Оформлення пояснювальної записки та ілюстративного матеріалу	01.06.22	13.06.22	ПЗ, додатки, презентація

6 Матеріали, що подаються до захисту ДР

Пояснювальна записка КБДР, графічні та ілюстративні матеріали, протокол попереднього захисту роботи на кафедрі, відзив наукового керівника, рецензія опонента, анотації до КБДР українською та іноземною

мовами, нормоконтроль про відповідність оформлення КБДР діючим вимогам.

7 Порядок контролю виконання та захисту ДР

Виконання етапів графічної та розрахункової документації ДР контролюється науковим керівником згідно зі встановленими термінами. Захист ДР відбувається на засіданні Державної екзаменаційної комісії, затвердженою наказом ректора.

8 Вимоги до оформлення ДР

Вимоги викладені в методичних вказівках до дипломного проектування, ДСТУ_3008-2015, ДСТУ 3974-2000 «Правила виконання дослідно-конструкторських робіт. Загальні положення» та діючого ГОСТ 2.114-95 ЕСКД.

9 Вимоги щодо технічного захисту інформації в КБДР з обмеженим доступом відсутні.

Технічне завдання до виконання прийняв ______ Андрій Рижков

ДОДАТОК Б

Структурна схема високорівневої архітектури клієнт-серверної інформаційної системи

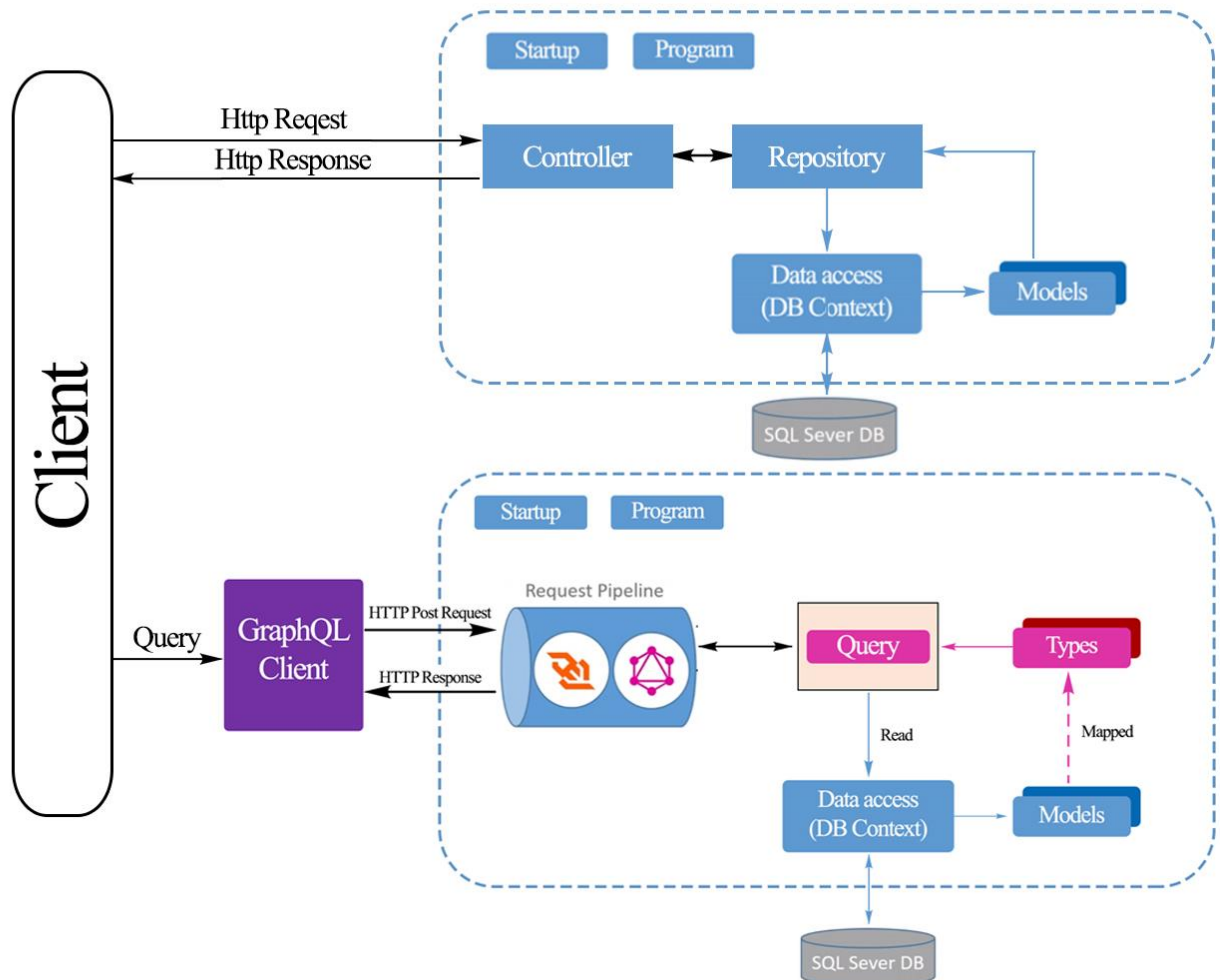


Рисунок Б.1 — Структурна схема високорівневої архітектури клієнт-серверної інформаційної системи

ДОДАТОК В

Діаграма класів розподіленого слабкозв’язаного серверного додатку

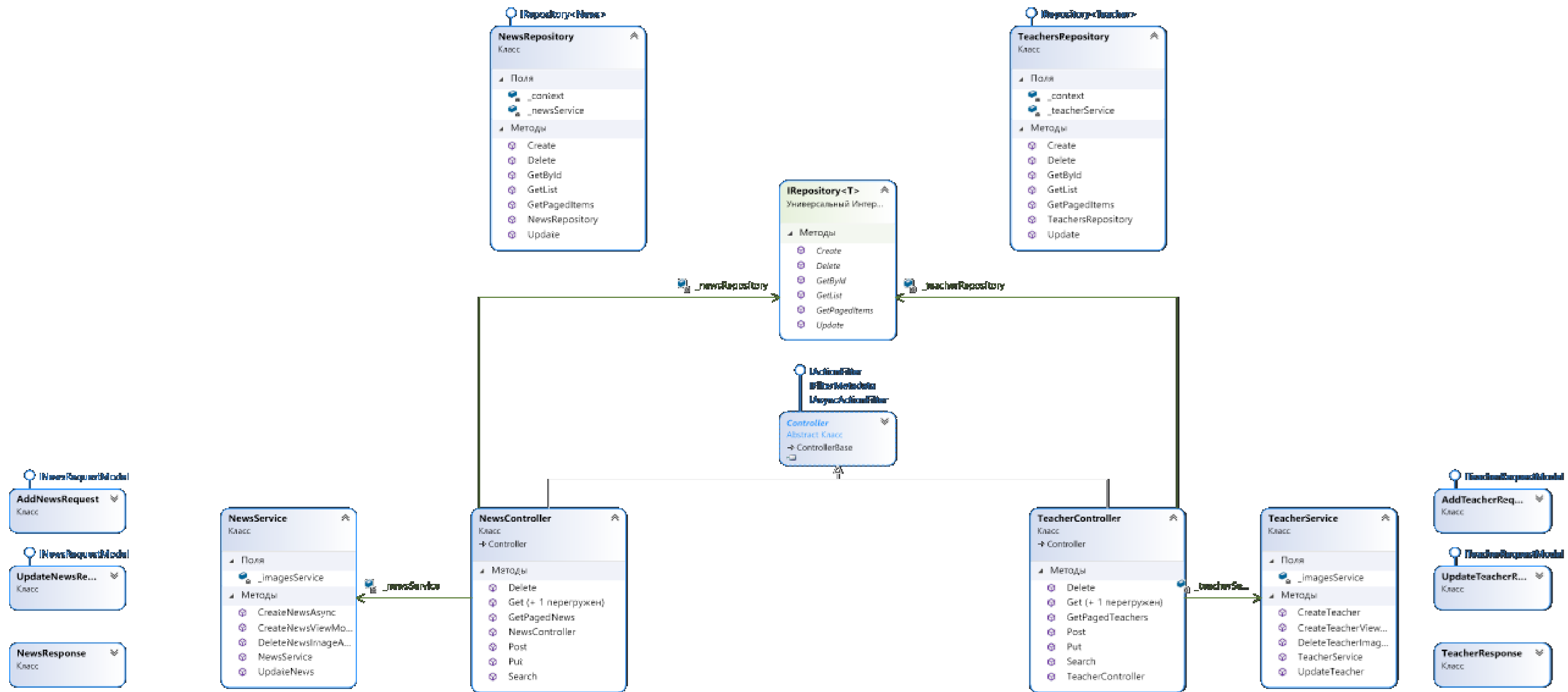


Рисунок В.1 — Діаграма класів розподіленого слабкозв’язаного серверного додатку

ДОДАТОК Г

Схема взаємодії запиту GraphQl із моделями даних

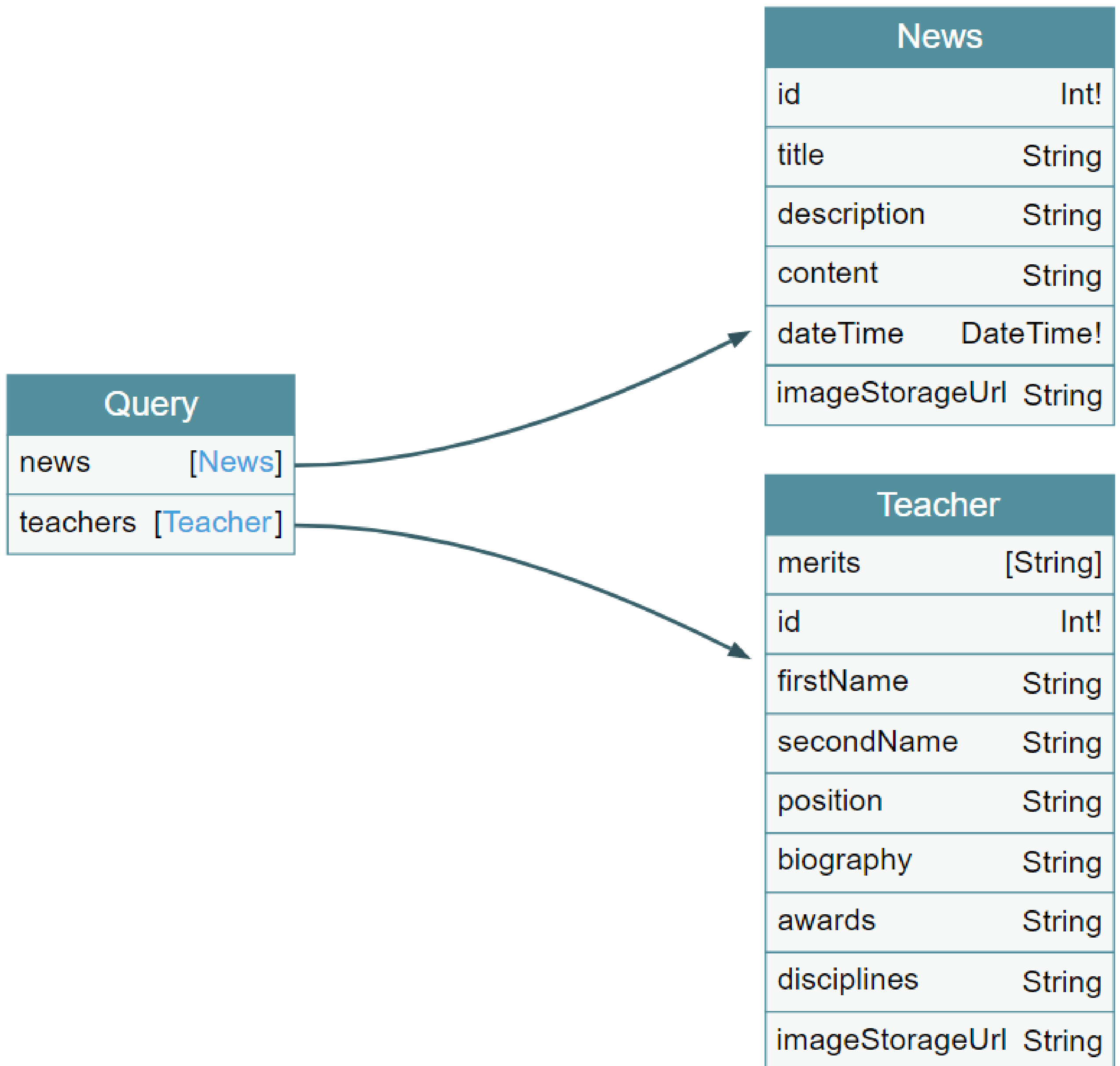


Рисунок Г.1 — Схема взаємодії запиту GraphQl із моделями даних

ДОДАТОК Д

Структура бази даних слабкозв’язаного серверного додатку

Teachers

	Имя столбца	Тип данных	Разрешить ...
🔑	Id	int	☐
	FirstName	nvarchar(MAX)	☑
	SecondName	nvarchar(MAX)	☑
	MiddleName	nvarchar(MAX)	☑
	Position	nvarchar(MAX)	☑
	Merits	nvarchar(MAX)	☑
	Biography	nvarchar(MAX)	☑
	Awards	nvarchar(MAX)	☑
	Disciplines	nvarchar(MAX)	☑
	ImageStorageUrl	nvarchar(MAX)	☑
	ImageName	nvarchar(MAX)	☑
			☐

News

	Имя столбца	Тип данных	Разрешить ...
🔑	Id	int	☐
	Title	nvarchar(MAX)	☑
	Description	nvarchar(MAX)	☑
	[Content]	nvarchar(MAX)	☑
	DateTime	datetime2(7)	☐
	ImageStorageUrl	nvarchar(MAX)	☑
	ImageName	nvarchar(MAX)	☑
			☐

Рисунок Д.1 —Структура бази даних слабкозв’язаного серверного додатку

ДОДАТОК Е

Мапа веб-сайту

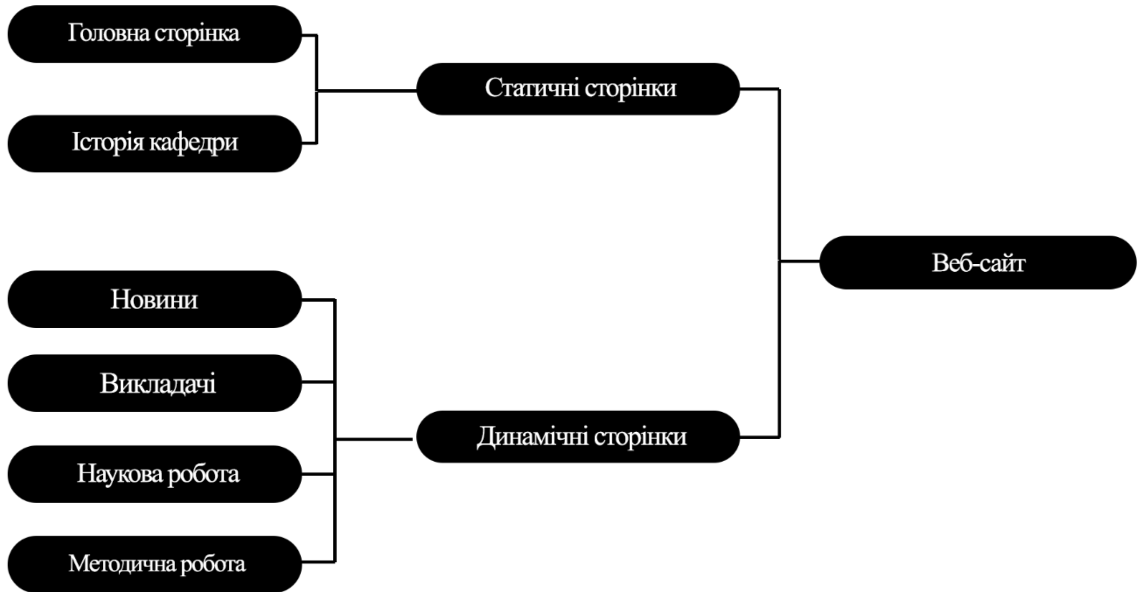


Рисунок Е. 1 — Мапа веб-сайту

ДОДАТОК Ж

Лістинг інтерфейсу NewsController

```
using Microsoft.AspNetCore.Mvc;
using ot_api.Entities;
using ot_api.Models;
using ot_api.Models.Pagination;
using ot_api.Repositories;
using ot_api.Services;
using System;
using System.Collections.Generic;
using System.Linq;

namespace ot_api.Controllers
{
    public class NewsController : Controller
    {
        private IRepository<News> _newsRepository;
        private NewsService _newsService;

        public NewsController(IRepository<News> newsRepository, NewsService newsService)
        {
            _newsService = newsService;
            _newsRepository = newsRepository;
        }

        /// <summary>
        /// Get all list of news from db
        /// </summary>
        /// <returns>List of news</returns>
        [HttpGet("News")]
        public ActionResult Get()
        {
            var news = _newsRepository.GetList();
```

```
            if (news == null)
            {
                return BadRequest();
            }

            List<NewsResponse> newsViewModels = new List<NewsResponse>();
            foreach (var item in news)
            {
                newsViewModels.Add(_newsService.CreateNewsViewModel(item));
            }
            return Ok(newsViewModels);
        }

        /// <summary>
        /// Get news on sepcific page
        /// </summary>
        /// <returns>News on sepcific page</returns>
        [HttpGet("PagedNews")]
        public ActionResult GetPagedNews(PageInfoRequest pageRequest)
        {
            var count = _newsRepository.GetList().Count;

            if (pageRequest.Page > (int)Math.Ceiling(count / (double)pageRequest.Page))
            {
                return NotFound("Can't find such page");
            }

            var newsInPage = _newsRepository.GetPagedItems(pageRequest.PageSize,
pageRequest.Page);
            return Ok(newsInPage);
        }

        /// <summary>
        /// Get news that are matching searched string from db
        /// </summary>
```

```
        /// <returns>List of news</returns>
        [HttpGet("SearchedNewsString")]
        public ActionResult Search(string searchedString)
        {
            return Ok(_newsRepository.GetList()
                .Where(news =>
news.Title.ToLower().Trim().Contains(searchedString.ToLower().Trim())
                || news.Description.ToLower().Trim().Contains(searchedString.ToLower().Trim())));
        }

        /// <summary>
        /// Get specific news from db
        /// </summary>
        /// <param name="id">Id of news you are looking for</param>
        /// <returns>News with specific id</returns>
        [HttpGet("NewsId")]
        public ActionResult Get(int id)
        {
            var news = _newsRepository.GetById(id);

            if (news == null)
            {
                return NotFound();
            }
            var newsViewModel = _newsService.CreateNewsViewModel(news);
            return Ok(newsViewModel);
        }

        /// <summary>
        /// Insert news into db
        /// </summary>
        /// <param name="news">News that you want to insert into db</param>
        /// <returns>Inserted news</returns>
        [HttpPost("InsertedNews")]
        public ActionResult Post(AddNewsRequest newsModel)
```

```
{
    if (newsModel == null)
    {
        return BadRequest();
    }
    var news = _newsService.CreateNewsAsync(newsModel).Result;

    _newsRepository.Create(news);
    return Ok(news);
}

/// <summary>
/// Update specific news from db
/// </summary>
/// <param name="news">Updated news</param>
/// <returns>Updated news</returns>
[HttpPut("EditedNews")]
public ActionResult Put(UpdateNewsRequest newsModel)
{
    var newsToUpdate = _newsRepository.GetById(newsModel.Id);
    if (newsModel == null)
    {
        return BadRequest();
    }
    if (newsToUpdate == null)
    {
        return NotFound();
    }

    var news = _newsService.CreateNewsAsync(newsModel).Result;
    _newsRepository.Update(news, newsModel.DeletePicture, newsToUpdate.ImageName);
    return Ok(news);
}

/// <summary>
```

```
    /// Deletes news with specific id from db
    /// </summary>
    /// <param name="id">Id of news that you want to delete</param>
    /// <returns>Id of deleted news</returns>
    [HttpDelete("DeleteNewsId")]
    public ActionResult Delete(int id)
    {
        if (!_newsRepository.GetList().Any(n => n.Id == id))
        {
            return NotFound();
        }
        _newsRepository.Delete(id);
        return Ok(id);
    }
}}}
```

ДОДАТОК К

Лістинг інтерфейсу TeacherController

```
using Microsoft.AspNetCore.Mvc;
using ot_api.Entities;
using ot_api.Models;
using ot_api.Repositories;
using ot_api.Services;
using System;
using System.Collections.Generic;
using System.Linq;

namespace ot_api.Controllers
{
    public class TeacherController : Controller
    {
        private IRepository<Teacher> _teacherRepository;
        private TeacherService _teacherService;

        public TeacherController(IRepository<Teacher> teacherRepository, TeacherService
newsService)
        {
            _teacherService = newsService;
            _teacherRepository = teacherRepository;
        }

        /// <summary>
        /// Get all list of teachers from db
        /// </summary>
        /// <returns>List of teachers</returns>
        [HttpGet("Teachers")]
        public ActionResult Get()
        {
            var teachers = _teacherRepository.GetList();
```

```
    if (teachers == null)
    {
        return BadRequest();
    }

    List<TeacherResponse> teacherViewModels = new List<TeacherResponse>();
    foreach (var item in teachers)
    {
        teacherViewModels.Add(_teacherService.CreateTeacherViewModel(item));
    }
    return Ok(teacherViewModels);
}

/// <summary>
/// Get specific teacher from db
/// </summary>
/// <param name="id">Id of Teacher you are looking for</param>
/// <returns>Teacher with specific id</returns>
[HttpGet("TeacherId")]
public ActionResult Get(int id)
{
    var teacher = _teacherRepository.GetById(id);
    if (teacher == null)
    {
        return NotFound();
    }
    var newsViewModel = _teacherService.CreateTeacherViewModel(teacher);
    return Ok(newsViewModel);
}

/// <summary>
/// Get teachers on sepcific page
/// </summary>
/// <returns>Teachers on sepcific page</returns>
[HttpGet("PagedTeachers")]
```

```
        public ActionResult GetPagedTeachers(int page, int pageSize)
        {
            var count = _teacherRepository.GetList().Count;
            if (page > (int)Math.Ceiling(count / (double)pageSize))
            {
                return NotFound("Can't find such page");
            }
            var teachersInPage = _teacherRepository.GetPagedItems(pageSize, page);
            return Ok(teachersInPage);
        }

        /// <summary>
        /// Get teacher that are matching searched string from db
        /// </summary>
        /// <returns>List of teachers</returns>
        [HttpGet("SearchedTeacherString")]
        public ActionResult Search(string searchedString)
        {
            return Ok(_teacherRepository.GetList()
                .Where(teacher =>
teacher.FirstName.ToLower().Trim().Contains(searchedString.ToLower().Trim())
                ||
teacher.SecondName.ToLower().Trim().Contains(searchedString.ToLower().Trim())));
        }

        /// <summary>
        /// Insert teachers into db
        /// </summary>
        /// <param name="teacher">Teachers that you want to insert into db</param>
        /// <returns>Inserted teachers</returns>
        [HttpPost("InsertedTeacher")]
        public ActionResult Post(AddTeacherRequest teacherModel)
        {
            if (teacherModel == null)
            {
```

```
            return BadRequest();
        }
        var news = _teacherService.CreateTeacher(teacherModel).Result;
        _teacherRepository.Create(news);
        return Ok(news);
    }

    /// <summary>
    /// Update specific teacher from db
    /// </summary>
    /// <param name="teacher">Updated teacher</param>
    /// <returns>Updated teacher</returns>
    [HttpPut("EditedTeacher")]
    public ActionResult Put(UpdateTeacherRequest teacherModel)
    {
        var teacherToUpdate = _teacherRepository.GetById(teacherModel.Id);
        if (teacherModel == null)
        {
            return BadRequest();
        }
        if (teacherToUpdate == null)
        {
            return NotFound();
        }
        var teacher = _teacherService.CreateTeacher(teacherModel).Result;
        _teacherRepository.Update(teacher, teacherModel.DeletePicture,
teacherToUpdate.ImageName);
        return Ok(teacher);
    }

    /// <summary>
    /// Deletes teacher with specific id from db
    /// </summary>
    /// <param name="id">Id of teacher that you want to delete</param>
    /// <returns>Id of deleted teacher</returns>
```

```
        [HttpDelete("DeleteTeacherId")]
        public ActionResult Delete(int id)
        {
            if (!_teacherRepository.GetList().Any(n => n.Id == id))
            {
                return NotFound();
            }
            _teacherRepository.Delete(id);
            return Ok(id);
        }
    }
}
```

ДОДАТОК Л

Лістинг інтерфейсу NewsService

```
using Microsoft.Extensions.Configuration;
using ot_api.Entities;
using ot_api.Models;
using ot_api.Models.Interfaces;
using System.Threading.Tasks;

namespace ot_api.Services
{
    public class NewsService
    {
        private ImagesService _imagesService;

        public NewsService(IConfiguration config)
        {
            _imagesService = new ImagesService(config);
        }

        public async Task<News> CreateNewsAsync(INewsRequestModel newsEditingModel)
        {
            var news = new News();

            if (newsEditingModel is UpdateNewsRequest newsWithId)
            {
                news.Id = newsWithId.Id;
            }

            news.Title = newsEditingModel.Title;
            news.Description = newsEditingModel.Description;
            news.Content = newsEditingModel.Content;

            if (newsEditingModel.ImageFile != null)
            {
```

```
            var imageInfo = await
_imagesService.SaveImagesAsync(newsEditingModel.ImageFile);
            news.ImageStorageUrl = imageInfo.FileLink;
            news.ImageName = imageInfo.FileName;
        }

        return news;
    }

    public void UpdateNews(News updatedNews, News news)
    {
        news.Title = updatedNews.Title;
        news.Description = updatedNews.Description;
        news.Content = updatedNews.Content;

        if (updatedNews.ImageStorageUrl != null)
        {
            news.ImageStorageUrl = updatedNews.ImageStorageUrl;
            news.ImageName = updatedNews.ImageName;
        }

    }

    public async Task DeleteNewsImageAsync(string imageName)
    {
        await _imagesService.DeleteImageAsync(imageName);
    }

    public NewsResponse CreateNewsViewModel(News news)
    {
        var newsViewModel = new NewsResponse();

        newsViewModel.Id = news.Id;
        newsViewModel.Title = news.Title;
        newsViewModel.Description = news.Description;
```

```
            newsViewModel.Content = news.Content;
            newsViewModel.DateTime = news.DateTime;
            newsViewModel.ImageStorageUrl = news.ImageStorageUrl;

            return newsViewModel;
        }
    }
}
```

ДОДАТОК М

Лістинг інтерфейсу TeacherService

```
using Microsoft.Extensions.Configuration;
using ot_api.Entities;
using ot_api.Models;
using ot_api.Models.Interfaces;
using System.Linq;
using System.Threading.Tasks;

namespace ot_api.Services
{
    public class TeacherService
    {
        private ImagesService _imagesService;
        public TeacherService(IConfiguration config)
        {
            _imagesService = new ImagesService(config);
        }
        public async Task<Teacher> CreateTeacher(ITeacherRequestModel teacherEditingModel)
        {
            var teacher = new Teacher();
            if (teacherEditingModel is UpdateTeacherRequest teacherWithId)
            {
                teacher.Id = teacherWithId.Id;
            }

            teacher.FirstName = teacherEditingModel.FirstName;
            teacher.SecondName = teacherEditingModel.SecondName;
            teacher.MiddleName = teacherEditingModel.MiddleName;
            teacher.Biography = teacherEditingModel.Biography;
            teacher.Disciplines = teacherEditingModel.Disciplines;
            teacher.Position = teacherEditingModel.Position;
```

```
            for (int i = 0; i < teacherEditingModel.Merits.Count; i++)
            {
                teacher.Merits += teacherEditingModel.Merits[i];
                if (i <teacherEditingModel.Merits.Count -1)
                {
                    teacher.Merits += ";";
                }
            }
            for (int i = 0; i < teacherEditingModel.Awards.Count; i++)
            {
                teacher.Awards += teacherEditingModel.Awards[i];
                if (i < teacherEditingModel.Awards.Count - 1)
                {
                    teacher.Awards += ";";
                }
            }
            if (teacherEditingModel.ImageFile != null)
            {
                var imageInfo = await
_imagesService.SaveImagesAsync(teacherEditingModel.ImageFile);
                teacher.ImageStorageUrl = imageInfo.FileLink;
                teacher.ImageName = imageInfo.FileName;
            }
            return teacher;
        }
        public void UpdateTeacher(Teacher updatedTeacer, Teacher teacher)
        {
            teacher.FirstName = updatedTeacer.FirstName;
            teacher.SecondName = updatedTeacer.SecondName;
            teacher.MiddleName = updatedTeacer.MiddleName;
            teacher.Position = updatedTeacer.Position;
            teacher.Merits = updatedTeacer.Merits;
            teacher.Awards = updatedTeacer.Awards;
            teacher.Biography = updatedTeacer.Biography;
            teacher.Disciplines = updatedTeacer.Disciplines;
```

```
        if (updatedTeacer.ImageStorageUrl != null)
        {
            teacher.ImageStorageUrl = updatedTeacer.ImageStorageUrl;
            teacher.ImageName = updatedTeacer.ImageName;
        }
    }
    public async Task DeleteTeacherImageAsync(string imageName)
    {
        await _imagesService.DeleteImageAsync(imageName);
    }
    public TeacherResponse CreateTeacherViewModel(Teacher teacher)
    {
        var teacherViewModel = new TeacherResponse();
        teacherViewModel.Id = teacher.Id;
        teacherViewModel.FirstName = teacher.FirstName;
        teacherViewModel.SecondName = teacher.SecondName;
        teacherViewModel.MiddleName = teacher.MiddleName;
        teacherViewModel.Biography = teacher.Biography;
        teacherViewModel.Disciplines = teacher.Disciplines;
        teacherViewModel.Position = teacher.Position;
        teacherViewModel.ImageStorageUrl = teacher.ImageStorageUrl;
        if (teacher.Merits != null)
        {
            teacherViewModel.Merits = teacher.Merits.Split(';').ToList();
        }
        if (teacher.Awards != null)
        {
            teacherViewModel.Awards = teacher.Awards.Split(';').ToList();
        }

        return teacherViewModel;
    }
}}
```

ДОДАТОК Н
ПРОТОКОЛ
ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА НАЯВНІСТЬ ТЕКСТОВИХ
ЗАПОЗИЧЕНЬ

Назва роботи: Клієнт-серверна інформаційна система підрозділу навчального закладу з можливістю розгортання в хмарному середовищі. Частина 2. «Розробка розподіленого слабкозв'язаного серверного додатку мовою C#

Тип роботи: бакалаврська дипломна робота
(БДР, МКР)

Підрозділ кафедра обчислювальної техніки
(кафедра, факультет)

Показники звіту подібності
Unicheck

Оригінальність 97% Схожість 3%

Аналіз звіту подібності (відмітити потрібне):

- ✓ Запозичення, виявлення у роботі, оформлені коректно і не містять ознак плагіату.
- □ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.
- □ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку ___________ Захарченко С.М.
(підпис) (прізвище, ініціали)

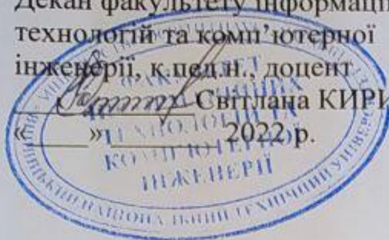
Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

Автор роботи ___________ Рижков А.К.
(підпис) (прізвище, ініціали)

Керівник роботи ___________ Войцеховська О. В.
(підпис) (прізвище, ініціали)

ДОДАТОК Р
АКТ ВПРОВАДЖЕННЯ

УЗГОДЖЕНО
Декан факультету інформаційних технологій та комп'ютерної інженерії, к.пед.н., доцент
_________ Світлана КИРИЛАЩУК
«____»________ 2022 р.



ЗАТВЕРДЖУЮ
Проректор з науково-педагогічної роботи та організації освітнього процесу ВНТУ, к.т.н., доцент
_________ Олександр ПЕТРОВ
«____»________ 2022 р.

АКТ ВПРОВАДЖЕННЯ № ____

результатів комплексної бакалаврської дипломної роботи

Замовник ___Вінницький національний технічний університет________________
(найменування організації)

Цим актом підтверджується, що результати роботи – «Клієнт-серверна інформаційна система підрозділу навчального закладу з можливістю розгортання в хмарному середовищі. Частина 2. «Розробка розподіленого слабкозв'язаного серверного додатку мовою C#»,
що виконана студентом гр. 1КІ – 20мс, ВНТУ, Рижковим А. К.
(виконавець)
впроваджено у Вінницькому національному технічному університеті___________
(найменування організації, де здійснювалося впровадження)

1.Вид впроваджених результатів сайт кафедри обчислювальної техніки ВНТУ_____
(експлуатація виробу, роботи, технології)

2. Характеристика масштабу впровадження _____одиничне________________
(унікальне, одиничне, партія, масове, серійне)

3. Форма впровадження ________програмний продукт__________________

4. Новизна результатів роботи модернізація старих розробок_
(піонерські, принципово нові, якісно нові, модифікації, модернізація старих розробок)

5. Впроваджені: _________в мережі ВНТУ___________

6. Річний економічний ефект____-_____________________________________

Від виконавця:
Студент групи 1КІ-20мс

____________Андрій РИЖКОВ

Від організації:
Завідувач кафедри обчислювальної техніки ВНТУ, д.т.н., професор

____________Олексій АЗАРОВ

Науковий керівник
________доц. Олена ВОЙЦЕХОВСЬКА