МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

Кафедра комп'ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ

Завідувач кафедри

_____Аліна САВЧЕНКО

«___»_____2021 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ

«МАГІСТРА»

ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ "ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ"

Тема: «Мобільний додаток для організації роботи з мережами кінотеатрів»

Виконавець: Мельник Владислав Євгенович

Керівник: к.т.н., доцент Холявкіна Тетяна Володимирівна

Нормоконтролер:_____ Ігор РАЙЧЕВ

Київ 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютерних інформаційних технологій

Галузь знань, спеціальність, освітньо-професійна програма: 12 "Інформаційні технології", 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології".

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Аліна САВЧЕНКО
«_____»_____ 2021 р.

ЗАВДАННЯ

на виконання дипломної роботи студента

Мельника Владислава Євгеновича
(прізвище, ім'я, по батькові)

1. Тема роботи: «Мобільний додаток для організації роботи з мережами кінотеатрів» затверджена наказом ректора від 12.10.2021р. за № 2228/ст.

2. Термін виконання роботи: з 12.10.2021 по 31.12.2021р.

3. Вихідні дані до роботи: мова розробки додатку – Swift, технології розробки додатку – MVVM, SwiftUI, JSON, середовище розробки XCode.

4. Зміст пояснювальної записки: вступ, аналітичний огляд і постановка завдання, аналіз реалізації основних компонентів програмного продукту, розробка програмної реалізації роботи додатку, розробка інтерфейсу та підтримка додатку, висновки.

5. Перелік обов'язкового ілюстративного матеріалу: модель роботи шаблону MVVM, схема роботи модулю екрану, реалізація роботи модуля покупки.

6. Календарний план-графік

№ п/п	***Завдання***	***Термін виконання***	***Підпис керівника***
1.	Проаналізувати літературу та джерела за темою дипломної роботи	11.10.21 – 14.10.21р.	
2.	Розроблення та затвердження плану дипломної роботи	15.10.21 – 17.10.21р.	
3.	Привести консультації з науковим керівником щодо створення першого розділу	18.10.21 – 19.10.21р.	
4.	Розробка розділу 1	20.10.21 – 29.10.21р.	
5.	Розробка розділу 2	30.10.21 – 09.11.21р.	
6.	Розробка розділу 3	09.11.21 – 12.11.21р.	
7.	Розробка розділу 4	12.11.21 – 21.11.21р.	
8.	Висновки та оформлення пояснювальної записки дипломної роботи	28.11.21 – 10.12.21р.	
9.	Підписання необхідних документів у встановленому порядку	11.12.21 –19.12.21р.	
10.	Підготовка до захисту та попередній захист дипломного проекту на випусковій кафедрі дипломної роботи	20.12.21 – 21.12.21р.	

7. Дата видачі завдання 12 жовтня 2021р.

Керівник дипломної роботи _____ Тетяна ХОЛЯВКІНА
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання _____ Владислав МЕЛЬНИК
(підпис випускника) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Мобільний додаток для організації роботи з мережами кінотеатрів» викладена на 117 сторінках, містить 56 рисунків, 25 літературних джерел.

Об'єкт дослідження: додаток для організації роботи з мережами кінотеатрів.

Мета роботи: формування, аналіз та розробка мобільного додатку для організації роботи з мережами кінотеатрів, що покращить сервіс та збільшить продуктивність пошуку, вибору та купівлі білету для користувача.

Предмет дослідження: структуризація та розробка системи організації роботи з кінотеатрами.

Ключові слова: МОБІЛЬНИЙ ДОДАТОК, АРХІТЕКТУРА ПРОГРАМИ, ОРГАНІЗАЦІЯ РОБОТИ, КІНОТЕАТР, ІНТЕРФЕЙС КОРИСТУВАЧА.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

IOS – мобільна операційна система розроблена компанією Apple.

Objective-C – мова програмування розроблена компанією Apple.

Swift – сучасна мова програмування, заміна Objective-C.

Face ID – технологія захисту, призначена для сканування обличчя.

ARC – технологія керування пам’яттю.

NoSQL – база даних відмінна від підходу таблиць-відношень.

MVC – шаблон коду: модель, представлення, контролер.

MVP – шаблон коду: модель, представлення, доповідач.

MVVM – шаблон коду: модель, представлення, модель представлення.

VIPER – покращений шаблон коду MVVM.

JSON – формат обміну даними у вигляді структурного тексту.

XCode – середовище розробки.

NeXT – файл мови розмітки.

NIB – файл, який описує візуальні елементи інтерфейсу.

XML – розширювана мова розмітки.

UI – інтерфейс користувача.

ВСТУП

Згадка про перші кінотеатри датуються з 1900-х років. На сьогоднішній день, заклади для перегляду фільму, можна назвати одним із найпопулярніших типів надання розважальних послуг. З кожним днем, кількість кінотеатрів збільшується, додаються нові типи кінотеатрів, змінюються їхня модель бізнесу, покращується сервіс. Статистика визначає, що світові касові збори кінотеатрів заробили 42 мільярда доларів, а на домашні та мобільні розваги - 59 мільярда доларів у всьому світі.

Зважаючи на те, що кількість кінотеатрів, кожного року, збільшується, змінюється і їх пропозиція та конкуренція. Тому, для просування, мережі кінотеатрів, повинні заохочувати клієнтів відвідувати саме їх заклади.

Одним із найпопулярнішим засобом комунікації, поширення та отримання інформації, на сьогоднішній день, є мобільний телефон. Саме з ним, більшість людей починає та закінчує свій день. Він завжди знаходиться у нас під рукою, та, в разі потреби, ним користуємось для полегшення буденних задач.

Сучасний мобільний телефон значно відрізняється від своїх попередників. За допомогою нього можна, переглядати сторінки в мережі, знаходити місця, та орієнтуватись на картах, проводити оплату, та шукати цікавий медіа контент.

Виходячи з описаного вище, використання мобільного телефону в якості додатку для просування мережі кінотеатрів та поширення власних послуг, є гарною ідеєю. Адже для того, щоб знайти цікавий фільм, купити білет та використати його, клієнту достатньо мати телефон з потрібним додатком. А у випадку, великою кількості відвідувачів, цей функціонал значно покращить продуктивність роботи кінотеатру, спростить та підвищить обслуговування клієнтів.

Отже, розроблений додаток буде проводити розширений пошук на базі одного, або багатьох кінотеатрів, з можливістю перегляду усіх доступних варіантів показу вибраного фільму. Програма повинна забезпечувати користувача функціоналом для покупки білетів, відповідно до ціни кінотеатру. Додатково повинна бути передбачена можливість формування QR-коду для його показу на касі кінотеатру.

Виходячи з цього, програма буде забезпечувати просування мереж кінотеатрів, їх цін, асортименту та послуг, а користувачу, в свою чергу, буде легше і зручніше зорієнтуватися у великому виборі фільмів, знайти кращий кінотеатр відповідно до рейтингу та ціни.

Реалізація сервісу забезпечить автоматизацію покупки та продажу білетів, формування статистичних даних, просування кіно-прем'єр. А отже, збільшить прибуток та зменшить витрати на обслуговування, покращить сервіс та зменшить кількість черг.

Розробка додатку буде проводитися на базі операційної системи iOS, з використанням мови програмування – Swift. Розробка програми проводиться з використання середовища розробки XCode. Все це забезпечує формування швидкого та зрозумілого інтерфейсу, що дозволить користувачу зменшити час, витрачений на роботу з програмою. А система захисту iOS дозволить залишити особисті дані користувача приватними. Для роботи з базою даних буде використана Cloud Firestore.

При дослідженні були застосовані методи синтезу, які можна побачити під час аналізу і оцінки цифрових пристроїв, існуючих додатків та методів для розробки програмного забезпечення. Порівняльний та абстрактний метод проявляються у знаходженні характеристик системи, коректності роботи бази даних, обробки отриманої інформації.

Актуальність теми обґрунтована тим, що в сучасних умовах, наявна тенденція збільшення кількості кінотеатрів, а разом з цим, збільшення конкуренції на ринку розважальних послуг. Це змушує індустрію кіно, знаходити нові ніші для розширення власної аудиторії. Крім того, збільшення кількості кінотеатрів, спричиняє перенасичення ринку та дезорієнтує клієнта при виборі комфортного місця перегляду фільму. Саму тому розробка додатку, який, зі сторони клієнта, полегшив би вибір фільму та кінотеатру, покупку білету та його використання, а, зі сторони індустрії, дав можливість запровадження нових послуг та розширення власної аудиторії, є актуальним завданням.

Отже, **метою дипломної роботи** є формування, аналіз та розробка мобільного додатку для організації роботи з мережами кінотеатрів, що покращить

сервіс та збільшить продуктивність пошуку, вибору та купівлі білету для користувача.

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД І ПОСТАНОВКА ЗАВДАННЯ

1.1. Дослідження предметної області

На сьогоднішній день, індустрія кіно розвивається значно швидше, ніж будь-коли. Кожен заклад, намагається залучити все більше аудиторії відвідувачів, покращуючи власні умови перегляду. Разом з цим збільшується обсяг кінотеатру, змінюється його рейтинг та ціна.

Для того, щоб попит відвідувачів не відставав від пропозиції, закладам необхідно проводити рекламні компанії, збільшувати свою аудиторію. Одним із найкращих варіантів, є інтеграція в сучасні технологічні рішення. Адже, сьогодні, кожна людина, так чи інакше зв’язана з цією сферою діяльності.

1.1.1. Аналіз сучасних цифрових пристроїв

На сьогоднішній день, у ролі технологічних рішень, які зв’язують людину з світом технологій є цифровий пристрій. Його визначають як електронний пристрій, який може: створювати, генерувати, формувати, надсилати, поширювати, отримувати, зберігати, відображати, або ж обробляти інформацію.

До таких технологій відносять настільні комп’ютери, ноутбуки, планшети, периферійні пристрої, сервери, мобільні телефони, смартфони та будь-які подібні пристрої зберігання даних, які зараз існують або можуть існувати у міру розвитку технологій [1].

На сьогоднішній день, найпоширенішим пристроєм, що зв’язує людину з цифровим світом є смартфон. Близько 94,9 % з користувачів володіють цим пристроєм.

За статистикою, у світі, наявно більше 8 мільярдів активних пристроїв. Отже більшість людей мають хоча б один мобільний телефон.

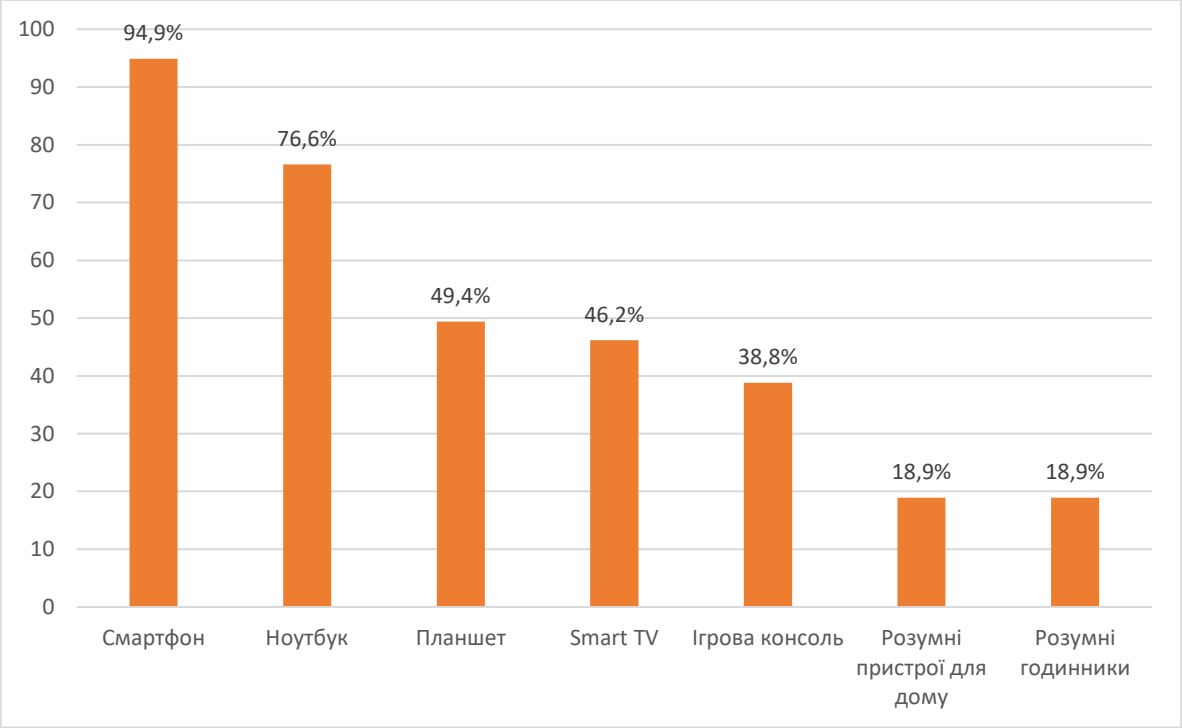


Рис. 1.1. Розподіл цифрових пристроїв у світі

Комп'ютери (як настільні, так і ноутбуки) та планшети, відповідно, другі та треті за популярністю електронні пристрої. Більше трьох з чотирьох людей у світі володіють комп’ютером, і майже половина володіє планшетами.

Отже смартфони займають центральне місце в цифровому житті людини. Це найпопулярніший інформаційний та комунікаційний пристрій сучасності. Вони є потужними засобами комунікації та мають певні переваги над персональними комп’ютерами та ноутбуками [2]. Розглянимо деякі із них:

Універсальність. Смартфон - робить все потроху. Він може бути телефоном, фотоапаратом, портативним музичним плеєром, GPS-пристроєм, електронний зчитувач чи будь-чим іншим, що вам потрібно.

Текстові повідомлення. Разом з голосовими дзвінками, смартфони дозволяють спілкуватися за допомогою текстових повідомлень. Це швидше, ніж електронна пошта, але менш нав'язливе, ніж голосовий дзвінок. Це зручний спосіб інформувати інших або отримувати відповіді на прості запитання. Ви можете

зробити це за допомогою кількох пальців і дотиків, а не використовувати комп’ютер.

Зв’язок. Смартфони – це, перш за все, мобільні телефони, а потім - портативні обчислювальні пристрої. Хоча деякі планшети і навіть кілька ноутбуків обладнані технологією для підключення до стільникових мереж. Як би нині не були поширені мережі Wi-Fi, все ще є величезні простори, де немає бездротової мережі, але ваш смартфон все ще може отримувати сигнал 3G (або, можливо, навіть 4G). Багато смартфонів також можуть виступати в якості точки доступу Wi-Fi, що дає змогу надавати доступ до мобільного зв’язку з вашим ноутбуком або планшетом для підключення практично з будь-якого місця.

Доступність. Куди б ви не йшли, він завжди буде при вас, у вашій кишені, або гаманці. Якщо ви хочете переглянути веб-сторінки, перевірити електронну пошту, дізнатись, яка погода буде в майбутньому, або просто подзвонити, смартфон майже завжди є під рукою.

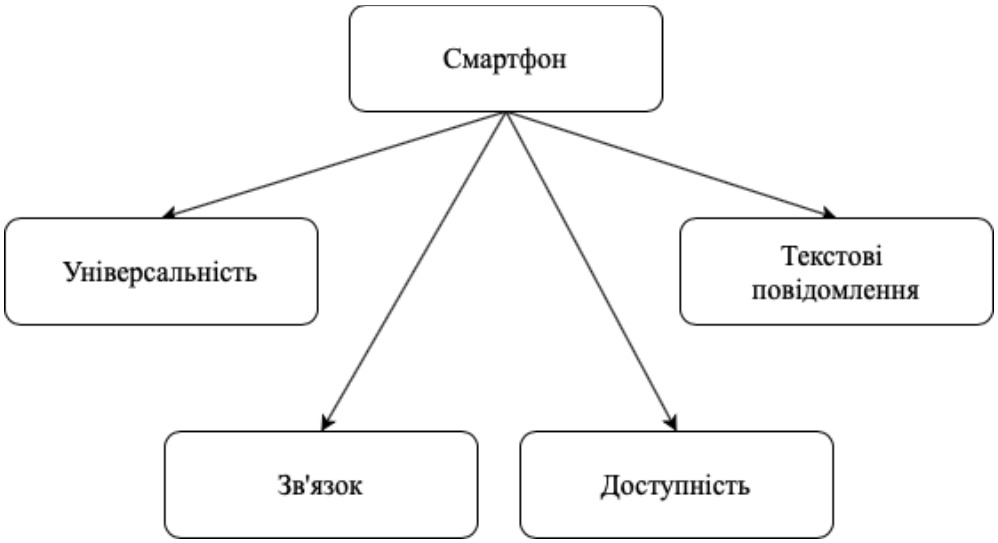


Рис. 1.2. Переваги смартфону

Отже смартфон, це універсальний та найбільш поширений цифровий пристрій у світі, що займає значне місце у житті людини. Виходячи з цього, найкращим варіантом поширення і просування свого продукту, є його інтеграція зі смартфоном. Достатньо створити та підтримувати мобільний додаток, щоб певна аудиторія людей мала змогу, використовуючи його.

1.1.2. Аналіз сучасних мобільних операційних систем

Мобільна операційна система - це набір низькорівневих програм, які дозволяють абстрагуватись від конкретних особливостей мобільного телефону та надають певні можливості мобільним додаткам, що працюють на ньому. Так само, як і персональні комп'ютери, які використовують Windows, Linux або Mac OS, мобільні пристрої мають свої операційні системи, такі як Android, iOS, Windows Phone тощо. Мобільні операційні системи набагато простіші та орієнтовані на бездротове з'єднання, мультимедійні формати та інші способи введення інформації до них.

На сьогоднішній день найпопулярнішими мобільними операційними системами є Android та iOS [3]. Їх загальна частка ринку складає 98,7%. База користувачів Android дещо більша, що багато в чому пояснюється її доступністю на пристроях різних виробників, включаючи HTC, LG, Motorola, Samsung тощо. З іншого боку, iOS працює лише на продуктах Apple. Розглянимо про кожну окремо:

Android - це мобільна операційна система на основі ядра Linux та іншого програмного забезпечення з відкритим кодом. Він був розроблений для мобільних пристроїв із сенсорним екраном, таких як смартфони, планшети, розумні годинники.

iOS - це мобільна операційна система від багатонаціональної компанії Apple Inc. Спочатку розроблена для iPhone (iPhone OS), пізніше вона використовувалася в таких пристроях, як iPod touch та iPad. Apple не дозволяє встановлювати iOS на стороннє обладнання.

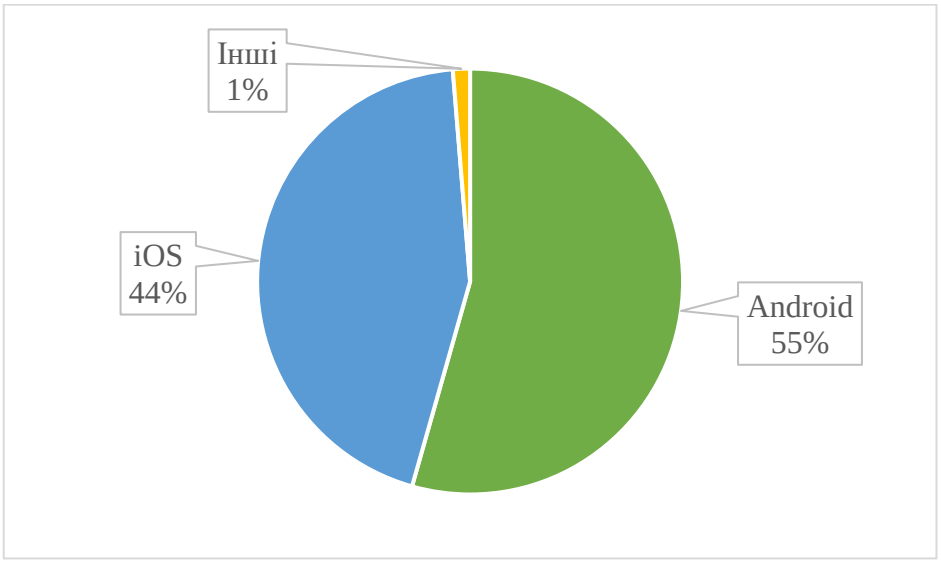


Рис. 1.3. Розподіл ОС на мобільному сегменті

Операційні системи iOS та Android, досить схожі, проте кожна з них має свої переваги та недоліки. Розглянемо переваги iOS.

Безпека. Apple перевіряє кожну програму, яка розміщена в її App Store, що є однією з причин того, що її менша ймовірність бути зламаною, ніж Android.

Конфіденційність. Apple не займається рекламним бізнесом, а це означає, що їй не потрібні ваші особисті дані, щоб заробляти гроші або продавати вам деякі продукти. Крім того, послідній час, компанія приділяє більше уваги конфіденційності у мережі, додавши систему блокування від відстежування.

Інтеграція програмного та апаратного забезпечення. Apple виробляє власні мобільні пристрої та програмне забезпечення. Завдяки цій суміші пристрої Apple працюють безперебійно. Наприклад - Face ID. Багато сторонніх додатків підтримують Face ID, тому що імпортувати цю бібліотеки у свою програму.

Зручний для користувача. iOS-найбільш зручна операційна система. Незважаючи на те, що виробники телефонів Android спрощують свою ОС, вона все одно не працює більш плавніше, ніж iPhone.

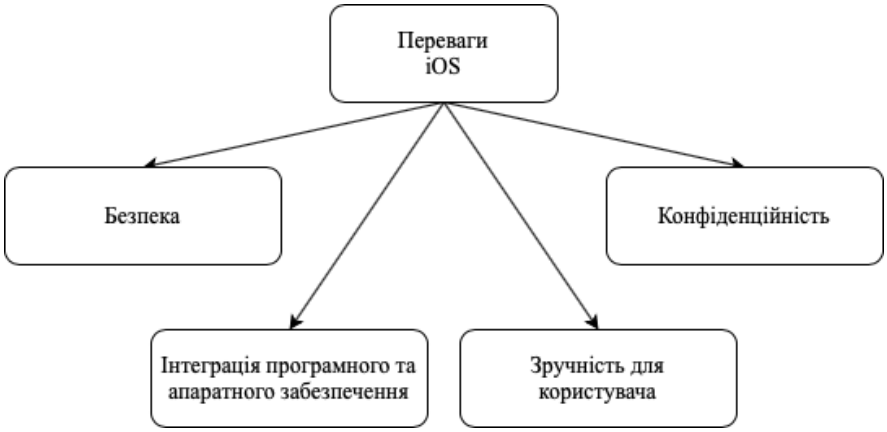


Рис. 1.4. Переваги iOS

До недоліків можна віднести закритість системи, залежність використання функцій телефону від Apple та наявність недостатньо інтерактивної файлової системи. Розглянемо переваги Android.

Краща ціна. Так як Android є відкритою операційною системою, її легше імпортувати на різні телефони. А тому ціновий діапазон пристроїв з даною операційною системою значно нижчий.

Модульність системи. Android підтримує модифікацію звичайних елементів системи, їх зміну зовнішнього вигляду.

Можливість зміни стандартних налаштувань. Відкрита операційна система дає можливість розробникам відкривати більший обсяг функцій доступних користувачу та підлаштовувати власний пристрій під себе.

Рис. 1.5. Переваги Android

До недоліків Android можна віднести складну підтримку великого різноманіття пристроїв, наявність шкідливого програмного забезпечення, доступ до вразливих компонентів системи.

Отже розглянувши переваги та недоліки систем iOS та Android, можна визначити, що iOS є більш захищеною, продуктивною та зручною для користувача ОС. Тому, в першу чергу, більш доцільно, розробляти додаток на базі iOS, з подальшою підтримкою Android.

1.2. Огляд і аналіз використання існуючих додатків

На сьогоднішній день вибір мобільних додатків, для пошуків кінотеатрів та замовлення квитків, досить невеликий. Кожен сервіс має як переваги так і свої недоліки. Основна частина таких програм зорієнтована на двох магазинах – Play Market та AppStore. Розглянемо деякі із них.

KinoZone. [4] Сервіс, який дозволяє користувачеві отримувати інформацію про актуальні покази фільмів, зберігати їх в бібліотеку, вибирати кінотеатр та дивитись наявні вільні місця у ньому. Крім цього програма дає змогу користувачеві продивитись інформацію про стрічку.

Перевагами сервісу є:

- простота інтерфейсу;
- можливість збереження стрічки;
- перегляд опису фільму;
- перегляд наявних кінотеатрів та вільних місць.

До недоліків можна віднести:

- неможливість купівлі білету;
- наявність проблем в дизайні;
- необхідність вибору міста розташування.

Рис. 1.6. Головний екран додатку KinoZone

Кіноріум. [5] Мобільний додаток, який дозволяє в зручній форму переглядати усі наявні кінострічки, інформацію про них, додавати до бібліотеки, отримувати пропозиції, та переглядати новинки.

До переваг сервісу можна віднести:

- зручність інтерфейсу;
- рейтингова система;
- рекомендаційна система;
- можливість збереження стрічок;
- деталізований опис фільму.

Недоліками сервісу є:

- неможливість знайти кінотеатр;

- неможливість купити білет на показ.

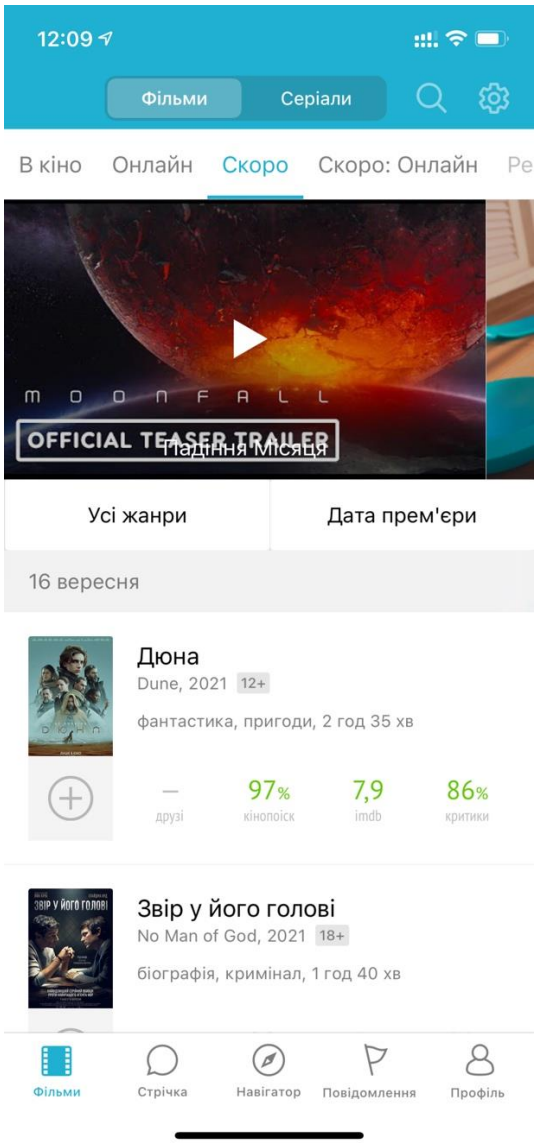


Рис. 1.7. Головний екран додатку Кіноріум

Fander. [6] Сервіс для підбору фільмів на основі ваших вподобань. Він дає змогу користувачу знайти новий фільм, що найбільш підходе по рейтингу його перегляду. Програма має можливості збереження відібраних та проглянутих стічок.

До переваг можна віднести:

- зручність інтерфейсу;
- можливість збереження проглянутих та відібраних стрічок;
- рейтингова система пошуку;

- стрічка новин;
- список прем’єр та показів;
- бонусна програма від партнерів;
- можливість пошуку кінотеатру;
- можливість купівлі білету.

Недоліками додатку є:

- насиченість інтерфейсу;
- показ тільки відібраних кінотеатрів;
- можливість купівлі тільки в певних кінотеатрах.

Рис. 1.8. Головний екран додатку Fander

Розглянувши декілька існуючих варіантів реалізації додатку, можна визначити, що кожен сервіс виконує певне завдання, але не перекриває загальні

проблеми користувача в цілому. Проглянувши основні недоліки можна узагальнити наступні із них:

- навантаженість інтерфейсу;
- не повна реалізація купівлі квитка та пошуку кінотеатру;
- прив’язаність до місця розташування.

До переваг розробленого мобільного додатку можна віднести:

- простоту інтерфейсу;
- відсутність прив’язки до місця розташування;
- функціонал пошуку кінотеатру;
- можливість замовлення квитка.

Наразі на платформі iOS немає програм, які б задовольняли і могли б виконувати наведені вище функції. Тому проблематика розробки додатку є актуальним завданням.

1.3. Дослідження засобів реалізації додатків на базі iOS

Розробка програмного забезпечення починається з визначення цільової платформи, мови програмування, та інструментів для цього. Визначена раніше, захищеність, продуктивність та зручність користування операційною системою iOS, дає цілком реальні причини для її використання під час створення додатку.

Отже визначившись з платформою, необхідно використовувати мову програмування, яка б задовольняла характеристики і потреби програми. Нативна розробка під iOS виконується на базі мов програмування Objective-C, або Swift.

Objective-C - це об'єктно-орієнтована мова програмування, написана на C. В основному він використовується при розробці операційних систем iOS та Mac OS, а також його додатків. Будь-яку програму, написану на C, можна компілювати за допомогою компілятора Objective-C, і ви також можете вільно включати код у C у межах Obejctive-C класу.

У 2014 році Apple представили нову програмування – Swift, яка була спроектована для покращення розробки додатків на базі Apple систем, та для заміни уже застарілої Objective-C.

Swift - це мова програмування загального призначення, складена з багатьох парадигм, розроблена компанією Apple. та спільнотою з відкритим кодом. Swift працює з рамками Apple Cocoa та Cocoa Touch, а ключовим аспектом дизайну Swift стала здатність взаємодіяти з величезною кількістю існуючого коду Objective-C, розробленого для продуктів Apple останні десятиліття.

Swift скасовує використання вказівників Objective-C та інших небезпечних доступів, відкидає раннє застосування граматики у стилі Smalltalk у Objective C і повністю змінює її на позначення крапок. Водночас він надає простори імен, загальні дані та перевантаження операторів, подібні до тих, що є у C ++ та C#.

Свіфт [7] – це більш сучасна мова, з простішим синтаксисом. Більш того, визначено, що одна і та ж програма, яка написана на Swift, матиме лише 30% рядків коду порівняно з тією, що написана на Objective-C. Зменшення кількості коду не тільки прискорює процес розробки, але також призводить до меншої кількості помилок та більш швидкого тестування.

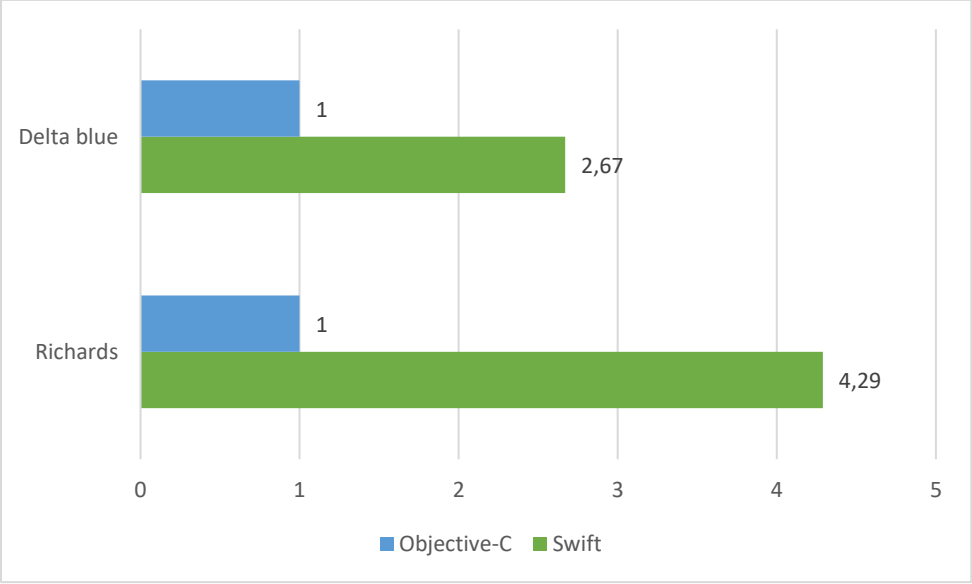


Рис. 1.9. Швидкість Swift проти Objective-C згідно з дослідженнями Apple

Простіший синтаксис та перевірка типів під час компіляції допомагають Swift бути кращою за Objective-C. Крім того, для оптимізації управління пам'яттю Swift використовує ARC (Automatic Counting Reference). В додаток, Swift підтримує динамічні бібліотеки, що також підвищують продуктивність програм.

Тому для створення мобільного додатку під платформу iOS, доцільно використовувати мову програмування Swift. Вона забезпечує більш стабільний, продуктивний та оптимізований підхід до розробки програмного забезпечення. А тому покращить швидкодію роботи додатку, його захищеність та простоту у використанні.

Написання коду на мові програмування найкраще проводити у спеціалізованому середовищі розробки. Гарним прикладом цього, є XCode. Це інтегроване середовище для розробки програмного забезпечення для macOS, iOS, watchOS та tvOS.

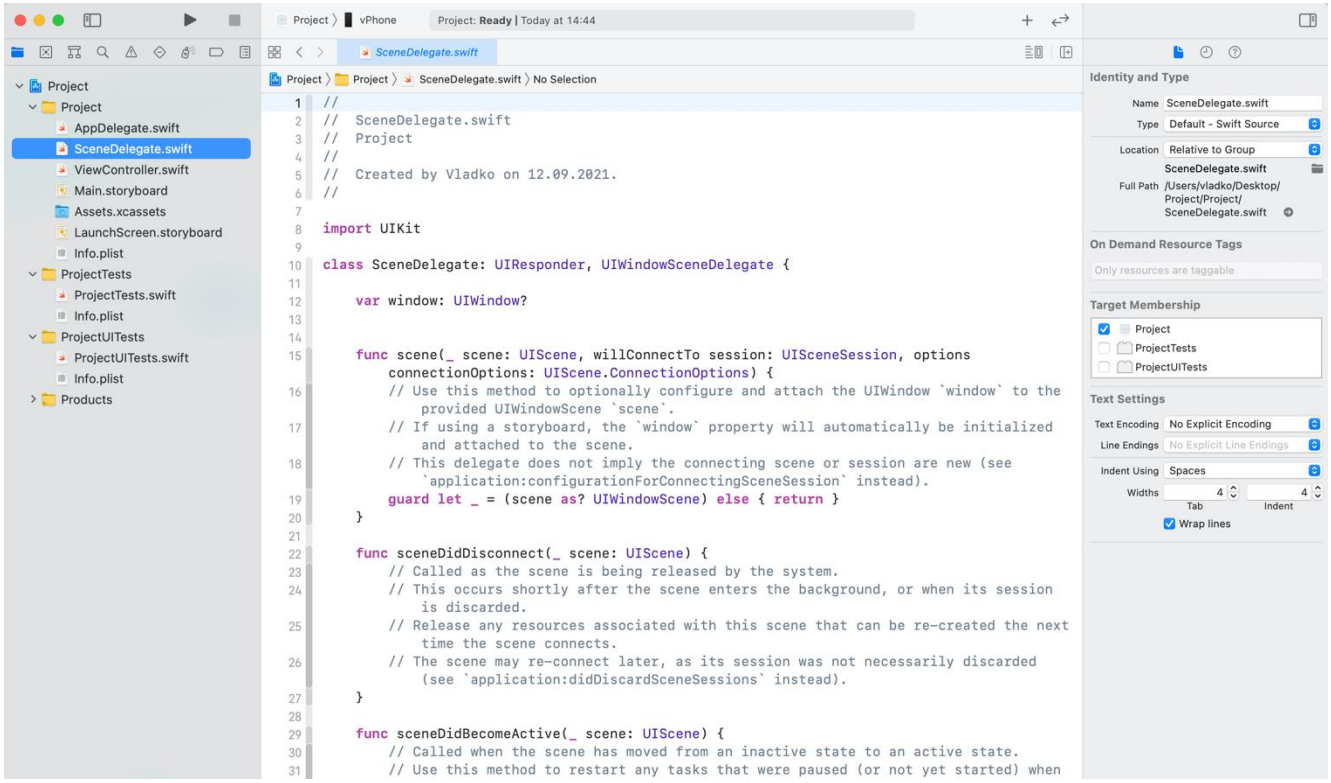


Рис. 1.10. Середовище розробки XCode

XCode [8] дозволяє редагувати проект, здійснювати пошук та навігацію по ньому, редагувати файли, створювати та налагоджувати всі типи проектів під час розробки, включаючи програми, інструменти, схеми, бібліотеки, плагіни, розширення ядра та драйвери пристроїв. XCode містить інструмент Interface Builder, який є графічним редактором для розробки компонентів інтерфейсу користувача.

Xcode має набір «розширень» для розробки, ці «розширення» називаються SDK і надаються компанією Apple. Компанії та незалежні розробники, належним чином зареєстровані в програмі розробників iOS для Apple під назвою iOS Developer Program, можуть розповсюджувати програми в магазині AppStore.

Так як, кількість кінотеатрів, вільних місць, список прем'єр та показів, а також їх розкладів – це значний об'єм даних, тому, є необхідністю їх впорядкувати у більш структурну форму. Для цього доцільно використати базу даних. Проте, знаючи, що інформація буде масштабуватись та змінюватись, а також, у майбутньому, використовуватись на платформі Android, потрібно реалізувати її у вигляді хмарного сховища. Найкращим варіантом для цього є сервіс Firebase.

Firebase [9] - це платформа для розробки та підтримки мобільних додатків. Вона надає інструменти розробки програмного забезпечення (SDK), призначеного для того, щоб дозволити розробникам легше та ефективніше надавати функції за допомогою інтерфейсів програмування на різних платформах.

Cloud Firestore - це серверна база даних сервісу Firebase, яка використовується для додавання, читання, оновлення та видалення даних. Вона заснований на NoSQL і розділена на колекції. Такі колекції містять документи, які, у свою чергу, містять власні колекції, які називаються під-колекціями, та різні поля даних. Усередині документів є поля, які містять усі види типів, такі як логічні, рядки, числа тощо.

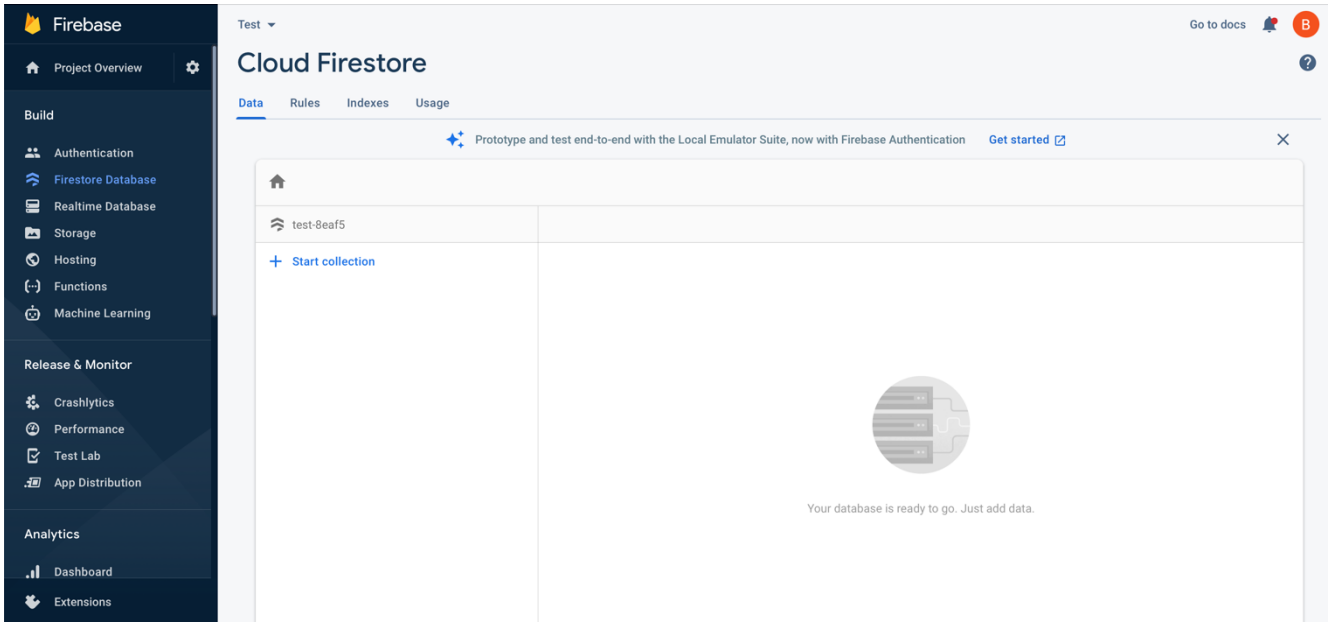


Рис. 1.11. Сервіс Firebase

ВИСНОВОК ДО РОЗДІЛУ 1

В цьому розділі було досліджено сучасні цифрові пристрої, визначено найбільш зручний пристрій для роботи. Проаналізовано розподіл сучасних мобільних операційних систем, сформовано їх переваги та недоліки. Було розглянуто використання існуючих додатків та визначено їх особливості.

Досліджено методи і засоби реалізації мобільних додатків на базі iOS. Визначено найзручнішу мову програмування, середовище розробки та сервіс для роботи з віддаленою базою даних.

РОЗДІЛ 2

АНАЛІЗ РЕАЛІЗАЦІЇ ОСНОВНИХ КОМПОНЕНТІВ ПРОГРАМНОГО ПРОДУКТУ

Кожна програма складається з сукупності команд, що взаємодіють одна з одною, на відповідно дозволених рівнях роботи системи. Кожна з них, виконує визначену функцію, наприклад збереження інформації у змінній, або ж формування елементу інтерфейсу. Ця сукупність, сформованих та працездатних команд, називається вихідним кодом програми. Саме він визначає, як додаток буде реагувати на ті чи інші дії користувача.

Проте зі збільшенням функцій додатку, екранів системи, масштабуванням, виникає проблема оптимізації. А саме, розробнику стає дедалі важче зрозуміти структуру програми, відловити проблему, що може виникнути, або ж, просто, додати новий функціонал в існуючу послідовність.

Отже, виходячи з цього, виникає необхідність упорядкувати вихідний код програми, сформувати структуру, визначити основні аспекти роботи системи та налаштувати взаємодію додатку з нею. Це дасть змогу, пришвидшити роботу розробника, покращити інтегрування нового функціоналу, полегшити знаходження вразливих місць та проблем у роботі системи, та покращити майбутню підтримку додатку. Тому аналіз реалізації, формування архітектури компонентів продукту, є важливою частиною циклу розробки програмного забезпечення.

2.1. Аналіз інформативності даних та їх використання

Екран додатку - це структура розташування компонентів інтерфейсу за

певними параметрами. Проте самого розташування елементів недостатньо, щоб задовольнити інформативність інтерфейсу. Для цього необхідно заповнити його вмістом. До інформаційного вмісту можна віднести, надписи елементів, опис отриманих даних, а також, зображення та відео, звукове та тактильне супроводження.

Більшість користувачів засвоюють інформацію читаючи текст опису, назву, або відгуки до вмісту. Проте, при великих обсягах, людина шукає ключові слова та вирази, для швидкого засвоєння інформації. Зважаючи на це, інтерфейс повинен задовольняти простоту та зрозумілість, давати змогу користувачу швидко засвоювати інформацію.

В свою чергу, використання зображень, не повинно відволікати людину від основного обсягу даних, а лише покращувати інформативність вмісту. Наприклад, це досягається шляхом використання невеликих за розміром зображень у певній виборці пошуку, або ж використання тільки відповідного зображення у детальному описі кіно-стрічки.

Відео та аудіо контент не повинен змушувати користувача, зупиняти його перегляд та прослуховування, а бути лише у вигляді допоміжного елементу. Також значне використання відео фрагментів у списку може спричини навантаження самого пристрою. Адже завантаження та відтворення відео-контенту затрачає достатню частину ресурсів.

До тактильного супроводження можна віднести технологію вібро-відгуку девайсу. Помірне використання цих елементів покращує досвід користувача, спрощує навігацію та відтворює псевдо звучання при відсутності аудіо. Проте, при імплементації вібро-відгуку у значній кількості може тільки погіршити зрозумілість системи.

Розроблений мобільний додаток містить реалізацію представлення списку фільмів. Кожен його елемент має достатній розмір зображення та підпис назви стрічки, що покращує навігацію екрану. Вікно детального перегляду представляє більш розгорнутий вигляд зображення, а також більш інформативну частину тексту його опису. При переході між екранами та використанні відповідних

елементів інтерфейсу можна відчути вібро-відгук, який забезпечує інтуїтивне орієнтуванні у навігації додатку.

2.2. Розробка архітектури додатку

Задля забезпечення продуктивності, оптимізації, швидкості роботи та лаконічності інтерфейсу програми, доцільно використовувати мову програмування Swift. Вона дає змогу швидко та результативно розробляти додатки на платформі iOS. Вона забезпечує захищеність даних користувача та швидку роботу системи.

Для розробки програмного забезпечення необхідно вибрати зручну та доступну архітектурний шаблон коду. Найпоширенішими такими представниками є MVC, MVP, MVVM, VIPER [10]. Розглянемо кожен окремо.

MVC (Model – View - Controller). Шаблон дозволяє нам сформувати проект структурним і простим у обслуговуванні. Усі файли в проекті знаходяться у відповідних папках або місцях, тому будь-який розробник може легко знайти файли та продовжити роботу з ним. MVC використовує для організації програмного проекту три частини: модель, представлення і контролер.

Модель — це компонентом зберігання даних шаблону, який використовує структуру даних. Він також безпосередньо керує правилами та даними програми.

Представлення — це інтерфейс або його елемент, за допомогою якого користувач взаємодіє з програмою.

Контролер використовується як міст між моделлю та представленням. Тому і модель, і представлення можуть взаємодіяти один з одним лише через контролер.

Розглянемо роботу шаблону MVC. Коли користувач натискає на кнопку, він сповіщає контролер, який в свою чергу перейде до компонента моделі і змусить компонент моделі виконати відповідну дію. Після чого компонент інформує контролер, який відповідно оновлює представлення.

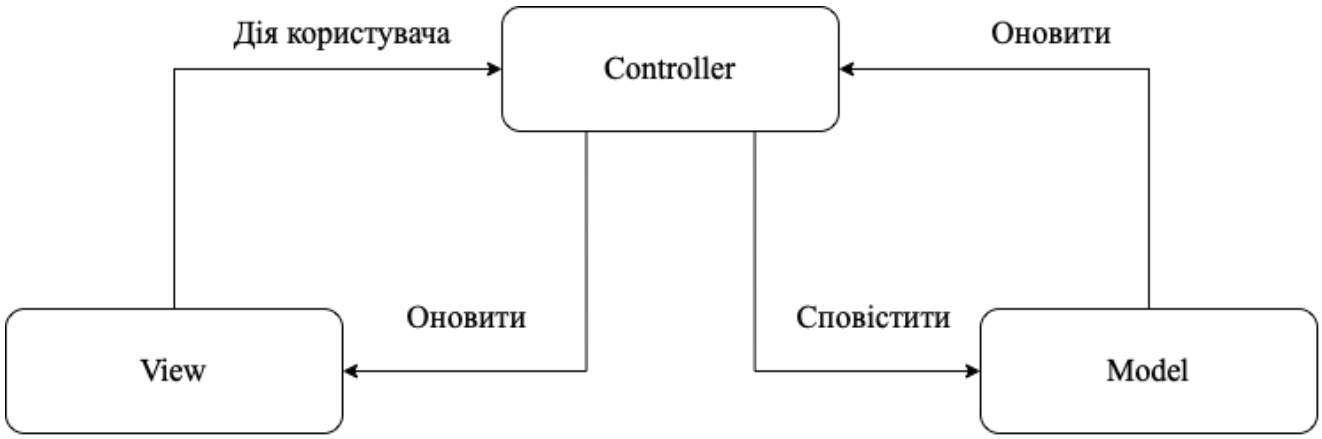


Рис. 2.1. Модель роботи шаблону MVC

До переваг MVC можна віднести простоту роботи та мінімальну кількість файлів для підтримки. Недоліками цього шаблону є складність тестування та складність реалізації на великих обсягах коду.

MVP (Model – View – Presenter). Цей патерн також складається з трьох компонентів – моделі, представлення та доповідача. У шаблоні проектування MVP контролер (у MVC) замінений доповідачем.

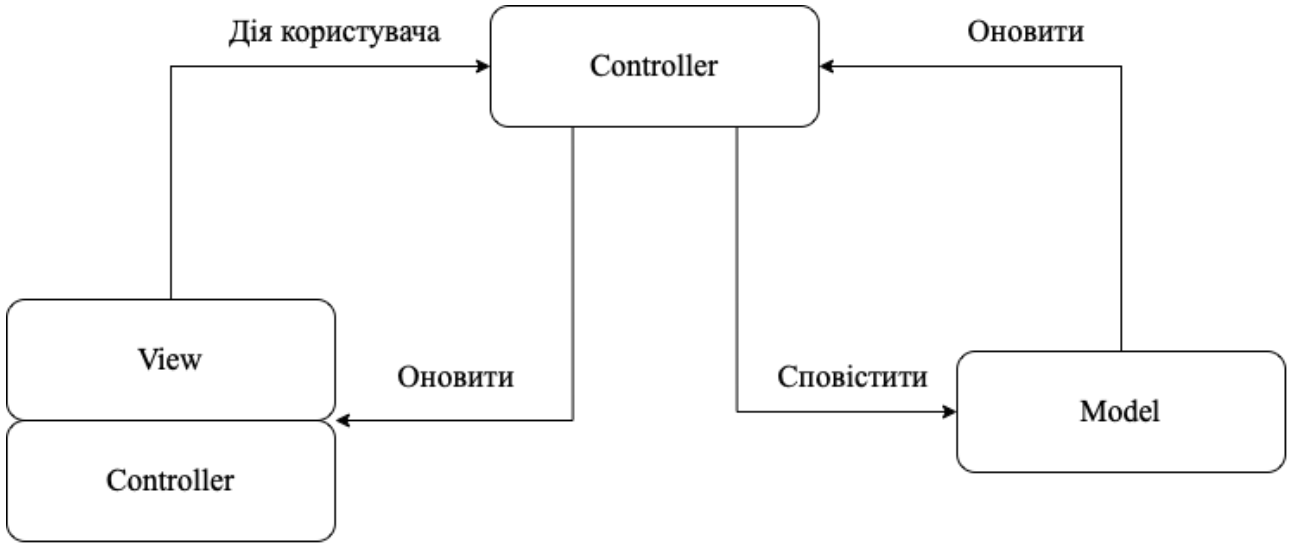


Рис. 2.2. Модель роботи шаблону MVP

На відміну від шаблону проектування MVC, Presenter повертається до подання. У шаблоні проектування MVP доповідач керує моделлю, а також

оновлює подання. Завдяки цьому реалізація представлення та модульне тестування стають легшими.

До переваг MVP можна віднести простоту роботи та тестування, поступове масштабування усіх компонентів архітектури. Недоліками цього шаблону є складність пере використання та більшу кількість файлів.

MVVM (Model – View – ViewModel). MVVM є покращеним варіантом дизайну MVC, а ViewModel в MVVM використовується для спрощення та розділення роботи з інформацією. У MVVM вся логіка роботи зберігається в ViewModel, а представлення повністю ізольоване від моделі.

У шаблоні проектування MVVM представлення є активним і містить інформацію про поведінку, події та інформацію про прив’язку даних. Модель перегляду в MVVM відповідає за розділення презентації та надає методи та команди для керування станом подання та керування моделлю.

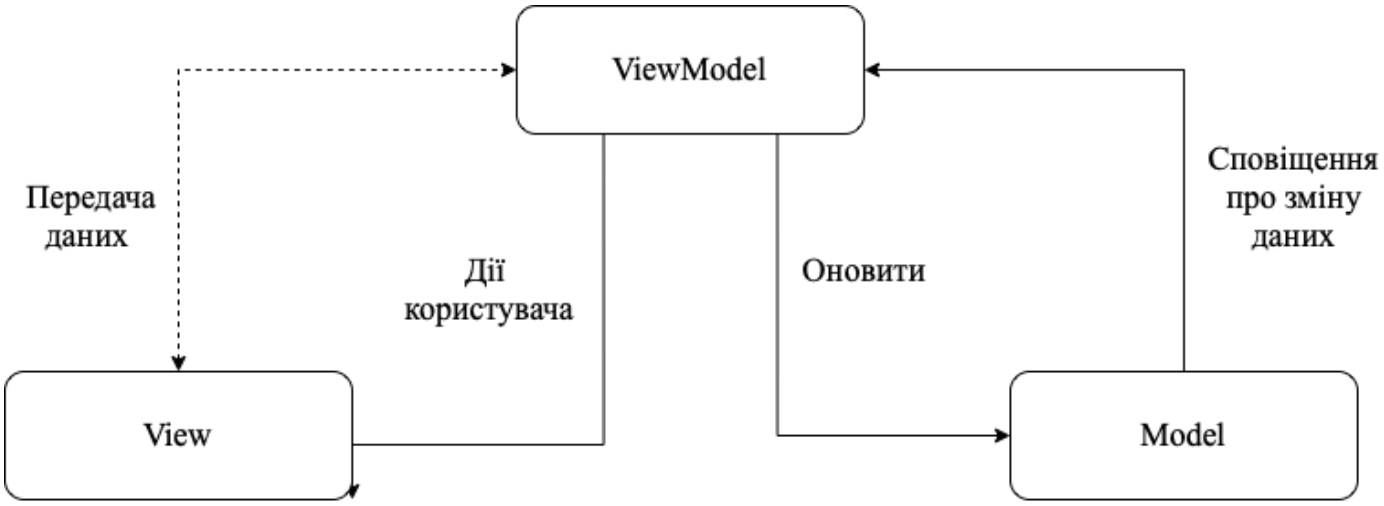


Рис. 2.3. Модель роботи шаблону MVVM

До переваг MVVM можна віднести простоту слідкування за даними, тестування, представлення та модель – повністю незалежні, можливість пере використання файлів. Недоліками цього шаблону є складність реалізації.

VIPER (View – Interactor – Presenter – Entity - Routing). Цей паттерн часто відносять до чистої архітектури. Він дійсно представляє сегментований спосіб

розподілу обов’язків, добре поєднує модульним тестування і робить код більш придатним для повторного використання.

Порівняно з MVVM, Viper має кілька ключових відмінностей у розподілі обов'язків. Він представляє маршрутизатор, який відповідає за навігацію, видаляючи його з перегляду. Сутності є простими структурами даних, які передають інтерактору логіку доступу, яка зазвичай належить моделі. А обов’язки ViewModelController розподіляються між інтерактором і доповідачем.

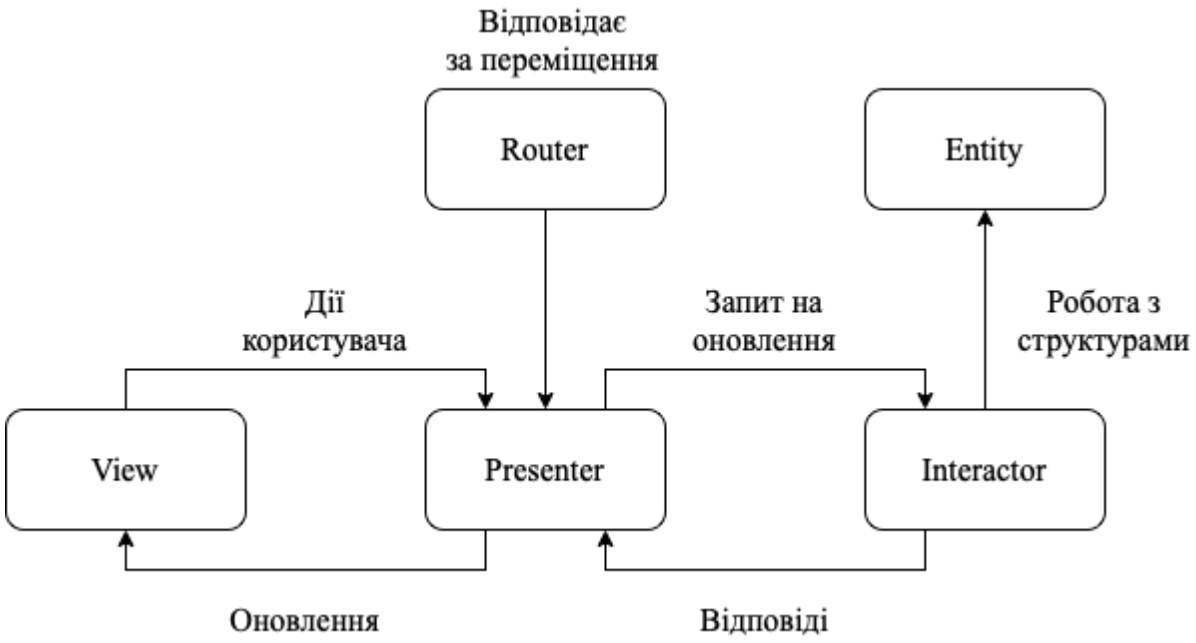


Рис. 2.4. Модель роботи шаблону VIPER

До переваг VIPER можна віднести, простоту модульного тестування, повністю незалежні компоненти, можливість пере використання файлів. Недоліками цього шаблону є складність реалізації, та велика кількість файлів.

Отже розглянувши найбільш поширені паттерни формування коду програми, можна визначити, що MVC досить зручно використовувати в незначних та невеликих програмах, MVP – використовується для реалізації, більш простого до тестування, додатку, а MVVM та VIPER – для роботи з великими програмами, що містять складний функціонал. Тому, розуміючи, що кількість екранів програмного продукту досить велика, буде зручно використовувати архітектурний шаблон MVVM.

2.3. Аналіз та вибір бібліотек

Для спрощення розробки програмного забезпечення, часто використовують стороні бібліотеки коду. Це дає змогу покращити роботу та спростити реалізацію додатку. Крім цього, для використання певних функцій сторонніх сервісів, необхідною умовою є використання їхнього коду роботи з системо. Тому використання бібліотек є звичайною практикою у реалізації програми.

Для підключення сторонніх бібліотек у проект XCode, можна використовувати функціонал CocoaPods [11]. Це система керування залежностями на рівні програми для Objective-C, Swift. Вона Забезпечує стандартний формат для керування зовнішніми бібліотеками, а також зосереджується на розповсюдженні стороннього коду на основі автоматичної інтеграції в проекти XCode.

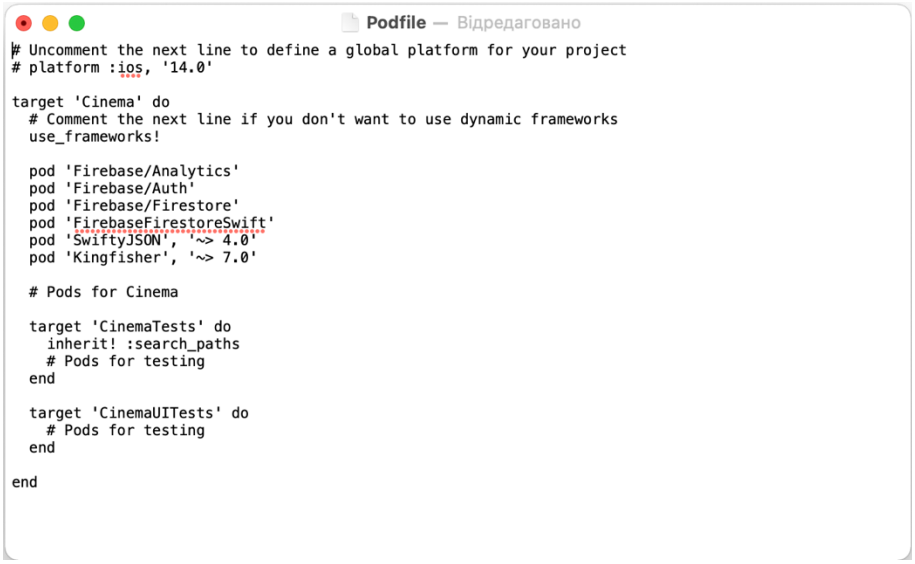


Рис. 2.5. Файл Pods з використаними бібліотеками

CocoaPods запускається з командного рядка, а також інтегрується в інтегроване середовище розробки XCode, JetBrains або AppCode. Для інсталяцій з багатьох різних джерел «головне» сховище специфікацій (включаючи метадані для багатьох бібліотек з відкритим кодом) та керується як репозиторій Git і розміщується на GitHub.

Аналітика, робота з базою даних та сесійна активність користувача відбувається за використання сервісу Firebase. Це служба від Google, яка використовується для полегшення розробники додатків. Завдяки Firebase розробники можуть зосередитися на реалізації програмного забезпечення, не докладаючи особливих зусиль. Дві цікаві функції Firebase – це віддалена конфігурація Firebase та база даних Firebase Firestore. Крім того, існують допоміжні функції для програм, які потребують сповіщень, а саме Firebase Cloud Messages.

Для роботи з аналітикою використовувався сервіс Firebase Analytics. Це безкоштовне рішення для дослідження додатків, яке надає уявлення про використання і залучення нових користувачів. Звіти Analytics допомагають чітко зрозуміти, як поводяться користувачі, що дає змогу приймати зважені рішення щодо маркетингу додатків та оптимізації ефективності. До списку Pods входить – “Firebase/Analytics”.

Для роботи з базою даних використовується сервіс Firestore. Його розроблено для збереження та відображення створеного користувачами вмісту, наприклад фотографії чи відео. Розробники можуть використовувати його для зберігання вмісту безпосередньо з використання клієнтської бібліотеки. Він забезпечує віддалене збереження інформації на серверах Google. До списку Pods входить – “Firebase/Firestore” та “FirebaseFirestoreSwift”

Більшість програм повинні ідентифікувати користувача. Знаючи особу користувача, програма може безпечно зберігати його дані в хмарі та забезпечувати однаковий персоналізований досвід на кожному пристрої.

Тому для авторизації користувача було використано сервіс Firebase Authentication. Це системна служба аутентифікації реалізує код на стороні клієнта, щоб користувачі могли зареєструватися та ввійти в програми через стороні сервіси – Facebook, GitHub, Twitter і Google. Крім того, Firebase підтримує керування користувачами. Це дає розробникам можливість увімкнути авторизацію користувачів за допомогою логінів електронної пошти та паролів, що зберігаються у на серверах Google. До списку Pods входить – “Firebase/Auth”

Для формування даних, а також їх отримання необхідно зберігати та передавати інформацію у вигляді словників даних. Це ускладнює розробку та чистоту коду. Тому для вирішення цієї проблеми та уніфікування інформації було вирішено використовувати формат JSON. Це текстовий формат, зрозумілий для людей, який використовується для представлення об’єктів та інших структур даних і в основному – для передачі структурованих даних по мережі, процес називається серіалізацією. JSON є найпростішою, найлегшою альтернативою XML.

Для спрощення кодування та декодування даних JSON, було використано бібліотеку SwiftyJSON. Вона реалізує безпечне конвертування текстового формату JSON у вигляд типових змінних мови Swift. Це забезпечує збереження простоти коду, а також більш інформативне представлення даних у структурі програми. До списку Pods входить – “SwiftyJSON”

Збереження медійних даних може займати значну частину сховища даних. А тому доцільно використовувати завантаження зображень по посиланню. Проте, роблячи це синхронно у головному потоці роботи програми, ми створюємо затримку та блокування інтерфейсу користувача. Тому більш доцільно використовувати асинхронне завантаження, яке не буде впливати на роботу людини у програмі. Для такої реалізації, було використано бібліотеку Kingfisher, що уже має реалізовані елементи зображень з відповідним функціоналом. До списку Pods входить – “Kingfisher”.

Для більш продуктивної роботи шаблону проектування MVVM доцільно використовувати реактивне програмування. Це декларативний шаблон програмування, який має справу з потоком даних і поширенням змін у ньому. За допомогою цього шаблону легко представляти статичні (наприклад, масив) або динамічні потоки даних (наприклад, випромінювачі подій), а також повідомляти, що існує залежність. Для його реалізації на базі SwiftUI доцільно використовувати нативну бібліотеку Apple – Combine. Структура Combine надає декларативний API Swift для обробки значень та їх змін у часі. Вони можуть представляти багато видів асинхронних подій. Combine оголошує, що Publishers отримують значення,

які можуть змінюватися з часом, а Subscribers отримують ці значення від Publishers. Для підключення бібліотеки достатньо скористатися командою import.

2.4. Огляд та формування моделей даних

Формування бази даних та робота з нею націлена на зберігання і оперуваннями масивами даних. А для їх упорядкування потрібно визначитися зі структурою моделей даних. Необхідно сформувати основні елементи інформації та поля у цих таблицях.

Перш за все, додаток організації роботи з мережами кінотеатрів, повинен оперувати інформацією про самі кінотеатри, наявні кінострічки та категорії до яких вони відносяться, а після покупки білету – і інформацією про нього. Тому визначимо такі основні структури даних:

- CinemaModel – модель кінотеатру;
- MovieModel – модель кінострічки;
- GenreModel – модель жанру кінострічки;
- TicketModel – модель білету.

Для оперуванням списків кінотеатрів та перегляду наявних кіносеансів у них потрібно використовувати модель CinemaModel. Тому для відображення усіх параметрів доцільно використовувати наступні поля:

- ідентифікатор – унікальний текстовий рядок;
- назва – текстова назва закладу;
- зображення – текстове посилання на зображення закладу;
- розташування – текстова адреса кінотеатру;
- координати – текстові координати довготи та широти кінотеатру;
- елемент фільмів – список наявних показів фільмів.

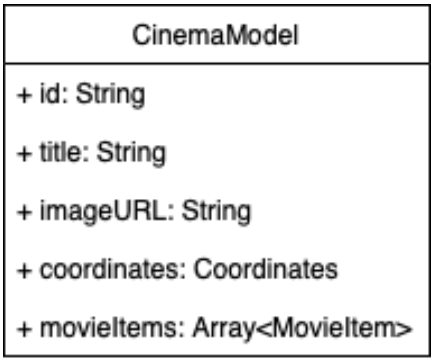


Рис. 2.6. Модель CinemaModel

Координати представлені у вигляді структури LocationModel. Вона має два текстових поля:

- довгота – поле довготи типу Double;
- широта – поле широти типу Double.

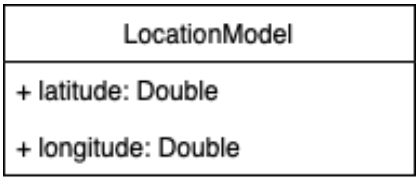


Рис. 2.7. Модель LocationModel

Елемент фільму, використовується для перегляду наявних сеансів та часу у поточному кінотеатрі, визначення ціни та кількісті вільних місць. Ця структура має наступні поля:

- ідентифікатор – унікальне текстове поле;
- ціна – числове поле ціни білету;
- сеанси – список текстових часів у форматі Timestamp;
- місця кінотеатру – модель розташування кінотеатру.

Timestamp - час Unix, який вказує кількість секунд з 1 січня 1970 року 00:00 UTC без високосних секунд.

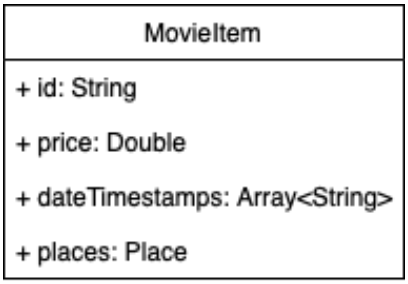


Рис. 2.8. Модель MovieItem

Модель Place використовується для визначення кількості місць по горизонталі та вертикалі у кінозалі. Відповідно ця модель має наступні поля:

- вертикальне значення – кількість рядів;
- горизонтальне значення – кількість місць у ряді.

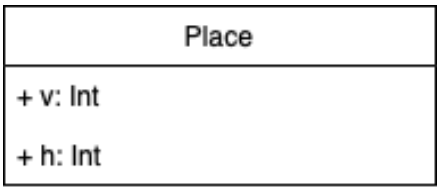


Рис. 2.9. Модель Place

Для оперування списком наявних кінострічок, або майбутніх прем’єр, їх пошуку, отримання по жанру, а також перегляду деталей фільму і замовлення квитка виникла необхідність сформувати окрему модель кінострічки. Вона містить наступні поля:

- ідентифікатор – унікальне текстове поле;
- назву – текстове поле назви фільму;
- опис – текстове поле опису фільму;
- зображення – текстове посилання на зображення стрічки;
- оцінка – оцінка фільму типу double;
- дата прем’єри - текстовий час у форматі Timestamp;
- студії – список текстових назв студій фільму;
- жанри – список ідентифікаторів жанру.

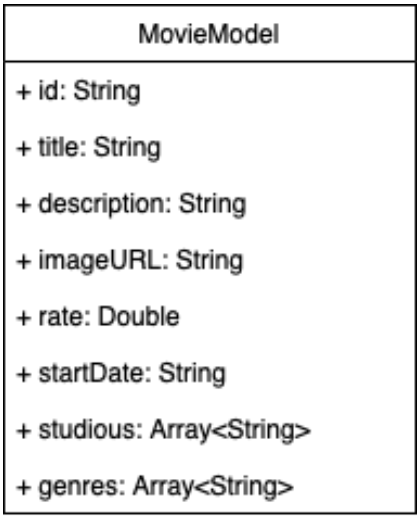


Рис. 2.10. Модель MovieModel

Список жанрів використовується для визначення відповідної категорії фільму а також пошуку стрічок за цим типом. Для цього у моделі фільму присутній список ідентифікаторів жанрів, до яких він відноситься. В загальному модель жанру має вигляд:

- ідентифікатор – унікальне текстове поле;
- назва – текстове поле назви жанру;
- емоджі – текстове відображення жанру.

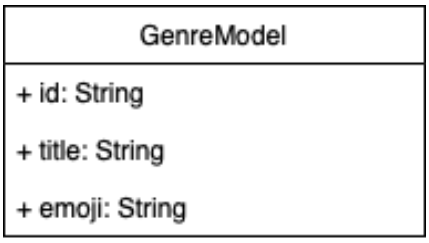


Рис. 2.11. Модель GenreModel

Під час проходження покупки нам необхідно зібрати інформацію про вибраний фільм, кінотеатр, відповідний час показу. Тому доцільно сформувати структуру для збереження цих даних до моменту покупки білету. Модель покупки виглядає наступним чином:

- ідентифікатор фільму – ідентифікатор вибраного фільму;
- ідентифікатор кінотеатру - ідентифікатор вибраного кінотеатру;
- ідентифікатор елемента фільму - ідентифікатор вибраного елементу фільму;
- час показу – вибраний час показу.

Date – нативний тип даних, що використовує Swift.

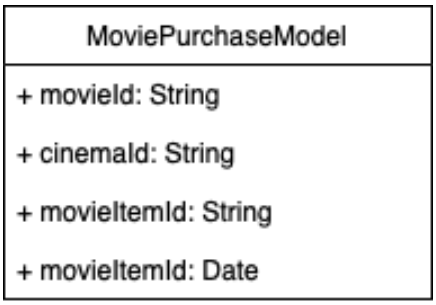


Рис. 2.12. Модель MoviePurchaseModel

Після оформлення і проходження покупки необхідно зберігати білет користувача, для подальшого його використання. Тому доцільно створити модель для купленого елементу. Вона має наступні поля:

- ідентифікатор – унікальне текстове поле;
- ідентифікатор кінотеатру - ідентифікатор вибраного кінотеатру;
- ідентифікатор фільму - ідентифікатор вибраного фільму;
- час - текстовий час у форматі Timestamp;
- ціна - числове поле ціни білету;
- місце – модель розташування місця;
- статус деактивації – перевірка деактивації білету.

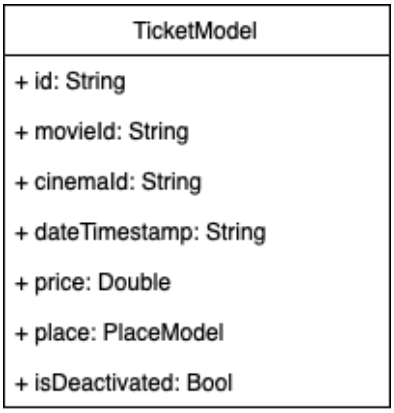


Рис. 2.13. Модель TicketModel

Модель Place використовується для визначення розташування місця по горизонталі та вертикалі. Відповідно ця модель має наступні поля:

- ряд – числове значення відповідного ряду;
- секція – числове значення відповідної секції.

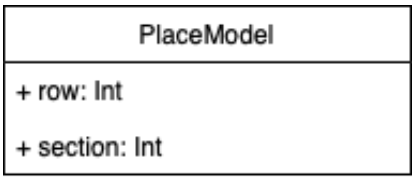


Рис. 2.14. Модель PlaceModel

2.5. Розробка структури бази даних

База даних відіграє важливу роль у роботі додатку. Адже завантаження інформації, її структуризація, оформлення та повторне використання є невід'ємною частиною роботи будь-якої програми. Тому правильне формування основи БД вплине на подальшу швидкодію та продуктивність продукту.

Як уже зазначалось в розділі, для роботи з БД був використаний сервіс Firebase. Firestore є базою даних, що містять документи, та являється NoSQL форматом [12]. Вона створена для високої продуктивності, автоматичного масштабування та простоти розробки і підтримки додатків. Firestore, як база даних NoSQL, відрізняється від традиційних тим, як описує відносини між

об’єктами даних. Проте інтерфейс він також має багато тих же функцій, що й звичайні бази даних.

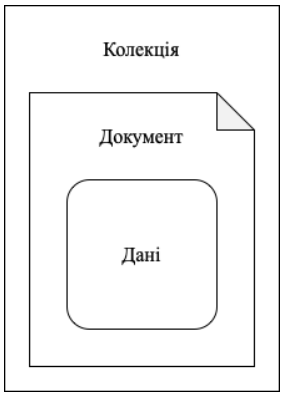


Рис. 2.15. Структура розташування файлів

Firestore може містити два формати файлів – колекцію, або документ. Кожен з цих елементів формує свій рівень у базі.

Документ – це одна із основних одиниць даних Firestore. Кожен такий елемент містить унікальну назву, та не може бути повторно продубльований.

Всі документи повинні зберігатися в колекціях. Документи можуть містити під колекції та вкладені об’єкти, обидва з яких можуть включати примітивні поля, як-от рядки, або складні об’єкти, як-от списки. Він може включати поля відповідно наступні типи:

- string – текстовий тип;
- number – числовий тип;
- boolean – булевий тип.

Крім звичайних, слід розглянути такі як map, array, null, timestamp, geopoint.

- map – словниковий тип;
- array – тип списку;
- null – пустий тип;
- timestamp – тип часової позначки;
- geopoint – тип координатної позначки.

Map – тип даних масиву карт, де пара ключ-значення зберігається в масиві. «Ключ» — це ідентифікатор певного типу даних, а «значення» — вміст, який ідентифікується або зберігається. Кожен ключ унікальний, тому зберігати

дублікати неможливо. Відповідно значення можуть містити усі визначені вище типи. А отже, його зручно використовувати для вкладення нових структур інформації.

Array – це тип даних, об’єктами якого є набори компонентів. Вони доступні за допомогою одного або кількох індексів (також званих ключами), обчислюваних під час виконання програми. Об'єкти масиву типу даних називаються "змінною масиву", "значенням масиву" - але найчастіше просто "масивом.

Null – це тип без посилання на значення або об’єкт, поведінка якого визначена як нейтральна (нульова). Шаблон проектування нульових об’єктів описує використання цих об’єктів та їх поведінку. Наприклад, функція може витягти список файлів з папки і виконати дію над кожним з них. У випадку порожньої папки однією з можливих відповідей є кинути виняток або повернути нульове посилання, а не список.

Timestamp – це тип закодованої інформації, яка визначає, коли сталася подія, з часткою секунди, як дата і час. Цей термін походить від проставлення дати та часу на вхідних документах або інших документах. У Unix форматі, цей тип вказує кількість секунд з 1 січня 1970 року 00:00 UTC без високосних секунд.

Geopoint – це тип даних, що подібний до Map, проте з завчасно визначеними ключами – довготи та широти. Значення, які можуть знаходитись під цими ключами будуть містити числовий тип. Таким чином спрощується логіка обробки даних та зникають проблеми із зміною значень.

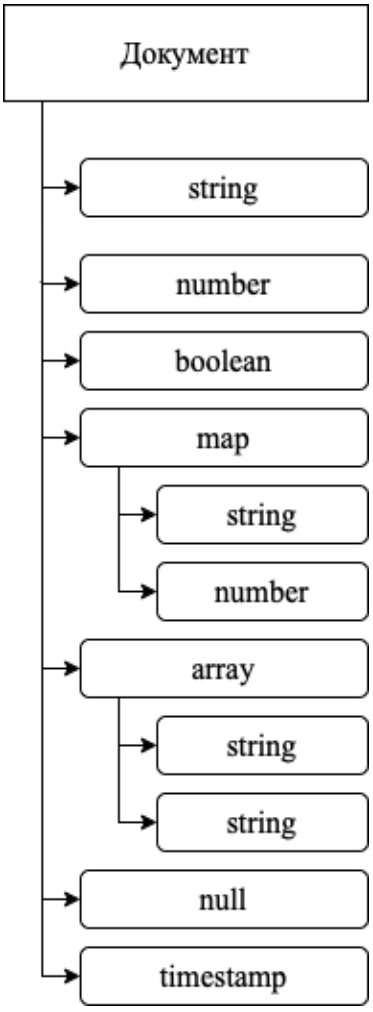


Рис. 2.16. Структура документу

Другим форматом файлів, що може містити Firestore є колекції. Кожен документ повинен знаходитись в колекції. Тому її можна визначити як контейнер для файлів. Наприклад ми можемо мати колекцію “полиця”, документами на ній будуть “книги.

Кожен документ в одній і тій же колекції можуть містити різні поля, або зберігати різні типи даних у них. Колекція містить документи і нічого більше. Вона не може містити поля зі значеннями, або інші колекції. Тому початковим елементом структури будь-якої бази даних Firestore є колекція.

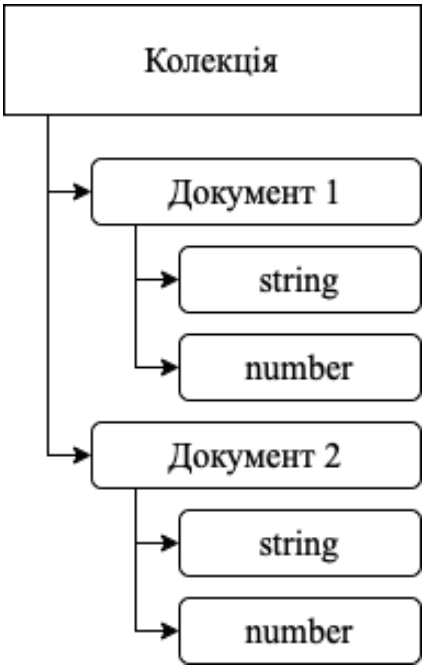


Рис. 2.17. Структура колекції

Розробимо структуру бази даних для роботи з додатком. Як було визначено раніше, нам необхідно зберігати чотири основних моделей бази даних. До першої ми відносимо модель кінотеатру. Це спричинене тим, що нам необхідно проводити пошук наявних кінострічок, а також давати можливість користувачеві переглядати ціни, рейтинги а також купувати самі білети у кіно.

Так як, кінотеатрів у нас може декілька, потрібно створити колекцію “cinemas”, яка буде містити цей список.

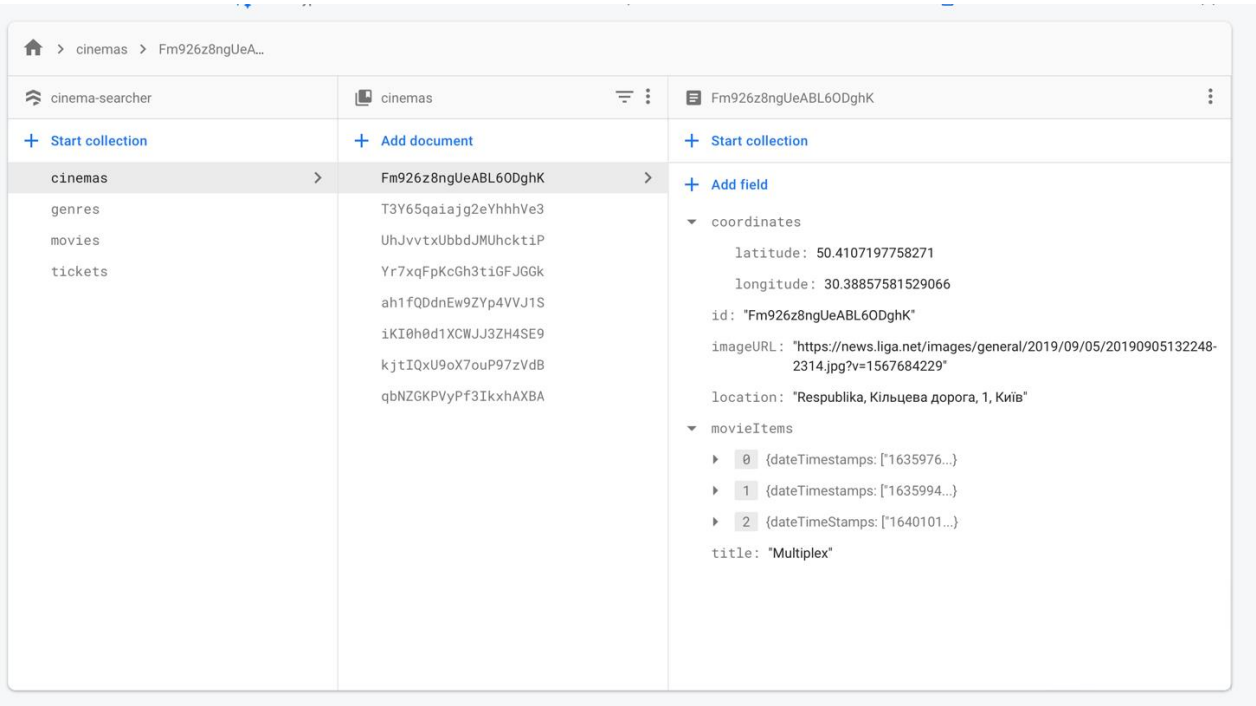


Рис. 2.18. Структура колекції та документів кінотеатрів

У ній знаходяться документи, що містить структуру кінотеатру визначеної, раніше, моделі. Поле “movieItems” буде містити список словників, які відповідають моделі “MovieItemModel”.

Звичайно, кожен кінотеатр відтворює певну кінострічку. Розуміючи, що фільм є незалежною одиницею, необхідно сформувати окрему колекцію “cinemas”. Так само, як і у минулому списку, кожен документ відповідає окремій, розглянутій раніше, структурі “MovieModel”. Де поля “genres” та “studious” – це масиви текстових значень.

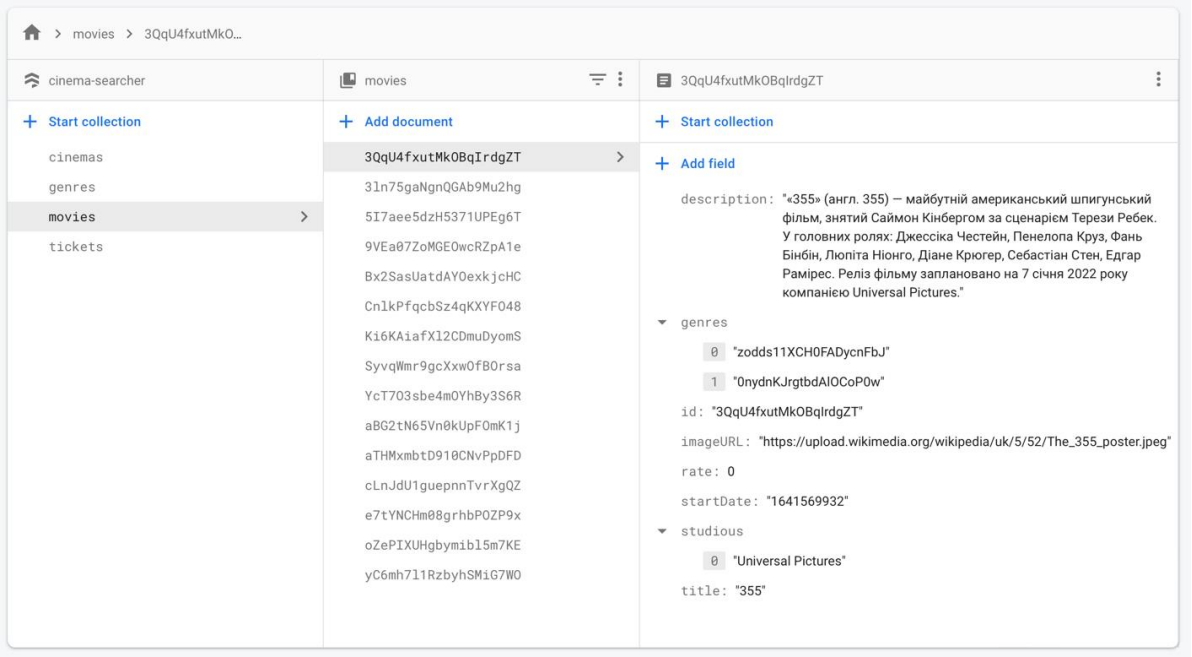


Рис. 2.19. Структура колекції та документів кінострічок

Крім цього, є необхідність у створені списку жанрів, до яких відносяться фільми. Тому було реалізовано окрему колекцію “genres” зі списком документів. Кожен елемент цієї колекції відповідає структурі “GenreModel”.

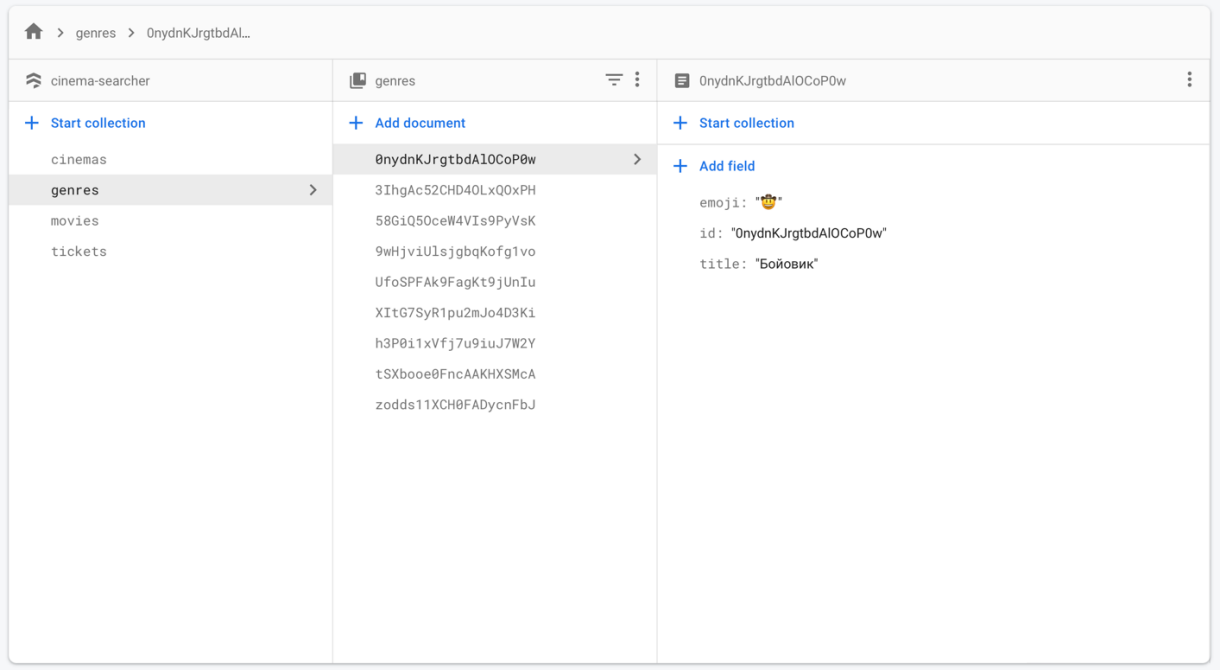


Рис. 2.20. Структура колекції та документів жанрів

Після того, як користувач купує білет, необхідно його зберігати. А отже, створимо колекцію “tickets”, де буде знаходитись файл “users”.

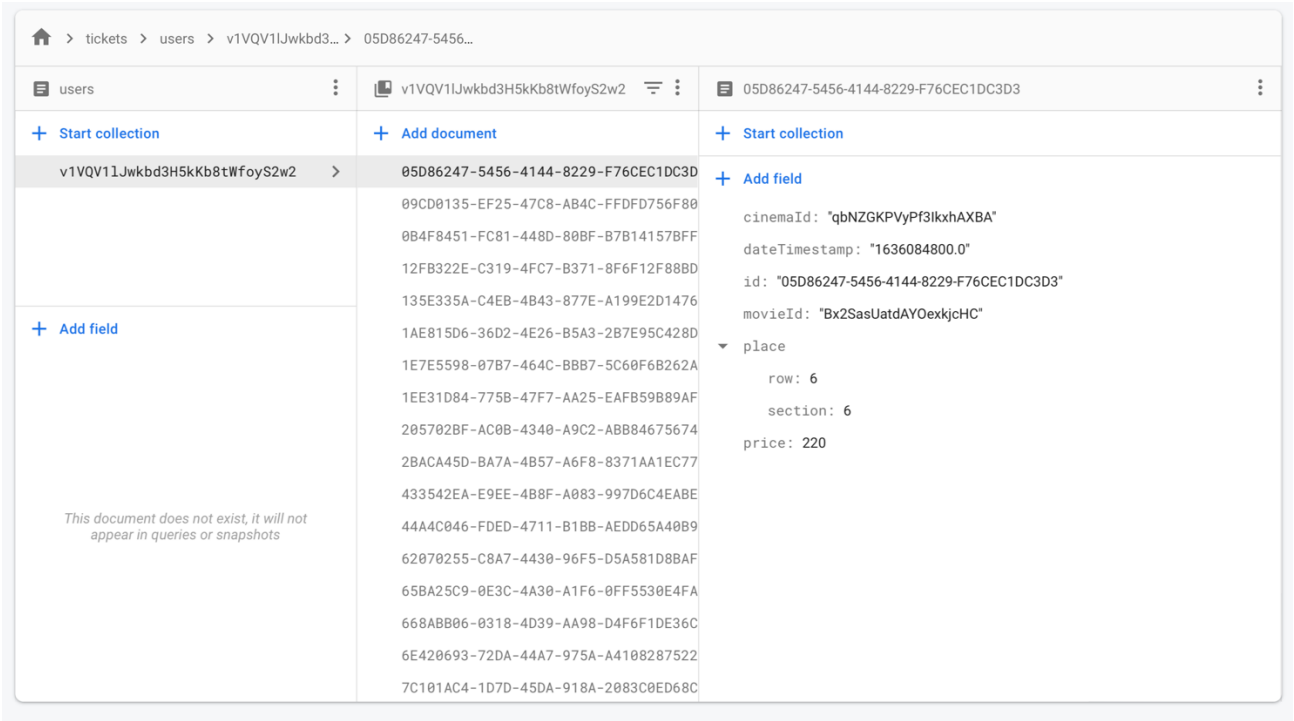


Рис. 2.21. Структура колекції та документів білетів

Файл буде містити окрему колекцію документів з ідентифікатором користувача. Це дає змогу додавати нові білети без перемішування їх між користувачами, а також пришвидшує читання самого списку. Тому елементами будуть документи, що формують структуру моделі “TicketModel”. Де поле “place” відповідає за модель “PlaceModel”.

ВИСНОВОК ДО РОЗДІЛУ 2

В цьому розділі було проаналізовано інформативність даних та розглянуто їх використання при роботі з мобільним додатком. Досліджено і розроблено найбільш оптимальну, відповідно до завдання, архітектуру системи.

Було визначено бібліотеки, для спрощення розробки програмного забезпечення. Оглянуто та сформовано моделі даних для роботи додатку та структуризації інформації. Досліджено і оцінено продуктивність роботи Firestore та розроблено структуру бази даних.

РОЗДІЛ 3

РОЗРОБКА ПРОГРАМНОЇ РЕАЛІЗАЦІЇ РОБОТИ ДОДАТКУ

Робота будь-якого додатку залежить від великою кількості досить простих задача. Саме ця сукупність і є основою для програмного коду. Проте, недостатньо визначити і написати команди, необхідно придати їх певної логіки та структури. Тому, досить часто, однакові за формою виконання частини коду об’єднують у певні блоки програм. Кожна з яких виконує відведену їй функцію та відповідає за визначену область роботи програми.

Для прикладу, можна привести модуль роботи з сервером. Його головна мета – це відправити запит та отримати відповідь. Після цього ця відповідь буде оброблена програмою та надана користувачеві. До блоку може входити запит відправки запиту, його обробки, генерації помилки, оновлення ключа авторизації та інше.

Виходячи з цього, стає зрозумілою необхідність до структуризації завдань, які необхідно виконувати, по відповідним модулям програми. Адже така реалізація буде сприяти продуктивності роботи, швидкодії додатку, кращої сумісності та тестованості у майбутньому.

У додатку для організації роботи з мережами кінотеатрів можна виділити п’ять основних модулів коду:

- модуль роботи з базою даних;
- модуль автентифікації користувача;
- модуль роботи з налаштуваннями;
- модуль оплати;
- модуль формування QR-коду

3.1. Реалізація модуля роботи з базою даних

Для роботи з базою даних Firestore необхідно підключити відповідну бібліотеку Firebase до проекту. Для цього ми використовуємо відповідну команду “import [назва бібліотеки]”.

На початку роботи стороннього коду необхідно ініціалізувати середовище. Це можна зробити виконуючи відповідну команду на початку роботи додатку. Тому, для спрощення та можливості використання в будь-якому місці коду, ми створимо відповідний клас “FirebaseService”.

Розглянемо детальніше елементи класу. Для отримання доступу з частини коду програми ми використовуємо Singletone. Це шаблон проектування, який використовується в об’єктно-орієнтованому програмуванні. Він гарантує, що в системі в будь-який момент часу існує лише один екземпляр об’єкта. Singletone корисний для таких речей, як реєстратори, диспетчери станів та інші об’єкти, які мають бути глобально доступними через програму.

Далі ми створюємо функцію “configure”, що викликає команду для ініціалізації та налаштування Firebase. Це необхідно для коректної роботи бібліотеки.

В загальному внутрішня взаємодія з Firestore розподілена на три етапи, кожен з яких спрощує використання наступного. Така реалізація дає змогу декомпозувати складну задачу на більш прості та мати можливість модульного тестування додатку.

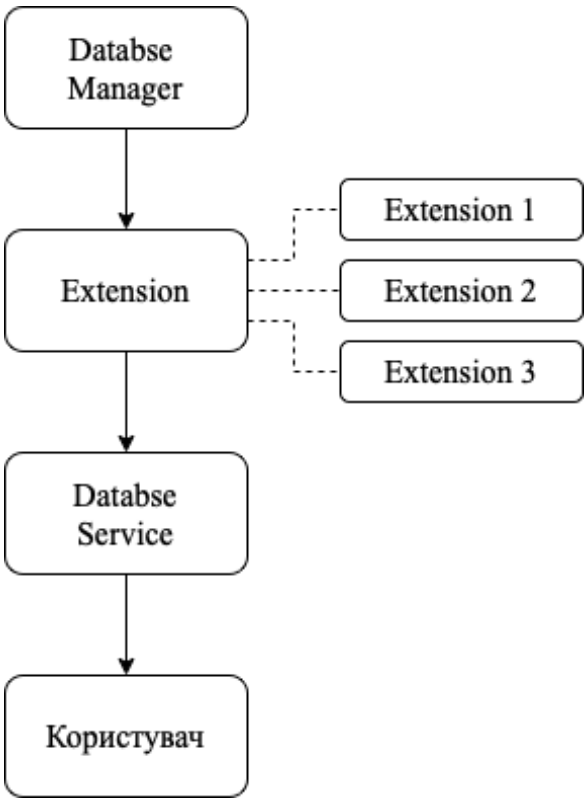


Рис. 3.1. Структура модулю роботи з БД

3.1.1. Реалізація ланки менеджера роботи з базою даних

Першою ланкою, яка напряму працює з бібліотекою, є “DatabaseManager”. Він надає можливість наступним ланкам уніфіковано використовувати поставлені запити до відповідних структур даних. Задля забезпечення структурованості файл розділений на три частини:

- ініціалізація;
- запис;
- читання.

Ініціалізація представляє з себе змінну, в яку записується об’єкт взаємодії з Firestore. Це відбувається при кожному створені класу “DatabaseManager”.

Запис складається з функції, що повертає викликає один із двох блоків. Перший у випадку успішного оброблення результату, а інший – в разі невдачі, з відповідною помилкою. Кожне виконання функції використовує відповідний шлях до файлу, або до колекції файлів у сервісі Firestore.

Для визначення документу ми використовуємо ключову команду “document([назва])”, а для колекції - “collection([назва])”. Після чого перевіряємо наявність помилок у відповіді бібліотеки, та повертаємо отриманий результат.

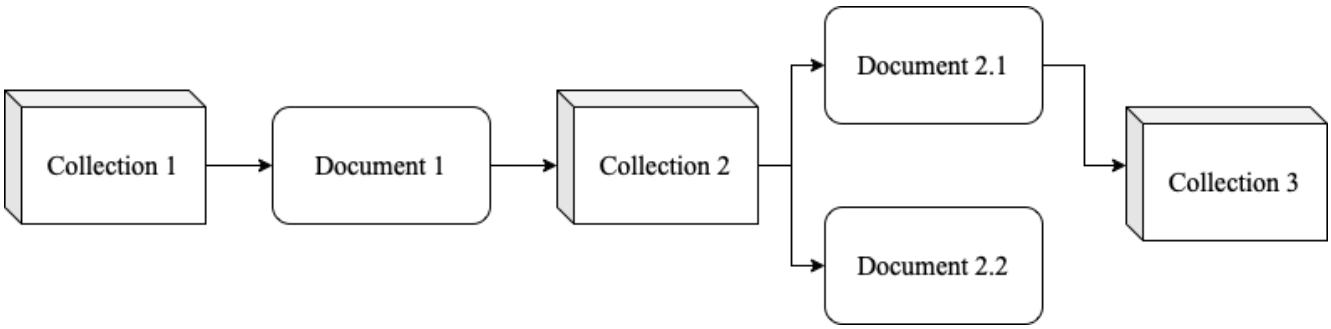


Рис. 3.2. Приклад формування шляху до колекції

Для отримання списку документів ми, по аналогією з функцією читання, вказуємо відповідно блок успішного виконання, та помилки. Після цього, використовуючи, раніше ініціалізовану, змінну “database”, та вказуючи шлях до колекції, читаємо вміст. У разі успішного виконання – конвертуємо кожен елемент списку у вигляді словника в JSON формат, для подальшого його використання. А у разі помилки – повертаємо її у блоці “failure”.

Структура та усі наявні функції менеджера модуля роботи з базою даних представлено нижче.

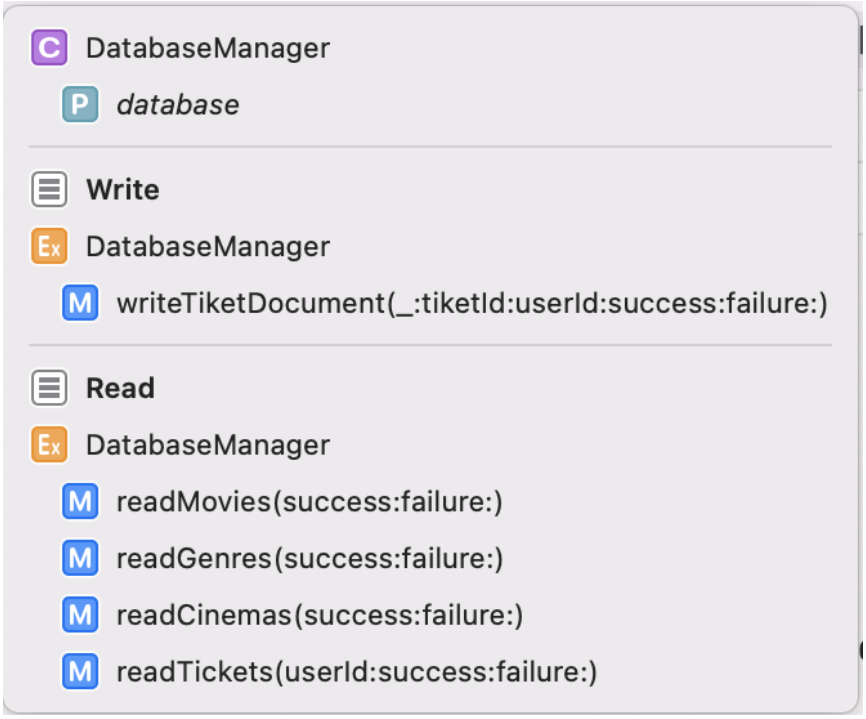


Рис. 3.3. Структура менеджера модуля роботи з БД

3.1.2. Реалізація ланки розширення роботи з базою даних

Ланка розширення складається з сукупності файлів. Кожен з яких відповідає роботі з певною структурою даних, наприклад кінострічка, кінотеатр, білети і т.д. Кожен із файлів імпортує бібліотеку “SwiftyJSON”, для більш зручної роботи з даними.

Функції, які представлені у цьому класі виконують роль фільтрування та сортування даних у разі потреби. Кожна з яких викликає відповідний метод менеджера модуля роботи з БД. Та ініціалізує два блоки – успішного і помилкового виконання. У першому випадку – відбувається другорядна робота з даними, тоді як у другому помилка передається на ланку вище.

Рис. 3.4. Структура роботи функції розширення

3.1.3. Реалізація ланки сервісу роботи з базою даних

Ланка сервісу включає в себе окремі файли для роботи з наявними типами даних. Таке розділення дає змогу, один незалежно від одного, змінювати та розширювати роботу класу. В загальному декомпозиція включає 4-ри сервіси:

- CinemaCervice – сервіс для роботи з даними кінотеатрів;
- MovieService – сервіс для роботи з даними кінострічок;
- GenreService – сервіс для роботи з даними жанрів;
- TicketService – сервіс для роботи з білетами користувача;

До кожного сервісу створюється та реалізується протокол. Він визначає які змінні та функції може виконувати той чи інший клас. Це дає змогу відповідати одній із парадигм програмування – інкапсуляції.

CinemaService використовується для взаємодії та отримання інформації по кінотеатрах. Сюди входить – завантаження усіх моделей, завантаження моделі з

відповідним ідентифікатором, завантаження кінотеатрів, що показують цю кінострічку, та кінотеатрів, що мають відповідний ідентифікатор та фільм.

Розглянемо кожен окремо. Для отримання даних усіх кінотеатрів, викликається метод нижчої ланки розширення “getCinemas”. Після чого, у випадку отримання позитивної відповіді вигляду масиву JSON об’єктів, кожний елемент декодуємо у відповідну модель даних та повертаємо до екрану користувача.

Процес декодування складається з трьох аспектів. Першим – це використання блоку “do-catch”, який запобігає падінню програми у разі неправильної трансформації блоку “do”. Наступним етапом є перетворення JSON формату у тип Data. Де Data – це тип значення даних, що дозволяє простим буферам байтів приймати поведінку об'єктів Foundation. І послідним етапом є декодування типу Date в модель “CinemaModel”.

Отримання кінотеатру по ідентифікатору фільму та подібне до отримання списку усіх елементів. Відмінність проявляється у додатковому фільтруванні елементів по полю “id”. Воно передбачає пошук елемента структури “MovieItemModel”, ідентифікатор фільму якого, збігається з переданим значенням функції. Після чого, якщо елементи не були знайдені – повертається пустий масив, а при наявності – повертається список “CinemaModel”.

Отримання кінотеатру по полях “cinemaId” та “movieId” відбувається шляхом отримання списку об’єктів “CinemaModel” та додатковому фільтруванню по цих параметрах. Якщо в параметр “movieId” не передали значення, то пошук відбувається тільки по “cinemaId”. І після успішної відповіді повертається пустий, або наповнений масив моделей кінотеатру.

Для спрощення був доданий метод пошуку, який включає в себе тільки параметр “cinemaId”. В цьому випадку викликається розглянутий метод з передачею параметру “movieId” у вигляді “nil” значення.

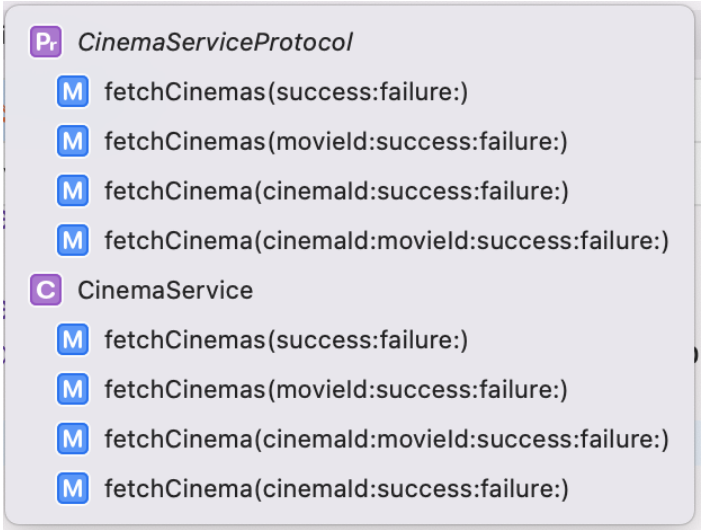


Рис. 3.5. Структура сервісу “CinemaService”

MovieService використовується для отримання списку кінострічок. Він включає можливість завантаження усіх, поточних, майбутніх фільмів, та визначених моделей по ідентифікатору, а також дає змогу проводити пошук по елементам отриманих структур.

Розглянемо кожен окремо. Для отримання даних усіх стрічок, викликається метод нижчої ланки розширення “fetchAllMovies”. Відбувається отримання списку відповіді вигляду масиву JSON об’єктів. Далі елементи масиву декодується у відповідну модель даних, як це було продемонстровано у методі “getCinemas”, з використанням функціоналу “do-catch”. І в разі безпомилкового перетворення, список структур “MovieModel” повертається до екрану користувача.

Для отримання елементів стрічок, які уже вийшли в прокат, використовуємо метод “fetchCurrentMovies”. Та передаємо рядок пошуку, якщо він необхідний. Після декодування отримані моделі сортуємо по полю “startDate”, від новіших до старіших та перевіряємо, щоб дата була не більшою за теперішню. Якщо поле пошуку пусте – показуємо усі елементи списку, а в іншому випадку – фільтруємо по ньому. Далі повертаємо список до користувача.

Для отримання елементів стрічок, які ще не вийшли в прокат, використовуємо метод “fetchFutureMovies”. Який подібний до попередньо

описаної функції. Відмінність тільки у фільтруванні “startDate” так, щоб дата була більшою за теперішню. Після чого повертаємо елементи на екран.

Щоб отримати список кінострічок по жанру, потрібно використати метод “fetchGenreMovies”. Йому необхідно передати ідентифікатор жанру. Який, після завантаження, фільтрує наявність у масиві поля “genres”, елементу отриманого значення “genreId” та повертає результат користувачу.

Для формування списку збережених кінострічок, відбувається отримання списку усіх елементів “MovieModel”, та фільтрації їх по наявності в масиві “likedMoviesIds” сервісу “UserSettingsService”, що буде розглянутий далі.

Для завантаження фільму по ідентифікатору, метод отримує список усіх фільмів та фільтрує їх по наявності “movieId”. Далі, з отриманого набору моделей вибирається перший і повертається на екран користувача.

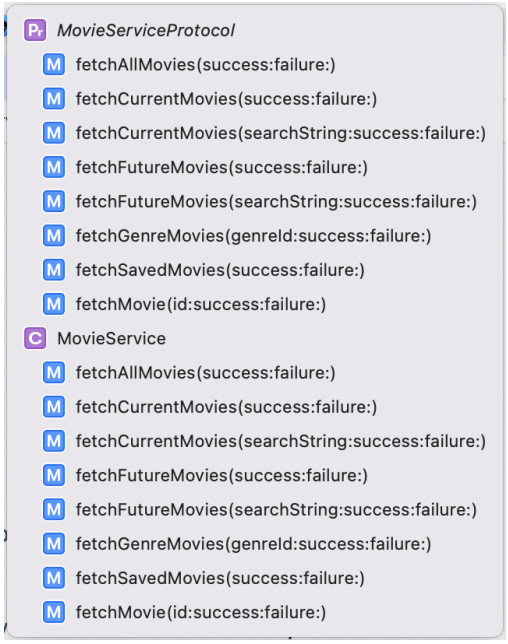


Рис. 3.6. Структура сервісу “MovieService”

GenreService використовується для отримання списку усіх наявних жанрів. Для цього використовується метод “fetchGenres”, який відправляє запит на минулу ланку для відповідного типу моделі. Після отримання списку відповіді вигляду масиву JSON об’єктів, елементи масиву декодується, як це було продемонстровано у минулих методах, з використанням функціоналу “do-catch”. І

в разі успішного перетворення, список структур “GenreModel” повертається до користувача.

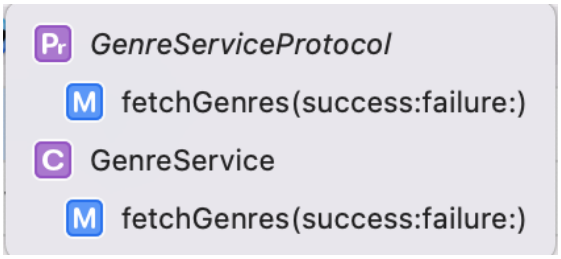


Рис. 3.7. Структура сервісу “GenreService”

TicketService виконує роль завантаження списку білетів що купив користувач, отримання відповідної моделі “TicketModel” по ідентифікатору, а також відправки результату купівлі білету в Firestore.

Для збереження елементу, після покупки, використовується функція “addTicket”, яка приймає модель “TicketModel”. Структура коду відправки поділяється на три етапи. Перший є використання блоку “do-catch”, що запобігає раптовому падінню програми у разі неправильної трансформації блоку. Другим етапом можна вважати кодування “TicketModel” у тип Data та перетворення його у формат JSON. Третім етапом є відправка JSON файлу в базу даних, вказавши необхідний метод нижчої ланки. У разі помилки – повертаємо її на екран користувача.

Завантаження списку усіх білетів, а також визначеного елементу по ідентифікатору відбувається подібно до минуло описаних функцій. Але відмінністю є, в першому випадку, вказання типу “TicketModel”, а іншому – фільтруванню по відповідному ідентифікатору.

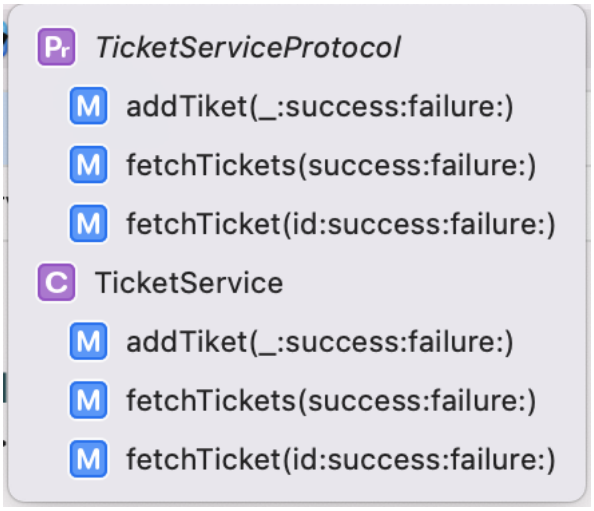


Рис. 3.8. Структура сервісу “TicketService”

3.2. Реалізація модуля автентифікації користувача

Кожна програма, яка прив’язує дії користувача до свого функціоналу неодмінно повинна присвоїти йому певний ідентифікатор. Така поведінка дає змогу відслідковувати взаємодію людини з системою та зберігати дані, які пов’язані з ним.

Одним із найкращих варіантів збереження ідентифікатора за користувачем є його авторизація у системі. На сьогоднішній день є декілька видів цього процесу:

- типова авторизація;
- авторизація за стороннім сервісом;
- системна авторизація;
- прихована авторизація.

До типової авторизації можна віднести використання почтової скриньки та паролю до системи. Користувач використовує сторонній менеджер роботи з повідомленнями, отримує листа та підтверджує свою авторизацію або реєстрацію. До переваг цього методу можна віднести простоту використання, зрозумілість та традиційність. В недоліки входить необхідність реєстрації в почтовому менеджері.

Авторизація за сторонніми сервісами – це більш сучасний підхід, який передбачає використання соціальних мереж для отримання інформації про

користувача. Переваги включають простоту використання та новизну. В недоліки можна віднести необхідність мати сторінку в визначених соціальних мереж.

Системна авторизація представляє з себе використання уже реалізованих функцій системи. Для прикладу можна привести Apple Authorization [13]. Вона полегшує користувачам вхід у програми та веб-сайти за допомогою використання наявного Apple ID. Перевагами є надійність, безпечність та простота. До недоліків входить прив’язаність до операційної системи.

Прихована авторизація не вимагає від користування виконання будь-яких додаткових дій. Ідентифікатор отримується за рахунок використання статичних системних змінних, як от серійний номер пристрою. Переваги цього способу, включають простоту та надійність. А недоліки – неможливість використання декількох профілів на одному пристрої.

Під час розробки додатку був використаний традиційний спосіб реєстрації та авторизації з використанням почтової скриньки та пароля. Такий процес був реалізований разом з бібліотекою Firebase Authorization. Яка контролює потоки відправки даних на сервер, записи профілів а також сесію користувача.

Для підключення бібліотеки необхідно виконати команду “import FirebaseAuth”. Робота обліку сесії користувача виконується у сервісі “UserSessionService”.

Використовуючи шаблон Singletone, створюємо змінну для доступу з будь-якої точки програми. Ініціалізуємо об’єкт “Auth”, який є ключовим елементом роботи з базою даних користувачів. Формуємо змінну “listener”, що реагує на стан авторизації, наприклад реєстрацію, або вихід із профіля. У разі деініціалізації сервісу – вимикаємо прослуховування статусу авторизації. Це зроблено для того, щоб не виникали, так звані, “витіки пам’яті”. Додатково, для зручності зробимо змінну “isAuthorized”, яка перевіряє стан профілю.

Реалізація сервісу “UserSessionService” складається з чотирьох основних методів:

- перевірка поля “email” та “password”;
- реєстрація;
- авторизація;

- вихід із профіля.

Функція перевірки поля “email” та “password” отримує відповідно ці два значення. На першому етапі перевіряється наявність цих полів, а також кількість їх символів. Якщо значення рівне нулю – повертається негативна відповідь та помилка. У іншому випадку починається наступний етап, де перевіряється правильність введення поля “email”. Для цього використовується предикат та регулярний вираз - "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,64}". Якщо поле проходить перевірку – повертається позитивна відповідь.

Реєстрація користувача проходить через функцію “signUp”. На початку виконання відбувається перевірка полів “email” та “password” описаним вище методом. Після чого, за використанням об’єкту “Auth”, викликаємо функцію бібліотеки “createUser”, куди передаємо відповідні поля. Якщо у блоці виконання помилок не з’явилось – повертаємо позитивну відповідь. А у разі помилки – негативну.

Процес авторизації починається з перевірки полів “email” та “password” описаним вище методом. Та, за використанням методу “signIn” об’єкту “Auth”, проводимо авторизацію користувача. Якщо у блоці виконання виникла помилка – повертаємо її на екран, а у іншому випадку – продовжуємо роботу.

Метод виходу з профіля користувача виконується, відповідно, за використанням методу “signOut” об’єкту “Auth”.

Після авторизації, реєстрації, або виходу профілю, раніше ініціалізована змінна, “listener” викликає блок виконання, що містить збереження отриманого користувача системи, позначення статусу “авторизовано/неавторизовано”, та відправку повідомлення про дію зі сторони “UserSessionService”.

Додатково файл сервісу включає структуру власних помилок, які можуть бути викликані у розглянутих раніше методах. Це дає користувачу змогу більш детально зрозуміти походження помилки, та виправити її.

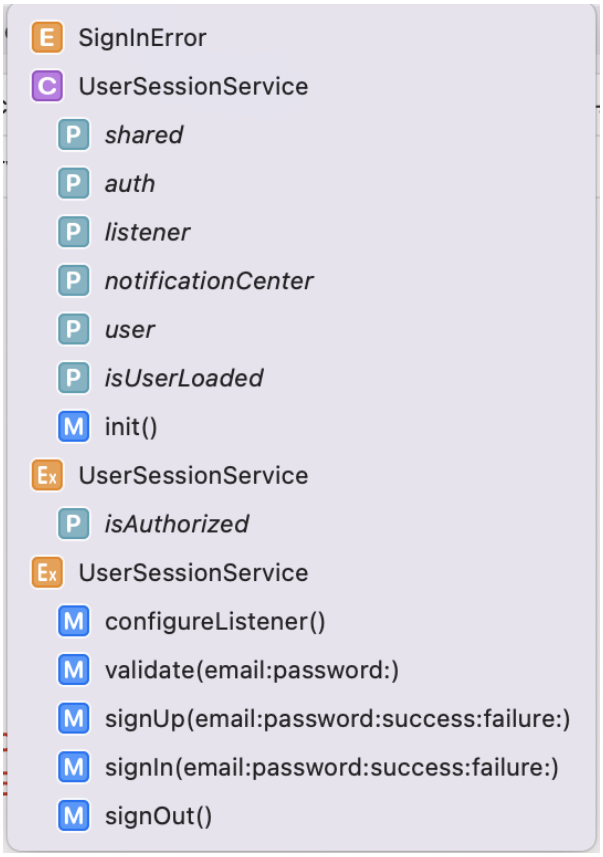


Рис. 3.9. Структура сервісу “UserSessionService”

3.3. Реалізація модуля налаштувань користувача

Часто, з’являється необхідність зберегти вибір користувача в тих чи інших ситуаціях. Наприклад увімкнення, або вимкнення вібро відгуку телефону, вибір теми інтерфейсу, тощо. Тому для групування усіх можливих варіантів, використовують окремий клас, доступ до якого можуть звертатися всі інші елементи програми.

У розробленому додатку, робота налаштувань користувача базується на системному сервісі “UserDefaults” [14]. Він призначений для збереження невеликих фрагментів даних, які використовуються під час запуску програми. “UserDefaults” дозволяє зберігати пари ключ-значення, де ключ завжди є рядком, а значенням може бути один із таких типів даних: дані, рядок, число, дата, масив або словник.

Робота сервісу налаштувань користувача створена на основі класу “UserSettingsService”. На початку роботи він ініціалізує константу “shared”, яка виступає в ролі шаблону проектування “Singletone”, що був розглянутий раніше.

Для спрощення роботи сервісу “UserSettingsService”, було розроблено приватний перелічувальний тип даних. Перерахування — це тип, який містить визначену кількість значень, наприклад рядки, або цілі числа. Ця змінна містить ключі доступу до елементів, що зберігатимуться в “UserDefaults”.

Кожна змінна сервісу “UserSettingsService” представляє з себе блок виконання, який базується на формуванні “get-set”. У частині коду “get” відбувається отримання даних, та їх перетворення. Після чого елемент “get” зобов’язується передати тип, що визначений в основі змінної. З іншої сторони – структура “set”, отримує значення, які були задані відповідному елементу, та обробляє їх, зберігаючи їх в іншому вигляді, або іншій частині коду.

Наразі, у програмі знаходиться два можливих варіанта використання налаштувань додатку:

- likedMoviesIds;
- isNotificationEnabled.

Перший представляє з себе масив текстових значень. Кожен елемент, цього списку, є ідентифікатором моделі “MovieModel”. Таке формування дозволяє зберігати кінострічки, які найбільше сподобались користувачу.

У блоці “get”, змінної “likedMoviesIds”, програма викликає змінну “standart” класу “UserDefaults”, яка являє з себе шаблон “Singletone”. Після отримання змінної відбувається виклик методу “string(forKey:)”, який, по переданому ключу, повертає відповідний елемент типу “string”. Якщо ж такої змінної не існує отримуємо значення “nil”. Так як ми отримали текстовий рядок, його необхідно перетворити в масив. Для цього, використовується системна команда “components(separatedBy:)”, яка створює масив із рядка, що розділяється певним символом. Після чого, отриманий результат повертається до користувача.

У блоці “set” змінної “likedMoviesIds”, за використанням функції “joined(separator:)”, із масиву формується текстове значення та зберігається у відповідне поле, за допомогою класу “UserDefaults”.

Параметр “isNotificationEnabled” – це змінна типу “Bool”, яка використовується для ідентифікування вибору користувача увімкнення, або вимкнення повідомлень у програмі.

У блоці “get”, змінної “isNotificationEnabled”, за використанням класу “UserDefaults”, витягується з комірки пам’яті значення типу “Bool”, та повертається на екран. А у блоці “set”, отриманий елемент зберігається за відповідним ключем, без жодних перетворень.

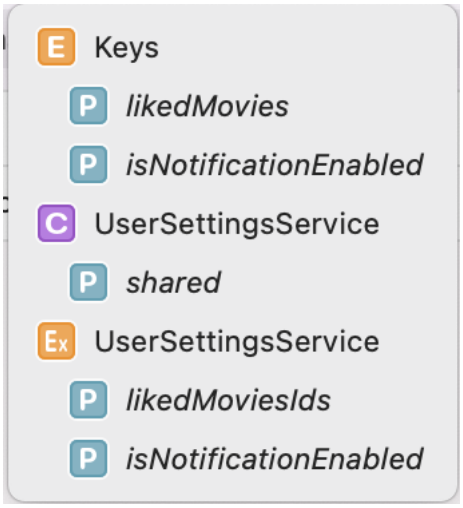


Рис. 3.10. Структура сервісу “UserSettingsService”

3.4. Реалізація модуля оплати

Покупка є однією з основних функцій додатку. Адже користувач, знайшовши цікаву кінострічку, та, вибравши найзручніший кінотеатр, хотів би купити білет прямо зі свого телефону. Тому цілком логічно розробити функціонал, який давав би змогу зробити це.

У системі iOS покупку можна зробити трьома шляхами. Кожен з яких має свої особливості та недоліки. До них можна віднести:

- покупка через AppStore;
- покупка через сторонній сервіс;
- покупка через Apple Pay.

Перший варіант, ще називають “in app purchase” [15]. Він дозволяє розробникам стягувати з користувачів плату за певну функціональність або вміст під час використання програми. На даний час існують чотири види покупки:

- consumable - це елементи, які можна купувати знову і знову після того, як користувачі використав їх;
- non-consumable - це товари, які купуються лише один раз;
- auto-renewable subscription – це підписки, які автоматично поновлюються, стягаючи повторну оплату кожен період;
- non-renewable subscription – це підписки, які не можуть автоматично поновлюватись.

Розглядаючи кожен із варіантів, можна зрозуміти, що покупка білету таким чином не зручна. Адже розробникам необхідно створювати для кожної кінострічки свою покупку у відповідних кінотеатрах. Тому така реалізація не доцільна з точки зору часу.

Покупка через сторонній сервіс являє собою використання зовнішнього інструменту, для списання коштів. Проте додаток з таким варіантом не завжди може задовольнити перевірку Apple. Це все може посприяти недопущенню розміщення додатку у AppStore.

Покупка за використанням інструменту Apple Pay – це системний спосіб перевірки та отримання реквізитів користувача з подальшим списанням коштів на сторонньому сервісі [16]. Такий спосіб реалізації, дозволяє користувачам безпечно передавати дані для оплати, адже їх отримання відбувається у зашифрованому вигляді.

З огляду усіх варіантів оплати у додатках IOS, найбільш доцільним і зручним способом списання коштів за білет, можна визначити функціонал Apple Pay. Він дозволить спростити використання оплати у програмі, а також збереже усю інформацію про користувача зашифрованою.

Для роботи з Apple Pay, нам необхідно підключити відповідну системну бібліотеку, використовуючи команду “import Passkit”. Також для зручності визначимо типи даних за допомогою ключового слова “typealias”. До них входять:

- PaymentSuccess – замість блоку “() -> Void”, успішне отримання результату;
- PaymentFailure – замість блоку “(String) -> Void”, помилкове отримання результату;
- PaymentCancelled – замість блоку “() -> Void”, отримання результату, що був відмінений.

Розглянемо змінні та константи, які були визначені та використанні у роботі сервісу “ApplePayService”. До них входить:

- merchantIdentifier – ідентифікатор мерчанту, який використовується для визначення додатку на серверах Apple;
- supportedNetworks – список доступних мереж оплати, таких як “masterCard”, або “visa”;
- merchantCapabilities – тип шифрування даних користувача, який підтримує Apple, визначений як “3ds”;
- paymentSuccess – блок успішного виконання;
- paymentFailure – блок помилки.

Розглянемо роботу оплати, сервісу “ApplePayService”. Для виклику функції “paymentRequest” необхідно передати параметри назви товару, ціни, ідентифікатор валюти, та коду. Результатом виконання запиту є об’єкт “PKPaymentRequest” бібліотеки “Passkit”.

У тілі методу, виконується створення запиту “PKPaymentRequest”, під час якого записуються відповідні значення полів мерчанту, доступних мереж, типу шифрування, ідентифікатору валюти та країни, а також заголовок та ціну товару.

Для виклику функції “makePayment” необхідно передати параметри назви товару, ціни, блоків успішного, помилкового та відміненого виконання. У тілі методу відбувається перевірка можливості зробити оплату в такий спосіб. При помилці – повертаємо її користувачу.

Наступним кроком функції “makePayment” є ініціалізація запиту оплати, уже розглянутим способом “paymentRequest”. Після чого відбувається створення контролера показу екрану “PKPaymentAuthorizationViewController”, який приймає змінну “PKPaymentRequest”. Отримавши цей об’єкт, програма зберігає класові

значення “paymentSuccess” та “paymentCancelled”, а також задає значення делегату та виводить екран на пристрій.

Коли користувач авторизується за допомогою біометричних методів системи, буде викликаний відповідний метод, делегату “PKPaymentAuthorizationViewControllerDelegate”. У разі успішного виконання – реквізити надходять до зовнішнього сервісу оплати. А при помилці – результат повертається на екран користувача.

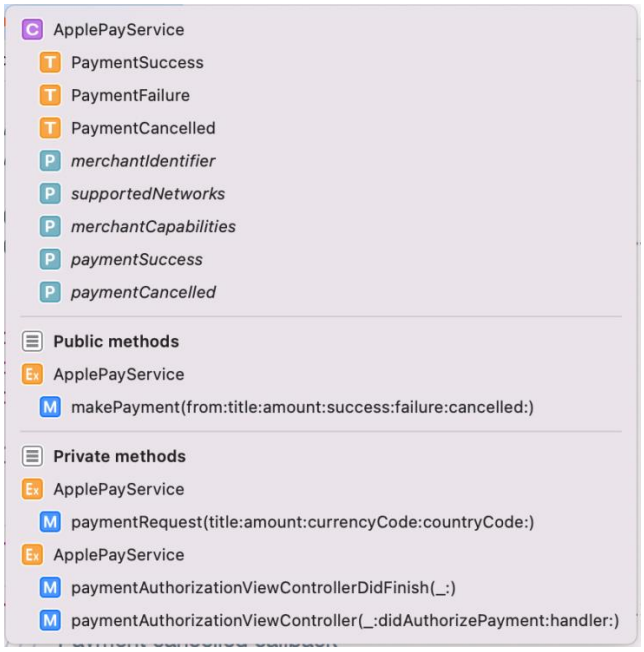


Рис. 3.11. Структура сервісу “ApplePayService”

3.5. Реалізація модуля формування QR-коду

Коли користувач купує білет, дані про це відправляються у Firebase. Після цього, кінотеатри-партнери можуть перевіряти данні, покупок їхніх кінострічок. Проте, для зручності та оптимізації процесу, доцільно пришвидшити перевірку наявності та актуальності білету. Одним із варіантів, для покращення процесу перегляду квитка є формування QR-коду [17].

Він, зазвичай складається з чорних елементів на білому фоні. Декодуючи записане зображення, вміст штрих-коду може направити користувача на адресу в Інтернеті, електронної пошти, номер телефону, контактну інформацію, SMS,

MMS, або інформацію про геолокацію. Наразі, на ринку є багато генераторів і зчитувачів QR-кодів, тому використання такого пристрою у кіноіндустрії не є проблемою.

Рис. 3.12. Приклад QR-коду білету

Купивши білет, користувач отримує власний QR-код, який містить дані про цю покупку. А саме інформацію знаходиться у вигляді JSON об’єкту “TicketModel”, яка перетворена у QR значення. Далі, підходячи на касу, людина показує це зображення. Відповідальна особа сканує його, після чого, програмне забезпечення кінотеатру перевіряє наявність інформації про покупку.

Проблема фальсифікації та дублювання кодів зникає, адже кожен білет формується з унікальним ідентифікатором. І після зчитування, сам квиток, стає дезактивованим. Крім цього білет перевіряється на актуальність дати показу. Саме така реалізація блокує можливість повторного використання коду.

Модуль формування QR-коду розроблений за використанням класу “QRCodeService”. Для його роботи використовуються наступні бібліотеки:

- Foundation – робота з базовими типами даних;
- CoreImage – бібліотека формування зображень;
- SwiftUI – робота з інтерфейсом користувача;
- UIKit – робота з старим типом інтерфейсів користувача;

Для створення QR-коду використовується метод “generateQRCode”. Його вхідними параметрами є текстовий рядок, який буде кодуватися. Для

перетворення моделі в символи використовується клас “JSONEncoder”, що був розглянутий раніше.

У тілі методу ініціалізується об’єкт типу “CIContext”, який генерує зображення. Крім того, створюється константа фільтру генерації QR-коду, який реалізує перетворення тексту у відповідну послідовність байтів. Далі, перетворюючи отриманий текстовий рядок у тип “Data”, передаємо його на вхід до фільтру під ключем “inputMessage”.

Якщо вхідні дані відповідають заданим вимогам, то в результаті отримуємо об’єкт “CIImage”. Це представлення зображення, яке буде обробляється або створюється фільтрами Core Image.

Далі отримані дані, за допомогою контексту, генеруємо у тип “CGImage”, який являє з себе растрове зображення або його маску.

У процесі виконання функції, це значення перетворюється на “UIImage” та “Image”. Тобто об’єкт, який керує даними зображення у програмі. Результат виконання передаємо користувачеві на екран.

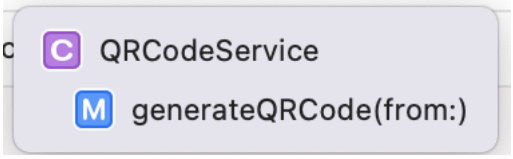


Рис. 3.13. Структура сервісу “QRCodeService”

ВИСНОВОК ДО РОЗДІЛУ 3

В розділі було оглянуто модуль роботи з базою даних, використовуючи розділення на три основні ланки – менеджера, розширення та сервісів роботи з інформацією. Розглянуто основні варіанти авторизації та реєстрації в додатку. Проаналізовано реалізацію системи автентифікації.

Досліджено роботу модуля налаштувань програми. Оцінено та визначено найзручніший спосіб покупок у додатках. Реалізовано системи оплати в програмі. Оглянуто клас для формування QR-коду по визначеному текстовому рядку.

РОЗДІЛ 4

РОЗРОБКА ІНТЕРФЕЙСУ ТА ПІДТРИМКА ДОДАТКУ

Сучасний мобільний додаток - це сукупності екранів, які взаємодіють між собою. Кожна сторінка програми виконує завчано завдану функцію. Для цього, на початку проектування, визначають ціль програмного забезпечення, його функціонал, мету роботи, основну логіку та формують зовнішній вигляд.

Кожен екран – це компонування інтерфейсу та програмного коду, який визначає його роботу та поведінку. На сьогоднішній день набули поширеності середовища розробки, які дозволяють за допомогою інструментів створити сторінку додатку, без написання коду. Для прикладу, можна навести програмне забезпечення XCode, та його функціонал – Interface Builder.

Interface Builder [18] робить палітри або колекції Objective-C доступними для розробника. Ці елементи графічного інтерфейсу включають текстові поля, таблиці або спливаючі меню. Палітри Interface Builder мають можливість розширюватись.

Interface Builder зберігає інтерфейс програми як пакет, що містить об’єкти інтерфейсу та зв’язки, які використовуються в програмі. Ці об’єкти архівуються у файл XML або у файл списку властивостей стилю NeXT з розширенням .nib. Після запуску програми відповідні об’єкти NIB розархівовуються та пов’язуються з бінарним файлом програми.

Проте, все біль поширенішим стає розробка інтерфейсу, напряму, через написання коду програми. Це значно пришвидшує компілювання та роботу додатку. Гарним прикладом цієї реалізації можна визначити бібліотеку SwiftUI. Це набір інструментів користувальницького інтерфейсу, який дозволяє декларативно проектувати програми. SwiftUI [19] має абсолютно нову систему компонування.

Кожен інтерфейс може масштабуватися та коригуватися відповідно до таких факторів, як пристрій, на якому виконується код, змінювати параметри доступності, які ввімкнув користувач, і використовувати локаль, що вибрана в даний час.

Виходячи з усіх можливих варіантів розробки екранів додатку, доцільно зупинись на використанні декларативної мови написання інтерфейсу SwiftUI у поєднанні з додатковими файлами, якій буть супроводжувати та підтримувати роботу сторінки програми.

4.1. Аналіз та розробка екранів додатку

Структура екранів мобільного додатку для організації роботи з мережами кінотеатрів, складається з чотирьох основних модулів. Кожен ж яких, виконує відведену йому функцію, та працює незалежно від інших. До цього списку входять:

- модуль автентифікації користувача;
- модуль списку кінострічок;
- модуль покупки білету;
- модуль профіля користувача.

Кожна, окрема сукупність екранів, була розроблена за використанням декларативного інтерфейсу SwiftUI. Переходи між екранами відбуваються, або за використанням відкриття повного представлення вище по ієрархії за поточний, або шляхом реалізації функціоналу системної навігації.

Кожен екран інтерфейсу складається з сукупності, щонайменше, чотирьох основних класів:

- Configurator;
- Router;
- ViewModel;
- View.

Configurator відповідає за формування і налагодження зв’язків на поточному блоці. Для його ініціалізації можуть бути передані додаткові конфігуратори, що

створюють нові сукупності інтерфейсу. Кожен такий елемент повинен задовольняти протокол, який задає усі можливі варіанти формування екранів, на які можна перейти з поточного.

Router відповідає за перехід, або відкриття нового блоку інтерфейсу. Отримавши завчано визначений шлях від моделі представлення, router вирішує який та яким способом відкрити той чи інший екран. Сюди можуть також можуть входити сповіщення, спливаючі вікна, або прості елементи інтерфейсу.

ViewModel відіграє роль центру керування екраном, яка відповідає за інформацію, що представлена на дисплеї, її завантаження та обробку, реакцію на дії користувача, поведінку додатку та надсилання шляху до Router.

View являється представлення екрану, яка містить структуру розташування елементів, їх реакцію на зміну інформації у ViewModel. Крім того, View додає компоненти переходів, до яких можна віднести “NavigationLink”.

Для комунікації екрану з моделлю представлення та об’єктом навігації, використовується реактивне програмування. При його використанні, інформаційні потоки складають основу програми. Події, повідомлення та помилки передаються через потік. За допомогою реактивного програмування, додаток спостерігаєте за цими ними та реагуєте, коли відбувається завчасно визначена дія.

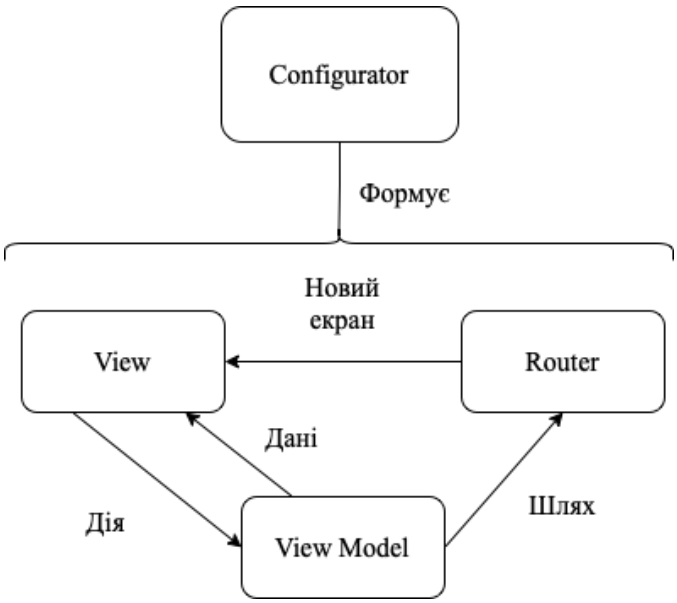


Рис. 4.1. Схема роботи модуля екрану

4.1.1. Реалізація головного блоку відкриття екранів.

Під час запуску, будь який додаток відкриває початково заданий екран. Найчастіше, таким елементом є модуль завантаження. Він інформує користувача про те, що програма отримує і обробляє усю необхідну інформацію для її запуску.

Розроблене програмне забезпечення не є виключенням. Проте, для такої реалізації, було використано рішення додати новий модуль, який буде маніпулювати представленнями в залежності від того, який стан має програма.

Наприклад, під час запуску додатку, можна побачити екран завантаження, що відкривається або на початку роботи, або під час синхронізації даних. Після процесу отримання і обробки інформації, якщо користувач не авторизований, буде показаний модуль авторизації. В іншому випадку – домашня сторінка додатку. Тому можна виділити три стани:

- launch – завантаження або синхронізація даних;
- login – авторизація користувача;
- home – домашній екран.

Для відслідковування процесів, які виникають в додатку, використовується сервіс “Notification Manager”. За допомогою нього, передається інформація, про статус виконаної синхронізації, авторизації або виходу з аккаунту користувача.

Така реалізація виконана шляхом використання бібліотеки “Combine”, яка була розглянута раніше. Елементи “Publishers” відслідковують зміну або появу нової події та реагують способом, який був завчасно заданий.

Після запуску, отримавши сповіщення про успішну синхронізацію, “NotificationManager”, відправляє повідомлення про це. Далі, модель представлення, перевіряє статус автентифікації користувача. У авторизованому випадку на об’єкт, типу Router, передається відповідний шлях “home”. Якщо ж, користувач ще не увійшов у систему – відправляється “login”.

Таким чином використання основного блоку “AppFlow”, дає можливість на початковому рівні маніпулювати екранами, відповідно до вхідних статусів програми.

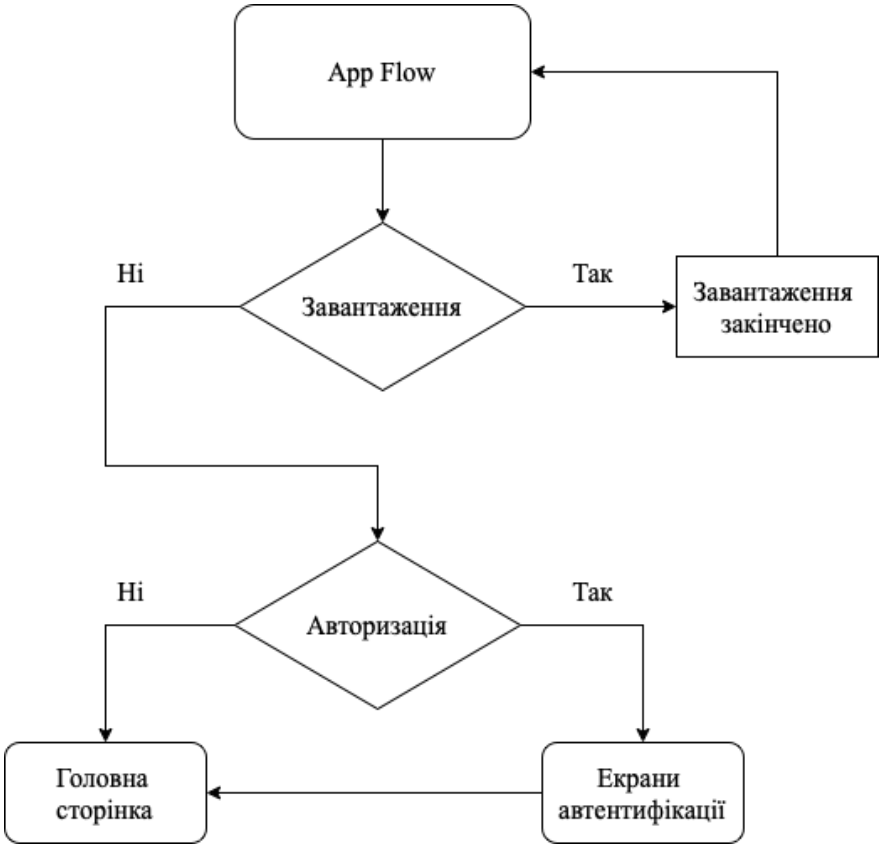


Рис. 4.2. Реалізація роботи модулю AppFlow

4.1.2. Розробка екранів автентифікації

Пройшовши етап завантаження та синхронізації даних, користувач попадає на екран авторизації, якщо раніше він не був автентифікований. Цей етап є необхідною стадією для ідентифікації людини у системі та визначені даних, які їй належать. Така реалізація забезпечує правильність отримання інформації з бази даних та фактор її безпеки.

Модуль автентифікації користувача складається з двох основних екранів – авторизації та реєстрації. Відповідно, користувач має змогу вибрати той, який йому більш необхідний. Кожен із них використовує менеджер для роботи з сесією користувача “UserSessionService”, що був розглянутий раніше.

Екран авторизації складається з двох, основних, полів для вводу інформації, які були використані під час авторизації користувача, кнопки авторизації, текстового заголовку та зображення сторінки.

Перше місце для введення інформації – це текстові дані електронної пошти. Перевірка вхідних даних відбувається за використанням предикату та регулярного виразу “[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,64}”, який передається йому. Таким чином користувач може використати символи, які відповідають наступним вимогам:

- блок символів до “@” може містити великі, або малі літери;
- блок символів до “@” може містити цифри від 0 до 9;
- блок символів до “@” може містити символи “.”, “_”, “%”, “+”, “-”;
- блок символів до “@” повинен містити хоча б одну літеру, або цифру;
- блок символів після “@” може містити великі, або малі літери;
- блок символів між “@” та “.” може містити цифри від 0 до 9;
- блок символів між “@” та “.” може містити символи “.”, “-”;
- блок символів між “@” та “.” може містити символи “.”, “-”;
- блок символів між “@” та “.” повинен містити хоча б одну літеру, або цифру;
- блок символів після “.” повинен містити від 2-х до 64-х великих або малих літер.

Друге поле відповідає за інформацію про пароль користувача, та перевіряється, щоб кількість символів була більшою за 8. Крім того, для забезпечення приватності, місце введення маску отриманий текст у вигляд крапок.

Після внесення інформації про почтову скриньку та пароль, користувач може натиснути на кнопку авторизації. Вона відправляє відповідний потік даних у модель представлення, де він відслідковується.

У випадку отримання відповідної події, ViewModel перевіряє вхідну інформацію та викликає метод “signIn”, сервісу для роботи з сесіями користувачів. Якщо відповідь позитивна, через “NotificationManager” надсилається сповіщення про авторизацію. Далі, модуль “AppFlow” переадресує користувача на головну сторінку.

Якщо відбулася помилка у перевірці даних, або під час автентифікації користувача, ViewModel надішле шлях “error(message)” у Router, де “message” –

це текст повідомлення. Після чого, Router, обробляючи вхідну інформацію, змінює відповідні поля “isAlertShowed” та “AlertView”. У свою чергу, отримавши потік даних про зміну, View презентує сповіщення користувачу. Таким чином відбувається взаємодія на екрані “SignIn”.

Модуль “SignUp” сформований за використання трьох полів – почтової скриньки, паролю та його повторного введення. Перевірка вхідних даних відбувається по тому ж алгоритму, що був описаний на екрані “SignIn”. Проте, додатково, поля паролю та його повторного введення перевіряються на ідентичність. Це зроблено для того, щоб користувач не між помилитись у введені символів, літер або цифр.

Взаємодія користувача з екраном “SignUp” відбувається за використанням кнопки реєстрації. Під час її натиску, у модель представлення, відправляється запит на реєстрацію. Отримавши його, ViewModel викликає функцію “signUp”, сервісі роботи з сесіями користувачів. У разі позитивної відповіді, через “NotificationManager”, надсилається сповіщення про реєстрацію, а користувача переадресовує на домашню сторінку. В іншому випадку ViewModel надсилає помилку на Router, який зміню відповідні поля, а View відкриває інформацію про помилку.

Таким чином відбувається авторизація та реєстрація користувача, що супроводжується використанням сервісу “UserSessionService” та “NotificationManager”. Які надсилають відповідні дані до бібліотеки Firebase Authorization, яка, в свою чергу, обробляє дані користувача та повертає отриману відповідь від серверу.

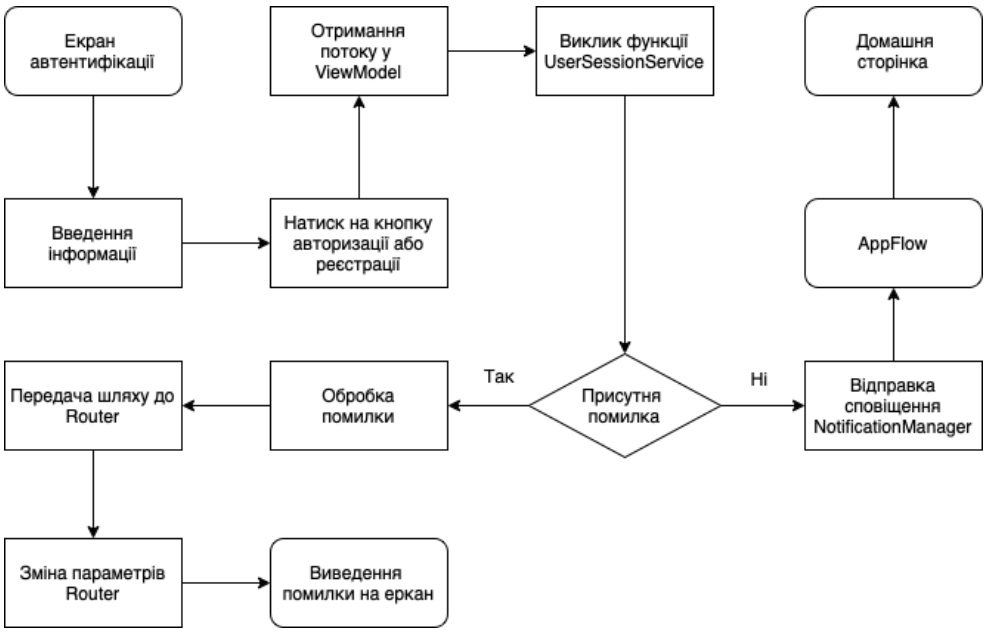


Рис. 4.3. Реалізація роботи екрану автентифікації

4.1.3. Реалізація екранів для перегляду списку кінострічок

Екрани для перегляду списку кінострічок, складаються із чотирьох основних модулів. Кожен із яких, виконує відведену йому функцію. До них входять:

- домашня сторінка;
- список кінострічок;
- деталі фільму;
- пошук кінострічок;

Першим екраном після авторизації, або реєстрації є домашня сторінка. На ній знаходяться секції пошуку, категорій, поточних і майбутніх кінопрем’єр.

Перша розглянута структура містить поле, при натиску на яке, відбувається швидкий перехід на екран пошуку.

Друга секція включає у себе список жанрів, які завантажуються під час відкриття сторінки. Кожен елемент представлений у вигляді блоку з emoji та текстовим надписом. Користувач має змогу натиснути на одну із структур, після чого відкриється список кінострічок по відповідному жанру.

Далі йдуть два блоки – список поточних та майбутніх кінопрем’єр. Дані, для кожної з яких, завантажуються окремо у менеджері представлення, після того, як відкрився поточний екран. Користувач має змогу натиснути на кінострічку, що змусить програму відкрити деталі відповідного фільму. Крім того, кожна така секція містить кнопку “більше”, яка викликає екран зі розширеним списком кінопрем’єр.

Отримання даних, для кожної із секцій відбувається під час запуску екрану, а тому, коли інформація ще не завантажилась, на місцях жанрів і кінострічок – можна побачити індикатори процесу, які сповіщають користувача, що елементи ще не були ініціалізовані.

Для коректної роботи, використовується модель представлення, яка відслідковує натиски кнопок, завантажує відповідні дані, та керує процесом передачі шляху для переходу на новий екран. Крім звичайного відкриття нового модуля, використовується навігаційний елемент “NavigationLink”. Це елемент SwiftUI, який запускає навігаційну презентацію. Він виконується під час відкриття екранів деталей книги, списку кінострічок, пошуку, або профілю користувача.

Сторінка інформації про фільм, містить його обложку, назву, студії розробники, жанри та детальний опис. Крім цього, екран містить рейтинг кінострічки, якщо вона уже вийшла в прокат. Також користувач може натиснути на кнопку покупки білету або додавання фільму у список улюбленого. В першому випадку, відкривається модуль придбання квитка. А в другому – кінопрем’єра, за використанням сервісу “UserSettingsService”, який був розглянутий раніше, оновлюється змінна “likedMoviesIds”, шляхом отримання та видалення або додавання поточного ідентифікатору фільму.

Завантаження даних відбувається за отриманням вхідного параметру ідентифікатору кінострічки, який надсилається функції сервісу “MovieService”. У разі отримання відповіді, результат буде показаний на екрані.

Сторінки списку фільмів розроблені з урахуванням типу запиту. Це означає, що відкриття екрану супроводжується визначенням його вмісту до початку завантаження даних. До основного вибору входять:

- поточні кінострічки;
- майбутні кінострічки;
- збережені кінострічки;
- кінострічки за жанром.

Таким чином спрощується використання та запобігається дублювання коду екрану, які подібні за змістом. Достатньо лише завчасно вказати визначений тип даних.

Робота екрану починається з того, що модель представлення по отриманому значенню, використовуючи сервіс “MovieService”, завантажує відповідний список кінострічок. У разі помилки, шлях “error(message)” відправляється в Router, де визначається що показати користувачу на дисплеї. Якщо результат запиту “MovieService” позитивний, дані зберігаються в масив, а представлення “View”, оновлює вміст екрану. У проміжку, між відправкою і отриманням відповіді, користувач бачить тільки область навігації та індикатор активності.

Коли користувач натискає на одну із вибраних кінострічок, модель представлення відслідковує це сповіщення, отримує відповідний об’єкт та передає шлях “movie(id)” на Router. Після чого відбувається відкриття модулю деталей фільму.

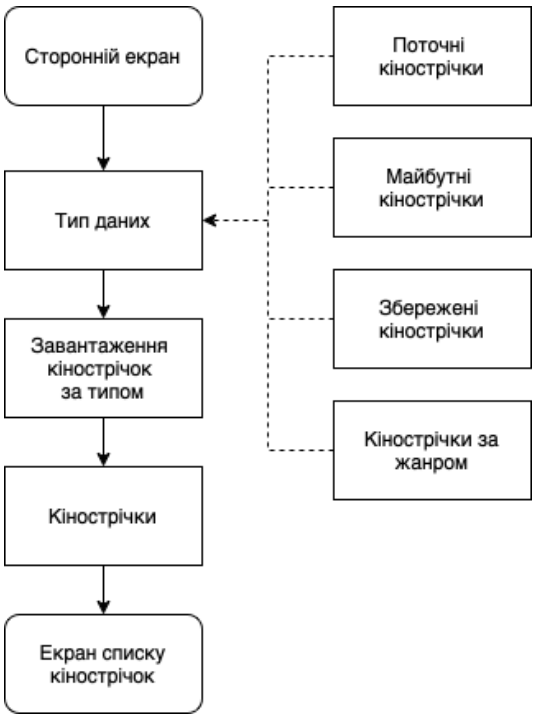


Рис. 4.4. Реалізація роботи екрану списку кінострічок

Сторінка пошуку реалізована шляхом поєднання двох секцій фільмів. До першої відносяться ті, які уже є в прокаті, до другої – тих, що ще не вийшли. Також екран містить поле для введення текстового значення, по якому буде відбуватись пошук кінострічок.

Реалізація функціоналу починається з завантаження списку кінопрем’єр з пустим полем пошуку. Після того, як користувач вводить необхідний запит та натискає на кнопку, у модель представлення приходе потік інформації, що містить текстовий рядок, який був уведений. Далі, використовуючи сервіс “MovieService”, послідовно викликаються дві функції – завантаження наявних і майбутніх кінострічок. Кожній з яких передаються отримані текстові дані.

У разі помилки, шлях передається на Router і відповідно відкривається сповіщення на представлені. Якщо ж список прийшов, він записується у відповідний масив та відображається на екрані. Завантаження супроводжується індикатором активності. Користувач має можливість вибрати потрібну кінострічку. Після чого, відкриється відповідне вікно деталей фільму.

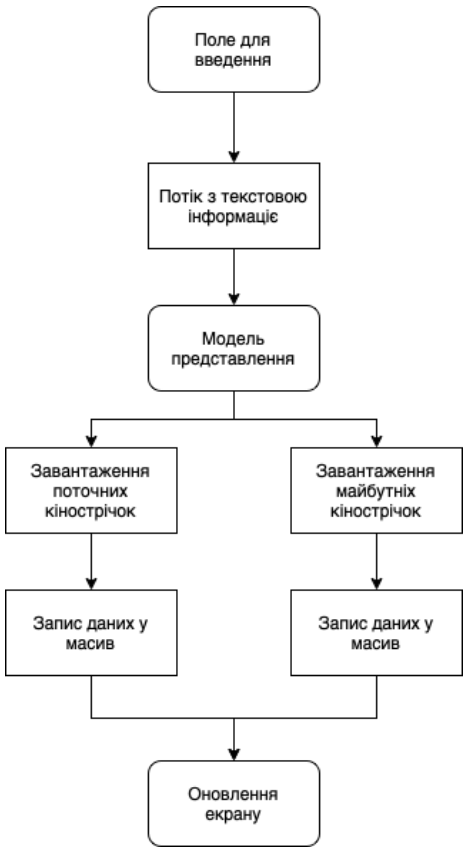


Рис. 4.5. Реалізація роботи екрану пошуку

4.1.4. Аналіз та розробка екранів покупки

Користувач має змогу купити білет на кінострічку, яка йому сподобалася. Для цього достатньо натиснути на відповідну кнопку екрану деталей про фільм. Після чого, панель навігації, відкриє модуль для покупки квитка.

Сукупність чотирьох послідовних екранів формують функціонал, що забезпечує можливість користувачу:

- вибрати кінотеатр;
- вибрати час показу;
- вибрати місце;
- оплатити білет;

Першою сторінкою, модуля покупки білету, є вибір кінотеатру. Він представлений списком усіх можливих місць, де даний фільм буде показаний. Завантаження даних починається з моменту відкриття екрану. За допомогою сервісу “CinemaService”, викликається функція “fetchCinemas”, якій передається ідентифікатор поточної кінострічки. Розглянутий раніше функціонал, фільтруючи по об’єктах “MovieItem”, структури “CinemaModel”, поверне список кінотеатрів, які містять цей фільм. Якщо ж виникне помилка – вона буде показана на екрані.

Отримана інформація записується в масив, який завчасно фільтрується по алфавіту. Після цього, представлення View оновлює екран. У проміжку завантаження даних, користувач буде бачити індикатор активності.

Натискаючи на елемент списку, представлення View сповіщає ViewModel, про те, що була вибрана відповідна модель даних. У цьому випадку формується модель покупки, яка включає в себе вибрані ідентифікатори фільму та кінотеатру. Після чого, отримавши шлях для переходу, Router викликає навігаційний елемент “NavigationItem”, який змінює екран, передаючи поточні дані.

Наступним етапом покупки білету є вибір часу показу кінострічки. Для цього, за допомогою сервісу “MovieService”, із моделей “MovieItem”, отримують сеанси показу та групують по днях. У проміжку завантаження даних, користувач

буде бачити індикатор активності. Таким чином користувач отримує, усі можливі, дати та години для покупки квитка.

Після вибору одного із сеансу, у отриману раніше модель даних покупки додається поле ідентифікатору “MovieItem”, який буде використаний для формування білету. Крім того, натиск супроводжується переходом на наступний етап.

Кран вибору місця складається з блока рядів та секцій сидінь, зображення розміщення дисплея кінотеатру, вибраної позиції та кнопки покупки. Перший елемент дає можливість користувачеві наглядно побачити які місця уже зайняті, та які він може купити. Другий елемент використовується для визначення напрямку сидінь. Третій елемент з’являється та зникає, коли користувач додає, або прибирає вибране місце у залі. Він допомагає зрозуміти користувачеві вартість місця а також його позицію.

Кнопка покупки дає можливість виконати оплату через розглянутий раніше сервіс “ApplePayService”. Після натиску на неї, у потік даних передається відповідно вибране місце у залі. Модель представлення викликає функцію “makePurchase”, яка відкриває системний сервіс оплати. Після успішного списання коштів, ViewModel формує білет з усіма вхідними параметрами моделі покупки. Далі, за допомогою сервісу “TicketService”, він зберігається базі даних Firebase Firestore. Де надалі його можна відкрити та використати.

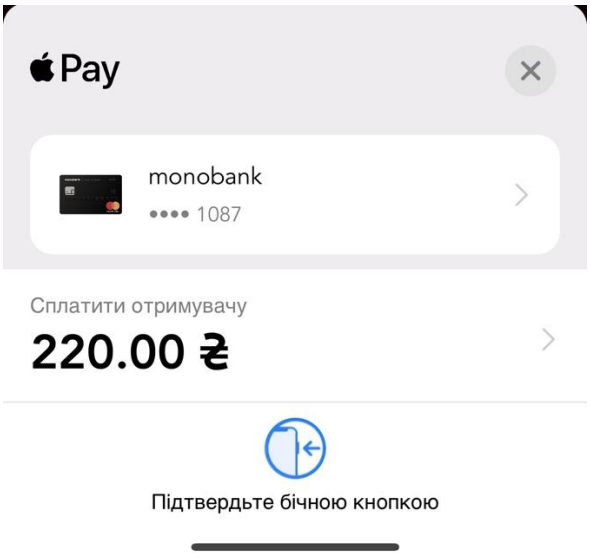


Рис. 4.6. Сторінка списання коштів

Успішне виконання покупки та відправки квитка супроводжується формування сповіщення для “NotificationManager”, та його відправкою. Домашня сторінка, отримавши цей потік даних, та, використовуючи елемент навігації, закриває усі поточні вікна. Після чого показує користувачеві повний екран деталей білету, який був тільки що куплений.

Така реалізація функціоналу дає можливість, без зайвих проблем, вибрати кінотеатр із можливих варіантів. Базуючись на показі сеансу, вибрати необхідну дату та час. Отримавши список вільних місць, дати користувачеві вибір для покупки. І за наявності результату оплатити і зберегти кінцевий білет.

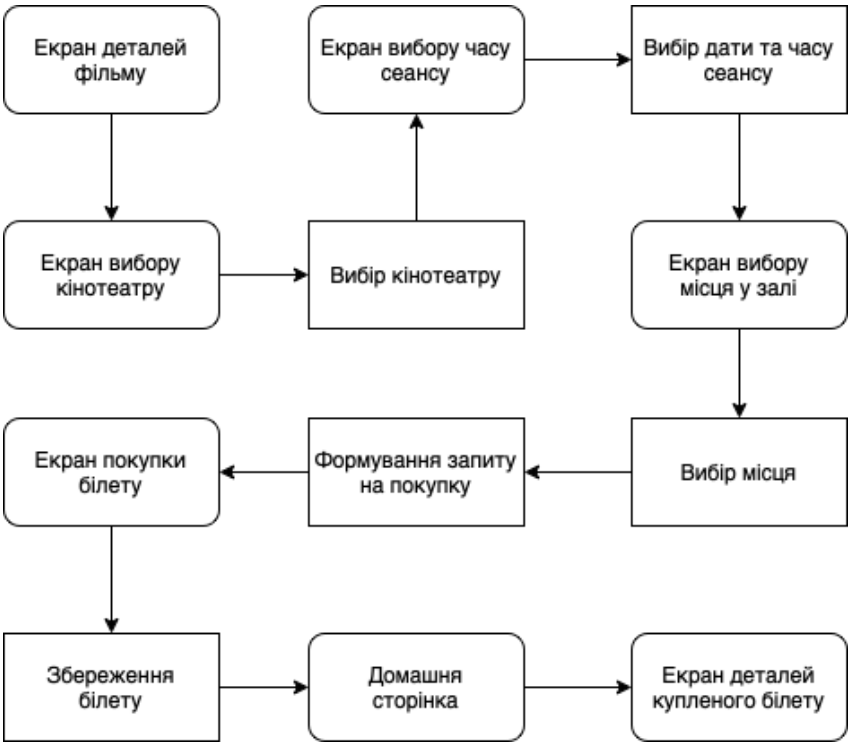


Рис. 4.7. Реалізація роботи модуля покупки

4.1.5. Огляд роботи екрану профіля користувача

Екран профілю використовується для відмежовування дій, які зв’язані напряму з користувачем. Перейти на цей модуль можна натиснувши іконку домашньої сторінки. Його структура складається з зображення користувача та опцій, які можуть бути вибрані ним. До них входять:

- сповіщення – дає можливість користувачеві вимкнути, або увімкнути отримання сповіщень у додатку;
- збережені стрічки – елемент навігації який відправляє до списку фільмів, що сподобалися користувачеві;
- придбані квитки – вибір, натиснувши на який людина попадає до переліку усіх куплених ним білетів;
- вийти з аккаунту – ініціалізує вихід з профілю користувача.

Опція сповіщення використовується для зміни статусу отримання повідомлення користувачем. Для її реалізації використовується сервіс “UserSessionService”, який був розглянутий раніше. Ініціалізація повзунка, а також його переключення відбувається шляхом встановлення значення типу Bool змінної “isNotificationsEnabled”.

При натиску на опцію збережених стрічок, відбувається виклик відповідної поді в моделі представлення, яка передає шлях “savedMovies” до Router. За допомогою конфігуратора, ініціалізується екран списку кінострічок з типом “saved”, та показується на екрані.

Опція виходу з ауккаунту, викликає функцію “signOut”, сервісу “UserSessionService”. Вона розпочинає процес очищення сесії користувача. У випадку успішного завершення, відправляється відповідне сповіщення сервісу “NotificationManager”. Отримавши його, модуль AppFlow, змінює поточний екран користувача на сторінку авторизації.

При натиску на опцію придбаних квитків, модель представлення ViewModel, надсилає шлях “tickets”, який перехвачує Router. Опрацювавши дані, та, змінивши змінні, він, за допомогою Configurator, ініціалізує модуль білетів користувача та показує його на екрані.

Екран списку куплених квитків використовується для перегляду усіх актуальних і деактивованих покупок. На ньому представлені структури, що містять назву фільму, розташування кінотеатру, дату показу, а також місце та ряд сидіння у залі.

Робота ViewModel розпочинається з отримання списку моделей “TicketModel”, шляхом використання сервісу “TicketService”. Крім цього,

відбувається завантаження кінотеатрів та білетів для показу відповідної інформації на екрані. Якщо виникає відповідь, відповідний шлях “error” надсилається до Router, де в подальшому – відкривається на дисплеї.

Білети, дата яких менша за поточну, а також ті, що мають статус “true” змінної “isDisactivated”, буть позначені прозорішим кольором та не зможуть активуватись у кінотеатрі. Така реалізація дає змогу прибрати повторне використання квитків, які уже були задіяні, або дата показу яких, спливла.

При натиску на один із елементів списку модель представлення формує відповідний “TicketModel”, та передає його, разом із шляхом “ticket”, до Router, де відбувається ініціалізація модулю деталей білету та подальшим його відкриттям на дисплеї.

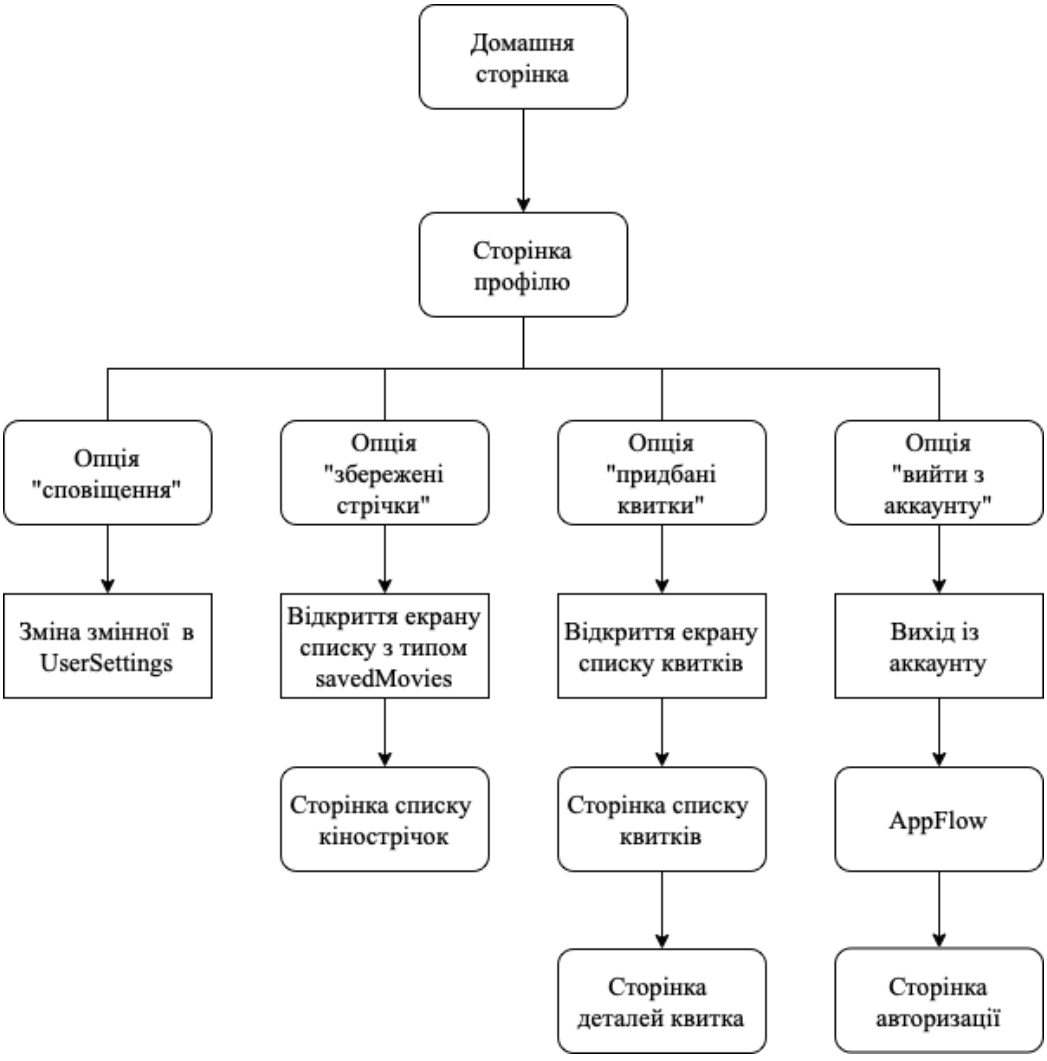


Рис. 4.8. Реалізація роботи модуля профіля

Екран деталей білету містить зображення QR коду, блок місця та ціни, деталі назви фільму, дати, кінотеатру та адресу. Крім цього навігаційна панель містить елемент для перегляду розташування кінотеатру на карті.

Для отримання QR коду використовується сервіс “QRCodeService”, який був розглянутий раніше. Отримання зображення проводиться шляхом виклику функції “generateQRCode”, якій передається “TicketModel” у вигляді JSON форматування. Місце та ціна білету визначається з тієї ж моделі. Крім того, для визначення фільму та кінотеатру, а також їх деталей, проводиться завантаження відповідних структур даних по отриманих ідентифікаторах.

Користувач має можливість побачити розташування кінотеатру на карті. Для цього достатньо натиснути кнопку “locate”, на навігаційній панелі. Відповідний запит відправляється у ViewModel, який генерує шлях “map”, передаючи модель з координатами “LocationModel”, до Router. В свою чергу відбувається відкриття карти на дисплеї.

Модель представлення модуля розташування кінотеатру, за допомогою системної бібліотеки “MapKit”, генерує структуру карти. Їй передаються поточне місце користувача а також отримані координати квитка. Таким чином спрощується пошук розташування, де буде показаний фільм.

4.2. Проведення тестувань додатку

Тестування є невід’ємним етапом процесу розробки додатку. Адже воно дозволяє знайти вразливі місця, дослідити поведінку програми в різних умовах, а також визначити причину падіння коду [20]. Крім того, тестування програмного забезпечення, формує об’єктивне та незалежне уявлення про продукт, що розробляється, таким чином даючи можливість зрозуміти та оцінити ризики, пов’язані з його впровадженням.

Існує велика кількість можливих тестів для різних компонентів додатку. Усі методи тестування програмного забезпечення використовують певний алгоритм для вибору відповідних тестів відповідно до наявного часу та ресурсів. Під час тестування, зазвичай намагаються запустити програму, що знаходить помилки в

роботі додатку. Вона надає об’єктивну та незалежну інформацію про якість програмного забезпечення, а також визначає ризики їх збою.

Одним із найпоширенішим методом тестування, є розробка Unit тестів. Це метод тестування додатку, при якому блоки вихідного коду (один або кілька наборів комп’ютерних програмних модулів) перевіряються разом із відповідними вхідними даними, використанням операційних процедур, щоб визначити, чи досягають вони запланованого значення [21].

Ідея Unit тестування полягає в тому, щоб написати випадки для кожної нетривіальної функції, або ж методу в модулі, щоб кожен випадок був незалежним від решти. Тоді за допомогою інтеграційних тестів можна забезпечити правильну роботу системи чи підсистеми.

Для прикладу розробимо Unit тести для перевірки email та отримання коректного списку усіх кінострічок. Для цього використаємо середовище розробки XCode, яке уже має налаштовані області для цього.

Для першого випадку створюємо змінну email, яка буде містити правильно написану почтову скриньку – “test@mailforspam.com”. Після чого у константу “isValid”, записуємо результат виконання функції “isValidEmail”, яка буде містити інформацію про перевірку email по регулярному виразі, визначеному раніше.

Для отримання значення тестування, буде використана функція “XCTAsserTrue”, вхідними даними якої буде константа “isValid”. Результат буде мати значення “true”, а тому Unit тест завершиться з позитивним статусом.

Замінивши email на “test@”, можна побачити негативну відповідь, яка проявляється червоною іконкою в полі статусу. Це означає, що під час виконання програми, виникла помилка. Місце її виникнення позначається відповідним кольором.

```
8   import XCTest
9   @testable import Cinema
10
12  class CinemaTests: XCTestCase {
12
14      func testEmailValidationCorrect() throws {
14
15          // Given
16          let email = "test@mailforspam.com"
17
18          // Wait
19          let isValid = email.isValidEmail
20
21          // Then
22          XCTAssertTrue(isValid)
23
24      }
25
27      func testEmailValidationUncorrect() throws {
27
28          // Given
29          let email = "test@"
30
31          // Wait
32          let isValid = email.isValidEmail
33
34          // Then
35          XCTAssertTrue(isValid)
36
37      }
38
39  }
40
```

Рис. 4.9. Реалізація перевірки email

Розробимо Unit тест для перевірки наявності завантаженого списку кінострічок. На початку виконання, створимо об’єкт сервісу “MovieService”. Після чого виконаємо функцію “getAllMovies”, для отримання списку усіх фільмів.

У блоці позитивного виконання, перевіримо наявність елементів, шляхом використання виразу “XCTAssertEqueal”. Першим вхідним параметром, передаймо значення “!movies.isEmpty”, яке буде перевіряти наявність об’єктів у масиві. Другим полем відправимо значення “true”. Таким чином функція “XCTAssertEqueal”, перевіряє еквівалентність отриманих параметрів.

У блоці негативного виконання використаємо вираз “XCTAssertNil”, на вхід якому передаймо значення отриманої помилки. Ця функція перевіряє наявність вхідних параметрів. А тому у разі помилки – ми отримаємо саме її.

Результатом виконання цього Unit тесту буде позитивний статус, адже сервіс “MovieService” повертає список фільмів, отриманих з бази даних Firestore.

```
8   import XCTest
9   @testable import Cinema
10
11  class CinemaTests: XCTestCase {
12
13      func testMoviesLoading() throws {
14
15          /// Given
16          let moviesService = MovieService()
17
18          // Wait
19          moviesService.fetchAllMovies { movies in
20
21              // Then
22              XCTAssertEqual(!movies.isEmpty, true)
23          } failure: { error in
24
25              // Then
26              XCTAssertNil(error)
27          }
28      }
29
30  }
```

Рис. 4.10. Реалізація перевірки завантаження кінострічок

Таким чином Unit тестування дає можливість завчасно перевірити правильність реалізації програмного коду, визначити місце помилки, а також передбачити можливість падіння додатку.

4.3. Поширення програми у AppStore

Для того, щоб користувачі мали змогу завантажити свій додаток на операційну систему iOS, необхідно розмістити його у магазині AppStore. Це онлайн магазин програмного забезпечення, який був розроблений Apple [22]. Додатки можуть бути платними, або безкоштовними. AppStore дозволяє користувачам переглядати та завантажувати програми з iTunes Store.

Але перед відправкою програми у AppStore, необхідно купити аккаунт, який відкриє можливості для цього. Після отримання статусу розробника, необхідно

визначити ідентифікатор програми, який буде унікальним, та використовуватиметься для розміщення додатку на телефонах користувачів.

Створивши сторінку програми у AppStore, розробник має можливість вивантажити бінарний код програми. Його отримати шляхом архівування проекту розробленого в XCode. Далі за використанням цієї програми, або ж додатку Transporter, потрібно вивантажити програму, вказавши усі необхідні сертифікати дистрибуції аккаунту.

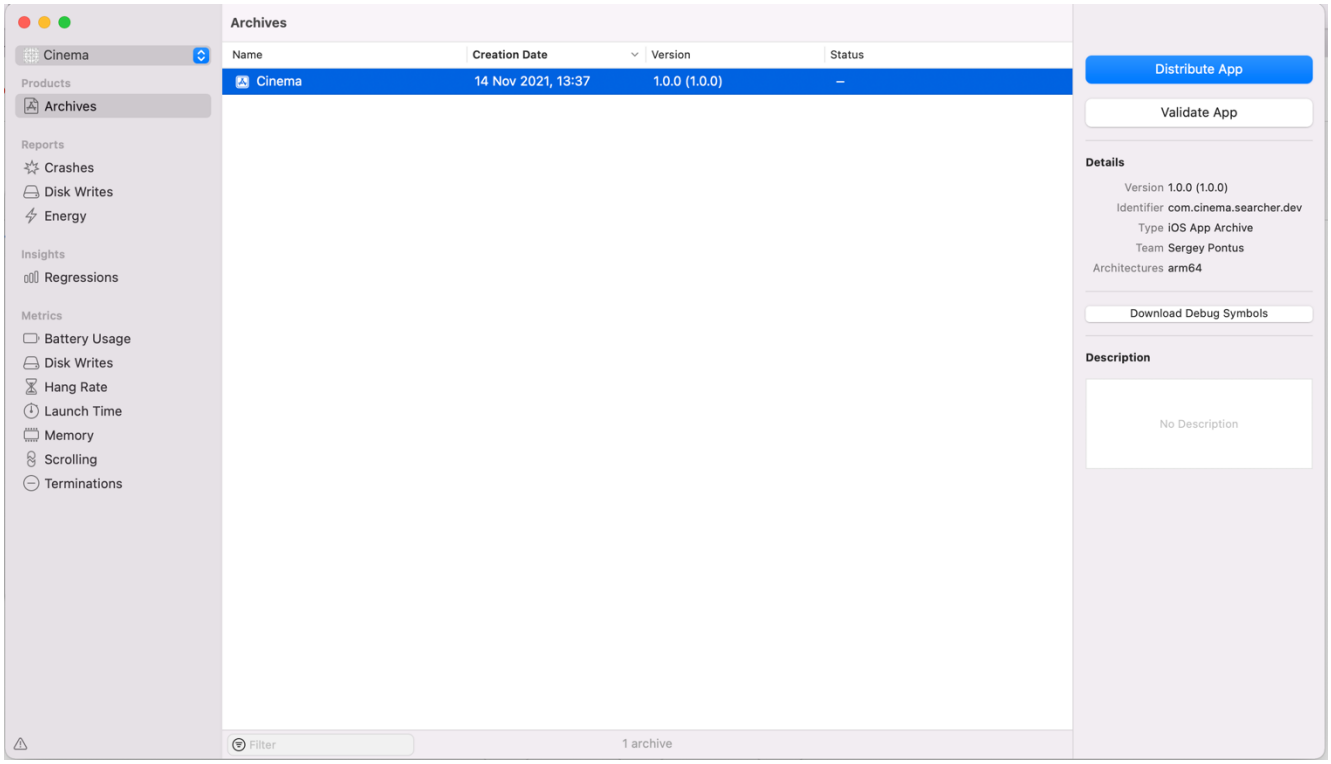


Рис. 4.11. Екран вивантаження архіву

Пройшовши перший етап перевірки AppStore на програмному рівні, додаток допускається до внутрішнього або зовнішнього тестування, а також створенню версії для публікації. TestFlight [23] — це програма, розроблена компанією Apple Inc. для операційної системи iOS, за допомогою якої можна встановлювати та тестувати додатки через AppStore, які знаходяться в бета-версії. Розробники можуть розповсюджувати свої програми невеликій кількості людей, які, у свою чергу, можуть надсилати відгуки та збої, безпосередньо на сервіс.

Сформувавши версію для публікації, а також заповнивши усі необхідні поля, розробник має змогу відправити реліз додатку на перевірку. Вона включає в себе людське тестування додатку зі сторони Apple, а також перевірку проходження усіх правил AppStore [24].

Після успішного перегляду програми, вона буде опублікована на сервісі, а усі користувачі матимуть змогу завантажити її. Сторінка додатку буде містити інформацію, заповнену раніше, список необхідних доступів та відгуки користувачів.

Таким чином відбувається формування додатку, його вивантаження, тестування бета-версії, перевірка та публікація у AppStore.

ВИСНОВОК ДО РОЗДІЛУ 4

В розділі було розглянуто основні аспекти розробки, та проаналізовано варіанти реалізації екранів додатку на базі iOS. Досліджено головний екран додатку та екрани авторизації. Розроблено домашню сторінку та списки кінострічок. Проаналізовано роботу екранів покупки білету та профілю користувача.

Досліджено реалізацію тестування програмного забезпечення, розроблено Unit тести для перевірки даних додатку. Розглянуто процес вивантаження, тестування, та публікації додатку у магазині AppStore.

ВИСНОВКИ

В даній дипломній роботі було проаналізовано розподіл сучасних цифрових пристроїв у світі та визначено найзручніший із переліку. Було розглянуто статистику використання мобільних операційних систем, досліджено їх недоліки та переваги. Переглянуто використання та особливості існуючих додатків операційної системи iOS. Було досліджено засоби та методи розробки програмного забезпечення, визначено найбільш підходящу мову програмування та середовище розробки. Оцінено та вибрано сервіс для роботи з віддаленою базою даних.

Було розглянуто інформативність даних та проаналізовано їх використання при роботі з мобільними додатками. Оцінено та визначено, відповідно до завдання, найоптимальніший шаблон архітектури для розробки додатку на базі iOS. Проаналізовано використання додаткових бібліотек, для спрощення розробки програмного забезпечення. Розроблено структури та даних для їх використання у додатку. Визначено модель і оцінено роботу віддаленої бази даних Firestore.

Було проаналізовано та сформовано модуль для роботи з базою даних на основі трьох ланок – менеджеру, розширень та сервісів. Розглянуто головні варіанти авторизації та реєстрації в системі. Оцінено реалізацію системи автентифікації користувача. Розроблено модуль налаштувань програми. Проаналізовано способи покупки у програмі та вибрано найзручніший. Реалізовано функціонал для проведення операцій оплати у додатку. Розглянуто модуль для формування QR-коду по отриманому текстовому значенню.

Було визначено основні елементи розробки та реалізації інтерфейсу користувача мобільного додатку на базі iOS. Визначено та реалізовано роботу головного екрану додатку і екранів автентифікації. Досліджено структуру домашньої сторінки та списків кінострічок. Проведено аналіз послідовності покупки білету та профілю користувача. Оцінено тестування програмного забезпечення та розглянуто розробку Unit тестів, що використовуються для перевірки роботи додатку. Було розглянуто роботу по просуванню додатку,

описано процес вивантаження, тестування та публікації додатку у магазині AppStore.

В подальшому розвитку розробленого додатку має бути реалізація функціоналу покупки білету у вигляді подарунку, розширення мережі кінотеатрів, покращення системи автентифікації за допомогою соціальних мереж, а також підвищення якості коду та модернізація надійності, системи безпеки користувача, пришвидшення роботи додатку та впровадження нових технологій.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Цифровий пристрій – Режим доступу: https://uk.wikipedia.org/wiki/Цифровий_пристрій (дата звернення 12.09.2021). – Назва з екрана

2. The device trends to know – Режим доступу: https://www.gwi.com/reports/device (дата звернення 12.09.2021). – Назва з екрана

3. Mobile operating systems' market share worldwide from January 2012 to June 2021 – Режим доступу: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. (дата звернення 12.09.2021). – Назва з екрана

4. KinoZone - Кіноафіша України – Режим доступу: https://apps.apple.com/us/app/kinozone-кіноафіша-україни/id1446673381 (дата звернення 12.09.2021). – Назва з екрана

5. Кіноріум: Всі фільми і серіали – Режим доступу: https://apps.apple.com/ua/app/id1093171715?l=uk (дата звернення 12.09.2021). – Назва з екрана

6. Fander – Режим доступу: https://apps.apple.com/ua/app/fander-гид-в-мир-кино/id1185465114?l=uk (дата звернення 12.09.2021). – Назва з екрана

7. Swift vs Objective-C – Режим доступу: https://medium.com/swiftify/swift-vs-objective-c-comparison-32aba9dad4e3 (дата звернення 12.09.2021). – Назва з екрана

8. XCode – Режим доступу: https://developer.apple.com/xcode (дата звернення 12.09.2021). – Назва з екрана

9. Firebase – Режим доступу: https://uk.wikipedia.org/wiki/Firebase (дата звернення 12.09.2021). – Назва з екрана

10. iOS Architecture Patterns – Режим доступу: https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52 (дата звернення 05.11.2021). – Назва з екрана

11. Cocoapods – Режим доступу: https://cocoapods.org (дата звернення 05.11.2021). – Назва з екрана

12. Firestore – Режим доступу: https://cloud.google.com/firestore (дата звернення 06.11.2021). – Назва з екрана

13. Sign in with Apple – Режим доступу: https://developer.apple.com/sign-in-with-apple/ (дата звернення 06.11.2021). – Назва з екрана

14. Working with UserDefaults in Swift – Режим доступу: https://learnappmaking.com/userdefaults-swift-setting-getting-data-how-to/ (дата звернення 07.11.2021). – Назва з екрана

15. In-App Purchase – Режим доступу: https://developer.apple.com/in-app-purchase/ (дата звернення 08.11.2021). – Назва з екрана

16. Apple Pay – Режим доступу: https://www.apple.com/uk/apple-pay/ (дата звернення 08.11.2021). – Назва з екрана

17. QR-код – Режим доступу: https://uk.wikipedia.org/wiki/QR-код (дата звернення 09.11.2021). – Назва з екрана

18. Interface Builder: – Режим доступу: https://en.wikipedia.org/wiki/Interface_Builder (дата звернення 13.11.2021). – Назва з екрана

19. SwiftUI: https://developer.apple.com/xcode/swiftui/ (дата звернення 13.11.2021). – Назва з екрана

20. Тестування програмного забезпечення – Режим доступу: https://uk.wikipedia.org/wiki/Тестування_програмного_забезпечення (дата звернення 14.11.2021). – Назва з екрана

21. Unit testing – Режим доступу: https://en.wikipedia.org/wiki/Unit_testing (дата звернення 14.11.2021). – Назва з екрана

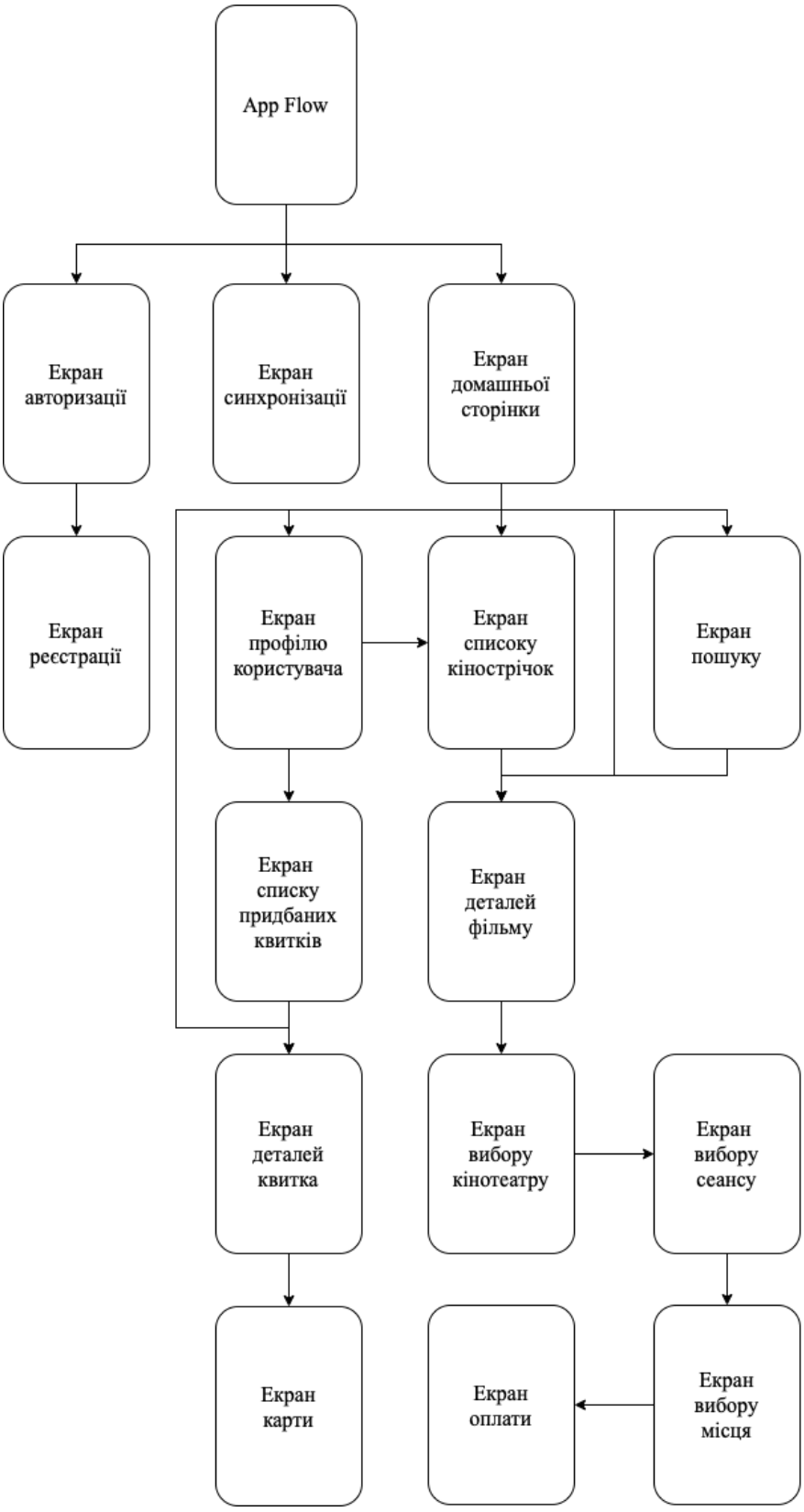
22. AppStore – Режим доступу: https://www.apple.com/ua/app-store/ (дата звернення 14.11.2021). – Назва з екрана

23. Бета тестування за допомогою TestFlight – Режим доступу: https://training.qatestlab.com/blog/technical-articles/beta-testing-via-testflight/ (дата звернення 14.11.2021). – Назва з екрана

24. App Store Review Guidelines – Режим доступу: https://developer.apple.com/app-store/review/guidelines/ (дата звернення 14.11.2021). – Назва з екрана

ДОДАТОК А

Структура екранів розробленого мобільного додатку



Код програми модуля роботи з сесією користувача

```
import FirebaseAuth
import Combine

private enum SignInError: String {
    case none = ""
    case empty = "Email або пароль пустий"
    case emailNotValid = "Невалідний Email"
}


class UserSessionService: ObservableObject {

    static let shared = UserSessionService()

    /// Auth instance
    private var auth = Auth.auth()
    /// User listener
    private var listener : AuthStateDidChangeListenerHandle? = nil
    /// Notification center
    private lazy var notificationCenter = NotificationCenter.default

    /// User
    var user: User? = nil

    /// Is user loaded
    var isUserLoaded: Bool = false

    init() {
        self.configureListener()
    }

    deinit {
        if let listener = self.listener {
            auth.removeStateDidChangeListener(listener)
            self.listener = nil
        }
    }
}

extension UserSessionService {

    var isAuthorized: Bool {
        return user != nil
    }

}

extension UserSessionService {


    /// Configure User Listener
    private func configureListener() {
        listener = auth.addStateDidChangeListener { auth, user in
            self.user = user
            self.isUserLoaded = true
```

```
            self.notificationCenter.post(name: user == nil ? .userAuthorized : .userUnauthorized,
object: user)
        }
    }

    /// Validation
    private func validate(email: String, password: String) -> (result: Bool, error: SignInError) {
        if email.isEmpty || password.isEmpty {
            return (false, .empty)
        }

        if !email.isValidEmail {
            return (false, .emailNotValid)
        }

        return (true, .none)
    }

    /// Sign Up
    func signUp(email: String, password: String, success: (() -> Void)? = nil, failure: ((String)
-> Void)? = nil) {
        let validation = validate(email: email, password: password)
        if !validation.result {
            failure?(validation.error.rawValue)
            return
        }
        auth.createUser(withEmail: email, password: password) { (result, error) in
            if let _ = result {
                success?()
            } else {
                failure?(error?.localizedDescription ?? "Sign up error")
            }
        }
    }

    /// Sign In
    func signIn(email: String, password: String, success: (() -> Void)? = nil, failure: ((String)
-> Void)? = nil) {
        let validation = validate(email: email, password: password)
        if !validation.result {
            failure?(validation.error.rawValue)
            return
        }
        auth.signIn(withEmail: email, password: password) { (result, error) in
            if let _ = result {
                success?()
            } else {
                failure?(error?.localizedDescription ?? "Sign in error")
            }
        }
    }

    /// Sign Out
    func signOut() {
        try? auth.signOut()
    }
}
```

Код програми менеджера роботи з базою даних

```
import Foundation
import FirebaseFirestore
import SwiftyJSON

final class DatabaseManager {

    private let database = Firestore.firestore()

}

// MARK: - Write
extension DatabaseManager {

    /// Write document
    func writeTiketDocument(_ document: [String: Any],
                            tiketId: String,
                            userId: String,
                            success: (() -> Void)?,
                            failure: ((String) -> ())?) {

        database.collection("tickets").document("users").collection(userId).document(tiketId).setD
ata(document) { error in
            if let error = error {
                failure?(error.localizedDescription)
            } else {
                success?()
            }
        }
    }

}

// MARK: - Read
extension DatabaseManager {

    func readMovies(success: (([JSON]) -> Void)?, failure: ((String) -> ())?) {
        database.collection("movies").getDocuments { querySnapshot, error in
            if let error = error {
                failure?(error.localizedDescription)
            } else {
                let list = querySnapshot!.documents.map { JSON($0.data()) }
                success?(list)
            }
        }
    }

    func readGenres(success: (([JSON]) -> Void)?, failure: ((String) -> ())?) {
        database.collection("genres").getDocuments { querySnapshot, error in
            if let error = error {
                failure?(error.localizedDescription)
            } else {
                let list = querySnapshot!.documents.map { JSON($0.data()) }
                success?(list)
            }
        }
```

```
    }

    func readCinemas(success: (([JSON]) -> Void)?, failure: ((String) -> ())?) {
        database.collection("cinemas").getDocuments { querySnapshot, error in
            if let error = error {
                failure?(error.localizedDescription)
            } else {
                let list = querySnapshot!.documents.map { JSON($0.data()) }
                success?(list)
            }
        }
    }

    func readTickets(userId: String, success: (([JSON]) -> Void)?, failure: ((String) -> ())?) {
        database.collection("tickets").document("users").collection(userId).getDocuments {
querySnapshot, error in
            if let error = error {
                failure?(error.localizedDescription)
            } else {
                let list = querySnapshot!.documents.map { JSON($0.data()) }
                success?(list)
            }
        }
    }

}
```

Код програми розширення та сервісу кінострічок для роботи з базою даних

```
import Foundation
import SwiftyJSON
```

```
protocol MovieServiceProtocol {

    func fetchAllMovies(success: (([MovieModel]) -> Void)?,
                        failure: ((String) -> Void)?)

    func fetchCurrentMovies(success: (([MovieModel]) -> Void)?,
                            failure: ((String) -> Void)?)

    func fetchCurrentMovies(searchString: String?,
                            success: (([MovieModel]) -> Void)?,
                            failure: ((String) -> Void)?)

    func fetchFutureMovies(success: (([MovieModel]) -> Void)?,
                           failure: ((String) -> Void)?)

    func fetchFutureMovies(searchString: String?,
                           success: (([MovieModel]) -> Void)?,
                           failure: ((String) -> Void)?)

    func fetchGenreMovies(genreId: String,
                          success: (([MovieModel]) -> Void)?,
                          failure: ((String) -> Void)?)

    func fetchSavedMovies(success: (([MovieModel]) -> Void)?,
                          failure: ((String) -> Void)?)

    func fetchMovie(id: String,
                    success: ((MovieModel?) -> Void)?,
                    failure: ((String) -> Void)?)

}
```

```
final class MovieService: MovieServiceProtocol {

    func fetchAllMovies(success: (([MovieModel]) -> Void)?,
                        failure: ((String) -> Void)?)
    {
        DatabaseService.shared.getMovies { json in
            let jsonDecoder = JSONDecoder()
            let models = json.compactMap { element -> MovieModel? in
                do {
                    let jsonData = try element.rawData()
                    return try jsonDecoder.decode(MovieModel.self, from: jsonData)
                } catch {
                    print("❌ Decoder error - \(error)")
                    return nil
                }
            }
            success?(models)
        } failure: { error in
            failure?(error)
```

```
        }
    }

    func fetchCurrentMovies(success: (([MovieModel]) -> Void)?, failure: ((String) -> Void)?) {
        fetchCurrentMovies(searchString: nil, success: success, failure: failure)
    }

    func fetchCurrentMovies(searchString: String?,
                            success: (([MovieModel]) -> Void)?,
                            failure: ((String) -> Void)?)
    {
        fetchAllMovies { movies in
            var movies = movies
                .filter { $0.startDate < Date() }

            if let searchString = searchString, !searchString.isEmpty {
                movies = movies
                    .filter { $0.title.lowercased().contains(searchString.lowercased()) }
            }

            success?(movies)
        } failure: { error in
            failure?(error)
        }
    }

    func fetchFutureMovies(success: (([MovieModel]) -> Void)?, failure: ((String) -> Void)?) {
        fetchFutureMovies(searchString: nil, success: success, failure: failure)
    }

    func fetchFutureMovies(searchString: String?,
                           success: (([MovieModel]) -> Void)?,
                           failure: ((String) -> Void)?)
    {
        fetchAllMovies { movies in
            var movies = movies.filter { $0.startDate > Date() }

            if let searchString = searchString, !searchString.isEmpty {
                movies = movies
                    .filter { $0.title.lowercased().contains(searchString.lowercased()) }
            }

            success?(movies)
        } failure: { error in
            failure?(error)
        }
    }

    func fetchGenreMovies(genreId: String,
                          success: (([MovieModel]) -> Void)?,
                          failure: ((String) -> Void)?)
    {
        fetchAllMovies { movies in
            let movies = movies.filter { $0.genres.contains(genreId) }
            success?(movies)
        } failure: { error in
```

```
            failure?(error)
        }
    }

    func fetchSavedMovies(success: (([MovieModel]) -> Void)?,
                          failure: ((String) -> Void)?)
    {
        let savedIds = UserSettingsService.shared.likedMoviesIds
        fetchAllMovies { movies in
            let movies = movies.filter { savedIds.contains($0.id) }
            success?(movies)
        } failure: { error in
            failure?(error)
        }
    }

    func fetchMovie(id: String,
                    success: ((MovieModel?) -> Void)?,
                    failure: ((String) -> Void)?)
    {
        fetchAllMovies { movies in
            let movies = movies.filter { $0.id == id }
            success?(movies.first)
        } failure: { error in
            failure?(error)
        }
    }

}
```

```
extension DatabaseService {

    func getMovies(success: (([JSON]) -> Void)?, failure: ((String) -> Void)?) {
        database.readMovies { json in
            success?(json)
        } failure: { _ in
            failure?("Не вдалось завантажити список стрічок")
        }
    }

}
```

Код програми сервісу роботи з налаштуваннями користувача

```
import Foundation

private enum Keys {
    static let likedMovies = "liked.movies.user.settings.key"
    static let isNotificationEnabled = "is.notifications.enabled.settings.key"
}

final class UserSettingsService {

    static let shared = UserSettingsService()

}

extension UserSettingsService {

    var likedMoviesIds: [String] {
        get {
            let string = UserDefaults.standard.string(forKey: Keys.likedMovies)
            return string?.components(separatedBy: ",") ?? []
        }
        set {
            let string = newValue.joined(separator: ",")
            UserDefaults.standard.set(string, forKey: Keys.likedMovies)
        }
    }

    var isNotificationEnabled: Bool {
        get {
            UserDefaults.standard.bool(forKey: Keys.isNotificationEnabled)
        }
        set {
            UserDefaults.standard.set(newValue, forKey: Keys.isNotificationEnabled)
        }
    }


}
```

Код програми сервісу формування QR-коду

```
import Foundation
import CoreImage.CIFilterBuiltins
import SwiftUI
import UIKit

final class QRCodeService {

    static func generateQRCode(from string: String) -> Image {
        let context = CIContext()
        let filter = CIFilter.qrCodeGenerator()
        let data = Data(string.utf8)

        filter.setValue(data, forKey: "inputMessage")

        if let outputImage = filter.outputImage {
```

```
            if let cgimg = context.createCGImage(outputImage, from:
outputImage.extent) {
                return Image(uiImage: UIImage(cgImage: cgimg))
            }
        }

        return Image(uiImage: UIImage(systemName: "xmark.circle") ?? UIImage())
    }
}
```

Код моделі представлення структури головної сторінки

```
import Foundation
import Combine

final class HomePageViewModel: BaseViewModel {

    // MARK: - Input

    /// Profile command
    let onProfileCommand = PassthroughSubject<Void, Never>()

    /// Search command
    let onSeachCommand = PassthroughSubject<Void, Never>()

    /// Movie details command
    let onMovieCommand = PassthroughSubject<MovieModel, Never>()

    /// Genre movies list command
    let onGenreCommand = PassthroughSubject<GenreModel, Never>()

    /// More current movies command
    let onMoreCurrenMoviesCommand = PassthroughSubject<Void, Never>()

    /// More future movies command
    let onMoreFutureMoviesCommand = PassthroughSubject<Void, Never>()

    // MARK: - Output

    @Published var username: String = ""

    /// Genres items
    @Published var genres: [GenreModel] = []

    /// Current movies items
    @Published var currentMovies: [MovieModel] = []

    /// Future movies items
    @Published var futureMovies: [MovieModel] = []

    /// Like command
    let onPopToRootTrigger = PassthroughSubject<Void, Never>()

    // MARK: - Private

    /// Router
    private let router: HomePageRouter

    /// Genre service
    private let genreService: GenreServiceProtocol

    /// Movie service
    private let movieService: MovieServiceProtocol

    // MARK: - Initializations & Setups

    init(router: HomePageRouter) {
```

```
        self.router = router
        self.genreService = GenreService()
        self.movieService = MovieService()
        super.init()
    }

    override func setupBinding() {
        super.setupBinding()

        let moviePurchaseEnded = NotificationCenter.default
            .publisher(for: .moviePurchasedEnded)

        onProfileCommand
            .map { .profile }
            .assign(to: \.router.route, on: self)
            .store(in: &bag)

        onSeachCommand
            .map { .search }
            .assign(to: \.router.route, on: self)
            .store(in: &bag)

        onMovieCommand
            .map { .movie(id: $0.id) }
            .assign(to: \.router.route, on: self)
            .store(in: &bag)

        onMoreCurrenMoviesCommand
            .map { .moreMovies(dataType: .current) }
            .assign(to: \.router.route, on: self)
            .store(in: &bag)

        onMoreFutureMoviesCommand
            .map { .moreMovies(dataType: .future) }
            .assign(to: \.router.route, on: self)
            .store(in: &bag)

        onGenreCommand
            .map { .moreMovies(dataType: .genre(id: $0.id, title: $0.title)) }
            .assign(to: \.router.route, on: self)
            .store(in: &bag)

        moviePurchaseEnded
            .compactMap { $0.object as? String }
            .sink { [weak self] _ in
                self?.onPopToRootTrigger.send()
            }
            .store(in: &bag)

        moviePurchaseEnded
            .compactMap { $0.object as? String }
            .map { .ticket(id: $0) }
            .assign(to: \.router.route, on: self)
            .store(in: &bag)
```

```
        loadData()
    }

}

// MARK: - Data loading
private extension HomePageViewModel {

    func loadData() {
        loadGenres()
        loadCurrentMovies()
        loadFutureMovies()
    }

    func loadGenres() {
        genreService.fetchGenres { [weak self] genres in
            self?.genres = genres
        } failure: { [weak self] error in
            self?.router.route = .error(string: error)
        }
    }

    func loadCurrentMovies() {
        movieService.fetchCurrentMovies { [weak self] movies in
            self?.currentMovies = movies
        } failure: { [weak self] error in
            self?.router.route = .error(string: error)
        }
    }

    func loadFutureMovies() {
        movieService.fetchFutureMovies { [weak self] movies in
            self?.futureMovies = movies
        } failure: { [weak self] error in
            self?.router.route = .error(string: error)
        }
    }

}
```

Код конфігуратора головної сторінки

```
import Foundation

protocol HomePageChildConfigurator {

    func makeMovieDetails(id: String) -> MovieDetailsView

    func makeTicketDetails(ticketId: String) -> TicketDetailsView

    func makeMoviesList(dataType: MoviesListViewModel.DataType) -> MoviesListView

    func makeSearch() -> SearchView

    func makeProfile() -> ProfileView

}

final class HomePageConfigurator {

    private let movieDetailsConfigurator: MovieDetailsConfigurator

    private let tickeDetailsConfigurator: TicketDetailsConfigurator

    private let moviesListConfigurator: MoviesListConfigurator

    private let searchConfigurator: SearchConfigurator

    private let profileConfigurator: ProfileConfigurator

    init(movieDetailsConfigurator: MovieDetailsConfigurator,
         tickeDetailsConfigurator: TicketDetailsConfigurator,
         moviesListConfigurator: MoviesListConfigurator,
         searchConfigurator: SearchConfigurator,
         profileConfigurator: ProfileConfigurator)
    {
        self.movieDetailsConfigurator = movieDetailsConfigurator
        self.tickeDetailsConfigurator = tickeDetailsConfigurator
        self.moviesListConfigurator = moviesListConfigurator
        self.searchConfigurator = searchConfigurator
        self.profileConfigurator = profileConfigurator
    }

    func make() -> HomePageView {
        let router = HomePageRouter(configurator: self)
        let viewModel = HomePageViewModel(router: router)
        let view = HomePageView(viewModel: viewModel, router: router)
        return view
    }

}

extension HomePageConfigurator: HomePageChildConfigurator {

    func makeMovieDetails(id: String) -> MovieDetailsView {
        movieDetailsConfigurator.make(id: id)
    }
```

```
    func makeTicketDetails(ticketId: String) -> TicketDetailsView {
        tickeDetailsConfigurator.make(ticketId: ticketId)
    }

    func makeMoviesList(dataType: MoviesListViewModel.DataType) -> MoviesListView {
        moviesListConfigurator.make(dataType: dataType)
    }

    func makeSearch() -> SearchView {
        searchConfigurator.make()
    }

    func makeProfile() -> ProfileView {
        profileConfigurator.make()
    }

}
```

Код представлення головної сторінки

```
import SwiftUI

struct HomePageView: View {

    /// View model
    @ObservedObject var viewModel: HomePageViewModel

    /// Router
    @ObservedObject var router: HomePageRouter

    var body: some View {
        ScrollView {
            VStack(spacing: 30) {
                HomePageNavigationView(onProfileCommand:
viewModel.onProfileCommand)
                HomePageSearchView(onSeachCommand: viewModel.onSeachCommand)
                HomePageCategoriesView(genres: viewModel.genres, onGenreCommand:
viewModel.onGenreCommand)
                HomePageMovieSegmentView(title: "Зараз у кіно",
                                         movies: viewModel.currentMovies,
                                         onMovieCommand: viewModel.onMovieCommand,
                                         onMoreCommand:
viewModel.onMoreCurrenMoviesCommand)
                HomePageMovieSegmentView(title: "Скоро у кіно",
                                         movies: viewModel.futureMovies,
                                         onMovieCommand: viewModel.onMovieCommand,
                                         onMoreCommand:
viewModel.onMoreFutureMoviesCommand)
                Spacer()
            }
            .padding(.top, 50)
            .padding(.bottom, 30)

            NavigationLink(destination: router.navigationView,
                           isActive: $router.isNavigationShowed,
                           label: { Color.clear })
                    .frame(height: 0)
        }
        .edgesIgnoringSafeArea(.all)
        .backgroundColor(.c2B080A)
        .rootNavigationBarView(title: "", isHidden: true)
        .fullScreenCover(isPresented: $router.isFullScreenShowed, content: {
router.fullScreen })
        .alert(isPresented: $router.isAlertShowed, content: { router.alertView })
        .onReceive(viewModel.onPopToRootTrigger) { NavigationUtil.popToRootView()
}
    }

}
```

Код роутера головної сторінки

```
import Foundation
import SwiftUI

final class HomePageRouter: BaseRouter {

    enum Route {
        case none
        case error(string: String)
        case movie(id: String)
        case ticket(id: String)
        case moreMovies(dataType: MoviesListViewModel.DataType)
        case search
        case profile
    }

    // MARK: - Input

    @Published var route: Route = .none

    // MARK: - Output

    // MARK: - Private

    private var configurator: HomePageChildConfigurator

    // MARK: - Initializations and setups

    init(configurator: HomePageChildConfigurator) {
        self.configurator = configurator
        super.init()
        setupBinding()
    }

    override func setupBinding() {
        setupPresentations()
        setupValues()
    }

    func setupPresentations() {

        $route
            .map {
                switch $0 {
                case .error:
                    return true
                default:
                    return false
                }
            }
            .assign(to: \.isAlertShowed, on: self)
            .store(in: &bag)

        $route
            .map {
                switch $0 {
```

```
                case .ticket:
                    return true
                default:
                    return false
                }
            }
            .assign(to: \.isFullScreenShowed, on: self)
            .store(in: &bag)

        $route
            .map {
                switch $0 {
                case .movie, .moreMovies, .search, .profile:
                    return true
                default:
                    return false
                }
            }
            .assign(to: \.isNavigationShowed, on: self)
            .store(in: &bag)

    }

    func setupValues() {

        $route
            .compactMap {
                switch $0 {
                case .error(let message):
                    return Alert(title: Text("Помилка"), message: Text(message), dismissButton:
.cancel(Text("OK")))
                default:
                    return nil
                }
            }
            .assign(to: \.alertView, on: self)
            .store(in: &bag)

        $route
            .compactMap {
                switch $0 {
                case .ticket(let ticketId):
                    return AnyView(self.configurator.makeTicketDetails(ticketId: ticketId))
                default:
                    return nil
                }
            }
            .assign(to: \.fullScreen, on: self)
            .store(in: &bag)

        $route
            .compactMap { [weak self] in
                guard let self = self else { return nil }
                switch $0 {
                case .movie(let id):
                    return AnyView(self.configurator.makeMovieDetails(id: id))
```

```
                case .moreMovies(let dataType):
                    return AnyView(self.configurator.makeMoviesList(dataType: dataType))
                case .search:
                    return AnyView(self.configurator.makeSearch())
                case .profile:
                    return AnyView(self.configurator.makeProfile())
                default:
                    return nil
                }
            }
            .assign(to: \.navigationView, on: self)
            .store(in: &bag)

    }


}
```