Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Розробка веб-застосунку з використанням технології Spring Boot та React

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
ст. викладач Борозений С.О.

“____” _____ 2024 р.

Виконав студент
Андрусенко Анатолій

“____” _____ 2024 р.

Київ 2024

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Старший викладач
_____С. О. Борозений
(підпис)
„____”_____2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Андрусенку Анатолію Дмитровичу факультету інформатики
3-го курсу
Тема Розробка веб-застосунку з використанням технології
Spring Boot та React

Зміст ТЧ до курсової роботи:

1. Індивідуальне завдання
2. Анотація
3. Вступ
4. Розділ 1. Дизайн бази даних
5. Розділ 2. Розробка backend застосунку
6. Розділ 3. Розробка frontend застосунку
7. Висновки
8. Список використаної літератури

Дата видачі „___” _____ 2023 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Розробка веб-застосунку з використанням технології Spring Boot та React

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової	10.04.2024	
2.	Аналіз тематичної літератури	16.04.2024	
3.	Написання першого розділу	20.04.2024	
4.	Написання другого розділу	30.04.2024	
5.	Написання третього розділу	8.05.2024	
6.	Завантаження роботи	16.05.2024	

Студент Андрусенко А. Д.
Керівник Борозений С.О.
“_____” _____

Зміст

Анотація

Ця курсова робота має на меті розробку веб-застосунку з використанням backend фреймворку Spring Boot та frontend бібліотеки React.

Результатом цієї роботи є full stack застосунок-соцмережа, що реалізує такий функціонал, як аутентифікація та авторизація користувачів, створення та перегляд постів, коментарів тощо. На прикладі цього застосунку продемонстровано підхід до рішень типових проблем з якими стикаються веб розробники в процесі створення застосунків, а також показано переваги та недоліки використаних технологій, їх архітектурних особливостей. Окрема увага приділяється захисту API від несанкціонованого доступу за допомогою можливостей Spring Security.

Для налаштування frontend застосунку використано інструмент Vite, для backend – Spring Initializr.

Версії використаних технологій:

- Spring boot 3.2.5
- React 18.2.0

Ключові слова: backend, frontend, full stack, React, Spring boot, Spring Framework, Spring Security

Вступ

Актуальність теми. Швидкий розвиток мережі останніх десятиліть йде пліч-о-пліч зі зростанням потреб ринку до веб-сайтів. Кожна велика компанія має щонайменше один сайт для рекламування, надання послуг, тощо. На зміну звичайним статичним веб-сайтам поступово приходили повноцінні веб застосунки, спроможні не тільки відображати інформацію, а й обмінюватися даними з клієнтом. Веб застосунок має переваги перед звичайним додатком, адже може бути відображений на будь-якому пристрою за наявності лише інтернет браузеру, через що для адаптування застосунку для різних систем потрібна лише одна команда; а також не потребує попереднього встановлення на пристрій.

Зі зростом потреб до сайтів з’являлися та розвивалися численні технології та фреймворки для розробки frontend та backend застосунків. Серед frontend технологій однією з найчастіше використовуваних є бібліотека React, що базується на розділенні коду на невеликі компоненти, призначені для багаторазового використання. Ця технологія була обрана для використання у цій роботі через її популярність, значну кількість бібліотек створених спільнотою та відносну простоту у використанні. Spring boot був вибраний у якості backend фреймворку через його широку екосистему та масштабованість.

Мета і завдання дослідження. Мета цієї роботи полягає у розробці full stack застосунку, що охоплює вирішення задач та проблем притаманних такій розробці. Серед основних завдань, вирішених у цій роботі, є наступні:

- Дизайн бази даних для зберігання даних на серверній стороні;

- Створення ефективного та захищеного API на серверній стороні для обміну даними з клієнтським застосунком, що повністю реалізує необхідну бізнес логіку;
- Розробка frontend застосунку для відображення ресурсу на стороні клієнта.

Об’єктом дослідження є відповідні технології, використані в практичній частині курсової роботи, а також їх використання для виконання поставлених задач.

Розділ 1. Дизайн бази даних

Початком роботи над застосунком є аналіз функціоналу, що має бути реалізованим, та дизайн відповідної бази даних.

1.1 Аналіз функціоналу

Першочергово, розроблюваний застосунок зберігатиме користувачів, які зможуть писати текстові пости, а також коментувати їх. Унікальний ідентифікатор користувача – це короткий тег, який користувач створює власноруч та використовує для авторизації. Кожен пост має заголовок, текстовий контент та теги. Користувач може коментувати сам пост, а також інші коментарі, таким чином кожен коментар може мати коментарі-відповіді. Користувач може додавати пости та коментарі до тих, що сподобалися натиснувши на відповідну кнопку.

1.2 Аналіз таблиць бази даних

Опишемо таблиці, що будуть використані в реляційній базі даних:

- **User**. Ця таблиця представляє користувача та має наступні поля:
  - userId - це унікальний ідентифікатор юзера, що створюється ним самим. Обов’язкове поле;
  - nickname - псевдонім користувача, що відображається для інших користувачів. На відміну від userId, він не є унікальним, тобто кілька користувачів можуть мати той самий псевдонім. Обов’язкове поле;
  - email – адрес електронної скриньки користувача, що не відображається для інших користувачів з метою забезпечення безпеки. Обов’язкове поле;
  - role – роль користувача, використовується для виконання захищених операцій, наприклад, додання та видалення тегів. Звичайний користувач має роль «USER», користувач з необмеженими правами має роль «ADMIN». Обов’язкове поле;
  - avatar – посилання на картинку профіля юзера. Картинки зберігаються у файловій системі сервера. Необов’язкове поле;

    - password – пароль користувача, що використовується для авторизації. Захищене поле, API в жодному випадку не має передавати пароль. Обов’язкове поле;
    - registeredDate – дата реєстрації користувача в системі. Генерується автоматично. Обов’язкове поле;
- **Post**. Це таблиця постів. Поля таблиці:
    - postId – ідентифікатор поста, число. Генерується автоматично, обов’язкове поле;
    - title – заголовок поста. Обов’язкове поле;
    - content – наповнення поста. Обов’язкове поле;
    - postAuthorId – зовнішній ключ, userId автора поста.
    - postedDate – час, коли пост був створений. Генерується автоматично. Обов’язкове поле;
- **PostLike** – таблиця лайків під постами, пов’язує між собою користувача, що вподобав пост, та пост. Поля таблиці:
    - userId – ідентифікатор користувача, що вподобав пост, зовнішній ключ, частина комбінованого первинного ключа таблиці. Обов’язкове поле;
    - postId – ідентифікатор поста, зовнішній ключ, частина комбінованого первинного ключа таблиці. Обов’язкове поле;
    - likeDate – дата, коли пост було вподобано. Генерується автоматично. Обов’язкове поле;
- **Tag –** таблиця тегів, які можуть мати пости. Поля таблиці:
    - tagId – ідентифікатор тега. Обов’язкове поле;
    - name – назва тега. Обов’язкове поле;
- **PostTag** – таблиця, що пов’язує пости та їх теги. Поля:
    - postId – ідентифікатор поста, зовнішній ключ, частина комбінованого первинного ключа. Обов’язкове поле;
    - tagId – ідентифікатор тега, зовнішній ключ, частина комбінованого первинного ключа. Обов’язкове поле;
- **Comment –** таблиця коментарів до постів. Поля таблиці:
    - commentId – унікальний ідентифікатор коментаря, первинний ключ, обов’язкове поле;
    - content – наповнення коментаря, обов’язкове поле.
    - postId – ідентифікатор поста, під яким користувач залишив цей коментар. Зовнішній ключ, обов’язкове поле;
    - commentAuthorId – ідентифікатор користувача, котрий є автором коментаря. Зовнішній ключ, обов’язкове поле;

    - commentedDate – дата, коли було створено коментар. Генерується автоматично, обов’язкове поле;
    - edited – поле, що відмічає чи коментар було змінено. За замовчуванням має значення false. Обов’язкове поле;
    - replyTo – ідентифікатор коментаря, на який цей коментар є відповіддю. Зовнішній ключ, необов’язкове поле
- **CommentLike** – таблиця уподобань під коментарями, пов’язує між собою користувача, що вподобав коментар, та коментар. Поля таблиці:
    - userId – ідентифікатор користувача, що вподобав коментар, зовнішній ключ, частина комбінованого первинного ключа таблиці. Обов’язкове поле;
    - commentId – ідентифікатор вподобаного коментаря, зовнішній ключ, частина комбінованого первинного ключа таблиці. Обов’язкове поле;
    - likeDate – дата, коли коментар було вподобано. Генерується автоматично. Обов’язкове поле.

1.3 Аналіз зв’язків

Користувачі, що вподобали пости, пов’язані з постами зв’язком «Багато до багатьох» через проміжну таблицю PostLike. Користувач може мати довільну кількість вподобаних постів, також пости можуть мати довільну кількість вподобань.

Користувачі мають зв’язок «Один до багатьох» з постами, авторами яких вони є. У користувача може бути довільна кількість постів, а у поста завжди є рівно один автор.

Пости мають зв’язок з «Багато до багатьох» з їх тегами через проміжну таблицю PostTag. У кожного поста може бути довільна кількість тегів, і навпаки.

Таблиця постів має зв’язок «Один до багатьох» з таблицею коментарів. Кожен пост може мати довільну кількість коментарів, коментар обов’язково має зв’язок зі своїм постом.

Користувачі мають зв’язок «Один до багатьох» з коментарями, авторами яких вони є. У користувача може бути довільна кількість коментарів, у коментаря є рівно один автор.

Коментарі мають рекурсивний зв’язок «Один до багатьох з коментарем, відповіддю до якого вони є. Коментар може мати або не мати відповіді, коментар може мати або не мати батьківський коментар.

Коментарі мають зв’язок «Багато до багатьох» з їх користувачами, які їх вподобали, через таблицю CommentLike. Кожен користувач має довільну кількість вподобаних коментарів, і навпаки.

1.4 Підсумковий дизайн бази даних

Загалом, розроблена база даних має наступний вигляд (Рисунок 1.4.1):

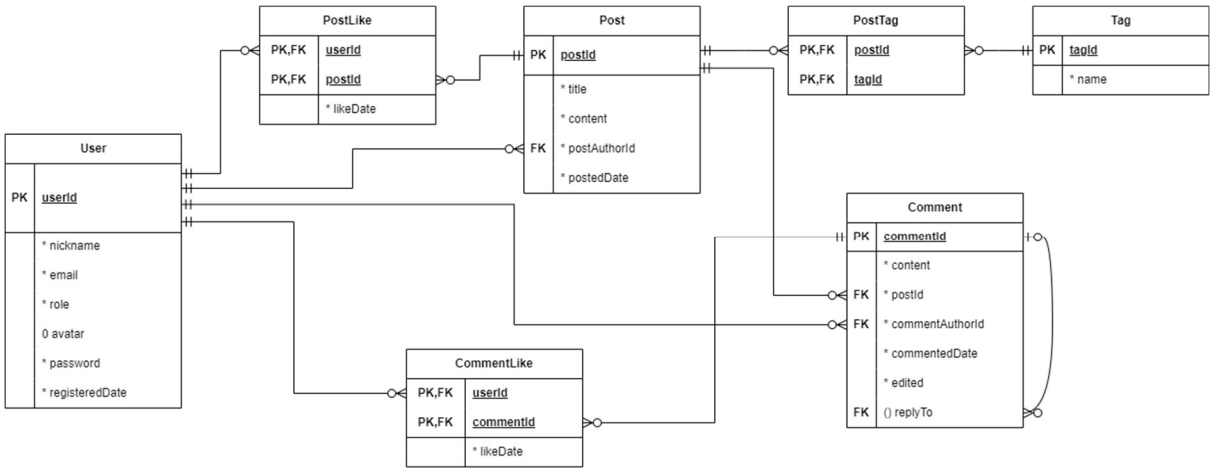

Рисунок 1.4.1

Розділ 2. Розробка backend застосунку

Spring Boot – це зручний фреймворк що базується на мові програмування Java. Цей фреймворк працює на базі Spring Framework, спрощує його конфігурацію та зменшує кількість коду, потрібного для роботи програми [1].

У статті [2] було оглянуто найпопулярніші backend фреймворки, а саме Spring, Lavarel, Ruby On Rails та Django. Результати показали, що незважаючи на високий поріг входу Spring він широко використовується через його масштабованість, інтегрованість зі сторонніми технологіями, докладну документацію тощо.

Для роботи з базами даних Spring boot використовує технологію Hibernate, представляючи записи з бази даних у вигляді Java об’єктів.

У цій роботі Spring boot використовується для створення RESTful API. REST – це архітектура API побудована на HTTP протоколі, що створює зручний інтерфейс для обміну даними між сервером та клієнтом.

2.1 Ініціалізація проекту

Для ініціалізації Spring boot проекту використано утиліту Spring Initializr, котра забезпечує спрощену конфігурацію проекту. Це можна зробити з середовища розробки IntelliJ IDEA (Рисунок 2.1.1)

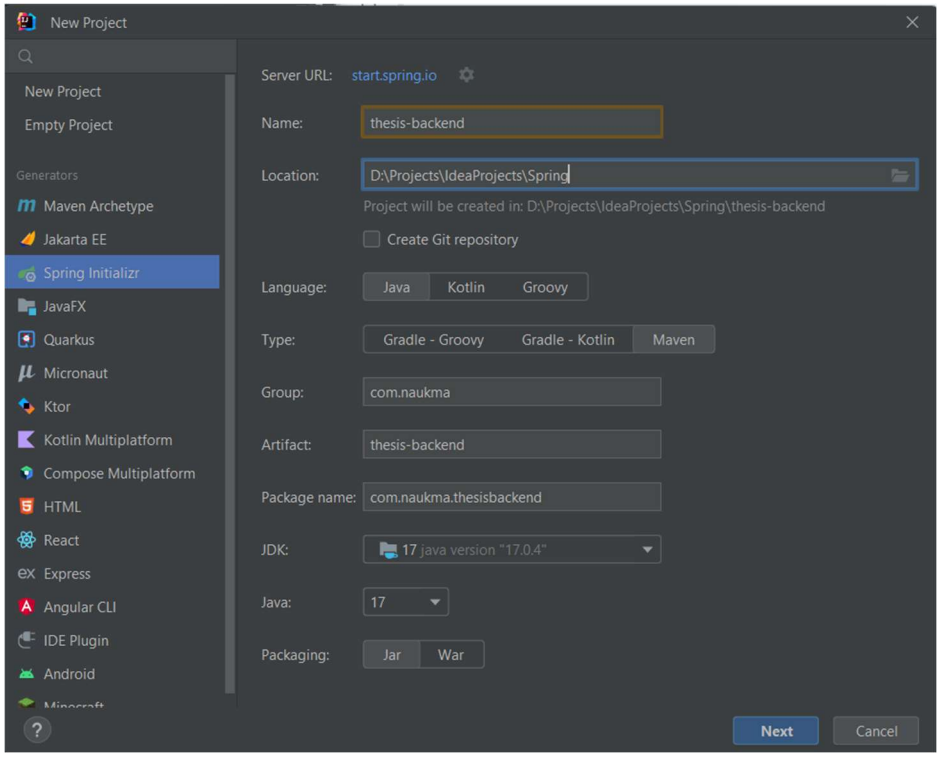

Рисунок 1.1.1

Для зменшення кількості повторів коду, таких як методи getters та setters в проекті використовується бібліотека Lombok.

У цілях спрощення тестування в якості системи керування базами даних (СКБД) використовується H2. Відповідні налаштування з файлу application.yml наведено на Рисунку 2.1.2:

```yaml
spring:
  datasource:
    url: jdbc:h2:file:~/spring-boot-h2-db
    username: u
    password: p
    driverClassName: org.h2.Driver
  jpa:
    database-platform: org.hibernate.dialect.H2Dialect
    hibernate:
      ddl-auto: update
```

Рисунок 2.1.2

2.2 Сутності Spring boot

Сутність – це об’єкт, що являє собою представлення запису з бази даних. Класи сутностей в Spring boot позначаються анотацією @Entity.

Проект вміщує 6 класів сутностей – User, Post, Tag, PostLike, CommentLike. На прикладі класу User продемонструємо процес створення сутності.

Створимо клас User (Рисунок 2.2.1):

```
@Getter
@Setter
@Entity
@Table(name = "app-user")
@NoArgsConstructor
public class User {
```

Рисунок 2.2.1

У цьому прикладі анотації @Getter, @Setter, @NoArgsConstructor належать до бібліотеки Lombok, для створення геттерів, сеттерів та конструктора без параметрів. Анотація @Table додана для зміни назви таблиці. За замовчуванням, назва таблиці у базі даних називається так саме, як і клас відповідної їй сутності, але ім’я User є зарезервованим в СКБД.

```
@Id
@Column(name = "user_id", nullable = false)
private String userId;

1 usage
@Column(name = "nickname", nullable = false)
private String nickname;
```

Рисунок 2.2.2

На рисунку 2.2.2 зображено, як у класі сутності користувача додаються поля. Поле userId є первинним ключем, тож воно має анотацію @Id. Анотація @Column дозволяє змінювати спосіб, у який поле

відображатиметься в базі даних. Вона має набір параметрів, таких як name, nullable, length, unique тощо.

Для збереження ролі користувача використовується enum UserRole, який має 2 значення: USER та ADMIN. Для збереження ролі в базі даних у вигляді рядкового значення, використовується налаштування @Enumerated(EnumType.STRING) (Рисунок 2.2.3)

```
@Column(name = "role", nullable = false)
@Enumerated(EnumType.STRING)
private UserRole role;
```

Рисунок 2.2.3

Двосторонні зв'язки «Один до багатьох» реалізовуються за допомогою анотацій @OneToMany та @ManyToOne. Їх використання на прикладі коментарів та їх автора продемонстровано на рисунку 2.2.4, у класі User, та рисунку 2.2.5, у класі Comment:

```
@OneToMany(fetch = FetchType.LAZY,
        mappedBy = "commentAuthor",
        cascade = CascadeType.ALL,
        orphanRemoval = true)
@JsonIgnore
private List<Comment> comments = new ArrayList<>();
```

Рисунок 2.2.4

```
@ManyToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "user_id", nullable = false)
private User commentAuthor;
```

Рисунок 2.2.5

Завдяки налаштуванню «Fetch = FetchType.LAZY», зв'язана сутність не буде отримана з бази даних одразу разом з основною сутністю, а тільки тоді коли вона знадобиться, що покращує продуктивність програми. Параметр «optional = false» означає, що у коментаря обов'язково повинен бути автор.

Налаштування cascade = CascadeType.ALL значить, що усі операції, проведені над батьківською сутністю будуть розповсюджуватися на дочірні сутності. Таким чином при видаленні користувача будуть видалені усі коментарі, написані ним, а також його пости та вподобання. «orphanRemoval = true» означає, що якщо сутності будуть роз'єднані, то дочірня сутність буде видалена.

Для реалізації зв'язку «Багато до багатьох» у Spring boot існує два різних підходи: через анотацію @ManyToMany та через проміжну таблицю. Зв'язок між таблицями Post та Tag допускає використання першого підходу, оскільки проміжна таблиця між ними не має жодних рядків, окрім зовнішніх ключів postId і tagId.

Реалізація в класі Post (Рисунок 2.2.6):

```
@ManyToMany
@JoinTable(
        name = "post_tag",
        joinColumns = @JoinColumn(name = "post_id"),
        inverseJoinColumns = @JoinColumn(name = "tag_id"))
private List<Tag> tags = new ArrayList<>();
```

Рисунок 2.2.6

Реалізація в класі Tag (Рисунок 2.2.7)

```
@JsonIgnore
@ManyToMany(mappedBy = "tags")
private List<Post> posts = new ArrayList<>();
```

Рисунок 2.2.7

Цей код створює проміжну таблицю «post_tag» автоматично. Анотація @JsonIgnore належить до бібліотеки Jackson. Spring використовує цю бібліотеку під час серіалізації об'єктів. Згадана анотація використовується для полів, котрі не повинні знаходитися у серіалізованому об'єкті.

Другий спосіб потребує створення таблиці власноруч. Цей спосіб буде продемонстровано на прикладі таблиць User, Comment і CommentLike. Оскільки таблиця має два первинних ключі, використовується клас CommentLikeKey з анотацією @Embeddable (Рисунок 2.2.8)

```
@Embeddable
@Getter
@Setter
public class CommentLikeKey implements Serializable {

    3 usages
    @Column(name = "user_id")
    private String userId;

    3 usages
    @Column(name = "comment_id")
    private Long commentId;
```

Рисунок 2.2.8

Цей ключ об’єднує ключі таблиць користувачів та коментарів. Таблиця CommentLike об’єднана з таблицями Comment та User зв’язками @ManyToOne (Рисунок 2.2.9):

```
@EmbeddedId
private CommentLikeKey id;

1 usage
@ManyToOne
@MapsId("userId")
@JoinColumn(name = "user_id")
private User user;

1 usage
@ManyToOne
@MapsId("commentId")
@JoinColumn(name = "comment_id")
private Comment comment;
```

Рисунок 2.2.9

Анотації @EmbededId та @MapsId використовується для налаштування комбінованого ключа.

2.3 Репозиторії

Репозиторії в Spring boot являють собою інтерфейси для взаємодії з базою даних. Вони наслідують інтерфейс Repository. Кожен метод такого класа являє собою конкретний тип запиту до бази даних. Методи репозиторію не реалізовуються розробником, а генеруються автоматично

```
@Repository
public interface CommentRepository extends JpaRepository<Comment, Long> {

    2 usages  AnatoliyAndrus
    Optional<Comment> findCommentByCommentId(Long commentId);
}
```

Рисунок 2.3.2

на основі одного з двох: назви методу або запиту, описаного анотацією @Query [3].

На рисунку 2.3.2 зображено репозиторій CommentRepository та його метод для отримання коментаря за його commentId. Він приймає id коментаря та повертає об’єкт, якщо коментар з таким id існує.

Проект має репозиторії для кожної сутності (Рисунок 2.3.3)

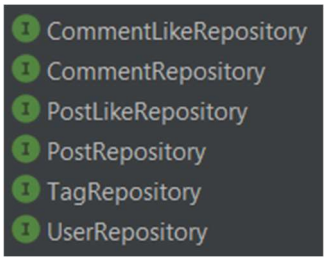

Рисунок 2.3.3

2.4 Впровадження залежностей

Dependency injection, або впровадження залежностей, це важлива техніка, на яку спирається Spring boot. Вона полягає в тому, щоб замість

того, щоб об’єкт брав на себе створення іншого об’єкта, він отримував той з зовнішнього джерела. У Spring boot існують класи, що називаються Beans, створення і керування якими бере на себе Spring ApplicationContext [4]. До них належать компоненти, такі як репозиторії, сервіси, контролери.

Впровадження залежностей можна виконати через конструктор. Детальні приклади будуть продемонстровані в наступних розділах.

2.5 Шаблон проектування DTO

DTO (Data Transfer Object) – це об’єкт, котрий використовується для передачі даних між частинами системи. Це простий Java об’єкт, який не реалізовує жодну бізнес логіку. Він може знадобитися для передачі усіх необхідних даних через один запит до серверу [5]. У цьому проекті він має широке використання, наприклад, CommentDto (Рисунок 2.5.1):

```
public record CommentDto(
    Long commentId,
    Long postId,
    String content,
    String authorUserId,
    String authorNickname,
    boolean edited,
    List<CommentDto> replies,
    Long replyTo,
    int likes,
    boolean isLiked,
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd'T'HH:mm:ss")
    LocalDateTime commentedDate
) {
}
```

Рисунок 2.5.1

2.6 Сервіси

Сервіс у Spring boot – це клас, який відповідає за виконання бізнес логіки. Він не відповідає за пряму роботу з HTTP запитами, але слугують

проміжним шаром між репозиторіями та контролерами. Прикладом слугуватиме UserService (Рисунок 2.6.1)

```
@Service
public class UserService {

    6 usages
    private final UserRepository userRepository;

    AnatoliyAndrus
    public UserService(UserRepository userRepository){
        this.userRepository = userRepository;
    }
```

Рисунок 2.6.1

Сервіси мають анотацію @Service. На рисунку 16 зображено впровадження залежності з UserRepository через конструктор. Завдяки цьому сервіс може використовувати методи репозиторію, наприклад, видалення користувача (Рисунок 2.6.2):

```
public void delete(String userId) {
    userRepository.deleteById(userId);
}
```

Рисунок 2.6.2

2.7 Контролери

Контролер – клас, який відповідає за безпосередню обробку запитів до API.

```
@RestController
@RequestMapping("api/v1/users")
public class UserController {

    new *
    @GetMapping("/{userId}")
    public ResponseEntity<UserBasicInfoDto> getUserBasicInfo(@PathVariable String userId){
        return ResponseEntity.ok(
                userService
                        .getUserById(userId) Optional<User>
                        .orElseThrow(()-> new EntityNotFoundException("No such user")) User
                        .toUserBasicInfoDto());
    }
```

Рисунок 2.7.1

На рисунку 2.7.1 зображений метод з класа UserController, що відповідає за отримання базової інформації юзера. Метод приймає ідентифікатор юзера в якості параметра шляху, за що відповідає аннотація @PathVariable. Він повертає DTO обернений в ResponseEntity. ResponseEntity це зручний клас, що являє собою повну HTTP відповідь, може вміщувати код відповіді, заголовки та тіло відповіді [6]. Метод контролера може приймати різні типи даних та параметри. Для отримання тіла запиту використовують @RequestBody, для отримання параметрів запиту (query parameters), або даних типу multipart/form-data - @RequestParam.

```
@PostMapping("/{userId}/avatar")
public ResponseEntity<?> saveOrUpdateUserAvatar(@PathVariable String userId,
                                                @RequestParam("avatar") MultipartFile avatarFile) throws IOException {
    if(avatarFile == null || avatarFile.getContentType() == null){
        return ResponseEntity.badRequest().body("Invalid input file");
    }
    if (!avatarFile.getContentType().startsWith("image/")) {
        return ResponseEntity.badRequest().body("Only image files are allowed, got " + avatarFile.getContentType());
    }

    User user = userService
            .getUserById(userId)
            .orElseThrow(()-> new EntityNotFoundException("No such user"));

    String newAvatar = avatarService.saveImage(avatarFile);
    String prevAvatar = user.getAvatar();

    user.setAvatar(newAvatar);

    if(prevAvatar!=null){
        avatarService.deleteImage(prevAvatar);
    }

    userService.save(user);

    return ResponseEntity.ok( body: null);
}
```

Рисунок 2.7.2

На рисунку 2.7.2 зображено метод, що обробляє запит на зміну картинки профіля юзера. Він отримує файл картинки, переданий за допомогою типу даних FormData, що є зручним типом для відправки даних, отриманих з форми.

2.8 Spring Security

Spring boot дозволяє детальне налаштування доступу до API. Для цього проекту була обрана автентифікація та авторизація через JWT. JSON Web Token – це спосіб представлення даних автентифікації, що складається з заголовка, що зберігає тип токена та алгоритм хешування; корисне навантаження (payload), що вміщує закодовані дані, та підпис, що перевіряється на справжність за допомогою ключа на сервері [7]. Для роботи з JWT використано бібліотеку java-jwt.

```java
@Service
public class TokenProvider {
    2 usages
    @Value("${security.jwt.token.secret-key}")
    private String JWT_SECRET;

    1 usage  AnatoliyAndrus
    public String generateAccessToken(UserDetails user) {
        try {
            Algorithm algorithm = Algorithm.HMAC256(JWT_SECRET);
            return JWT.create()
                    .withSubject(user.getUsername())
                    .withClaim( name: "username", user.getUsername())
                    .withExpiresAt(generateAccessExpirationDate())
                    .sign(algorithm);
        } catch (JWTCreationException exception) {
            throw new JWTCreationException("Error while generating token", exception);
        }
    }

    1 usage  new *
    private Instant generateAccessExpirationDate() {
        return LocalDateTime.now().plusDays(2).toInstant(ZoneOffset.of( offsetId: "+03:00"));
    }
```

Рисунок 2.8.1

Метод generateAccessToken (Рисунок 2.8.1) використовується для створення JWT токена для користувача. Цей токен закінчує термін дії через 2 дні після генерації, таким чином користувач мусить періодично проходити операцію авторизації повторно.

З Spring Security кожен запит, отриманий сервером, спочатку проходить ряд фільтрів (Рисунок 2.8.2):

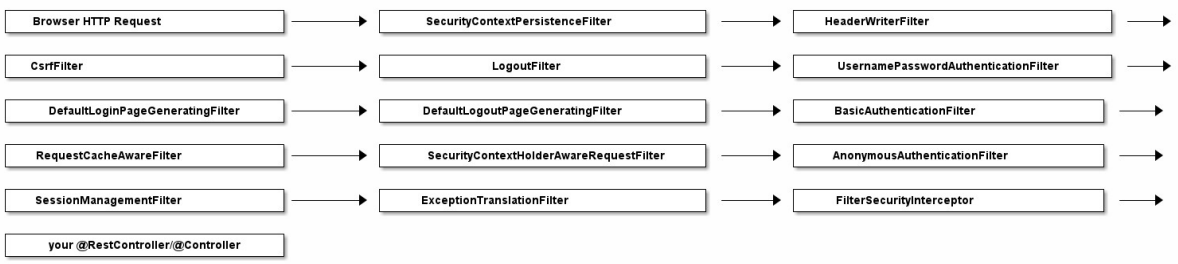

Рисунок 2.8.2

Для проходження авторизації/автентифікації створимо власний фільтр. Це клас CustomSecurityFilter, що наслідує абстрактний клас OncePerRequestFilter, що перевизначає метод doFilterInternal (Рисунок 2.8.3):

```
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
    var token :String = this.recoverToken(request);
    if (token != null) {
        var userId :String = tokenProvider.validateToken(token);

        if(userId!=null) {
            var user :Optional<User> = userRepository.findByUserId(userId);

            if (user.isPresent()) {
                var userDetails = new CustomUserDetails(user.get());
                var authentication = new UsernamePasswordAuthenticationToken(userDetails,
                        credentials: null,
                        userDetails.getAuthorities());
                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        }
    }
    filterChain.doFilter(request, response);
}
```

Рисунок 2.8.3

Цей метод отримує токен з заголовку запиту, отримує з нього закодований userId, перевіряє, чи такий юзер є в базі даних, і додає його в контекст безпеки. SecurityContextHolder призначений для збереження інформації про автентифікованого користувача [8]. Він може бути отриманий з будь-якої точки застосунку. Завдяки інформації автентифікованого користувача за допомогою конфігурації безпеки можна налаштувати захист конкретних шляхів у застосунку через Security Filter Chain (Рисунок 2.8.4).

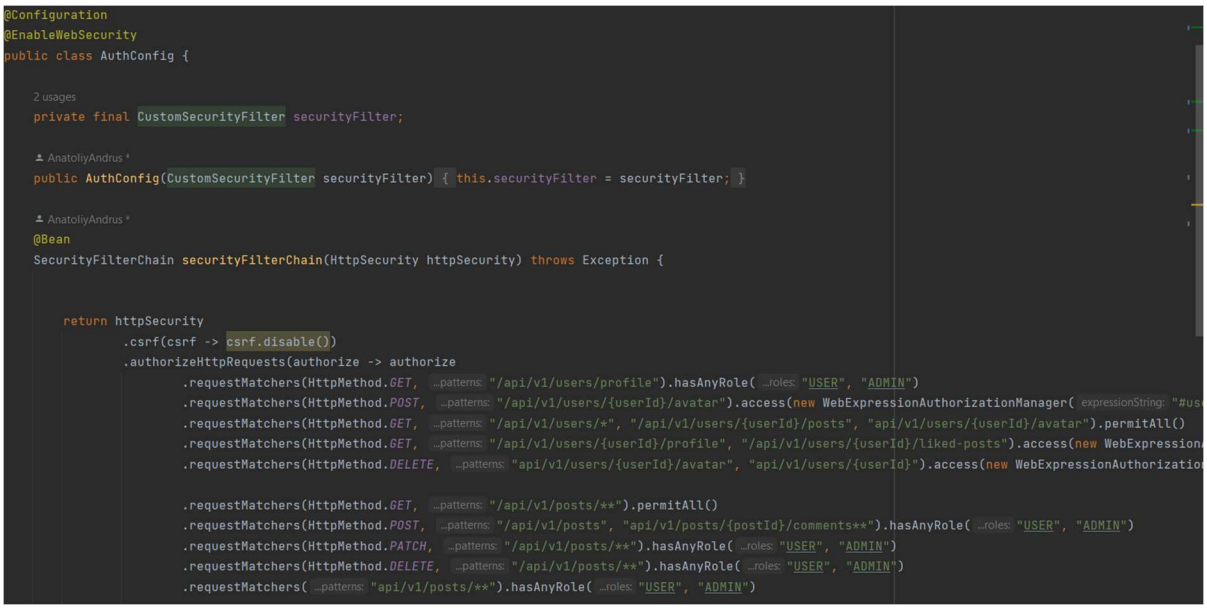

Рисунок 2.8.4

Для отримання запитів на реєстрацію/логін створимо AuthController та відповідні методи signUp, signId. Метод реєстрації лише зберігає користувача в базі даних, тоді як signId перевіряє, чи такий користувач існує, після чого генерує токен для нього та повертає. В подальшому для авторизації/автентифікації користувач мусить додавати цей токен до заголовку запиту «Authorization» (Рисунок 2.8.5).

Рисунок 2.8.5

2.9 Загальний опис API

У цьому розділі описані усі шляхи API, що має застосунок, з їх HTTP методами

- [POST] /auth/sign-up реєстрація користувача
- [POST] /auth/sign-in логін користувача
- [POST] /posts створення поста
- [GET] /posts отримання сторінки фільтрованих постів
- [PATCH] /posts/{postId} оновлення поста
- [DELETE] /posts/{postId} оновлення поста
- [GET] /posts/{postId} отримання поста
- [POST] /posts/{postId}/comments створення коментаря до поста
- [GET] /posts/{postId}/comments отримання усіх коментарів до поста
- [PATCH] /posts/{postId}/toggle-like вподобання поста
- [GET] /comments/{commentId} отримання коментаря
- [PATCH] /comments/{commentId} оновлення коментаря
- [DELETE] /comments/{commentId} видалення коментаря
- [PATCH] /comments/{commentId}/toggle-like вподобання коментаря
- [GET] /users/profile отримання власної інформації користувача
- [DELETE] /users/{userId} видалення юзера
- [POST] /users/{userId}/avatar збереження картинки профіля
- [GET] /users/{userId}/avatar отримання картинки профіля
- [DELETE] /users/{userId}/avatar видалення картинки профіля
- [GET] /users/{userId}/liked-posts отримання постів, вподобаних користувачем
- [GET] /users/{userId}/posts отримання постів за авторством користувача

- [GET] /users/{userId}/profile отримання повної інформації користувача
- [POST] /tags створення тегу
- [GET] /tags отримання усіх тегів
- [GET] /tags/{tagId} отримання конкретного тегу
- [PATCH] /tags/{tagId} зміна тегу
- [DELETE] /tags/{tagId} видалення тегу

3. Розробка frontend застосунку

Для розробки frontend застосунку для цієї роботи використовується бібліотека React. React засновується на ідеї компонентів, що можуть бути використані декілька разів. React дозволяє зручне керування станом компонентів. Створені компоненти можна використовувати як html теги в інших компонентах.

Існують два варіанти створення компонентів у React: функціональні компоненти та класові компоненти. Життєвий цикл функціонального компоненту керується за допомогою функцій hooks, тоді як у класових компонентах використовуються методи життєвого циклу. Класові компоненти вважаються менш ефективними та більш складними у використанні, через що частіше використовуються функціональні [9].

В основі React лежить ідея односторінкового застосунку (single page application, SPA). На відміну від традиційного багатосторінкового застосунку, котрий може складатися з багатьох html сторінок, котрі завантажуються на вимогу користувача, увесь необхідний код для SPA завантажується одразу. Односторінковий застосунок не перезавантажується у відповідь на дії користувача, що позитивно впливає на досвід користування (user experience) [10].

3.1 Створення проекту

Для створення проекту використано утиліту vite, яка виконує початкові налаштування. Для створення проекту з останньою версією необхідно ввести в командний рядок команду «npm create vite@latest».

3.2 Налаштування шляхів

Для маршрутизації у React використовується бібліотека react-router-dom. У нашому випадку, шляхи та компоненти, що відображаються при переході за цими шляхами, визначаються в компоненті App за допомогою компонентів Routes, Route (рисунок 3.2.1):

```
<Routes>
    <Route index element={<InfinitePostsScrollerPage/>}></Route>
    <Route path="home" element={<InfinitePostsScrollerPage/>}></Route>
    <Route path="register" element={<RegisterForm/>}></Route>
    <Route path="login" element={<LoginForm/>}></Route>
    <Route path="create-post" element={<CreatePost/>}></Route>
    <Route path="posts/:postId" element={<PostPage/>}></Route>
    <Route path="profiles/:userId/liked-posts" element={<UserLikedPostsPage/>}></Route>
    <Route path="profiles/:userId/posts" element={<UserPostsPage/>}></Route>
    <Route path="profiles/:userId" element={<UserProfile currentUser={currentUser}/>}>
    </Route>
    <Route path="*" element={<NoMatch/>}></Route>
</Routes>
```

Рисунок 3.2.1

Наприклад, при переході за шляхом /register, відобразиться компонент RegisterForm, а за шляхом profiles/someUserId/posts - усі пости конкретного користувача.

3.3 Робота з API сервера

Для надсилання запитів до backend застосунка використано бібліотеку axios. Усі запити реалізуються в сервісах, тобто JavaScript файлах, котрі експортують усі необхідні функції.

```
export function getUserBasicInfo(userId){
    return api.get(
        url: `users/${userId}`
    )
}
```

Рисунок 3.3.1

На рисунку 3.3.1 зображений запит, що повертає інформацію користувача за його userId. Тут api – це екземпляр axios з налаштованим базовим шляхом до сервера (Рисунок 3.3.2):

```
const api = axios.create({
    baseURL: backendBaseLink,
});
```

Рисунок 3.3.2

3.4 Компоненти

Для роботи з автентифікацією та авторизацією користувача використовуються функції з файлу authService.js та компоненти LoginForm та RegisterForm.

Hooks, або хуки, це функції в React, що використовуються для керування станом компоненту та розширенням його базового функціоналу.

У компоненті форми реєстрації хук useState зв’язує дані з полів форми та об’єкт formData

```
const [formData, setFormData] = useState( initialState: {
    userId: '',
    nickname: '',
    email: '',
    password: ''
});
```

Рисунок 3.4.1

Наприклад, поле форми userId пов’язано з відповідним значенням через наступні параметри (Рисунок 3.4.2):

```
<input
    value={formData.userId}
    onChange={handleChangeField}
```

Рисунок 3.4.2

Завдяки параметру value у полі вводу завжди відображається значення formData.userId. onChange відповідає за дію, що виконується, коли змінюється наповнення поля вводу, і пов’язане з функцією handleChangeField (рисунок 3.4.3):

```
const handleChangeField = (e) => {
    const {name, value} = e.target;
    setFormData( value: {
        ...formData,
        [name]: value
    });
};
```

Рисунок 3.4.3

Таким чином, при зміні значення поля, його значення в об’єкті теж змінюється.

При натисненні на кнопку відправки форми, викликається функція handleFormSubmit (Рисунок 3.4.4). Вона надсилає запит на сервер, після чого, якщо відповідь успішна, за допомогою хука useNavigate перенаправляє користувача на шлях /login. У випадку невдачі вона показує користувачеві сповіщення з текстом помилки за допомогою бібліотеки react-toastify.

```
const handleFormSubmit = (e) => {
    e.preventDefault();

    authService
        .signUp(formData)
        .then(res => {
            if(res.success){
                navigate("/login")
            }else{
                toast(res.message)
            }
        })
      };
```

Рисунок 3.4.4

LoginForm має 2 поля: userId та password. При відправленні форми виконується запит на сервер, після кого у випадку успіху отриманий з сервера токен зберігається у localStorage. LocalStorage – це механізм зберігання даних у браузері користувача упродовж деякого часу. Завдяки тому, що значення у LocalStorage зберігаються між перезавантаженнями сторінки та повторними відвідування сайту, користувач не мусить входити в систему на кожному відвідуванні.

Для відображення постів та їх фільтрації використовується компонент InfinitePostsScrollerPage. Цей компонент використовує запит до серверу, котрий повертає одну сторінку постів. Це зроблено для того, щоб у теоретичному випадку відповідь не мала забагато постів, що могло би призвести до надмірного використання трафіку. Для відображення постів використовується компонент InfiniteScroll з бібліотеки react-infinite-scroll-component. Ціль цього компоненту – відображення даних порціями, коли користувач прокручує сторінку до низу, виконується функція з параметру next, для отримання нової порції (Рисунок 3.4.5).

```
<InfiniteScroll
    next={getNextData}
    hasMore={hasMorePosts}
    loader={<CircleLoader/>}
    dataLength={posts.length}
    endMessage={<h3>No more posts found</h3>}
>
    {posts.map(post => <Post key={post.postId} postData={post}></Post>)}
</InfiniteScroll>
```

Рисунок 3.4.5

Функція getNextData відповідає за завантаження наступної порції постів з серверу (Рисунок 3.4.6). Вона використовує вибрані фільтри, сторінку, та розмір сторінки для запиту. Отримані пости вона додає до списку вже існуючих.

```
const getNextData = () => {
    fetchPostsByFilters(
        filters: {authorId, minDate, maxDate, title, tagIds: selectedTags, page: page + 1, size}
    ).then(res => {
        if (res.data.last) setHasMorePosts( value: false)
        setPosts( value: posts => posts.concat(res.data.content))
    })
    setPage( value: prev => prev + 1)
};
```

Рисунок 3.4.6

Компонент Post відображає один пост за його даними, такими як заголовок, наповнення, ім’я автору, кількістю вподобань тощо.

Компонент CreatePost являє собою форму створення поста. В цій формі вводиться заголовок, контент та теги поста. Після створення поста користувач автоматично переноситься на шлях /home.

Компонент PostPage відповідає за відображення сторінки поста, що разом з постом відображає також коментарі до нього. На цій сторінці користувач може написати коментар до поста або до іншого коментаря. Для відображення коментарів використано компонент Comment

Comment – це компонент, що відображає коментар разом з його дочірніми коментарями. Для того, щоби відобразити окрім коментаря також відповіді на нього, потрібно використати компонент Comment рекурсивно в середині власного методу return (Рисунок 3.4.7):

```
{commentData.replies.length > 0 && (
    <div className="comment-replies">
        {commentData.replies.map((reply) => (
            <div key={reply.commentId}>
                <Comment commentData={reply}/>
            </div>
        ))}
    </div>
)}
```

Рисунок 3.4.7

Таким чином, якщо commentData.length непустий, то всередині коментаря відображаються також відповіді на нього.

Компоненти UserLikedPostsPage та UserPostsPage відповідають за відображення списків улюблених постів користувача та постів за авторством користувача відповідно. Використовують компонент Post для відображення.

UserProfile використовується для відображення профілю користувача. У випадку, коли відображуваний користувач співпадає з поточним користувачем, його профіль також буде мати кнопку оновлення картинки профіля, і відобразить email користувача.

3.5 Результуючий сайт

Цей розділ містить зображення веб-застосунку, що був отриманий у процесі розробки.

Форми реєстрації/входу в систему зображені на рисунку 3.5.1

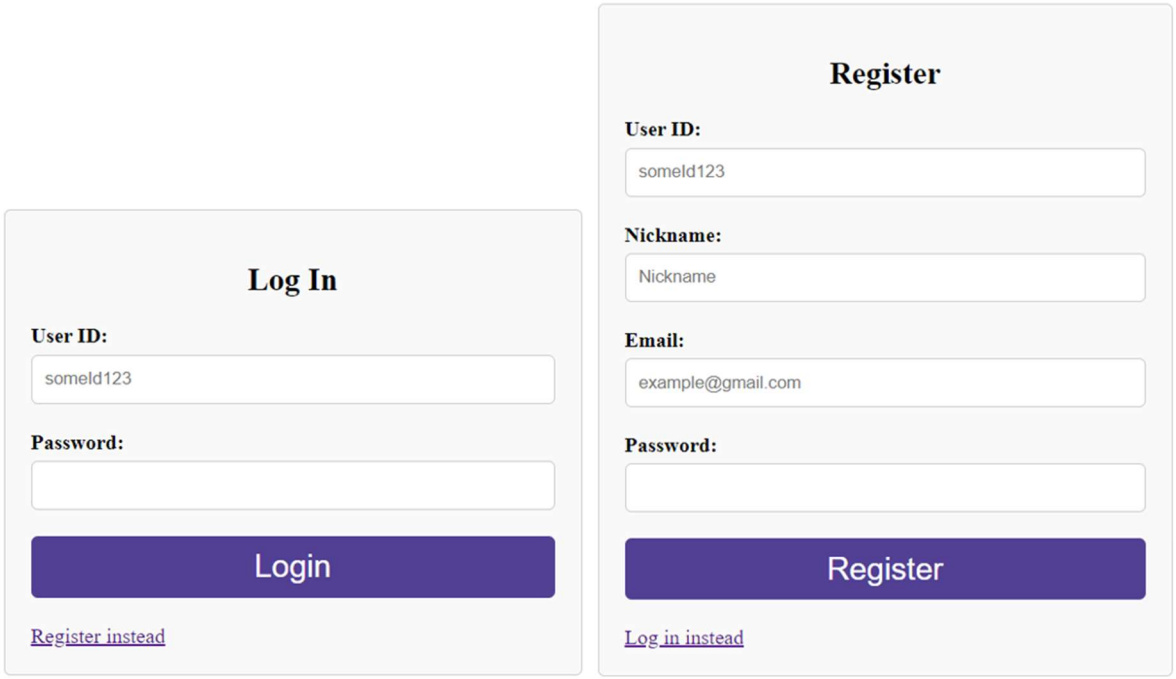

Рисунок 3.5.1

Сторінка home, що відображує відфільтровані пости зображена на рисунку 3.5.2. На ній можна відфільтрувати пости за авторством, назвою, часом створення, тегами.

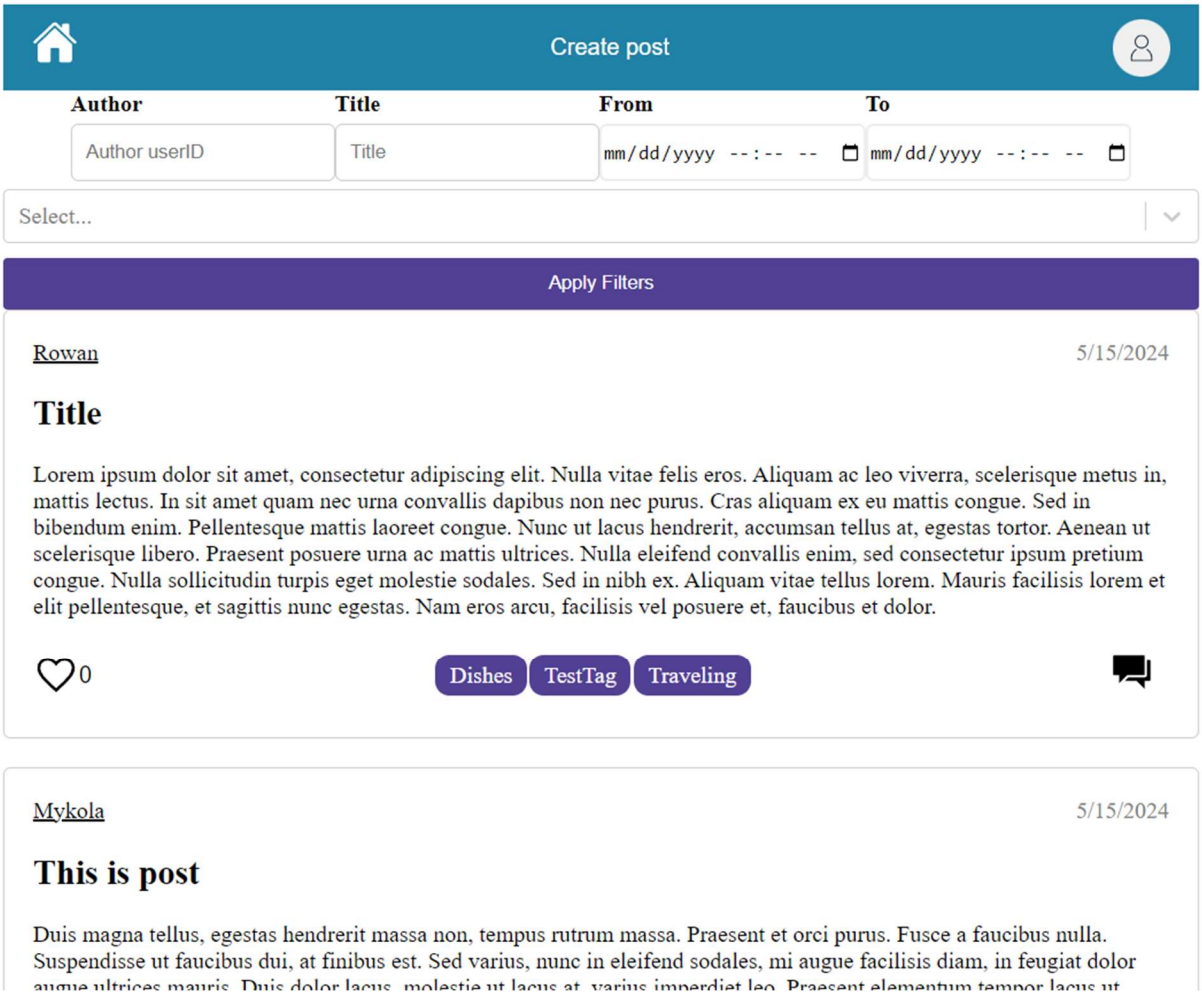

Рисунок 3.5.2

При натисненні на іконку коментаря на пості, відбувається перехід на сторінку цього поста. Ця сторінка вміщує пост, а також коментарі до нього (Рисунок 3.5.3).

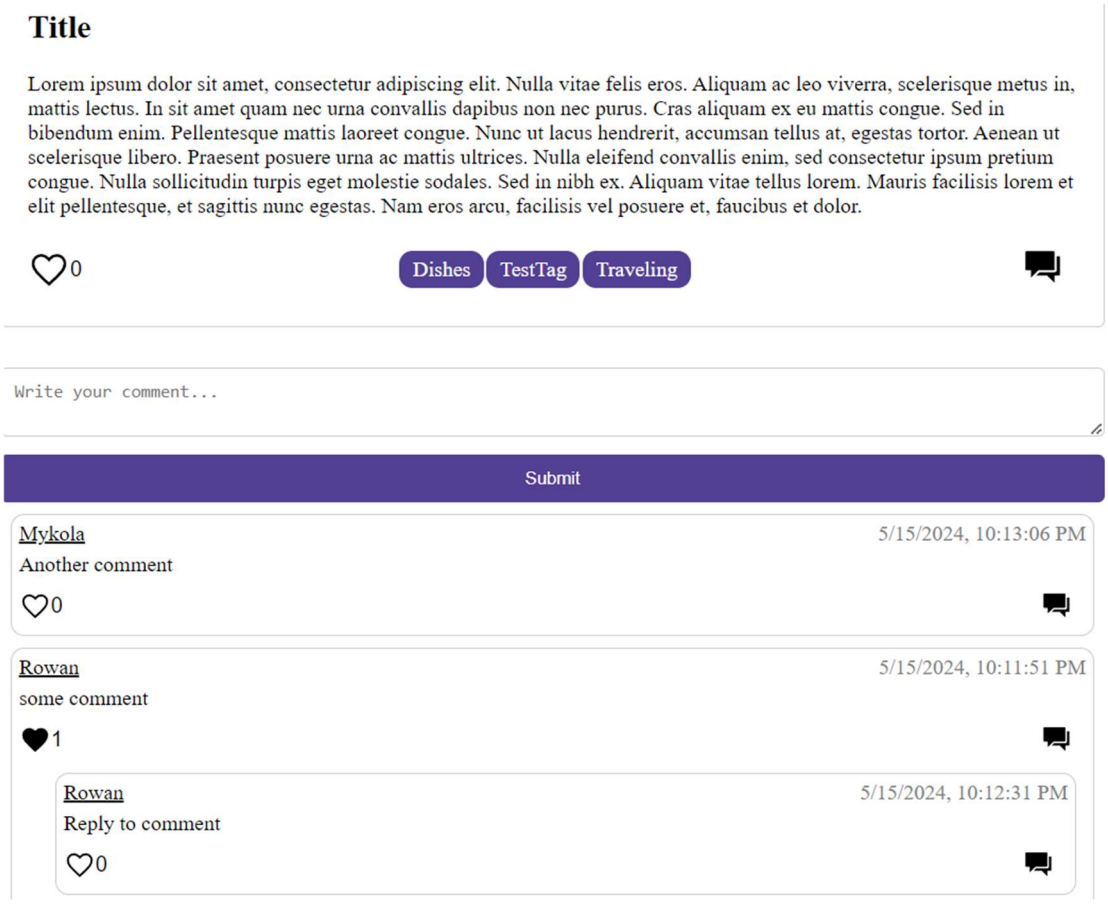

Рисунок 3.5.3

Сторінка створення поста, на якій авторизований користувач може створити пост, ввівши назву, наповнення та його теги, зображена на рисунку 3.5.4:

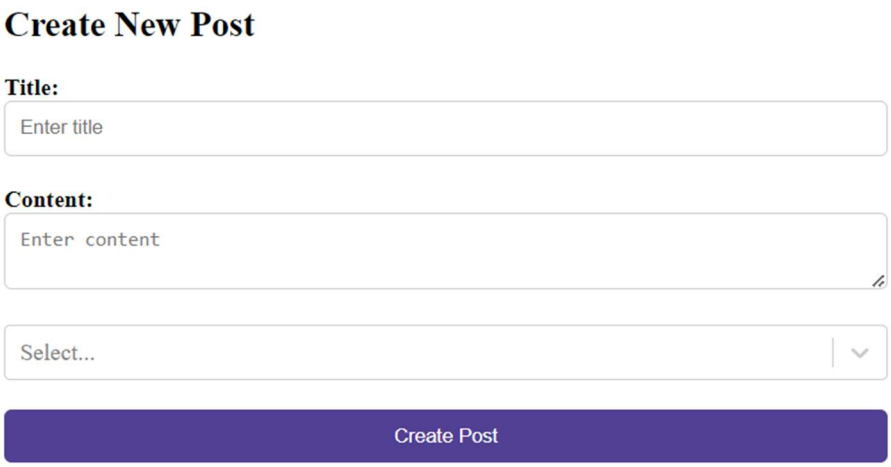

Рисунок 3.5.4

Висновки

Ця курсова робота є результатом дослідження процесу розробки веб застосунка, його етапів. Були вирішені задачі, що зазвичай супроводжують таку розробку, такі як дизайн бази даних, розробка захищеного та ефективного RESTful API, налаштування безпеки застосунку, розробка користувацького інтерфейсу, використання API на клієнтській стороні.

Використані у процесі розробки технології пропонують сучасні та ефективні рішення поставлених задач.

Простий застосунок-соцмережа, створений в процесі, реалізує такий функціонал, як:

- Реєстрація/вхід користувача
- Написання постів/коментарів до постів
- Вподобання постів/коментарів
- Перегляд постів за фільтрами, перегляд постів що сподобалися, перегляд постів за авторством користувача
- Відображення користувацького профілю, зміна картинки профілю

Застосунок має достатньо вдалу архітектуру, що дозволяє легке масштабування у разі необхідності.

Список літератури

1. [Електронний ресурс] – Режим доступу: https://spring.io/projects/spring-boot#overview
2. Kaluža M. A comparison of back-end frameworks for web application development [Електронний ресурс] / M. Kaluža, M. Kalanj, B. Vukelić // Journal of the Polytechnic of Rijeka, Vol. 7 No. 1. – 2019. – Режим доступу до ресурсу: https://hrcak.srce.hr/en/clanak/321176%3F.
3. [Електронний ресурс] – Режим доступу: https://docs.spring.io/spring-data/jpa/reference/repositories/query-methods-details.html
4. [Електронний ресурс] – Режим доступу: https://www.baeldung.com/spring-application-context
5. [Електронний ресурс] – Режим доступу: https://www.baeldung.com/java-dto-pattern
6. [Електронний ресурс] – Режим доступу: https://www.baeldung.com/spring-response-entity
7. [Електронний ресурс] – Режим доступу: https://jwt.io/
8. [Електронний ресурс] – Режим доступу: https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html
9. Anggraini D. Modern Front End Web Architectures with React.Js and Next.Js [Електронний ресурс] / Dyah Anggraini // International Research Journal of Advanced Engineering and Science. – 2022. – Режим доступу до ресурсу: http://irjaes.com/wp-content/uploads/2022/02/IRJAES-V7N1P162Y22.pdf.
10. Erolin J. React Single Page Application [Електронний ресурс] / Justice Erolin – Режим доступу до ресурсу: https://www.bairesdev.com/blog/react-spa-single-page-application/.