Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Модель мобільного додатку з застосуванням фреймворку Flutter

(тема)

Виконав:

студент II курсу, групи СПм-22-4

Бешта В.С.
(прізвище, ініціали)

Спеціальність

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

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

Освітня програма

Системне програмування
(повна назва освітньої програми)

Керівник: доц. Філімончук Т.В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

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

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

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Бешті Вячеславу Сергійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Модель мобільного додатку з застосуванням фреймворку Flutter

затверджена наказом по університету від “ 01 ” квітня 2024 р. № 257Ст

2. Термін подання студентом роботи до екзаменаційної комісії 15 червня 2024 р.

3. Вхідні дані до роботи 1) архітектурний шаблон BloC; 2) сервіс Firebase; 3) фреймворк Flutter; 4) база даних Sqflite; 5) мова програмування Dart; 6) інтегроване середовище розробки Visual Studio Code

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз переваг та недоліків операційних платформ, що використовуються для створення мобільних додатків;
2) порівняння нативних та кросплатформних підходів до розробки мобільних додатків;
3) інструменти для розробки мобільних додатків;
4) огляд компонентів фреймворку Flutter;
5) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів)

Слайд-презентація – 12 слайдів

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

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

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд операційних платформ, які використову-	02.04.24-08.04.24	
	ються при розробці мобільних додатків		
2	Огляд інструментарію для розробки	09.04.24-16.04.24	
	мобільного застосунку		
3	Огляд принципів об'єктно-орієнтованого	17.04.24-22.04.24	
	програмування		
4	Розробка моделі мобільного застосунку	23.04.24-24.05.24	
	з використанням фреймворку Flutter		
5	Оформлення матеріалів кваліфікаційної роботи	25.05.24-03.06.24	
6	Подання кваліфікаційної роботи керівникові	04.06.24-07.06.24	
	та її попередній захист		
7	Подання роботи на рецензування	8.06.24-12.06.24	

Дата видачі завдання 01 квітня 2024 р.

Студент _____
(підпис)

Керівник роботи _____ доц. Філімончук Т.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 73 с., 8 рис., 1 дод., 10 джерел.

FLUTTER, IOS, ANDROID, SDK, МОБІЛЬНИЙ ДОДАТОК, МОДЕЛЬ, ФРЕЙМВОРК, BLOC, API, ВІДЖЕТ.

Метою кваліфікаційної роботи є дослідження та аналіз сучасних тенденцій у розробці мобільних додатків з використанням фреймворку Flutter. Основною задачею дослідження є виявлення переваг та обмежень цього фреймворку, а також розробка та тестування моделі мобільного додатку на основі Flutter з використанням різних складових, включаючи інтерфейс користувача, бізнес-логіку, базу даних, хмарні сервіси та механізми безпеки.

У ході виконання кваліфікаційної роботи проводився аналіз актуальних тенденцій у ринку розробки мобільних додатків, вивчення основних характеристик фреймворку Flutter та порівняння його з іншими інструментами розробки. На основі проведеного аналізу була розроблена модель мобільного додатку, яка включає в себе всі необхідні компоненти для його успішної реалізації та функціональності з урахуванням всіх переваг та недоліків, які було отримано на етапі аналізу.

Основними результатами роботи є модель мобільного додатку з використанням фреймворку Flutter, де чітко розмежовано відповідні компоненти: інтерфейс користувача, логіка взаємодії, база даних та хмарні сервіси. Включення в модель модулю аналітики та моніторингу надає командам розробників безцінну інформацію про поведінку користувачів та продуктивність додатків, що полегшує прийняття рішень на основі даних для постійного покращення та доопрацювання додатку.

ABSTRACT

Master’s thesis:: 73 pages, 8 figures, 1 appendices, 10 sources.

FLUTTER, IOS, ANDROID, SDK, MOBILE APPLICATION, MODEL, FRAMEWORK, BLOC, API, WIDGET.

The purpose of the qualification work is to research and analyze current trends in the development of mobile applications using the Flutter framework. The main goal is to identify the advantages and limitations of this framework, as well as to develop and test a model of a mobile application based on Flutter using various components, including the user interface, business logic, database, cloud services, and security mechanisms.

The qualification work involved analyzing current trends in the mobile application development market, studying the main characteristics of the Flutter framework and comparing it with other development tools. Next, we developed a model of a mobile application that includes all the necessary components for its successful implementation and functionality.

The main results of the work are the developed model of a mobile application using the Flutter framework, as well as the advantages and disadvantages of this framework in comparison with alternative development tools. The work includes the analysis and justification of each component of the model, as well as the study of its impact on the functionality and performance of the mobile application.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 8
ВСТУП 9
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 10
1.1 Актуальність мобільних технологій 10
1.2 Тенденції ринку розробки додатків 11
1.3 Основні вимоги користувачів до мобільних додатків 12
1.4 Нативні засоби побудови мобільного додатку 13
1.5 Кросплатформні засоби побудови мобільного додатку 15
2 ТЕХНОЛОГІЧНІ ОСНОВИ РОЗРОБКИ МОБІЛЬНИХ ДОДАТКІВ 17
2.1 Мова програмування Dart 17
2.2 Кросплатформний фреймворк Flutter 19
2.2.1 Переваги та обмеження використання фреймворку Flutter 20
2.2.2 Рушії рендерингу Flutter 21
2.2.3 Архітектура Flutter 22
2.3 BLoC архітектура 23
3 МОДЕЛЬ МОБІЛЬНОГО ДОДАТКУ НА ОСНОВІ FLUTTER 26
3.1 Існуюча модель 26
3.2 Модель мобільного застосунку, що запропонована 27
3.2.1 Інтерфейс користувача 27
3.2.2 Клієнтська частина на мобільній платформі 29
3.2.3 Логіка взаємодії застосунку 30
3.2.4 Бізнес-логіка мобільного додатку 31
3.2.5 Застосування бази даних у мобільному додатку 32
3.2.6 Інтерфейс прикладного програмування мобільних додатків 35
3.2.7 Хмарні сервіси при розробці мобільних додатків 37
3.2.8 Використання механізмів безпеки в мобільних додатках 38
3.2.9 Механізми аналітики та моніторингу 40

3.2.10 Модуль тестування для мобільного додатку 42
4 РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ ДЛЯ ТЕСТУВАННЯ МОДЕЛІ 44
4.1 Структура тестового додатку 44
4.2 Модуль інтерфейсу користувача: SignInPage віджет 45
4.3 Визначення типу мережевого з'єднання 48
4.4 Оптимізація інтерфейсу користувача через логіку валідації 52
4.5 Використання бізнес логіки 53
4.6 Використання сервісу баз даних 55
4.7 Використання хмарних сервісів 56
4.9 Сервіси аналітики та моніторингу 61
ВИСНОВКИ 64
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 66
ДОДАТОК А Графічний матеріал кваліфікаційної роботи 67

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – інтерфейс прикладного програмування (англ. Application Programming Interface)

BLOC – архітектурний підхід розробки мобільного додатку (англ. Business Logic Component)

HTTP – протокол передачі гіпер-текстових документів, що використовується в комп'ютерних мережах (англ. Hyper Text Transfer Protocol)

SDK – комплект для розробки програмного забезпечення (англ. Software Development Kit)

ВСТУП

У сучасному інформаційному суспільстві розробка мобільних додатків стає все більш важливою та актуальною задачею. Розмаїття пристроїв та платформ, швидкі темпи технологічного розвитку та постійні зміни у вимогах користувачів створюють потребу в ефективних інструментах для розробки програмного забезпечення. Одним із сучасних рішень у цій області є використання фреймворку Flutter, розробленого компанією Google.

Фреймворк Flutter надає зручний та швидкий спосіб розробки крос-платформних мобільних додатків з високоякісним інтерфейсом користувача. Його особливість полягає у тому, що він дозволяє створювати однаковий інтерфейс для різних платформ, таких як iOS та Android, з використанням одного і того ж коду. Це дозволяє розробникам значно економити час та зусилля при створенні та підтримці мобільних додатків.

Проте використання фреймворку Flutter вимагає вивчення його архітектурних принципів та використання певних підходів при розробці додатків. У зв'язку з цим, проведення науково-дослідницької практики з аналізу та розвитку моделі мобільного додатку з використанням фреймворку Flutter є актуальним та перспективним напрямком.

Це дослідження розглядає потенціал Flutter шляхом розробки та аналізу моделі мобільного додатку. Ця модель включатиме різні компоненти, включаючи дизайн інтерфейсу користувача, реалізацію бізнес-логіки, інтеграцію з базами даних, використання хмарних сервісів та заходи безпеки. Оцінюючи цю модель, дослідження має на меті визначити переваги та обмеження використання Flutter для розробки мобільних додатків та сприяти глибшому розумінню його практичного значення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Розвиток мобільних технологій та поширення смартфонів призвели до значного збільшення популярності мобільних додатків у різних сферах життя. Сучасне суспільство стає все більш мобільним, зростає попит на зручні та функціональні додатки, що задовольняють різноманітні потреби користувачів. Особливо це стосується розваг, освіти, бізнесу, медицини та інших галузей.

Мобільні додатки стали не лише зручними інструментами для виконання певних задач, але й необхідним елементом сучасного життя. Завдяки їм, користувачі можуть отримати доступ до різноманітних сервісів, інформації та розваг у будь-який час та в будь-якому місці, просто з використанням свого мобільного пристрою. Із зростанням популярності мобільних додатків також збільшується конкуренція серед розробників, що ставить перед ними задачу створення продуктів, які не лише відповідають потребам користувачів, але й виходять за їхні очікування [1].

Тому аналіз предметної області стає ключовим етапом при розробці мобільних додатків з використанням фреймворку Flutter. Ретельне вивчення потреб цільової аудиторії, тенденцій ринку їх розробки та конкурентного середовища дозволяє зрозуміти, які саме функції та можливості повинен мати мобільний додаток, щоб він був успішним.

1.1 Актуальність мобільних технологій

Мобільні технології перетворили спосіб, яким ми спілкуємося, працюємо та розважаємося, змінюючи наш підхід до комунікації, вирішення завдань та відпочинку. Вони стали не лише невід'ємною частиною нашого повсякденного життя, але й необхідним інструментом для багатьох бізнес-процесів. За даними дослідження "Mobile Economy 2024" від GSMA Intelligence кількість активних мобільних абонентів світу перевищила

6 мільярдів, що підкреслює важливість мобільних технологій у сучасному суспільстві [2].

Завдяки швидкому зростанню популярності мобільних пристроїв, попит на мобільні додатки стрімко зростає. За даними "Global Mobile App Revenue 2021" від App Annie, загальний дохід від мобільних додатків у світі досяг понад 400 мільярдів доларів США у 2021 році, що відображає постійне зростання популярності цього сегмента ринку [3].

Сьогодні люди використовують мобільні додатки для комунікації, споживання контенту, здійснення покупок, вирішення фінансових питань та багато іншого. Згідно з дослідженням "DIGITAL 2023: GLOBAL OVERVIEW REPORT" від DataReportal середній користувач витрачає понад 5 години на мобільних пристроях щодня, що підкреслює значущість мобільних додатків у сучасному суспільстві [4].

Нові технології, такі як швидке мобільне Інтернет-з'єднання, розширена реальність (Augmented Reality) та штучний інтелект, роблять їх ще потужнішими та функціональнішими. Наприклад, додатки для розширеної реальності можуть створювати інтерактивні дослідницькі або розважальні простори, тоді як застосування штучного інтелекту може забезпечити персоналізовані рекомендації та покращити досвід користувача.

Однак разом із зростанням популярності мобільних додатків збільшується і їх конкуренція. Компанії, що бажають залишатися конкурентоспроможними на ринку, мають надавати високоякісні, інноваційні та привабливі додатки для своїх користувачів. І тут важливою стає можливість швидкої та ефективної розробки та їх розгортання на різних платформах.

1.2 Тенденції ринку розробки додатків

Ринок мобільних додатків постійно змінюється та розвивається, відображаючи нові технологічні можливості та вимоги користувачів. Для розуміння тенденцій ринку розробки мобільних додатків необхідно

проаналізувати кілька ключових аспектів.

По-перше, важливо врахувати зростаючий попит на персоналізовані додатки. Користувачі очікують, що вони будуть пропонувати індивідуалізований досвід, відповідно до їхніх унікальних потреб та переваг. Крім того, зростає значення безпеки даних в них. Користувачі стають все більш обізнаними щодо приватності та безпеки своїх особистих даних, тому важливо, щоб додатки забезпечували високий рівень захисту та дотримувалися відповідних стандартів безпеки.

По-друге, на ринку спостерігається зростаючий попит на додатки, які спрощують життя та підвищують продуктивність. Як правило, користувачі шукають додатки, які допоможуть їм ефективно керувати часом, планувати задачі та організовувати робочі процеси.

По-третє, важливим аспектом є розуміння змін у способі використання мобільних додатків користувачами. Сьогодні користувачі вимагають не лише функціональних додатків, але й таких, які надають емоційний досвід та взаємодію з ними. Дизайн та досвід користувача (UX) стають ключовими факторами їх успіху. Розробники повинні ретельно вивчати потреби та побажання своїх користувачів, враховувати останні тенденції у дизайні та взаємодії з ними.

Також варто відзначити зростаючий тренд використання штучного інтелекту (ШІ) та машинного навчання: ШІ може використовуватися для покращення функціональності додатків, враховуючи поведінку користувачів та надаючи персоналізовані рекомендації [5].

Загалом, розуміння тенденцій ринку дозволяє розробникам створювати додатки, які відповідають потребам та очікуванням користувачів, а також використовувати найновітніші технології для досягнення цієї мети.

1.3 Основні вимоги користувачів до мобільних додатків

При розробці мобільних додатків з використанням фреймворку Flutter важливо враховувати основні вимоги користувачів, оскільки вони

визначають успіх та їх прийняття на ринку. Дотримання цих вимог допомагає забезпечити зручність використання, задоволення від користування та позитивне сприйняття продукту серед аудиторії.

Однією з найважливіших аспектів у розробці мобільних додатків є задоволення основних вимог аудиторії. Користувачі мають високі очікування щодо функціональності, продуктивності та їх дизайну. Зокрема, вони очікують, що додаток буде легким у використанні, інтуїтивно зрозумілим та ефективним.

Іншою з основних вимог клієнтів є швидка реакція додатку на відповідні дії [6], тобто аудиторія очікує, що він буде реагувати миттєво на їхні взаємодії без зайвих затримок чи зависань. Крім того, вони вимагають, щоб інтерфейс користувача був зрозумілим та легким у навігації, а всі функції були доступні на достатньо високому рівні зручності.

Для багатьох клієнтів також важливою є безпека їхніх даних та конфіденційність інформації [7]. Вони очікують, що додаток буде забезпечувати надійний захист їхніх персональних даних та не допустить їх незаконного використання чи втрати.

Окрім цього, зовнішній вигляд додатку також має велике значення для користувачів. Вони хочуть, щоб інтерфейс був естетичним, сучасним та привабливим, а дизайн відповідав останнім трендам у світі UX/UI.

Узагальнюючи, основні вимоги аудиторії до мобільних додатків включають швидкість та продуктивність, зрозумілість та легкість у використанні, надійну безпеку даних та естетичний дизайн. Врахування цих вимог у процесі розробки дозволить створити успішний та конкурентоздатний продукт.

1.4 Нативні засоби побудови мобільного додатку

Нативні засоби побудови мобільних додатків використовуються для розробки програмного забезпечення, яке спеціально оптимізоване для

конкретної мобільної платформи, такої як iOS або Android. Ці засоби дозволяють розробникам використовувати офіційні інструменти та мови програмування, надаючи повний доступ до функціоналу платформи, максимальну швидкість та продуктивність додатку.

На платформі iOS нативні додатки розробляються мовою програмування Swift або Objective-C, використовуючи інтегроване середовище розробки Xcode. Це надає розробникам доступ до усіх функцій та можливостей iOS, таких як фреймворки Cocoa Touch та інструменти розробки Apple.

На платформі Android нативні додатки розробляються мовою програмування Java або Kotlin, використовуючи Android Studio як інтегроване середовище розробки. Це дозволяє розробникам максимально використовувати можливості платформи Android та доступатися до всіх інструментів та функцій, які надаються Google.

Переваги нативних засобів включають високу продуктивність та швидкість роботи додатків, повний доступ до функціоналу платформи та можливість використання останніх технологій та інструментів розробки. Однак нативний підхід також має свої обмеження, зокрема, необхідність розробки окремих версій додатку для кожної платформи, що може збільшувати час та витрати на розробку та підтримку.

Крім того, нативні додатки зазвичай забезпечують кращу інтеграцію з екосистемою конкретної платформи, що може бути важливим фактором для користувачів. Наприклад, на операційній системі iOS нативні додатки можуть використовувати функції, які доступні тільки для пристроїв Apple, такі як Touch ID або Face ID для аутентифікації. Аналогічно на Android нативні додатки можуть легко інтегруватися з сервісами Google, такими як Google Maps або Google Pay.

Однак, розробка нативних додатків може бути витратною через потребу в написанні окремого коду для кожної платформи. Це також може ускладнити підтримку та їх оновлення, оскільки будь-які зміни повинні бути

внесені в обидві версії окремо. За останні кілька років на ринку з'явилися альтернативи нативному підходу, такі як кросплатформні фреймворки, які дозволяють розробникам створювати один додаток, який може працювати на різних платформах, таких як iOS та Android з використанням єдиної бази коду.

1.5 Кросплатформні засоби побудови мобільного додатку

Кросплатформні фреймворки, такі як React Native, Flutter та Xamarin надають розробникам можливість використовувати одну мову програмування та єдиний кодову базу для розробки додатків для різних платформ. Наприклад, фреймворк React Native оперує мовою JavaScript для розробки додатків, що дозволяє розробникам використовувати знайомі інструменти та бібліотеки для швидкої розробки.

Однією з переваг кросплатформних фреймворків є скорочення часу розробки та витрат на підтримку. Розробники можуть писати один і той самий код та обирати його для різних платформ, що дозволяє ефективно використовувати ресурси та прискорює процес розробки. Крім того, ці фреймворки надають широкий вибір інструментів та бібліотек для розробки, що сприяє розширенню можливостей додатків.

Розроблений компанією Google, фреймворк Flutter став дуже популярним серед розробників мобільних додатків. Він відомий своєю швидкістю та продуктивністю, а також забезпечує можливість створення красивих та динамічних інтерфейсів користувача.

Фреймворк React Native, розроблений компанією Facebook, використовує мову JavaScript. Він надає можливість використовувати компоненти React для швидкої розробки додатків для iOS та Android.

Розроблений компанією Microsoft, фреймворк Xamarin використовує мову програмування C# для створення кросплатформних додатків. Він дозволяє розробникам використовувати одну кодову базу для розробки кінцевого продукту для iOS, Android та Windows.

Ionic – це фреймворк який використовує мови HTML, CSS та JavaScript для розробки кросплатформних додатків. Він базується на фреймворках Angular та Cordova і надає розробникам можливість швидко створювати додатки з красивим інтерфейсом.

Однак, кросплатформність може мати свої недоліки, такі як обмежені можливості доступу до функцій платформи або втрата продуктивності порівняно з нативними додатками. Також, для вирішення деяких конкретних задач може бути потрібно писати додатковий код або використовувати платформозалежні рішення.

У будь-якому випадку, вибір між нативною розробкою та кросплатформним підходом залежить від конкретних потреб проєкту, ресурсів та стратегії розробки. Обидва підходи мають свої переваги та недоліки та важливо враховувати їх при виборі оптимального шляху розробки мобільних додатків.

2 ТЕХНОЛОГІЧНІ ОСНОВИ РОЗРОБКИ МОБІЛЬНИХ ДОДАТКІВ

При виборі інструментів для розробки мобільних додатків, важливо враховувати ряд ключових факторів. По-перше, мова програмування визначає можливості та функціональність фреймворку. Важливо обирати мови, які є популярними серед розробників та надають широкі можливості для реалізації потрібного функціоналу.

Крім того, слід враховувати наявність відомих та надійних фреймворків та бібліотек, які можуть значно спростити розробку та забезпечити стабільність фреймворку. Інтеграція систем автоматизованого тестування також є важливим аспектом, що дозволяє забезпечити високу якість та надійність фреймворку.

Крім технічних аспектів, важливо також мати належну документацію та активну спільноту розробників, що сприяє поширенню та використанню фреймворку, а також дозволяє обмінюватися досвідом та вирішувати проблеми разом. Обираючи оптимальні інструменти для створення фреймворку, розробник забезпечує успішну та ефективну розробку, яка відповідає потребам та вимогам проекту.

2.1 Мова програмування Dart

Dart – це мова програмування, що створена для розробки веб та мобільних застосунків. Розроблена компанією Google, вона може використовуватися для створення серверних та настільних додатків. Ця об'єктно-орієнтована мова програмування включає збирач сміття (garbage collector), базується на класах та має синтаксис у стилі C. Dart підтримує різні функціональні можливості, такі як інтерфейси, міксини, абстрактні класи, уніфіковані дженерики та виведення типу. Мова Dart була представлена на конференції GOTO в Орхусі, Данія, у жовтні 2012 року. Засновниками мови є Ларс Баком та Каспер Лундом, а версія Dart 1.0 була випущена 14 листопада

2013 року. Початкові плани щодо включення веб-машини Dart у браузер Google Chrome викликали неоднозначні реакції, але в 2015 році вони були скасовані з виходом версії Dart 1.9. У серпні 2018 року була випущена версія Dart 2.0 зі значними змінами, включаючи впровадження системи суворої перевірки типів, відомої як Sound null safety. Ця функція запобігає помилкам, що виникають внаслідок ненавмисного доступу до змінних, які мають значення null.

Нове розширення, dart2native, було представлено в Dart 2.6, воно дозволяє компілювати код Dart в автономний рідний код для платформ Linux, macOS та Windows. Розробники тепер можуть створювати автономні файли, що виконуються та компонувати програму Dart без необхідності встановлення Dart SDK. Нове розширення також інтегроване з інструментарієм Flutter, що дає можливість використовувати його у невеликих сервісах, наприклад, для підтримки бекенда.

Ecma International створила технічний комітет TC52 для роботи над стандартизацією мови Dart. Dart може бути скомпільована до стандартної JavaScript, і вона ефективно працює в будь-якому сучасному браузері. Перше видання специфікації мови Dart було затверджено Ecma International у липні 2014 року, а друге – у грудні 2014 року [9]. В останній доступній версії специфікації мови Dart включені всі необхідні відомості щодо мови.

Існує чотири способи запуску Dart-коду: web, Stand-alone, Ahead-of-time compiled та Native. Кожен з них має свої особливості та використовується для різних цілей та платформ.

Запуск веб-додатків зазвичай відбувається за допомогою Dart VM, яка вбудована в браузер або використовується через Dart DevTools. Stand-alone додатки можуть бути виконані з командного рядка за допомогою Dart VM та їх можна зібрати у файли, що виконуються, для різних операційних систем. Нарешті, Dart Native дозволяє створювати додатки для платформи (Android та iOS) шляхом компіляції коду Dart безпосередньо в нативний машинний код.

Останні версії Dart принесли численні вдосконалення та нововведення, такі як покращення системи типів, підтримка нульової безпеки, а також оновлення документації та інструментів розробки. Команда розробників постійно працює над вдосконаленням мови та її екосистеми, щоб забезпечити розробникам найкращі інструменти для створення сучасних програм.

2.2 Кросплатформний фреймворк Flutter

Flutter – це кросплатформний набір інструментів для створення інтерфейсів користувача, який призначений для можливості повторного використання коду на різних операційних системах, таких як iOS та Android, а також для можливості додатків взаємодіяти безпосередньо з основними сервісами платформи. Основна мета фреймворку полягає в тому, щоб дозволити розробникам створювати застосунки високої продуктивності, які будуть "натурально" відчуватися на різних платформах, враховуючи відмінності там, де вони існують, одночасно ділячи якомога більше коду.

Під час розробки додатків на Flutter вони виконуються в віртуальній машині, яка пропонує перезавантаження в режимі реального часу зі збереженням стану без необхідності повної перекомпіляції. Для релізу додатку Flutter компілюються безпосередньо в машинний код, будь то інструкції Intel x64 або ARM, або в JavaScript, якщо ціль веб. Фреймворк Flutter є відкритим для використання, має ліцензію BSD з дозволом на вільне поширення та має активну екосистему сторонніх пакетів, які доповнюють функціональність основної бібліотеки.

Dart – мова програмування, на якій ґрунтується Flutter, має різні методи компіляції, що дозволяють використовувати код на різних платформах та для різних цілей. Перш за все, є Just-in-Time (JIT) компіляція, яка використовується під час розробки для швидкого виявлення помилок гарячого перезавантаження та швидкого відлагодження. Код JIT компілюється на льоту під час виконання програми.

Другий метод – це Ahead-of-Time (AOT) компіляція, яка використовується для підготовки додатку до публікації. Слід розуміти, що код AOT компілюється в машинний код перед запуском додатку, що поліпшує його продуктивність та ефективність.

2.2.1 Переваги та обмеження використання фреймворку Flutter

Оскільки Flutter є одним із найбільш популярних та швидко зростаючих фреймворків для створення перехідних мобільних додатків, його розгляд як інструменту для розробки моделі мобільного додатку стає надзвичайно актуальним. Тому, докладне вивчення переваг та обмежень Flutter у контексті моделювання мобільних додатків стане корисним для розуміння його ефективності та придатності для конкретних вимог та задач.

Переваги використання фреймворку Flutter у розробці мобільних додатків є значними. Перш за все, він надає можливість розробки кросплатформних додатків: один і той же код може бути використаний додатком для різних платформ, таких як iOS та Android. Це значно зменшує час розробки та зусилля, оскільки розробники можуть уникнути необхідності писати окремий код для кожної платформи. Крім того, Flutter має високу швидкість рендерингу інтерфейсу користувача, що робить його ідеальним вибором для створення додатків з високою реактивністю та плавною анімацією. Він також має багатий набір готових компонентів, що спрощує розробку та дозволяє створювати привабливий та функціональний інтерфейс.

Незважаючи на ці переваги, важливо враховувати деякі обмеження використання Flutter. Однією з найбільших проблем може бути обмежена підтримка сторонніх бібліотек та модулів, особливо якщо ці бібліотеки специфічні для певної платформи. Крім того, Flutter ще молодий фреймворк порівняно з іншими, тому можуть виникати проблеми з сумісністю та стабільністю. Не всі функції, які доступні на платформах iOS та Android, можуть бути доступні у Flutter без додаткових налаштувань або обхідних рішень. Також слід враховувати, що хоча Flutter дозволяє розробникам

писати кросплатформний код, для деяких випадків все ж може знадобитися писати платформозалежний код для різних особливостей та функцій. Це може призвести до збільшення складності проєкту та зниження продуктивності.

Не зважаючи на ці обмеження, велика кількість проєктів та компаній успішно використовують фреймворк Flutter для розробки мобільних додатків, що свідчить про його значний потенціал у цій області.

2.2.2 Рушії рендерингу Flutter

Impeller та Skia – це два різних рендерингові рушії, які використовуються в Flutter для відображення графічного інтерфейсу користувача. Основна відмінність між ними полягає в тому, як вони компілюють та виконують шейдери – програми, які запускаються на GPU і визначають, як відображати різні форми та кольори на екрані. Skia – це рушій рендерингу 2D, розроблений Google, який використовується в багатьох продуктах, таких як Chrome, Android та Flutter, і має широкий набір функцій для рендерингу та роботи з графікою. Skia компілює шейдери під час виконання, тобто коли додаток вже запущено на пристрої. Це дозволяє Skia адаптуватися до різних графічних API, таких як OpenGL, Vulkan або Metal, і генерувати оптимальні шейдери для кожного з них.

Однак цей процес компіляції може займати значний час, що призводить до погіршення продуктивності та затримок у відтворенні анімацій, особливо при першому запуску додатка або першому використанні шейдера.

Impeller – це новий рушій рендерингу, розроблений командою Flutter спеціально з метою поліпшення продуктивності та усунення затримок у відтворенні анімацій. Його API та алгоритми рендерингу розроблені з урахуванням особливостей Flutter та спрямовані на мінімізацію накладних витрат і оптимізацію продуктивності. Impeller призначений для заміни Skia та потенційно забезпечує підтримку 3D, яка раніше була неможлива з Skia, оскільки він виключно підтримує 2D. На відміну від Skia, Impeller компілює

шейдери під час збірки самого рушія Flutter. Це означає, що додатки, які використовують Impeller, вже мають усі необхідні шейдери в скомпільованому вигляді, що дозволяє уникнути затримок у відтворенні анімацій. При цьому Impeller не впливає на розмір додатка або на швидкість його запуску в порівнянні зі Skia.

2.2.3 Архітектура Flutter

Фреймворк Flutter має складну архітектуру (рисунок 2.1), що включає в себе кілька ключових компонентів, які спільно працюють, щоб забезпечити високу продуктивність та ефективність розробки мобільних додатків.

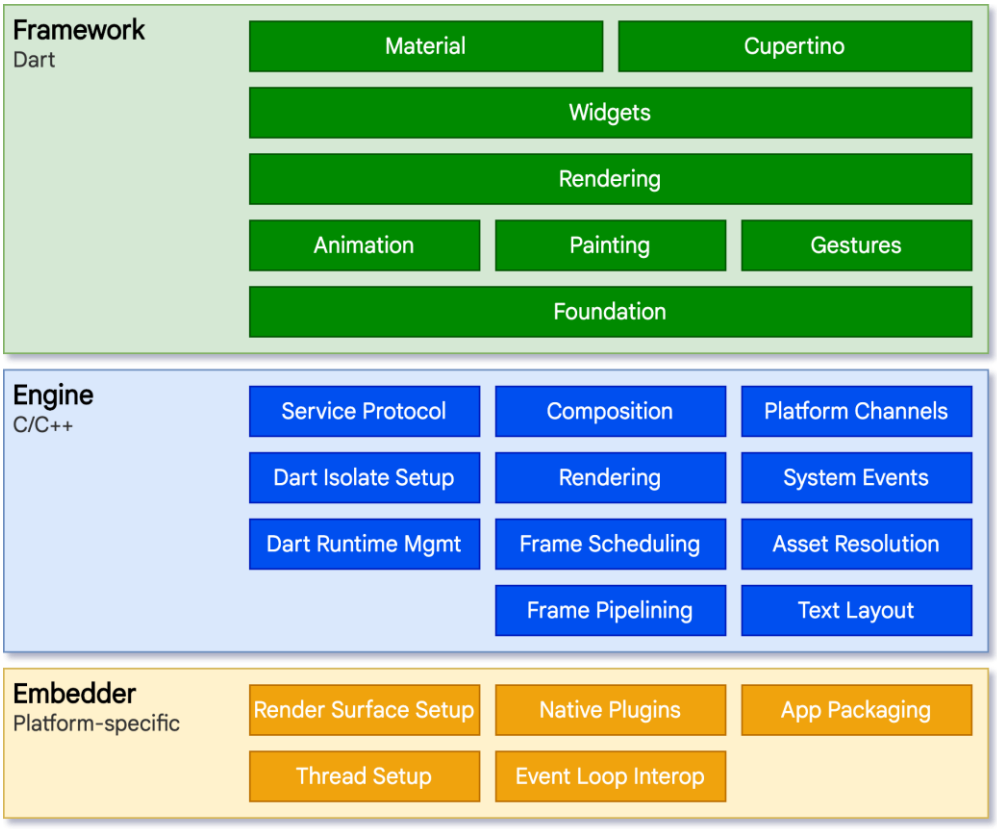

Рисунок 2.1 – Архітектура Flutter

Основними архітектурними компонентами виступають Embedder, Engine та Framework. Embedder відповідає за вбудовування гібридної програми в рідну оболонку платформи, таку як Android або iOS, та використовує мову, специфічну для кожної платформи, таку як Java для

Android або Swift/Objective-C для iOS.

Engine, написаний переважно мовою C++, відповідає за низькорівневу реалізацію API Flutter та забезпечує високу продуктивність та швидкість виконання додатків Flutter.

Основна частина функціональності Flutter відбувається через Framework, який включає багато бібліотек для роботи з анімацією, жестами та макетами. Кожен елемент у Flutter є віджетом, що дозволяє розробникам налаштовувати їх за потребою. Flutter також має функцію Hot Reload, яка дозволяє розробникам динамічно маніпулювати об'єктами візуалізації та бачити зміни миттєво без необхідності перезавантаження. Ці компоненти працюють разом, щоб забезпечити розробникам зручну та ефективну платформу для створення мобільних додатків на Flutter.

2.3 BLoC архітектура

Архітектура BLoC (Business Logic Component) – це методологія розробки програмного забезпечення, яка набула великої популярності в мобільній розробці, зокрема в середовищі Flutter. Ця архітектурна парадигма надає зручну структуру для організації коду (рисунок 2.2), розділяючи логіку додатка на невеликий набір блоків, які можна легко використовувати.

У центрі архітектури BLoC знаходяться блоки, кожен з яких відповідає за певну частину функціоналу. Вони обробляють вхідні події, такі як натискання кнопок або отримання даних з сервера, і реагують на них, видаючи змінений стан. Крім того, ці блоки можуть взаємодіяти з іншими частинами додатка, такими як інтерфейс користувача чи шар доступу до даних.

Ключовими складовими архітектури BLoC є події (events), стани (states) та потоки даних (data streams). Події визначають зміни стану, які відбуваються в додатку, стани відображають поточний стан блоку, а потоки даних забезпечують ефективну комунікацію між різними частинами додатку.

Цей підхід дозволяє розробникам створювати більш передбачувані та

керовані додатки, де бізнес-логіка чітко відокремлена від презентаційного шару. Це сприяє зручності тестування та повторного використання компонентів. Завдяки цьому підходу, зміни в одній частині коду не впливають на інші частини, що зменшує ймовірність виникнення помилок та спрощує підтримку додатка в довгостроковій перспективі.

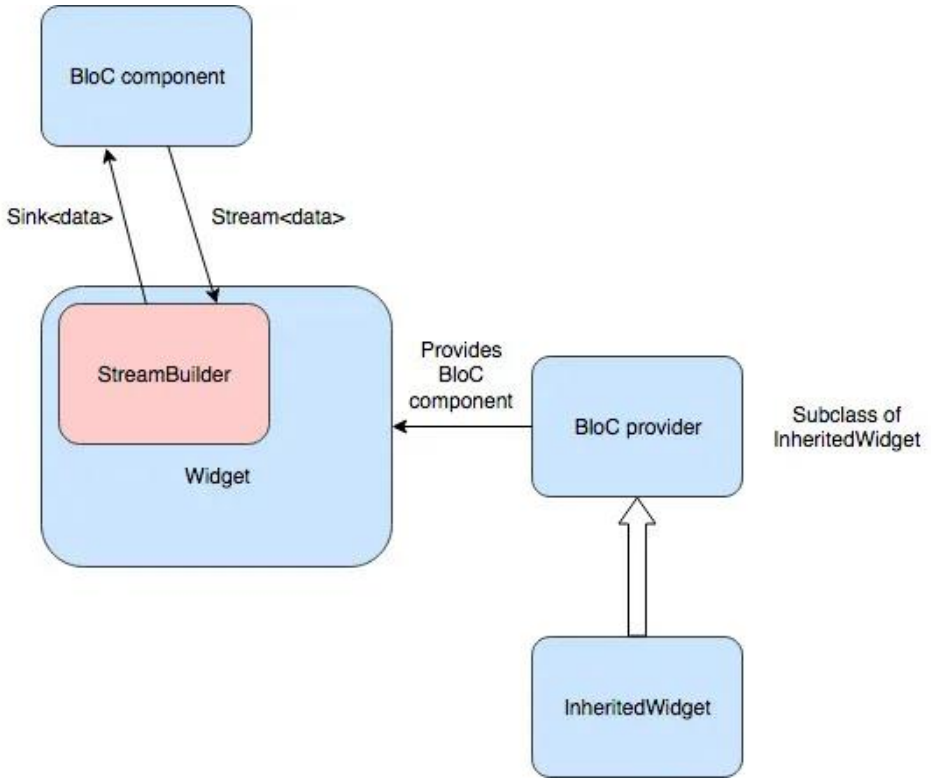

Рисунок 2.2 – Архітектура BLoC

InheritedWidget – це спеціальний вид віджета в Flutter, який дозволяє передавати дані вглиб дерева віджетів без необхідності передачі їх через кожен окремий віджет. Він створює залежність між батьківським та дочірнім віджетами, дозволяючи дочірнім віджетам отримувати доступ до спільного стану, який зберігається в InheritedWidget.

InheritedWidget відіграє важливу роль у передачі блоків даних вглиб дерева віджетів. Це дозволяє ефективно передавати доступ до екземплярів BLoC всім віджетам у додатку, які його потребують, без необхідності передачі їх через кожен окремий віджет. Наприклад, для використання BLoC, можна створити спеціалізований клас, який унаслідується від

InheritedWidget, і міститиме екземпляр BLoC.

Використання архітектури BLoC допомагає зробити код додатка більш структурованим, легким для розуміння та підтримки. Дана архітектура сприяє впровадженню принципів розділення відповідальностей та перевикористання коду, що є ключовими для успішної розробки складних мобільних додатків. За допомогою архітектури BloC розробники можуть створювати додатки, які є більш легкими для обслуговування, масштабування та розширення з часом.

3 МОДЕЛЬ МОБІЛЬНОГО ДОДАТКУ НА ОСНОВІ FLUTTER

3.1 Існуюча модель

Модель мобільних додатків на основі Flutter являє собою складну структуру, яка пояснює комплексну архітектуру, операційну функціональність і технічні складові, притаманні життєвому циклу розробки мобільних додатків.

По суті, модель мобільних додатків на основі Flutter визначає основні компоненти та їх синергетичну взаємодію, з акцентом на оптимізацію ефективності та задоволення очікувань користувачів. Модель охоплює набір критично важливих елементів, включаючи, але не обмежуючись ними, інтерфейс користувача (UI), бізнес-логіку (BL), базу даних (DB) та інтерфейс прикладного програмування (API). Ці компоненти ретельно розроблені для спільної роботи, гарантуючи, що додаток не тільки забезпечує безперебійну роботу користувача, але й точно виконує свої функції.

Цілісне представлення моделі мобільного додатку на основі Flutter може бути інкапсульоване кортежем 3.1:

$$M = \{ UI, BL, SEC, MT \}, \quad (3.1)$$

де: UI (User Interface) – інтерфейс взаємодії користувача з додатком;

BL (Business Logic) – компонент, що відповідає за виконання основних функцій програми;

SEC (Security) – механізм безпеки, який включає аутентифікацію та авторизацію;

MT (Mobile Testing) – модуль для створення та проведення тестів для перевірки роботи компонентів додатку і їх взаємодії.

Таким чином, модель мобільних додатків на основі Flutter пропонує комплексну схему для розробки мобільних додатків, які є не тільки візуально привабливими та зручними для користувача, але й функціонально надійними,

безпечними та ретельно протестованими. Адаптивність цієї моделі до різних методологій розробки та її здатність задовольнити широкий спектр типів мобільних додатків підкреслюють її універсальність та ефективність у сучасному ландшафті мобільної розробки.

3.2 Модель мобільного застосунку, що запропонована

Проводячи аналіз предметної області, можна сформулювати модель мобільного застосунку, яка базується на фреймворку Flutter:

$$M = \{ UI, CF, IL, BL, DB, CS, API, SEC, AM, MT \}, \quad (3.2)$$

де DB (Database) – база даних, яка відповідає за зберігання та організацію інформації;

API (Application Programming Interface) – інтерфейс програмування застосунків, який забезпечує комунікацію між клієнтською та серверною частинами;

CF (Client Frontend) – частина додатку, яка працює на мобільних платформах (iOS та Android);

IL (Interaction Logic) – логіка взаємодії, яка включає обробку даних та комунікацію з сервером через API;

CS (Cloud Services) – хмарні сервіси для зберігання файлів користувачів та управління ресурсами;

AM (Analytics and Monitoring) – інструмент для аналізу та моніторингу використання додатку.

3.2.1 Інтерфейс користувача

UI (User Interface) або інтерфейс користувача, є важливою складовою будь-якого мобільного додатку, оскільки він забезпечує зв'язок між користувачем та програмою. Цей аспект відповідає за відображення інформації на екрані пристрою та створення зручного середовища для

взаємодії з додатком.

У своїй основі UI містить різноманітні елементи, такі як кнопки, поля введення, меню, списки та інше, які відображаються на екрані пристрою і дозволяють користувачеві виконувати різні дії та отримувати потрібну інформацію. Основні задачі UI – це створення зручного та естетичного інтерфейсу, який би відповідав потребам користувачів та сприяв зручності взаємодії з програмою.

У Flutter, UI побудовано за допомогою набору віджетів, які можуть бути поєднані та налаштовані для створення різноманітних інтерфейсів. Віджети в Flutter є найбільшими будівельними блоками інтерфейсу, вони можуть включати в себе кнопки, тексти, картинки, поля введення, списки та інші елементи, що визначають зовнішній вигляд та функціональність додатку (рисунок 3.1). Крім того, Flutter надає можливість створювати власні віджети або модифікувати наявні, що дозволяє розробникам налаштовувати інтерфейс додатку відповідно до їх потреб та дизайну.

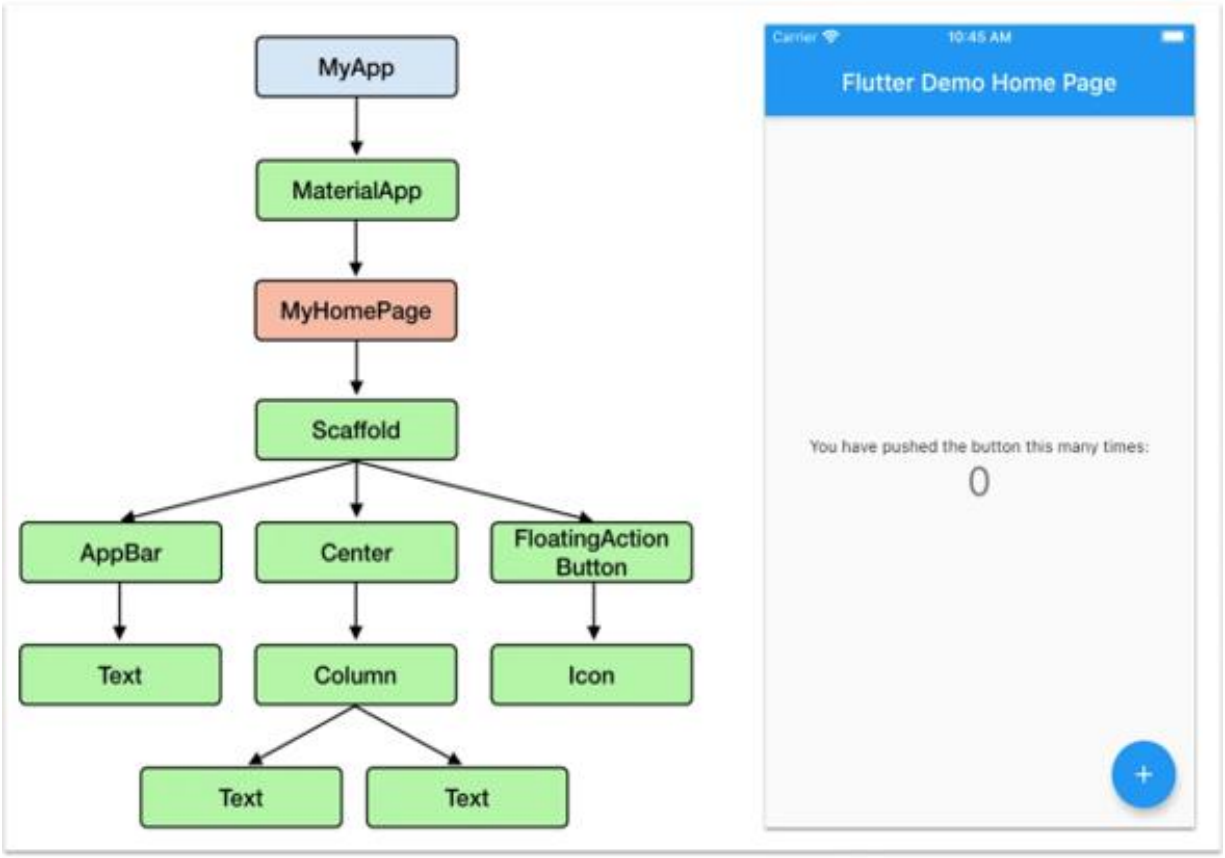

Рисунок 3.1 – Ієрархія віджетів

Завдяки гнучкості та потужному набору інструментів для розробки інтерфейсу Flutter став популярним вибором для розробників мобільних додатків, які прагнуть створити красиві, швидкі та користувацько-орієнтовані додатки для різних платформ.

3.2.2 Клієнтська частина на мобільній платформі

Клієнтська частина на мобільній платформі (CF) є невід'ємною складовою моделі мобільного додатку. Цей компонент відповідає за реалізацію інтерфейсу користувача на конкретних мобільних пристроях, таких як iOS та Android. У контексті розробки мобільних додатків CF забезпечує взаємодію з користувачем та обробку його взаємодії з додатком. Це означає, що CF відповідає за відображення різних екранів, кнопок, полів введення, списків, тощо, які користувач може бачити та взаємодіяти з ними.

Під час розробки CF, розробники зазвичай використовують можливості Flutter для створення кросплатформного інтерфейсу, що однаково добре працює як на пристроях з iOS, так і на пристроях з Android. Це дозволяє зменшити зусилля, необхідні для розробки та підтримки двох окремих версій інтерфейсу для кожної платформи [11]. Крім того, CF включає в себе реалізацію анімацій, переходів між екранами, оптимізацію швидкості роботи інтерфейсу та інші аспекти, що забезпечують зручність та ефективність використання додатку користувачем на мобільних платформах iOS та Android. CF також надає набір API, які дозволяють розробникам отримати доступ до специфічних функцій пристрою, таких як камера, GPS та контакти.

Наприклад, у сценарії взаємодії мобільного застосунку, який передбачає використання QR-кодів для коректної роботи, компонент клієнтського інтерфейсу (CF) може активувати функцію камери пристрою з метою захоплення зображення QR-коду. Після цього отримані дані, інтерпретовані як QR-код, піддаються подальшій обробці за допомогою внутрішнього програмного інтерфейсу (API) додатку. Такий підхід до інтеграції можливостей пристрою та функціоналу серверної частини стає

однією з ключових переваг використання фреймворку Flutter для створення мобільних додатків.

3.2.3 Логіка взаємодії застосунку

Компонент (Interaction Logic, IL) відповідає за обробку взаємодії користувача з додатком та сервером через API. Він включає в себе компоненти, які обробляють дані, що були введені користувачем через інтерфейс та ініціюють необхідні дії на основі цих даних. Наприклад, при натисканні кнопки на інтерфейсі користувача, IL може виконувати певну дію, таку як відправлення запиту на сервер для отримання додаткової інформації або збереження змін у базі даних.

Також компонент IL відповідає за взаємодію з серверною частиною додатку через API. Він інтерпретує дані, отримані від сервера, та забезпечує їх правильне відображення на інтерфейсі користувача. Крім того, IL може відповідати за обробку різних станів додатку, таких як підключення до мережі, обробка помилок або оновлення даних у реальному часі.

Основною метою IL є забезпечення плавної та зручної взаємодії користувача з додатком, а також ефективної комунікації з серверною частиною.

Інтерфейс користувача також отримує дані з сервера і відповідно до них оновлює інтерфейс програми. Наприклад, якщо користувач шукає товар у додатку для електронної комерції, інтерфейс користувача надсилає пошуковий запит на сервер, отримує результати пошуку, а потім відображає їх в інтерфейсі додатку.

Основна мета IL – це забезпечити безперебійну та зручну роботу користувача, а також ефективний зв'язок з сервером. Щоб досягти цього, інтерфейс повинен бути здатним обробляти широкий спектр взаємодій користувача, від простих натискань кнопок до складних задач з обробки даних.

3.2.4 Бізнес-логіка мобільного додатку

Компонент бізнес-логіки (Business Logic, BL) є ядром додатку (рисунок 3.2), визначаючи його функціональність та забезпечуючи надійність. Він відповідає за реалізацію основних функцій і бізнес-процесів, які визначають призначення та поведінку програми. BL слугує мозковим центром, який перетворює бізнес-вимоги та правила на дієвий код, забезпечуючи ефективну та результативну роботу додатку.

Крім того, компонент бізнес-логіки часто включає в себе обробку даних, взаємодію з базою даних та іншими системами, а також забезпечує валідацію введених даних та управління станом додатку. Його правильна організація та реалізація є критично важливою для стабільної та ефективної роботи мобільного додатку.

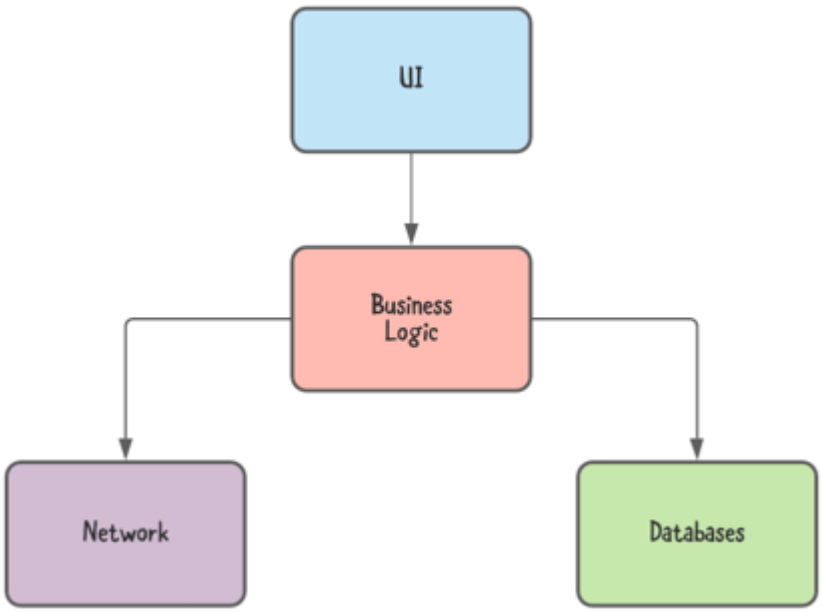

Рисунок 3.2 – Місто компоненту "Бізнес логіка" у застосунку

Однією з ключових переваг BL є її здатність забезпечувати дотримання бізнес-правил та логіки додатків. Ці правила визначають межі та обмеження, в яких працює додаток, узгоджуючи його з бізнес-цілями організації та галузевими стандартами. Наприклад, у фінансовому додатку BL забезпечує дотримання правил, пов'язаних з перевіркою транзакцій, авторизацією та

дотриманням нормативних вимог. Включаючи ці правила в логіку програми, BL гарантує, що програма буде ефективно та безпечно функціонувати в реальних умовах.

Всі дії та робочі процеси в додатку підтримуються та керуються BL. Цей компонент обробляє, перевіряє та трансформує дані, забезпечуючи їх безперебійну передачу між різними компонентами та системами. BL також керує станом програми, відстежуючи сеанси користувачів, їхні вподобання та іншу контекстну інформацію, щоб забезпечити персоналізований та послідовний досвід роботи.

3.2.5 Застосування бази даних у мобільному додатку

У сфері розробки мобільних додатків база даних (БД) відіграє ключову роль, слугуючи наріжним камінцем для ефективного управління та пошуку даних. В рамках Flutter база даних є ключовим компонентом, на який покладається відповідальність за організацію та зберігання життєво-важливої інформації, що забезпечує функціональність додатку.

Основна мета бази даних в контексті мобільного додатку – це забезпечити структурований та ефективний механізм для зберігання даних. Вона діє як централізоване сховище, що містить основні дані, від яких залежить безперебійне функціонування додатку. Ці дані охоплюють різноманітні елементи, включаючи вподобання користувача, налаштування програми, мультимедійні файли та динамічний контент. Застосовуючи системний підхід до організації даних, база даних гарантує, що інформація є легкодоступною та може бути ефективно використана різними компонентами програми.

Значення бази даних виходить за рамки простого зберігання даних. Вона дає можливість різним модулям програми безперешкодно взаємодіяти зі збереженою інформацією. Завдяки чітко визначеним інтерфейсам та механізмам пошуку даних, база даних полегшує обмін даними та їх доступність, забезпечуючи безперешкодну інтеграцію та координацію між

функціональними можливостями програми. Це, в свою чергу, покращує загальний досвід користувачів, надаючи їм негайний доступ до потрібної інформації.

При розробці бази даних для мобільного додатку розробники можуть обирати між різними типами баз даних. Реляційні бази даних, такі як SQLite, використовуються для зберігання структурованих даних у вигляді таблиць з реляційними зв'язками між ними. Наприклад, мобільний додаток для управління задачами може використовувати SQLite для зберігання списку завдань та їх атрибутів, таких як назва, опис та дата додавання.

NoSQL бази даних, наприклад Firebase Realtime Database або MongoDB, набувають популярності у мобільній розробці завдяки своїй гнучкості та здатності працювати з неструктурованими або поліморфними даними. Наприклад, мобільний додаток для соціального медіа може використовувати Firebase Realtime Database для зберігання потокових даних, таких як повідомлення, коментарі та оцінки, які постійно оновлюються та синхронізуються між користувачами.

Гібридні системи баз даних поєднують в собі переваги реляційних та NoSQL підходів, щоб забезпечити оптимальний баланс між структурою та гнучкістю управління даними. Наприклад, мобільний додаток для електронної комерції може використовувати гібридну базу даних для зберігання інформації про товари та замовлення, поєднуючи структуровані дані товарів із динамічною інформацією про поточний статус замовлень.

У контексті розробки додатків Flutter використовуються як локальні, так і віддалені бази даних. Локальні бази даних, прикладом яких є SQLite, знаходяться на пристрої користувача та добре підходять для управління невеликими та середніми наборами даних або для полегшення офлайн-функціональності. З іншого боку, віддалені бази даних, такі як Firebase, використовують хмарні рішення для зберігання даних, пропонуючи такі переваги, як синхронізація даних на різних пристроях та оновлення в режимі реального часу.

Щоб оптимізувати обробку даних у додатках Flutter, розробники можуть використовувати можливості вбудованих локальних сховищ даних, таких як SQLite, або обрати сторонні пакети, такі як Hive. Ці інструменти дозволяють розробникам створювати структуровані бази даних, які зберігають дані навіть тоді, коли додаток закрито або коли пристрій працює в автономному режимі. Крім того, підтримка Flutter прямого доступу до бази даних за допомогою SQL-запитів або використання бібліотек об'єктно-реляційного відображення (ORM), таких як Moor або Hive, спрощує управління даними, надаючи інтуїтивно зрозумілі абстракції, які безперешкодно відображають операції з базою даних на об'єкти Dart.

Забезпечення безпеки даних має першорядне значення, особливо коли розробник має справу з конфіденційною інформацією. Додатки Flutter можуть використовувати методи шифрування для захисту даних, що зберігаються в базі даних, таким чином захищаючи приватність користувачів та зберігаючи конфіденційність критично важливої інформації.

При розробці бази даних для програми Flutter необхідно ретельно моделювати дані, беручи до уваги взаємозв'язки та взаємодію між різними об'єктами даних. Цей продуманий процес проєктування оптимізує ефективність зберігання та швидкість пошуку, що в кінцевому підсумку підвищує загальну продуктивність додатку. Крім того, стратегії оптимізації продуктивності, такі як індексування, кешування та розумний вибір даних, відіграють вирішальну роль у максимальному використанні обмежених ресурсів пристрою, що призводить до швидкої та плавної роботи користувача. Нарешті, включення офлайн-функціональності в мобільні додатки стає все більш важливим. Надаючи користувачам доступ до даних та взаємодію з ними навіть за відсутності інтернет-з'єднання, додатки можуть забезпечити безперебійне надання послуг. Механізми синхронізації можуть бути впроваджені для безперешкодного оновлення віддаленої бази даних після відновлення зв'язку, забезпечуючи узгодженість даних та задоволеність користувачів.

3.2.6 Інтерфейс прикладного програмування мобільних додатків

Сучасні мобільні додатки, особливо ті, що розробляються в рамках Flutter, значною мірою покладаються на інтерфейси прикладного програмування (API) для встановлення каналів зв'язку із зовнішніми сервісами та джерелами даних. API слугують посередниками, що полегшують взаємодію між клієнтським додатком та функціями на стороні сервера, дозволяючи обмінюватися даними, використовувати сервіси і, зрештою, збагачувати можливості додатку.

У розробці мобільних додатків переважають кілька архітектурних стилів API. RESTful API характеризуються дотриманням архітектурного стилю Representational State Transfer, використовує методи HTTP для маніпуляцій з ресурсами та широко підтримуються в екосистемі Flutter за допомогою таких пакетів, як "http". В якості альтернативи, API GraphQL пропонують більш гнучкий та ефективний підхід до отримання даних, а бібліотеки на кшталт graphql_flutter надають необхідні інструменти для інтеграції. Вибір архітектури API часто залежить від таких факторів, як складність вимог до даних, інфраструктура на стороні сервера та бажані характеристики продуктивності.

Інтеграція API у Flutter-додатки вимагає врахування низки міркувань. Асинхронна природа викликів API вимагає надійних рішень для управління станом (таких як BLoC або Provider, щоб ефективно обробляти потоки даних) і оновлення інтерфейсу користувача. Крім того, механізми серіалізації та десеріалізації даних, які часто полегшуються пакетами на кшталт json_serializable, мають вирішальне значення для перетворення між об'єктами Dart та форматом JSON, який зазвичай використовується у спілкуванні з API.

Крім того, міркування безпеки є першочерговими при інтеграції API. Механізми автентифікації, включаючи ключі API, протоколи OAuth або інші форми авторизації, забезпечують безпечний доступ до ресурсів та запобігають несанкціонованому маніпулюванню даними. OAuth (Open

Authorization) – це протокол авторизації, який дозволяє стороннім сервісам отримувати обмежений доступ до ресурсів користувача без необхідності надавати їм свій пароль. Наприклад, мобільний додаток може використовувати OAuth для авторизації користувача через його обліковий запис у популярних соціальних мережах, таких як Facebook або Google. Після успішної авторизації OAuth надає додатку токен доступу, який може бути використаний для отримання обмеженого доступу до ресурсів користувача на сервері (рисунок 3.3). Цей процес дозволяє забезпечити безпеку та зручність для користувачів, а також дозволяє додаткам використовувати функціональність інших сервісів без необхідності надавати свої дані для авторизації.

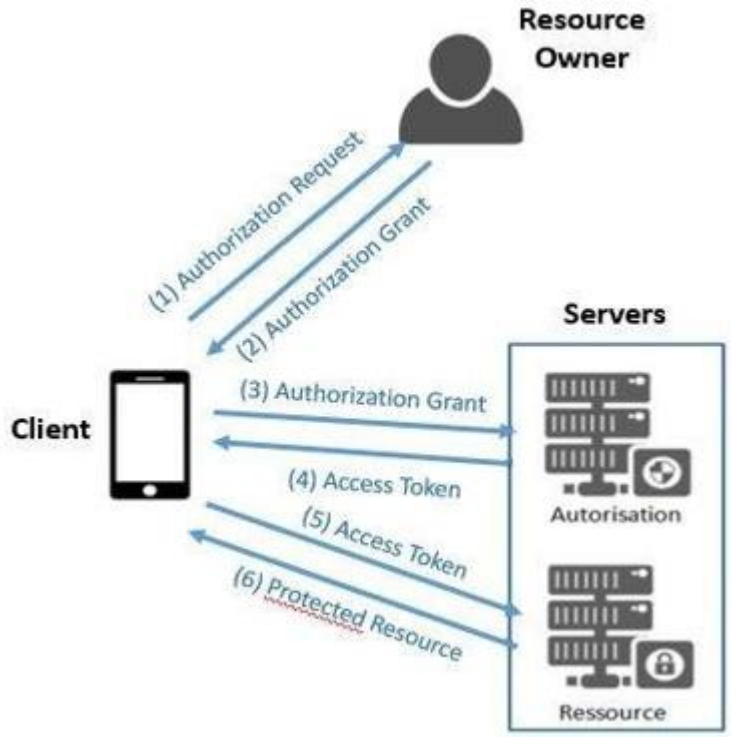

Рисунок 3.3 – Принцип дії OAuth

Надійні стратегії обробки помилок також мають важливе значення для вирішення потенційних проблем мережі, помилок API та неочікуваних відповідей, забезпечуючи стабільність роботи додатків та зручність для користувачів.

Крім цього, важливо регулярно оновлювати та аудитувати механізми

безпеки для виявлення потенційних вразливостей та забезпечення високого рівня захищеності додатків у змінних умовах кібербезпеки.

Основна мета API полягає у створенні стандартизованого та зручного інтерфейсу, що дозволяє різним частинам додатку взаємодіяти одна з одною без необхідності знання деталей реалізації інших компонентів. Це спрощує розробку та підтримку додатків, дозволяючи різним командам працювати над окремими частинами без взаємозалежності.

3.2.7 Хмарні сервіси при розробці мобільних додатків

Хмарні сервіси стали незамінними компонентами в сучасній розробці мобільних додатків, особливо в контексті фреймворку Flutter. Ці сервіси пропонують широкий спектр функціональних можливостей, включаючи зберігання даних, управління мультимедійним контентом та ресурсоємні можливості обробки. Інтегруючи хмарні сервіси в додатки Flutter, розробники можуть значно підвищити масштабованість, надійність та доступність, що в кінцевому підсумку сприяє кращому досвіду користувача.

Одна з головних переваг використання хмарних сервісів полягає у відокремленні зберігання даних від фізичних обмежень мобільних пристроїв. Таке відокремлення полегшує безперешкодне резервне копіювання, відновлення та синхронізацію даних на різних пристроях користувача, забезпечуючи їхню цілісність та доступність. Крім того, хмарні сервіси мають властиву їм масштабованість, що дозволяє додаткам динамічно адаптуватися до коливань обсягу даних без шкоди для продуктивності. У міру зростання вимог до програми хмарні ресурси можуть бути легко надані для задоволення зростаючого робочого навантаження, забезпечуючи ефективну роботу та швидкість реагування.

Крім того, хмарні сервіси пропонують безліч інструментів і технологій для розширеного аналізу та обробки даних. Сюди входить інтеграція механізмів штучного інтелекту (ШІ) та машинного навчання (МН), що дозволяє розробникам розширювати функціональність додатків за рахунок

інтелектуальних функцій. Наприклад, чат-боти на основі ШІ можуть надавати персоналізовану підтримку клієнтам, а алгоритми машинного навчання можуть аналізувати поведінку користувачів і надавати їм персоналізовані рекомендації та контент. Такі можливості не лише покращують досвід користувача, але й надають цінну інформацію для оптимізації додатків та бізнес-аналітики.

В екосистемі Flutter численні плагіни та пакети сприяють безперешкодній інтеграції з популярними постачальниками хмарних послуг. Наприклад, плагін firebase_core забезпечує основу для підключення додатків Flutter до різних сервісів Firebase, включаючи автентифікацію, хмарне сховище та бази даних в режимі реального часу. Аналогічно, пакет amplify_flutter забезпечує інтеграцію з AWS Amplify, пропонуючи комплексне рішення BaaS (Backend-as-a-Service). Ці інструменти абстрагуються від складнощів взаємодії з хмарними сервісами, дозволяючи розробникам зосередитися на створенні основних функцій додатків.

Іншими важливими інтеграціями є Microsoft Azure та Google Cloud Platform, які пропонують широкі хмарні сервіси та підтримку для додатків Flutter. Ці інтеграції дозволяють розробникам використовувати сильні сторони різних хмарних провайдерів, забезпечуючи гнучкість і вибір у виборі найбільш підходящих сервісів для своїх додатків.

Наявність численних плагінів та пакетів в екосистемі Flutter ще більше спрощує цю інтеграцію, дозволяючи розробникам ефективно інтегрувати хмарні сервіси в свої додатки. Оскільки хмарні технології продовжують розвиватися, їхня роль у розробці мобільних додатків, безсумнівно, стане ще більш важливою, стимулюючи інновації та відкриваючи нові можливості як для розробників, так і для користувачів.

3.2.8 Використання механізмів безпеки в мобільних додатках

SEC (Security) – це механізми безпеки, що включають в себе різноманітні заходи для захисту мобільного додатку та даних користувачів

від потенційних загроз та атак. У моделі мобільного додатку на основі фреймворку Flutter, використання механізмів безпеки відіграє критичну роль у забезпеченні конфіденційності, цілісності та доступності даних.

Комплексний механізм безпеки в мобільному додатку охоплює низку аспектів, включаючи, але не обмежуючись ними: аутентифікацію користувача, авторизацію доступу до ресурсів, шифрування даних, захист від вразливостей та моніторинг подій для виявлення аномальної активності. Ці заходи безпеки допомагають забезпечити захист додатку та його користувачів відповідно до вимог безпеки та законодавства, тим самим зменшуючи ризик витоку даних, несанкціонованого доступу та інших інцидентів, пов'язаних з безпекою.

Зокрема, модель мобільного додатку може реалізовувати низку механізмів безпеки, серед яких:

- двофакторна автентифікація (2FA): надійний механізм автентифікації, який вимагає від користувачів надання двох форм верифікації, таких як пароль та біометричний ідентифікатор (наприклад, відбиток пальця або розпізнавання обличчя), щоб отримати доступ до додатку. Такий підхід значно підвищує рівень безпеки додатку, зменшуючи ризик несанкціонованого доступу;

- токени доступу: безпечний підхід до автентифікації та авторизації запитів до сервера, який гарантує, що обробляються лише легітимні запити. Маркери доступу, такі як JSON Web Tokens (JWT), можуть використовуватися для перевірки автентичності запитів та запобігання несанкціонованому доступу до конфіденційних ресурсів;

- шифрування даних: критично важливий механізм безпеки, який передбачає шифрування даних як в дорозі (під час передачі), так і в стані спокою (при зберіганні) за допомогою надійних алгоритмів шифрування, таких як Advanced Encryption Standard (AES) та Secure Sockets Layer/Transport Layer Security (SSL/TLS). Такий підхід гарантує, що дані залишаються конфіденційними та захищеними від несанкціонованого

доступу навіть у випадку несанкціонованого доступу;

- захист від вразливостей: проактивний підхід до виявлення та усунення вразливостей у коді програми, сторонніх бібліотеках та залежностях. Це передбачає проведення регулярних аудитів безпеки, тестування на проникнення та оглядів коду для виявлення потенційних слабких місць у безпеці та впровадження заходів щодо їх усунення.

В контексті розробки додатків Flutter, механізми безпеки можуть бути реалізовані за допомогою різних методів, включаючи:

- вбудовані функції безпеки Flutter: фреймворк надає ряд вбудованих функцій безпеки, таких як шифрування та безпечне зберігання, які можна використовувати для підвищення рівня безпеки програми;

- сторонні бібліотеки: PointyCastle та FlutterSecureStorage можуть бути використані для реалізації надійних механізмів безпеки, включаючи шифрування, безпечну автентифікацію та контроль доступу;

- безпечні практики кодування: впровадження безпечних методів кодування, таких як перевірка вхідних даних та обробка помилок, може допомогти запобігти поширеним вразливостям безпеки, таким як SQL-ін'єкції та міжсайтовий скриптинг (XSS).

Цей підхід дозволяє створювати додатки, які можна безпечно використовувати в різних сферах діяльності, включаючи фінанси, охорону здоров'я та електронну комерцію, де безпека та конфіденційність мають першорядне значення.

3.2.9 Механізми аналітики та моніторингу

Analytics and Monitoring (AM) є важливим компонентом розробки мобільних додатків, задачею якого є аналіз та моніторинг функціонування для збору цінної інформації про його використання та продуктивність. У контексті моделі, заснованої на фреймворку Flutter, AM відіграє важливу роль у зборі та аналізі даних, що дозволяє розробникам зрозуміти, як користувачі взаємодіють з додатком і як його функції можуть бути

оптимізовані для покращення досвіду користувача.

Механізми аналітики, що є критично важливим аспектом компоненту АМ, передбачають систематичний збір даних про різні аспекти використання додатків. Сюди входить відстеження використання функцій застосунку для виявлення сфер, до яких користувачі звертаються найактивніше та можливостей для покращення, а також моніторинг поведінки користувачів, зокрема тривалості сеансу, частоти використання та навігаційних патернів, для прийняття рішень щодо дизайну та оптимізації продуктивності додатку. Крім того, механізми аналітики включають аналіз ефективності функцій та інтерфейсів додатків у стимулюванні бажаних дій користувачів, таких як покупки, реєстрація або завершення, а також моніторинг взаємодії користувачів з рекламою, включаючи показники кліків, конверсії та отримання доходу, для оптимізації стратегій розміщення та монетизації реклами. Також механізми аналітики допомагають відстежувати ключові показники ефективності, такі як збої в роботі додатків, час завантаження та споживання заряду акумулятора, щоб визначити сфери для оптимізації та вдосконалення.

Моніторинг, ще один важливий аспект АМ, передбачає безперервний нагляд за додатком для виявлення помилок, відстеження критичних показників продуктивності та забезпечення надійності. Це включає виявлення та діагностику помилок, збоїв і винятків для швидкого вирішення та мінімізації часу простою, а також відстеження ключових показників продуктивності, таких як час відгуку, використання пам'яті та завантаження процесора, виявлення областей для оптимізації та вдосконалення. Крім того, моніторинг передбачає впровадження заходів для забезпечення стабільності та ефективності роботи програми, включаючи стратегії обходу відмов, балансування навантаження та оптимізацію мережі доставки контенту (CDN).

Основна мета АМ полягає в тому, щоб надати розробникам практичну інформацію для прийняття рішень на основі даних щодо подальшого

розвитку застосунку, тим самим покращуючи досвід користувача та сприяючи успіху бізнесу. Цей компонент має вирішальне значення для забезпечення успішної роботи та розвитку мобільних застосунків в умовах зростаючої конкуренції.

У контексті розробки застосунків з використанням фреймворку Flutter, АМ може бути реалізований за допомогою різних технік. Наприклад, розробники можуть використовувати вбудовані аналітичні можливості Flutter, такі як плагін Firebase Analytics для збору та аналізу даних про використання застосунків. Крім того, вони можуть інтегрувати сторонні аналітичні інструменти, такі як Google Analytics, Mixpanel або Adobe Analytics, щоб отримати більш глибоке розуміння використання та продуктивності додатків. Крім того, розробники можуть створювати власні аналітичні рішення, використовуючи архітектуру плагінів та API Flutter для збору та аналізу даних для конкретних програмних рішень.

3.2.10 Модуль тестування для мобільного додатку

MT (Mobile Testing) – це одна з ключових складових моделі мобільного додатку, яка відіграє важливу роль у створенні та запуску тестів для перевірки коректної роботи окремих компонентів та їх взаємодії. У контексті моделі мобільних додатків, заснованої на фреймворку Flutter, MT відіграє першорядну роль у забезпеченні якості та надійності додатку на різних етапах розробки.

Модуль тестування охоплює розробку тестових сценаріїв, автоматизацію виконання тестів та аналіз результатів. Сюди входить ряд аспектів тестування, таких як функціональне тестування для перевірки правильності реалізації бізнес-логіки, UI-тестування для перевірки правильності відображення інтерфейсу користувача, а також тестування продуктивності та безпеки. Функціональне тестування передбачає перевірку виконання функціональних вимог програми, тоді як UI-тестування фокусується на тому, щоб інтерфейс користувача програми відображався

коректно та був зручним для користувача. Тестування продуктивності, з іншого боку, включає в себе оцінку продуктивності додатку при різних навантаженнях і умовах, а тестування безпеки включає в себе виявлення вразливостей та забезпечення його безпеки.

Мобільне тестування – це складний процес, який передбачає тестування на різних платформах, включаючи iOS та Android, що вимагає врахування особливостей кожної платформи та їх взаємодії з мобільним додатком. Модуль тестування допомагає виявити та виправити помилки та проблеми, які можуть виникати на різних пристроях і платформах, забезпечуючи однаково якісну роботу додатку для всіх користувачів. Це включає тестування на різних пристроях, таких як смартфони та планшети, з різними розмірами екранів, роздільною здатністю та операційними системами.

Загальна мета МТ – переконатися, що додаток відповідає вимогам, стандартам та очікуванням користувачів, а також забезпечити його стабільність та ефективність в реальних умовах. Тестування є важливим етапом життєвого циклу розробки додатку та допомагає запобігти проблемам і забезпечити його успішний випуск на ринок.

В контексті розробки додатків на Flutter, МТ може бути реалізовано за допомогою різних технік, включаючи модульне тестування, тестування віджетів та інтеграційне тестування.

Модульне тестування передбачає тестування окремих компонентів програми, таких як віджети та функції, щоб переконатися, що вони працюють правильно. Тестування віджетів передбачає тестування інтерфейсу користувача та взаємодії з користувачем, тоді як інтеграційне тестування передбачає тестування компонентів програми та їхньої взаємодії.

4 РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ ДЛЯ ТЕСТУВАННЯ МОДЕЛІ

4.1 Структура тестового додатку

Щоб продемонструвати практичну реалізацію фреймворкової моделі, було розроблено додаток, призначений для купівлі товарів в інтернет-магазині. Цей додаток слугує тестовим майданчиком для оцінки ефективності та функціональності різних модулів, описаних у попередньому розділі. За допомогою цього додатку буде надано детальний аналіз того, як кожен модуль впливає на загальну продуктивність та надійність мобільного додатку, створеного з використанням фреймворку Flutter (рисунок 4.1).

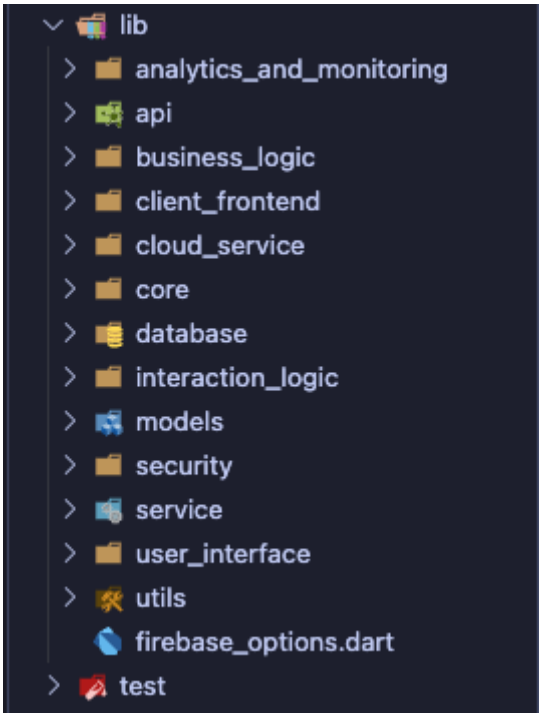

Рисунок 4.1 – Структура директорій проєкту тестового застосунку

Директорія lib містить у собі такі під-директорії:

- user_interface – директорія для User Interface компонентів;

- client_frontend – директорія для Client Frontend логіки, реалізація специфічних для платформи (iOS, Android) інтерфейсів та логіки;

- interaction_logic – директорія для Interaction Logic;

- business_logic – директорія для Business Logic;

- database – директорія для Database взаємодії;

- cloud_services – директорія для Cloud Services;

- api – директорія відповідає за логіку API запитів;

- analytics_and_monitoring – директорія для Analytics and Monitoring;

- core – містить налаштування навігації та налаштування залежностей.

Також в кореневій директорії проєкту міститься директорія test для створення та проведення тестів для перевірки роботи компонентів додатку.

4.2 Модуль інтерфейсу користувача: SignInPage віджет

Модуль інтерфейсу користувача у Flutter визначає візуальні та інтерактивні елементи, які складають екран входу в додаток. На лістингу 4.1 продемонстровано використання адаптивного дизайну та зручної для користувача взаємодії з використанням різних макетів для малих та великих екранів. Віджет SignInPage слугує основною точкою входу для автентифікації користувача. Він адаптує свій макет відповідно до розміру екрану, щоб забезпечити оптимальний досвід користувача (рисунки 4.2, 4.3).

Лістинг 4.1 – Віджет SignInPage

```
class SignInPage extends StatelessWidget {
        const SignInPage({super.key});
        static const path = '/sign_in';

        @override
        Widget build(BuildContext context) {
          final bool isSmallScreen = MediaQuery.of(context).size.width < 600;

          return Scaffold(
            body: Center(
              child: isSmallScreen
                  ? const _SmallScreenLayout()
                  : const _LargeScreenLayout(),
            ),
          );
        }
      }
```

Для невеликих екранів віджет _SmallScreenLayout організовує компоненти в одну колонку, щоб усі елементи залишалися легко доступними та видимими.

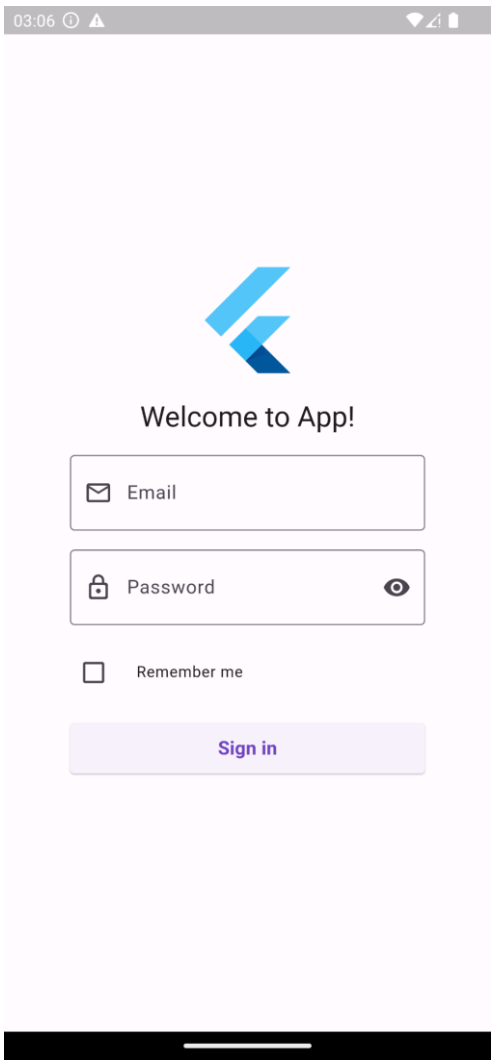

Рисунок 4.2 – Вигляд авторизації на малому екрані

Віджет _Logo (лістинг 4.2) відображає логотип програми та налаштовує його розмір і стиль відповідно до розміру екрану за допомогою MediaQuery від Flutter.

Лістинг 4.2 – Віджет _Logo віджет

```
class _Logo extends StatelessWidget {
        const _Logo();
        @override
        Widget build(BuildContext context) {
          final bool isSmallScreen = MediaQuery.of(context).size.width < 600;
          return Column(
            mainAxisSize: MainAxisSize.min,
            children: [
              FlutterLogo(size: isSmallScreen ? 100 : 200),
              Padding(
                padding: const EdgeInsets.all(16.0),
                child: Text(
                  'Welcome to App!',
                  textAlign: TextAlign.center,
                  style: isSmallScreen
```

```
                ? Theme.of(context).textTheme.headlineSmall:
      Theme.of(context).textTheme.headlineMedium?.copyWith(color:
      Colors.black),
            ),        ),
        ],    );
    }}
```

Віджет _FormContent (лістинг 4.3) містить поля для введення адреси електронної пошти та пароля, прапорець "Запам'ятати мене" та кнопку входу. Він керує станом форми та взаємодією з віджетом Flutter's Form, а також обробляє зміни стану для видимості пароля та функції "запам'ятати мене".

Лістинг 4.3 – _FormContent віджет

```
class _FormContent extends StatefulWidget {
      const _FormContent();
      @override
      State<_FormContent> createState() => __FormContentState();
    }
    class __FormContentState extends State<_FormContent> {
      bool _isPasswordVisible = false;
      bool _rememberMe = false;
      final GlobalKey<FormState> _formKey = GlobalKey<FormState>();
      @override
      Widget build(BuildContext context) {
        return Container(
          constraints: const BoxConstraints(maxWidth: 300),
          child: Form(
            key: _formKey,
            child: Column(
              mainAxisSize: MainAxisSize.min,
              mainAxisAlignment: MainAxisAlignment.center,
              children: [
                EmailField(),
                const SizedBox(height: 16),
                PasswordField(
                  isPasswordVisible: _isPasswordVisible,
                  togglePasswordVisibility: () {
                    setState(() {
                      _isPasswordVisible = !_isPasswordVisible;         });
                  },            ),
                const SizedBox(height: 16),
                RememberMeCheckbox(
                  value: _rememberMe,
                  onChanged: (value) {
                    if (value == null) return;
                    setState(() {
                      _rememberMe = value;                });
                  },            ),
                const SizedBox(height: 16),
                SignInButton(
                  formKey: _formKey,
                  onPressed: () {
                    try {
                      router.go(NewsFeedPage.path);
                    } catch (e) {
                      print(e.toString());
                    }  },  ),   ],   ),   ),   );   }  }
```

EmailField – це поле користувача TextFormField для введення електронної пошти з перевіркою.

PasswordField – це поле користувача TextFormField для введення пароля з перевіркою та перемикачем видимості.

RememberMeCheckbox – це спеціальний віджет CheckboxListTile для запам'ятовування облікових даних користувача.

SignInButton – це віджет ElevatedButton для надсилання форми, керування валідацією форми та навігацією.

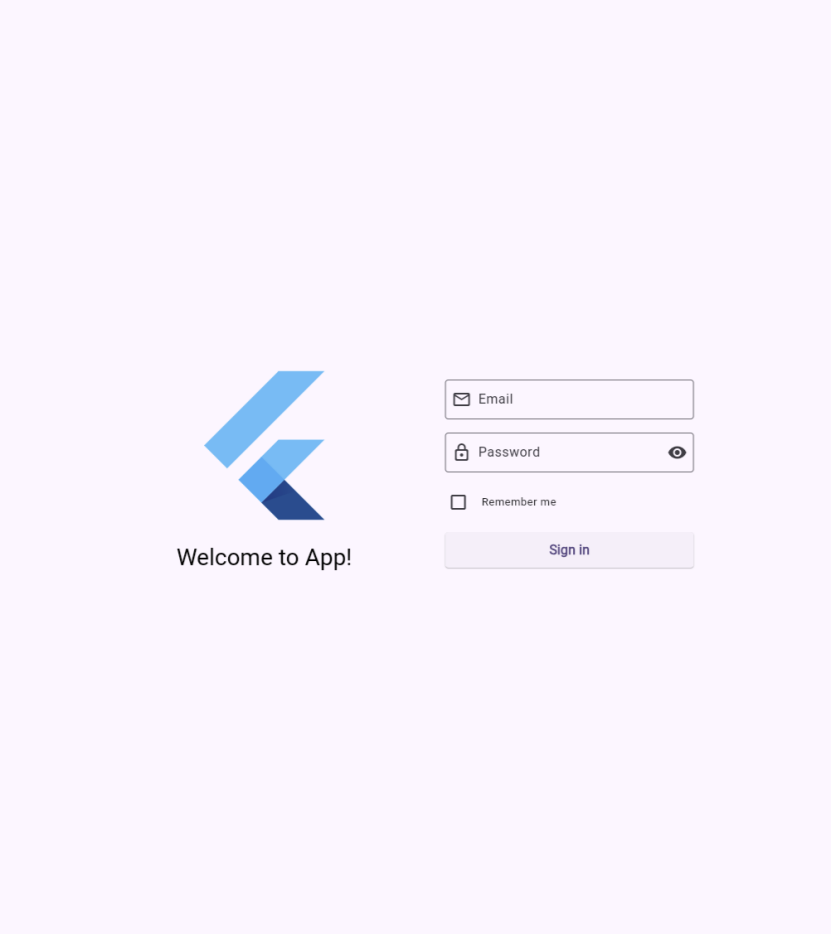

Рисунок 4.3 – Вигляд авторизації на великому екрані

4.3 Визначення типу мережевого з'єднання

У контексті розробки мобільного додатку з використанням фреймворку Flutter, модуль Client Frontend (CF) відповідає за обробку специфічних особливостей платформи, які мають вирішальне значення для

функціональності додатку. Одним із прикладів є визначення типу мережного з'єднання (WiFi або стільниковий зв'язок), до якого підключений користувач. Ця функціональність використовує нативні API платформ iOS та Android, що робить її ідеальним варіантом використання для реалізації каналу методів у Flutter.

Здатність визначати, до якої мережі підключений користувач (Wi-Fi чи стільникової) є важливою для оптимізації поведінки та продуктивності програми. Наприклад, певні функції програми можуть вимагати високошвидкісного підключення до Інтернету через WiFi, тоді як інші можуть бути розроблені для ефективної роботи через стільникові мережі. Така диференціація може значно покращити досвід користувача та забезпечити оптимальну роботу програми.

Інкапсуляція специфічних для платформи функцій у модулі клієнтського інтерфейсу гарантує, що основна логіка програми залишається незалежною від платформи. Таке розділення завдань є ключовим принципом програмної інженерії, що підвищує зручність супроводу кодової бази.

Підхід платформних каналів у Flutter полегшує безперешкодну інтеграцію специфічних для платформи функцій, розширюючи можливості додатку, використовуючи при цьому нативні API (лістинг 4.4).

Лістинг 4.4 – Приклад використання платформних каналів у Flutter

```
class PlatformChannel {
        static Future<String> getNetworkStatus(MethodChannel channel) async {
          try {
            final String result = await
      channel.invokeMethod('getNetworkStatus');
            return result;
          } on PlatformException catch (e) {
            return "Failed to get network status: '${e.message}'.";
          }  }}
```

Коли користувач створює канал у Dart, йому потрібно зробити те саме у Kotlin (лістинг 4.5) та Swift (лістинг 4.6).

Лістинг 4.5 – Налаштування каналів платформи в Kotlin (Android)

```
class MainActivity: FlutterActivity() {
          override fun configureFlutterEngine(flutterEngine: FlutterEngine) {
```

```
            super.configureFlutterEngine(flutterEngine)
            val channelHandler = ChannelHandler(this, flutterEngine)
            channelHandler.setMethodCallHandler()
    }   }
```

Лістинг 4.6 – Налаштування каналів платформи в Swift (iOS)

```
@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?
  ) -> Bool {
      GeneratedPluginRegistrant.register(with: self)
      let controller: FlutterViewController = window?.rootViewController as!
FlutterViewController
      let channelHandler = ChannelHandler(messenger:
controller.binaryMessenger)
      channelHandler.setMethodCallHandler()
      return super.application(application, didFinishLaunchingWithOptions:
launchOptions)
  }   }
```

Тепер слід переходити до самої суті отримання даних з нативу назад в Dart та Flutter (лістинги 4.7, 4.8).

Лістинг 4.7 – Обробка викликів нативних методів у Flutter за допомогою каналів платформи (iOS)

```
class ChannelHandler {
    private let methodChannel: FlutterMethodChannel
    init(messenger: FlutterBinaryMessenger) {
        methodChannel = FlutterMethodChannel(name: "channel-name",
binaryMessenger: messenger)    }
    func setMethodCallHandler() {
        methodChannel.setMethodCallHandler(handle)    }
    private func handle(call: FlutterMethodCall, result: @escaping
FlutterResult) {
        switch call.method {
        case "getNetworkStatus":
            getNetworkStatus { status in
                result(status)            }
        default:
            result(FlutterMethodNotImplemented)        }    }
    private func getNetworkStatus(completion: @escaping (String) -> Void) {
        let monitor = NWPathMonitor()
        let queue = DispatchQueue(label: "NetworkMonitor")
        monitor.pathUpdateHandler = { path in
            var networkStatus: String
            if path.status == .satisfied {
                if path.usesInterfaceType(.wifi) {
                    networkStatus = "Connected to Wi-Fi"
                } else if path.usesInterfaceType(.cellular) {
                    networkStatus = "Connected to Cellular"
                } else if path.usesInterfaceType(.wiredEthernet) {
                    networkStatus = "Connected to Wired Ethernet"
                } else {
                    networkStatus = "Connected to Unknown"        }
```

```
            } else {
                networkStatus = "No connection"   }
            completion(networkStatus)
            monitor.cancel()        }
        monitor.start(queue: queue)    }   }
```

Лістинг 4.8 – Обробка викликів нативних методів у Flutter за допомогою каналів платформи (Android):

```
class ChannelHandler(private val context: Context, flutterEngine:
FlutterEngine) {
    private val methodChannel =
MethodChannel(flutterEngine.dartExecutor.binaryMessenger, "channel-name")
    fun setMethodCallHandler() {
        methodChannel.setMethodCallHandler { call, result ->
            when (call.method) {
                "getNetworkStatus" -> {
                    val networkStatus = getNetworkStatus()
                    result.success(networkStatus)        }
                else -> result.notImplemented()       }  }  }
    private fun getNetworkStatus(): String {
        val connectivityManager =
context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
        val network = connectivityManager.activeNetwork
        val capabilities =
connectivityManager.getNetworkCapabilities(network)
        return when {
            capabilities == null -> "No connection"
            capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI) ->
"Connected to Wi-Fi"
            capabilities.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR)
-> "Connected to Cellular"
            else -> "Unknown connection"   }   }   }
```

Статус мережі, що повертається , залежить від наступних умов:

- якщо статус мережного шляху (path.status) визначено як .satisfied, що вказує на активне мережеве з'єднання:

1) якщо мережа використовує Wi-Fi, повертається рядок стану "Connected to Wi-Fi";

2) якщо мережа використовує стільниковий зв'язок, повертається рядок стану "Connected to Cellular";

3) якщо мережа використовує дротовий Ethernet, повертається рядок стану "Connected to Wired Ethernet";

4) якщо конкретний тип інтерфейсу не може бути визначено, повертається рядок стану "Connected to Unknown";

- якщо стан мережного шляху вказує на відсутність активного з'єднання, повертається рядок стану "No connection".

Такий структурований підхід гарантує, що додаток Flutter отримує точні та своєчасні оновлення щодо статусу мережного підключення пристрою, що дозволяє швидко адаптувати поведінку інтерфейсу користувача або мережево-залежні операції в додатку.

4.4 Оптимізація інтерфейсу користувача через логіку валідації

Відокремлюючи логіку взаємодії (IL) від бізнес-логіки (BL) комплексна модель сприяє чіткому розмежуванню завдань. Кожен модуль відповідає за певний аспект функціональності програми, що спрощує розробку, налагодження та обслуговування.

Логіка валідації в модулі логіки взаємодії гарантує, що вхідні дані користувача відповідають попередньо визначеним правилам, перш ніж вони будуть оброблені додатком. Ця практика не тільки покращує цілісність даних та зручність роботи користувача, але й допомагає підтримувати надійність програми. Відокремивши логіку валідації від бізнес-логіки та компонентів інтерфейсу користувача, можливо досягти модульного дизайну, який легше підтримувати та розширювати.

Щоб проілюструвати реалізацію логіки валідації, розглянемо поширений сценарій, в якому потрібно валідувати форму входу з полями для електронної пошти та пароля. Клас FormValidators (лістинг 4.9) інкапсулює різні функції перевірки. Кожна функція отримує значення поля і контекст збірки, повертаючи повідомлення про помилку, якщо перевірка не пройшла, або null, якщо пройшла успішно.

Лістинг 4.9 – Приклад логіки валідації

```
class FormValidators {
  static String? validateEmail(String? value) {
    if (value == null || value.isEmpty) {
      return 'Please enter some text';    }
    final bool emailValid = RegExp(
      r"^[a-zA-Z0-9.a-zA-Z0-9.!#$%&'*+-/=?^_`{|}~]+@[a-zA-Z0-9]+\.[a-zA-Z]+",
    ).hasMatch(value);
    if (!emailValid) {
      return 'Please enter a valid email';    }
    return null;  }
  static String? validatePassword(String? value) {
```

```
    if (value == null || value.isEmpty) {
      return 'Please enter some text';    }
    if (value.length < 6) {
      return 'Password must be at least 6 characters';    }
    return null;  }  }
```

Підключення функцій валідації до відповідних полів наведено у лістингу 4.10

Лістинг 4.10 – Приклад використання валідації

```
class PasswordField extends TextFormField {
  PasswordField({
    super.key,
    required bool isPasswordVisible,
    required VoidCallback togglePasswordVisibility,
  }) : super(
          validator: (value) => FormValidators.validatePassword(value),
          obscureText: !isPasswordVisible,
          decoration: InputDecoration(
          border: const OutlineInputBorder(),
            suffixIcon: IconButton(
              icon: Icon(
                isPasswordVisible ? Icons.visibility_off : Icons.visibility,),
              onPressed: togglePasswordVisibility,         ),     ),    );  }
class EmailField extends TextFormField {
  EmailField({super.key})
      : super(
          validator: (value) => FormValidators.validateEmail(value),
          decoration: InputDecoration(
            prefixIcon: const Icon(Icons.email_outlined),
            border: const OutlineInputBorder(),     ),    );   }
```

Клас FormValidators забезпечує структурований підхід до перевірки полів форм у додатках. Завдяки інкапсуляції логіки валідації в цьому класі, код стає більш модульним та легшим в обслуговуванні. Кожна функція валідації перевіряє певні критерії, повертаючи відповідні повідомлення про помилки, якщо вхідні дані не відповідають вимогам.

4.5 Використання бізнес логіки

Основна відповідальність модуля бізнес-логіки полягає в обробці даних та виконанні обчислювальних завдань, які підтримують основні функції програми. Сюди входить управління потоком даних між інтерфейсом користувача та базою даних, реалізація алгоритмів маніпулювання даними, та забезпечення дотримання бізнес-правил та робочих процесів.

Класи BaseCubit та BaseBloc (лістинг 4.11) – це абстрактні обгортки навколо базових класів, що використовуються в архітектурі управління станом у Flutter-додатках, які розширюють базовий клас. Він включає в себе можливість використання міксинів, зокрема DiProvider, що спрощує інтеграцію залежностей та інших функціональних можливостей.

Міксін DiProvider ілюструє застосування принципів ін'єкції залежностей (DI) в контексті програми Flutter, використовуючи пакет GetIt для реєстрації та вирішення сервісів. Ін'єкція залежностей – це патерн проєктування програмного забезпечення, який полегшує вільний зв'язок шляхом екстерналізації залежностей класу або модуля, тим самим сприяючи багаторазовому використанню, тестуванню та підтримці.

Лістинг 4.11 – Класи BaseCubit та BaseBloc

```
abstract class BaseCubit<State> extends Cubit<State> with DiProvider {
  BaseCubit(State initialState) : super(initialState);
}

abstract class BaseBloc<Event, State> extends Bloc<Event, State> with
DiProvider {
  BaseBloc(State initialState) : super(initialState);
}
```

Включення DiProvider (лістинг 4.12) до класів BaseCubit та BaseBloc відображає високий рівень абстракції та повторного використання коду в архітектурі програми. Це сприяє чіткому розділенню обов'язків та забезпечує прозорість у впровадженні залежностей.

Лістинг 4.12 – Міксін ін'єкції залежностей

```
mixin DiProvider {
  Future<void> initDiProvider() async {
    GetIt.I.registerSingletonAsync<LocalStorageService>(
      () async => LocalStorageService().init(),    );
    GetIt.I.registerSingletonAsync<AuthService>(
      () async => AuthServiceImpl(),    );
    GetIt.I.registerSingletonAsync<UserService>(
      () async => UserServiceImpl(),    );
    GetIt.I.registerSingletonWithDependencies<UserRepository>(
      () => UserRepository(
        GetIt.instance<LocalStorageService>(),
        GetIt.instance<AuthService>(),
        GetIt.instance<UserService>(),      ),
      dependsOn: [LocalStorageService, AuthService, UserService],    );
```

```
    await GetIt.instance.allReady();  }
  UserRepository get userRepository => GetIt.I<UserRepository>();}
```

Клас LoginScreenCubit (лістинг 4.13) ілюструє практичне застосування патерну BLoC для керування станом та поведінкою екрану входу у додатку Flutter.

Лістинг 4.13 – Приклад використання патерну BLoC

```
class LoginScreenCubit extends BaseCubit<LoginScreenState> {
  LoginScreenCubit() : super(const LoginScreenInitial());

  Future<void> login(String email, String password) async {
    emit(const LoginScreenLoading());
    final error = await userRepository.login(email, password);
    if (error != null) {
      emit(LoginScreenError(error));
    } else {
      router.go(userRepository.readUserData()!.homeScreenPath);
    }
  }
}
```

4.6 Використання сервісу баз даних

Модуль бази даних відповідає за зберігання всіх даних програми, забезпечуючи їх швидкий та точний пошук. БД гарантує, що дані структуровані таким чином, щоб оптимізувати продуктивність та підтримувати вимоги програми.

БД застосовує обмеження та взаємозв'язки між даними для підтримки цілісності та запобігання невідповідностей. Модуль бази даних повинен бути здатний масштабуватися разом з програмою по мірі зростання обсягу даних та кількості користувачів.

Абстрактний клас визначатиме методи, які повинна надавати будь-яка реалізація бази даних. Це включає базові операції CRUD (створення, читання, оновлення, видалення) та будь-які інші необхідні методи (лістинг 4.14).

Лістинг 4.14 – Приклад абстрактного інтерфейсу сервісу бази даних

```
abstract class DatabaseService {
Future<void> openDatabase(String dbName);
Future<void> closeDatabase();
```

```
Future<void> insert(String table, Map<String, dynamic> data);
Future<Map<String, dynamic>> get(String table, String id);
Future<List<Map<String, dynamic>>> getAll(String table);
Future<void> update(String table, String id, Map<String, dynamic> data);
Future<void> delete(String table, String id); }
```

У лістингу 4.15 наведено приклад реалізації бази даних для sqflite (популярний пакет SQLite для Flutter).

Лістинг 4.15 – Приклад реалізації інтерфейсу DatabaseService за допомогою пакету sqflite

```
class SQLiteDatabaseServiceImpl implements DatabaseService {
  SQLiteDatabaseServiceImpl(this._database);
  final Database _database;
  @override
  Future<void> insert(String table, JsonMap data) async {
    await _database.insert(table, data);  }
  @override
  Future<JsonMap?> query(String table, String id) async {
    final results = await _database.query(table, where: 'id = ?', whereArgs:
[id]);
    return results.isNotEmpty ? results.first : null;  }
  @override
  Future<List<JsonMap>> queryAll(String table) async {
    return await _database.query(table);  }
  @override
  Future<void> update(String table, String id, JsonMap data) async {
    await _database.update(table, data, where: 'id = ?', whereArgs: [id]);  }
  @override
  Future<void> delete(String table, String id) async {
    await _database.delete(table, where: 'id = ?', whereArgs: [id]);  } }
```

Реалізація моделі бази даних з використанням SQFLite у Flutter передбачає створення сервісного рівня, який абстрагує основні операції з базою даних. Такий підхід покращує модульність, тестованість та масштабованість рівня доступу до даних додатку.

4.7 Використання хмарних сервісів

Модуль Cloud Services (CS) інкапсулює різні функції, що надаються платформами хмарних обчислень, такі як зберігання, обробка та синхронізація даних. Цей модуль дозволяє додатку взаємодіяти з хмарними API та сервісами, забезпечуючи безпечне зберігання даних, ефективний доступ до них та синхронізацію між різними пристроями в режимі реального

часу. Інтеграція хмарних сервісів в архітектуру мобільного додатку підвищує його здатність обробляти великі масиви даних, виконувати складні обчислення та забезпечувати безперебійну роботу користувачів.

Розглянемо базовий приклад модуля хмарних сервісів (лістинг 4.16) з використанням Firebase, популярної платформи backend-as-a-service (BaaS), що фокусується на автентифікації та операціях з базами даних в режимі реального часу.

Лістинг 4.16 – Приклад абстрактного інтерфейсу хмарного сервісу

```
abstract class CloudService {
  Future<void> signInWithEmailAndPassword(String email, String password);
  Future<void> signOut();
  Stream<User?> get authStateChanges;
  Future<void> setData(String path, Map<String, dynamic> data);
  Future<Map<String, dynamic>?> getData(String path);
  Stream<Map<String, dynamic>> streamData(String path);
  Future<void> updateData(String path, Map<String, dynamic> data);
  Future<void> deleteData(String path);   }
```

Для реалізації інтерфейсу CloudService з використанням Firebase в якості бекенда, створено клас FirebaseCloudService (лістинг 4.17), який інтегрує функції Firebase Authentication та Firebase Realtime Database. Firebase пропонує надійні функції для автентифікації, синхронізації даних у реальному часі та хмарного зберігання, що робить її ідеальною для багатьох додатків Flutter.

Лістинг 4.17 – Приклад реалізації інтерфейсу CloudService використовуючи Firebase

```
class FirebaseCloudService implements CloudService {
  final FirebaseAuth _auth = FirebaseAuth.instance;
        final DatabaseReference _database =
      FirebaseDatabase.instance.reference();
  @override
  Future<void> signInWithEmailAndPassword(String email, String password)
async {
    try {
      await _auth.signInWithEmailAndPassword(email: email, password:
password);
    } catch (e) {
      throw e; // Handle sign-in errors appropriately    }  }
  @override
  Future<void> signOut() async {
    await _auth.signOut();  }
  @override
```

```
  Stream<User?> get authStateChanges => _auth.authStateChanges();
  @override
  Future<void> setData(String path, Map<String, dynamic> data) async {
    await _database.child(path).set(data);  }
  @override
  Future<Map<String, dynamic>?> getData(String path) async {
    DataSnapshot snapshot = await _database.child(path).once();
    return snapshot.value;  }
  @override
  Stream<Map<String, dynamic>> streamData(String path) {
    return _database.child(path).onValue.map((event) => event.snapshot.value
?? {});  }
  @override
  Future<void> updateData(String path, Map<String, dynamic> data) async {
    await _database.child(path).update(data);  }
  @override
  Future<void> deleteData(String path) async {
    await _database.child(path).remove();  }  }
```

Інтеграція Firebase у додаток Flutter через FirebaseCloudService забезпечує надійну автентифікацію та управління базами даних у режимі реального часу. Цей модульний підхід забезпечує масштабованість та зручність обслуговування, дозволяючи додаткам безпечно автентифікувати користувачів та синхронізувати дані з мінімальними зусиллями. Інкапсулюючи функціональність Firebase в інтерфейс CloudService, розробники отримують гнучкість для адаптації внутрішніх сервісів або безперешкодного розширення функціональності.

4.8 Мережний сервіс

Розширена формула явно включає модуль API, визнаючи його особливу та вирішальну роль у полегшенні комунікації між клієнтом і сервером. Така деталізація дозволяє краще сфокусуватися та оптимізувати кожен компонент.

Відокремлюючи модуль API від інших компонентів, таких як бізнес-логіка (BL), формула сприяє чіткому розділенню завдань. Таке розділення покращує ремонтопридатність та масштабованість, оскільки кожен модуль можна розробляти, тестувати та оновлювати незалежно.

Завдяки спеціальному модулю API розробники можуть створювати цільові тести та процедури налагодження спеціально для взаємодії з API. Це підвищує надійність та стійкість комунікаційного рівня програми.

Детальна формула забезпечує більшу гнучкість у розширенні та інтеграції нових функцій. Наприклад, включенням нових кінцевих точок API або зміною існуючих можна керувати в межах спеціального модуля API, не впливаючи на інші частини програми.

У контексті додатку, що розробляється, модуль API слугує посередником, який обробляє запити від клієнтського фронтенду та надає відповіді від бекенд-сервісів. Він гарантує, що додаток може ефективно отримувати, маніпулювати та відображати дані, зберігаючи при цьому безпеку та ефективність.

Реалізація абстрактного мережного сервісу разом з конкретною імплементацією за допомогою пакета Dio в Dart є методологічно обґрунтованою практикою з кількох причин. До них відносяться забезпечення абстракції та гнучкості, покращення можливості тестування та покращення супроводу коду.

Абстрактний інтерфейс мережного сервісу визначає основні методи HTTP, необхідні для мережної комунікації. Ця абстракція відокремлює мережну логіку від інших компонентів програми, тим самим полегшуючи її модифікацію та вдосконалення. Визначаючи мережеві операції в абстрактному інтерфейсі (лістинг 4.18), код програми, який залежить від цих команд, залишається незалежним від конкретних деталей реалізації. Ця абстракція полегшує заміну Dio на інший HTTP-клієнт, якщо це необхідно, без необхідності внесення змін до залежного коду.

Лістинг 4.18 – Приклад абстрактного інтерфейсу мережевого сервісу

```
typedef HeaderMap = Map<String, String>;
typedef JsonMap = Map<String, dynamic>;
abstract class NetworkService {
Future<Response<T>> get<T>(String url, { required HeaderMap headers, });
Future<Response<T>> post<T>(String url, { required HeaderMap headers,
required JsonMap body, });
Future<Response<T>> put<T>(String url, { required HeaderMap headers, required
JsonMap body, });
Future<Response<T>> delete<T>(String url, { required HeaderMap headers, });
Future<Response<T>> patch<T>(String url, { required HeaderMap headers,
required JsonMap body, });
Future<Response<T>> options<T>(String url, { required HeaderMap headers, });}
```

Пакет Dio – це потужний HTTP-клієнт для Dart, відомий своїм багатим набором функцій, включаючи перехоплювачі, глобальну конфігурацію, FormData, скасування запитів, завантаження файлів та багато іншого (лістинг 4.19). Гнучкість та простота використання роблять його чудовим вибором для реалізації мережевих сервісів.

Лістинг 4.19 – Приклад реалізації інтерфейсу NetworkService за допомогою пакету Dio

```
import 'package:dio/dio.dart';
import 'network_service.dart';
class DioNetworkServiceImpl implements NetworkService {
  DioNetworkServiceImpl(this._dio);
  final Dio _dio;
  @override
  Future<Response<T>> delete<T>(String url, {
    required HeaderMap headers,  }) =>
      _dio.delete<T>(url,
        options: Options(headers: headers),      );
  @override
  Future<Response<T>> get<T>(String url, {
    required HeaderMap headers,  }) =>
      _dio.get<T>(url,
        options: Options(headers: headers),      );
  @override
  Future<Response<T>> post<T>(String url, {
    required HeaderMap headers,required JsonMap body,  }) =>
      _dio.post<T>(url,
        data: body,
        options: Options(headers: headers),      );
  @override
  Future<Response<T>> put<T>(String url, {
    required HeaderMap headers, required JsonMap body,  }) =>
      _dio.put<T>(url,
        data: body,
        options: Options(headers: headers),      );
  @override
  Future<Response<T>> patch<T>(String url, {
    required HeaderMap headers,required JsonMap body,  }) =>
      _dio.patch<T>(url,
        data: body,
        options: Options(headers: headers),      );
  @override
  Future<Response<T>> options<T>(String url, {
    required HeaderMap headers,  }) =>
      _dio.options<T>(url,
        options: Options(headers: headers),      );}
```

Під час модульного тестування імітаційна реалізація NetworkService може імітувати відповіді мережі, що дозволяє ізольовано тестувати логіку роботи програми. Це покращує покриття тестів та надійність, усуваючи залежність від доступності мережі.

Чітке розділення між мережним рівнем та іншою логікою додатку дотримується принципу єдиної відповідальності, тим самим покращуючи підтримку та зрозумілість коду. Обов'язки кожного компонента чітко визначені, що зменшує ймовірність помилок та спрощує налагодження та розширення.

4.9 Сервіси аналітики та моніторингу

Реалізація модуля аналітики та моніторингу передбачає інтеграцію сторонніх сервісів, таких як Google Analytics, Firebase Analytics та Sentry. Ці інструменти надають широкий функціонал для відстеження, аналізу та реагування на події в додатку. Наступна реалізація демонструє, як ці сервіси можуть бути інтегровані в додаток Flutter.

Модуль аналітики та моніторингу (лістинг 4.20) використовує абстрактний клас сервісу для визначення структури та методів інтеграції різних постачальників аналітики та моніторингу в додаток.

Лістинг 4.20 – Приклад абстрактного інтерфейсу моніторингового сервісу

```
abstract class MonitoringService {
  Future<void> initialize();
  Future<void> logEvent(String eventName, Map<String, dynamic> parameters);
  Future<void> logLogin(String userId);
  Future<void> setUserProperties(String userId, String userRole);
  Future<void> captureException(dynamic exception, dynamic stackTrace);
  Future<void> captureMessage(String message);}
```

Реалізація абстрактного класу сервісу використовує Firebase Analytics для відстеження поведінки користувачів та Sentry для моніторингу помилок у реальному часі (лістинг 4.21). Firebase Analytics відстежує взаємодію користувачів та надає детальні звіти про їхню активність та поведінку.

Лістинг 4.21 – Приклад моніторингового сервісу використовуючи Firebase Analytics

```
class FirebaseMonitoringService implements MonitoringService {
  final FirebaseAnalytics _analytics = FirebaseAnalytics();
  @override
  Future<void> initialize() async {
    // Firebase Analytics is initialized by default when the app starts.
```

```
    // No additional initialization is needed.
  }
  @override
  Future<void> logEvent(String eventName, Map<String, dynamic> parameters)
async {
    try {
      await _analytics.logEvent(name: eventName, parameters: parameters);
      print('Event logged: $eventName with parameters: $parameters');
    } catch (e) {
      print('Failed to log event: $e');    }  }
  @override
  Future<void> logLogin(String userId) async {
    try {
      await _analytics.logLogin(loginMethod: 'email');
      await _analytics.setUserId(userId);
      print('User logged in with ID: $userId');
    } catch (e) {
      print('Failed to log login: $e');    }  }
  @override
  Future<void> setUserProperties(String userId, String userRole) async {
    try {
      await _analytics.setUserId(userId);
      await _analytics.setUserProperty(name: 'role', value: userRole);
      print('User properties set: ID - $userId, Role - $userRole');
    } catch (e) {
      print('Failed to set user properties: $e');    }  }
  @override
  Future<void> captureException(dynamic exception, dynamic stackTrace) async {
    try {
      await FirebaseCrashlytics.instance.recordError(exception, stackTrace);
      print('Exception captured: $exception');
    } catch (e) {
      print('Failed to capture exception: $e');    }  }
  @override
  Future<void> captureMessage(String message) async {
    try {
      await FirebaseCrashlytics.instance.log(message);
      print('Message logged: $message');
    } catch (e) {
      print('Failed to capture message: $e');    }  }  }
```

Sentry надає можливості відстеження помилок у реальному часі та моніторингу продуктивності (лістинг 4.22).

Лістинг 4.22 – Приклад моніторингового сервісу використовуючи Sentry

```
class SentryMonitoringService implements MonitoringService {
  @override
  Future<void> initialize() async {
    await SentryFlutter.init(
      (options) {options.dsn = 'your_dsn_here'; //Replace with your Sentry DSN
        options.environment = 'production'; // Set the environment
        options.debug = true; // Enable debug mode if needed      },
      appRunner: () => runApp(MyApp()),    );  }
  @override
  Future<void> logEvent(String eventName, Map<String, dynamic> parameters)
async {
    try { Sentry.configureScope((scope) {
        scope.setExtras(parameters);      });
      await Sentry.captureEvent(
        SentryEvent(
```

```
          message: SentryMessage(message: eventName),
          tags: parameters,        ),      );
      print('Event logged to Sentry: $eventName with parameters:
$parameters');
    } catch (e) { print('Failed to log event to Sentry: $e');    }  }
  @override
  Future<void> logLogin(String userId) async {
    // Capture login event in Sentry
    try {
      Sentry.configureScope((scope) {scope.setUser(SentryUser(id: userId));});
      await Sentry.captureMessage('User logged in with ID: $userId');
      print('Login event logged to Sentry for user ID: $userId');
    } catch (e) {
      print('Failed to log login to Sentry: $e');    }  }
  @override
  Future<void> setUserProperties(String userId, String userRole) async {
    // Set user properties in Sentry
    try { Sentry.configureScope((scope) {
        scope.setUser(SentryUser(id: userId, extra: {'role': userRole})); });
      print('User properties set in Sentry: ID - $userId, Role - $userRole');
    } catch (e) { print('Failed to set user properties in Sentry: $e');  }  }
  @override
  Future<void> captureException(dynamic exception, dynamic stackTrace) async
{  try {await Sentry.captureException(exception, stackTrace: stackTrace);
      print('Exception captured in Sentry: $exception');
    } catch (e) { print('Failed to capture exception in Sentry: $e');    }  }
  @override
  Future<void> captureMessage(String message) async {
    try { await Sentry.captureMessage(message);
      print('Message captured in Sentry: $message');
    } catch (e) { print('Failed to capture message in Sentry: $e');  }  }  }
```

Модуль аналітики та моніторингу з реалізаціями для Firebase та Sentry, забезпечує надійну основу для фіксації та аналізу поведінки користувачів, показників продуктивності та помилок у роботі додатків. Таке налаштування гарантує, що розробники можуть підтримувати високі стандарти якості, безпеки та досвіду користувача додатків завдяки комплексним можливостям моніторингу та аналітики.

ВИСНОВКИ

З огляду на дослідження ринку додатків та аналіз статистичних даних можна зробити висновок, що на сучасному ринку спостерігається зростаючий попит на додатки, спрямовані на підвищення продуктивності та спрощення повсякденних робочих процесів. Користувачі шукають інструменти, які допоможуть їм ефективно керувати часом, планувати задачі та організовувати комунікацію в команді. Цей тренд підтверджується як статистикою скачувань та використання додатків, так і результатами опитувань користувачів.

Разом з тим, розробники мобільних додатків мають великий потенціал для створення інноваційних рішень, спрямованих на задоволення потреб сучасного користувача. Враховуючи зростаючий попит на додатки для підвищення продуктивності, компанії можуть активно вкладати ресурси в розробку новаторських програм, які будуть відповідати потребам та очікуванням користувачів, сприяючи їхній ефективності та успішності. Тому створення гнучкої моделі розробки додатків, яка дозволяє швидко адаптуватися до змінних потреб користувачів та ринкових умов, важливе для компаній у сфері мобільної розробки. Гнучка модель дозволяє розробникам ефективно реагувати на зміни в запитах користувачів та швидко впроваджувати нові функції та покращення, що сприяє збільшенню конкурентоспроможності та задоволенню потреб клієнтів.

Фреймворк Flutter обирається для розробки мобільних додатків завдяки своїм унікальним перевагам, які забезпечують ефективність та якість проекту. Його кросплатформна природа дозволяє писати код один раз та використовувати його для створення додатків під різні платформи, що економить час та ресурси. Крім того, можливості Flutter для створення привабливого інтерфейсу та гнучкості налаштування віджетів забезпечують високу якість дизайну. Підтримка з боку Google та активна спільнота розробників додають впевненості щодо майбутнього розвитку та підтримки

платформи. Тому в контексті зростаючого попиту на мобільні додатки, використання Flutter стає стратегічним вибором для команд, які прагнуть успішно розвиватися та конкурувати на ринку.

Запропонована модель (3.2) ще більше посилює ці переваги, пропонуючи комплексну та адаптивну структуру для розробки додатків Flutter. Чітко розмежовуючи такі компоненти, як інтерфейс клієнтувача (CF), логіка взаємодії (IL), база даних (DB) та хмарні сервіси (CS). Вона сприяє модульному підходу, який спрощує розробку, тестування та майбутні вдосконалення.

Включення аналітики та моніторингу (AM) в модель надає командам розробників безцінну інформацію про поведінку користувачів та продуктивність додатків полегшуючи прийняття рішень на основі даних для постійного покращення та доопрацювання додатку. Компонент AM діє як петля зворотного зв'язку, гарантуючи, що додаток завжди відповідає потребам та очікуванням користувачів. Така орієнтація на користувача в поєднанні з притаманною моделі масштабованістю позиціонує запропоновану модель як надійну основу для розробки високоякісних, конкурентоспроможних та зручних для користувача Flutter-додатків на ринку мобільних додатків, що постійно розвивається.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Бешта В.С., Комаричев А.В., Філімончук Т.В., Баранєй Д.І. Модель мобільного додатку, яка орієнтована на обробку даних // Системи управління, навігації та зв'язку. Збірник наукових праць. Полтава: ПНТУ, 2024. Випуск 3 (77). С. 43-47. doi: 10.26906/SUNZ.2024.2.135.

2. GSMA Intelligence 2023, "The Mobile Economy". URL: https://www.gsma.com/solutions-and-impact/connectivity-for-good/mobile-economy/wp-content/ uploads/2024/02/260224-The-Mobile-Economy-2024.pdf

3. Lindner J. (2021), "Must-Know App Revenue Statistics". URL: https://gitnux.org/app-revenue-statistics

4. DataReportal (2023), "Digital 2023: Global Overview Report". URL: https://datareportal.com/reports/digital-2023-global-overview-report

5. Miquido (2023), "Current Trends of AI in Mobile Apps to Watch in 2024". URL: https://www.miquido.com/blog/ai-in-mobile-apps

6. Gibson E. (2023), "Importance of response time in customer service". URL: https://docflite.com/importance-of-response-time-in-customer-service

7. AIContentfy team (2023), "The impact of data privacy and security on customer acquisition". URL: https://aicontentfy.com/en/blog/impact-of-data-privacy-and-security-on-customer-acquisition

8. Carmine Zaccagnino (2020), "Programming Flutter: Native, Cross-Platform Apps the Easy Way (The Pragmatic Programmers)".United States of America: Pragmatic Bookshelf, 370 p.

9. Ecma International (2014), "TC52". URL: https://ecma-international.org/technical-committees/tc52

10. Flutter (2023), "Impeller rendering engine". URL: https://docs.flutter.dev/perf/impeller