МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

Навчально-науковий Інститут фізико-технічних та комп’ютерних наук

Відділ комп’ютерних технологій

Кафедра комп’ютерних наук

**Обчислювальна практика**

**з дисципліни «Об’єктно-орієнтоване програмування»**

**на тему «Платформа для оренди житла»**

Студента 2 курсу, групи 244

напряму підготовки «Комп’ютерні науки»

\_\_\_\_\_\_\_\_\_\_\_\_Кульчицького А. С.\_\_\_\_\_\_\_

(ініціали студента)

Керівник      к.ф.-м.н. Томка Ю. Я.

 (посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна шкала:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_Оцінка: ECTS \_\_\_

Члени комісії:

\_\_\_\_\_\_\_\_\_\_\_\_ Ушенко Ю. О.

(підпис) (прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_ Томка Ю. Я.

(підпис) (прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_ Ковальчук М. Л.

(підпис) (прізвище та ініціали)

Чернівці – 2023

**Чернівецький національний університет імені Юрія Федьковича**

(назва вузу)

|  |  |
| --- | --- |
| Навчально-Науковий Інститут фізико-технічних та комп’ютерних наук  Кафедра комп’ютерних наук  **ЗАТВЕРДЖУЮ**  Зав. кафедрою \_\_\_\_\_\_\_\_\_\_\_\_ Ушенко Ю.О.  “\_\_ ” \_\_\_\_\_\_ 2023 р. | |
| **ІНДИВІДУАЛЬНЕ ЗАВДАННЯ** | |
| **на обчислювальну практику** | |
| Студента \_Кульчицького Андрія Сергійовича\_групи\_244\_другого курсу | |
| (прізвище, ім‘я, по батькові) | |
| **1. Тема роботи: «Платформа для оренди житла»** | |
| **Керівник роботи** к.ф.-м.н. Томка Юрій Ярославович                                                                                         (прізвище, ім’я, по батькові, науковий ступінь, вчене звання) | |
| **2. Строк подання** студентом проекту: **30.06.2023** | |
| **3. Вихідні дані до роботи:** | літературні джерела |
| об’єктно-орієнтовані підходи до розробки | |
| програмного забезпечення із використанням мови C# у програмному середовищі Microsoft Visual Studio 2022 | |
| **4.  Зміст розрахунково-пояснювальної записки** (перелік питань, що їх належить розробити): | |
| Опис предметної області | |
| Аналіз аналогів програмного забезпечення | |
| Функціональні вимоги до додатку | |
| Модульно-аналітична схема додатку | |
| Проектування бази даних | |
| Вибір архітектури додатку | |
| Вибір стеку технологій | |
| Реалізація програмного продукту | |
| Розробка графічного інтерфейсу користувача | |
| Висновки | |
| **5.  Перелік наочного матеріалу** (з точним зазначенням обов’язкових креслень, плакатів): | |
| * графічна складова користувацького інтерфейсу програмного забезпечення; | |
| * код програми; | |
| * пояснювальна записка, презентація. | |
| Дата видачі завдання   **19.06.2023** | |

|  |  |  |  |
| --- | --- | --- | --- |
| **КАЛЕНДАРНИЙ ПЛАН** | | | |
| № з/п | Назва етапів обчислювальної практики | Строк виконання етапів проекту | Примітка |
| 1 | Отримання завдання на обчислювальну практику | 19.06.2023 |  |
| 2 | Огляд джерел технічної інформації за тематикою | 19.06.2023 |  |
| 3 | Виконання аналізу методів реалізації завдання | 20.06.2023 |  |
| 4 | Аналіз шляхів рішення поставленої задачі | 20.06.2023 |  |
| 5 | Програмна розробка додатку. Розробка Data Access Layer | 21.06.2023 |  |
| 6 | Програмна розробка додатку. Розробка Business Logic Layer | 22.06.2023 |  |
| 7 | Програмна розробка додатку. Розробка Views | 24.06.2023 |  |
| 8 | Програмна розробка додатку. Розробка ViewsModels | 27.06.2023 |  |
| 9 | Тестування розробленого програмного продукту | 28.06.2023 |  |
| 10 | Корегування роботи за результатами розгляду керівника. Написання пояснювальної записки. | 29.06.2023 |  |
| 11 | Розробка доповіді та презентації | 29.06.2023 |  |
| 12 | Захист роботи з обчислювальної практики | 30.06.2023 |  |
|  |  |  |  |
| Студент   Кульчицький А. С.  / П.І.Б. / Науковий керівник  Томка Ю. Я.  / П.І.Б. / | | | |
| (підпис)                                                            (підпис) | | | |
| Завдання видане    «**19**» **червня 2023р.** | | | |

# АНОТАЦІЯ

Десктоп-додатки забезпечує ефективне та зручне керування готельними операціями та покращення взаємодії з клієнтами. Десктоп-додатки дозволяють готелям автоматизувати багато процесів, таких як управління бронюваннями, розподілом номерів, облік фінансів та інші аспекти їх діяльності.

Задача полягає у тому щоб створити додаток для Windows у якому будуть оголошення про оренду житла чи номерів готелів. Потрібно реалізувати можливість створення, видалення та зміни бронювання житла.

Це все буде реалізовано з архітектурним патерном MVVM та за допомогою графічної підсистеми WPF.

**Ключові слова:** ДЕКСТОП-ДОДАТОК, АВТОМАТИЗАЦІЯ ПРОЦЕСІВ,

АРХІТЕКТУРНИЙ ПАТЕРН MVVM, ГРАФІЧНА ПІДСИСТЕМА WPF.

# ЗМІСТ

[ВСТУП 4](#_heading=h.3znysh7)

[РОЗДІЛ 1. ПРЕДМЕТНА ОБЛАСТЬ 5](#_heading=h.2et92p0)

[1.1 Формування проблематики. Опис предметної області 5](#_heading=h.tyjcwt)

[1.2 Аналіз аналогів програмного забезпечення 6](#_heading=h.3dy6vkm)

[1.2.1 Booking 6](#_heading=h.1t3h5sf)

[1.2.2 Airbnb 8](#_heading=h.4d34og8)

[1.2.3 Підсумки аналізу 9](#_heading=h.2s8eyo1)

[1.3 Функціональні вимоги до проекту. User Stories 9](#_heading=h.17dp8vu)

[1.3.1 User Stories Аноніма 10](#_heading=h.3rdcrjn)

[1.3.2 User Stories Читача 11](#_heading=h.26in1rg)

[1.3.3 User Stories Автора 12](#_heading=h.lnxbz9)

[1.3.4 User Stories Адміністратора 13](#_heading=h.35nkun2)

[1.4 Модульно-аналітична схема додатку, що розробляється 13](#_heading=h.1ksv4uv)

[1.5 Проектування бази даних 15](#_heading=h.44sinio)

[1.6 Вибір архітектури додатку 17](#_heading=h.2jxsxqh)

[ВИСНОВКИ ДО РОЗДІЛУ 20](#_heading=h.z337ya)

[РОЗДІЛ 2. BACK-END РОЗРОБКА 21](#_heading=h.3j2qqm3)

[2.1 Вибір стеку технологій 21](#_heading=h.1y810tw)

[2.2 Реалізація з використанням ADO.NET та Dapper 22](#_heading=h.4i7ojhp)

[2.2.1 Збережувані процедури 23](#_heading=h.2xcytpi)

[2.2.2 Підключення до бази даних 23](#_heading=h.1ci93xb)

[2.2.3 Generic Repository. Порівняння запитів на ADO.NET і Dapper 24](#_heading=h.3whwml4)

[2.2.4 Контролери 27](#_heading=h.2bn6wsx)

[2.3 Основна реалізація з використанням Entity Framework Core 28](#_heading=h.qsh70q)

[2.3.1 Code-First підхід до створення бази даних 28](#_heading=h.3as4poj)

[2.3.2 Конфігурація сутностей з використанням Fluent API 29](#_heading=h.1pxezwc)

[2.3.3 Generic Repository. Unit of Work 31](#_heading=h.49x2ik5)

[2.3.4 ASP.NET Core Identity 32](#_heading=h.2p2csry)

[2.3.5 Міграції. Зміни структури бази даних у процесі розробки 32](#_heading=h.147n2zr)

[2.3.6 Бізнес-логіка. DTO. AutoMapper 34](#_heading=h.3o7alnk)

[2.3.7 Авторизація на основі JWT-токенів 36](#_heading=h.23ckvvd)

[2.3.8 Пагінація, фільтрація, пошук даних 37](#_heading=h.ihv636)

[2.3.9 Контролери. Завантаження файлів. Swagger 39](#_heading=h.32hioqz)

[2.3.10 Валідація. Локалізація 42](#_heading=h.1hmsyys)

[2.4 Порівняння ADO.NET (Dapper) та Entity Framework Core 44](#_heading=h.41mghml)

[ВИСНОВКИ ДО РОЗДІЛУ 46](#_heading=h.2grqrue)

[РОЗДІЛ 3. FRONT-END РОЗРОБКА 47](#_heading=h.vx1227)

[3.1 Вибір стеку технологій 47](#_heading=h.3fwokq0)

[3.2 Робота із компонентами 48](#_heading=h.1v1yuxt)

[3.2.1 Компоненти MudBlazor 48](#_heading=h.4f1mdlm)

[3.2.2 Користувацькі компоненти 49](#_heading=h.2u6wntf)

[3.3 Сторінки 53](#_heading=h.19c6y18)

[3.4 Формування запитів до back-end 55](#_heading=h.3tbugp1)

[3.5 Авторизація 57](#_heading=h.28h4qwu)

[3.5.1 Формування авторизаційного запиту до back-end 57](#_heading=h.nmf14n)

[3.5.2 Робота з JWT-токеном 57](#_heading=h.37m2jsg)

[3.5.3 Надсилання запиту із JWT-токеном 58](#_heading=h.1mrcu09)

[ВИСНОВКИ ДО РОЗДІЛУ 59](#_heading=h.46r0co2)

[ВИСНОВКИ 60](#_heading=h.2lwamvv)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 61](#_heading=h.111kx3o)

[ДОДАТКИ 63](#_heading=h.3l18frh)

[Додаток А1. Запит на створення бази даних MS SQL Server 63](#_heading=h.206ipza)

[Додаток А2. Збережені процедури 65](#_heading=h.4k668n3)

[Додаток Б1. Контекст даних 67](#_heading=h.2zbgiuw)

[Додаток Б2. Generic Repository 69](#_heading=h.1egqt2p)

[Додаток Б3. Unit of Work 70](#_heading=h.3ygebqi)

[Додаток Б4. Профіль AutoMapper 71](#_heading=h.2dlolyb)

[Додаток Б5. Сервіс Identity 72](#_heading=h.sqyw64)

[Додаток В1. Графічні компоненти 74](#_heading=h.3cqmetx)

[Додаток В2. Головна сторінка графічного інтерфейсу 74](#_heading=h.1rvwp1q)

[Додаток В3. Допоміжний клієнт HTTP 77](#_heading=h.4bvk7pj)

[Додаток В4. JwtAuthenticationStateProvider 78](#_heading=h.2r0uhxc)

# ВСТУП

В сучасному світі існує велика потреба у платформах, які дозволяють різним готелям та власникам житла розміщувати свої об'яви і надавати можливість орендувати їх користувачам. Це дозволяє забезпечити зручну інтерактивну платформу для бронювання житла на певну дату, на заданий період часу та вибір категорії.

На платформі можна знайти об'яви про різноманітне житло від різних готелів та власників. Користувачі можуть шукати житло за певною категорією, вибирати дату, яка їм підходить, і орендувати його на зазначений період часу. Кожна об'ява має заголовок, опис, фотографії та інші деталі, які допомагають користувачеві зробити відповідне рішення.

Темою роботи є платформа для розміщення пропозицій оренди житла та його пошуку.

Метою даного проекту є розробка інформаційної системи платформи для розміщення об'яв про готелі та житло, доступного для оренди. Основним завданням є створення зручного сервісу, де готелі та власники житла зможуть розміщувати свої об'яви, а користувачі зможуть орендувати житло на певну дату та тривалість, вибираючи серед різних категорій.

**Метою** роботиє проектування і розробка платформи для оренди житла.

Для досягнення мети поставлено такі **завдання:**

* проаналізувати предметну область та аналоги ПЗ;
* сформулювати функціональні вимоги до системи;
* продумати структуру проекту відповідно до вимог;
* реалізувати back-end додатку на платформі .NET;
* реалізувати front-end з використання технології WPF.

**Структура та обсяг роботи.**  Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел (24 найменування), 11 додатків. Загальний обсяг роботи становить 78 сторінок, серед них 36 рисунків.

# РОЗДІЛ 1. ПРЕДМЕТНА ОБЛАСТЬ

## Формування проблематики. Опис предметної області

Інтернет відіграє важливу роль у готельному бізнесі, забезпечуючи широкі можливості для реклами, взаємодії з клієнтами та надання послуг. Платформа для оренди житла створює зручні умови для клієнтів, які шукають житло на певну дату та тривалість. Це дає їм можливість вибирати серед різноманітних пропозицій, які орендодавці розміщують на платформі.

Платформа дозволяє клієнтам швидко та зручно знайти і орендувати житло, а також обирати за категоріями, такими як тип житла, розташування, ціновий діапазон та інші параметри. Це забезпечує комфортний процес вибору для клієнтів, оскільки всі пропозиції зібрані в одному місці і можуть бути легко сортовані та фільтровані.

Одним із ключових аспектів платформи для оренди житла є забезпечення надійності та безпеки для клієнтів. Шахраї та недостовірні пропозиції можуть становити загрозу, тому важливо мати перевірені джерела інформації та механізми перевірки орендодавців. Надійність платформи та її репутація є критичними факторами для залучення клієнтів.

Крім того, платформа для оренди житла може забезпечувати додаткові функції, такі як онлайн-оплата, відгуки та рейтинги, контакт з орендодавцем, картографічне представлення об'єктів житла тощо. Це полегшує процес оренди та покращує взаємодію між клієнтами та орендодавцями.

Загалом, платформа для оренди житла сприяє ефективному використанню ресурсів, забезпечує зручність та широкі можливості для користувачів. Вона створює вигоду як для клієнтів, які шукають житло, так і для орендодавців, які пропонують свої об'єкти оренди.

Для цієї предметної області потрібно реалізувати::

1. Оголошення, їх створення та редагування орендодавцями.

2. Реалізація відгуків на орендодавців та системи рейтингу

3. Пошук з фільтрами оголошень за критеріями,.

4. Система реєстрації, автентифікації та авторизації користувачів.

5. Профіль користувача

6. Зворотній звязок

## Аналіз аналогів програмного забезпечення

Аналіз аналогів ПЗ дасть змогу виділити сильні сторони, які можна перейняти, та з’ясувати слабкі сторони які не потрібно реалізовувати у своєму додатку. Як аналоги є такі платформи як Booking та Airbnb.

### Booking

Booking[1] — система інтернет-бронювання житла (готелі, хостели), заснована в Амстердамі 1996 року, придбана 2005 року за $133 млн компанією Priceline (NASDAQ: PCLN). Штаб-квартира компанії знаходиться в Амстердамі, Нідерланди. Штат працівників складає понад 15 000 осіб у 198 офісах та 70 країнах.

Сильною стороною Booking.com є великий асортимент готелів, апартаментів, гостьових будинків та інших типів помешкань по всьому світу. Платформа зібрала велику кількість пропозиції, та створила систему що дозволяє орендодавцям додавати свої пропозиції та користувачам знаходити різноманітне житло.

Також розробники подбали про користувацький досвід[2], Booking.com ставить на перше місце зручність користувача. Інтуїтивно простий інтерфейс, зручний пошук житла, зображення високої якості та детальний опис про кожну пропозицію - ці важливо врахувати при розробці декстоп-додатку, для комфорту користувачів.

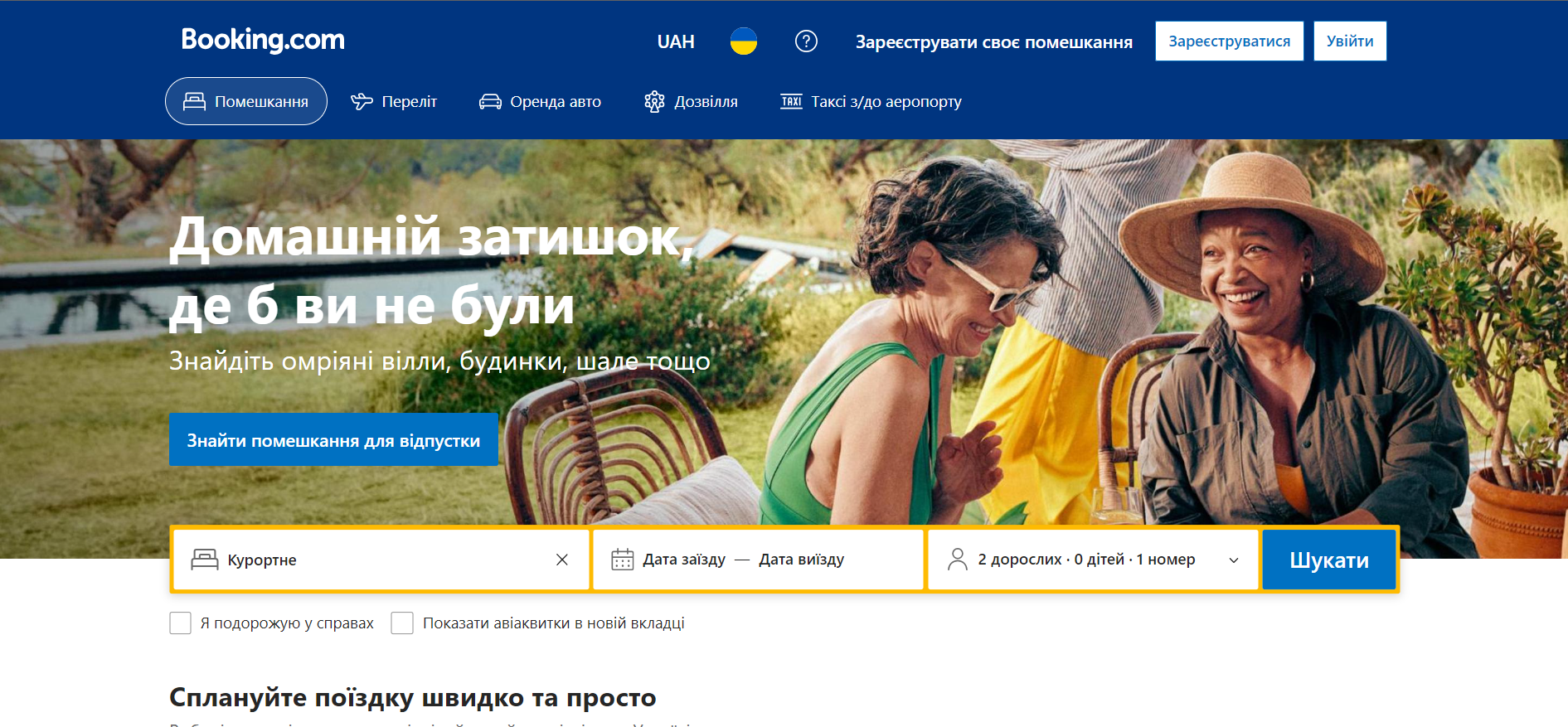


Рисунок 1.1 – Користувацький інтерфейс платформи Booking

### Airbnb

Ейрбіенбі (англ. Airbnb) — онлайн-сервіс з розміщення, пошуку та короткострокової оренди житла по всьому світі, що працює за парадигмою економіки спільної участі. Назва розшифровується як «AirBed and Breakfast» — «Надувний матрац та сніданок». Сервіс допомагає винайняти будинки, квартири, кімнати або ліжка у більш ніж 190 країнах світу.

З сильних сторін можна виділити систему відгуків та рейтингу. Платформа Airbnb надає можливість користувачам залишати відгуки про житло та господарів. Це допомагає обрати житло спираючись на досвід попередніх клієнтів.

Чудовом плюсом є управління бронюванням. Платформа довзоляє не лише створювати але й редагувати бронювання: змінювати дату, скасовувати або змінювати резервацію.

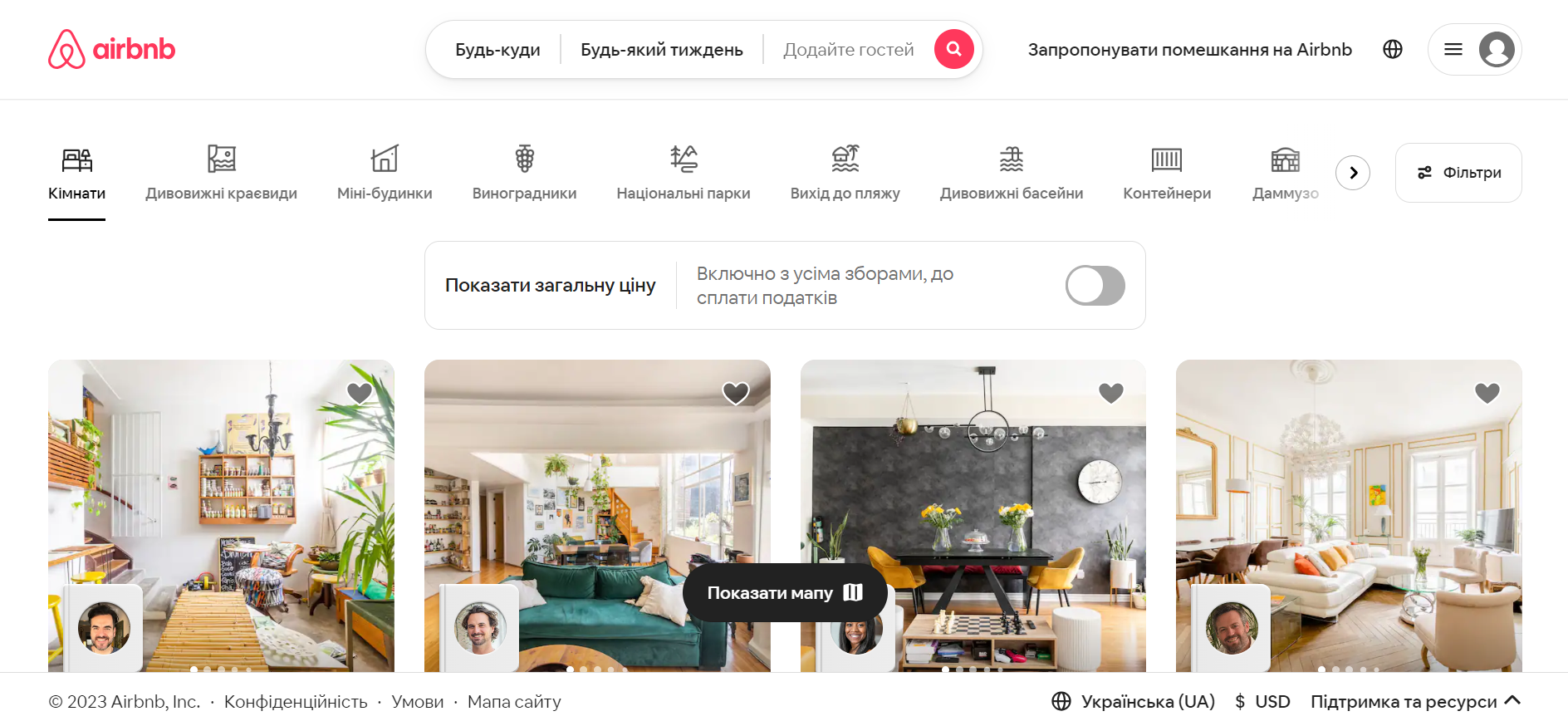


Рисунок 1.2 – Головна сторінка платформи Airbnb

### Підсумки аналізу

Аналізуючи подібні проекти можна виділити їхні сильні сторони, та

## Функціональні вимоги до проекту. User Stories

User Stories (історії користувача) [4] – це спосіб представлення функціональних вимог до програмного забезпечення на етапі проектування; набір коротких речень простою мовою предметної області, які чітко описують потреби користувача у функціоналі. User Stories широко застосовуються в [гнучких методологіях](https://uk.wikipedia.org/wiki/%D0%93%D0%BD%D1%83%D1%87%D0%BA%D0%B0_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D1%96%D1%8F) розробки ПЗ. Ціль окремої історії користувача – з’ясувати, *хто* має функціональну вимогу, *що* вона собою являє та *чому* її необхідно реалізувати. Стандартний вигляд User Stories є таким: «Як *певний тип користувача*, я хочу *мати певну можливість*, щоб *отримати певну вигоду*».

У рамках даної предметної області виділено чотири основні типи (ролі) користувачів: Анонім, Читач, Автор, Адміністратор. Кожен наступний тип наділений всім функціоналом попереднього та новим. Анонім – неавторизований відвідувач ресурсу, що має права перегляду. Читач, окрім цього, має власний профіль, може коментувати та оцінювати публікації. Автор публікує пости та може редагувати власні. Адміністратор має повні права доступу.

Далі наведено перелік всіх User Stories, сформованих для розробки проекту. Деякі з них було відібрано для подальшого розвитку; такі історії позначено «НР» (не реалізовано).

### User Stories Аноніма

1. Як анонім, я хочу переглядати всі публікації від найновішої до найстарішої, щоб отримати необхідну мені інформацію.

2. Як анонім, я хочу мати посторінковий перегляд публікацій, щоб заощаджувати ресурси мого комп’ютера при завантаженні вебсайту.

3. Як анонім, я хочу проводити пошук серед публікацій по назві, щоб швидко знаходити потрібну інформацію.

4. Як анонім, я хочу бачити перелік найбільш популярних публікацій, щоб знати, на що варто звернути увагу.

5. Як анонім, я хочу бачити перелік категорій, щоб мати уявлення про наявність необхідного мені контенту.

6. Як анонім, я хочу фільтрувати публікації по категоріях, щоб пришвидшити пошук необхідної публікації.

7. Як анонім, я хочу бачити картки з достатньою інформацією про публікації, щоб орієнтуватися в ній без перечитування вмісту. Зокрема, повинні відображатися заголовок, автор, час публікування, зображення заставки.

8. Як анонім, я хочу бачити час публікування у зручному форматі, щоб швидко розуміти актуальність матеріалу. Наприклад, «щойно», «16 днів тому».

9. Як анонім, я хочу мати довідкову інформацію про вебсайт, щоб швидко та доступно ознайомитися з ним. (НР)

10. Як анонім, я хочу отримати контакти власника вебсайту, щоб могти зв’язатися з ним зі скаргами та пропозиціями. Бажано мати вбудовану форму зворотного зв’язку. (НР)

11. Як анонім, я хочу обирати мою мову зі списку, щоб вільно розуміти вміст вебресурсу. При цьому повинно бути перекладено все, окрім вмісту користувачів (публікацій, коментарів, даних профілю).

12. Як анонім, я хочу реєструватися в системі, щоб створити власний обліковий запис із додатковими привілеями.

13. Як анонім, я хочу входити в систему зі своїми даними, щоб отримувати доступ до власного облікового запису.

14. Як анонім, я бажаю доступних та чітких повідомлень про неправильно введені дані, щоб швидко розуміти, в чому я допустив помилку.

15. Як анонім, я хочу бачити в публікації не тільки суцільний текст, а й форматування, зображення, медіа. Це допоможе більш наочно донести мені інформацію.

16. Як анонім, я хочу бачити середню користувацьку оцінку публікації, щоб орієнтуватися в її релевантності та змістовності.

17. Як анонім, я хочу читати коментарі інших користувачів, щоб отримувати різні думки щодо прочитаної публікації.

18. Як анонім, я хочу бачити список категорій, до яких належить конкретна публікація, щоб швидко складати враження про її вміст.

19. Як анонім, я хочу бачити історію змін публікації, щоб бути впевненим, що я не пропустив важливої нової інформації. (НР)

20. Як анонім, я хочу бачити перелік найкращих авторів, щоб могти переглянути їх роботи. Для цього також повинен бути пошук всіх публікацій користувача за його іменем. (НР)

21. Як анонім, я хочу мати доступ до гнучкого пошуку публікацій по даті, щоб швидше знаходити потрібну інформацію. Так, бажано шукати по конкретній даті, по певному місяцю, року, по часовому проміжку. (НР)

### User Stories Читача

1. Як читач, я хочу мати доступ до налаштувань власного профілю, де я міг би редагувати особисті дані та вподобання, щоб мати контроль над моєю взаємодією з додатком. (НР)

2. Як читач, я хочу зберігати публікації в закладки, щоб швидко отримувати доступ до обраних публікацій. (НР)

3. Як читач, я хочу переглядати профілі інших користувачів і бачити їх улюблені пости, щоб складати думку про певного користувача та ознайомлюватися з його інтересами. (НР)

4. Як читач, я хочу залишати коментарі до публікацій, щоб висловлювати свою думку щодо них.

5. Як читач, я хочу відповідати на коментарі, щоб вести спілкування із конкретним користувачем. При цьому він повинен бути повідомлений про мою відповідь. (НР)

6. Як читач, я хочу відзначати коментарі, які мені сподобалися, щоб висловлювати підтримку їх авторам. (НР)

7. Як читач, я хочу оцінювати публікації, щоб брати участь у їх просуванні.

8. Як читач, я хочу бачити власну історію переглядів публікацій, щоб повертатися до переглянутої інформації в разі потреби. (НР)

9. Як читач, я хочу отримувати необхідні оновлення у вигляді електронних листів, щоб слідкувати за виходом цікавих мені публікацій. (НР)

10. Як читач, я хочу подавати заявку на роль автора, щоб взяти безпосередню участь у житті ресурсу. (НР)

### User Stories Автора

1. Як автор, я хочу мати перелік власних публікацій з інструментами для роботи з ними, щоб зручно проводити свою діяльність на ресурсі.

2. Як автор, я хочу створювати публікацію на єдиній формі, включно з додаванням категорій та зображення, щоб зробити швидшим процес публікування.

3. Як автор, я хочу редагувати публікацію, зокрема замінювати зображення заставки, щоб могти виправляти свої помилки.

4. Як автор, я хочу мати попередній перегляд створюваної або редагованої публікації, щоб мати уявлення про зовнішній вигляд результату роботи. (НР)

5. Як автор, я повинен мати посібник по розмітці в публікації, щоб навчитися грамотно організовувати її вміст. (НР)

6. Як автор, я хочу відображати власні публікації в моєму профілі, щоб інші користувачі могли мати легкий доступ до них. (НР)

7. Як автор, я хочу мати загальний рейтинг на вебсайті, базований на оцінках моїх публікацій, щоб давати уявлення іншим про мою діяльність. (НР)

### User Stories Адміністратора

1. Як адміністратор, я хочу оперативно видаляти будь-який коментар, щоб уникнути неприємних ситуацій.

2. Як адміністратор, я хочу керувати користувачами, зокрема призупиняти дію облікових записів, щоб обмежити їх у разі необхідності. (НР)

3. Як адміністратор, я хочу робити публікації прихованими, щоб їх автори могли виправити критичні проблеми. (НР)

4. Як адміністратор, я хочу мати перелік вхідних заявок на роль автора, щоб зручно і оперативно на них відповідати. (НР)

5. Як адміністратор, я хочу мати перелік потенційно небажаних коментарів, щоб оперативно реагувати на порушення. (НР)

6. Як адміністратор, я хочу отримувати статистику відвідуваності сайту, щоб робити судження про аудиторію та бізнес-рішення. (НР)

## Модульно-аналітична схема додатку, що розробляється

Додаток можна розділити на такі логічні модулі (рис. 1.3):

1. Робота з обліковими записами (User Account Module):

* організація створення облікових записів (реєстрація);
* зберігання даних про облікові записи;
* автентифікація користувача в системі;
* призначення користувачеві ролі, блокування запису;
* операції з особистими даними (редагування, зміна пароля).

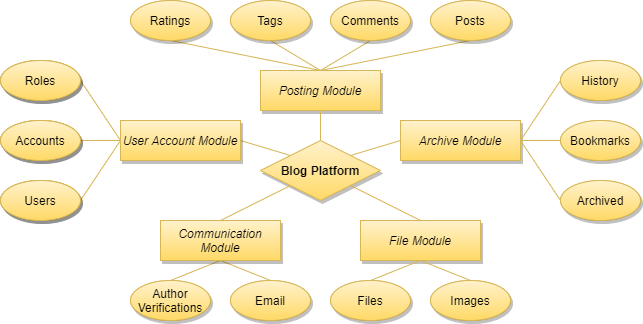


Рисунок 1.3 – Модульно-аналітична схема додатку

1. Робота з публікаціями (Posting Module):

* зберігання даних про публікації, коментарі, оцінки, категорії;
* CRUD-операції з цими даними;
* реалізація необхідної бізнес-логіки;
* пагінація, фільтрування, пошук, сортування даних.

1. Зв’язок (Communication Module):

* надсилання електронної пошти (email-листів);
* обробка заявок на роль автора;
* зберігання історії підтвердження авторів.

1. Збереження файлів (File Module):

* локальне зберігання користувацьких зображень, інших файлів;
* отримання файлу за його посиланням;
* публікування користувацьких файлів.

1. Архів (Archive Module):

* зберігання історії переглядів користувачів;
* зберігання закладок (улюблених публікацій);
* розміщення архівованих публікацій;
* зберігання інформації про відвідування ресурсу;
* зберігання історії змін публікацій.

## Проектування бази даних

Для реалізації сутностей предметної області обрано реляційну СКБД Microsoft SQL Server. Перша версія бази даних, отримана в процесі розробки, була сформована Transact-SQL (T-SQL) запитом. Синтаксичні конструкції для вивчення та повторення взято з довідника [5]. Уривок із запиту показано на рис. 1.4. Даний фрагмент коду створює таблицю Comments з відповідними полями, ключами та обмеженнями.

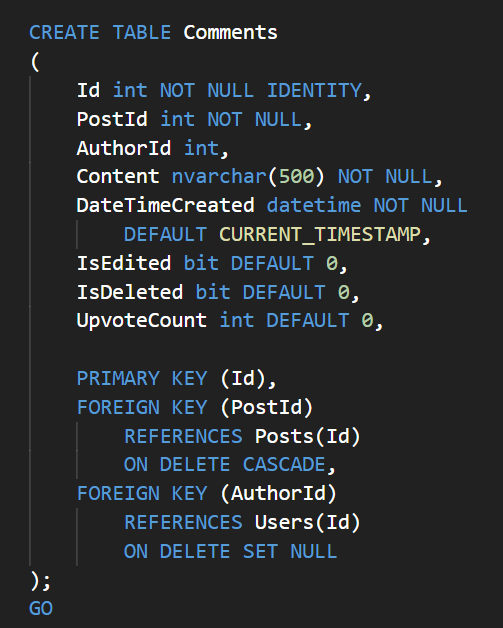


Рисунок 1.4 – Запит на створення таблиці Comments

Повний запит зі структурою таблиць та зв’язків наведено в додатку А1. Результатом виконання запиту є повністю функціональна база даних (рис. 1.5). Варто розтлумачити призначення кожної структурної одиниці:

1. Users (Користувачі) – тимчасова таблиця, яка створювалася для цілісності структури бази даних. Надалі замінена вбудованим механізмом ASP.NET Core Identity (наведено в підрозділі 2.3). Містила ім’я користувача (Username) та логічне значення, чи є користувач автором блогу (IsBlogAuthor).

2. Posts (Публікації) – основна сутність предметної області. Вона містить записи про опубліковані пости, а саме заголовок (Title), шлях до зображення заставки (ThumbnailPath), дату створення (DateTimePublished). Містить зв'язок «один до багатьох» з таблицею Users через поле AuthorId – у користувача може бути багато публікацій, але у публікації один автор.

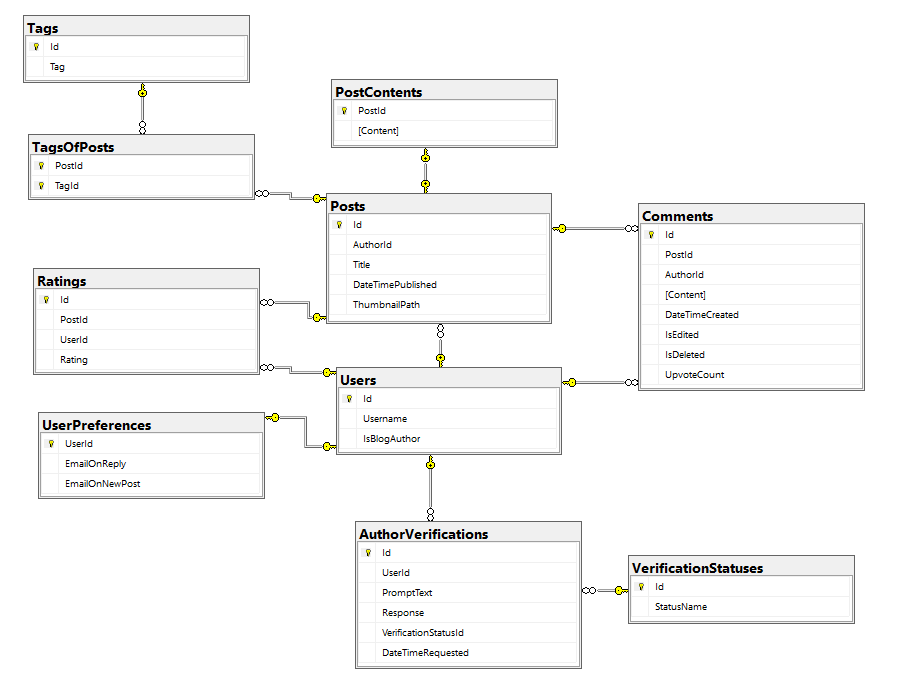


Рисунок 1.5 – Діаграма створеної бази даних

3. PostContents (Вміст публікацій) – допоміжна сутність для збереження вмісту постів (Content), яка виникла внаслідок нормалізації таблиці Posts – контент зберігається окремо від публікації через гарантовано великі об’єми даних, що зберігаються в цьому полі. Таблиці мають зв’язок «один до одного» - кожному запису в Posts відповідає один запис в PostContents.

4. Comments (Коментарі) – одна з основних сутностей, яка відповідає за зберігання даних про коментарі користувачів до публікацій. Містить текст коментаря (Content), дату створення (DateTimePublished), логічні значення, чи є коментар редагованим або видаленим (IsEdited, IsDeleted), сумарна оцінка (UpvoteCount). Містить зв’язки «один до багатьох» із таблицями Posts та Users – пост і користувач можуть мати багато коментарів, однак коментар належить до одного поста та має одного автора.

5. Ratings (Оцінки) – сутність, що зберігає користувацькі оцінки публікацій в полі Rating. Має аналогічні зв’язки з таблицями Posts та Users, як у таблиці Comments.

6. Tags (Теги) – сутність відповідає за збереження назв категорій постів, іншими словами, тегів. Містить зв’язок «багато до багатьох» з таблицею Posts – публікація може мати багато тегів, і тег може належати до багатьох публікацій. Зв’язок реалізовано за допомогою проміжної таблиці TagsOfPosts (Теги публікацій).

7. AuthorVerifications (Підтвердження авторів) – сутність, що являє собою заявку від користувача на роль автора. Містить текст заявки та відповідь на неї (PromptText, Response), дату подачі (DateTimeRequested). Має два зв’язки «один до багатьох» – з таблицею Users (автор заявки) та VerificationStatuses (статус підтвердження).

8. VerificationStatuses (Статуси підтвердження) – допоміжна таблиця, що зберігає можливі статуси обробки заявки на роль автора.

9. UserPreferences (Вподобання користувача) – таблиця зі зв’язком «один до одного» із Users, що відповідала за профіль та налаштування користувацьких вподобань. Містила логічні значення, чи варто сповіщати електронною поштою про нову публікацію та відповідь на коментар (EmailOnNewPost, EmailOnReply).

## Вибір архітектури додатку

Дослідивши джерела [6-8], було прийнято рішення обрати для реалізації вебдодатку монолітну тришарову архітектуру.

Монолітна архітектура (часто «моноліт») передбачає реалізацію продукту у вигляді єдиної, модульно неподіленої системи. Так, додаток розгортається на одному сервері, має єдину точку входу у вигляді прикладного інтерфейсу (API), працює з єдиною базою даних. Також очевидно, що програмний код у такому випадку написано з використанням тої самої технології. При цьому система містить набір сервісів, які таким чином вільно обмінюються даними.

Часто монолітну архітектуру розглядають у протиставленні до мікросервісної, у рамках якої система модульно розділяється на набір мікросервісів, кожен з яких зазвичай розгорнутий окремо та має власну базу даних із набором сутностей. Це набагато покращує масштабованість та відмовостійкість додатку, проте також вимагає ретельного проектування всіх необхідних модулів, а особливо їх взаємодії між собою.

З урахуванням потреб проекту, на даному етапі було обрано «моноліт». Звісно, є ризики сильної зв’язаності компонентів, при виникненні несправності в одній частині відмовить увесь додаток цілком. Однак при цьому зберігається простота розробки, оскільки є можливість реалізувати необхідний функціонал засобами єдиної технології або стеку технологій, а також простота розгортання у разі необхідності, не доводиться підтримувати декілька незалежних сервісів та баз даних одночасно. Варто додати, що часто розробка мікросервісних додатків починається саме з «моноліту», який потім розділяють на логічні частини. Для свого проекту в майбутньому я також планую перехід на мікросервісну архітектуру.

Тришарова архітектура є видом багатошарової (n-layer / n-tier) архітектури [8], яка застосовується до внутрішньої організації програмних компонентів системи. Таким чином, програмний код поділяється на три шари:

1. Data Access Layer (DAL) – шар доступу до даних. Вся взаємодія з базами даних повинна відбуватися винятково в його межах. Надає інтерфейс взаємодії з даними для наступних шарів.

2. Business Logic Layer (BLL) – шар бізнес-логіки. Бізнес-логіка являє собою реалізацію функціональних вимог до проекту з точки зору предметної області. Часто містить набір сервісів для виконання різних завдань, інтерфейс роботи з якими надається для наступних шарів.

3. Presentation Layer (PL) – шар представлення, що надає інтерфейс для роботи з додатком у цілому. Для вебдодатків він часто реалізується за допомогою графічних компонентів, що взаємодіють з даними. Поширеною практикою є відокремлення графічного інтерфейсу користувача (GUI) у вигляді обособленого клієнта. Тоді замість PL застосовується також назва Web API Layer, який надає клієнту програмний інтерфейс взаємодії.

Тришарова архітектура цілком задовольняє потреби проекту, забезпечує достатній ступінь ізольованості частин, дозволяє окремо організувати back-end та front-end (докладно в розділах 2 та 3 відповідно).

# ВИСНОВКИ ДО РОЗДІЛУ

Для додатку, що розробляється, обрано предметну область, а саме колективний блог, її досліджено та описано. Виділено основні моменти предметної області, на які варто звернути увагу під час реалізації. Обґрунтовано проблематику та доцільність вибору теми.

Розглянуто наявні програмні аналоги: Habr та CodeProject. Проаналізовано їх переваги та недоліки, важливі аспекти роботи. Сильні та слабкі сторони взято до уваги при формуванні вимог до кінцевого продукту, щоб намагатися використати найкращі практики при розробці.

Сформовано перелік User Stories, що визначають повний список функціональних вимог, у тому числі тих, що будуть реалізовані в майбутньому. Визначено вимоги для таких типів користувачів системи: Анонім, Читач, Автор, Адміністратор.

Наведено модульно-аналітичну схему додатку, яка дозволяє розділити його на п’ять окремих логічних частин. У подальшому її можна застосувати при переході до мікросервісної архітектури.

Спроектовано первинну базу даних засобами СКБД MS SQL Server та мови запитів Transact-SQL. Наведено структуру та тлумачення сутностей і зв’язків у базі даних.

Вибрано для додатку монолітну тришарову архітектуру. Розглянуто переваги та недоліки «моноліту» та мікросервісів, обґрунтовано вибір з урахуванням вимог проекту. Наведено переваги використання тришарової архітектури для функціональної ізоляції частин додатку.

# РОЗДІЛ 2. BACK-END РОЗРОБКА

## Вибір стеку технологій

Для розробки back-end частини за основу я взяв платформу .NET 5 та, відповідно, мову програмування С# (версії 9.0). .NET є технологією компанії Microsoft з 20-річною історією, що показала себе як надійне та потужне середовище для розробки Enterprise-додатків [9]. У 2016 році на подальшу зміну .NET Framework з підтримкою винятково Windows вийшов крос-платформний та модульний .NET Core, що відтоді підтримує також Linux та macOS. Ця подія та подальші якісні зміни тільки затвердили його релевантність, і в 2020 році відбувся реліз .NET 5, що затвердив початок об’єднання всіх .NET технологій та закріпив їх позицію на ринку програмного забезпечення. Окрім того, привабливим також є інтегроване середовище розробки Visual Studio, що надає повний набір інструментів для продуктивного програмування.

Ще одною технологією серед основних є вебфреймворк ASP.NET Core 5 (Web API), що широко застосовується для розробки сучасних вебдодатків. Працюючи на базі .NET 5, він також розгортається на всіх необхідних платформах, використовуючи традиційний сервер IIS або крос-платформний Kestrel. Перевагами ASP.NET Core є модульність, що досягається з використанням пакетного менеджера NuGet, вбудований IoC-контейнер [10], а також механізми роботи з користувачами за допомогою технології Identity.

Оскільки використовується СКБД SQL Server, також продукт Microsoft, то додаткова перевага .NET виявляється у вбудованій підтримці підключення до неї у вигляді технології ADO.NET. Вона надає вичерпний функціонал з надсилання запитів до бази даних, підтримує підключений та відключений режими роботи. Проте для додатків, що стрімко розростаються, популярним вибором є Entity Framework (EF) – ORM-технологія (об'єктно-реляційної проекції) [11], в центрі якої – концепція сутностей, що переносяться з реляційних таблиць до C#-об’єктів та відслідковуються фреймворком на зміни.

На цьому етапі виникло питання про доцільність вибору тої чи іншої моделі роботи з базою даних SQL Server, враховуючи наявність для ADO.NET також micro-ORM бібліотеки Dapper, що спрощує надсилання запитів та бере на себе проекцію даних. Розв’язання цієї проблеми я вирішив знайти практично, використавши обидві технології для реалізації проекту поступово. Базова реалізація за допомогою ADO.NET + Dapper наведена в підрозділі 2.2. Розробка повноцінного вебдодатку засобами EF Core наведена в підрозділі 2.3. Отримані результати аналізуються в підрозділі 2.4.

Також використано низку допоміжних бібліотек для .NET та ASP.NET Core, а саме:

1. AutoMapper. Надає інструменти для проекції даних між різнотипними об’єктами C# та її конфігурації.

2. Swagger (OpenAPI). Формує документацію API та створює зручний графічний інтерфейс для тестування точок входу.

3. FluentValidation. Замінює стандартну валідацію вхідних даних, засновану на анотаціях, гнучкими наборами правил для властивостей окремих класів.

4. MimeTypes. Зіставляє розширення файлу із MIME-типом HTTP-вмісту [12], що використовується для надсилання файлів у мережі.

## Реалізація з використанням ADO.NET та Dapper

При розробці використано підключений режим роботи ADO.NET. Матеріали для ознайомлення з технологіями взято з джерел [13-15].

Реалізовано патерн Generic Repository – абстракцію з CRUD-операціями до бази даних для будь-якої сутності – на ADO.NET і окремо на Dapper. Таким чином, можна порівняти, наскільки останній спрощує роботу та скорочує кількість програмного коду.

### Збережувані процедури

Для взаємодії з базою даних було використано ряд збережуваних процедур, що дозволяють звертатися до будь-якої таблиці, що має ключове поле Id, та виконувати з нею CRUD-операції. На рис. 2.1 наведено SQL-запит на створення однієї з таких процедур (Update).



Рисунок 2.1 – Запит на створення процедури Update

Параметри даної процедури – назва таблиці, рядок зі списком стовпців для оновлення та Id таблиці. На основі цих даних динамічно формується та виконується рядок з SQL-запитом. За подібним принципом працюють і решта процедур (GetAll, GetById, Insert, Delete). Повний запит на їх створення наведено в додатку А2.

### Підключення до бази даних

Підключення до SQL Server здійснюється за допомогою System.Data.SqlConnection із вказанням рядка підключення до бази даних. Над цією логікою побудовано абстракцію фабрики підключень (рис. 2.2).

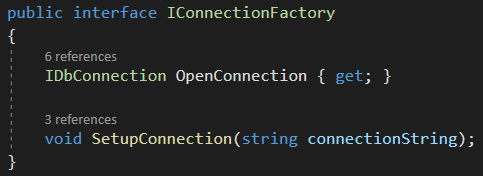


Рисунок 2.2 – Інтерфейс фабрики підключень

Властивість OpenConnection повертає налаштоване та відкрите підключення. Метод SetupConnection встановлює рядок підключення до бази даних. Конкретна реалізація фабрики у властивості містить логіку створення підключення саме до SQL Server, як показано на рис. 2.3.

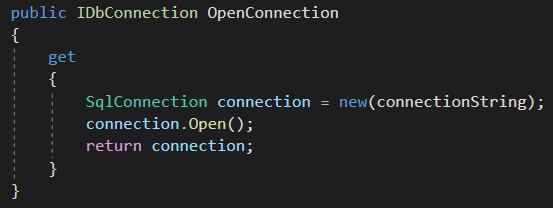


Рисунок 2.3 – Створення та відкриття підключення

У подальшому об’єкт фабрики передається до репозиторію за допомогою ін’єкції залежності [16] та підключення отримується шляхом виклику властивості OpenConnection.

### Generic Repository. Порівняння запитів на ADO.NET і Dapper

Generic Repository – це патерн проектування, який дозволяє абстрагувати логіку звертання до таблиць бази даних через набір методів. При цьому він повинен типізуватися певною моделлю сутності, яка представлена програмним класом та стосовно якої виконуватимуться операції. Інтерфейс такого репозиторію наведено на рис. 2.4.

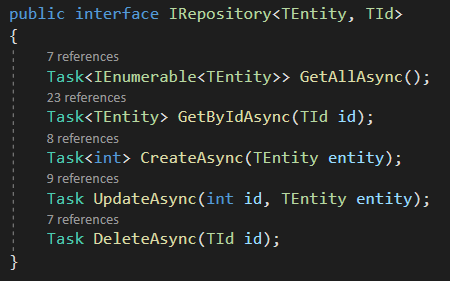


Рисунок 2.4 – Інтерфейс Generic Repository

Реалізація цього інтерфейсу, у свою чергу, містить логіку звертання до бази даних у кожному з методів. Поглянемо на реалізацію з ADO.NET, без допоміжних методів Dapper (рис. 2.5, 2.6).

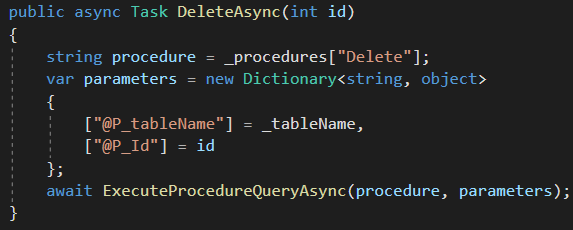


Рисунок 2.5 – Реалізація методу Delete (ADO.NET)

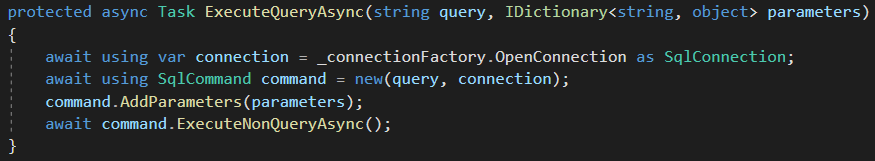


Рисунок 2.6 – Допоміжний метод для виконання запиту

Як бачимо, для коректної роботи необхідний допоміжний метод, а таких для інших операцій оголошено ще декілька. У даному, ExecuteQueryAsync, відкривається підключення за допомогою фабрики, створюється SqlCommand, що репрезентує запит на виконання, за допомогою методу розширення до нього додаються необхідні параметри і він виконується.

Окрім всього, в конкретному репозиторії сутності необхідно визначати абстрактні методи для проекції даних та отримання параметрів сутності для запиту (рис. 2.7).

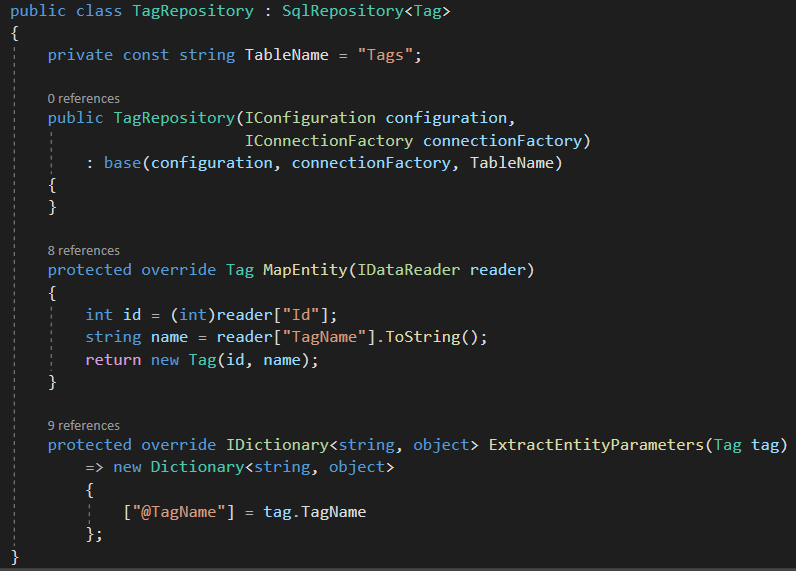


Рисунок 2.7 – Клас TagRepository

Поглянемо на реалізацію методу за допомогою Dapper (рис. 2.8). Можна помітити, що виконання запиту значно скорочується, оскільки функціонал для передавання параметрів і проекції даних вже реалізовано. Крім цього, більше не потрібні допоміжні методи. Через метод розширення Dapper – ExecuteAsync – передається назва процедури, параметри у вигляді анонімного об’єкта та тип команди, чого вже достатньо для виконання запиту до бази даних.

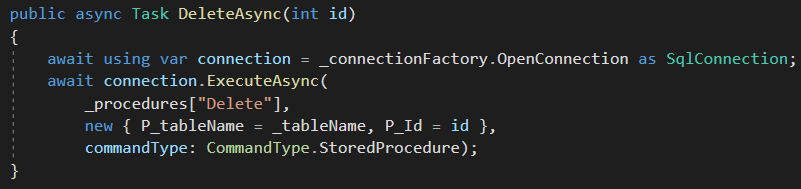


Рисунок 2.8 – Реалізація методу Delete (Dapper)

### Контролери

Після реалізації репозиторіїв, на ADO.NET чи на Dapper, достатньо лише отримати їх в контролері за допомогою dependency injection (ін’єкції залежності) та викликати описані методи. Приклади наведено на рис. 2.9, 2.10.

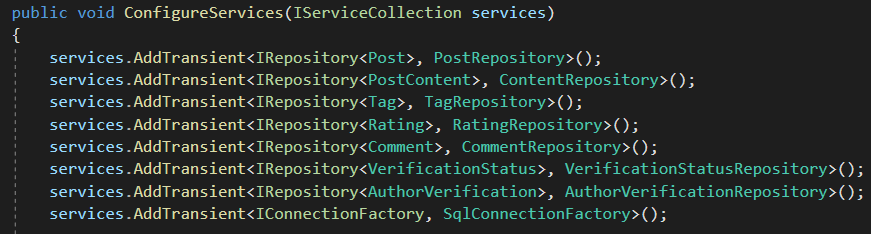


Рисунок 2.9 – Реєстрація репозиторіїв та фабрики для DI

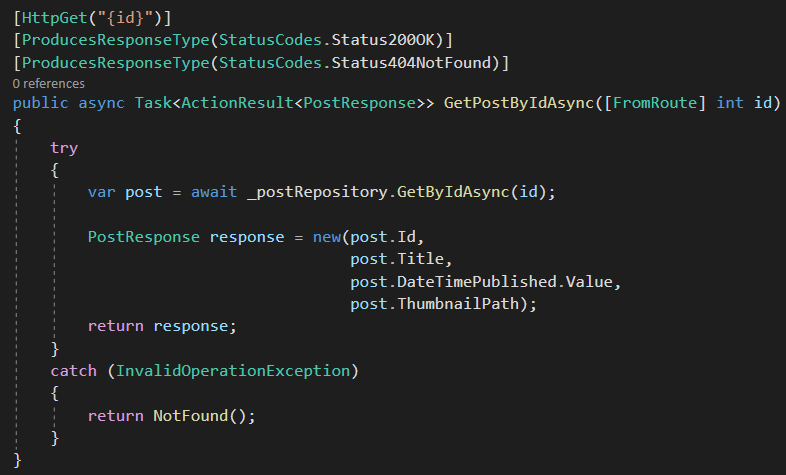


Рисунок 2.10 – Get-метод контролера

Тут за ідентифікатором з бази даних отримується запис публікації, і далі над нею проводяться будь-які необхідні дії. Варто зауважити, що ця версія проекту демонстраційна, оскільки в ній опущено шар бізнес-логіки та інші необхідні елементи повноцінного вебдодатку, однак вона дає нам зрозуміти роботу ADO.NET та Dapper, а також в чому перевага використання останнього.

## Основна реалізація з використанням Entity Framework Core

### Code-First підхід до створення бази даних

При роботі з EF Core зазвичай застосовується Code-First підхід – спершу оголошуються класи сутностей, і, з урахуванням налаштованої конфігурації, автоматично створюється база даних.

Кожна сутність у додатку має набір спільних полів – ідентифікатор, дат створення да останньої зміни. Нижче наведено код абстрактної сутності EntityBase, яка є базовим класом для інших.

public abstract class EntityBase<TId>

{

public TId Id { get; init; }

public DateTime CreatedOn { get; init; }

public DateTime UpdatedOn { get; set; }

}

Також варто навести деякі конкретні сутності: Post, Comment. Решта сутностей реалізовані схожим чином і повторюють структуру бази даних (див. підрозділ 1.5). Окрім властивостей, що повторюють поля таблиці, є також навігаційні властивості, які використано для конфігурації зв'язків між сутностями.

public class Post : EntityBase<int>

{

public string AuthorId { get; set; }

public User Author { get; set; }

public string Title { get; set; }

public string TitleIdentifier { get; set; }

public string ThumbnailPath { get; set; }

public PostContent ContentEntity { get; set; }

public IList<Tag> Tags { get; set; }

public IEnumerable<Comment> Comments { get; set; }

public IEnumerable<Rating> Ratings { get; set; }

}

public class Comment : EntityBase<int>

{

public int PostId { get; set; }

public Post Post { get; set; }

public string AuthorId { get; set; }

public User Author { get; set; }

public string Content { get; set; }

public int UpvoteCount { get; set; }

}

Центральний клас при використанні Entity Framework Core – контекст даних, або DbContext. Він містить набір властивостей типу DbSet, що моделюють таблиці в базі даних. У додатку повинен бути власний клас контексту, що наслідується від DbContext. Повний код класу контексту наведений у додатку Б1. Нижче буде розглянуто деякі важливі уривки. Наприклад, оголошення властивостей DbSet:

public DbSet<Post> Posts { get; set; }

public DbSet<PostContent> PostContents { get; set; }

public DbSet<Comment> Comments { get; set; }

public DbSet<Tag> Tags { get; set; }

public DbSet<Rating> Ratings { get; set; }

public DbSet<AuthorVerification> AuthorVerifications { get; set; }

public DbSet<VerificationStatus> VerificationStatuses { get; set; }

public DbSet<UserProfile> UserProfiles { get; set; }

Наведеного вище коду достатньо, щоб EF Core міг створити окремі таблиці з набором полів. Однак для накладання обмежень та ключів також застосовується конфігурування контексту.

### Конфігурація сутностей з використанням Fluent API

Сутності можна конфігурувати або безпосередньо в методі OnModelCreating, або створити класи конфігурації для кожної сутності та винести логіку в них, що дозволяє розділити обов'язки та зменшити об'єм контексту. Так реєструються в OnModelCreating всі класи конфігурацій зі збірки:

modelBuilder.ApplyConfigurationsFromAssembly(this.GetType().Assembly);

Клас конфігурації сутності Post наведено нижче.

internal class PostConfiguration : BaseEntityConfiguration<Post, int>

{

public override void Configure(EntityTypeBuilder<Post> builder)

{

base.Configure(builder);

builder.HasIndex(p => p.Title)

.IsUnique();

builder.HasIndex(p => p.TitleIdentifier)

.IsUnique();

builder.Property(p => p.Title)

.IsRequired()

.HasMaxLength(200);

builder.Property(p => p.TitleIdentifier)

.IsRequired()

.HasMaxLength(100);

builder.Property(p => p.ThumbnailPath)

.HasMaxLength(500);

builder.HasOne(p => p.Author)

.WithMany(u => u.Posts)

.HasForeignKey(p => p.AuthorId)

.OnDelete(DeleteBehavior.SetNull);

builder.HasOne(p => p.ContentEntity)

.WithOne(pc => pc.Post)

.HasForeignKey<PostContent>(pc => pc.Id)

.OnDelete(DeleteBehavior.Cascade);

builder.HasMany(p => p.Tags)

.WithMany(t => t.Posts)

.UsingEntity(j => j.ToTable("TagsOfPosts"));

}

}

За допомогою Fluent API тут конфігуруються обмеження унікальності заголовка, необхідність його введення та максимальна довжина 200 символів тощо. В трьох останніх ланцюгах методів можна побачити налаштування трьох типів зв’язків: «один до багатьох», «один до одного», «багато до багатьох».

Також Fluent API дозволяє проводити seeding – введення початкових даних у таблиці при створенні. Наприклад, так у методі OnModelCreating задаються ролі:

modelBuilder.Entity<IdentityRole>(role =>

{

role.HasData(

new() { Id = "9341400c-1856-46d2-a17f-841be8d59d59", Name = "Reader", NormalizedName = "READER" },

new() { Id = "0428f7f5-3ddd-4892-bfaf-adb1bf0ed1c6", Name = "Author", NormalizedName = "AUTHOR" },

new() { Id = "2186fb46-ec78-4ff3-87d5-63196ee65bdf", Name = "Admin", NormalizedName = "ADMIN" }

);

});

### Generic Repository. Unit of Work

У поєднанні з Entity Framework Core, Generic Repository по суті виступає додатковою абстракцією над DbSet, щоб послабити залежність. Разом із ним також використовується підхід Unit of Work («одиниця роботи») – передача об’єкту, що містить посилання на всі конкретні репозиторії, і є абстракцією над DbContext. Таким чином, через Unit of Work можна отримати доступ до будь-яких потрібних даних. Крім того, він відкидає потребу в оновленні окремих репозиторіїв, оскільки сам повинен містити метод збереження всіх змін. Повний код Generic Repository надано в додатку Б2, Unit of Work – в додатку Б3.

Конкретні репозиторії наслідують клас Generic Repository, тобто мають стандартний набір методів, а також надають власні, що стосуються певної більш точкової роботи з даною сутністю. Наприклад, один із них:

public class TagRepository : EntityRepository<Tag, int>, ITagRepository

{

public TagRepository(BlogContext context)

: base(context)

{

}

public async Task<IEnumerable<Tag>> GetRelevantTagsAsync()

{

var tags = await \_set.Include(t => t.Posts).ToListAsync();

return tags.Where(t => t.Posts != null && t.Posts.Any());

}

public async Task<Tag> GetTagByNameAsync(string name)

{

return await EnsureEntityResultAsync(() =>

{

return \_set.SingleAsync(t => t.TagName == name);

});

}

}

Тут за допомогою технології LINQ to Entities відбувається робота з сутностями через DbSet, а також застосовано метод Include для eager loading («жадібного завантаження») зв’язаних сутностей через навігаційні властивості.

Для використання отриманих класів необхідно зареєструвати їх для dependency injection у конфігурації проекту ASP.NET Core:

services.AddTransient<IPostRepository, PostRepository>();

services.AddTransient<IPostContentRepository, PostContentRepository>(); services.AddTransient<ICommentRepository, CommentRepository>();

services.AddTransient<IRatingRepository, RatingRepository>();

services.AddTransient<ITagRepository, TagRepository>();

services.AddTransient<IVerificationRepository, VerificationRepository>();

services.AddTransient<IVerificationStatusRepository, VerificationStatusRepository>();

services.AddTransient<IProfileRepository, ProfileRepository>();

services.AddTransient<IBloggingUnitOfWork, UnitOfWork>();

### ASP.NET Core Identity

Технологія Identity додає до контексту даних інструменти для роботи з користувачами та автоматичної генерації таблиць у базі даних. Для цього необхідно наслідувати контекст від класу IdentityDbContext, з можливістю типізації власними реалізаціями користувача (User) та ролі (Role) за необхідності. Наприклад, для зв’язування користувача з іншими сутностями визначено таку реалізацію User:

public class User : IdentityUser

{

public UserProfile Profile { get; set; }

public IEnumerable<Post> Posts { get; set; }

public IEnumerable<Comment> Comments { get; set; }

public IEnumerable<Rating> Ratings { get; set; }

public IEnumerable<AuthorVerification> Verifications { get; set; }

}

Тепер до контексту, а також до Unit of Work, можливо додати менеджерів користувачів, ролей та входу, що містять відповідні методи для роботи:

public UserManager<User> UserManager { get; set; }

public RoleManager<IdentityRole> RoleManager { get; set; }

public SignInManager<User> SignInManager { get; set; }

### Міграції. Зміни структури бази даних у процесі розробки

За допомогою механізму міграцій EF Core дозволяє ітеративно оновлювати структуру базу даних, залежно від змін у сутностях та їх конфігурації. У консолі менеджера пакетів, встановивши пакет EntityFrameworkCore.Tools, можна ввести команди:

1. *add-migration MigrationName* – створює нову міграцію з усіма незбереженими змінами структури бази даних.

2. *update-database* – оновлює структуру базу даних відповідно до останньої міграції.

Внаслідок процесу розробки і послідовного застосування міграцій, структура бази даних (наведено в підрозділі 1.5) змінилася. Оновлену діаграму наведено на рис. 2.11.

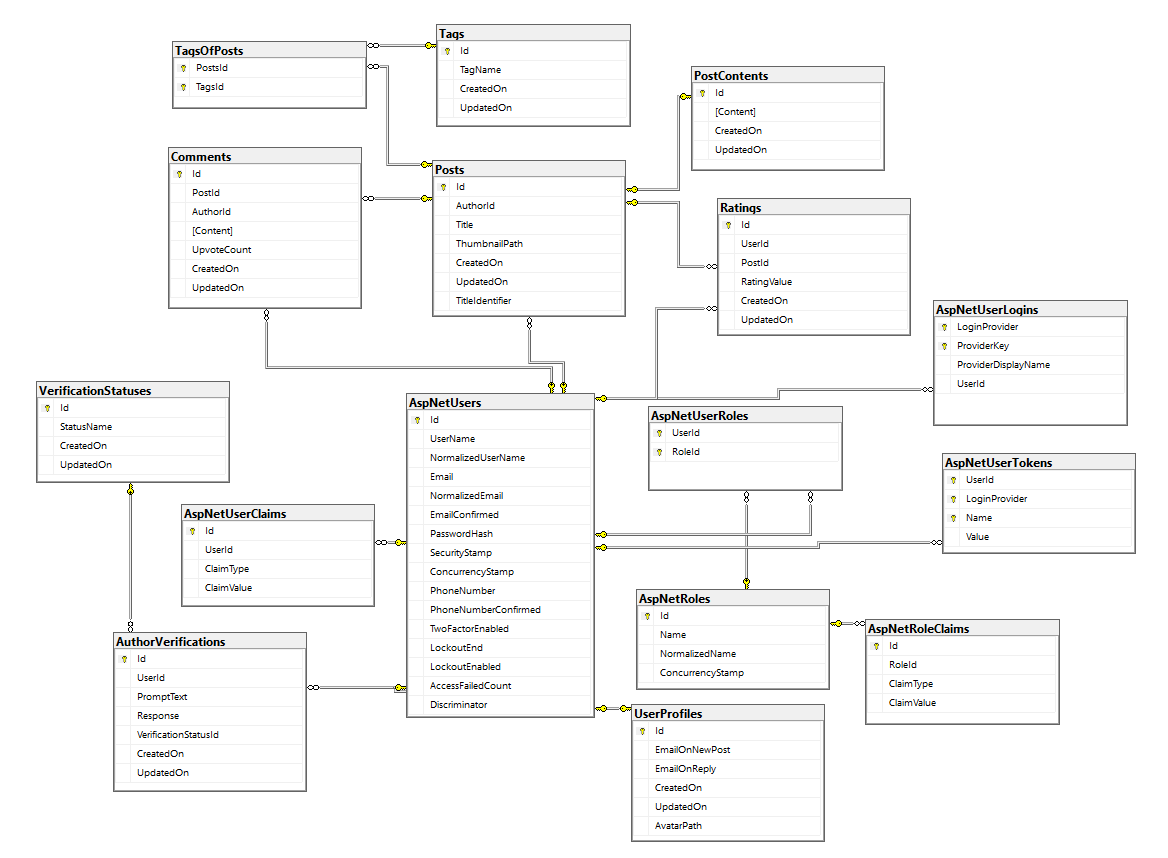


Рисунок 2.11 – Оновлена діаграма бази даних

Основні зміни відбулися при інтеграції Identity – додано 7 згенерованих таблиць для користувачів, ролей тощо. Також до всіх сутностей додано поля аудиту – дата та час створення і зміни запису. Додано кілька нових полів для відповідності вимогам предметної області.

### Бізнес-логіка. DTO. AutoMapper

У шарі бізнес-логіки ми отримуємо доступ до даних через ін’єкцію Unit of Work до сервісів, та з його використанням формуємо потрібні методи. Приклад бізнес-логіки (створення публікації в PostService):

public async Task<PostResponse> PublishPostAsync(PostRequest postDto, string authorId)

{

Post post = \_mapper.Map<Post>(postDto);

post.ContentEntity = \_mapper.Map<PostContent>(postDto);

post.AuthorId = authorId;

string cutTitle = post.Title.Length switch

{

> 100 => post.Title[..100],

\_ => post.Title

};

string cutLowerTitle = cutTitle.ToLowerInvariant();

string wordOnlyString = Regex.Replace(cutLowerTitle, @"[^\w\s]", string.Empty);

post.TitleIdentifier = Regex.Replace(wordOnlyString, @"\s+", "-");

await \_unitOfWork.Posts.CreateAsync(post);

await \_unitOfWork.CommitAsync();

var response = \_mapper.Map<PostResponse>(post);

this.AddRelativeTimeToResponse(response);

return response;

}

Метод приймає дані про публікацію, встановлює її автора та вміст, формує заголовковий ідентифікатор (форма заголовку, що використовується для ідентифікації поста в URL-адресі), створює запис та зберігає зміни. Створена публікація повертається з методу.

Тут, очевидно, стикаємося з потребою перетворення форми даних у процесі роботи додатку. Так, дані, що надходять до сервісу ззовні, відрізняються від структури сутностей, аналогічно й дані, що сервіс видає назовні. Для таких цілей використовуються класи DTO (Data Transfer Object – об’єкт транспортування даних). У даному проекті вони поділені на дві групи: вхідні дані (Requests, або запити) та вихідні (Responses, або відповіді).

Задачу перетворення даних із сутності в DTO та навпаки виконує бібліотека AutoMapper (потребує реєстрації сервісів для ін’єкції). Вона надає набір методів та способи конфігурації «маппінгів». Профіль конфігурації наведено в додатку Б4. Приклад нижче:

CreateMap<Comment, CommentResponse>()

.ForMember(res => res.Author, opt => opt.MapFrom(c => c.Author.UserName)) .ForMember(res => res.PublishedOn, opt => opt.MapFrom(c => c.CreatedOn)) .ForMember(res => res.IsEdited, opt => opt.MapFrom(c => c.UpdatedOn > c.CreatedOn));

Дана конфігурація вказує, за яким правилом перетворювати ті чи інші властивості. За замовчуванням значення копіюються в однойменні властивості, проте інші випадки необхідно налаштовувати. Так, тут автор коментаря береться із завантаженої навігаційної властивості, дата оприлюднення із властивості з іншою назвою, а значення відредагованості коментаря задається логічним виразом.

Набір сервісів шару бізнес-логіки можна побачити на рис. 2.12. Вони зареєстровані для ін'єкції у конфігурації.

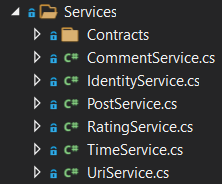


Рисунок 2.12 – Сервіси додатку

Функції даних сервісів:

1. CommentService – коментування.

2. PostService – робота з публікаціями.

3. RatingService – оцінювання.

4. UriService – формування необхідних Uri-адрес.

5. TimeService – робота з часом публікування.

6. IdentityService – робота з користувачами.

### Авторизація на основі JWT-токенів

IdentityService виконує операції над користувачами системи, такі як реєстрація, вхід, зміна пароля тощо. Програмний код сервісу надано в додатку Б5. Варто розглянути деякі уривки.

JWT-токен (JSON Web Token) описано у джерелі [17]. Це стандарт зберігання і надсилання даних, що часто використовується для передавання даних про користувача у мережі.

Так, після реєстрації та входу формується токен для відправки тверджень (claims) користувача у методі GenerateJwtToken. Розглянемо його:

private async Task<string> GenerateJwtToken(User user)

{

DateTime expirationTime = DateTime.UtcNow.AddDays(\_daysToExpire);

string secret = \_configuration.GetValue<string>("TokenSecret");

var keyBytes = Encoding.UTF8.GetBytes(secret);

var userRoles = await \_unitOfWork.UserManager.GetRolesAsync(user);

List<Claim> claims = new()

{

new Claim("id", user.Id),

new Claim(ClaimTypes.Name, user.UserName),

new Claim(ClaimTypes.Email, user.Email),

new Claim(ClaimTypes.Role, string.Join(", ", userRoles))

};

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new(claims),

Expires = expirationTime,

SigningCredentials = new(new SymmetricSecurityKey(keyBytes),

SecurityAlgorithms.HmacSha256Signature)

};

var tokenHandler = new JwtSecurityTokenHandler();

var token = tokenHandler.CreateToken(tokenDescriptor);

return tokenHandler.WriteToken(token);

}

З даних користувача формується список тверджень: ID, ім'я, електронна пошта, роль. Саме задання ролі дозволяє працювати механізму авторизації – надання певних прав доступу користувачеві залежно від ролі. Далі список перетворюється на токен з визначеним терміном життя, він підписується та створюється, повертаючись у форматі рядка. В подальшому контролери перехоплюють токен із HTTP-заголовка Authorization та декодують дані про поточного користувача, таким чином за потреби обмежуючи доступ до певних ресурсів на основі ролі. Для цього використовується атрибут контролера Authorize, який можливо налаштувати на пропуск певних ролей:

[Authorize(Roles = "Admin, Author")]

### Пагінація, фільтрація, пошук даних

Пагінація (поділ на сторінки) дозволяє обмежити кількість даних, що надсилаються клієнту, тим самим зменшивши навантаження на його ресурси. Пагінацію реалізовано на рівні бізнес-логіки, наприклад, для публікацій:

public async Task<Page<PostResponse>> GetPageOfPostsAsync(PostFilter filter = null)

{

Uri GetPageUri(int pageNumber)

{

PostFilter anotherFilter = filter.CopyWithDifferentPage(pageNumber);

return \_uriService.GetPostsPageUri(anotherFilter);

}

var filteredPosts = await \_unitOfWork.Posts.GetFilteredPostsAsync(filter);

var pagedPosts = await filteredPosts.Paginate(filter).ToListAsync();

var responseList = \_mapper.Map<List<Post>, List<PostResponse>>(pagedPosts);

responseList.ForEach(this.AddRelativeTimeToResponse);

Uri previousPageUri = GetPageUri(filter.PageNumber - 1);

Uri nextPageUri = GetPageUri(filter.PageNumber + 1);

return new(responseList, filteredPosts.Count(), filter, previousPageUri, nextPageUri);

}

Тут репозиторій повертає фільтровані пости за переданими критеріями (розглянемо нижче), вони пагінуються методом розширення (за допомогою методів LINQ Skip та Take) відповідно до параметрів сторінки та повертаються, обгорнуті в допоміжний об'єкт Page. Він містить власне дані та метадані пагінації в цілому:

public class Page<T>

{

private IEnumerable<T> \_data;

public IEnumerable<T> Data

{

get => \_data;

set

{

if (value is null)

throw new ArgumentNullException(nameof(value));

\_data = value;

}

}

public int PageNumber { get; set; }

public int PageSize { get; set; }

public int TotalRecords { get; set; }

public int TotalPages

{

get

{

if (PageSize == 0)

return 0;

return (int)Math.Ceiling(TotalRecords \* 1d / PageSize);

}

}

public string PreviousPage { get; set; }

public string NextPage { get; set; }

public Page(IEnumerable<T> data,

int totalRecords,

PaginationFilter filter = null,

Uri previousPageUri = null,

Uri nextPageUri = null)

{

Data = data;

TotalRecords = totalRecords;

PageNumber = filter?.PageNumber ?? 0;

PageSize = filter?.PageSize ?? 0;

if (PageNumber > 1 && Data.Any())

PreviousPage = previousPageUri?.AbsoluteUri;

if (PageNumber < TotalPages)

NextPage = nextPageUri?.AbsoluteUri;

}

}

Зокрема, передаються також URL-адреси наступної та попередньої сторінок (з урахуванням фільтрації), які використовуватимуться для запитів зі сторони клієнта. Формуванням адрес займається сервіс UriService.

Розглянемо також класи фільтрів, що поєднують параметри пагінації, фільтрації та пошуку:

public class PaginationFilter

{

public int PageNumber { get; set; }

public int PageSize { get; set; }

public PaginationFilter()

{

PageNumber = 1;

PageSize = 10;

}

public PaginationFilter CopyWithDifferentPage(int pageNumber)

{

var copy = (PaginationFilter)this.MemberwiseClone();

copy.PageNumber = pageNumber;

return copy;

}

}

public class PostFilter : PaginationFilter

{

public string Title { get; set; }

public string Author { get; set; }

public int? Year { get; set; }

public int? Month { get; set; }

public int? Day { get; set; }

public string Tag { get; set; }

public new PostFilter CopyWithDifferentPage(int pageNumber)

=> (PostFilter)base.CopyWithDifferentPage(pageNumber);

}

PostFilter є нащадком класу PaginationFilter, тобто успадковує параметри номеру та розміру сторінки, яку необхідно повернути. Ці дані обробляються у методі, наведеному вище. Решта параметрів дозволяють фільтрувати публікації за іменем автора, тегом, днем, місяцем і роком оприлюднення, а також проводити пошук по заголовку. Цей функціонал знаходиться у репозиторії та реалізується засобами LINQ to Entities та Eager Loading:

public async Task<IQueryable<Post>> GetNewestPostsWithAuthorsAndTagsAsync()

{

var posts = \_set.Include(p => p.Author)

.Include(p => p.Tags)

.OrderByDescending(p => p.CreatedOn);

return await Task.FromResult(posts);

}

public async Task<IQueryable<Post>> GetFilteredPostsAsync(PostFilter filter)

{

var posts = await this.GetNewestPostsWithAuthorsAndTagsAsync();

if (filter is null)

return posts;

return posts.Where(p => filter.Title == null || p.Title.Contains(filter.Title))

.Where(p => filter.Author == null || p.Author.UserName == filter.Author)

.Where(p => filter.Year == null || p.CreatedOn.Year == filter.Year.Value)

.Where(p => filter.Month == null || p.CreatedOn.Month == filter.Month.Value)

.Where(p => filter.Day == null || p.CreatedOn.Day == filter.Day.Value)

.Where(p => filter.Tag == null || p.Tags.Any(t => t.TagName == filter.Tag));

}

### Контролери. Завантаження файлів. Swagger

У проекті містяться такі контролери: PostController, CommentController, TagController, RatingController, VerificationController – надають «кінцеві точки» (endpoints) для роботи з відповідними сутностями; UserController – для реєстрації, входу, роботи з даними користувачів; FileController – для завантаження та локального розміщення файлів (рис. 2.13).

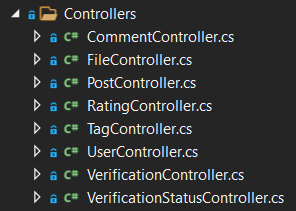


Рисунок 2.13 – Контролери додатку

До прикладу, розглянемо один із методів контролера коментарів:

[HttpDelete("{id}")]

[ProducesResponseType(StatusCodes.Status204NoContent)]

[ProducesResponseType(StatusCodes.Status401Unauthorized)]

[ProducesResponseType(StatusCodes.Status403Forbidden)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

public async Task<ActionResult> DeleteComment([FromRoute] int id)

{

try

{

bool userIsPermitted = await CheckIsAuthorOfCommentOrAdmin(id);

if (!userIsPermitted)

return Forbid();

await \_commentService.DeleteCommentAsync(id);

return NoContent();

}

catch (EntityNotFoundException)

{

return NotFound();

}

}

Він має набір атрибутів, а саме для визначення HTTP-методу та шляху (HttpDelete) і для опису можливих статусних кодів, що ідентифікують результат запиту (ProducesResponseType). У тілі методу перехоплюється користувацьке виключення, якщо запис сутності за ID не знайдено, а також перед викликом сервісу перевіряється, чи дозволено поточному користувачу видалення коментаря, тобто чи є він власне його автором або адміністратором. Дані про користувача отримуються з JWT-токена, який необхідний для авторизації в контролері. Врешті-решт викликається метод для повернення того чи іншого статусного коду, залежно від результату виконання.

Описані дані, а саме про HTTP-метод, статусні коди, типи вхідних та вихідних даних утилізує бібліотека Swagger (OpenAPI). На основі них вона формує документацію API (swagger.json), а завдяки їй – графічний інтерфейс для зручного виклику методів контролера (рис. 2.14).

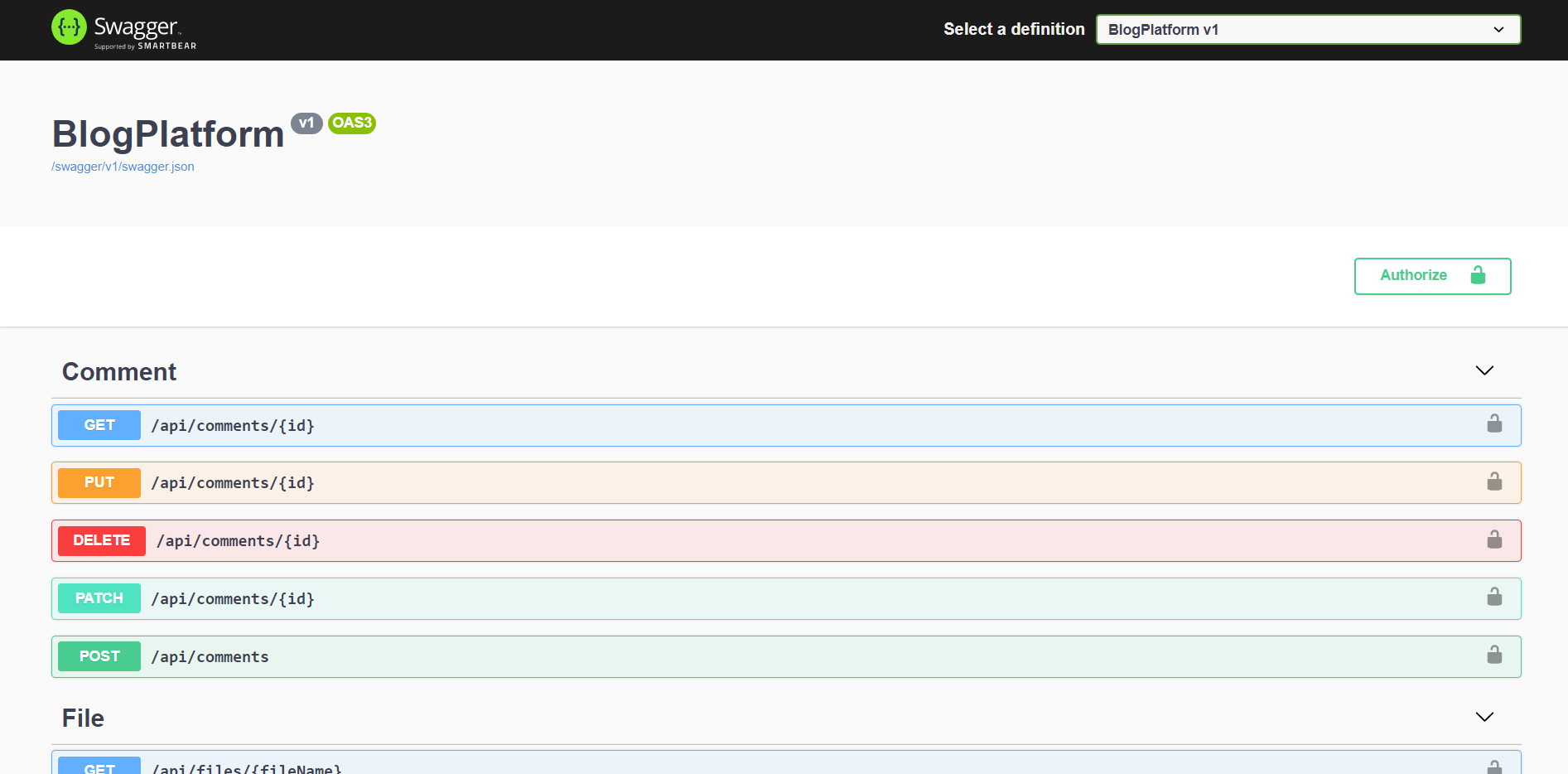


Рисунок 2.14 – Графічний інтерфейс Web API, сформований Swagger

Завантаження користувацьких файлів на сервер реалізовано у FileController. Нижче наведено його методи: отримання файлу за іменем та збереження переданого файлу.

[HttpGet("{fileName}")]

[AllowAnonymous]

[ProducesResponseType(StatusCodes.Status200OK)]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

public async Task<ActionResult> GetImage([FromRoute] string fileName)

{

string filePath = Path.Combine(Directory.GetCurrentDirectory(), "StaticFiles", "Images", fileName);

if (!System.IO.File.Exists(filePath))

return BadRequest();

byte[] imageBytes = await System.IO.File.ReadAllBytesAsync(filePath);

string mimeType = MimeTypes.GetMimeType(fileName);

return File(imageBytes, mimeType);

}

[HttpPost]

[DisableRequestSizeLimit]

[ProducesResponseType(StatusCodes.Status200OK)]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

[ProducesResponseType(StatusCodes.Status401Unauthorized)]

[ProducesResponseType(StatusCodes.Status403Forbidden)]

public async Task<ActionResult<FileUploadResponse>> UploadImage([FromForm] FileUploadRequest fileRequest)

{

IFormFile file = fileRequest.Files;

var dispositionHeader = ContentDispositionHeaderValue.Parse(file.ContentDisposition);

string fileName = dispositionHeader.FileName.Trim('\"');

string constructedFileName = DateTime.UtcNow.ToString("yyyyMMddhhmmssfff") + Path.GetExtension(fileName);

string fileFolderPath = Path.Combine("StaticFiles", "Images");

string filePath = Path.Combine(Directory.GetCurrentDirectory(), fileFolderPath, constructedFileName);

using FileStream stream = new(filePath, FileMode.Create);

await file.CopyToAsync(stream);

return new FileUploadResponse() { LocalPath = constructedFileName };

}

Завантаження файлу реалізовано шляхом його зчитування з даних форми, конструювання нового безпечного локального імені з поточної дати та часу, запис до файлової системи через файловий потік. У відповідь користувач отримує локальну назву. Її можна передати до першого методу, щоб за нею проводився пошук файлу у файловій системі по імені, у разі успіху він зчитувався і повертався у формі файлового HTTP-вмісту. Тут застосовано допоміжну бібліотеку MimeTypes, що знаходить MIME-тип за розширенням файлу.

### Валідація. Локалізація

Важливим моментом при отриманні зовнішніх вхідних даних правильно валідувати їх, щоб не наразити на небезпеку систему через їх неправильний формат. ASP.NET Core традиційно пропонує валідацію на основі атрибутів, проте вона має ряд недоліків, серед них порушення принципу єдиного обов’язку класів [18]. На зміну цьому популярності набула бібліотека FluentValidation. Вона дозволяє підмінити стандартну валідацію своїми засобами, зокрема набором валідаторів для класів вхідних даних (DTO). При її реєстрації в конфігурації також налаштовується автоматична реєстрація всіх валідаторів у збірці. Розглянемо один із них:

public class PostRequestValidator : AbstractValidator<PostRequest>

{

public PostRequestValidator(IStringLocalizer<ValidationResource> locale)

{

RuleFor(pr => pr.Title).NotEmpty()

.Length(3, 200);

RuleFor(pr => pr.ThumbnailPath).Must(BeValidPath)

.When(pr => pr.ThumbnailPath is not null)

.WithMessage(locale["NotAValidPath"]);

RuleFor(pr => pr.Content).NotEmpty()

.MaximumLength(20000);

}

private bool BeValidPath(string path)

{

return path.IndexOfAny(Path.GetInvalidPathChars()) == -1

|| Uri.IsWellFormedUriString(path, UriKind.Absolute);

}

}

Даний клас наслідує абстрактний валідатор, наданий бібліотекою, типізований DTO-класом PostRequest. Прописані правила валідації постановляють, що заголовок не може бути порожнім та повинен мати 3-200 символів, що шлях до зображення заставки повинен бути валідним шляхом (файловим або URL), що вміст поста не може бути порожнім та довшим за 20 тис. символів. При цьому вбудовані правила мають стандартні повідомлення про помилки, які до того ж автоматично локалізуються низкою мов та повертаються зі статусом «Bad Request» у разі невалідності. Проте власні повідомлення, як-от у другому правилі наведеного валідатора, необхідно локалізувати окремо.

Локалізація працює на основі файлів ресурсів (.resx), що являють собою набори ключів-значень необхідними мовами та обираються залежно від системної культури (System.Globalization.CultureInfo.CurrentUICulture). Значення для поточної культури можна отримати за допомогою ін’єкції IStringLocalizer, типізованого необхідним для локалізації класом. Тут ValidationResource є класом-заглушкою, який розташовано в просторі імен валідаторів та дозволяє об’єднати повідомлення валідації в одному файлі на культуру.

Налаштування локалізації при її реєстрації в проекті, як видно на рис. 2.15, дозволяють задати шлях до ресурсів у збірці, підтримувані культури, мову за замовчуванням. На рис. 2.16, 2.17 наведено приклад файлів ресурсів, англійською та українською мовами відповідно.

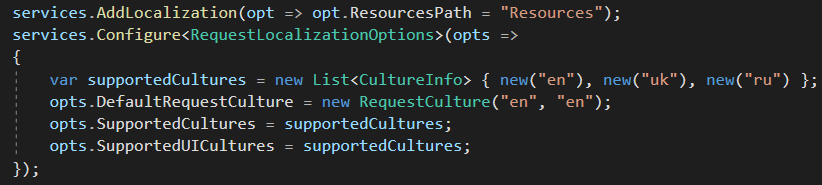


Рисунок 2.15 – Налаштування локалізації



Рисунок 2.16 – Англійська локалізація повідомлень валідації

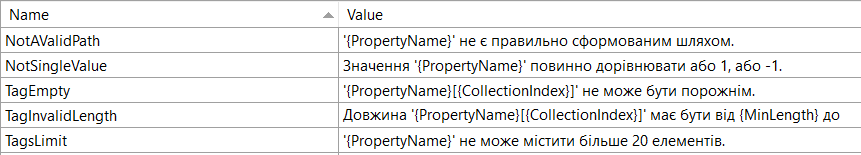


Рисунок 2.17 – Українська локалізація повідомлень валідації

Ще одна локалізована одиниця – TimeService, що повертає відносний час публікації («1 day ago» - «1 день тому» тощо).

## Порівняння ADO.NET (Dapper) та Entity Framework Core

У результаті проведеної роботи над back-end можна проаналізувати використання технології підключення до бази даних. Первинна реалізація на ADO.NET показала, що використання Dapper набагато спрощує роботу із запитами та є по суті необхідною заміною. Тоді питання можна поставити так: «Що краще – Dapper чи EF Core?». Однозначної відповіді немає.

Dapper є легким та швидким аналогом для роботи з базою даних. Швидкість надсилання та обробки запиту суттєво перевищує ту, що досягається при використанні ORM. Простота також на його боці, оскільки налаштування контексту даних для EF Core займає незрівнянно більше часу та потребує більшого об'єму знань про технологію. З іншого боку, коли проект розростається до великих масштабів, підтримувати кожен окремий запит на Dapper стає все важче. Саме в цьому ORM-підхід є сильнішим – він забезпечує цілісність структури бази даних, надає інструменти для її безболісної зміни та легше масштабується. Крім того, EF Core практично абстрагує розробника від безпосередньої роботи з базою даних.

Тому все залежить від конкретного проекту. Якщо робота з БД не є особливо обширною, то Dapper буде чудовим рішенням, що прискорить та спростить її. Для великих же проектів надійність та масштабованість EF Core є пріоритетом.

Для даного додатку другий критерій є важливішим, тому використання Entity Framework Core як основної технології можна вважати доцільним.

# ВИСНОВКИ ДО РОЗДІЛУ

Описано обраний стек back-end технологій, а саме: .NET 5 (C# 9.0), ASP.NET Core, ADO.NET / Dapper, Entity Framework Core, додаткові бібліотеки. Обґрунтовано доцільність його вибору, поставлено запитання про доцільність того чи іншого підходу роботи з базою даних, яке в подальшому досліджено.

Реалізовано базову версію програмного забезпечення з використанням технології ADO.NET та бібліотеки Dapper. Після їх порівняння на однакових задачах з’ясовано, що Dapper як надбудова є більш привабливим варіантом для застосування.

Для основної реалізації було обрано Entity Framework Core, що, як з’ясувалось у підсумку, стало правильним рішенням. Описано структуру, функціонал та особливості роботи всіх трьох шарів внутрішньої архітектури. Розібрано вихідний код додатку та акцентовано увагу на реалізації необхідних вимог. У результаті обсяг виконано не повністю, проте його вибрано для подальшої розробки та розвитку.

Проаналізувавши технології доступу до даних, вдалося виробити критерій вибору. Dapper варто використовувати у проектах помірних масштабів для швидкодії, а повноцінну ORM, як Entity Framework Core, у значних за розмірами проектах, де важлива стабільність.

# РОЗДІЛ 3. FRONT-END РОЗРОБКА

## Вибір стеку технологій

Для розробки клієнтського графічного інтерфейсу вебдодатку було вибрано технологію Blazor Server. Це фреймворк на базі платформи .NET, що дозволяє будувати front-end застосунки із застосуванням розмітки HTML/CSS та компонентів Razor, що використовують мову програмування C# для інтерактивності сторінок [19]. Blazor також підтримує роботу з JavaScript та реалізує компонентний підхід до побудови графічного інтерфейсу, що робить його конкурентоспроможним на фоні SPA фреймворків для JS. Крім того, платформа .NET використана також при розробці back-end, і побудова всієї системи на основі однієї технології, з доступом до однієї кодової бази, бібліотек, є зручною. Версія Server заснована на ASP.NET Core та взаємодіє з браузером через підключення SignalR, тобто має всі можливості фреймворка.

Для більшої продуктивності розробки застосовано фреймворк Bootstrap 4.3.1. За основу для компонування взято шаблон блогу у вільному доступі [22].

Використано перспективну бібліотеку компонентів MudBlazor [20], що надає свою інтерпретацію Material Design [23]. Вона не набула великого поширення станом на 2021 рік, проте стрімко розвивається, має значний набір готових рішень для веброзробників і потенціал майбутнього росту.

Оскільки публікування постів вимагає відповідного форматування, залучено бібліотеку Markdig. Вона являє собою набір утиліт для роботи з Markdown-розміткою [21], що широко застосовується для форматованого редагування текстів у сучасному ПЗ.

Також вжито ресурси двох допоміжних пакетів: Blazored.LocalStorage для спрощеної роботи з локальним сховищем браузера; Blazored.FluentValidation для інтеграції бібліотеки FluentValidation у форми Blazor.

## Робота із компонентами

У проекті широко застосовувалися готові компоненти MudBlazor. Буде описано основні з них, а також розроблені користувацькі компоненти.

### Компоненти MudBlazor

Інформацію про налаштування та використання компонентів бібліотеки наведено в офіційній документації [20]. MudBlazor надає три глобальні компоненти, що розміщуються в App:

1. MudThemeProvider – для налаштування користувацької теми.

2. MudSnackbarProvider – для розміщення на сторінці «снекбара» (динамічного повідомлення про помилку).

3. MudDialogProvider – для роботи з компонентами діалогових вікон.

Для їх коректної роботи необхідно зареєструвати сервіси при конфігуруванні проекту за допомогою методу AddMudServices. Його також перевантажено для налаштування користувацької конфігурації. Рядки підключення стилів та сценаріїв MudBlazor також варто додати до \_Host.

Найчастіше використовуваний компонент – MudTextField, що являє собою текстове поле для вводу з приємним дизайном, можливістю задання надписів («лейблів»), розміщення інтерактивних іконок, підтримка валідації. На рис. 3.1, 3.2 зображено застосування цього компонента на сторінках.

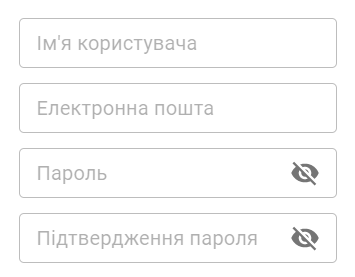


Рисунок 3.1 – Текстові поля MudBlazor на формі реєстрації

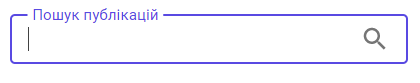


Рисунок 3.2 – Текстове поле MudBlazor як рядок пошуку

Також часто вживаними є компоненти MudButton та MudIconButton – кнопки, представлені надписом та іконкою відповідно. Для другого доступний весь набір Material Icons та декілька авторських іконок бібліотеки, що є суттєвою перевагою використання MudBlazor. MudPaper – контейнер для вмісту з білим фоном та контрольованою елевацією (рівнем затінення) – обрано для тла публікацій та коментарів. MudProgress використано для екрану завантаження, MudRating – для оцінювання публікацій, MudChipset (і MudChip як його складова) – для відображення тегів публікацій. На рис. 3.3, 3.4 показано деякі з них.



Рисунок 3.3 – Компонент MudRating як оцінка публікації

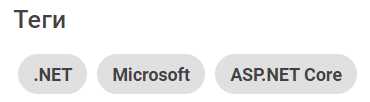


Рисунок 3.4 – Компонент MudChipset як представлення тегів

### Користувацькі компоненти

Декілька компонентів було окремо виділено з метою повторного використання та уникнення повного копіювання розмітки. До них зокрема належить стандартний компонент NavMenu, меню навігації, доповнене і пророблене для власних потреб. Розглянемо код компонента:

@inject IStringLocalizer<NavMenu> \_locale

<nav class="navbar navbar-expand-lg navbar-light fixed-top" id="mainNav">

<div class="container">

<NavLink class="navbar-brand" href="/" Match="NavLinkMatch.All">

Prog Blog

</NavLink>

<button class="navbar-toggler navbar-toggler-right" type="button" data-toggle="collapse"

data-target="#navbarResponsive" aria-controls="navbarResponsive" aria-expanded="false" aria-label="Toggle navigation">

@\_locale["Menu"]

<i class="fas fa-bars"></i>

</button>

<div class="collapse navbar-collapse" id="navbarResponsive">

<ul class="navbar-nav ml-auto">

<li class="nav-item">

<NavLink class="nav-link" href="/" Match="NavLinkMatch.All">

@\_locale["AllPosts"]

</NavLink>

</li>

<li class="nav-item">

<NavLink class="nav-link" href="about">

@\_locale["About"]

</NavLink>

</li>

<li class="nav-item">

<NavLink class="nav-link" href="contact">

@\_locale["Contact"]

</NavLink>

</li>

<AuthorizeView>

<Authorized>

<li class="nav-item">

<NavLink class="nav-link" href="profile">

@\_locale["Profile"]

</NavLink>

</li>

</Authorized>

<NotAuthorized>

<li class="nav-item">

<NavLink class="nav-link" href="login">

@\_locale["SignIn"]

</NavLink>

</li>

</NotAuthorized>

</AuthorizeView>

<AuthorizeView Roles="Author, Admin">

<Authorized>

<li class="nav-item">

<NavLink class="nav-link" href="author">

@\_locale["Author"]

</NavLink>

</li>

</Authorized>

</AuthorizeView>

</ul>

<MudHidden Breakpoint="Breakpoint.MdAndUp" Invert="true">

<LanguageMenu IconColor="Color.Tertiary" />

</MudHidden>

<MudHidden Breakpoint="Breakpoint.MdAndUp">

<LanguageMenu IconColor="Color.Default" />

</MudHidden>

</div>

</div>

</nav>

Як можна побачити, звичайна HTML-розмітка поєднується з Bootstrap, стандартними компонентами Blazor та бібліотеки MudBlazor, а також користувацькими. Для навігації традиційно використано NavLink, як і для авторизованого перегляду AuthorizeView. Так, залежно від того, авторизований користувач чи ні, та в якій ролі, змінюються пункти меню та додаються нові. Реалізовано також локалізацію: ін'єктується IStringLocalizer та повертає різні рядкові значення, залежно від поточної культури. Варто звернути увагу на користувацький компонент LanguageMenu. Він часто повторювано використовується та відповідає за логіку зміни мови на сайті.

@using BlogPlatform.UI.Helpers

@inject NavigationManager \_navManager

<MudMenu Icon="@Icons.Material.Filled.Language" Color="IconColor"

Class="ml-lg-5" OffsetY="true" Dense="true" Size="Size.Medium">

<MudMenuItem OnClick="@(() => SelectCulture("en"))">English</MudMenuItem>

<MudMenuItem OnClick="@(() => SelectCulture("uk"))">Українська</MudMenuItem>

<MudMenuItem OnClick="@(() => SelectCulture("ru"))">Русский</MudMenuItem>

</MudMenu>

@code {

[Parameter]

public Color IconColor { get; set; }

void SelectCulture(string language)

{

string redirectUri = new Uri(\_navManager.Uri).GetComponents(

components: UriComponents.PathAndQuery,

format: UriFormat.Unescaped

);

UriQueryBuilder queryBuilder = new();

queryBuilder.AppendParameter("culture", language);

queryBuilder.AppendParameter("redirectUri", redirectUri);

string query = queryBuilder.ToString();

\_navManager.NavigateTo($"/culture/set{query}", forceLoad: true);

}

}

Маємо компонент, який приймає як параметр колір іконки та має метод для зміни культури. Він формує запит із перенаправленням до контролера культури (рекомендований Microsoft підхід для локалізації) з таким методом:

[Route("culture")]

public class CultureController : Controller

{

[Route("set")]

public ActionResult SetCulture(string culture, string redirectUri)

{

if (culture is not null)

{

HttpContext.Response.Cookies.Append(

CookieRequestCultureProvider.DefaultCookieName,

CookieRequestCultureProvider.MakeCookieValue(

new RequestCulture(culture)

));

}

return LocalRedirect(redirectUri ?? "/");

}

}

Тобто відбувається встановлення нового значення культури в cookie-файл локалізації та зворотне перенаправлення. Налаштування сервісів локалізації, розміщення файлів ресурсів аналогічне до back-end (див. пункт 2.3.10).

Користувацькі компоненти зображено на рис. 3.5-3.8. Код розмітки компонентів, що не був поданий вище, наведено в додатку В1.

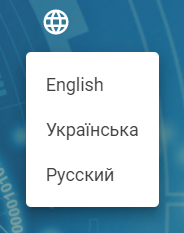


Рисунок 3.5 – Компонент LanguageMenu



Рисунок 3.6 – Компонент NavMenu

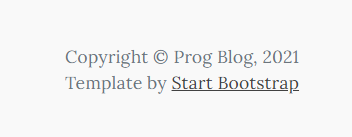


Рисунок 3.7 – Компонент Footer



Рисунок 3.8 – Компонент AuthorBar

## Сторінки

Перелік розроблених сторінок, їхнє функціональне призначення та особливості:

1. Index – головна сторінка. Відображено меню навігації, заголовок сайту, список карток публікацій з пагінацією, поле пошуку по заголовку публікації, перелік 5 найбільш популярних постів, список наявних тегів та фільтрування по них при натисканні (виборі).

2. Post – сторінка окремої публікації. Меню навігації, заголовок поста, автор та відносна дата, загальний рейтинг поста; представлення вмісту за допомогою MarkdigPipeline та конвертації тексту в форматі розмітки Markdown до HTML; перелік тегів, оцінювання публікації, картки коментарів. Можливість для адміністратора видалити коментар, для авторизованого користувача – залишити власний, для неавторизованого – завантажити більше коментарів, якщо є.

3. Login / Register – форми для входу та реєстрації відповідно. Необхідні поля, валідація за допомогою FluentValidation.

4. Profile – тимчасова сторінка профілю користувача. Наявне привітання з виведенням даних користувача та кнопка «Вийти», що виконує вихід з облікового запису.

5. Author – сторінка автора блогу. Список публікацій та інтерфейс для дій з ними: створення, редагування, видалення.

6. AddPost / EditPost – форми для створення і редагування публікацій. Поля для заголовку, вмісту, додавання тегів та кнопка додавання зображення заставки.

7. NotFound – сторінка, що відображається, якщо ресурс по шляху не знайдено.

Основні сторінки зображені на рис. 3.9-3.13. Код розмітки головної сторінки наведено в додатку В2.

Сторінки використовують прив’язку моделі [24] та самі класи моделей, які відповідають формі DTO на back-end (рис. 3.14).

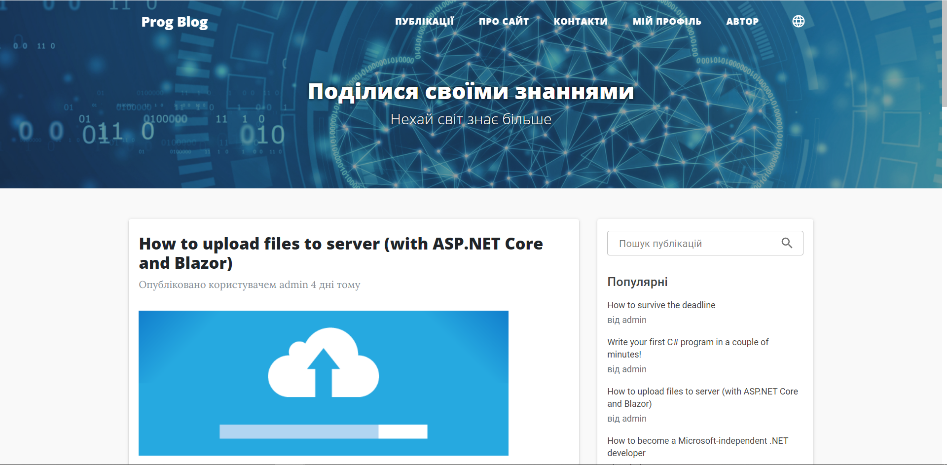


Рисунок 3.9 – Головна сторінка

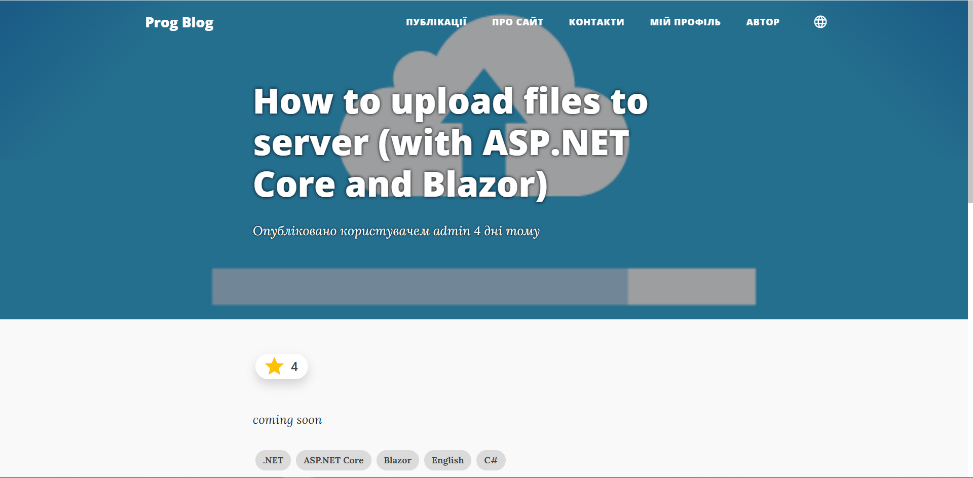


Рисунок 3.10 – Сторінка публікації

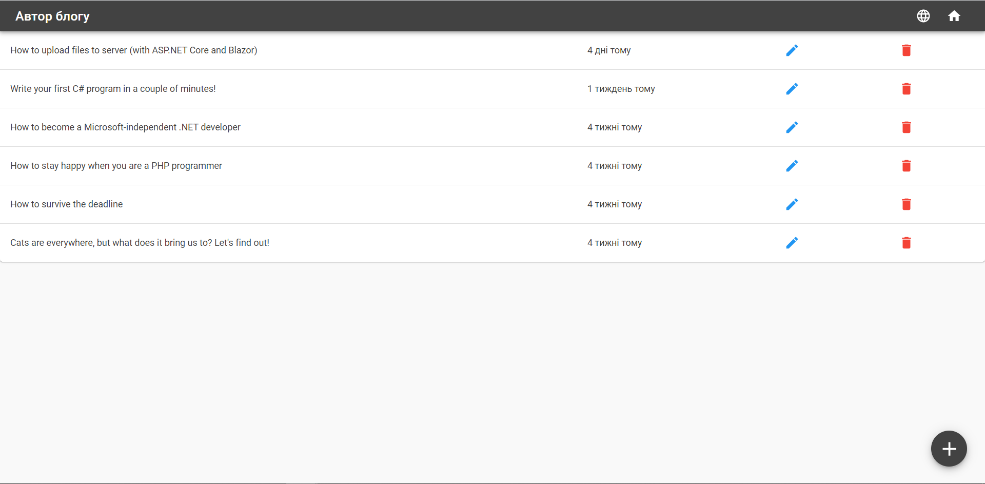


Рисунок 3.11 – Сторінка автора

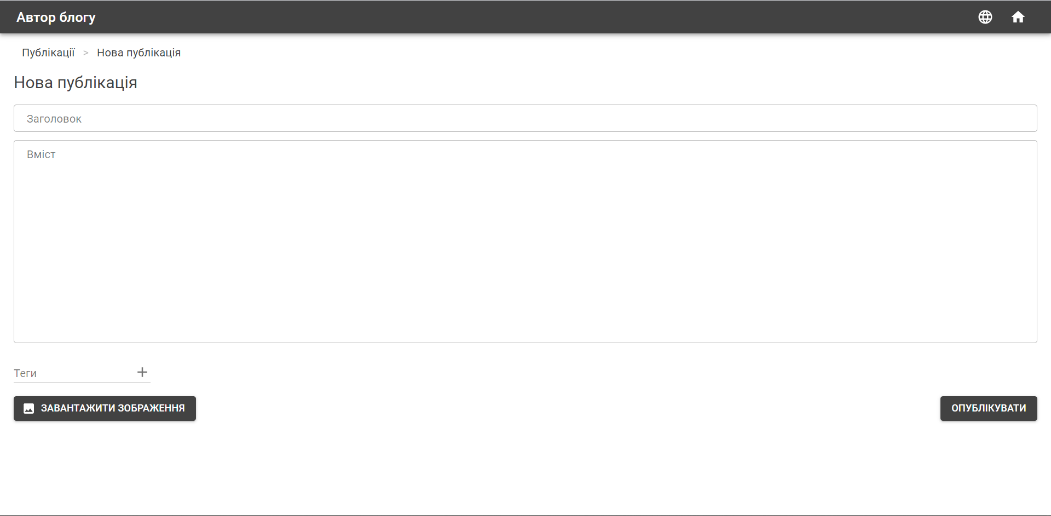


Рисунок 3.12 – Сторінка додавання публікації



Рисунок 3.13 – Сторінка входу

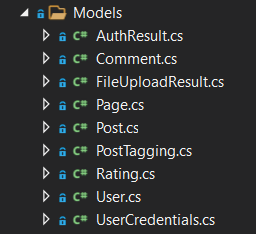


Рисунок 3.14 – Наявні класи моделей

## Формування запитів до back-end

Для отримання даних з Web API використовується набір сервісів, що є типізованими HTTP-клієнтами. Сервіси викликаються на сторінках додатку за допомогою dependency injection. Метод для їх реєстрації наведено нижче.

internal static IServiceCollection AddApiServices(this IServiceCollection services, string apiUrl)

{

void SetBaseApiRequestAddress(HttpClient client, string endpoint)

{

client.BaseAddress = new Uri($"{apiUrl}/{endpoint}");

}

services.AddTransient<IApiClient, ApiClient>();

services.AddHttpClient<IPostService, PostService>(

client => SetBaseApiRequestAddress(client, "posts")

);

services.AddHttpClient<ICommentService, CommentService>(

client => SetBaseApiRequestAddress(client, "comments")

);

services.AddHttpClient<IRatingService, RatingService>(

client => SetBaseApiRequestAddress(client, "ratings")

);

services.AddHttpClient<ITagService, TagService>(

client => SetBaseApiRequestAddress(client, "tags")

);

services.AddHttpClient<IAuthService, AuthService>(

client => SetBaseApiRequestAddress(client, "users")

);

services.AddHttpClient<IFileService, FileService>(

client => SetBaseApiRequestAddress(client, "files")

);

return services;

}

Варто приділити увагу допоміжному класові ApiClient. У ньому є набір загальних методів для надсилання HTTP-запитів, які використовуються всіма сервісами. Крім того, він працює з local storage, дістаючи збережений JWT-токен, та заповнює ним HTTP-заголовок Authorization для надсилання авторизованого запиту до back-end. Код класу наведено в додатку В3.

Методи звичайних сервісів являють собою виклик методів ApiClient. Відрізняються методи FileService:

public string GetImageUrl(string fileName)

{

if (Uri.IsWellFormedUriString(fileName, UriKind.Absolute))

return fileName;

string fileUrl = \_apiClient.HttpClient.BaseAddress.AbsoluteUri;

return $"{fileUrl}/{fileName}";

}

public async Task<string> PublishFile(IBrowserFile file)

{

using MultipartFormDataContent formContent = new();

using StreamContent streamContent = new(file.OpenReadStream(MaxFileSize));

formContent.Add(streamContent, "\"files\"", file.Name);

await \_apiClient.EnsureAuthorizationHeader();

HttpResponseMessage response = await \_apiClient.HttpClient.PostAsync("", formContent);

using Stream contentStream = await response.Content.ReadAsStreamAsync();

var result = await JsonSerializer.DeserializeAsync<FileUploadResult>(contentStream, options: new()

{

PropertyNameCaseInsensitive = true

});

return result.LocalPath;

}

Замість надсилання StringContent у вигляді JSON-серіалізованого об’єкта, обраний файл додається до даних форми як потоковий вміст. Результатом виконання методу після десеріалізації результату буде локальний шлях файлу на сервері. При передачі його в перший наведений метод буде сформовано URL-адресу, при зверненні до якої Web API поверне файловий вміст.

## Авторизація

### Формування авторизаційного запиту до back-end

За авторизацію користувача відповідає сервіс AuthService. Він використовує допоміжні методи класу ApiClient для надсилання POST-запитів з даними реєстрації та входу, очікуючи як результат або токен, або помилки авторизації. При вході в систему у компоненті Login викликається метод входу AuthService, що дозволяє встановити токен у local storage, щоб користувач вважався автентифікованим:

public async Task SubmitCredentials()

{

AuthResult result = await \_authService.LoginAsync(enteredCredentials);

if (result.Token is not null && result.Errors is null)

{

var provider = \_authStateProvider as JwtAuthenticationStateProvider;

await provider.AuthenticateUserAsync(result.Token);

\_navigationManager.NavigateTo("/");

return;

}

foreach (string errorMessage in result.Errors)

\_snackbar.Add(errorMessage, Severity.Error);

}

### Робота з JWT-токеном

З наведеного вище методу видно, що для операцій з токеном використовується підстановка власного провайдера стану автентифікації JwtAuthenticationStateProvider замість стандартного. Клас містить стандартний метод для повернення стану автентифікації, логіка якого змінена так, щоб цей стан формувався із тверджень (claims), які можна отримати зі збереженого JWT-токена. Також він містить методи для збереження та видалення токена з local storage. Код класу надано в додатку В4.

Як результат, всі компоненти AuthorizeView отримують дані про користувача з токена і відображають відповідний вміст. Також всі сторінки автора та профілю користувача захищені атрибутом Authorize, що не допускає переходу для неавторизованих користувачів та тих, хто не має відповідної ролі.

### Надсилання запиту із JWT-токеном

Як було згадано у підрозділі 3.4, додаванням JWT-токена до запиту займається ApiClient. Його метод EnsureAuthorizationHeader додає актуальний токен до заголовка авторизації.

# ВИСНОВКИ ДО РОЗДІЛУ

Описано стек технологій для front-end, обґрунтовано використання фреймворку Blazor. Наведено допоміжні бібліотеки, що використовуються в проекті.

Досліджено застосовані компоненти, як з бібліотеки MudBlazor, так і користувацькі. Надано опис та вигляд сторінок графічного інтерфейсу, їх призначення, перелік прив’язаних моделей.

Описано процес формування запитів до back-end, набір сервісів, що їх здійснюють, допоміжний клас ApiClient. Розглянуто алгоритм відправки файлів (зображень) до Web API та їх отримання за посиланням. Пояснено механізм авторизації в застосунку.

Як результат, частина запланованих вимог до front-end залишилася невиконаною, проте закладено надійну основу, яка дозволяє продовжувати розробку та розвивати проект у майбутньому.

# ВИСНОВКИ

В ході обчислювальної практики спроектовано та розроблено вебдодаток колективного блогу на IT-тематику, вихідний код якого знаходиться за посиланням: [https://github.com/AndrewTheM/BlogPlatform/tree/entity](https://github.com/AndrewTheM/BlogPlatform/tree/entity/).

Він складається з back-end та front-end частин. Розробка відбувалася на стеку технологій платформи .NET, а саме: MS SQL Server, ASP.NET Core, ADO.NET (Dapper), Entity Framework Core, Blazor, ряд допоміжних бібліотек.

Сформовані функціональні вимоги було виконано в достатньому обсязі, зокрема розроблено головну сторінку сайту зі списком публікацій і можливістю фільтрування та пошуку; сторінку публікації з форматованим вмістом, оцінками та коментарями; інструменти автора блогу для роботи з публікаціями; форми входу та реєстрації користувача; відповідні програмні компоненти серверної частини, що забезпечують механізми роботи з даними, реєстрації, автентифікації, авторизації, глобалізації та локалізації.

При цьому є перспективи для розвитку продукту у вигляді реалізації решти поставлених вимог. Серед них публічний профіль користувача та його налаштування, панель адміністратора з обширним інструментарієм, відповіді до коментарів та оцінювання тощо. Інтеграція з популярними онлайн-сервісами також буде доречною.

Розроблений додаток може використовуватися як платформа для колективного ІТ-блогу, що в подальшому стане базою для створення ІТ-спільноти, об’єднаної задля взаємного обміну знаннями та вирішенням проблем професійної діяльності. В цьому полягала мета роботи, і, таким чином, її можна вважати досягнутою.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

* 1. Booking- Вікіпедія [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Booking.com
  2. Користувацький досвід [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%94%D0%BE%D1%81%D0%B2%D1%96%D0%B4\_%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F
  3. Airbnb [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Airbnb
  4. What? (Що?) User Story [Електронний ресурс] – Режим доступу до ресурсу: <https://www.quality-assurance-group.com/user-stories>.
  5. Довідник із Transact-SQL - SQL Server | Microsoft Docs [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.microsoft.com/ru-ru/sql/t-sql/language-reference?view=sql-server-ver15>.
  6. Порівняльний аналіз деяких видів архітектури програмного забезпечення / Блог компанії OTUS / Хабр [Електронний ресурс] – Режим доступу до ресурсу: [https://habr.com/ru/company/otus/blog/476024](https://habr.com/ru/company/otus/blog/476024/).
  7. Клапчук Р. Г. Монолітні веб-сервіси та мікросервіси: порівняння та вибір / Р. Г. Клапчук, В. С. Харченко. // Радіоелектронні і комп'ютерні системи. – 2017. – №1. – С. 51–56.
  8. Багатошарова архітектура [Електронний ресурс] – Режим доступу до ресурсу: <https://eopearhiiv.edu.ee/e-kursused/eucip/arendus_vk/1631_.html>.
  9. Why .Net is the most used platform for Enterprise Application Development [Електронний ресурс] – Режим доступу до ресурсу: <https://www.janbask.com/blog/why-net-is-the-most-used-platform-for-enterprise-application-development>.
  10. IoC Контейнери [Електронний ресурс] – Режим доступу до ресурсу: <https://www.tutorialsteacher.com/ioc/ioc-container>.
  11. Об'єктно-реляційне відображення - Вікіпедія [Електронний ресурс] – Режим доступу до ресурсу: [https://uk.wikipedia.org/wiki/Об%27єктно-реляційне\_відображення](https://uk.wikipedia.org/wiki/%D0%9E%D0%B1%27%D1%94%D0%BA%D1%82%D0%BD%D0%BE-%D1%80%D0%B5%D0%BB%D1%8F%D1%86%D1%96%D0%B9%D0%BD%D0%B5_%D0%B2%D1%96%D0%B4%D0%BE%D0%B1%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F).
  12. MIME тип - HTTP | MDN [Електронний ресурс] – Режим доступу до ресурсу: <https://developer.mozilla.org/ru/docs/Web/HTTP/Basics_of_HTTP/MIME_types>.
  13. Вступ до ADO.NET [Електронний ресурс] – Режим доступу до ресурсу: <https://metanit.com/sharp/adonet/1.1.php>.
  14. Dapper Tutorial | Dapper Tutorial and Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://dapper-tutorial.net>.
  15. Welcome To Learn Dapper | Learn Dapper [Електронний ресурс] – Режим доступу до ресурсу: <https://www.learndapper.com>.
  16. Короткий вступ до ін’єкції залежності: що це таке, і коли її використовувати [Електронний ресурс] – Режим доступу до ресурсу: <https://hackit-ukraine.com/26-a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it>.
  17. JSON Web Token - Вікіпедія [Електронний ресурс] – Режим доступу до ресурсу: <https://uk.wikipedia.org/wiki/JSON_Web_Token>.
  18. Принципи SOLID. Принцип єдиного обов’язку [Електронний ресурс] – Режим доступу до ресурсу: <https://stackify.com/solid-design-principles>.
  19. Що таке Blazor? - Blazor University [Електронний ресурс] – Режим доступу до ресурсу: <https://blazor-university.com/overview/what-is-blazor>.
  20. MudBlazor - Бібліотека компонентів Blazor [Електронний ресурс] – Режим доступу до ресурсу: <https://mudblazor.com>.
  21. Markdown - Вікіпедія [Електронний ресурс] – Режим доступу до ресурсу: <https://uk.wikipedia.org/wiki/Markdown>.
  22. Розмітка блогу «Clean Blog» від Start Bootstrap [Електронний ресурс] – Режим доступу до ресурсу: [https://github.com/StartBootstrap/startbootstrap-clean-blog](https://github.com/StartBootstrap/startbootstrap-clean-blog/).
  23. Вступ до Material Design - Material.io [Електронний ресурс] – Режим доступу до ресурсу: <https://material.io/design/introduction>.
  24. Прив'язка до даних в ASP.NET Core Blazor | Microsoft Docs [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.microsoft.com/ru-ru/aspnet/core/blazor/components/data-binding?view=aspnetcore-5.0>.

# ДОДАТКИ

## Додаток А1. Запит на створення бази даних MS SQL Server

USE master

GO

CREATE DATABASE BlogPlatformInitial

GO

USE BlogPlatformInitial

GO

CREATE TABLE Users

(

    Id int NOT NULL IDENTITY,

    Username nvarchar(100) NOT NULL,

    IsBlogAuthor bit DEFAULT 0,

    PRIMARY KEY (Id),

    UNIQUE (Username)

);

GO

CREATE TABLE Posts

(

    Id int NOT NULL IDENTITY,

    AuthorId int,

    Title nvarchar(100) NOT NULL,

    DateTimePublished datetime NOT NULL

        DEFAULT CURRENT\_TIMESTAMP,

    ThumbnailPath nvarchar(100) NOT NULL,

    PRIMARY KEY (Id),

    FOREIGN KEY (AuthorId)

        REFERENCES Users(Id)

        ON DELETE SET NULL

);

GO

CREATE TABLE PostContents

(

    Id int NOT NULL,

    Content nvarchar(max) NOT NULL,

    PRIMARY KEY (Id),

    FOREIGN KEY (Id)

        REFERENCES Posts(Id)

        ON DELETE CASCADE

);

GO

CREATE TABLE Ratings

(

    Id int NOT NULL IDENTITY,

    PostId int NOT NULL,

    UserId int,

    RatingValue int NOT NULL,

    PRIMARY KEY (Id),

    FOREIGN KEY (PostId)

        REFERENCES Posts(Id)

        ON DELETE CASCADE,

    FOREIGN KEY (UserId)

        REFERENCES Users(Id)

        ON DELETE SET NULL

);

GO

CREATE TABLE Comments

(

    Id int NOT NULL IDENTITY,

    PostId int NOT NULL,

    AuthorId int,

    Content nvarchar(500) NOT NULL,

    DateTimeCreated datetime NOT NULL

        DEFAULT CURRENT\_TIMESTAMP,

    IsEdited bit DEFAULT 0,

    IsDeleted bit DEFAULT 0,

**Продовження додатку А1. Запит на створення бази даних MS SQL Server**

    UpvoteCount int DEFAULT 0,

    PRIMARY KEY (Id),

    FOREIGN KEY (PostId)

        REFERENCES Posts(Id)

        ON DELETE CASCADE,

    FOREIGN KEY (AuthorId)

        REFERENCES Users(Id)

        ON DELETE SET NULL

);

GO

CREATE TABLE UserPreferences

(

    UserId int NOT NULL,

    EmailOnReply bit DEFAULT 1,

    EmailOnNewPost bit DEFAULT 0,

    PRIMARY KEY (UserId),

    FOREIGN KEY (UserId)

        REFERENCES Users(Id)

        ON DELETE CASCADE

);

GO

CREATE TABLE Tags

(

    Id int NOT NULL IDENTITY,

    TagName nvarchar(50) NOT NULL,

    PRIMARY KEY (Id),

    UNIQUE (TagName)

);

GO

CREATE TABLE TagsOfPosts

(

    PostId int NOT NULL,

    TagId int NOT NULL,

    PRIMARY KEY (PostId, TagId),

    FOREIGN KEY (PostId)

        REFERENCES Posts(Id)

        ON DELETE CASCADE,

    FOREIGN KEY (TagId)

        REFERENCES Tags(Id)

        ON DELETE CASCADE

);

GO

CREATE TABLE VerificationStatuses

(

    Id int NOT NULL IDENTITY,

    StatusName nvarchar(50) NOT NULL,

    PRIMARY KEY (Id),

    UNIQUE (StatusName)

);

GO

CREATE TABLE AuthorVerifications

(

    Id int NOT NULL IDENTITY,

    UserId int NOT NULL,

    PromptText nvarchar(500),

    Response nvarchar(500),

    VerificationStatusId int,

    DateTimeRequested datetime NOT NULL

        DEFAULT CURRENT\_TIMESTAMP,

    PRIMARY KEY (Id),

    FOREIGN KEY (UserId)

        REFERENCES Users(Id)

        ON DELETE CASCADE,

    FOREIGN KEY (VerificationStatusId)

        REFERENCES VerificationStatuses(Id)

        ON DELETE SET NULL

);

## Додаток А2. Збережені процедури

USE BlogPlatformInitial  
GO

CREATE OR ALTER PROCEDURE [dbo].[SP\_GetAllRecordsFromTable]

    @P\_tableName nvarchar(50) = NULL

AS

BEGIN

    SET NOCOUNT ON;

    DECLARE @V\_table nvarchar(50) = NULL;

    IF (@P\_tableName IS NOT NULL)

        SELECT @V\_table = QUOTENAME( TABLE\_NAME )

        FROM INFORMATION\_SCHEMA.TABLES

        WHERE TABLE\_NAME = @P\_tableName;

    DECLARE @V\_sql AS nvarchar(MAX) = NULL;

    IF (@V\_table IS NOT NULL)

        SELECT @V\_sql = 'SELECT \* FROM ' + @V\_table;

    IF (@V\_sql IS NOT NULL)

        EXEC(@V\_sql);

    ELSE

        SELECT -1;

END

CREATE OR ALTER PROCEDURE [dbo].[SP\_GetRecordByIdFromTable]

    @P\_tableName nvarchar(50) = NULL,

    @P\_Id nvarchar(50) = NULL

AS

BEGIN

    SET NOCOUNT ON;

    DECLARE @V\_table nvarchar(50) = NULL;

    IF (@P\_tableName IS NOT NULL)

        SELECT @V\_table = QUOTENAME( TABLE\_NAME )

        FROM INFORMATION\_SCHEMA.TABLES

        WHERE TABLE\_NAME = @P\_tableName;

    DECLARE @V\_sql AS nvarchar(MAX) = NULL;

    IF (@V\_table IS NOT NULL AND @P\_Id IS NOT NULL)

        SELECT @V\_sql = 'SELECT \* FROM ' + @V\_table + ' WHERE Id = ' + @P\_Id;

    IF (@V\_sql IS NOT NULL)

        EXEC(@V\_sql);

    ELSE

        SELECT -1;

END

CREATE OR ALTER PROCEDURE [dbo].[SP\_InsertRecordToTable]

    @P\_tableName nvarchar(50) = NULL,

    @P\_columnsString nvarchar(MAX) = NULL,

    @P\_propertiesString nvarchar(MAX) = NULL

AS

BEGIN

    SET NOCOUNT ON;

    DECLARE @V\_table nvarchar(50) = NULL;

    IF (@P\_tableName IS NOT NULL)

        SELECT @V\_table = QUOTENAME( TABLE\_NAME )

        FROM INFORMATION\_SCHEMA.TABLES

        WHERE TABLE\_NAME = @P\_tableName;

    DECLARE @V\_sql as nvarchar(MAX) = NULL;

    IF (@V\_table IS NOT NULL AND @P\_columnsString IS NOT NULL AND @P\_propertiesString IS NOT NULL)

        SET @V\_sql = 'INSERT INTO ' + @V\_table + ' (' + @P\_columnsString + ') OUTPUT INSERTED.ID VALUES (' + @P\_propertiesString + ')';

    IF (@V\_sql IS NOT NULL)

        SELECT @V\_sql;

    ELSE

        SELECT -1;

END

**Продовження додатку А2. Збережені процедури**

CREATE OR ALTER PROCEDURE [dbo].[SP\_UpdateRecordInTable]

    @P\_tableName nvarchar(50) = NULL,

    @P\_columnsString nvarchar(MAX) = NULL,

    @P\_Id nvarchar(50) = NULL

AS

BEGIN

    SET NOCOUNT ON;

    DECLARE @V\_table nvarchar(50) = NULL;

    IF (@P\_tableName IS NOT NULL)

        select @V\_table = QUOTENAME( TABLE\_NAME )

        FROM INFORMATION\_SCHEMA.TABLES

        WHERE TABLE\_NAME = @P\_tableName;

    DECLARE @V\_sql AS nvarchar(MAX) = NULL;

    IF (@V\_table IS NOT NULL AND @P\_columnsString IS NOT NULL AND @P\_Id IS NOT NULL)

        SET @V\_sql = 'UPDATE ' + @V\_table + ' SET ' + @P\_columnsString + ' WHERE Id = ' + @P\_Id;

    IF (@V\_sql IS NOT NULL)

        SELECT @V\_sql;

    ELSE

        SELECT -1;

END

CREATE OR ALTER PROCEDURE [dbo].[SP\_DeleteRecordFromTable]

    @P\_tableName nvarchar(50) = NULL,

    @P\_Id nvarchar(50) = NULL

AS

BEGIN

    SET NOCOUNT ON;

    DECLARE @V\_table nvarchar(50) = NULL;

    IF (@P\_tableName IS NOT NULL)

        SELECT @V\_table = QUOTENAME( TABLE\_NAME )

        FROM INFORMATION\_SCHEMA.TABLES

        WHERE TABLE\_NAME = @P\_tableName;

    DECLARE @V\_sql AS nvarchar(MAX) = NULL;

    IF (@V\_table IS NOT NULL and @P\_Id IS NOT NULL)

        SELECT @V\_sql = 'DELETE FROM ' + @V\_table + 'WHERE Id = ' + @P\_Id;

    IF (@V\_sql IS NOT NULL)

        EXEC(@V\_sql);

    ELSE

        SELECT 0;

END

## Додаток Б1. Контекст даних

public class BlogContext : IdentityDbContext<IdentityUser>

{

private readonly AdminUserOptions \_adminOptions;

public DbSet<Post> Posts { get; set; }

public DbSet<PostContent> PostContents { get; set; }

public DbSet<Comment> Comments { get; set; }

public DbSet<Tag> Tags { get; set; }

public DbSet<Rating> Ratings { get; set; }

public DbSet<AuthorVerification> AuthorVerifications { get; set; }

public DbSet<VerificationStatus> VerificationStatuses { get; set; }

public DbSet<UserProfile> UserProfiles { get; set; }

public UserManager<User> UserManager { get; set; }

public RoleManager<IdentityRole> RoleManager { get; set; }

public SignInManager<User> SignInManager { get; set; }

public BlogContext()

{

}

public BlogContext(IOptions<AdminUserOptions> adminOptions,

DbContextOptions contextOptions)

: base(contextOptions)

{

\_adminOptions = adminOptions?.Value;

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.ApplyConfigurationsFromAssembly(this.GetType().Assembly);

modelBuilder.Entity<IdentityRole>(role =>

{

role.HasData(

new() { Id = "9341400c-1856-46d2-a17f-841be8d59d59", Name = "Reader", NormalizedName = "READER" },

new() { Id = "0428f7f5-3ddd-4892-bfaf-adb1bf0ed1c6", Name = "Author", NormalizedName = "AUTHOR" },

new() { Id = "2186fb46-ec78-4ff3-87d5-63196ee65bdf", Name = "Admin", NormalizedName = "ADMIN" }

);

});

if (\_adminOptions is null)

return;

modelBuilder.Entity<User>(user =>

{

PasswordHasher<User> hasher = new();

user.HasData(

new User

{

Id = \_adminOptions.Id,

UserName = \_adminOptions.Username,

NormalizedUserName = \_adminOptions.Username.ToUpper(),

PasswordHash = hasher.HashPassword(null, \_adminOptions.Password),

Email = \_adminOptions.Email,

NormalizedEmail = \_adminOptions.Email.ToUpper()

}

);

});

**Продовження додатку Б1. Контекст даних**

modelBuilder.Entity<IdentityUserRole<string>>(ur =>

{

ur.HasData(

new IdentityUserRole<string>

{

UserId = \_adminOptions.Id,

RoleId = "2186fb46-ec78-4ff3-87d5-63196ee65bdf"

}

);

});

}

public override async Task<int> SaveChangesAsync(CancellationToken cancellationToken = default)

{

ChangeTracker.DetectChanges();

var entities = ChangeTracker.Entries()

.Where(entry => entry.State == EntityState.Modified)

.Select(entry => entry.Entity)

.OfType<EntityBase<int>>();

foreach (var entity in entities)

entity.UpdatedOn = DateTime.Now;

return await base.SaveChangesAsync(cancellationToken);

}

}

## Додаток Б2. Generic Repository

public class EntityRepository<TEntity, TId> : IRepository<TEntity, TId>

where TEntity : EntityBase<TId>

{

protected readonly BlogContext \_context;

protected readonly DbSet<TEntity> \_set;

public EntityRepository(BlogContext context)

{

\_context = context;

\_set = context.Set<TEntity>();

}

public Task<IQueryable<TEntity>> GetAllAsync()

{

var records = \_set.AsNoTracking();

return Task.FromResult(records);

}

public async Task<TEntity> GetByIdAsync(TId id)

{

return await EnsureEntityResultAsync(() =>

{

return \_set.FindAsync(id).AsTask();

});

}

public async Task CreateAsync(TEntity entity)

{

await \_set.AddAsync(entity);

}

public async Task DeleteAsync(TId id)

{

var entity = await GetByIdAsync(id);

\_set.Remove(entity);

}

/// <summary>

/// Accepts a delegate that retrieves an entity asynchronously.

/// Returns the execution result of the delegate.

/// Throws <see cref="EntityNotFoundException" />

/// if the result is null or the operation is invalid.

/// </summary>

protected async Task<TEntity> EnsureEntityResultAsync(Func<Task<TEntity>> entityDelegate)

{

try

{

return await entityDelegate()

?? throw new EntityNotFoundException();

}

catch (InvalidOperationException ex)

{

throw new EntityNotFoundException("An exception occured while retrieving the entity.", ex);

}

}

}

## Додаток Б3. Unit of Work

public class UnitOfWork : IBloggingUnitOfWork

{

private bool \_isDisposed;

private readonly BlogContext \_context;

public IPostRepository Posts { get; }

public IPostContentRepository PostContents { get; }

public ICommentRepository Comments { get; }

public ITagRepository Tags { get; }

public IRatingRepository Ratings { get; }

public IVerificationRepository AuthorVerifications { get; }

public IVerificationStatusRepository VerificationStatuses { get; }

public IProfileRepository UserProfiles { get; }

public UserManager<User> UserManager { get; }

public RoleManager<IdentityRole> RoleManager { get; }

public SignInManager<User> SignInManager { get; }

public UnitOfWork(BlogContext context,

IPostRepository posts,

IPostContentRepository postContents,

ICommentRepository comments,

ITagRepository tags,

IRatingRepository ratings,

IVerificationRepository authorVerifications,

IVerificationStatusRepository verificationStatuses,

IProfileRepository userProfiles,

UserManager<User> userManager,

RoleManager<IdentityRole> roleManager,

SignInManager<User> signInManager)

{

\_context = context;

Posts = posts;

PostContents = postContents;

Comments = comments;

Tags = tags;

Ratings = ratings;

AuthorVerifications = authorVerifications;

VerificationStatuses = verificationStatuses;

UserProfiles = userProfiles;

UserManager = userManager;

RoleManager = roleManager;

SignInManager = signInManager;

}

public async Task CommitAsync()

{

await \_context.SaveChangesAsync();

}

public void Dispose()

{

Dispose(disposing: true);

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (!\_isDisposed)

{

if (disposing)

{

\_context.Dispose();

}

\_isDisposed = true;

}

}

}

## Додаток Б4. Профіль AutoMapper

public class BlogMappingProfile : Profile

{

public BlogMappingProfile()

{

ConfigurePost();

ConfigureComment();

ConfigureRating();

ConfigureTag();

ConfigureUser();

}

private static void SkipNullValues<TSource, TDestination, TMember>(

IMemberConfigurationExpression<TSource, TDestination, TMember> opt)

{

opt.Condition((src, dest, prop) => prop is not null);

}

private void ConfigurePost()

{

CreateMap<PostRequest, Post>()

.AfterMap((req, p) =>

{

if (p.ContentEntity is not null && req.Content is not null)

p.ContentEntity.Content = req.Content;

})

.ForAllMembers(SkipNullValues);

CreateMap<PostRequest, PostContent>()

.ForAllMembers(SkipNullValues);

CreateMap<Post, PostResponse>()

.ForMember(res => res.PublishedOn, opt => opt.MapFrom(p => p.CreatedOn))

.ForMember(res => res.IsEdited, opt => opt.MapFrom(p => p.UpdatedOn > p.CreatedOn));

CreateMap<Post, CompletePostResponse>()

.IncludeBase<Post, PostResponse>()

.ForMember(res => res.Content, opt => opt.MapFrom(p => p.ContentEntity.Content))

.ForMember(res => res.Author, opt => opt.MapFrom(p => p.Author.UserName))

.ForMember(res => res.Tags, opt => opt.MapFrom(p => p.Tags.Select(t => t.TagName)));

}

private void ConfigureComment()

{

CreateMap<CommentRequest, Comment>()

.ForMember(c => c.PostId, opt => opt.Condition(req => req.PostId > 0))

.ForAllOtherMembers(SkipNullValues);

CreateMap<Comment, CommentResponse>()

.ForMember(res => res.Author, opt => opt.MapFrom(c => c.Author.UserName))

.ForMember(res => res.PublishedOn, opt => opt.MapFrom(c => c.CreatedOn))

.ForMember(res => res.IsEdited, opt => opt.MapFrom(c => c.UpdatedOn > c.CreatedOn));

}

private void ConfigureRating()

{

CreateMap<RatingRequest, Rating>()

.ForMember(r => r.PostId, opt => opt.Condition(req => req.PostId > 0))

.ForAllOtherMembers(SkipNullValues);

CreateMap<Rating, RatingResponse>()

.ForMember(res => res.User, opt => opt.MapFrom(r => r.User.UserName));

}

private void ConfigureTag()

{

CreateMap<TagRequest, Tag>();

CreateMap<Tag, TagResponse>();

}

private void ConfigureUser()

{

CreateMap<UserRequest, User>()

.ForMember(u => u.UserName, opt => opt.MapFrom(req => req.Username));

}

}

## Додаток Б5. Сервіс Identity

public class IdentityService : IIdentityService

{

private const int \_daysToExpire = 7;

private readonly IBloggingUnitOfWork \_unitOfWork;

private readonly IMapper \_mapper;

private readonly IConfiguration \_configuration;

private readonly IStringLocalizer<IdentityService> \_locale;

public IdentityService(IBloggingUnitOfWork unitOfWork,

IMapper mapper,

IConfiguration configuration,

IStringLocalizer<IdentityService> locale)

{

\_unitOfWork = unitOfWork;

\_mapper = mapper;

\_configuration = configuration;

\_locale = locale;

}

public async Task<AuthResponse> RegisterAsync(UserRequest userDto)

{

AuthResponse response = new();

User userToRegister = \_mapper.Map<User>(userDto);

var createResult = await \_unitOfWork.UserManager.CreateAsync(userToRegister, userDto.Password);

if (!createResult.Succeeded)

{

response.Errors = createResult.Errors.Select(er => er.Description).ToArray();

return response;

}

await \_unitOfWork.UserManager.AddToRoleAsync(userToRegister, "Reader");

response.Token = await GenerateJwtToken(userToRegister);

return response;

}

public async Task<AuthResponse> LoginAsync(UserCredentialsRequest credentials)

{

AuthResponse response = new();

User existingUser = await \_unitOfWork.UserManager.FindByNameAsync(credentials.Username);

if (existingUser is null)

{

response.Errors = new[] { \_locale["UserDoesNotExist"].Value };

return response;

}

var signInResult = await \_unitOfWork.SignInManager.PasswordSignInAsync(credentials.Username,credentials.Password, isPersistent: true,lockoutOnFailure: true);

if (!signInResult.Succeeded)

{

string message = signInResult.IsLockedOut

? \_locale["UserLockedOut"].Value

: \_locale["IncorrectPassword"].Value;

response.Errors = new[] { message };

return response;

}

response.Token = await GenerateJwtToken(existingUser);

return response;

}

public async Task<AuthResponse> ChangePasswordAsync(UserCredentialsRequest credentials)

{

AuthResponse response = new();

User existingUser = await \_unitOfWork.UserManager.FindByNameAsync(credentials.Username);

**Продовження додатку Б5. Сервіс Identity**

if (existingUser is null)

{

response.Errors = new[] { \_locale["UserDoesNotExist"].Value };

return response;

}

string resetToken = await \_unitOfWork.UserManager.GeneratePasswordResetTokenAsync(existingUser);

var resetResult = await \_unitOfWork.UserManager.ResetPasswordAsync(existingUser,

resetToken,

credentials.Password);

if (!resetResult.Succeeded)

{

response.Errors = resetResult.Errors.Select(er => er.Description).ToArray();

}

return response;

}

public async Task<bool> CheckIfUserHasName(string userId, string username)

{

User user = await \_unitOfWork.UserManager.FindByIdAsync(userId);

return user?.UserName == username;

}

private async Task<string> GenerateJwtToken(User user)

{

DateTime expirationTime = DateTime.UtcNow.AddDays(\_daysToExpire);

string secret = \_configuration.GetValue<string>("TokenSecret");

var keyBytes = Encoding.UTF8.GetBytes(secret);

var userRoles = await \_unitOfWork.UserManager.GetRolesAsync(user);

List<Claim> claims = new()

{

new Claim("id", user.Id),

new Claim(ClaimTypes.Name, user.UserName),

new Claim(ClaimTypes.Email, user.Email),

new Claim(ClaimTypes.Role, string.Join(", ", userRoles))

};

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new(claims),

Expires = expirationTime,

SigningCredentials = new(new SymmetricSecurityKey(keyBytes),

SecurityAlgorithms.HmacSha256Signature)

};

var tokenHandler = new JwtSecurityTokenHandler();

var token = tokenHandler.CreateToken(tokenDescriptor);

return tokenHandler.WriteToken(token);

}

}

## Додаток В1. Графічні компоненти

**AuthorBar**

@inject IStringLocalizer<AuthorBar> \_locale

<MudAppBar Color="Color.Dark" Dense="true" Fixed="false">

<MudText Typo="Typo.h6">@\_locale["BlogAuthor"]</MudText>

<MudAppBarSpacer />

<LanguageMenu IconColor="Color.Tertiary" />

<MudTooltip Text="@\_locale["HomePage"]">

<MudIconButton Color="Color.Inherit" Icon="@Icons.Material.Filled.Home" Link="/" />

</MudTooltip>

</MudAppBar>

**Loading**

<style>

html, body {

height: 100%;

}

</style>

<div class="d-flex justify-content-center align-items-center w-100 h-100">

<MudProgressCircular Size="Size.Large" Indeterminate="true" />

</div>

**Footer**

<footer class="container">

<div class="row">

<div class="col-lg-8 col-md-10 mx-auto">

<p class="copyright text-muted">

Copyright © Prog Blog, 2021

<br>

Template by

<a href="https://github.com/StartBootstrap/startbootstrap-clean-blog" target="\_blank">

Start Bootstrap

</a>

</p>

</div>

</div>

</footer>

## Додаток В2. Головна сторінка графічного інтерфейсу

@page "/"

@using BlogPlatform.UI.Models

@inject IPostService \_postService

@inject IFileService \_fileService

@inject ITagService \_tagService

@inject IStringLocalizer<Index> \_locale

<NavMenu />

<header class="masthead">

<div class="overlay"></div>

<div class="container">

<div class="row">

<div class="col-lg-8 col-md-10 mx-auto">

<div class="site-heading">

<h1>@\_locale["Motto"]</h1>

<span class="subheading">

@\_locale["MottoSub"]

</span>

</div>

</div>

</div>

</div>

</header>

<main class="content">

<div class="container">

**Продовження додатку В2. Головна сторінка графічного інтерфейсу**

@if (currentPage is not null)

{

<div class="row">

<div class="col-sm" id="posts">

@foreach (Models.Post post in currentPage.Data)

{

<MudPaper Class="post-preview mb-4 pa-4" Elevation="2">

<NavLink href="@($"post/{post.TitleIdentifier}")">

<h2 class="post-title my-2">

@post.Title

</h2>

</NavLink>

<p class="post-meta">

@\_locale["PostedBy"] @post.Author @post.RelativePublishTime

</p>

<img class="img-fluid" src="@\_fileService.GetImageUrl(post.ThumbnailPath)">

</MudPaper>

}

@if (currentPage.TotalPages > 1)

{

<div class="mt-5">

<ul class="pagination justify-content-center">

<li class="page-item @(currentPage.PreviousPage is null ? "disabled" : "")">

<a class="page-link" href="#posts" @onclick="GoToPreviousPage">

@\_locale["PreviousPage"]

</a>

</li>

@for (int i = 1; i <= currentPage.TotalPages; ++i)

{

int j = i;

<li class="page-item @(currentPage.PageNumber == j ? "active" : "")">

<a class="page-link" href="#posts" @onclick="async () => await GoToPage(j)">@i</a>

</li>

}

<li class="page-item @(currentPage.NextPage is null ? "disabled" : "")">

<a class="page-link" href="#posts" @onclick="GoToNextPage">

@\_locale["NextPage"]

</a>

</li>

</ul>

</div>

}

</div>

<div class="col-lg-4 mt-md-0 mt-5">

<MudPaper Class="pa-4" Elevation="2">

<MudTextFieldString @bind-Value="searchQuery" OnKeyUp="DetectEnterOnSearch" OnAdornmentClick="SearchPosts" Label="@\_locale["SearchPosts"]" Variant="Variant.Outlined" Margin="Margin.Dense" Adornment="Adornment.End" AdornmentIcon="@Icons.Filled.Search" />

<p class="h5 mb-3">@\_locale["Trending"]</p>

@foreach (var post in trendingPosts)

{

<div class="my-4">

<NavLink href="@($"/post/{post.TitleIdentifier}")">

@post.Title

</NavLink>

<div class="mt-1 text-muted">

@\_locale["TrendingBy"] @post.Author

</div>

</div>

}

<p class="h5 mb-3">@\_locale["Tags"]</p>

<div class="d-flex flex-wrap font-weight-bold">

<MudChipSet SelectedChip="selectedTagChip" SelectedChipChanged="SearchByTagChip">

@foreach (string tag in tags)

{

<MudChip Text="@tag" />

}

</MudChipSet>

</div>

</MudPaper>

</div>

</div>

}

</div>

**Продовження додатку В2. Головна сторінка графічного інтерфейсу**

</main>

<Footer />

@code {

Page<Models.Post> currentPage;

string searchQuery;

MudChip selectedTagChip;

IEnumerable<Models.Post> trendingPosts;

IEnumerable<string> tags;

protected override async Task OnInitializedAsync()

{

await LoadPosts();

trendingPosts = await \_postService.GetTrendingPostsAsync();

tags = await \_tagService.GetTagNames();

}

async Task LoadPosts(int pageNumber = 1, int pageSize = 5)

{

currentPage = await \_postService.GetPostsAsync(pageNumber,pageSize,title: searchQuery,

tag: selectedTagChip?.Text);

}

async Task LoadPosts(string pageUrl)

{

currentPage = await \_postService.GetPostsAsync(pageUrl);

}

async Task GoToPreviousPage()

{

if (currentPage?.PreviousPage is null)

return;

await LoadPosts(currentPage.PreviousPage);

}

async Task GoToPage(int pageNumber)

{

if (currentPage is null || currentPage.PageNumber == pageNumber)

return;

await LoadPosts(pageNumber, currentPage.PageSize);

}

async Task GoToNextPage()

{

if (currentPage?.NextPage is null)

return;

await LoadPosts(currentPage.NextPage);

}

async Task SearchPosts()

{

await LoadPosts();

}

async Task DetectEnterOnSearch(KeyboardEventArgs e)

{

if (e.Code != "Enter" && e.Code != "NumpadEnter")

return;

await LoadPosts();

}

async Task SearchByTagChip(MudChip chip)

{

selectedTagChip = chip;

await LoadPosts();

}

}

## Додаток В3. Допоміжний клієнт HTTP

public class ApiClient : IApiClient

{

private readonly ILocalStorageService \_localStorage;

public HttpClient HttpClient { get; set; }

public ApiClient(ILocalStorageService localStorage)

{

\_localStorage = localStorage;

HttpClient = new();

}

public async Task<T> SendGetApiRequestAsync<T>(string endpoint)

{

await EnsureAuthorizationHeader();

string urlWithCulture = this.AppendCulture(endpoint);

HttpResponseMessage response = await HttpClient.GetAsync(urlWithCulture);

response.EnsureSuccessStatusCode();

using Stream contentStream = await response.Content.ReadAsStreamAsync();

return await JsonSerializer.DeserializeAsync<T>(contentStream, options: new()

{

PropertyNameCaseInsensitive = true

});

}

public async Task<TResponse> SendPostApiRequestWithResultAsync<TRequest, TResponse>(string endpoint, TRequest body, bool ensureSuccess = true)

{

await EnsureAuthorizationHeader();

string urlWithCulture = this.AppendCulture(endpoint);

string bodyJson = JsonSerializer.Serialize(body);

using StringContent jsonContent = new(bodyJson, Encoding.UTF8, "application/json");

HttpResponseMessage response = await HttpClient.PostAsync(urlWithCulture, jsonContent);

if (ensureSuccess)

response.EnsureSuccessStatusCode();

using Stream contentStream = await response.Content.ReadAsStreamAsync();

return await JsonSerializer.DeserializeAsync<TResponse>(contentStream, options: new()

{

PropertyNameCaseInsensitive = true

});

}

public async Task SendModifyingApiRequestAsync<T>(HttpMethod method, string endpoint, T body)

{

HashSet<HttpMethod> allowedMethods = new() { HttpMethod.Post, HttpMethod.Put, HttpMethod.Patch };

if (!allowedMethods.Contains(method))

throw new ArgumentException("HTTP method not allowed.", nameof(method));

await EnsureAuthorizationHeader();

string bodyJson = JsonSerializer.Serialize(body);

using StringContent jsonContent = new(bodyJson, Encoding.UTF8, "application/json");

HttpRequestMessage request = new(method, endpoint) { Content = jsonContent };

HttpResponseMessage response = await HttpClient.SendAsync(request);

response.EnsureSuccessStatusCode();

}

public async Task SendDeleteRequestAsync(string endpoint)

{

await EnsureAuthorizationHeader();

HttpResponseMessage response = await HttpClient.DeleteAsync(endpoint);

response.EnsureSuccessStatusCode();

}

public async Task EnsureAuthorizationHeader()

{

string accessToken = await \_localStorage.GetItemAsync<string>("accessToken");

HttpClient.DefaultRequestHeaders.Authorization = new("Bearer", accessToken);

}

private string AppendCulture(string url)

{

return $"{url}{(url.Contains('?')?'&':'?')}culture={CultureInfo.CurrentUICulture.Name}";

}

}

## Додаток В4. JwtAuthenticationStateProvider

public class JwtAuthenticationStateProvider : AuthenticationStateProvider

{

private const string TokenKey = "accessToken";

private readonly ILocalStorageService \_localStorage;

public JwtAuthenticationStateProvider(ILocalStorageService localStorage)

{

\_localStorage = localStorage;

}

public async override Task<AuthenticationState> GetAuthenticationStateAsync()

{

string token = await \_localStorage.GetItemAsStringAsync(TokenKey);

return this.BuildAuthenticationStateFromToken(token);

}

public async Task AuthenticateUserAsync(string userToken)

{

await \_localStorage.SetItemAsync(TokenKey, userToken);

var authState = this.BuildAuthenticationStateFromToken(userToken);

NotifyAuthenticationStateChanged(Task.FromResult(authState));

}

public async Task UnauthenticateAsync()

{

await \_localStorage.RemoveItemAsync(TokenKey);

var authState = this.BuildAuthenticationStateFromToken(string.Empty);

NotifyAuthenticationStateChanged(Task.FromResult(authState));

}

private AuthenticationState BuildAuthenticationStateFromToken(string userToken)

{

ClaimsIdentity userIdentity = null;

if (!string.IsNullOrWhiteSpace(userToken))

{

var userClaims = this.ParseClaimsFromJwt(userToken);

userIdentity = new ClaimsIdentity(userClaims, "apiauth\_type");

}

userIdentity ??= new ClaimsIdentity();

ClaimsPrincipal principal = new(userIdentity);

AuthenticationState authState = new(principal);

return authState;

}

private IEnumerable<Claim> ParseClaimsFromJwt(string jwt)

{

string payload = jwt.Split('.')[1];

payload += (payload.Length % 4) switch

{

2 => "==",

3 => "=",

\_ => string.Empty

};

byte[] jsonBytes = Convert.FromBase64String(payload);

var pairs = JsonSerializer.Deserialize<Dictionary<string, object>>(jsonBytes);

var claims = pairs.Select(pair => new Claim(pair.Key, pair.Value.ToString()));

Claim roleClaim = claims.FirstOrDefault(claim => claim.Type == "role");

return claims.Append(new Claim(ClaimTypes.Role, roleClaim.Value));

}

}