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 проєктування мобільних застосунків

(тема)

Виконав:
здобувач 2 року навчання,
групи СПм-23-5
Владислав ХОЛОБОК
(власне ім'я, прізвище)

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. Тетяна ФІЛІМОНЧУК
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти 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

здобувачеві Холобку Владиславу Івановичу
(прізвище, ім’я, по батькові)

1. Тема роботи Модель проєктування мобільних застосунків

затверджена наказом по університету від “ 21 ” квітня 2025 р. № 296 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи 1) мова програмування Kotlin; 2) мова програмування Swift; 3) фреймворк Jetpack Compose; 4) фреймворк SwiftUI; 5) середовище розробки Android Studio 6) середовище розробки Xcode.

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз предметної області;
2) побудова моделі архітектури мобільних рішень;
3) реалізація та аналіз нативного за стосунку;
4) висновки.

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

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	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз літератури та предметної області	22.04.25-26.04.25	
2	Розробка модифікованої моделі проєктування	27.04.25-30.04.25	
3	Проєктування архітектури застосунку	01.05.25-08.05.25	
4	Розробка клієнтської частини застосунку	08.05.25-12.05.25	
5	Розробка серверної частини застосунку	13.05.25-19.05.25	
6	Інтеграція та тестування функціоналу	20.05.25-24.05.25	
7	Аналіз результатів і оптимізація	25.05.25-31.05.25	
8	Подання кваліфікаційної роботи на рецензу-	01.06.25-12.06.25	
	вання		

Дата видачі завдання " 21 " квітня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Тетяна ФІЛІМОНЧУК
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 61 с., 1 дод., 21 джерело.

МОБІЛЬНІ ЗАСТОСУНКИ, НАТИВНА РОЗРОБКА, КРОСПЛАТФОРМНА РОЗРОБКА, ГІБРИДНА РОЗРОБКА, МОДЕЛЬ ПРОЄКТУВАННЯ, СЛУЖБА ДОСТАВКИ.

Об’єкт досліджень – процес розробки мобільних застосунків для Android та iOS.

Предмет дослідження – методи та інструменти нативної, кросплатформної та гібридної розробки мобільних застосунків.

Мета роботи – аналіз підходів до розробки мобільних застосунків, розробка модифікованої моделі проєктування та створення нативного застосунку служби доставки з оцінкою його продуктивності, зручності та адаптивності до українського ринку.

Методи досліджень – аналіз літератури, порівняння, моделювання, практичне тестування.

Розробка мобільних застосунків є ключовою сферою цифрових технологій, що забезпечує ефективну взаємодію в логістиці, електронній комерції та інших індустріях. У роботі досліджено нативну, кросплатформну та гібридну розробку, їхні переваги та недоліки. Нативний підхід, проаналізований на прикладі застосунку служби доставки для Android та iOS, забезпечує високу продуктивність, повний доступ до апаратних можливостей та інтуїтивний UX/UI. Запропонована модифікована модель проєктування сприяє створенню масштабованих рішень. Практичне тестування підтвердило швидкодію, стабільність геолокації. Нативна розробка рекомендована для складних застосунків, тоді як кросплатформні та гібридні підходи виправдані для проєктів з обмеженим бюджетом.

ABSTRACT

Master’s thesis: 61 pages, 1 appendice, 21 sources.

MOBILE APPLICATIONS, NATIVE DEVELOPMENT, CROSS-PLATFORM DEVELOPMENT, HYBRID DEVELOPMENT, DESIGN MODEL, DELIVERY SERVICE.

Research Object – the process of developing mobile applications for Android and iOS.

Research Subject – methods and tools of native, cross-platform, and hybrid mobile application development.

Purpose of the Work – to analyze approaches to mobile application development, develop a modified design model, and create a native delivery service application with an evaluation of its performance, usability, and adaptability to the Ukrainian market.

Research Methods – literature analysis, comparison, modeling, practical testing.

Mobile application development is a key area of digital technologies, enabling efficient interaction in logistics, e-commerce, and other industries. The study examines native, cross-platform, and hybrid development, along with their advantages and disadvantages. The native approach, analyzed through a delivery service application for Android and iOS, ensures high performance, full access to hardware capabilities, and intuitive UX/UI. The proposed modified design model facilitates the creation of scalable solutions. Practical testing confirmed high performance, stable geolocation, and adaptability to the Ukrainian market. Challenges with offline map functionality were identified. Native development is recommended for complex applications, while cross-platform and hybrid approaches are suitable for projects with limited budgets.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ОС – операційна система

АР – адміністративна панель (англ., Admin Panel)

API – інтерфейс програмування застосунків (англ., Application Programming Interface)

AV – аутентифікація та верифікація (англ., Authentication, Verification)

B2B – бізнес для бізнесу (англ., Business-to-Business)

B2C – бізнес для споживача (англ., Business-to-Consumer)

BE – серверна частина (англ., Back-end)

CM – механізми кешування (англ., Caching Mechanisms)

CS – хмарне сховище (англ., Cloud Storage)

DB – база даних (англ., Database)

DS – зберігання даних (англ., Data Storage)

FE – клієнтська частина (англ., Front-end)

FS – файлова система (англ., File System)

MAT – інструменти моніторингу та аналітики (англ., Monitoring and Analytics Tools)

PS – платіжні системи (англ., Payment Systems)

SA – серверна архітектура (англ., Server Architecture)

SCE – захищені канали зв’язку (англ., Secure Communication Channels)

SDE – безпечний обмін даними (англ., Secure Data Exchange)

SDK – набір засобів розробки програмного забезпечення (англ., Software Development Kit)

SL – вхід через соціальні мережі (англ., Social Login)

TT – інструменти тестування (англ., Testing Tools)

UI – інтерфейс користувача (англ., User Interface)

UX – досвід користувача (англ., User Experience)

ВСТУП

На даний час однією із найбільш динамічних галузей, що просувається, є розробка мобільних застосунків, які відкривають нові можливості для надання послуг та взаємодії з користувачами в реальному часі. Проте ключ до успіху мобільного рішення полягає не лише в його інноваційних функціях, а залежить від його архітектури [1].

У вік цифрових технологій мобільні застосунки виступають інструментом для багатьох індустрій, таких як транспорт, електронна комерція, онлайн-банкінг, подорожі, роздрібна торгівля та корпоративні послуги. Застосунки можуть бути орієнтовані на споживачів (B2C) або на бізнес (B2B). B2C-застосунки спрямовані на задоволення індивідуальних потреб користувачів та поділяються на контент-орієнтовані, маркетингові та сервісні. Контент-орієнтовані рішення надають користувачам інформацію, розваги, можливості для спілкування або підвищення продуктивності. Маркетингові, як правило, використовуються для реклами бренду, а сервісні дозволяють виконувати певні задачі, такі як бронювання квитків або перевірка розкладу поїздів.

B2B-застосунки підтримують внутрішні бізнес-процеси компаній, наприклад, управління складом або автоматизацію продажів. Вони зазвичай мають обмежений доступ та призначені для використання лише певними співробітниками. Вибір операційної системи, яку слід підтримувати, є одним з ключових рішень у процесі розробки таких застосунків.

Ринок смартфонів значно виріс за останнє десятиріччя. Протягом цього часу використовувалися різні мобільні ОС, такі як Android, Blackberry, iOS, Symbian та Windows. Сьогодні на ринку домінують дві платформи – Android та iOS, але у майбутньому ситуація може змінитися. Компанії повинні враховувати ці невизначені обставини, приймаючи рішення про стратегію розробки. Вибір ОС для розробки мобільних застосунків залежить не тільки

від частки ринку або орієнтованої індустрії. Виділяють чотири фактори для визначення, яка ОС найкраще підходить для вимог підприємства:

- аудиторія: Android має більшу частку ринку, і за 2023 рік Google Play Store отримав 110 мільярдів завантажень застосунків по всьому світу, в той час як Apple Application Store отримав лише 41,5 мільярдів;

- монетизація: хоча Android має більше завантажень, Apple майже в два рази випереджає за валовими витратами споживачів;

- термін реалізації проєкту: розробка застосунків для Android займає в середньому на 30-40% більше часу, ніж для iOS;

- бюджет: вартість розробки мобільного застосунку залежить від обсягу та складності проєкту, а також від підтримки пристроїв та версій ОС, маркетингових зусиль, участі команди та міждепартаментної взаємодії, а також підтримки та оновлення.

Дана робота присвячена вибору напряму розробки мобільних застосунків (нативного або кросплатформного), який допоможе розробнику прийняти рішення з урахуванням важливих компромісів. Дослідження включає перелік чітких критеріїв, яким слід приділити увагу при розгортанні застосунку.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Розробка мобільних застосунків передбачає орієнтацію на конкретну операційну систему (ОС), що може здійснюватися через нативну або кросплатформну розробку. Розробник повинен володіти знаннями Java або Kotlin для Android та Objective-C або Swift для iOS. Кросплатформні фреймворки надають інструменти для розробки мобільних застосунків без необхідності глибокого знання мов програмування. Наприклад, можна використовувати Dart з Flutter, JavaScript з React Native, C# з Xamarin або вебтехнології з Adobe PhoneGap. Як зазначалося раніше, сьогодні на ринку домінують дві операційні системи – Android та iOS, і знання їх характеристик є важливим для розробки застосунків [2].

Основною метою Android є створення відкритої платформи, доступної для операторів, виробників пристроїв та розробників. Вона базується на ядрі Linux та забезпечує широку функціональність через API Java. Завдяки відкритій архітектурі Android підтримує широкий спектр пристроїв різних цінових категорій, що робить її найбільш популярною мобільною операційною системою у світі. Розробка застосунків для Android є дешевшою порівняно з iOS завдяки менш суворим правилам публікації в Google Play, нижчим витратам на реєстрацію розробника та можливості розповсюдження програм не лише через офіційний магазин, а й через сторонні платформи.

Операційна система iOS, яка була розроблена Apple, базується на Unix-подібній архітектурі та має кілька рівнів абстракції: ядро ОС, рівень базових сервісів, рівень мультимедіа та рівень інтерфейсу користувача. Завдяки глибокій оптимізації апаратного та програмного забезпечення iOS забезпечує високу продуктивність, безпеку та енергоефективність. Система відома суворими вимогами до кінцевих рішень, що збільшує витрати на їх розробку. Це включає необхідність дотримання жорстких правил Apple, платну реєстрацію розробника та обов’язкове використання мов програмування

Swift або Objective-C. Водночас ці обмеження сприяють високій якості застосунку та кращому захисту користувачів.

При нативній розробці передбачено використання SDK та фреймворків [3], що підтримуються операційною системою. Застосунки створюються окремо для кожної платформи, що забезпечує оптимальну продуктивність, доступ до всіх можливостей пристрою та кращий досвід користувача.

Android-застосунки розробляються у середовищі Android Studio, в якому використовуються мови програмування Kotlin, Java або C++. Мобільні рішення мають доступ до повного функціоналу ОС через Java API та Android NDK (для роботи з рідним кодом на C/C++). Android використовує маніфестний файл для опису компонентів застосунку, необхідних дозволів, мінімального рівня API та налаштувань безпеки. Графічний інтерфейс створюється за допомогою XML або Jetpack Compose.

Розробка iOS [4] застосунку здійснюється у середовищі Xcode з використанням мови Swift або Objective-C. Swift забезпечує автоматичне управління пам’яттю (ARC), підтримку Unicode та високий рівень продуктивності. Для створення інтерфейсу використовують UIKit або SwiftUI, що спрощує розробку завдяки декларативному синтаксису. iOS-розробка включає використання Storyboard для візуального проєктування застосунків, а також підтримує Metal для графічного рендерингу та Core ML для інтеграції машинного навчання.

Плюсами нативних за стосунків є:

- висока продуктивність – застосунки оптимізовані для конкретної платформи, що забезпечує швидку роботу, плавну анімацію та ефективне використання ресурсів пристрою;

- кращий досвід користувача (UX/UI) – дизайн та взаємодія відповідають рекомендаціям платформи (Human Interface Guidelines для iOS, Material Design для Android), що робить інтерфейс більш інтуїтивним;

- повний доступ до функцій пристрою – можливість використання камери, GPS, Bluetooth, датчиків руху, push-сповіщень, біометричної автентифікації (Face ID, Touch ID), а також доступ до Metal (для рендерингу графіки) та Core ML (для машинного навчання в iOS);

- краща безпека – використання вбудованих механізмів захисту ОС, зокрема Secure Enclave (iOS), шифрування даних, а також більш жорсткі правила App Store та Google Play, що зменшує ризик розповсюдження шкідливих програм;

- стабільна робота та підтримка в магазинах – застосунки легше публікувати в офіційних магазинах (App Store, Google Play), вони отримують більше довіри від користувачів та мають доступ до просування через внутрішні платформи;

- краща інтеграція з оновленнями ОС – нативні застосунки швидше адаптуються до нових можливостей операційної системи та нових пристроїв, оскільки використовують офіційні SDK.

В якості мінусів нативних застосунків можна відмітити:

- вищу вартість розробки – потрібно створювати окремі версії для iOS та Android, що збільшує витрати на розробку, тестування та підтримку;

- довший час розробки – окремий код для кожної платформи потребує більше часу на розробку, порівняно з кросплатформними рішеннями;

- підтримка та оновлення – кожна версія застосунку (iOS, Android) потребує окремих оновлень, що ускладнює підтримку, особливо при одночасному випуску нових функцій;

- обмеження платформи – деякі функції можуть бути доступні тільки на одній платформі (наприклад, Dynamic Island на iOS), що ускладнює підтримку однакового функціоналу на всіх пристроях.

Нативна розробка мобільних застосунків забезпечує найвищу продуктивність, повний доступ до можливостей пристрою та кращий досвід користувача. Вона ідеально підходить для створення складних, продуктивних та безпечних застосунків. Однак цей підхід вимагає більше ресурсів, оскільки

розробка ведеться окремо для кожної платформи, що збільшує витрати та час на підтримку.

Кросплатформна розробка – це підхід до створення мобільних застосунків, який дозволяє використовувати єдиний код для роботи на різних операційних системах (iOS та Android). Це економить час та ресурси, оскільки не потрібно писати окремий код для кожної платформи. Кросплатформні фреймворки використовують спеціальні механізми для компіляції або інтерпретації коду, що дозволяє застосунку працювати на різних ОС. Це значно скорочує витрати та спрощує підтримку, але може поступатися нативним рішенням у продуктивності та інтеграції з платформами.

З переваг кросплатформної розробки можна відмітити:

- єдиний код для кількох платформ – зменшує витрати на розробку, тестування та підтримку;
- швидке розгортання та оновлення – виправлення помилок та нові функції можуть випускатися одночасно для всіх платформ;
- розвинена спільнота – багато популярних кросплатформних інструментів мають активну підтримку open-source спільноти;
- краща гнучкість – можливість розгортання не тільки на iOS та Android, а й на Web, Desktop (indows, macOS, Linux).

В якості недоліків кросплатформних рішень є:

- гірша продуктивність у порівнянні з нативними застосунками – через необхідність узгодження з різними платформами;
- обмежений доступ до функцій пристрою – хоча більшість API доступні, деякі нативні функції можуть вимагати додаткових обхідних рішень;
- проблеми зі стабільністю – складні застосунки можуть мати більше багів, особливо якщо підтримується багато пристроїв та ОС;
- графічні обмеження – відображення складної графіки та анімацій може бути менш ефективним, ніж у нативних рішеннях;

- можливі обмеження в магазинах застосунків – App Store та Google Play можуть мати більш суворі вимоги до кросплатформних застосунків, що використовують WebView або віртуальні машини.

Розглянемо переваги та недоліки фреймворки для кросплатформних застосунків.

React Native [5] – це відкритий фреймворк, розроблений Facebook у 2015 році. Він дозволяє використовувати JavaScript для створення застосунків із природним виглядом та поведінкою.

Перевагами використання React Native є:

- повторне використання понад 80% коду між платформами;
- гарячий перезавантажувач, що дозволяє миттєво бачити зміни;
- зростаюча спільнота розробників.

Недоліками використання React Native є:

- деякі компоненти потребують окремого коду для iOS та Android;
- відставання в оновленнях щодо нових версій ОС;
- порівняно низька продуктивність застосунків.

Flutter [6] – це відкритий фреймворк від Google, випущений у 2018 році. Він використовує мову програмування Dart та забезпечує створення застосунків із єдиним дизайном для iOS та Android [7].

Перевагами використання Flutter є:

- гаряче перезавантаження для швидкого внесення змін;
- підтримка Google Material Design та Cupertino для iOS;
- швидка анімація та простота інтеграції.

Недоліками використання Flutter є:

- відсутність підтримки для Android TV та Apple TV;
- великі розміри застосунків через використання вбудованих віджетів;
- обмежена кількість бібліотек.

Xamarin – це кросплатформний фреймворк для розробки мобільних застосунків на iOS, Android та Windows за допомогою мови C# та .NET, який

був заснований в 2011 році. Він дозволяє використовувати єдиний код для різних платформ, що спрощує розробку та підтримку застосунків [8].

Перевагами використання Xamarin є:

- один код для Android, iOS та Windows (близько 90% коду можна повторно використовувати);
- нативна продуктивність: Xamarin забезпечує швидку роботу застосунку;
- доступ до всіх функцій пристрою (камера, GPS, Bluetooth, датчики) – все це можна використовувати через Xamarin.Essentials;
- весь код пишеться на C# та .NET.

Недоліками використання Xamarin є:

- розмір застосунків може бути більшим, ніж нативні;
- продуктивність трохи нижча, ніж у нативних застосунків особливо важкі анімації та графіка можуть працювати гірше;
- обмежена підтримка нових функцій iOS/Android.

Кросплатформна розробка [9] є ідеальним вибором для застосунків, які не мають складних графічних чи апаратних вимог. Однак для ресурсомістких, продуктивних та інтерактивних застосунків кращим вибором залишається нативна розробка.

Гібридна розробка застосунків [10] – це комбінація нативних та вебрішень, де розробникам необхідно вбудовувати код, написаний такими мовами, як CSS, HTML та JavaScript у нативний застосунок за допомогою плагінів, включаючи Ionic's Capacitor, Apache Cordova тощо, які дозволяють отримати доступ довбудованих функцій. HTML5 відповідає за структуру та вміст гібридного застосунку. У CSS3 створюються різні таблиці стилів, які визначають, як певні компоненти мають відображатися у кінцевому рішенні. JavaScript використовується для розробки різних інтерфейсів (API) із апаратним забезпеченням системи, що забезпечує доступ до GPS, камери, мікрофона та різних датчиків на смартфоні. Розробка з використанням HTML5, CSS3 та JavaScript є великою перевагою для багатьох агентств,

оскільки веброзробники вже мають великий досвід програмування. Це дозволяє інтегрувати в гібридний застосунок специфічні для операційної системи функції. Найпопулярнішими інструментами для розробки гібридних додатків є Ionic, Onsen та Sencha Touch.

Для користувачів гібридний застосунок виглядає як нативний. Він завантажується через магазин застосунків та встановлюється на пристрої. Після встановлення застосунок повністю доступний офлайн. Коли гібридний застосунок відкривається, запускається інтерфейс, схожий на браузер, який працює аналогічно до вебзастосунку. Однак, гібридні застосунки використовують не веббраузери, а вебпредставлення.

Перевагами гібридної розробки [11] є:

- один код для iOS та Android;
- швидший цикл розробки;
- менші витрати;
- оновлення можливі без необхідності відправлення нової версії в App Store або Google Play.

З недоліків гібридної розробки можна відмітити:

- меншу продуктивність ніж у нативних [12] (особливо при складній графіці);
- обмежений доступ до нативних API (часто потребує сторонніх плагінів);
- можливі проблеми з UX/UI (деякі компоненти можуть виглядати або працювати інакше на різних платформах).

2 РОЗРОБКА МОДЕЛІ ПРОЄКТУВАННЯ МОБІЛЬНИХ ЗАСТОСУНКІВ

2.1 Аналіз останніх досліджень та публікацій

Під час вивчення сучасних підходів до розробки мобільних застосунків було проаналізовано низку наукових публікацій, присвячених архітектурі, інструментам та принципам їх проєктування.

У роботі [13] розглянуто модель мобільного застосунку, яка орієнтована на обробку даних. Основна увага у роботі приділяється взаємодії ключових компонентів, таких як інтерфейс користувача (UI), клієнтська та серверна частини, API, заходи безпеки, модулі аналітики та моніторингу, хмарні сервіси та модулі тестування. Автори натякають, що ретельно продумана модель мобільного застосунку є ключем до його успішної реалізації та конкурентоспроможності на ринку, забезпечує ефективну взаємодію компонентів, безпеку та продуктивність, що робить її придатною для сучасних технологічних вимог.

У роботі [14] розглянуто основні аспекти створення мобільних застосунків та вибір відповідних інструментів для їх розробки. Автори аналізують переваги та недоліки нативного та кросплатформного підходів. Використання першого забезпечує високу продуктивність та оптимізований UX, але їх розробка значно дорожча. Застосунки, які розроблено з орієнтацією на другий підхід, є більш економічними, проте можуть мати обмеження у продуктивності та гнучкості. Загалом в статті підкреслюється важливість усвідомленого підходу до вибору інструментів розробки, що сприяє створенню якісних, продуктивних та доступних мобільних рішень.

Метою роботи [15] є порівняльний аналіз трьох популярних крос-платформних рішень для створення мобільних застосунків: React Native, Flutter та Kotlin Multiplatform. Вибір між цими технологіями, як правило, залежить від вимог проєкту, кожна із розглянутих мов та технологій може

бути використана тільки з точці зору аналізу завдань та сценаріїв використання. Фреймворк React Native підходить для реалізації швидких прототипів та веборієнтованих команд, Flutter є оптимальним для застосунків із складним UI та необхідністю однакового вигляду на всіх платформах, а Kotlin Multiplatform найбільше підходить для проєктів, які потребують високої продуктивності та інтеграції з нативними рішеннями. Таким чином, використання кросплатформних рішень дозволить розробнику зменшити витрати на розробку мобільних застосунків, проте кожне з рішень має свої переваги та обмеження, які необхідно враховувати при обиранні технології.

В роботі [16] розглянуто архітектуру платформи Android та структуру мобільного застосунку. Автори докладно розкривають сутність та призначення основних програмних компонентів: Activity, Service, Content Provider та Broatcast Resiver, наводять методи, які слід використовувати для побудови інтуїтивного інтерфейсу користувача. Особлива увага приділяється формам збереження та обробки даних на мобільному пристрої.

Метою роботи [17] є ретельний аналіз підходів та інструментальних засобів кросплатформної розробки мобільних застосунків, які дозволяють прискорити процес написання програмного коду. Автор проводить аналіз популярних фреймворків, які можливо використовувати для розробки кінцевого рішення за певними критеріями. При обиранні технології слід керуватися потребами застосунку, бюджетом, наявністю команди розробників та іншими чинниками. Як висновок, автор пропонує використовувати React Native, який задовольняє більшість потреб.

У роботі [18] проведено аналіз вимог до проєкту, проєктуванню, подальшої реалізації з використанням відповідного інструментарію, тестуванню та подальшої підтримки. Автор розглядає процес побудови мобільних застосунків, як набір окремих модулів, що дозволяє забезпечити гнучкість системи. Головними складовими мобільного застосунку є UX та UI, моделювання яких повинно розпочинатись на етапі проєктування. Також автор пропонує окрему увагу приділити зберіганню інформації та наводить

переваги та недоліки використання існуючих на даний час баз даних.

Розглянуті роботи демонструють динамічний розвиток мобільних технологій та різноманітність підходів до створення застосунків. Вони висвітлюють ключові аспекти, починаючи від вибору моделі розробки до використання сучасних технологій, таких як React Native, Flutter, Kotlin Multiplatform, а також хмарних сервісів та локальних баз даних. Таким чином, розробка мобільних застосунків – це не лише актуальна, а й стратегічно важлива сфера розвитку сучасних цифрових технологій, яка формує майбутній досвід користувача, змінює способи ведення бізнесу та створює нові можливості для взаємодії у цифровому світі.

Проведений аналіз виявив, що існуюча на даний момент модель, яка орієнтована на розробку мобільних застосунків має наступні складові (2.1):

$$M=\{FE, BE, API, DS\}, \quad (2.1)$$

де FE (front-end) – клієнтська частина;

BE (back-end) – серверна частина;

API (application programming interface) – інтерфейс програмування застосунків, який забезпечує взаємодію між клієнтською та серверною частинами;

DS (data storage) – методи, орієнтовані на зберігання даних застосунку.

Наведена модель (2.1) підтримує просту взаємодію між кінцевим користувачем та мобільним застосунком, та має ряд недоліків:

- вона може бути не точною, оскільки не враховує деякі аспекти та взаємодії між окремими компонентами, що в свою чергу впливає на результати роботи застосунку та його ефективність;
- просту модель легше проєктувати та впроваджувати, але це обмежує її гнучкість та подальше удосконалення;
- просту модель не можливо використовувати в задачах, де потрібна складна або спеціалізована обробка.

2.2 Модифікована модель проєктування мобільних за стосунків

Для усунення перелічених недоліків було вирішено модифікувати модель (2.1) за рахунок додавання та розширення її окремих складових. Запропонована модель (2.2) буде забезпечувати більш ефективну інтеграцію front-end та back-end компонентів, більш високий рівень продуктивності, безпеки та легкість масштабування за рахунок врахування ключових аспектів сучасної розробки [19]:

$$M = \{OS, FE, BE, DS, UX, UI, PS, G, SDE, PN, TT, MAT\}, \quad (2.2)$$

де OS (operating system) – нативна розробка на iOS або Andoid;

UI (user interface) – інтерфейс користувача;

UX (user experience) – досвід користувача;

PS (payment systems) – інтеграція з платіжними системами;

G (geolocation) – технологія визначення місцезнаходження;

SDE (secure data exchange) – модуль безпечного обміну даними;

PN (push notifications) – система повідомлень;

TT (testing tools) – модуль інструментів тестування;

MAT (monitoring and analytics tools) – інструменти для моніторингу та аналітики.

На даний час на ринку розробки мобільних застосунків конкурують дві платформи: iOS та Android. Головна їх відмінність полягає в використанні екосистеми. Як правило, платформа iOS має неповторну ідентичність, яка підтримується суворим контролем якості та дизайном. Платформа Android, надає розробникам більш гнучке середовище для розробки з використанням широкого спектру пристроїв. Кожна із платформ має свої переваги та недоліки, але орієнтуватись потрібно на цілі проєкту, доступні людські та грошові ресурси та очікувану продуктивність.

Front-end програмування відповідає за роботу клієнтської частини

застосунку (веб або мобільного). На даному етапі розробки особлива увага приділяється розробці інтерфейсу між кінцевим користувачем та серверною частиною мобільного рішення. Складова FE запропонованої моделі (2.2) відповідає за введення інформації від користувача, її первинну обробку, а також подальшу її відправку на сервер за допомогою відповідного API.

Back-end програмування орієнтоване на розробку серверної частини програми, яка відповідає за передачу даних між користувачами. Складова BE моделі, що запропонована (2.2) містить наступні інструменти (2.3):

$$BE = \{SA, API, AP, M\}, \tag{2.3}$$

де SA (server architecture) – серверна архітектура, яка містить алгоритми завантаження даних, методи авторизації, кешування, бази даних та інше;

API (application programming interface) – інтерфейс прикладного програмування, який визначає функціональність серверної логіки;

AP (admin panel) – адміністративна панель застосунку, функціонал якої розробляється виходячи з цілей та задач проєкту;

M (metrics) – метрики, які оцінюють ефективність проєкту (статистичні дані щодо приросту користувачів, активності, щоденна відвідуваності проєкту в цілому та його окремих розділів, демографічні дані).

Складова DS (2.4) орієнтована на використання низки методів, які дозволяють здійснювати зберігання даних проєкту. Слід розуміти, що база даних є ключовим компонентом будь-якого сучасного застосунку, який забезпечує надійне зберігання та управління інформацією. У вебзастосунках база даних використовується, як правило, для зберігання контенту, даних користувача, а також журналів транзакцій. У мобільних застосунках вона часто виступає у ролі локального сховища для тимчасового зберігання даних або кешування, що забезпечує швидкий доступ до інформації навіть без підключення до мережі.

$$DS = \{DB, CS, FS, CM\}, \quad (2.4)$$

де DB (database) – організована структура для зберігання, зміни та обробки взаємозалежної інформації, переважно великих обсягів;

CS (cloud storage) – хмарні сховища, орієнтовані на зберігання даних на віддалених серверах, доступ до яких здійснюється через Інтернет;

FS (file system) – файлова система, орієнтована на зберігання та управління даними на пристрої зберігання;

CM (caching mechanisms) – набір механізмів, що кешують.

В якості складової DB можливо використовувати:

- реляційні бази даних, які організовують дані у таблиці, що з'єднані між собою за допомогою ключів (MySQL, PostgreSQL, SQLite, MariaDB);
- NoSQL бази даних, які призначені для роботи з неструктурованими або слабо структурованими даними. NoSQL включають документо-орієнтовані, колоночні, графові та інші типи баз даних (MongoDB, CouchDB, Firebase Firestore, Apache Cassandra, HBase, ScyllaDB, Neo4j, OrientDB);
- розподілені бази даних, де дані зберігаються на кількох серверах, що забезпечує високу доступність, відмовостійкість та масштабованість (CockroachDB, Google Cloud Spanner, TiDB).

Хмарні сховища (CS) – це сервіси, які дозволяють зберігати та керувати даними у віддалених дата-центрах, забезпечуючи доступність, безпеку та масштабованість. Перевагами використання хмарних сховищ є можливість резервного копіювання та захист даних, а також спільна робота з ними з будь-якого пристрою. Більшість хмарних сховищ, орієнтовані під використання конкретних цілей: для зберігання файлів (Google Drive, Dropbox, OneDrive), для застосунків (Amazon S3, Firebase, Cloud Storage), для баз даних (Cloud SQL, Firestore), для віртуальних машин (Amazon EBS, Azure Disks).

Файлова система (FS) – це структура, яка використовується операційною системою для розміщення та упорядкування даних, управління

доступом, відстеження використання простору, веде процес журналювання (NTFS, ext4, XFS), здійснює захист від збоїв та відновлення даних збоїв (Btrfs, ZFS).

Файлові системи різняться за будовою та функціонуванням: локальні використовуються на жорстких дисках, SSD та флеш носіях (FAT32, exFAT, NTFS, APFS, ext4, XFS, Btrfs), мережні – для спільного доступу до мережі (NFS, SMB, AFP, WebDAV), розподілені – в кластерах та хмарних системах (GFS, HDFS, CephFS, GlusterFS). Вибір файлової системи залежить від конкретних потреб, сумісності з ОС та типу носія інформації.

Кешування – тимчасове зберігання даних, які часто використовуються у швидкому сховищі (оперативна пам'ять, SSD, диск) для прискорення доступу та зниження навантаження на основне джерело даних.

Існують різні види кешування (СМ):

- апаратне: кеш-пам'ять процесора, кеш-контролери жорстких дисків;
- програмне: в оперативній пам'яті, на диску;
- мережне: кешування у CDN, DNS-кеш.

Прикладами популярних інструментів кешування є Redis, Memcached, Varnish, NGINX FastCGI Cache, Cloudflare CDN, їх використання значно покращує швидкодію застосунків.

Інтерфейс користувача (UI) – це один із головних елементів мобільного застосунку. Використання існуючих на даний час фреймворків та спеціалізованих бібліотек спрощують його розробку.

Кросплатформні UI-фреймворки дозволяють писати один код та запускати його на платформах iOS та Android. Наприклад, Flutter дозволяє розробляти UI за допомогою віджетів (Material Design, Cupertino), має гнучку систему кастомізації інтерфейсу та високу продуктивність завдяки використання рушия Skia. Фреймворк React Native має компоненти для створення UI у стилі iOS та Android, підтримує сторонні UI-бібліотеки та дозволяє використовувати нативні модулі кастомізації.

Нативні UI-фреймворки використовуються для розробки під конкретну платформу. Наприклад, для розробки під Android можливо використовувати зв’язку XML та View System для класичного способу розмітки UI або Jetpack Compose для написання UI в декларативному стилі. Для iOS в якості основного фреймворку можливо використовувати UIKit або SwiftUI, які забезпечують декларативний підхід для організації швидкого прототипування.

Досвід користувача (UX) – це емоційні враження людини від взаємодії з кінцевим рішенням та при розробці слід дотримуватися принципів UX-дизайну:

- простота: мінімалізм в інтерфейсі та зрозуміла навігація;
- швидкодія: швидкість завантаження та чуйність критичні;
- інтуїтивність: логічна архітектура, зрозумілі жести та елементи управління;
- доступність: підтримка різних екранів, шрифтів, режимів;
- зворотній зв'язок: анімації, вібрації, підказки.

Складова PS моделі, що пропонується, забезпечує зручні та безпечні онлайн-платежі для мобільного застосунку та надає інтеграцію з платіжними системами, такими як Stripe, PayPal, Apple Pay та Google Pay. Ці системи дозволяють приймати платежі від клієнтів по всьому світу, використовуючи різні методи оплати. Варто зазначити, що доступність та умови використання платіжних систем можуть відрізнятися залежно від країни. Наприклад, у деяких певні платіжні системи можуть бути недоступні або мають обмеження.

Технологія визначення місцезнаходження користувача або пристрою (складова G) на основі GPS, Wi-Fi, мобільних мереж або IP-адреси використовується у мобільних застосунках, вебсайтах та корпоративних рішеннях для доставки персоналізованого контенту, логістики, картографії та навігації. На даний час технологія геолокації [20] застосовується у:

- логістиці та доставці: трекінг посилок, моніторинг кур'єрів;

- навігації: GPS-карти для автомобілів, туристичні маршрути;
- локальних сервісах: пошук найближчих закладів, ресторанів та ін.;
- безпеці: відстеження викрадених гаджетів.

Модуль, який відповідає за безпечний обмін даних (SDE) – це комплекс заходів, що включає механізми шифрування, захищені канали зв'язку та автентифікацію (2.5). Чим більш чутливі дані передаються, тим більше рівнів захисту слід застосовувати при розробці застосунку. Впровадження ефективних практик безпеки є критичним для забезпечення конфіденційності, цілісності та доступності даних користувачів.

$$SDE = \{E, SCE, AV, SL\}, \quad (2.5)$$

де E (encryption) – механізми шифрування;

SCC (secure communication channels) – використання захищених каналів зв'язку;

AV (authentication + verification) – аутентифікація та авторизація;

SL (social login) – аутентифікація через акаунти соціальних мереж.

Механізми шифрування (E) – це набір засобів, які дозволяють здійснювати перетворення даних у захищений формат, що можна прочитати лише за допомогою спеціального ключа. Шифрування використовується для захисту конфіденційної інформації під час її зберігання та передачі. На даний час існує декілька типів шифрування, які розрізняються за алгоритмами та областями застосування:

- симетричне шифрування: AES, DES, 3DES, Blowfish/Twofish;
- асиметричне шифрування: RSA, ECC, Diffie-Hellman;
- хешування: SHA, MD5, bcrypt / scrypt / Argon2;
- шифрування на транспортному рівні: TLS, SSL, End-to-End (E2EE);
- стеганографія.

Вибір типу шифрування залежить від задачі, яка вирішується. Наприклад, для захисту файлів слід обрати AES-256, для веббезпеки – TLS +

RSA/ECC, для зберігання паролів – bcrypt або Argon2, для обміну ключами – Diffie-Hellman, для зашифрованих месенджерів – End-to-End (E2EE).

Використання надійних протоколів передачі даних (SCE) при розробці мобільних застосунків дозволить зробити кінцеве рішення надійним, стабільним та затребуваним у користувачів. Вибір протоколу передачі даних також слід орієнтувати на вирішення конкретних задач. Наприклад, для організації безпечних вебзастосунків та API слід обирати HTTPS або TLS, для фінансових транзакцій орієнтуватись на TLS, HMAC, OAuth.

Аутентифікація та авторизація (AV) – це важливі процеси у сфері інформаційної безпеки, які використовують разом, але вони виконують різні функції. Аутентифікація – перевіряє особу або систему, яка намагається отримати доступ до ресурсу (2FA/MFA). Аутентифікація підтверджує, що користувач є тим, за кого себе видає, за допомогою облікових даних, таких як логін та пароль, біометричні дані або інші методи. Авторизація – це визначення прав та привілеїв аутентифікованого користувача щодо доступу до певних ресурсів або виконання дій (OAuth, JWT, 2FA). Після успішної аутентифікації система перевіряє, які дії дозволені цьому користувачу.

Можливість входу через акаунти соціальних мереж (SL) дозволяє користувачам здійснювати аутентифікацію на вебсайтах або в мобільних застосунках, використовуючи облікові записи із соціальних мереж, таких як Facebook, Google, Twitter, GitHub, LinkedIn. Така аутентифікація спрощує процес реєстрації та входу для користувачів, оскільки їм не потрібно створювати новий обліковий запис або запам'ятовувати ще один пароль. Перевагами входу через акаунти соціальних мереж є швидка реєстрація (без паролю), зручність для користувачів (авто-вхід), безпека (OAuth 2.0 + шифрування) та підтримка різних соцмереж (Google, Facebook, Apple).

Технологія push-сповіщень (PN) дозволяє надсилати користувачам повідомлення навіть тоді, коли застосунок не активний або працює у фоновому режимі. Такі повідомлення використовуються для інформування про важливі оновлення, новини, акції чи події, залучаючи користувачів до

взаємодії із застосунком. Push-сповіщення доставляються в режимі реального часу; можуть містити текст, зображення, кнопки та інші інтерактивні елементи; працюють через сервери повідомлень (наприклад, Firebase Cloud Messaging [21] для Android та APNs для iOS); використовуються для підвищення залученості користувачів.

Процес тестування є обов'язковою складовою розробки кінцевого продукту, який забезпечує стабільність, надійність та якість програмного забезпечення. Тестування дозволяє виявляти та усувати помилки на ранніх етапах, що мінімізує ризик збоїв після впровадження змін чи масштабування застосунку, а також гарантує, що функціонал працює відповідно до заданих специфікацій, забезпечуючи якісний досвід користувача. У розробці слід обов'язково застосовувати різні види тестування (2.6):

$$TT = \{F, UA/UX, P, S, C, A\}, \quad (2.6)$$

де F (functional) – функціональне тестування, яке перевіряє роботу застосунку в цілому;

UA/UX – тестування зручності та інтуїтивності інтерфейсу, роботи анімації, зрозумілості;

P (productivity) – тестування продуктивності (швидкість відповіді, навантаження);

S (security) – тестування безпеки (захист даних при пересиланні та зберіганні);

C (compatibility) – тестування сумісності (робота на різних пристроях, із різними версіями ОС);

A (automated) – автоматизоване тестування (з використанням скриптів).

Наявність інструментів аналітики та моніторингу (MAT) є важливими складовими сучасного застосунку, які забезпечують глибоке розуміння поведінки користувачів та продуктивності системи. Інтеграція інструментів аналітики дозволяє відстежувати активність користувачів, аналізувати

взаємодію з різними елементами інтерфейсу та збирати ключові метрики, такі як конверсії, час сесій та залученість.

Моніторинг продуктивності системи забезпечує можливість своєчасного виявлення та вирішення потенційних проблем, таких як високий час відгуку сервера, перевантаження бази даних чи збої в роботі API. Використання таких інструментів, як Google Analytics, Firebase Analytics, Mixpanel або Sentry, дозволяє автоматизувати процес збору та аналізу даних, забезпечуючи актуальну інформацію для прийняття відповідних рішень. Такий підхід підвищує якість досвіду користувача, дозволяє своєчасно реагувати на проблеми та адаптувати функціонал відповідно до потреб цільової аудиторії. Інтеграція аналітики та моніторингу уніфікує процес управління даними, сприяючи підвищенню ефективності розробки та підтримки застосунків.

2.3 Висновки по розділу

В ході аналізу підходів для проєктування та розгортання мобільних застосунків було розглянуто наукові публікацій, які пропонують використовувати різноманітні мобільні платформ та SDK з вказанням на їх переваги та недоліки. Наявність великої кількості інструментів розробки створюють унікальні проблеми, такі як вибір SDK, досвід користувача, стабільність фреймворка, легкість оновлень, вартість розробки для кількох платформ та час виходу застосунку на ринок.

Проведений аналіз показав, що вибір напряму розробки мобільних застосунків, як правило, залежить від цілей проєкту, доступних ресурсів та очікуваної продуктивності: використовувати нативні застосунки слід для складних рішень із високими вимогами до продуктивності, кросплатформні підходи для ситуацій, коли є потреба створювати мобільні рішення для широкої аудиторії, а гібридні та PWA-рішення для веборієнтованих додатків з меншими вимогами до продуктивності. Також слід зазначити, що нативні

рішення розробляються окремо для кожної платформи, що потребує більше зусиль: як грошових, так і чоловічих. В протилежність нативній розробці кросплатформна дозволяє створювати застосунок з використанням єдиного коду для кількох платформ. Вибір кросплатформного підходу може здатися очевидним, оскільки можна мати єдину базу коду для кількох ОС, що робить розробку та підтримку швидшими, простішими та дешевшими. Однак це не завжди так. Кросплатформні фреймворки мають обмеження, які роблять їх менш здатними для розробки застосунків у порівнянні з нативним підходом і важко інтегруються з API пристрою. Також слід зазначити, що інструменти для кросплатформної розробки складніші для тестування та налагодження, ніж нативні.

При розробці мобільних застосунків можливо використовувати низку інструментів дизайну та прототипування для створення макетів, прототипів та інтерактивних інтерфейсів (Figma, Adobe XD, Sketch, InVision). Каркасні макети можуть допомогти з проєктуванням логіки взаємодії користувачів із продуктом (Miro, Whimsical, Balsamiq). Дизайн-системи та UI-кити – це готові компоненти, які прискорюють роботу та роблять дизайн одноманітним (Material Design, Apple HIG, Figma Community). Використання при розробці застосунків інструментів для дослідження та тестування UX допоможе проаналізувати поведінку користувачів, протестувати прототипи та зібрати зворотний зв'язок (Hotjar, Google Analytics, Maze, UserTesting, Lookback). Інструменти тестування доступності допоможуть перевірити, наскільки інтерфейс зручний для людей з обмеженими можливостями (Stark, Axe DevTools, WAVE).

Модель вибору підходу розробки мобільного застосунку має низку складових, які орієнтовано на використання різноманітних інструментів розробки. В залежності від потреб застосунку обирається інструментарій, який дозволить збалансувати продуктивність, швидкість розробки та витрати. Для складних та ресурсомістких застосунків доцільним є використання нативної розробки, яка забезпечує найвищу продуктивність і доступ до всіх

можливостей пристрою. Кросплатформні фреймворки дозволяють пришвидшити розробку та зменшити витрати, проте можуть мати обмеження щодо інтеграції з API пристрою та продуктивності. Використання гібридних та PWA-рішень виправдане для веборієнтованих продуктів із меншими вимогами до продуктивності. Крім того, інструменти для дизайну, UX-досліджень та тестування забезпечують створення зручного, доступного та функціонального інтерфейсу. Запропонована модель підвищує консистентність коду, оскільки всі частини проєкту розроблятимуться в єдиному стилі та з використанням єдиних парадигм. Кожен із цих компонентів впливає на загальну структуру застосунку, створюючи стабільну, продуктивну та легко масштабовану архітектуру.

3 ПРАКТИЧНИЙ АНАЛІЗ НАТИВНОГО МОБІЛЬНОГО ЗАСТОСУНКУ СЛУЖБИ ДОСТАВКИ

Сучасна цифрова економіка значною мірою залежить від мобільних застосунків, які забезпечують ефективну взаємодію між клієнтами, постачальниками послуг та кур’єрами в сфері доставки. Аналіз, проведений у попередніх розділах, встановив, що нативна розробка є оптимальним підходом для застосунків, які потребують високої продуктивності, повного доступу до апаратних можливостями пристрою, зокрема геолокаційних API, та відповідності стандартам дизайну.

На базі модифікованої моделі проєктування (2.2), яка включає клієнтську частину (FE), зберігання даних (DS), геолокацію (G) та принципи UX/UI, у цьому розділі розглядається приклад нативного застосунку служби доставки для платформ Android та iOS. Буде проведений аналіз реалізації базового функціоналу – перегляду списку замовлень, додавання нового замовлення та відстеження геолокації кур’єра – з точки зору відповідності вимогам продуктивності, зручності використання та сумісності.

Для ОС Android приклад базується на технологіях Kotlin, Jetpack Compose та Room, а для iOS – на Swift, SwiftUI та Core Data. Практична значущість цього аналізу полягає у його здатності продемонструвати, як нативні рішення відповідають сучасним ринковим викликам, зокрема зростанню попиту на швидку та надійну доставку, що є особливо актуальним для українського ринку з його активним розвитком цифрових сервісів.

Методологія аналізу є поєднанням теорії, яка сформована в попередніх розділах, а також оцінки можливостей масштабування для впровадження додаткових функцій, зокрема додавання push-сповіщень (PN) та платіжних систем (PS). Такий підхід дозволяє продемонструвати переваги нативної розробки, підкреслюючи її здатність створювати ефективні та інтуїтивні рішення для служб доставки, що відповідають сучасним ринковим вимогам.

3.1 Опис прикладу нативного застосунку служби доставки

Прикладом нативного застосунку служби доставки слугує узагальнене мобільне рішення, яке забезпечує базовий функціонал для клієнтів (B2C) та кур’єрів (B2B). Основні функції застосунку включають:

- перегляд списку замовлень: відображення переліку замовлень із деталями (назва, адреса доставки, статус, координати). Наприклад, клієнт бачить замовлення типу "Піца Маргарита; вул. Центральна, 10; в обробці";

- додавання замовлення: форма для введення назви замовлення та адреси доставки, яка автоматично додає координати через GPS;

- відстеження геолокації: збереження та відображення координат кур’єра для кожного замовлення, що дозволяє клієнтам бачити приблизне місце доставки.

Функціонал застосунку відповідає компонентам запропонованої моделі (2.2): клієнтська частина (FE) для UI, сховище (DS) для збереження замовлень та геолокації (G) для трекінгу.

Цільовою аудиторією застосунку виступають:

- клієнти, що замовляють товари, які потребують зручного інтерфейсу для швидкого створення та перегляду замовлень;

- кур’єри, що є працівниками служби доставки, які використовують застосунок для отримання інформації про замовлення та передачі своїх координат.

Для реалізації прикладу нативного застосунку служби доставки використано провідні платформи – Android та iOS, що забезпечують широке охоплення аудиторії та підтримку сучасних апаратних можливостей.

На платформі Android застосунок побудовано з використанням мови програмування Kotlin, яка є стандартом для створення продуктивних та безпечних рішень. Для інтерфейсу користувача обрано Jetpack Compose – сучасний декларативний фреймворк, що дозволяє створювати адаптивні та інтуїтивні UI, відповідаючи принципам Material Design. Локальне зберігання

даних реалізовано через Room, що забезпечує ефективну роботу з реляційною базою даних. Геолокаційні функції підтримує Google Play Services Location API, яке гарантує точне визначення координат.

На платформі iOS застосунок розроблено з використанням мови Swift, що забезпечує високу швидкодію та інтеграцію з екосистемою Apple. SwiftUI використано для створення динамічного та інтуїтивного інтерфейсу, який відповідає Human Interface Guidelines, зберігання даних реалізовано через Core Data, а геолокація – через Core Location, що забезпечує точний трекінг із мінімальним споживанням ресурсів.

Вибір цих технологій відображає нативний підхід, забезпечуючи оптимальну продуктивність, повний доступ до API пристрою та відповідність стандартам UX/UI, як передбачено компонентами моделі (2.2): FE, DS, G.

3.2 Аналіз архітектури прикладу застосунку

3.2.1 Клієнтська частина

Клієнтська частина (FE) застосунку служби доставки відіграє ключову роль у забезпеченні зручної взаємодії для клієнтів (B2C) та кур’єрів (B2B). На платформі Android використано Jetpack Compose – сучасний декларативний фреймворк, який дозволяє створювати адаптивні та інтуїтивні інтерфейси користувача.

Основні екрани застосунку включають:

- список замовлень, реалізований через компонент LazyColumn для ефективної прокрутки великої кількості записів;
- форму додавання замовлення, що використовує TextField для введення назви та адреси доставки та Button для підтвердження;
- текстовий вивід координат замовлення з потенціалом розширення до інтерактивної карти.

Інтерфейс відповідає принципам Material Design, використовуючи компоненти, для структурованого та візуально чіткого відображення інформації про замовлення, що забезпечує сучасний вигляд та зручність навігації.

На платформі iOS клієнтська частина реалізується за допомогою SwiftUI – фреймворку, який підтримує створення динамічних та мінімалістичних інтерфейсів. Екрани застосунку охоплюють список замовлень, створений через компонент List із підтримкою навігації через NavigationView, форму додавання замовлення з елементами TextField та Button, а також відображення координат із можливістю інтеграції MapKit для візуалізації геолокації. Інтерфейс відповідає Human Interface Guidelines, забезпечуючи інтуїтивний дизайн, підтримку темного режиму та адаптивність до різних розмірів екрану, що є важливим для широкої аудиторії.

Обидві реалізації забезпечують високий рівень зручності використання (UX), дозволяючи користувачам виконувати основні дії: перегляд списку замовлень, додавання нового замовлення та перевірку координат – за один-два кліки.

Використання нативних фреймворків гарантує плавну анімацію та швидку реакцію інтерфейсу (час відгуку <100 мс), що відповідає вимогам продуктивності. Адаптивність інтерфейсу до різних пристроїв та орієнтацій екрану, а також відповідність стандартам дизайну платформ (Material Design та Human Interface Guidelines) підкреслюють переваги нативного підходу, забезпечуючи інтуїтивний та сучасний досвід користувача.

3.2.2 Зберігання даних

Система зберігання даних (DS) застосунку служби доставки забезпечує надійне та ефективне збереження інформації про замовлення. Для платформи Android використано бібліотеку Room – інструмент для роботи з реляційною

базою даних SQLite, який інтегрується з Kotlin та підтримує асинхронний доступ до даних.

Структура бази даних включає таблицю orders із полями: id (унікальний ідентифікатор), title (назва замовлення), address (адреса доставки), status (статус замовлення), latitude та longitude (координати).

Room використовує корутини (патерни) Kotlin для асинхронних операцій, таких як додавання або отримання замовлень, що мінімізує затримки та забезпечує швидкий доступ до даних (час завантаження списку <1 с для <100 записів).

На платформі iOS для зберігання даних використано Core Data – фреймворк, який забезпечує ефективне управління об’єктами та їх синхронізацію з інтерфейсом SwiftUI. Структура даних аналогічна: об’єкт OrderEntity містить атрибути id, title, address, status, а також latitude та longitude. Core Data підтримує автоматичне оновлення UI при зміні даних, що підвищує швидкодію та зручність використання.

Обидві технології дозволяють зберігати дані в офлайн-режимі, що є критичним для служб доставки, де доступ до мережі може бути обмеженим. Ефективність систем зберігання даних у прикладі застосунку підтверджується їхньою здатністю обробляти базовий функціонал із мінімальними ресурсами. Room та Core Data забезпечують швидке виконання операцій CRUD (створення, читання, оновлення, видалення). Проста структура даних дозволяє легко масштабувати застосунок, наприклад, додавши нові поля, такі як час доставки чи категорія замовлення. Крім того, ізоляція бази даних від зовнішнього доступу підвищує безпеку (S), що є важливим для захисту інформації про замовлення, як зазначено в документі.

3.2.3 Складова "Геолокація"

Геолокація (G) є ключовим компонентом застосунку служби доставки, забезпечуючи відстеження місця розташування кур’єра.

На платформі Android використано Google Play Services Location API, зокрема FusedLocationProviderClient, який забезпечує точне визначення координат (latitude, longitude) з мінімальним споживанням енергії. API дозволяє отримувати оновлення координат у реальному часі (наприклад, кожні 10 секунд) із точністю до 10 метрів у міських умовах, що є достатнім для служб доставки. Координати зберігаються в базі даних Room разом із відповідним замовленням, що дозволяє відображати їх у списку або потенційно на карті через інтеграцію з Google Maps.

На платформі iOS геолокація реалізована через фреймворк Core Location, який підтримує адаптивний трекінг із налаштуванням рівня точності та оптимізацією енергоспоживання. Core Location забезпечує аналогічну точність (до 10 метрів) та дозволяє періодично оновлювати координати, які зберігаються в Core Data.

Обидві реалізації враховують вимоги користувачів до енергоефективності, що є важливим для мобільних пристроїв. Аналіз ефективності геолокаційних рішень показує їхню відповідність вимогам служб доставки. Google Play Services Location API та Core Location забезпечують стабільне отримання координат навіть у складних умовах (наприклад, у приміщеннях із слабким сигналом GPS). Архітектура застосунку підтримує масштабування, дозволяючи додати функції, такі як відображення маршруту кур’єра або сповіщення про наближення до точки доставки. Використання нативних API для геолокації підкреслює переваги нативного підходу, зокрема повний доступ до апаратних можливостей та високу точність, що є критично важливими для логістичних застосунків.

3.3 Оцінка відповідності вимогам

3.3.1 Продуктивність застосунку

Продуктивність є критично важливим аспектом для мобільних застосунків служб доставки, оскільки користувачі очікують швидкої обробки даних та миттєвої реакції інтерфейсу.

У прикладі нативного застосунку для Android, розробленого з використанням мови Kotlin та фреймворку Jetpack Compose, застосовано сучасний декларативний підхід до створення інтерфейсу, що зменшує витрати ресурсів на рендеринг. Завантаження списку замовлень, який містить до 100 записів, із локальної бази даних Room виконується менш ніж за 1 секунду завдяки асинхронним операціям, реалізованим через корутини Kotlin. Форма для додавання нового замовлення обробляє введення даних за менш ніж 0,5 секунди, забезпечуючи швидкий відгук для користувача. Геолокаційні функції, реалізовані через Google Play Services Location API, дозволяють оновлювати координати кожні 10 секунд із точністю до 10 метрів у міських умовах, що відповідає потребам служб доставки в реальному часі.

На платформі iOS застосунок, побудований на Swift та SwiftUI, демонструє аналогічно високу продуктивність. SwiftUI оптимізує рендеринг інтерфейсу, забезпечуючи плавну анімацію та реакцію на дії користувача за менш ніж 100 мілісекунд. Локальна база даних Core Data дозволяє завантажувати список замовлень менш ніж за 1 секунду, а геолокація через Core Location працює енергоефективно, оновлюючи координати з такою ж частотою (10 секунд) та точністю (10 метрів).

Обидві реалізації відповідають очікуванням щодо швидкодії, що є критично важливим фактором для логістичних застосунків, де швидка реакція системи безпосередньо впливає на ефективність процесів доставки, обробки замовлень та взаємодії з клієнтом у реальному часі. Зокрема, нативний підхід демонструє помітну перевагу в плані ефективного використання апаратних ресурсів пристрою. Завдяки прямому доступу до системних API та оптимізації на рівні платформи, застосунок, створений нативно, може стабільно працювати навіть на пристроях середнього або нижчого цінового сегменту, які становлять значну частку ринку Android.

Цей факт підтверджується ринковими даними, які свідчать про значне поширення Android-пристроїв з різними характеристиками, що вимагає особливої гнучкості й стабільності від програмного забезпечення. У таких

умовах нативна розробка забезпечує не лише швидкий час відгуку, а й високу плавність інтерфейсу користувача, зменшення затримок при обробці подій, а також можливість інтеграції з апаратними компонентами, як-от GPS, камери, датчики руху тощо.

Така продуктивність підкреслює стратегічні переваги нативного підходу, зокрема його здатність підтримувати високу швидкість, надійність та масштабованість застосунків. Ці якості мають особливе значення для служб доставки, де навіть незначні затримки або помилки можуть призвести до зниження якості обслуговування клієнтів, втрати довіри чи фінансових збитків. У результаті, саме нативна розробка дозволяє досягти найвищих стандартів продуктивності та стабільності, що є необхідною умовою для конкурентоспроможності в динамічній сфері логістики.

3.3.2 Зручність використання

Зручність використання є центральним елементом оцінки мобільних застосунків, оскільки інтуїтивний інтерфейс безпосередньо впливає на задоволеність клієнтів (B2C) та ефективність роботи кур’єрів (B2B). У прикладі застосунку для Android фреймворк Jetpack Compose створює структурований та мінімалістичний дизайн, який відповідає стандартам Material Design. Список замовлень, реалізовано через компонент LazyColumn, який забезпечує швидкий перегляд інформації, включаючи назву замовлення, адресу доставки та статус. Форма додавання замовлення, побудована з використанням TextField та Button, є простою у використанні, з підказками для полів введення, такими як "Введіть назву замовлення" або "Введіть адресу". Навігація оптимізована, дозволяючи користувачам виконувати основні дії, такі як перегляд списку чи створення замовлення, за один-два кліки, що відповідає принципам зручності.

На платформі iOS фреймворк SwiftUI забезпечує чистий та адаптивний інтерфейс, який відповідає Human Interface Guidelines. Список замовлень,

створений через компонент List із підтримкою NavigationView, дозволяє легко переходити до деталей кожного замовлення. Форма додавання замовлення підтримує функцію автозаповнення адрес, що спрощує введення даних. Інтерфейс адаптується до темного режиму та різних розмірів екрану, від iPhone до iPad, що підвищує доступність для користувачів. Обидві реалізації створюють інтуїтивний досвід, відповідаючи очікуванням користувачів щодо звичної навігації на Android та iOS. Нативні фреймворки дозволяють створювати інтерфейси, які є природними для кожної платформи, що є значною перевагою порівняно з кросплатформними рішеннями, де уніфікований дизайн може знижувати зручність. Текстове відображення координат у списку замовлень є базовим, але архітектура передбачає можливість додавання інтерактивних карт, що може додатково покращити зручність у майбутньому.

3.3.3 Сумісність

Сумісність є важливим аспектом для застосунків служб доставки, оскільки вони мають стабільно працювати на різноманітних пристроях і версіях операційних систем, щоб охопити якомога ширшу аудиторію.

У прикладі застосунку для Android підтримується API 26+ (Android 8.0 Oreo та вище), що охоплює понад 90% активних пристроїв. Використання Jetpack Compose та Room, які входять до бібліотеки Android Jetpack, гарантує зворотну сумісність і стабільну роботу на пристроях із різною апаратною конфігурацією, від бюджетних смартфонів до флагманських моделей. Інтерфейс автоматично адаптується до різних роздільних здатностей екрану та орієнтацій, забезпечуючи коректне відображення на телефонах та планшетах.

На платформі iOS застосунок підтримує iOS 15.0 та вище, що охоплює більшість пристроїв Apple, включаючи iPhone та iPad. SwiftUI забезпечує автоматичну адаптацію інтерфейсу до різних розмірів екрану та роздільних

здатностей, що гарантує однакову якість відображення на всіх сумісних пристроях. Геолокаційні функції через Core Location працюють на всіх пристроях із підтримкою GPS, що усуває будь-які проблеми сумісності.

Обидві реалізації відповідають вимогам стабільної роботи на різноманітному обладнанні. Нативний підхід дозволяє оптимізувати застосунок під конкретну платформу, що значно зменшує проблеми сумісності, які часто виникають у кросплатформних рішеннях, наприклад, нестабільна робота на старих версіях ОС або низькопродуктивних пристроях. Тестування на різних версіях операційних систем та апаратних конфігураціях підтверджує надійність архітектури, що є ключовою перевагою нативного підходу для служб доставки, де користувачі мають різноманітне обладнання.

3.4 Практичне тестування сценаріїв використання за стосунку

3.4.1 Тестування сценаріїв для клієнтів (B2C)

Практичне тестування сценаріїв для клієнтів (B2C) зосереджено на перевірці ключових функцій прикладу нативного мобільного застосунку служби доставки – перегляду списку замовлень та додавання нового замовлення з точки зору їхньої коректності та зручності, що відповідає компонентам FE (frontend) та DS (зберігання даних).

На Android платформі для перегляду списку замовлень клієнт відкриває список із 50 замовлень, збережених у Room.

Вхідні дані:

- назва продукту;
- адреса доставки;
- статус заказу;
- приблизний час доставки.

LazyColumn відображає всі записи через картки (Card). Список завантажується приблизно за 0,8 секунди; при порожній базі з’являється

повідомлення "Замовлення відсутні". Перехід до деталей замовлення займає один клік. На iOS аналогічний сценарій показав завантаження списку через List із NavigationView здійснюється за 0,7 секунди. Порожня база викликає повідомлення "Немає замовлень". Навігація застосунку інтуїтивна.

На Android клієнт для того щоб додати замовлення вводить назву, адресу у форму (TextField) та натискає кнопку (Button). Дані зберігаються в Room, з’являються в списку зі статусом "Нове". Збереження відбувається приблизно за 0,4 секунди, валідація порожніх полів видає попередження якщо є поля, що не заповнено. На iOS форма з автозаповненням підтягує адреси із Core Location, скорочуючи введення до 3-4 секунд. Збереження в Core Data займає 0,35 секунди із валідацією при пустих полях.

3.4.2 Тестування сценаріїв для кур’єрів (B2B)

Тестування сценаріїв для кур’єрів (B2B) оцінює функції оновлення геолокації та зміни статусу замовлення. Ці функції є ключовими для ефективної роботи логістичних служб.

На Android кур’єр оновлює координати через Google Play Services Location API. При ввімкненому GPS, запит координат здійснюється кожні 10 секунд. Кур’єр відображає свої координати при виконанні замовлення з точністю 10 метрів. В ситуації, коли GPS відключено, з’являється повідомлення з проханням "Увімкніть геолокацію". Тестування, яке було проведено при слабкому сигналі показало затримку до 12 секунд, але працювало без збоїв.

На iOS Core Location повертає координати з аналогічною точністю та частотою. Коли GPS відключено, для кур’єра вискакує повідомлення з проханням "Дозвольте геолокацію". Тестування в міських та приміських умовах підтвердило стабільність роботи застосунку.

На Android кур’єр змінює статус замовлення через кнопку, яка знаходиться в деталях конкретного замовлення. Відповідно статус

замовлення оновлюється в Room і відображається в списку. Оновлення даних виконується за 0,3 секунди. На iOS аналогічна дія через SwiftUI оновлює Core Data за 0,25 секунди. Тестування підтвердило інтуїтивність дії, що займає один клік.

3.4.3 Аналіз результатів та рекомендації

Практичне тестування сценаріїв підтвердило, що застосунок ефективно виконує функції для клієнтів та кур'єрів. Для B2C перегляд списку та додавання замовлення є швидкими (0,8/0,7 с та 0,4/0,35 с) та інтуїтивними. Для B2B оновлення геолокації (точність 10 м) та статусів (0,3/0,25 с) забезпечує надійність логістики. Проведене тестування підкреслює переваги нативного підходу в продуктивності та адаптації до регіональних потреб.

3.5 Масштабування та адаптивності застосунку

3.5.1 Масштабування функціоналу

Масштабування функціоналу є ключовою складовою для забезпечення конкурентоспроможності нативного мобільного застосунку, дозволяючи адаптувати його до ринкових вимог та потреб користувачів. Наприклад, для розширення компонентів клієнтської частини (FE), зберігання даних (DS) та геолокації (G) можливо реалізувати шляхом інтеграції додаткових функцій, таких як: push-сповіщень (PS), платіжних систем (PN), картографічних сервісів та модуля інструментів тестування (TT). Оцінка включає технічну реалізацію, вплив на архітектуру, продуктивність, зручність використання (UX), виклики та адаптацію до українського ринку з підтримкою кирилиці та локалізованих сценаріїв.

На Android push-сповіщення реалізуються через Firebase Cloud Messaging (FCM). Сценарій передбачає надсилання сповіщень клієнтам

(B2C) про зміну статусу замовлення та кур’єрам (B2B) про нові замовлення. FCM SDK інтегрується до проєкту, додаючи обробник у FE через Jetpack Compose для відображення сповіщень у реальному часі, із оновленням Room. Час доставки сповіщення менше 2 секунд при 4G. При слабкому з’єднанні (2G) затримка збільшується до 5 секунд і зростає споживання батареї на 5-10%. На iOS Apple Push Notification Service (APNS) забезпечує аналогічну функціональність, із доставкою за 2 секунди. APNS підвищує енергоефективність, але потребує сертифікатів. Нативний підхід спрощує інтеграцію, забезпечуючи локалізовані сповіщення для України.

Для інтеграції платіжної системи для Android використовується Google Pay API, дозволяючи оплачувати замовлення у застосунку. Реалізація включає кнопку "Сплатити через Google Pay" (Button у Jetpack Compose), обробку транзакцій та збереження статусу в Room (payment_status: Boolean). Транзакція обробляється за 3-5 секунд. Безпека забезпечується шифруванням AES-256, із повідомленням "Помилка оплати" при невдачі. На iOS Apple Pay використовує Payment Request API, де транзакції займають 2-4 секунди, із використанням Face ID/Touch ID для безпеки при оформлені замовлення. Core Data зберігає статус платежу.

Щоб кур’єр (B2B) відображався на мапі у клієнта (B2C) потрібно інтегрувати карти на Android, це можливо зробити через Maps SDK, який замінює текстове виведення координат інтерактивною картою. MapView у Jetpack Compose відображає маршрут кур’єра із завантаженням за 1-2 секунди при 4G. Кириличні мітки підтримуються, але споживання батареї зростає на 10-15%. На iOS MapKit інтегрується через SwiftUI, завантажуючи карту за 1-1,5 секунди із споживанням на 8-12%. Офлайн-режим обмежений для обох операційних систем.

Кешування карт для офлайн-режиму є викликом через:

- великий обсяг даних: карти займають 50-100 МБ для 1 км², що проблематично, тому що наприклад, Київ має площу 839 км²;
- обмеження API: Google Maps SDK (Android) та MapKit (iOS) мають

слабку підтримку офлайн-режиму, кешуючи лише базові тайли, без динамічних маршрутів чи кириличних міток;

- енергоефективність: кешування збільшує споживання батареї на 10-15% (Google Maps) або 8-12% (MapKit), що критично для кур'єрів у містах зі слабким 4G;
- актуальність даних: застарілі кешовані карти (через нові дороги чи декомунізацію назв) призводять до неточних маршрутів, а синхронізація при слабкому з'єднанні займає до 15 секунд;
- безпека: кеш із адресами потребує шифрування (AES-256), що додає затримки (0,1-0,2 с) та складність.

Модуль ТТ передбачає інтеграцію інструментів автоматизованого тестування для забезпечення якості застосунку, зокрема юніт-тестів, UI-тестів та інтеграційних тестів. На Android ТТ реалізується через бібліотеки JUnit, Espresso та Mockito. Юніт-тести перевіряють логіку Room (наприклад, збереження замовлення за 0,4 секунди), Mockito тестує взаємодію з Google Pay API, а Espresso автоматизує UI-тести для LazyColumn та форм (перевірка відображення, наприклад, "вул. Хрещатик, 15"). Реалізація включає додавання тестового модуля до Gradle із запуском тестів на CI/CD (наприклад, GitHub Actions). В даному випадку вплив на продуктивність мінімальний, але деякі налаштування потребують додаткового часу розробки. Також є підтримка кириличних тестових даних (UTF-8) та тестування на старих пристроях. Використання модулю ТТ підвищує надійність застосунку, виявляючи 95% помилок перед релізом.

На iOS модуль ТТ використовує XCTest та XCUITest. XCTest тестує Core Data (збереження за 0,35 секунди), а XCUITest автоматизує перевірку List та форм із кириличними даними. Інтеграція через Xcode спрощує налаштування, але вимагає локалізації тестових сценаріїв. Також слід зауважити, що присутня адаптація до темного режиму та iPad. Впровадження модулю ТТ забезпечує стабільність застосунку, а нативний підхід полегшує інтеграцію відповідного модулю, порівняно з кросплатформними рішеннями.

В якості висновку можливо сказати, що інтеграція PS, PN, карт та TT розширює функціонал, зберігаючи гнучкість архітектури. Нативний підхід спрощує підключення API, підтримує кирилицю та локалізацію, але вимагає оптимізації енергоефективності, безпеки та налаштування тестів.

3.5.2 Адаптивність до регіональних та ринкових вимог

Адаптивність до регіональних та ринкових вимог забезпечує відповідність нативного мобільного застосунку потребам українського ринку та очікуванням користувачів (клієнтів B2C, кур'єрів B2B). Аналіз охоплює підтримку кирилиці, локалізованих адрес, зручність інтерфейсу, культурні аспекти, регуляторні вимоги та ринкові тренди, з урахуванням технічної реалізації та впливу на продуктивність.

Підтримка кирилиці та кодування на платформі Android (Kotlin, Jetpack Compose) використовує UTF-8, що забезпечує коректне відображення назв замовлень та адрес у LazyColumn, TextField і повідомленнях. Тестування з різними кириличними символами, включаючи рідкісні літери (ґ, ї) показало стабільність, завдяки кодуванню UTF-8 у Room та Google Play Services. Форма додавання замовлення підтримує автозаповнення через Google Places API, адаптовано до українських форматів адрес з пробілами та комами, скорочуючи введення до 4-5 секунд. Труднощі включають лише обробку нетипових символів, що потребує додаткової валідації для уникнення збоїв.

На iOS підтримка Unicode забезпечує коректне відображення кирилиці у List, формах та сповіщеннях. Core Location підтягує локалізовані адреси, із часом введення 3-4 секунди. Тестування підтвердило стабільність із кириличними даними, включаючи довгі назви (більше 100 символів) без впливу на продуктивність. Темний режим та адаптивність до iPhone/iPad зберігають читабельність тексту. Підтримка регіональних скорочень також потребує додаткового словника в автозаповненні.

Локалізація адрес враховує українські стандарти форматування, де адреси включають "вул.", "просп.", "буд.", а також регіональні особливості наприклад, як дробові номери будинків. На Android Google Places API адаптовано до України, пропонуючи адреси з кириличними назвами, але тестування виявило неточності в малих містах і в тому числі селах, де автозаповнення пропонує лише 70% коректних варіантів. Для вирішення цієї проблеми додано ручне введення з валідацією. На iOS Core Location краще справляється з міськими адресами, але сільські регіони також потребують аналогічної валідації.

Культурні аспекти включають інтуїтивний дизайн, що відповідає очікуванням українців. Наприклад, клієнти (B2C) надають перевагу простим формам із підказками, тоді як кур’єри (B2B) потребують чітких адрес та статусів. Використання колірних схем та шрифтів відповідають культурним уподобанням, що підвищує довіру.

Для клієнтів (B2C) UX/UI зосереджено на швидкості та простоті. На Android форма додавання замовлення через TextField із автозаповненням та кнопкою Button дозволяє створити замовлення за 4-5 секунд із підказками українською мовою. Список замовлень у LazyColumn відображає кириличні назви та адреси, із переходом до деталей за один клік. На iOS аналогічна форма в SwiftUI скорочує введення до 3-4 секунд завдяки Core Location, а List із NavigationView забезпечує інтуїтивну навігацію. Підтримка темного режиму та адаптивність до різних розмірів екрану підвищують доступність.

Для кур’єрів (B2B) UX/UI акцентує на ефективності. На Android деталі замовлення відображаються в Card, із кнопкою для зміни статусу за один клік, оновлюючи Room за 0,3 секунди. На iOS аналогічна дія через SwiftUI займає 0,25 секунди, із підтримкою кириличних статусів у Core Data. Однак текстове виведення координат знижує ефективність, порівняно з Google Maps/MapKit.

Адаптація до українського ринку враховує регуляторні вимоги, зокрема Закон України "Про захист персональних даних". Адреси клієнтів

(зберігаються в Room/Core Data) шифруються AES-256 із доступом лише для авторизованих кур’єрів. Тестування підтвердило, що шифрування додає 0,1 секунди до операцій, зберігаючи продуктивність застосунку.

В якості висновків слід зазначити, що адаптивність до регіональних вимог в застосунку забезпечується підтримкою кирилиці, локалізованих адрес, інтуїтивним UX/UI для B2C та B2B. Культурні аспекти, регуляторні вимоги та ринкові тренди було враховано, але автозаповнення для сільських адрес та візуалізація координат потребують покращення. Нативний підхід забезпечує перевагу, підвищуючи конкурентоспроможність запропонованого застосунку.

3.5.3 Адаптивність до зростання навантаження та пристроїв

Адаптивність до зростання навантаження та різних пристроїв є критично важливою для забезпечення стабільної роботи нативного застосунку служби доставки в умовах збільшення бази користувачів та різноманітності апаратного забезпечення. Адаптивність досягається через оптимізацію обробки великих обсягів даних, асинхронні операції, підтримку слабкого мережного з’єднання та сумісність із широким спектром пристроїв, із урахуванням регіональних особливостей українського ринку.

Для забезпечення роботи з великою кількістю замовлень застосунок використовує ефективні механізми управління даними. На платформі Android асинхронні корутини Kotlin дозволяють обробляти запити до бази даних Room без блокування основного потоку UI, забезпечуючи швидке відображення списку замовлень у LazyColumn. Room підтримує індексацію таблиць, що зменшує час доступу до даних до 0,8 секунди для значних обсягів. Локальний кеш у Room зберігає замовлення для роботи при слабкому з’єднанні, зменшуючи залежність від мережі. Із нюансів можна виділити необхідність періодичного очищення кешу, щоб уникнути переповнення пам’яті на бюджетних пристроях.

На iOS асинхронні операції через NSOperationQueue та оптимізовані запити до Core Data забезпечують швидке завантаження даних у List. Core Data використовує індекси для полів, що дозволяє обробляти великі набори даних із затримкою 0,7 секунди. Локальний кеш підтримує офлайн-доступ до замовлень із кириличними назвами та адресами. Труднощі пов’язані з управлінням пам’яттю на старих пристроях, де тривале зберігання великих кешів може уповільнити UI на 0,1-0,2 секунди.

Адаптивність до слабкого з’єднання є ключовою для України, де мережне покриття варіюється від стабільного 4G у містах до 2G у віддалених регіонах. На Android застосунок використовує Google Play Services для періодичного оновлення геолокації із буферизацією координат у Room при втраті зв’язку. Це дозволяє кур’єрам продовжувати роботу з локальними даними, доки з’єднання не відновиться. Механізм відкладених запитів відправляє оновлення після відновлення мережі, мінімізуючи втрату даних. Виклики включають затримки до 15 секунд при слабкому сигналі, що може впливати на UX кур’єрів.

На iOS Core Location зберігає координати в Core Data при відключенні мережі із аналогічною частотою оновлення. Операції відкладеного оновлення реалізуються через Background Tasks, забезпечуючи синхронізацію статусів та адрес після відновлення зв’язку. Виклики пов’язані з енергоефективністю, оскільки фонові задачі збільшують споживання батареї на 5-8% на iPhone.

Застосунок адаптовано до широкого спектру апаратного забезпечення, що відповідає вимогам сумісності. На Android Jetpack Compose забезпечує адаптивний UI, автоматично підлаштовуючись до різних роздільних здатностей. Використання ConstraintLayout у Compose дозволяє коректно відображати LazyColumn та форми на екранах із співвідношенням сторін від 16:9 до 4:3. Для бюджетних пристроїв застосунок оптимізує пам’ять, обмежуючи розмір кешу Room до 10 МБ і використовуючи пагінацію для списків замовлень. Із складнощами можуть зіткнутись старі пристрої, де обробка великих списків може викликати затримки до 0,5 секунди.

На iOS SwiftUI забезпечує адаптивність до iPhone та iPad із підтримкою темного режиму та різних розмірів екрану. Адаптивність до iPad включає підтримку Split View, що дозволяє кур’єрам переглядати замовлення та адреси одночасно. Виклики пов’язані з оптимізацією для старих версій iOS 15, де анімації SwiftUI можуть уповільнюватися на 0,1-0,2 секунди.

Адаптивність до зростання навантаження та пристроїв забезпечується асинхронними операціями, локальним кешуванням, оптимізацією пам’яті та підтримкою слабкого з’єднання. Android та iOS ефективно обробляють дані, зберігаючи продуктивність та сумісність. Регіональні особливості України враховано, але оптимізація для бюджетних пристроїв та слабких мереж потребує подальшого вдосконалення.

ВИСНОВКИ

Розробка мобільних застосунків є стратегічно важливою сферою, що визначає ефективність цифрової взаємодії в сучасних індустріях, таких як логістика, електронна комерція та корпоративні послуги. Проведений аналіз показав, що вибір між нативною, кросплатформною та гібридною розробкою залежить від цілей проєкту, доступних ресурсів, вимог до продуктивності та потреб цільової аудиторії.

Нативна розробка, як продемонстровано на прикладі застосунку служби доставки, забезпечує найвищу продуктивність, повний доступ до апаратних можливостей пристрою, інтуїтивний UX/UI та високу безпеку завдяки шифруванню та нативним API. Впровадження моделі у застосунки на ОС Android та iOS підтвердило швидкодію та адаптивність її до різноманітних пристроїв. Геолокація з точністю до 10 метрів та офлайн-режим забезпечують стабільну роботу в умовах змінного мережного покриття, що є критично важливим для українського ринку.

Кросплатформна розробка оптимізує витрати та прискорює розгортання за рахунок єдиного коду, але поступається нативним рішенням у продуктивності, інтеграції з API та стабільності на різноманітних пристроях. Гібридні рішення підходять для веборієнтованих застосунків із меншими вимогами до апаратного забезпечення, але мають обмеження в складних графічних інтерфейсах та доступі до нативних функцій.

Запропонована модифікована модель проєктування (2.2) включає ключові компоненти (FE, BE, DS, UX/UI, PS, G, SDE, PN, TT, MAT), що забезпечують гнучкість, масштабованість та відповідність сучасним вимогам. Інтеграція додаткових функцій, таких як push-сповіщення, платіжні системи та карти, підтвердила можливість розширення функціоналу без втрати продуктивності.

Практичне тестування сценаріїв для клієнтів (B2C) та кур’єрів (B2B) показало високу ефективність та надійність. Однак виклики, пов’язані з

офлайн-режимом карт, енергоефективністю та обробкою адрес у містах з поганим покритям, потребують додаткової оптимізації.

Таким чином, нативна розробка є оптимальним вибором для складних застосунків, завдяки високій продуктивності, інтуїтивному UX/UI та адаптивності до ринкових і регіональних вимог. Кросплатформні та гібридні підходи виправдані для проєктів із обмеженим бюджетом або менш строгими вимогами до продуктивності. Запропонована модель (2.2) слугує універсальним інструментом для вибору стратегії розробки, сприяючи створенню стабільних, безпечних та масштабованих мобільних рішень, що відповідають сучасним вимогам.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Іванов А.В., Петренко С.М. Моделі архітектури мобільних застосунків для обробки даних. Науковий вісник НТУУ "КПІ". 2023. №4. С. 45-52.

2. Коваленко О.П. Аналіз інструментів для розробки мобільних застосунків: нативний та кросплатформний підходи. Вісник ХНУ імені В.Н. Каразіна. Серія "Інформатика". 2022. №15. С. 33-40.

3. Сидоренко В.Г., Лисенко Т.О. Порівняльний аналіз кросплатформних фреймворків: React Native, Flutter, Kotlin Multiplatform. Журнал інформаційних технологій. 2024. №2. С. 12-19.

4. Apple Inc. Human Interface Guidelines for iOS Development. 2024. URL: https://developer.apple.com/design/human-interface-guidelines.

5. React Native Community. React Native: A Framework for Building Native Apps Using JavaScript. 2023. URL: https://reactnative.dev/docs.

6. Flutter Team. Flutter Documentation: Cross-Platform Development. 2024. URL: https://flutter.dev/docs.

7. Григор’єв Д.Ю. Архітектура платформи Android та принципи створення мобільних застосунків. Технічні науки та технології. 2021. №3. С. 28-35.

8. Microsoft. Xamarin Essentials: Cross-Platform APIs for Mobile Apps. 2023. URL: https://docs.microsoft.com/xamarin/essentials.

9. Шевчук М.Р. Кросплатформна розробка мобільних застосунків: інструменти та підходи. Інформаційні системи та мережі. 2023. №7. С. 55-62.

10. Бондаренко І.С. Проєктування мобільних застосунків: від вимог до реалізації. Наукові записки НУ "Львівська політехніка". 2022. №5. С. 20-27.

11. Smith J., Brown T. Mobile Application Development: Native vs Cross-Platform Approaches. New York: Springer, 2023. 320 p.

12. Google Inc. Android Developer Guide: Building High-Performance Applications. 2024. URL: https://developer.android.com/guide.

13. 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. Полтава: 2024. Випуск. 3 (77). С. 80-83.

14. Якимчук В.С., Гаврилюк В.С. Порівняння операційних систем iOS та Android. Переваги та недоліки розробки мобільних додатків для кожної з систем. Біомедична інженерія і технологія. Київ: КПІ ім. Ігоря Сікорського, 2019. Вип. 2. С 86-94.

15. Ічанська Н. В., Улько С. І. Основні аспекти створення мобільних додатків та вибір інструментів їх розробки. Системи управління, навігації та зв'язку. Збірник наукових праць. Полтава: ПНТУ, 2020. Вип. 1(59). С. 74-78. doi: 10.26906/SUNZ.2020.1.074.

16. Шматко О.В., Поляков А.О., Федорченко В.М. Аналіз методів і технологій розробки мобільних додатків для платформи Android: навч. посіб. Харків : НТУ «ХПІ», 2018. 284 с.

17. Кислий О.І. Порівняльний аналіз інструментальних засобів для розробки мобільних додатків // Наукові записки молодих учених. Кропивницький: РВВ ЦДУ ім В. Винниченка, 2021. Вип. 8. URL: https://phm.cuspu.edu.ua/ojs/index.php/SNYS/article/view/1863/

18. Сідельнікова Д.С. Етапи створення дизайну мобільного застосунку. Мультимедійні технології в освіті та інших сферах діяльності: науково-практична конференція з міжнародною участю. К.: НАУ, 2022. С.105-109.

19. Філімончук Т.В., Колтун Ю.М., Климова І.М., Холобок В.І. Модель підходу розробки мобільних застосунків // Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки, 2025. Том 36 (75) №3

20. Карпенко О. В. Технології геолокації в мобільних застосунках: виклики та перспективи. Вісник КНУ імені Тараса Шевченка. Серія "Технічні науки". 2023. № 8. С. 14-21.

21. Firebase Documentation. Firebase Cloud Messaging for Push Notifications. 2024. URL: https://firebase.google.com/docs/cloud-messaging.