Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Веб-застосунок з використанням Web API
для адміністрування кінотеатру
(тема)

Виконав:
здобувач 3 року навчання,
групи КІУКІу-22-1
Костянтин ПОПОВ
(власне ім’я, прізвище)

Спеціальність
123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Комп’ютерна інженерія
(повна назва освітньої програми)

Керівник: ас. Олег ЖУРИЛО
(посада, власне ім’я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ ________________ Андрій КОВАЛЕНКО
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп'ютерна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ____________
(підпис)
"____" ______________ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Попову Костянтину Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Веб-застосунок з використанням Web API для адміністрування кінотеатру

затверджена наказом по університету від " 26 " травня 2025 р. № 425 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 14 липня 2025 р.

3. Вхідні дані до роботи 1) архітектура системи: клієнт-серверна; 2) стек протоколів: TCP/IP; 3) аутентифікація/авторизація: JSON WEB Tokens

4. Перелік питань, що потрібно опрацювати у роботі

1) аналіз проблеми та огляд існуючих рішень;
2) вибір технології розробки та інструментальних засобів;
3) розробка алгоритмічного забезпечення;
4) розробка програмних модулів;
5) відлагодження програмних модулів;
6) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп’ютерних ілюстрацій

Слайд-презентація – 10 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	10.06.25-13.06.25	
2	Вибір технології розробки та	14.06.25-17.06.25	
	інструментальних засобів		
3	Розробка алгоритмічного забезпечення	18.06.25-21.06.25	
4	Розробка програмних модулів	23.06.25-28.06.25	
5	Відлагодження програмних модулів	30.06.25-02.07.25	
6	Оформлення матеріалів кваліфікаційної роботи	03.07.25-05.07.25	
7	Подання кваліфікаційної роботи керівникові та	07.07.25-09.07.25	
	її попередній захист		
8	Подання кваліфікаційної роботи на рецензу-	10.07.25-11.07.25	
	вання		

Дата видачі завдання “ 09 ” червня 2025 р.

Здобувач ____________________
(підпис)

Керівник роботи ____________________ ас. Олег ЖУРИЛО
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 70 с., 7 рис., 1 дод., 7 джерел.

ВЕБ-ЗАСТОСУНОК, КІНОТЕАТР, БАЗА ДАНИХ, БЕКЕНД, ФРОНТЕНД, АУТЕНТИФІКАЦІЯ, АВТОРИЗАЦІЯ, API, JWT, CRUD, CRM.

Метою кваліфікаційної роботи є розробка та реалізація веб-застосунку для ефективного адміністрування кінотеатру, що надасть зручний інтерфейс для взаємодії клієнтів та адміністрації закладу.

У ході виконання кваліфікаційної роботи було спроєктовано та реалізовано повноцінний веб-застосунок, що складається з серверної частини (Backend) на базі ASP.NET Core Web API та клієнтської частини (Frontend) на Angular.

Розроблено реляційну базу даних (MS SQL Server) для надійного зберігання інформації про фільми, сеанси, кінозали, клієнтів та адміністрацію.

Реалізовано комплексний функціонал для управління цими сутностями, включаючи додавання, видалення та редагування даних.

Важливим аспектом стало впровадження механізмів аутентифікації та авторизації користувачів з розмежуванням прав доступу за ролями за допомогою JWT-токенів.

Frontend забезпечує зручний інтерфейс для перегляду розкладу фільмів та бронювання квитків для клієнтів, а також повний доступ до управлінського функціоналу для адміністрації.

ABSTRACT

Bachelor’s thesis: 70 pages, 7 figures, 1 appendices, 7 sources.

WEB APPLICATION, CINEMA, DATABASE, BACKEND, FRONTEND, AUTHENTICATION, AUTHORIZATION, API, JWT, CRUD, CRM.

The major goal of this thesis is to develop and implement a web application for efficient cinema administration, providing a user-friendly interface for interaction among clients and the cinema's administration.

In the course of this thesis, a full-fledged web application was designed and implemented, consisting of a server-side (Backend) based on ASP.NET Core Web API and a client-side (Frontend) built with Angular.

A relational database (MS SQL Server) was developed for reliable storage of information about movies, showtimes, cinema halls, clients, and administration.

Comprehensive functionality for managing these entities, including adding, deleting, and editing data, was implemented.

A crucial aspect was the integration of user authentication and authorization mechanisms with role-based access control using JWT tokens.

The Frontend provides a convenient interface for clients to view schedules and book tickets, as well as full access to administrative functionality for the cinema's management.

Functional testing, including API verification via Swagger UI, confirmed the system's correct operation.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 8
ВСТУП 9
1 СИСТЕМНІ ВИМОГИ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 11
1.1 Аналіз предметної області та існуючих рішень 11
1.1.1 Опис предметної області 11
1.1.2 Огляд існуючих рішень та їх аналіз 12
1.2 Визначення системних вимог до веб-застосунку 13
1.2.1 Опис функціональних вимог 13
1.2.2 Опис нефункціональних вимог 15
1.3 Моделювання бізнес-процесів кінотеатру 17
2 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ОБҐРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ 19
2.1 Вимоги до програмного забезпечення 19
2.1.1 Опис функцій програмного забезпечення 19
2.1.2 Вимоги до обробки даних 19
2.1.3 Вимоги до інтерфейсу користувача 20
2.2 Огляд сучасних середовищ розробки (IDE) 21
2.3 Обґрунтування вибору технологій та засобів розробки 22
2.3.1 Обґрунтування вибору Frontend (Angular) 22
2.3.2 Обґрунтування вибору Backend (C# WEB API) 23
2.3.3 Обґрунтування вибору бази даних (MS SQL) 24
2.3.4 Обґрунтування вибору середовищ розробки 25
2.3.5 Обґрунтування вибору допоміжного ПЗ та сервісів 26
3 АРХІТЕКТУРА ПРОГРАМНОЇ СИСТЕМИ 28
3.1 Загальна архітектура веб-застосунку 28
3.2 Проектування структури бази даних 30
3.3 Проектування архітектури Backend 33

3.3.1 Структура C# WEB API та основні компоненти 33
3.3.2 Аутентифікація та авторизація 38
3.4 Проектування архітектури Frontend 39
4 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 44
4.1 Розробка програмних компонентів Frontend 44
4.1.1 Опис інтерфейсу взаємодії ПЗ з користувачем 44
4.1.2 Реалізація функціоналу для клієнтів 45
4.1.3 Реалізація функціоналу для адміністрації 46
4.2 Розробка програмних компонентів Backend 48
4.2.1 Реалізація бізнес-логіки програмних компонентів 48
4.2.2 Розробка компонентів для взаємодії з базою даних 49
4.2.3 Реалізація механізмів авторизації та аутентифікації користувачів 50
4.3 Опис використаних алгоритмів та структур даних 50
5 ТЕСТУВАННЯ ТА ВВЕДЕННЯ В ЕКСПЛУАТАЦІЮ 53
5.1. Методи та стратегії тестування ПЗ 53
5.2. Розробка тестових прикладів та сценаріїв 54
5.3. Проведення тестування (функціональне, API-тестування) 55
5.4. Аналіз результатів тестових експериментів та виявлення помилок 57
5.5. Опис налагодження ПЗ 57
5.6. Інструкції щодо розгортання ПЗ (системні вимоги, мінімальна конфігурація апаратного та програмного забезпечення) 58
5.7. Рекомендації щодо використання та супроводу розроблених програм 60
ВИСНОВКИ 62
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 64
ДОДАТОК А ГРАФІЧНИЙ МАТЕРІАЛ КВАЛІФІКАЦІЙНОЇ РОБОТИ 65

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

БД – База даних

ПЗ – Програмне забезпечення

API – Application Programming Interface

ASP.NET Core Web API – Асинхронний веб-фреймворк для створення веб-сервісів та API на платформі .NET

CRUD – Create, Read, Update, Delete (Створити, Прочитати, Оновити, Видалити)

DI – Dependency Injection

DTO – Data Transfer Object

EF Core – Entity Framework Core

HTTPS – Hypertext Transfer Protocol Secure

IDE – Integrated Development Environment

JSON – JavaScript Object Notation

JWT – JSON Web Token

LINQ – Language Integrated Query

MS SQL Server – Microsoft SQL Server

ORM – Object-Relational Mapper

RAM – Random Access Memory

SDK – Software Development Kit

SPA – Single Page Application

SSD – Solid State Drive

UI – User Interface

VS Code – Visual Studio Code

ВСТУП

Сучасні кінотеатри перетворилися з простих місць для перегляду фільмів на багатофункціональні розважальні комплекси, де кожен відвідувач шукає не лише якісного кінопоказу, а й комфорту, високого рівня обслуговування та персоналізованого підходу.

У таких умовах, ефективне управління закладом стає неможливим без впровадження спеціалізованих програмних рішень. Ці системи виступають невидимими, але незамінними помічниками, що забезпечують безперебійну та злагоджену роботу всіх внутрішніх процесів.

Щодня адміністратори та персонал кінотеатру стикаються з величезним обсягом рутинних операцій: реєстрація нових клієнтів, формування та оновлення розкладів кіносеансів, детальний контроль продажу квитків, управління різноманітними типами абонементів/програм лояльності та обробка платежів.

Будь-яка пропущена деталь, некоректний запис або помилка у документації може призвести до значних фінансових втрат, марнування часу, а головне – до зниження довіри та лояльності клієнтів, що є критично важливим для конкурентного ринку.

Саме тому сучасний кінотеатр потребує не просто електронної бази даних, а комплексної, інтелектуальної системи управління, яка стане надійним та автоматизованим партнером у щоденній операційній діяльності, а також включатиме CRM систему для роботи з клієнтами.

У цій роботі буде розроблено інноваційний веб-застосунок, спроектований спеціально для адміністрування кінотеатру.

Цей потужний інструмент гармонійно поєднує інтуїтивно зрозумілий та сучасний інтерфейс користувача з розширеною функціональністю корпоративного рівня. Система дозволить у реальному часі ефективно керувати всіма ключовими аспектами роботи закладу: від ведення детальної

клієнтської бази даних та формування гнучких розкладів показів до моніторингу завантаженості кінозалів та управління персоналом. Особливу актуальність цей проєкт набуває в умовах постійного та стрімкого зростання конкуренції на ринку кінопоказів, де якість обслуговування, швидкість реагування на запити клієнтів та оптимізація внутрішніх процесів стають вирішальними факторами для успіху та сталого розвитку бізнесу.

Для клієнтів впровадження цієї системи означатиме новий рівень сервісу та комфорту, де кожен відвідувач відчуватиме індивідуальний підхід, зручність у бронюванні квитків та доступ до актуального розкладу фільмів.

Для персоналу кінотеатру це буде не просто програмне забезпечення, а потужний та зручний інструмент, який трансформує рутинні та часозатратні операції на ефективну та зосереджену роботу з клієнтами.

Розробка даного рішення ґрунтується на сучасному та перевіреному стеку технологій (Angular для Frontend, ASP.NET Core Web API для Backend та MS SQL Server для бази даних), що гарантує високу стабільність роботи, надійну безпеку даних та надає широкі можливості для подальшого масштабування та розвитку системи відповідно до зростаючих потреб бізнесу у майбутньому.

1 СИСТЕМНІ ВИМОГИ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної області та існуючих рішень

1.1.1 Опис предметної області

Предметна область «Кінотеатр» характеризується наявністю кількох ключових сутностей та їх взаємодією:

- клієнти – особи, що відвідують кінотеатр та купують квитки. Важливо мати їхні персональні дані (ПІБ, контактна інформація), а також інформацію для доступу до системи (логін, пароль) та можливість бронювання/купівлі квитків;
- адміністрація – персонал, що керує роботою кінотеатру. Сюди входить роль адміністратора, яка має свої права доступу до системи та можливість адміністрування інформації про фільми, сеанси, кінозали та клієнтів;
- кінозали – приміщення, де проводяться кіносеанси. Важливо мати інформацію про доступні кінозали, їх місткість та розташування;
- фільми – кінострічки, що демонструються. Необхідно зберігати інформацію про назву, опис, тривалість, жанр, вікові обмеження, постер, а також статус (в прокаті, очікується);
- сеанси – заплановані покази фільмів у конкретному кінозалі в певний час. Включає інформацію про фільм, кінозал, дату та час початку, ціну квитка, кількість вільних місць.

Основні процеси, що потребують автоматизації: ведення обліку клієнтів, фільмів та сеансів, формування та управління розкладом показів, бронювання та продаж квитків, контроль доступу користувачів до різної інформації та функціоналу системи.

1.1.2 Огляд існуючих рішень та їх аналіз

На сучасному ринку представлено ряд спеціалізованих CRM-систем та програмних комплексів для кінотеатрів. До найбільш відомих глобальних рішень належать такі, як Veezi (by Vista Cinema), Spectra Ticketing & Fan Engagement (частина Vivendi Village), UCC (Ultimate Cinema Control) та інші. Ці системи є комплексними та використовуються великими мережами кінотеатрів по всьому світу.

В Україні також існують розробки та адаптовані рішення для кінотеатрів. До прикладу, можна згадати TicketsBox (також відомий як kassa.in.ua), який є платформою для продажу квитків та може мати певні CRM-функції для організаторів подій, включаючи кінотеатри. Також багато кінотеатрів використовують власні або індивідуально розроблені системи, або адаптують існуючі рішення для управління бронюваннями та продажами.

Аналіз цих систем (як глобальних, так і доступних в Україні) дозволяє виділити як їхні сильні сторони, так і існуючі обмеження, що стосуються управління специфічними потребами кінотеатрів.

Сильні сторони існуючих рішень:

- більшість систем ефективно справляються з базовими завданнями, такими як ведення клієнтської бази, формування загального розкладу сеансів, облік квитків та фінансовий моніторинг;
- наявність функцій онлайн-бронювання, контролю продажу, SMS/E-mail розсилок та генерації базових звітів значно автоматизує рутинні операції;
- деякі рішення пропонують мобільні додатки для клієнтів, що підвищує зручність бронювання квитків та перегляду особистої інформації;
- інтеграція з платіжними системами та можливості обліку доходів/витрат є важливим аспектом;
- часто існують вбудовані CRM-функції для управління програмами лояльності та ведення історії покупок клієнтів.

Виявлені недоліки та обмеження існуючих систем:

- існуючі рішення часто мають жорстко визначені ролі користувачів, що не дозволяє реалізувати тонке налаштування прав доступу для різних категорій персоналу кінотеатру;

- часто системи не дозволяють зручно переглядати та фільтрувати розклад сеансів одночасно за фільмами та за кінозалами, що ускладнює процес вибору квитків для клієнтів та планування для адміністрації;

- більшість комерційних рішень є «коробковими» продуктами з обмеженими можливостями інтеграції із зовнішніми системами через відкритий API, що ускладнює їх адаптацію під специфічні потреби конкретного кінотеатру або розширення функціоналу;

- ліцензування та підтримка деяких корпоративних рішень є досить дорогими, що робить їх недоступними для малих та середніх кінотеатрів, які шукають більш бюджетні та адаптовані варіанти;

- адаптація наявних систем під унікальні бізнес-процеси та вимоги конкретного кінотеатру часто вимагає значних зусиль та додаткових фінансових вкладень.

1.2 Визначення системних вимог до веб-застосунку

1.2.1 Опис функціональних вимог

Функціональні вимоги визначають, що саме має робити система, які функції вона повинна виконувати для задоволення потреб користувачів. Веб-застосунок має забезпечувати наступний функціонал:

а) управління інформацією про фільми:

1) реєстрація нових фільмів із зазначенням назви, опису, тривалості, жанру, року випуску, акторського складу, ціна фільму, жанрів, постеру та трейлера;

2) можливість зміни всіх вищезазначених даних існуючих фільмів;

3) можливість видалення даних про фільми;

4) перегляд детальної інформації про всі фільми;

б) Управління інформацією про кінозали:

1) реєстрація нових кінозалів із зазначенням їх назви, опису, місткості (кількості місць);

2) можливість зміни даних існуючих кінозалів;

3) можливість видалення даних про кінозали;

4) перегляд інформації про всі доступні кінозали;

в) управління інформацією про сеанси:

1) створення нових сеансів із зазначенням фільму, кінозалу, дати та часу проведення, та максимальної кількості місць (автоматично з кінозалу);

2) можливість зміни всіх деталей існуючих сеансів;

3) можливість видалення сеансів з розкладу;

г) управління інформацією про клієнтів (CRM функціонал):

1) реєстрація нових клієнтів із зазначенням ПІБ, дати народження, контактної інформації (телефон, email) та облікових даних (логін, пароль);

2) можливість зміни всіх вищезазначених даних існуючих клієнтів;

3) можливість видалення даних про клієнтів;

4) перегляд детальної інформації про всіх клієнтів та їхню історію покупок;

5) управління програмами лояльності, знижками та спеціальними пропозиціями для клієнтів;

6) можливість розсилки інформації про нові фільми та акції;

ґ) можливість перегляду розкладу сеансів для клієнтів та адміністрації;

д) функціонал бронювання та продажу квитків:

1) вибір фільму та сеансу;

2) вибір місць у кінозалі;

3) бронювання квитків користувачем;

4) скасування бронювання квитків;

е) взаємодія користувачів:

1) доступ до персонального кабінету для кожного зареєстрованого користувача;

2) механізми авторизації та аутентифікації користувачів (клієнтів та адміністраторів).

1.2.2 Опис нефункціональних вимог

Нефункціональні вимоги визначають, наскільки добре система повинна виконувати свої функції, її характеристики та обмеження:

а) продуктивність:

1) час відгуку – система повинна забезпечувати швидкий час відгуку (не більше 2-3 секунд) на основні запити (завантаження розкладу, бронювання квитків, пошук фільмів/сеансів).

2) обробка навантаження – здатність системи коректно функціонувати при одночасній роботі до 50 користувачів;

б) надійність:

1) стійкість до збоїв – система повинна бути стійкою до непередбачених помилок та збоїв, мінімізуючи втрату даних;

2) резервне копіювання – передбачити механізми регулярного резервного копіювання бази даних для запобігання втраті інформації;

3) відновлення – можливість швидкого відновлення системи після збою;

в) зручність використання (Usability):

1) інтуїтивність – інтерфейс користувача має бути простим, логічним та інтуїтивно зрозумілим для всіх категорій користувачів (клієнтів та адміністрації);

2) зворотній зв’язок – система повинна надавати чіткий зворотний зв’язок користувачеві про стан його операцій (успішне збереження, помилка, очікування);

3) адаптивний дизайн – інтерфейс повинен коректно відображатися

на різних пристроях;

г) безпека:

1) контроль доступу – реалізація надійної системи аутентифікації та авторизації з ролевим контролем доступу (для клієнтів та адміністраторів);

2) захист даних – забезпечення конфіденційності, цілісності та доступності даних, включаючи персональні дані користувачів та дані про транзакції;

3) запобігання атакам – захист від поширених веб-атак;

4) шифрування – використання захищених протоколів передачі даних;

ґ) супроводжуваність:

1) модульна архітектура – код повинен бути добре структурований та модульний, що спростить подальше внесення змін та розширення функціоналу;

2) документація – наявність технічної документації для розробників та адміністраторів;

3) легкість виправлення помилок – можливість швидкого виявлення та виправлення помилок;

д) портативність/сумісність:

1) крос-браузерність – система повинна коректно працювати у всіх сучасних веб-браузерах (Chrome, Firefox, Edge, Safari);

2) підтримка ОС – застосунок має бути доступним з різних операційних систем, що підтримують сучасні веб-браузери;

е) масштабованість:

1) збільшення обсягів даних – можливість розширення бази даних та інфраструктури для зберігання зростаючих обсягів інформації;

2) збільшення кількості користувачів – архітектура повинна дозволяти масштабування для підтримки зростаючої кількості одночасних користувачів.

1.3 Моделювання бізнес-процесів кінотеатру

Моделювання бізнес-процесів є важливим етапом проектування програмного забезпечення, що дозволяє візуалізувати та проаналізувати потік робіт, взаємодії між учасниками та дані, які обробляються.

Це допомагає ідентифікувати ключові етапи, потенційні вузькі місця та вимоги до автоматизації. Для веб-застосунку для адміністрування кінотеатру моделювання бізнес-процесів дозволяє чітко визначити, як буде відбуватися взаємодія між клієнтами та адміністрацією в контексті використання системи.

В рамках даного проекту основна увага приділяється моделюванню таких ключових бізнес-процесів:

а) процес реєстрації користувача та входу в систему:

1) початок – незареєстрований користувач бажає отримати доступ до функціоналу системи або адміністратор входить в систему;

2) дії – введення реєстраційних даних (для клієнта) або отримання облікових даних від адміністрації (для адміністратора). Підтвердження реєстрації. Вхід у систему за допомогою логіна та пароля;

3) кінець – успішна авторизація користувача з відповідними правами доступу;

б) процес управління розкладом сеансів:

1) початок – необхідність сформувати або оновити розклад показів;

2) дії – адміністратор створює нові сеанси, вказуючи фільм, дату, час, тривалість, кінозал. Можливість редагування або видалення існуючих сеансів;

3) кінець – оновлений розклад доступний для перегляду всіма користувачами;

в) процес бронювання/купівлі квитка клієнтом:

1) початок – клієнт бажає забронювати або купити квиток;

2) дії – клієнт переглядає доступний розклад сеансів, вибирає бажаний фільм та сеанс, обирає місця. Система реєструє бронювання/продаж квитка;

3) кінець – клієнт успішно забронював/придбав квиток на сеанс;

г) процес управління даними про клієнтів, фільми, сеанси та кінозали:

1) початок – необхідність додати, змінити або видалити інформацію про клієнтів, фільми, сеанси або кінозали;

2) дії – адміністратор виконує відповідні операції (CRUD - Create, Read, Update, Delete) з даними сутностей;

3) кінець – інформація в системі оновлена.

2 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ОБҐРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ

2.1 Вимоги до програмного забезпечення

2.1.1 Опис функцій програмного забезпечення

Програмне забезпечення повинно реалізовувати весь комплекс функцій, необхідних для ефективного адміністрування кінотеатру та взаємодії з його користувачами.

Детальний перелік та опис цих функцій, що включають управління даними про фільми, кінозали, сеанси, клієнтів та адміністративний персонал, а також функціонал управління розкладом та бронювання/продажу квитків, наведено у пункті 1.2.1 (Опис функціональних вимог).

Система забезпечить підтримку всіх визначених у цьому пункті операцій, таких як додавання, редагування, видалення та перегляд відповідної інформації, а також механізми авторизації та аутентифікації користувачів.

2.1.2 Вимоги до обробки даних

Програмне забезпечення повинно забезпечувати коректну, цілісну та безпечну обробку всіх даних, що циркулюють у системі. Це включає:

- збір та зберігання даних – системні компоненти повинні коректно збирати, валідувати та зберігати в базі даних всю необхідну інформацію, визначену в описі предметної області пункт 1.1.1 (Опис предметної області) та функціональних вимогах пункт 1.2.1 (Опис функціональних вимог). Це стосується персональних даних користувачів, інформації про фільми, сеанси, кінозали та бронювання квитків;

- валідація даних – вхідні дані мають проходити обов’язкову валідацію як на стороні клієнта, так і на сервері для забезпечення їх коректності, відповідності заданим форматам та уникнення внесення некоректної або шкідливої інформації;

- цілісність даних – система має підтримувати реляційну цілісність даних у базі даних, запобігаючи неконсистентності та забезпечуючи коректність зв’язків між сутностями. Мають бути реалізовані механізми транзакцій для операцій, що впливають на декілька пов’язаних сутностей;

- обробка бізнес-логіки – логіка обробки даних повинна строго відповідати бізнес-правилам кінотеатру, зокрема, щодо обмежень на уникнення часових конфліктів у розкладі сеансів, коректного управління місцями у залі та застосування прав доступу, визначених для різних ролей користувачів.

2.1.3 Вимоги до інтерфейсу користувача

Інтерфейс користувача розроблюваного веб-застосунку має бути інтуїтивно зрозумілим, функціональним та забезпечувати високий рівень досвіду користувача для всіх категорій користувачів. Основні вимоги до інтерфейсу включають:

- загальні принципи UI/UX – відповідність принципам зручності використання, інтуїтивності, послідовності та естетики, які є частиною нефункціональних вимог пункт 1.2.2 (Опис нефункціональних вимог);

- навігація – забезпечення чіткої та логічної системи навігації, що дозволяє користувачам легко переміщатися між розділами та функціональними блоками;

- візуалізація – ефективне та зручне представлення інформації, зокрема розкладу сеансів, списків фільмів та клієнтів;

- зворотній зв’язок – система повинна надавати чіткий, своєчасний та інформативний зворотний зв’язок на дії користувача, підтверджуючи успішні

операції або повідомляючи про помилки з рекомендаціями щодо їх виправлення;

- адаптивність – інтерфейс повинен бути повністю адаптивним (responsive design) для коректного та зручного відображення та функціонування на різних пристроях – від настільних комп’ютерів до мобільних телефонів.

2.2 Огляд сучасних середовищ розробки (IDE)

Сучасна розробка програмного забезпечення значною мірою спирається на використання інтегрованих середовищ розробки (Integrated Development Environment, IDE), які надають комплексний набір інструментів для написання, налагодження, тестування та розгортання коду. Вибір відповідного IDE є критично важливим для продуктивності розробника, ефективності проекту та якості кінцевого продукту. На сьогоднішній день існує широкий спектр IDE, призначених для різних мов програмування, платформ та типів додатків. До найбільш популярних та потужних відносяться:

- Microsoft Visual Studio – повноцінне інтегроване середовище розробки від Microsoft, призначене для створення широкого спектру додатків, включаючи настільні, мобільні, веб- та хмарні рішення. Особливо ефективне для розробки на платформі .NET (C#, VB.NET, F#) та C++. Пропонує потужні засоби для налагодження, тестування, профілювання, інтеграції з системами контролю версій (Git, Azure DevOps) та розширену підтримку баз даних;

- Visual Studio Code (VS Code) – легкий, але потужний текстовий редактор коду, розроблений Microsoft, який підтримує широкий спектр мов програмування та технологій завдяки великій екосистемі розширень. Хоча це не повноцінне IDE в традиційному розумінні, VS Code пропонує функціонал IDE (підсвічування синтаксису, автодоповнення, вбудований термінал,

засоби налагодження) і є надзвичайно гнучким для веб-розробки (HTML, CSS, JavaScript, TypeScript, Angular), Python, Node.js та багатьох інших;

- JetBrains серія IDE (IntelliJ IDEA, PyCharm, WebStorm, Rider) – компанія JetBrains пропонує лінійку спеціалізованих IDE, які відомі своїми інтелектуальними функціями, глибоким аналізом коду, розвиненими засобами рефакторингу та інструментами продуктивності. Наприклад, WebStorm оптимальний для Frontend-розробки (Angular, React, Vue.js), а Rider – для розробки на .NET;

- Eclipse – популярне відкрите IDE, переважно використовуване для розробки на Java, але також підтримує інші мови через плагіни;

- Xcode – інтегроване середовище розробки від Apple для створення додатків для macOS, iOS, watchOS та tvOS.

Вибір конкретних середовищ розробки для реалізації даного проекту, а також обґрунтування їх використання в комплексі з іншими технологіями, буде детально висвітлено в наступному розділі.

2.3 Обґрунтування вибору технологій та засобів розробки

2.3.1 Обґрунтування вибору Frontend (Angular)

Для розробки клієнтської частини веб-застосунку (Frontend) було обрано фреймворк Angular.

Переваги:

- компонентний підхід – Angular базується на компонентній архітектурі, що сприяє створенню модульного, повторно використовуваного та легко супроводжуваного коду. Це важливо для великих та середніх проектів;

- TypeScript – використання TypeScript (надмножина JavaScript з підтримкою статичної типізації) підвищує надійність коду, спрощує його читання та дозволяє виявляти помилки на етапі компіляції, що прискорює

розробку;

- багатий функціонал «з коробки» – Angular надає повний набір інструментів та бібліотек для створення складних односторінкових додатків (Single Page Applications, SPA), включаючи маршрутизацію, управління станом, HTTP-клієнт та інше;

- продуктивність – завдяки Ivy Renderer та оптимізації компіляції, Angular-додатки є високопродуктивними;

- велика спільнота та документація – широка підтримка розробників та якісна документація полегшують навчання та вирішення проблем.

Альтернативи та причини відмови: React (більша гнучкість, але менше «з коробки», що вимагає більшого досвіду у виборі бібліотек), Vue.js (більш простий для старту, але менш підходить для великих корпоративних проектів). Angular був обраний за його структурованість, надійність та готовність до масштабування.

2.3.2 Обґрунтування вибору Backend (C# WEB API)

Для розробки серверної частини веб-застосунку (Backend) було обрано технологію C# WEB API на платформі .NET.

Переваги:

- потужність та продуктивність .Net – Платформа .NET є високопродуктивною та надійною, забезпечуючи високу швидкість виконання коду та ефективну обробку запитів;

- мова C# – сучасна, об’єктно-орієнтована мова програмування з багатою бібліотекою класів, що дозволяє швидко та ефективно розробляти складну бізнес-логіку;

- RESTful API – ASP.NET Core Web API дозволяє легко створювати RESTful сервіси, які є стандартом для взаємодії між клієнтськими та серверними частинами веб-додатків;

- кросплатформність – сучасні версії .NET є кросплатформними, що

дозволяє розгортати Backend на різних операційних системах;

- безпека – вбудовані механізми безпеки для аутентифікації та авторизації (наприклад, JWT-токени), що критично важливо для веб-додатків;

- розширюваність – можливість легкого інтегрування з базами даних (Entity Framework Core), сторонніми сервісами та іншими компонентами.

Альтернативи та причини відмови: Node.js (JavaScript на сервері, але може бути менш продуктивним для складних обчислень), Python/Django/Flask (відмінно для швидкої розробки, але для великих корпоративних рішень C#/.NET часто є більш оптимальним вибором з точки зору продуктивності та супроводу), Java/Spring (потужне, але має вищий поріг входу та може бути більш ресурсоємним). C# WEB API був обраний за його високу продуктивність, зрілість екосистеми та ефективність для корпоративних рішень.

2.3.3 Обґрунтування вибору бази даних (MS SQL)

Для зберігання та управління даними веб-застосунку було обрано реляційну систему управління базами даних (СУБД) Microsoft SQL Server. Цей вибір ґрунтується на низці ключових переваг, що забезпечують високу ефективність, надійність та зручність розробки в рамках обраного стеку технологій.

Переваги:

- надійність та масштабованість – MS SQL Server є зрілою, надійною та високомасштабованою СУБД, що підходить для проектів будь-якого розміру, забезпечуючи цілісність та доступність даних;

- сумісність з .Net – ідеальна інтеграція з платформою .NET та Visual Studio, що значно спрощує розробку та взаємодію з базою даних через EF Core;

- потужні інструменти управління – наявність SQL Server Management

Studio (SSMS) надає широкі можливості для адміністрування, моніторингу, оптимізації та безпеки бази даних;

- безпека – розвинені функції безпеки для захисту даних та контролю доступу;
- підтримка транзакцій – надійна підтримка ACID-транзакцій.

Альтернативи та причини відмови: PostgreSQL (потужна Open Source СУБД, але може вимагати додаткових налаштувань для інтеграції з .NET), MySQL (популярна для веб, але MS SQL краще інтегрована в екосистему Microsoft), MS SQL обрано за його надійність, продуктивність та безшовну інтеграцію з обраним стеком технологій.

2.3.4 Обґрунтування вибору середовищ розробки

Для ефективної розробки проекту було обрано наступні інтегровані середовища розробки та інструменти, кожен з яких відіграє ключову роль у відповідному аспекті створення веб-застосунку:

- Microsoft Visual Studio: використовується як основне IDE для розробки серверної частини (Backend) на C# WEB API. Його вибір обумовлений глибокою інтеграцією з .NET, потужними засобами налагодження, профілювання та інструментами для роботи з базами даних;
- Visual Studio Code (VS Code): обрано як основне середовище для розробки клієнтської частини на Angular. Його перевагами є відмінна підтримка TypeScript, широка екосистема розширень, швидкість роботи та вбудований термінал, що спрощує взаємодію з Angular CLI;
- SQL Server Management Studio (SSMS): використовується як спеціалізований інструмент для адміністрування, проектування та взаємодії з базою даних MS SQL. SSMS є галузевим стандартом для роботи з SQL Server і надає розширені можливості, недоступні в інших IDE.

Комбінація цих потужних та спеціалізованих технологій та інструментів забезпечує синергію, дозволяючи розробникам зосередитись на

своїй частині проекту з найефективнішими засобами. Цей підхід дозволяє розробити високоефективний, надійний, масштабований та безпечний веб-застосунок для адміністрування кінотеатру, оптимізуючи кожен етап розробки.

2.3.5 Обґрунтування вибору допоміжного ПЗ та сервісів

Окрім основних технологій та середовищ розробки, успішна реалізація веб-застосунку вимагає використання допоміжного програмного забезпечення та сервісів, які оптимізують процеси розробки, управління залежностями та тестування:

а) менеджери пакетів:

1) npm (Node Package Manager) – для розробки клієнтської частини на Angular критично важливим є використання npm. npm дозволяє ефективно керувати бібліотеками та залежностями JavaScript/TypeScript, які необхідні для функціонування Angular-додатку. Це забезпечує легке встановлення, оновлення та управління всіма зовнішніми компонентами, такими як Angular CLI, сторонні UI-бібліотеки, та інші модулі;

2) NuGet: Для серверної частини на C# WEB API використовується NuGet – менеджер пакетів для платформи .NET. NuGet надає можливість легко додавати, оновлювати та видаляти .NET бібліотеки (пакети) у проекті. Це включає такі важливі пакети, як Entity Framework Core для роботи з базою даних, пакети для аутентифікації, та інші допоміжні бібліотеки, що значно прискорює розробку та забезпечує доступ до перевірених рішень;

б) інструмент для тестування API – для тестування та взаємодії з розробленим C# WEB API використовуватися вбудований Swagger UI. Цей інструменти дозволить надсилати HTTP-запити до API-методів, перевіряти відповіді та документувати функціонал API, що є важливим для ефективної розробки та інтеграції Frontend-частини.

Вибір цих допоміжних засобів обумовлений їхньою інтеграцією з обраними основними технологіями (Angular, C# WEB API, MS SQL), що забезпечує безперервний та ефективний процес розробки та тестування веб-застосунку.

3 АРХІТЕКТУРА ПРОГРАМНОЇ СИСТЕМИ

3.1 Загальна архітектура веб-застосунку

Веб-застосунок для адміністрування кінотеатру розробляється на основі класичної клієнт-серверної архітектури (Client-Server Architecture). Цей підхід дозволяє розділити логіку представлення даних та бізнес-логіку від логіки зберігання та обробки даних, забезпечуючи гнучкість, масштабованість та ефективне управління ресурсами.

Загальна архітектура системи складається з трьох основних логічних рівнів (рисунок 3.1):

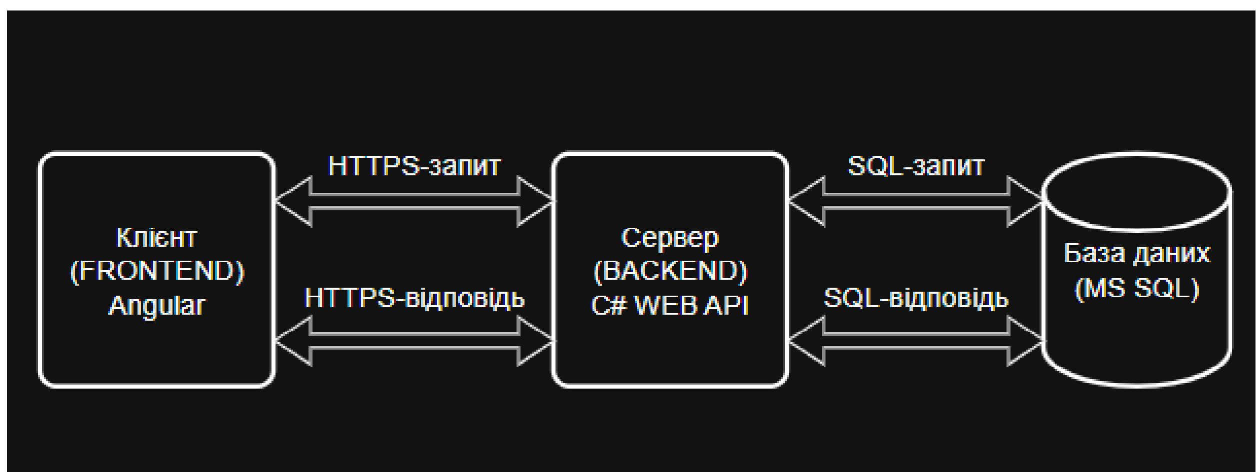

Рисунок 3.1 – Загальна клієнт-серверна архітектура веб-застосунку

а) клієнтський рівень (Frontend) – представляє собою інтерфейс користувача, з яким взаємодіють клієнти та адміністрація. Цей рівень реалізовано як односторінковий додаток (SPA) на базі фреймворку Angular. Він відповідає за:

1) відображення інформації (розклад фільмів, списки, форми);
2) обробку дій користувача (кліки, введення даних);
3) відправлення запитів до серверної частини (Backend);

4) динамічне оновлення інтерфейсу без повного перезавантаження сторінки;

б) серверний рівень (Backend / Business Logic Layer) – відповідає за обробку запитів від клієнтського рівня, виконання бізнес-логіки та взаємодію з базою даних. Цей рівень реалізовано за допомогою C# WEB API. Його ключові функції:

1) прийом HTTP-запитів від Frontend;

2) валідація вхідних даних;

3) реалізація бізнес-правил та логіки;

4) взаємодія з базою даних для отримання, збереження, оновлення та видалення інформації;

5) формування відповідей у форматі JSON для клієнтського рівня;

6) забезпечення безпеки (аутентифікація, авторизація);

в) рівень даних (Database Layer) – відповідає за постійне зберігання всіх даних системи. Для цього використовується реляційна база даних MS SQL Server. Цей рівень забезпечує:

1) надійне зберігання інформації про клієнтів, фільми, сеанси, кінозали та адміністративний персонал;

2) забезпечення цілісності та узгодженості даних;

3) підтримка транзакцій для атомарних операцій;

4) надання механізмів для ефективного пошуку та вибірки даних.

Взаємодія між рівнями відбувається за допомогою стандартизованих протоколів. Клієнтський рівень обмінюється даними з серверним рівнем через RESTful API, використовуючи протокол HTTPS та формат передачі даних JSON. Серверний рівень взаємодіє з рівнем даних за допомогою технологій доступу до баз даних, таких як Entity Framework.

Ця архітектура забезпечує високу гнучкість, оскільки Frontend і Backend можуть розроблятися та розгортатися незалежно. Вона також сприяє масштабованості та дозволяє легко інтегрувати нові функціональності або змінювати існуючі компоненти без впливу на всю систему.

3.2 Проектування структури бази даних

Проектування структури бази даних є фундаментальним етапом, що визначає, як дані будуть зберігатися та взаємодіяти у системі. Для веб-застосунку для адміністрування кінотеатру використовується реляційна база даних MS SQL Server. Її структура базується на наступних ключових сутностях та їх взаємозв’язках, відображених у ER-моделі (Entity-Relationship Model), яка представлена на рисунку 3.2.

Основою для управління користувачами, їх ролями та аутентифікацією є інтеграція з системою ASP.NET Core Identity. Це забезпечує надійне та стандартизоване рішення для роботи з користувацькими обліковими записами та їх безпекою, включаючи розмежування доступу для ролей «Клієнт» та «Адмін».

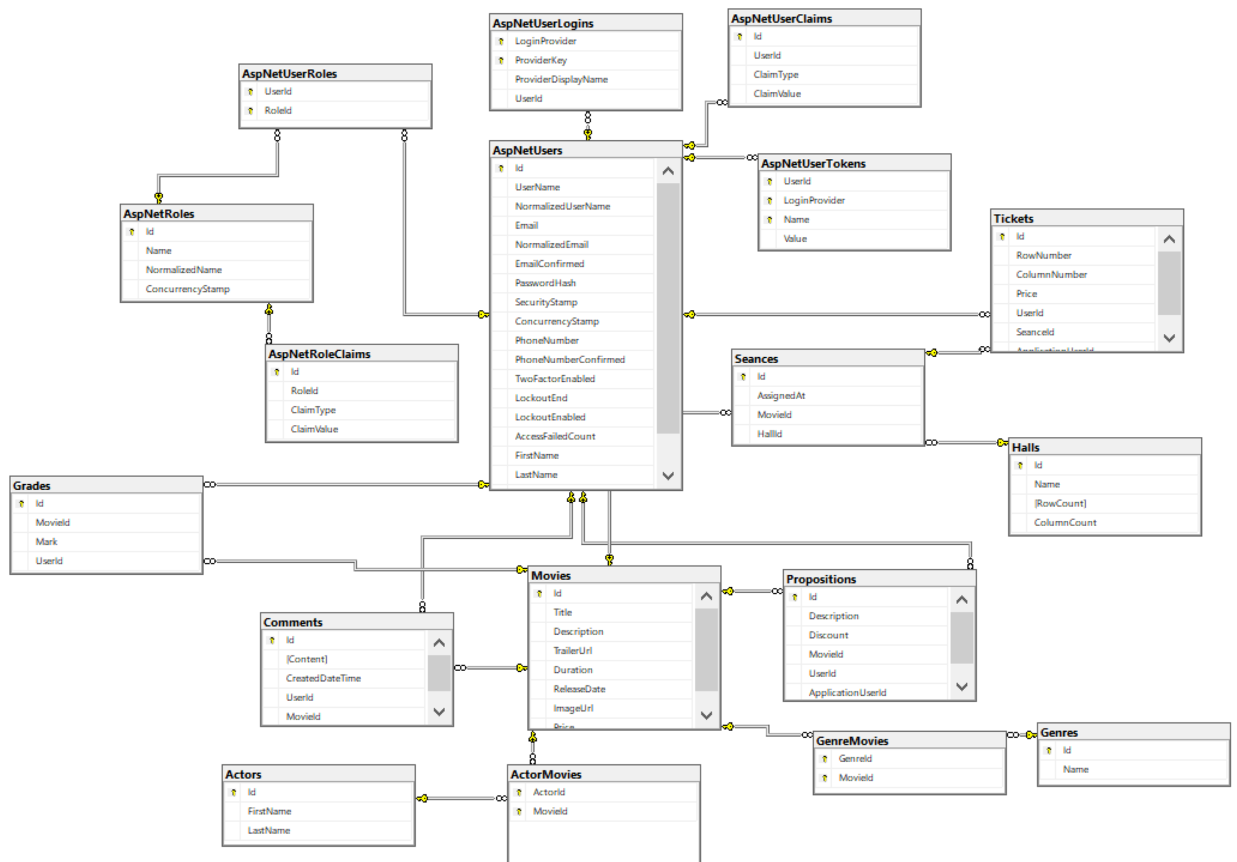

Рисунок 3.2 – Схема ER-моделі бази даних

Основні сутності та їх взаємозв’язки:

а) AspNetUsers – центральна таблиця для зберігання базової інформації про всіх користувачів системи. Включає такі поля, як Id (PRIMARY KEY), FirstName, LastName, Email, PhoneNumber, UserName, а також поля для безпеки аутентифікації (PasswordHash, SecurityStamp тощо). Зв’язки:

1) має зв’язок «один-до-багатьох» (1:N) з таблицями Tickets, Propositions, Grades та Comments, оскільки кожен користувач може мати багато квитків, пропозицій, оцінок та коментарів;

2) має зв’язок «багато-до-багатьох» (N:M) з AspNetRoles через проміжну таблицю AspNetUserRoles, що дозволяє одному користувачеві мати декілька ролей;

3) пов’язана з іншими Identity-таблицями (AspNetUserClaims, AspNetUserLogins, AspNetUserTokens) для розширених функцій аутентифікації та авторизації;

б) AspNetRoles – таблиця для визначення ролей користувачів у системі. Включає поля Id та Name;

в) Actors – зберігає інформацію про акторів. Поля: Id (PRIMARY KEY), FirstName, LastName. Зв’язки: має зв'язок «багато-до-багатьох» (N:M) з Movies через проміжну таблицю ActorMovies.г) Administrations – зберігає специфічну інформацію для користувачів з роллю «Admin». Має Id (PRIMARY KEY, зовнішній ключ до AspNetUsers.Id) та Status;

г) ActorMovies – проміжна таблиця для реалізації зв'язку «багато-до-багатьох» між Actors та Movies, що вказує, які актори знімались у яких фільмах. Поля: ActorId (PRIMARY KEY, FOREIGN KEY до Actors.Id), MovieId (PRIMARY KEY, FOREIGN KEY до Movies.Id);

г) Comments – зберігає коментарі користувачів до фільмів. Поля: Id (PRIMARY KEY), Content (текст коментаря), CreatedDateTime (дата та час створення), UserId (FOREIGN KEY до AspNetUsers.Id), MovieId (FOREIGN KEY до Movies.Id). Зв’язки: «багато-до-одного» (N:1) з ApplicationUser (коментар належить одному користувачу) та «багато-до-одного» (N:1) з

Movie;

д) Genres – зберігає інформацію про жанри фільмів. Поля: Id (PRIMARY KEY), Name (назва жанру, унікальна). Зв’язки: має зв'язок «багато-до-багатьох» (N:M) з Movies через проміжну таблицю GenreMovies;

е) GenreMovies – проміжна таблиця для реалізації зв'язку «багато-до-багатьох» між Genres та Movies, що вказує, до яких жанрів належить кожен фільм. Поля: GenreId (PRIMARY KEY, FOREIGN KEY до Genres.Id), MovieId (PRIMARY KEY, FOREIGN KEY до Movies.Id);

є) Grades – зберігає оцінки фільмів користувачами. Поля: Id (PRIMARY KEY), MovieId (FOREIGN KEY до Movies.Id), Mark (оцінка), UserId (FOREIGN KEY до AspNetUsers.Id). Зв’язки: «багато-до-одного» (N:1) з Movie (оцінка до одного фільму) та «багато-до-одного» (N:1) з ApplicationUser;

ж) Halls – інформація про кінозали. Поля: Id (PRIMARY KEY), Name (назва залу), RowCount (кількість рядів), ColumnCount (кількість місць у ряду). Зв’язки: «один-до-багатьох» (1:N) з Seances, оскільки в одному залі може проходити багато сеансів;

з) Movies – інформація про фільми. Поля: Id (PRIMARY KEY), Title, ImageUrl, Description, TrailerUrl, Duration, ReleaseDate, Price (базова ціна). Зв’язки:

1) має зв'язок «один-до-багатьох» (1:N) з Seances1;

2) має зв'язок «один-до-багатьох» (1:N) з Propositions, Grades та Comments;

3) має зв'язок «багато-до-багатьох» (N:M) з Actors через ActorMovies;

4) має зв'язок «багато-до-багатьох» (N:M) з Genres через GenreMovies;

и) Propositions – інформація про спеціальні пропозиції/знижки для фільмів. Поля: Id (PRIMARY KEY), Description, Discount, MovieId (FOREIGN KEY до Movies.Id), UserId (FOREIGN KEY до AspNetUsers.Id). Зв’язки:

«багато-до-одного» (N:1) з Movie та «багато-до-одного» (N:1) з ApplicationUser;

і) Seances – інформація про заплановані покази фільмів. Поля: Id (PRIMARY KEY), AssignedAt (дата та час сеансу), MovieId (FOREIGN KEY до Movies.Id), HallId (FOREIGN KEY до Halls.Id). Зв’язки:

1) «багато-до-одного» (N:1) з Movie;

2) «багато-до-одного» (N:1) з Hall;

3) «один-до-багатьох» (1:N) з Tickets;

ї) Tickets – інформація про придбані/заброньовані квитки. Поля: Id (PRIMARY KEY), RowNumber, ColumnNumber (місце в залі), Price (фактична ціна квитка), UserId (FOREIGN KEY до AspNetUsers.Id), SeanceId (FOREIGN KEY до Seances.Id). Зв’язки: «багато-до-одного» (N:1) з ApplicationUser (квиток належить одному користувачу) та «багато-до-одного» (N:1) з Seance;

й) інші таблиці ASP.NET Identity:

1) AspNetUserRoles – зв’язує користувачів з їхніми ролями;

2) AspNetRoleClaims, AspNetUserClaims, AspNetUserLogins, AspNetUserTokens: Стандартні таблиці ASP.NET Identity, що використовуються для зберігання клеймів, логінів сторонніх провайдерів та токенів користувачів.

3.3 Проектування архітектури Backend

3.3.1 Структура C# WEB API та основні компоненти

Архітектура Backend дотримується принципів багатошарової архітектури (N-Tier Architecture), представленої на рисунку 3.3. Цей підхід дозволяє застосункам встановлювати чіткі обмеження на те, які шари можуть взаємодіяти з іншими шарами, що є фундаментальним для досягнення ефективної організації програмного коду. Основна ідея багатошарової архітектури полягає у поділі функціональності системи на логічно

відокремлені, ієрархічно розташовані шари, де кожен шар має чітко визначену відповідальність та взаємодіє лише з сусідніми шарами.

Завдяки такій структурі досягається інфкапсуляція, коли внутрішня реалізація кожного шару прихована від інших. Це мінімізує вплив змін в одному шарі на інші компоненти системи, оскільки лише ті шари, які безпосередньо працюють зі зміненим шаром, мають бути адаптовані. Таким чином, замість того, щоб одна зміна могла вплинути на весь додаток, вплив обмежується, що значно спрощує процеси розробки, тестування, налагодження та подальшого супроводу. Це також сприяє розділенню відповідальності (Separation of Concerns), роблячи кожен компонент системи більш сфокусованим на виконанні своїх конкретних завдань. Перевагами такого підходу є підвищена масштабованість, оскільки шари можуть бути розгорнуті на окремих серверах; краща тестувальність, оскільки кожен шар можна тестувати ізольовано; та гнучкість у виборі технологій для кожного шару.

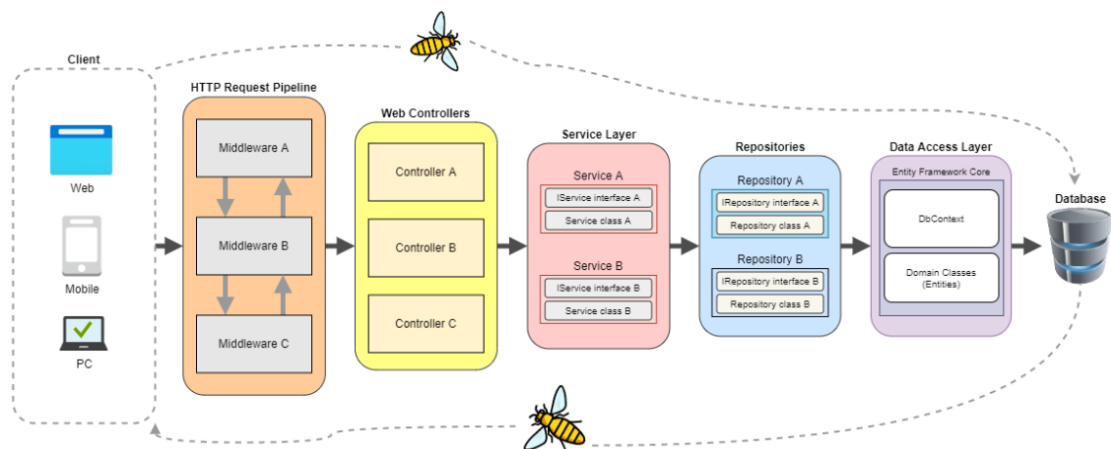

Рисунок 3.3 – Архітектура Backend-компонентів

Основні шари та компоненти Backend:

а) HTTP Pipeline – це конвеєр, через який проходять запити до того, як вони досягнуть контролерів. У застосунках .NET Core запити проходять

через цей конвеєр, що дозволяє реалізовувати скрізну функціональність;

б) Middleware (Проміжне програмне забезпечення):

1) Middleware в ASP.NET Core контролює, як застосунок реагує на HTTP-запити. Воно може також контролювати вигляд застосунку у випадку помилки і є ключовим елементом у процесі аутентифікації та авторизації користувача для виконання певних дій. Middleware виконується послідовно в HTTP-конвеєрі;

2) приклади Middleware: аутентифікація (перевірка JWT-токенів), авторизація (перевірка прав доступу), обробка винятків, логування, CORS-налаштування;

в) Presentation Layer (API Controllers):

1) Web API Контролери (Web API Controllers): схожі на контролери ASP.NET MVC. Вони обробляють вхідні HTTP-запити та надсилають відповіді назад до того, хто їх викликав. Контролер – це клас, назва якого має закінчуватися на «Controller». Всі публічні методи контролера називаються методами дії (action methods). Контролери відповідають за маршрутизацію запитів до відповідних методів та формування HTTP-відповідей, делегуючи складну бізнес-логіку Сервісному шару;

2) приклади: TicketController, HallController, AuthController, SeanceController, MovieController;

г) Service Layer (Business Logic Layer):

1) сервісний шар (Service Layer): це додатковий шар у Web API застосунку, який є проміжним між контролерами та репозиторіями. Сервісний шар містить бізнес-логіку і може також містити логіку валідації. Сервіси інкапсулюють правила роботи з даними, перевіряють коректність операцій, реалізують складні алгоритми та координують взаємодію між репозиторіями. Вони не мають прямого доступу до бази даних, а взаємодіють з нею через репозиторії.

2) приклади: MovieService, TicketService, HallService, AuthService;

3) відповідальність: валідація даних, обробка прав доступу, логіка

запису/відміни, агрегація даних з кількох репозиторіїв;

ґ) Data Access Layer (Repository Layer):

1) репозиторії (Repositories) – це класи, які приховують логіку, необхідну для зберігання або отримання даних. Кожен репозиторій відповідає за CRUD-операції (Create, Read, Update, Delete) для певної сутності. Репозиторії використовують ORM-інструмент – Entity Framework Core – для роботи з базою даних об’єктно-орієнтованим способом, мінімізуючи написання SQL-запитів вручну. Основне призначення цього шару – ізолювати логіку доступу до даних;

2) приклади: IMovieRepository, MovieRepository;

3) Entity Framework (EF) – це фреймворк об’єктно-реляційного відображення (ORM). Він дозволяє працювати з реляційними даними як з об'єктами, усуваючи більшість коду доступу до даних, який зазвичай доводиться писати;

4) DbContext – об’єкт Entity Framework Core, що представляє сесію взаємодії з базою даних і дозволяє запитувати та зберігати об’єкти;

д) Models/DTOs (Data Transfer Objects):

1) моделі сутностей (Entity Models) – представляють структуру даних у базі даних.

2) DTOs (Data Transfer Objects) – об’єкти, що використовуються для передачі даних між шарами (між Frontend та Backend, або між Контролерами та Сервісами). DTOs дозволяють контролювати, які дані передаються, та уникати витоку зайвої інформації.

Впровадження Dependency Injection. DI є важливим аспектом розробки сучасних додатків, оскільки він дозволяє знизити залежності між компонентами та підвищити їх тестованість. У бекенді DI використовується для впровадження сервісів у контролери та інші компоненти. Це забезпечує більш гнучку архітектуру та полегшує заміну або оновлення залежностей.

Реєстрація сервісів здійснюється у файлі Program.cs, де використовується метод розширення для налаштування залежностей.

Основна конфігурація знаходиться у класі Configuration.cs у просторі імен GymManagement.Business. Приклад додавання залежностей у DI контейнер реалізовано на лістингу 3.1.

Лістинг 3.1 – Статичний клас для додавання залежностей

```
     public static class Configuration
     {
         public  static  void  Configure(this  IServiceCollection
serviceCollection, string connectionString)
         {
             Data.Configuration.Configure(serviceCollection,
connectionString);

serviceCollection.AddAutoMapper(typeof(MapperProfile));
             serviceCollection.AddTransient<IActorService,
ActorService>()
                 .AddTransient<ICommentService, CommentService>()
                 .AddTransient<IGenreService, GenreService>()
                 .AddTransient<IGradeService, GradeService>()
                 .AddTransient<IHallService, HallService>()
                 .AddTransient<IMovieService, MovieService>()
                 .AddTransient<IPropositionService,
PropositionService>()
                 .AddTransient<ISeanceService, SeanceService>()
                 .AddTransient<ITicketService, TicketService>()
                 .AddTransient<IAuthenticationService,
AuthenticationService>()
                 .AddTransient<ITokenService, TokenService>();
         }
     }
```

У цьому лістингу:

- Data.Configuration.Configure: викликається для налаштування залежностей рівня даних, таких як репозиторії та контекст бази даних;

- AddAutoMapper: додає AutoMapper для автоматизації процесу мапінгу моделей і DTO;

- AddTransient: використовується для реєстрації сервісів із короткочасним життєвим циклом. (це означає, що новий екземпляр сервісу створюється кожного разу, коли він запитується);

- TokenService та AuthenticationService: сервіси, які забезпечують функціональність аутентифікації та створення токенів доступу.

Завдяки використанню DI забезпечується розділення відповідальностей

у системі, що підвищує гнучкість, знижує залежності та спрощує тестування компонентів. Це підхід, який дозволяє створювати більш масштабовані та підтримувані додатки.

3.3.2 Аутентифікація та авторизація

У будь-якому веб-застосунку, що передбачає взаємодію з користувачами, важливими складовими є аутентифікація (перевірка особи користувача) та авторизація (визначення прав доступу до певних ресурсів). Для реалізації безпечного керування доступом у даному проєкті використано технології ASP.NET Core Identity у поєднанні з JWT Token (JSON Web Token).

ASP.NET Core Identity – це вбудована система аутентифікації та управління користувачами в ASP.NET Core. Вона дозволяє:

- реєструвати та входити в систему;
- працювати з ролями користувачів (адміністратор, клієнт);
- захищати API-методи на основі ролей;
- хешувати та зберігати паролі безпечно;
- працювати з email-підтвердженнями, блокуванням, відновленням паролю тощо.

У проєкті було створено власну модель користувача, яка розширює стандартний IdentityUser для додавання додаткових полів, інтегруючи її зі структурою бази даних (див. AspNetUsers у 3.2. Проектування структури бази даних).

JWT-токени використовуються для захисту API і забезпечення безпечної комунікації між клієнтською та серверною частинами додатку. Після успішної автентифікації система генерує токен, який містить усю необхідну інформацію про користувача, включаючи його ідентифікатор, роль і додаткові клейми. Цей токен підписується за допомогою секретного ключа і має обмежений термін дії, що значно підвищує безпеку системи. Клієнтський додаток отримує цей токен і використовує його для подальших запитів до

API, додаючи його в заголовок `Authorization`.

На Рисунку 3.4 представлено схему роботи механізму аутентифікації з використанням Access та Refresh токенів.

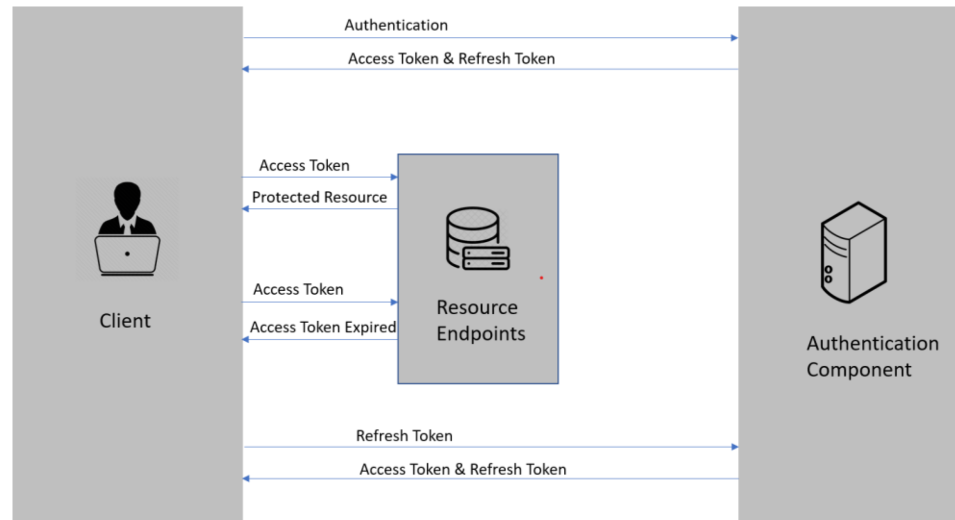

Рисунок 3.4 – Схема роботи JWT аутентифікації

Авторизація в системі реалізована на декількох рівнях. На рівні контролерів використовуються атрибути `Authorize` з вказівкою необхідних ролей, що дозволяє обмежити доступ до певних методів API. Наприклад, функціонал управління клієнтами доступний лише для адміністраторів, тоді як звичайні клієнти можуть лише переглядати інформацію про себе. Додатково реалізована політика ресурсів.

Інтеграція Identity з JWT дозволяє створити сучасну, безпечну і зручну систему управління доступом, яка повністю відповідає вимогам нашого проєкту. Цей підхід забезпечує не лише надійний захист даних, але й гнучкість у керуванні правами користувачів, що особливо важливо для системи з такою складною структурою ролей і дозволів, як у нашому веб-додатку для адміністрування кінотеатру.

3.4 Проектування архітектури Frontend

Клієнтська частина веб-застосунку реалізується як Односторінковий

Додаток (Single Page Application, SPA) за допомогою фреймворку Angular. Вибір архітектури SPA обґрунтований її перевагами в забезпеченні динамічного та інтерактивного користувацького досвіду, що максимально наближений до десктопних застосунків. SPA дозволяє завантажувати всі необхідні ресурси (HTML, CSS, JavaScript) лише один раз при першому зверненні, а потім динамічно оновлювати контент сторінки без її повного перезавантаження. Це значно підвищує швидкість відгуку інтерфейсу та покращує взаємодію з користувачем. Архітектура Angular-додатку базується на сучасному компонентному підході, який є фундаментальним для розробки складних інтерфейсів. Кожен елемент інтерфейсу користувача інкапсулюється у власний компонент, що має свою логіку, шаблон та стилі. Це сприяє модульності та розділенню відповідальності, роблячи код більш зрозумілим, керованим та легким для тестування. Важливою особливістю реалізації Frontend є використання Standalone Components – сучасної можливості Angular, яка дозволяє визначати компоненти незалежно від Angular Modules.

Такий підхід забезпечує модульність, можливість повторного використання коду та легкість супроводу, підвищуючи загальну ефективність розробки. Загальна структура Frontend-додатку, яка відображає ці архітектурні принципи та організацію компонентів, представлена на рисунку 3.5. Архітектура Angular-додатку.

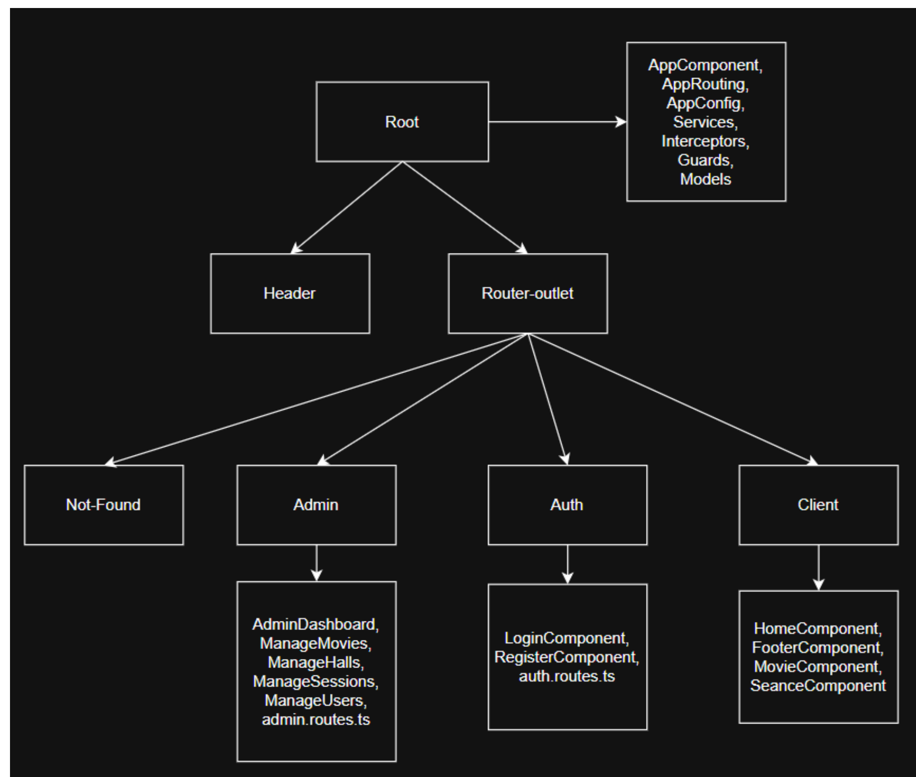

Рисунок 3.5 – Архітектура Angular-додатку

Згідно з представленою архітектурою, додаток організовано наступним чином:

а) кореневий рівень (Root):

1) Root – представляє основу додатку, з якої ініціюється робота системи;

2) AppComponent – кореневий компонент додатку, який є точкою входу для відображення UI;

3) AppRouting – визначає основні маршрути для навігації по додатку;

4) AppConfig – файл конфігурації застосунку;

5) Services, Interceptors, Guards, Models – ці елементи є основними службами, які надають функціональність, яка може бути використана всім додатком, включаючи взаємодію з Backend, захист маршрутів та структуру даних;

б) загальні компоненти та маршрутизація:

1) Header – компонент, який відповідає за відображення верхньої частини інтерфейсу, що є загальною для всього додатку;

2) Router-outlet – спеціальний елемент Angular, в якому динамічно відображаються компоненти, відповідні поточному маршруту. Це центральний елемент для навігації в SPA;

в) розділення за функціональними областями (Feature Areas) – ці блоки підключаються через Router-outlet та представляють окремі функціональні модулі або сторінки додатку:

1) Not-Found – компонент для відображення сторінки «404 Not Found» у випадку, якщо запитуваний маршрут не знайдено;

2) Admin – функціональний блок для адміністраторів, що включає: AdminDashboard, ManageMovies, ManageHalls, ManageSessions, ManageUsers, admin.routes.ts (файл маршрутів);

3) Auth – функціональний блок для аутентифікації, що включає: LoginComponent, RegisterComponent, auth.routes.ts (файл маршрутів для функціоналу аутентифікації);

4) Client – функціональний блок для клієнтів, що включає: MovieComponent, SeanceComponent, HomeComponent, FooterComponent;

Ключові принципи та компоненти Frontend-архітектури:

- компоненти (Components) – є основними будівельними блоками UI. Кожен компонент контролює частину екрану (view) та містить HTML-шаблон, CSS-стилі та клас TypeScript, який містить логіку компонента, властивості, методи та взаємодію з даними;

- сервіси (Services) – містять бізнес-логіку або логіку для отримання даних, яку можна використовувати в різних компонентах. Сервіси є синглтонами і надаються за допомогою механізму Dependency Injection. Вони відповідають за взаємодію з Backend API через HttpClient;

- маршрутизація (Routing) – використовується для навігації між різними «сторінками» або видами в SPA. Angular Router дозволяє визначати маршрути на основі URL, що ведуть до конкретних компонентів. Файли `app.routes.ts` та специфічні `*.routes.ts` для функціональних модулів (наприклад, `admin.routes.ts`) визначають ці маршрути, дозволяючи також

реалізувати Lazy Loading (відкладене завантаження) для оптимізації продуктивності, коли функціональні області завантажуються лише при потребі;

- охоронці маршрутів (Route Guards) – розташовані у блоці Core, забезпечують контроль доступу до певних маршрутів на основі ролі користувача або стану аутентифікації, запобігаючи несанкціонованому доступу до захищених частин додатку;

- HTTP Interceptors – знаходяться у блоці Core, використовуються для перехоплення HTTP-запитів та відповідей. Це ідеальне місце для централізованої обробки помилок, додавання заголовків або логування.

Ця архітектура, заснована на Standalone Components та чіткому розподілі за функціональними областями, забезпечує високу організованість коду, полегшує розробку, тестування та масштабування клієнтської частини застосунку.

4 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Розробка програмних компонентів Frontend

4.1.1 Опис інтерфейсу взаємодії ПЗ з користувачем

Інтерфейс користувача (UI) спроектований з метою забезпечення інтуїтивно зрозумілої навігації, привабливого дизайну та зручності використання для всіх категорій користувачів: клієнтів та адміністрації.

Дизайн та зовнішній вигляд:

- використано сучасний та чистий дизайн, що відповідає естетиці кіно-індустрії;

- забезпечено адаптивний дизайн (Responsive Web Design), що дозволяє коректно відображати та взаємодіяти з інтерфейсом на різних пристроях – від мобільних телефонів до широкоформатних моніторів. Це досягається за допомогою гнучких сіток, медіа-запитів CSS та Angular Material;

- кольорова палітра та типографіка підібрані для створення приємного візуального сприйняття та читабельності.

Навігація:

- навігація реалізована за допомогою Angular Router (рисунок 4.1) , що забезпечує плавні переходи між «сторінками» без повного перезавантаження браузера;

- верхнє навігаційне меню (HeaderComponent у client/header/) забезпечує доступ до основних розділів (головна, сеанси, фільми, вхід/реєстрація);

- навігація адаптується до ролі користувача: після авторизації з’являються додаткові пункти меню, доступні тільки для відповідної ролі.

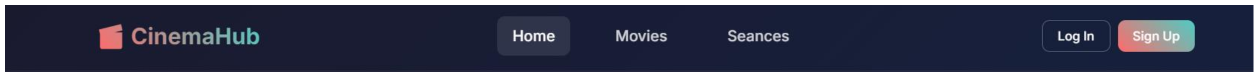

Рисунок 4.1 – Навігаційне меню

Форми:

- всі форми (реєстрація, вхід, редагування даних) розроблені з урахуванням зручності введення даних;
- використовуються стандартні Angular Forms (Reactive Forms) для управління станом форми, валідації та надсилання даних;
- клієнтська валідація даних реалізована для негайного зворотного зв’язку користувачеві щодо коректності введених даних, що зменшує кількість помилок та запитів до сервера;
- повідомлення про помилки чіткі та зрозумілі, з’являються поруч із відповідними полями введення.

4.1.2 Реалізація функціоналу для клієнтів

Функціонал для клієнтів реалізовано в блоці `client/` та включає наступні можливості, спрямовані на зручну взаємодію з кінотеатром:

а) редагування власної інформації:

1) клієнти мають можливість переглядати та оновлювати свої персональні дані, такі як ім'я, прізвище, електронна пошта, та за потреби, змінювати пароль;

2) реалізовано інтерактивну форму редагування в особистому кабінеті, яка завантажує поточні дані клієнта з Backend та дозволяє їх змінити, з подальшим збереженням оновленої інформації;

б) перегляд розкладу кіносеансів:

1) клієнти можуть переглядати актуальний розклад усіх доступних кіносеансів;

2) розклад відображається в зручному та інтуїтивно зрозумілому

форматі, з можливістю фільтрації;

3) для кожного сеансу відображається детальна інформація: назва фільму, кінозал, дата та час початку, тривалість, ціна квитка та доступні місця;

4) дані розкладу динамічно отримуються з Backend, забезпечуючи актуальність інформації;

в) бронювання та купівля квитків:

1) після вибору бажаного фільму та сеансу, клієнти можуть переглянути інтерактивну схему кінозалу з відображенням вільних та зайнятих місць;

2) користувачі можуть обирати одне або декілька бажаних місць у кінозалі;

3) реалізована можливість бронювання квитків або їх негайної купівлі онлайн;

4) для бронювання/купівлі відправляється POST-запит на Backend з усіма необхідними даними (ідентифікатори клієнта, сеансу, номери обраних місць, ціна);

5) після успішного бронювання або оплати, клієнт отримує підтвердження, а інформація про його квитки відображається у його персональному кабінеті в розділі «My tickits».

4.1.3 Реалізація функціоналу для адміністрації

Функціонал для адміністрації є найбільш розширеним і реалізований у блоці admin/ з відповідними підкомпонентами, надаючи повний контроль над операційною діяльністю кінотеатру.

Управління фільмами (manage-movies/):

- дозволяє адміністраторам додавати нові фільми до бази даних, включаючи всі необхідні атрибути: назву, детальний опис, тривалість, посилання на зображення (постер), посилання на трейлер, дату виходу,

базову ціну;

- надається можливість редагувати інформацію про існуючі фільми, а також видаляти їх з прокату;

- інтерфейс включає табличний перегляд усіх фільмів, форми для додавання/редагування фільмів, а також механізми для прив’язки фільмів до відповідних жанрів та акторів, які в них знімались;

- взаємодія з Backend відбувається через API з використанням POST-запитів для створення, PUT-запитів для оновлення та DELETE-запитів для видалення даних.

Управління кінозалами (manage-halls/):

- надає адміністрації можливість додавати нові кінозали, редагувати їхні назви, а також задавати кількість рядів (RowCount) та кількість місць у кожному ряду (ColumnCount), що дозволяє точно моделювати схему залу для бронювання квитків;

- аналогічно іншим управлінським модулям, реалізовано табличний перегляд усіх кінозалів та форми для виконання CRUD-операцій;

- взаємодія з Backend здійснюється через API: POST для створення, PUT для оновлення, DELETE для видалення;

Управління сеансами (manage-seances/):

- дозволяє адміністраторам створювати нові кіносеанси, вказуючи фільм, який демонструватиметься, кінозал, дату та точний час проведення сеансу;

- надається можливість редагувати всі деталі існуючих сеансів, а також скасовувати або видаляти їх з розкладу;

- реалізовано можливість перегляду списку проданих та заброньованих квитків для кожного конкретного сеансу, що дозволяє контролювати заповнюваність залу;

- взаємодія з Backend відбувається через API: POST для створення, PUT для оновлення, DELETE для видалення.

Управління користувачами (manage-users/):

- адміністрація має повний доступ до інформації про всіх користувачів системи, включаючи клієнтів та інших адміністраторів. Це дозволяє переглядати, додавати нових користувачів, редагувати їхні облікові дані (ім'я, прізвище, email, ролі) та, за потреби, деактивувати або видаляти профілі;
- цей модуль є центральною базою даних користувачів та відіграє ключову роль у функціоналі CRM, дозволяючи адміністраторам ефективно керувати даними користувачів, вирішувати питання з реєстрацією, оновленням контактної інформації. Можливість ручного додавання нових користувачів є корисною для офлайн-реєстрацій або для швидкого надання доступу персоналу;
- взаємодія з Backend реалізована через API: POST, PUT, DELETE.

Управління пропозиціями (manage-propositions/):

- адміністратори можуть створювати, редагувати та видаляти спеціальні пропозиції та знижки, які можуть бути застосовані до певних фільмів або спрямовані на конкретних користувачів;
- це дозволяє кінотеатру запускати маркетингові кампанії та програми лояльності, керуючи їхньою актуальністю та умовами;
- взаємодія з Backend відбувається через API: POST для створення, PUT для оновлення, DELETE для видалення.

4.2 Розробка програмних компонентів Backend

4.2.1 Реалізація бізнес-логіки програмних компонентів

Бізнес-логіка реалізується переважно у Сервісному шарі (Service Layer), який відокремлений від контролерів та репозиторіїв. Це забезпечує чистий код, легкість тестування та супроводу.

Контролери є точками входу API, приймають HTTP-запити, виконують базову валідацію вхідних DTOs та делегують подальшу обробку відповідним сервісам. Вони відповідають за повернення стандартизованих HTTP-

відповідей. Сервіси – кожен сервіс інкапсулює бізнес-правила для певної предметної області.

Приклади бізнес-логіки в сервісах:

- AuthService – обробка реєстрації, входу, генерації та валідації JWT-токенів, управління сесіями;
- ClientService, MovieService, SessionService: валідація даних при створенні/оновленні профілів користувачів, перевірка унікальності email/телефону при реєстрації, логіка прив'язки користувача до відповідної ролі (клієнт, адміністратор) під час реєстрації або зміни ролі;
- валідація – реалізована на двох рівнях: Data Annotations для простих правил валідації на DTOs, бізнес-валідація в сервісах: Для складних, контекстно-залежних правил, які вимагають доступу до бази даних або інших сервісів.

4.2.2 Розробка компонентів для взаємодії з базою даних

Взаємодія з базою даних MS SQL Server реалізується через Data Access Layer, який використовує патерн «Репозиторій» (Repository Pattern) та Entity Framework Core (EF Core) як ORM (Object-Relational Mapper).

Кожен репозиторій відповідає за CRUD-операції для однієї конкретної сутності. Вони абстрагують логіку доступу до даних від бізнес-логіки, забезпечуючи чистий інтерфейс для сервісів.

Entity Framework Core (EF Core) – це потужний ORM, який дозволяє розробникам працювати з реляційними даними (таблицями, записами) як з об'єктами C# (моделями сутностей). `DbContext` є основним компонентом EF Core, що представляє сесію взаємодії з базою даних. Він відповідає за відстеження змін у об'єктах, виконання запитів та збереження змін. LINQ (Language Integrated Query) – активно використовується для написання запитів до бази даних. LINQ-запити, написані на C# з використанням синтаксису, подібного до звичайних колекцій, трансліруються EF Core у

відповідні оптимізовані SQL-запити, що виконуються на MS SQL Server. Це значно зменшує необхідність писати сирий SQL-код, підвищує продуктивність розробника та забезпечує типобезпеку запитів. EF Core підтримує транзакції, що дозволяє забезпечити атомарність складних операцій, які зачіпають декілька таблиць, гарантуючи цілісність даних.

4.2.3 Реалізація механізмів авторизації та аутентифікації користувачів

Детальна реалізація механізмів аутентифікації та авторизації винесена в окремий підрозділ 3.3.2. Аутентифікація та Авторизація. Проте, на етапі реалізації це включає:

- ASP.NET Core Identity – інтеграція стандартних можливостей Identity для управління користувачами, ролями, хешуванням паролів та пов'язаними таблицями бази даних (AspNetUsers, AspNetRoles тощо);
- JWT – реалізація генерації, підписання та валідації Access та Refresh токенів. Це включає налаштування JWT Bearer Authentication у HTTP-конвеєрі Backend для перехоплення та перевірки токенів у кожному запиті;
- авторизація на основі ролей і використання атрибуту `[Authorize(Roles = "Admin")]` на контролерах та методах дій для обмеження доступу до ресурсів на основі ролі користувача, отриманої з JWT-токену;
- політики авторизації для складніших сценаріїв авторизації, що виходять за рамки простої перевірки ролей, реалізовано політики авторизації, які дозволяють виконувати довільну логіку (наприклад, «чи користувач є власником ресурсу?»).

4.3 Опис використаних алгоритмів та структур даних

У контексті даного веб-застосунку, більшість складної логіки обробки даних делегується базі даних та фреймворкам, які оптимізовані для цих

завдань. Проте, можуть бути використані наступні загальні алгоритми та структури даних:

а) структури даних:

1) List/Array (Списки/Масиви) для тимчасового зберігання колекцій об’єктів під час обробки запитів як на Backend, так і на Frontend. Наприклад, списки фільмів, сеансів, квитків, користувачів, акторів або жанрів тимчасово зберігаються та маніпулюються в пам'яті для подальшої обробки або відображення;

2) Dictionary/HashTable (Словники/Хеш-таблиці) які використовуються для швидкого пошуку об’єктів за унікальним ключем або для агрегації даних. Наприклад, можуть використовуватися для ефективного відображення схеми кінозалу, де ключ може бути комбінацією ряду та місця, а значення – статусом зайнятості місця. Також корисні для кешування даних або швидкого доступу до певних сутностей за їх ID;

3) DateTime (Дата/Час) які широко використовується для роботи з часовими даними, такими як дата виходу фільму, час початку та закінчення кіносеансів, тривалість фільмів, дата реєстрації користувача або час створення коментаря;

б) алгоритми:

1) алгоритми сортування та фільтрації – ці алгоритми широко використовуються як на Backend, так і на Frontend. На Backend, через LINQ (Language Integrated Query), вони дозволяють формувати оптимізовані SQL-запити до бази даних, що виконують такі операції, як сортування списку фільмів за датою виходу, сортування сеансів за часом початку, фільтрація квитків за користувачем або сеансом, а також пошук фільмів за назвою або жанром. Entity Framework Core ефективно переводить ці LINQ-запити у високоефективні SQL-команди, що дозволяє базі даних виконувати сортування та фільтрацію на рівні джерела даних, тим самим мінімізуючи обсяг переданих даних та підвищуючи загальну продуктивність системи. На Frontend, сортування та фільтрація можуть бути реалізовані для покращення

користувацького досвіду, дозволяючи динамічно організовувати відображену інформацію (наприклад, фільтрувати розклад залів або жанрів, сортувати список акторів) без необхідності повторних запитів до сервера, якщо дані вже завантажені.

2) алгоритми хешування (Hashing) – використовуються для безпечного зберігання паролів користувачів. Важливо, що паролі ніколи не зберігаються у відкритому вигляді. ASP.NET Core Identity використовує надійні, криптографічно стійкі алгоритми хешування (наприклад, PBKDF2), що унеможливлює відновлення оригінального пароля з хешу. Це забезпечує високий рівень безпеки облікових записів та захист персональних даних;

3) генерація токенів (JWT) – складний криптографічний алгоритм для створення та підписання JWT-токенів, який забезпечує їхню цілісність та автентичність;

4) алгоритми валідації які застосовуються для перевірки вхідних даних;

5 ТЕСТУВАННЯ ТА ВВЕДЕННЯ В ЕКСПЛУАТАЦІЮ

5.1. Методи та стратегії тестування ПЗ

Для забезпечення коректної роботи розробленого веб-застосунку було застосовано стратегію функціонального тестування, зосереджену на перевірці відповідності функціональності системи встановленим вимогам. Це передбачає тестування, чи виконує застосунок те, що від нього очікується, згідно з бізнес-вимогами.

Види тестування, що проводились:

а) функціональне тестування (Functional Testing):

1) спрямоване на перевірку, чи виконує застосунок усі заявлені функції відповідно до специфікації;

2) включає тестування усіх користувацьких сценаріїв (реєстрація, вхід, бронювання квитків, управління фільмами, сеансами, кінозалами, користувачами тощо) для всіх ролей (клієнт, адміністратор);

3) це тестування проводилося переважно ручним способом, зосереджуючись на взаємодії з інтерфейсом користувача (Frontend) та безпосередньо з API (Backend);

б) тестування API (API Testing):

1) сфокусоване на перевірці коректності роботи кожного окремого API-методу, його вхідних параметрів та вихідних даних;

2) для цього використовувався інтегрований у Backend інструмент Swagger UI, який дозволяв зручно надсилати запити до API та аналізувати відповіді;

в) тестування безпеки (Security Testing):

1) Проводилася перевірка базових механізмів аутентифікації та авторизації, включаючи коректність використання JWT-токенів. Це включало перевірку успішного входу в систему з валідними обліковими даними, а

також перевірку відмови у доступі з невірними даними. Особлива увага приділялася передачі та валідації JWT-токенів для кожного захищеного запиту, щоб переконатися в їхній цілісності та терміні дії;

2) тестувався доступ до ресурсів відповідно до ролей користувачів (адміністратор, клієнт), щоб переконатися, що неавторизовані користувачі не можуть отримати доступ до захищених даних або функцій. Наприклад, тестувалося, чи може клієнт отримати доступ до адміністративної панелі або до даних інших користувачів. Перевірялися сценарії, коли спроба доступу до неавторизованого ресурсу повертає відповідний HTTP-статус (наприклад, 401 Unauthorized або 403 Forbidden).

5.2. Розробка тестових прикладів та сценаріїв

Для проведення тестування були розроблені тестові приклади та сценарії, які охоплювали ключові функціональні можливості системи. Тестовий приклад, який описував конкретний набір вхідних даних, умови виконання та очікуваний результат для перевірки певної функції. Тестовий сценарій який представляв послідовність дій користувача для перевірки функціональності на більш високому рівні, імітуючи реальні кейси використання.

Етапи розробки тестових прикладів:

- аналіз вимог: визначення усіх функціональних вимог до системи;
- ідентифікація функцій для тестування: визначення конкретних функцій або API-методів, які потребують перевірки;
- формування вхідних даних: створення наборів вхідних даних, що включають як коректні, так і некоректні значення для перевірки обробки помилок;
- визначення очікуваних результатів: чітке формулювання, як система повинна повестися або які дані повернути за певних умов;
- опис кроків тестування: покрокові інструкції для виконання тесту.

Приклади тестових сценаріїв:

а) сценарій 1 (Успішна реєстрація нового клієнта):

1) дії – перейти на сторінку реєстрації, ввести валідні дані для нового клієнта та натиснути кнопку «Register»;

2) очікуваний результат – успішна реєстрація, перенаправлення на сторінку входу. Новий клієнт відображається у базі даних (перевірка через API адміністратора або безпосередньо у БД);

б) сценарій 2 (Бронювання квитка на кіносеанс):

1) дії – авторизуватися як «Клієнт», перейти на сторінку або «Seances», вибрати бажаний фільм та конкретний сеанс, обрати одне або декілька місць на інтерактивній схемі кінозалу та натиснути кнопку «Book the ticket»;

2) очікуваний результат – бронювання/купівля квитка успішно виконано, з'являється повідомлення про успішне бронювання/купівлю, квиток/сеанс відображається у персональному кабінеті клієнта в розділі «My tickits»;

в) сценарій 3 (Додавання нового фільму адміністратором):

1) дії – авторизуватися як «Адміністратор», перейти до розділу «Manage movies», натиснути кнопку «Add Movie», заповнити всі обов'язкові поля фільму (назва, опис, тривалість, посилання на постер та трейлер, дата виходу, базова ціна) та натиснути «Create»;

1) очікуваний результат – новий фільм успішно додано до системи, відображається у списку фільмів, є можливість його редагувати або видалити.

5.3. Проведення тестування (функціональне, API-тестування)

Тестування проводилося поетапно, після реалізації ключових функціональних блоків Frontend та Backend. Функціональне тестування: виконувалися всі розроблені тестові сценарії для перевірки відповідності

реалізованого функціоналу бізнес-вимогам. Проводилася візуальна перевірка інтерфейсу користувача, коректності відображення даних та зручності взаємодії. Тестування API за допомогою Swagger UI: для перевірки коректності роботи Backend API. Swagger був доданий до проєкту шляхом конфігурації у файлі Program.cs (Лістинг 3.3).

Лістинг 5.1 – Додавання Swagger до проєкту

```
     services.AddSwaggerGen();
     var app = builder.Build();
     if (app.Environment.IsDevelopment()){
         app.UseSwagger();
         app.UseSwaggerUI(c =>{
             c.SwaggerEndpoint("/swagger/v1/swagger.json",  "  My
API V1 ");
             c.OAuthAppName("ScreenScene API ");
       });
     }
```

Це дозволило автоматично генерувати документацію API та інтерфейс для взаємодії з ним. Процес тестування через Swagger UI: після запуску Backend-додатку, Swagger UI відкривався у веб-браузері, надаючи доступ до усіх контролерів та їхніх методів. Для тестування конкретного API-методу (наприклад, POST /api/authentication/registration/client), тестувальник обирав відповідний ендпоінт, натискав кнопку «Try it out» для активації інтерфейсу введення параметрів. Вводилися необхідні вхідні дані (наприклад, JSON-тіло запиту для реєстрації клієнта), після чого натискалася кнопка «Execute» для відправлення запиту на сервер. Отримана відповідь від сервера (у форматі JSON) негайно відображалася в інтерфейсі Swagger UI, дозволяючи візуально перевірити її коректність, коди статусу та вміст. Рисунок 3.3 демонструє приклад відповіді сервера на такий запит. Авторизація в Swagger: для тестування захищених API-методів, що вимагають авторизації (наприклад, усі методи управління, доступні лише адміністратору), використовувалася функція «Authorize» у Swagger UI. Це дозволяло ввести валідний Bearer токен (отриманий після успішної аутентифікації), який автоматично додавався до заголовків подальших запитів. Це забезпечувало перевірку

механізмів авторизації.

Code Details

201
Undocumented

Response body


```
{
  "message": "Client created successfully"
}
```


Рисунок 5.1 – Відповідь від сервера

5.4. Аналіз результатів тестових експериментів та виявлення помилок

Під час виконання тестів, на Backend велося детальне логування подій та помилок, що дозволяло ідентифікувати проблеми, які не були очевидними з боку інтерфейсу. На Frontend використовувалася консоль розробника браузера для відстеження помилок JavaScript та мережевих запитів/відповідей. При тестуванні API через Swagger UI відповіді сервера у форматі JSON були безпосередньо доступні для аналізу, що дозволяло швидко виявляти некоректні дані, помилкові коди статусу або відсутність очікуваної інформації. Виявлені дефекти та відхилення від очікуваної поведінки аналізувалися для подальшого виправлення. Виявлені помилки були пріоритизовані за їхнім впливом на функціональність та критичністю.

5.5. Опис налагодження ПЗ

Налагодження (debugging) було інтегровано в процес розробки та виконувалося ітеративно з тестуванням. Активно використовувалися вбудовані засоби налагодження інтегрованих середовищ розробки (IDE) – Visual Studio для Backend (C#) та VS Code для Frontend (Angular). Встановлення точок зупинки (breakpoints) в коді дозволяло призупиняти

виконання програми в певних місцях, щоб аналізувати стан змінних, стек викликів та потік виконання. Функції покрокового виконання коду (Step Over, Step Into, Step Out) використовувалися для детального відстеження логіки програми. При виникненні неперехоплених винятків, аналізувався стек-трейс для визначення точного місця в коді, де виникла проблема. Додаткові лог-повідомлення тимчасово додавалися в код для трасування складних сценаріїв та відстеження проміжних значень. Основна мета налагодження полягала не просто у виправленні симптомів, а у виявленні та усуненні першопричини помилки, щоб запобігти її повторному виникненню в майбутньому.

5.6. Інструкції щодо розгортання ПЗ (системні вимоги, мінімальна конфігурація апаратного та програмного забезпечення)

Розробка та тестування програмного забезпечення здійснювалися на персональному комп'ютері з операційною системою Windows 10, який слугував єдиним середовищем для функціонування як Backend, так і Frontend компонентів, а також бази даних.

Для успішного функціонування розробленого веб-застосунку були використані наступні характеристики апаратного та програмного забезпечення розробницької машини.

- процесор: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz, 2.40 GHz, що є 4-ядерним процесором, достатнім для розробки та локального тестування застосунку;
- оперативна пам’ять (RAM): 16,0 ГБ (доступно для використання: 15,9 ГБ), що забезпечує комфортну роботу IDE, Backend-сервісу, бази даних та веб-браузера одночасно;
- сховище: 466 ГБ SSD (Samsung SSD 970 EVO Plus 500GB) та 932 ГБ HDD (WDC WD10SPZX-24Z10). SSD диск був використаний для операційної системи та проектних файлів, забезпечуючи високу швидкість завантаження

та доступу до даних;

- графічна плата: NVIDIA GeForce GTX 1050 (3 ГБ) та Intel(R) UHD Graphics 630 (128 МБ). Хоча для веб-розробки потужна відеокарта не є критичною, наявність цих плат забезпечувала стабільну роботу графічного інтерфейсу та інструментів розробника. Тип системи: 64-розрядна операційна система, процесор на базі архітектури x64, що є стандартом для сучасних розробницьких середовищ. Перо та дотики: ввід за допомогою пера та сенсорний ввід недоступні на цьому дисплеї.

Для Backend: .NET SDK / .NET Runtime: проект був розроблений та запускався на відповідній версії .NET 9.0. Для Бази даних: MS SQL Server: була встановлена відповідна версія SQL Server 2022). Для Frontend (клієнтська частина): тестування здійснювалося у сучасних веб-браузерах Mozilla Firefox та Microsoft Edge, з увімкненим JavaScript, які забезпечують повну сумісність з Angular-додатком. Мережеві вимоги: для локального доступу до веб-застосунку використовувалися стандартні мережеві протоколи. Відкриття портів 80/443 не було потрібно для локальної розробки, оскільки Backend запускався на іншому порту, а Frontend на своєму 4200.

Процес розгортання для розробки та тестування:

- налаштування локального середовища: встановлення Windows 10, .NET SDK, MS SQL Server, Node.js та Angular CLI;

- запуск Backend: ASP.NET Core Web API запускався локально за допомогою Visual Studio або командного рядка;

- налаштування бази даних: створення бази даних та виконання міграцій Entity Framework для ініціалізації схеми БД;

- запуск Frontend: Angular-додаток запускався локально за допомогою Angular CLI (ng serve), що забезпечувало гарячу перезавантаження та зручність розробки;

- тестування: здійснювалося через локальний доступ до Frontend у браузерах та безпосередньо через Swagger UI для API-методів Backend.

5.7. Рекомендації щодо використання та супроводу розроблених програм

Для ефективного використання та забезпечення довгострокового функціонування розробленого програмного забезпечення рекомендуються наступні підходи: регулярне резервне копіювання даних: налаштувати автоматичне щоденне або щотижневе резервне копіювання бази даних MS SQL Server. Зберігати копії на віддаленому сховищі. Моніторинг продуктивності та стабільності: використовувати системи моніторингу для відстеження навантаження на сервер, використання пам’яті, часу відгуку API та помилок. Відстежувати логі додатку на предмет аномалій або частих помилок. Регулярне оновлення компонентів: своєчасно оновлювати фреймворки (Angular, ASP.NET Core), бібліотеки та операційні системи, щоб отримувати оновлення безпеки та нові функції. Це допомагає запобігти потенційним вразливостям та забезпечити сумісність. Управління версіями: весь вихідний код зберігати в системі управління версіями з використанням репозиторіїв. Документування: підтримувати актуальну технічну документацію (опис архітектури, ER-діаграма, API-документація (через Swagger), інструкції з розгортання та супроводу). Планування масштабування: у міру зростання кількості користувачів та обсягів даних, планувати масштабування системи (вертикальне – збільшення ресурсів сервера; горизонтальне – додавання нових екземплярів Backend, реплікація БД). Обслуговування бази даних: регулярно проводити оптимізацію бази даних (індексування, дефрагментація, очищення логів) для забезпечення оптимальної продуктивності. Навчання користувачів: надати користувачам (адміністраторам) чіткі інструкції та, за потреби, провести навчання щодо використання функціоналу системи. Підтримка та зворотний зв’язок:

забезпечити канал для зворотного зв’язку від користувачів та оперативну підтримку для вирішення проблем та реагування на запити.

ВИСНОВКИ

У результаті виконання даного проєкту був успішно розроблений та реалізований повноцінний веб-застосунок для ефективного адміністрування кінотеатру. Комплексне рішення охоплює як потужну серверну частину (Backend), так і зручний та функціональний клієнтський інтерфейс (Frontend).

Основна увага була зосереджена на створенні надійного Backend-сервісу на базі ASP.NET Core Web API, який реалізовано згідно з принципами багаторівневої архітектури (трирівнева архітектура). Це забезпечило чітке розділення відповідальності між шарами та високу гнучкість системи. Backend повністю підтримує виконання всіх необхідних CRUD-операцій (створення, читання, оновлення, видалення) для ключових сутностей, таких як фільми, сеанси, кінозали, актори, жанри, квитки та користувачі. Окрім базових операцій, було розроблено та інтегровано механізми аутентифікації та авторизації на основі JWT-токенів, що гарантує безпечний доступ до ресурсів та управління правами користувачів відповідно до їхніх ролей.

Паралельно з Backend-частиною, була повністю розроблена клієнтська частина на фреймворку Angular, яка реалізована як Односторінковий Додаток (SPA) з використанням сучасного компонентного підходу (Standalone Components). Це дозволило створити інтуїтивно зрозумілий та динамічний користувацький інтерфейс, що забезпечує безперебійну взаємодію з Backend API через асинхронні HTTP-запити. Налаштована маршрутизація, реалізовані функціональні блоки для клієнтів (перегляд розкладу фільмів, бронювання/купівля квитків, управління профілем та історією квитків, оцінки та коментарі до фільмів), а також для адміністраторів (повне управління фільмами, сеансами, кінозалами, користувачами та спеціальними пропозиціями).

Таким чином, реалізоване рішення є завершеним та функціональним

веб-застосунком, який забезпечує повноцінну цифровізацію ключових аспектів управління кінотеатром. Розроблена система є гнучкою, масштабованою та легкою для подальшого супроводу та розвитку. У майбутньому проєкт може бути доповнений розширеною аналітикою, інтеграцією платіжних систем, системою сповіщень та іншими можливостями, що дозволить ще більше поглибити автоматизацію та підвищити ефективність роботи кінотеатру.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Miller J. Dependency Injection Principles, Practices, and Patterns. Manning Publications, 2019. 360 с. ISBN 978-1617294738.
2. Swagger API Documentation. OpenAPI Specification. URL: https://swagger.io (дата звернення: 18.06.2025).
3. Thorstad L. Web APIs with ASP.NET Core in Action. Manning Publications, 2021. 450 с. ISBN 978-1617298194.
4. Richardson C. Microservices Patterns: With Examples in .NET. Manning Publications, 2019. 520 с. ISBN 978-1617294547.
5. Microsoft Documentation. Entity Framework Core. URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 20.06.2025).
6. Microsoft Learn. Tutorial: Create a web API with ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api (дата звернення: 16.06.2025).
7. Angular Documentation. Overview. URL: https://angular.dev/overview (дата звернення: 20.06.2025).