Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра автоматизації та інтелектуальних інформаційних технологій

Бакалаврська кваліфікаційна робота на тему:

«ІНФОРМАЦІЙНА СИСТЕМА УПРАВЛІННЯ РЕСУРСАМИ КІНОТЕАТРУ»

Виконав: студент IV курсу, групи 1ІСТ-21б
спеціальності 126 – Інформаційні системи та технології
Слободиський В.В.

Керівник: к.т.н., доцент кафедри АІІТ
Богач І.В.
«16» 06 2025 р.

Рецензент: д.т.н., професор кафедри КН
Іванчук Я.В.
«17» 06 2025 р.

Допущено до захисту
завідувач кафедри АІІТ
«18» 06 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
(повне найменування вищого навчального закладу)
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра автоматизації та інтелектуальних інформаційних технологій
Рівень вищої освіти перший (бакалаврський)
Галузь знань 12 – Інформаційні технології
Спеціальність 126 – Інформаційні системи та технології
Освітньо-професійна програма Інтелектуальні інформаційні системи

ЗАТВЕРДЖУЮ
завідувач кафедри АІІТ
_____ д.т.н., проф. Бісікало О. В.
«18» 06 2025 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Слободиському Владиславу Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Інформаційна система управління ресурсами кінотеатру»
керівник роботи Богач Ілона Віталіївна, к.т.н, доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від «20» березня 2025 року №97
2. Термін подання студентом роботи 10.06.2024 р.
3. Вихідні дані до роботи:
- наявність пристрою з операційною системою Windows, Linux або macOS;
- наявність застосунків JHipster,Spring Boot, React і PostgreSQL для написання додатку.
4. Зміст текстової частини (перелік питань, які потрібно розробити) Аналіз преметної галузі. Проєктування системи управління ресурсами кінотеатру. Реалізація системи управління ресурсами кінотеатру. Тестування системи управління ресурсами кінотеатру
5. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень) Архітектура системи управління ресурсами кінотеатру. Діаграма класів системи. Структура бази даних (ER-діаграма). Діаграма стану для процесу бронювання. Діаграма розгортання системи

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Богач І.В., доц. каф. АІІТ		

7. Дата видачі завдання 20.03.2025 р.

КАЛЕНДАРНИЙ ПЛАН

/п	Назва етапів бакалаврської квафіліфікаційної роботи	Термін виконання		Примітка
		початок	закінчення	
1	Вибір, узгодження та затвердження теми БКР	03.10.2024	09.10.2024	вик
2	Аналіз предметної області та опрацювання літературних джерел.	10.03.2025	28.03.2025	вик
3	Постановка задач для вирішення в БКР	01.04.2025	25.04.2025	вик
4	Вирішення поставлених задач	28.04.2025	09.05.2025	вик
5	Підтвердження достовірності отриманих результатів	12.05.2025	23.05.2025	вик
7	Оформлення пояснювальної записки та графічної частини	27.05.2025	31.05.2025	вик
8	Перевірка ПЗ на відповідність вимогам	03.06.2025	07.06.2025	вик
9	Попередній захист, доопрацювання, та рецензування БКР	10.06.2025	15.06.2025	вик
10	Захист БКР	16.06.2025	23.06.2025	вик

Студент _____ Слободиський В.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Богач І.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота складається з 148 сторінок формату А4 включаючи додатки, на яких є 43 рисунки і 4 таблиці, список використаних джерел містить 38 найменувань.

У бакалаврській кваліфікаційній роботі розроблено інформаційну систему управління ресурсами кінотеатру із використанням фреймворку JHipster 8.0. Досліджено проблему оптимізації внутрішніх процесів кінотеатру, включно з управлінням персоналом, продажем квитків, контролем запасів снеків і плануванням розкладу показів. В обґрунтуванні актуальності використано порівняльний аналіз існуючих систем. Запропоноване рішення побудоване на Spring Boot для серверної частини та React TypeScript на клієнті, що дозволяє ефективно керувати даними через REST API і підтримує JWT-аутентифікацію. Реалізовано можливості інтеграції з Apache Kafka для обробки подій у реальному часі та з PostgreSQL для зберігання даних. Проєкт включає UML-діаграми для формалізації логіки взаємодії (послідовності, варіантів використання, класів, компонентів тощо). В результаті створено повноцінне рішення, яке забезпечує масштабованість, адаптивність під вимоги конкретних кінотеатрів і високий рівень безпеки.

Ключові слова: система управління ресурсами, кінотеатр, JHipster, Spring Boot, React, JWT, Apache Kafka, PostgreSQL, UML-діаграми.

ABSTRACT

The Bachelor's degree qualification work consists of 148 pages of A4 format, including appendices, which contain 43 figures and 6 tables, and the list of references consists of 38 titles.

This bachelor’s qualification thesis presents the development of a cinema resource management information system based on the JHipster 8.0 framework. The work addresses the optimization of internal cinema processes, including staff management, ticket sales, snack inventory control, and screening schedule planning. A comparative analysis of existing systems (Fandango, Multiplex, and KyivKinoFilm) is provided to justify the relevance of the proposed solution. The system employs Spring Boot on the server side and React TypeScript on the client side, enabling efficient data handling via a REST API and supporting JWT-based authentication. Integration with Apache Kafka is implemented for real-time event processing, and PostgreSQL is used for data storage. The project includes UML diagrams to formalize interaction logic (sequence, use-case, class, component, etc.). As a result, a comprehensive solution has been created that ensures scalability, adaptability to specific cinema requirements, and a high level of security.

Keywords: resource management system, cinema, JHipster, Spring Boot, React, JWT, Apache Kafka, PostgreSQL, UML diagrams.

ЗМІСТ

ВСТУП

Актуальність роботи. Управління ресурсами кінотеатру є багатогранним процесом, що охоплює контроль за розкладом сеансів, продажем квитків, обслуговуванням клієнтів, утриманням обладнання, роботою персоналу та запасами снеків. Ефективність цього процесу безпосередньо впливає на економічні показники закладу та на рівень задоволеності відвідувачів. У сучасних умовах, коли використання інформаційних технологій стало ключем до конкурентної переваги, кінотеатри змушені швидко адаптуватися, запроваджуючи інноваційні методи автоматизації та аналітики.

Слід зазначити, що галузь кіноіндустрії традиційно відзначається певною складністю: необхідно враховувати сезонні коливання попиту, умови співпраці з дистриб'юторами та особливості локального ринку (зокрема, наявність чи відсутність державних обмежень, ситуацію з платоспроможністю населення, культурні уподобання) [1]. Додатковим викликом є швидка цифровізація сфери розваг та конкуренція зі стрімінговими сервісами, які надають користувачам можливість переглядати широкий вибір контенту вдома.

У такій ситуації впровадження сучасних інформаційних систем стає одним із найефективніших способів оптимізації процесів: автоматизовані інструменти дають змогу швидко обробляти транзакції, узгоджувати розклад, відстежувати наявність місць у залі, аналізувати купівельну поведінку клієнтів та забезпечувати гнучку систему лояльності. Крім того, застосування розвиненого аналітичного інструментарію дозволяє керівництву кінотеатру вчасно реагувати на зміни в попиті, удосконалювати маркетингові стратегії та підтримувати якість обслуговування.

Проте наявні на ринку програмні рішення не завжди повною мірою охоплюють усі аспекти управління ресурсами. Значна частина сервісів зосереджена лише на реалізації квитків, залишаючи поза увагою управління персоналом чи логістику запасів [2]. Крім того, багато з них виявляються недостатньо гнучкими й орієнтованими переважно на вузькі сегменти або великі мережі, що ускладнює використання малими чи незалежними кінотеатрами з

індивідуальною специфікою. Також проблемою може бути обмежена підтримка локальних платіжних систем та відсутність української локалізації – це створює бар’єри для широкого запровадження і відлякує потенційних користувачів [4].

З огляду на це, вивчення можливостей створення власного рішення, яке б задовольняло потреби українських кінотеатрів і розв’язувало низку типових для галузі завдань, набуває стратегічної важливості. Такий підхід дає змогу врахувати локальні особливості, додати необхідний функціонал, забезпечити масштабованість та інтеграцію з іншими сервісами. Водночас застосування сучасних технологій (зокрема, фреймворку JHipster, мов програмування Java та TypeScript, рішень для мікросервісної або монолітної архітектури) дозволить створити високопродуктивну й безпечну платформу, здатну оперативно реагувати на виклики та підвищувати рівень автоматизації бізнес-процесів.

Таким чином, актуальність і доцільність теми «Розробка інформаційної системи управління ресурсами кінотеатру» зумовлені потребою підвищення ефективності діяльності кінотеатрів, а також відсутністю комплексного рішення, що відповідало б вимогам локального ринку і підтримувало повний спектр функцій для управління розкладом, персоналом, запасами та фінансово-маркетинговими напрямами. Тому обґрунтований підхід до створення власної системи може сприяти розвитку галузі, покращенню сервісу та досягненню стабільнішого фінансового результату.

Метою даної роботи є вдосконалення інформаційних систем управління ресурсами кінотеатру, що на відміну від існуючих забезпечить підвищення ефективності ключових бізнес-процесів, оптимізацію взаємодії з клієнтами та зменшення витрат на експлуатацію.

Для досягнення мети роботи, потрібно **розв’язати наступні задачі**:

1. Провести аналітичний огляд існуючих платформ і визначити їх переваги та недоліки в контексті вимог українського ринку;

- обґрунтувати вибір архітектури та для розробки системи;
- спроєктувати систему управління ресурсами кінотеатру, в т.ч. створити UML-діаграми (діаграми варіантів використання, послідовності, класів,

компонентів, розгортання тощо) для формалізованого опису структурних та поведінкових аспектів системи

- реалізувати систему управління ресурсами кінотеатру;
- провести тестування програмного модуля з метою оцінки його продуктивності та відповідності функціональним вимогам.

Об’єкт дослідження – процес управління ресурсами в кінотеатрі (контроль запасів, розклад сеансів, персонал, взаємодія з клієнтами).

Предмет дослідження – методи та інструментарій розробки інформаційних систем для автоматизації та оптимізації бізнес-процесів у кіноіндустрії.

Методи дослідження. Під час роботи використано методи системного аналізу для визначення вимог та побудови архітектури, UML-моделювання для формалізації структури та поведінки системи, об’єктно-орієнтоване програмування та фреймворк JHipster для реалізації прототипу інформаційної системи. Для аналізу продуктивності та функціональності застосовано методи тестування на реальних сценаріях використання.

В роботі розроблено підхід до інтеграції ключових процесів кінотеатру (продаж квитків, облік запасів, маркетинг, управління персоналом) у єдину інформаційну платформу, адаптовану до умов локального ринку та запропоновано структуру монолітного застосунку з використанням сучасних інструментів (JHipster, Kafka, React, Spring Boot), що забезпечує високий рівень масштабованості й адаптивності завдяки відкритому API та модульному принципу побудови.

Практичний результат роботи полягає в тому, що на основі проведених досліджень і запропонованої архітектури було розроблено програмні модулі для управління ресурсами кінотеатру (зокрема, системи бронювання, обліку персоналу й запасів снеків), що можуть бути інтегровані в реальні або демонстраційні середовища. Застосована схема, на відміну від існуючих, дозволяє масштабувати рішення під потреби кінотеатрів різного розміру, передбачає зручні інструменти для аналітики й маркетингу, а також може бути

використана як фундамент для подальшого розширення функціоналу та інтеграції з новими сервісами.

Апробація результатів дослідження. Результати роботи було представлено на LIV Всеукраїнській науково-технічній конференції підрозділів Вінницького національного технічного університету (2025) [20] та на Міжнародній науково-практичній Інтернет-конференції студентів, аспірантів та молодих науковців «Молодь в науці: дослідження, проблеми, перспективи (МН-2025)» [37].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Основні принципи управління ресурсами в кінотеатрах

Ресурси – це всі активи, засоби, матеріальні й нематеріальні елементи, які використовуються для досягнення цілей організації. У контексті кінотеатрів ресурси включають кілька видів [5]:

Людські ресурси – персонал, який обслуговує клієнтів, працює на касах, займається управлінням кінотеатром, забезпеченням роботи техніки, а також персонал для прибирання та забезпечення безпеки.

Матеріальні ресурси – технічне обладнання, кінострічки, зали для показу фільмів, інфраструктура кінотеатру (меблі, кіноекрани, звукові системи, проєктори).

Фінансові ресурси – кошти, необхідні для підтримки операційної діяльності, розвитку інфраструктури, виплати заробітної плати, а також фінансування маркетингових кампаній та інших проєктів.

Інформаційні ресурси – дані, аналітика, звіти про продажі, відгуки відвідувачів, статистика переглядів, а також системи управління інформацією для ефективного ведення бізнесу.

Управління ресурсами є комплексним процесом, що включає планування, організацію, розподіл і контроль для досягнення встановлених цілей. У контексті кінотеатрів ресурси охоплюють персонал, фінансові ресурси, технічне обладнання, кінострічки та інфраструктурні компоненти.

Відмінність управління ресурсами в кінотеатральній сфері полягає в залежності від факторів сезонності, нових кінопрем'єр, рівня попиту та активності глядачів, що робить його менш стабільним у порівнянні з іншими галузями, де попит та споживання ресурсів є відносно передбачуваними.

Пандемія COVID-19, яка розпочалася у 2019 році, суттєво змінила підходи до управління ресурсами в кінотеатрах. Одним із ключових наслідків стало значне скорочення відвідуваності через карантинні обмеження та нові санітарні вимоги. Це спонукало кінотеатри до оптимізації використання залів, розвитку

цифрових рішень, таких як онлайн-продаж квитків та попереднє бронювання місць, а також до зменшення контактів під час обслуговування клієнтів. З'явилася потреба у нових формах маркетингу, націлених на залучення аудиторії в умовах обмежень, а також у коригуванні фінансових стратегій через значне зниження доходів.

Починаючи з 2022 року, війна в Україні створила ще більш складні умови для управління ресурсами кінотеатрів [2]. Серед ключових проблем виникли нестабільність у роботі, загрози для безпеки як відвідувачів, так і співробітників, проблеми з постачанням фільмів та необхідних матеріалів, а також фінансовий тиск у зв'язку зі зниженням купівельної спроможності населення.

У багатьох регіонах кінотеатри були змушені тимчасово закритися, а в більш безпечних районах було зроблено акцент на підтримці суспільного духу через соціальні ініціативи та покази для місцевих громад.

Зміни в структурі управління також передбачали більше уваги до гнучких робочих графіків, зниження витрат, утримання ключових працівників, а також пошук нових джерел доходу через співпрацю з онлайн-платформами та організацію заходів, які виходять за рамки звичайних кінопоказів.

Управління ресурсами в кінотеатрах є складним і багатоаспектним процесом, що охоплює організацію та координацію людських, матеріальних, фінансових і інформаційних ресурсів. У сучасних умовах, з урахуванням викликів, які постають перед цією сферою, ефективне управління стає особливо актуальним для забезпечення стабільної роботи, підтримки конкурентоспроможності та задоволення потреб аудиторії.

Основні принципи управління ресурсами в кінотеатрах відображені в таблиці 1.1.

Таблиця 1.1 – Основні принципи управління ресурсами в кінотеатрах

Принцип	Характеристика
Ефективне планування ресурсів	Ключовим елементом управління є планування, яке включає складання детальних розкладів показів фільмів, графіків роботи персоналу, а також визначення потреб у технічному обслуговуванні обладнання. Це дозволяє зменшити непродуктивні витрати, забезпечити оптимальне використання наявних ресурсів і задовольнити потреби відвідувачів.
Управління людськими ресурсами	Для кінотеатру важливо забезпечити якісний сервіс, тому особлива увага приділяється найму, навчання та мотивації персоналу. Управління графіками роботи, навчання нових співробітників та підтримка їх професійного розвитку сприяють підвищенню ефективності обслуговування клієнтів і підтримці високого рівня задоволеності відвідувачів.
Фінансове планування та контроль	Фінансове управління включає складання бюджету, контроль витрат і прогнозування доходів. Це допомагає кінотеатрам визначити, які витрати необхідні для підтримки обладнання, проведення маркетингових кампаній та забезпечення комфорту для глядачів. Важливим є також аналіз рентабельності окремих кінопоказів і заходів, щоб оптимізувати фінансові потоки.
Управління технічними ресурсами	Сучасні кінотеатри використовують складне обладнання, таке як цифрові проектори, аудіосистеми та системи кондиціонування повітря. Тому важливим аспектом є регулярне обслуговування і модернізація обладнання для забезпечення високої якості показу. Це включає планування профілактичних перевірок, своєчасне оновлення техніки та вирішення технічних проблем.
Оптимізація управління простором	Кінотеатри повинні раціонально використовувати свої приміщення, оптимально розміщуючи зони очікування, каси, кіоски з напоями та попкорном, а також забезпечуючи комфортні умови в залах. Управління простором також включає планування використання залів для різних видів заходів, таких як прем'єри, корпоративні події або спеціальні покази.
Інформаційні технології та автоматизація	Використання сучасних інформаційних систем дозволяє оптимізувати роботу кінотеатру. Це включає впровадження систем автоматизації продажу квитків, CRM-систем для управління взаємовідносинами з клієнтами, а також аналітичних інструментів для моніторингу продажів і ефективності маркетингових кампаній. Впровадження систем онлайн-продажу та резервування квитків також сприяє підвищенню рівня зручності для клієнтів.

Продовження таблиці 1.1

1	2
Маркетинг і управління попитом	Ефективне управління ресурсами в кінотеатрах також передбачає застосування маркетингових стратегій для залучення відвідувачів. Це може включати гнучке ціноутворення, акції, програми лояльності, співпрацю з медіа та проведення спеціальних заходів, таких як прем'єри чи тематичні вечори. Управління попитом дозволяє забезпечити максимальне заповнення залів у різні дні тижня та часи доби.

На рисунку 1.1 представлені зміни активності кінотеатрів у період з 2018 до 2023 року. Загальна кількість активності різко впала під час пандемії і залишалася низькою під час наступних років через карантинні обмеження і наслідки війни. Натомість кількість онлайн-активності значно зросла, оскільки кінотеатри адаптували свої послуги до цифрових форматів.

Рисунок 1.1 – Зміни активності кінотеатрів

Таким чином, ефективне управління ресурсами в кінотеатрах спрямоване на забезпечення безперебійної роботи, підвищення рентабельності та задоволення потреб відвідувачів [1, 2]. Воно базується на інтегрованому підході до управління всіма видами ресурсів, оптимізації операційних процесів та використанні сучасних інформаційних технологій.

1.2 Дослідження інформаційних систем управління ресурсами кінотеатру

Інформаційна система управління ресурсами кінотеатру – це комплексне програмне рішення, яке інтегрує та автоматизує всі аспекти управління кінотеатром, включаючи продаж квитків, управління персоналом, контроль за запасами снеків і напоїв, а також планування показів фільмів. Система призначена для підвищення ефективності операцій, зниження витрат та покращення задоволеності клієнтів [1].

Основні особливості таких систем відображені в таблиці 1.2

Таблиця 1.2 – Основні особливості інформаційних систем управління ресурсами кінотеатру [3]

Особливості	Характеристика
Інтеграція	Система об'єднує різні функціональні сфери кінотеатру в єдине програмне рішення, що дозволяє централізовано управляти всіма ресурсами
Автоматизація	Від автоматичного розкладу фільмів до управління запасами і обліку персоналу, система мінімізує необхідність ручної роботи.
Звітність та аналітика	Потужні аналітичні інструменти дозволяють керівництву кінотеатру отримувати детальні звіти про продажі, популярність фільмів, ефективність персоналу тощо.
Масштабованість	Система легко адаптується до зміни розмірів або потреб кінотеатру, дозволяючи легко розширювати функціонал або інтегрувати нові технології.

Інформаційні системи управління можна класифікувати за кількома критеріями.

1. За масштабом використання бувають локальні системи, які обслуговують один кінотеатр, і мережеві, що використовуються для управління групою кінотеатрів.

2. За функціональністю можуть бути спеціалізовані системи, призначені для окремих завдань, таких як продаж квитків, або інтегровані, що покривають усі аспекти діяльності кінотеатру.

3. За розміщенням існують локальні системи, встановлені на серверах кінотеатру, та хмарні рішення, які використовують інтернет для хостингу.

Кіноіндустрія проходить через значні трансформації, обумовлені цифровізацією, пандемією COVID-19, війною в Україні та зростанням популярності стрімінгових платформ. Ці фактори змушують кінотеатри швидко адаптуватися до нових умов:

– відбувається масове впровадження цифрових технологій, що включає онлайн-продаж квитків та використання цифрового обладнання для показу фільмів;

– зростає потреба в підвищенні задоволеності клієнтів через особистісний підхід, програми лояльності та покращення загального досвіду відвідування;

– розвивається співпраця з онлайн-платформами для стрімінгу фільмів, що стає частиною гібридної моделі розповсюдження контенту.

Щоб впровадити нові інформаційні системи, важливо розуміти, які проблеми спровокували цю потребу. Багато кінотеатрів досі використовують застарілі системи для управління запасами, розкладами сеансів та роботою персоналу, що веде до зайвих витрат та втрати потенційного доходу. Відсутність інтегрованого підходу до управління розкладом, персоналом та маркетингом спричиняє плутанину та помилки, що негативно впливає на клієнтський досвід. Крім того, недоліки традиційних систем в аналізі даних обмежують можливості планування та оптимізації маркетингових заходів [6, 7].

Для подолання цих викликів, інформаційна система управління ресурсами кінотеатру може стати дієвим інструментом, що забезпечить автоматизацію, інтеграцію та аналітику, необхідні для сучасного управління. Впровадження інформаційної системи управління ресурсами дозволить значно оптимізувати внутрішні процеси, покращити обслуговування клієнтів та підвищити ефективність роботи персоналу.

Основні компоненти такої системи включають управління персоналом, автоматизацію продажу квитків, контроль запасів та планування розкладу показів. Інформаційна система дозволяє автоматизувати розклад роботи співробітників, що полегшує процес розподілу змін і враховує потреби кінотеатру. Вона також забезпечує ефективне відстеження робочого часу та інших аспектів управління персоналом, що звільняє час менеджерів для стратегічних завдань.

Система управління квитками інтегрується з онлайн-платформами та касами кінотеатру, дозволяючи клієнтам зручно планувати своє відвідування, а кінотеатру – автоматично оновлювати дані про наявність місць у реальному часі. Це суттєво покращує досвід користувачів і дозволяє підвищити ефективність продажу квитків.

Контроль запасів включає управління продуктами харчування та іншими товарами, що продаються в кінотеатрі. Система може прогнозувати потреби на основі минулих даних, автоматично відстежувати запаси та забезпечувати своєчасне поповнення, що дозволяє уникнути нестачі товарів або їх надлишку.

Система планування показів фільмів допомагає оптимізувати розклад таким чином, щоб забезпечити максимальну заповнюваність залів і задовольнити потреби аудиторії. Інтеграція з аналітичними інструментами дозволяє ефективніше відслідковувати популярність фільмів і швидко реагувати на зміни в попиті.

Впровадження інформаційної системи управління ресурсами в кінотеатрах приносить значні переваги завдяки автоматизації, підвищенню ефективності та забезпеченню високого рівня задоволення клієнтів. Сучасна система управління

ресурсами є невід'ємною частиною стратегії кінотеатру, що дозволяє адаптуватися до змін у ринкових умовах та зберегти конкурентні позиції.

Існує кілька видів інформаційних систем для кінотеатрів. Розглянемо готові рішення більш детально.

1. Квиткові системи – автоматизують продаж квитків через різні канали, включаючи онлайн, каси та мобільні додатки. Прикладом можуть бути системи «BoxOffice» або «Vista Cinema».

2. CRM-системи (Customer Relationship Management) – збирають та аналізують дані про клієнтів, їхні вподобання та поведінку. Використання CRM-систем, як-от «Salesforce» або «Key CRM», дозволяє автоматизувати взаємодію з клієнтами та підтримувати їхню лояльність.

3. Системи управління контентом (CMS) – використовуються для управління вебсайтами та мобільними додатками кінотеатрів, що дозволяє глядачам переглядати розклад, бронювати місця та купувати квитки.

4. Системи управління ресурсами (ERP) – автоматизують внутрішні процеси кінотеатру, такі як бухгалтерський облік, управління запасами, логістика та HR-менеджмент.

5. Маркетингові платформи – автоматизують маркетингові кампанії, управління акціями та знижками, моніторинг ефективності рекламних заходів і аналіз клієнтської бази.

Ці системи дозволяють кінотеатрам оптимізувати свою роботу, поліпшити якість послуг і задовольнити вимоги сучасного ринку, адаптуючись до швидких змін у сфері розваг. Інформаційні системи стають невід'ємною частиною управління сучасними кінотеатрами, сприяючи їхній конкурентоспроможності та розвитку.

На рисунку 1.2 продемонстровано використання різних видів інформаційних систем у кінотеатрах. Вона показує, як розподіляються системи для продажу квитків, управління ресурсами, кібербезпеки та аналітичні системи для маркетингу [1-6].

Рисунок 1.2 – Частка використання різних видів інформаційних систем у кінотеатрах

Після впровадження інформаційних систем багато кінотеатрів змогли суттєво покращити свою діяльність. Наприклад, кінотеатр «Сінема Сіті» в Україні запровадив систему автоматизації продажу квитків та управління касами, що дозволило підвищити точність прогнозів щодо продажів на 25% та знизити витрати на операційне управління на 15%. Це сприяло значному покращенню ефективності роботи, особливо під час періодів пікового навантаження.

Американська мережа кінотеатрів «IMAX» впровадила CRM-систему, завдяки якій збираються дані про відвідувачів, що дозволило значно покращити персоналізацію маркетингових повідомлень. У результаті відвідуваність збільшилася на 18% завдяки таргетованим акціям і електронним розсилкам.

«Планета Кіно», українська мережа кінотеатрів, використовує хмарні системи для автоматизації управління логістикою, запасами та бухгалтерськими процесами. Це дозволило зменшити адміністративні витрати на 20% і значно підвищити якість обслуговування клієнтів, оптимізувавши процеси планування.

Інформаційні системи надають кінотеатрам гнучкість і можливість швидкої адаптації до змін ринкових умов, що є важливим фактором збереження

конкурентних позицій. Автоматизація основних процесів, таких як продаж квитків, маркетинг, управління запасами та ресурсами, стала невід'ємною частиною сучасного кінотеатру.

У сучасному динамічному світі кіноіндустрії інформаційні системи управління ресурсами кінотеатру є незамінним інструментом для оптимізації бізнес-процесів, підвищення задоволеності клієнтів та ефективного управління персоналом і матеріальними ресурсами. Такі системи інтегрують усі ключові функції управління, зокрема планування розкладу показів фільмів, продаж квитків, облік запасів снеків і напоїв, а також управління персоналом. Це забезпечує зниження витрат і підвищує загальну ефективність діяльності кінотеатру.

Автоматизація цих процесів сприяє зменшенню помилок, звільняє час співробітників для виконання важливіших завдань та дозволяє кінотеатру швидко реагувати на зміни в попиті та вподобаннях клієнтів. Аналітичні можливості системи надають менеджерам глибоке розуміння поведінки клієнтів та ефективності різних аспектів роботи, що допомагає приймати обґрунтовані управлінські рішення. Як результат, кінотеатри, що впроваджують сучасні інформаційні системи управління ресурсами, можуть не лише значно підвищити якість обслуговування та рівень задоволеності відвідувачів, а й збільшити свою конкурентоспроможність на ринку.

1.3 Порівняльний аналіз аналогів

Для підтвердження актуальності розробки власної реалізації необхідно проаналізувати деякі популярні програмні засоби, які використовуються для організації процесу бронювання квитків у кінотеатрах. Такий аналіз дозволяє визначити переваги, недоліки та обмеження існуючих рішень, що, своєю чергою, допоможе обґрунтувати вибір архітектурних підходів, функціонального наповнення та технологій для створення нової системи. Розглянемо три актуальні приклади, що представляють різні підходи до вирішення завдання.

Сервіс Fandango (рис. 1.3) є одним із найвідоміших прикладів онлайн-платформ для придбання квитків у кінотеатри, який активно функціонує на території США [4].

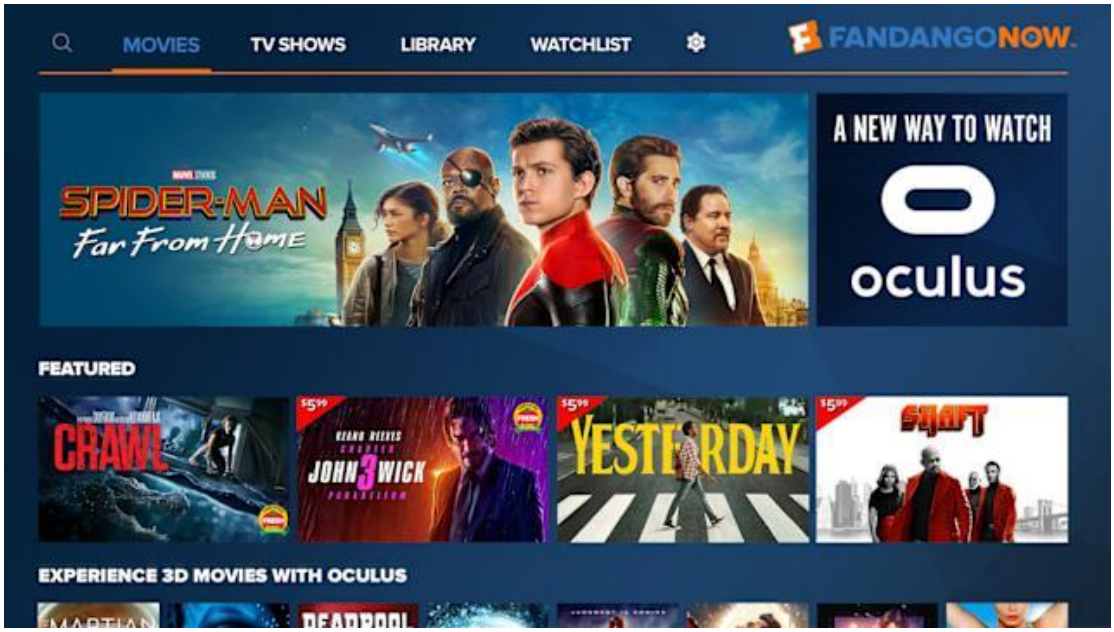

Рисунок 1.3 – Вікно програми «Fandango»

Його широке розповсюдження обумовлене високим рівнем зручності для кінцевого користувача, оскільки платформа дозволяє здійснювати повний цикл взаємодії з кіномережами: від вибору фільму й сеансу до оплати замовлення та збереження електронного квитка в мобільному додатку або гаманці пристрою. Важливою перевагою цього рішення є його інтеграція з додатковими сервісами, що надають доступ до рецензій, рейтингів і трейлерів, дозволяючи користувачу приймати обґрунтоване рішення при виборі контенту. Також платформа демонструє високу стійкість до навантажень і здатна ефективно обслуговувати значну кількість користувачів одночасно, що особливо актуально під час прем’єрних показів або розпродажів.

Разом з тим, існують і обмеження, які знижують придатність Fandango для локального впровадження в умовах українського ринку. По-перше, система є повністю закритою, її вихідний код недоступний, що унеможливлює адаптацію або інтеграцію з наявною інфраструктурою кінотеатру без погодження з

правовласниками. Також функціонал сервісу орієнтований винятково на англомовного користувача, а інтерфейс та система оплати не підтримують локалізацію або роботу з національними банківськими інструментами. Ще одним суттєвим недоліком є залежність Fandango від обмеженого переліку партнерських кіномереж, що виключає можливість використання платформи незалежними або малими кінотеатрами без попереднього укладення окремих ліцензійних угод. Крім того, модель співпраці передбачає обов'язкові фінансові зобов'язання, що робить впровадження сервісу економічно недоцільним для закладів з обмеженим бюджетом.

Таким чином, хоча Fandango забезпечує користувачів сучасним і надійним механізмом бронювання квитків, його замкненість, вузька орієнтація на зовнішній ринок і відсутність інструментів для адаптації під локальні умови створюють серйозні бар'єри для використання в українських реаліях. Це підтверджує необхідність розробки власного інформаційного рішення, здатного враховувати специфіку внутрішнього ринку, забезпечувати гнучкість інтеграції з інфраструктурою кінотеатрів та дозволяти масштабування залежно від конкретних потреб.

Сервіс Multiplex (рис. 1.4) є прикладом сучасної української системи онлайн-бронювання та продажу квитків до кінотеатрів однойменної мережі. Його ключовою перевагою є повна інтеграція з внутрішніми процесами мережі Multiplex, що дозволяє забезпечити безперервний обіг інформації між клієнтською частиною сайту або мобільного додатку і бекстейдж-системами кінотеатру [5].

Інтерфейс платформи вирізняється зручністю користування, адаптивністю для мобільних пристроїв і підтримкою української мови, що робить її повністю придатною для внутрішнього ринку. Крім того, Multiplex активно використовує програми лояльності, персоналізовані пропозиції та push-сповіщення, що сприяє залученню та утриманню клієнтів. Можливість інтеграції з Apple Wallet або Google Pay для зберігання квитків, а також підтримка українських платіжних сервісів, таких як LiqPay, робить процес купівлі квитків швидким і безпечним.

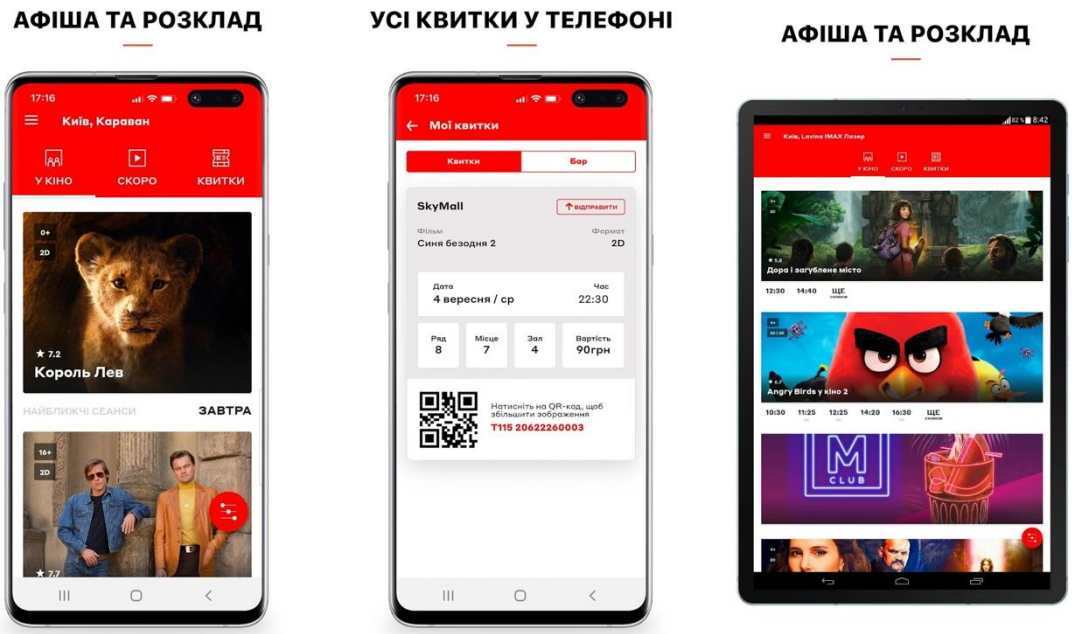

Рисунок 1.4 – Вікна програми «Multiplex»

Водночас, незважаючи на очевидну зручність і ефективність, система Multiplex має обмеження, які пов’язані з її закритістю та вузькою спрямованістю. Платформа призначена виключно для використання в межах власної мережі кінотеатрів, що унеможливлює її застосування іншими гравцями ринку без доступу до внутрішніх API або адміністративної панелі. Архітектура системи не передбачає можливості кастомізації або масштабування для сторонніх організацій, що ставить перепону для впровадження подібного рішення на регіональному рівні або в незалежних кінотеатрах. Крім того, її функціональність орієнтована на типову модель комерційного мультиплексу, без урахування потреб нестандартних форматів закладів, таких як малі кінохаби, open-air майданчики чи мобільні кінотеатри.

Отже, Multiplex є ефективною платформою для конкретної комерційної структури, яка реалізує централізовану модель управління. Проте для загального застосування на відкритому ринку потрібне більш гнучке та адаптивне рішення з відкритим API, можливістю локального розгортання і кастомізації під конкретні потреби кожного закладу. Це підтверджує доцільність створення нової

системи управління ресурсами кінотеатру, яка дозволить вирішити проблему сумісності, масштабованості та економічної ефективності.

Ще одним прикладом україномовного аналогу, який доцільно включити в порівняльний аналіз, є сервіс Київкінофільм – офіційна система бронювання та продажу квитків для муніципальних кінотеатрів Києва [7]. Цей вебсервіс (рис. 1.5) надає базову функціональність для користувачів: перегляд розкладу, ознайомлення з фільмами, вибір місць у залі та попереднє бронювання квитків.

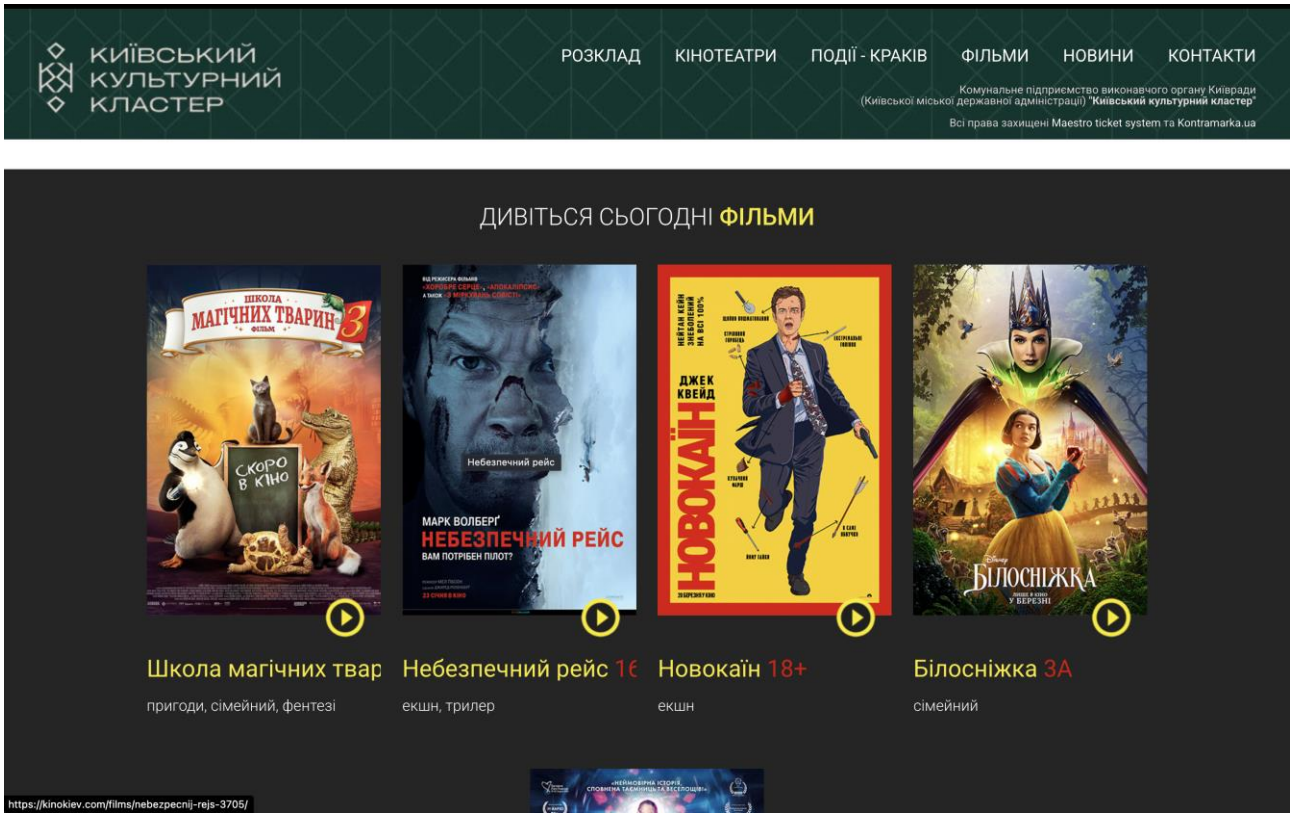

Рисунок 1.5 – Інтерфейс «Київкінофільм»

Інтерфейс сайту доступний українською мовою, що робить його зручним для внутрішньої аудиторії, а також відповідає вимогам локалізації державних та комунальних сервісів.

Основною перевагою системи є її простота та доступність – користувачам не потрібно проходити обов’язкову реєстрацію для оформлення бронювання. Це дозволяє швидко взаємодіяти з платформою навіть у випадках разового використання. Крім того, платформа орієнтована на доступні ціни та

популяризацію культурного дозвілля, що робить її зручною для широких верств населення.

Водночас система має низку функціональних обмежень. Вона не підтримує онлайн-оплату – користувач може лише забронювати квиток, а викупити його потрібно безпосередньо в касі перед початком сеансу. Це створює ризики втрати броні в разі запізнення або змін у планах. Крім того, відсутня підтримка мобільних застосунків, що суттєво знижує зручність для користувачів, які звикли до сучасного мобільного досвіду. Адміністративна частина платформи також обмежена – немає можливості управляти запасами товарів, змінювати склад персоналу чи будувати аналітичні звіти.

Отже, платформа Київкінофільм є прикладом мінімалістичного україномовного сервісу для бронювання квитків, проте її функціональність не охоплює всіх потреб сучасного кінотеатру. Це підтверджує доцільність створення розширеного, автономного та більш гнучкого рішення, яке дозволило б ефективно управляти як процесами продажу, так і ресурсами всередині закладу.

Таким чином, проведемо порівняльний аналіз між згаданими програмними засобами та власною розробкою (табл.1.3).

Таблиця 1.3 – Порівняльні характеристики програмних продуктів

Критерій	Fandango	Multiplex	Київкінофільм	Власна розробка
Підтримка української мови	–	+	+	+
Підтримка мобільного застосунку	+	+	–	+
Можливість онлайн-оплати	+	+	+	+
Інтеграція з національними платіжками	–	+	+	+
Наявність API для стороннього доступу	–	–	–	+
Підтримка кастомізації	–	–	–	+
Підтримка зовнішніх кінотеатрів	–	–	–	+
Програми лояльності та бонуси	+	+	–	+
Інтеграція з обліком запасів	–	–	–	+
Можливість роботи в офлайн-режимі	+	+	–	+
Загальна гнучкість системи	+	+	–	+
Сума	6	7	5	12

Проведений порівняльний аналіз програмних засобів, представлених у таблиці 1.3, свідчить про наявність суттєвих відмінностей між існуючими платформами та пропонованою власною розробкою системи управління ресурсами кінотеатру. Загальна кількість реалізованих функціональних критеріїв у сервісах Fandango, Multiplex та Київкінофільм становить відповідно 6, 7 та 5 пунктів із 11 можливих, що вказує на часткову реалізацію ключових можливостей, необхідних для сучасного кінотеатрального менеджменту.

У той же час власна розробка задовольняє всі визначені критерії, набираючи максимальні 12 балів, що підтверджує її повноцінність, універсальність і перспективність для впровадження.

Сервіс Fandango демонструє високу інтегрованість та підтримку мобільних платформ, однак його слабкими сторонами є відсутність української локалізації, неможливість адаптації під сторонні системи та недоступність для локальних операторів. Multiplex, як представник вітчизняного ринку, забезпечує високу якість обслуговування в рамках власної мережі, проте має обмеження у масштабуванні, розширенні та доступі для незалежних кінотеатрів. Київкінофільм, у свою чергу, є прикладом базового рішення з обмеженою функціональністю, орієнтованого переважно на муніципальні установи.

Власна система не лише включає підтримку всіх перелічених функцій, а й передбачає повну гнучкість архітектури: від доступу до API для сторонніх розробників і підтримки кастомізації до можливості інтеграції з обліком запасів, бонусною системою, офлайн-доступом і підтримкою будь-яких типів кінотеатрів – як мережевих, так і незалежних. Це робить її універсальним рішенням, яке відповідає сучасним вимогам ринку, забезпечує гнучкість адміністрування, адаптацію до бізнес-логіки закладу та спрощує масштабування в регіональному або національному масштабі.

Наведені характеристики підтверджують актуальність і конкурентоспроможність власної розробки, яка здатна не лише заповнити прогалини існуючих рішень, а й запропонувати якісно новий рівень автоматизації управління ресурсами кінотеатру.

1.4 Аналіз методів розв’язання завдання

Основні бізнес-процеси кінотеатру охоплюють підбір і програмування контенту, управління касовими операціями, продаж квитків, обслуговування відвідувачів, реалізацію продуктів харчування та напоїв, а також маркетинг і просування.

Процес вибору фільмів для показу ґрунтується на аналізі попиту, переговорах із дистриб'юторами та укладанні ліцензійних угод. Формування розкладу кіносеансів враховує популярність фільмів, прогнозований попит і наявність залів. Продаж квитків здійснюється через каси, термінали самообслуговування та онлайн-платформи, що дозволяє полегшити процес для клієнтів та забезпечити точність у відстеженні продажів.

Управління залами та забезпечення якості обслуговування включає контроль за наповненістю залів, підтримання чистоти та комфорту, координацію роботи персоналу для гарантування безпеки, надання допомоги відвідувачам і контроль дотримання правил. Продаж снеків і напоїв є значним джерелом доходу, де ефективність залежить від швидкості обслуговування та відповідності асортименту очікуванням клієнтів.

Маркетингові активності кінотеатру включають рекламні кампанії, спрямовані на залучення аудиторії, з використанням соціальних мереж, рекламних банерів та електронного маркетингу. Особливу увагу приділяється програмам лояльності, які допомагають залучати постійних клієнтів і стимулювати повторні відвідування [8]. Такі підходи сприяють підтриманню високого рівня інтересу до кінотеатру та розвитку стійких взаємовідносин із клієнтами.

Приклад схеми роботи декількох залів у кінотеатрі відображена на рисунку 1.6.

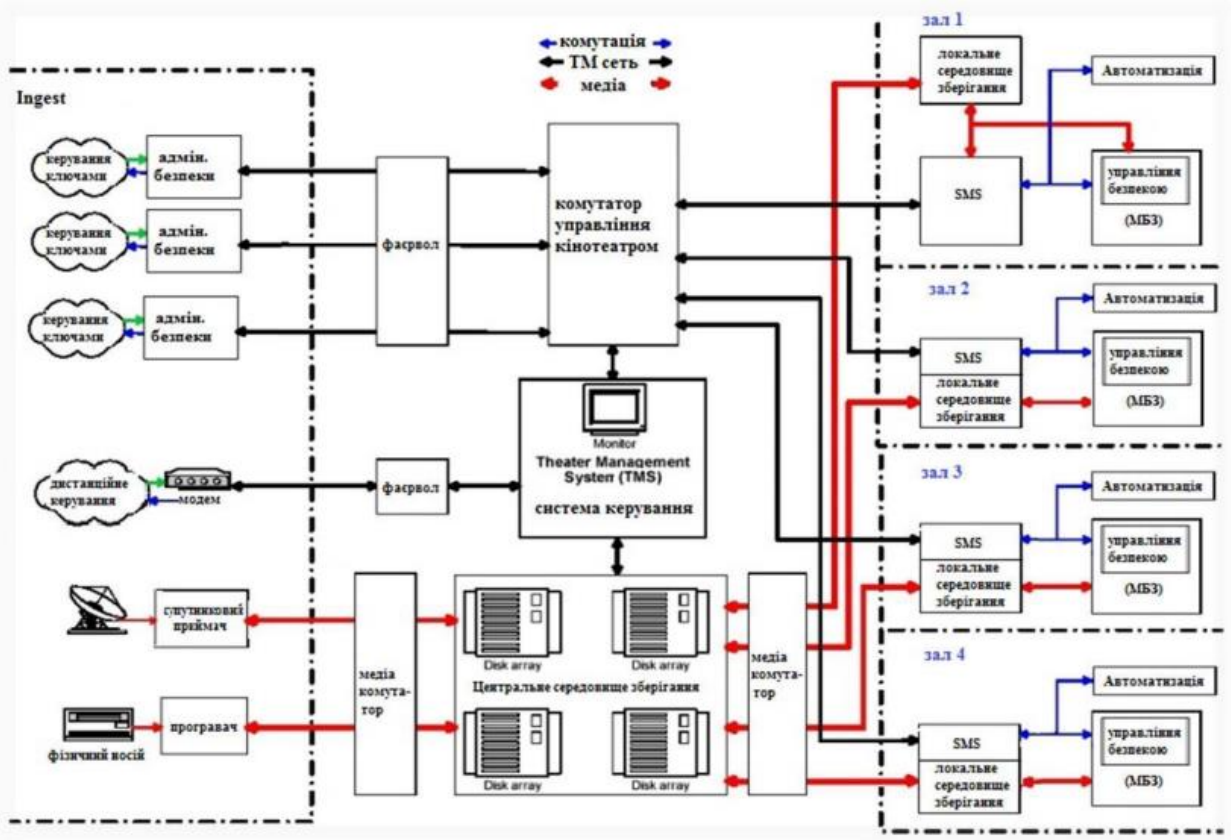

Рисунок 1.6 – Структура залів в кінотеатрі

Зали в кінотеатрі комфортні і мають широкі екрани. Основні виклики включають відсутність узгодженості між касовими та онлайн-продажами, що може призводити до дублювання або втрати бронювань. Крім того, є необхідність оптимізації управління персоналом для зниження витрат та підвищення продуктивності роботи [9].

Важливо також підвищити якість обслуговування клієнтів через впровадження мобільних додатків і сучасних інформаційних технологій.

Проведемо аналіз існуючою інформаційної системи управління ресурсами кінотеатру «Світ Сінема», яка представляє собою комплекс програмних і апаратних рішень, спрямованих на ефективну організацію і керування усіма аспектами діяльності кінотеатру. Система включає управління фінансовими, матеріально-технічними, людськими ресурсами, а також забезпечує високий рівень обслуговування клієнтів.

Для фінансового управління застосовується програмний продукт «QuickBooks» (рис. 1.7), що дозволяє вести бухгалтерський облік, зокрема фіксувати доходи і витрати, вести розрахунки з постачальниками та клієнтами, контролювати основні засоби та проводити інвентаризацію.

Це програмне забезпечення також підтримує фінансове планування, аналітику, створення прогнозів і оцінку фінансового стану кінотеатру. Завдяки таким рішенням «Світ Сінема» може ефективно керувати своїми фінансовими процесами, забезпечуючи стійкий розвиток і конкурентоздатність.

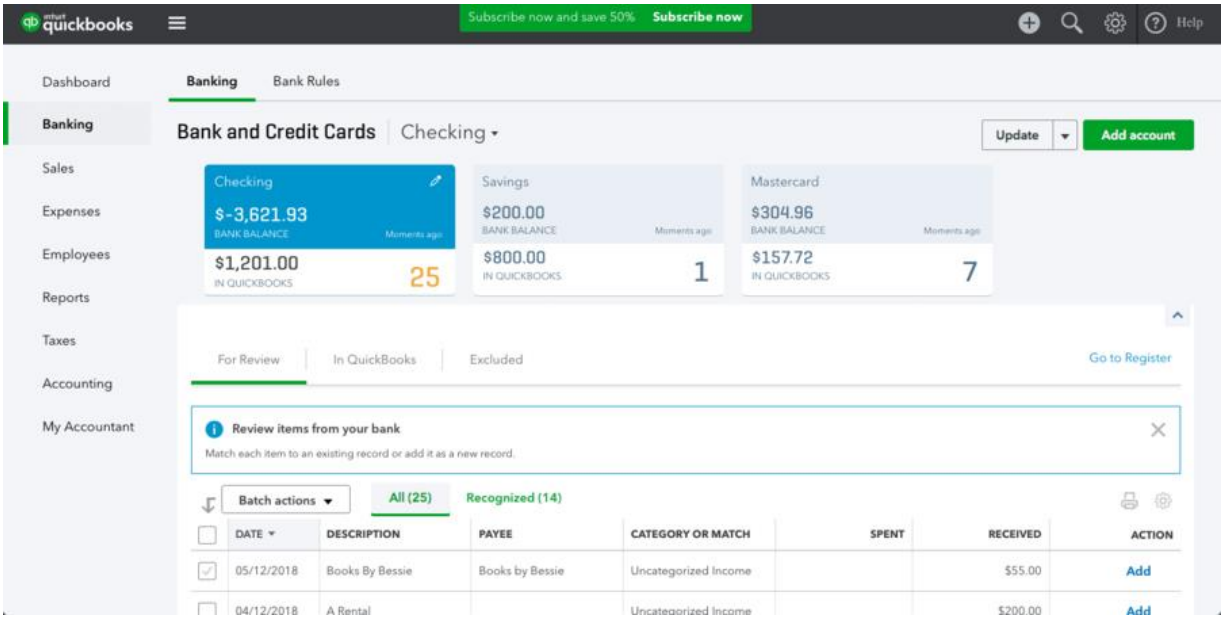

Рисунок 1.7 – Робоче вікно програмного забезпечення «QuickBooks»

Система «Ticketscloud» (рис. 1.8) використовується для автоматизації управління продажами та маркетингом, забезпечуючи можливість продажу квитків як через каси, так і через інтернет та мобільні додатки [10]. Ця платформа підтримує функції бронювання місць, вибір сеансів та розміщення у залі. Крім того, програми лояльності для постійних клієнтів управляються цією системою, що дозволяє організовувати акції та спеціальні пропозиції.

«Ticketscloud» також надає інструменти для аналізу ефективності маркетингових кампаній, управління рекламною діяльністю та проведення соціологічних опитувань серед клієнтів.

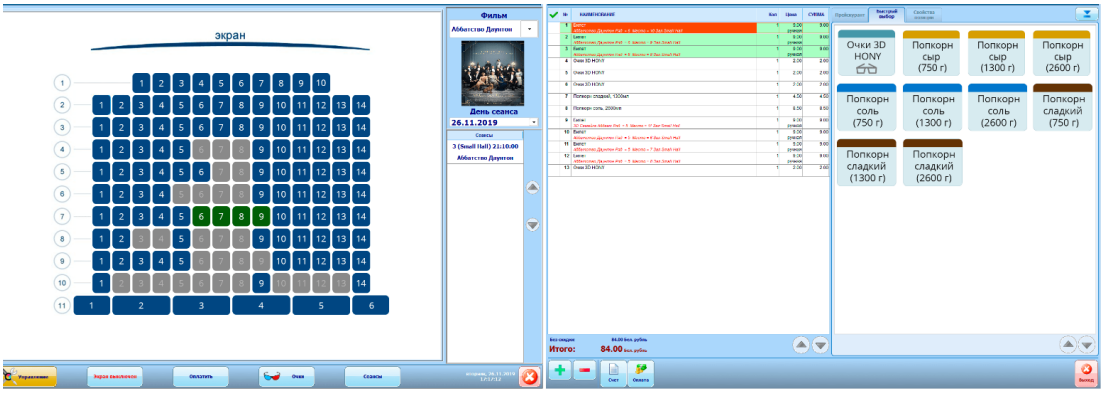

Рисунок 1.8 – Система продажів квитків та їжі

Захист інформації у кінотеатрі здійснюється за допомогою «McAfee Total Protection» (рис.1.9), яка включає функції шифрування даних, контроль доступу до інформаційної системи, резервне копіювання та відновлення інформації. Система також забезпечує постійний моніторинг для виявлення й усунення потенційних загроз безпеці [11].

Аналіз ефективності сучасного програмного забезпечення, що використовувалося кінотеатром у період з 2021 до 2024 року, охоплює оцінку інструментів для фінансового управління, продажів, маркетингу та захисту даних. Детально розглянемо такі програми: QuickBooks, Ticketscloud і McAfee Total Protection, їхні переваги й недоліки, динаміку використання протягом цих років та вплив на розвиток бізнесу.

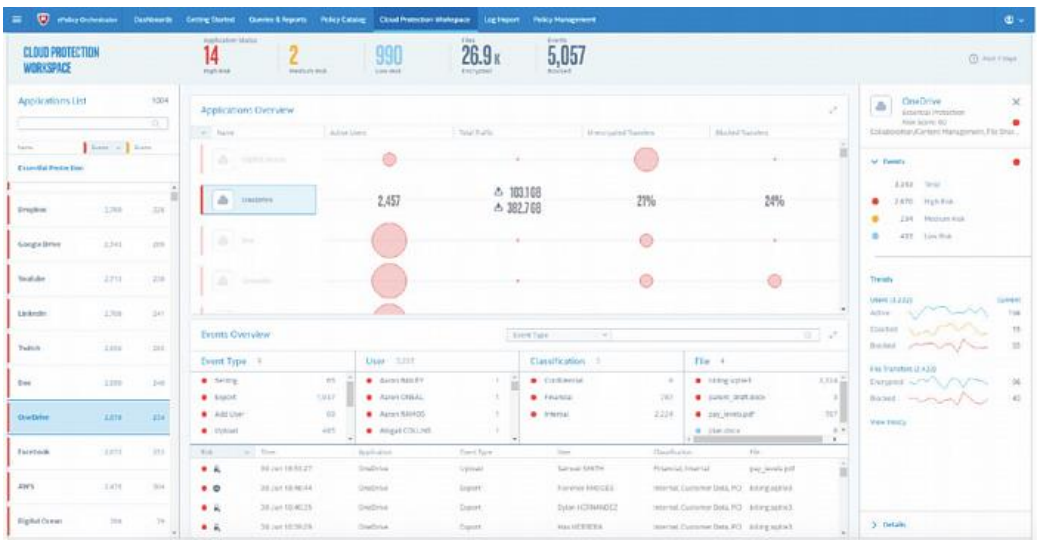

Рисунок 1.9 – Програма McAfee Total Protection

Програмне забезпечення «QuickBooks» використовувалося з 2021 по 2024 роки для ведення бухгалтерського обліку, включаючи облік доходів і витрат, розрахунки з постачальниками та клієнтами, а також облік основних засобів та інвентаризацію. Крім того, QuickBooks надавала можливість здійснювати фінансове планування, аналіз і прогнозування.

У 2021 році основна роль QuickBooks полягала у веденні бухгалтерського обліку та підготовці фінансової звітності, що дало можливість структурувати фінансові дані, скоротити витрати на бухгалтерські послуги на 15% і зменшити ймовірність помилок на 20%. Протягом 2022–2023 років функціонал програми було розширено – додано можливість прогнозування фінансових показників, що дозволило покращити фінансове планування та зменшити витрати на непередбачувані обставини на 10%. У 2024 році QuickBooks стала важливим інструментом для моніторингу фінансового стану підприємства та частково інтегрувала фінансові дані з іншими системами [12]. Простота управління бухгалтерією та можливість ведення обліку без залучення великої бухгалтерської команди дозволила зекономити на витратах на персонал близько 120 000 грн щорічно.

Система «Ticketscloud» автоматизувала процес продажу квитків через каси, інтернет та мобільні додатки, а також дозволила керувати програмами лояльності, проводити маркетингові кампанії та аналізувати їхню ефективність. У 2021 році Ticketscloud використовувалась переважно для продажу квитків, що підвищило зручність для клієнтів завдяки можливості онлайн-купівлі, що призвело до збільшення частки онлайн-продажів з 25% до 40%. У 2022 році функціонал був розширений для підтримки програм лояльності та акцій, що дозволило збільшити утримання клієнтів на 18%. У 2023–2024 роках Ticketscloud використовувалась для аналізу маркетингових кампаній і проведення соціологічних опитувань клієнтів, що дозволило покращити рекламні стратегії та підвищити конверсію на 12%. Серед переваг системи можна відзначити ефективне підвищення середнього чеку продажу на 10% за рахунок впровадження акцій і спеціальних пропозицій. Проте, недоліком є відсутність

прямої інтеграції з ERP або CRM-системами для комплексного аналізу даних. Також, у разі проблем із інтернет-з'єднанням, виникають труднощі з продажем квитків, що впливає на рівень обслуговування клієнтів – близько 2% клієнтів можуть втратити можливість купівлі через технічні збої.

«McAfee Total Protection» використовувалася для захисту даних кінотеатру з 2021 по 2024 роки, включаючи функції шифрування, контроль доступу, резервне копіювання та моніторинг загроз. У 2021 році McAfee забезпечувала базовий рівень безпеки, що включав антивірусний захист і контроль доступу. У 2022 році функціонал був розширений, зокрема додано резервне копіювання, що забезпечувало безперервність роботи й швидке відновлення даних у разі збоїв, що зменшило ризики втрат даних на 30%. У 2023–2024 роках McAfee зосередилася на вдосконаленні моніторингу загроз і забезпеченні відповідності сучасним стандартам кібербезпеки. Це дозволило знизити ймовірність кібератак на підприємство на 25% і зменшити кількість інцидентів, пов'язаних із безпекою, на 40%.

Необхідність впровадження інформаційної системи (ІС) управління ресурсами в кінотеатрі «Світ Сінема» має стратегічне обґрунтування, оскільки система (Enterprise Resource Planning) дозволяє інтегрувати всі основні бізнес-процеси підприємства в єдину платформу. Це охоплює такі ключові аспекти діяльності, як управління фінансами, продаж квитків, маркетинг, інвентаризацію, обслуговування клієнтів і кадрову політику. Впровадження такої системи надає суттєві переваги, що в перспективі покращать операційну ефективність та конкурентоспроможність підприємства.

Далі додамо таблицю 1.4, яка підсумовує використання кожного програмного забезпечення, його переваги, недоліки та вартість. Також обґрунтуємо необхідність впровадження інформаційної системи (ІС) управління ресурсами в кінотеатрі для підвищення ефективності роботи та забезпечення стійкого розвитку бізнесу.

Таблиця 1.4 – Оцінка ефективності використовуваного програмного забезпечення в кінотеатрі (2021–2024)

Програмне забезпечення	Функції	Переваги	Недоліки	Вартість обслуговування
QuickBooks	Фінансове управління, облік доходів та витрат, прогнозування	Інтегровані інструменти обліку Фінансове планування	Обмежені можливості масштабування Інтеграція з іншими системами вимагає додаткових модулів	300–700 доларів/рік
Ticketscloud	Продаж квитків, програми лояльності, маркетинг	Універсальність продажів Програми лояльності та маркетинг	Обмежена інтеграція з іншими системами Залежність від інтернету	100–500 доларів/місяць
McAfee Total Protection	Захист даних, контроль доступу, шифрування	Повний захист системи Захист конфіденційних даних	Високе споживання ресурсів Необхідність регулярних оновлень	100–150 доларів/рік

Система дозволяє об'єднати всі основні процеси кінотеатру в одну інтегровану інформаційну платформу. Це допомагає зменшити дублювання даних та підвищити їхню узгодженість, що, у свою чергу, дозволяє знизити витрати на управління інформацією на 25%. Крім того, це сприяє більш точному управлінню фінансовими та матеріальними ресурсами, оптимізуючи процеси інвентаризації та управління продажами квитків. Інформаційна система управління забезпечує централізоване управління ресурсами підприємства, включаючи планування розкладу показів, управління персоналом і контроль закупівель. Це дозволяє краще використовувати наявні ресурси та оптимізувати витрати на розподіл ресурсів на 20%. Наприклад, оптимізація графіка роботи персоналу допомагає скоротити витрати на оплату праці приблизно на 10-15% завдяки більш ефективному розподілу змін.

система надає точні дані в режимі реального часу, що дозволяє керівництву швидко приймати обґрунтовані рішення на основі актуальних показників. За допомогою аналітичних інструментів керівники можуть відстежувати ключові показники ефективності, аналізувати рентабельність кожного сеансу, оцінювати ефективність програм лояльності та маркетингових кампаній. Наприклад, зниження витрат на маркетингові кампанії на 15% стало можливим завдяки аналізу даних щодо ефективності рекламних акцій та можливості швидкої корекції стратегії. Інтеграція всіх бізнес-процесів у єдину систему сприяє зниженню витрат на обробку інформації, оскільки система автоматизує більшість рутинних завдань. Це дозволяє зменшити витрати на оплату праці і уникнути помилок, спричинених людським фактором, на 20-30%. Крім того, це зменшує адміністративні витрати, пов’язані з ручною обробкою даних, та підвищує точність фінансової звітності. система може значно підвищити рівень обслуговування клієнтів. Наприклад, автоматизація продажів квитків дозволяє надавати точну інформацію про доступність місць у режимі реального часу, що знижує ймовірність дублювання продажів та підвищує рівень задоволення клієнтів на 10-15%. Наявність актуальної інформації про акції та знижки допомагає залучати нових відвідувачів та стимулювати повторні візити, що, у свою чергу, збільшує доходи на 12%.

Станом на сьогодні відсутність єдиної інформаційної системи управління ресурсами спричиняє ряд проблем. Зокрема, використання різних програмних рішень для окремих бізнес-процесів (наприклад, QuickBooks для фінансів і Ticketscloud для продажу квитків) призводить до розрізненості даних, що ускладнює їх аналіз і, відповідно, процес прийняття стратегічних рішень. У випадку зростання обсягів продажу квитків або розширення бізнесу, відсутність єдиної системи ускладнює управління більшою кількістю клієнтів, залів і співробітників, що може призвести до збільшення операційних витрат на 15-20%.

Вибір системи, яка підтримує інтеграцію з уже існуючими системами (такими як QuickBooks і Ticketscloud), є важливим аспектом для збереження

неперервності роботи. рішення має підтримувати можливість інтеграції з іншими компонентами кінотеатру та забезпечувати функціонування без збоїв під час переходу. Це дозволить знизити ризик втрати даних під час впровадження нових систем на 25%. Загалом, впровадження інформаційної системи управління ресурсами дозволить «Світ Сінема» забезпечити більш ефективне управління бізнес-процесами, підвищити узгодженість та прозорість даних, що в свою чергу сприятиме покращенню якості обслуговування клієнтів, скороченню витрат і збільшенню прибутковості компанії. Інтеграція всіх функцій у єдину систему є важливим кроком для забезпечення стійкого розвитку та конкурентоспроможності на ринку.

Таким чином, для підвищення ефективності роботи кінотеатру рекомендується впровадити комплексне рішення для інтеграції всіх ключових бізнес-процесів. Важливо також розширити маркетингові можливості системи та забезпечити високий рівень безпеки даних для зменшення кіберризиків і захисту інформації підприємства.

1.5 Постановка задач розробки системи для системи управління ресурсами кінотеатру

Розробка інформаційної системи управління ресурсами кінотеатру потребує чіткого визначення мети та кола завдань, спрямованих на реалізацію ефективної автоматизації бізнес-процесів. Насамперед необхідно формалізувати вимоги, враховуючи аспекти управління розкладами сеансів, продажу квитків, контролю запасів (наприклад, снеків), а також планування і роботи персоналу. Далі слід розробити модель предметної області, що відображатиме ключові сутності та забезпечуватиме зв’язки між ними, з урахуванням операцій CRUD і потреб у масштабованості.

Важливим завданням є розробка архітектури застосунку, яка втілюватиме вибір сучасних інструментів (зокрема, Spring Boot на бекенді, React із TypeScript на фронтенді) і забезпечуватиме гнучкість обробки даних завдяки REST API й

підтримці інтеграційного механізму. Одночасно слід реалізувати надійну систему автентифікації та авторизації з використанням JWT, що надасть змогу розмежувати права доступу залежно від ролі користувача (адміністратор, касир, клієнт). Потрібно також передбачити можливість під'єднання платіжних сервісів для онлайн-оплати й унормувати процедуру взаємодії з зовнішніми модулями (наприклад, системами лояльності або сервісами розсилки повідомлень).

Необхідно створити інтуїтивно зрозумілий інтерфейс користувача, де відображається інформація про фільми та їх розклади, а також контроль запасів чи розрахунків, передбачаючи підтримку кількох мов (української, англійської) та можливість використання технологій PWA для мобільного досвіду. На завершальному етапі важливо провести тестування й налагодження програмного рішення, аби перевірити його продуктивність, надійність та відповідність функціональним вимогам. Після цього доцільно розглянути перспективи подальшого розвитку системи й інтеграції з іншими сервісами або переходу до мікросервісної архітектури, якщо обсяги даних чи навантаження суттєво зростатимуть.

1.6 Висновки

У першому розділі обґрунтовано актуальність теми дослідження, зокрема потребу у створенні сучасної інформаційної системи управління ресурсами кінотеатру, яка враховує особливості українського ринку. Було охарактеризовано специфіку управління ресурсами в кінотеатральній сфері, вказано на виклики, що виникли внаслідок пандемії COVID-19, ситуації в країні, а також конкуренції зі стрімінговими сервісами. Підкреслено важливість цифрової трансформації кінотеатрів задля забезпечення стабільної роботи, підвищення ефективності та конкурентоспроможності.

Розглянуто основні типи інформаційних систем, що використовуються в управлінні ресурсами кінотеатрів, включаючи квиткові системи, CRM, CMS, ERP та маркетингові платформи. Проведено детальний огляд функціональних

вимог до таких систем, визначено їх ключові компоненти та особливості інтеграції. Представлено порівняльний аналіз існуючих рішень – Fandango, Multiplex та Київкінофільм. За результатами аналізу встановлено, що жодна з розглянутих систем не забезпечує повного охоплення потреб сучасного кінотеатру, особливо з огляду на підтримку кастомізації, API, локалізації та інтеграції з обліком запасів.

Оцінено програмне забезпечення, що вже використовується в кінотеатрі. На основі аналізу їхнього функціоналу, переваг, недоліків та вартості обслуговування зроблено висновок про фрагментарність існуючої IT-інфраструктури та обґрунтовано необхідність впровадження єдиної системи для оптимізації внутрішніх процесів, покращення якості обслуговування клієнтів і зменшення операційних витрат.

Поставлено завдання для подальшої розробки інформаційної системи, визначено ключові функціональні модулі, технології реалізації та архітектурні вимоги. Сформульовано технічне завдання, що забезпечить підґрунтя для наступних етапів проектування і реалізації програмного рішення.

2 ПРОЄКТУВАННЯ СИСТЕМИ УПРАВЛІННЯ РЕСУРСАМИ КІНОТЕАТРУ

2.1 Розробка структури програмного модулю системи управління ресурсами кінотеатру

У процесі створення сучасних інформаційних систем особливу увагу необхідно приділяти розробці їх внутрішньої структури. Від того, наскільки продумано організовано архітектуру програмного модуля, залежить не лише стабільність функціонування всієї системи, а й її здатність до масштабування, адаптації до нових вимог, зручність підтримки та інтеграції з іншими сервісами [13]. Це особливо актуально для складних багатокомпонентних систем, таких як система управління ресурсами кінотеатру, яка повинна забезпечувати безперервну взаємодію між численними користувачами, ефективну обробку даних у реальному часі та підтримку критично важливих бізнес-процесів [14, 15].

Вибір сучасного фреймворку Jhipster [16] як платформи для проєктування програмного модуля є обґрунтованим кроком, що дозволяє інтегрувати перевірені інструменти й шаблони для генерації надійного коду з урахуванням кращих практик у сфері веброзробки. Монолітна архітектура, реалізована на основі Spring Boot [17], забезпечує цілісність системи, спрощує управління залежностями та дає змогу сконцентрувати ресурси на оптимізації логіки програми без необхідності підтримки окремих сервісів на початковому етапі.

Важливим аспектом побудови такої структури є узгодженість між клієнтською та серверною частинами. Реалізація фронтенду з використанням React та TypeScript сприяє створенню гнучкого та інтерактивного інтерфейсу, що забезпечує користувачам швидкий та інтуїтивно зрозумілий доступ до всіх функцій системи. У свою чергу, серверна частина, завдяки використанню REST API, дозволяє чітко організувати логіку обробки даних та взаємодію з базою, авторизацією й зовнішніми системами.

Розробка структури програмного модуля виступає ключовим етапом у формуванні повнофункціональної системи управління кінотеатром [19]. Вона дозволяє не лише реалізувати вимоги до роботи з базовими бізнес-сутностями, але й забезпечує технологічну основу для подальшого розвитку – додавання нових можливостей, переходу до мікросервісної архітектури або інтеграції зі сторонніми платформами. Тому структура програмного модуля, представлена у цьому розділі, є результатом глибокого аналізу та практичної оптимізації відповідно до сучасних вимог інженерії програмного забезпечення.

У результаті проведеного дослідження [20] розроблено детальну структуру програмного модуля системи управління ресурсами кінотеатру на основі фреймворку JHipster 8.0. Вибір технології JHipster з монолітною архітектурою дозволив створити інтегроване рішення з високою ефективністю, забезпечуючи стабільну роботу веб-застосунку та можливість подальшого масштабування.

Отримана структура застосунку:

```
application {
  config {
    jhipsterVersion "8.0.0"
    baseName cinemaCityManagement
    applicationType monolith
    buildTool maven
    packageName com.cinemacity.management
    authenticationType jwt
    databaseType sql
    devDatabaseType h2Disk
    prodDatabaseType postgresql
    enableHibernateCache true
    enableTranslation true
    clientPackageManager npm
    languages [en, ua]
    nativeLanguage ua
    dtoSuffix DTO
    enableSwaggerCodegen true
    messageBroker kafka
    reactive false
    serviceDiscoveryType no
    skipClient false
    skipServer false
    skipUserManagement false
    websocket false
    serverPort 8080
    testFrameworks [junit]
  }
  entities *
}
```

Структура програмного модуля складається з таких компонентів, як фронтенд-частина, реалізована за допомогою React, та бекенд-частина, яка

базується на Spring Boot із postgresql. React-інтерфейс відповідає за взаємодію з користувачем, забезпечуючи швидке відображення інформації та інтуїтивно зрозуміле управління ресурсами кінотеатру. Використання TypeScript дозволило знизити кількість помилок та покращити підтримуваність коду [21, 22].

Бекенд-частина включає REST API, що реалізує CRUD-операції для ключових сутностей, таких як фільми, зали, сеанси, бронювання місць, працівники та клієнти. Використання JWT-аутентифікації гарантує безпечний доступ до ресурсів системи та забезпечує високий рівень захисту персональних даних користувачів [23]. Також до структури входить база даних, яка працює з технологією Hibernate [24], що забезпечує ефективну роботу з даними та високу продуктивність запитів.

Додатково реалізовано інтеграцію Apache Kafka [25] для ефективного управління повідомленнями та подіями в реальному часі, що суттєво підвищує швидкість реакції системи на зміну стану ресурсів кінотеатру. В системі передбачена підтримка багатомовності, що сприяє розширенню потенційної аудиторії користувачів та покращенню загального користувацького досвіду [26].

Комплексна модель предметної області включає сукупність сутностей, які максимально точно відображають бізнес-процеси діяльності кінотеатру [27]. Сутність Movie містить важливі характеристики фільмів, такі як назва, опис, тривалість показу, дата прем'єри та рейтинг, що дозволяє створювати афіші та проводити каталогізацію кінофільмів. Сутність Hall описує приміщення кінотеатру, включаючи назву та місткість кожного залу, тоді як Seat відповідає за управління місцями у залах, зберігаючи інформацію про номер, ряд і статус доступності конкретного місця. Screening використовується для планування сеансів, визначаючи конкретні дату, час та ціну перегляду фільму. Сутність Ticket забезпечує облік квитків з інформацією щодо дати покупки, вартості та поточного статусу (куплений, використаний, скасований), що важливо для контролю продажів та відвідуваності. Snack охоплює асортимент закусок і напоїв, доступних до покупки, а Inventory забезпечує контроль над запасами продукції, відслідковуючи рівень товарів і вчасне їх поповнення. Employee

дозволяє ефективно управляти персоналом кінотеатру, зберігаючи особисті дані, посади та дати прийняття на роботу співробітників. Schedule застосовується для ведення робочого графіка працівників, що включає час початку та завершення робочих змін. Сутність Sale відповідає за фінансовий облік та містить дані щодо дати продажу та сум проведених операцій. Customer включає в себе інформацію про клієнтів, зокрема контактні дані та відомості для впровадження програм лояльності. Це дозволяє проводити цілеспрямовані маркетингові кампанії та покращувати якість взаємодії з клієнтами кінотеатру.

Запропонована структура запропонована на рисунку 2.1.

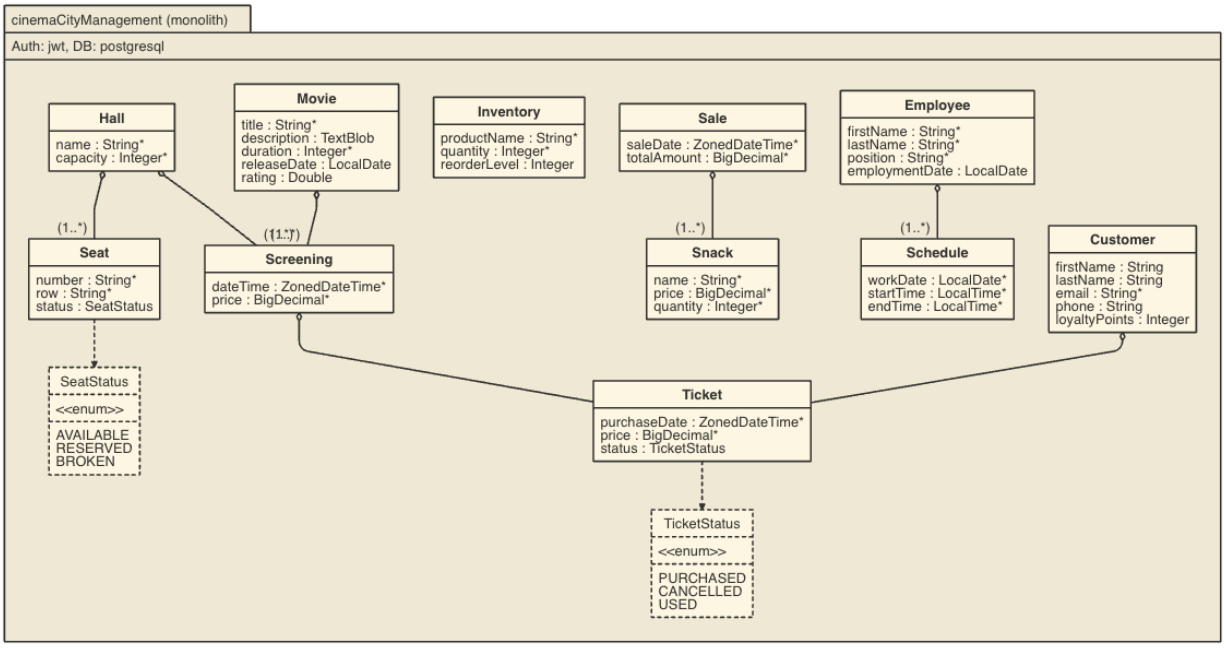

Рисунок 2.1 – Структура програмного модулю системи управління ресурсами кінотеатру

Для забезпечення якості та стабільності роботи застосунку структура включає автоматизовані тести з використанням фреймворку JUnit, що дозволяє регулярно контролювати працездатність компонентів системи та вчасно виявляти помилки. Для документування REST API використовується Swagger, що спрощує розробку та підтримку API, забезпечуючи зрозумілу інтерактивну документацію [28].

Використання JHipster для розробки системи управління ресурсами кінотеатру дозволило значно прискорити процес створення програмного рішення завдяки використанню готових шаблонів та інтегрованих технологій. JHipster є сучасним генератором веб-застосунків, який комбінує переваги технологій Java на бекенді та популярних JavaScript фреймворків на фронтенді, що робить його особливо зручним для реалізації повноцінних інтегрованих систем.

У розробці системи управління ресурсами кінотеатру було використано восьму версію JHipster, яка пропонує монолітну архітектуру з використанням Java Spring Boot для бекенд-частини та React з TypeScript на фронтенді. Обраний монолітний тип архітектури забезпечує простоту розгортання та управління застосунком на початкових етапах розробки і впровадження. Інструмент надає автоматично створювану основу проєкту, включаючи структуру REST API, налаштування бази даних, механізми автентифікації, а також налаштоване робоче середовище для тестування та документування коду.

Для побудови програмного забезпечення було вибрано Maven як інструмент автоматизації збірки проєкту, що забезпечує стабільне керування залежностями та процесами інтеграції. Для розробки бази даних застосовано технологію Hibernate із підтримкою кешування, що дозволяє значно скоротити час відповіді на запити та підвищити продуктивність роботи системи. Для середовища розробки використано базу даних H2, тоді як для промислового використання – PostgreSQL, що дозволяє забезпечити стабільність та надійність роботи з даними.

З метою безпеки та керування доступом було застосовано JWT-аутентифікацію, яка забезпечує безпечну взаємодію користувачів із сервером та дозволяє ефективно контролювати доступ до різних функцій системи. Використання JWT-токенів дозволяє розмежувати рівні доступу та надає гнучку систему налаштувань прав доступу для різних категорій користувачів: клієнтів, адміністраторів та працівників кінотеатру.

Використання Apache Kafka як брокера повідомлень дозволяє ефективно керувати потоками подій та обміном даними між компонентами системи. Kafka надає можливість реалізації асинхронної взаємодії сервісів, що значно підвищує швидкість роботи застосунку та дозволяє оперативно реагувати на зміни стану сутностей системи, таких як бронювання квитків або зміни у розкладі кіносеансів.

Додатково JHipster надає інтегровану підтримку багатомовності. Реалізована система підтримує українську та англійську мови, що забезпечує зручний та зрозумілий інтерфейс як для локальних, так і для міжнародних користувачів. Це дозволяє значно розширити потенційну аудиторію застосунку, залучаючи більше клієнтів та покращуючи загальний користувацький досвід.

Таким чином, використання JHipster для розробки системи управління ресурсами кінотеатру дозволило отримати цілісну, ефективну та масштабовану програмну платформу, яка повністю відповідає сучасним вимогам розробки веб-застосунків та забезпечує гнучкість і швидкість впровадження нових функцій. Цей інструмент виявився оптимальним вибором для реалізації бізнес-процесів кінотеатру завдяки великій кількості інтегрованих технологій та можливості подальшого розвитку й масштабування створеного рішення.

Таким чином, розроблена структура програмного модуля із застосуванням JHipster 8.0 дозволяє ефективно реалізувати бізнес-процеси управління ресурсами кінотеатру, забезпечуючи при цьому високу надійність, продуктивність та масштабованість.

2.2 Розробка ключових бізнес-процесів та проєктування системи управління ресурсами кінотеатру

У сучасному процесі проєктування інформаційних систем надзвичайно важливою складовою є побудова UML-діаграм, які дозволяють формалізувати логіку функціонування програмного забезпечення, структуру його компонентів та способи взаємодії між ними. Розробка таких діаграм є не лише способом

документування системи, а й дієвим інструментом для верифікації архітектурних рішень, забезпечення узгодженості між логічною моделлю та реалізацією, а також для полегшення комунікації між членами команди розробки, аналітиками та зацікавленими сторонами.

UML (Unified Modeling Language) діаграми охоплюють як структурні, так і поведінкові аспекти системи. Завдяки їм можна описати структуру класів, зв’язки між об'єктами, варіанти використання, алгоритми дій, зміни станів об'єктів у часі, сценарії взаємодії компонентів та фізичну архітектуру програмного комплексу. Такий підхід дозволяє перейти від неформального опису вимог до формальної моделі, яка є базисом для якісної реалізації програмного продукту.

Для розробки системи управління ресурсами кінотеатру побудова UML-діаграм сприяє чіткому відображенню ключових бізнес-процесів, таких як вибір і бронювання місць, управління сеансами, облік клієнтів, продаж квитків та обробка оплат. Це дозволяє моделювати як типові сценарії взаємодії користувачів із системою, так і внутрішню логіку обробки даних та підтримку адміністративних функцій.

Доцільно розглянути низку діаграм, що розкривають структуру та поведінку системи: діаграма послідовності (Sequence Diagram), діаграма варіантів використання (Use Case Diagram), діаграма класів (Class Diagram), діаграма об’єктів (Object Diagram), діаграма діяльності (Activity Diagram), компонентна діаграма (Component Diagram), діаграма розгортання (Deployment Diagram), діаграма станів (State Diagram) та діаграма часу (Timing Diagram). Кожна з них розроблена відповідно до вимог стандарту UML та адаптована до предметної області – управління ресурсами кінотеатру.

Кожна із запропонованих діаграм відіграє специфічну роль у моделюванні системи: від опису сценаріїв взаємодії користувачів до формування фізичної моделі її розгортання в середовищі виконання. Їхня розробка дає змогу підвищити надійність, масштабованість і прозорість архітектури застосунку, що

є запорукою ефективної розробки та подальшої підтримки програмного забезпечення.

2.2.1 Розробка UML Sequence Diagram

Sequence Diagram, або діаграма послідовності, є одним із ключових типів UML-діаграм, що використовується для візуалізації взаємодії між різними частинами системи в хронологічному порядку.

Наведена діаграма на рисунку 2.2 демонструє процес вибору та бронювання квитків у системі управління ресурсами кінотеатру.

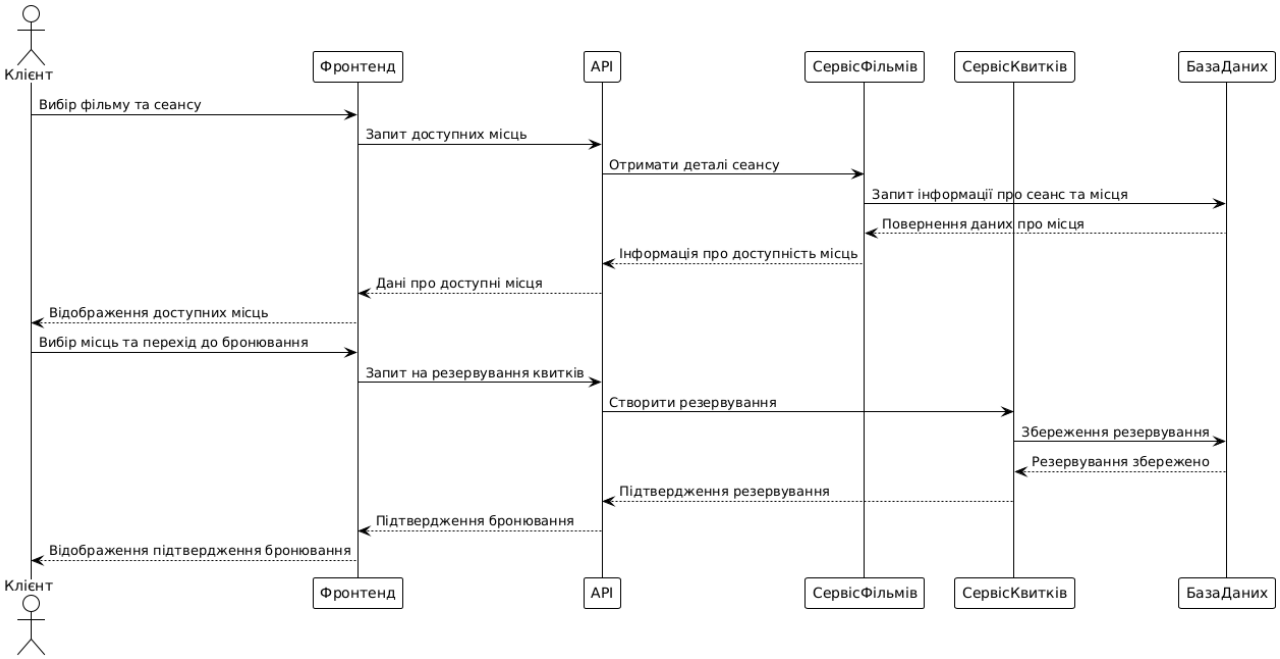

Рисунок 2.2 – UML-діаграма послідовності (Sequence Diagram) взаємодії компонентів системи управління ресурсами кінотеатру

На діаграмі представлені наступні елементи: Клієнт, Фронтенд, API, СервісФільмів, СервісКвитків та БазаДаних. Взаємодія починається з вибору клієнтом фільму та сеансу через інтерфейс користувача. Після цього фронтенд надсилає запит до API для отримання доступних місць.

Ця діаграма демонструє етапи взаємодії між користувачем системи та її основними компонентами під час процесу вибору фільму, перегляду доступних місць і подальшого бронювання квитків. Вона є ключовим елементом моделювання логіки взаємодії в інформаційній системі управління ресурсами кінотеатру.

Учасники (актор та компоненти):

- Клієнт – кінцевий користувач, який взаємодіє з системою через інтерфейс.
- Фронтенд – користувацький інтерфейс, реалізований у вигляді SPA-додатку на React, що відповідає за обробку дій клієнта.
- API – серверна точка взаємодії, яка приймає запити від фронтенду і спрямовує їх до відповідних сервісів.
- СервісФільмів – бекенд-сервіс, що обробляє запити, пов’язані з фільмами, сеансами та доступністю місць.
- СервісКвитків – сервіс, що відповідає за логіку бронювання квитків.
- БазаДаних – централізоване сховище даних, що містить інформацію про всі сутності системи.

Етап 1 – Отримання доступних місць:

1. Клієнт обирає фільм і відповідний сеанс через Фронтенд.
2. Фронтенд надсилає запит на API для отримання списку вільних місць.
3. API звертається до СервісФільмів, щоб дізнатися інформацію про конкретний сеанс.
4. СервісФільмів здійснює запит до БазиДаних для отримання даних про сеанс та стан місць.
5. БазаДаних повертає відповідь із інформацією про місця до СервісФільмів.
6. СервісФільмів формує відповідь і надсилає її до API.
7. API пересилає дані до Фронтенду, де Клієнт бачить список доступних місць.

Етап 2 – Бронювання квитків:

1. Клієнт обирає конкретні місця й ініціює бронювання.
2. Фронтенд передає запит до API для створення бронювання.
3. API направляє запит до СервісКвитків, що відповідає за обробку бронювання.
4. СервісКвитків зберігає нове бронювання у БазіДаних.
5. БазаДаних підтверджує успішне збереження даних.
6. СервісКвитків надсилає підтвердження назад до API.
7. API повідомляє Фронтенд про успішне бронювання.
8. Фронтенд відображає Клієнту підтвердження з деталями замовлення.

Розглянемо переваги такої діаграми.

- Дозволяє візуалізувати логіку взаємодії між компонентами системи.
- Дає змогу виявити зайві або дубльовані запити.
- Підтримує ефективне документування бізнес-логіки.
- Полегшує комунікацію між розробниками, дизайнерами та аналітиками.

Таким чином, діаграма чітко відображає порядок взаємодій між компонентами системи, допомагаючи зрозуміти, як саме відбувається процес обробки бронювання квитків, а також дозволяє виявити потенційні проблемні місця та оптимізувати роботу системи.

2.2.2 Розробка UML Use-case Diagram

Use-case діаграма – це один з основних типів UML-діаграм, який використовується для моделювання взаємодії користувачів (акторів) з інформаційною системою. Вона дозволяє візуально представити функціональні вимоги до системи через прості сценарії використання (use cases), що описують, як система повинна реагувати на дії користувача.

У контексті системи бронювання кінотеатру, представлена use-case діаграма відображає типові дії користувачів різних ролей – Клієнта, Адміністратора та Касира – та функціональність, яку система має підтримувати. Кожна роль (актор) зображена у вигляді чоловічка зліва від системи, а сама система представлена у вигляді прямокутника з усіма варіантами використання, які ця система виконує.

Діаграма варіантів використання (Use Case Diagram) моделює основні функціональні можливості системи бронювання кінотеатру. Вона створена відповідно до стандартів UML та вимог ВНТУ, з дотриманням структури, де актори розташовані ліворуч, використані коректні зв'язки <<include>> та <<extend>>, а варіанти використання згруповані в межах прямокутника системи.

Use Case діаграма наведена на рисунку 2.3.

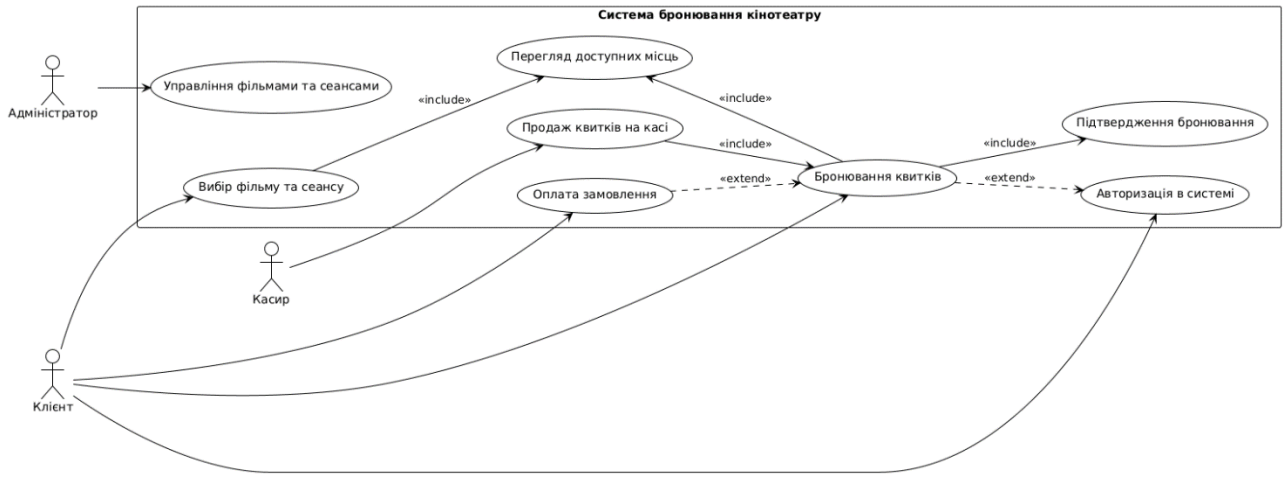

Рисунок 2.3 – Use Case діаграма

Розглянемо більш детально акторів діаграми.

- Клієнт – звичайний користувач системи, який взаємодіє з нею через інтерфейс для вибору фільмів, бронювання та оплати квитків.
- Касир – працівник, який здійснює продаж квитків безпосередньо через касу, використовуючи внутрішній доступ до функцій бронювання.
- Адміністратор – відповідальна особа, яка керує репертуаром, фільмами та розкладом сеансів.

Основні варіанти використання (use cases):

- “Вибір фільму та сеансу” (ВибірСеансу) – стартовий сценарій, у якому клієнт обирає бажаний фільм із доступного репертуару. Обов’язково включає перегляд місць.
- “Перегляд доступних місць” (ПереглядМісць) – підпроцес, який автоматично виконується при виборі фільму або бронюванні. Покращує інформування користувача перед покупкою.
- “Бронювання квитків” (БронюванняКвитків) – ключовий процес, який реалізується як клієнтом, так і касиром. Включає перегляд місць та підтвердження.
- “Підтвердження бронювання” (Підтвердження) – логічне завершення бронювання, що виконується перед оплатою.
- “Авторизація в системі” (Авторизація) – розширення (<<extend>>) для дій, які потребують підтвердження особи (наприклад, бронювання чи оплата квитків).
- “Оплата замовлення” (Оплата) – розширення до сценарію бронювання, яке виконується опціонально, залежно від моделі бізнес-процесу (наприклад, бронювання без попередньої оплати).
- “Продаж квитків на касі” (ПродажНаКасі) – специфічний сценарій для касира, що використовує вже наявну логіку бронювання.
- “Управління фільмами та сеансами” (УправлінняФільмами) – адміністративний варіант використання, що дозволяє адміністратору додавати, змінювати або видаляти фільми та розклад.

Розглянемо зв’язки між use cases.

- <<include>> позначає обов’язкове включення одного варіанту в інший. Наприклад, бронювання квитка обов’язково включає перегляд місць та підтвердження.
- <<extend>> використовується для позначення додаткових (необов’язкових) сценаріїв. Наприклад, оплата не завжди потрібна для бронювання – її реалізація залежить від налаштувань системи.

Клієнт має можливість обирати фільм і сеанс, переглядати доступні місця, здійснювати бронювання, авторизовуватися в системі та оплачувати замовлення. Деякі функції доповнюють одна одну за допомогою зв’язків <<include>> або <<extend>>. Наприклад, перед бронюванням обов’язковим є перегляд доступних місць (<<include>>), а оплата є необов’язковим продовженням бронювання (<<extend>>).

Адміністратор системи має доступ до функцій управління контентом – зокрема, фільмами та сеансами, що відображаються клієнтам. Касир, у свою чергу, виконує продаж квитків на касі, що включає частину дій, аналогічних клієнтському бронюванню (наприклад, створення замовлення).

Використання <<include>> показує обов’язковість виконання певного підпроцесу в межах основного варіанту використання. Наприклад, “Бронювання квитків” обов’язково включає “Перегляд місць” і “Підтвердження бронювання”. Зв’язок <<extend>> використовується, коли додаткова функція не є обов’язковою, але може бути виконана за певних умов – як, наприклад, авторизація або оплата.

Загалом, така діаграма дозволяє на початковому етапі розробки чітко сформулювати вимоги до системи, визначити ключових користувачів і передбачити логіку взаємодії з програмним продуктом. Це робить її важливим інструментом у процесі проєктування і розробки систем, зокрема систем управління кінотеатром.

2.2.3 Розробка UML Class Diagram

Діаграма класів є одним з ключових інструментів моделювання програмного забезпечення в об'єктно-орієнтованому підході. Вона дозволяє візуально представити структуру системи, показуючи класи, їх атрибути, методи та зв’язки між ними. Основна мета цієї діаграми – описати статичну будову системи, підкреслити логічні зв’язки між об’єктами та забезпечити основу для подальшої реалізації в коді.

Використання діаграм класів є важливим під час аналізу вимог до системи, її проєктування та документування. Вони допомагають зрозуміти, які сутності існують у системі, які властивості та поведінку вони мають, як об'єкти взаємодіють між собою, і які залежності між ними існують. Це дає змогу виявити потенційні проблеми на ранніх етапах проєктування, уникнути дублювання функціоналу та забезпечити узгодженість усіх елементів системи.

Особливо актуальним є використання діаграм класів у складних програмних рішеннях, де взаємозв’язки між об’єктами численні й важко піддаються уявному аналізу. Завдяки цим діаграмам команда розробки може ефективно комунікувати, ділити обов’язки та спільно створювати єдину архітектуру проєкту. Таким чином, діаграма класів виступає важливим етапом у побудові стабільної, масштабованої та підтримуваної програмної системи.

UML-class діаграма наведена на рисунку 2.4.

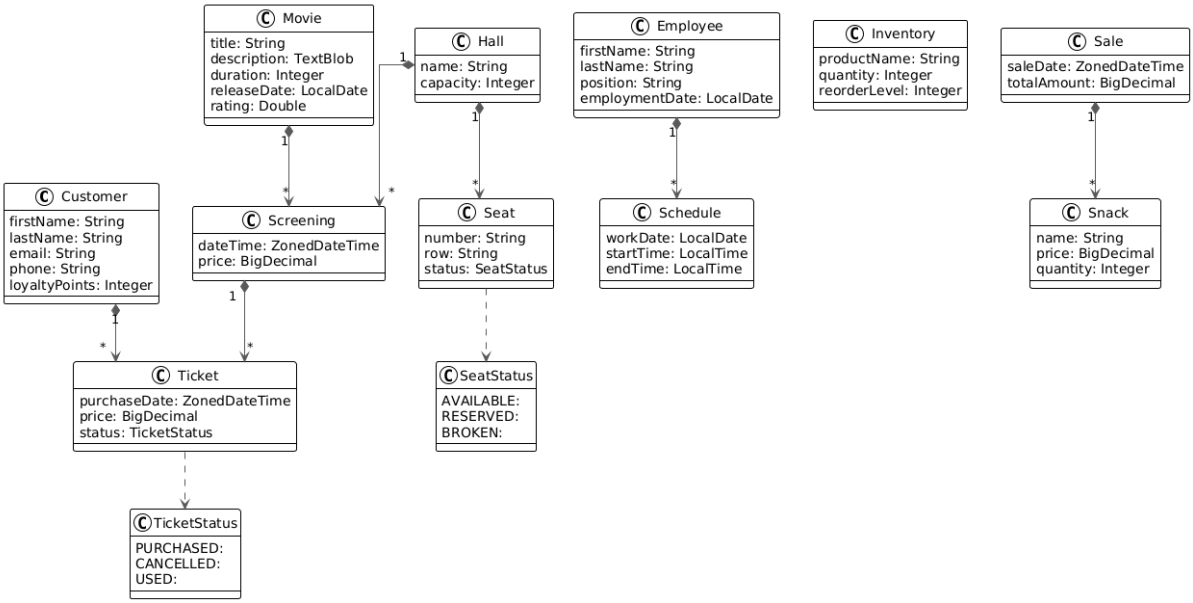

Рисунок 2.4 – UML-class діаграма

Запропонована діаграма класів моделює структуру об’єктів системи бронювання кінотеатру, ілюструючи сутності та зв’язки між ними. Угорі розташовані основні учасники – клієнт та співробітники, які фіксуються в об’єктах Customer та Employee. Клієнт зберігає базову контактну інформацію та

кількість бонусних балів, що можуть бути використані в системі лояльності. Співробітник має дані про посаду та дату прийому на роботу, що необхідно для керування персоналом.

Сутність Movie описує фільми, які демонструються в кінотеатрі, включаючи назву, опис, тривалість, дату прем’єри та рейтинг. З ними пов’язані об’єкти Screening, які містять інформацію про час показу фільму та його вартість. Кожен сеанс закріплюється за певним залом (Hall), що має свою назву та місткість. До залу також належать конкретні місця (Seat), кожне з яких має свій номер, ряд і статус, що вказує, чи доступне воно для бронювання, зарезервоване або недоступне.

Коли клієнт купує квиток, створюється об’єкт Ticket, який містить дату покупки, ціну та статус – придбаний, анульований або використаний. Кожен квиток асоціюється з конкретним сеансом та клієнтом. В системі визначено статуси місць (SeatStatus) і квитків (TicketStatus) як окремі класи, що дозволяє чітко моделювати зміну станів об’єктів.

Крім цього, модель включає об’єкт Inventory, який забезпечує облік запасів снеків, з вказанням назви продукту, кількості та критичного рівня поповнення. Снеки (Snack) можуть бути частиною продажу (Sale), де фіксується дата транзакції та загальна сума. Для обліку робочого часу співробітників використовується клас Schedule, який визначає дату та години роботи.

Така структура забезпечує повноцінне представлення внутрішньої логіки системи кінотеатру, дозволяючи ефективно управляти процесами від демонстрації фільмів до контролю ресурсів і обслуговування клієнтів.

2.2.4 Розробка UML Object Diagram

Діаграма об'єктів (Object Diagram) є однією з допоміжних структурних діаграм у мові UML, що дозволяє фіксувати стан системи на певний момент часу. Основне її призначення полягає у наочному представленні конкретних екземплярів класів, які беруть участь у функціонуванні системи, із зазначенням

значень їхніх атрибутів і встановлених між ними зв’язків. Вона є фактичним знімком частини пам’яті об’єктно-орієнтованої системи, що демонструє, як її логічна структура реалізується у вигляді конкретних об’єктів під час виконання.

На відміну від діаграми класів, яка описує загальну архітектуру системи та визначає типи об’єктів і їх взаємозв’язки, діаграма об'єктів фокусується на вже створених елементах – об’єктах, що існують у певний момент часу. Це дозволяє наочно відобразити сценарій використання системи, актуальні стани даних та фактичні зв’язки між екземплярами класів. Важливість таких діаграм проявляється в процесі тестування, верифікації логіки, документування функціоналу та побудови прикладів для розуміння поведінки системи.

Object diagram є особливо корисною у складних інформаційних системах, де важливо простежити взаємодію між численними сутностями в динаміці, а також перевірити коректність ініціалізації, збереження та обробки даних. У контексті системи управління ресурсами кінотеатру така діаграма дозволяє змоделювати конкретну ситуацію – наприклад, придбання клієнтом квитка на певний фільм, з уточненням зали, місця, дати сеансу та додаткових покупок, що дозволяє виявити і логічно узгодити зв’язки між об’єктами.

Таким чином, діаграма об’єктів є невід’ємним елементом моделювання поведінки програмного забезпечення, що забезпечує візуальну підтримку процесу розробки, підвищує якість проектної документації та сприяє точнішому формуванню архітектурних рішень. Її використання на ранніх етапах дозволяє зменшити ризики логічних помилок і покращити загальну узгодженість моделі.

Object-діаграма, представлена на рисунку 2.5, відображає конкретний фрагмент роботи системи бронювання кінотеатру в певний момент часу. Вона є знімком поточного стану об’єктів, що моделюють типовий сценарій взаємодії користувача із системою, зокрема оформлення замовлення на перегляд фільму та придбання снеків.

На діаграмі змодельовано ситуацію, в якій клієнт Іван Петренко обирає фільм «Інтерстеллар» і здійснює бронювання квитка на сеанс, що відбудеться 15 квітня 2025 року о 20:00 в залі «IMAX», що має місткість 150 місць.

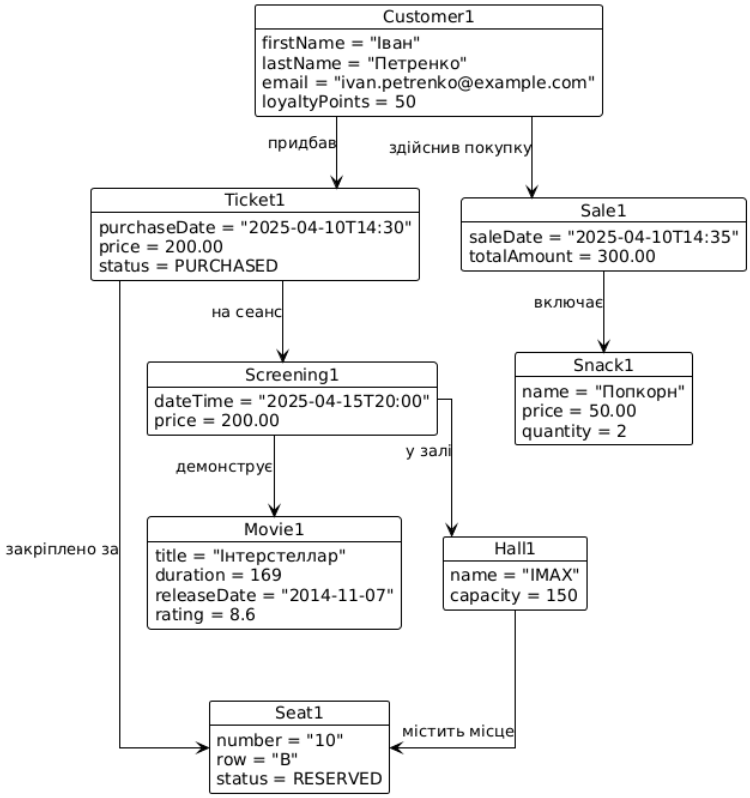

Рисунок 2.5 – UML-object діаграма

Обране місце – номер 10 у ряду B – має статус «зарезервовано», що свідчить про його тимчасову недоступність для інших користувачів. Квиток, прив’язаний до цього сеансу та місця, має статус «придбаний» і зберігає інформацію про дату покупки – 10 квітня 2025 року, а також ціну – 200 грн.

Кожен об’єкт представлений із зазначенням атрибутів і відповідних значень. Наприклад, фільм «Інтерстеллар» має тривалість 169 хвилин, дату релізу 7 листопада 2014 року та рейтинг 8.6, що дозволяє враховувати його популярність. Важливими є зв’язки між об’єктами, які уточнюють їхню роль у сценарії: квиток пов’язаний із конкретним клієнтом, місцем у залі та відповідним сеансом; сам сеанс – із фільмом та залом.

Окрім бронювання, діаграма ілюструє додаткову взаємодію клієнта з системою – зокрема, здійснення покупки снеків. Іван Петренко придбав два попкорни по 50 грн, що відображено у відповідному об’єкті Snack. Пов’язана з цим транзакція фіксується в об’єкті Sale з вказаною загальною сумою 300 грн і часом покупки – 14:35 того ж дня, коли було придбано квиток. Це свідчить про

інтеграцію обліку продажів додаткових товарів у межах єдиної системи обслуговування клієнта.

Таким чином, Object-діаграма дає змогу наочно відобразити життєвий цикл об'єктів у межах конкретного процесу, перевірити коректність їхніх взаємозв’язків та переконатися у відповідності структури об’єктної моделі логіці бізнес-процесів кінотеатру. Також слід зосередитися на її практичному значенні в контексті розробки інформаційної системи. Об’єктна діаграма не лише забезпечує фіксацію поточного стану сутностей у системі, а й виступає інструментом перевірки відповідності між логічною структурою, змодельованою на рівні класів, та реальною взаємодією об’єктів під час виконання. У випадку системи кінотеатру, така діаграма дозволяє проаналізувати наскрізний сценарій обслуговування клієнта – від вибору фільму до придбання додаткових товарів – і переконатися, що кожна ланка цього процесу належним чином представлена у системі.

Додатковим аспектом, який демонструє ця діаграма, є послідовність операцій у рамках бізнес-процесу. Зв’язки між об’єктами ілюструють взаємозалежність даних: зокрема, неможливо створити квиток без наявного сеансу й клієнта, як і продаж не відбудеться без обраного товару та зафіксованої транзакції. Завдяки цьому забезпечується цілісність даних у системі, що є критично важливим для її стабільної роботи.

Object-діаграма також виступає у ролі засобу для візуалізації типових сценаріїв використання, що особливо корисно під час етапів валідації та демонстрації функціональності замовнику або членам команди. Вона дозволяє представити логіку взаємодії у доступній формі, не вдаючись до складної термінології або фрагментів коду. У такий спосіб полегшується процес прийняття рішень щодо оптимізації, модифікацій або розширення функціоналу системи.

Загалом, наведена Object-діаграма є інформативною моделлю поточного стану ключових об’єктів, яка дозволяє здійснити якісне проектування, перевірити відповідність бізнес-вимогам і підвищити прозорість внутрішньої

логіки застосунку. Її використання сприяє ефективному контролю як на етапі розробки, так і під час подальшої підтримки програмного забезпечення.

2.2.5 Розробка UML Activity diagram

Діаграми діяльності (Activity diagram) є важливою частиною моделювання поведінки програмного забезпечення, що базується на специфікації UML (Unified Modeling Language). Цей тип діаграми призначений для зображення послідовності дій, логіки керування потоками та умов переходу між станами в межах певного процесу або функціонального сценарію. На відміну від діаграми послідовностей, яка фокусується на взаємодії між об’єктами у часі, діаграма діяльності показує внутрішню логіку виконання задачі незалежно від об'єктів, які її реалізують.

Кожна діаграма діяльності будується з елементів, що відображають дії (activity), рішення (decision), початкові та кінцеві стани, паралельні або умовні гілки. Візуально це виглядає як схема блоків, з’єднаних стрілками, які визначають послідовність переходу між діями. Такий підхід дозволяє формалізовано описати, як система поводиться у відповідь на зовнішні дії користувача або внутрішні події, з урахуванням гілкування, циклів та умов виконання.

У сфері розробки інформаційних систем діаграма діяльності є інструментом для фахівців різного рівня: аналітики використовують її для опису бізнес-процесів, архітектори – для побудови логіки систем, розробники – для реалізації програмного коду відповідно до визначених дій. Такі діаграми значно спрощують розуміння складних процесів, забезпечують узгодженість між командою розробки, полегшують тестування та документування.

Особливо важливими діаграми діяльності є для сценаріїв, що містять умови, перевірки або кілька варіантів виконання. Наприклад, у системі бронювання кінотеатру користувач проходить низку дій: обирає фільм, перевіряє доступність місць, авторизується, бронює квиток і здійснює оплату. Діаграма

діяльності дозволяє візуально описати всі ці кроки, включно з альтернативними шляхами (наприклад, відсутність місць або помилка оплати), що критично для надійності системи.

Activity diagram є універсальним засобом представлення динамічної поведінки системи, який забезпечує чітке, структуроване та інтуїтивно зрозуміле бачення процесу, сприяючи підвищенню ефективності розробки, аналізу та впровадження програмного забезпечення.

Діаграма діяльності (UML-activity діаграма), представлена на рисунку 2.6, моделює типовий сценарій взаємодії клієнта із системою бронювання кінотеатру та демонструє логічну послідовність дій, що виконуються при оформленні замовлення. Вона є результатом формалізації процесу, де кожен крок відображено як дія, а переходи між діями реалізуються за допомогою умовних розгалужень.

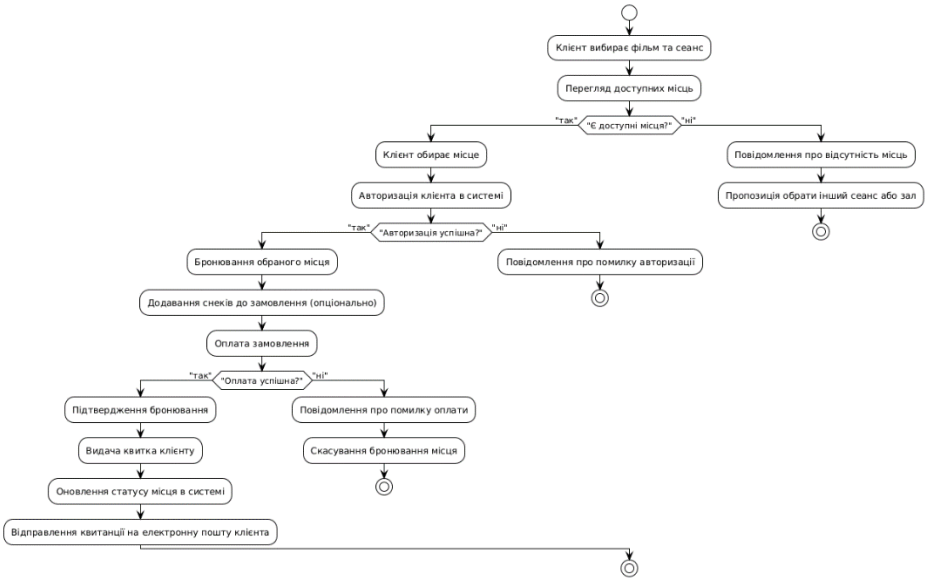

Рисунок 2.6 – UML-activity діаграма

На представленій компонентній UML-діаграмі (рис. 2.6) відображено загальну архітектуру програмного забезпечення системи управління ресурсами кінотеатру. Діаграма складається з кількох основних пакетів, кожен з яких включає компоненти, відповідальні за певний аспект функціонування системи.

Перший пакет, що позначений як «Клієнтська частина (Frontend React)», містить компонент «Користувацький інтерфейс». Цей компонент забезпечує взаємодію користувача з системою, реалізує графічний інтерфейс та дозволяє виконувати операції авторизації, перегляду доступних сеансів, бронювання квитків і проведення платежів. Інтерфейс взаємодіє з серверною частиною системи через HTTP-запити.

Другий пакет, «Сервер застосунку (Spring Boot)», містить компоненти, які розподілені на два рівні: контролери та сервіси. Контролери (авторизації, сеансів, квитків і оплат) є точками входу для обробки запитів від фронтенду. Після отримання запитів контролери делегують виконання бізнес-логіки відповідним сервісам. Сервіси авторизації відповідають за аутентифікацію користувачів та перевірку прав доступу. Сервіси сеансів забезпечують отримання інформації про фільми та сеанси. Сервіси квитків виконують операції бронювання і управління квитками. Сервіси оплат відповідають за обробку платежів і взаємодію з платіжними системами.

Третій пакет, «База даних (PostgreSQL)», включає таблиці, які використовуються для постійного зберігання інформації. Кожен із сервісів має безпосередній доступ до відповідної таблиці, наприклад, сервіс авторизації зберігає та отримує дані користувачів з таблиці користувачів, а сервіс квитків працює з таблицею квитків. Такий поділ дозволяє забезпечити чітке розмежування відповідальності та ефективну роботу з даними.

Останній пакет, «Зовнішні сервіси», демонструє взаємодію системи з зовнішніми компонентами. До них належить «Сервіс email-розсилки», який використовується для повідомлень клієнтам про підтвердження бронювання, та «Платіжна система», що обробляє фінансові транзакції. Сервіси оплати та квитків взаємодіють з цими зовнішніми системами, забезпечуючи інтеграцію з додатковими функціями, необхідними для повноцінного обслуговування клієнтів.

Загалом, дана компонентна діаграма наочно демонструє логічну структуру та взаємодію окремих частин системи, допомагаючи розробникам чітко

зрозуміти архітектуру проєкту, що, у свою чергу, полегшує підтримку, розширення та масштабування застосунку.

Сценарій розпочинається з ініціації взаємодії, коли клієнт обирає фільм та відповідний сеанс. Після цього здійснюється перевірка доступності місць. Якщо вільні місця є, користувач обирає конкретне місце та проходить авторизацію. У разі успішної авторизації виконується етап резервування місця, з можливістю додати снеки до замовлення. Наступним кроком є процес оплати. Якщо оплата проходить успішно, система підтверджує бронювання, видає квиток, оновлює статус місця та надсилає клієнту електронну квитанцію.

У разі помилки на будь-якому з ключових етапів, як-от авторизація чи оплата, відповідно генерується повідомлення про помилку, і сценарій завершується без завершення бронювання. Якщо ж на початку не виявлено доступних місць, клієнту пропонується альтернативний варіант – обрати інший сеанс або зал.

Діаграма діяльності, представлена на рисунку 2.6, моделює типовий бізнес-процес оформлення замовлення клієнтом у системі кінотеатру, фокусуючись на етапах вибору сеансу, перевірки доступності місць, бронювання, авторизації, оплати та обробки покупки. Це наочно демонструє логіку послідовності дій та варіанти розвитку подій у разі виникнення помилок або відсутності ресурсів.

У початковій точці сценарію користувач ініціює процес, обираючи фільм і сеанс. Наступним кроком є перегляд вільних місць. На цьому етапі виконується перевірка наявності доступних місць. Якщо система виявляє наявність таких, користувач має змогу обрати конкретне місце та переходить до авторизації в системі. Успішна авторизація відкриває доступ до бронювання місця, після чого клієнт має можливість додати до замовлення снеки – ця дія є необов’язковою, однак важлива з точки зору підвищення прибутковості сервісу.

Далі процес переходить до етапу оплати. У випадку успішного завершення транзакції система підтверджує бронювання, видає квиток, оновлює статус

обраного місця та надсилає клієнту квитанцію на електронну пошту. Це забезпечує завершення циклу взаємодії та підтверджує його результат.

Однак, якщо на будь-якому з ключових етапів виникає помилка, сценарій розвивається за альтернативною гілкою. Наприклад, у разі помилки авторизації користувач отримає повідомлення про невдалу спробу входу й процес буде завершено без продовження. Аналогічно, помилка оплати спричинить скасування бронювання, що дозволяє уникнути некоректного резервування ресурсу. У випадку, коли доступних місць немає, клієнту пропонується обрати інший сеанс або зал.

Ця діаграма дозволяє дослідити логіку поведінки системи з урахуванням як основного, так і альтернативних сценаріїв, і є ключовим інструментом у процесі формального опису динаміки функціонування прикладного програмного забезпечення. Вона також сприяє виявленню критичних точок, де потрібна перевірка валідності дій або введення додаткових обробників помилок. Завдяки цьому можна досягти вищого рівня надійності системи та забезпечити її відповідність вимогам користувачів.

Запропонований приклад ілюструє здатність діаграми діяльності не лише відображати прямолінійну послідовність кроків, але й ефективно моделювати складні логічні гілки, альтернативи та умови. Вона є цінним інструментом для формалізації користувацьких сценаріїв, перевірки логіки роботи системи та комунікації між учасниками розробки на всіх етапах життєвого циклу програмного забезпечення.

2.2.6 Розробка UML Component diagram

Компонентна діаграма (Component diagram) є одним з основних типів структурних діаграм у мові моделювання UML, яка використовується для візуального відображення високорівневої архітектури програмного забезпечення. Її основна мета – представити логічну структуру системи у вигляді взаємопов’язаних компонентів, що реалізують певну функціональність або

інтерфейси. Кожен компонент може відповідати модулю, підсистемі, бібліотеці або сервісу, який виконує певну роль у загальній системі.

Компонентна діаграма дозволяє розробникам, аналітикам та архітекторам отримати загальне уявлення про організацію програмної системи, її залежності, точки інтеграції з іншими модулями або сторонніми сервісами. Вона відображає, як компоненти взаємодіють між собою за допомогою інтерфейсів, що забезпечує гнучкість у підтримці, розширенні та тестуванні системи. Зазвичай діаграма включає такі елементи, як компоненти, інтерфейси (надавані та вимагані), з'єднання між ними, а також залежності.

У сучасних системах, зокрема тих, що базуються на мікросервісній архітектурі або багаторівневих застосунках, компонентна діаграма стає інструментом для проектування логіки взаємодії між фронтендом, бекендом, базами даних та зовнішніми API.

Вона допомагає розмежувати відповідальність між модулями, визначити публічні точки входу для сервісів і оцінити вплив змін у системі. Компоненти можуть бути деталізовані окремими класами або реалізовані як незалежні частини коду, що розгортаються автономно.

Таким чином компонентна діаграма відіграє важливу роль у процесі моделювання складних інформаційних систем, оскільки забезпечує чітке структурування логіки, підвищує масштабованість та дозволяє уникнути дублювання функціоналу за рахунок повторного використання незалежних частин. Вона є також невід’ємною частиною документації системи, що забезпечує її зрозумілість для всієї команди розробки.

На представленій компонентній UML-діаграмі (рис. 2.7) відображено загальну архітектуру програмного забезпечення системи управління ресурсами кінотеатру. Діаграма складається з кількох основних пакетів, кожен з яких включає компоненти, відповідальні за певний аспект функціонування системи.

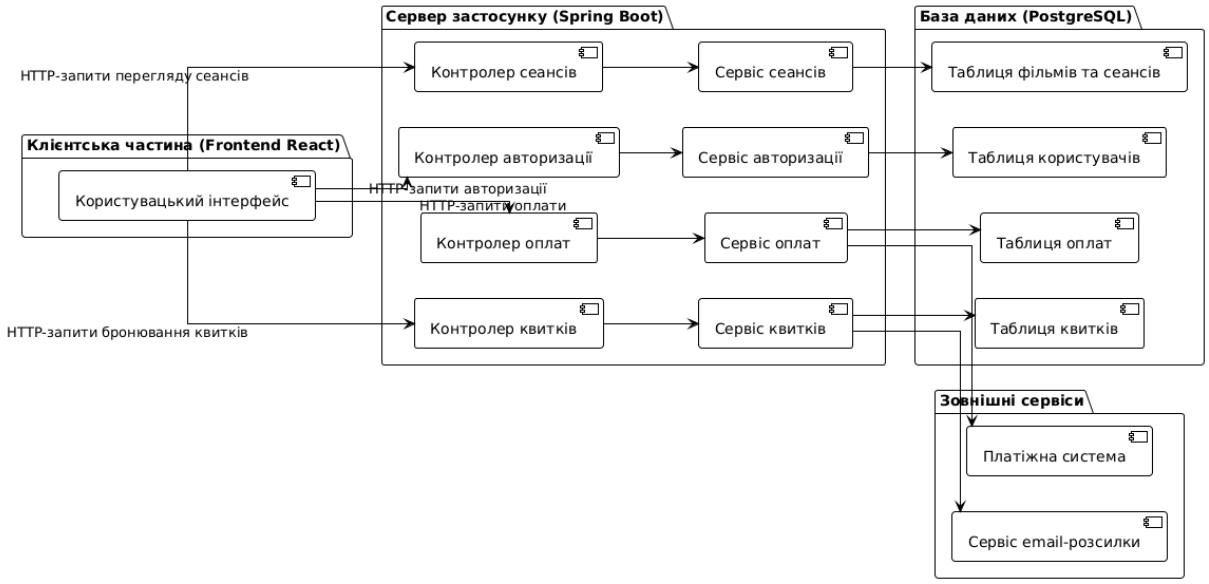

Рисунок 2.7 – UML-component діаграма

Перший пакет, що позначений як «Клієнтська частина (Frontend React)», містить компонент «Користувацький інтерфейс». Цей компонент забезпечує взаємодію користувача з системою, реалізує графічний інтерфейс та дозволяє виконувати операції авторизації, перегляду доступних сеансів, бронювання квитків і проведення платежів. Інтерфейс взаємодіє з серверною частиною системи через HTTP-запити.

Другий пакет, «Сервер застосунку (Spring Boot)», містить компоненти, які розподілені на два рівні: контролери та сервіси. Контролери (авторизації, сеансів, квитків і оплат) є точками входу для обробки запитів від фронтенду. Після отримання запитів контролери делегують виконання бізнес-логіки відповідним сервісам. Сервіси авторизації відповідають за аутентифікацію користувачів та перевірку прав доступу. Сервіси сеансів забезпечують отримання інформації про фільми та сеанси. Сервіси квитків виконують операції бронювання і управління квитками. Сервіси оплат відповідають за обробку платежів і взаємодію з платіжними системами.

Третій пакет, «База даних (PostgreSQL)», включає таблиці, які використовуються для постійного зберігання інформації. Кожен із сервісів має безпосередній доступ до відповідної таблиці, наприклад, сервіс авторизації зберігає та отримує дані користувачів з таблиці користувачів, а сервіс квитків

працює з таблицею квитків. Такий поділ дозволяє забезпечити чітке розмежування відповідальності та ефективну роботу з даними.

Останній пакет, «Зовнішні сервіси», демонструє взаємодію системи з зовнішніми компонентами. До них належить «Сервіс email-розсилки», який використовується для повідомлень клієнтам про підтвердження бронювання, та «Платіжна система», що обробляє фінансові транзакції. Сервіси оплати та квитків взаємодіють з цими зовнішніми системами, забезпечуючи інтеграцію з додатковими функціями, необхідними для повноцінного обслуговування клієнтів.

Загалом, дана компонентна діаграма наочно демонструє логічну структуру та взаємодію окремих частин системи, допомагаючи розробникам чітко зрозуміти архітектуру проєкту, що, у свою чергу, полегшує підтримку, розширення та масштабування застосунку.

2.2.7 Розробка UML Deployment diagram

У мові моделювання UML діаграма розгортання (Deployment diagram) є важливим інструментом для представлення фізичної архітектури програмної системи. Її основне призначення полягає у відображенні розміщення програмних компонентів на апаратних вузлах, а також комунікацій між ними. Вона моделює інфраструктуру системи, яка включає сервери, пристрої користувачів, бази даних, зовнішні сервіси та інші апаратно-програмні ресурси.

На відміну від логічних діаграм, таких як діаграма класів чи компонентів, діаграма розгортання фокусується на фізичному розміщенні частин системи та їх взаємозв’язках у реальному середовищі виконання. У рамках цієї діаграми вузли (nodes) представляють фізичні чи віртуальні машини, сервери або пристрої, на яких розгортаються програмні компоненти, що моделюються у вигляді прямокутників із написом «component».

У сфері розробки вебзастосунків, таких як система управління кінотеатром, діаграма розгортання дозволяє показати, як клієнтські компоненти

(наприклад, React-інтерфейс у браузері або мобільний PWA-застосунок) взаємодіють із серверною частиною (Spring Boot застосунок), яка обробляє запити, керує бізнес-логікою, працює з базою даних, кешує сесії у Redis, відправляє повідомлення через Kafka, і звертається до зовнішніх сервісів – платіжних шлюзів, поштових серверів та служб SMS.

Завдяки цій діаграмі можна проаналізувати навантаження на різні частини системи, виявити потенційні вузькі місця в інфраструктурі, визначити шляхи масштабування та забезпечити відповідність вимогам безпеки й надійності. Також вона є важливим елементом технічної документації, який використовується для комунікації між архітекторами, DevOps-фахівцями та розробниками, а також для підготовки до впровадження системи в промислове середовище. Таким чином, UML Deployment diagram сприяє глибшому розумінню того, як саме система функціонує у фізичному середовищі, і є невід’ємною частиною процесу проєктування сучасних розподілених інформаційних систем.

Представлена діаграма розгортання на рисунку 2.8 моделює фізичну архітектуру системи бронювання кінотеатру, реалізованої за допомогою фреймворку Spring Boot з клієнтським інтерфейсом на базі React та мобільним PWA-додатком. Вона демонструє розміщення основних компонентів системи на відповідних вузлах та типи з'єднань між ними. Це дає змогу зрозуміти, як логіка застосунку реалізується у фізичному середовищі, і забезпечує візуалізацію ключових точок інтеграції. У структурі системи виділено кілька рівнів. Клієнтський рівень представлений веб-браузером з React-інтерфейсом, а також мобільним прогресивним застосунком (PWA), що працює на пристроях користувачів. Обидва канали взаємодіють із сервером застосунку через захищені HTTPS-з’єднання. Це забезпечує зручну і безпечну взаємодію кінцевого користувача з системою.

Серверначастина складається з кількох логічно незалежних вузлів. Основний сервер працює на базі Spring Boot, де реалізовано REST API-контролери, бізнес-сервіси та окремі модулі для обробки JWT-аутентифікації.

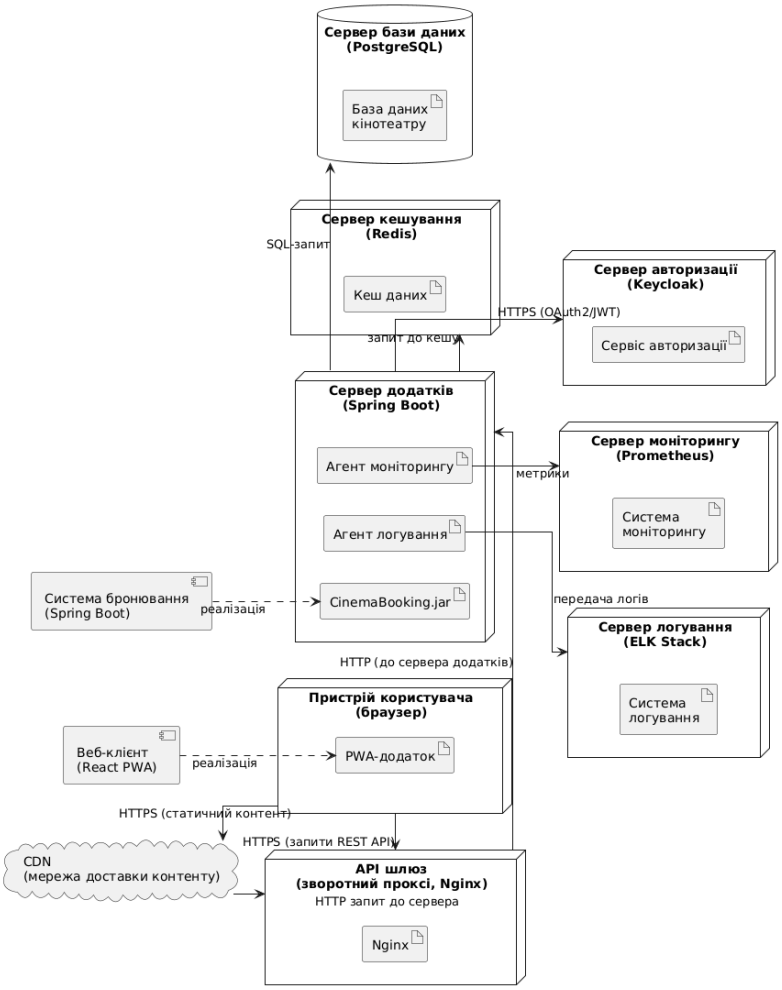

Рисунок 2.8 – UML-deployment діаграма

Обробка запитів користувачів проходить через контролери, які передають дані у відповідні сервіси або взаємодіють із модулями автентифікації. Для забезпечення реактивної обробки подій у реальному часі використовується Apache Kafka як брокер повідомлень, до якого мають доступ бізнес-сервіси. Кешування даних реалізовано через Redis, що підвищує загальну продуктивність і дозволяє ефективно управляти тимчасовими сесіями.

Представлена UML-діаграма розгортання моделює фізичну інфраструктуру системи бронювання кінотеатру, реалізованої з використанням вебтехнологій та фреймворку Spring Boot. Основна мета цієї діаграми – наочно показати, як компоненти системи розміщуються на реальних або віртуальних апаратних вузлах, які типи взаємодій між ними існують, і які сервіси підтримують основні бізнес-процеси.

На діаграмі виділено клієнтський пристрій, на якому виконується PWA-додаток (Progressive Web App), створений за допомогою фреймворку React. Цей додаток, що є частиною веб-клієнта, отримує статичний контент (HTML, CSS, JS) з мережі доставки контенту (CDN). Надалі всі динамічні запити відправляються через API-шлюз (реалізований як Nginx), що виконує функції зворотного проксі, керує маршрутизацією, обробляє HTTPS-з'єднання та передає запити до внутрішньої інфраструктури.

Запити до бізнес-логіки обробляються на сервері застосунку, де розгорнуто артефакт CinemaBooking.jar – основна збірка Spring Boot-застосунку, яка реалізує систему бронювання. Цей застосунок також містить агент логування, що надсилає журнали подій до централізованої системи логування (ELK Stack), та агент моніторингу, який збирає метрики продуктивності та передає їх на сервер моніторингу (Prometheus).

Для реалізації функціоналу автентифікації використовується окремий сервер авторизації, де розміщено компонент Keycloak, що забезпечує OAuth2-автентифікацію з використанням JWT-токенів. Комунікація між застосунком і сервером авторизації здійснюється через захищене з'єднання HTTPS.

Зберігання даних відбувається на сервері бази даних PostgreSQL, де розгорнуто артефакт “База даних кінотеатру”, що містить усі необхідні сутності – фільми, сеанси, користувачів, квитки тощо. Додатково інтегровано Elasticsearch, призначений для зберігання журналів аудиту та забезпечення швидкого пошуку логів.

Кешування запитів реалізовано за допомогою Redis, що дозволяє тимчасово зберігати дані та зменшити навантаження на основну базу. Застосунок звертається до Redis при обробці повторних запитів або зберіганні тимчасових сесій.

На діаграмі також показано зв’язки між усіма елементами системи:

- користувач отримує інтерфейс через CDN та взаємодіє з API через HTTPS;
- API-шлюз передає запити до Spring Boot-серверу;

- застосунок звертається до Redis для кешування, до PostgreSQL – для доступу до постійних даних, до Elasticsearch – для збереження логів;
- метрики надсилаються на Prometheus, журнали – до ELK Stack;
- запити авторизації обробляються Keycloak-сервером.

Дана діаграма демонструє чітке розмежування ролей між вузлами, їх функціональність та спосіб взаємодії. Такий підхід дозволяє забезпечити масштабованість, надійність, інформаційну безпеку та розширюваність системи, що особливо важливо для промислової експлуатації платформи управління ресурсами кінотеатру.

2.2.8 Розробка UML State diagram

Діаграма станів (State Diagram) є однією з поведінкових діаграм UML, яка призначена для моделювання життєвого циклу об’єкта в системі – від його створення до знищення – шляхом відображення послідовних змін його стану у відповідь на певні події. Основною метою цієї діаграми є візуалізація можливих станів, у яких може перебувати об’єкт, а також умов, що спричиняють перехід між цими станами.

Цей тип діаграм особливо корисний у ситуаціях, де поведінка об’єкта визначається його поточним станом. Вона застосовується для опису систем, де є складна логіка переходів, як у фінансових процесах, системах керування користувачами, модулях бронювання, а також у будь-яких інтерфейсах, де об’єкти змінюють свої властивості залежно від дій користувача або внутрішніх умов.

На діаграмі станів використовуються такі базові елементи: стани (які відображають ситуацію або умови, в яких перебуває об'єкт у певний момент часу), події (що викликають зміну стану), переходи (які з'єднують стани та відображають рух об'єкта між ними), а також початковий і кінцевий стан (що визначають межі життєвого циклу об'єкта).

Перевагою використання діаграм станів є те, що вони дозволяють глибоко проаналізувати поведінку об’єктів у системі, виявити критичні гілки логіки, уніфікувати бізнес-правила та забезпечити більшу передбачуваність у поведінці застосунку. Такі діаграми сприяють чіткому розумінню станів об’єкта як розробниками, так і аналітиками та тестувальниками.

У контексті системи бронювання квитків, діаграма станів може відображати послідовні етапи обробки квитка: наприклад, від його створення, через етапи резервування, оплати, активації або анулювання. Це дозволяє структурувати логіку роботи з квитками та гарантувати цілісність бізнес-процесу.

Таким чином, діаграма станів є потужним інструментом для проєктування та аналізу поведінки складних систем, дозволяючи мінімізувати логічні помилки, узгодити поведінку компонентів і підвищити якість архітектури програмного забезпечення.

UML-діаграма станів (State Diagram), представлена на рисунку 2.9, моделює життєвий цикл об’єкта замовлення квитка в системі бронювання кінотеатру. Вона демонструє послідовні зміни станів, які проходить замовлення – від моменту його створення до фінального використання або анулювання. Така діаграма дозволяє чітко відобразити логіку переходів між станами залежно від дій користувача або внутрішніх подій у системі, що є критично важливим для гарантування коректності виконання бізнес-процесів.

Життєвий цикл починається з початкового стану, позначеного символом [*], який переходить у стан Created унаслідок ініціації процесу створення замовлення. Всередині цього стану відбувається підпроцес підготовки даних, де замовлення збирає всі необхідні атрибути. Стан Підготовка моделює внутрішній перехід у межах об'єкта Created, що відображає логіку формування об’єкта до завершення цього етапу.

Після завершення підготовки можливий перехід у стан Reserved, який означає, що квитки успішно заброньовані.

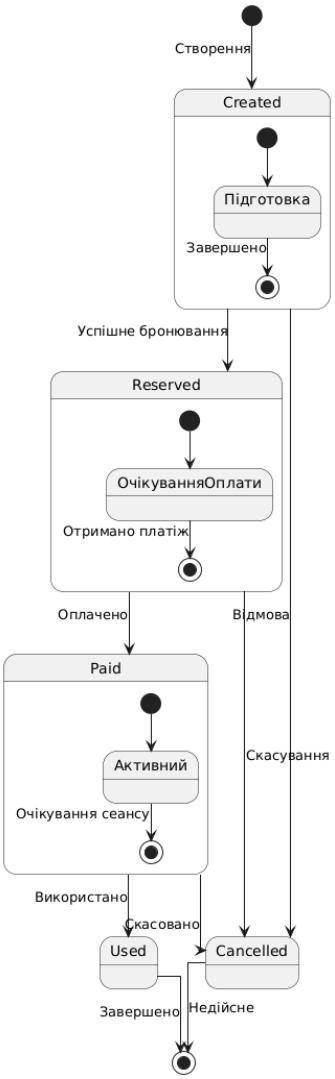

Рисунок 2.9 – UML-state діаграма

Альтернативний перехід із Created веде до стану Cancelled, що позначає ситуацію, коли замовлення не було завершено – через відмову користувача, системну помилку або порушення вимог. Таким чином, уже на ранньому етапі життєвого циклу об'єкта визначаються дві можливі гілки розвитку: продовження до оплати або завершення з помилкою.

У стані Reserved відбувається очікування платежу. Початковий підстан тут – ОчікуванняОплати, який завершується переходом до зовнішнього стану Paid, якщо платіж надійшов успішно. У разі неуспішної транзакції або скасування бронювання користувачем передбачений перехід до Cancelled, що унеможливлює подальше використання квитка. Таким чином, саме цей вузол логіки відображає критичний момент – очікування фінансової дії від клієнта.

Стан Paid означає завершене успішне бронювання з підтвердженою оплатою. Усередині цього стану моделюється підстан Активний, який ілюструє період очікування події – початку сеансу. Вихід зі стану Paid можливий у двох напрямках: до Used, коли квиток відсканована під час відвідування кінотеатру, або до Cancelled, коли оплата була повернута, або квиток був скасований згідно з політикою повернення. Стан Used вказує на завершене використання квитка, після чого життєвий цикл об’єкта досягає фінального завершення.

З іншого боку, будь-який перехід до Cancelled означає припинення дії об’єкта. Цей стан має єдину поведінку – він відзначає квиток або замовлення як недійсне, завершуючи його життєвий цикл.

Таким чином, запропонована діаграма відображає повний життєвий шлях замовлення квитка, моделюючи всі ключові стани: створення, резервування, оплату, активацію, використання або анулювання. Це дозволяє розробникам та аналітикам відстежити, в яких станах може перебувати замовлення в будь-який момент, і які події впливають на зміну цих станів. Такий підхід забезпечує прозорість бізнес-логіки, підвищує надійність програмного забезпечення і дозволяє формалізувати перевірку граничних умов для кожного сценарію користування системою.

2.2.9 Розробка UML Timing diagram

Timing Diagram (Діаграма часу) – це один із видів поведінкових діаграм у мові моделювання UML, який використовується для відображення змін станів об'єктів або значень змінних у часовій шкалі. Основне призначення такої діаграми полягає у наочному показі, як об'єкти поводяться з плином часу, у якому порядку змінюються їх стани та скільки часу вони перебувають у кожному з них. Це дозволяє дослідити синхронізацію, затримки, часові обмеження та інші аспекти, пов’язані з часовою поведінкою системи.

На відміну від діаграми станів, яка зосереджена на послідовності переходів між станами об’єкта, діаграма часу показує ті ж самі переходи, але в контексті

часу. Її структура зазвичай складається з горизонтальної шкали часу, вертикальних ліній життєвих циклів об’єктів або змінних та прямокутників, що позначають активні стани або значення.

Розглянемо типові сценарії використання Timing Diagram.

- Аналіз тимчасових залежностей між об’єктами в системах реального часу.
- Верифікація поведінки інтерфейсів користувача або мережевих протоколів.
- Моделювання сценаріїв, де важливо відслідковувати час реакції, затримки або час виконання дій.

У випадку системи бронювання кінотеатру Timing Diagram може бути використана для моделювання тривалості кожного етапу взаємодії клієнта з системою – наприклад, час на вибір фільму, резервування місця, проведення оплати, підтвердження замовлення. Це дозволяє аналізувати, чи вкладаються процеси в допустимі межі часу, які затримки можуть бути критичними та де можлива оптимізація.

Timing Diagram (діаграма часових характеристик) у нотації UML – це тип поведінкової діаграми, що дозволяє описати зміну станів об’єктів у часі. Вона ідеально підходить для моделювання сценаріїв, у яких важливо простежити, як саме змінюється стан учасників системи під впливом подій у чітко визначеній часовій послідовності. На відміну від діаграм послідовності або діяльності, де основна увага зосереджена на порядку дій, діаграма часу дозволяє відстежити саме затримки, синхронність і тривалість певних станів або операцій. Це робить її особливо корисною в розподілених або інтерактивних системах.

На рисунку 2.10 представлено діаграму часового розвитку подій, яка ілюструє взаємодію чотирьох основних учасників системи бронювання кінотеатру: Клієнта, Браузера, API-сервера та Платіжної системи.

Діаграма описує типовий сценарій – від моменту, коли користувач відкриває інтерфейс і починає вибір фільму, до фінального підтвердження успішного замовлення після оплати.

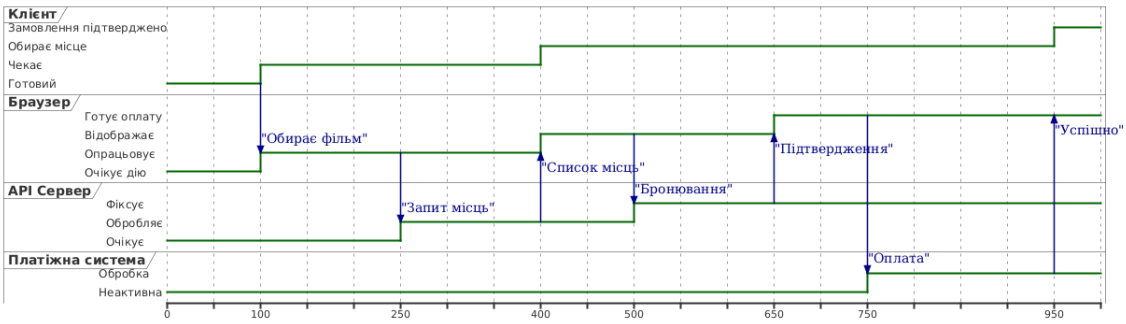

Рисунок 2.10 – UML-timing діаграма

У початковий момент часу всі компоненти системи перебувають у неактивному або очікуючому стані: клієнт готовий до взаємодії, браузер очікує дії користувача, API сервер не має запитів, а платіжна система не задіяна. Далі клієнт ініціює взаємодію, обираючи фільм. Це призводить до переходу браузера в стан активної обробки, а клієнта – у стан очікування відповіді. Через певний проміжок часу браузер звертається до API для отримання доступних місць у залі.

На цьому етапі API переходить у стан обробки запиту. Після обробки дані про доступні місця повертаються до браузера, який починає відображення результату. Клієнт у цей момент здійснює вибір конкретного місця, що супроводжується новим запитом до API на бронювання. API, отримавши запит, фіксує бронювання у базі даних, після чого повертає підтвердження до браузера.

Наступний етап – це підготовка до оплати. Браузер формує запит і надсилає його до зовнішньої платіжної системи. Платіжна система змінює свій стан на «обробка» й через деякий час повертає підтвердження успішної транзакції. У відповідь клієнт бачить повідомлення про підтверджене замовлення, що означає завершення життєвого циклу даного сценарію. Дана діаграма чітко відображає часову структуру взаємодії між учасниками, демонструє як змінюється їхній стан протягом обробки одного сценарію, дозволяє побачити затримки між запитами та реакціями, що може бути корисним під час оптимізації продуктивності та узгодженості логіки.

Таким чином, UML-timing діаграма є важливим інструментом для аналізу тимчасових характеристик системи, забезпечує чітке уявлення про динаміку обміну повідомленнями між компонентами та допомагає виявити потенційні

проблеми на ранніх етапах розробки. У системах із високими вимогами до реактивності, синхронізації чи взаємодії з кількома службами одночасно вона є невід'ємною частиною архітектурного моделювання.

2.3 Автоматизація процесів системи управління ресурсами кінотеатру

Автоматизація є ключовим етапом у розробці інформаційної системи управління ресурсами кінотеатру, оскільки дозволяє знизити витрати часу та зусиль на виконання рутинних операцій, забезпечити більшу точність обробки даних та підвищити загальну ефективність функціонування підприємства. Основна мета автоматизації полягає в оптимізації роботи персоналу, покращенні взаємодії з клієнтами, зменшенні кількості помилок і підвищенні якості обслуговування.

Процес автоматизації розпочинається з планування розкладу сеансів. В інформаційній системі реалізовано механізм для зручного додавання фільмів, автоматичного призначення часу показу з урахуванням тривалості фільму, технічних пауз, частоти повтору та поточного навантаження залів. Це дає змогу ефективно використовувати наявний простір, адаптувати розклад до попиту та уникати конфліктів у розміщенні сеансів. Також передбачена можливість змінювати розклад у режимі реального часу.

Процес бронювання квитків повністю автоматизовано. Користувач може обрати фільм, час показу, вільні місця та здійснити оплату онлайн. Система автоматично оновлює наявність місць у залі, формує електронний квиток, надсилає його клієнту та додає запис до журналу продажів. Для зручності передбачено інтеграцію з українськими платіжними системами, а також можливість зберігати квитки у форматах, сумісних із мобільними додатками та електронними гаманцями.

Управління запасами снеків також реалізовано в автоматизованому режимі. Система контролює кількість товарів на складі, відстежує продажі, формує попередження про мінімальні залишки та генерує звіти для замовлення

нових партій. За допомогою механізмів аналітики система аналізує пікові періоди попиту на різні види продукції та дозволяє робити прогнози щодо майбутніх потреб.

Облік персоналу здійснюється через спеціалізований інтерфейс. Кожен співробітник має власний профіль, де зберігаються дані про розклад змін, години роботи, роль у системі, рівень доступу та історію виконання завдань. Автоматичний розподіл змін дозволяє зменшити адміністративне навантаження, а механізм оцінювання продуктивності допомагає виявляти сильні сторони працівників і своєчасно реагувати на проблеми.

Автоматизовано також процеси, пов’язані з обробкою знижок, бонусів та програм лояльності. Система самостійно застосовує відповідні знижки під час оплати, веде облік бонусних балів клієнтів, дозволяє накопичувати або списувати їх, формує персоналізовані пропозиції та надсилає сповіщення про акції. Це сприяє підвищенню рівня задоволеності клієнтів і стимулює повторні покупки.

Важливим компонентом автоматизації є формування статистичних звітів і проведення бізнес-аналітики. Інформаційна система регулярно генерує зведення про динаміку продажів, відвідуваність фільмів, ефективність акцій та використання ресурсів. Звіти формуються у зручному вигляді з візуалізацією графіків, що допомагає керівництву приймати обґрунтовані управлінські рішення.

Централізоване адміністрування та контроль доступу забезпечується шляхом використання механізму авторизації на основі ролей. Система чітко визначає права доступу до певних розділів і функцій відповідно до ролі користувача — адміністратора, касира, менеджера або клієнта. Це дозволяє захистити дані від несанкціонованого доступу й гарантує безпеку виконання операцій.

Діаграма діяльності, що зображено на рисунку 2.11, описує автоматизований алгоритм процесу бронювання квитка в системі управління

ресурсами кінотеатру. Вона ілюструє послідовність дій користувача та відповідних кроків системи, які виконуються автоматично після кожного етапу.

Процес розпочинається з того, що користувач заходить на вебсайт кінотеатру. Після цього він переглядає розклад доступних фільмів та обирає бажаний фільм і конкретний сеанс. Наступним кроком є вибір місця в залі. Система автоматично перевіряє, чи обране місце є вільним.

Якщо місце доступне, відбувається його бронювання. Далі користувач вводить платіжні дані, після чого запускається процедура обробки оплати. Якщо оплата проходить успішно, система формує електронний квиток і надсилає його користувачу на електронну пошту або в мобільний додаток.

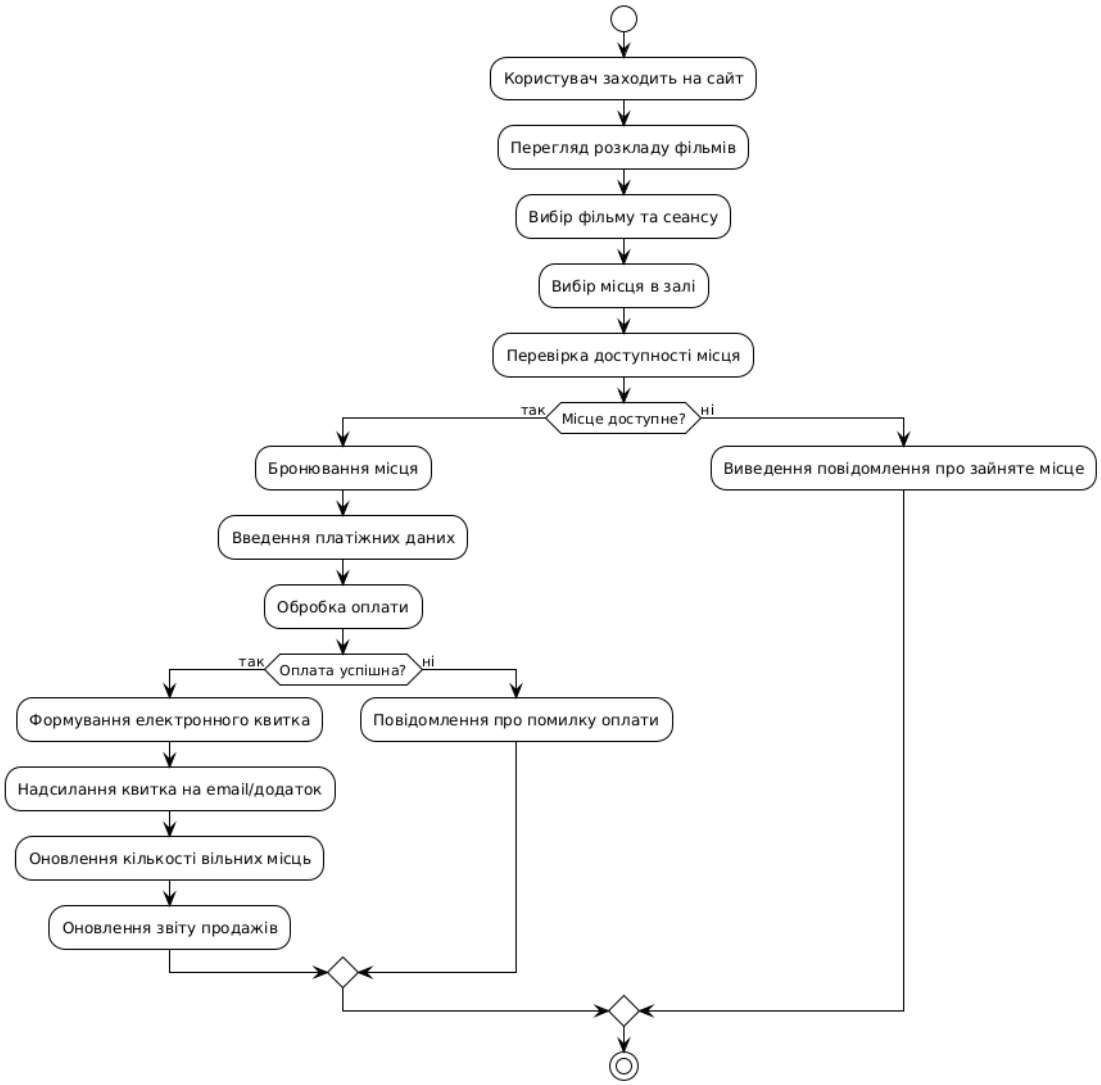

Рисунок 2.11 – Алгоритм автоматизації процесу бронювання квитка

Після цього система оновлює дані про кількість вільних місць у залі та вносить інформацію в звіт продажів для подальшої аналітики.

Якщо ж оплата не проходить, користувач отримує повідомлення про помилку під час транзакції. У випадку, коли місце в залі вже зайняте, система також повідомляє про це, не допускаючи дублювання бронювання.

Запропонована діаграма демонструє ефективну автоматизацію ключового бізнес-процесу – від вибору фільму до підтвердження покупки, з урахуванням перевірок, обробки транзакцій та оновлення стану системи, що суттєво підвищує зручність для клієнта та зменшує навантаження на персонал.

Автоматизація процесу обліку та поповнення запасів снеків є однією з ключових складових ефективного управління ресурсами сучасного кінотеатру. Торговий сегмент, пов’язаний із реалізацією напоїв, попкорну, цукерок та інших супутніх товарів, забезпечує значну частину прибутку закладу, особливо в умовах сезонних коливань відвідуваності. Тому своєчасне поповнення запасів, контроль рівня товарів на складі та зменшення ризиків дефіциту або надлишків є стратегічно важливими завданнями для стабільної роботи підприємства. У межах запропонованої інформаційної системи управління ресурсами кінотеатру передбачено спеціалізований модуль, який відповідає за моніторинг наявності товарів, формування заявок та облік змін у запасах. Цей модуль працює в автоматичному режимі, що суттєво знижує навантаження на персонал та усуває людський фактор у прийнятті рутинних операційних рішень.

Процес обліку та поповнення товарів зображено на рисунку 2.12 і починається з періодичного запуску перевірки, який може здійснюватися щоденно або кілька разів на добу, залежно від інтенсивності продажів.

Система автоматично ініціює аналіз даних про залишки товарів на складі, звертаючись до бази даних, у якій зберігається інформація про кількість кожного виду снеків, терміни придатності та попередні операції закупівлі. Для кожного товару окремо виконується перевірка на відповідність встановленим пороговим значенням.

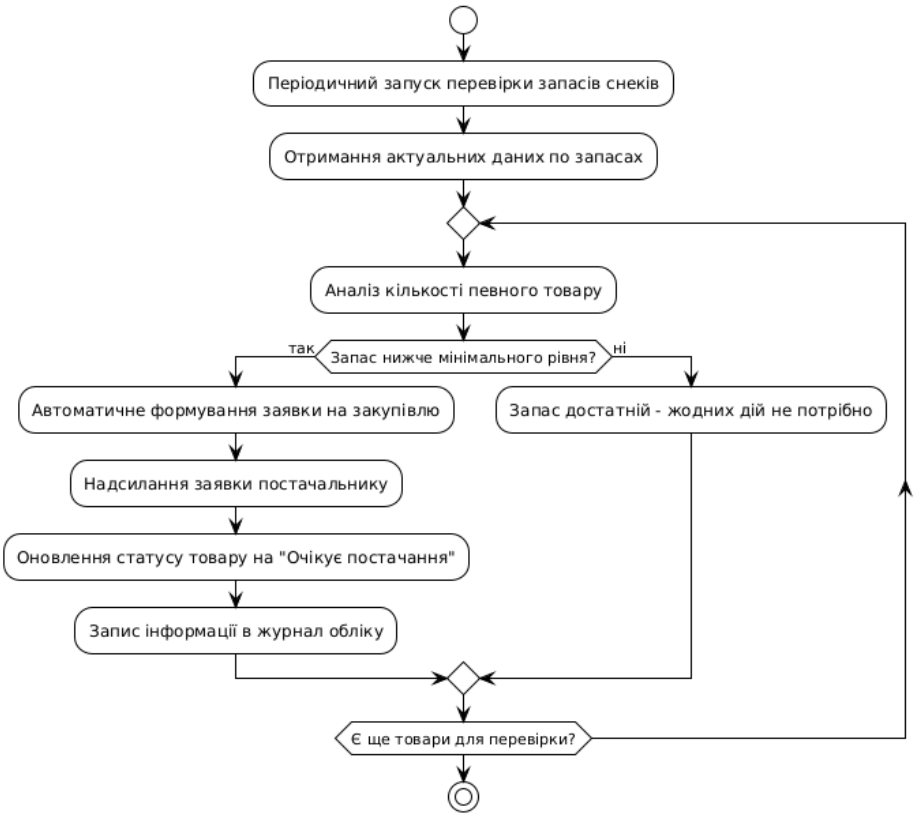

Рисунок 2.12 – Алгоритм автоматизації обліку та поповнення запасів снеків

Ці значення задаються адміністратором системи й можуть бути гнучко налаштовані залежно від категорії товару, його популярності, частоти закупівлі та середнього обсягу споживання.

У разі, якщо система виявляє, що поточна кількість одиниць певного товару опустилася нижче допустимого мінімуму, автоматично формується заявка на закупівлю. У заявці вказується назва товару, кількість, яку необхідно поповнити, бажана дата доставки та контактна інформація для постачальника. Далі ця заявка в автоматичному режимі надсилається обраному постачальнику через відповідний API або електронну пошту, залежно від технічних можливостей інтеграції. Після відправлення заявки статус товару в системі змінюється на «Очікує постачання», що дозволяє менеджерам відстежувати процес виконання замовлення без дублювання дій. Окрім цього, у внутрішній електронний журнал записується інформація про подію: дата й час перевірки, назва товару, рівень залишку, зміни статусу та деталі сформованої заявки. Це дає змогу формувати докладні звіти для подальшого аналізу.

Якщо в ході перевірки система виявляє, що кількість товару відповідає нормі, вона просто переходить до наступної одиниці, не виконуючи жодних додаткових дій. Завдяки цьому досягається економія обчислювальних ресурсів і забезпечується швидке завершення циклу перевірки. Усі операції виконуються у фоновому режимі й не потребують втручання оператора, що особливо важливо в періоди підвищеного навантаження на персонал кінотеатру.

На діаграмі, що представлена у вигляді UML-діаграми діяльності, чітко зображено алгоритм дій автоматизованої системи: починаючи від ініціації перевірки запасів, система виконує цикл обробки кожного товару, виявляє дефіцит, формує відповідні дії та завершує перевірку після обробки всіх позицій. Такий підхід забезпечує прозорість процесів, підвищує точність обліку, мінімізує ризики, пов’язані з людськими помилками, і дозволяє оперативно реагувати на зміну рівня попиту. Згодом, на основі накопичених даних, цей модуль може бути розширено для прогнозного аналізу — наприклад, із застосуванням машинного навчання для передбачення пікових періодів і попередньої закупівлі відповідних обсягів продукції.

Таким чином, автоматизація процесу обліку та поповнення запасів снеків у рамках інформаційної системи кінотеатру є інструментом стратегічного управління, який сприяє підвищенню операційної ефективності, покращенню обслуговування клієнтів і формуванню цілісної, надійної логістичної системи. Це особливо актуально в умовах конкурентного ринку, де швидкість, точність та безперебійність роботи є вирішальними факторами успішності бізнесу.

Усі модулі системи взаємодіють між собою через внутрішній API, що забезпечує обмін даними в реальному часі, а також дозволяє легко інтегрувати сторонні сервіси за потреби. Такий підхід забезпечує стабільну роботу платформи, простоту масштабування та можливість розширення функціоналу без зміни основної архітектури.

У підсумку, автоматизація процесів у рамках запропонованої інформаційної системи забезпечує високу ефективність, надійність та зручність

управління ключовими ресурсами кінотеатру, створюючи конкурентні переваги та підвищуючи якість обслуговування клієнтів.

2.4 Висновки

У другому розділі було виконано комплексну розробку інформаційної системи управління ресурсами кінотеатру, яка охоплює всі ключові аспекти архітектури, функціонування та автоматизації бізнес-процесів. У межах підрозділу 2.1 сформовано структуру програмного модуля системи на основі фреймворку JHipster, що забезпечило інтеграцію сучасних технологій для побудови монолітного веб-застосунку. Завдяки використанню React та Spring Boot досягнуто високого рівня інтерактивності, масштабованості та стабільності функціонування системи. Детальне проектування сутностей, ролей користувачів, типових сценаріїв взаємодії та доступу до даних забезпечило повноту логіки бізнес-процесів кінотеатру.

Розроблено UML-діаграми, які формалізують функціональну та структурну модель системи. Зокрема, реалізовано діаграми варіантів використання, послідовностей, класів, об’єктів, діяльності, компонентів, розгортання, станів і часу. Це дозволило наочно відобразити взаємодію між користувачами, сервісами та базою даних, забезпечивши повне охоплення процесів від вибору фільму й бронювання квитків до обліку персоналу та продажу снеків. Усі діаграми відповідають стандартам UML і враховують специфіку предметної області.

Особливу увагу приділено автоматизації критично важливих процесів. Реалізовано алгоритми автоматичного бронювання місць, формування електронних квитків, перевірки наявності вільних місць та проведення оплати. Окремо розроблено механізм автоматизованого обліку й поповнення запасів снеків, що дозволяє формувати заявки на закупівлю без участі оператора. Запропоновані рішення спрямовані на зменшення впливу людського чинника,

оптимізацію часу обслуговування, забезпечення точності даних та підвищення якості обслуговування клієнтів.

У результаті розробки створено цілісну інформаційну систему, орієнтовану на потреби кінотеатру, яка поєднує в собі гнучку архітектуру, ефективну логіку функціонування та високу ступінь автоматизації. Це забезпечує конкурентні переваги, знижує операційні витрати, спрощує адміністрування та створює передумови для подальшого розвитку платформи із залученням інтелектуальних технологій прогнозування та машинного навчання.

3 РЕАЛІЗАЦІЯ СИСТЕМИ УПРАВЛІННЯ РЕСУРСАМИ КІНОТЕАТРУ

3.1 Варіантний аналіз і обґрунтування вибору мови програмування

Вибір мови програмування та середовища розробки є одним із ключових рішень на початковому етапі створення програмного забезпечення, оскільки саме ці технології визначають технічні обмеження, гнучкість архітектури, швидкість розробки та подальшу підтримку системи. Під час обґрунтування вибору слід враховувати не лише функціональні можливості, а й надійність, продуктивність, доступність бібліотек і досвід команди.

Для реалізації даного застосунку було обрано мову Java, яка зарекомендувала себе як стабільне, багатоцільове середовище програмування з активною підтримкою спільноти. Java є однією з найпопулярніших мов, завдяки чому існує розвинена екосистема інструментів, бібліотек і фреймворків, що значно пришвидшує розробку. Її головною перевагою є незалежність від платформи: додатки, написані на Java, можуть працювати на будь-якій операційній системі, що підтримує JVM (Java Virtual Machine), без необхідності модифікації коду.

Для порівняння було також проаналізовано інші популярні мови – C# та C++. Основні характеристики, які було враховано при порівнянні, наведено в таблиці 3.1.

Результати порівняння свідчать про перевагу Java за більшістю критеріїв, включно з мультиплатформеністю, ефективністю управління пам’яттю та підтримкою сучасних парадигм розробки. Хоча такі мови, як C++ або C#, також мають низку переваг, вони вимагають додаткових зусиль у налаштуванні середовища, обробці пам’яті або створенні кросплатформеного рішення. Java ж дозволяє уникнути зайвих складностей і зосередитись безпосередньо на логіці застосунку.

Таблиця 3.1– Порівняння мов програмування

Мова / Характеристика	Java	C#	C++
Об’єктно-орієнтована	+	+	+
Наявність інтерфейсів	+	+	+
Мультиплатформеність	+	+	-
Управління пам’яттю	+	-	+
Інструкція goto	+	+	-
Багатопотокова компіляція	+	-	+
Використання покажчиків	+	+	-
Сума	7	5	4

Для покращення швидкості обробки та компіляції нового продукту використання багатопотокової компіляції є важливою перевагою. Однак, використання інструкції goto не є перевагою, оскільки це може ускладнити розуміння коду для пересічного програміста, який буде працювати з ним у майбутньому. Автоматичне управління пам'яттю у Java дозволяє уникнути проблем, пов'язаних з роботою оперативної пам'яті. Крім того, можливість працювати без вказівників спрощує і прискорює процес створення нової програми.

Сьогодні Java вважається однією з найкращих мов програмування [30]. Вона забезпечує багатоцільове застосування та незалежність від платформи. Переваги Java, які були вказані в таблиці 3.1, підтверджують її вибір для даного проекту. Тому було прийнято рішення обрати Java як мову програмування для розробки програмного продукту.

Середовище розробки відіграє важливу роль у процесі створення програмного забезпечення, адже воно має забезпечувати зручність роботи, підтримку налагодження, автоматичну компіляцію та інтеграцію з системами контролю версій. Одним із найефективніших інструментів для цього є середовище JetBrains IntelliJ IDEA, яке завдяки своїм можливостям значно

підвищує продуктивність розробника. Цей програмний продукт автоматизує багато рутинних дій, надає підказки при написанні коду, попереджає про можливі помилки та пропонує шляхи їх вирішення. Завдяки вбудованій підтримці фреймворків і технологій, IntelliJ IDEA дозволяє розробникам зосередитися безпосередньо на реалізації функціоналу, не витрачаючи час на налаштування середовища.

Особливо зручною ця система є для створення графічних інтерфейсів у Java, оскільки підтримує інструменти для проєктування форм Swing, що значно спрощує роботу з графічним оформленням. Розумний редактор з автодоповненням прискорює набір коду і зменшує кількість синтаксичних помилок. Інтеграція з системами керування версіями, зокрема Git, дозволяє ефективно організовувати командну роботу, фіксувати зміни у коді й відстежувати історію.

Також інтегровані засоби рефакторингу сприяють безпечному зміненню структури коду без порушення його логіки. Окрім цього, інструменти аналізу якості коду дають змогу виявити недоліки ще до етапу тестування, що значно скорочує цикл розробки й підвищує якість кінцевого продукту. Усе це робить IntelliJ IDEA оптимальним середовищем для реалізації інформаційної системи управління ресурсами кінотеатру.

Таким чином, для розробки програмного продукту було обрано мову програмування Java і середовище розробки JetBrains IntelliJ IDEA. Вибір Java був зумовлений його багатоплатформеністю, ефективним управлінням пам'яттю, широкою підтримкою та великою кількістю ресурсів для розробки. Середовище розробки JetBrains IntelliJ IDEA було обрано через його потужність, інтегровані інструменти, зручний інтерфейс і підтримку важливих функціональних можливостей для розробки графічних додатків. Така комбінація мови програмування та середовища розробки забезпечує ефективний та продуктивний процес розробки програмного забезпечення.

3.2 Розробка інтерфейсу інформаційної системи

Створення зручного й інтуїтивно зрозумілого інтерфейсу є одним із найважливіших етапів розробки інформаційної системи. Саме інтерфейс визначає якість взаємодії користувача з програмним продуктом і суттєво впливає на ефективність виконання завдань. У межах даного проєкту реалізовано повноцінну інтеграцію між фронтендом і бекендом, що забезпечує адаптацію під конкретні бізнес-вимоги та зручну роботу кінцевого користувача.

Фронтенд-частина застосунку створена з використанням бібліотеки React у поєднанні з TypeScript, що забезпечує високий рівень типізації, спрощує підтримку коду та підвищує загальну надійність інтерфейсу. Основою побудови користувацьких сторінок є компонентний підхід, де кожен елемент системи — сторінка фільмів, форма бронювання, список квитків, панель адміністрування — є окремим самодостатнім компонентом, який легко масштабується й розширюється.

Інтерфейс побудований за принципами адаптивного дизайну, що забезпечує коректне відображення як на десктопах, так і на мобільних пристроях. Це реалізується за допомогою бібліотеки Bootstrap 5, яка дозволяє створювати гнучку сітку розташування елементів та дотримуватися принципу mobile-first. Також інтерфейс підтримує багатомовність, з можливістю перемикання між українською та англійською мовами, при цьому текстові ресурси зручно винесено у файли локалізації.

Кожен інтерфейсний компонент взаємодіє з серверною частиною системи через REST API. При виборі фільму, перегляді доступних місць або оформленні замовлення React-компоненти надсилають запити до відповідних контролерів, отримують необхідні дані й динамічно оновлюють вміст сторінки. Такий підхід забезпечує швидкість обробки запитів і безперервність користувацької взаємодії без потреби у перезавантаженні.

Таким чином, завдяки використанню JHipster було створено цілісну структуру користувацького інтерфейсу, яка відповідає сучасним вимогам до

функціональності, ергономіки та гнучкості. Вона легко адаптується до розширення системи, дозволяє швидко додавати нові модулі та сутності, зберігаючи при цьому єдиний стиль і логіку побудови всіх компонентів застосунку.

Макет головного вікна застосунку було розроблено з дотриманням цих принципів, як показано на рисунку 3.1.

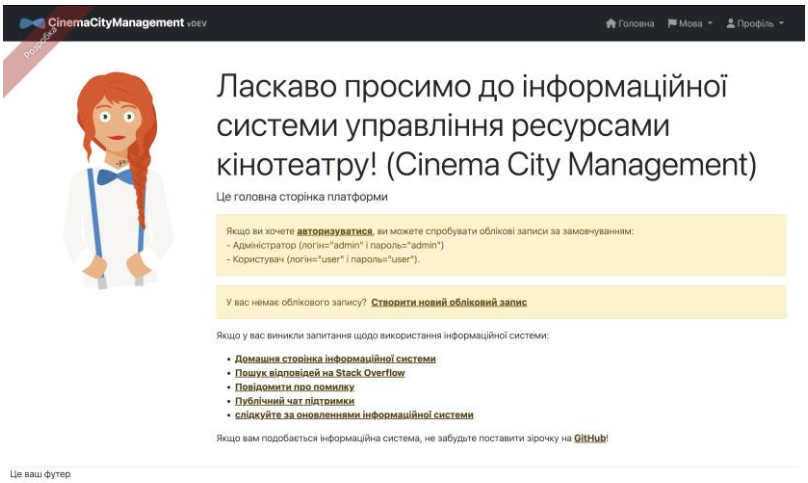

Рисунок 3.1 – Макет головного вікна застосунку

Важливою частиною стала реалізація механізму входу до системи, який забезпечує автентифікацію користувачів і захищений доступ до функціональних можливостей відповідно до їх ролей. Вхід до системи реалізовано у вигляді модального вікна з формою авторизації, яка містить поля для введення логіна (електронної пошти) та пароля, як це зображено на рисунку 3.2.

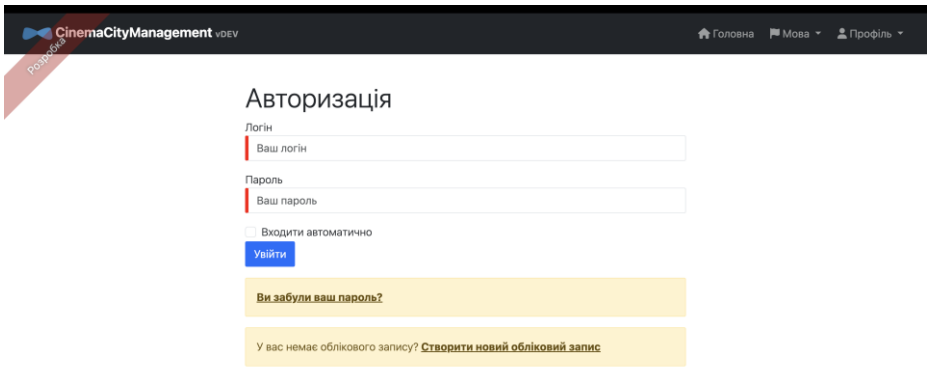

Рисунок 3.2 – Модель вікна з формою авторизації

Після успішного входу користувач отримує персоналізований доступ до інтерфейсу, а його роль визначає обсяг доступних функцій.

Механізм автентифікації базується на токенах безпеки, що передаються клієнтською частиною до сервера при кожному запиті. Токен зберігається у локальному сховищі браузера, що дозволяє підтримувати сесію користувача без повторного введення пароля під час навігації між сторінками. Захищені маршрути забезпечують обмеження доступу до певних частин системи залежно від прав доступу, що підвищує рівень безпеки та запобігає несанкціонованому перегляду даних.

Реалізація входу до системи є важливою складовою взаємодії з користувачем, оскільки вона визначає перший етап доступу до сервісу. У рамках проєкту забезпечено не лише функціональність авторизації, а й зручність її використання, а також відповідність сучасним вимогам безпеки для подібних інформаційних систем.

Після входу користувач бачить навігаційне меню з розділами, що відповідають його ролі. Наприклад, звичайний користувач може переглядати розклад фільмів, бронювати квитки, переглядати замовлення, тоді як адміністратор отримує доступ до розділів управління користувачами, репертуаром, залами, продажами та запасами як це зображено на рисунку 3.3. Інтерфейс дозволяє вийти з системи в один клік, після чого токен знищується й доступ до персональних даних блокується.

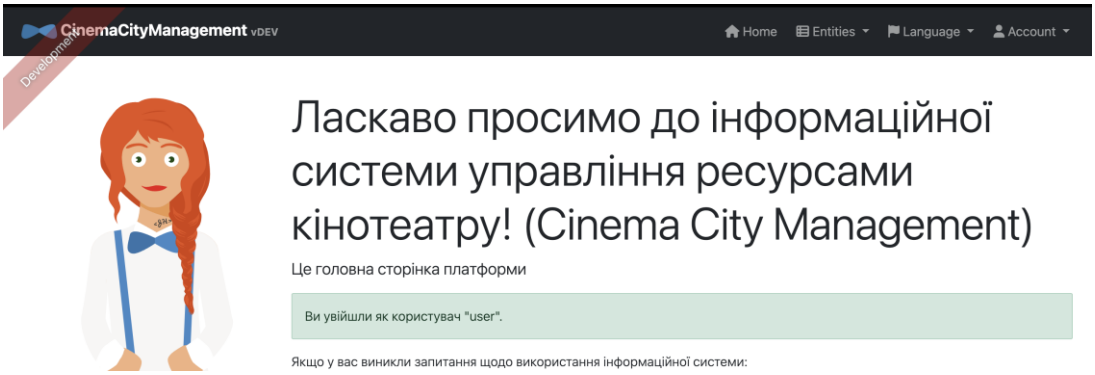

Рисунок 3.3 – Модель вікна звичайного користувача

Особливої уваги в інформаційній системі управління ресурсами кінотеатру заслуговує реалізація адміністративного входу, який забезпечує доступ до функціональності, недоступної для звичайних користувачів. Після автентифікації з обліковими даними адміністратора інтерфейс автоматично підлаштовується під розширений набір прав, і користувачу відкриваються додаткові розділи керування сутностями системи.

Адміністративна частина застосунку включає інтерфейси для створення, редагування та видалення фільмів, розкладів, залів, товарних запасів, а також перегляду статистики продажів, історії замовлень і керування користувачами як це зображено на рисунку 3.4.

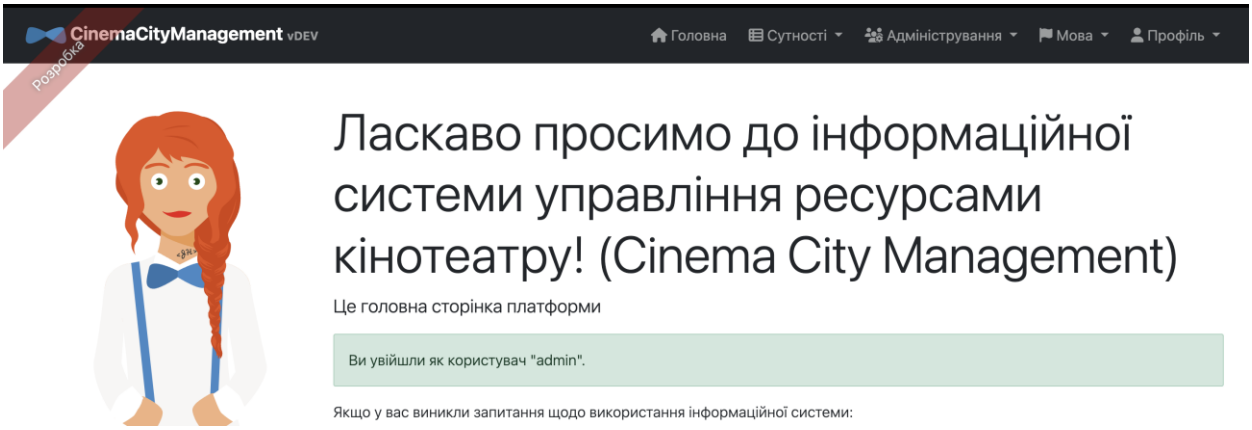

Рисунок 3.4 – Модель головного вікна адміністратора

Кожен розділ містить таблиці з можливістю фільтрації, сортування та пагінації, що полегшує обробку великих обсягів даних. Інтерфейс розроблено з урахуванням зручності щоденного користування, передбачено швидкий перехід між сутностями, форму попереднього перегляду та спливаючі повідомлення про результати операцій.

Управління сутностями системи можна виконувати за допомогою вибору відповідної назви у панелі «Entities», як це зображено на рисунку 3.5.

Усі операції, що виконуються в адміністративному режимі, супроводжуються перевіркою прав доступу на сервері, а також логуються для забезпечення прозорості змін.

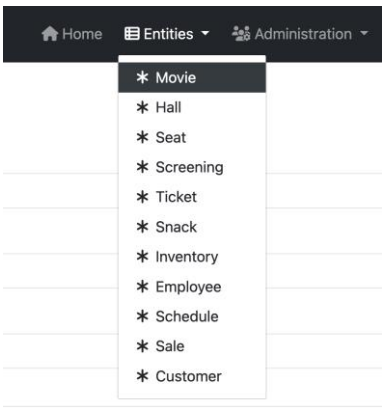

Рисунок 3.5 – Вибір керування сутностями

Таким чином, адміністратор може працювати із критичними даними системи, не побоюючись помилкових змін чи втрати інформації з боку інших користувачів. Крім того, у рамках адміністративного входу реалізовано доступ до розширених налаштувань системи, включаючи параметри безпеки, інтернаціоналізації, оновлення конфігурації й запуск службових операцій, як це зображено на рисунку 3.6.

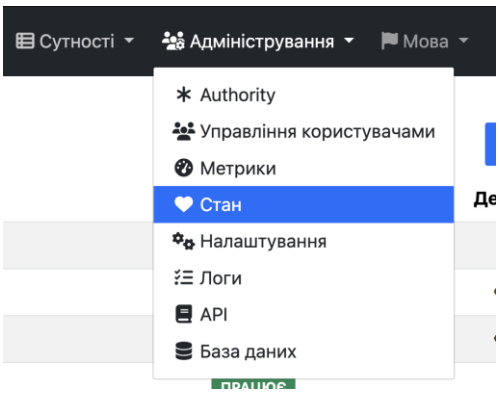

Рисунок 3.6 – Додаткові налаштування вікна адміністратора

Для інтерфейсу інформаційної системи управління ресурсами кінотеатру була створена окрема сторінка для роботи з клієнтами, яка дозволяє ефективно відображати й керувати даними зареєстрованих користувачів, що зображено на рисунку 3.7. Основна мета цієї частини системи полягає в забезпеченні швидкого доступу до інформації про клієнтів для адміністративного персоналу, а також у наданні інструментів для фільтрації, пошуку, перегляду та редагування записів.

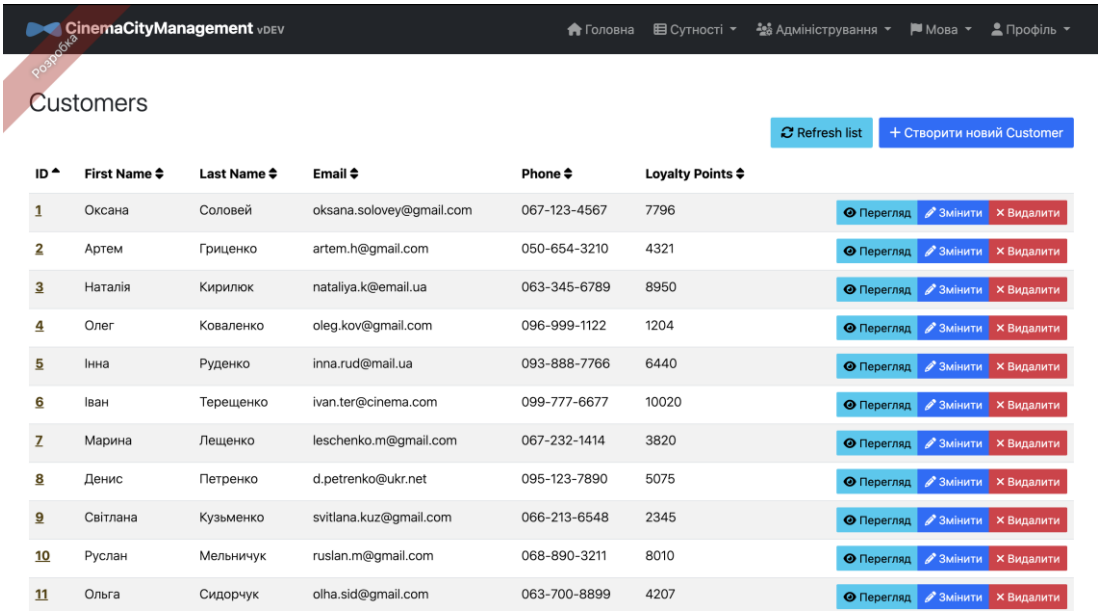

Рисунок 3.7 – Сторінка адміністрування користувачів

Сторінка відображає таблицю, яка містить ключові поля, пов’язані з клієнтами кінотеатру, включно з ідентифікатором запису (ID), ім’ям (First Name), прізвищем (Last Name), адресою електронної пошти (Email), номером телефону (Phone) та накопиченими балами лояльності (Loyalty Points). Завдяки табличному поданню даних адміністратор може швидко зорієнтуватися у списку користувачів і при потребі перейти до детального перегляду або редагування профілю окремого клієнта.

Для зручності використання реалізовано можливість сортування даних у таблиці за будь-яким зі стовпців, що дозволяє, наприклад, швидко знайти клієнтів з найбільшою кількістю балів або відсортувати користувачів за алфавітом. Окремий механізм фільтрації забезпечує пошук за іменем, прізвищем чи адресою електронної пошти, що особливо корисно у випадках великої кількості зареєстрованих користувачів.

Інтерфейс також передбачає відображення кількості балів лояльності, які клієнти можуть використовувати для участі в акційних пропозиціях або отримання знижок. Це дозволяє менеджеру відстежувати активність постійних клієнтів і оперативно реагувати на зміни у системі лояльності. Розширені функції

редагування дозволяють оновлювати контактні дані, змінювати статус активності профілю або анулювати доступ користувача при необхідності.

Аналогічно до сторінки клієнтів, в адміністративному інтерфейсі було реалізовано окремі таблиці для управління іншими ключовими сутностями системи.

На прикладі зображення 3.8 показано сторінку, що відповідає за облік снеків, які реалізуються в кінотеатрі. Візуально таблиця побудована за тим же принципом: кожен рядок представляє окремий товар, а стовпці містять інформацію про унікальний ідентифікатор, назву, ціну, кількість на складі та наявність акційної знижки.

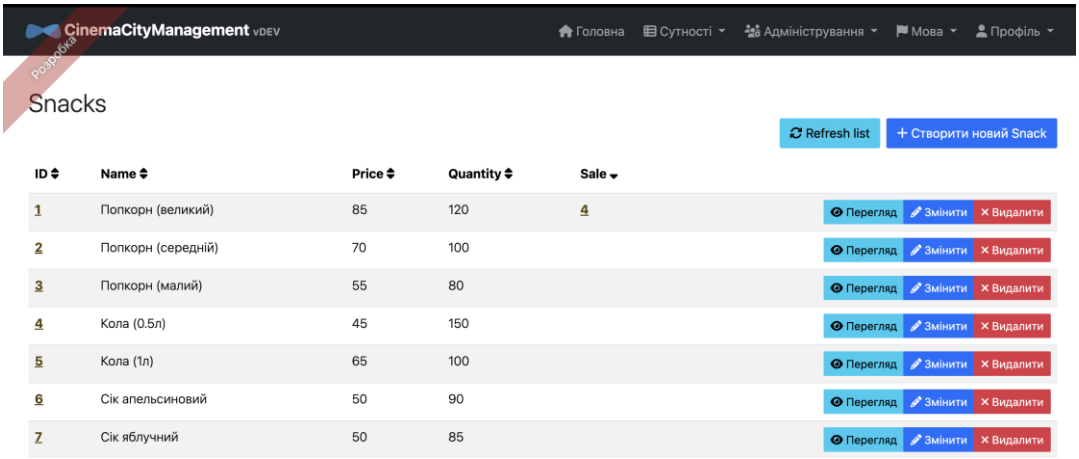

Рисунок 3.8 – Інтерфейс сторінки обліку снеків

Користувачеві доступні основні операції над кожним записом: перегляд деталей, редагування значень або видалення елемента зі списку. Додаткові елементи керування дозволяють оновити таблицю в один клік або створити новий запис про товар. Завдяки єдиному підходу до побудови інтерфейсів адміністратор швидко орієнтується в усіх функціональних розділах, а повторне використання шаблонів забезпечує послідовність і передбачуваність взаємодії з системою.

Кожна таблиця є логічно ізольованою, але водночас взаємопов’язана з іншими через загальні бізнес-процеси: наприклад, облік снеків пов’язано з формуванням продажів і замовлень, тому швидке оновлення цих даних має

вирішальне значення для точності фінансової звітності та коректності роботи касових модулів. Такий підхід дозволяє ефективно автоматизувати управління ресурсами та забезпечити повну прозорість усіх внутрішніх процесів у межах інформаційної системи кінотеатру.

У розглянутому прикладі інтерфейсу сторінки обліку снеків, зображеному на рисунку 3.8, в останньому стовпці таблиці, що позначений як “Sale”, вказано значення «4». Це значення не є числовим показником кількості чи вартості, а натомість слугує інтерактивним елементом (ID), який виступає у ролі посилання на конкретну транзакцію продажу. Інтерфейс сторінки продажу зображено на рисунку 3.9.

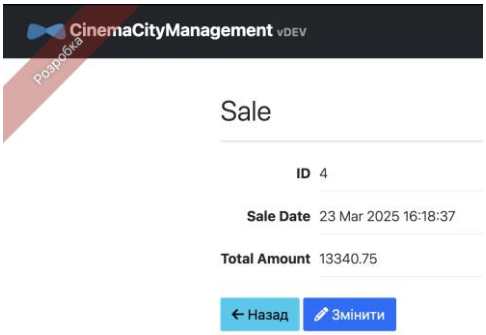

Рисунок 3.9 – Інтерфейс сторінки продажу

Зокрема, число «4» в даному випадку є унікальним ідентифікатором відповідного запису на сторінці “Продажі” (Sales). При натисканні на нього користувач переходить до детальної інформації про обрану операцію. Наприклад, для ідентифікатора 4 буде відкрито сторінку, де зазначено, що продаж відбувся 23 березня 2025 року о 16:18:37, а загальна сума операції склала 13340,75 грн.

Запропонований функціонал дозволяє швидко відстежити, в рамках якої саме транзакції був проданий певний снек, що суттєво полегшує контроль залишків, перевірку касових записів та аудит фінансових операцій. Водночас це забезпечує логічний зв’язок між сутностями «Снеки» та «Продажі», що є

важливою складовою внутрішньої узгодженості інформаційної системи управління кінотеатром.

Ще однією з важливих функцій інтерфейсу інформаційної системи управління ресурсами кінотеатру є підтримка багатомовності, що дозволяє охопити ширше коло користувачів, зокрема працівників та клієнтів, які володіють різними мовами. Для цього було реалізовано механізм вибору мови інтерфейсу, який представлено у вигляді випадаючого меню у верхній панелі навігації, як показано на рисунку 3.10.

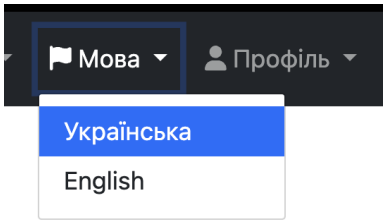

Рисунок 3.10 – Компонент вибору мови інтерфейсу

Користувач має можливість миттєво перемикатися між доступними мовами — у цьому випадку між українською та англійською. Зміна мови відбувається без перезавантаження сторінки: всі текстові елементи, такі як назви розділів, підписи до полів, повідомлення та кнопки, автоматично адаптуються до обраної локалі.

Механізм реалізовано шляхом використання файлів локалізації, у яких зберігаються всі текстові ресурси інтерфейсу. Під час перемикання мови система звертається до відповідного набору ресурсів і оновлює інтерфейс відповідно до обраного перекладу. Це значно полегшує підтримку багатомовного застосунку, дає змогу централізовано оновлювати переклади та додавати нові мови у разі потреби.

Такий підхід особливо корисний у випадку, коли персонал кінотеатру включає працівників, що користуються різними мовами, або ж застосунок розгортається у багатомовному регіоні. Забезпечення зручності сприйняття інформації користувачем сприяє покращенню загального досвіду взаємодії із

системою, зменшує кількість помилок під час введення даних та полегшує навчання нових працівників.

Важливим компонентом інтерфейсу інформаційної системи управління ресурсами кінотеатру є функціональність налаштувань профілю користувача. Доступ до цих налаштувань здійснюється через випадаюче меню “Профіль”, розташоване у верхній правій частині інтерфейсу, як показано на рисунку 3.11.

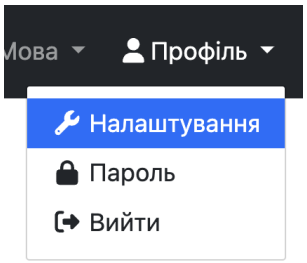

Рисунок 3.11 – Випадаюче меню налаштувань профілю користувача

Меню надає три основні опції: “Налаштування”, “Пароль” та “Вийти”. Вибравши пункт “Налаштування”, користувач переходить до окремої сторінки, де може переглянути та змінити інформацію власного облікового запису, зокрема ім'я, прізвище, адресу електронної пошти, а також обрану мову інтерфейсу. Приклад сторінки налаштувань користувача представлений на рисунку 3.12.

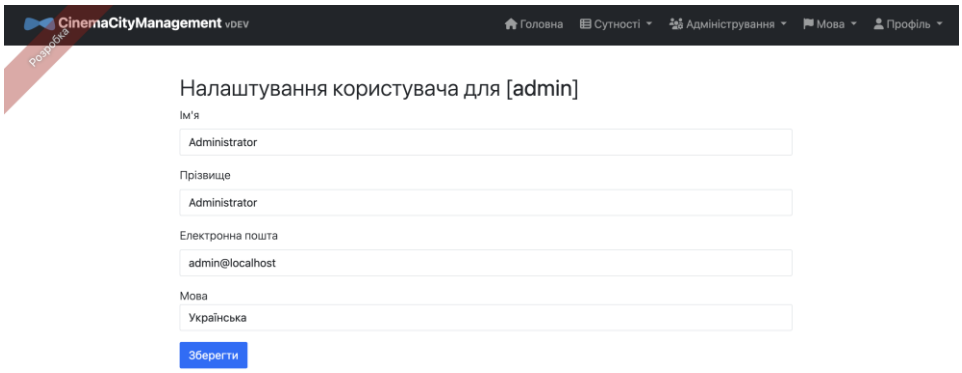

Рисунок 3.12 – Сторінка налаштувань користувача

Інтерфейс цієї сторінки розроблено максимально просто і зрозуміло, що дозволяє користувачу швидко та без зайвих складнощів змінювати потрібні

параметри. Для оновлення інформації необхідно внести відповідні зміни у текстові поля і натиснути кнопку “Зберегти”. Система автоматично перевіряє коректність введених даних та зберігає зміни у профілі.

Особливо важливою є можливість зміни мови інтерфейсу у налаштуваннях профілю користувача. Ця функція дублює швидкий вибір мови у верхньому меню, проте дозволяє закріпити вибрану мову для конкретного профілю, що забезпечує її використання одразу після наступного входу в систему без додаткових дій з боку користувача.

Отримана реалізована система налаштувань профілю сприяє персоналізації інтерфейсу, покращує користувацький досвід і дозволяє зручно й оперативно керувати власною інформацією, що є важливим елементом у загальній структурі інформаційної системи управління кінотеатром.

Ще одним важливим аспектом налаштувань профілю є можливість зміни пароля користувача. Ця функція доступна через меню “Профіль”, обравши пункт “Пароль”. Після цього відкривається окрема сторінка з формою для зміни пароля, що зображено на рисунку 3.13.

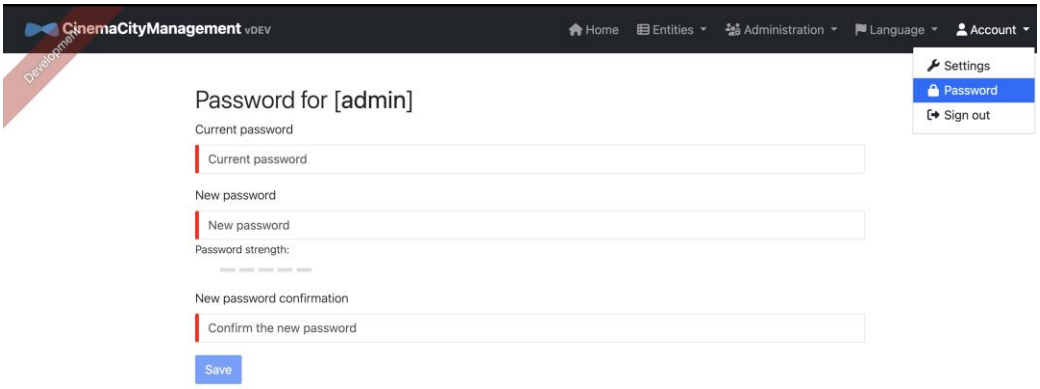

Рисунок 3.13 – Інтерфейс зміни пароля користувача

Інтерфейс сторінки зміни пароля включає три поля введення: поточний пароль (Current password), новий пароль (New password) і підтвердження нового пароля (New password confirmation). Така організація форми спрямована на підвищення безпеки користувацьких даних та запобігання помилок під час введення нового пароля.

Додатковою функцією форми є інтерактивний індикатор надійності нового пароля (Password strength), який динамічно показує ступінь складності та захищеності введених символів. Це допомагає користувачеві створити більш надійний пароль і підвищує загальну безпеку інформаційної системи.

Після введення всіх необхідних даних користувач натискає кнопку “Save” для підтвердження зміни пароля. Система автоматично перевіряє правильність поточного пароля, збіг нового пароля та його підтвердження, а також оцінює складність нового пароля. У разі успішної перевірки пароль оновлюється в системі, після чого користувач може використовувати нові дані для подальшого входу.

Реалізація механізму зміни пароля забезпечує додатковий рівень безпеки облікового запису та дозволяє користувачам оперативно реагувати на можливі загрози несанкціонованого доступу. Це є ключовим аспектом у підтримці надійності та безпеки даних у межах інформаційної системи управління кінотеатром.

Окремою частиною адміністративного інтерфейсу інформаційної системи управління ресурсами кінотеатру є доступ до документації API, яка реалізована з використанням інструмента Swagger. Цей інструмент надає повну інформацію про доступні REST-контролери та методи, які можуть бути викликані з клієнтської частини застосунку або інших зовнішніх сервісів. Інтерфейс цієї сторінки зображено на рисунку 3.14.

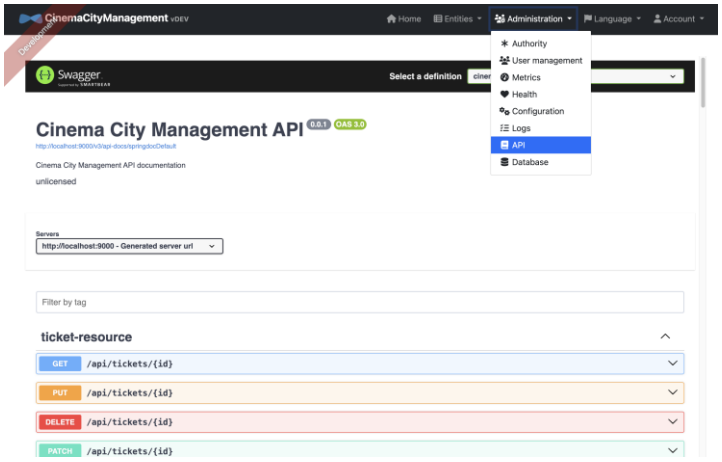

Рисунок 3.14 – Інтерфейс документації API

Сторінка документації дозволяє переглядати всі доступні методи та ресурси, які забезпечують взаємодію клієнтської та серверної частин. Вона містить перелік доступних HTTP-запитів (GET, POST, PUT, PATCH, DELETE) для кожного ресурсу, наприклад, для ресурсу квитків (ticket-resource). Кожен метод супроводжується детальним описом, що включає параметри запиту, формат відповіді, приклади виконання запитів та відповіді сервера.

Завдяки інтерактивній документації адміністратор або розробник може перевірити роботу API безпосередньо у браузері, надсилаючи тестові запити до сервера та отримуючи результати в реальному часі. Це суттєво спрощує процес інтеграції нових компонентів системи, відлагодження вже наявних, а також забезпечує прозорість та зручність документування можливостей інформаційної системи.

Даний підхід дозволяє забезпечити високий рівень сумісності з іншими програмними системами, спрощує підтримку та розвиток застосунку, а також зменшує ймовірність виникнення помилок під час взаємодії клієнтських і серверних компонентів інформаційної системи управління кінотеатром.

Ще важлива є сторінка моніторингу стану роботи основних сервісів і компонентів, яка реалізована під назвою “Health Checks”. Ця сторінка надає адміністратору швидкий огляд поточного стану критично важливих сервісів системи, таких як база даних, брокер повідомлень, доступний дисковий простір та загальний стан застосунку. Візуально ця сторінка представлена на рисунку 3.15.

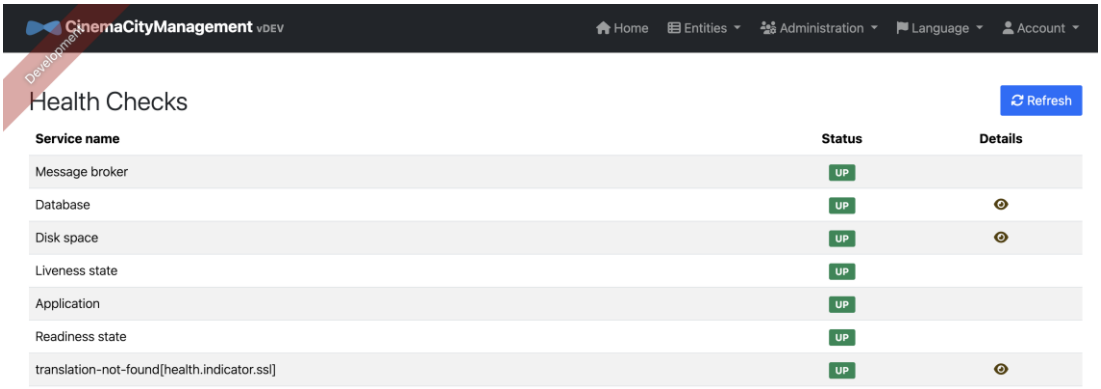

Рисунок 3.15 – Сторінка перевірки стану сервісів (Health Checks)

На сторінці відображається перелік сервісів та їх статус у реальному часі. Статуси позначаються за допомогою кольорових індикаторів: статус «UP» (зелений колір) означає, що сервіс працює нормально, а будь-який інший статус (наприклад, «DOWN») свідчить про наявність проблеми. Для кожного сервісу також доступна детальніша інформація, яку можна переглянути, натиснувши на відповідний значок у стовпці “Details”.

Такий моніторинг є необхідним для оперативного реагування на можливі збої або нестабільності у роботі інформаційної системи, дозволяючи адміністратору швидко виявляти та усувати неполадки. Це особливо важливо для забезпечення стабільності й безперервності роботи системи, що прямо впливає на якість обслуговування клієнтів кінотеатру та загальну ефективність управління ресурсами.

3.3 Розробка компонент системи управління ресурсами кінотеатра

Серверна частина інформаційної системи управління ресурсами кінотеатру є ключовим елементом програмної архітектури, оскільки саме вона забезпечує обробку бізнес-логіки, взаємодію з базою даних, а також відповідає за безпечну й надійну комунікацію між користувачем і системою. Для реалізації серверної частини було використано стек технологій Spring Boot у поєднанні з JHipster — генератором застосунків, що дозволяє швидко створювати надійні корпоративні рішення з підтримкою безпеки, інтернаціоналізації, розширюваної структури даних і REST API.

Основою серверної частини є набір контролерів, сервісів і репозиторіїв, які реалізують основні функції системи: управління репертуаром фільмів, розкладами, залами, замовленнями, продажами, клієнтами, обліком товарів та адмініструванням користувачів. Завдяки архітектурі REST кожна функція має відповідний кінцевий пункт API, що дозволяє клієнтській частині взаємодіяти з даними через HTTP-запити у форматі JSON.

Важливою особливістю є підтримка CRUD-операцій (Create, Read, Update, Delete) для кожної сутності. Усі дані зберігаються у базі даних PostgreSQL, з якою серверна частина працює через ORM-технологію JPA/Hibernate. Це забезпечує зручну абстракцію над реляційною моделлю та дозволяє уникнути написання складних SQL-запитів. При кожній зміні моделі база даних оновлюється за допомогою Liquibase, що автоматизує керування міграціями та гарантує стабільність структури даних під час розвитку системи.

Серверна частина також забезпечує автентифікацію та авторизацію користувачів. Для цього застосовується JWT-токен, який генерується після успішного входу в систему та передається з кожним запитом до захищених ресурсів. Усі ролі та права доступу визначені централізовано й перевіряються на рівні контролерів і сервісів, що забезпечує надійність та захист конфіденційної інформації.

Для покращення продуктивності та підтримки масштабованості були використані кешування відповідей та асинхронна обробка запитів там, де це доцільно, наприклад, при генерації статистичних звітів або обробці великих обсягів транзакцій. Серверна частина інтегрована з системою моніторингу, що надає адміністратору можливість переглядати стан сервісів, кількість запитів, тривалість виконання операцій і швидко виявляти потенційні вузькі місця в системі.

Завдяки використанню Spring Boot і JHipster структура серверної частини залишається гнучкою та розширюваною: кожен новий функціональний модуль додається як окрема сутність із власними маршрутами, сервісами, DTO й репозиторіями. Це дозволяє масштабувати проєкт без порушення існуючої логіки, а також легко тестувати окремі компоненти як у модульному, так і в інтеграційному середовищі.

У межах розробки інформаційної системи управління ресурсами кінотеатру важливим елементом є конфігурація застосунка, яка реалізується за допомогою YAML-файлів. Вони дозволяють централізовано керувати параметрами середовища, безпеки, взаємодії з базами даних, API, метрик,

кешування та багатьох інших аспектів. Завдяки підтримці профілів Spring Boot конфігурації адаптуються під різні середовища, зокрема dev, prod, api-docs.

Розглянемо налаштування запуску серверної частини в режимі розробки системи:

```
spring:
  devtools:
    restart:
      enabled: true
      additional-exclude: static/**,.h2.server.properties
    livereload:
      enabled: false
  docker:
    compose:
      enabled: true
      profiles:
        active: dev
  jackson:
    serialization:
      indent-output: true
  datasource:
    type: com.zaxxer.hikari.HikariDataSource
    url: jdbc:h2:file:./target/h2db/db/cinemaCityManagement;DB_CLOSE_DELAY=-1
    username: cinemaCityManagement
    password:
    hikari:
      poolName: Hikari
      auto-commit: false
  h2:
    console:
             enabled: true
  liquibase:
        contexts: dev, faker
  mail:
    host: localhost
    port: 25
    username:
    password:
  messages:
    cache-duration: PT1S # 1 second, see the ISO 8601 standard
  thymeleaf:
    cache: false
server:
  port: 8080
  forward-headers-strategy: native
jhipster:
  cache: # Cache configuration
    ehcache: # Ehcache configuration
      time-to-live-seconds: 3600 # By default objects stay 1 hour in the cache
      max-entries: 100 # Number of objects in each cache entry
  cors:
    allowed-origins:
'http://localhost:8100,https://localhost:8100,http://localhost:9000,https://loca
lhost:9000,http://localhost:4200,https://localhost:4200'
    # Enable CORS when running in GitHub Codespaces
    allowed-origin-patterns: 'https://*.githubpreview.dev'
    allowed-methods: '*'
    allowed-headers: '*'
    exposed-headers: 'Authorization,Link,X-Total-Count,X-
${jhipster.clientApp.name}-alert,X-${jhipster.clientApp.name}-error,X-
```

```
${jhipster.clientApp.name}-params'
    allow-credentials: true
    max-age: 1800
  security:
    authentication:
      jwt:
        =        token-validity-in-seconds: 86400
        token-validity-in-seconds-for-remember-me: 2592000
  mail:    base-url: http://127.0.0.1:8080
  logging:
    use-json-format: false # By default, logs are not in Json format
    logstash: # Forward logs to logstash over a socket, used by
LoggingConfiguration
      enabled: false
      host: localhost
      port: 5000
      ring-buffer-size: 512
```

Однією з ключових переваг є гнучкість профілювання. Наприклад, конфігурація application.yml є базовою, але може бути доповнена або перевизначена специфічними файлами, як-от application-prod.yml, залежно від активного профілю. Це дозволяє, наприклад, у розробницькому середовищі активувати документацію API або розширену діагностику, а в продакшн — вимкнути ці можливості задля підвищення безпеки та продуктивності.

Параметри spring.application.name, spring.jpa.properties.hibernate.*, spring.datasource.* задають назву програми, поведінку ORM-шару та з’єднання з базою даних PostgreSQL. Автоматичні міграції структури бази даних виконуються за допомогою Liquibase. У секції spring.jpa.hibernate.naming застосовано стратегію CamelCaseToUnderscores, що забезпечує зрозуміле відображення Java-сутностей у вигляді SQL-таблиць.

Блок spring.cloud.stream.kafka відповідає за інтеграцію з брокером повідомлень Kafka. Це забезпечує обробку асинхронних подій, зокрема обробку повідомлень між модулями системи — наприклад, для розсилки повідомлень, оновлення запасів чи логування подій. Включено конфігурації продюсера та консьюмера, а також автогенерацію тем.

Безпека конфігурується через spring.security.oauth2.resourceserver.jwt, зокрема вказано префікси прав доступу, використання JWT-токенів і ролі для доступу до певних ендпоінтів. Тривалість життя токена також визначено — 24 години для звичайної сесії та 30 днів для режиму «запам’ятати мене».

Окрему увагу приділено системі моніторингу: у конфігурації management.endpoints.web.exposure явно задано перелік ендпоінтів, які можна переглядати через /management. Наприклад, відображаються метрики, журнал логів, конфігурація, інформація про систему та статуси сервісів (health, loggers, threaddump, prometheus, тощо). Стан сервісів (liveness, readiness) згруповано в окремі підрозділи й доступний лише користувачам із правами ROLE_ADMIN.

Файл також містить параметри кешування через Ehcache (jhipster.cache.ehcache), де зазначено кількість записів у кеші та час їхнього зберігання. Це підвищує швидкодію при роботі з даними, які часто повторюються. У конфігурації також прописано використання поштового сервера, параметри логування, а також можливість активувати CORS (Cross-Origin Resource Sharing), що корисно для інтеграції з фронтенд-застосунками.

Таким чином, конфігурація застосунка забезпечує гнучке налаштування всієї інформаційної системи — від підключення до бази даних і моніторингу до безпеки, Kafka, кешу та документації API. Централізоване управління параметрами через YAML-файли спрощує підтримку, масштабування та адаптацію системи до різних середовищ розгортання, зберігаючи високу надійність і керованість проєкту.

Розглянемо детальніше java код серверного застосунку. Пакет com.cinemacity.management.domain містить фундаментальні класи, що визначають модель даних інформаційної системи управління ресурсами кінотеатру. Одним із базових класів, що забезпечує важливі можливості для всіх сутностей, є абстрактний клас AbstractAuditingEntity:

```
package com.cinemacity.management.domain;
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
@JsonIgnoreProperties(value = { "createdBy", "createdDate", "lastModifiedBy",
"lastModifiedDate" }, allowGetters = true)
public abstract class AbstractAuditingEntity<T> implements Serializable {
    private static final long serialVersionUID = 1L;
    public abstract T getId();
    @CreatedBy
    @Column(name = "created_by", nullable = false, length = 50, updatable =
false)
    private String createdBy;
    @CreatedDate
    @Column(name = "created_date", updatable = false)
    private Instant createdDate = Instant.now();
```

```
    @LastModifiedBy
    @Column(name = "last_modified_by", length = 50)
    private String lastModifiedBy;
    @LastModifiedDate
    @Column(name = "last_modified_date")
    private Instant lastModifiedDate = Instant.now();
}
```

Даний клас відіграє ключову роль у побудові єдиної логіки для автоматичного відстеження історії змін будь-яких об'єктів бази даних. Його головне призначення полягає в централізованому зберіганні інформації про те, коли і ким були створені або змінені відповідні записи.

Використання анотації @MappedSuperclass дозволяє наслідувати властивості класу без створення окремої таблиці в базі даних для самого класу AbstractAuditingEntity. Це означає, що кожна сутність, яка успадковує цей клас, автоматично отримує поля для аудиту:

- createdBy — ім'я користувача, який створив запис;
- createdDate — дата й час створення запису;
- lastModifiedBy — ім'я користувача, який останнім вносив зміни;
- lastModifiedDate — дата й час останньої зміни запису.

Для автоматичного заповнення цих полів використовуються спеціалізовані анотації з бібліотеки Spring Data JPA:

- @CreatedBy та @LastModifiedBy — вказують на користувача, який виконав операцію (створення або зміну);
- @CreatedDate та @LastModifiedDate — автоматично заповнюються поточною датою і часом виконання операції відповідно.

Додатково клас використовує анотацію @EntityListeners (AuditingEntityListener.class), що забезпечує прослуховування подій життєвого циклу сутностей для автоматичної фіксації інформації про створення та зміни записів.

Поля класу також позначені спеціальними атрибутами в анотації @Column, які додатково гарантують коректне створення таблиць і стовпців у базі даних:

- updatable = false означає, що поля createdBy і createdDate не змінюються після їх первинного запису в базу;

- length = 50 вказує максимальну довжину текстових полів, що є оптимальним для зберігання імен користувачів системи;
- nullable = false гарантує, що поле createdBy не може бути порожнім, забезпечуючи таким чином обов'язковість його заповнення.

Клас містить метод getId(), який реалізується в конкретних сутностях і дозволяє однозначно ідентифікувати запис у базі даних, підтримуючи при цьому типову для JPA структуру сутностей.

Застосування такого підходу забезпечує єдиний стандарт ведення журналу змін сутностей, що значно спрощує аудит змін у системі, забезпечує надійність і прозорість операцій, а також зменшує повторення коду у конкретних сутностях. В результаті це суттєво спрощує подальшу підтримку та розвиток програмного рішення.

Клас SecurityMetersService, який знаходиться у пакеті com.cinemacity.management.management, є частиною системи моніторингу безпеки інформаційної системи управління кінотеатром. Основним завданням цього сервісу є відстеження та реєстрація помилок, що виникають при обробці JWT-токенів. Для забезпечення цього функціоналу застосовується бібліотека Micrometer, що дозволяє формувати та передавати детальні метрики про стан безпеки системи у різноманітні інструменти моніторингу, такі як Prometheus.

Частина коду SecurityMetersService:

```
package com.cinemacity.management.management;
@Service
public class SecurityMetersService {
    public static final String INVALID_TOKENS_METER_NAME =
"security.authentication.invalid-tokens";
    public static final String INVALID_TOKENS_METER_DESCRIPTION =
        "Indicates validation error count of the tokens presented by the
clients.";
    public static final String INVALID_TOKENS_METER_BASE_UNIT = "errors";
    public static final String INVALID_TOKENS_METER_CAUSE_DIMENSION = "cause";
    private final Counter tokenInvalidSignatureCounter;
    private final Counter tokenExpiredCounter;
    private final Counter tokenUnsupportedCounter;
    private final Counter tokenMalformedCounter;
    public SecurityMetersService(MeterRegistry registry) {
        this.tokenInvalidSignatureCounter =
invalidTokensCounterForCauseBuilder("invalid-signature").register(registry);
        this.tokenExpiredCounter =
invalidTokensCounterForCauseBuilder("expired").register(registry);
        this.tokenUnsupportedCounter =
invalidTokensCounterForCauseBuilder("unsupported").register(registry);
```

```
        this.tokenMalformedCounter =
invalidTokensCounterForCauseBuilder("malformed").register(registry);
    }
    private Counter.Builder invalidTokensCounterForCauseBuilder(String cause) {
        return Counter.builder(INVALID_TOKENS_METER_NAME)
            .baseUnit(INVALID_TOKENS_METER_BASE_UNIT)
            .description(INVALID_TOKENS_METER_DESCRIPTION)
            .tag(INVALID_TOKENS_METER_CAUSE_DIMENSION, cause);  }
    public void trackTokenInvalidSignature() {
        this.tokenInvalidSignatureCounter.increment();  }
    public void trackTokenExpired() {
        this.tokenExpiredCounter.increment();}
    public void trackTokenUnsupported()
{this.tokenUnsupportedCounter.increment();}
    public void trackTokenMalformed() {this.tokenMalformedCounter.increment();}
}
```

Сервіс має окремі лічильники, кожен з яких відповідає за реєстрацію специфічного типу помилок при роботі з токенами. Ці лічильники створюються під час запуску застосунку і пов'язані з конкретними причинами помилок. Наприклад, існують лічильники для некоректного підпису токена, простроченого терміну дії, використання токенів із непідтримуваним форматом, а також пошкоджених або некоректно сформованих токенів.

Методи класу дозволяють збільшувати значення відповідних лічильників, що дає змогу чітко ідентифікувати кількість та причини помилок автентифікації у реальному часі. Такий підхід сприяє підвищенню оперативності реагування на інциденти безпеки, забезпечує прозорість стану автентифікації користувачів та дозволяє адміністраторам системи завчасно вживати заходів для уникнення або швидкого виправлення проблем. Завдяки цьому значно підвищується загальний рівень захищеності системи, а також спрощується аналіз її стану в процесі експлуатації.

Таким чином, серверна частина інформаційної системи управління ресурсами кінотеатру виконує роль ядра застосунку, забезпечуючи централізовану обробку запитів, керування даними, автентифікацію користувачів, контроль бізнес-логіки та підтримку комунікації між компонентами. Її реалізація відповідає сучасним вимогам до гнучкості, безпеки та продуктивності, що дозволяє ефективно розвивати систему в умовах реального функціонування кінотеатру.

3.4 Висновки

У третьому розділі було проведено детальний аналіз та реалізацію програмних компонентів інформаційної системи управління ресурсами кінотеатру. Обґрунтовано вибір мови програмування Java завдяки її мультиплатформеності, ефективному управлінню пам'яттю та широкій підтримці інструментів для розробки. Також обрано середовище розробки JetBrains IntelliJ IDEA, що забезпечує високу продуктивність, зручність роботи з кодом і графічним інтерфейсом, а також ефективну інтеграцію з інструментами контролю версій.

Було реалізовано інтуїтивний та зручний користувацький інтерфейс на основі бібліотеки React із TypeScript, що дозволило підвищити стабільність і простоту підтримки фронтенд-частини системи. Інтерфейс враховує принципи адаптивності, забезпечує багатомовність, простоту використання та інтегрований механізм авторизації з використанням токенів. Адміністративна частина інтерфейсу має розширений функціонал для управління сутностями, зручну фільтрацію й навігацію, що забезпечує швидку й надійну роботу з великими обсягами даних.

Серверну частину побудовано з використанням сучасних технологій Spring Boot та JHipster. Це дозволило швидко та якісно реалізувати основні модулі системи, забезпечити стабільну роботу з базою даних PostgreSQL через ORM Hibernate та автоматизувати міграцію схеми за допомогою Liquibase. Безпека реалізована на основі JWT-токенів, а додатковий моніторинг і кешування забезпечили ефективність та оперативність роботи системи.

Важливими компонентами стали конфігурації застосунку, організовані за допомогою YAML-файлів, що забезпечують гнучкість налаштування для різних середовищ. Було використано механізми аудиту змін сутностей і спеціальні засоби моніторингу безпеки за допомогою класів AbstractAuditingEntity та SecurityMetersService. Це дозволило підвищити прозорість, захищеність і керованість інформаційної системи.

Таким чином, у результаті виконаних робіт отримано програмне рішення, що відповідає сучасним технологічним стандартам і здатне ефективно підтримувати управління ресурсами кінотеатру в умовах реальної експлуатації. Розроблені компоненти забезпечують достатню гнучкість для подальшого розширення функціоналу, а також зручність експлуатації та підтримки системи.

4 ТЕСТУВАННЯ СИСТЕМИ УПРАВЛІННЯ РЕСУРСАМИ КІНОТЕАТРУ

4.1 Тестування програмного забезпечення

Тестування програмного забезпечення є обов’язковим етапом створення якісного веб-застосунку. Існує кілька теоретичних підходів до тестування, які можуть бути застосовані при перевірці системи управління ресурсами кінотеатру. В першу чергу можна застосовувати функціональне тестування, яке перевіряє відповідність застосунку заданим вимогам і бізнес-логіці [30]. Такий підхід охоплює різні типи перевірок, включаючи позитивне та негативне тестування, що забезпечує ретельне дослідження роботи компонентів за звичайних і виняткових умов.

Наступним важливим підходом є нефункціональне тестування, спрямоване на перевірку аспектів системи, що не пов’язані напряму з її функціональністю, але значно впливають на користувацький досвід. Це включає тестування продуктивності, навантажувальне тестування, перевірку безпеки та тестування зручності використання.

Також важливим підходом є структурне або white-box тестування, коли тестувальник має доступ до вихідного коду й перевіряє внутрішню структуру та логіку роботи програмних компонентів, використовуючи модульні та інтеграційні тести. Протилежним до нього є black-box тестування, при якому перевіряється лише зовнішня поведінка системи без урахування внутрішньої структури. Корисним є також підхід наскрізного (end-to-end) тестування, що дозволяє перевірити роботу застосунку як єдиного цілого через типові сценарії взаємодії користувача із системою. Цей підхід є важливим для виявлення помилок, які не можуть бути знайдені при окремій перевірці модулів або компонентів. Важливим напрямом також є регресійне тестування, що виконується після внесення змін до коду, і має на меті перевірку того, що внесені зміни не призвели до появи нових помилок у вже протестованих функціональностях [31].

Ще одним важливим аспектом є автоматизоване тестування, що забезпечує швидке й регулярне виконання тестових сценаріїв, зменшує людський фактор, прискорює перевірку та підтримує якість у процесах безперервної інтеграції (CI/CD) [32]. Таким чином, для системи управління кінотеатром є доцільним застосування комплексного підходу, що охоплює функціональні й нефункціональні перевірки, модульні, інтеграційні та наскрізні тести, а також автоматизовані регресійні перевірки, що забезпечить високу якість, стабільність і надійність створеного програмного забезпечення.

Методи тестування використовуються для перевірки правильності роботи програмного забезпечення, виявлення потенційних помилок та оцінки загальної якості системи. Тестування є важливим етапом розробки, оскільки дозволяє переконатися у відповідності програмного продукту початковим вимогам і бізнес-задачам. В процесі тестування системи управління ресурсами кінотеатру було застосовано комплексний підхід, який включає ручне та автоматизоване тестування, що забезпечило високу надійність роботи застосунку.

Ручне тестування проводилося для перевірки коректності роботи інтерфейсу користувача та взаємодії окремих компонентів системи. Це дозволило своєчасно виявити помилки інтерфейсу, помилки логіки, а також недоліки в дизайні та зручності використання [33]. Особлива увага приділялася перевірці сценаріїв роботи користувачів із різними ролями (звичайний користувач і адміністратор), включаючи авторизацію, роботу зі списками сутностей, оформлення замовлень і редагування даних.

Автоматизоване тестування забезпечувало стабільність ключових функціональних модулів серверної частини системи. Для автоматизації було використано бібліотеку JUnit та Spring Test, що дозволило створити модульні й інтеграційні тести, які виконуються автоматично після кожної зміни в коді. Це суттєво прискорило процес виявлення та виправлення помилок, забезпечило підтримку високої якості програмного забезпечення на всіх етапах розробки.

Таким чином, обрані методи тестування дають змогу отримати надійну інформаційну систему, що відповідає заявленим вимогам і здатна ефективно працювати у реальних умовах експлуатації.

Підходи до тестування програмного забезпечення мають на меті забезпечити високий рівень якості та стабільності розробленої системи. У процесі тестування інформаційної системи управління ресурсами кінотеатру були застосовані різноманітні методологічні підходи, що дозволили комплексно оцінити її функціональність, продуктивність та відповідність початковим вимогам [34].

Спочатку використовувався функціональний підхід, що передбачає перевірку відповідності реалізованих функцій програмного продукту визначеним бізнес-вимогам. Цей підхід охоплює перевірку коректності роботи інтерфейсу користувача, взаємодію фронтенду та бекенду, точність обробки даних та виконання операцій, таких як створення, читання, оновлення та видалення інформації. Для цього були створені конкретні сценарії тестування, що відображають реальні випадки використання системи.

Наступним важливим етапом став підхід на основі інтеграційного тестування. Він забезпечує перевірку спільної роботи компонентів, таких як REST-контролери, сервіси, репозиторії, та брокери повідомлень. Під час інтеграційного тестування було перевірено, як окремі модулі взаємодіють один з одним, чи коректно виконуються запити до бази даних, а також чи правильно обробляються асинхронні події в системі (наприклад, через Kafka).

Для перевірки стійкості системи до високих навантажень та її здатності обробляти велику кількість одночасних запитів використовувався підхід до навантажувального тестування. В рамках цього тестування моделювалися ситуації активного використання застосунку багатьма користувачами одночасно, що дозволило оцінити продуктивність сервера, швидкість відповіді та стабільність роботи компонентів.

Ще одним ключовим підходом стало використання автоматизованих тестів на рівні модулів. Це дало змогу виявляти помилки ще на ранніх стадіях розробки

та забезпечити стабільність змін, що вносяться в код. Використання автоматизованих тестів дозволило швидко проводити регресивне тестування після кожного оновлення коду, гарантуючи, що нові функції не впливають негативно на вже реалізовану функціональність [35].

Окрім цього, особливу увагу було приділено безпеці системи, для чого застосовувався спеціалізований підхід до безпекового тестування. У цьому випадку перевірялась автентифікація, авторизація, управління сесіями, а також захист даних при передачі та зберіганні. Зокрема, тестувалася робота з JWT-токенами, а також забезпечення належного рівня захисту конфіденційної інформації користувачів.

Завдяки комплексному застосуванню цих підходів тестування можливо забезпечити високу якість, продуктивність і надійність системи управління ресурсами кінотеатру. Такий підхід гарантує стабільну та безпечну експлуатацію розробленого програмного продукту, дозволяючи своєчасно виявляти та усувати потенційні проблеми.

4.2 Тестування функціональності

Тестування функціональності є важливим етапом перевірки розробленого веб-застосунку для управління ресурсами кінотеатру, оскільки саме функціональні характеристики визначають, наскільки система відповідає початковим вимогам та потребам користувачів. Основною метою функціонального тестування є перевірка коректності роботи реалізованих компонентів і процесів застосунку з точки зору кінцевого користувача. В процесі тестування функціональності проводиться детальна перевірка реалізації ключових операцій: реєстрації та автентифікації користувачів, бронювання квитків на фільми, перегляду та редагування репертуару, формування звітів про продажі, ведення обліку товарів та адміністрування ресурсів. Перевірка проводиться шляхом виконання типових сценаріїв взаємодії із системою, таких як авторизація користувача, вибір фільму, часу сеансу, оформлення бронювання

квитків, оплата замовлення, а також адміністративні операції з керування даними. Особливу увагу приділено перевірці коректності введення даних та адекватності реакції системи на помилкові або некоректні дані, зокрема при спробах бронювання вже зайнятих місць або введенні недійсних реквізитів при оплаті замовлення.

Юніт-тестування є одним із ключових підходів до перевірки працездатності окремих компонентів програмного забезпечення. Його основна ідея полягає в тому, щоб протестувати кожен компонент програми (наприклад, класи, методи чи функції) ізольовано від інших елементів. Головна мета таких тестів — переконатися у правильності реалізації логіки на найнижчому рівні, виявити помилки якомога раніше та мінімізувати ризики виникнення складних дефектів у майбутньому.

Під час розробки інформаційної системи для управління ресурсами кінотеатру юніт-тестування застосовувалося для перевірки окремих методів бізнес-логіки, роботи сервісних класів, обчислювальних алгоритмів та маперів, які перетворюють сутності на DTO-об’єкти. Для реалізації таких тестів використовувався фреймворк JUnit — популярний інструмент у Java-розробці, який дозволяє легко писати та запускати юніт-тести. Також активно використовувалися бібліотеки для створення «заглушок» і мок-об’єктів, наприклад Mockito, які дозволяють імітувати поведінку залежностей, що не входять у тестований клас.

При написанні юніт-тестів було охоплено всі ключові сутності та сервіси: наприклад, методи, що здійснюють розрахунок вартості квитків, бронювання місць, обробку замовлень, логіку обліку снеків, перевірку автентифікації користувачів, тощо. Використовуючи JUnit, вдалося швидко перевірити правильність роботи цих елементів та переконатися в тому, що вони виконують поставлені задачі.

Результати тестування серверної частини системи управління ресурсами кінотеатру зображено на рисунку 4.1.

Рисунок 4.1 – Результати тестування серверної частини системи управління ресурсами кінотеатру

Застосування модульного тестування дало змогу перевірити роботу окремих складових програмної системи, зокрема сервісів, відповідальних за бронювання квитків, розрахунок їх вартості, контроль запасів снеків та автентифікацію користувачів. Отримані результати дозволили підтвердити правильність реалізації бізнес-логіки та виявити потенційні дефекти ще під час процесу розробки.

Під час інтеграційного тестування проводилася перевірка взаємодії REST API з базою даних PostgreSQL, зовнішніми асинхронними сервісами Kafka та механізмами автентифікації на основі JWT-токенів. Використання Spring Test дало можливість переконатися в ефективній взаємодії окремих компонентів, своєчасно виявити можливі проблеми транзакцій та перевірити стабільність системи за умов підвищеного навантаження.

Наскрізне тестування із застосуванням інструменту Cypress включало основні сценарії взаємодії користувачів із системою: вибір і резервування місць у залі, здійснення оплат, керування ресурсами кінотеатру та адміністрування персоналу. Проведені тести засвідчили стабільну роботу веб-інтерфейсу,

коректну перевірку введених користувачем даних, зручність взаємодії з застосунком та повну відповідність висунутим вимогам.

Крім того, було здійснено перевірку серверної частини веб-застосунку, розробленого із застосуванням JHipster, Spring Boot та React. Тестування передбачало генерацію технічної документації за допомогою Javadoc, аналіз відповідності стилю коду через Checkstyle, виконання модульних та інтеграційних тестів з використанням JUnit, а також оцінку покриття коду тестами через інструмент JaCoCo.

У ході роботи було виконано аналіз відповідності проєкту вимогам до документації, що підтвердилось успішним генеруванням технічної документації за допомогою інструменту Javadoc. Аналіз стилістики написання коду за допомогою засобу Checkstyle продемонстрував повну відповідність стандартам оформлення Java-коду, що свідчить про добре організовану структуру та зрозумілість програмного забезпечення.

Було також проведено модульні тести, які перевіряли функціональність основних бекенд-компонентів системи, включаючи сутності, DTO-об’єкти та механізми перетворення даних на базі MapStruct. Загальна кількість виконаних модульних тестів становить 87, і всі вони завершились успішно без помилок чи критичних зауважень, що підтвердило правильність реалізації ключових сутностей: фільмів, залів, квитків, місць та персоналу.

Інтеграційне тестування охопило перевірку взаємодії REST API із базою даних, системою асинхронного обміну повідомленнями Kafka, а також компонентами автентифікації на основі JWT-токенів. Загалом було виконано 305 інтеграційних тестів, які завершилися без помилок, що підтвердило надійність та стабільність взаємодії між різними компонентами серверної частини застосунку.

Під час тестування були виявлені незначні зауваження від MapStruct щодо неповних перетворень окремих полів даних. Ці зауваження не є критичними для функціонування системи, проте можуть бути використані як рекомендації для подальшого вдосконалення програмного забезпечення.

Комплексні результати тестування серверної частини, які наведені на рисунку 4.1, підтвердили високу якість розробленого веб-застосунку та його готовність до подальшого використання у виробничому середовищі.

Додатково було проведено комплексне оцінювання покриття програмного коду автоматизованими тестами з використанням інструментів Maven та JUnit. Це включало інформаційні, стилістичні, модульні, документаційні та інтеграційні перевірки. Підсумкові результати покриття тестами коду представлені на рисунку 4.2.

Coverage: management in cinema-city-management

Element	Class, %	Method, %	Line, %	Branch, %
com.cinemacity.management	95% (99/104)	94% (854/901)	91% (2355/2566)	76% (481/630)
aop.logging	0% (0/1)	0% (0/6)	0% (0/21)	0% (0/10)
broker	100% (2/2)	100% (6/6)	94% (16/17)	100% (0/0)
config	82% (14/17)	81% (52/64)	79% (205/258)	56% (37/66)
domain	100% (16/16)	100% (278/278)	100% (499/499)	86% (92/106)
management	100% (1/1)	100% (6/6)	100% (13/13)	100% (0/0)
repository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
security	100% (4/4)	93% (15/16)	94% (47/50)	75% (12/16)
service	100% (31/31)	94% (319/336)	93% (834/890)	73% (88/120)
web	100% (30/30)	95% (175/183)	92% (733/789)	82% (250/302)
ApplicationWebXml	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
CinemaCityManagementApp	100% (1/1)	60% (3/5)	29% (8/27)	20% (2/10)
GeneratedByJHipster	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)

Рисунок 4.2 – Результати покриття тестами серверної частини CI/CD системи управління ресурсами кінотеатру

Згідно з результатами, усі виконані перевірки завершились успішно, без жодних помилок чи виняткових ситуацій. Всього було здійснено 305 тестів, що дозволяє констатувати стабільність реалізованої логіки, правильну роботу взаємодії між сутностями та контролерами REST API. Було ретельно протестовано класи, що реалізують ключові бізнес-функції системи: управління квитками, фільмами, розкладами, продажами, клієнтами та запасами, включаючи відповідні сервіси, мапери та DTO.

Перевірка рівня покриття тестами була здійснена за допомогою модуля JaCoCo. Показники охоплення склали 81% за рядками коду, 88,9% за методами та 56,1% за умовними розгалуженнями. Високі показники покриття, що

перевищують 90%, були досягнуті для пакетів web.rest, service, domain та security, що свідчить про повноту автоматизованого тестування основних елементів системи.

Проте, деякі пакети, зокрема aop.logging та service.api.dto, наразі не мають повного тестового покриття, що надає можливість для подальшого покращення якості програмного забезпечення через реалізацію додаткових тестів.

Отже, застосування комплексного підходу до тестування підтвердило високу ефективність і якість створеного веб-застосунку, реалізованого з використанням платформи JHipster, та забезпечило надійність системи для подальшого використання в умовах реальної експлуатації кінотеатру. В результаті загальне покриття коду становило понад 80%, що свідчить про високу надійність розроблених компонентів.

Таким чином, юніт-тестування допомогло швидко виявити та усунути дефекти на ранніх стадіях розробки, забезпечило високу якість коду та сприяло створенню надійного й стабільного програмного забезпечення для управління ресурсами кінотеатру.

4.3 Оцінка ефективності використання інформаційної системи

Зробимо детальний аналіз ефективності використання системи та розглянемо рекомендації щодо подальшого розвитку, враховуючи ресурсні витрати та очікуваний ефект.

Для впровадження системи будуть задіяні такі ресурси:

- Вартість базової ліцензії системи складає близько 300 000 гривень на рік. Це включає доступ до основних модулів (управління фінансами, продажами, інвентарем та персоналом).

- Вартість інтеграційних робіт оцінюється у 150 000 гривень. Це включає налаштування інтеграції з системою продажу квитків Ticketscloud та платіжною системою.

- Проведення тренінгів для персоналу кінотеатру склало близько 60 000 гривень. Включало навчання роботи з с, а також з модулями управління фінансами, інвентарем та розкладом показів.

- Щорічні витрати на технічну підтримку та обслуговування системи оцінюються у 90 000 гривень. Це включає оновлення системи та усунення можливих помилок.

Загальні витрати на впровадження та підтримку системи склали близько 600000 гривень за перший рік.

Система сприятиме покращенню управління розкладом показів, оптимізації маркетингових активностей та більш ефективному використанню ресурсів, що сприяло збільшенню відвідуваності. В таблиці 4.1 представимо основні витрати.

Таблиця 4.1 - Витрати та ефективності системи

Витрати/Показники	Сума (грн)	Очікуваний ефект
Ліцензія системи (річна)	300 000	Автоматизація управління бізнес-процесами
Інтеграція з існуючими системами	150 000	Оптимізація обліку продажів та фінансових операцій
Навчання персоналу	60 000	Збільшення ефективності роботи персоналу
Технічна підтримка та обслуговування	90 000	Підтримка безперебійної роботи системи
Загальні витрати за перший рік	600 000	-

Враховуючи зростання чистого прибутку, система окупилася протягом приблизно 2 років, оскільки додатковий прибуток поступово покриває витрати на впровадження.

З урахуванням попередніх результатів від впровадження системи в кінотеатрі «Світ Сінема», можна зробити прогноз на 2024 та 2025 роки, беручи до уваги динаміку зростання доходів та прибутковості, а також додаткові ефекти, досягнуті завдяки автоматизації бізнес-процесів.

4.4 Висновки

Проведене тестування інформаційної системи управління ресурсами кінотеатру підтвердило високу якість та стабільність розробленого веб-застосунку. Використання модульних, інтеграційних і наскрізних тестів дозволило детально оцінити працездатність окремих компонентів, взаємодію між ними, а також роботу системи з точки зору кінцевого користувача.

Юніт-тести забезпечили швидке виявлення помилок на етапі розробки, а інтеграційні перевірки REST API з базою даних PostgreSQL і сервісами Kafka підтвердили надійність взаємодії компонентів. Наскрізне тестування за допомогою Cypress довело зручність і стабільність роботи інтерфейсу. Автоматизація процесу тестування через Maven і JUnit із використанням JaCoCo та Checkstyle показала високий рівень відповідності коду стандартам та досягла хорошого показника покриття (81% за рядками).

Виявлені зони для покращення (неповні мапінги MapStruct і недостатнє покриття окремих пакетів) відкривають перспективи для подальшого розвитку якості системи.

Таким чином, комплексний підхід до тестування забезпечив створення надійного рішення, готового до впровадження у практичну експлуатацію.

ВИСНОВКИ

У бакалаврській кваліфікаційній роботі проведено комплексне дослідження і розробку інформаційної системи управління ресурсами кінотеатру. Виконано аналіз існуючих програмних рішень, визначено їх переваги й недоліки, що підтвердило необхідність створення адаптивної та гнучкої системи, здатної враховувати особливості українського ринку.

Запропонована система побудована з використанням сучасних технологій: серверна частина реалізована за допомогою фреймворку Spring Boot, а клієнтська – із застосуванням React TypeScript. Це дозволило забезпечити високу продуктивність, легкість масштабування та адаптації під конкретні потреби кінотеатрів. Інтеграція з Apache Kafka дала змогу організувати обробку подій у реальному часі, а використання PostgreSQL забезпечило надійність зберігання та швидкий доступ до даних.

Було створено детальні UML-діаграми, які формалізують основні бізнес-процеси та логіку роботи системи, що дозволило чітко визначити поведінкові й структурні аспекти роботи програмного комплексу.

Розроблене програмне забезпечення успішно пройшло тестування, яке підтвердило його відповідність усім функціональним вимогам, високий рівень безпеки та зручність використання. Додатково складено інструкцію користувача, яка містить докладні рекомендації щодо використання системи.

Практичне значення виконаної роботи полягає в тому, що розроблене рішення може бути впроваджено як у невеликих незалежних кінотеатрах, так і в великих мережевих структурах, сприяючи підвищенню ефективності операційних процесів, зменшенню витрат на управління ресурсами та покращенню сервісу для клієнтів.

Отримані результати рекомендується використовувати для подальшого розвитку інформаційних технологій у кіноіндустрії, а також як основу для подальшого вдосконалення та інтеграції із зовнішніми сервісами і платформами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Левицька Т.Л., Сидоренко О.В. Розвиток індустрії кінотеатрів в Україні: сучасний стан та перспективи. Економіка і організація управління. 2021. № 1(41). С. 84-92.

2. Петров В.І. Цифрова трансформація культурної індустрії України. Київ: Наукова думка, 2023. 256 с.

3. Гавриленко А.В., Шевченко М.П. Методології управління ресурсами в сфері розваг. Вісник ВНТУ. 2022. № 3. С. 25-32.

4. Fandango Media, LLC: офіційний сайт компанії. URL: https://www.fandango.com/ (дата звернення: 10.04.2025).

5. Multiplex Holding: система управління мережею кінотеатрів. URL: https://multiplex.ua/ (дата звернення: 10.04.2025).

6. KyivKinoFilm: офіційний сайт. URL: https://kievkino.com.ua/ (дата звернення: 10.04.2025).

7. Ковальчук О.М. Порівняльний аналіз систем управління ресурсами кінотеатрів. Інформаційні технології та комп'ютерна інженерія. 2023. №2. С. 45-53.

8. Шевчук І.Б., Васьків О.М. Інноваційні технології в індустрії розваг: проблеми впровадження та перспективи розвитку. Ефективна економіка. 2022. № 11. URL: http://www.economy.nayka.com.ua/?op=1&z=10642 (дата звернення: 12.04.2025).

9. Бородіна О. А., Коваленко І. В. Автоматизовані системи управління ресурсами підприємств індустрії розваг. Економіка і менеджмент культури. 2022. №2. С. 45-52.

10. Ticketscloud: офіційна документація системи продажу квитків. URL: https://ticketscloud.com/docs/ (дата звернення: 15.04.2025).

11. McAfee Total Protection: комплексний захист інформаційних систем. URL: https://www.mcafee.com/consumer/uk-ua/total-protection.html (дата звернення: 18.04.2025).

12. Сидоренко М. Я., Петренко В. О. Економічна ефективність впровадження інформаційних систем управління ресурсами в кінотеатрах України. Національна економіка України: теорія та практика управління. 2023. Вип. 47. С. 113-124.

13. Fowler M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2003. 533 p.

14. Richardson L., Ruby S. RESTful Web Services. O'Reilly Media, 2007. 454p.

15. Мейер Б. Об'єктно-орієнтоване програмування. Київ: Фенікс, 2019. 672 с.

16. JHipster: офіційна документація. URL: https://www.jhipster.tech/ (дата звернення: 15.04.2025).

17. Walls C. Spring Boot in Action. Manning Publications, 2022. 592 p.

18. Banks A., Porcello E. Learning React: Functional Web Development with React and Redux. O'Reilly Media, 2020. 350 p.

19. Вігерс К., Бітті Д. Розробка вимог до програмного забезпечення. 3-є вид. Київ: Видавнича група BHV, 2019. 736 с.

20. Розробка структури застосунку для управління ресурсами кінотеатру із використанням JHipster. /В.В. Слободиський, І.В. Богач // Матеріали LIV науково-технічної конференції підрозділів Вінницького національного технічного університету (НТКП ВНТУ-2025). Збірник доповідей [Електронний ресурс], Вінниця : ВНТУ, 2025. Режим доступу: https://conferences.vntu.edu.ua/index.php/all-fksa/all-fksa-2025/paper/view/24478

21. Apache Maven: офіційна документація. URL: https://maven.apache.org/guides/ (дата звернення: 18.04.2025).

22. PostgreSQL: The World's Most Advanced Open Source Relational Database. URL: https://www.postgresql.org/docs/ (дата звернення: 18.04.2025).

23. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT): офіційна специфікація. URL: https://tools.ietf.org/html/rfc7519 (дата звернення: 20.04.2025).

24. Bauer C., King G. Java Persistence with Hibernate. Manning Publications, 2015. 608 p.

25. Narkhede N., Shapira G., Palino T. Kafka: The Definitive Guide. O'Reilly Media, 2017. 322 p.

26. Fink G., Flatow I. Pro Single Page Application Development: Using Backbone.js and ASP.NET. Apress, 2014. 372 p.

27. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008. 464 p.

28. Swagger Documentation: офіційна документація. URL: https://swagger.io/docs/ (дата звернення: 22.04.2025).

29. Куліков С. С. Тестування програмного забезпечення. Базовий курс. Мінськ: Чотири чверті, 2017. 312 с.

30. JUnit 5: офіційна документація. URL: https://junit.org/junit5/docs/current/user-guide/ (дата звернення: 24.05.2025).

31. Crispin L., Gregory J. Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley Professional, 2009. 576 p.

32. Selenium: офіційна документація. URL: https://www.selenium.dev/documentation/ (дата звернення: 25.04.2025).

33. Molyneaux I. The Art of Application Performance Testing. O'Reilly Media, 2014. 278 p.

34. OWASP: The Open Web Application Security Project. URL: https://owasp.org/ (дата звернення: 25.04.2025).

35. Коберн А. Сучасні методи опису функціональних вимог до систем. Київ: Лорі, 2018. 264 с.

36. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. URL: https://www.iso.org/standard/35733.html (дата звернення: 25.04.2025).

37. Тестування web-застосунків із використанням JHipster на прикладі системи для управління ресурсами кінотеатру. /Богач І.В., Слободиський В.В. // Молодь в науці: дослідження, проблеми, перспективи (МН-2025), Вінниця,

2025. [Електронний ресурс]. URL: https://conferences.vntu.edu.ua/index.php/mn/mn2025/paper/view/25744.

38. Методичні вказівки до виконання бакалаврських кваліфікаційних робіт (проєктів) для студентів спеціальностей: 126 «Інформаційні системи та технології», 151 «Автоматизація та комп'ютерно-інтегровані технології» [Електронний ресурс] / Уклад. К. В. Овчинников, О. В. Бісікало. – Вінниця : ВНТУ, 2022. – 50 с.

ДОДАТКИ

Додаток А (обов’язковий)

ІЛЮСТРАТИВНА ЧАСТИНА

ІНФОРМАЦІЙНА СИСТЕМА

УПРАВЛІННЯ РЕСУРСАМИ КІНОТЕАТРУ

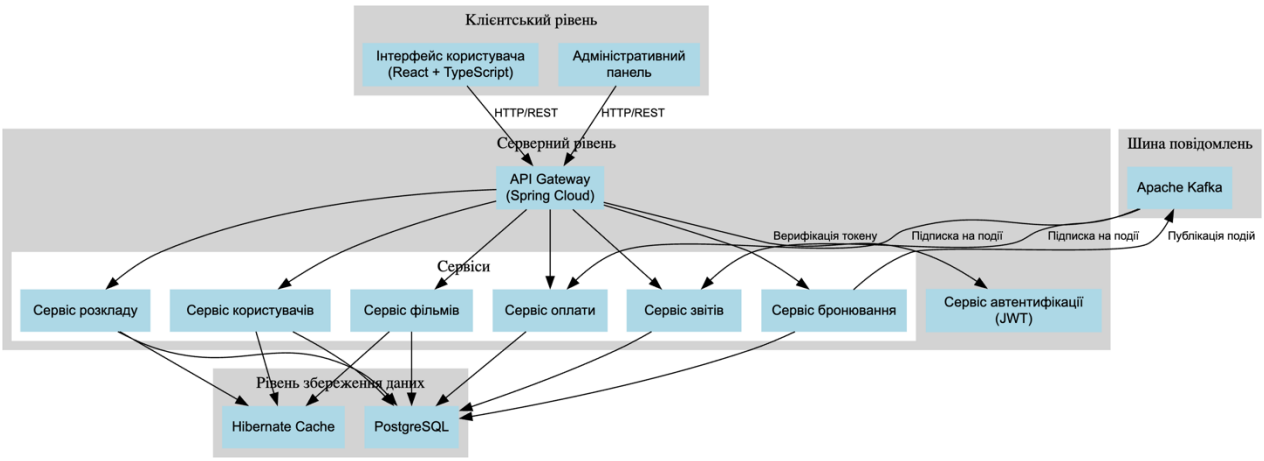

Рисунок А.1 – Архітектура системи управління ресурсами кінотеатру

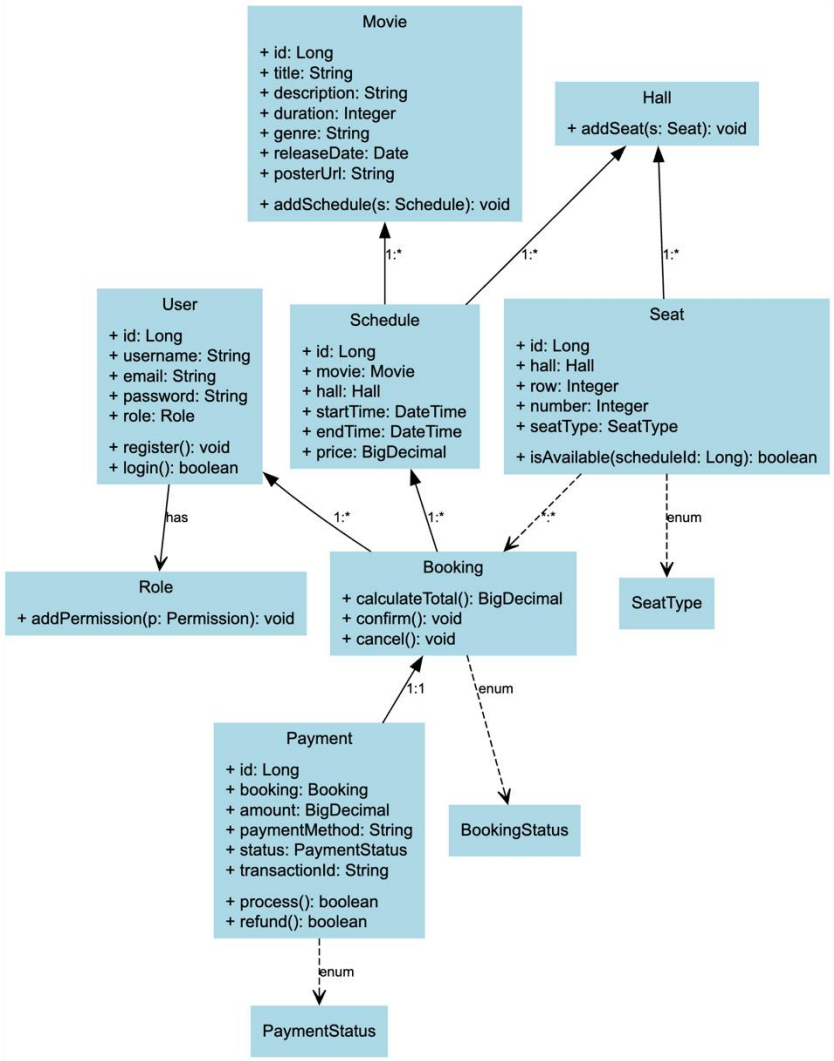

Рисунок А.2 – Діаграма класів системи

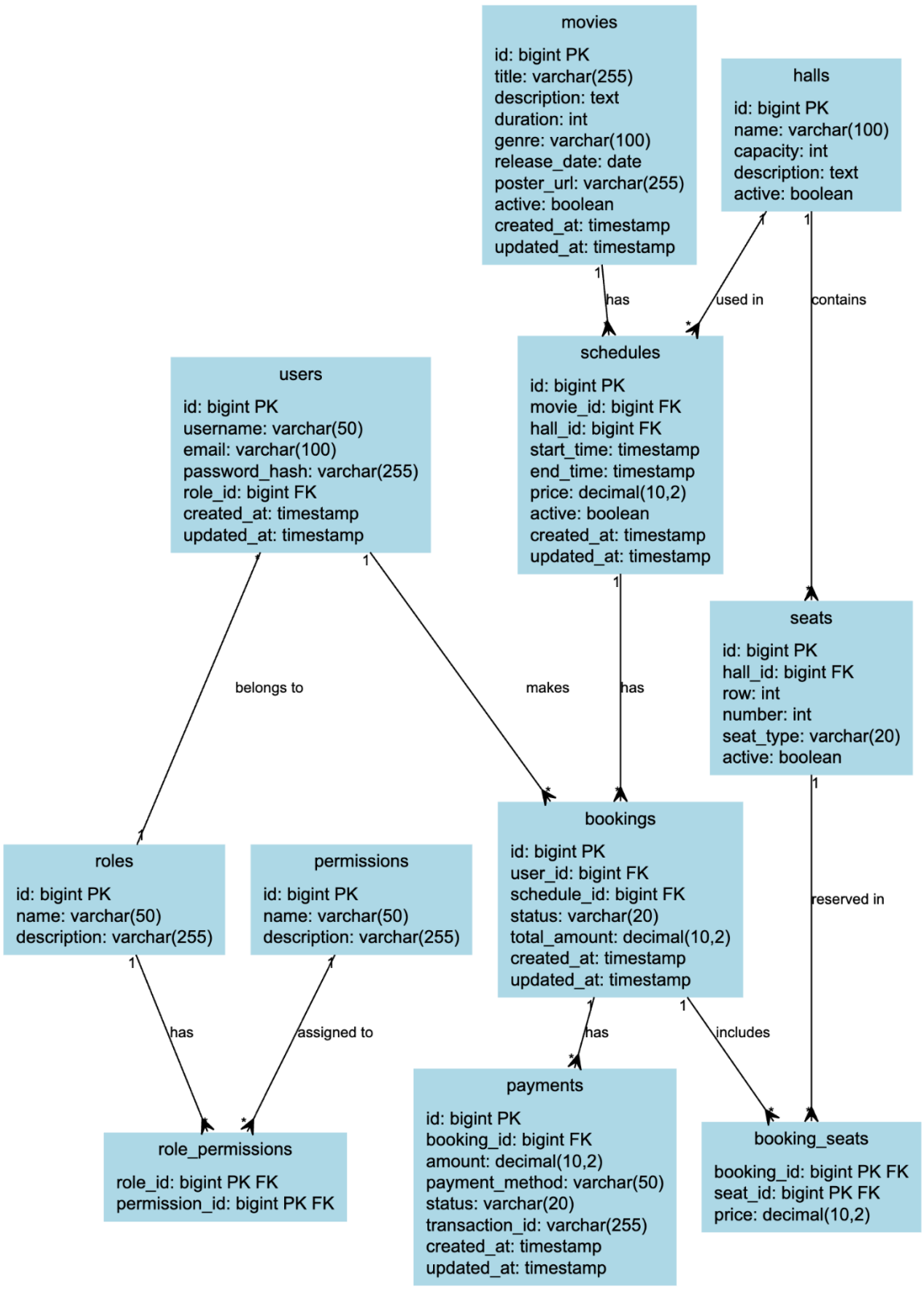

Рисунок А.3 – Структура бази даних (ER-діаграма)

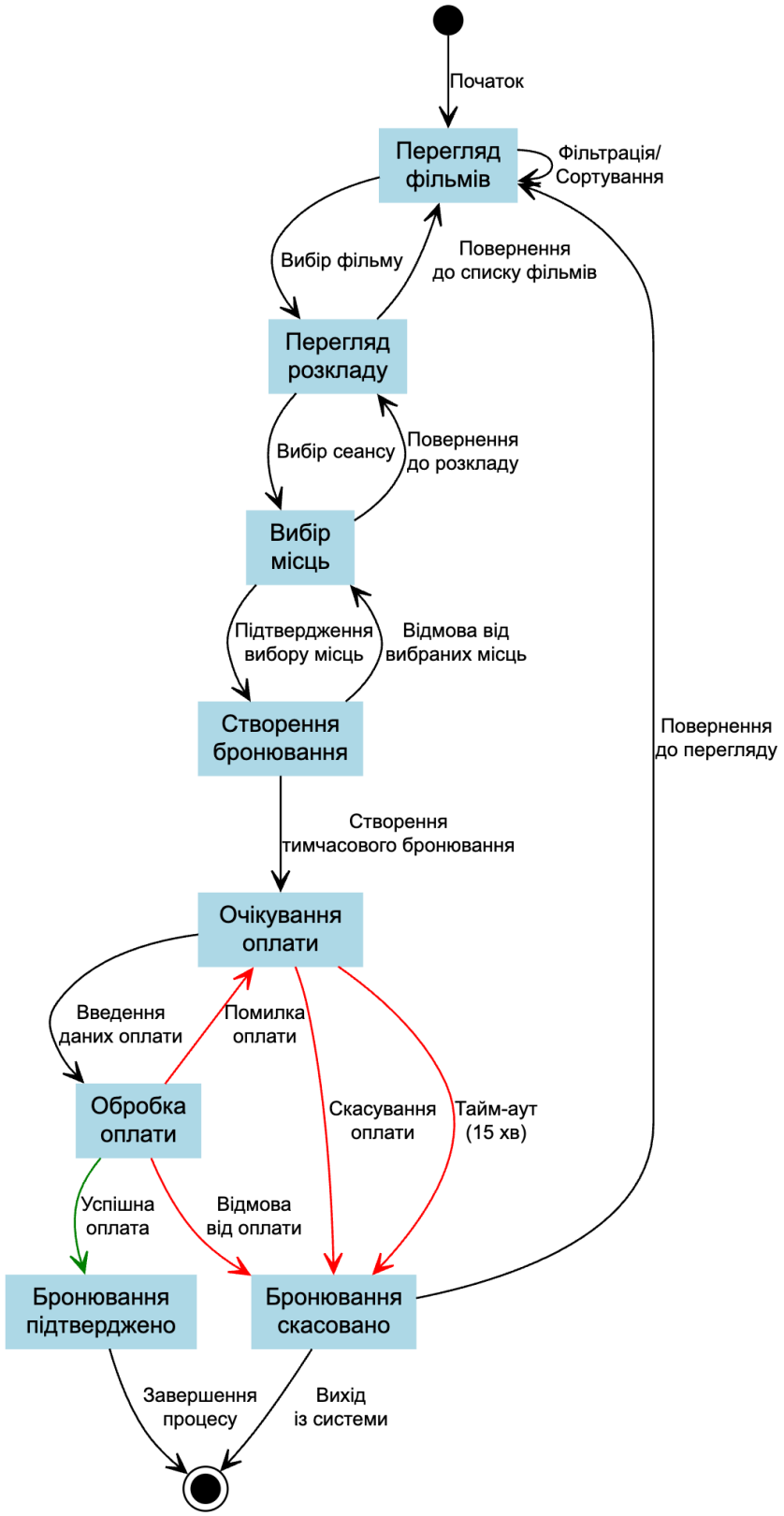

Рисунок А.4 – Діаграма стану для процесу бронювання

Рисунок А.5 – Діаграма розгортання системи

Додаток Б (обов'язковий)

Лістинг основних функцій

```
application {
  config {
    jhipsterVersion "8.0.0"
    baseName cinemaCityManagement
    applicationType monolith
    buildTool maven
    packageName com.cinemacity.management
    authenticationType jwt
    databaseType sql
    devDatabaseType h2Disk
    prodDatabaseType postgresql
    enableHibernateCache true
    enableTranslation true
    clientPackageManager npm
    languages [en, ua]
    nativeLanguage ua
    dtoSuffix DTO
    enableSwaggerCodegen true
    messageBroker kafka
    reactive false
    serviceDiscoveryType no
    skipClient false
    skipServer false
    skipUserManagement false
    websocket false
    serverPort 8080
    testFrameworks [junit]
  }
  entities *
}

entity Movie {
  title String required
  description TextBlob
  duration Integer required
  releaseDate LocalDate
  rating Double
}

entity Hall {
  name String required
  capacity Integer required
}

entity Seat {
  number String required
  row String required
  status SeatStatus
}

enum SeatStatus {
  AVAILABLE,
  RESERVED,
  BROKEN
}

entity Screening {
  dateTime ZonedDateTime required
  price BigDecimal required
}

entity Ticket {
  purchaseDate ZonedDateTime required
  price BigDecimal required
  status TicketStatus
}

enum TicketStatus {
  PURCHASED,
  CANCELLED,
  USED
}
```

```
entity Snack {
  name String required
  price BigDecimal required
  quantity Integer required
}

entity Inventory {
  productName String required
  quantity Integer required
  reorderLevel Integer
}

entity Employee {
  firstName String required
  lastName String required
  position String required
  employmentDate LocalDate
}

entity Schedule {
  workDate LocalDate required
  startTime LocalTime required
  endTime LocalTime required
}

entity Sale {
  saleDate ZonedDateTime required
  totalAmount BigDecimal required
}

entity Customer {
  firstName String
  lastName String
  email String required
  phone String
  loyaltyPoints Integer
}

relationship OneToMany {
  Hall{seats} to Seat{hall}
  Hall{screenings} to Screening{hall}
  Movie{screenings} to Screening{movie}
  Screening{tickets} to Ticket{screening}
  Customer{tickets} to Ticket{customer}
  Sale{snacks} to Snack{sale}
  Employee{schedules} to Schedule{employee}
}

dto * with mapstruct
service * with serviceClass
paginate * with pagination

package com.cinemacity.management;

import com.cinemacity.management.config.ApplicationProperties;
import com.cinemacity.management.config.CRLFLogConverter;
import jakarta.annotation.PostConstruct;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Arrays;
import java.util.Collection;
import java.util.Optional;
import org.apache.commons.lang3.StringUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration;
import org.springframework.boot.autoconfigure.liquibase.LiquibaseProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.core.env.Environment;
import tech.jhipster.config.DefaultProfileUtil;
import tech.jhipster.config.JHipsterConstants;

@SpringBootApplication(exclude = { H2ConsoleAutoConfiguration.class })
@EnableConfigurationProperties({ LiquibaseProperties.class, ApplicationProperties.class })
public class CinemaCityManagementApp {

    private static final Logger LOG = LoggerFactory.getLogger(CinemaCityManagementApp.class);
```

```
    private final Environment env;

    public CinemaCityManagementApp(Environment env) {
        this.env = env;
    }

    /**
     * Initializes cinemaCityManagement.
     * <p>
     * Spring profiles can be configured with a program argument --spring.profiles.active=your-
active-profile
     */
    @PostConstruct
    public void initApplication() {
        Collection<String> activeProfiles = Arrays.asList(env.getActiveProfiles());
        if (
            activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_DEVELOPMENT) &&
            activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_PRODUCTION)
        ) {
            LOG.error(
                "You have misconfigured your application! It should not run " + "with both the 'dev'
and 'prod' profiles at the same time."
            );
        }
        if (
            activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_DEVELOPMENT) &&
            activeProfiles.contains(JHipsterConstants.SPRING_PROFILE_CLOUD)
        ) {
            LOG.error(
                "You have misconfigured your application! It should not " + "run with both the 'dev'
and 'cloud' profiles at the same time."
            );
        }
    }

    /**
     * Main method, used to run the application.
     *
     * @param args the command line arguments.
     */
    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(CinemaCityManagementApp.class);
        DefaultProfileUtil.addDefaultProfile(app);
        Environment env = app.run(args).getEnvironment();
        logApplicationStartup(env);
    }

    private static void logApplicationStartup(Environment env) {
        String protocol = Optional.ofNullable(env.getProperty("server.ssl.key-store")).map(key ->
"https").orElse("http");
        String applicationName = env.getProperty("spring.application.name");
        String serverPort = env.getProperty("server.port");
        String contextPath = Optional.ofNullable(env.getProperty("server.servlet.context-path"))
            .filter(StringUtils::isNotBlank)
            .orElse("/");
        String hostAddress = "localhost";
        try {
            hostAddress = InetAddress.getLocalHost().getHostAddress();
        } catch (UnknownHostException e) {
            LOG.warn("The host name could not be determined, using `localhost` as fallback");
        }
        LOG.info(
            CRLFLogConverter.CRLF_SAFE_MARKER,
            """

            ----------------------------------------------------------
            \tApplication '{}' is running! Access URLs:
            \tLocal: \t\t{}://localhost:{}{}
            \tExternal: \t{}://{}:{}{}
            \tProfile(s): \t{}
            ----------------------------------------------------------""",
            applicationName,
            protocol,
            serverPort,
            contextPath,
            protocol,
            hostAddress,
            serverPort,
            contextPath,
```

```
            env.getActiveProfiles().length == 0 ? env.getDefaultProfiles() : env.getActiveProfiles()
        );
    }
}
```

```
package com.cinemacity.management.service;

import com.cinemacity.management.config.Constants;
import com.cinemacity.management.domain.Authority;
import com.cinemacity.management.domain.User;
import com.cinemacity.management.repository.AuthorityRepository;
import com.cinemacity.management.repository.UserRepository;
import com.cinemacity.management.security.AuthoritiesConstants;
import com.cinemacity.management.security.SecurityUtils;
import com.cinemacity.management.service.dto.AdminUserDTO;
import com.cinemacity.management.service.dto.UserDTO;
import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.*;
import java.util.stream.Collectors;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cache.CacheManager;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import tech.jhipster.security.RandomUtil;

/**
 * Service class for managing users.
 */
@Service
@Transactional
public class UserService {

    private static final Logger LOG = LoggerFactory.getLogger(UserService.class);

    private final UserRepository userRepository;

    private final PasswordEncoder passwordEncoder;

    private final AuthorityRepository authorityRepository;

    private final CacheManager cacheManager;

    public UserService(
        UserRepository userRepository,
        PasswordEncoder passwordEncoder,
        AuthorityRepository authorityRepository,
        CacheManager cacheManager
    ) {
        this.userRepository = userRepository;
        this.passwordEncoder = passwordEncoder;
        this.authorityRepository = authorityRepository;
        this.cacheManager = cacheManager;
    }

    public Optional<User> activateRegistration(String key) {
        LOG.debug("Activating user for activation key {}", key);
        return userRepository
            .findOneByActivationKey(key)
            .map(user -> {
                // activate given user for the registration key.
                user.setActivated(true);
                user.setActivationKey(null);
                this.clearUserCaches(user);
                LOG.debug("Activated user: {}", user);
                return user;
            });
    }

    public Optional<User> completePasswordReset(String newPassword, String key) {
        LOG.debug("Reset user password for reset key {}", key);
        return userRepository
            .findOneByResetKey(key)
            .filter(user -> user.getResetDate().isAfter(Instant.now().minus(1, ChronoUnit.DAYS)))
            .map(user -> {
```

```
            user.setPassword(passwordEncoder.encode(newPassword));
            user.setResetKey(null);
            user.setResetDate(null);
            this.clearUserCaches(user);
            return user;
        });
}

public Optional<User> requestPasswordReset(String mail) {
    return userRepository
        .findOneByEmailIgnoreCase(mail)
        .filter(User::isActivated)
        .map(user -> {
            user.setResetKey(RandomUtil.generateResetKey());
            user.setResetDate(Instant.now());
            this.clearUserCaches(user);
            return user;
        });
}

public User registerUser(AdminUserDTO userDTO, String password) {
    userRepository
        .findOneByLogin(userDTO.getLogin().toLowerCase())
        .ifPresent(existingUser -> {
            boolean removed = removeNonActivatedUser(existingUser);
            if (!removed) {
                throw new UsernameAlreadyUsedException();
            }
        });
    userRepository
        .findOneByEmailIgnoreCase(userDTO.getEmail())
        .ifPresent(existingUser -> {
            boolean removed = removeNonActivatedUser(existingUser);
            if (!removed) {
                throw new EmailAlreadyUsedException();
            }
        });
    User newUser = new User();
    String encryptedPassword = passwordEncoder.encode(password);
    newUser.setLogin(userDTO.getLogin().toLowerCase());
    // new user gets initially a generated password
    newUser.setPassword(encryptedPassword);
    newUser.setFirstName(userDTO.getFirstName());
    newUser.setLastName(userDTO.getLastName());
    if (userDTO.getEmail() != null) {
        newUser.setEmail(userDTO.getEmail().toLowerCase());
    }
    newUser.setImageUrl(userDTO.getImageUrl());
    newUser.setLangKey(userDTO.getLangKey());
    // new user is not active
    newUser.setActivated(false);
    // new user gets registration key
    newUser.setActivationKey(RandomUtil.generateActivationKey());
    Set<Authority> authorities = new HashSet<>();
    authorityRepository.findById(AuthoritiesConstants.USER).ifPresent(authorities::add);
    newUser.setAuthorities(authorities);
    userRepository.save(newUser);
    this.clearUserCaches(newUser);
    LOG.debug("Created Information for User: {}", newUser);
    return newUser;
}

private boolean removeNonActivatedUser(User existingUser) {
    if (existingUser.isActivated()) {
        return false;
    }
    userRepository.delete(existingUser);
    userRepository.flush();
    this.clearUserCaches(existingUser);
    return true;
}

public User createUser(AdminUserDTO userDTO) {
    User user = new User();
    user.setLogin(userDTO.getLogin().toLowerCase());
    user.setFirstName(userDTO.getFirstName());
    user.setLastName(userDTO.getLastName());
    if (userDTO.getEmail() != null) {
        user.setEmail(userDTO.getEmail().toLowerCase());
    }
```

```
        user.setImageUrl(userDTO.getImageUrl());
        if (userDTO.getLangKey() == null) {
            user.setLangKey(Constants.DEFAULT_LANGUAGE); // default language
        } else {
            user.setLangKey(userDTO.getLangKey());
        }
        String encryptedPassword = passwordEncoder.encode(RandomUtil.generatePassword());
        user.setPassword(encryptedPassword);
        user.setResetKey(RandomUtil.generateResetKey());
        user.setResetDate(Instant.now());
        user.setActivated(true);
        if (userDTO.getAuthorities() != null) {
            Set<Authority> authorities = userDTO
                .getAuthorities()
                .stream()
                .map(authorityRepository::findById)
                .filter(Optional::isPresent)
                .map(Optional::get)
                .collect(Collectors.toSet());
            user.setAuthorities(authorities);
        }
        userRepository.save(user);
        this.clearUserCaches(user);
        LOG.debug("Created Information for User: {}", user);
        return user;
    }

    /**
     * Update all information for a specific user, and return the modified user.
     *
     * @param userDTO user to update.
     * @return updated user.
     */
    public Optional<AdminUserDTO> updateUser(AdminUserDTO userDTO) {
        return Optional.of(userRepository.findById(userDTO.getId()))
            .filter(Optional::isPresent)
            .map(Optional::get)
            .map(user -> {
                this.clearUserCaches(user);
                user.setLogin(userDTO.getLogin().toLowerCase());
                user.setFirstName(userDTO.getFirstName());
                user.setLastName(userDTO.getLastName());
                if (userDTO.getEmail() != null) {
                    user.setEmail(userDTO.getEmail().toLowerCase());
                }
                user.setImageUrl(userDTO.getImageUrl());
                user.setActivated(userDTO.isActivated());
                user.setLangKey(userDTO.getLangKey());
                Set<Authority> managedAuthorities = user.getAuthorities();
                managedAuthorities.clear();
                userDTO
                    .getAuthorities()
                    .stream()
                    .map(authorityRepository::findById)
                    .filter(Optional::isPresent)
                    .map(Optional::get)
                    .forEach(managedAuthorities::add);
                userRepository.save(user);
                this.clearUserCaches(user);
                LOG.debug("Changed Information for User: {}", user);
                return user;
            })
            .map(AdminUserDTO::new);
    }

    public void deleteUser(String login) {
        userRepository
            .findOneByLogin(login)
            .ifPresent(user -> {
                userRepository.delete(user);
                this.clearUserCaches(user);
                LOG.debug("Deleted User: {}", user);
            });
    }

    /**
     * Update basic information (first name, last name, email, language) for the current user.
     *
     * @param firstName first name of user.
     * @param lastName  last name of user.
```

```
     * @param email     email id of user.
     * @param langKey   language key.
     * @param imageUrl  image URL of user.
     */
    public void updateUser(String firstName, String lastName, String email, String langKey, String
imageUrl) {
        SecurityUtils.getCurrentUserLogin()
            .flatMap(userRepository::findOneByLogin)
            .ifPresent(user -> {
                user.setFirstName(firstName);
                user.setLastName(lastName);
                if (email != null) {
                    user.setEmail(email.toLowerCase());
                }
                user.setLangKey(langKey);
                user.setImageUrl(imageUrl);
                userRepository.save(user);
                this.clearUserCaches(user);
                LOG.debug("Changed Information for User: {}", user);
            });
    }

    @Transactional
    public void changePassword(String currentClearTextPassword, String newPassword) {
        SecurityUtils.getCurrentUserLogin()
            .flatMap(userRepository::findOneByLogin)
            .ifPresent(user -> {
                String currentEncryptedPassword = user.getPassword();
                if (!passwordEncoder.matches(currentClearTextPassword, currentEncryptedPassword)) {
                    throw new InvalidPasswordException();
                }
                String encryptedPassword = passwordEncoder.encode(newPassword);
                user.setPassword(encryptedPassword);
                this.clearUserCaches(user);
                LOG.debug("Changed password for User: {}", user);
            });
    }

    @Transactional(readOnly = true)
    public Page<AdminUserDTO> getAllManagedUsers(Pageable pageable) {
        return userRepository.findAll(pageable).map(AdminUserDTO::new);
    }

    @Transactional(readOnly = true)
    public Page<UserDTO> getAllPublicUsers(Pageable pageable) {
        return userRepository.findAllByIdNotNullAndActivatedIsTrue(pageable).map(UserDTO::new);
    }

    @Transactional(readOnly = true)
    public Optional<User> getUserWithAuthoritiesByLogin(String login) {
        return userRepository.findOneWithAuthoritiesByLogin(login);
    }

    @Transactional(readOnly = true)
    public Optional<User> getUserWithAuthorities() {
        return
SecurityUtils.getCurrentUserLogin().flatMap(userRepository::findOneWithAuthoritiesByLogin);
    }

    /**
     * Not activated users should be automatically deleted after 3 days.
     * <p>
     * This is scheduled to get fired every day, at 01:00 (am).
     */
    @Scheduled(cron = "0 0 1 * * ?")
    public void removeNotActivatedUsers() {
        userRepository

.findAllByActivatedIsFalseAndActivationKeyIsNotNullAndCreatedDateBefore(Instant.now().minus(3,
ChronoUnit.DAYS))
            .forEach(user -> {
                LOG.debug("Deleting not activated user {}", user.getLogin());
                userRepository.delete(user);
                this.clearUserCaches(user);
            });
    }

    /**
     * Gets a list of all the authorities.
     * @return a list of all the authorities.
```

```
     */
    @Transactional(readOnly = true)
    public List<String> getAuthorities() {
        return authorityRepository.findAll().stream().map(Authority::getName).toList();
    }

    private void clearUserCaches(User user) {

Objects.requireNonNull(cacheManager.getCache(UserRepository.USERS_BY_LOGIN_CACHE)).evictIfPresent(us
er.getLogin());
        if (user.getEmail() != null) {

Objects.requireNonNull(cacheManager.getCache(UserRepository.USERS_BY_EMAIL_CACHE)).evictIfPresent(us
er.getEmail());
        }
    }
}

package com.cinemacity.management.service;

import com.cinemacity.management.domain.Customer;
import com.cinemacity.management.repository.CustomerRepository;
import com.cinemacity.management.service.dto.CustomerDTO;
import com.cinemacity.management.service.mapper.CustomerMapper;
import java.util.Optional;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * Service Implementation for managing {@link com.cinemacity.management.domain.Customer}.
 */
@Service
@Transactional
public class CustomerService {

    private static final Logger LOG = LoggerFactory.getLogger(CustomerService.class);

    private final CustomerRepository customerRepository;

    private final CustomerMapper customerMapper;

    public CustomerService(CustomerRepository customerRepository, CustomerMapper customerMapper) {
        this.customerRepository = customerRepository;
        this.customerMapper = customerMapper;
    }

    /**
     * Save a customer.
     *
     * @param customerDTO the entity to save.
     * @return the persisted entity.
     */
    public CustomerDTO save(CustomerDTO customerDTO) {
        LOG.debug("Request to save Customer : {}", customerDTO);
        Customer customer = customerMapper.toEntity(customerDTO);
        customer = customerRepository.save(customer);
        return customerMapper.toDto(customer);
    }

    /**
     * Update a customer.
     *
     * @param customerDTO the entity to save.
     * @return the persisted entity.
     */
    public CustomerDTO update(CustomerDTO customerDTO) {
        LOG.debug("Request to update Customer : {}", customerDTO);
        Customer customer = customerMapper.toEntity(customerDTO);
        customer = customerRepository.save(customer);
        return customerMapper.toDto(customer);
    }

    /**
     * Partially update a customer.
     *
     * @param customerDTO the entity to update partially.
```

```
     * @return the persisted entity.
     */
    public Optional<CustomerDTO> partialUpdate(CustomerDTO customerDTO) {
        LOG.debug("Request to partially update Customer : {}", customerDTO);

        return customerRepository
            .findById(customerDTO.getId())
            .map(existingCustomer -> {
                customerMapper.partialUpdate(existingCustomer, customerDTO);

                return existingCustomer;
            })
            .map(customerRepository::save)
            .map(customerMapper::toDto);
    }

    /**
     * Get all the customers.
     *
     * @param pageable the pagination information.
     * @return the list of entities.
     */
    @Transactional(readOnly = true)
    public Page<CustomerDTO> findAll(Pageable pageable) {
        LOG.debug("Request to get all Customers");
        return customerRepository.findAll(pageable).map(customerMapper::toDto);
    }

    /**
     * Get one customer by id.
     *
     * @param id the id of the entity.
     * @return the entity.
     */
    @Transactional(readOnly = true)
    public Optional<CustomerDTO> findOne(Long id) {
        LOG.debug("Request to get Customer : {}", id);
        return customerRepository.findById(id).map(customerMapper::toDto);
    }

    /**
     * Delete the customer by id.
     *
     * @param id the id of the entity.
     */
    public void delete(Long id) {
        LOG.debug("Request to delete Customer : {}", id);
        customerRepository.deleteById(id);
    }
}
package com.cinemacity.management.service;

import com.cinemacity.management.domain.Employee;
import com.cinemacity.management.repository.EmployeeRepository;
import com.cinemacity.management.service.dto.EmployeeDTO;
import com.cinemacity.management.service.mapper.EmployeeMapper;
import java.util.Optional;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * Service Implementation for managing {@link com.cinemacity.management.domain.Employee}.
 */
@Service
@Transactional
public class EmployeeService {

    private static final Logger LOG = LoggerFactory.getLogger(EmployeeService.class);

    private final EmployeeRepository employeeRepository;

    private final EmployeeMapper employeeMapper;

    public EmployeeService(EmployeeRepository employeeRepository, EmployeeMapper employeeMapper) {
        this.employeeRepository = employeeRepository;
        this.employeeMapper = employeeMapper;
```

```
}

/**
 * Save a employee.
 *
 * @param employeeDTO the entity to save.
 * @return the persisted entity.
 */
public EmployeeDTO save(EmployeeDTO employeeDTO) {
    LOG.debug("Request to save Employee : {}", employeeDTO);
    Employee employee = employeeMapper.toEntity(employeeDTO);
    employee = employeeRepository.save(employee);
    return employeeMapper.toDto(employee);
}

/**
 * Update a employee.
 *
 * @param employeeDTO the entity to save.
 * @return the persisted entity.
 */
public EmployeeDTO update(EmployeeDTO employeeDTO) {
    LOG.debug("Request to update Employee : {}", employeeDTO);
    Employee employee = employeeMapper.toEntity(employeeDTO);
    employee = employeeRepository.save(employee);
    return employeeMapper.toDto(employee);
}

/**
 * Partially update a employee.
 *
 * @param employeeDTO the entity to update partially.
 * @return the persisted entity.
 */
public Optional<EmployeeDTO> partialUpdate(EmployeeDTO employeeDTO) {
    LOG.debug("Request to partially update Employee : {}", employeeDTO);

    return employeeRepository
        .findById(employeeDTO.getId())
        .map(existingEmployee -> {
            employeeMapper.partialUpdate(existingEmployee, employeeDTO);

            return existingEmployee;
        })
        .map(employeeRepository::save)
        .map(employeeMapper::toDto);
}

/**
 * Get all the employees.
 *
 * @param pageable the pagination information.
 * @return the list of entities.
 */
@Transactional(readOnly = true)
public Page<EmployeeDTO> findAll(Pageable pageable) {
    LOG.debug("Request to get all Employees");
    return employeeRepository.findAll(pageable).map(employeeMapper::toDto);
}

/**
 * Get one employee by id.
 *
 * @param id the id of the entity.
 * @return the entity.
 */
@Transactional(readOnly = true)
public Optional<EmployeeDTO> findOne(Long id) {
    LOG.debug("Request to get Employee : {}", id);
    return employeeRepository.findById(id).map(employeeMapper::toDto);
}

/**
 * Delete the employee by id.
 *
 * @param id the id of the entity.
 */
public void delete(Long id) {
    LOG.debug("Request to delete Employee : {}", id);
    employeeRepository.deleteById(id);
```

```
    }
}

package com.cinemacity.management.domain;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import jakarta.persistence.*;
import jakarta.validation.constraints.*;
import java.io.Serializable;
import java.time.LocalDate;
import java.util.HashSet;
import java.util.Set;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

/**
 * A Movie.
 */
@Entity
@Table(name = "movie")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@SuppressWarnings("common-java:DuplicatedBlocks")
public class Movie implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "sequenceGenerator")
    @SequenceGenerator(name = "sequenceGenerator")
    @Column(name = "id")
    private Long id;

    @NotNull
    @Column(name = "title", nullable = false)
    private String title;

    @Lob
    @Column(name = "description")
    private String description;

    @NotNull
    @Column(name = "duration", nullable = false)
    private Integer duration;

    @Column(name = "release_date")
    private LocalDate releaseDate;

    @Column(name = "rating")
    private Double rating;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "movie")
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    @JsonIgnoreProperties(value = { "tickets", "hall", "movie" }, allowSetters = true)
    private Set<Screening> screenings = new HashSet<>();

    public Long getId() {
        return this.id;
    }

    public Movie id(Long id) {
        this.setId(id);
        return this;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getTitle() {
        return this.title;
    }

    public Movie title(String title) {
        this.setTitle(title);
        return this;
    }

    public void setTitle(String title) {
        this.title = title;
    }
```

```
public String getDescription() {
    return this.description;
}

public Movie description(String description) {
    this.setDescription(description);
    return this;
}

public void setDescription(String description) {
    this.description = description;
}

public Integer getDuration() {
    return this.duration;
}

public Movie duration(Integer duration) {
    this.setDuration(duration);
    return this;
}

public void setDuration(Integer duration) {
    this.duration = duration;
}

public LocalDate getReleaseDate() {
    return this.releaseDate;
}

public Movie releaseDate(LocalDate releaseDate) {
    this.setReleaseDate(releaseDate);
    return this;
}

public void setReleaseDate(LocalDate releaseDate) {
    this.releaseDate = releaseDate;
}

public Double getRating() {
    return this.rating;
}

public Movie rating(Double rating) {
    this.setRating(rating);
    return this;
}

public void setRating(Double rating) {
    this.rating = rating;
}

public Set<Screening> getScreenings() {
    return this.screenings;
}

public void setScreenings(Set<Screening> screenings) {
    if (this.screenings != null) {
        this.screenings.forEach(i -> i.setMovie(null));
    }
    if (screenings != null) {
        screenings.forEach(i -> i.setMovie(this));
    }
    this.screenings = screenings;
}

public Movie screenings(Set<Screening> screenings) {
    this.setScreenings(screenings);
    return this;
}

public Movie addScreenings(Screening screening) {
    this.screenings.add(screening);
    screening.setMovie(this);
    return this;
}

public Movie removeScreenings(Screening screening) {
    this.screenings.remove(screening);
```

```
        screening.setMovie(null);
        return this;
    }


    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Movie)) {
            return false;
        }
        return getId() != null && getId().equals(((Movie) o).getId());
    }

    @Override
    public int hashCode() {
        return getClass().hashCode();
    }

    // prettier-ignore
    @Override
    public String toString() {
        return "Movie{" +
            "id=" + getId() +
            ", title='" + getTitle() + "'" +
            ", description='" + getDescription() + "'" +
            ", duration=" + getDuration() +
            ", releaseDate='" + getReleaseDate() + "'" +
            ", rating=" + getRating() +
            "}";
    }
}
```

```
package com.cinemacity.management.domain;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import jakarta.persistence.*;
import jakarta.validation.constraints.*;
import java.io.Serializable;
import java.math.BigDecimal;
import java.time.ZonedDateTime;
import java.util.HashSet;
import java.util.Set;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

/**
 * A Sale.
 */
@Entity
@Table(name = "sale")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@SuppressWarnings("common-java:DuplicatedBlocks")
public class Sale implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "sequenceGenerator")
    @SequenceGenerator(name = "sequenceGenerator")
    @Column(name = "id")
    private Long id;

    @NotNull
    @Column(name = "sale_date", nullable = false)
    private ZonedDateTime saleDate;

    @NotNull
    @Column(name = "total_amount", precision = 21, scale = 2, nullable = false)
    private BigDecimal totalAmount;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "sale")
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    @JsonIgnoreProperties(value = { "sale" }, allowSetters = true)
    private Set<Snack> snacks = new HashSet<>();

    public Long getId() {
        return this.id;
```

```
}

public Sale id(Long id) {
    this.setId(id);
    return this;
}

public void setId(Long id) {
    this.id = id;
}

public ZonedDateTime getSaleDate() {
    return this.saleDate;
}

public Sale saleDate(ZonedDateTime saleDate) {
    this.setSaleDate(saleDate);
    return this;
}

public void setSaleDate(ZonedDateTime saleDate) {
    this.saleDate = saleDate;
}

public BigDecimal getTotalAmount() {
    return this.totalAmount;
}

public Sale totalAmount(BigDecimal totalAmount) {
    this.setTotalAmount(totalAmount);
    return this;
}

public void setTotalAmount(BigDecimal totalAmount) {
    this.totalAmount = totalAmount;
}

public Set<Snack> getSnacks() {
    return this.snacks;
}

public void setSnacks(Set<Snack> snacks) {
    if (this.snacks != null) {
        this.snacks.forEach(i -> i.setSale(null));
    }
    if (snacks != null) {
        snacks.forEach(i -> i.setSale(this));
    }
    this.snacks = snacks;
}

public Sale snacks(Set<Snack> snacks) {
    this.setSnacks(snacks);
    return this;
}

public Sale addSnacks(Snack snack) {
    this.snacks.add(snack);
    snack.setSale(this);
    return this;
}

public Sale removeSnacks(Snack snack) {
    this.snacks.remove(snack);
    snack.setSale(null);
    return this;
}


@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if (!(o instanceof Sale)) {
        return false;
    }
    return getId() != null && getId().equals(((Sale) o).getId());
}
```

```
    @Override
    public int hashCode() {
        return getClass().hashCode();
    }

    // prettier-ignore
    @Override
    public String toString() {
        return "Sale{" +
            "id=" + getId() +
            ", saleDate='" + getSaleDate() + "'" +
            ", totalAmount=" + getTotalAmount() +
            "}";
    }
}
```

```
package com.cinemacity.management.domain;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import jakarta.persistence.*;
import jakarta.validation.constraints.*;
import java.io.Serializable;
import java.util.Objects;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;
import org.springframework.data.domain.Persistable;

/**
 * A Authority.
 */
@Entity
@Table(name = "jhi_authority")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@JsonIgnoreProperties(value = { "new", "id" })
@SuppressWarnings("common-java:DuplicatedBlocks")
public class Authority implements Serializable, Persistable<String> {

    private static final long serialVersionUID = 1L;

    @NotNull
    @Size(max = 50)
    @Id
    @Column(name = "name", length = 50, nullable = false)
    private String name;

    @org.springframework.data.annotation.Transient
    @Transient
    private boolean isPersisted;

    public String getName() {
        return this.name;
    }

    public Authority name(String name) {
        this.setName(name);
        return this;
    }

    public void setName(String name) {
        this.name = name;
    }

    @PostLoad
    @PostPersist
    public void updateEntityState() {
        this.setIsPersisted();
    }

    @Override
    public String getId() {
        return this.name;
    }

    @org.springframework.data.annotation.Transient
    @Transient
    @Override
    public boolean isNew() {
        return !this.isPersisted;
    }
```

```
    public Authority setIsPersisted() {
        this.isPersisted = true;
        return this;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Authority)) {
            return false;
        }
        return getName() != null && getName().equals(((Authority) o).getName());
    }

    @Override
    public int hashCode() {
        return Objects.hashCode(getName());
    }

    // prettier-ignore
    @Override
    public String toString() {
        return "Authority{" +
            "name=" + getName() +
            "}";
    }
}
```

```
package com.cinemacity.management.domain;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import jakarta.persistence.*;
import jakarta.validation.constraints.*;
import java.io.Serializable;
import java.math.BigDecimal;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

/**
 * A Snack.
 */
@Entity
@Table(name = "snack")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@SuppressWarnings("common-java:DuplicatedBlocks")
public class Snack implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "sequenceGenerator")
    @SequenceGenerator(name = "sequenceGenerator")
    @Column(name = "id")
    private Long id;

    @NotNull
    @Column(name = "name", nullable = false)
    private String name;

    @NotNull
    @Column(name = "price", precision = 21, scale = 2, nullable = false)
    private BigDecimal price;

    @NotNull
    @Column(name = "quantity", nullable = false)
    private Integer quantity;

    @ManyToOne(fetch = FetchType.LAZY)
    @JsonIgnoreProperties(value = { "snacks" }, allowSetters = true)
    private Sale sale;

    public Long getId() {
        return this.id;
    }

    public Snack id(Long id) {
```

```
    this.setId(id);
    return this;
}

public void setId(Long id) {
    this.id = id;
}

public String getName() {
    return this.name;
}

public Snack name(String name) {
    this.setName(name);
    return this;
}

public void setName(String name) {
    this.name = name;
}

public BigDecimal getPrice() {
    return this.price;
}

public Snack price(BigDecimal price) {
    this.setPrice(price);
    return this;
}

public void setPrice(BigDecimal price) {
    this.price = price;
}

public Integer getQuantity() {
    return this.quantity;
}

public Snack quantity(Integer quantity) {
    this.setQuantity(quantity);
    return this;
}

public void setQuantity(Integer quantity) {
    this.quantity = quantity;
}

public Sale getSale() {
    return this.sale;
}

public void setSale(Sale sale) {
    this.sale = sale;
}

public Snack sale(Sale sale) {
    this.setSale(sale);
    return this;
}

@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if (!(o instanceof Snack)) {
        return false;
    }
    return getId() != null && getId().equals(((Snack) o).getId());
}

@Override
public int hashCode() {
    return getClass().hashCode();
}

// prettier-ignore
@Override
public String toString() {
    return "Snack{" +
```

```
            "id=" + getId() +
            ", name='" + getName() + "'" +
            ", price=" + getPrice() +
            ", quantity=" + getQuantity() +
            "}";
    }
}

package com.cinemacity.management.management;

import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;
import org.springframework.stereotype.Service;

@Service
public class SecurityMetersService {

    public static final String INVALID_TOKENS_METER_NAME = "security.authentication.invalid-tokens";
    public static final String INVALID_TOKENS_METER_DESCRIPTION =
        "Indicates validation error count of the tokens presented by the clients.";
    public static final String INVALID_TOKENS_METER_BASE_UNIT = "errors";
    public static final String INVALID_TOKENS_METER_CAUSE_DIMENSION = "cause";

    private final Counter tokenInvalidSignatureCounter;
    private final Counter tokenExpiredCounter;
    private final Counter tokenUnsupportedCounter;
    private final Counter tokenMalformedCounter;

    public SecurityMetersService(MeterRegistry registry) {
        this.tokenInvalidSignatureCounter = invalidTokensCounterForCauseBuilder("invalid-
signature").register(registry);
        this.tokenExpiredCounter =
invalidTokensCounterForCauseBuilder("expired").register(registry);
        this.tokenUnsupportedCounter =
invalidTokensCounterForCauseBuilder("unsupported").register(registry);
        this.tokenMalformedCounter =
invalidTokensCounterForCauseBuilder("malformed").register(registry);
    }

    private Counter.Builder invalidTokensCounterForCauseBuilder(String cause) {
        return Counter.builder(INVALID_TOKENS_METER_NAME)
            .baseUnit(INVALID_TOKENS_METER_BASE_UNIT)
            .description(INVALID_TOKENS_METER_DESCRIPTION)
            .tag(INVALID_TOKENS_METER_CAUSE_DIMENSION, cause);
    }

    public void trackTokenInvalidSignature() {
        this.tokenInvalidSignatureCounter.increment();
    }

    public void trackTokenExpired() {
        this.tokenExpiredCounter.increment();
    }

    public void trackTokenUnsupported() {
        this.tokenUnsupportedCounter.increment();
    }

    public void trackTokenMalformed() {
        this.tokenMalformedCounter.increment();
    }
}

package com.cinemacity.management.web.rest;

import com.cinemacity.management.repository.MovieRepository;
import com.cinemacity.management.service.MovieService;
import com.cinemacity.management.service.dto.MovieDTO;
import com.cinemacity.management.web.rest.errors.BadRequestAlertException;
import jakarta.validation.Valid;
import jakarta.validation.constraints.NotNull;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.List;
import java.util.Objects;
import java.util.Optional;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.domain.Page;
```

```
import org.springframework.data.domain.Pageable;
import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;
import tech.jhipster.web.util.HeaderUtil;
import tech.jhipster.web.util.PaginationUtil;
import tech.jhipster.web.util.ResponseUtil;

/**
 * REST controller for managing {@link com.cinemacity.management.domain.Movie}.
 */
@RestController
@RequestMapping("/api/movies")
public class MovieResource {

    private static final Logger LOG = LoggerFactory.getLogger(MovieResource.class);

    private static final String ENTITY_NAME = "movie";

    @Value("${jhipster.clientApp.name}")
    private String applicationName;

    private final MovieService movieService;

    private final MovieRepository movieRepository;

    public MovieResource(MovieService movieService, MovieRepository movieRepository) {
        this.movieService = movieService;
        this.movieRepository = movieRepository;
    }

    /**
     * {@code POST  /movies} : Create a new movie.
     *
     * @param movieDTO the movieDTO to create.
     * @return the {@link ResponseEntity} with status {@code 201 (Created)} and with body the new
movieDTO, or with status {@code 400 (Bad Request)} if the movie has already an ID.
     * @throws URISyntaxException if the Location URI syntax is incorrect.
     */
    @PostMapping("")
    public ResponseEntity<MovieDTO> createMovie(@Valid @RequestBody MovieDTO movieDTO) throws
URISyntaxException {
        LOG.debug("REST request to save Movie : {}", movieDTO);
        if (movieDTO.getId() != null) {
            throw new BadRequestAlertException("A new movie cannot already have an ID", ENTITY_NAME,
"idexists");
        }
        movieDTO = movieService.save(movieDTO);
        return ResponseEntity.created(new URI("/api/movies/" + movieDTO.getId()))
            .headers(HeaderUtil.createEntityCreationAlert(applicationName, true, ENTITY_NAME,
movieDTO.getId().toString()))
            .body(movieDTO);
    }

    /**
     * {@code PUT  /movies/:id} : Updates an existing movie.
     *
     * @param id the id of the movieDTO to save.
     * @param movieDTO the movieDTO to update.
     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and with body the updated
movieDTO,
     * or with status {@code 400 (Bad Request)} if the movieDTO is not valid,
     * or with status {@code 500 (Internal Server Error)} if the movieDTO couldn't be updated.
     * @throws URISyntaxException if the Location URI syntax is incorrect.
     */
    @PutMapping("/{id}")
    public ResponseEntity<MovieDTO> updateMovie(
        @PathVariable(value = "id", required = false) final Long id,
        @Valid @RequestBody MovieDTO movieDTO
    ) throws URISyntaxException {
        LOG.debug("REST request to update Movie : {}, {}", id, movieDTO);
        if (movieDTO.getId() == null) {
            throw new BadRequestAlertException("Invalid id", ENTITY_NAME, "idnull");
        }
        if (!Objects.equals(id, movieDTO.getId())) {
            throw new BadRequestAlertException("Invalid ID", ENTITY_NAME, "idinvalid");
        }

        if (!movieRepository.existsById(id)) {
```

```
            throw new BadRequestAlertException("Entity not found", ENTITY_NAME, "idnotfound");
        }

        movieDTO = movieService.update(movieDTO);
        return ResponseEntity.ok()
            .headers(HeaderUtil.createEntityUpdateAlert(applicationName, true, ENTITY_NAME,
movieDTO.getId().toString()))
            .body(movieDTO);
    }

    /**
     * {@code PATCH  /movies/:id} : Partial updates given fields of an existing movie, field will
ignore if it is null
     *
     * @param id the id of the movieDTO to save.
     * @param movieDTO the movieDTO to update.
     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and with body the updated
movieDTO,
     * or with status {@code 400 (Bad Request)} if the movieDTO is not valid,
     * or with status {@code 404 (Not Found)} if the movieDTO is not found,
     * or with status {@code 500 (Internal Server Error)} if the movieDTO couldn't be updated.
     * @throws URISyntaxException if the Location URI syntax is incorrect.
     */
    @PatchMapping(value = "/{id}", consumes = { "application/json", "application/merge-patch+json"
})
    public ResponseEntity<MovieDTO> partialUpdateMovie(
        @PathVariable(value = "id", required = false) final Long id,
        @NotNull @RequestBody MovieDTO movieDTO
    ) throws URISyntaxException {
        LOG.debug("REST request to partial update Movie partially : {}, {}", id, movieDTO);
        if (movieDTO.getId() == null) {
            throw new BadRequestAlertException("Invalid id", ENTITY_NAME, "idnull");
        }
        if (!Objects.equals(id, movieDTO.getId())) {
            throw new BadRequestAlertException("Invalid ID", ENTITY_NAME, "idinvalid");
        }

        if (!movieRepository.existsById(id)) {
            throw new BadRequestAlertException("Entity not found", ENTITY_NAME, "idnotfound");
        }

        Optional<MovieDTO> result = movieService.partialUpdate(movieDTO);

        return ResponseUtil.wrapOrNotFound(
            result,
            HeaderUtil.createEntityUpdateAlert(applicationName, true, ENTITY_NAME,
movieDTO.getId().toString())
        );
    }

    /**
     * {@code GET  /movies} : get all the movies.
     *
     * @param pageable the pagination information.
     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and the list of movies in
body.
     */
    @GetMapping("")
    public ResponseEntity<List<MovieDTO>>
getAllMovies(@org.springdoc.core.annotations.ParameterObject Pageable pageable) {
        LOG.debug("REST request to get a page of Movies");
        Page<MovieDTO> page = movieService.findAll(pageable);
        HttpHeaders headers =
PaginationUtil.generatePaginationHttpHeaders(ServletUriComponentsBuilder.fromCurrentRequest(),
page);
        return ResponseEntity.ok().headers(headers).body(page.getContent());
    }

    /**
     * {@code GET  /movies/:id} : get the "id" movie.
     *
     * @param id the id of the movieDTO to retrieve.
     * @return the {@link ResponseEntity} with status {@code 200 (OK)} and with body the movieDTO,
or with status {@code 404 (Not Found)}.
     */
    @GetMapping("/{id}")
    public ResponseEntity<MovieDTO> getMovie(@PathVariable("id") Long id) {
        LOG.debug("REST request to get Movie : {}", id);
        Optional<MovieDTO> movieDTO = movieService.findOne(id);
        return ResponseUtil.wrapOrNotFound(movieDTO);
```

```
    }

    /**
     * {@code DELETE  /movies/:id} : delete the "id" movie.
     *
     * @param id the id of the movieDTO to delete.
     * @return the {@link ResponseEntity} with status {@code 204 (NO_CONTENT)}.
     */
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteMovie(@PathVariable("id") Long id) {
        LOG.debug("REST request to delete Movie : {}", id);
        movieService.delete(id);
        return ResponseEntity.noContent()
            .headers(HeaderUtil.createEntityDeletionAlert(applicationName, true, ENTITY_NAME,
id.toString()))
            .build();
    }
}
```

```
package com.cinemacity.management.broker;

import static org.springframework.web.servlet.mvc.method.annotation.SseEmitter.event;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;
import java.util.function.Consumer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.mvc.method.annotation.SseEmitter;

@Component
public class KafkaConsumer implements Consumer<String> {

    private static final Logger LOG = LoggerFactory.getLogger(KafkaConsumer.class);

    private Map<String, SseEmitter> emitters = new HashMap<>();

    public SseEmitter register(String key) {
        LOG.debug("Registering sse client for {}", key);
        SseEmitter emitter = new SseEmitter();
        emitter.onCompletion(() -> emitters.remove(key));
        emitters.put(key, emitter);
        return emitter;
    }

    public void unregister(String key) {
        LOG.debug("Unregistering sse emitter for: {}", key);
        Optional.ofNullable(emitters.get(key)).ifPresent(SseEmitter::complete);
    }

    @Override
    public void accept(String input) {
        LOG.debug("Got message from kafka stream: {}", input);
        emitters
            .entrySet()
            .stream()
            .map(Map.Entry::getValue)
            .forEach((SseEmitter emitter) -> {
                try {
                    emitter.send(event().data(input, MediaType.TEXT_PLAIN));
                } catch (IOException e) {
                    LOG.debug("error sending sse message, {}", input);
                }
            });
    }
}
```

```
application.yaml – налаштування застосунка
name: cinemacitymanagement
services:
  app:
    image: cinemacitymanagement
    environment:
      - _JAVA_OPTIONS=-Xmx512m -Xms256m
```

```
    - SPRING_PROFILES_ACTIVE=prod,api-docs
    - MANAGEMENT_PROMETHEUS_METRICS_EXPORT_ENABLED=true
    - SPRING_DATASOURCE_URL=jdbc:postgresql://postgresql:5432/cinemaCityManagement
    - SPRING_LIQUIBASE_URL=jdbc:postgresql://postgresql:5432/cinemaCityManagement
    - SPRING_CLOUD_STREAM_KAFKA_BINDER_BROKERS=kafka:9092
  ports:
    - 127.0.0.1:8080:8080
  healthcheck:
    test:
      - CMD
      - curl
      - -f
      - http://localhost:8080/management/health
    interval: 5s
    timeout: 5s
    retries: 40
  depends_on:
    postgresql:
      condition: service_healthy
postgresql:
  extends:
    file: ./postgresql.yml
    service: postgresql
kafka:
  extends:
    file: ./kafka.yml
    service: kafka
```

Додаток В (обов'язковий)
ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Інформаційна система управління ресурсами кінотеатру

Тип роботи: бакалаврська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ. АІІТ, ФІІТА, 1ІСТ-21Б
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 20,36 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

☑ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

Олег БІСІКАЛО, зав. каф. АІІТ _____ (підпис)
(прізвище, ініціали, посада)

Любов ЯНЧУК, секретар. каф. АІІТ _____ (підпис)
(прізвище, ініціали, посада)

Особа, відповідальна за перевірку _____ Костянтин ОВЧИННИКОВ
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Ілона БОГАЧ, к. т. н., доц. каф. АІІТ
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Владислав СЛОБОДИСЬКИЙ
(підпис) (прізвище, ініціали)