КВАЛІФІКАЦІЙНА РОБОТА

Група ІПЗс-20-2

Кудла Р.О.

2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Кудла Роман Олегович

УДК 004.4

Розробка веб-орієнтованої системи керування діяльністю кінотеатру засобами Ruby on Rails

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації бакалавр

Нормоконтроль	Студент
_____________Стисло О.В.	______________Кудла Р.О.
(підпис, дата, розшифрування підпису)	(підпис, дата, розшифрування підпису)
Допускається до захисту Завідувач кафедри	Керівник роботи
______________к.т.н., доц. Ващишак С.П.	______________к.т.н., доц. Слабінога М.О.
(підпис, дата, розшифрування підпису)	(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

Освітній ступінь: «бакалавр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

____

«____ » ______________ 2024 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Кудла Роман Олегович

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:

Розробка веб-орієнтованої системи керування діяльністю кінотеатру засобами Ruby on Rails

керівник роботи:

Слабінога Мар'ян Остапович, к.т.н., доцент

затверджена наказом вищого навчального закладу від « 12 » березня 2024 року

№ 19/1

2. Термін подання студентом роботи 05.06.2024
3. Вихідні дані роботи: мова програмування Ruby, фреймворк Ruby on Rails, мова програмування JavaScript
4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)
   1. Огляд предметної області
   2. Проєктування та план розробки додатку
   3. Програмна реалізація веб-додатку
5. Дата видачі завдання 14.03.2024

КОНСУЛЬТАНТИ РОЗДІЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Розділ	Консультант (прізвище, ініціали та посада)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	Огляд предметної області	22.03.2024	Виконано
2.	Проєктування та план розробки додатку	29.03.2024	Виконано
3.	Ініціалізація проєкту та розробка базової частини	12.04.2024	Виконано
4.	Розробка адміністративної частини додатку	03.05.2024	Виконано
5.	Розробка користувацького порталу	15.05.2024	Виконано
6.	Оформлення пояснювальної записки	20.05.2024	Виконано
7.	Оформлення графічного матеріалу та підготовка до захисту роботи	24.05.2024	Виконано

Студент ______________ Кудла Р.О.
(підпис) (прізвище та ініціали)

Керівник роботи ______________ Слабінога М.О.
(підпис) (прізвище та ініціали)

Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Сторінка	Опис графічного матеріалу	Сторінка	Опис графічного матеріалу
14	Сторінка розкладу кінотеатру “Галичина”	54	Модульне вікно створення залу
15	Сторінка розкладу кінотеатру “Movieland”	57	Модальне вікно створення залу з схемою місць
16	Сторінка розкладу сеансів кінотеатру “Люм’єр”	58	Модальне вікно створення сеансу
19	Основні компоненти User Story	60	Сторінка бронювання квитків в панелі адміністратора
50	Форма для додавання фільму вручну	63	Сторінка бронювання квитків для відвідувачів
51	Сторінка додавання фільму за допомогою інтеграції з TMDB	65	Сторінка оплати Stripe

АНОТАЦІЯ

У кваліфікаційній роботі досліджується, аналізується та розробляється веб-сайт для керування діяльністю кінотеатру. Впровадження такого веб-сайту дозволить зменшити витрати часу на налаштування залів, додавання фільмів та створення сесій, а також підвищити загальну ефективність роботи кінотеатру.

В першому розділі проведено аналіз веб-сайтів кінотеатрів. Розглянуто основні компоненти та можливості існуючих веб-платформ. Проведено порівняльний аналіз з метою виявлення переваг та недоліків кожної з розглянутих систем. На основі цього аналізу сформульовано чіткі вимоги та задачі для подальшого дослідження та розробки власного рішення.

У другому розділі розглянуто написання User Stories для визначення вимог до системи та функціональних можливостей веб-сайту, описано використання фреймворку Rails, зокрема його переваги. Проведено аналіз вибору інструментів для розробки та для розгортання додатку у виробниче середовище.

У третьому розділі детально описано програмну реалізацію веб-сайту для управління діяльністю кінотеатру, включаючи розробку його основних модулів та компонентів. Використовуючи фреймворк Rails, було створено моделі для управління базою даних, контролери для обробки запитів та представлення для відображення інформації користувачам.

В результаті проведеної роботи було створено функціональний веб-сайт для управління діяльністю кінотеатру, який значно оптимізує процеси налаштування залів, додавання фільмів та створення сеансів. Веб-сайт забезпечує ефективну обробку платежів та отримання актуальної інформації про фільми через інтеграції з сторонніми сервісами. Розроблений веб-сайт не лише підвищує ефективність робочих процесів кінотеатру, але й забезпечує покращений досвід користувачів для адміністраторів та відвідувачів.

КЛЮЧОВІ СЛОВА: КІНОТЕАТР, RUBY, RUBY ON RAILS, STIMULUS, STRIPE, THE MOVIE DATABASE

SUMMARY

In the qualification work, a website for managing the activities of a cinema is researched, analyzed, and developed. The implementation of such a website will reduce the time spent on setting up auditoriums, adding movies, and creating sessions, as well as increase the overall efficiency of the cinema's operations.

In the first section, an analysis of cinema websites is conducted. The main components and capabilities of existing web platforms are considered. A comparative analysis is performed to identify the advantages and disadvantages of each system reviewed. Based on this analysis, clear requirements and tasks for further research and development of our own solution are formulated.

In the second section, writing User Stories to define system requirements and website functionalities is discussed, and the use of the Rails framework, particularly its advantages, is described. An analysis of the selection of tools for development and deployment of the application in a production environment is also conducted.

The third section provides a detailed description of the software implementation of the website for managing the cinema's activities, including the development of its main modules and components. Using the Rails framework, models for database management, controllers for request processing, and views for information display to users were created.

As a result of the work done, a functional website for managing the cinema's activities was created, significantly optimizing the processes of setting up auditoriums, adding movies, and creating sessions. The website ensures efficient payment processing and access to current movie information through integrations with third-party services. The developed website not only improves the efficiency of the cinema's operational processes but also provides an enhanced user experience for administrators and visitors.

KEYWORDS: CINEMA, RUBY, RUBY ON RAILS, STIMULUS, STRIPE, THE MOVIE DATABASE

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ........ 7
ВСТУП........ 8
РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ........ 11
1.1 Аналіз веб-сайтів кінотеатрів........ 11
1.2 Загальний опис веб-сайту........ 12
1.3 Огляд та виявлення недоліків існуючих аналогів........ 14
1.4 Постановка задачі........ 17
Висновки до розділу 1........ 18
РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА ПЛАН РОЗРОБКИ ДОДАТКУ........ 19
2.1 Написання User Stories........ 19
2.2 Особливості Rails та вибір інструментів для розробки........ 21
2.3 Розгортання проєкту у виробниче середовище........ 23
Висновки до розділу 2........ 25
РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ........ 26
3.1 Ініціалізація та налаштування проєкту........ 26
3.2 Створення моделей Movie, Hall та Session........ 31
3.3 Автентифікація та авторизація користувачів........ 37
3.4 Написання сервісів для додавання фільмів........ 42
3.5 Створення адміністративної панелі для основних моделей........ 48
3.6 Розробка функціоналу для бронювання квитків........ 60
Висновки до розділу 3........ 67
ВИСНОВКИ........ 68
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ........ 69

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – Програмні інтерфейси

БД – База даних

Гем – бібліотека або інструмент

Екшин – метод контролера, який обробляє запит та формує відповідь

Компілятор – програма, яка перетворює вихідний код, написаний мовою програмування високого рівня, у машинний код, зрозумілий для виконання комп'ютером

Паршал – фрагмент шаблону, який можна повторно використовувати в інших шаблонах для зменшення дублювання коду

Скоуп – метод, який дозволяє визначити умови вибору даних з бази даних для моделі

Ендпоінт – це URL, за яким можна отримати доступ до певного функціоналу у веб-додатку чи API

Колбек – функція, яка викликаються автоматично в певних точках життєвого циклу об'єктів моделей чи контролерів

Рендер – процес відображення веб-сторінки або компоненту на веб-сайті на основі відповідного шаблону та даних

Таб – вкладка, яка містить певну інформацію

Неймспейс – механізм організації коду, який дозволяє групувати класи, модулі та інші об'єкти відповідно до їхнього призначення або функціональності

ВСТУП

Актуальність теми. Розробка веб-орієнтованої системи керування діяльністю кінотеатру засобами Ruby on Rails обумовлена необхідністю цифровізації та оптимізації процесів управління в сфері розваг та дозвілля. В умовах постійного зростання конкуренції у розважальному бізнесі, ефективність та швидкість обслуговування клієнтів, а також гнучкість управління ресурсами кінотеатру стають вирішальними факторами успіху. Сучасні технології веб-розробки, зокрема Ruby on Rails, дозволяють створити високофункціональну, легко масштабовану та інтуїтивно зрозумілу систему, яка може ефективно задовольнити потреби відвідувачів та керівництва кінотеатру.

Впровадження такої системи дозволить оптимізувати робочі процеси, зокрема планування показів, управління квитками, аналіз відвідуваності та прибутковості, що сприятиме підвищенню ефективності роботи кінотеатру. Крім того, цифровізація дозволяє розширити можливості для маркетингових кампаній та покращення сервісу для клієнтів, забезпечуючи більшу лояльність відвідувачів та збільшення прибутків. Отже, розробка та впровадження веб-орієнтованої системи керування діяльністю кінотеатру є актуальною задачею, що відповідає сучасним тенденціям розвитку розважальної індустрії та потребам ринку.

Мета і завдання дослідження. Розробка веб-орієнтованої системи управління діяльністю кінотеатру на основі технології Ruby on Rails, що спрямована на оптимізацію внутрішніх процесів, покращення якості обслуговування клієнтів та збільшення загальної ефективності роботи кінотеатру. Для досягнення мети покладені такі задачі:

– визначення ключових вимог до функціональності веб-орієнтованої системи управління кінотеатром, заснованої на аналізі потреб користувачів;

– розробка архітектури системи, що враховує сучасні принципи веб-розробки та специфіку управління діяльністю кінотеатру;

– реалізація модулів для управління сеансами, продажу квитків, управління контентом та аналітики, забезпечуючи високий рівень безпеки та користувацької взаємодії;

– тестування системи з метою виявлення та усунення помилок, перевірка її ефективності та зручності для кінцевих користувачів.

Об'єкт дослідження. Процес управління діяльністю кінотеатру, що включає організацію показів фільмів та продаж квитків.

Предмет дослідження. Методи та інструменти веб-орієнтованої системи управління, реалізовані за допомогою технології Ruby on Rails, що спрямовані на оптимізацію внутрішніх процесів управління кінотеатру та підвищення якості обслуговування його клієнтів.

Методи дослідження. Для досягнення мети дослідження та вирішення поставлених завдань у роботі застосовувалися наступні методи дослідження:

– аналіз – використовувався для визначення сучасних тенденцій розробки веб-систем, особливостей використання Ruby on Rails в проєктах подібного типу, а також для дослідження вимог до систем управління діяльністю кінотеатрів;

– моделювання – метод застосовувався для розробки архітектури системи та побудови моделей даних, що включає структуру бази даних, взаємозв'язки між елементами системи та алгоритми обробки інформації;

– програмування – основний метод реалізації системи, включаючи написання коду для фронтенду та бекенду, інтеграцію з зовнішніми сервісами та реалізацію функціоналу згідно з зазначеними вимогами;

– тестування – використовувалось для перевірки функціональності та надійності системи, включаючи юніт-тестування, інтеграційне тестування та приймальне тестування з метою виявлення та усунення помилок.

Ці методи дослідження дозволили всебічно підійти до розробки та впровадження веб-орієнтованої системи керування діяльністю кінотеатру, оцінити її ефективність та визначити шляхи подальшого розвитку.

Практичне значення одержаних результатів. Результат цього дослідження полягає у можливості впровадження функціоналу для оптимізації рутинних процесів, таких як налаштування залів, додавання фільмів та створення сеансів, що призведе до ефективного використання робочого часу. Крім того, інтеграція зі сторонніми сервісами для обробки платежів та отримання інформації про фільми дозволить забезпечити зручний та безперебійний досвід для відвідувачів кінотеатру. В результаті, підвищиться конкурентоспроможність кінотеатру на ринку та покращиться задоволеність його клієнтів.

Апробація результатів дослідження. Матеріали кваліфікаційної роботи були представлені на XI Міжнародної наукової конференції «Студентські наукові дискусії поза форматом», яка відбулася 11 квітня 2023 року в Університеті Короля Данила.

Структура. Розділи – 3. Загальний обсяг основної частини – 62 сторінки. Список використаних джерел – 21.

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз веб-сайтів кінотеатрів

Веб-сайти кінотеатрів представляють собою комплексні цифрові платформи, призначені для задоволення потреб як відвідувачів, так і керівництва кінотеатрів. Вони об'єднують в собі функціонал для просування фільмів, продажу квитків, взаємодії з аудиторією та аналітики. Сучасний веб-сайт кінотеатру є не лише інформаційним ресурсом, але й важливим елементом у створенні загального враження від кінотеатру.

Загальний дизайн веб-сайтів кінотеатрів зазвичай фокусується на візуальній привабливості, з легкістю навігації та доступністю всієї необхідної інформації. Головна сторінка часто використовується для висвітлення актуальних прем'єр та спеціальних пропозицій, в той час як окремі розділи сайту містять детальну інформацію про розклад сеансів, описи фільмів, відгуки, ціни на квитки та доступні послуги (наприклад, бронювання залів для приватних подій).

Функціональність продажу квитків є ключовою для кожного веб-сайту кінотеатру, дозволяючи користувачам вибрати фільм, сеанс, місце в залі та оплатити квитки онлайн без необхідності стояти в черзі. Ця система часто інтегрована з CRM-системами кінотеатру для кращого управління клієнтською базою та програмами лояльності.

Додатково, сучасні веб-сайти кінотеатрів включають розділи для новин та блогів, де публікуються огляди фільмів, анонси подій і інша корисна інформація, яка сприяє залученню відвідувачів та підтримці інтересу до кінотеатру. Інтеграція з соціальними медіа та можливість поділитися враженнями у соцмережах роблять сайт ефективним інструментом маркетингу та спілкування з аудиторією.

З точки зору технічного втілення, веб-сайти кінотеатрів мають бути оптимізовані для різних пристроїв, від настільних комп'ютерів до смартфонів та планшетів, забезпечуючи зручний перегляд та використання незалежно від пристрою користувача. Висока швидкість завантаження, захист даних користувачів та інтеграція з різноманітними платіжними системами також є критично важливими для забезпечення високого рівня задоволення користувачів і безпеки.

1.2 Загальний опис веб-сайту

Веб-сайт кінотеатру планується розробити з метою забезпечення зручності та ефективності як для відвідувачів, так і для персоналу. Він складатиметься з двох основних секцій: інтерфейс для відвідувачів та адміністративний розділ для персоналу.

Частина для відвідувачів веб-сайту кінотеатру буде ретельно розроблена, щоб забезпечити максимально зручний та інтуїтивно зрозумілий інтерфейс для користувачів. Вона дозволить відвідувачам переглядати доступні сеанси в реальному часі, обирати місця у кінозалі та придбавати квитки онлайн із мінімальними зусиллями.

У майбутньому відвідувачі зможуть легко навігувати по сайту, використовуючи різноманітні фільтри для пошуку фільмів за жанром, датою показу, рейтингом, а також шукати спеціальні пропозиції та знижки. Ця система допоможе користувачам швидко знайти найбільш відповідний для них контент, зекономивши час та підвищивши задоволення від користування сайтом.

Придбання квитка стане простим і зрозумілим процесом, завдяки інтеграції безпечної платіжної системи Stripe. Після вибору сеансу та місць у кінозалі, відвідувачам буде запропоновано пройти швидку процедуру оплати, яка підтримуватиме всі основні платіжні системи, включаючи кредитні картки та електронні гаманці. Захист персональних даних та фінансових транзакцій гарантуватиметься за допомогою сучасних технологій шифрування.

Особистий кабінет відвідувача стане централізованим місцем для управління їхніми квитками, історією покупок та персональними налаштуваннями. Користувачі зможуть переглядати деталі своїх майбутніх сеансів, перевіряти статус квитків, а також легко скасовувати бронювання.

Частина веб-сайту, розроблена для персоналу кінотеатру, буде включати набір інструментів для ефективного управління контентом та ресурсами кінотеатру. Основою цієї системи стане зручна адміністративна панель, яка дозволить персоналу легко додавати нові фільми, керувати сеансами та залами, а також налаштовувати ціни на квитки.

Для додавання нових фільмів персонал зможе використовувати просту форму, заповнюючи необхідну інформацію вручну, або ж скористатися інтеграцією зовнішнього сервісу для швидкого додавання. Також цей сервіс дозволить виконувати пошук за ключовими словами та отримувати інформацію про популярні новинки.

Керування сеансами передбачає можливість налаштовувати час та дати показу, вибирати зал для сеансу та встановлювати ціни на квитки з урахуванням різних факторів, таких як час сеансу або попит на конкретний фільм. Адміністративний інтерфейс дозволить також легко керувати конфігурацією залів, включаючи розташування місць.

Сторінка продажу квитків офлайн є ключовим компонентом системи для персоналу кінотеатру, що дозволяє ефективно управляти продажами квитків безпосередньо з адміністративної панелі. Цей інструмент спеціально розроблений для спрощення процесу продажу квитків співробітниками кінотеатру, які працюють безпосередньо з відвідувачами. На сторінці продажу квитків офлайн персонал зможе швидко вибирати сеанси з актуального розкладу, переглядати доступні місця на інтерактивному плані залу та проводити продаж квитків.

Окрім того, система надасть можливість керування доступом персоналу до різних функцій адміністративної панелі. Це дозволить встановити рівні

доступу в залежності від ролі співробітника у кінотеатрі, забезпечуючи таким чином безпеку даних та ефективність управління ресурсами.

Ця частина веб-сайту покликана забезпечити персонал кінотеатру максимально зручними та ефективними інструментами для управління всіма аспектами діяльності кінотеатру, підвищуючи якість обслуговування та забезпечуючи гладке функціонування усіх процесів.

1.3 Огляд та виявлення недоліків існуючих аналогів

Під час аналізу існуючих веб-сайтів кінотеатрів було проведено огляд різноманітних платформ і виявлено декілька ключових недоліків, які часто зустрічаються та впливають на загальне задоволення користувачів і ефективність використання сайтів.

Деякі веб-сайти кінотеатрів мають складний або не інтуїтивний інтерфейс, що ускладнює процес знаходження інформації про сеанси, вибору місць та придбання квитків. Важливі елементи інтерфейсу часто замасковані або занадто засмічені, що спонукає користувачів залишати сайт без здійснення покупки (рис. 1.1).

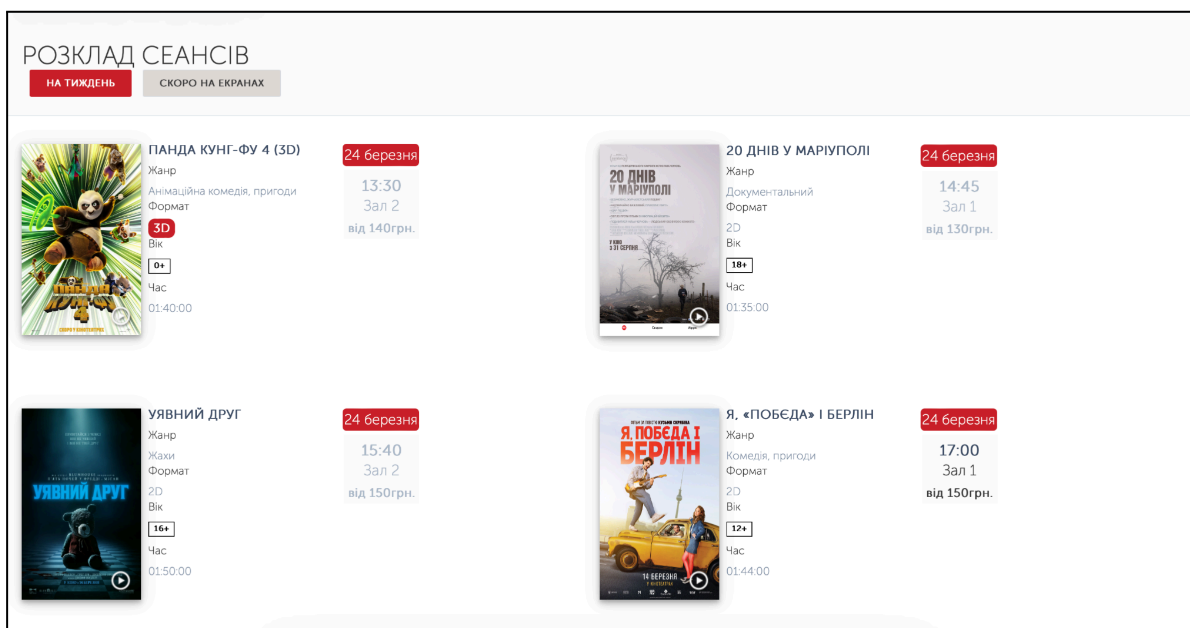


Рисунок 1.1 – Сторінка розкладу кінотеатру “Галичина”

Як можна помітити з рисунку 1.1 веб-сайт має простий але не інтуїтивний інтерфейс. При наведенні курсора на заголовок фільму він змінює колір, але при цьому не є клікабельним. Не зразу зрозуміло як перейти до придбання квитка. Дуже мало інформації про фільм, а та інформація яка надається є не зручною для читання. Наприклад, текст зливається з фоном, великі відступи між елементами (жанр, формат, вік, час) і їх значеннями, та відсутність відступу після значення. Постери фільму в правому нижньому куті мають знак програвання, припускаю що це було зроблено для перегляду трейлеру, але при натисканні відкривається пусте модульне вікно. Ще одним недоліком є те, що веб-сайт показує сеанси, які вже розпочались або закінчились, забираючи у користувачів час на перегляд неактуальної інформації [1].

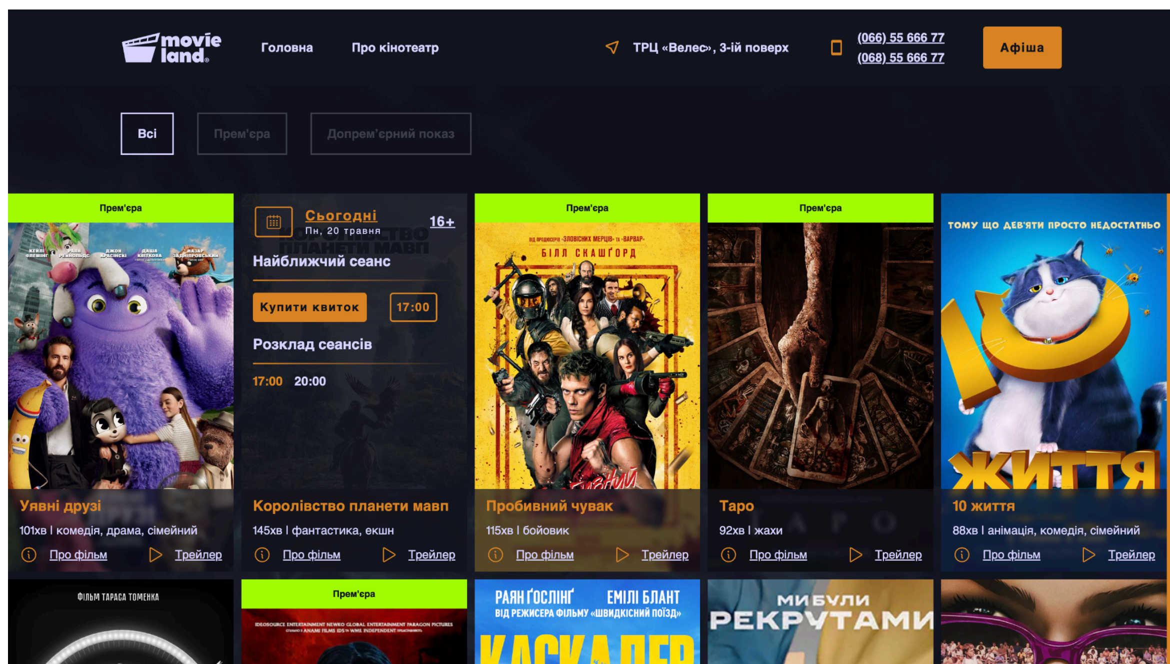


Рисунок 1.2 – Сторінка розкладу кінотеатру “Movieland”

Наступний кінотеатр, який я хотів би розглянути це Movieland. Серед переваг MovieLand приваблює представленням інформації про фільми, сеанси, та навіть можливістю переглянути трейлер. Також, можливість попереднього перегляду схеми залу та вибору місць робить процес покупки квитків онлайн ще більш зручним. Зверніть увагу на рисунок 1.2, на перший погляд інтерфес

вибору сеансів інтуітивно зрозумілий, але це до поки ми не захочемо забронювати квиток на любий інший день, крім сьогоднішнього. Щоб це зробити потрібно нажати на “Сьогодні”, після чого з’явиться дропдавн меню, де ми зможемо обрати потрібний день [2].

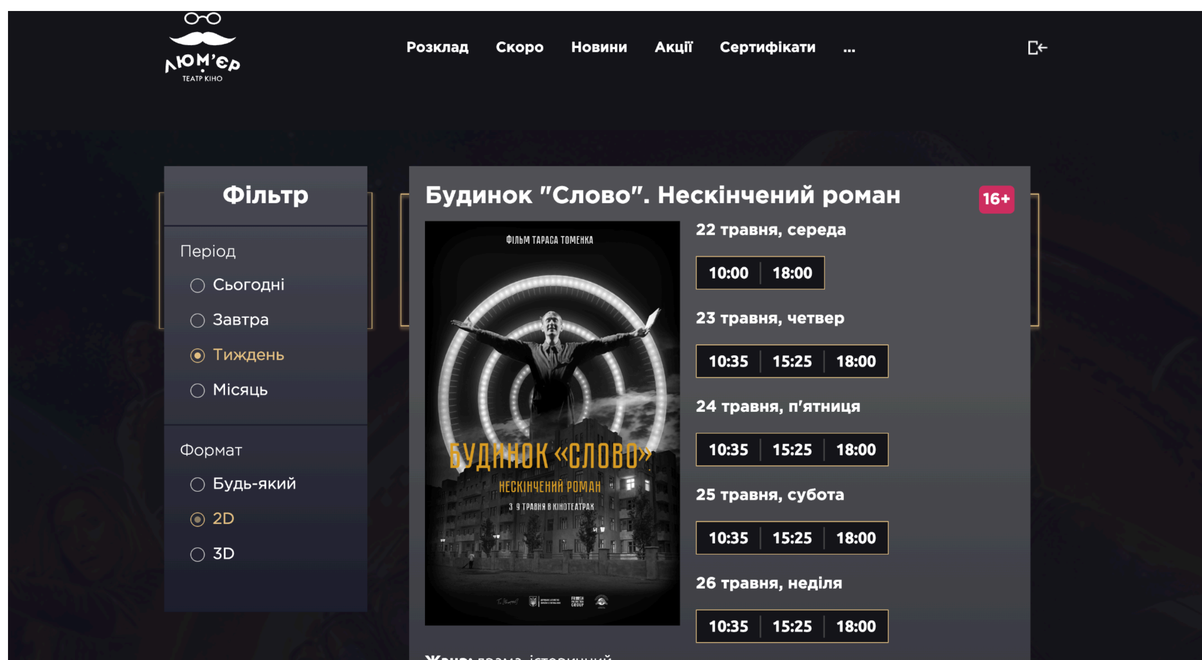


Рисунок 1.3 – Сторінка розкладу сеансів кінотеатру “Люм’єр”

Сайт кінотеатру “Люм’єр” зручний тим, що пропонує чітку і зрозумілу структуру з розділами для афіші, новин, акцій та сертифікатів. Можна легко переглянути розклад сеансів, купити квитки онлайн, дізнатися про майбутні прем'єри та акційні пропозиції. Окрім зручної навігації та можливості купівлі квитків онлайн, можна відмітити можливість фільтрації сеансів за періодом показу та формату (рис. 1.3). Серед недоліків, які можуть бути не зручними, є перегляд доступних місць у залі, для цього потрібна авторизація [3].

На деяких сайтах відсутній або обмежений функціонал для онлайн-бронювання квитків. Це змушує відвідувачів здійснювати покупки через сторонні сервіси або приходити в кінотеатр заздалегідь, що знижує загальну привабливість сервісу.

Незважаючи на зростаючу популярність використання смартфонів для відвідування веб-сайтів, деякі платформи кінотеатрів все ще не оптимізовані для мобільних пристроїв. Це створює незручності для користувачів, які переважно використовують телефони для веб-серфінгу.

Системи управління вмістом для кінотеатрів часто мають обмежену функціональність або складні в управлінні, що не дозволяє персоналу кінотеатру ефективно управляти контентом, сеансами, залами, а також швидко реагувати на зміни в розкладі.

1.4 Постановка задачі

Основна задача кваліфікаційної роботи полягає у розробці веб-орієнтованої системи керування діяльністю кінотеатру за допомогою мови програмування Ruby та фреймворку Ruby on Rails, що включає створення ефективного інструменту для управління контентом, сеансами, залами та квитками. Система повинна надавати зручний інтерфейс для відвідувачів для перегляду розкладу сеансів та покупки квитків онлайн, а також функціональну адміністративну панель для персоналу кінотеатру для керування кінолінійкою, розкладом сеансів, залами та ціноутворенням. Таким чином, робота спрямована на покращення якості обслуговування відвідувачів кінотеатру та оптимізацію робочих процесів персоналу.

Для досягнення поставленої мети у кваліфікаційній роботі необхідно вирішити наступні задачі:

- визначення вимог до системи. Оцінка потреб користувачів та персоналу кінотеатру для формування функціональних і технічних вимог;
- проєктування архітектури системи. Розробка структури бази даних, схеми взаємодії компонентів системи, інтерфейсу користувача та адміністративної панелі;

– програмування веб-орієнтованої системи керування кінотеатром на мові Ruby on Rails, реалізація клієнтської частини (frontend) та серверної частини (backend);

– інтеграція з зовнішніми сервісами. Підключення системи до зовнішніх API для розширення функціоналу, наприклад, інтеграція з TMDB для отримання інформації про фільми та інтеграція платіжної системи для обробки онлайн-платежів;

– тестування системи, виявлення та усунення помилок, перевірка відповідності реалізованої системи встановленим вимогам;

– впровадження та налаштування системи. Деплоймент системи на сервері, налаштування робочого середовища.

Висновки до розділу 1

У першому розділі було проведено детальний огляд предметної області, зокрема, аналіз веб-сайтів кінотеатрів, що дозволило виявити поточні тенденції та ключові функціональні можливості. Було здійснено загальний опис веб-сайту кінотеатру, визначено основні вимоги та очікування користувачів. Огляд існуючих аналогів допоміг ідентифікувати їхні недоліки. На основі цього аналізу було сформульовано чітку постановку задачі для розробки нового веб-додатку, що дозволить усунути виявлені недоліки та забезпечити покращений користувацький досвід.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА ПЛАН РОЗРОБКИ ДОДАТКУ

2.1 Написання User Stories

User stories, або історії користувачів, є основним елементом розробки, який допомагає командам зрозуміти потреби та бажання кінцевих користувачів. Це короткі, описові заяви, що викладають функціональні вимоги з перспективи користувача, описуючи, що користувач хоче зробити, використовуючи продукт, і чому це важливо для нього. Використання user stories дозволяє розробникам та дизайнерам фокусуватися на створенні цінності для користувача, забезпечує краще розуміння потреб користувача і сприяє гнучкості у процесі розробки.

WHO are we building it for? Who is the user?	As a <type of user>
WHAT are we building? What is the intention?	I want <some goal or objective>
WHY are we building it? What is the value for the customer?	So that <benefit/value>

Рисунок. 2.1 – Основні компоненти User Story

Кожна історія користувача включає три основні компоненти: опис того, хто є користувачем (роль), що користувач хоче досягти (функція), і для чого це йому потрібно (цінність) (рис. 2.1). Це допомагає команді підтримувати зосередженість на користувачі під час усього процесу розробки та гарантувати, що кінцевий продукт відповідатиме реальним потребам користувачів. Використання user stories також спрощує планування, оцінку та пріоритизацію роботи, роблячи процес розробки більш адаптивним і відповідним до змін у вимогах чи очікуваннях користувачів.

Щоб покрити основні потреби користувачів і персоналу кінотеатру в контексті веб-сайту, від простого перегляду афіші до комплексного управління кінотеатральними процесами, я написав кілька User Stories:

1. Як користувач, я хочу переглядати актуальну афішу кінотеатру, щоб дізнатися про поточні фільми та сеанси.
2. Як користувач, я хочу мати можливість читати описи фільмів та дивитися трейлери, щоб вирішити, чи хочу я відвідати сеанс.
3. Як зареєстрований користувач, я хочу купувати квитки онлайн, щоб забезпечити собі місце на сеанс без необхідності відвідування каси кінотеатру.
4. Як користувач, я хочу знати ціну квитка до початку процесу бронювання, щоб я міг врахувати свій бюджет і прийняти обґрунтоване рішення про покупку без несподіваних витрат.
5. Як зареєстрований користувач, я хочу переглядати історію моїх покупок квитків, щоб мати змогу згадати інформацію про відвідані сеанси.
6. Як керівник кінотеатру, я хочу керувати доступом персоналу до різних функцій адміністративної панелі, щоб забезпечити виконання конкретних завдань без ризику втрати даних або зловживань.
7. Як співробітник кінотеатру, я хочу додавати нові фільми до афіші, щоб утримувати актуальний репертуар.
8. Як співробітник кінотеатру, я хочу керувати розкладом сеансів, щоб забезпечити ефективне використання залів, враховуючи попит відвідувачів.
9. Як співробітник кінотеатру, я хочу оновлювати ціни на квитки, щоб оптимізувати прибутковість кінотеатру.
10. Як співробітник кінотеатру, я хочу обробляти покупки квитків, здійснені в касі, щоб видавати квитки відвідувачам і забезпечувати актуальне відображення доступних місць у реальному часі.
11. Як співробітник кінотеатру, я хочу налаштовувати конфігурацію залів, щоб правильно відображати місця в залі для відвідувачів.

2.2 Особливості Rails та вибір інструментів для розробки

Ruby on Rails є одним з найпопулярніших фреймворків для розробки веб-додатків на мові програмування Ruby. Його популярність забезпечена цілою низкою особливостей, що спрощують і прискорюють процес розробки, а також забезпечують високу якість кінцевого продукту.

Однією з найважливіших особливостей Rails є принцип "Convention over Configuration" (Конвенція над конфігурацією). Цей принцип означає, що розробникам не потрібно витрачати час на детальну конфігурацію кожного аспекта додатку, якщо вони слідують певним загальноприйнятим конвенціям. Це спрощує процес розробки та зменшує кількість потенційних помилок.

Також важливим є наявність вбудованої підтримки для розробки відповідно до архітектурного патерну Model-View-Controller (MVC), що допомагає організувати код додатку, розділивши його на три основні компоненти: моделі (бізнес-логіка і взаємодія з базою даних), контролери (логіка обробки запитів) та відображення (інтерфейс користувача).

Безпека додатку є ще одним критичним аспектом, на який потрібно звернути увагу. Rails надає багато вбудованих механізмів безпеки, таких як захист від CSRF-атак (Cross-Site Request Forgery), SQL-ін'єкцій та інших поширених вразливостей. Проте, важливо регулярно оновлювати залежності та слідувати кращим практикам безпеки, щоб забезпечити захист даних користувачів та системи в цілому [4].

В свій веб-сайт я хочу інтегрувати зовнішній сервіс для автоматизації процесу оновлення контенту. Онлайн бази даних фільмів, такі як The Movie Database (TMDb), IMDb (Internet Movie Database), або Rotten Tomatoes надають обширні дані про фільми, серіали, акторів, режисерів, сценаристів, а також відгуки користувачів і критиків. Ці бази даних стали невід'ємною частиною сучасної культури, оскільки вони забезпечують швидкий доступ до інформації про кіно, що спрощує пошук і вибір фільмів для перегляду. Багато з цих онлайн

баз даних пропонують API, що дозволяє розробникам веб-сайтів, мобільних додатків або інших сервісів інтегрувати дані про фільми прямо у свої продукти.

Я зупинився на виборі TMDb, оскільки він пропонує відносно легкий у використанні API, що дозволяє розробникам інтегрувати дані у свої додатки та має ряд переваг над своїми аналогами. Серед них можна виділити те що це одна з найбільших баз даних фільмів та телешоу, що охоплює велику кількість інформації, включаючи деталі про виробництво, акторський склад, відгуки тощо. Має активну спільноту користувачів, що гарантує швидке оновлення інформації та надає широкі можливості для виконання запитів, дозволяючи отримувати деталізовану інформацію за конкретними критеріями пошуку.

Ця інтеграція дозволяє виконувати пошук фільмів використовуючи ключові слова, переглядати перелік фільмів, що наразі користуються популярністю в інших кінотеатрах, а також ознайомитися з новинками, які незабаром з'являться в прокаті, та додавати їх в свою колекцію одним кліком, замість того щоб вручну заповнювати дані, що значно спрощує процес оновлення асортименту кінопоказів [5].

Інтеграція платіжних систем у Rails додаток є важливою частиною розробки сучасних веб-застосунків, які потребують обробки платежів. Є кілька популярних платіжних провайдерів, які можна інтегрувати з Rails, включаючи Stripe, PayPal, Braintree, Square та інші. Кожен з них пропонує свої переваги та особливості, що можуть впливати на вибір провайдера в залежності від конкретних потреб проєкту.

Зупинка на Stripe для інтеграції у свій проєкт зумовлена кількома ключовими факторами. По-перше, Stripe пропонує простий і зрозумілий процес інтеграції, з великою кількістю прикладів і документації, що значно скорочує час на інтеграцію. По-друге, Stripe забезпечує широким набором функцій, включаючи підтримку різних платіжних методів, підписок і навіть інвойсів, що робить його універсальним рішенням для різних типів бізнесу. По-третє, високий рівень безпеки та автоматична обробка вимог PCI DSS знімає значний обсяг роботи з розробників щодо відповідності стандартам безпеки. Нарешті,

хороша підтримка та постійні оновлення від Stripe роблять його надійним вибором для довгострокового використання [6].

Rails залишається відмінним вибором для розробки веб-орієнтованої системи керування діяльністю кінотеатру обумовлюється його гнучкістю, швидкістю розробки, а також великою активною спільнотою, яка може надати велику кількість готових рішень у вигляді гемів, що можуть значно пришвидшити розробку.

2.3 Розгортання проєкту у виробниче середовище

Розгортання Rails додатка у виробниче середовище є критичним етапом розробки, який включає кілька важливих кроків для забезпечення безперебійної роботи, безпеки та масштабованості. Виробниче середовище суттєво відрізняється від середовища розробки або тестування тим, що вимагає більш високих стандартів надійності та продуктивності, оскільки додаток буде взаємодіяти з реальними користувачами.

Першим кроком у розгортанні є вибір хостинг-провайдера. Популярні варіанти включають Heroku, AWS (Amazon Web Services), DigitalOcean, та інші. Heroku є популярним вибором для Rails додатків завдяки своїй простоті у використанні та підтримці автоматизованих розгортань. AWS надає більшу гнучкість і масштабованість, але вимагає більш детального налаштування. DigitalOcean пропонує баланс між простотою та контролем, дозволяючи розгортати додатки на віртуальних приватних серверах (VPS).

Наступним кроком є налаштування середовища додатку. Це включає конфігурацію бази даних, веб-сервера та інших необхідних сервісів. У виробничому середовищі зазвичай використовуються реляційні бази даних, такі як PostgreSQL або MySQL, замість SQLite, яка часто використовується в середовищі розробки. Веб-сервери, такі як Puma або Unicorn, використовуються для обробки HTTP-запитів, і можуть бути розгорнуті за допомогою Nginx або Apache як зворотного проксі-сервера.

Безпека є ключовим аспектом виробничого середовища. Це включає налаштування SSL для шифрування трафіку, забезпечення захисту від атак CSRF і XSS, а також регулярне оновлення залежностей для усунення відомих вразливостей. Використання інструментів на зразок Let's Encrypt для автоматичного отримання та оновлення SSL сертифікатів може значно спростити цей процес.

Для розгортання свого веб-додатку було обрано сервіс Heroku. Однією з головних переваг Heroku є можливість швидкого розгортання додатку без потреби в складних налаштуваннях інфраструктури. Використовуючи просту інтеграцію з Git, розробники можуть розгорнути свій додаток за допомогою кількох команд у терміналі, таких як git push heroku master. Heroku автоматично обробляє інсталяцію залежностей, компіляцію активів і запуск серверу, що значно скорочує час на розгортання і дозволяє зосередитися на написанні коду.

Додатковою перевагою є масштабованість платформи. Heroku дозволяє легко адаптувати додаток до зростання трафіку завдяки можливості швидкого збільшення кількості динамо-екземплярів (dynos) для обробки більшої кількості запитів. Це означає, що ви можете налаштувати ваш додаток для автоматичного масштабування в залежності від навантаження, забезпечуючи стабільну роботу навіть під час пікових навантажень. Крім того, інтеграція з різноманітними аддонами, такими як бази даних, кешування і моніторинг, дозволяє легко додавати необхідні сервіси для підтримки додатку.

Heroku також забезпечує високий рівень безпеки і надійності, що є критично важливим для будь-якого веб-додатку. Всі з'єднання до додатків на Heroku здійснюються через HTTPS, що забезпечує шифрування даних в транзиті. Автоматичні оновлення безпеки і регулярні резервні копії баз даних допомагають підтримувати високу надійність і швидке відновлення у разі непередбачених проблем [7].

Завдяки цим перевагам Heroku стає ідеальним вибором для розробників, які хочуть швидко розгортати, масштабувати і підтримувати свої Rails додатки з мінімальними зусиллями.

Висновки до розділу 2

У другому розділі було висвітлено основні етапи та аспекти проєктування і планування розробки додатку. Спочатку детально розглянуто процес написання User Stories, що дозволяє чітко визначити вимоги до функціональності додатку з точки зору користувача. Цей підхід сприяє кращому розумінню очікувань кінцевих користувачів та забезпечує більш структурований підхід до розробки. Було обговорено особливості фреймворку Rails та обґрунтовано вибір інструментів для розробки, що оптимально підходять для реалізації запланованих функцій додатку. Також було проаналізовано переваги використання Rails. Особливу увагу було приділено розгортанню додатку у виробниче середовище. Включаючи вибір хостинг-провайдера, налаштування середовища додатку, забезпечення безпеки та масштабованості.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ

3.1 Ініціалізація та налаштування проєкту

Для реалізації проєкту буде використовуватись остання версію Ruby on Rails. Для її встановлення потрібно скористатися командою:

```
gem install rails -v 7.1.1
```

Для створення типового Rails додатку зі стандартними параметрами та компонентами використовується команда rails new. Однак, якщо ми заздалегідь знаємо, які конкретні налаштування та компоненти нам потрібні для нашого проєкту, ми можемо скористатися опціями команди. Це дозволяє нам налаштувати додаток відразу під наші потреби, зекономивши час та забезпечивши гнучкість розробки з самого початку процесу. Тому для створення додатку я додам кілька опцій до команди:

– T – ця опція пропускає створення файлів стандартного фреймворку для тестування (заснованого на Minitest), замість нього ми налаштуємо RSpec, який надає більшу гнучкість і читабельність при написанні тестів;

– d postgresql – підключає бібліотеку і генерує потрібні файли конфігурації для використання PostgreSQL, замість стандартної SQLite;

– j esbuild – ця опція вказує, що в проєкті використовуватиметься esbuild для компіляції JavaScript файлів. esbuild – це швидкий компілятор JavaScript і може бути використаний для оптимізації та збільшення продуктивності веб-сторінок [8];

– css bootstrap – вказує, що веб-додаток буде використовувати фреймворк стилів Bootstrap для оформлення сторінок та елементів інтерфейсу. Bootstrap – це популярний фреймворк фронтенд-розробки, який надає набір готових стилів, компонентів та JavaScript плагінів для швидкого створення сучасних веб-сайтів [9].

Після ініціалізації проєкту додаємо кілька бібліотек, для цього відкриваємо Gemfile. Gemfile у Ruby on Rails є ключовим файлом для управління залежностями веб-додатку. Цей файл розташований в кореневій папці проєкту та містить перелік гемів (бібліотек і розширень), які використовуються в додатку. У Gemfile ви вказуєте геми, що необхідно встановити, разом з їхніми версіями. Rails рекомендує використовувати цей файл для задання гемів, таких як rails для самого фреймворку, а також для додаткових бібліотек, які використовуються в різних середовищах: development, test та production. Коли ви редагуєте Gemfile, ви можете встановити або оновити геми за допомогою команди bundle install, що забезпечує однорідність середовища розробки та допомагає уникнути конфліктів версій. Після встановлення гемів, Rails також генерує файл Gemfile.lock, який фіксує конкретні версії гемів, що встановлені в проєкті, забезпечуючи стабільність та портативність проєкту між різними середовищами розробки. Для development і test середовища було додано ці бібліотеки:

```
gem 'pry', '~> 0.14.2'
gem 'faker', '~> 3.2', '>= 3.2.1'
gem 'rubocop', '~> 1.57', '>= 1.57.2'
gem 'rubocop-rails', '~> 2.21', '>= 2.21.2'
gem 'rubocop-rspec', '~> 2.24', '>= 2.24.1'
```

Pry – це інтерактивний дебагер та консольний інтерпретатор. Він дозволяє взаємодіяти з кодом в режимі реального часу, вставляти точки зупинки та досліджувати змінні під час виконання програми. Pry розширює стандартні можливості дебагера Ruby, надаючи більше можливостей та зручностей.

Faker – це генератор випадкових даних, який корисний під час розробки та тестування, особливо при створенні фейкових записів в базі даних для тестування функціональності додатка. Зазвичай цю бібліотеку додають в тестове середовище, але в деяких випадках і для середовища розробки, щоб наповнити вміст сайту випадковими даними через seeds файл [10].

RuboCop – це інструмент для автоматичного аналізу та виправлення коду, який допомагає дотримуватися стандартів коду та знижує кількість можливих помилок під час розробки. Також було додано два розширення до цього гему:

- RuboCopRails – розширює RuboCop специфічно для Ruby on Rails проєктів, надаючи правила та стилі для відповідності стандартам Rails
- RuboCopRSpec – це розширення для аналізу та виправлення коду, який відповідає стандартам RSpec [11].

Для test середовища було додано:

```
gem 'rspec-rails', '~> 6.0', '>= 6.0.3'
gem 'factory_bot_rails', '~> 6.2'
gem 'capybara', '~> 2.7', '>= 2.7.1'
gem 'database_cleaner', '~> 2.0.1'
```

Замість стандартного фреймворку Minitest, генерацію якого було відключено при створенні проєкту, буде використовуватись Rspec. RSpec є дуже популярним інструментом для тестування і розглядається як один з найкращих фреймворків для тестування в Ruby-екосистемі. Він надає зручний та експресивний спосіб писати тести, які допомагають вам перевіряти функціональність вашого коду та переконатися, що він працює правильно [12].

FactoryBotRails допомагає спростити процес створення тестових даних та об'єктів для тестів, забезпечуючи зручність та ефективність [13].

Capybara – це бібліотека для тестування веб-додатків у Ruby. Вона дозволяє автоматизувати імітацію взаємодії користувача з веб-додатком у тестах і перевіряти очікуваний результат. Capybara надає простий та експресивний інтерфейс для виконання дій, таких як заповнення форм, натискання на посилання, навігація між сторінками і багато інших [14].

DatabaseCleaner допомагає видаляти дані з бази даних між тестами, це забезпечує чистоту та ізольованість тестового середовища, сприяючи надійному виконанню тестів та уникненню конфліктів даних.

Після додавання бібліотек до файлу Gemfile, необхідно виконати команду bundle install. Ця команда встановлює всі зазначені бібліотеки та їх залежності. Деякі геми, які я додав, працюватимуть зразу без додаткового налаштування. Однак, для гемів, таких як rubocop, rspec, factory_bot, capybara і database_cleaner, необхідно провести додаткову конфігурацію.

Якщо в проєкті відсутній конфігураційний файл для RuboCop, то застосовуватимуться значення за замовчуванням, і він буде перевіряти ваш код на основі стандартних правил. Однак можна створити власний конфігураційний файл .rubocop.yml в кореневій директорії проєкту та налаштувати правила RuboCop згідно з потребами.

Я Вимкнув перевірку довжини методів та блоків коду, встановивши параметр Enabled для правил Metrics/MethodLength та Metrics/BlockLength у значення false. Це дозволяє більшу гнучкість у написанні функцій та блоків коду без обмежень за їхньою довжиною. Обмежено максимальну довжину рядка коду до 140 символів для правила Layout/LineLength. Це допомагає зробити код більш читабельним і відповідає стандартам стилю. Вказав шляхи до файлів і директорій, які повинні бути виключені з перевірок RuboCop, включаючи файли db/schema.rb, файли в директорії config та файли в директорії bin. Це дозволяє уникнути непотрібних перевірок у важливих частинах проєкту. Вимкнув обов'язковий коментар щодо використання "заморожених" рядків у коді, встановивши EnforcedStyle для правила Style/FrozenStringLiteralComment в значення never. Вимкнув правила, пов'язані з документацією коду, шляхом встановлення параметра Enabled для рубрики Documentation в значення false.

Переходимо до налаштування бібліотек для тестів, для цього генеруємо файли для rspec виконавши команду rails generate rspec:install. Тепер в відкриваємо файл spec/rails_helper.rb який щойно згенерувався і підключемо capybara додавши рядок require 'capybara/rspec' на початку файлу. Для підключення і налаштування factory_bot і database_cleaner потрібно в цьому ж файлі додати цей код:

```
RSpec.configure do |config|
    config.include FactoryBot::Syntax::Methods
end
RSpec.configure do |config|
    config.before(:suite) do
        DatabaseCleaner.clean_with(:truncation)
    end
    config.before(:each) do
        DatabaseCleaner.strategy = :transaction
    end
    config.before(:each, type: :feature) do
        DatabaseCleaner.strategy = :truncation
    end
    config.before(:each) do
        DatabaseCleaner.start
    end
    config.append_after(:each) do
        DatabaseCleaner.clean
    end
end
```

Для створення бази даних потрібно ввести команду rails database:create, ця команда створить одразу дві бази даних для тестового та розробницького середовища. Запускаємо сервер командою rails s, за замовчуванням rails додатки використовують порт 3000 для запуску розробницького сервера. Тому перевіряємо чи наш додаток доступний та чи немає ніяких помилок за адресою http://localhost:3000 у веб-браузері.

Загалом, з допомогою встановлених гемів і налаштувань було створено добре налаштований та готовий до розробки веб-додаток, який відповідає моїм потребам. Це дозволяє мені зосередитися на розробці функціональності мого веб-сайту, знаючи, що основні аспекти конфігурації та робочого середовища вже налаштовані та готові до використання.

3.2 Створення моделей Movie, Hall та Session

Моделі в Ruby on Rails є фундаментальним елементом в розробці веб-додатків, і вони відіграють ключову роль у взаємодії з базою даних. Модель представляє собою клас, який описує об'єкти та їх властивості, і вона визначає структуру даних, які зберігаються в базі даних.

Однією з ключових переваг моделей в Rails є можливість використовувати мову програмування Ruby для взаємодії з базою даних, замість SQL-запитів. Це спрощує роботу розробників і дозволяє писати більш читабельний і зрозумілий код.

Моделі також сприяють забезпеченню безпеки даних та перевірці на валідність, виконуючи правила, встановлені розробником. Це допомагає уникнути некоректних або шкідливих даних у базі, забезпечуючи цілісність і надійність інформації.

Один з інших важливих аспектів моделей в Rails – це зв'язки між ними. Реляційні бази даних дозволяють створювати складні структури, де об'єкти моделей пов'язані між собою, що дозволяє реалізувати різноманітні функціональні можливості, такі як звіти, пошук, фільтрація тощо.

У підсумку, моделі в Ruby on Rails є основою для створення потужних та надійних веб-додатків, і правильне визначення моделей є важливим завданням для будь-якого розробника. Вони дозволяють працювати з даними більш ефективно та допомагають забезпечити якість та безпеку вашого додатку.

Для реалізації мого проєкту мені потрібні такі моделі Movie, Session, Hall. Щоб їх згенерувати в Ruby on Rails можна використати команду rails generate model movie, яка створить файл моделі та міграції. Міграції – це спосіб управління структурою бази даних в Rails, і вони дозволяють змінювати схему бази даних відповідно до розвитку додатку. В кожній тільки що створеній міграції ми повинні написати поля та їхні типи даних які потрібно додати до відповідної таблиці. Наприклад для своєї моделі Movie я додам поля title, description, genre, duration, directors, actors, countries, release_date.

```
class CreateMovies < ActiveRecord::Migration[7.1]
 def change
   create_table :movies do |t|
     t.string :title
     t.string :description
     t.string :genre, array: true, default: []
     t.integer :duration
     t.string :directors, array: true, default: []
     t.string :actors, array: true, default: []
     t.string :countries, array: true, default: []
     t.date :release_date
     t.timestamps
   end
 end
end
```

В кінці міграції є рядок з кодом t.timestamps, цей рядок створюється при генерації міграції. В цьому прикладі, t.timestamps додає два поля: created_at та updated_at до таблиці "movies". Коли створюється новий запис в цій таблиці, Rails автоматично встановлює значення для created_at і updated_at на поточний час. Коли запис змінюється, Rails оновлює значення в полі updated_at. Використання timestamps дозволяє легко відстежувати час створення та оновлення об'єктів у базі даних, що може бути корисно для реалізації логіки в системі або для відслідковування активності в додатку.

Таблиця залів має поле name типу string і поле scheme типу jsonb для зберігання схеми залу. Тип стовпця jsonb в базі даних PostgreSQL, призначений для зберігання даних у форматі JSON. Використовується для зберігання неструктурованих або змінюваних даних, де ключі можуть бути різними для кожного запису. Цей тип стовпця дозволяє ефективно здійснювати операції пошуку та фільтрації даних в форматі JSON, а також розширює можливості моделей в Rails для роботи зі структурованими даними у вигляді хешів або масивів JSON [15].

В міграцію для сеансів було додано поле start_datetime типу datetime для зберігання дати та часу початку сеансу. Поле seats_data типу JSON яке буде

зберігати інформацію про вільні і заброньовані місця. Поле price типу float для зберігання ціни квитка. Також додано два поля для встановлення зв'язку між таблицями movies і halls за допомогою зовнішнього ключа, прописавши:

```
t.references :movie
t.references :hall
```

це створить два стовпця movie_id та hall_id.

Кожний фільм повинен мати зображення, для цього буде використовуватись вбудована в Ruby on Rails бібліотека ActiveStorage. Ця бібліотека призначена для зберігання та керування файлами. Вона спрощує роботу з різними хмарними та локальними сховищами, надаючи зручний інтерфейс для завантаження, зберігання та отримання файлів. ActiveStorage підтримує інтеграцію з популярними хмарними сервісами, такими як Amazon S3, Google Cloud Storage та Microsoft Azure, що дозволяє розробникам легко масштабувати свої додатки.

Основною перевагою ActiveStorage є його тісна інтеграція з іншими частинами Rails, такими як Active Record. Це дозволяє легко додавати файли до моделей, а також забезпечує автоматичне створення необхідних асоціацій та міграцій бази даних. ActiveStorage також підтримує різноманітні типи файлів, включаючи зображення, відео, документи та інші, і надає вбудовані інструменти для обробки та перетворення файлів, такі як зміна розміру зображень або конвертація форматів.

Крім того, ActiveStorage забезпечує безпечне зберігання та доступ до файлів, підтримуючи такі функції, як підписані URL для тимчасового доступу до приватних файлів. Це дозволяє забезпечити конфіденційність та безпеку даних користувачів. Завдяки простоті використання та гнучкості, ActiveStorage є потужним інструментом для будь-якого додатка, який потребує роботу з файлами та мультимедійними даними.

Для того щоб ActiveStorage запрацював, потрібно виконати команду rails active_storage:install, яка згенерує міграцію для створення таблиць бази даних,

необхідних для зберігання прикріплених файлів. А в моделі Movie потрібно додати рядок has_one_attached :poster, який дозволить до кожного фільму прикріпити один постер [16].

Щоб виконати міграції та оновити базу даних згідно визначених змін потрібно виконати команду rails db:migrate.

Подивитись поточну структуру бази даних додатку можна в файлі schema.rb. Він включає інформацію про всі таблиці, стовпці, індекси та інші параметри бази даних. Коли запускаються міграції, Rails автоматично оновлює schema.rb, щоб відобразити нові зміни в базі даних. Цей файл є частиною системи керування версіями схеми даних в Rails.

Файл schema.rb є корисним для того, щоб швидко отримати уявлення про структуру бази даних та порівняти її між різними версіями додатку. Також цей файл може бути використаний при перенесенні бази даних між різними середовищами, такими як робочий сервер та локальна розробка.

Валідації це спосіб перевіряти, чи відповідають дані в об'єктах моделей певним критеріям перед збереженням їх у базі даних. Вони допомагають забезпечити цілісність даних та уникнути збереження некоректних записів.

Класи моделей мають метод validates, який використовується для оголошення стандартних валідацій для атрибутів моделі, таких як перевірка наявності, довжини, формату, числового діапазону та інші. Для основних полів моделей було додано валідацію presence, щоб переконатися що об'єкт не буде створено або оновлено, якщо значення цих атрибутів будуть нульовим, порожнім рядком, або рядком який складається з пробілів.

Для валідації зображення в моделі фільмів, я використовую бібліотеку active_storage_validations, це додатковий інструмент для роботи з файлами в Rails додатках, який надає зручні методи перевірки наявності та типу завантажених файлів, що зберігаються за допомогою Active Storage. Це дозволяє забезпечити валідацію файлів і переконатися, що вони відповідають вимогам перед збереженням. Додавший цей рядок до моделі validates :poster, content_type: %i[png jpg jpeg], я перевіряю чи прикріплений файл відповідає

формату зображень PNG, JPG або JPEG. Якщо тип файлу не відповідає жодному з цих дозволених типів то валідація не пройде і файл не буде прикріплено [17].

Також в Ruby є метод validate, який використовується для встановлення власних методів валідації для об'єктів класу. Для моделі сеансів було створено власний метод валідації, який перед створенням об'єкта перевіряє чи зал в якому буде проводитись сеанс вільний у відповідний період, що дозволить уникнути накладок.

Для реалізації цієї логіки спочатку з бази даних достаються всі доступні сеанси, які відбуваються в тому ж залі де і новий сеанс. Потім за допомогою методу select, відбираються сеанси які потенційно конфліктують з новим сеансом. Для цього перевіряємо чи потрапляє початок сеансу в діапазон часу існуючих сеансів. Діапазон часу обчислюється як період від початку існуючого сеансу мінус тривалість фільму нового сеансу до початку плюс тривалість існуючого сеансу. Якщо виявлено будь-які конфліктуючі ситуації, додається помилка, що вказує на те, що зал недоступний у цей час.

```
validate :hall_available_during_time
def hall_available_during_time
     conflicting_sessions = Session.where(hall_id: hall_id).select {
|x| start_datetime.between?(x.start_datetime, x.start_datetime +
x.movie.duration.minutes) }
     errors.add(:base, 'The hall is not available during this time')
if conflicting_sessions.present?
end
```

Щоб уникнути створення не актуальних сеансів було додано ще один метод валідації який дозволяє створювати сеанси тільки в майбутньому. Цей метод валідації перевіряє чи дата і час нового сеансу знаходиться в майбутньому порівняно з поточним часом, якщо дата та час, вказані у полі start_datetime, менші або дорівнюють поточному часу, то метод додає

повідомлення про помилку з вказівкою на те, що дата та час мають бути в майбутньому.

```
validate :start_datetime_is_future

def start_datetime_is_future
  errors.add(:start_datetime, 'must be in the future') if
start_datetime <= DateTime.now
end
```

Життєвий цикл об'єктів моделі у Rails може бути поділений на різні етапи, такі як створення, оновлення, видалення тощо. Для кожного з цих етапів можна визначити колбеки, які будуть виконуватися автоматично на цих етапах. Колбеки в Ruby on Rails – це зручний спосіб автоматично викликати певний метод перед або після виконання певних подій у життєвому циклі об'єктів.

Щоб забезпечити що фільми і зали не можуть бути видалені допоки пов’язані з ними сеанси є доступними, було створено колбеки які викликаються перед видаленням.

```
before_destroy :have_sessions, prepend: true do
  throw(:abort) if errors.present?
end

def have_sessions
  errors.add(:base, 'The movie cannot be deleted until all sessions
have ended') if sessions.available.any?
end
```

У даному колбеці before_destroy з параметром prepend: true викликається метод have_sessions перед тим, як об'єкт буде видалено з бази даних. Параметр prepend: true вказує, що цей колбек має бути викликаний перед іншими колбеками, що можуть бути визначені для цієї ж події.

Сам колбек have_sessions перевіряє наявність активних сеансів для фільму. Якщо є будь-які активні сеанси, то до об'єкта моделі додається помилка

з повідомленням, яке говорить про те, що фільм не може бути видалений, поки є активні сесії. Якщо в об'єкті моделі є помилки, то операція видалення об'єкта переривається методом throw(:abort), і видалення не відбувається.

Аналогічний колбек був зроблений і для моделі залів.

3.3 Автентифікація та авторизація користувачів

Автентифікація – це процес перевірки ідентичності користувача з метою надання йому доступу до певних ресурсів чи функціоналу. Основною метою автентифікації є перевірка того, що особа, яка намагається отримати доступ, дійсно є тією, за кого вона себе видає. Використання ефективної системи автентифікації дозволяє забезпечити безпеку та конфіденційність у веб-додатках, а також керувати доступом користувачів до функціоналу відповідно до їхніх ролей та прав.

Для автентифікації користувачів буде використовуватись бібліотека Device. Вона надає готові рішення для багатьох аспектів автентифікації, таких як реєстрація, вхід, вихід, відновлення пароля та управління користувачами. Devise є однією з найпопулярніших бібліотек для автентифікації в екосистемі Ruby on Rails. Щоб використовувати Devise, спочатку необхідно додати його до Gemfile та встановити. Після цього слід виконати генерацію конфігурацій та міграцій для коректної роботи з базою даних [18].

До згенерованої міграції було додано два поля: role і permissions для визначення прав доступу кожного користувача. Наприклад, адміністратор повинен мати доступ до редагування фільмів та інших адміністративних функцій, касир повинен мати доступ до продажу квитків офлайн, тоді як звичайний користувач може тільки переглядати та купувати квитки.

В додатку буде 3 ролі user який не має доступу до панелі адміністратора, admin який має доступ до панелі адміністратора, але тільки до визначеного функціоналу в полі permissions, для кожного користувача він різний і superadmin

ця роль має повний доступ до всього функціоналу і визначає permissions для інших користувачів.

Організація атрибутів з певним обмеженим набором значень для поля role, зроблена за допомогою enum. Використання enum додає методи для зручної роботи з цими значеннями, автоматично перетворюючи їх між збереженими в базі даних цілочисельними індексами та символьними іменами в коді, що спрощує логіку перевірку ролей об'єктів. Також це дозволяє легко додавати валідацію, скоупи для запитів та використовувати допоміжні методи для перевірки або встановлення значень, забезпечуючи чистоту і ефективність коду при роботі з обмеженими наборами значень.

В панелі адміністратора, було створено сторінку staff, це і буде сторінка для додавання адміністраторів і розподілення їхніх прав. Ця сторінка доступна тільки для користувачів з роллю superadmin, тому посилання на неї не повинно відображається в навігаційній панелі інших користувачів. Тому для хедера додано перевірку

```
<% if current_user.superadmin? %>
  <li class="nav-item">
    <%= link_to 'Staff', admin_users_path, class: "nav-link #{'active'
if content_for(:active_tab) == 'staff'}" %>
  </li>
<% end %>
```

Цей код забезпечує відображення посилання на цю сторінку тільки для користувачів які мають доступ, але інші користувачі все ще можуть потрапити сюди ввівши url адресу вручну. Тому щоб перевірити доступу користувачів до певних функції в додатку буде використовуватись бібліотека Pundit.

Pundit – це легкий і гнучкий gem для Ruby on Rails, який надає прості засоби для реалізації авторизації. Він використовує шаблон проєктування політик (policy objects), який дозволяє розміщувати логіку авторизації в окремих класах, замість того, щоб інтегрувати її безпосередньо у контролерах або моделях. Кожна політика відповідає певній моделі та визначає правила

доступу до різних дій, пов'язаних з цією моделлю. Використання Pundit сприяє чистоті коду та його легкості для розуміння, оскільки вся логіка авторизації зосереджена в одному місці. Це дозволяє розробникам легко управляти правами доступу, забезпечує вищий рівень безпеки та підтримує чистоту і легкість підтримки коду. Його простий API інтегрується безпосередньо з контролерами Rails, що робить його зручним у використанні і дуже ефективним для управління складними вимогами авторизації [19].

Було створено базовий контроллер адміністратора, всі інші контролери які відносяться до панелі адміністратора наслідуються від нього. Це дозволяє централізувати спільні налаштування, фільтри та методи авторизації для всіх контролерів, які обслуговують адміністративний інтерфейс. Це спрощує управління кодом, підвищує його читабельність та забезпечує єдину точку контролю доступу і безпеки для адміністративної частини додатка, оскільки всі контролери панелі адміністратора наслідуються від цього базового контроллера, забезпечуючи консистентність і знижуючи дублікацію коду, отримуючи тим самим спільні методи та функціональність. Це робить код більш структурованим, зручним для розширення та підтримки. Базовий контроллер для адміністраторів виглядає так:

```
module Admin
 class BaseController < ApplicationController
   include Pundit
   rescue_from Pundit::NotAuthorizedError, with: :user_not_authorized
   before_action :authenticate_user!
   after_action :verify_authorized
   layout 'admin'
   private
   def user_not_authorized
     flash[:danger] = 'You are not authorized to perform this action.'
     redirect_to(request.referer || root_path)
   end
 end
end
```

Тут я підключаю бібліотеку Pundit, який включає в себе метод authorize, цей метод прописується в кожному окремому контролері, а не в базовому оскільки метод приймає аргумент який є об'єктом або символом класу, для якого потрібно перевірити політику доступу, для кожного контроллера вона різна. Цей аргумент визначає контекст, у якому буде виконано перевірку прав користувача. Наприклад, якщо ви хочете перевірити, чи має поточний користувач право редагувати певний запис, ви передаєте цей запис як аргумент методу authorize. Pundit найде відповідний клас політики (наприклад, UserPolicy для об'єкта User) і викличе метод, назва якого відповідає екшину (наприклад, index?), передавши в нього об'єкт як аргумент для перевірки правил визначених в цій політиці. Якщо користувач попробує виконати дію для якої в нього немає прав доступу, виникне помилка Pundit::NotAuthorizedError, яка буде вирішина викликом метода user_not_authorized. Цей метод встановлює повідомлення для користувача про те, що він не має права виконувати цю дію, використовуючи механізм флеш-повідомлень і перенаправляє користувача на попередню сторінку, з якої він прийшов або на головну сторінку, якщо реферер недоступний. Ще один метод бібліотеки Pundit verify_authorized, який викликається після кожного екшина. Цей метод перевіряє, чи була виконана авторизація за допомогою методу authorize в цьому екшині. Якщо метод authorize не був викликаний, verify_authorized викличе помилку Pundit::AuthorizationNotPerformedError, сигналізуючи про те, що авторизація не була перевірена. Цей підхід забезпечує додатковий рівень безпеки, вимагаючи, щоб розробник явно виконав перевірку авторизації у кожному екшині, що допомагає запобігти випадковому наданню доступу до дій або даних, які повинні бути обмежені. Використання after_action :verify_authorized є чудовим способом забезпечити, що політики авторизації застосовуються послідовно по всьому додатку.

В моледі User, створено константу, де в масиві перелічені усі можливі права доступу:

```
PERMISSION_LEVELS = %w[create_movie_with_tmdb create_and_update_movie
remove_movie create_hall update_hall destroy_hall manage_sessions
ticket_sales]
```

В AplicationPolicy, було додано такий код:

```
User::PERMISSION_LEVELS.each do |x|
  define_method :"permission_#{x}?" do
    @user.superadmin? || @user.permissions.include?(x)
  end
end
```

За допомогою метапрограмування у цьому коді динамічно створюються методи для перевірки рівнів дозволів користувача. Використовуючи цикл each, проходиться через кожен елемент масиву User::PERMISSION_LEVELS і для кожного з них створюється метод з назвою permission_#{x}?, де `x` є конкретним рівнем дозволу. Всі ці методи визначаються за допомогою методу define_method. Внутрішня логіка цих методів перевіряє, чи є користувач супер-адміністратором (@user.superadmin?) або ж чи міститься відповідний рівень дозволу в масиві дозволів користувача (`@user.permissions.include?(x)`).

Створені таким чином методи будуть доступні в контексті об'єкта, до якого вони були додані, тобто клас ApplicationPolicy або будь-який інший клас, який наслідує від нього. Це означає, що всі політики доступу, які наслідують ApplicationPolicy, матимуть доступ до методів, створених динамічно. Наприклад, якщо в `User::PERMISSION_LEVELS` є рівні дозволів create_movie_with_tmdb, manage_sessions та ticket_sales, то у політиці будуть доступні методи permission_create_movie_with_tmdb?, permission_manage_sessions? та permission_ticket_sales?, які дозволять перевіряти відповідні дозволи для користувача.

Також в базовому контролері я використовую метод authenticate_user!. Бібліотека Devise надає цей допоміжний метод, щоб допомогти розробникам легко контролювати доступ до певних частин додатка, вимагаючи, щоб користувач був аутентифікованим перед тим, як здійснити певні дії або

отримати доступ до певних контролерів чи екшинів. Якщо користувач не аутентифікований, Devise перенаправить його на сторінку входу.

3.4 Написання сервісів для додавання фільмів

Сервіс-об'єкт в Rails – це підхід до організації логіки додатка, який полягає в винесенні комплексної функціональності з контролерів або моделей у власні об'єкти, які називаються сервісами. Основна мета сервіс-об'єктів – це забезпечити чистоту коду, уникнути перевантаження контролерів та моделей, а також полегшити тестування і підтримку коду.

Сервіс-об'єкти використовуються для виконання конкретних завдань або операцій, які не відносяться безпосередньо до операцій бази даних або обробки введення/виведення. Це можуть бути різноманітні завдання, такі як інтеграція з зовнішніми API, оптимізація зображень, обробка файлів, відправка електронних листів, генерація звітів тощо.

Використання сервіс-об'єктів дозволяє підтримувати код додатка чистим, гнучким і добре структурованим. Вони допомагають розділити функціональність на логічні компоненти, що сприяє покращенню читабельності коду та зниженню його складності. Крім того, вони полегшують тестування, оскільки окремі частини функціональності можуть бути протестовані незалежно один від одного.

Для отримання доступу до великого обсягу даних з фільмами та різноманітну інформацію про них, таку як заголовки, рейтинги, актори, режисери та багато іншого, мною було написано сервіс з інтеграцією The Movie Database API.

Перш за все щоб перейти до написання сервісу потрібно отримати API ключ за допомогою якого ми отримаємо доступу до їхньої бази даних, це можна зробити після реєстрації. Ключі API – це конфіденційна інформація, тому важливо зберігати їх у безпечному місці і уникати їхнього прямого використання в коді. Для цього в своєму додатку я використовую бібліотеку

config, яка спрощує управління конфігурацією, забезпечуючи безпеку, чистоту коду та легкість у налаштуванні. Вона також надає можливість використовувати значення змінних середовища, що дозволяє легко налаштовувати ваше додаток для різних середовищ (розробка, тестування, виробництво) без необхідності зміни коду [20].

```
tmdb:
  api_key: <%= ENV['TMDB_API_KEY'] %>
  api_read_access_token: <%= ENV['TMDB_API_READ_ACCESS_TOKEN'] %>
```

Значення самого ключа встановлюється в змінних середовища, що забезпечує безпеку, адже ключ не зберігається безпосередньо у коді. Потім в файлі конфігурації ми встановлюємо пару ключ і значення, де значення в даному випадку це ключ API, взятий з змінної середовища за допомогою синтаксису ERB. Тепер коли нам потрібно буде використовувати ключ в коді ми можемо безпечно отримати до нього доступ за допомогою такого рядка: Settings.tmdb.api_key.

Сервіс для взаємодії з TMDB API приймає три параметра action який є обов’язковим і відповідає за те на який ендпоін буде надісланий запит, movie_id який не є обов'язковим і використовується в ендпоїнах для отримання детальнішої інформації про фільм та про акторський і режисерський склад, opts використовується для передавання додаткових параметрів до запиту. Всі ці параметри записуються в змінні на етапі створення сервіс об'єкта.

```
def initialize(action:, movie_id: nil, opts: {})
   @action = action
   @movie_id = movie_id
   @opts = opts
 end
```

Після створення йде виклик сервісу який викликає метод send_request.

```
def call
  response = send_request
```

```
  JSON.parse(response.body)
end
def send_request
  url = URI("#{Settings.tmdb.url}/3/#{path}?#{http_params}")
  http = Net::HTTP.new(url.host, url.port)
  http.use_ssl = true
  request = Net::HTTP::Get.new(url)
  request['accept'] = 'application/json'
  request['Authorization'] = "Bearer
#{Settings.tmdb.api_read_access_token}"
  http.request(request)
end
```

Цей метод збирає посилання відповідно до переданих параметрів за допомогою допоміжних методів path і http_params, налаштовує HTTP-клієнт та відправляє GET запит до TMDB API. Відповідь надісланого запиту я записую в змінну response, а потім перетворюю JSON-рядок з тіла response в ruby хеш. Це дозволяє подальше оброблення цієї відповіді в додатку за допомогою зручного для роботи формату даних.

```
def http_params
   params = { language: 'en-US', region: 'UA' }
   params.merge!(@opts)
   params.to_query
 end

 def path
   {now_playing: 'movie/now_playing',
     upcoming: 'movie/upcoming',
     search: 'search/movie',
     details: "movie/#{@movie_id}",
     credits: "movie/#{@movie_id}/credits"}[@action]
 end
```

Написаний сервіс можна викликати таким рядком коду: TmdbApiService.call(action: :now_playing)

Наступний сервіс який було створено відповідає за побудову об'єкта фільму на основі даних отриманих з TMDB API. Його основна мета – це виконання складних завдань, пов'язаних з отриманням та обробкою даних, та забезпечення модульності та гнучкості коду.

Під час ініціалізації сервісу передається зовнішній ідентифікатор фільму external_id, який використовується для здійснення запитів до TMDB API. Параметр external_id я отримую в результатах виконання попереднього сервісу.

```
def initialize(external_id:)
  @external_id = external_id
end
```

Виклик сервісу створює об'єкт фільму на основі виклику двох приватних методів movie_data та credits_data.

```
def call
  Movie.new({ external_id: @external_id }.merge(movie_data,
credits_data))
end
```

Метод movie_data викликає TmdbApiService для отримання детальних даних про фільм з TMDB API, включаючи заголовок, опис, жанр, тривалість, країни виробництва, дату виходу та посилання на постер. Потім ці дані обробляються відповідно до назв колонок та їх типу в базі даних. Наприклад цей рядок коду створює масив країн виробництва для фільму на основі отриманих даних: countries: attributes['production_countries'].map { |country| country['name'] }.

Він використовує метод map для ітерації по кожному елементу масиву production_countries, який містить інформацію про країни виробництва фільму. Для кожної країни програма витягує назву країни з об'єкта country за ключем name і додає його до нового масиву. Після завершення цього процесу отримується масив, який містить назви країн виробництва для даного фільму.

Розглянемо ще один приклад: DateTime.parse(attributes['release_date']). Цей код перетворює рядкове значення дати, яке міститься у змінній release_date, на об'єкт класу DateTime. Він використовує метод parse класу DateTime, який автоматично розпізнає формат дати у рядку і створює відповідний об'єкт що дозволяє коректно зберегти дату.

```
def movie_data
   attributes = TmdbApiService.call(action: :details, movie_id:
@external_id)
   {
     title: attributes['title'],
     description: attributes['overview'],
     genre: attributes['genres'].map { |genre| genre['name'] },
     duration: attributes['runtime'],
     countries: attributes['production_countries'].map { |country|
country['name'] },
     release_date: DateTime.parse(attributes['release_date']),
     poster_path: attributes['poster_path']
   }
end
```

Метод credits_data також викликає TmdbApiService, але цього разу для отримання та в подальшому обреленню даних про акторів та режисерів фільму. Спосіб отримання даних про акторський склад ідентичний до того як я отримував дані про країни виробників фільмів в методі movie_data. А от отримання данів про режисерів трохи відрізняється, тому що в масиві crew отримується інформація про всю команду яка брала участь в зйомці фільму, крім акторів. Наприклад люди які відповідали за камери, звук, спец. ефекти і інші. Це дуже багато людей і серед них мені потрібно вибрати тільки режисерів.

Давайте розберем код за допомогою якого це було зроблено: attributes['crew'].filter_map { |crew| crew['name'] if crew['job'] == 'Director' }.
В даному коді замість методу map, я використовую filter_map. В блоці цього методу перевіряється роль кожного елементу. Якщо роль рівна 'Director' (режисер), то вираз crew['name'] (ім'я режисера) додається до нового масиву, в

іншому випадку нічого не відбудеться і запишеться значення nil. Цей метод також проходить по кожному елементу масиву та виконує переданий блок, але його особливістю є те що він фільтрує значення, видаляючи nil, тобто він включає в результат лише ті елементи, які не є nil.

```
def credits_data
  attributes = TmdbApiService.call(action: :credits, movie_id: 
@external_id)
  {
    directors: attributes['crew'].filter_map { |crew| crew['name'] if 
crew['job'] == 'Director' },
    actors: attributes['cast'].map { |actor| actor['name'] }
  }
end
```

Правильно оброблені дані в цих двох методах повертаються в вигляді масиву та використовуються для створення об'єкту фільма. При створенні цього об’єкту також записується id фільму з TMDB в поле external_id моделі фільму. Цієї колонки в БД поки що не існує, тому потрібно створити міграцію яка його додасть. Зробити це можна за допомогою команди rails g migration AddExternalIdToMovies external_id:integer:uniq. Ця команда створить новий файл міграції де буде міститися код для додавання стовпця external_id до таблиці movies з типом даних integer, а також для створення унікального індексу для цього стовпця.

В полі external_id зберігаються тільки унікальні значення, але об'єкт фільму можна буде створити не тільки за допомогою сервісу, а і вручну. Це означає що значення поля external_id для таких випадків буде nil (пустим). Створенні другого об'єкту вручну закінчиться помилкою через порушення вимог унікальності (спроба створити ще один об'єкт з значенням nil в колонці external_id). Для вирішення цієї проблеми мною була додана наступна валідація: validates :external_id, uniqueness: true, allow_nil: true

3.5 Створення адміністративної панелі для основних моделей

Адміністративна панель на веб-сайті кінотеатру є ключовим інструментом для забезпечення ефективного керування контентом, підтримки актуальної та релевантної інформації на веб-сайті та забезпечення задоволення потреб користувачів. Перш за все перед створенням в панелі адміністратора основної функції для управління сеансами, потрібно розробити залежні панелі, це управління фільмами та кінозалами.

Так як я вже маю сервіси для отримання інформації та створення фільмів, то почну з панелі управління фільмами. Для цього перейдемо до файлу config/routes.rb і додаємо такий код:

```
namespace :admin do
  resources :movies do
    post :search, on: :collection
    post 'create_with_tmdb/:external_id', on: :collection, as: 
:create_with_tmdb, action: :create_with_tmdb
  end
end
```

Цей код використовується для створення адміністративної області з веб-інтерфейсом для керування ресурсом “фільми”. Використання ключового слова namespace дозволяє групувати різні маршрути та контролери відповідно до певного простору імен. У даному випадку, ми визначаємо простір імен admin, що означає, що всі маршрути та контролери, включені в цю групу, будуть доступні за адресою “/admin/…”.

Далі, за допомогою resources :movies, вказуємо, що для цього адміністративного простору імен будуть доступні стандартні REST-маршрути для керування ресурсом "фільми". Це означає, що буде створено адміністративний інтерфейс для керування фільмами, доступний за автоматично створеними маршрутами, такими як “/admin/movies”, “/admin/movies/new”, “/admin/movies/:id/edit” тощо. А також будуть створені

відповідні методи контролеру для керування фільмами (index, show, new, create, edit, update, destroy).

Далі генеруємо контроллер за допомогою команди rails g controller admin/movie. Наслідуємо його від базового контроллера панелі адміністратора створеного раніше:

class MoviesController < Admin::BaseController

та додаємо такий рядок коду before_action -> { authorize(:movie) }, який означає, що перед виконанням будь-якого екшину у контролері, автоматично викликається метод authorize(:movie). Цей метод використовується для перевірки дозволу на виконання певної операції над об'єктом. У даному випадку, перевіряється дозвіл на доступ до ресурсу movie.

Тепер перейдемо до сторінки створення/додавання нових фільмів до колекції. Я хочу мати можливість додавати фільм вручну заповнюючи форму або на основі отриманих даних від сервісу, для цього я в контролері створено такі ешкшини:

```
def new
  @movie = Movie.new
  @now_playing_list = TmdbApiService.call(action:
:now_playing).dig('results')
  @upcoming_list = TmdbApiService.call(action:
:upcoming).dig('results')
end

def create
  movie = Movie.new(movie_params)
  if movie.save
    flash[:success] = 'The movie has been added to our collection.'
    redirect_to admin_movies_path
  else
    flash.now[:danger] = 'Something went wrong.'
    render :new
  end
end
```

В екшині new я записую в інстансну змінну @movie новий екземпляр об'єкту, який буде використовується для відображення форми створення нового фільму (рис. 3.1). Після надсилання форми викликається екшин create. Якщо всі дані валідні то фільм буде створено, а користувач буде перенаправлений на індексну сторінку і попереджений флеш повідомленням про те що фільм додано. В іншому випадку сторінка перерендериться і корстувач побаче помилки, через які він не зміг додати фільм.

Рисунок 3.1 – Форма для додавання фільму вручну

Було додано колбеки з приватними методами:

```
before_action :load_movies, only: %i[index new search create_with_tmdb]
before_action :load_tmdb_movies, only: %i[new create_with_tmdb]
before_action :load_movie, only: %i[edit update destroy]
def load_movies
  @movies ||= Movie.all
end
def load_movie
  @movie = Movie.find(params[:id])
```

```
end
def load_tmdb_movies
  @now_playing_list = TmdbApiService.call(action:
:now_playing).dig('results')
  @upcoming_list = TmdbApiService.call(action:
:upcoming).dig('results')
end
```

Колбеки load_movies і load_movie, дістають з бази даних фільми або фільм відповідно і записують ці дані в інстансні змінні. Колбек load_tmdb_movies, викликає сервіси з потрібними параметрами, результати викликів також записуються в інстанси.

За допомогою даних з колбеку load_tmdb_movies, я презентую на сторінці new фільми які зараз йдуть у кіно та фільми які незабаром вийдуть за даними TMDB, та надаю можливість додати їх до колекції в один клік (рис. 3.2).

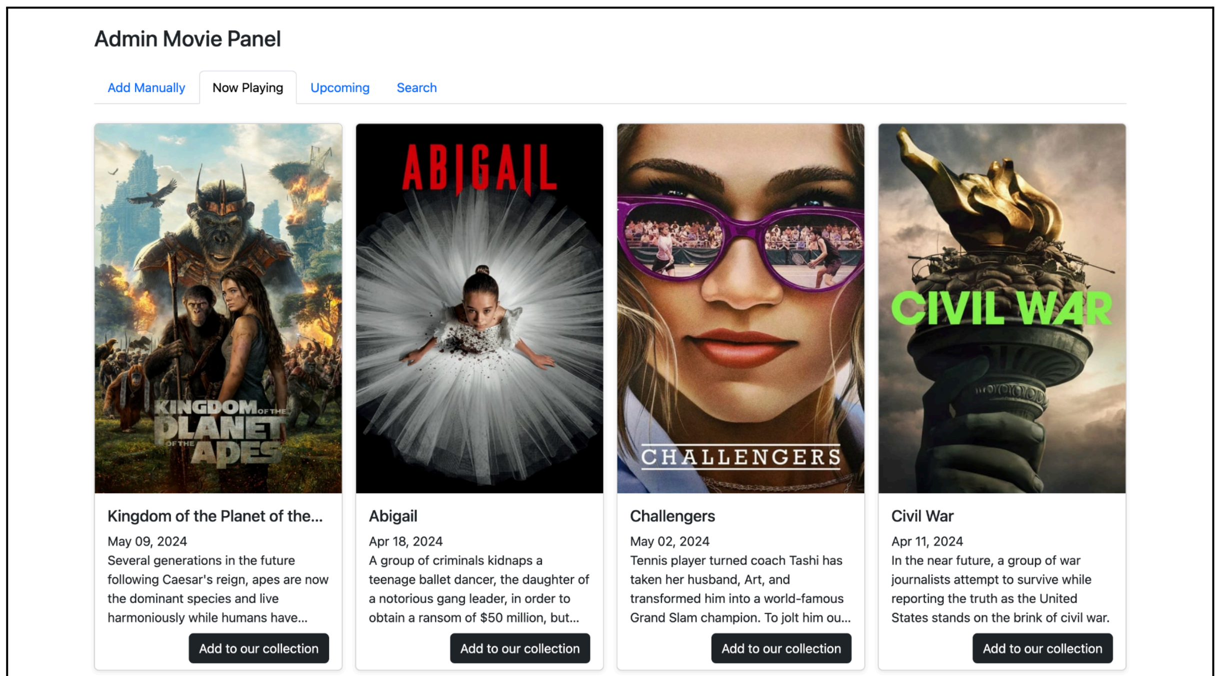


Рисунок 3.2 – Сторінка додавання фільму за допомогою інтеграції з TMDB

Є можливість скористатися пошуку фільму за назвами, для випадків коли потрібного фільму немає в списках. Для цього створено екшин search, який також використовує TMDB сервіс, але вже з параметрами для пошуку.

```
def search
  @search_list = TmdbApiService.call(action: :search, opts: { query:
params[:query] }).dig('results')
end
```

Після пошуку фільми з’являються на сторінці без перезавантаження самої сторінки. Це зроблено за допомогою Turbo Streams методом turbo_stream.replace 'search_list', який знаходить на сторінці HTML елемент з ідентифікатором search_list та оновлює його вміст.

```
<%= turbo_stream.replace 'search_list' do %>
  <div id="search_list">
    <div class="row row-cols-1 row-cols-sm-2 row-cols-lg-4 g-3">
      <% @search_list.each do |movie| %>
        <%= render 'card', movie: movie %>
      <% end %>
    </div>
  </div>
<% end %>
```

При натисканні на кнопку “Add to our collection” відправляється POST запит до екшину create_with_tmdb, де за допомогою другого сервісу TmdbConstructionService, створюється екземпляр об'єкта Movie. Так само як і в методі create встановлюється флеш повідомлення при вдалій спробі зберегти об’єкт до бази даних, але в цьому випадку переадресації на іншу сторінку немає. Можна припустити, що TMDB може не мати дані для деяких обов’язкових полів і об’єкт не створотиться через помилку валідації. В такому випадку, таб користувача переключиться на таб з формою для ручного створення, де всі валідні дані з TMDB будуть заповненні, а над не валідноми полями буде повідомлення про помилку.

```
def create_with_tmdb
  @movie = TmdbConstructionService.call(external_id:
params[:external_id])
```

```
  if @movie.save
    @movie = Movie.new
    flash.now[:success] = 'The movie has been copied to our
collection.'
    render :new, status: :created
  else
    cookies[:active_movie_tab] = 'Add Manually'
    flash.now[:danger] = 'All required fields must be filled'
    render :new, status: :unprocessable_entity
  end
```

Зручне переключення між способами додавання фільмів зроблено за допомогою табів. Кукі зберігають інформацію який таб користувач відвідав останнім, це потрібно для того щоб при ререндері сторінки, після успішного додавання фільму методом create_with_tmdb, користувач залишався на попередньо відкритому табі. Запис кукі відбувається за допомогою JavaScript функції, яка викликається при зміні табу.

```
setLastActiveTab(event) {
  event.preventDefault();
  document.cookie = "active_movie_tab=" +
event.currentTarget.textContent.trim() + ";path=/";
}
```

Ешкин update, зроблено просто. Користувач отримує флеш повідомлення і буде перенаправлений на сторінку index якщо спроба оновити атрибути фільму вдала і сторінка edit ререндериться так само як це було з екшином create і сторінкою new. Екшин destroy, так само покаже флеш повідомлення про успішне видалення, або про проблему чого видалення не відбулось. Так як кнопка видалення присутня при перегляді колекції кінотеатру та при перегляді табів із списками фільмів від TMDB на сторінці new, перенаправлення користувача буде відбуватись на ту сторінку звідки він викликав екшин destroy. Зроблено це за допомогою рядка: redirect_to request.referer || admin_movies_path. Якщо реферер відсутній то переадресація відбувається на сторінку колекцій.

Адміністративна панель для залів розроблена з одною сторінкою index, та модальними вікнами для створення і редагування залів (рис. 3.3).

Рисунок 3.3 – Модульне вікно створення залу

З рисунку 3.3 можна побачити що в формі модульного вікна є два поля name і size. Size інпут складається з двох частин, перше поле для встановлення кількості рядків та друге для колонок.

```
<%= text_field_tag 'hall[scheme][rows_count]',
f.object&.scheme['rows_count'], { data: { action:
'input->seats#buildSeats', seats_target: 'rowsCount' }, class:
'form-control' } %>
<%= text_field_tag 'hall[scheme][columns_count]',
f.object&.scheme['columns_count'], { data: { action:
'input->seats#buildSeats', seats_target: 'columnsCount' }, class:
'form-control' } %>
```

Заповнення цих полів викликають екшен buildSeats в Stimulus контролері seats [21]. Цей метод отримує введені дані за допомогою встановлених таргетів та генерує схему місць на їхній основі, створюючи об'єкт seats, де кожний ряд представляється масивом із заданої кількості місць (всі місця позначені як доступні за допомогою значення true).

```
  const rows_count = this.rowsCountTarget.value
```

```
  const columns_count = this.columnsCountTarget.value

  const seats = {}

  for (let i = 1; i <= rows_count; i++) {
    seats[`row_${i}`] = Array(Number(columns_count)).fill(true);
  }
```

Далі формується об'єкт data, що містить ідентифікатор залу і створену схему місць.

```
  const data = {
    hall: {
      id: this.hallIdTarget.value,
      scheme: {
        columns_count: columns_count,
        rows_count: rows_count,
        seats
      }
    }
  }
```

Цей об'єкт відправляється на сервер через POST-запит на екшин build_chart з необхідними заголовками, включаючи CSRF-токен для безпеки.

```
  fetch('/admin/halls/build_chart', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'X-CSRF-Token':
document.querySelector("meta[name='csrf-token']").getAttribute("content
")
    },
    body: JSON.stringify(data) })
    .then((response) => response.text()).then((html) => {
Turbo.renderStreamMessage(html); })}
```

Rails отримує запит та надсилає HTML з використанням Turbo Stream replace у відповідь, далі JS цю відповідь рендерить за допомогою Turbo.renderStreamMessage.

Цей фрагмент шаблону, отриманий з відповіді, відповідає за відображення схеми місць залу (рис 3.4):

```
<% hall['scheme']['seats'].values.each.with_index(1) do |row,
row_index| %>
  <ul class="d-flex showcase m-0 p-0">
    <% row.each.with_index do |seat, i| %>
      <li>
        <div class="seat <%= 'space' unless seat %>"
             data-row="<%= row_index %>"
             data-column="<%= i %>"
             data-action="click->seats#changeSeat">
        </div>
      </li>
    <% end %>
  </ul>
<% end %>
<%= hidden_field_tag 'hall[scheme][seats]', hall['scheme']['seats']
.to_json, data: { seats_target: 'seatsScheme' } %>
```

Тут ми відображаємо кожне місце за допомогою методу each, а в низу коду є скрите поле яке заповнено схемаю в форматі JSON. При кліку на місце викликається stimulus action для зміни його доступності (вільне місце або порожній простір).

```
changeSeat(e) {
  const seatsScheme = JSON.parse(this.seatsSchemeTarget.value);
  const row = `row_${e.target.dataset.row}`
  const column = Number(e.target.dataset.column)
  e.target.classList.toggle('space');
  seatsScheme[row][column] = !seatsScheme[row][column]
  this.seatsSchemeTarget.value = JSON.stringify(seatsScheme);
}
```

Спочатку метод отримує поточну схему місць із прихованого поля та перетворює її з JSON-формату в об'єкт. Далі визначає рядок і колонку натиснутого місця, використовуючи дані з атрибутів data-row та data-column. За допомогою методу toggle перемикається клас space. Значення відповідного місця в схемі також оновлюється (інвертується) та зберігається в приховане поле у вигляді JSON-рядка, що дозволяє зберегти актуальний стан схеми для подальшої обробки. Таким чином забираючи сидячі місця можна зробити схему будь якої складності, наприклад зробити прохід між рядами (рис 3.4).

Рисунок 3.4 – Модальне вікно створення залу з схемою місць

Адміністративна панель для створення сеансів була розроблена після можливість додавання фільмів і створення залів. Так само як і для залів розроблено основну index сторінку для відображення всіх доступних сеансів та модальні вікна для їх створення/оновлення.

В формі потрібно заповнити такі дані як фільм, зал, ціну, дату та час початку сеансу. Також в формі є checkbox “Repeat to”, при встановленні

прапорця з’являється ще одне поле. Це поле потрібно для встановлення дати до якої повторювати сеанс. Дана функція розроблена щоб не витрачати час на створення кожного сеансу окремо, різниця між якими буде лише дата його проведення (рис. 3.5).

Рисунок 3.5 – Модальне вікно створення сеансу

Якщо прапорець встановлений, в екшині create викликається метод create_repeated_sessions який створює повторювані сеанси.

```
def create_repeated_sessions(session)
  repeat_to = Date.new(*params.fetch(:session).slice(:'repeat_to(1i)',
:'repeat_to(2i)', :'repeat_to(3i)').values.map(&:to_i))
  if repeat_to > session.start_datetime.to_date
    begin
      Session.transaction do
        (session.start_datetime.to_date..repeat_to).each do |date|
          Session.create!(session_params.merge('start_datetime(1i)':
date.year.to_s, 'start_datetime(2i)': date.month.to_s,
'start_datetime(3i)': date.day.to_s))
        end
      end
      flash.now[:success] = 'Sessions successfully created'
    rescue => e
      flash.now[:danger] = e.message
```

```
    end
  else
    flash.now[:danger] = 'Please select a valid date range'
  end
end
```

Якщо дата більша за дату початку вхідного сеансу, запускається транзакція. В межах цієї транзакції для кожного створюється новий сеанс, використовуючи параметри вхідного сеансу та змінюючи тільки дату початку. Якщо всі сеанси успішно створені, користувач отримує повідомлення про успіх, в іншому випадку транзакція відкочується, і користувач отримує повідомлення про помилку. Якщо ж дата завершення повторення є некоректною (раніше або дорівнює даті початку), користувач отримує відповідне повідомлення про помилку без спроби створення сеансів.

До моделі сеансів було додано вебхук build_seats_data, який створює структуру даних для зберігання інформації про місця в залі. Він ітерує через ряди та місця в схемі залу, і на основі наявності або відсутності місця формує відповідні записи в хеші. Для кожного ряду створюється вкладений хеш, де ключами є номери колонок. Якщо місце присутнє, до цього хеша додається запис з інформацією про його номер (порядковий номер в ряду), позначка, що місце не заброньоване і позначка, що це не порожнє місце. Якщо місце це порожній простір, додається запис із позначками booked: nil, number: nil, і space: true. Отриманий хеш присвоюється атрибуту seats_data.

```
def build_seats_data
  data = {}
  hall.scheme['seats'].values.each.with_index(1) do |seats, row_index|
    data["row_#{row_index}".to_sym] = {}
    number = 1
    seats.each.with_index(1) do |seat, index|
      if seat
        data["row_#{row_index}".to_sym]["column_#{index}".to_sym] = {
booked: false, number: number, space: false }
        number += 1
```

```
        else
          data["row_#{row_index}".to_sym]["column_#{index}".to_sym] = {
booked: nil, number: nil, space: true }
        end
      end
    end
  self.seats_data = data
end
```

3.6 Розробка функціоналу для бронювання квитків

Функціонал для бронювання квитків розроблений як для персоналу в панелі адміністратора так і для користувачів. Перед реалізацією функцілалу була згенеровано нава модель командою rails g model ticket з полями row і seat для зберігання інформації про ряд і місце квитка та асоціативні поля session_id і user_id для встановлення зв’язку.

Почнемо розбирати код з панелі адміністратора, для цього в файлі route.rb було додано нові маршрути до неймспейсу admin/sessions, такі як get :tickets, on: :member та post :book, on: :member. Перший GET шлях потрібний для перегляду та вибору доступних місць кінотеатру (рис. 3.6), а POST для бронювання.

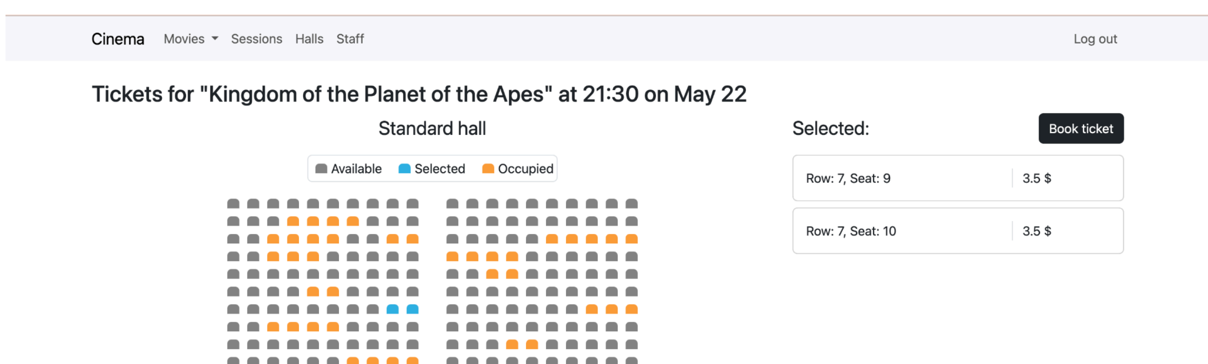


Рисунок 3.6 – Сторінка бронювання квитків в панелі адміністратора

Сторінка бронювання містить детальну інформацію про сеанс, таку як назву фільму, зал, дату та час показу. Доступні місця відображаються сірим

кольором, а заброньовані оранжевим. Адміністратор може натиснути на доступні місця для подальшого бронювання, колір цих місць зміниться на голубий, а в правій частині екрану з'явиться інформація про вибрані місця, така як ряд, місце та ціна. Натиснувши кнопку "Book tickets", виконається Stimulus метод adminBookSeat, цей метод відправляє дані про квитки, які формуються методом buildTicketsData, POST-запитом на сервер для бронювання.

```
adminBookSeat(e) {
  fetch(`/admin/sessions/${e.target.dataset.sessionId}/book`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'X-CSRF-Token':
document.querySelector("meta[name='csrf-token']").getAttribute("content
")
    },
    body: JSON.stringify(this.buildTicketsData())
  }).then((response) => response.text()).then((html) => {
    document.documentElement.innerHTML = html;
  })
}
```

Після успішного виконання запиту, сервер повертає HTML-відповідь, яка містить оновлені дані сторінки. Метод обробляє цю відповідь, замінюючи вміст головного елемента на новий HTML-вміст, отриманий з сервера. Це оновлює вміст сторінки з оновленою інформацією про квитки після бронювання.

Запит обробляється методом book, він знаходить об'єкт сеансу за допомогою переданого в параметрах його ідентифікатора. Потім перевіряє наявність необхідних параметрів. Якщо параметри присутні, перевіряється доступність квитків для бронювання на вибрані місця, це зроблено для того щоб не було накладки бронювання місць, пізніше розберемо це більш детальніше.

```
  @session = Session.find(params[:id])
```

```
  if params[:scheme].present? && params[:tickets].present?
    unless @session.tickets_available?(params[:scheme])
      flash.now[:danger] = 'Tickets not available'
      render :tickets
      return
    end
```

Якщо квитки недоступні, видається відповідне повідомлення про помилку, і відбувається перенаправлення на сторінку вибору квитків. У випадку успішності перевірки виконується транзакція, позначки бронювання в схемі сеансу змінюються на true, та створюються квитки без прив'язки до користувача. Якщо виникає помилка, транзакція відкочується, і відображається повідомлення про помилку. В кінці відбувається перенаправлення на сторінку вибору квитків.

```
ActiveRecord::Base.transaction do
  expire_payment_session(params[:scheme], params[:id])
  update_session(@session, params[:scheme])
  book_tickets(params[:tickets], params[:id])
  flash.now[:success] = 'Tickets successfully booked'
rescue => e
  flash.now[:danger] = e.message
end
```

Методи зміни схеми та бронювання квитків:

```
def update_session(session, scheme)
  scheme.each do |row, columns|
    columns.each do |column|
      session.seats_data[row][column]['booked'] = true
    end
  end
  session.save
end
def book_tickets(tickets, session_id, user_id = nil)
  tickets.each do |row, seats|
    seats.each do |seat|
```

```
        Ticket.create!(row: row, seat: seat, session_id: session_id ,
user_id: user_id)
    end
  end
end
```

Із сторони відвідувача, після обрання фільму, йому відкривається сторінка з детальною інформацією про фільм та із списком доступних сеансів, та доступних місць на цих сеансах (рис 3.7). При зміні вибору дня та часу сеансу, викликається метод swap, який заміняє схему на відповідну.

```
def swap
  @session = Session.find(params[:id])
  respond_to do |format|
    format.turbo_stream do
      render turbo_stream: turbo_stream.replace('seats-chart', partial:
'/movie_sessions/seats_chart', locals: { admin: :false })
    end
  end
end
```

Рисунок 3.7 – Сторінка бронювання квитків для відвідувачів

Сама схема ідентична до тієї що в панелі адміністратора, тому цей HTML код було винесено в паршал для зменшення дублювання коду. Кнопка бронювання так само відправляє дані про вибрані квитки, але на інший екшин.

```
def payment
  unless current_user
    flash[:danger] = 'You must log in before purchasing tickets'
    render json: { redirect_to: new_user_session_path }
    return
  end
  @session = Session.find(params[:id])
  unless @session.tickets_available?(params[:scheme])
    flash[:danger] = 'Tickets not available'
    render json: { redirect_to: movie_session_url(@session.movie) }
    return
  end
 ...
```

Метод payment обробляє процес оплати квитків для користувачів. Спочатку він перевіряє, чи користувач увійшов у систему. Якщо користувач не авторизований, видається повідомлення про необхідність входу, і здійснюється перенаправлення на сторінку входу. Далі метод знаходить сеанс за переданим ідентифікатором і перевіряє наявність доступних квитків для вказаної схеми місць. Якщо квитки не доступні, користувачу отримує повідомлення про відсутність квитків, і здійснюється перенаправлення на сторінку сеансу фільму.

```
  checkout = Stripe::Checkout::Session.create(
    customer: current_user.stripe_customer,
    line_items: stripe_products,
    metadata: stripe_metadata,
    payment_intent_data: { metadata: stripe_metadata },
    mode: 'payment',
    success_url:  my_tickets_url,
    cancel_url: movie_session_url(@session.movie)
  )
```

```
render json: { redirect_to: checkout.url }
```

Якщо користувач авторизований і квитки доступні, метод ініціює створення сесії оплати через Stripe. Він налаштовує сесію з параметрами поточного користувача, включаючи інформацію про квитки, метадані, URL для успішної оплати, який перенаправляє користувача на сторінку з їхніми квитками(рис 3.9), та URL для скасування оплати, який повертає їх на сторінку сеансу фільму. Після створення сесії користувача перенаправить на сторінку оплати Stripe для завершення процесу оплати (рис 3.8).

Рисунок 3.8 – Сторінка оплати Stripe

Після успішного бронювання квитків виконується вебхук який на основі раніше доданих метаданих змінює схему доступних місць сеансу і створює квитки.

```
metadata = event.data.object.metadata
scheme = JSON.parse(metadata.scheme)
tickets = JSON.parse(metadata.tickets)
session = Session.find(metadata.id)

update_session(session, scheme)
book_tickets(tickets, metadata.id, metadata.user_id)
```

```
expire_payment_session(scheme, metadata.id)
```

Для запобігання одночасного бронювання тих самих квитків було розроблено метод expire_payment_session. Цей метод завершує активні сесії оплати в Stripe, які конфліктують із поточною схемою місць для зазначеного сеансу. Спочатку метод отримує список відкритих сесій оплати в Stripe. Потім для кожної сесії перевіряється, чи існує конфлікт із тільки що заброньованими квитками для даного сеансу, використовуючи метод tickets_conflict?.

```
def tickets_conflict?(checkout, scheme, session_id)
  return false if session_id != checkout.metadata.id
  JSON.parse(checkout.metadata.scheme).each do |row, columns|
    columns.each do |column|
      return true if scheme[row]&.include?(column)
    end
  end

  false
end
```

Якщо виявляється конфлікт, метод завершує цю сесію, викликаючи Stripe::Checkout::Session.expire з ідентифікатором конфліктної сесії. Цей підхід гарантує, що будь-які відкриті сесії оплати, які можуть привести до подвійного бронюванню місць, будуть закриті.

Захищаючи систему від змінення або скасування сеансу після того, як квитки вже були заброньовані, забезпечуючи консистентність даних та уникнення конфліктів, для моделі Session, було додано метод have_tickets. Цей метод викликається новим колбеком і валідацією, яка виконується при оновленні сеансу тільки у випадку, якщо є якісь зміни в атрибутах, крім атрибуту seats_data. Метод have_tickets перевіряє, чи сеанс має заброньовані квитки, якщо такі квитки існують, то додається помилка з повідомленням, про те що сеанс не може бути оновлений або скасований.

```
before_destroy :have_tickets, prepend: true do
  throw(:abort) if errors.present?
```

```
end
validate :have_tickets, on: :update, if: -> {
changed.excluding('seats_data').any? }
def have_tickets
  errors.add(:base, 'The session cannot be updated/canceled after
booking the tickets') if tickets.any?
end
```

Висновки до розділу 3

Розроблений веб-сайт значно оптимізує рутинні процеси кінотеатру, такі як налаштування залів, додавання фільмів та створення сеансів. Інтеграція зі Stripe для обробки платежів забезпечує надійність і безпеку фінансових транзакцій, а інтеграція з TMDB дозволяє отримувати актуальну інформацію про фільми. Це не лише знижує витрати часу на ручне введення даних, але й забезпечує точність та актуальність інформації, що відображається на сайті. Система аутентифікації та авторизації забезпечує безпеку користувачів та дозволяє адміністратору ефективно керувати доступами. Інтуїтивно зрозумілий інтерфейс веб-сайту покращує користувацький досвід як для відвідувачів, так і для адміністраторів. Відвідувачі можуть легко переглядати інформацію про фільми, купувати квитки та вибирати місця в залі, що робить процес відвідування кінотеатру приємнішим.

ВИСНОВКИ

В рамках виконання кваліфікаційної роботи було розроблено веб-сайт для кінотеатру. Процес розробки був зосереджений на забезпеченні зручності та ефективності користувацького досвіду як для відвідувачів, так і для адміністрації кінотеатру.

На сторінці головного меню користувачі можуть легко переглядати розклад сеансів, інтуїтивний дизайн та зручний інтерфейс дозволяють користувачам швидко знаходити необхідну інформацію. Основний акцент зроблено на системі бронювання квитків, де ключовим елементом є інтерактивна схема залу, яка дозволяє користувачам бачити вільні та зайняті місця у реальному часі, легко їх вибирати та здійснювати оплату онлайн. Важливою частиною функціоналу є інтеграція з платіжною системою Stripe, що забезпечує безпечні і швидкі транзакції.

Для адміністрації розроблено панель управління, яка значно полегшує внутрішні операції кінотеатру. Ця панель дозволяє адміністраторам створювати та редагувати сеанси, керувати залами, та обробляти бронювання квитків. Однією з ключових функцій є інтеграція з базою даних TMDB, що забезпечує легке та швидке додавання нових фільмів до колекції кінотеатру.

Для забезпечення безперебійної роботи та масштабованості веб-сайт було розгорнуто на платформі Heroku. Це дозволяє швидко і безпечно розгортати нові версії додатку, автоматично масштабувати ресурси у відповідь на зростання трафіку, а також інтегрувати додаткові сервіси.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Веб-сайт кінотеатру Галичина. веб-сайт. URL: https://galychyna.net/ (дата звернення: 19.03.2024)
2. Веб-сайт кінотеатру MovieLand. веб-сайт. URL: https://www.movieland.com.ua/ (дата звернення: 19.03.2024))
3. Веб-сайт кінотеатру Люм'єр. веб-сайт. URL: https://www.kinolumiere.com/ (дата звернення: 19.03.2024))
4. Документація Ruby on Rails. веб-сайт. URL: https://guides.rubyonrails.org (дата звернення: 25.03.2024)
5. Документація The Movie DataBase API. веб-сайт. URL: https://developer.themoviedb.org/docs/getting-started (дата звернення: 26.03.2024)
6. Документація Stripe API. веб-сайт. URL: https://docs.stripe.com/api (дата звернення: 26.03.2024)
7. Документація Heroku. веб-сайт. URL: https://devcenter.heroku.com/categories/ruby-support (дата звернення: 28.03.2024)
8. Документація ESBuild. веб-сайт. URL: https://esbuild.github.io/getting-started/ (дата звернення: 1.04.2024)
9. Документація Bootstrap. веб-сайт. URL: https://getbootstrap.com/docs/5.3/getting-started/introduction/ (дата звернення: 1.04.2024)
10. Документація бібліотеки Faker. веб-сайт. URL: https://github.com/faker-ruby/faker (дата звернення: 2.04.2024)
11. Документація бібліотеки RuboCop. веб-сайт. URL: https://docs.rubocop.org/rubocop/1.63/index.html (дата звернення: 2.04.2024)
12. Документація бібліотеки RSpec. веб-сайт. URL: https://rspec.info/documentation/ (дата звернення: 3.04.2024)
13. Документація бібліотеки FactoryBotRails. веб-сайт. URL: https://github.com/thoughtbot/factory_bot_rails (дата звернення: 3.04.2024)

14. Документація бібліотеки Capybara. веб-сайт. URL: https://github.com/teamcapybara/capybara (дата звернення: 5.04.2024)

15. Документація бази даних PostgreSQL. веб-сайт. URL: https://www.postgresql.org/docs/ (дата звернення: 12.04.2024)

16. Документація бібліотеки ActiveStorage. веб-сайт. URL: https://edgeguides.rubyonrails.org/active_storage_overview.html (дата звернення: 15.04.2024)

17. Документація бібліотеки active_storage_validations. веб-сайт. URL: https://github.com/igorkasyanchuk/active_storage_validations (дата звернення: 15.04.2024)

18. Документація бібліотеки Devise. веб-сайт. URL: https://github.com/heartcombo/devise (дата звернення: 22.04.2024)

19. Документація бібліотеки Pundit. веб-сайт. URL: https://github.com/varvet/pundit (дата звернення: 22.04.2024)

20. Документація бібліотеки config. веб-сайт. URL: https://github.com/rubyconfig/config (дата звернення: 25.04.2024)

21. Документація Stimulus JS. веб-сайт. URL: https://stimulus.hotwired.dev/ (дата звернення: 1.05.2024)

StrikePlagiarism.com King Danylo University

Дата звіту 5/24/2024
Дата редагування ---

Звіт не був оцінений.

метадані

Заголовок
Розробка веб-орієнтованої системи керування діяльністю кінотеатру засобами Ruby on Rails

Автор
Кудла Роман Науковий керівник / Експерт

підрозділ
King Danylo University

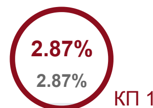
Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

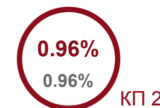
Заміна букв	ß	0
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	26

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



КП 1



КП 2

25 Довжина фрази для коефіцієнта подібності 2

13480 Кількість слів

100376 Кількість символів

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз

Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	http://repository.ukd.edu.ua/bitstream/handle/123456789/395/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0%20%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0%20%D0%A1%D1%82%D0%B5%D0%BF%D0%B0%D0%BD%D1%8E%D0%BA.pdf?sequence=1	56	0.42 %
2	http://repository.ukd.edu.ua/bitstream/handle/123456789/394/%D0%A0%D1%83%D0%B4%D0%B8%D0%B9%20%D0%90%D0%BD%D0%B4%D1%80%D1%96%D0%B9%20%D0%B4%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0.pdf?sequence=1	41	0.30 %
3	http://repository.ukd.edu.ua/bitstream/handle/123456789/395/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0%20%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0%20%D0%A1%D1%82%D0%B5%D0%BF%D0%B0%D0%BD%D1%8E%D0%BA.pdf?sequence=1	32	0.24 %