Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

Допущено до захисту

Завідувач кафедри комп'ютерних наук,
кандидат технічних наук, доцент,
_____ Ірина МАШКІНА

(підпис)

« ___ » _____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «бакалавр» Спеціальність 122 Комп'ютерні науки Освітня програма 122.00.01 Інформатика

Тема:

Розробка вебзастосунку для музичної групи з функціоналом електронної комерції

Виконав
студент групи ІНб-2-21-4.0д
Когутко Сергій Сергійович

(ПІБ)

_____

(підпис)

Науковий керівник
кандидат технічних наук, доцент

_____ Ірина МЕЛЬНИК

(підпис)

Київ – 2025

Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

Допущено до захисту
Завідувач кафедри комп'ютерних наук,
кандидат технічних наук, доцент,
_____ Ірина МАШКІНА
(підпис)
« ___ » _____ 2025 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІНУ РОБОТУ БАКАЛАВРА

«Розробка вебзастосунку для музичної групи з функціоналом електронної комерції»

Виконавець – студент групи ІНб-2-21-4.0д
Когутко Сергій Сергійович

Вихідні дані: провести аналітичний огляд наукових публікацій з обраної теми, визначити та обґрунтувати мету, об'єкт, предмет і теоретичну базу дослідження; сформувати завдання, структуру та зміст бакалаврської роботи; обрати й обґрунтувати методи та інструменти дослідження; підготувати матеріали до відповідних розділів роботи згідно з індивідуальним завданням; здійснити практичну реалізацію проєкту, пов'язаного з представлення веб-застосунку для музичного гурту.

Пояснювальна записка: Обсяг – до 52 стор. формату А4 комп'ютерного набору з дотриманням вимог стандарту і методичних рекомендацій кафедри.

- Графічні матеріали: Презентація, Ілюстрації
- Строк подання роботи на кафедру: «5» червня 2025 р.

Науковий керівник:
кандидат технічних наук
доцент
_____ Яскевич В.О.
дата

Виконавець:
_____
дата

Календарний план

№	Назва етапу дипломної роботи	Строк виконання етапу роботи	Примітка
1	Формування теми дипломної роботи бакалавра	25.12.2024	Створення теми для бакалаврської дипломної роботи
2	Ознайомлення з методичними рекомендаціями до дипломної роботи	12.01.2025	Створення архітектури роботи
3	Складання змісту та календарного плану роботи	24.01.2025	Вибір стеку технологій
4	Підбір літератури, складання завдання	01.02.2025	Пошук онлайн-курсів для підготовки
5	Написання реферату та вступу	07.02.2025	Створення концепт-документу
6	Написання першого розділу	20.02.2025	Створення MVP
7	Написання другого розділу	6.03.2025	Завершення MVP та перехід до повноцінної роботи над модулями
8	Написання третього розділу	29.03.2025	Завершення написання коду
9	Написання висновків	07.04.2025	Розгортання роботи в хмарі

10	Виправлення зауважень	14.04.2025	Початок написання дипломної роботи
11	Створення презентації	19.05.2025	
13	Офіційний захист матеріалів дипломної роботи на засіданні екзаменаційної комісії	05.06.2025	

Здобувач_____

Керівник роботи _____

РЕФЕРАТ

Кваліфікаційна робота: 51 сторінка, 23 джерела, 11 ілюстрацій.

Актуальність роботи зумовлена критичною важливістю ефективної онлайн-присутності для музичних гуртів у сучасній цифровій музичній індустрії. Існує потреба в централізованих платформах, що дозволяють гуртам контролювати представлення своєї творчості, безпосередньо взаємодіяти з фанатами та вести комерційну діяльність, долаючи обмеження розрізнених сторонніх сервісів.

Об’єктом дослідження є процес розробки вебзастосунку для музичного гурту з акцентом на забезпеченні ефективної взаємодії з фанатами, продажу продукції та представленні творчості. Предметом дослідження є функціональні можливості, архітектура, технології, процес розробки та результати тестування вебзастосунку Threllia.

Мета роботи: проєктування, розробка та тестування повнофункціонального вебзастосунку Threllia з використанням сучасних веб-технологій (React.js для Frontend та Spring Boot для Backend) для створення централізованої онлайн-платформи, що надає фанатам повний доступ до інформації про музичний гурт, його творчість та можливості придбання офіційного мерчу. Для досягнення мети вирішено завдання: аналіз вимог; проектування архітектури клієнтської (React.js) та серверної (Spring Boot) частин та їх взаємодії через REST API; розробка функціональних модулів ("Музика", "Пісні", "Новини", "Тури", "Магазин", "Медіа-галерея", "Панель адміністратора"); реалізація системи автентифікації та авторизації (Spring Security, JWT); інтеграція з базою даних MySQL на AWS RDS та сервісом Cloudinary для медіафайлів; комплексне тестування.

Спроєктовано багаторівневу архітектуру вебзастосунку Threllia з розділенням на Frontend (React.js) та Backend (Spring Boot), що взаємодіють через REST API. Розроблено повнофункціональний вебзастосунок з реалізованими модулями, системою керування станом Redux, базою даних

MySQL (AWS RDS) та системою безпеки Spring Security. Проведено комплексне тестування, що підтвердило відповідність вимогам, зручність використання та ефективність. Вебзастосунок успішно розгорнуто на хмарних платформах Netlify та Render.

Розроблений вебзастосунок Threllia рекомендується для використання музичними гуртами як інструмент для забезпечення ефективної онлайн-присутності, централізованого представлення творчості, розширення аудиторії, покращення взаємодії з фанатами та ведення комерційної діяльності через власний онлайн-магазин.

Подальший розвиток проєкту може включати розширення функціоналу електронної комерції (наприклад, інтеграція з різними платіжними системами, розширена аналітика продажів), впровадження інтерактивних елементів для спілкування фанатів (форум, коментарі під новинами), розробку мобільної версії застосунку, а також інтеграцію з потоковими музичними сервісами та соціальними мережами.

Ключові слова: ВЕБ-ЗАСТОСУНОК, МУЗИЧНИЙ ГУРТ, ОНЛАЙН-ПЛАТФОРМА, REACT.JS, SPRING BOOT, ЕЛЕКТРОННА КОМЕРЦІЯ, REST API, JWT, MYSQL, CLOUDINARY.

ЗМІСТ

ВСТУП

У цифрову епоху веб-технології стали невід'ємною частиною музичної індустрії, трансформуючи способи взаємодії артистів зі своєю аудиторією. Ефективна онлайн-присутність є критично важливою для музичних гуртів, оскільки дозволяє їм не лише просувати свою творчість, але й будувати міцніші відносини з фанатами, розширювати аудиторію та генерувати прибуток.

Аналіз сучасного стану проблеми показує, що хоча існує багато інструментів для онлайн-представлення музикантів, таких як соціальні мережі та сторонні платформи для продажу мерчу, ці рішення часто є розрізненими. Це ускладнює користувачам отримання повної та актуальної інформації та негативно впливає на їхній досвід, а також обмежує контроль гуртів над власним контентом та взаємодією з аудиторією. Провідні музичні колективи часто створюють власні веб-платформи (наприклад, вебсайти Metallica) для консолідації своєї присутності. Однак, розробка таких комплексних рішень потребує значних зусиль та використання сучасних технологічних підходів.

Актуальність даної кваліфікаційної роботи зумовлена необхідністю створення централізованих онлайн-платформ для музичних гуртів. Це пов'язано зі зростанням цифрового споживання музики, де потокові сервіси та онлайн-магазини домінують, вимагаючи від гуртів наявності власних платформ для контролю над творчістю та максимізації прибутку. Також критично важливою є можливість прямої взаємодії з фанатами, оскільки власний вебсайт дозволяє створити унікальний досвід, надавати ексклюзивний контент та здійснювати прямий продаж мерчу, що сприяє формуванню лояльної спільноти, на відміну від обмежених можливостей соціальних мереж. Існуючі розрізнені рішення часто не задовольняють потребу в єдиній точці доступу до всієї інформації про гурт.

Метою цієї роботи є проектування, розробка та тестування повнофункціонального вебзастосунку Threllia з використанням сучасних веб-технологій (React.js для Frontend та Spring Boot + Spring Security для

Backend) для створення централізованої онлайн-платформи для надання фанатам повного доступу до інформації про гурт, його творчість та можливості придбання офіційного мерчу. Для досягнення поставленої мети необхідно вирішити наступні завдання:

Провести аналіз функціональних та нефункціональних вимог до вебзастосунку Threllia та спроєктувати його багаторівневу архітектуру з використанням React.js для клієнтської частини та Spring Boot для серверної, визначивши REST API для їх взаємодії.

Розробити ключові функціональні модулі вебзастосунку (включаючи "Музика", "Пісні", "Новини", "Тури", "Магазин", "Медіа-галерея", "Панель адміністратора"), реалізувати користувацький інтерфейс з використанням Tailwind CSS та ShadCN UI, систему керування станом Redux, інтеграцію з базою даних MySQL (AWS RDS) та систему аутентифікації й авторизації на базі Spring Security та JWT.

Провести комплексне тестування (модульне, інтеграційне, системне) розробленого вебзастосунку для оцінки його працездатності, безпеки та відповідності вимогам.

Об’єктом дослідження є процес розробки вебзастосунку для музичного гурту з акцентом на забезпеченні ефективної взаємодії з фанатами, продажу продукції та представленні творчості.

Предметом дослідження є функціональні можливості, архітектура, технології, процес розробки та результати тестування вебзастосунку Threllia.

Для вирішення поставлених завдань у роботі застосовувалися методи аналізу науково-технічної літератури та існуючих рішень, принципи об'єктно-орієнтованого проектування та розробки багаторівневої архітектури, використання сучасних фреймворків та бібліотек (React.js, Spring Boot, Spring Security, Redux, Axios, Tailwind CSS), методи розробки REST API, а також методи тестування програмного забезпечення, зокрема функціональне тестування API за допомогою Postman, інтеграційне тестування взаємодії компонентів та тестування користувацького інтерфейсу.

Практичне значення одержаних результатів полягає у створенні готового до впровадження вебзастосунку Threllia, який може слугувати сучасним інструментом для музичних гуртів у їхній взаємодії з аудиторією, просуванні власної творчості та веденні комерційної діяльності, сприяючи зміцненню їхніх позицій у цифровій музичній індустрії.

Галузь застосування розробленого вебзастосунку охоплює забезпечення ефективної онлайн-присутності музичних гуртів, централізоване представлення їхньої творчості та новин, розширення аудиторії, покращення взаємодії з фанатами та організацію електронної комерції через власний онлайн-магазин.

РОЗДІЛ 1 ПРОЄКТУВАННЯ THRELLIA WEB APP

1.1 Визначення функціоналу вебзастосунку

Вебзастосунок **Threllia** розробляється з метою створення централізованої цифрової платформи для взаємодії між музичним гуртом та його фанатами. Функціональні можливості застосунку орієнтовані як на користувачів, так і на адміністраторів.

Основні функціональні модулі застосунку:

- **Музика** — розділ, у якому представлені всі офіційні релізи гурту: альбоми, сингли, EP. Кожен реліз має обкладинку, треклист, дату виходу.
- **Пісні** — каталог усіх пісень гурту в алфавітному порядку з можливістю перегляду, у яких релізах вона зустрічається, та де виконувалася наживо.
- **Новини** — стрічка актуальних оновлень гурту.
- **Тури** — інтерактивний календар з майбутніми виступами, можливістю купівлі квитків та переглядом архіву минулих концертів.
- **Магазин (Shop)** — онлайн-магазин з офіційним мерчем, функціональністю кошика та оформлення замовлення.
- **Медіа-галерея** — колекція фотографій, відео, закулісних матеріалів.
- **Панель адміністратора** — закритий розділ для керування контентом сайту (CRUD-операції для кожного розділу: додавання, редагування, видалення новин, турів, пісень тощо).

Кожен з функціональних модулів має чітку взаємодію з базою даних та враховує рольову модель доступу користувачів (адміністратор / гість / авторизований користувач).

1.2 Архітектура застосунку

Застосунок Threllia розроблено з використанням багаторівневої архітектури, що забезпечує чітке розмежування між клієнтською та серверною частинами. Такий підхід сприяє кращій організації коду, полегшує його тестування та обслуговування, а також забезпечує масштабованість системи. Взаємодія між Frontend та Backend здійснюється за допомогою REST API, що є стандартним підходом для розробки сучасних вебзастосунків. [1]

1.2.1 Архітектура Backend

Серверна частина вебзастосунку Threllia побудована на основі фреймворку Spring Boot 3.4.1 та мови програмування Java 17. Spring Boot є потужним та популярним фреймворком для розробки Java-застосунків, що дозволяє швидко створювати виробничі системи з мінімальною конфігурацією.

Backend:

Основа проекту:

- **Spring Boot 3.4.1** - базовий фреймворк
- **Java 17** - версія мови програмування

Ключові компоненти:

1. **Spring Web** - для створення REST API та веб-застосунків
2. **Spring Security** - для автентифікації та авторизації
3. **Spring Data JPA** - для роботи з базою даних через JPA
4. **Spring Data JDBC** - для роботи з базою даних через JDBC
5. **MySQL** - база даних
6. **Spring Mail** - для надсилання електронних листів

Інтеграції з зовнішніми сервісами:

- **Cloudinary** - для керування мультимедійними файлами в хмарі
- **Stripe** - для обробки платежів

Утиліти та інструменти:

- **JWT (jsonwebtoken)** - для реалізації токенів автентифікації
- **Jackson** - для роботи з JSON, включаючи підтримку Java 8 Date/Time API
- **Lombok** - для зменшення шаблонного коду через анотації
- **Spring Boot DevTools** - для прискорення розробки

Для організації коду в Backend використовується багатошарова архітектура (Layered Architecture), яка є стандартною практикою у Spring Boot розробці. Ця архітектура розділяє застосунок на логічні шари, кожен з яких виконує певну функцію та має чітко визначені обов'язки. [2]

Основні шари Backend:

1. Контролери (Controllers): Цей шар відповідає за обробку HTTP-запитів від клієнтів. Контролери отримують запити, валідують вхідні дані, делегують бізнес-логіку сервісам та повертають HTTP-відповіді. Для визначення контролерів використовуються анотації @RestController, @Controller та @RequestMapping.
2. Сервіси (Services): Шар сервісів містить бізнес-логіку застосунку. Сервіси координують роботу репозиторіїв та інших компонентів, трансформують дані між різними шарами та управляють транзакціями. Анотації @Service та @Transactional використовуються для визначення сервісів.
3. Репозиторії (Repositories): Шар репозиторіїв забезпечує доступ до бази даних. Репозиторії виконують операції CRUD, інкапсулюють складні запити до бази даних та можуть використовувати Spring Data JPA, JDBC або інші технології доступу до даних. Для визначення репозиторіїв використовуються анотації @Repository та інтерфейси JpaRepository.
4. Сутності/Моделі (Entities/Models): Сутності представляють доменні об'єкти та відображають структуру таблиць бази даних. Вони містять бізнес-правила, специфічні для сутностей, та анотуються за допомогою @Entity, @Table та @Column.
5. DTO (Data Transfer Objects): Об'єкти передачі даних використовуються для передачі даних між шарами та клієнтами. DTO допомагають

приховати внутрішню структуру даних та дозволяють вибірково передавати дані, що особливо корисно в API.

6. Конфігурація (Configuration): Шар конфігурації відповідає за налаштування Spring контейнера, конфігурацію безпеки, баз даних та інших компонентів застосунку. Анотації @Configuration та @Bean використовуються для визначення класів конфігурації.
7. Винятки (Exceptions): Цей шар містить спеціальні класи винятків та забезпечує глобальну обробку виняткових ситуацій у застосунку. Анотації @ControllerAdvice та @ExceptionHandler використовуються для обробки винятків.
8. Утиліти (Utilities): Шар утиліт містить допоміжні класи для різних завдань, розширення функціональності, статичні методи та константи.

Переваги багатошарової архітектури:

- Розділення відповідальності: Кожен шар має чітко визначені обов'язки, що спрощує розробку, тестування та обслуговування коду.
- Простіше тестування: Кожен шар можна тестувати окремо, що полегшує виявлення та виправлення помилок.
- Зрозуміла структура: Багатошарова архітектура забезпечує чітку та зрозумілу структуру проєкту, що полегшує навігацію кодом.
- Масштабованість і підтримуваність: Ця архітектура полегшує масштабування та підтримку застосунку, оскільки зміни в одному шарі не обов'язково впливають на інші шари.
- Можливість паралельної розробки: Різні команди можуть працювати над різними шарами одночасно, що прискорює процес розробки.

Багатошарова архітектура добре поєднується з принципами інверсії залежностей та впровадження залежностей, які є ключовими в Spring Boot та сприяють створенню слабко зв'язаних та гнучких компонентів. [3]

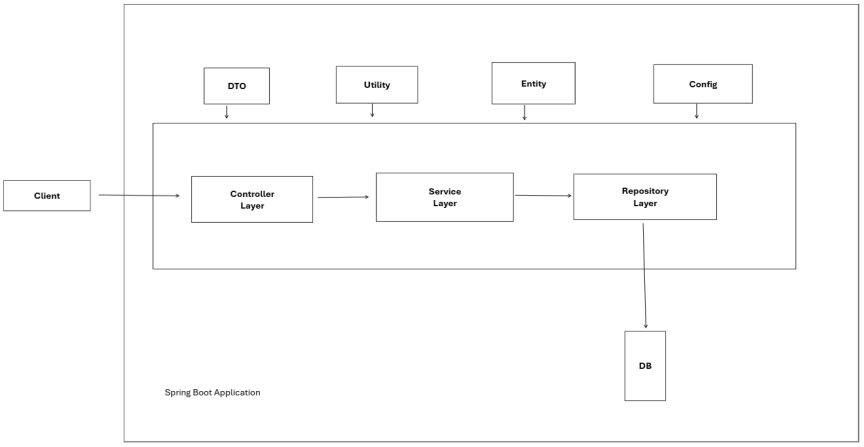


Рисунок 1.1 – Шарова архітектура

Рисунок 1.2 – Огляд шару контролер

Рисунок 1.3 – Огляд шару репозиторіїв

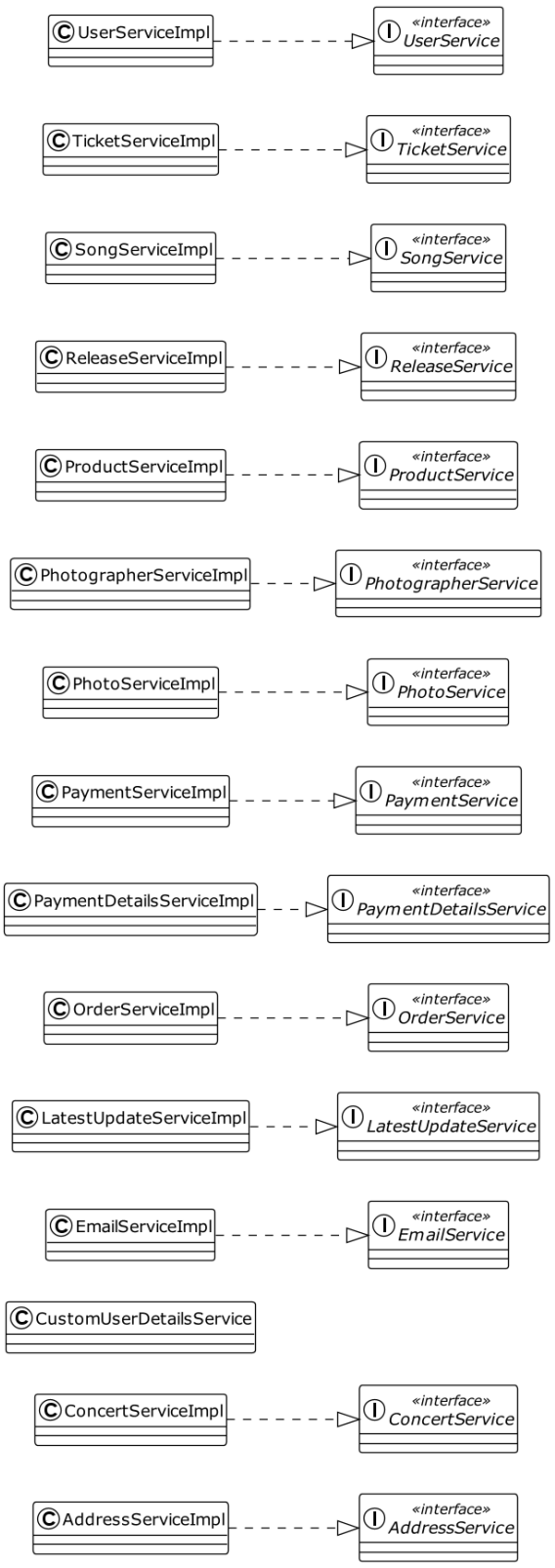


Рисунок 1.4 – Огляд шару сервісів

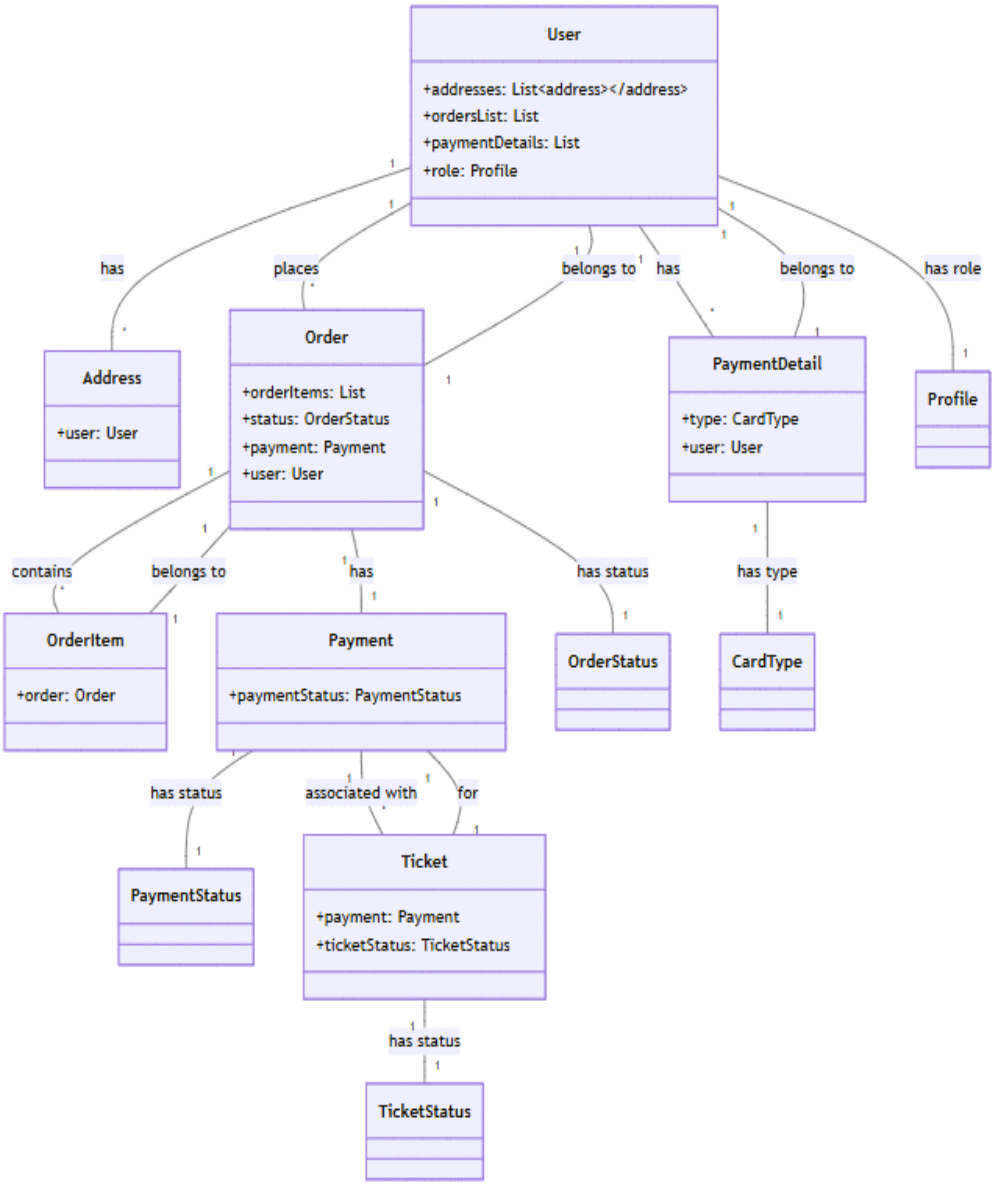


Рисунок 1.5 – Огляд шару моделі

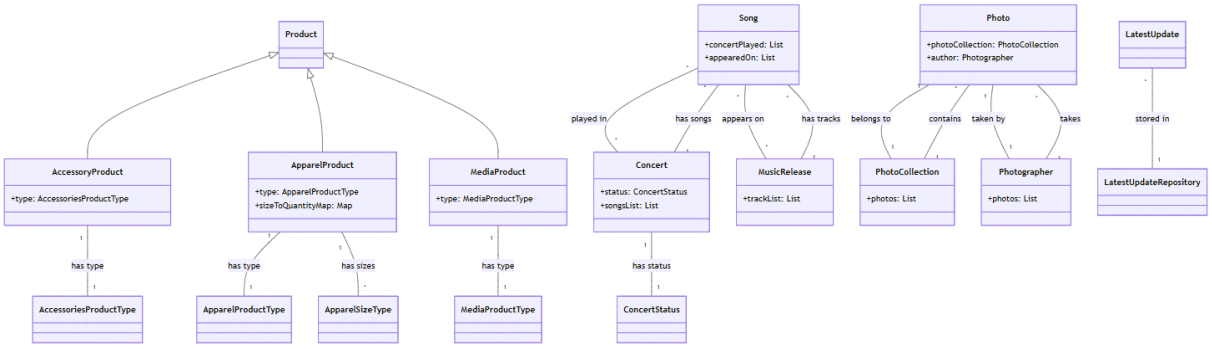


Рисунок 1.6 – Огляд шару моделі

Основні сутності:

- User: Представляє користувача системи та містить інформацію, таку як id, username, email, password, addresses, ordersList, paymentDetails, role.
- Order: Представляє замовлення, зроблене користувачем, та містить інформацію про orderItems, status, payment, user.
- OrderItem: Представляє окремий товар у замовленні та містить інформацію про order.
- Product: Представляє товар, доступний для придбання.
- Payment: Представляє платіж за замовлення та містить інформацію про paymentStatus.
- PaymentDetail: Представляє деталі платіжної інформації користувача та містить інформацію про type, user.

- Ticket: Представляє квиток на концерт та містить інформацію про payment, ticketStatus.
- Address: Представляє адресу користувача та містить інформацію про user.
- Profile: Представляє роль користувача в системі.
- OrderStatus: Представляє статус замовлення.
- PaymentStatus: Представляє статус платежу.
- CardType: Представляє тип платіжної картки.
- TicketStatus: Представляє статус квитка.

Основні зв'язки:

- User has Address (один користувач може мати багато адрес).
- User has Order (один користувач може мати багато замовлень).
- User has PaymentDetail (один користувач може мати багато платіжних деталей).
- User belongs to Profile (один користувач належить до однієї ролі).
- Order contains OrderItem (одне замовлення містить багато товарів).
- Order belongs to User (одне замовлення належить одному користувачу).
- Order has Payment (одне замовлення має один платіж).
- Order has OrderStatus (одне замовлення має один статус).
- Payment is associated with Ticket (один платіж може бути пов'язаний з одним квитком).
- Payment has PaymentStatus (один платіж має один статус).
- PaymentDetail belongs to User (одна платіжна деталь належить одному користувачу).
- PaymentDetail has CardType (одна платіжна деталь має один тип картки).
- Ticket has TicketStatus (один квиток має один статус).

Ця модель даних спроєктована для ефективного зберігання та управління інформацією про користувачів, замовлення, товари, платежі, квитки та інші сутності, необхідні для функціонування вебзастосунку Threllia.

1.2.2 Архітектура Frontend

Клієнтська частина вебзастосунку Threllia розроблена з використанням бібліотеки React.js. React.js є потужною та популярною JavaScript-бібліотекою для створення користувацьких інтерфейсів, що забезпечує високу продуктивність, модульність та зручність розробки.

Основні технології та інструменти, використані у Frontend розробці:

Frontend:

- Розробка ведеться за допомогою бібліотеки **React.js** та локальним сервером Vite 6.
- **React Router:** Для реалізації навігації між сторінками застосунку використовується React Router. React Router дозволяє керувати URL-адресами та відображати відповідні компоненти на основі поточного маршруту. [4]
- **Redux Toolkit:** Керування станом застосунку (даними, які використовуються компонентами) здійснюється за допомогою Redux Toolkit. Redux Toolkit є ефективною бібліотекою для управління складними станами у React-застосунках, що забезпечує передбачуваність та полегшує налагодження. [5]
- **Axios:** Для виконання HTTP-запитів до Backend використовується бібліотека Axios. Axios надає зручний інтерфейс для асинхронної взаємодії з сервером. [6]
- **Tailwind CSS:** Стилізація компонентів застосунку реалізована за допомогою Tailwind CSS. Tailwind CSS є CSS-фреймворком, що базується на утилітах, та дозволяє швидко створювати кастомні дизайни. [7]
- **ShadCN UI:** Для створення повторно використовуваних UI-компонентів використовується бібліотека ShadCN UI. ShadCN UI надає набір готових компонентів, стилізованих за допомогою Tailwind CSS. [8]
- **Шрифти:** У застосунку використовуються кастомні шрифти: Delicious Handrawn, Rubik Wet Paint, Trade Winds. Вибір цих шрифтів обумовлений

їхньою естетичною привабливістю та відповідністю загальному стилю вебсайту.

- **Іконки:** Для відображення іконок використовується бібліотека Lucide React. Lucide React надає великий набір векторних іконок, які легко інтегруються в React-застосунки.

Вибір React.js для Frontend розробки обумовлений його популярністю, продуктивністю, великою спільнотою розробників та наявністю численних бібліотек та інструментів, що спрощують розробку складних вебзастосунків. Використання Tailwind CSS та ShadCN UI дозволяє прискорити процес розробки UI та забезпечити консистентність дизайну.

Загалом, архітектура вебзастосунку Threllia спроєктована таким чином, щоб забезпечити масштабованість, зручність в обслуговуванні, високу продуктивність та позитивний досвід користувача.

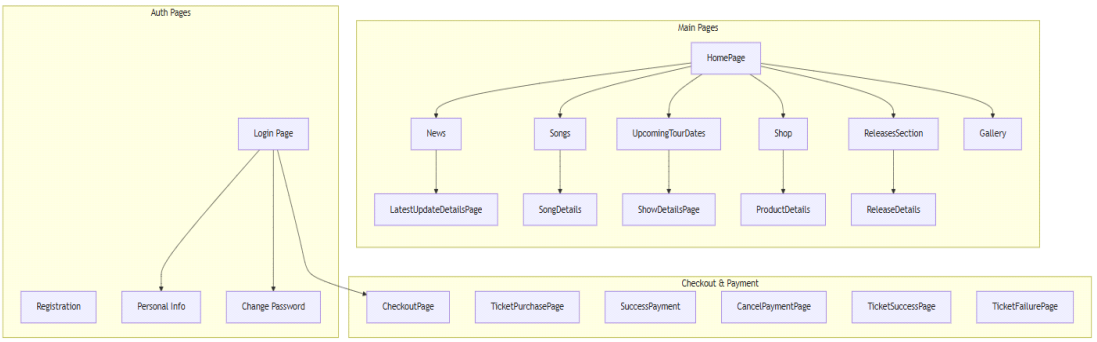


Рисунок 1.7 – Високорівневий огляд сторінок

Схема розділена на три основні частини, що відображають різні функціональні області сайту:

- Auth Pages (Сторінки аутентифікації): Ця частина схеми описує сторінки, пов'язані з процесом аутентифікації користувачів.
    - LoginPage: Сторінка входу користувача в систему. З неї доступні переходи на:
        - Registration: Сторінка реєстрації нового користувача.
        - Personal Info: Сторінка з інформацією профілю користувача.
        - Change Password: Сторінка зміни пароля користувача.
- Main Pages (Основні сторінки): Ця частина схеми описує основні сторінки сайту, доступні після аутентифікації.
    - HomePage: Головна сторінка сайту. З неї доступні переходи на:
        - News: Сторінка з останніми новинами гурту. З неї доступний перехід на:
            - LatestUpdateDetailsPage: Сторінка з детальною інформацією про конкретну новину.
        - Songs: Сторінка з переліком пісень гурту. З неї доступний перехід на:
            - SongDetailsPage: Сторінка з детальною інформацією про конкретну пісню.
        - UpcomingTourDates: Сторінка з інформацією про майбутні тури гурту. З неї доступний перехід на:
            - ShowDetailsPage: Сторінка з детальною інформацією про конкретний виступ.
        - Shop: Сторінка інтернет-магазину гурту. З неї доступний перехід на:
            - ProductDetails: Сторінка з детальною інформацією про конкретний товар.

        - ReleaseSection: Сторінка з переліком музичних релізів гурту. З неї доступний перехід на:
            - ReleaseDetails: Сторінка з детальною інформацією про конкретний реліз.
        - Gallery: Сторінка з фото- та відеоматеріалами гурту.
- Checkout & Payment (Оформлення та оплата): Ця частина схеми описує сторінки, пов'язані з процесом оформлення замовлення та оплати.
    - CheckoutPage: Сторінка оформлення замовлення.
    - TicketPurchasePage: Сторінка купівлі квитків.
    - SuccessPayment: Сторінка успішної оплати.
    - CancelPaymentPage: Сторінка скасування оплати.
    - TicketSuccessPage: Сторінка успішної купівлі квитка.
    - TicketFailurePage: Сторінка невдалої купівлі квитка.

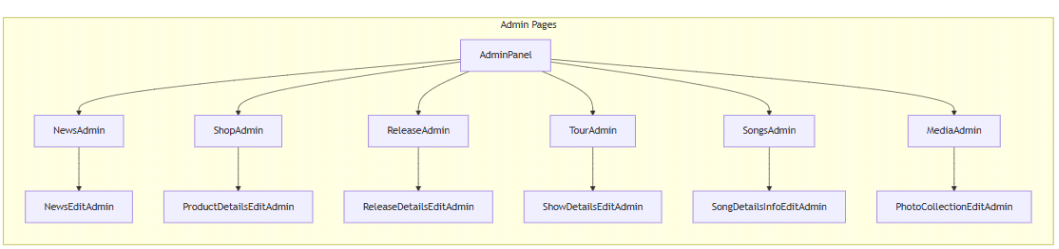


Рисунок 1.8 – Admin-сторінки

Admin Pages надає інтерфейс для адміністраторів вебзастосунку Threllia для керування контентом та даними. Основні сторінки Admin Pages:

- AdminPanel: Головна сторінка Admin Pages, що надає навігацію до інших сторінок.
- NewsAdmin: Сторінка для керування новинами (перегляд, додавання, редагування, видалення).
- ShopAdmin: Сторінка для керування товарами (перегляд, додавання, редагування, видалення).
- ReleaseAdmin: Сторінка для керування релізами музики (альбоми, сингли, EP) (перегляд, додавання, редагування, видалення).
- TourAdmin: Сторінка для керування інформацією про тури та концерти (перегляд, додавання, редагування, видалення).
- SongsAdmin: Сторінка для керування інформацією про пісні (перегляд, додавання, редагування, видалення).
- MediaAdmin: Сторінка для керування медіа-контентом (фото, відео) (перегляд, додавання, редагування, видалення).
- NewsEditAdmin: Сторінка для редагування конкретної новини.
- ProductDetailsEditAdmin: Сторінка для редагування деталей конкретного товару.
- ReleaseDetailsEditAdmin: Сторінка для редагування деталей конкретного релізу.
- ShowDetailsEditAdmin: Сторінка для редагування деталей конкретного концерту.
- SongDetailsInfoEditAdmin: Сторінка для редагування деталей конкретної пісні.
- PhotoCollectionEditAdmin: Сторінка для редагування деталей конкретної колекції фотографій.

Admin Pages забезпечує зручний та інтуїтивно зрозумілий інтерфейс для адміністраторів, що дозволяє їм ефективно керувати контентом вебзастосунку Threllia.

РОЗДІЛ 2 БЕЗПЕКА ДОДАТКУ

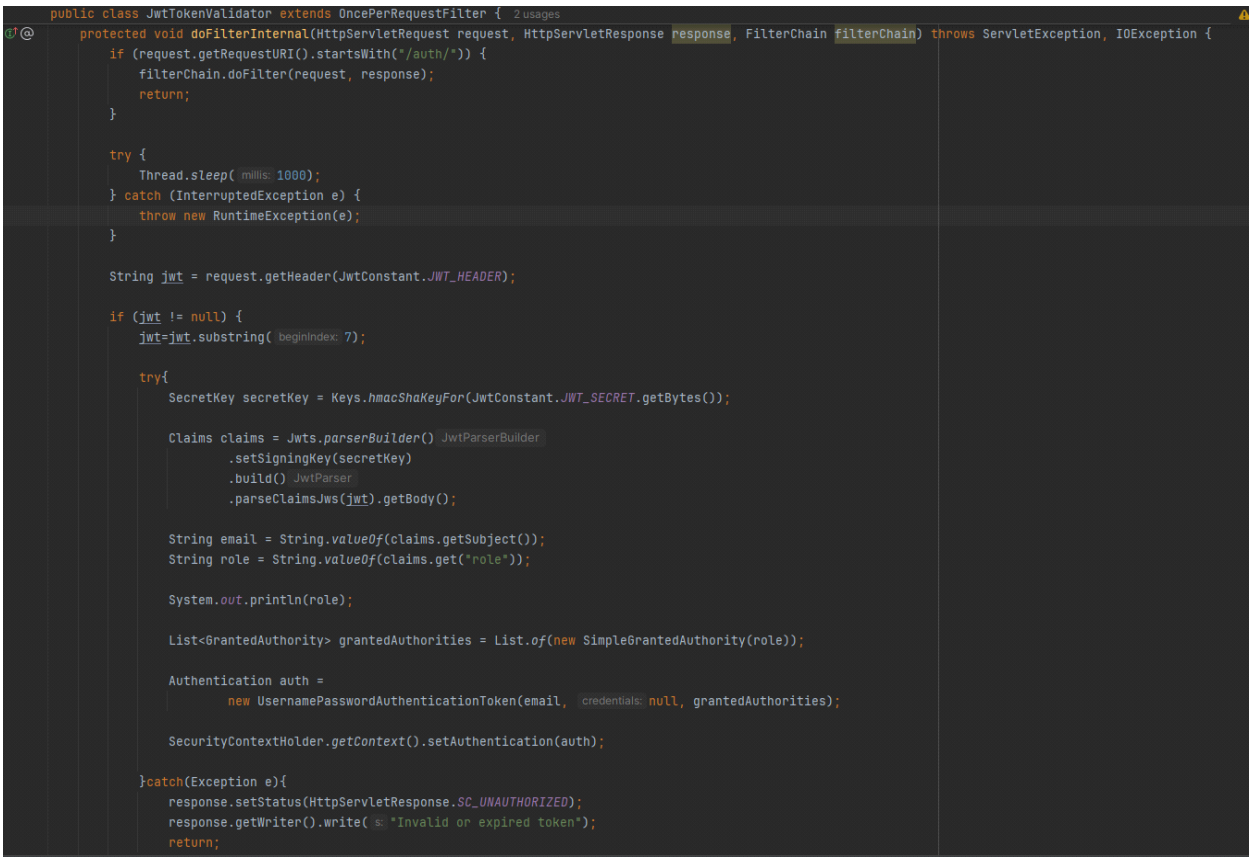
2.1 Реалізація системи автентифікації та авторизації

Для забезпечення безпеки вебзастосунку Threllia та контролю доступу до його ресурсів була реалізована система автентифікації та авторизації з використанням фреймворку Spring Security у поєднанні з технологією JSON Web Tokens (JWT). Цей підхід дозволяє створити гнучку та безпечну систему для stateless архітектури, де стан користувача не зберігається на сервері між запитами.

Основні налаштування безпеки визначені у конфігураційному класі Configuration. [9] Ключові аспекти конфігурації включають:

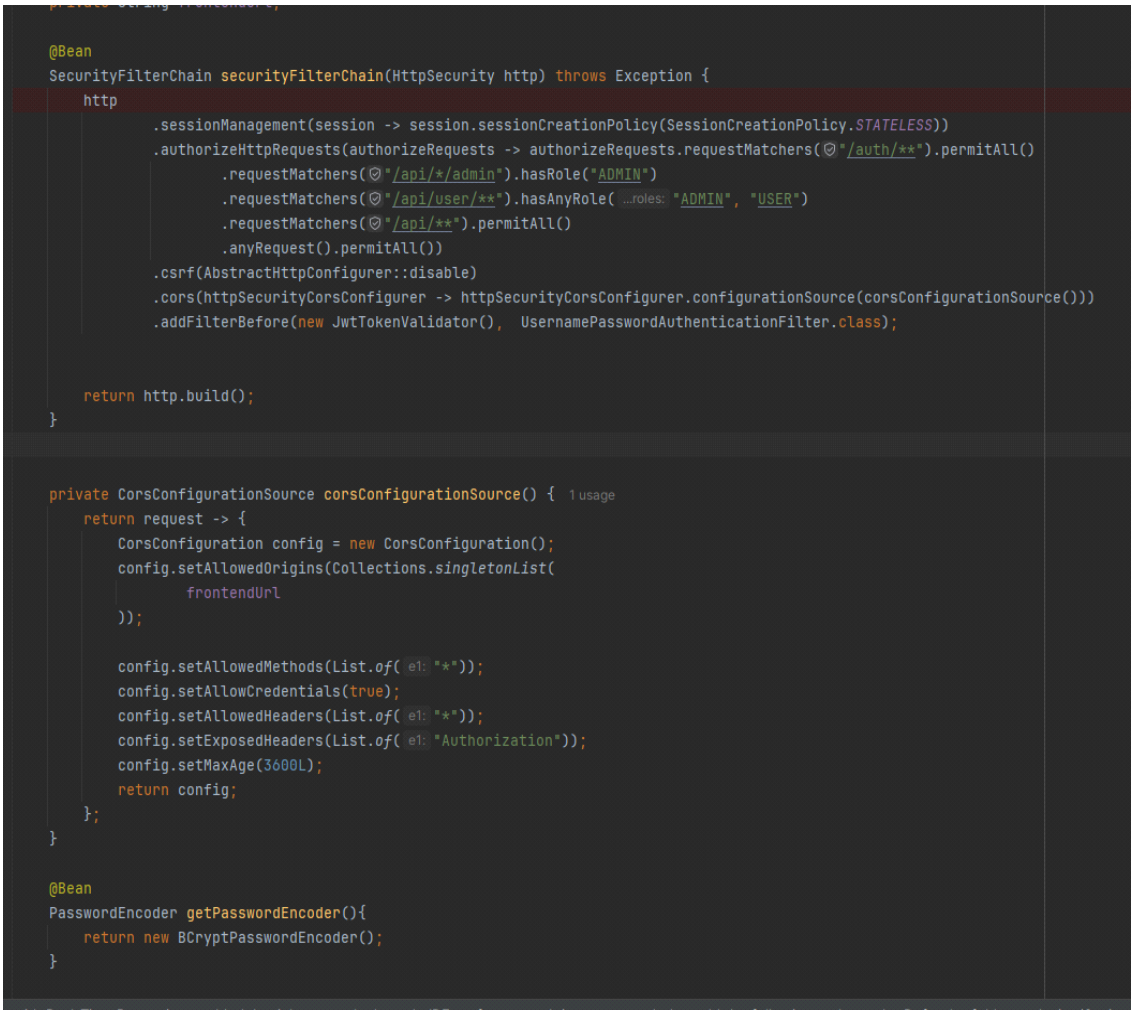
1. **Керування сесіями:** Застосунок налаштований на використання політики STATELESS для сесій (sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)). Це означає, що сервер не зберігає інформацію про сесію користувача. Автентифікація відбувається для кожного запиту окремо на основі наданого JWT.
2. **Правила авторизації запитів:** Визначено правила доступу до різних URL-шляхів застосунку (authorizeHttpRequests):
    - Шляхи, що починаються з /auth/** (призначені для процесів входу та реєстрації), є загальнодоступними (permitAll()).
    - Доступ до шляхів, що відповідають шаблону /api/*/admin, вимагає наявності у користувача ролі "ADMIN" (hasRole("ADMIN")).
    - Доступ до шляхів /api/user/** дозволено користувачам з ролями "ADMIN" або "USER" (hasAnyRole("ADMIN", "USER")).
    - Інші шляхи, що відповідають шаблону /api/**, а також будь-які інші запити (anyRequest()) є загальнодоступними (permitAll()).

3. **Захист CSRF:** Захист від атак типу Cross-Site Request Forgery вимкнено (csrf(AbstractHttpConfigurer::disable)), що є поширеною практикою для stateless API, які використовують токени для автентифікації.
4. **Конфігурація CORS:** Налаштовано політику Cross-Origin Resource Sharing (cors()) для взаємодії з клієнтською частиною (Frontend), розміщеною за адресою frontendUrl. Дозволено використання всіх HTTP-методів (List.of("*")), передачу облікових даних (credentials) (setAllowCredentials(true)) та використання всіх заголовків (List.of("*")). Важливо, що заголовок Authorization явно дозволено (setExposedHeaders) для того, щоб клієнт міг його отримати та використовувати.
5. **Додавання JWT фільтра:** Спеціальний фільтр JwtTokenValidator інтегровано в ланцюжок фільтрів Spring Security. Він додається *перед* стандартним фільтром UsernamePasswordAuthenticationFilter (addFilterBefore), що забезпечує перевірку JWT на ранньому етапі обробки запиту.
6. **Кодувальник паролів:** Для безпечного зберігання та перевірки паролів користувачів визначено бін PasswordEncoder з реалізацією BCryptPasswordEncoder.



```
public class JwtTokenValidator extends OncePerRequestFilter {  2 usages
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
        if (request.getRequestURI().startsWith("/auth/")) {
            filterChain.doFilter(request, response);
            return;
        }

        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        String jwt = request.getHeader(JwtConstant.JWT_HEADER);

        if (jwt != null) {
            jwt=jwt.substring( beginIndex: 7);

            try{
                SecretKey secretKey = Keys.hmacShaKeyFor(JwtConstant.JWT_SECRET.getBytes());

                Claims claims = Jwts.parserBuilder() JwtParserBuilder
                        .setSigningKey(secretKey)
                        .build() JwtParser
                        .parseClaimsJws(jwt).getBody();

                String email = String.valueOf(claims.getSubject());
                String role = String.valueOf(claims.get("role"));

                System.out.println(role);

                List<GrantedAuthority> grantedAuthorities = List.of(new SimpleGrantedAuthority(role));

                Authentication auth =
                        new UsernamePasswordAuthenticationToken(email,  credentials: null, grantedAuthorities);

                SecurityContextHolder.getContext().setAuthentication(auth);

            }catch(Exception e){
                response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
                response.getWriter().write( s: "Invalid or expired token");
                return;
```

Рисунок 2.1 - Налаштування JwtTokenValidatorFilter



```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(authorizeRequests -> authorizeRequests.requestMatchers("/auth/**").permitAll()
                    .requestMatchers("/api/*/admin").hasRole("ADMIN")
                    .requestMatchers("/api/user/**").hasAnyRole( ...roles: "ADMIN", "USER")
                    .requestMatchers("/api/**").permitAll()
                    .anyRequest().permitAll())
            .csrf(AbstractHttpConfigurer::disable)
            .cors(httpSecurityCorsConfigurer -> httpSecurityCorsConfigurer.configurationSource(corsConfigurationSource()))
            .addFilterBefore(new JwtTokenValidator(),  UsernamePasswordAuthenticationFilter.class);


    return http.build();
}


private CorsConfigurationSource corsConfigurationSource() {  1 usage
    return request -> {
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowedOrigins(Collections.singletonList(
                frontendUrl
        ));

        config.setAllowedMethods(List.of( e1: "*"));
        config.setAllowCredentials(true);
        config.setAllowedHeaders(List.of( e1: "*"));
        config.setExposedHeaders(List.of( e1: "Authorization"));
        config.setMaxAge(3600L);
        return config;
    };
}

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

Рисунок 2.2 - Налаштування SecurityFilterChain

2.2 Валідація JWT за допомогою фільтра JwtTokenValidator

Клас JwtTokenValidator розширює OncePerRequestFilter, гарантуючи виконання логіки перевірки токена один раз для кожного запиту. Процес валідації відбувається наступним чином [10]:

1. **Перехоплення запитів:** Фільтр перехоплює всі вхідні HTTP-запити до захищених ресурсів.
2. **Виключення шляхів автентифікації:** Запити до кінцевих точок, що починаються з /auth/, ігноруються цим фільтром, оскільки вони відповідають за початкову автентифікацію (наприклад, логін за іменем користувача та паролем) та генерацію JWT.

3. **Вилучення токена:** Фільтр намагається отримати рядок токена з HTTP-заголовка Authorization (request.getHeader(JwtConstant.JWT_HEADER)). Очікується, що токен буде передано у форматі "Bearer <токен>".
4. **Перевірка та обробка токена:**
    - Якщо заголовок Authorization існує та починається з префікса "Bearer ", префікс видаляється (jwt.substring(7)), залишаючи чистий JWT.
    - Використовуючи секретний ключ (JwtConstant.JWT_SECRET), що зберігається в константах, та можливості бібліотеки JJWT (Jwts.parserBuilder()), токен розбирається (парситься) та валідується його підпис і термін дії.
    - У разі успішної валідації, з корисного навантаження токена (claims) витягуються дані користувача: електронна адреса (email, що використовується як ім'я користувача/subject) та роль (role).
    - На основі отриманої ролі створюється список повноважень (List<GrantedAuthority>) для системи безпеки Spring.
    - Створюється об'єкт Authentication типу UsernamePasswordAuthenticationToken. В якості principal встановлюється email користувача, credentials встановлюються в null, а також передається список наданих повноважень.
    - Створений об'єкт Authentication зберігається у SecurityContextHolder. Це сигналізує Spring Security, що користувач успішно автентифікований для поточного запиту.
5. **Обробка помилок валідації:** Якщо токен відсутній у запиті, має невірний формат, невірний підпис, або його термін дії закінчився, бібліотека JJWT генерує виняток. Фільтр перехоплює ці винятки. У відповідь клієнту надсилається HTTP-статус 401 Unauthorized разом із повідомленням про помилку ("Invalid or expired token"). Подальша обробка запиту припиняється.

6. **Передача запиту далі:** Якщо токен успішно валідований і контекст безпеки встановлено, або якщо токен не був наданий (для публічних кінцевих точок), запит передається далі по ланцюжку фільтрів Spring Security (filterChain.doFilter(request, response)). На наступних етапах Spring Security застосовує правила авторизації, визначені в класі Configuration, перевіряючи, чи має автентифікований користувач необхідні повноваження для доступу до запитуваного ресурсу.

Загальна схема роботи:

Користувач проходить автентифікацію (ймовірно, через кінцеву точку /auth/**), надаючи свої облікові дані. У разі успіху сервер генерує JWT та повертає його клієнту. Клієнтська частина зберігає цей токен і додає його до заголовка Authorization кожного наступного запиту до захищених ресурсів Backend. Фільтр JwtTokenValidator на стороні Backend перехоплює ці запити, валідує токен і, якщо він дійсний, встановлює дані про автентифікованого користувача та його ролі в SecurityContextHolder. Це дозволяє Spring Security коректно застосовувати правила авторизації для доступу до різних частин API.

ОК, давайте доопрацюємо цей текст, щоб він мав більш формальний та академічний стиль, придатний для дипломної роботи, зберігаючи при цьому всі ключові технічні деталі. [11]

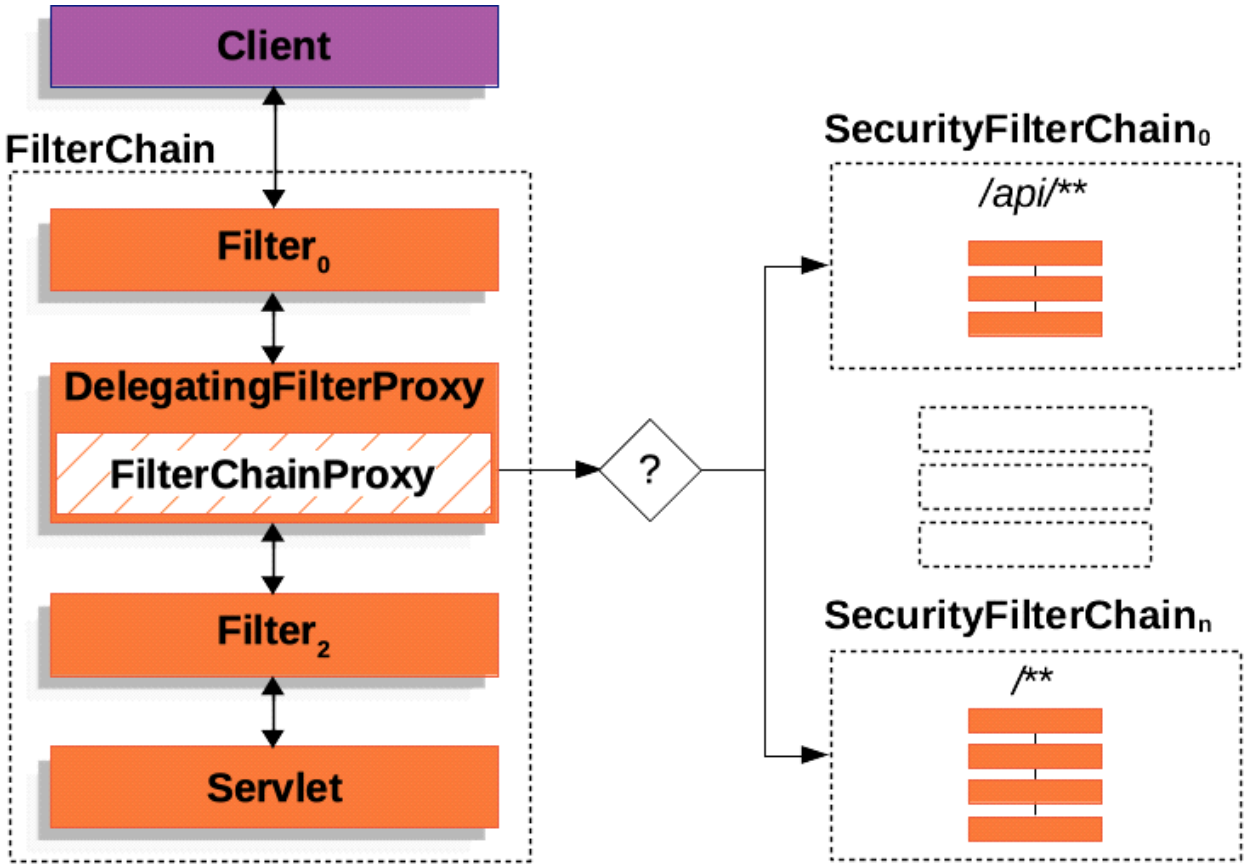


Рисунок 2.3 - Spring Security Filters Architecture

2.3 Реалізація системи автентифікації та авторизації на клієнтській стороні

Система автентифікації та авторизації на клієнтській стороні вебзастосунку Threllia спроєктована для забезпечення безпечної взаємодії з серверною частиною з використанням механізму JSON Web Tokens та підходу stateless автентифікації. Управління станом автентифікації та взаємодія з API реалізовані за допомогою бібліотеки керування станом Redux Toolkit та HTTP-клієнта Axios.

2.3.1 Процес автентифікації та керування токеном

Після успішного надсилання облікових даних користувача на відповідну кінцеву точку Backend (наприклад, /auth/login), сервер у відповідь надсилає JWT. Цей токен є підтвердженням успішної автентифікації та містить необхідну інформацію про користувача та його права доступу (роль).

Для забезпечення персистентності стану автентифікації між сеансами користувача та після оновлення сторінки, отриманий JWT зберігається у локальному сховищі браузера (localStorage):

Одночасно зі збереженням токена в localStorage, відповідний стан автентифікації (наприклад, інформація про користувача, ознака isAuthenticated) оновлюється у глобальному сховищі Redux за допомогою відповідного редьюсера (наприклад, authReducer).

2.3.2 Використання токена для авторизованих запитів

Для взаємодії із захищеними кінцевими точками API серверної частини кожен запит повинен містити дійсний JWT. Щоб автоматизувати цей процес та уникнути дублювання коду, використовується спеціально налаштований екземпляр бібліотеки Axios. Цей екземпляр містить глобальний перехоплювач запитів (request interceptor), який автоматично додає заголовок Authorization до кожного вихідного запиту. Токен для заголовка отримується або безпосередньо з localStorage, або з відповідного стану Redux.

2.3.3 Обробка закінчення терміну дії токена та автоматичний вихід

Важливою частиною системи є обробка ситуацій, коли JWT стає недійсним (наприклад, через закінчення терміну дії). Для цього використовується перехоплювач відповідей (response interceptor) Axios. Цей перехоплювач аналізує відповіді від сервера. Якщо сервер повертає помилку, що свідчить про недійсність токена (наприклад, статус 401 Unauthorized або специфічне повідомлення, як Invalid or expired token, що повертається реалізованим JwtTokenValidator на Backend), перехоплювач ініціює процес виходу користувача. Відповідна Redux-дія (logoutUserAction) виконує наступні кроки:

- Видаляє JWT з localStorage.
- Скидає стан автентифікації у Redux-сховищі до початкового.
- За потреби, перенаправляє користувача на сторінку входу або головну публічну сторінку.

Такий підхід централізує логіку обробки помилок автентифікації, звільняючи окремі компоненти від необхідності самостійно перевіряти валідність токена при кожній взаємодії з API.

2.3.4 Захист маршрутів на клієнтській стороні

Вебзастосунок використовує механізми захисту маршрутів, реалізовані за допомогою бібліотеки React Router. Компоненти та сторінки, що вимагають автентифікації користувача , обгорнуті у спеціальні компоненти вищого порядку або використовують логіку умовного рендерингу. Доступ до таких маршрутів надається лише за наявності дійсного стану автентифікації у Redux-сховищі. В іншому випадку користувач автоматично перенаправляється на сторінку входу.

Також при початковому завантаженні застосунку виконується перевірка наявності збереженого токена в localStorage. Якщо токен знайдено, може бути ініційована спроба автоматичного відновлення сесії, що покращує досвід користувача, уникаючи необхідності повторного входу.

2.3.5 Аспекти безпеки: CSRF та CORS

Оскільки застосунок використовує stateless автентифікацію на основі JWT, традиційні механізми захисту від CSRF, що базуються на сесіях, не є необхідними і не використовуються на клієнтській стороні. Безпека забезпечується перевіркою JWT на сервері при кожному запиті. [12]

Політика CORS налаштована на серверній стороні, як описано у розділі X.Y.1, що дозволяє клієнтській частині безпечно взаємодіяти з API, зокрема передавати заголовок Authorization. [13]

2.3.6 Переваги реалізованого підходу

Обраний підхід до реалізації системи автентифікації та авторизації на клієнтській стороні має наступні переваги:

- **Stateless:** Відповідає архітектурі REST API, не вимагає зберігання сесійних даних на сервері, що полегшує масштабування.

- **Безпека:** Перевірка JWT при кожному запиті до захищених ресурсів забезпечує належний рівень контролю доступу.
- **Централізована обробка:** Використання перехоплювачів Axios дозволяє уніфікувати додавання токенів до запитів та обробку помилок автентифікації.
- **Гнучкість:** JWT дозволяє передавати додаткову інформацію (claims), таку як ролі користувача, що використовується для реалізації контролю доступу на основі ролей.
- **Покращений досвід користувача:** Можливість збереження стану автентифікації та автоматичного відновлення сесії.

Загалом, реалізована система забезпечує надійний та сучасний механізм автентифікації та авторизації для клієнтської частини вебзастосунку Threllia, створюючи основу для подальшого розвитку функціоналу безпеки. [14]

РОЗДІЛ 3. ПІДГОТОВКА ПРОЕКТУ ДО ДЕПЛОЮ

3.1 Тестування

Тестування є невід'ємним етапом розробки програмного забезпечення, метою якого є забезпечення якості, надійності та відповідності розробленого вебзастосунку Threllia визначеним функціональним та нефункціональним вимогам. У процесі розробки було застосовано комплексний підхід до тестування, що охоплював різні рівні та аспекти системи.

3.1.1 Тестування API (Backend)

Тестування серверної частини (Backend API), реалізованої на Spring Boot, проводилося здебільшого на рівні функціонального тестування кінцевих точок (endpoints). Основним інструментом для цього слугував Postman.

Методи та об'єкти тестування:

- **Перевірка коректності HTTP-запитів та відповідей:** Тестувалися різні HTTP-методи (GET, POST, PUT, DELETE) для всіх реалізованих кінцевих точок API. Перевірялася відповідність структури та типів даних у тілі запитів та відповідей (JSON), а також коректність HTTP-статусів.
- **Валідація вхідних даних:** Перевірялася обробка коректних та некоректних вхідних даних, включаючи граничні значення та обов'язкові поля.
- **Тестування логіки автентифікації та авторизації:** Особлива увага приділялася перевірці механізму JWT-автентифікації. Тестувалися сценарії з валідними, невалідними та простроченими токенами. Перевірявся контроль доступу на основі ролей ("ADMIN", "USER") до відповідних кінцевих точок API, як визначено у конфігурації Spring Security.
- **Обробка помилок:** Тестувалася коректність обробки виняткових ситуацій та повернення інформативних повідомлень про помилки клієнту.

Використання Postman дозволило ефективно перевіряти окремі компоненти API ізольовано від клієнтської частини, що сприяло ранньому виявленню помилок у бізнес-логіці та механізмах безпеки Backend.

3.1.2 Інтеграційне тестування

Метою інтеграційного тестування була перевірка коректності взаємодії між клієнтською та серверною частинами застосунку, а також перевірка наскрізних користувацьких сценаріїв.

Основні перевірені сценарії:

- Реєстрація нового користувача.
- Вхід користувача в систему (логін) та отримання JWT.
- Перегляд захищених даних після успішного входу.
- Виконання операцій, що потребують авторизації.

- Обробка ситуації з недійсним/простроченим токеном (автоматичний вихід користувача) .
- Коректність роботи CRUD-операцій через інтерфейс адміністратора.

Це тестування здебільшого виконувалося вручну шляхом взаємодії з користувацьким інтерфейсом та відстеженням відповідних запитів/відповідей у інструментах розробника браузера, а також аналізу стану даних після виконання операцій.

3.1.3 Тестування користувацького інтерфейсу (Frontend)

Тестування клієнтської частини, реалізованої на React.js, було спрямоване на перевірку візуальної коректності, зручності використання та функціональності інтерфейсу.

Аспекти тестування:

- **Візуальна відповідність:** Перевірка відповідності реалізованого інтерфейсу розробленим макетам та прототипам. Оцінювалася коректність відображення елементів, шрифтів , іконок , кольорової схеми та загального стилю з використанням Tailwind CSS та ShadCN UI .
- **Функціональність компонентів:** Перевірка працездатності інтерактивних елементів (кнопок, форм, посилань, меню навігації). Тестувалася коректність обробки подій користувача та оновлення стану за допомогою Redux.
- **Адаптивність та кросбраузерність:** Ручна перевірка коректності відображення та функціонування інтерфейсу на різних розмірах екранів за допомогою інструментів розробника браузера.
- **Зручність використання:** Оцінка інтуїтивності навігації, зрозумілості елементів керування та загального досвіду взаємодії користувача з застосунком .

- **Обробка станів:** Перевірка відображення індикаторів завантаження під час асинхронних операцій та коректного показу повідомлень про помилки, отриманих від Backend.

3.1.4 Використання логування для діагностики

Протягом усього процесу розробки та тестування активно використовувалися механізми логування. На клієнтській стороні аналізувалися повідомлення у консолі браузера (за допомогою console.log та інструментів розробника) для відстеження стану компонентів, даних Redux та діагностики помилок JavaScript. На серверній стороні використовувалися стандартні механізми логування Spring Boot для моніторингу запитів, винятків та стану виконання бізнес-логіки. Логи були важливим інструментом для швидкого виявлення та усунення проблем, особливо під час інтеграційного тестування.

Підсумок тестування: Застосований багаторівневий підхід до тестування дозволив виявити та виправити низку помилок на різних етапах розробки, що сприяло підвищенню стабільності, надійності та безпеки вебзастосунку Threllia та його відповідності поставленим вимогам.

3.2 Управління даними та зберігання

Ефективне управління даними є критично важливим для функціонування вебзастосунку Threllia, який оперує різноманітною інформацією: дані користувачів, новини, музичні релізи, товари, замовлення, медіафайли тощо. Для забезпечення надійного зберігання, доступу та управління цими даними була спроєктована та реалізована система, що використовує реляційну базу даних, хмарний сервіс для її хостингу, ORM-фреймворк для взаємодії з даними на рівні застосунку та спеціалізований сервіс для зберігання медіафайлів.

3.2.1 Система управління базами даних: MySQL

В якості основної системи управління базами даних для вебзастосунку Threllia була обрана MySQL. MySQL є реляційною СУБД, що добре підходить для зберігання структурованих даних, таких як інформація про користувачів, їхні замовлення, адреси, товари в магазині, новини гурту, інформація про тури та пісні.

Обґрунтування вибору MySQL:

- **Реляційна модель:** Дозволяє ефективно моделювати зв'язки між різними сутностями застосунку (наприклад, користувач та його замовлення, замовлення та товари в ньому).
- **Стабільність та надійність:** MySQL є перевіреною часом, стабільною СУБД, що широко використовується у веб-розробці.
- **Продуктивність:** Забезпечує належний рівень продуктивності для операцій читання та запису, необхідних для застосунку.
- **Сумісність:** Добре інтегрується зі стеком технологій, що використовуються у Backend (Java, Spring Boot).
- **Спільнота та документація:** Наявність великої спільноти та вичерпної документації спрощує розробку та вирішення можливих проблем.

Структура бази даних була спроєктована відповідно до визначених сутностей та їх взаємозв'язків, що відображено у діаграмі моделі даних.

3.2.2 Хостинг бази даних: Amazon RDS

Замість самостійного налаштування та адміністрування сервера бази даних було прийнято рішення використовувати хмарний сервіс керованих баз даних Amazon Relational Database Service. Розгортання екземпляра MySQL на AWS RDS надає низку переваг:

- **Спрощене адміністрування:** AWS бере на себе завдання з оновлення програмного забезпечення СУБД, застосування патчів безпеки, налаштування резервного копіювання та моніторингу.
- **Масштабованість:** Дозволяє легко масштабувати ресурси бази даних (потужність процесора, обсяг пам'яті та сховища) відповідно до зростання навантаження на застосунок.
- **Висока доступність та надійність:** Надає можливості для налаштування відмовостійких конфігурацій (наприклад, Multi-AZ), що підвищує доступність бази даних.
- **Безпека:** Пропонує вбудовані механізми для забезпечення безпеки бази даних, включаючи шифрування даних та контроль доступу.

Параметри підключення до бази даних на AWS RDS (URL, ім'я користувача, пароль) безпечно передаються до Backend-застосунку на платформі Render через змінні середовища, як описано у розділі 4.4.2.

3.2.3 Рівень доступу до даних: Spring Data JPA та Hibernate

Взаємодія між Backend-застосунком на Spring Boot та базою даних MySQL реалізована за допомогою Spring Data JPA. Spring Data JPA є частиною екосистеми Spring, що значно спрощує створення рівня доступу до даних. [15]

Ключові аспекти використання Spring Data JPA:

- **Абстракція над JPA:** Надає високорівневу абстракцію над стандартним Java Persistence API (JPA).
- **Репозиторії:** Дозволяє визначати інтерфейси репозиторіїв, розширюючи базові інтерфейси Spring Data (такі як JpaRepository). Spring Data JPA автоматично генерує реалізацію стандартних CRUD-операцій та дозволяє

легко створювати кастомні запити за допомогою іменування методів або анотації @Query.

- **Зменшення шаблонного коду:** Значно скорочує обсяг коду, необхідного для взаємодії з базою даних, порівняно з використанням чистого JDBC або JPA EntityManager.

В якості реалізації JPA під капотом Spring Data JPA за замовчуванням використовується **Hibernate**. Hibernate є потужним та популярним фреймворком Object-Relational Mapping. [16]

Роль Hibernate:

- **Мапінг об'єктів на реляційні таблиці:** Hibernate відповідає за відображення Java-класів, анотованих як @Entity, на відповідні таблиці у базі даних MySQL.
- **Автоматична генерація SQL:** Генерує та виконує необхідні SQL-запити для операцій з об'єктами, звільняючи розробника від написання SQL вручну.
- **Управління транзакціями:** Інтегрується з механізмами управління транзакціями Spring (анотація @Transactional), забезпечуючи цілісність даних.
- **Додаткові можливості:** Надає розширені функції, такі як кешування даних, ліниве завантаження (lazy loading) пов'язаних сутностей та оптимістичне блокування.

Використання Spring Data JPA разом з Hibernate дозволило створити ефективний, гнучкий та легкий у підтримці рівень доступу до даних.

3.2.4 Зберігання медіафайлів: Cloudinary

Для зберігання медіафайлів, таких як зображення товарів, обкладинки альбомів, фотографії для новин та медіагалереї, було вирішено не зберігати їх безпосередньо у базі даних MySQL (що є неефективним для великих бінарних об'єктів), а використати спеціалізований хмарний сервіс – **Cloudinary**. [17]

Функції Cloudinary:

- **Зберігання медіа:** Надає хмарне сховище, оптимізоване для зберігання зображень та відео.
- **Обробка та трансформація:** Дозволяє виконувати різноманітні операції над медіафайлами "на льоту" (зміна розміру, обрізка, накладання водяних знаків, оптимізація формату тощо).
- **Доставка контенту:** Інтегрований з мережею доставки контенту для швидкого завантаження медіафайлів користувачами по всьому світу.

Інтеграція з застосунком: Backend-застосунок взаємодіє з Cloudinary API. Коли користувач (наприклад, адміністратор) завантажує зображення через інтерфейс застосунку, Backend відправляє цей файл до Cloudinary. Cloudinary зберігає файл і повертає унікальну URL-адресу цього файлу. Саме ця URL-адреса зберігається у відповідному полі в базі даних MySQL. Клієнтська частина потім використовує ці URL для відображення зображень.

Такий підхід дозволяє розвантажити основний сервер застосунку та базу даних від зберігання великих файлів, покращити швидкість завантаження медіа для користувачів та отримати доступ до потужних інструментів обробки зображень.

Висновок: Комбінація реляційної бази даних MySQL, розміщеної на надійній платформі AWS RDS, використання Spring Data JPA та Hibernate для ефективної взаємодії з даними на рівні коду, а також інтеграція зі спеціалізованим сервісом Cloudinary для управління медіафайлами, забезпечує комплексну, масштабовану та продуктивну систему управління даними для вебзастосунку Threllia.

3.3 Розгортання вебзастосунку Threllia

Розгортання є фінальним етапом процесу розробки, що робить вебзастосунок доступним для кінцевих користувачів через мережу Інтернет. Для вебзастосунку Threllia було обрано сучасні хмарні платформи як послуги, які

спрощують процес розгортання, налаштування та масштабування: Netlify для клієнтської частини та Render для серверної частини. Також була використана технологія контейнеризації Docker для пакування та розгортання Backend.

3.3.1 Розгортання клієнтської частини на Netlify

Клієнтська частина застосунку, розроблена з використанням React.js, є статичним односторінковим застосунком, що робить платформу Netlify оптимальним вибором для її розміщення. Netlify спеціалізується на хостингу статичних сайтів та надає інструменти для автоматизації збірки та розгортання, а також глобальну мережу доставки контенту для швидкого доступу користувачів з будь-якої точки світу. [18]

Процес розгортання на Netlify:

1. **Підключення репозиторію:** Проєкт клієнтської частини, розміщений у системі контролю версій Git, був підключений до платформи Netlify.
2. **Налаштування збірки:** У налаштуваннях сайту на Netlify були вказані команди для збірки проєкту та директорія, в якій зберігаються статичні файли після збірки.
3. **Конфігурація змінних середовища:** Були налаштовані необхідні змінні середовища, зокрема VITE_API_BASE_URL, яка вказує на URL розгорнутої серверної частини на платформі Render. Це дозволяє клієнтській частині коректно відправляти запити до API.
4. **Автоматичне розгортання (CI/CD):** Netlify автоматично відстежує зміни у підключеній гілці Git-репозиторію. Кожне оновлення коду у цій гілці автоматично ініціює процес збірки та розгортання нової версії клієнтської частини.

В результаті клієнтська частина вебзастосунку Threllia стала доступною за унікальною URL-адресою, наданою Netlify.

3.3.2 Розгортання серверної частини на Render з використанням Docker

Серверна частина, розроблена на Spring Boot, потребує середовища виконання Java та можливості підключення до бази даних. Платформа Render була обрана завдяки гнучкості, підтримці Docker-контейнерів, можливості розміщення вебсервісів та керованих баз даних. Використання Docker дозволило запакувати застосунок разом з усіма залежностями та середовищем виконання, забезпечуючи консистентність між середовищем розробки та розгортання. [19]

Процес розгортання на Render:

1. **Докеризація застосунку:** Був створений файл Dockerfile для серверної частини. Цей файл містить інструкції для створення Docker-образу:
    - Визначення базового образу з необхідною версією Java.
    - Копіювання зібраного JAR-файлу застосунку всередину образу.
    - Визначення порту, який слухає застосунок Spring Boot.
    - Вказівка команди для запуску застосунку при старті контейнера.
2. **Створення вебсервісу на Render:** На платформі Render був створений новий сервіс типу "Web Service".
3. **Підключення репозиторію:** До створеного сервісу був підключений Git-репозиторій, що містить код серверної частини та Dockerfile.
4. **Налаштування середовища:** Render автоматично визначив наявність Dockerfile та був налаштований на використання Docker як середовища виконання. [20]
5. **Конфігурація змінних середовища:** Через інтерфейс Render були налаштовані всі необхідні змінні середовища для роботи Spring Boot застосунку:
    - Параметри підключення до бази даних (SPRING_DATASOURCE_URL, SPRING_DATASOURCE_USERNAME, SPRING_DATASOURCE_PASSWORD), яка може бути розгорнута

як окремий керований сервіс баз даних на Render або на зовнішній платформі.

- Секретний ключ для генерації та валідації JWT (JWT_SECRET), що використовується JwtTokenValidator.
- URL клієнтської частини (FRONTEND_URL), необхідний для коректного налаштування CORS у Spring Security [source: конфігурація Backend].
- Ключі доступу до зовнішніх сервісів, якщо вони використовуються (наприклад, Cloudinary, Stripe).

6. **Автоматичне розгортання:** Аналогічно до Netlify, Render відстежує зміни у підключеній гілці репозиторію. Кожне оновлення коду ініціює автоматичну збірку нового Docker-образу та його розгортання у створеному вебсервісі.

Після успішного розгортання серверна частина стала доступною за стабільною URL-адресою, наданою Render.

3.3.3 Завершальні налаштування та інтеграція

На завершальному етапі було перевірено, що:

- Клієнтська частина на Netlify коректно налаштована для взаємодії з URL серверної частини на Render (через змінну середовища VITE_API_BASE_URL).
- Конфігурація CORS на серверній частині дозволяє отримувати запити з домену, на якому розміщена клієнтська частина на Netlify.

Висновок: Обрана стратегія розгортання з використанням платформ Netlify та Render, а також технології Docker, дозволила ефективно, автоматизовано та надійно розмістити повнофункціональний вебзастосунок Threllia в мережі Інтернет. Це забезпечує легкість подальшого оновлення, обслуговування та потенційного масштабування застосунку.

ВИСНОВКИ

У даній кваліфікаційній роботі було розглянуто актуальну проблему забезпечення ефективної онлайн-присутності для музичних гуртів в умовах цифрової трансформації музичної індустрії. Було відзначено, що існуючі рішення, такі як соціальні мережі та сторонні платформи, часто є фрагментованими та мають обмеження у функціональності, що ускладнює пряму взаємодію з фанатами та централізоване представлення творчості й продаж продукції .

Основною метою роботи було проєктування, розробка та тестування повнофункціонального вебзастосунку Threllia, який слугує централізованою онлайн-платформою для музичного гурту. Для досягнення цієї мети було вирішено низку завдань , що дозволило отримати наступні результати:

1. На основі аналізу вимог та огляду існуючих рішень було спроєктовано [1] багаторівневу архітектуру вебзастосунку. Обрана архітектура з чітким розділенням на клієнтську частину (Frontend) на базі React.js та серверну частину (Backend) на базі Spring Boot , що взаємодіють через REST API, забезпечує гнучкість, масштабованість та зручність супроводу системи.
2. Реалізовано серверну частину застосунку з використанням Java 17 та Spring Boot . Впроваджено надійну систему управління даними на основі реляційної СУБД MySQL, розміщеної на хмарній платформі AWS RDS , з використанням Spring Data JPA та Hibernate для об'єктно-реляційного відображення. Для ефективного зберігання медіафайлів інтегровано хмарний сервіс Cloudinary.
3. Розроблено безпечну систему автентифікації та авторизації користувачів (фанатів та адміністраторів) на основі Spring Security та JSON Web Tokens (JWT). Реалізовано механізми захисту кінцевих точок API та розмежування доступу на основі ролей, що забезпечує належний рівень безпеки даних.

4. Створено інтерактивну та адаптивну клієнтську частину застосунку за допомогою бібліотеки React.js та сучасних інструментів Frontend-розробки (Vite, React Router, Axios). Ефективне управління станом застосунку реалізовано за допомогою Redux Toolkit . Для стилізації інтерфейсу використано Tailwind CSS та бібліотеку компонентів ShadCN UI , що дозволило створити консистентний та естетично привабливий дизайн.
5. Реалізовано всі ключові функціональні модулі, визначені на етапі аналізу вимог: "Музика", "Пісні", "Новини", "Тури", "Магазин" (з кошиком та оформленням замовлення), "Медіа-галерея" та "Панель адміністратора" з CRUD-операціями для керування контентом .
6. Проведено комплексне тестування застосунку, включаючи функціональне тестування API за допомогою Postman, інтеграційне тестування наскрізних сценаріїв взаємодії Frontend та Backend, а також ручне тестування користувацького інтерфейсу. Результати тестування підтвердили працездатність усіх модулів, коректність обробки даних, належну роботу системи безпеки та відповідність застосунку функціональним та нефункціональним вимогам. Це обґрунтовує достовірність отриманих результатів розробки.
7. Розроблений вебзастосунок успішно розгорнуто на хмарних платформах: клієнтська частина на Netlify, серверна частина (у вигляді Docker-контейнера) та база даних на Render/AWS RDS . Обрана стратегія розгортання забезпечує доступність, надійність та можливість легкого оновлення й масштабування застосунку.

Таким чином, у результаті виконання кваліфікаційної роботи було створено повноцінний програмний продукт – вебзастосунок Threllia, який вирішує поставлене завдання створення централізованої онлайн-платформи для музичного гурту. Практична значущість роботи полягає у наданні гуртам сучасного інструменту для взаємодії з аудиторією, промоції власної творчості та

ведення комерційної діяльності, що сприяє зміцненню їх позицій у цифровій музичній індустрії.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Яскевич, Владислав Олександрович (2023) *JavaScriptбібліотека React Навчальний посібник для студентів спеціальності Комп'ютерні науки* Київський університет імені Бориса Грінченка, Україна, Київ. https://elibrary.kubg.edu.ua/id/eprint/48587/

2. В Машкіна, ВО Абрамов, ТІ Носенко, ЄВ Іваніченко.Навчально-методичний посібник до виконання і захисту кваліфікаційної роботи бакалавра спеціальностей 122 Комп'ютерні науки для здобувачів вищої освіти денної форми навчання, Київський столичний університет імені Бориса Грінченка. Київ, 2024.- 43 с.

3. Теоретичні та практичні аспекти використання математичних методів та інформаційних технологій в освіті й науці:моногр. / за заг. ред. О. Литвин. — К.: Київ. ун-т ім. Б. Грінченка,2021. — 332 с.

4. ОБ Придибайло, РВ Придибайло, Владислав Олександрович Яскевич, Юрій Владиславович Яскевич, «Архітектура нульової довіри: логічні компоненти та підходи запровадження», Зв'язок , № 3, 2024, стор 7-11

5. CodeSnippet. Spring Security Architecture: The Bigger Picture Explained! , 2025. *YouTube*. URL: https://www.youtube.com/watch?v=7eSQd7me6QI (дата звернення: 21.04.2025).
6. Hudge S. Layers in software architecture. *Medium*. URL: https://medium.com/@sagar.hudge/layers-in-software-architecture-c8cc16329ff6 (дата звернення: 21.04.2025).
7. React router home | react router. *React Router Official Documentation*. URL: https://reactrouter.com/home (дата звернення: 21.04.2025).
8. Getting Started with React Redux | React Redux. *React Redux | React Redux*. URL: https://react-redux.js.org/introduction/getting-started (дата звернення: 21.04.2025).
9. Починаючи | axios docs. *Axios*. URL: https://axios-http.com/uk/docs/intro (дата звернення: 21.04.2025).
10. Installing with Vite - Installation. *Tailwind CSS - Rapidly build modern websites without ever leaving your HTML*. URL:

https://tailwindcss.com/docs/installation/using-vite (дата звернення: 21.04.2025).

11. Introduction. *Build your component library - shadcn/ui*. URL: "https://ui.shadcn.com/docs".shadcn.com/docs (date of access: 21.04.2025).
12. Java Configuration :: Spring Security. *Spring | Home*. URL: https://docs.spring.io/spring-security/reference/s (дата звернення: 21.04.2025).
13. CodeSnippet. Spring Security Architecture: The Bigger Picture Explained! , 2025. *YouTube*. URL: https://www.youtube.com/watch?v=7eSQ (дата звернення: 21.04.2025).
14. Architecture :: spring security. *Spring | Home*. URL: https://docs.spring.io/spring-security/reference/servlet/architecture.html (дата звернення: 21.04.2025).
15. Що таке атаки CSRF (міжсайтова підробка запиту) і як їм запобігти - CoreWin. *CoreWin*. URL: https://corewin.ua/blog/cross-site-request-forgery-csrf/ (дата звернення: 21.04.2025).
16. Cross-Origin Resource Sharing (CORS) - HTTP | MDN. *MDN Web Docs*. URL: https://developer.mozilla.org/ru/docs/Web/HTTP/Guides/CORS (дата звернення: 21.04.2025).
17. ByteByteGo. Stateless Architecture: The Key to Building Scalable and Resilient Systems. *ByteByteGo Newsletter | Alex Xu | Substack*. URL: https://blog.bytebytego.com/p/stateless-architecture-the-key-to (дата звернення: 21.04.2025).
18. Spring Data JPA. *Spring Data JPA*. URL: https://spring.io/projects/spring-data-jpa (дата звернення: 21.04.2025).
19. Your relational data. Objectively. - Hibernate ORM. *Hibernate*. URL: https://hibernate.org/orm/ (дата звернення: 21.04.2025).

20. Cloudinary Image & Video Management - Documentation Home | Cloudinary. *Image and Video Upload, Storage, Optimization and CDN*. URL: https://cloudinary.com/documentation (дата звернення: 21.04.2025).
21. Welcome to Netlify. *Welcome to Netlify | Netlify Docs*. URL: https://docs.netlify.com/ (дата звернення: 21.04.2025).
22. Docs + Quickstarts – Render Docs. *Docs + Quickstarts – Render Docs*. URL: https://render.com/docs (дата звернення: 21.04.2025).
23. Store configuration data using Docker Configs. *Docker Documentation*. URL: https://docs.docker.com/engine/swarm/configs/ (дата звернення: 21.04.2025).