Приватний заклад вищої освіти
Одеський технологічний університет «ШАГ»
Кафедра інформаційних технологій та фундаментальної підготовки

Випускна кваліфікаційна робота бакалавра

«ІНФОРМАЦІЙНА СИСТЕМА КЕРУВАННЯ АВТОРСЬКИМИ МІКРОБЛОГАМИ»

на здобуття бакалаврського рівня вищої освіти
зі спеціальності «122 Комп'ютерні науки»

Виконавці проєкту

№	П.І.Б.	Група
1	Синяков Д. О.	КН-П-191
2	Луста А. С.	КН-П-191
3	Антонішин Д. О.	КН-П-191
4	Потапенко К. В.	КН-Д-191
5	Кечко О. С.	КН-Д-191
6	Батула Д. Ф.	КН-А-191

Автор звіту

_____ Синяков Дмитро Олександрович

Одеса – 2023

АНОТАЦІЯ

Основною метою створення проєкту є відтворення соціальної мережі, як способу поширення масової інформації, висловлення загальної або особистої думки та перегляд подібного контенту. Для визначення функціонала системи було проаналізовано подібні продукти, що вже існують, такі як Twitter та Instagram.

Розробка системи велась із метою надати користувачам комфортний та доступний спосіб розповсюдження інформації, при цьому надаючи їм конфіденційність збереження їх особистих даних.

Система має наступний функціонал: реєстрація, авторизація, автентифікація, редагування особистого профілю, створення постів та їх поширення у межах особистого профілю, збереження та поширення постів інших користувачів, оцінка опублікованих постів та їх коментування, надсилання особистих повідомлень іншим користувачам, перегляд отриманих повідомлень та постів.

Основним завданням при розробці системи було відтворення архітектури для зберігання, зміни, видалення та отримання цифрової інформації при обмежених ресурсах. Предметом дослідження є можливість створення такої архітектури додатка, що мала б надати розробнику спосіб її розширення при зміні або додаванні нового функціоналу системи. У якості задач дослідження було обрано:

- Переглянути функціонал додатків, що реалізують системи поширення інформації, з метою виділення їх спільних характеристик.
- Визначити технології створення програмного забезпечення, що підходять для реалізації заданого функціонала, з можливістю його розширення.
- Ознайомитись із хмарними технологіями та порівняти їх з метою обрати найбільш вдалий варіант для реалізації системи.
- Розробити додаток використовуючи результати, отримані у попередніх задачах.
- Запобігти цілісності інформації, що зберігатиметься у базі даних, впровадивши технології захисту даних.

Для розробки серверного додатка було використано ASP .NET Core 6 та мову програмування C#. За базу даних обрано MySql, що була розташована за допомогою сервісів хмарних обчислень AWS. Також для збереження медіафайлів було обрано сервіс S3 від AWS. Для реалізації клієнтської частини клієнтської частини було обрано створення мобільного додатку за допомогою технології .NET MAUI та мови програмування C#.

ЗМІСТ

ВСТУП

На сьогодні переважна більшість тих, хто користується пристроями, що мають доступ до мережі Інтернет, володіє обліковим записом у щонайменше одній з сучасних соціальних мереж, таких як Твіттер або Інстаграм. Безсумнівно, можна запевнити, що соціальні мережі стали невід'ємною частиною нашого повсякденного життя. Вони постачають інформацію про події у світі, допомагають користувачам дізнаватись більше інформації на теми, що їх цікавлять, розважатись, ділитись враженнями, висловлювати свою думку, та вислуховувати думку інших. Додатки що надають нам такі можливості займають, займають дуже високі сходинки у рейтингах завантажень інтернет-магазинів, та хоча й мають деякий негативний вплив на оточення, заперечення про їх необхідність не мають жодного сенсу у сучасному цифровому суспільстві.

Оцінюючи найбільш вдалі приклади соціальних мереж, ми прийшли до висновку, що кожна з них має свій менталітет та принципи, якими керується при розповсюдженні інформації. Таким чином, для нашого проєкту був обраний принцип надання користувачам ресурсу, за допомогою якого вони можуть займатись тим, що полюбляють робити в мережі Інтернет – пліткувати та ділитись новинами.

З технічної сторони, головною метою стало створити гнучку архітектуру, яка давала б можливість її удосконалення і масштабування для поширення функціонала додатка без зайвих труднощів. Надалі у документі описано життєвий цикл створення додатка, а також розглядаються труднощі, з якими ми зустрілися у процесі роботи.

У рамках даного звіту розглядається серверна частина додатка, що становить собою Web-API, розгорнуте за допомогою хмарних сервісів. Звіт складається з трьох розділів, що описують загальну характеристику системи, деталі щодо створення серверного додатка, а також результати її розгортання та тестування. Також у звіті присутні детальне технічне завдання проєкту та висновки.

ТЕХНІЧНЕ ЗАВДАННЯ ДО ПРОЄКТУ

Загальний опис

Програмне забезпечення (ПЗ) для поширення інформації шляхом спілкування з іншими користувачами

Призначення

Надання можливості поширення та розповсюдження інформації в мережі Інтернет. Збереження даних та надання конфіденційності користувачам, забезпечуючи безпеку та захист особистої інформації

Вимоги до функціонала

- Реєстрація та авторизація
  - Реєстрація
  - Підтвердження особистості
  - Авторизація
- Особистий акаунт користувача
  - Особистий профіль
  - Зміна особистих даних
  - Зміна пароля
- Інформаційні пости
  - Створення постів
  - Поширення особистих постів
  - Поширення постів, що сподобалися
  - Поширення постів інших користувачів
  - Зберігання постів
  - Видалення постів
  - Коментування постів
- Підпис та підписники
  - Підпис на інших користувачів
  - Відписка від інших користувачів

Архітектура

- Сервер бази даних – серверне програмне забезпечення для збереження та обробки поточних даних
- Сервіс Web-API – серверне програмне забезпечення, що надає програмний інтерфейс для взаємодії (REST-API) з іншим програмним забезпеченням.

- Мобільний додаток керування контентом - додаток що встановлюється на мобільні пристрої та надає користувачам інтерфейс для взаємодії з функціоналом.

Технології

- Для кожного з модулів рекомендується (з можливістю часткової зміни) наступні технології та методології створення
    - Бази даних – MySQL
    - Сервіс Web-API – REST-API, ASP.NET Core, C#
    - Мобільний додаток – .NET MAUI, C#
- При розгортанні програмного забезпечення рекомендується залучити хмарні технології
- При розробці програмного забезпечення необхідно залучити зберігання вихідних кодів ПЗ повинно здійснюватись за допомогою системи контролю версій (вибір конкретної системи узгоджується групою проєкт).
- Вихідний код повинен оформлятися відповідно до загальноприйнятих стандартів, на зразок стандартів GNU http://www.gnu.org/prep/standards/, у тому числі з додержанням вимог до коментування та документування коду.

РОЗДІЛ 1
ЗАГАЛЬНА ХАРАКТЕРИСТИКА СИСТЕМИ

За вимогами технічного завдання (ТЗ) архітектура системи була розділена на декілька частин:

- «Design»
- «Frontend»
- «Backend»

Частина «Design» являє собою модель інтерфейсу взаємодії користувачів із функціональною частиною додатка. Ця частина складається із моделей візуального інтерфейсу, що повинні бути реалізовані у додатку. При створенні інтерфейсів було використано принципи UI/UX, що забезпечують інтуїтивну взаємодію користувача із функціоналом системи. Частина розміщена за адресою https://www.figma.com/file/DUaLAVhz9ZvboEtRyyWfJG/Wireframes?type=design&node-id=108-953&mode=design&t=YifX2FbAJF6qPGLe-0.

Частина «Frontend» реалізована у вигляді мобільного додатку, що надає користувачам візуальний інтерфейс для взаємодії із функціями системи. Для функціонування додатка потрібне підключення до мережі Інтернет. Основним принципом взаємодії додатка з серверною частиною є надсилання HTTP-запитів і отримання відповідей на їх основі. Додаток також повинен мати доступ до локального сховища пристрою для збереження інформації про підключення до серверної частини та токенів авторизації. Вихідні коди даної частини зберігаються на репозиторії https://github.com/plitkarka/client.git.

Частина «Backend» являє собою серверне програмне забезпечення (ПЗ), створене з метою збереження та обробки інформації, що надається користувачами системи. Вона реалізована у вигляді Web-API, що надає програмний інтерфейс для взаємодії із нею. ПЗ реалізує певну кількість кінцевих точок для обробки HTTP запитів отриманих від клієнтської частини, створення яких керувалося використанням принципів REST архітектури. Дані, якими керує додаток, зберігаються з використанням бази даних MySQL, до якої ПЗ має постійний доступ, а також сервісу AWS S3 для збереження медіафайлів. ПЗ створене з використанням мови програмування C#, а за його основу було використано технологію ASP .NET Core 6. Вихідні коди даної частини зберігаються на репозиторії https://github.com/plitkarka/Plitkarka.git.

Для збереження вихідних кодів системи було використано сервіс GitHub та систему контролю версій Git. Також було використано сервіс GitHub Actions для реалізації CI/CD, тобто постійної інтеграції внесених змін у готовий для використання додаток. Створений за допомогою GitHub Actions пайплайн автоматично розгортає серверний додаток за допомогою Docker та AWS EC2, перед цим тестуючи його на працездатність.

РОЗДІЛ 2
РЕАЛІЗАЦІЯ СЕРВЕРНОЇ ЧАСТИНИ

Перед початком роботи була поставлена задача виділити основні технології для створення серверної частини. Першим бажанням при створенні плану була реалізація мікросервісної архітектури, адже така архітектура дає багато можливостей для майбутнього масштабування додатка, шляхом створення нових компонентів/сервісів, та заміни старих компонентів на нові. Одним із перших питань було середовище розгортання, за яке спочатку було обрано хмарні сервіси Microsoft Azure, адже даний набір сервісів дає повний інструментарій для створення саме такої архітектури, що була обрана. На перевагу Microsoft Azure також вплинули інструменти Azure DevOps, що дозволяли реалізувати процес CI/CD та організацію командної роботи, а також функціонал репозитаріїв для збереження та контролю вихідних кодів. Базуючись на виборі сервісів компанії Microsoft, за основну технологію розробки додатка було обрано платформу .NET та мову програмування C#, адже вони мають повну підтримку сервісів Microsoft Azure та власну технологію для створення Web-API, якою є ASP .NET Core 6.

У ході аналізу було вирішено відмовитись від технологій Microsoft Azure, адже реалізація мікросервісної архітектури була значно складнішою за створення додатка з монолітною архітектурою і не було впевненості, що робота була б виконана за укладений час. Хоча й була можливість реалізувати подібний додаток з монолітною архітектурою за допомогою сервісів Microsoft Azure, замість них було вирішено використовувати AWS (Amazon Web Services), з причини кількості ресурсів що були б витрачені на розгортання додатка, яких у випадку AWS знадобилося б менше. Через відмову від Microsoft Azure, також було прийнято рішення відмовитися від Azure DevOps, на перевагу GitHub для реалізації процесу CI/CD та використання систему контролю версій Git.

Отже, у кінцевому результаті для створення серверної частини додатка було використано наступні технології:

- Платформа .NET та мова програмування C#. Технологія ASP .NET Core 6
- Модулі Entity Framework Core, MediatR, SignalR, Swagger, AutoMapper, MailKit, AWSSDK
- Середовище розробки Visual Studio Community 2022 та редактор коду Visual Studio Code
- Сервіси AWS – EC2, RDS та S3

Архітектура готового серверного ПЗ нагадує звичайну “Onion” архітектуру, адже складається з трьох рівнів:

- Application layer
- Domain layer
- Infrastructure layer

Application layer – рівень додатка. За допомогою механізму контролерів у ньому реалізовані кінцеві точки Web-API, що приймають HTTP-запити та

передають їх на обробку за допомогою модулю MidiatR. Також на цьому рівні відбувається конфігурація самого додатку до якої входять: створення об'єктів конфігурації, що зберігають дані про підключення до зовнішніх інфраструктурних елементів, налаштування конвеєра обробки запитів, а саме підключення middleware об'єктів та mapping контролерів, та конфігурація контейнера впровадження залежностей (DI Container).

Domain layer – рівень обробки запитів, або ж рівень бізнес-логіки. При розробці додатка було звернено увагу на дотримання принципів розробки S.O.L.I.D., отже для реалізації обробки запитів на цьому рівні було обрано вищезгаданий модуль MediatR. Головною ідеєю модуля є створення спеціальних об'єктів (Handler) що обробляють передані їм запити (Request). Кожен запит має лише один обробник, а кожен обробник обробляє лише один запит, що забезпечує дотримання принципу єдиної відповідальності (Single Responsibility). Також на цьому рівні реалізовані сервіси, що надають можливості використані у різних місцях додатка.

Infrastructure layer – рівень доступу до зовнішніх інфраструктурних компонентів. У звичайному прикладі реалізації “Onion” архітектури даний рівень має назву “Domain Model”, але до інфраструктури відноситься й використання бази даних, що у випадку цієї системи є головним завданням цього рівня. Infrastructure layer має у собі реалізацію сутностей (Entity) бази даних за допомогою модулю Entity Framework Core. Кожна сутність являю собою C# клас, що описує властиві їй поля, та дані що будуть зберігатися у базі. Також на цьому рівні реалізовані репозиторії доступу до сутностей бази даних. Кожен репозиторій є звичайним ASP .NET Core сервісом, що має доступ до контексту бази даних, та реалізує CRUD операції над об'єктами, що в ній зберігаються.

Демонстрація роботи додатку

Для демонстрації принципу роботи усього механізму було обрано запит на реєстрацію нового користувача.

```
[HttpPost]
[ModelStateValidation]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<ActionResult<StringResponse>> SignUp(
    [FromBody] SignUpRequestModel body)
{
    var email = await _mediator.Send(new SignUpRequest(
        Login: body.Login,
        Name: body.Name,
        Email: body.Email,
        Password: body.Password,
        BirthDate: body.BirthDate));

    return Ok(new StringResponse(email));
}
```

При отриманні запиту, контролер валідує отриманні дані за допомогою власного атрибута ModelStateValidation, що перевіряє всі поля об'єкта SignUpRequestModel, переданого до контролера через параметри методу, а саме значення цього параметру отримане із тіла HTTP-запиту. У випадку невдалої валідації буде згенероване виключення, що буде оброблено об'єктом ExceptionMiddleware, що вбудований у конвеєр обробки запиту. Додаток реагує на різного роду виключення як раз за допомогою цього класу, і в залежності від них, може створювати відповіді на невдалі запити. У випадку якщо отримані поля не пройдуть валідацію, користувач отримає статус код 400 BadRequest, що також вказано у методі контролеру. У випадку якщо обробка запиту пройде вдало, запит завершиться зі статус кодом 200 OK. Після валідації створюється запит для MediatR і обробка переходить на Domain layer.

```
private async Task ValidateEmailAndlogin(SignUpRequest request)
    {
        UserEntity? userExist;

        try
        {
            userExist = await _repository.GetAll().FirstOrDefaultAsync(
               user => user.Email == request.Email || user.Login == request.Login);

            if (userExist != null)
            {
                if (userExist.Email == request.Email)
                {
                    throw new ValidationException("This Email is already used", nameof(request.Email));
                }

                if (userExist.Login == request.Login)
                {
                    throw new ValidationException("This Login is already used", nameof(request.Login));
                }
            }
        }
        catch (Exception ex) when (ex is not ValidationException)
        {
            throw new MySqlException(ex.Message);
        }
    }
```

У MediatR обробнику на Domain layer першим чином відбувається валідація даних, що стосуються бізнес-логіки додатка. В системі не повинно бути користувачів з однаковими логінами або електронною поштою, тож перед тим як створити нового користувача, обробник звертається до бази даних із запитом, на перевірку факту існування користувачів з однаковим логіном або поштою, та у випадку, якщо такі користувачі існують, генерує виключення ValidationException з описом проблеми, що також буде оброблено за допомогою ExceptionMiddleware, і відповіддю на HTTP-запит у такому випадку також буде 400 BadRequest. Також при роботі з базою даних береться до уваги факт її відключення, тому якщо у процесі валідації логіну або пошти буде згенероване виключення відмінне від ValidationException, даний метод згенерує замість нього

MySqlException, на ExceptionMiddleware поверне до клієнтської частини статус код 500 InternalServerError.

```
public async Task<string> Handle(
        SignUpRequest request,
        CancellationToken cancellationToken)
    {
        await ValidateEmailAndlogin(request);

        var salt = _encryptionService.GenerateSalt();

        var newUser = new User()
        {
            Login = request.Login,
            Name = request.Name,
            Email = request.Email,
            EmailCode = _emailService.GenerateEmailCode(),
            Password = _encryptionService.Hash(request.Password + salt),
            Salt = salt,
            BirthDate = request.BirthDate,
            LastLoginDate = DateTime.UtcNow
        };

        newUser.Id = await _repository.AddAsync(
            _mapper.Map<UserEntity>(newUser));

        await _emailService.SendEmailAsync(
            newUser.Email,
            EmailTextTemplates.VerificationCodeText(newUser.Name, newUser.EmailCode),
            EmailTextTemplates.VerificationCode);

        return newUser.Email;
    }
```

Якщо всі етапи валідації було успішно пройдено MediatR запит перейде до створення нової моделі користувача.

Кожна модель даних має два типи – сутність та бізнес-модель, що можуть бути переведені один в одного за допомогою модуля AutoMapper. Це робиться з ціллю відокремити логіку роботи з даними, та логіку їх зберігання, адже первісна конструкція нової сутності, може вміщувати у себе деякі процедури, які було б краще зробити зі звичайним об'єктом, а не з об'єктом, що повинен після цього зберігатись у базі даних. Такий підхід виключає необхідність створювати методи роботи з сутностями бази даних які не є необхідними на рівні Infrastructure layer. У такому випадку створюється бізнес-модель користувача – User, що перед додаванням до бази даних буде перетворена на сутність UserEntity.

Паролі зберігаються у базі даних у хешованому вигляді. Для користувача створюється сіль – унікальний ідентифікатор, що поєднується з паролем перед хешуванням, що зменшує до мінімуму можливість отримати два однакові хешовані паролі.

Також при реєстрації нового користувача створюється спеціальний код із 6 символів, що відправляються на вказану пошту за допомогою EmailService для підтвердження дійсності пошти.

```
private IRepository<UserEntity> _repository { get; init; }
private IMapper _mapper { get; init; }
private IEmailService _emailService { get; init; }
private IEncryptionService _encryptionService { get; init; }

public SignUpHandler(
    IRepository<UserEntity> repository,
    IMapper mapper,
    IEmailService emailService,
    IEncryptionService encryptionService)
{
    _repository = repository;
    _mapper = mapper;
    _emailService = emailService;
    _encryptionService = encryptionService;
}
```

Конструювання кожного обробника відбувається за допомогою DI контейнеру. Він виконує роботу впровадження залежностей які потребує обробник. У цьому випадку даними залежностями є сервіси додатка. Використання сервісів було продемонстровано в описі роботи обробника запиту створення акаунту. Він залежить від чотирьох сервісів: репозиторію користувачів, мапперу, сервісу надсилання поштових повідомлень та криптографічного сервісу.

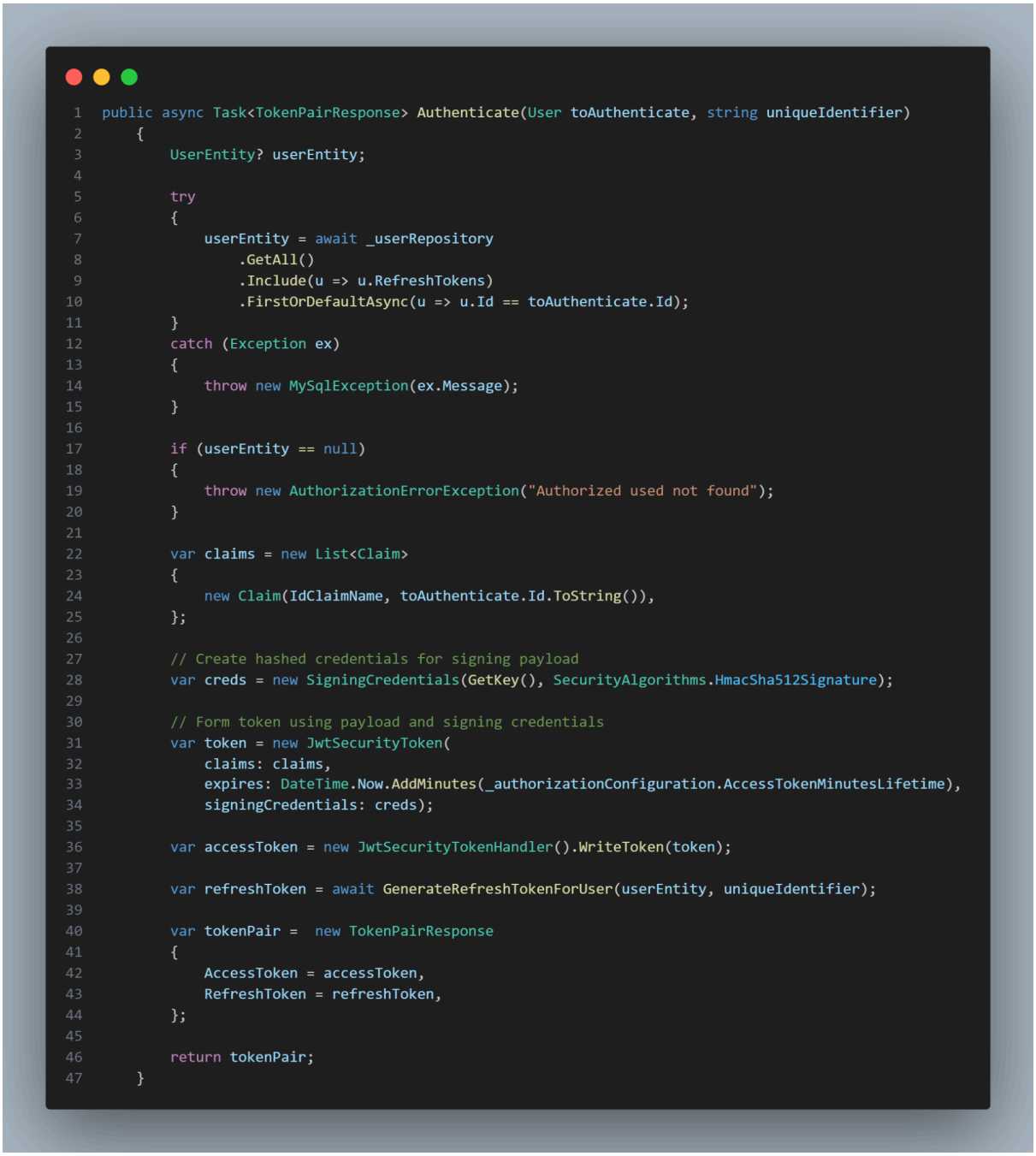
Репозиторій користувачів є сервісом Infrastructure layer. Він надає можливість працювати із сутностями користувачів, що знаходяться у базі даних. Жоден компонент додатка не повинен мати доступу до контексту базу даних, окрім репозиторіїв. У прикладі реєстрації нового користувача були використані два методи цього репозиторію – додавання користувача та отримання IQueryable об'єкту для створення запиту до бази даних. При використанні можливості додавання користувача за допомогою репозиторію обробка помилки та генерація виключення відбувається безпосередньо у репозиторії, так само як при видалені сутності, оновлені та отримані по Id користувача. Але у випадку створення запиту до бази даних через метод GetAll обробка помилки відбувається у місці звернення до репозиторію.

```
public async Task<Guid> AddAsync(UserEntity user)
{
    if (user == null)
    {
        throw new ArgumentNullException(nameof(user));
    }

    try
    {
        var res = await _db.Users.AddAsync(user);
        await _db.SaveChangesAsync();

        return res.Entity.Id;
    }
    catch(Exception ex)
    {
        throw new MySqlException(ex.Message);
    }
}
```

Авторизація та автентифікація

Для захисту особистих даних було впроваджено систему авторизації та автентифікації. За основу системи було обрано механізм JWT. Для реалізації механізму авторизації та автентифікації було прийнято рішення створити особисті компоненти, що будуть реалізовувати цей функціонал.

Автентифікація – це процедура підтвердження того ким є користувач, що проходить під час його входу до акаунту. Перевірка дійсності відбувається шляхом введення електронної пошти та паролю. Результатом процесу автентифікації є надання користувачеві токенів для авторизації. Під час процесу автентифікації клієнтська частина також надає серверному додатку спеціальний ідентифікатор пристрою, що дозволяє реалізувати вхід до особистого акаунту користувача з декількох пристроїв одночасно, адже в такому випадку токени авторизації створюються саме для певного пристрою. Таким ідентифікатором може виступати як певний набір символів, що якимось чином прив'язаний до пристрою з якого був відправлений запит на автентифікацію, так і штучно згенерований клієнтською частиною ідентифікатор, що після цього буде збережений у локальному сховищі пристрою. Автентифікація відбувається за допомогою механізму Claim, наданого базовим функціоналом ASP .NET Core 6.



```
public async Task<TokenPairResponse> Authenticate(User toAuthenticate, string uniqueIdentifier)
    {
        UserEntity? userEntity;

        try
        {
            userEntity = await _userRepository
                .GetAll()
                .Include(u => u.RefreshTokens)
                .FirstOrDefaultAsync(u => u.Id == toAuthenticate.Id);
        }
        catch (Exception ex)
        {
            throw new MySqlException(ex.Message);
        }

        if (userEntity == null)
        {
            throw new AuthorizationErrorException("Authorized used not found");
        }

        var claims = new List<Claim>
        {
            new Claim(IdClaimName, toAuthenticate.Id.ToString()),
        };

        // Create hashed credentials for signing payload
        var creds = new SigningCredentials(GetKey(), SecurityAlgorithms.HmacSha512Signature);

        // Form token using payload and signing credentials
        var token = new JwtSecurityToken(
            claims: claims,
            expires: DateTime.Now.AddMinutes(_authorizationConfiguration.AccessTokenMinutesLifetime),
            signingCredentials: creds);

        var accessToken = new JwtSecurityTokenHandler().WriteToken(token);

        var refreshToken = await GenerateRefreshTokenForUser(userEntity, uniqueIdentifier);

        var tokenPair =  new TokenPairResponse
        {
            AccessToken = accessToken,
            RefreshToken = refreshToken,
        };

        return tokenPair;
    }
```

У результаті процесу автентифікації користувач отримує два токени – токен доступу (Access Token) та токен відновлення (Refresh Token). Токен доступу є класичним прикладом JWT (JSON Web Token), що складається із заголовків з загальною інформацією, корисної частини, що зберігає Id користувача, та електронного підпису. JWT має певний час існування (10-20 хвилин у більшості випадків). Якщо система отримає токен доступу, час існування якого вийшов, вона відповість на запит зі статус кодом 401 Unauthorized. У такому випадку токен треба оновити надіславши певний запит з використанням обох токенів. Токен оновлення є лише певним ідентифікатором, що зберігається у базі даних та слугує підтвердженням дійсності оновлення токена доступу. Такий механізм запобігає втраті токена доступу, шляхом атак

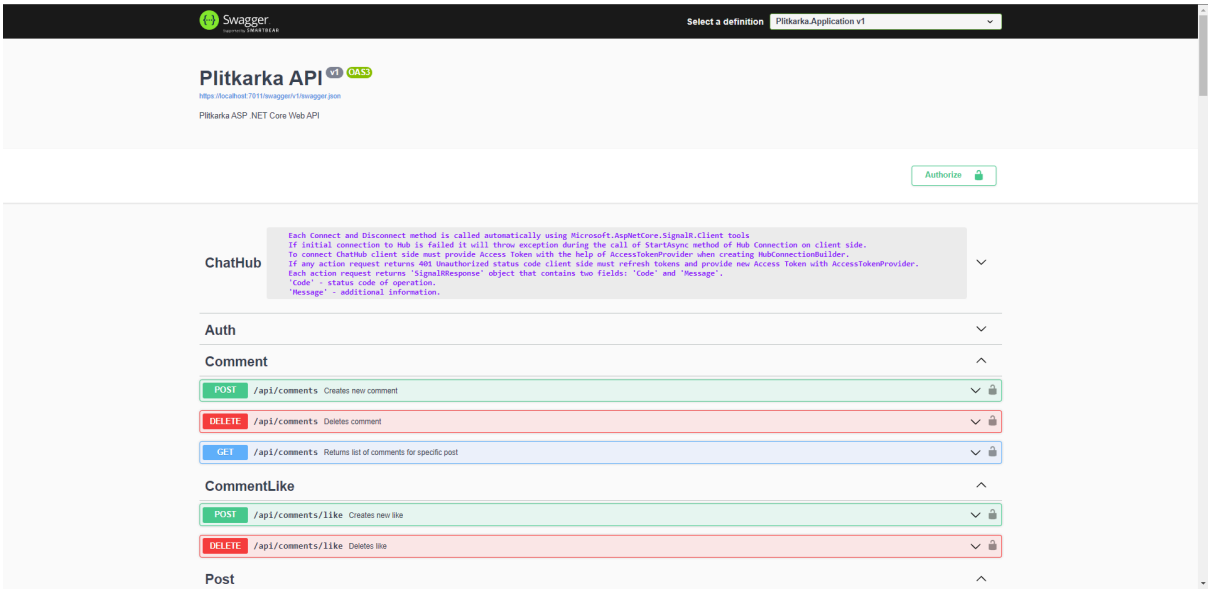
спрямованих на отримання токенів, адже у такому випадку токен перестане бути дієздатним через деякий час.

Авторизація – процес надання користувачеві прав на певні дії у додатку. Авторизація відбувається за допомогою токенів, що були створені у процесі автентифікації. При отриманні запиту AuthorizationMiddleware, що вбудований до конвеєра обробки запитів, перевірить наданий токен доступу (Access Token), та на його основі отримає Id користувача, що створив цей запит. У випадку якщо токен не надано або він не є вірним, система генерує виключення при якому користувач не отримає задовільну відповідь на запит, що запобігає шахрайству та отриманню приватної інформації.

```
virtual public async Task InvokeAsync(
    HttpContext context)
{
    var token = _contextAccessTokenService.AccessToken;

    if (token != null)
    {
        Guid userId;
        UserEntity? user;

        userId = _authorizationService.Authorize(token);

        if (userId == Guid.Empty)
        {
            await _next(context);
            return;
        }

        var userRepository = context
            .RequestServices
            .GetRequiredService<IRepository<UserEntity>>();

        user = await userRepository.GetByIdAsync(userId);

        if (user == null)
        {
            throw new AuthorizationErrorException("User not found");
        }

        if (user.EmailCode != EmailService.VerifiedCode)
        {
            throw new AuthorizationErrorException("Email is not verified");
        }

        user.LastLoginDate = DateTime.UtcNow.Date;

        await userRepository.UpdateAsync(user);

        _contextUserService.User = _mapper.Map<User>(user);
    }

    await _next(context);
}
```

РОЗДІЛ 3
РЕЗУЛЬТАТИ ВИПРОБУВАНЬ

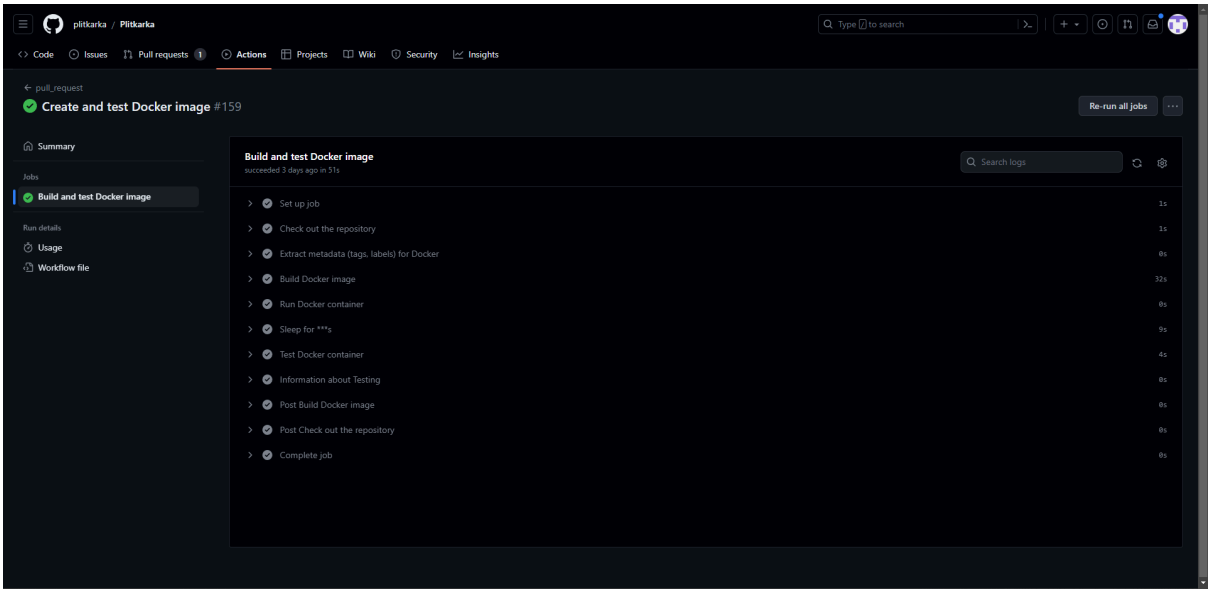
При розробці серверного додатка було використано модуль Swagger, що дозволяє отримати автоматично згенерований інтерфейс для створення запитів. За допомогою атрибутів, що надані цим модулем, Інтерфейс був налаштований для надання повної інформації про кожен запит. Даний інтерфейс є доступним тільки у випадку якщо додаток було увімкнено у середовищі Develop (розробка), адже інтерфейс Swagger не повинен бути доступний користувачам після розгортання додатка.



Також для тестування працездатності додатка перед його розгортанням було запроваджено механізм HealthCheck, що надає можливість перевірити стан додатка після його запуску через певний HTTP-запит.

```
public static IServiceCollection AddMyHealthChecks(
    this IServiceCollection services,
    IConfiguration configuration)
{
    var connectionString = configuration["MySql:ConnectionString"];

    services
        .AddHealthChecks()
        .AddCheck("self", () => HealthCheckResult.Healthy())
        .AddMySql(connectionString!);

    return services;
}
```

При додаванні нових змін до гілки “master” у системі контролю версій Git, сервіс GitHub Actions запускає пайплайн для збірки та розгортання додатка на екземплярі віртуальної машини, що налаштована за допомогою сервісу AWS EC2. Пайплайн створює зображення (Image) додатку за допомогою Docker, розміщує дане зображення на порталі Docker Hub, після чого приєднується до віртуальної машини за допомогою протоколу SSH. На віртуальній машині запускається скрипт, що розгортає додаток із порталу Docker Hub в ізольованому контейнері Docker. Контейнер та віртуальна машина мають налаштовані порти для отримання HTTP-запитів, що надає змогу клієнтській частині додатка,встановленій на пристроях користувачів, надсилати запити до серверного додатка, що таким чином знаходиться у публічному доступі.



Усі функції, що регламентовані технічним завданням були реалізовані та протестовані. Вихідний код додатка розміщений на репозиторії: https://github.com/plitkarka/Plitkarka.git.

ВИСНОВКИ

Було проведено аналіз інформаційних систем керування авторськими мікроблогами, виділено їх спільні та відмінні характеристики та виявлено, що найбільше уваги користувачі виділяють системам із найбільшою доступністю та впровадження засобів надання безпеки персональним даним.

Визначено програмні засоби та технології, що надають можливість створення інформаційних систем керування авторськими мікроблогами із підвищеним рівнем захисту особистих даних та можливістю наступного збагачення функціонала системи.

Спроєктовано, реалізовано та протестовано інформаційну систему керування авторськими мікроблогами. Були запроваджені усі функції затверджені технічним завданням. Система реалізована за роздільною клієнт-серверною архітектурою та складається з трьох частин, що можна знайти за адресами:

- «Design» – https://www.figma.com/file/DUaLAVhz9ZvboEtRyyWfJG/Wireframes?type=design&node-id=108-953&mode=design&t=HunYxcfxXrTAV5QT-0
- «Frontend» – https://github.com/plitkarka/client.git
- «Backend» – https://github.com/plitkarka/Plitkarka.git

У систему було запроваджено власний механізм авторизації та автентифікації для підвищення рівня безпеки персональних даних, механізми хешування даних, що не повинні бути відомі іншим, та засоби підтвердження електронною пошти.

Систему було випробувано шляхом розгортання за допомогою хмарних технологій та використання її функціоналу у публічному доступі.

Було обрано перелік функцій для майбутнього удосконалення системи:

- Механізм двофакторної автентифікації.
- Система блокування користувачів.
- Система закриття персонального обліковий запис від публічного доступу.
- Можлива перебудова додатку на мікросервісну архітектуру для масштабування продуктивності.
- Покриття серверного додатку Unit-тестами.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ ТА ТЕХНОЛОГІЙ

1. Джефрі Ріхтер "CLR VIA C#”
2. Роберт Мартін “Чиста архітектура”
3. Роберт Мартін “Чистий код”
4. Платформа .NET – https://dotnet.microsoft.com/en-us/
5. C# – https://learn.microsoft.com/en-us/dotnet/csharp/
6. ASP .NET Core 6 – https://learn.microsoft.com/en-us/aspnet/core/release-notes/aspnetcore-6.0?view=aspnetcore-7.0#minimal-apis
7. SignalR – https://learn.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr
8. Entity Framework – https://learn.microsoft.com/en-us/ef/
9. AutoMapper – https://automapper.org/
10. MediatR – https://github.com/jbogard/MediatR
11. MySQL – https://www.mysql.com/
12. Visual Studio – https://visualstudio.microsoft.com/
13. Visual Studio Code – https://code.visualstudio.com/
14. GitHub – https://github.com/
15. GitHub Actions – https://github.com/features/actions
16. Git – https://git-scm.com/
17. AWS – https://aws.amazon.com/
18. AWS RDS – https://aws.amazon.com/rds/
19. AWS S3 – https://aws.amazon.com/s3/
20. Docker – https://www.docker.com/
21. Docker Hub – https://hub.docker.com/