Вінницький національний технічний університет
(повне найменування вищого навчального закладу)

Факультет інформаційних технологій та комп'ютерної інженерії
(повне найменування інституту, назва факультету (відділення))

Кафедра обчислювальної техніки
(повна назва кафедри (предметної, циклової комісії))

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Серверна частина соціальної мережі»

08-54.БКР.022.00.000 ПЗ

Виконав: студент 2 курсу, групи 2КІ-23мс
спеціальності 123 — Комп'ютерна інженерія
(шифр і назва напряму підготовки, спеціальності)
_____Вовчук М. О.
(підпис)

Керівник ст. викл. каф. ОТ
_____Кисюк Д. В.
(підпис)
«_____» _____2025 р.

Рецензент: д.т.н., проф., зав. каф. ПЗ
_____Романюк О.Н.
(підпис)
«_____» _____2025 р.

Допущено до захисту
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.
«10» 06 2025 р.

Вінниця ВНТУ — 2025 рік

ВІННИЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Освітньо-кваліфікаційний рівень бакалавр

Галузь знань — 12 «Інформаційні технології»

Спеціальність — 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

д.т.н., проф. Азаров О. Д.

«21» березня 2025 р.

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Вовчуку Максиму Олеговичу

1 Тема роботи: «Серверна частина соціальної мережі», керівник роботи Кисюк Д.В.. ст. викл. каф. ОТ, затверджена наказом вищого навчального закладу від 20 березня 2025 р. №97.

2 Термін подання студентом роботи 13.05.2025 р.

3 Вихідні дані до роботи: Призначення — розробка серверної частини програмного забезпечення для соціальної мережі; основний функціонал: реєстрація/автентифікація користувачів, створення постів, коментування, система підписок, взаємодія з медіа.

4 Зміст розрахунково-пояснювальної записки: вступ, аналітичний огляд теорії та практики, технологічна реалізація, тестування, безпека, документація API, висновки, список використаних джерел, додатки.

5 Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): структура класів програмного засобу, скріншоти інтерфейсу програмного засобу.

6 Консультанти розділів роботи наведені в таблиці 1.

Таблиця 1 — Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Розділ 1–3	с. т. викл. каф. ОТ Кисюк Д.В.	16.03.2025	18.05.2025
Нормокотроль	ас. каф. ОТ Швець С. І.	14.05.25	30.05.25

7 Дата видачі завдання 12.03.2025 р.

8 Календарний план виконання БДР наведено в таблиці 2.

Таблиця 2 — Календарний план

№ етапу	Назва етапу	Термін виконання		Підпис
		Початок	Кінець	
1	Постановка задачі	21.03.25	21.03.25	вик
2	Аналіз соціальних мереж	21.03.25	30.03.25	вик
3	Технології розробки	21.03.25	30.04.25	вик
4	Розробка серверної частини мережі	31.03.25	13.04.25	вик
5	Реалізація	14.04.25	27.04.25	вик
6	Підготовка тезових матеріалів для публікації	21.04.25	28.04.25	вик
7	Оформлення пояснювальної записки та демонстраційної презентації для доповіді	29.04.25	12.05.25	вик
8	Підготовка супровідної документації, перевірка відповідності нормам та проходження тесту на плагіат	13.05.25	13.05.25	вик

Студент Вовчук М.О.

Керівник роботи ст. викл. каф. ОТ Кисюк Д. В.

АНОТАЦІЯ

УДК 621.3

Вовчук М.О. Серверна частина соціальної мережі з використанням Java та Spring Framework. Бакалаврська кваліфікаційна робота зі спеціальності 123 «Комп'ютерна інженерія», освітня програма «Комп'ютерна інженерія». Вінниця: ВНТУ, 2025. 77 с.

На укр. мові. Бібліогр.: 25; рис.: 6; табл. 2.

У бакалаврській кваліфікаційній роботі розроблено серверну частину соціальної мережі з використанням мови програмування Java та фреймворку Spring Boot. Реалізовано REST API для авторизації користувачів, створення та взаємодії з публікаціями, підписок, коментування, а також обробки медіа-контенту. Використано Spring Security для захисту даних і авторизації через JWT, PostgreSQL як систему керування базами даних, та Swagger для генерації технічної документації. Система протестована локально, спроєктована з урахуванням масштабованості та можливості подальшого розширення функціоналу. Розробка проводилась в середовищі IntelliJ IDEA.

Ключові слова: соціальна мережа, Java, Spring Boot, REST API, PostgreSQL, JWT, Spring Security, Swagger.

ABSTRACT

Vovchuk M.O. Backend of a Social Network Using Java and Spring Framework. Bachelor qualification work on specialty 123

«Computer Engineering», educational program «Computer Engineering». Vinnytsia: VNTU, 2025. 77 p.

In Ukrainian. Bibliography: 25; Figures: 6; Tables: 2.

In the bachelor's qualification work, a backend for a social networking application was developed using the Java programming language and the Spring Boot framework. A REST API was implemented to handle user authentication, creation and interaction with posts, subscriptions, comments, and multimedia content. Spring Security was used to protect data and provide JWT-based authorization. PostgreSQL was used as the database management system, and Swagger was applied for API documentation generation. The system was locally tested and designed with scalability and extensibility in mind. Development was carried out in IntelliJ IDEA.

Keywords: social network, Java, Spring Boot, REST API, PostgreSQL, JWT, Spring Security, Swagger.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

API — Application Programming Interface

REST — Representational State Transfer

JSON — JavaScript Object Notation

HTTP — HyperText Transfer Protocol

HTTPS — HyperText Transfer Protocol Secure

JWT — JSON Web Token

SQL — Structured Query Language

CRUD — Create, Read, Update, Delete

ORM — Object-Relational Mapping

JPA — Java Persistence API

JVM — Java Virtual Machine

IDE — Integrated Development Environment

AWS — Amazon Web Services

S3 — Simple Storage Service

SSL/TLS — Secure Sockets Layer/Transport Layer Security

CORS — Cross-Origin Resource Sharing

CSRF — Cross-Site Request Forgery

XSS — Cross-Site Scripting

BCrypt — Blowfish Crypt

SMTP — Simple Mail Transfer Protocol

DTO — Data Transfer Object

MVC — Model-View-Controller

ACID — Atomicity, Consistency, Isolation, Durability

Docker — Docker Containerization Platform

Redis — Remote Dictionary Server

Swagger — API Documentation Tool

JUnit — Java Unit Testing Framework

Spring Boot — Spring Boot Framework

ЗМІСТ

ВСТУП

У сучасному світі цифрові технології стали невід'ємною частиною повсякденного життя, а соціальні мережі — одним з основних інструментів комунікації між людьми. Вони дозволяють підтримувати зв'язок з друзями, ділитись новинами, брати участь у спільнотах за інтересами та отримувати актуальну інформацію. Популярність соціальних мереж зумовлює необхідність постійного вдосконалення їх функціональності, продуктивності та безпеки. Ключову роль у цьому відіграє серверна частина, яка відповідає за обробку запитів, зберігання даних, автентифікацію користувачів, забезпечення взаємодії між клієнтами та масштабованість системи.

Актуальність розробки полягає в необхідності створення надійних серверних рішень для соціальних мереж у зв'язку з експоненційним зростанням кількості користувачів та обсягів оброблюваної інформації. Розробка ефективної серверної архітектури дає можливість забезпечити стабільну роботу платформи під високим навантаженням, ефективне управління великими масивами даних та високий рівень захисту особистої інформації користувачів. Сучасні серверні рішення повинні підтримувати мільйони одночасних користувачів, обробляти терабайти контенту та гарантувати доступність сервісу 24/7.

У наукових публікаціях та дослідженнях наголошується на значущості застосування сучасних технологій, таких як мікросервісна архітектура, REST API, JWT-автентифікація та реляційні бази даних. Попередні роботи показують ефективність використання Spring Framework для побудови enterprise-рівня додатків, PostgreSQL для надійного зберігання даних та контейнеризації для забезпечення масштабованості системи.

Мета бакалаврської кваліфікаційної роботи полягає у підвищенні масштабованості та надійності серверної системи соціальної мережі за допомогою використання сучасних технологій та архітектурних підходів, що забезпечують високу продуктивність, надійність та безпеку, через проєктування багатошарової архітектури на базі Spring Boot, реалізацію RESTful API для клієнт-серверної взаємодії, впровадження JWT-автентифікації та авторизації, створення

оптимізованої структури PostgreSQL бази даних, а також налаштування системи моніторингу та розгортання.

Необхідні завдання для вирішення поставленої мети БКР:

— провести аналіз сучасних архітектурних підходів до побудови серверної частини веб-застосунків соціальних мереж, зокрема монолітної та мікросервісної архітектури;

— спроєктувати багатошарову архітектуру серверної частини соціальної мережі із чітким розподілом логіки між її компонентами;

— реалізувати RESTful API для забезпечення ефективної взаємодії між клієнтською та серверною частинами системи;

— впровадити механізм автентифікації та авторизації користувачів з використанням JWT для захисту особистих даних та контролю доступу;

— розробити та оптимізувати структуру бази даних PostgreSQL для ефективного зберігання та обробки інформації користувачів і контенту;

— налаштувати контейнеризоване середовище за допомогою Docker для забезпечення гнучкого розгортання та масштабування системи;

Об'єкт дослідження — це процес розробки та функціонування серверної частини веб-застосунків соціальних мереж.

Предметом дослідження є технології проєктування, реалізації та забезпечення безпеки серверної частини соціальної мережі з використанням Java Spring Framework, PostgreSQL та сучасних DevOps практик.

Новизна результатів полягає у розробці комплексного підходу до створення серверної архітектури соціальної мережі, що поєднує сучасні технології Spring Boot, JWT-автентифікацію, оптимізовану структуру бази даних PostgreSQL та контейнеризацію для забезпечення високої масштабованості та надійності системи.

Практичне застосування — розроблена серверна частина може бути використана як основа для створення повноцінної соціальної мережі, впровадження в існуючі веб-проєкти або як навчальний матеріал для вивчення сучасних підходів до backend розробки.

1 АНАЛІЗ СОЦІАЛЬНИХ МЕРЕЖ ТА ТЕХНОЛОГІЙ ЇХ РОЗРОБКИ

1.1 Розвиток соціальних мереж

Соціальні мережі є ключовим елементом сучасного цифрового суспільства, що фундаментально змінили способи людської комунікації, обміну інформацією та формування соціальних зв'язків. Їхній розвиток відображає не лише прогрес інформаційних технологій, але й еволюцію соціальних потреб, культурних змін та економічних моделей цифрової епохи.

У 1970-х роках перші зародки соціальних мереж з'явилися у вигляді електронних дошок оголошень (BBS — Bulletin Board Systems), які дозволяли користувачам обмінюватися повідомленнями через телефонні лінії. Ці системи, хоча й примітивні за сучасними стандартами, заклали фундаментальні принципи створення цифрових спільнот, асинхронного спілкування та можливості поділитися інформацією з невідомими людьми. Технічно BBS працювали на основі dial-up з'єднань, що обмежувало кількість одночасних користувачів та швидкість обміну даними, але створило перші прецеденти онлайн-взаємодії [1].

Розвиток мережі ARPANET у 1980-х роках та поява World Wide Web у 1990-х створили технологічну основу для масових соціальних платформ. Перші веб-форуми та чати (IRC) продемонстрували потенціал мережевого спілкування, але все ще були орієнтовані на технічно грамотних користувачів та специфічні інтереси.

У 1990-х роках SixDegrees.com (1997) став революційним сервісом, що вперше запровадив концепцію персональних профілів і візуальних зв'язків між користувачами. Платформа дозволяла користувачам створювати детальні профілі з фотографіями, особистою інформацією та списками друзів. Назва сервісу базувалася на теорії "шести ступенів розподілу" соціолога Стенлі Мілгрема, згідно з якою будь-які дві людини на планеті з'єднані ланцюжком не більше ніж з шести знайомств. Незважаючи на інноваційність концепції, SixDegrees не досягла комерційного успіху через обмежену інтернет-аудиторію того часу та відсутність широкосмугового доступу, що змушувало користувачів чекати хвилини для

завантаження однієї фотографії.

На початку 2000-х років з'явилися платформи, що кардинально розширили функціональність та аудиторію. Friendster (2002) вперше досягла мільйона користувачів, запровадивши концепцію "друзів друзів" та можливість знайомитися з людьми через взаємних знайомих. Однак технічні проблеми з масштабуванням серверів призвели до повільного завантаження сторінок та втрати користувачів. MySpace (2003) став першою справді масовою соціальною мережею, досягнувши 100 мільйонів користувачів до 2006 року. Революційність MySpace полягала в можливості повної персоналізації профілів: користувачі могли змінювати дизайн, додавати HTML-код, вставляти музику та відео. Це зробило MySpace особливо популярним серед молоді та музикантів, які використовували платформу для промоції своєї творчості [2].

Запуск Facebook у 2004 році став переломним моментом в історії соціальних мереж. Марк Цукерберг та команда Harvard студентів створили платформу, що поєднувала простоту використання з потужним функціоналом. На відміну від хаотичної персоналізації MySpace, Facebook запропонував чіткий, уніфікований інтерфейс, що зосереджувався на контенті, а не на дизайні. Ключові інновації Facebook включали концепцію "стіни" — централізованого місця для публікації оновлень, систему коментарів та вподобань, що створювала інтерактивність навколо контенту. У 2006 році Facebook запровадив News Feed — алгоритмічно організовану стрічку новин, що автоматично агрегувала оновлення від друзів.

У 2010-х роках соціальні мережі почали спеціалізуватися, з платформами, що зосереджувались на специфічних потребах. Twitter (2006) революціонізував мікроблогінг, обмеживши повідомлення до 140 символів, що змусило користувачів бути лаконічними та сприяло швидкому поширенню новин. Twitter став платформою для громадських рухів, політичних дискусій та реального часу репортажів з подій. LinkedIn (2003) захопив професійну нішу, створивши цифровий еквівалент мережевого спілкування в бізнес-середовищі, інтегрувавши функції резюме, пошуку роботи та професійного нетворкінгу. Instagram (2010) зосередився виключно на візуальному контенті, спочатку пропонуючи лише квадратні

фотографії з фільтрами, але простота та акцент на естетиці швидко привабили аудиторію [25].

Нове покоління платформ запровадило революційні концепції, що змінили правила гри. Snapchat (2011) популяризував ефемерний контент — повідомлення, фотографії та відео, які автоматично зникають після перегляду, знизивши психологічний бар'єр для публікації та створивши відчуття приватності. TikTok (2016) кардинально змінив ландшафт, зосередившись на коротких відео та використовуючи потужні алгоритми машинного навчання для створення персоналізованої стрічки "For You", що дозволило навіть новим креаторам досягати мільйонних аудиторій без попереднього фолловінгу.

Сучасні соціальні мережі інтегрують штучний інтелект для персоналізації контенту, аналізуючи терабайти даних для передбачення преференцій користувачів. Підтримка функцій реального часу, включаючи live streaming та відеочати, створює відчуття присутності та миттєвої взаємодії. Мультимедійність стала стандартом, з підтримкою AR/VR технологій, 360-градусних відео та інтерактивних Stories, що вимагає від серверної частини високої продуктивності, масштабованості та гнучкості для обробки мільйонів запитів щосекунди [3].

1.2 Архітектура та технології

Серверна частина соціальних мереж представляє собою складну технологічну екосистему, що повинна забезпечувати високу доступність, масштабованість та продуктивність для обслуговування мільйонів користувачів одночасно. Архітектурні рішення значною мірою визначають успіх платформи, впливаючи на швидкість відгуку, стабільність роботи та можливості розширення функціоналу.

Монолітна архітектура об'єднує всі компоненти системи — автентифікацію, управління контентом, повідомлення, аналітику — в єдину кодову базу та виконуваний модуль. Основні переваги включають простоту розробки, оскільки всі компоненти знаходяться в одному проєкті, спрощене тестування без необхідності налаштування складної інфраструктури, та легке розгортання через єдиний

артефакт. Монолітний підхід також забезпечує повну підтримку ACID транзакцій, оскільки всі операції відбуваються в рамках одної бази даних. Проте моноліти погано масштабуються при зростанні навантаження, оскільки всі компоненти повинні масштабуватись разом, та ускладнюють командну роботу при збільшенні команди розробників через можливі конфлікти в спільній кодовій базі.

Мікросервісна архітектура, яка використовується в Instagram, Twitter, Netflix та інших великих платформах, розподіляє функціонал між незалежними сервісами, кожен з яких відповідає за специфічну бізнес-функцію. Наприклад, окремі сервіси можуть обробляти автентифікацію користувачів, управління постами, систему коментарів, нотифікації та аналітику. Це забезпечує горизонтальну масштабованість, коли кожен сервіс може масштабуватись незалежно відповідно до навантаження, легке оновлення окремих модулів без впливу на всю систему, та стійкість до збоїв через ізоляцію сервісів. Однак мікросервіси ускладнюють координацію між компонентами через необхідність мережевої комунікації, вимагають складних механізмів моніторингу та логування, та створюють проблеми з консистентністю даних через розподілені транзакції [4].

Service-Oriented Architecture (SOA) фокусується на інтеграції різних систем через стандартизовані інтерфейси та протоколи обміну даними. Цей підхід особливо корисний для інтеграції сторонніх систем, таких як платіжні шлюзи, сервіси електронної пошти, аналітичні інструменти або системи модерації контенту. SOA дозволяє поступово модернізувати legacy системи та забезпечує гнучкість у виборі технологій для різних компонентів.

Для забезпечення взаємодії в реальному часі використовуються різні технології та протоколи. WebSocket технологія забезпечує двонаправлений зв'язок між клієнтом та сервером, що критично важливо для чатів, нотифікацій та live updates. Message queues, такі як Apache Kafka або RabbitMQ, використовуються для асинхронної обробки повідомлень, що дозволяє системі обробляти пікові навантаження та забезпечує надійність доставки повідомлень навіть при тимчасових збоях окремих компонентів.

REST API залишається золотим стандартом для клієнт-серверної взаємодії

завдяки простоті, зрозумілості та універсальності. RESTful принципи забезпечують предсказувану структуру ендпоінтів та статус-кодів, що спрощує інтеграцію для клієнтських додатків. GraphQL набуває популярності як альтернатива REST через гнучкість запитів, що дозволяє клієнтам запитувати лише необхідні дані, зменшуючи кількість мережевого трафіку та кількість запитів. Facebook використовує GraphQL для оптимізації запитів до стрічки новин, де один GraphQL запит може замінити десятки REST викликів [5].

Serverless архітектура з використанням AWS Lambda, Azure Functions або Google Cloud Functions підходить для обробки пікових навантажень та епізодичних задач, але рідко застосовується як основна архітектура соціальних мереж через складність управління станом та обмеження часу виконання функцій. Для наочного порівняння різних архітектурних підходів наведено таблицю 1.1.

Таблиця 1.1 — Порівняння архітектурних підходів

Критерій	Монолітна	Мікросервісна	SOA
Складність розробки	Низька	Висока	Середня
Швидкість розробки	Висока	Низька	Середня
Масштабованість	Обмежена	Відмінна	Хороша
Розгортання	Просте	Складне	Середнє
Тестування	Просте	Складне	Середнє
Підходить для	Малі/середні проєкти	Великі системи	Корпоративні системи

Для розробки серверної частини Java залишається одним з найпопулярніших вибіром завдяки надійності, кросплатформності, великій екосистемі бібліотек та інструментів. Spring Framework є де-факто стандартом для Java розробки корпоративних додатків. Spring Boot радикально спрощує створення REST API, надаючи вбудовані веб-сервери (Tomcat, Jetty), автоматичну конфігурацію та стартери для швидкого налаштування різних компонентів. Spring Security

забезпечує комплексну безпеку, підтримуючи різні механізми автентифікації та авторизації, включаючи JWT, OAuth2, SAML та рольовий доступ. Spring Data значно спрощує роботу з різними типами баз даних через Spring Data JPA для реляційних баз, Spring Data MongoDB для документних баз, та Spring Data Redis для кешування.

Екосистема інструментів навколо Java розробки включає системи збірки Maven та Gradle, які автоматизують управління залежностями, компіляцію, тестування та пакування додатків. Hibernate як ORM забезпечує об'єктно-реляційне відображення та оптимізацію запитів до бази даних. Контейнеризація через Docker та оркестрація через Kubernetes полегшують розгортання, масштабування та управління життєвим циклом додатків. Для асинхронної обробки застосовуються RabbitMQ для традиційного message passing або Apache Kafka для high-throughput event streaming. CI/CD пайплайни через GitHub Actions, Jenkins або GitLab CI автоматизують тестування, збірку та розгортання, забезпечуючи швидкі та надійні релізи [6].

Інструменти розробки та моніторингу включають Swagger/OpenAPI для автоматичної генерації документації API, Postman та JUnit для тестування функціоналу, Prometheus та Grafana для збору метрик та візуалізації продуктивності, ELK Stack (Elasticsearch, Logstash, Kibana) для централізованого логування та аналізу. Ці інструменти створюють повну екосистему для розробки, тестування, розгортання та підтримки соціальних мереж корпоративного рівня.

1.3 Зберігання даних та безпека

Вибір системи зберігання даних є одним з найкритичніших архітектурних рішень для соціальних мереж, оскільки він впливає на продуктивність, масштабованість, консистентність даних та загальну користувацьку експериенцію. Різні типи даних вимагають різних підходів до зберігання та оптимізації.

Проєкт використовує PostgreSQL як основну реляційну базу даних для зберігання структурованих даних, включаючи профілі користувачів, метадані постів, коментарі, інформацію про лайки та підписки. PostgreSQL вибрано завдяки

надійній підтримці ACID транзакцій, що критично важливо для забезпечення консистентності даних при одночасному доступі тисяч користувачів. Реляційна модель ідеально підходить для моделювання складних зв'язків між сутностями через зовнішні ключі, що забезпечує референційну цілісність та можливість виконання складних JOIN запитів для отримання агрегованих даних. PostgreSQL також підтримує індекси різних типів (B-tree, Hash, GiST, GIN), що дозволяє оптимізувати продуктивність запитів для різних сценаріїв використання.

У великих соціальних мережах застосовуються більш складні стратегії зберігання даних, що включають комбінацію реляційних та NoSQL баз даних. Instagram використовує Apache Cassandra для зберігання великих обсягів неструктурованих даних, таких як метадані фотографій та часових серій користувацької активності. Cassandra забезпечує лінійну масштабованість та високу доступність через розподілену архітектуру без єдиної точки відмови. MongoDB застосовується для зберігання документів змінної структури, таких як налаштування користувачів, конфігурації додатків або аналітичні дані, де схема може еволюціонувати без міграції всієї бази даних.

Redis відіграє критичну роль у архітектурі високонавантажених соціальних мереж як високопродуктивне in-memory сховище для кешування. Типові сценарії використання Redis включають кешування стрічки новин користувачів, де часто запитувані пости зберігаються в пам'яті для миттєвого доступу, зберігання сесій користувачів для швидкої автентифікації, кешування популярного контенту та трендів, а також реалізацію rate limiting для захисту від зловживань API. У поточному проєкті PostgreSQL є достатньою для обробки планованого навантаження, але в майбутньому Redis може бути інтегрований для оптимізації продуктивності критичних операцій [7].

Elasticsearch може бути використаний для реалізації потужного повнотекстового пошуку по контенту користувачів. Це особливо важливо для соціальних мереж, де користувачі очікують можливість швидко знайти пости, користувачів або хештеги за ключовими словами. Elasticsearch забезпечує швидкий пошук, автокомпліт, fuzzy matching та складні фільтри.

Гібридний підхід до зберігання даних є поширеною практикою серед великих технологічних компаній. Twitter комбінує MySQL для зберігання основних даних користувачів та зв'язків, внутрішню NoSQL систему Manhattan для зберігання твітів та Redis для кешування трендів та популярного контенту. Facebook використовує MySQL для соціального графу, Cassandra для повідомлень та Redis для кешування. Цей підхід дозволяє оптимізувати кожен тип даних відповідно до його специфічних вимог щодо консистентності, доступності та продуктивності.

Вибір системи зберігання залежить від кількох ключових факторів: обсягу даних, частоти та типу запитів, вимог до консистентності, латентності доступу та бюджетних обмежень. Для проєктів середнього масштабу PostgreSQL часто є оптимальним вибором завдяки балансу між функціональністю та складністю управління.

Безпека серверної частини є критично важливою для соціальних мереж, оскільки вони зберігають величезні обсяги персональних даних користувачів та є привабливими цілями для кіберзлочинців. Основні категорії загроз включають SQL-ін'єкції, які можуть дозволити зловмисникам отримати доступ до бази даних через вразливі запити, Cross-Site Scripting (XSS) атаки для виконання зловмисних скриптів в браузерах користувачів, Cross-Site Request Forgery (CSRF) для виконання небажаних дій від імені автентифікованих користувачів, витоки персональних даних через неправильно налаштовані права доступу, та DDoS атаки для порушення доступності сервісу [8].

Проєкт Five-o-gram реалізує комплексну систему захисту на кількох рівнях. Spring Security з JWT токенами забезпечує stateless автентифікацію та авторизацію, що критично важливо для масштабованості, оскільки сервер не зберігає стан сесій. BCrypt використовується для безпечного хешування паролів з автоматичним додаванням солі, що робить паролі практично неможливими для зворотного розшифрування навіть при компрометації бази даних. Hibernate Validator забезпечує валідацію всіх вхідних даних на рівні моделей, запобігаючи обробці некоректних або потенційно шкідливих даних.

OWASP Java Encoder застосовується для захисту від XSS атак у текстових

полях постів та коментарів, кодуючи всі потенційно небезпечні символи перед відображенням. HTTPS шифрування забезпечує конфіденційність передачі даних між клієнтом та сервером, а CORS (Cross-Origin Resource Sharing) обмежує доступ до API лише з дозволених доменів.

JWT токени, хоча й забезпечують ефективну stateless автентифікацію, потребують особливої уваги до безпеки. Токени повинні мати обмежений час життя, передаватися лише через захищені канали, та правильно валідуватися на сервері. Порівняння різних методів автентифікації показує, що JWT є більш ефективним за OAuth2 для монолітних додатків через простоту реалізації та менші накладні витрати, але OAuth2 може бути кращим вибором для мікросервісної архітектури з множинними клієнтськими додатками.

Для посилення безпеки рекомендується впровадження додаткових заходів: rate limiting для захисту від brute force атак та зловживання API, OWASP Dependency-Check для регулярного сканування залежностей проєкту на наявність відомих вразливостей, Web Application Firewall (WAF) для додаткового захисту від поширених атак, та регулярні security аудити коду та інфраструктури.

1.4 Задачі та цілі проєкту

Основна задача проєкту полягає в створенні повнофункціональної серверної частини соціальної мережі, що забезпечує весь спектр базового функціоналу сучасних соціальних платформ. Розроблювана система повинна підтримувати безпечну реєстрацію користувачів з обов'язковою email-верифікацією для забезпечення якості користувацької бази, надійну автентифікацію та авторизацію для захисту особистих даних, повнофункціональну систему створення та перегляду мультимедійного контенту, інтерактивні можливості коментування та вподобань для стимулювання користувацької взаємодії, а також механізм соціальних зв'язків через систему підписок.

Ключовою вимогою є забезпечення високого рівня безпеки на всіх рівнях системи, включаючи захист від поширених веб-атак, безпечне зберігання конфіденційних даних та надійні механізми контролю доступу. Система повинна

демонструвати відмінну продуктивність під навантаженням, забезпечуючи швидкий відгук навіть при обробці великої кількості одночасних запитів. Архітектура має бути спроектована з урахуванням майбутнього розширення функціоналу, передбачаючи можливість додавання складніших функцій, таких як система чатів в реальному часі, live streaming відео, розширена аналітика користувацької поведінки, або інтеграція з зовнішніми сервісами та API.

Перша стратегічна ціль передбачає розробку монолітної серверної частини з використанням Java та Spring Boot фреймворку як технологічної основи. Вибір Java обґрунтований її зрілістю як платформи для розробки корпоративних додатків, надійністю JVM для обробки високих навантажень, величезною екосистемою бібліотек та інструментів, а також широкою підтримкою спільноти розробників. Java забезпечує стабільну продуктивність при обробці великої кількості одночасних користувачів завдяки ефективним алгоритмам збирання сміття, оптимізаціям JIT-компілятора та можливостям паралельної обробки через багатопоточність [9].

Spring Boot обрано як основний фреймворк через його здатність значно спрощувати конфігурацію та розгортання додатків, надаючи автоматичну конфігурацію для більшості типових компонентів, вбудовані веб-сервери для швидкого запуску, та комплексну інтеграцію з екосистемою Spring. Це дозволяє команді розробників зосередитися на реалізації бізнес-логіки замість налаштування інфраструктурних компонентів, значно прискорюючи процес розробки та зменшуючи кількість потенційних помилок конфігурації.

Друга ціль включає створення комплексного REST API для всіх CRUD-операцій з основними сутностями системи, включаючи управління профілями користувачів, створення та модифікацію постів, систему коментарів та вподобань, а також механізм підписок. API повинен повністю відповідати принципам RESTful дизайну, забезпечуючи інтуїтивно зрозумілу структуру ендпоінтів, правильне використання HTTP методів та статус кодів, а також консистентне форматування JSON відповідей.

Критично важливою є реалізація підтримки пагінації для ефективної обробки

великих наборів даних, що дозволить клієнтським додаткам завантажувати контент частинами та забезпечить плавну користувацьку експериенцію навіть при роботі з тисячами записів. Система фільтрації повинна надавати гнучкі можливості для пошуку та відбору контенту за різними критеріями, включаючи часові періоди, типи контенту, авторів та інші атрибути.

Комплексна валідація всіх вхідних параметрів забезпечить цілісність даних та захист від некоректних або зловмисних запитів. Система повинна повертати зрозумілі та інформативні повідомлення про помилки з відповідними HTTP статус кодами, що допоможе розробникам клієнтських додатків швидко діагностувати та виправляти проблеми.

Особлива увага приділяється оптимізації запитів до бази даних для уникнення проблеми N+1, яка може значно погіршити продуктивність при роботі з пов'язаними сутностями. Це включає використання eager loading стратегій, batch fetching, та оптимізованих JOIN запитів для мінімізації кількості звернень до бази даних.

Третя ціль зосереджена на забезпеченні надійної безпеки через імплементацію JWT автентифікації та Spring Security фреймворку. JWT токени забезпечують stateless автентифікацію, що є критично важливим для горизонтального масштабування додатку, оскільки сервери не потребують збереження стану сесій користувачів. Це дозволяє легко додавати нові екземпляри серверів для обробки зростаючого навантаження без необхідності синхронізації сесійних даних [10].

Spring Security надає комплексний та перевірений часом фреймворк для управління автентифікацією та авторизацією, включаючи підтримку різних механізмів автентифікації, гнучку конфігурацію правил доступу, та інтеграцію з корпоративними системами безпеки. Система повинна підтримувати механізми refresh токенів для продовження сесій користувачів без повторного введення облікових даних, безпечний logout з інвалідацією токенів на стороні сервера, та можливість відкликання токенів у випадку компрометації облікового запису або підозрілої активності.

Четверта ціль передбачає створення високоефективної структури бази даних у PostgreSQL з ретельним моделюванням всіх зв'язків між сутностями та оптимізацією для специфічних потреб соціальної мережі. База даних повинна забезпечувати референційну цілісність через правильне використання зовнішніх ключей та обмежень, що гарантує консистентність даних навіть при одночасній роботі багатьох користувачів.

Стратегічне створення індексів для прискорення критичних запитів включає композитні індекси для складних умов пошуку, часткові індекси для оптимізації специфічних сценаріїв, та текстові індекси для повнотекстового пошуку. Використання відповідних типів даних PostgreSQL забезпечить оптимальне використання дискового простору та швидкість операцій.

Особлива увага приділяється оптимізації запитів для стрічки новин користувачів, яка є однією з найбільш навантажених функцій соціальних мереж. Це включає денормалізацію критичних даних для зменшення кількості JOIN операцій, кешування популярного контенту, та використання матеріалізованих представлень для складних агрегацій. Система пошуку повинна забезпечувати швидкий доступ до контенту за різними критеріями, включаючи повнотекстовий пошук по постах та пошук користувачів за іменами та username.

П'ята ціль включає створення вичерпної документації API через Swagger/OpenAPI та проведення багаторівневого тестування для забезпечення якості та надійності системи. Swagger повинен автоматично генерувати повну інтерактивну документацію з детальними описами всіх ендпоінтів, параметрів запитів, форматів відповідей, та прикладами використання. Це значно спростить процес інтеграції для розробників клієнтських додатків, frontend команд, та сторонніх розробників [11].

Комплексна стратегія тестування включає unit тести з використанням JUnit для перевірки бізнес-логіки окремих компонентів в ізоляції, mock-тестування залежностей через Mockito для імітації зовнішніх сервісів та репозиторіїв, інтеграційні тести для верифікації взаємодії між різними шарами додатку, та API тести через Postman або аналогічні інструменти для валідації функціональності з

точки зору клієнта.

Автоматизоване тестування повинно включатися в CI/CD пайплайн для забезпечення якості кожного релізу, з автоматичним запуском при кожному комміті та обов'язковим проходженням всіх тестів перед деплоєм на продакшн.

Шоста ціль передбачає впровадження Docker контейнеризації для забезпечення консистентності середовища розробки, тестування та продакшну. Docker гарантує, що додаток буде працювати ідентично незалежно від операційної системи хоста або встановлених бібліотек, вирішуючи класичну проблему "працює на моїй машині".

Docker Compose повинен дозволяти розробникам запускати всю необхідну інфраструктуру, включаючи базу даних, сервіси кешування, та сам додаток, однією командою. Це значно спрощує onboarding нових членів команди, налаштування CI/CD пайплайнів, та створення ідентичних середовищ для різних цілей.

Контейнеризація також забезпечує легке масштабування в продакшн середовищі через оркестратори контейнерів типу Kubernetes, дозволяючи автоматично масштабувати кількість екземплярів додатку відповідно до навантаження.

Сьома та найстратегічніша ціль зосереджена на закладанні архітектурного фундаменту для майбутнього масштабування без необхідності кардинальних змін в існуючій системі. Підготовка до інтеграції Redis як високопродуктивного in-memory сховища для кешування включає ідентифікацію критичних точок для кешування, таких як стрічка новин користувачів, популярний контент, сесії користувачів, та результати складних запитів [12].

Планування поступової міграції до мікросервісної архітектури передбачає виділення модульних компонентів з чіткими границями відповідальності, що дозволить в майбутньому витягувати окремі сервіси без порушення загальної функціональності. Розподілення навантаження читання та запису через read replicas бази даних забезпечить кращу продуктивність при зростанні користувацької бази.

Впровадження асинхронної обробки через message queues (Apache Kafka, RabbitMQ) для операцій, що не потребують миттєвого відгуку, таких як відправка

email нотифікацій, обробка зображень, генерація звітів, або аналітика користувацької поведінки, дозволить системі ефективно обробляти пікові навантаження.

Загальна мета проєкту — створення надійного, масштабованого та безпечного серверного рішення, що відповідає найкращим практикам розробки сучасних соціальних мереж та здатне обслуговувати тисячі одночасних користувачів з можливістю органічного зростання без кардинальних архітектурних змін. Система повинна забезпечувати швидкий відгук на запити користувачів, високу надійність роботи навіть при несподіваних навантаженнях, та максимальну зручність для розробників клієнтських додатків через добре документований та інтуїтивний API [13].

2 ПРОЄКТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ СОЦІАЛЬНОЇ МЕРЕЖІ

2.1 Архітектура серверної частини

Серверна частина проєкту побудована на основі монолітної архітектури з використанням фреймворку Spring Boot. Монолітна архітектура передбачає об'єднання всіх функціональних компонентів системи в єдиний виконуваний модуль, що включає автентифікацію користувачів, управління постами, систему коментарів, механізм вподобань та функціонал підписок. На рисунку 2.1 зображено монолітну архітектуру з згрупованими шарами та модулями.

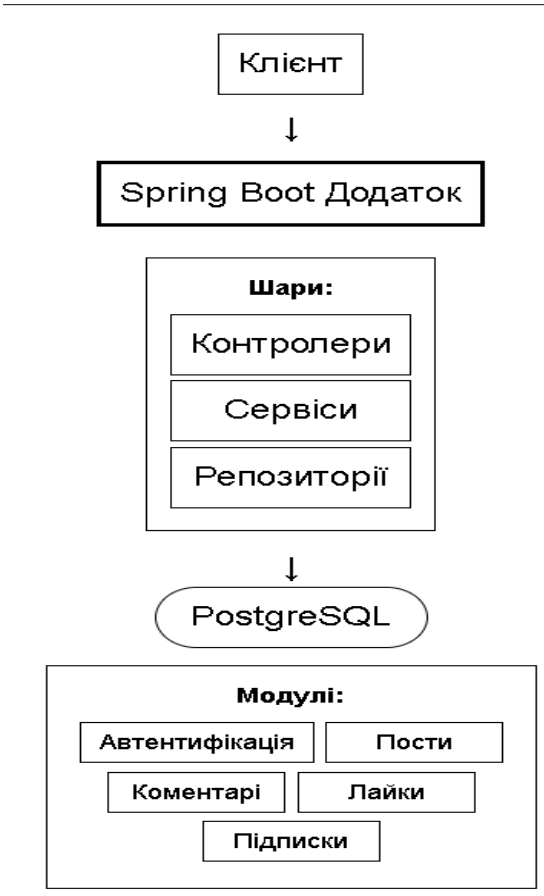

Рисунок 2.1 — Загальна структура архітектури

Основною перевагою вибраної архітектури є спрощення процесу розробки завдяки відсутності необхідності налаштування міжсервісної комунікації. Всі компоненти працюють в рамках одного процесу, що забезпечує високу швидкість виконання операцій та повну підтримку ACID-транзакцій. Тестування також суттєво спрощується, оскільки всі модулі можна тестувати в єдиному контексті без

необхідності емуляції мережевих викликів.

Розгортання монолітного додатку здійснюється шляхом створення єдиного JAR-файлу, що містить всі залежності та може бути запущений на будь-якому сервері з підтримкою Java. Це рішення особливо підходить для проєктів середнього масштабу, де складність мікросервісної архітектури була б невиправданою.

Архітектура додатку організована за принципом багатошарової структури. Шар контролерів відповідає за обробку HTTP-запитів та формування відповідей у форматі JSON. Бізнес-логіка зосереджена в сервісному шарі, який реалізує всі правила предметної області та управляє транзакціями. Шар доступу до даних представлений репозиторіями Spring Data JPA, які забезпечують абстракцію над операціями з базою даних [14].

2.2 Структура PostgreSql бд та зв’язки між сутностями

База даних проєкту побудована на PostgreSQL — реляційній системі управління базами даних, що забезпечує надійність, продуктивність та підтримку складних запитів. Структура бази даних складається з дванадцяти основних таблиць, кожна з яких відповідає за зберігання специфічного типу інформації.

Центральною таблицею системи є users, яка містить основну інформацію про користувачів: унікальний ідентифікатор, ім'я, прізвище, унікальне ім'я користувача, email-адресу та хешований пароль. Паролі зберігаються у вигляді BCrypt-хешів з високим рівнем складності, що забезпечує захист від атак перебору. Додатково таблиця містить поле ролі користувача, яке визначає рівень доступу в системі.

Таблиця posts зберігає основну інформацію про публікації користувачів, включаючи текстовий контент, час створення та зовнішній ключ на автора публікації. Кожен пост може мати декілька прикріплених зображень, інформація про які зберігається в таблиці pictures. Ця таблиця містить шлях до файлу зображення та зовнішній ключ на відповідний пост.

Система інтерактивності реалізована через таблиці likes та comments. Таблиця likes відстежує вподобання постів користувачами, використовуючи

композитний унікальний ключ для запобігання дублікатів. Таблиця comments зберігає коментарі до постів з можливістю створення вкладених коментарів через самопосилання.

Соціальні зв'язки між користувачами реалізовані через таблицю subscriptions, яка моделює відношення "багато до багатьох" між користувачами. Кожен запис містить ідентифікатор підписника та ідентифікатор користувача, на якого здійснюється підписка.

Функціонал Stories реалізований через таблицю stories, яка зберігає тимчасовий контент користувачів з полем is_expired для відстеження актуальності контенту. Спонсоровані публікації виділені в окрему таблицю sponsored_posts, що дозволяє відстежувати комерційний контент.

Система сповіщень базується на таблиці notifications, яка зберігає інформацію про події в системі (нові підписки, коментарі, вподобання) з типізацією через поле type. Процес реєстрації організований через проміжну таблицю user_to_register, яка тимчасово зберігає дані нових користувачів до підтвердження email-адреси.

Всі зовнішні ключі налаштовані з опцією CASCADE DELETE, що забезпечує автоматичне видалення залежних записів при видаленні основного об'єкта. Для оптимізації продуктивності створено індекси на найбільш часто використовувані поля для пошуку та фільтрації [15].

2.3 Реалізація REST API для реєстрації, автентифікації та роботи з постами

REST API проєкту побудовано згідно з принципами RESTful архітектури, забезпечуючи чіткий інтерфейс для взаємодії клієнтських додатків з сервером. API організовано за ресурсно-орієнтованим підходом, де кожен ендпоінт відповідає за операції з конкретним типом ресурсів.

Процес реєстрації користувачів реалізований через двоетапний механізм з підтвердженням email-адреси. На першому етапі клієнт надсилає POST-запит на /api/auth/register з базовою інформацією про користувача. Сервер валідує отримані дані, перевіряє унікальність email та username, генерує код підтвердження та

зберігає тимчасові дані в таблиці user_to_register. Код підтвердження надсилається на вказану email-адресу.

Другий етап реєстрації включає підтвердження email через ендпоінт /api/auth/verify, де користувач надає email та отриманий код. При успішній верифікації дані переносяться з тимчасової таблиці до основної таблиці users, а тимчасовий запис видаляється.

Автентифікація користувачів здійснюється через ендпоінт /api/auth/login, який приймає email та пароль. Сервер перевіряє коректність облікових даних, порівнюючи надісланий пароль з BCrypt-хешем, збереженим в базі даних. При успішній автентифікації генерується JWT-токен, який містить ідентифікатор користувача, роль та час закінчення дії токена [16].

Управління постами реалізовано через набір CRUD-операцій. Створення нового поста здійснюється через POST-запит на /api/posts з текстовим контентом та опціональними зображеннями. Сервер обробляє мультипарт-дані, зберігає зображення в файловій системі, витягує хештеги з тексту та створює відповідні записи в базі даних.

Отримання інформації про пост доступне через GET-запит на /api/posts/{id}, який повертає повну інформацію про публікацію, включаючи дані автора, прикріплені зображення, список коментарів та кількість вподобань. Оновлення та видалення постів дозволені лише авторам публікацій або адміністраторам системи.

Система коментарів реалізована через ендпоінт /api/posts/{id}/comments, який дозволяє додавати коментарі до конкретних постів. Підтримується створення вкладених коментарів через вказання ідентифікатора батьківського коментаря.

Механізм вподобань реалізований через ендпоінти /api/posts/{id}/likes та /api/comments/{id}/likes для постів та коментарів відповідно. Система автоматично відстежує унікальність вподобань та оновлює лічильники.

Всі відповіді API форматуються у JSON з відповідними HTTP-кодами статусу. Валідація вхідних даних здійснюється через Hibernate Validator з використанням анотацій типу @NotBlank, @Email, @Size [17].

2.4 Механізм авторизації та захисту API за допомогою JWT-токенів

Система авторизації побудована на основі JWT (JSON Web Tokens) з використанням Spring Security Framework. JWT забезпечує stateless автентифікацію, що дозволяє масштабувати додаток без необхідності зберігання сесій на сервері.

Після успішної автентифікації сервер генерує JWT-токен, який містить зашифровану інформацію про користувача. Токен підписується секретним ключем, що забезпечує його цілісність та автентичність. Клієнт зберігає отриманий токен та додає його до заголовка Authorization кожного наступного запиту у форматі "Bearer {token}".

Для обробки JWT-токенів реалізовано спеціальний фільтр JwtAuthenticationFilter, який перехоплює всі вхідні запити та перевіряє наявність валідного токена. Фільтр витягує токен з заголовка, валідує його підпис та термін дії, отримує інформацію про користувача та встановлює контекст безпеки для поточного запиту.

Система ролей включає два основні рівні доступу: USER та ADMIN. Звичайні користувачі (USER) мають право створювати пости, коментарі, ставити вподобання та керувати своїми підписками. Адміністратори (ADMIN) додатково можуть керувати скаргами користувачів, видаляти будь-який контент та модерувати систему [18].

Авторизація на рівні методів реалізована через анотації Spring Security. Захищені ендпоінти перевіряють не лише наявність аутентифікації, але й відповідність ролі користувача вимогам операції. Для операцій з власним контентом (редагування/видалення постів) додатково перевіряється, чи є поточний користувач автором контенту.

Паролі користувачів зберігаються у вигляді BCrypt-хешів з високою складністю обчислення, що робить їх практично неможливими для зворотного розшифрування. BCrypt автоматично додає "сіль" до кожного паролю, запобігаючи атакам через rainbow tables.

Для захисту від Cross-Site Request Forgery (CSRF) атак використовуються CSRF-токени для всіх POST-запитів. Cross-Origin Resource Sharing (CORS) налаштований для обмеження доступу до API лише з дозволених доменів.

Захист від XSS-атак реалізований через OWASP Java Encoder, який кодує всі користувацькі дані перед їх відображенням. Особлива увага приділяється текстовим полям постів та коментарів, які можуть містити потенційно небезпечний контент [19, 20].

2.5 Тестування серверної частини

Стратегія тестування включає декілька рівнів для забезпечення якості та надійності додатку. Юніт-тестування реалізовано з використанням JUnit 5 та Mockito для тестування окремих компонентів в ізоляції.

Тестування сервісного шару зосереджено на перевірці бізнес-логіки додатку. Використовуються mock-об'єкти для імітації залежностей, що дозволяє тестувати логіку без звернення до реальної бази даних. Тести покривають основні сценарії використання, включаючи створення постів, обробку коментарів, систему вподобань та управління підписками.

Для зручності ручного тестування та демонстрації API створено колекцію тестів в Postman, яка включає всі основні ендпоінти з прикладами запитів та очікуваних відповідей. Колекція організована за функціональними групами та включає тести для автентифікації, роботи з постами, коментарями та користувачами [21].

Документація API автоматично генерується через Swagger (OpenAPI 3.0), що створює інтерактивний веб-інтерфейс для дослідження та тестування ендпоінтів. Swagger UI доступний за адресою /swagger-ui.html та включає детальні описи всіх ендпоінтів, параметрів запитів та форматів відповідей.

2.6 Розгортання серверу

Контейнеризація базується на принципах ізоляції процесів на рівні операційної системи та забезпечує консистентність середовища між розробкою,

тестуванням та продакшном. Docker інкапсулює додаток з усіма його залежностями в портативну одиницю розгортання, розв'язуючи класичну проблему "works on my machine". Ця технологія використовує спільне ядро ОС, що робить контейнери значно легшими за віртуальні машини та забезпечує швидший запуск.

Багатоетапний Dockerfile реалізує концепцію multi-stage builds для оптимізації розміру образів. Перший етап включає всі build tools - JDK, Maven, тестові залежності, а другий етап використовує мінімальний runtime образ з JRE та копіює лише скомпільований JAR файл. Такий підхід може зменшити розмір фінального образу з кількох гігабайт до сотень мегабайт, що значно прискорює розгортання та зменшує споживання мережевого трафіку [22].

Docker Compose втілює принципи infrastructure as code на локальному рівні, дозволяючи описувати весь стек додатку в єдиному YAML файлі. Compose автоматично створює ізольовані мережі для міжсервісної комунікації, управляє порядком запуску згідно з залежностями та забезпечує service discovery через внутрішні DNS імена. Винесення всієї конфігурації в змінні середовища відповідає twelve-factor app methodology, що забезпечує чітке розділення коду та конфігурації і дозволяє використовувати один образ у різних середовищах. Використання секретів через Docker secrets або зовнішні системи управління ключами додатково підвищує безпеку конфіденційної інформації.

Постійні томи вирішують фундаментальну проблему stateful сервісів у контейнерному середовищі. Оскільки контейнери за природою є ephemeral та immutable, всі дані всередині контейнера втрачаються при перезапуску. PostgreSQL критично потребує persistent storage для збереження даних, що реалізується через монтування host volumes або named volumes до директорії /var/lib/postgresql/data всередині контейнера [23].

Liquibase впроваджує концепцію database version control, трактуючи схему бази даних як код, що може версіонуватися, рецензуватися та розгортатися автоматично. Система changesets забезпечує атомарність змін та idempotency операцій. Liquibase відстежує застосовані зміни через службову таблицю DATABASECHANGELOG, що запобігає повторному виконанню міграцій та

дозволяє безпечно розгортати зміни в різних середовищах незалежно від поточного стану схеми. Підтримка rollback функціональності дозволяє безпечно повертати зміни у випадку виявлення критичних помилок після розгортання.

AWS RDS надає повністю managed database service, що абстрагує складність управління інфраструктурою бази даних від розробників. Мульти-AZ розгортання забезпечує high availability через синхронну реплікацію між різними зонами доступності. При збої primary instance система автоматично виконує failover на standby replica протягом 60-120 секунд. Автоматичне резервне копіювання створює щоденні point-in-time snapshots та зберігає transaction logs, що дозволяє відновити базу до будь-якого моменту часу в межах retention period.

Моніторинг через Spring Boot Actuator базується на принципах observability, що включає метрики, логи та трейси. Actuator експортує широкий спектр метрик - JVM memory usage, garbage collection, HTTP request metrics, database connection pool statistics - у стандартизованому форматі, сумісному з Prometheus time-series database. Prometheus збирає метрики кожні 15 секунд та зберігає їх з можливістю довгострокового retention. Grafana візуалізує ці метрики через інтуїтивні дашборди та забезпечує систему алертинга для проактивного виявлення проблем продуктивності. Інтеграція з системами нотифікацій дозволяє автоматично сповіщати DevOps команду про критичні інциденти через Slack, email або SMS.

Міграція на AWS S3 представляє перехід від централізованого файлового сховища до розподіленої object storage архітектури. S3 забезпечує практично необмежену масштабованість та eleven nines durability через автоматичну реплікацію даних між кількома data centers. CloudFront CDN інтегрується з S3 для мінімізації latency через кешування контенту в понад 400 edge locations по всьому світу, реалізуючи принципи edge computing та значно покращуючи user experience для географічно розподілених користувачів.

Безпека продакшн-розгортання базується на концепції defense in depth - багаторівневому захисті з різними security controls на кожному рівні архітектури. SSL/TLS шифрування забезпечує конфіденційність та цілісність даних під час передачі. Web Application Firewall діє на application layer, фільтруючи malicious

traffic та захищаючи від OWASP Top 10 вразливостей. Регулярне оновлення всіх компонентів системи реалізує принцип security by design через своєчасне застосування критичних security patches. Впровадження принципу least privilege через IAM ролі та policies мінімізує потенційний impact від компрометації облікових записів. На рисунку 2.2 зображено схему розгортання серверної частини.

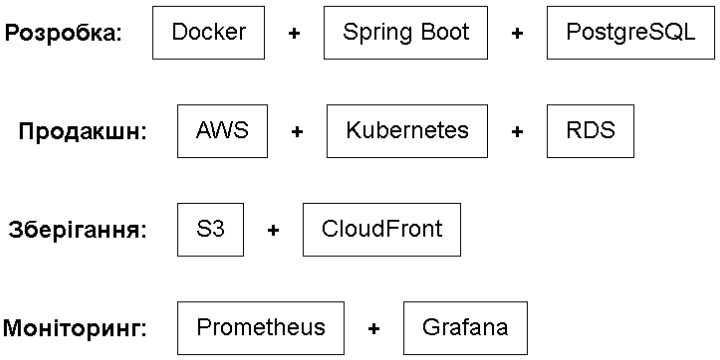

Рисунок 2.2 — Схема розгортання

Архітектура серверної частини соціальної мережі представляє собою комплексне рішення, побудоване на сучасних технологіях. Монолітна архітектура на базі Spring Boot забезпечує стабільність та простоту розробки, PostgreSQL гарантує надійне зберігання даних з оптимізованою структурою взаємозв'язків, REST API надає зручний інтерфейс для клієнтських додатків. Система безпеки з JWT авторизацією та багаторівневим захистом забезпечує конфіденційність користувацьких даних, комплексне тестування гарантує якість коду, а контейнеризоване розгортання через Docker забезпечує масштабованість та портабельність рішення. Вся архітектура спроектована з урахуванням майбутнього росту платформи і можливості інтеграції з сервісами [24].

3 РЕАЛІЗАЦІЯ СЕРВЕРНОЇ ЧАСТИНИ

3.1 Налаштування середовища та структури проєкту з використанням Spring Boot

Налаштування середовища розробки є критично важливим етапом, що визначає продуктивність та ефективність всього процесу створення серверної частини соціальної мережі. Правильний вибір інструментів розробки та організація проєктної структури забезпечують не лише швидку розробку, але й подальшу підтримку та масштабування системи.

Як основне середовище розробки обрано IntelliJ IDEA Ultimate — професійну IDE, що надає комплексну підтримку Java екосистеми та інтеграцію з усіма необхідними інструментами. IntelliJ IDEA забезпечує розширені можливості рефакторингу коду, інтелектуальне автодоповнення, вбудований дебагер з підтримкою віддаленого налагодження, інтеграцію з системами контролю версій Git, а також спеціалізовані плагіни для Spring Boot розробки. IDE включає вбудовані інструменти для роботи з базами даних, що дозволяє безпосередньо з середовища розробки виконувати SQL запити, переглядати структуру таблиць та аналізувати продуктивність запитів до PostgreSQL.

Вибір Java 19 як основної платформи розробки обґрунтований кількома факторами. По-перше, Java забезпечує високу продуктивність та стабільність при обробці великої кількості одночасних користувачів завдяки ефективній віртуальній машині JVM та зрілим алгоритмам збирання сміття. По-друге, Java 19 включає сучасні мовні функції, такі як pattern matching, records, sealed classes, що дозволяють писати більш виразний та безпечний код. По-третє, екосистема Java надає потужні бібліотеки для всіх аспектів розробки соціальних мереж: Jackson для обробки JSON, Apache Commons для утилітарних функцій, JavaMail для інтеграції з SMTP серверами, та WebSocket API для реального часу комунікації.

Maven обрано як систему збірки та управління залежностями через її зрілість, стандартизованість та широку підтримку в Java спільноті. Maven автоматизує весь життєвий цикл проєкту: управління залежностями з автоматичним завантаженням

бібліотек з центрального репозиторію, компіляцію вихідного коду з підтримкою різних версій Java, виконання unit та інтеграційних тестів з генерацією звітів покриття, пакування додатку в JAR або WAR файли для розгортання, а також інтеграцію з CI/CD пайплайнами для автоматичної збірки та розгортання.

Архітектура проєкту організована за принципом багатошарової структури з чітким розподілом відповідальності між компонентами. Пакет controllers містить REST контролери, що відповідають за обробку HTTP запитів від клієнтських додатків, валідацію вхідних параметрів, маршалінг/демаршалінг JSON даних та формування HTTP відповідей з відповідними статус кодами. Контролери реалізують тонкий шар, що делегує всю бізнес-логіку сервісному шару, забезпечуючи принцип єдиної відповідальності.

Пакет services інкапсулює всю бізнес-логіку додатку, включаючи правила предметної області, валідацію бізнес-правил, оркестрацію операцій між різними репозиторіями та управління транзакціями. Сервісний шар також відповідає за генерацію нотифікацій, обробку файлів, інтеграцію з зовнішніми сервісами та реалізацію складних алгоритмів, таких як формування стрічки новин або системи рекомендацій.

Пакет repositories забезпечує абстракцію доступу до даних через Spring Data JPA, що автоматично генерує імплементації репозиторіїв на основі інтерфейсів та методів-конвенцій. Репозиторії включають як стандартні CRUD операції, так і кастомні запити для специфічних потреб, таких як пошук користувачів за частиною імені або отримання популярних постів за певний період.

Пакет models містить JPA сутності, що відображають структуру бази даних в об'єктній моделі Java. Сутності включають анотації для налаштування відображення на таблиці, зв'язків між об'єктами, стратегій завантаження та валідації полів. Кожна сутність відповідає конкретній таблиці бази даних: User для users, Post для posts, Picture для pictures, та інші.

Пакет dto (Data Transfer Objects) забезпечує безпечну передачу даних між різними шарами додатку та зовнішніми клієнтами. DTO об'єкти інкапсулюють лише необхідні для конкретної операції поля, приховуючи внутрішню структуру

доменних об'єктів та забезпечуючи версіонування API.

Пакет config містить конфігураційні класи Spring, що налаштовують різні аспекти додатку: Spring Security для автентифікації та авторизації, Jackson для серіалізації JSON, CORS для cross-origin запитів, Swagger для документації API, та інші інфраструктурні компоненти. Файлову структуру проєкту зображено на рисунку 3.1.

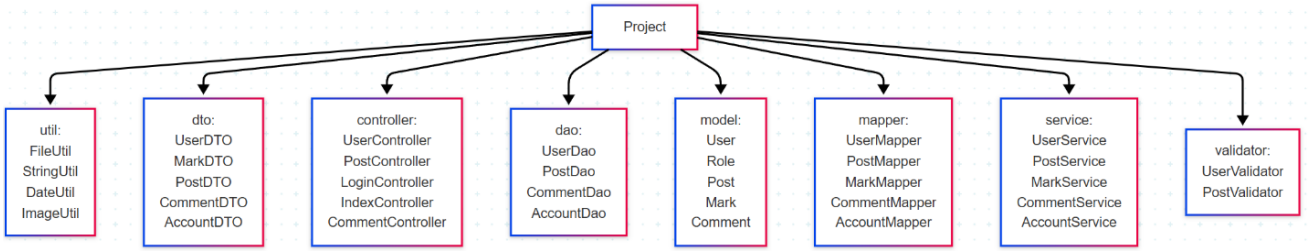

Рисунок 3.1 — Структура проєкту

Docker відіграє ключову роль в забезпеченні консистентності середовища розробки та продакшену. Контейнеризація гарантує, що додаток працюватиме ідентично на локальних машинах розробників, тестових серверах та продакшн інфраструктурі, незалежно від операційної системи хоста або встановлених бібліотек. Docker також спрощує процес налаштування середовища для нових членів команди, оскільки вся інфраструктура може бути запущена однією командою.

Лістинг 3.1 — Файл docker-compose.yml

```
version: '3'
services:
  app:
    build: ./
    ports:
      - 8080:8080
    depends_on:
      - postgres
    environment:
      - MYAPP_JDBC_URL=jdbc:postgresql://postgres:5432/Fiveogram
      - MYAPP_JDBC_USER=postgres
      - MYAPP_JDBC_PASS=postgres
    restart: unless-stopped
    networks:
      - backend
  postgres:
```

```
    image: postgres:13
    ports:
      - 5432:5432
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: Fiveogram
    restart: unless-stopped
    networks:
      - backend
networks:
  backend:
```

Docker Compose оркеструє запуск всієї інфраструктури, включаючи Spring Boot додаток та PostgreSQL базу даних. Конфігурація забезпечує ізольовану мережу backend для безпечної комунікації між контейнерами, автоматичний перезапуск сервісів при збоях, та правильну послідовність запуску з залежностями.

Лістинг 3.2 — Файл Dockerfile

```
FROM openjdk:19
EXPOSE 8080
COPY target/Five-o-gram-0.0.1-SNAPSHOT.jar /app.jar
CMD ["java", "-jar", "/app.jar"]
```

Dockerfile реалізований за принципом мінімалізму, використовуючи офіційний OpenJDK образ як базу та копіюючи лише необхідний JAR файл. Такий підхід забезпечує швидку збірку образу та мінімальний розмір фінального контейнера.

PostgreSQL вибрано як основну систему управління базами даних завдяки її надійності, продуктивності та розширеним можливостям. База даних спроектована з урахуванням принципів нормалізації та включає оптимізовані індекси для критичних запитів. Структура бази включає тринадцять основних таблиць, кожна з яких відповідає за зберігання специфічного типу даних: users для профілів користувачів, posts для публікацій, pictures для зображень, likes для вподобань, comments для коментарів, comment_likes для вподобань коментарів, hashtag для хештегів, marks для міток на зображеннях, stories для тимчасового контенту, avatars для аватарів користувачів, subscriptions для підписок, notifications для сповіщень, та user_to_register для тимчасового зберігання даних реєстрації. Діаграма таблиць зображена на рисунку 3.2.

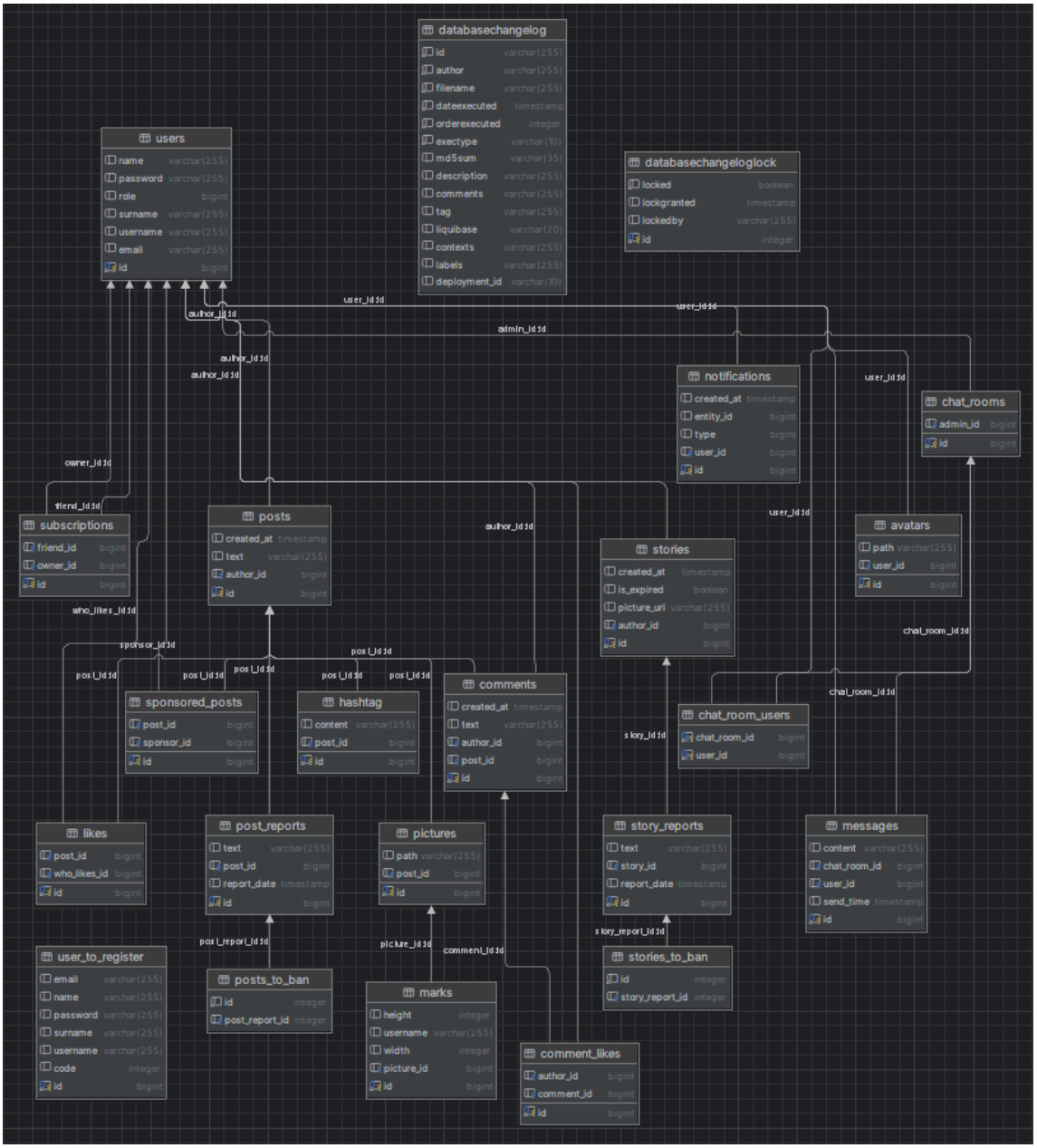

Рисунок 3.2 — Діаграма таблиць в базі даних

Liquibase забезпечує версіонування та еволюцію схеми бази даних через декларативні міграції. Система дозволяє відстежувати всі зміни в структурі бази даних, застосовувати їх поступово та при необхідності відкочувати до попередніх версій. Міграції зберігаються як SQL скрипти в папці src/main/resources/db/changelog та виконуються автоматично при запуску додатку.

Лістинг 3.3 — Файл changelog-v1.yaml

```
databaseChangeLog:
```

```
- changeSet:
    id: users creation
    author: maks
    changes:
      - sqlFile:
          path: 0001-user.sql
          relativeToChangelogFile: true

- changeSet:
    id: password reset token
    author: maks
    changes:
      - sqlFile:
         path: 0002-main-content.sql
         relativeToChangelogFile: true

- changeSet:
    id: parent
    author: maks
    changes:
      - sqlFile:
          path: 0003-report.sql
          relativeToChangelogFile: true

- changeSet:
    id: insert to parent profile
    author: maks
    changes:
      - sqlFile:
          path: 0004-chat.sql
          relativeToChangelogFile: true

- changeSet:
    id: changes in reg/login
    author: maks
    changes:
      - sqlFile:
          path: 0005-loginChanges.sql
          relativeToChangelogFile: true
```

Така організація інфраструктури забезпечує надійну основу для розробки, тестування та розгортання серверної частини соціальної мережі. Поєднання сучасних інструментів розробки, контейнеризації та управління базою даних створює ефективне середовище для командної роботи та швидкої ітеративної розробки. Модульна структура проєкту дозволяє легко орієнтуватися в коді та масштабувати систему при зростанні функціональних вимог.

3.2 Система безпеки та автентифікація

Безпека є фундаментальним аспектом будь-якої соціальної мережі, оскільки

платформа обробляє конфіденційні персональні дані користувачів, включаючи особисту інформацію, приватну переписку та поведінкові патерни. Ефективна система безпеки не лише захищає від зовнішніх загроз, але й забезпечує довіру користувачів до платформи, що є критично важливим для успіху соціальної мережі.

У сучасному цифровому середовищі соціальні мережі стикаються з широким спектром загроз безпеки. Хакери можуть намагатися отримати несанкціонований доступ до облікових записів користувачів через атаки перебору паролів, фішингові схеми або експлуатацію вразливостей в коді. SQL-ін'єкції можуть дозволити зловмисникам отримати доступ до всієї бази даних, включаючи паролі та особисту інформацію. Cross-Site Scripting (XSS) атаки можуть дозволити виконання шкідливого коду в браузерах користувачів, потенційно крадіючи сесійні токени або особисті дані.

Spring Security є основою системи безпеки проєкту, надаючи комплексний фреймворк для автентифікації та авторизації. Конфігурація Spring Security реалізована через декларативний підхід, що дозволяє точно контролювати доступ до різних частин API. Система розрізняє публічні ендпоінти, які доступні без автентифікації (реєстрація, логін, Swagger документація), захищені ендпоінти, які вимагають валідного JWT токена, та адміністративні функції, доступні лише користувачам з роллю ADMIN.

Лістинг 3.4 — Файл SecurityConfig

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    private final UserDetailsService userDetailsService;
    private final JWTFilter jwtFilter;

    public SecurityConfig(UserDetailsService userDetailsService, JWTFilter
jwtFilter) {
        this.userDetailsService = userDetailsService;
        this.jwtFilter = jwtFilter;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
                .authorizeRequests().antMatchers("/auth/login").permitAll()

.antMatchers("/report/**").hasAuthority(Role.ADMIN.getAuthority())
```

```
                .antMatchers("/auth/register").permitAll()
                .antMatchers("/auth/register/confirm").permitAll()
                .antMatchers("/swagger-ui.html").permitAll()
                .antMatchers("/webjars/**").permitAll()
                .antMatchers("/swagger-ui/**").permitAll()
                .antMatchers("/swagger-resources/**").permitAll()
                .antMatchers("/v2/**").permitAll()
                .antMatchers("/csrf").permitAll()
                .antMatchers("/").permitAll()
                .antMatchers("/test/**").permitAll()
                .anyRequest().authenticated()
                .and()
                .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        http.addFilterBefore(jwtFilter,
UsernamePasswordAuthenticationFilter.class);
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth.userDetailsService(userDetailsService)
                .passwordEncoder(getPasswordEncoder());
    }

    @Bean
    public PasswordEncoder getPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws
Exception {
        return super.authenticationManagerBean();
    }
}
```

JWT (JSON Web Token) автентифікація забезпечує stateless підхід до управління сесіями користувачів, що критично важливо для масштабованості системи. На відміну від традиційних серверних сесій, JWT токени містять всю необхідну інформацію про користувача в закодованому вигляді, що дозволяє серверу валідувати токен без звернення до бази даних або зберігання стану сесії. Це особливо важливо для горизонтального масштабування, коли запити користувача можуть обробляти різні сервери.

JWT фільтр є центральним компонентом системи автентифікації, що перехоплює всі вхідні HTTP запити та перевіряє наявність валідного токена автентифікації. Фільтр реалізований як наслідувач OncePerRequestFilter, що

гарантує його виконання рівно один раз для кожного запиту, навіть у випадку внутрішніх перенаправлень.

Лістинг 3.5 — Файл JWTFilter

```
@Component
public class JWTFilter extends OncePerRequestFilter {

    private final JWTUtil jwtUtil;
    private final AuthService authService;

    public JWTFilter(JWTUtil jwtUtil, AuthService authService) {
        this.jwtUtil = jwtUtil;
        this.authService = authService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse
            , FilterChain filterChain) throws ServletException, IOException
{
        String authHeader = httpServletRequest.getHeader("Authorization");

        String requestURI = httpServletRequest.getRequestURI();
        System.out.println(requestURI);
        List<String> list = List.of("/swagger-ui.html", "/swagger-
ui/index.html", "/v3/api-docs", "/favicon.ico", "/v2/api-docs",
"/webjars/**", "/swagger-resources/**","/test/**");
        if (list.contains(requestURI)) {
            filterChain.doFilter(httpServletRequest, httpServletResponse);
            return;
        }

        if (authHeader != null && !authHeader.isBlank() &&
authHeader.startsWith("Bearer ")) {
            String jwt = authHeader.substring(7);

            if (jwt.isBlank()) {

httpServletResponse.sendError(HttpServletResponse.SC_BAD_REQUEST,
                        "Invalid JWT Token in Bearer Header");
            } else {
                try {
                    String username =
jwtUtil.validateTokenAndRetrieveUsername(jwt);
                    UserDetails userDetails =
authService.loadUserByUsername(username);

                    UsernamePasswordAuthenticationToken authToken =
                            new
UsernamePasswordAuthenticationToken(userDetails,
                                    userDetails.getPassword(),
                                    userDetails.getAuthorities());
```

```
                    if
(SecurityContextHolder.getContext().getAuthentication() == null) {

SecurityContextHolder.getContext().setAuthentication(authToken);
                    }
                } catch (JWTVerificationException e) {

httpServletResponse.sendError(HttpServletResponse.SC_BAD_REQUEST,
                            "Invalid JWT Token");
                }
            }
        }
        filterChain.doFilter(httpServletRequest, httpServletResponse);
    }
}
```

Процес реєстрації користувачів реалізований як двоетапна процедура для забезпечення валідності email адрес та запобігання створенню фейкових облікових записів. Такий підхід також допомагає зменшити спам та зловживання системою. На першому етапі користувач надає основну інформацію (ім'я, прізвище, username, email, пароль), система валідує унікальність username та email, генерує випадковий код підтвердження та надсилає його на вказану email адресу.

Лістинг 3.6 — Код реєстрації з RegistrationService

```
public String baseRegister(UserDTO userDTO) throws
Status439UsernameBusyException, MessagingException {
    UserToRegister user = convertToUser(userDTO);
    validate(user);
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    userToRegisterRepository.save(user);
    mailSenderService.sendMessage("Registration", "Your code is " +
user.getCode(), user.getEmail());
    return user.getEmail();
}

public String endRegister(String email, int code) throws
Status455WrongCodeException {
    UserToRegister userToRegister =
userToRegisterRepository.findByEmail(email);
    if (userToRegister.getCode() == code) {
        User user = modelMapper.map(userToRegister, User.class);
        user.setRole(Role.USER);
        userRepository.save(user);
        userToRegisterRepository.delete(userToRegister);
        return jwtUtil.generateToken(user.getUsername(),
List.of(Role.USER));
    }
    throw new Status455WrongCodeException();
}
```

Другий етап реєстрації включає підтвердження email через введення отриманого коду. При успішній верифікації система переносить дані з тимчасової таблиці user_to_register до основної таблиці users, видаляє тимчасовий запис та автоматично генерує JWT токен для негайного входу користувача в систему без необхідності повторної автентифікації.

Процедура автентифікації (логіну) валідує облікові дані користувача та генерує JWT токен для подальших запитів. Система використовує email як основний ідентифікатор користувача, оскільки він є унікальним та верифікованим під час реєстрації.

Лістинг 3.7 — Код входу з LoginService

```
public String login(AuthenticationDTO authenticationDTO) throws
Status440WrongPasswordException {
    JwtUser userDetails;
    userDetails =
authService.loadUserByUsername(authenticationDTO.username());
    if (!passwordEncoder.matches(authenticationDTO.password(),
userDetails.getPassword())) {
        throw new Status440WrongPasswordException();
    }
    return jwtUtil.generateToken(userDetails.getUsername(),
(Collection<Role>) userDetails.getAuthorities());
}
```

BCrypt використовується для безпечного хешування паролів з автоматичним додаванням солі (salt), що робить паролі практично неможливими для зворотного розшифрування навіть при компрометації бази даних. BCrypt є адаптивною функцією хешування, що означає можливість налаштування складності обчислень для протидії атакам перебору з використанням більш потужного обладнання.

Система надсилання електронних листів є критично важливим компонентом для верифікації email адрес та інших комунікацій з користувачами. Реалізація базується на JavaMail API з налаштуванням для роботи через SMTP сервер Gmail, що забезпечує надійну доставку повідомлень.

Лістинг 3.8 — Код надсилання email з MailSenderServiceImpl

```
MailSenderServiceImpl() {
    Properties prop = new Properties();
    prop.put("mail.smtp.host", "smtp.gmail.com");
```

```
        prop.put("mail.smtp.port", "587");
        prop.put("mail.smtp.auth", "true");
        prop.put("mail.smtp.starttls.enable", "true");
        this.session = Session.getInstance(prop, new Authenticator() {
            @Override
            protected PasswordAuthentication getPasswordAuthentication() {
                return new PasswordAuthentication(username, password);
            }
        });
    }

    @Override
    public void sendMessage(String subject, String text, String recipient) {
        try {
            MimeMessage message = new MimeMessage(session);
            message.setText(text);
            message.setSubject(subject);
            message.addRecipient(Message.RecipientType.TO, new
    InternetAddress(recipient));
            message.setSentDate(new Date());
            message.setFrom(new InternetAddress("no-reply@gmail.com"));
            Transport.send(message);
        } catch (MessagingException e) {
            e.printStackTrace();
        }
    }
```

Система email сервісу налаштована з використанням STARTTLS для шифрованого з'єднання з SMTP сервером, що забезпечує конфіденційність передачі електронних листів. Аутентифікація до Gmail здійснюється через спеціальний пароль додатку для підвищення безпеки основного облікового запису.

Реалізована система безпеки забезпечує комплексний захист на всіх рівнях додатку — від безпечного зберігання паролів до захисту від веб-атак та надійної автентифікації користувачів. Поєднання JWT токенів з Spring Security створює масштабовану та ефективну систему управління доступом, а двоетапна реєстрація з email верифікацією гарантує якість користувацької бази. Така архітектура безпеки відповідає сучасним стандартам та забезпечує довіру користувачів до платформи.

3.3 Основний функціонал платформи

Управління профілями користувачів є центральним елементом персоналізації соціальної мережі, що дозволяє кожному учаснику створювати унікальну цифрову ідентичність та налаштовувати свою присутність на платформі. Система профілів

забезпечує не лише зберігання основної інформації про користувачів, але й складні взаємодії, такі як пошук людей, управління аватарами, редагування персональних даних та побудова соціальних зв'язків через підписки.

Профіль користувача включає базову інформацію (ім'я, прізвище, username, email), візуальне представлення через аватар, статистичні дані про активність (кількість постів, підписників, підписок), та налаштування приватності. Username служить як унікальний ідентифікатор для пошуку та згадувань в постах, тоді як email використовується для автентифікації та важливих нотифікацій.

Лістинг 3.9 — Основна логіка роботи з профілями

```
@Override
public User findUserById(Long id) throws Status437UserNotFoundException {
    User user = userRepository.findUserById(id);
    if (user == null) {
        throw new Status437UserNotFoundException();
    }
    return user;
}

@Override
public User setAvatar(String username, MultipartFile multipartFile) throws
Status441FileIsNullException {
    if (multipartFile == null) {
        throw new Status441FileIsNullException();
    }
    User user = userRepository.findUserByUsername(username);
    String uri = fileService.saveFile(user, multipartFile);
    Avatar avatar = new Avatar();
    avatar.setPath(uri);
    avatar.setUser(user);
    avatarRepository.save(avatar);
    user.addAvatar(avatar);
    return user;
}

@Override
public List<User> getFriendsList(String username) throws
Status437UserNotFoundException {
    User user = userRepository.findUserByUsername(username);
    return getUserSubscriptions(user.getId());
}

@Override
public String editMe(String username, UserDTO userDTO) throws
Status439UsernameBusyException {
    User user = userRepository.findUserByUsername(username);
    if (!username.equals(userDTO.username()) &&
userRepository.existsByUsername(userDTO.username())) {
```

```
            throw new Status439UsernameBusyException();
        }
        user.setName(userDTO.name());
        user.setUsername(userDTO.username());
        user.setSurname(userDTO.surname());
        user.setPassword(passwordEncoder.encode(userDTO.password()));
        return jwtUtil.generateToken(user.getUsername(),
    List.of(user.getRole()));
    }
```

Система завантаження аватарів реалізована з підтримкою різних форматів зображень (JPEG, PNG, GIF) та автоматичною валідацією типу та розміру файлів. Файли зберігаються в локальній файловій системі з унікальними іменами для запобігання конфліктам та оптимізації доступу. В майбутньому планується міграція до хмарного сховища AWS S3 для кращої масштабованості та надійності.

Редагування профілю дозволяє користувачам оновлювати свою інформацію з автоматичною валідацією унікальності нового username та regeneration JWT токена при зміні критичних даних. Це забезпечує безпеку та консистентність автентифікації після оновлення профілю.

Створення та управління контентом є основною функціональністю соціальної мережі, що дозволяє користувачам ділитися своїми думками, переживаннями та творчістю з аудиторією. Система підтримує публікацію мультимедійного контенту з текстовими описами, автоматичну обробку хештегів для категоризації, можливість позначення інших користувачів, та спеціальні типи контенту, такі як спонсоровані публікації.

Лістинг 3.10 — Основна логіка створення постів

```
@Override
public Post save(String username, PostDTO postDTO) throws
Status441FileIsNullException, Status436SponsorNotFoundException,
Status443DidNotReceivePictureException, Status446MarksBadRequestException,
Status437UserNotFoundException {
    User user = userRepository.findUserByUsername(username);
    List<MultipartFile> multipartFiles = postDTO.multipartFiles();
    if (multipartFiles==null || multipartFiles.isEmpty()) {
        throw new Status443DidNotReceivePictureException();
    }
    Long sponsorId = postDTO.sponsorId();
    User sponsor = null;
    if (sponsorId != null) {
        sponsor = userRepository.findUserById(sponsorId);
        if (sponsor == null) {
```

```
                throw new Status436SponsorNotFoundException();
            }
        }
        Post post = createAndSavePost(user, postDTO.text(), multipartFiles);
        hashtagService.saveAllHashtagsFromPost(post);
        if (sponsorId != null) {
            createAndSaveSponsoredPost(post, sponsor);
        }
        return post;
    }

    private Post createAndSavePost(User user, String text, List<MultipartFile>
    multipartFiles) throws Status441FileIsNullException {
        Post post = new Post();
        post.setAuthor(user);
        post.setText(text);
        post.setPubDate(LocalDateTime.now());
        postRepository.save(post);
        String uri;
        for (MultipartFile multipartFile : multipartFiles) {
            uri = fileService.saveFile(user, multipartFile);
            Picture picture = new Picture();
            picture.setPost(post);
            picture.setPath(uri);
            post.addPicture(picture);
            pictureRepository.save(picture);
        }
        checkForMarksInTextAndSendNotifications(post);
        return post;
    }
```

Процес створення поста включає кілька важливих етапів валідації та обробки. Спочатку система перевіряє наявність прикріплених зображень, оскільки платформа орієнтована на візуальний контент по типу Instagram. Далі обробляється текстовий опис з автоматичним витяганням хештегів та перевіркою на згадки інших користувачів через символ @.

Кожне зображення обробляється окремо через файловий сервіс, який забезпечує безпечне збереження, валідацію формату та розміру, генерацію унікальних шляхів для запобігання конфліктам імен файлів. Система також підтримує спонсоровані публікації, що дозволяє користувачам або брендам просувати свій контент через спеціальну мітку.

Система підписок є фундаментальним механізмом для побудови соціального графу платформи, що дозволяє користувачам формувати персоналізовані мережі зв'язків та отримувати релевантний контент від цікавих людей. Підписка створює

односторонній зв'язок, де підписник отримує оновлення від користувача, на якого підписався, але зворотний зв'язок не створюється автоматично, що відрізняє цю модель від традиційної "дружби".

Лістинг 3.11 — Основна логіка підписок

```
@Override
public User subscribe(String username, long id) throws
Status431SubscriptionException, Status437UserNotFoundException {
    User newFriend = userService.findUserById(id);
    User owner = userService.findUserByUsername(username);
    if (newFriend == null) {
        throw new Status437UserNotFoundException();
    }
    if (Objects.equals(owner, newFriend)) {
        throw new Status431SubscriptionException("You can`t subscribe to
yourself");
    }
    if (subscriptionRepository.existsByFriendAndOwner(newFriend, owner)) {
        throw new Status431SubscriptionException("You`ve already subscribe
this user");
    }
    Subscription subscription = new Subscription(owner, newFriend);
    subscriptionRepository.save(subscription);
    notificationService.sendNotification(
            new SubscriptionNotification(subscription)
    );
    return newFriend;
}
```

Система підписок включає кілька важливих валідацій для забезпечення коректності соціального графу. Перевірка на самопідписку запобігає створенню безглуздих зв'язків, коли користувач намагається підписатися на себе. Перевірка на дублікати гарантує, що між двома користувачами може існувати лише одна підписка в кожному напрямку. При успішному створенні підписки система автоматично генерує нотифікацію для користувача, на якого підписалися, інформуючи його про нового підписника.

Стрічка новин є однією з найскладніших та найважливіших функцій соціальної мережі, що агрегує контент від всіх підписок користувача в хронологічному порядку. Ефективна реалізація стрічки критично важлива для користувацького досвіду, оскільки більшість часу в соціальній мережі користувачі проводять саме переглядаючи стрічку.

Лістинг 3.12 — Основна логіка стрічки

```
@Override
public List<Post> getFeed(String username)
        throws Status442NoFeedPostsException, Status437UserNotFoundException
{
    List<Post> posts = new
ArrayList<>(getFriendsList(username).stream().flatMap
            (friend ->
postService.findAll(friend).stream().limit(5)).toList());
    if (posts.isEmpty()) {
        throw new Status442NoFeedPostsException();
    }
    posts.sort(Comparator.comparing(Post::getPubDate));
    return posts;
}
```

Поточна реалізація стрічки використовує простий алгоритм: система отримує список всіх користувачів, на яких підписаний поточний користувач, для кожного з них обмежує кількість постів (максимум 5 найновіших), об'єднує всі пости в один список та сортує їх за датою публікації. Такий підхід забезпечує справедливе представлення контенту від всіх підписок, але може бути оптимізований в майбутньому через впровадження кешування та більш складних алгоритмів ранжування.

Система хештегів забезпечує ефективну категоризацію та організацію контенту, дозволяючи користувачам легко знаходити публікації за певними темами або інтересами. Хештеги автоматично витягуються з тексту постів під час публікації та зберігаються як окремі сутності, пов'язані з відповідними постами.

Лістинг 3.13 — Основна логіка хештегів

```
@Override
public void saveAllHashtagsFromPost(Post post) {
    String[] texts = post.getText().split("#");
    List<String> hashtags = StringUtil.get(texts);
    for (String hashtagText : hashtags) {
        Hashtag hashtag = new Hashtag(hashtagText, post);
        save(hashtag);
    }
}

@Override
public List<Post> getPostsByHashtags(List<String> hashtags) {
    if (hashtags.isEmpty()) return new ArrayList<>();
    List<Hashtag> foundHashtags = new ArrayList<>();
    for (String hashtag : hashtags) {
```

```
            foundHashtags.addAll(hashtagRepository.findAllByContent(hashtag));
        }
        Set<Post> result = new HashSet<>();
        foundHashtags.stream().map(Hashtag::getPost).forEach(post -> {
            if
(post.getHashtags().stream().map(Hashtag::getContent).collect(Collectors.toS
et()).containsAll(hashtags)) {
                result.add(post);
            }
        });
        return result.stream().toList();
    }
```

Обробка хештегів включає розбиття тексту поста на фрагменти за символом #, фільтрацію валідних хештегів через утилітарний клас StringUtil, та створення окремих записів в таблиці hashtag для кожного унікального тегу. Пошук постів за хештегами підтримує як одиночні теги, так і комбінації кількох тегів, використовуючи логіку AND для знаходження постів, що містять всі вказані хештеги.

Система рекомендацій представляє інтелектуальний підхід до персоналізації контенту, аналізуючи поведінкові патерни користувачів для пропонування релевантних публікацій. Алгоритм базується на аналізі хештегів постів, які користувач вподобав раніше, припускаючи, що інтереси користувача відображаються через його взаємодії з контентом.

Лістинг 3.14 — Основна логіка рекомендацій

```
@Override
public Set<Post> getRecommendations(String username) {
    User user = userRepository.findUserByUsername(username);
    Set<Post> resultSet = new HashSet<>();
    List<List<String>> RawHashtags =
likeRepository.findAllByWhoLikes(user).stream().map(Like::getPost)
            .map(post ->
post.getHashtags().stream().map(Hashtag::getContent).collect(Collectors.toLi
st())).toList();
    Set<String> hashtags = new HashSet<>();
    for (List<String> rawHashtag : RawHashtags) {
        hashtags.addAll(rawHashtag);
    }
    for (String hashtag : hashtags) {
        List<Post> posts =
hashtagService.getPostsByHashtags(List.of(hashtag));
        resultSet.add(posts.stream().max(Comparator.comparingInt(post ->
post.getLikesList().size())).orElseThrow());
    }
```

```
    return resultSet;
}
```

Алгоритм рекомендацій працює в кілька етапів: спочатку система аналізує всі лайки користувача та витягує хештеги з вподобаних постів, формуючи профіль інтересів. Далі для кожного унікального хештегу з профілю система знаходить всі пости з цим тегом та вибирає найпопулярніший (з найбільшою кількістю лайків) як рекомендацію. Такий підхід забезпечує персоналізацію на основі продемонстрованих інтересів користувача, одночасно враховуючи популярність контенту серед спільноти.

Тимчасовий контент у вигляді Stories додає динамічний елемент до платформи, дозволяючи користувачам ділитися моментальними знімками свого життя без довгострокового зобов'язання зберігати контент. Stories автоматично зникають через 24 години, що знижує психологічний бар'єр для публікації та сприяє більш спонтанному контенту.

Лістинг 3.15 — Основна логіка історій

```
@Override
public Story createNewStory(String username, MultipartFile multipartFile)
throws Status441FileIsNullException {
    User author = userService.findUserByUsername(username);
    return storyRepository.save(
            Story.builder().author(author)
                    .pictureUrl(fileService.saveFile(author, multipartFile))
                    .createdAt(LocalDateTime.now())
                    .build());
}

@Override
@Transactional
public List<Story> getStoriesList(String username) {
    User user = userService.findUserByUsername(username);
    List<Story> stories = new ArrayList<>();
    List<Story> resultStories = new ArrayList<>();
    user.getSubscriptions().stream().map(Subscription::getFriend)
            .forEach(friend ->
stories.addAll(friend.getUnexpiredStories()));
    for (Story story : stories) {
        if (LocalDateTime.now().isAfter(story.getCreatedAt().plusDays(1))) {
            story.setExpired(true);
        } else {
            resultStories.add(story);
        }
    }
```

```
    return resultStories;
}
```

Система Stories включає автоматичну логіку для позначення застарілого контенту. При отриманні списку Stories система перевіряє час створення кожної історії та автоматично позначає як застарілі ті, що були створені більше 24 годин тому. Це забезпечує актуальність контенту та відповідає очікуванням користувачів щодо тимчасової природи Stories.

3.4 Інтерактивні функції та адміністрування

Система сповіщень є критично важливим компонентом для залучення користувачів та інформування їх про важливі події в соціальній мережі. Ефективна система нотифікацій збільшує активність користувачів, оскільки своєчасно інформує їх про нові взаємодії, такі як лайки їхніх постів, коментарі від інших користувачів, або нові підписки.

Архітектура системи сповіщень побудована на принципі розділення відповідальності між генерацією та доставкою нотифікацій. Коли відбувається подія, що потребує сповіщення (лайк, коментар, підписка), відповідний сервіс створює об'єкт нотифікації та передає його службі нотифікацій для обробки та збереження.

Лістинг 3.16 — Основна логіка сповіщень

```
@Override
public void sendNotification(Notification notification) {
    for (User recipient : notification.getRecipients()) {
        notificationRepository.save(new TextNotification(recipient,
                notification.getType(), notification.getEntityId(),
LocalDateTime.now()));
    }
    notification.clearRecipients();
}

@Override
public List<Notification> getAllNotifications(String username) {
    List<TextNotification> textNotifications =
            notificationRepository.findByOwnerUsername(username);
    List<Notification> notifications = new ArrayList<>();
    for (TextNotification textNotification : textNotifications) {
        switch (textNotification.getType()) {
            case LIKE -> {
```

```
                Optional<Like> likeOptional =

likeRepository.findById(textNotification.getEntityId());
                likeOptional.ifPresent(like ->
                        notifications.add(new LikeNotification(like)));
            }
            case COMMENT -> {
                Optional<Comment> commentOptional =

commentRepository.findById(textNotification.getEntityId());
                commentOptional.ifPresent(comment ->
                        notifications.add(new
CommentNotification(comment)));
            }
            case SUBSCRIPTION -> {
                Optional<Subscription> subscriptionOptional =

subscriptionRepository.findById(textNotification.getEntityId());
                subscriptionOptional.ifPresent(subscription ->
                        notifications.add(new
SubscriptionNotification(subscription)));
            }
        }
    }
    return notifications;
}

@Scheduled(fixedRate = 86400000) //24 hours
public void deleteOldNotifications() {
    List<TextNotification> notifications = notificationRepository.findAll();
    for (TextNotification notification : notifications) {
        if
(notification.getCreatedAt().plusDays(30).isBefore(LocalDateTime.now())) {
            notificationRepository.delete(notification);
        }
    }
}
```

Система використовує гібридний підхід до зберігання нотифікацій: основна інформація зберігається в компактній формі в таблиці TextNotification (тип події, ідентифікатор пов'язаної сутності, час створення), а детальна інформація відновлюється динамічно при отриманні списку нотифікацій через запити до відповідних репозиторіїв. Такий підхід оптимізує використання дискового простору та забезпечує актуальність інформації в нотифікаціях.

Автоматичне очищення застарілих нотифікацій реалізовано через Spring Scheduler, який кожні 24 години видаляє сповіщення старше 30 днів. Це запобігає безконтрольному зростанню бази даних та підтримує оптимальну продуктивність системи.

Система чату в реальному часі реалізована через WebSocket технологію, що забезпечує миттєвий двонаправлений зв'язок між клієнтом та сервером. На відміну від традиційних HTTP запитів, WebSocket підтримує постійне з'єднання, що дозволяє серверу ініціювати відправку повідомлень клієнту без попереднього запиту.

Лістинг 3.17 — Налаштування вебсокету

```
@ServerEndpoint(value = "/chat/{chatRoomId}", decoders =
MessageModelDecoder.class, encoders = MessageModelEncoder.class)
public class ChatWebsocketEndpoint {

    private static final Set<ChatRoom> chatRooms =
Collections.synchronizedSet(new HashSet<>());
    private final ChatRoomService chatRoomService;
    private final UserService userService;
    private final MessageService messageService;

    public ChatWebsocketEndpoint() {
        this.chatRoomService =
SpringContext.getApplicationContext().getBean(ChatRoomService.class);
        this.userService =
SpringContext.getApplicationContext().getBean(UserService.class);
        this.messageService =
SpringContext.getApplicationContext().getBean(MessageService.class);
    }

    @OnMessage
    public String onMessage(Session session, String message,
@PathParam("chatRoomId") long chatRoomId) throws IOException {
        System.out.println("Handling message: " + message);
        User user =
userService.findUserByUsername(session.getUserPrincipal().getName());
        ChatRoom chatRoom = chatRooms.stream().filter
                (chatRoom1 -> chatRoom1.getId() ==
chatRoomId).findAny().orElseThrow();
        MessageModel model = MessageModel.builder()
                .chatRoom(chatRoom).user(user).content(message).build();
        messageService.save(model);
        for (Session s : chatRoom.getSessions()) {
            s.getBasicRemote().sendText(message);
        }
        return null;
    }

    @OnOpen
    public void onOpen(Session session, @PathParam("chatRoomId") long
chatRoomId)
            throws Status453NotYourChatRoomException,
Status452ChatRoomNotFoundException {
        System.out.println("On open: " + session.getId());
        ChatRoom chatRoom = chatRooms.stream().filter
```

```
                        (chatRoom1 -> chatRoom1.getId() ==
chatRoomId).findAny()
                .orElse(chatRoomService.findById(chatRoomId,
session.getUserPrincipal().getName()));
        chatRoom.addSession(session);
        chatRooms.add(chatRoom);
    }

    @OnClose
    public void onClose(Session session, @PathParam("chatRoomId") long
chatRoomId) {
        System.out.println("On close: " + session.getId());
        ChatRoom chatRoom = chatRooms.stream().filter
                (chatRoom1 -> chatRoom1.getId() ==
chatRoomId).findAny().orElse(new ChatRoom());
        chatRoom.removeSession(session);
    }
}
```

WebSocket ендпоінт управляє життєвим циклом чат-кімнат та WebSocket сесій. При підключенні нового користувача система перевіряє його права доступу до конкретної чат-кімнати, додає його сесію до списку активних учасників, та зберігає кімнату в пам'яті для швидкого доступу. Коли користувач надсилає повідомлення, система зберігає його в базі даних для збереження історії та ретранслює всім активним учасникам кімнати в реальному часі.

Система використовує thread-safe колекції (Collections.synchronizedSet) для управління одночасним доступом від кількох користувачів, що забезпечує стабільність роботи під навантаженням.

Лістинг 3.18 — Код декодера та енкодера

```
public class MessageModelDecoder implements Decoder.Text<MessageModel> {
    Gson gson = new Gson();

    @Override
    public MessageModel decode(String s) {
        return gson.fromJson(s, MessageModel.class);
    }
}

public class MessageModelEncoder implements Encoder.Text<MessageModel> {
    Gson gson = new Gson();

    @Override
    public String encode(MessageModel message) {
        return gson.toJson(message);
    }
}
```

Кастомні декодери та енкодери забезпечують правильну серіалізацію та десеріалізацію повідомлень між JSON форматом, що використовує клієнт, та Java об'єктами на сервері. Використання Gson бібліотеки забезпечує ефективну обробку JSON даних з автоматичним мапінгом полів.

Система модерації контенту через скарги користувачів є важливим інструментом для підтримки безпечного та дружнього середовища в соціальній мережі. Адміністратори мають спеціальні інструменти для перегляду скарг, аналізу проблемного контенту та прийняття рішень щодо модерації.

Лістинг 3.19 — Код репортів для адміністраторів.

```
@Override
public List<PostReportDTO> getPostReports() {
    List<PostsToBan> postsToBan = postsToBanRepository.findAll();
    if (postsToBan.isEmpty()) {
        return new ArrayList<>();
    }
    List<PostReportDTO> reports = new ArrayList<>();

    postsToBan.sort((o1, o2) -> Math.toIntExact
            (o1.getPostReport().getPost().getId() -
o2.getPostReport().getPost().getId()));

    Post postWhichWeAreWorkingWith =
postsToBan.get(0).getPostReport().getPost();
    PostReportDTO postReportDTO = new
PostReportDTO(postWhichWeAreWorkingWith);

    for (PostsToBan postToBan : postsToBan) {
        PostReport report = postToBan.getPostReport();
        if (report.getPost() != postWhichWeAreWorkingWith) {
            postWhichWeAreWorkingWith = report.getPost();
            reports.add(postReportDTO);
            postReportDTO = new PostReportDTO(report.getPost());
        }
        postReportDTO.addReportText(report.getText());
    }
    reports.add(postReportDTO);
    reports.sort(Comparator.comparingInt(o -> o.reportTexts().size()));
    return reports;
}

@Override
@Transactional
public void acceptPostReport(Long id) throws
Status451ReportWithThisIdIsNotFound {
    if (!postReportRepository.existsByPost_Id(id)) {
        throw new Status451ReportWithThisIdIsNotFound();
```

```
    }
    postReportRepository.findByPost_Id(id).stream().map(PostReport::getId)
            .forEach(postsToBanRepository::deleteByPostReport_Id);
    postReportRepository.deleteAllByPost_Id(id);
    postService.banPost(id);
}
```

Система агрегує всі скарги на конкретний пост та представляє їх адміністратору в зручному форматі, групуючи за постом та сортуючи за кількістю скарг. Це дозволяє адміністраторам швидко ідентифікувати найпроблемніший контент та приймати обґрунтовані рішення про модерацію.

Лістинг 3.20 — Автоматична обробка скарг

```
@Override
public Post reportPost(Long id, String text) throws
Status435PostNotFoundException {
    Post post =
postRepository.findById(id).orElseThrow(Status435PostNotFoundException::new)
;
    PostReport postReport = PostReport.builder()
            .text(text)
            .post(post)
            .build();
    postReportRepository.save(postReport);
    additionalPostReport(postReport);
    return post;
}

private void additionalPostReport(PostReport postReport) {
    int thisPostReportsCount =
postReportRepository.countAllByPost_Id(postReport.getPost().getId());
    if (thisPostReportsCount == 10) {
        postsToBanRepository.saveAll(
                postReportRepository.findAllByPost(postReport.getPost())

.stream().map(PostsToBan::new).collect(Collectors.toList()));
    } else if (thisPostReportsCount > 10) {
        postsToBanRepository.save(new PostsToBan(postReport));
    }
}
```

Автоматична система реагування на скарги відстежує кількість скарг на кожен пост та автоматично передає контент на розгляд адміністраторів після досягнення порогу в 10 скарг. Це забезпечує швидку реакцію на потенційно проблемний контент навіть за відсутності постійного моніторингу з боку адміністраторів.

Централізована обробка помилок забезпечує консистентну та зрозумілу комунікацію помилок між сервером та клієнтськими додатками. Система використовує Spring @ControllerAdvice для глобального перехоплення винятків та їх перетворення в зрозумілі HTTP відповіді.

Лістинг 3.21 — Обробка помилок

```
@ControllerAdvice
public class ExceptionsHandler {
    @ExceptionHandler(CustomException.class)
    public ResponseEntity<Object>
handleAlreadyExistException(CustomException ex) {
        return ResponseEntity.status(ex.code).body(ex.getMessage());
    }

    @ExceptionHandler(SizeLimitExceededException.class)
    public ResponseEntity<Object> handleException(SizeLimitExceededException
ex){
        return ResponseEntity.status(430).body("File is too big");
    }

    @ExceptionHandler(UsernameNotFoundException.class)
    public ResponseEntity<Object> handleException(UsernameNotFoundException
ex){
        return ResponseEntity.status(429).body("User with this username is
not found");
    }
}

public class CustomException extends Exception {
    public final int code;

    public CustomException(int code, String message) {
        super(message);
        this.code = code;
    }
}
```

Система включає спеціальні винятки з кастомними HTTP кодами для різних типів помилок, що дозволяє клієнтським додаткам правильно реагувати на різні ситуації. CustomException є базовим класом для всіх бізнес-помилок з можливістю вказання специфічного статус коду.

Swagger документація служить центральним інструментом для документування та тестування API, автоматично генеруючи інтерактивну документацію на основі анотацій в коді контролерів. Це значно спрощує процес розробки клієнтських додатків та фронтенд інтеграції.

Лістинг 3.22 — Налаштування Swagger:

```
@Configuration

@EnableSwagger2
public class SwaggerConfig {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()

.apis(RequestHandlerSelectors.basePackage("com.fivesysdev.Fiveogram.controll
ers"))
                .paths(PathSelectors.any())
                .build()
                .apiInfo(apiInfo())
                .securitySchemes(List.of(apiKey()))

.securityContexts(Collections.singletonList(securityContext()));
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
                .title("API")
                .description("Vovchuk Maksym, 2KI-23ms, 2025")
                .version("1.0.0")
                .termsOfServiceUrl(null)
                .build();
    }

    private ApiKey apiKey() {
        return new ApiKey("JWT", "Authorization", "header");
    }

    private SecurityContext securityContext() {
        return SecurityContext.builder()
                .securityReferences(defaultAuth())
                .forPaths(PathSelectors.any())
                .build();
    }

    private List<SecurityReference> defaultAuth() {
        AuthorizationScope authorizationScope = new
AuthorizationScope("global", "accessEverything");
        AuthorizationScope[] authorizationScopes = new
AuthorizationScope[1];
        authorizationScopes[0] = authorizationScope;
        return List.of(new SecurityReference("JWT", authorizationScopes));
    }
}
```

Конфігурація Swagger включає налаштування JWT автентифікації, що дозволяє тестувати захищені ендпоінти безпосередньо через веб-інтерфейс

документації. Розробники можуть ввести JWT токен один раз та використовувати його для всіх наступних запитів, що значно спрощує процес тестування.

Swagger автоматично сканує всі контролери в вказаному пакеті, витягує інформацію про ендпоінти, параметри запитів, формати відповідей та генерує повну інтерактивну документацію. Документація оновлюється автоматично при зміні коду. Вигляд інтерфейсу Swagger зображено на рисунку 3.3.

Рисунок 3.3 — Вигляд інтерфейсу Swagger

Реалізація серверної частини соціальної мережі демонструє успішне поєднання сучасних технологій та архітектурних підходів для створення масштабованого та безпечного рішення. Використання Java з Spring Boot забезпечило швидку розробку та надійну роботу системи, PostgreSQL гарантує цілісність та консистентність даних, а Docker спрощує розгортання та підтримку. Впровадження JWT автентифікації, комплексної системи сповіщень, WebSocket чату та інструментів модерації створює повнофункціональну платформу, готову для обслуговування реальних користувачів. Swagger документація та централізована обробка помилок забезпечують зручність розробки клієнтських додатків та подальшої підтримки системи. Архітектура закладає міцний фундамент для майбутнього масштабування та додавання нових функцій без кардинальних змін в існуючій кодовій базі.

ВИСНОВКИ

У роботі проведено аналіз сучасних соціальних мереж, таких як Instagram, Twitter і TikTok, з акцентом на їхні серверні архітектури та функціональні можливості. Визначено ключові функції, включаючи управління профілями, створення контенту, персоналізовані стрічки, сповіщення, рекомендації та модерацію контенту, а також виклики, пов’язані з масштабуванням, безпекою та залученням користувачів. Успіх таких платформ залежить від інтеграції сучасних технологій, надійної інфраструктури та зручного користувацького досвіду.

Розроблено серверну частину проєкту соціальної мережі на основі монолітної архітектури з використанням Java 19 і Spring Boot. Реалізовано основні функції: управління профілями користувачів, створення та обробку контенту, автентифікацію й авторизацію з JWT, сповіщення в реальному часі, підписки, ефемерний контент у вигляді історій, систему рекомендацій на основі аналізу взаємодій користувачів, чат через WebSocket, відправлення email для верифікації, логіку стрічки з підтримкою хештегів, обробку скарг адміністраторами та централізовану обробку помилок. Для управління базою даних використано Liquibase, що забезпечує автоматизацію SQL-скриптів, а Docker — ізольоване розгортання. API задокументовано за допомогою Swagger, що полегшує інтеграцію для розробників клієнтських додатків.

Тестування системи підтвердило її стабільність і коректну роботу всіх основних функцій, включаючи створення контенту, доставку сповіщень, обробку скарг і реал-тайм взаємодію через чат. Проєкт є функціональним прототипом соціальної мережі з потенціалом для комерційного використання, зокрема в ніші платформ для творчого самовираження чи локальних спільнот. Система забезпечує гнучкість і може бути адаптована для ширшого застосування завдяки монолітній архітектурі та сучасним технологіям.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Сучасні підходи до розробки веб-додатків з використанням Spring Boot / О. В. Петренко, М. І. Іваненко // Матеріали конференції LIV Всеукраїнської науково-технічної конференції факультету інформаційних технологій та комп'ютерної інженерії. Вінниця 2025 р. [Електронний ресурс]. Режим доступу: https://conf.vntu.edu.ua/index.php/all-fitki/all-fitki2025/.

2. Walls C. Spring Boot in Action. 3rd ed. Manning Publications, 2023. 592 p.

3. Інформаційні технології та програмування: підручник / Азаров О. Д., Захарченко С. М., Кадук О. В., Орлова М. М., Тарасенко В. П. — Вінниця: ВНТУ, 2022. 412 с.

4. Коробейнікова Т. І., Захарченко С. М. Сучасні веб-технології та їх застосування. Львів: Видавництво Львівської політехніки, 2023. 284 с.

5. Richardson C., Smith F. Microservices Patterns: With examples in Java. Manning Publications, 2023. 520 p.

6. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2021. 560 p.

7. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2022. 432 p.

8. Fowler M., Lewis J. Microservices: a definition of this new architectural term. 2023. Режим доступу: https://martinfowler.com/articles/microservices.html

9. Spring Framework Documentation. Spring Boot Reference Guide. 2024. Режим доступу: https://spring.io/projects/spring-boot

10. Гейєр Д. Розробка RESTful веб-сервісів. К.: Діалектика, 2023. 324 с.

11. Freeman A. Pro Spring 6: An In-Depth Guide to the Spring Framework. 6th ed. Apress, 2024. 896 p.

12. RFC 7519. JSON Web Token (JWT). Network Working Group, 2023. Режим доступу: https://datatracker.ietf.org/doc/html/rfc7519

13. RFC 6749. The OAuth 2.0 Authorization Framework. Network Working Group, 2022. Режим доступу: https://datatracker.ietf.org/doc/html/rfc6749

14. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. ACM SIGCOMM Computer Communication Review, 2021, vol. 51(2), pp. 15-28.

15. PostgreSQL Global Development Group. PostgreSQL 15 Documentation. 2024. Режим доступу: https://www.postgresql.org/docs/15/

16. Docker Inc. Docker Documentation. Best Practices for Writing Dockerfiles. 2024. Режим доступу: https://docs.docker.com/develop/dev-best-practices/

17. Хант К. Java: повний довідник. 12-е вид. К.: Діалектика, 2023. 1248 с.

18. Bloch J. Effective Java. 4th ed. Addison-Wesley Professional, 2023. 416 p.

19. Newman S. Building Microservices: Designing Fine-Grained Systems. 3rd ed. O'Reilly Media, 2024. 624 p.

20. OWASP Foundation. OWASP Top Ten Web Application Security Risks. 2023. Режим доступу: https://owasp.org/www-project-top-ten/

21. NIST Special Publication 800-63B. Authentication and Lifecycle Management. 2023. Режим доступу: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf

22. Лін В. Д. Сучасні технології веб-розробки: архітектура, безпека, тестування. К.: Техніка, 2023. 524 с.

23. Beck K. Test Driven Development: By Example. 2nd ed. Addison-Wesley Professional, 2022. 240 p.

24. Swagger Documentation. OpenAPI Specification v3.1.0. 2024. Режим доступу: https://spec.openapis.org/oas/v3.1.0

25. Міщенко В. А., Петров О. І. Методи забезпечення інформаційної безпеки веб-додатків // Кібербезпека: освіта, наука, техніка. 2023. № 3(19). С. 45-58.

ДОДАТОК А

Технічне завдання

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

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
проф., д.т.н.. Азаров О.Д.
21 березня 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ
на виконання бакалаврської кваліфікаційної роботи
«СЕРВЕРНА ЧАСТИНА СОЦІАЛЬНОЇ МЕРЕЖІ»
08-54.БКР.022.00.000 ТЗ

Науковий керівник: ст. викл. каф. ОТ
_____Кисюк Д. В.

Студент групи 2КІ-23мс
_____Вовчук М. О.

м. Вінниця — 2025

1 Підстава для використання бакалаврської кваліфікаційної роботи (БКР)

1.1 Актуальність розробки полягає у зростанні потреби в сучасних соціальних платформах, що забезпечують безпечну та ефективну комунікацію між користувачами в умовах цифровізації суспільства. Розробка серверної частини соціальної мережі з використанням Java та Spring Boot дасть можливість створити надійну, масштабовану систему, що відповідає сучасним стандартам веб-розробки та забезпечує високий рівень безпеки користувацьких даних.

1.2 Головним завданням розробки є створення повнофункціональної серверної частини соціальної мережі з використанням монолітної архітектури, REST API, системи автентифікації та авторизації, ефективної структури бази даних та сучасних технологій розгортання.

1.3 Наказ про затвердження теми бакалаврської кваліфікаційної роботи від 11 березня 2025 р. №80.

2 Мета і призначення БКР

2.1 Мета роботи полягає у розробці серверної частини соціальної мережі з метою забезпечення високого рівня безпеки та продуктивності через створення монолітної архітектури на базі Spring Boot, реалізацію REST API для всіх основних операцій, впровадження JWT автентифікації та Spring Security, проєктування оптимізованої структури PostgreSQL бази даних, розробку системи тестування та документації, а також організацію контейнеризованого розгортання через Docker.

2.2 Призначення розробки полягає у виконанні бакалаврської кваліфікаційної роботи, за результатами якої буде створено повнофункціональну серверну частину соціальної мережі з можливістю подальшого розвитку та масштабування системи.

3 Вихідні дані для виконання БКР

3.1 Аналіз існуючих соціальних мереж та їх архітектурних рішень, включно з вивченням підходів до організації серверної частини, методів зберігання даних

та механізмів забезпечення безпеки.

3.2 Огляд і аналіз сучасних технологій веб-розробки: платформи Java, фреймворку Spring Boot, принципів REST API, систем управління базами даних (PostgreSQL), технологій автентифікації (JWT), методів тестування та контейнеризації (Docker).

3.3 Визначення вимог до серверного програмного забезпечення та архітектури системи, які відповідають потребам сучасної соціальної мережі і забезпечують високий рівень продуктивності, масштабованості та безпеки.

3.4 Планування структури бази даних, включаючи моделювання сутностей користувачів, постів, коментарів, вподобань та підписок, а також оптимізацію зв'язків між ними.

3.5 Оцінка технологічних рішень і вибір оптимального стеку технологій з погляду ефективності розробки та подальшої підтримки системи.

3.6 Розробка плану реалізації та тестування серверної частини з урахуванням необхідності перевірки всіх API ендпоінтів та функціональності системи.

4 Вимоги до виконання БКР

Головна вимога полягає в створенні ефективної та надійної серверної частини соціальної мережі з монолітною архітектурою, яка забезпечить:

— реалізацію повного функціоналу соціальної мережі: реєстрацію, автентифікацію, створення постів, коментування, вподобання та підписки;

— безпечну автентифікацію та авторизацію користувачів через JWT токени та Spring Security;

— ефективну структуру PostgreSQL бази даних з оптимізованими індексами та зв'язками;

— REST API з підтримкою всіх CRUD операцій, пагінації та валідації даних;

— комплексне тестування системи через unit та інтеграційні тести;

— повну документацію API через Swagger та контейнеризоване розгортання через Docker.Етапи БКР та очікувані результати

5 Етапи розробки БКР

Етапи розробки БКР та очікувані результати наведено у Таблиці А.1

Таблиця А.1 — Етапи БКР

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		Початок	Кінець	
1	Постановка задачі	21.03.25	21.03.25	Розділ 1
2	Аналіз соціальних мереж	21.03.25	30.03.25	Розділ 1
3	Технології розробки	21.03.25	30.04.25	Розділ 2
4	Розробка серверної частини мережі	31.03.25	13.04.25	Розділ 3
5	Реалізація	14.04.25	27.04.25	Розділ 4
6	Підготовка тезових матеріалів для публікації	21.04.25	28.04.25	Тези
7	Оформлення пояснювальної записки та демонстраційної презентації для доповіді	29.04.25	12.05.25	ПЗ, презентація
8	Підготовка супровідної документації, перевірка відповідності нормам та проходження тесту на плагіат	13.05.25	13.05.25	ПЗ, презентація

6 Матеріали, що подаються до захисту БКР

До захисту подаються: пояснювальна записка БКР, ілюстративні матеріали (схеми архітектури системи, діаграми бази даних, приклади API документації), протокол попереднього захисту БКР на кафедрі, відгук наукового керівника, анотації до БКР українською та іноземною мовами, довідка про відповідність оформлення БКР чинним вимогам.

7 Порядок контролю виконання та захисту БКР

Виконання етапів графічної та розрахункової документації БКР контролюється науковим керівником ст. викл. каф. ОТ Кисюк Д. В. згідно зі встановленими термінами. Захист БКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора ВНТУ.

8 Вимоги до оформлювання та порядок виконання БКР

8.1 При оформлюванні БКР використовуються:

— ДСТУ 3008 : 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302 : 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— методичні вказівки до виконання бакалаврських кваліфікаційних робіт для студентів спеціальності 123 «Комп'ютерна інженерія» (освітня програма «Комп'ютерна інженерія») — кафедра обчислювальної техніки, ВНТУ 2023.

— Порядок виконання БКР викладено в «Положення про кваліфікаційні роботи на першому (бакалаврському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21»

ДОДАТОК Б

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: «Серверна частина соціальної мережі»

Тип роботи: Бакалаврська кваліфікаційна робота
(БДР, МКР)

Підрозділ кафедра обчислювальної техніки
(кафедра, факультет)

Показники звіту подібності StrikePlagiarism

Оригінальність 98 % Схожість 2 %

Аналіз звіту подібності (відмітити потрібне):

✓ Запозичення, виявлення у роботі, оформлені коректно і не містять ознак плагіату.

☐ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.

☐ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

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

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

Автор роботи (підпис) Вовчук М. О (прізвище, ініціали)

Керівник роботи (підпис) Кисюк Д. В. (прізвище, ініціали)

ДОДАТОК В

ІЛЮСТРАТИВНА ЧАСТИНА

Серверна частина соціальної мережі

Рисунок В — Вигляд інтерфейсу Swagger

ДОДАТОК Г

Лістинги файлів

Лістинг Г.1 — Сутність користувача

```
@Getter
@Setter
@ToString
@Builder
@AllArgsConstructor
@Entity
@Table(name = "users")
public class User extends BaseEntity {
    @Column(name = "name")
    private String name;

    @Column(name = "surname")
    private String surname;

    @Column(name = "username")
    private String username;

    @Column(name = "password")
    private String password;

    @Column(name = "role")
    private Role role;

    @Column(name = "email")
    private String email;

    @LazyCollection(LazyCollectionOption.TRUE)
    @OneToMany(mappedBy = "user")
    @ToString.Exclude
    private List<Avatar> avatars;

    @LazyCollection(LazyCollectionOption.TRUE)
    @OneToMany(mappedBy = "owner")
    @ToString.Exclude
    private List<Subscription> subscriptions;

    @OneToMany(mappedBy = "author")
    @LazyCollection(LazyCollectionOption.TRUE)
    @JsonIgnore
    @ToString.Exclude
    private List<Story> stories;

    public User() {
        subscriptions = new ArrayList<>();
        avatars = new ArrayList<>();
        stories = new ArrayList<>();
    }

    public void addAvatar(Avatar avatar) {
        avatars.add(avatar);
```

```
    }

    @JsonIgnore
    public List<Story> getUnexpiredStories() {
        List<Story> unexpiredStories = new ArrayList<>();
        for (Story story : stories) {
            if (!story.isExpired()) {
                unexpiredStories.add(story);
            }
        }
        return unexpiredStories;
    }
```

Лістинг Г.2 — Сутність поста

```
@Getter
@Setter
@ToString
@Builder
@AllArgsConstructor
@Table(name = "posts")
@Entity
public class Post extends BaseEntity {
    @ManyToOne
    @JoinColumn(name = "author_id", referencedColumnName = "id")
    @JsonIgnoreProperties({"subscriptions"})
    private User author;

    @Column(name = "text")
    private String text;

    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval
= true)
    @ToString.Exclude
    private List<Picture> pictures;

    @Column(name = "created_at")
    @CreatedDate
    private LocalDateTime pubDate;

    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval
= true)
    @ToString.Exclude
    private List<Like> likesList;

    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval
= true)
    @ToString.Exclude
    private List<Comment> commentList;

    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval
= true)
    @ToString.Exclude
    private List<Hashtag> hashtags;

    public void addPicture(Picture picture) {
```

```
            pictures.add(picture);
        }

        public Post() {
            pictures = new ArrayList<>();
        }
```

Лістинг Г.3 — Конфігурація безпеки

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    private final UserDetailsService userDetailsService;
    private final JWTFilter jwtFilter;

    public SecurityConfig(UserDetailsService userDetailsService, JWTFilter
jwtFilter) {
        this.userDetailsService = userDetailsService;
        this.jwtFilter = jwtFilter;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
                .authorizeRequests().antMatchers("/auth/login").permitAll()

.antMatchers("/report/**").hasAuthority(Role.ADMIN.getAuthority())
                .antMatchers("/auth/register").permitAll()
                .antMatchers("/auth/register/confirm").permitAll()
                .antMatchers("/swagger-ui.html").permitAll()
                .antMatchers("/webjars/**").permitAll()
                .antMatchers("/swagger-ui/**").permitAll()
                .antMatchers("/swagger-resources/**").permitAll()
                .antMatchers("/v2/**").permitAll()
                .antMatchers("/csrf").permitAll()
                .antMatchers("/").permitAll()
                .antMatchers("/test/**").permitAll()
                .anyRequest().authenticated()
                .and()
                .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        http.addFilterBefore(jwtFilter,
UsernamePasswordAuthenticationFilter.class);

    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
        auth.userDetailsService(userDetailsService)
                .passwordEncoder(getPasswordEncoder());
    }

    @Bean
    public PasswordEncoder getPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }
```

```
    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
}
```

Лістинг Г.4 — Контролер постів

```
@RestController
@AllArgsConstructor
@RequestMapping("/post")
@Api(value = "Post endpoints", tags = {"Post"})
public class PostController {
    private final PostService postService;
    private final CommentService commentService;
    private final ReportService reportService;
    private final LikeService likeService;
    private final JWTUtil jwtUtil;
    private final HashtagService hashtagService;

    @PostMapping()
    public Response<Post> addNewPost(@ModelAttribute PostDTO postDTO,
                                     @ApiParam(hidden = true)
@RequestHeader(value = "Authorization") String token)
            throws Status441FileIsNullException,
Status436SponsorNotFoundException,
            Status443DidNotReceivePictureException,
Status446MarksBadRequestException, Status437UserNotFoundException {
        return new Response<>(postService.save(jwtUtil.getUsername(token),
postDTO));
    }

    @PostMapping("/addMarks")
    public Response<Post> addMarks(@RequestBody MarksToAddDTO marksToAddDTO,
                                   @ApiParam(hidden = true)
@RequestHeader(value = "Authorization") String token)
            throws Status449PictureNotFoundException,
Status433NotYourPostException, Status437UserNotFoundException {
        return new Response<>(postService.addMarks(jwtUtil.getUsername(token),
marksToAddDTO.markDTOs()));
    }

    @GetMapping("/{id}")
    public Response<Post> getPost(@PathVariable long id) throws
Status435PostNotFoundException {
        return new Response<>(postService.findPostById(id));
    }

    @PatchMapping("/{id}/edit")
    public Response<Post> editPost(@PathVariable long id, @RequestBody PostDTO
postDTO,
                                   @ApiParam(hidden = true)
@RequestHeader(value = "Authorization") String token)
            throws Status441FileIsNullException, Status435PostNotFoundException,
```

```
            Status433NotYourPostException, Status437UserNotFoundException,
Status446MarksBadRequestException {
        return new Response<>(postService.editPost(jwtUtil.getUsername(token),
postDTO, id));
    }

    @DeleteMapping("/{id}/delete")
    public Response<List<Post>> deletePost(@PathVariable long id,
                                           @ApiParam(hidden = true)
@RequestHeader(value = "Authorization") String token)
            throws Status435PostNotFoundException, Status433NotYourPostException
{
        return new Response<>(postService.deletePost(jwtUtil.getUsername(token),
id));
    }

    @PostMapping("/{id}/addComment")
    public Response<Comment> addComment(@PathVariable long id, @RequestParam
@Nullable String text,
                                        @ApiParam(hidden = true)
@RequestHeader(value = "Authorization") String token)
            throws Status435PostNotFoundException, Status448TextIsNullException
{
        return new Response<>(commentService.save(jwtUtil.getUsername(token),
id, text));
    }

    @PostMapping("/{id}/setLike")
    public Response<Post> addLike(@PathVariable long id,
                                  @ApiParam(hidden = true)
@RequestHeader(value = "Authorization") String token)
            throws Status437UserNotFoundException,
Status438PostAlreadyLikedException,
            Status435PostNotFoundException {
        return new Response<>(likeService.likePost(jwtUtil.getUsername(token),
id));
    }

    @PostMapping("/{id}/deleteLike")
    public Response<Post> deleteLike(@PathVariable long id,
                                     @ApiParam(hidden = true)
@RequestHeader(value = "Authorization") String token)
            throws Status435PostNotFoundException {
        return new Response<>(likeService.unlikePost(jwtUtil.getUsername(token),
id));
    }

    @GetMapping("/{id}/getLikes")
    public Response<Set<Like>> getLikes(@PathVariable long id) throws
Status435PostNotFoundException {
        return new Response<>(likeService.findAllPostLikes(id));
    }

    @PostMapping("/{id}/report")
    public Response<Post> report(@PathVariable long id, @RequestParam String
text)
```

```
            throws Status435PostNotFoundException {
        return new Response<>(reportService.reportPost(id,text));
    }
    @GetMapping("/search")
    public Response<List<Post>> searchByHashtags(@RequestBody List<String>
hashtags){
        return new Response<>(hashtagService.getPostsByHashtags(hashtags));
    }

    @GetMapping("/getRecommendations")
    public Response<Set<Post>> getMyRecommendations(@ApiParam(hidden = true)
@RequestHeader(value = "Authorization") String token){
        return new
Response<>(postService.getRecommendations(jwtUtil.getUsername(token)));
    }
}
```