МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЗАПОРІЗЬКА ПОЛІТЕХНІКА»

Факультет комп'ютерних наук та технологій
Кафедра комп'ютерних систем та мереж

Пояснювальна записка

до дипломного проєкту (роботи)
магістра
(ступінь вищої освіти)

на тему **АНДРОЇД-ДОДАТОК ДЛЯ МАГАЗИНУ З ДОСТАВКОЮ**

Виконав: студент 2 курсу, групи КНТ-513м
спеціальності
123 Комп'ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп'ютерні системи та мережі

СОКОЛОВ М.О.
(ПРІЗВИЩЕ та ініціали)
Керівник ТЯГУНОВА М.Ю.
(ПРІЗВИЩЕ та ініціали)
Рецензент МАЛИЙ О.Ю.
(ПРІЗВИЩЕ та ініціали)

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій

Кафедра «Комп'ютерні системи та мережі»

Ступінь вищої освіти магістерський

Спеціальність 123 Комп'ютерна інженерія

(код і найменування)

Освітня програма (спеціалізація) «Комп'ютерні системи та мережі»

(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Зав. кафедри Кудерметов Р.К.

"____" _____ 2024 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

СОКОЛОВА Михайла Олександровича

(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Андроїд-додаток для магазину з доставкою

керівник проєкту (роботи) к. т. н., доцент, ТЯГУНОВА Марія Юріївна

(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом вищого навчального закладу від "18" жовтня 2024 року № 149

2. Строк подання студентом проєкту (роботи) 10 грудня 2024 року

3. Вихідні дані до проєкту (роботи) наявні методи створення додатку на платформі Android для реалізації функціоналу магазину з доставкою

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) Аналіз предметної області;
2) Проєктування дизайну та функціоналу застосунку;
3) Дослідження ефективного розподілу ресурсів;
4) Реалізація додатку

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Презентація до пояснювальної записки

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	ТЯГУНОВА М.Ю., к.т.н., доцент		
нормоконтроль	ЩЕРБАК Н.В., ст. викл.		

7. Дата видачі завдання 01.10.2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз предметної області, дослідження сучасних	10.10.2024 р.	
	тенденцій та визначення вимог до додатку		
2	Проєктування дизайну та функціоналу	18.10.2024 р.	
3	Дослідження методів розподілу замовлень	08.11.2024 р.	
4	Реалізація додатку	14.11.2024 р.	
5	Оформлення отриманих результатів у ПЗ	25.11.2024 р.	
6	Оформлення графічного матеріалу	01.12.2024 р.	
7	Оформлення допоміжного матеріалу	05.12.2024 р.	

Студент _____ **Михайло СОКОЛОВ**
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ **Марія ТЯГУНОВА**
(підпис) (ім'я, ПРІЗВИЩЕ)

ЗМІСТ

РЕФЕРАТ

ПЗ: 116 с., 30 рис., 2 табл., 16 джерел.

ДОДАТОК, ДОСТАВКА, ЖАДІБНИЙ АЛГОРИТМ, ІНТЕРНЕТ-МАГАЗИН, ІНТЕРФЕЙС, УГОРСЬКИЙ АЛГОРИТМ, FIREBASE DATABASE

Об’єкт дослідження – додатки для магазинів з доставкою, алгоритми розподілення навантаження.

Предмет дослідження – додаток на платформі Android мовою Java, жадібний та угорський алгоритми для розподілу замовлень між кур’єрами.

Метою магістерської роботи є задовольнити потреби споживачів та зробити замовлення доставки більш доступним для різних шарів населення за рахунок розробки зручного додатку на базі ОС Android.

У першому розділі проведено аналіз сучасних тенденцій у додатках для магазинів з доставкою, технологій, які необхідні для розробки, тощо.

У другому розділі проведено проєктування дизайну та функціоналу застосунку, виявлено найбільш оптимальний архітектурний паттерн, визначена структура баз даних та змодельовано алгоритм роботи застосунку.

У третьому розділі проводиться дослідження ефективного розподілу ресурсів, а саме найбільш оптимального розташування точок видачі для того, щоб доставка могла проводитися по всьому місту, а також дослідження алгоритмів, за якими будуть розподілятися замовлення між кур’єрами – жадібним алгоритмом та угорським.

У четвертому розділі проводиться повна реалізація додатку, інтерфейсу, створюється весь необхідний функціонал.

В результаті виконання роботи розроблено додаток для магазину з доставкою на платформі Android мовою Java з реалізацією найбільш оптимального алгоритму розподілу замовлень між кур’єрами.

ABSTRACT

Explanatory note to the master’s work: 116 p., 30 figures, 2 tables, 16 sources.

APPLICATION, DELIVERY, FIREBASE DATABASE, GREEDY ALGORITHM, HUNGARIAN ALGORITHM, INTERFACE, ONLINE STORE

Object of research - applications for delivery stores, load balancing algorithms.

The subject of the research is an Android application in Java, greedy and Hungarian algorithms for distributing orders between couriers.

The purpose of the master's thesis is to meet the needs of consumers and make delivery orders more accessible to different segments of the population by developing a convenient Android application.

The first chapter analyzes current trends in applications for delivery stores, technologies required for development, etc.

The second section deals with the design and functionality of the application, identifying the most optimal architectural pattern, defining the database structure, and modeling the application algorithm.

In the third section, we study the efficient allocation of resources, namely the most optimal location of delivery points so that delivery can be carried out throughout the city, as well as the algorithms that will distribute orders between couriers - the greedy algorithm and the Hungarian algorithm.

In the fourth chapter, the full implementation of the application and interface is carried out, and all the necessary functionality is created.

As a result of the work, an application for a store with delivery on the Android platform was developed in Java with the implementation of the most optimal algorithm for distributing orders between couriers.

ВСТУП

Магазини з доставкою стають все більш популярними і більшість звичайних супермаркетів та гіпермаркетів переходять на торгівлю своєю продукцією онлайн. Однак деякі магазини та заклади використовують сторонні компанії для забезпечення доставки до користувачів. Це збільшує собівартість таких послуг. Додаток Monosell є актуальним, тому що він орієнтований на зменшення собівартості товарів та підвищення доступності замовлення доставки для споживачів. Завдяки організації управління і контролю за допомогою мобільного додатку вдасться зробити зручний менеджмент та функціонал, у якому розбереться будь-яка людина.

Загалом додаток розрахований на ОС Android оскільки нею користується близько 71% користувачів мобільних платформ за даними ресурсу Statcounter. Моделі телефонів на базі операційної системи Android є більш доступними з урахуванням цінової політики, тому ними користується більшість споживачів. Початкова ціль – задовольнити потреби більшості клієнтів, а потім портувати додаток на ОС iOS для охоплення всього можливого ринку. Також не варто забувати що для зручнішого користування додатком потрібно підключити сервіси Google.

Метою магістерської роботи є задовольнити потреби споживачів та зробити замовлення доставки більш доступним для різних шарів населення за рахунок розробки зручного додатку на базі ОС Android.

Для досягнення поставленої мети необхідно виконати такі завдання:

- проаналізувати предметну область та технології розробки мобільних додатків;
- визначити вимоги до розроблюваного додатку;
- спроєктувати структуру бази даних та користувацький інтерфейс додатку;

- провести дослідження найбільш оптимальних алгоритмів для розподілу наявних замовлень між кур’єрами з урахуванням відстані, необхідної для завершення замовлення та завантаженості кур’єра;

- реалізувати програмне забезпечення для платформи Android з урахуванням вимог споживачів для забезпечення роботи магазину з доставкою без посередників та унікальним функціоналом.

Особливістю роботи є моделювання місцезнаходження пунктів видачі та зберігання товару для більш зручної та швидкої доставки до користувачів по всьому місту Запоріжжя, а також порівняння різних алгоритмів для розподілу замовлень між кур’єрами. Назва магазину – Monosell. Слоган: «Monosell – магазин у вашому телефоні».

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Наразі багато супермаркетів та гіпермаркетів пропонують своїм користувачам замовляти необхідну продукцію з функцією доставки, що однозначно робить зручним процес купівлі. Проблема полягає в тому, що більшість з таких мереж магазинів використовують сторонні сервіси для доставки, що в свою чергу призводить до збільшення витрат на замовлення. Такий фактор є не дуже зручним для середньостатистичного покупця. Також не можна проходити повз той факт, що деякі мережі магазинів використовують для надання подібних послуг лише сайт, не маючи при цьому мобільного додатку. Застосунок для смартфону є більш зручним варіантом, оскільки він забезпечує користувачам можливість швидкого замовлення будь-де. Такий застосунок зберігає вашу інформацію, що значно прискорює процес та дозволяє не витрачати час на повторний вхід абощо.

Оскільки додаток для магазину Monosell розрахований на задоволення потреб більшої частини користувачів, а за статистикою, в Україні близько 71% людей користуються платформою Android, логічним є обрання саме цієї операційної системи. Основні переваги ОС Android при розробці мобільного додатку полягають у її відкритості та гнучкості. Оскільки це платформа з відкритим кодом, розробники мають доступ до широкого спектра інструментів і бібліотек, що дозволяє створювати кастомізовані рішення для різних потреб. Android підтримує інтеграцію з різними хмарними сервісами, що є важливим для масштабованих додатків, таких як застосунок для магазину Monosell. Крім того, ОС Android має велике покриття серед користувачів у світі, що забезпечує широкий ринок для поширення. Система також добре підходить для роботи з багатозадачністю та пропонує високий рівень персоналізації інтерфейсу, що дозволяє налаштувати взаємодію додатка відповідно до вимог різних користувачів або ролей, як-от адміністратор, кур'єр чи покупець [16].

1.1 Дослідження сучасних тенденцій доставки продуктів

Для розуміння сучасних тенденцій в сфері магазинів з доставкою продукції варто розглянути наявні маркетплейси та їхні особливості.

Rozetka є одним із найвідоміших українських маркетплейсів, що починав з електроніки та техніки, але зараз пропонує практично всі товари – від продуктів харчування до меблів і одягу. Мобільний додаток Rozetka дозволяє користувачам переглядати каталог, купувати товари та отримувати їх з доставкою по всій Україні. Клієнти можуть обрати доставку кур'єром або самовивіз з точок видачі, розташованих у різних містах. Додаток також пропонує персоналізовані рекомендації, історію замовлень і знижки, зручні для постійних покупців.

Ключовими особливостями є:

- великий асортимент товарів;

- можливість вибору з кількох варіантів доставки (кур'єрська, самовивіз);

- мобільний додаток підтримує онлайн-оплату, відстеження замовлень та повідомлення про акції.

Мережа супермаркетів Fozzy Group (Сільпо, Le Silpo) представлена кількома додатками, які дозволяють купувати продукти з доставкою додому. Наприклад, додаток Сільпо дозволяє клієнтам переглядати актуальні акції та пропозиції, збирати віртуальні бонусні картки, створювати списки покупок і замовляти доставку. Le Silpo – преміум-версія цієї мережі, яка орієнтована на якісніші продукти та більш ексклюзивний асортимент.

Ключові особливості:

- можливість зручного пошуку товарів за категоріями або через сканування штрих-кодів;

- аерсоналізовані пропозиції та бонуси для постійних клієнтів;

- інтеграція з системою лояльності мережі (Fozzy Group);

- підтримка самовивозу або доставки додому.

Zakaz.ua – це агрегатор, що об'єднує кілька супермаркетів, таких як METRO, Auchan, Novus та інші. Він дозволяє користувачам здійснювати покупки у будь-якому з доступних магазинів і замовляти доставку прямо до дверей. Це зручний сервіс для тих, хто хоче купувати продукти та побутові товари з кількох різних магазинів в одному місці. Додаток надає можливість зібрати кошик із товарів і обрати магазин для здійснення покупки.

Ключові особливості:

- об'єднання кількох популярних супермаркетів у одному додатку;

- підтримка доставки продуктів додому з можливістю вибору часу доставки;

- можливість замовлення товарів зі складів супермаркетів із найкращими цінами;

- відстеження замовлень у режимі реального часу.

Glovo – це міжнародний сервіс доставки, який працює в Україні й покриває різноманітні сфери. Користувачі можуть замовляти продукти, готову їжу, ліки та інші товари. Хоча основний акцент зроблено на доставці їжі з ресторанів, через Glovo можна також замовити продукти з місцевих супермаркетів або товари з інших магазинів.

Ключові особливості:

- великий вибір товарів та послуг для доставки – від їжі до одягу;

- швидка доставка з ресторанів і магазинів;

- можливість замовляти різні товари за допомогою однієї платформи;

- відстеження кур'єра у режимі реального часу.

АТБ – це одна з найбільших мереж супермаркетів в Україні, яка пропонує власний мобільний додаток для онлайн-покупок. Додаток АТБ дає змогу переглядати актуальний асортимент продуктів, вибирати товари та замовляти їх із доставкою додому. Також є можливість обирати час доставки або самовивіз з найближчого магазину.

Ключові особливості:

- широкий вибір товарів з мережі АТБ;

- підтримка доставки додому або самовивозу;

- зручний інтерфейс для пошуку товарів та перегляду актуальних акцій.

Varus – ще одна українська мережа супермаркетів, яка запустила власний мобільний додаток для покупок онлайн. Вона пропонує широкий асортимент продуктів і товарів першої необхідності. Додаток підтримує доставку додому або самовивіз. Varus активно розвиває програму лояльності, надаючи користувачам знижки та персоналізовані пропозиції.

Ключові особливості:

- зручний каталог товарів з різними фільтрами;

- доставка по Україні з можливістю вибору часу;

- персоналізовані знижки та бонуси через програму лояльності;

- можливість самовивозу з магазинів мережі.

Для кращого розуміння основних потреб користувачів пропонуємо вам розглянути порівняльну таблицю з вищевказаними маркетплейсами.

Таблиця 1.1 – порівняльна таблиця маркетплейсів

Магазин/Додаток	Асортимент	Доставка	Самовивіз	Програма лояльності	Особливості
Rozetka	Універсальний: електроніка, одяг, продукти, побутові товари, меблі, тощо	По всій Україні, кур'єрська доставка, поштові служби	Так, пункти видачі в різних містах	Знижки, персоналізовані рекомендації	Найбільший маркетплейс в Україні з широким асортиментом
Fozzy Group (Сільпо, Le Silpo)	Товари преміум-класу (Le Silpo)	Кур'єрська доставк	Так, пункти самовивозу в супермаркетах	Бонуси для постійних клієнтів, карта лояльності	Висока якість продуктів, персоналізовані пропозиції
Zakaz.ua	Продукти харчування, побутові товари з супермаркетів METRO, Auchan, Novus	Кур'єрська доставка по містах, вибір часу доставки	Ні	Відсутня єдина програма лояльності	Можливість замовляти товари з різних супермаркетів

Продовження таблиці 1.1

Магазин/Додаток	Асортимент	Доставка	Самовивіз	Програма лояльності	Особливості
Glovo	Продукти харчування, готова їжа, ліки	Швидка кур'єрська доставка	Ні	Немає	Велика кількість категорій товарів, відстеження кур'єра в реальному часі
АТБ	Продукти харчування, побутові товари	Кур'єрська доставка по містах	Так, у мережі супермаркетів	Персоналізовані знижки	Широка мережа магазинів, акції на популярні товари
Varus	Продукти харчування, товари для дому	Кур'єрська доставка по містах	Так, у мережі супермаркетів	Програма лояльності, знижки	Персоналізовані пропозиції, зручна система фільтрів товарів

Як можна зрозуміти з цієї таблиці основними тенденціями додатків для магазинів з функцією доставки є великий асортимент продукції, персоналізовані пропозиції для постійних клієнтів, можливість самовивозу та швидка кур’єрська доставка з функцією відстеження. Всі ці аспекти потрібно враховувати для створення додатку який задовольнить потреби більшості користувачів.

1.2 Аналіз технологій для розробки мобільного додатку

Перед початком написання коду додатку для магазину з доставкою варто визначитися з декількома пунктами, які дозволять обрати найбільш підходящу структуру та побудувати алгоритм дій, що спростить весь процес розробки. Мова йде про визначення архітектури додатку, бази даних, що використовується, обрання IDE, мови програмування. Найперший пункт – мова програмування,

адже саме вона є основою для написання коду і в залежності від обраної мови потрібно встановити правильний IDE, фреймворки.

Kotlin — офіційна мова програмування для Android, яку Google оголосив такою у 2017 році. Вона є сучасною і потужною мовою, яка забезпечує розробникам більш лаконічний і виразний синтаксис у порівнянні з Java. Основна перевага Kotlin полягає в його здатності спрощувати код, усуваючи зайву громіздкість, яка часто притаманна Java. Крім того, Kotlin надає ефективні засоби для обробки помилок і уникнення NullPointerException через використання системи типів, яка робить обробку null-безпечною. Kotlin має повну сумісність з Java, тому розробники можуть використовувати наявні Java-бібліотеки та проєкти, поступово мігруючи до Kotlin без необхідності переписувати код [1].

Java залишається однією з найпоширеніших мов програмування для Android-додатків. Протягом багатьох років вона була основною мовою, яку використовували для створення Android-додатків, і має велику кількість бібліотек та інструментів, які полегшують розробку. Однією з переваг Java є її стабільність і широка підтримка в Android SDK, що дозволяє розробникам працювати з нею без зайвих проблем. Однак синтаксис Java є більш громіздким у порівнянні з Kotlin, що може ускладнювати читання і підтримку великого коду. Java також відома своєю потужною моделлю багатозадачності, що дозволяє ефективно працювати з асинхронними процесами [2].

C++ використовується у тих випадках, коли продуктивність має вирішальне значення, наприклад, для розробки ігор, графічних додатків або інших додатків, що потребують складних обчислень. В Android C++ можна використовувати для написання нативного коду за допомогою інструменту Android NDK (Native Development Kit). Ця мова надає розробникам більше контролю над управлінням пам'яттю, що дозволяє створювати високопродуктивні програми. Проте, C++ є складнішою мовою для написання та відлагодження, а також потребує інтеграції через JNI (Java Native Interface), що може збільшувати складність розробки.

C# — це потужна мова програмування, яка використовується у середовищі розробки Xamarin для створення кросплатформених додатків, що працюють як на Android, так і на iOS. Xamarin дозволяє розробникам писати код один раз і використовувати його для різних платформ, що значно знижує час та витрати на розробку. Крім того, C# є частиною екосистеми .NET, що надає доступ до великої кількості бібліотек та інструментів. Однак, оскільки Xamarin є кросплатформною технологією, вона може не завжди надавати такий же рівень доступу до нативних функцій Android, як це роблять Kotlin або Java.

Dart використовується разом із фреймворком Flutter, що дозволяє створювати кросплатформенні додатки для Android і iOS з єдиною кодовою базою. Flutter відомий своєю високою продуктивністю і можливістю створювати красиві користувацькі інтерфейси завдяки власному механізму рендерингу. Dart — мова, розроблена Google, яка має зрозумілий синтаксис і високу швидкість компіляції. За допомогою Flutter розробники можуть легко адаптувати свої додатки для різних платформ, не жертвуючи продуктивністю. Однак, як молодий фреймворк, Flutter може інколи мати проблеми з підтримкою специфічних можливостей Android.

Python, завдяки таким фреймворкам, як Kivy або BeeWare, також може використовуватися для розробки Android-додатків. Python відомий своєю простотою у вивченні і високою читабельністю коду, що робить його привабливим вибором для початківців розробників. Kivy — це кросплатформовий фреймворк, який дозволяє створювати мобільні додатки на Python. BeeWare надає інструменти для створення нативних додатків на Python для Android. Проте варто зазначити, що продуктивність додатків на Python зазвичай нижча, ніж у додатків, написаних на Kotlin або Java, тому Python найкраще підходить для прототипування або для додатків зі спрощеною логікою.

JavaScript — одна з найбільш поширених мов програмування для веб-розробки, але її також можна використовувати для створення Android-додатків за допомогою фреймворків React Native, Ionic або PhoneGap. React Native, розроблений Facebook, дозволяє писати кросплатформенні додатки для Android

та iOS за допомогою JavaScript. Ionic і PhoneGap використовують веб-технології для створення мобільних додатків, що дозволяє розробникам використовувати вже знайомі інструменти та бібліотеки. Можуть виникати проблеми з продуктивністю, особливо для складних додатків, через обмежену інтеграцію з нативними функціями Android.

Rust — це мова, яка швидко набирає популярність завдяки своїй продуктивності та безпеці пам'яті. Rust активно використовується для написання високопродуктивних частин додатків або системних модулів, що потребують контролю за управлінням пам'яттю і ефективністю. У контексті Android Rust може бути використаний для створення нативних модулів, аналогічно до C++, але з більш сучасними інструментами для забезпечення безпеки та уникнення помилок управління пам'яттю. Rust поки що не має такої великої спільноти, як Java або Kotlin, але його потенціал для створення продуктивних і безпечних додатків є значним [8].

Таблиця 1.2 – Порівняльна таблиця мов програмування на Android

Мова	Підтримка Android	Тип додатків	Продуктивність	Легкість навчання	Кросплатформеність	Особливості/Обмеження
Kotlin	Офіційна	Нативні	Висока	Легка (схожа на Java)	Обмежена (через Kotlin Multiplatform	Основна мова для Android, сумісна з Java
Java	Офіційна	Нативні	Висока	Середня	Ні	Широка екосистема, громіздкий код
C++	Підтримується через NDK	Нативні (з NDK)	Дуже висока	Висока складність	Ні	Висока продуктивність, складне використання

Продовження таблиці 1.2

Мова	Підтримка Android	Тип додатків	Продуктивність	Легкість навчання	Кросплатформеність	Особливості/Обмеження

C#	Через Xamarin	Кросплатформенні	Висока	Середня	Так	Спільний код для Android і iOS, залежність від Xamarin
Dart	Через Flutter	Кросплатформенні	Висока	Легка	Так	Потужний UI-фреймворк, єдина кодова база для iOS і Android
Python	Через Kivy, BeeWare	Кросплатформенні	Низька	Дуже легка	Так	Простий для прототипування, не для складних додатків
JavaScript	Через React Native, Ionic	Кросплатформенні	Середня	Легка	Так	Використовується для веб-технологій,
Rust	Підтримується	Нативні (з NDK)	Дуже висока	Складна	Ні	Безпечна робота з пам'яттю, для продуктивних додатків
Swift	Через Jetpack Compose Multiplatform	Кросплатформенні	Висока	Середня	Так	Основна мова для iOS, обмежена підтримка Android

Для розробки додатків на платформі Android існує кілька популярних середовищ розробки (IDE), які забезпечують необхідні інструменти та функціональність для програмування, тестування та налагодження додатків.

Одне з найвідоміших середовищ розробки для Android — це Android Studio, офіційна IDE, розроблена Google спеціально для створення Android-додатків. Вона базується на платформі IntelliJ IDEA та пропонує повний набір інструментів для розробників, включаючи графічний інтерфейс для роботи з

дизайном інтерфейсів, вбудований емулюючий пристрій для тестування додатків, інтеграцію з Git для керування версіями, а також можливість використання нативних бібліотек і SDK. Android Studio повністю підтримує мови Kotlin та Java, а також включає в себе інструменти для роботи з Android NDK для розробки на C++ [3].

Іншою популярною IDE є IntelliJ IDEA від компанії JetBrains. Хоча IntelliJ не є спеціалізованим середовищем для Android, як Android Studio, вона також пропонує можливість розробки Android-додатків. IntelliJ IDEA має потужну систему автозаповнення, інструменти рефакторингу та інтеграцію з Android SDK, що робить її зручною для розробки як на Kotlin, так і на Java. Розробники можуть використовувати IntelliJ IDEA для тих самих завдань, що й Android Studio, але з гнучкішими можливостями налаштування середовища [5].

Для тих, хто створює кросплатформенні додатки на C#, відмінним варіантом є Microsoft Visual Studio разом із розширенням Xamarin. Visual Studio дозволяє розробляти додатки для Android та iOS одночасно, використовуючи єдиний код на C#. Середовище включає всі необхідні інструменти для написання, тестування та відлагодження мобільних додатків, а також підтримку хмарних сервісів. Xamarin забезпечує доступ до нативних функцій Android через API, що робить Visual Studio чудовим вибором для розробників, які хочуть охопити кілька платформ одночасно.

Для розробки кросплатформених додатків на Flutter, використовується середовище Visual Studio Code або ж Android Studio з відповідним плагіном. Visual Studio Code — легке середовище розробки, яке відрізняється швидким завантаженням та широким вибором плагінів. Воно має підтримку таких мов, як Dart (для Flutter), Python, JavaScript, і дозволяє розробляти мобільні додатки для кількох платформ одночасно. Visual Studio Code є дуже популярним серед розробників, які шукають легке і швидке середовище з мінімальними ресурсними вимогами, проте для роботи з Android тут можуть знадобитися додаткові налаштування [9].

Ще однією IDE для кросплатформенної розробки є Eclipse з плагіном ADT (Android Development Tools). Хоча Eclipse раніше було основним середовищем для Android, його популярність значно знизилася після виходу Android Studio. Тим не менш, Eclipse продовжує використовуватись деякими розробниками, оскільки підтримує велику кількість плагінів і мов програмування, таких як Java, C++ та Python. Незважаючи на це, Eclipse поступається Android Studio у зручності та можливостях, тому його рідше обирають для сучасних Android-проєктів.

Android Studio є найкращим варіантом для розробки Android-додатків завдяки кільком ключовим перевагам, які роблять це середовище розробки зручним, потужним і функціональним для будь-якого типу проєктів.

Перша і головна причина — офіційна підтримка Google. Android Studio є розробленою і підтримуваною компанією Google IDE, що гарантує її оптимальну сумісність з Android SDK, NDK та іншими інструментами екосистеми Android. Це означає, що всі нові можливості, нові версії операційної системи, інструменти та API спочатку будуть доступні в Android Studio. Офіційна підтримка також забезпечує стабільні оновлення і виправлення помилок, що є важливим для підтримання сучасних і безпечних додатків.

Друге значення має повна інтеграція з Android SDK. Android Studio надає всі необхідні інструменти для розробки, налагодження та тестування додатків без необхідності додаткових налаштувань або встановлення плагінів. Це включає унікальні можливості, такі як вбудований емулятор Android, який дозволяє запускати додатки на віртуальних пристроях різних конфігурацій і версій Android. Емулятор швидкий і підтримує різні функції, як-от тестування додатків з різними роздільними здатностями екранів, версіями ОС, або симуляцію дій, таких як дзвінки та зміна місця розташування.

Ще однією перевагою є зручність для дизайну інтерфейсу користувача (UI). Android Studio включає візуальний редактор інтерфейсів, який дозволяє легко створювати макети додатків за допомогою drag-and-drop елементів. Розробники можуть переключатися між кодовим і візуальним виглядом, що

дозволяє прискорити процес створення інтерфейсу. Вбудовані функції автоматичної адаптації під різні розміри екранів і пристрої (Adaptive Layout) роблять роботу з дизайном гнучкішою і зручнішою.

Також важливою перевагою є потужна система налагодження та профілювання. Android Studio пропонує інтегровані інструменти для налагодження коду (debugging), аналізу продуктивності додатка (profiling) та моніторингу використання ресурсів, як-от оперативна пам'ять, процесор або батарея. Інструменти, як Android Profiler і Logcat, дозволяють розробникам відстежувати поведінку додатка в реальному часі, виявляти проблеми та оптимізувати додаток для кращої продуктивності [4].

Крім того, Android Studio надає підтримку для кількох мов програмування, включаючи офіційну для Android мову — Kotlin, а також Java і C++. Це дозволяє розробляти як стандартні нативні додатки, так і високопродуктивні програми з використанням Android NDK для C++.

Окрім цього, Android Studio має інтеграцію з системами контролю версій, як-от Git, що дозволяє легко працювати в командах і керувати версіями коду. Це спрощує процес злиття змін, дозволяє відстежувати історію проєкту і працювати над різними версіями додатка одночасно.

Нарешті, Android Studio надає підтримку для нових технологій, таких як Jetpack Compose — сучасний інструмент для створення інтерфейсів на основі декларативного підходу. Це значно спрощує створення UI-компонентів та забезпечує більш зручний і швидкий процес розробки інтерфейсів.

Завдяки своїй повній інтеграції з інструментами Android, сучасним функціоналом для розробки, потужній системі налагодження та підтримці нових технологій, Android Studio є найкращим варіантом для створення Android-додатків.

Варто зазначити що окрім IDE важливим елементом для створення додатку для магазину з доставкою є фреймворки, які включають в себе купу корисних інструментів.

Фреймворк для написання додатку на платформі Android — це набір інструментів, бібліотек і правил, що полегшують процес розробки додатків. Він надає розробникам готові рішення для типових завдань, таких як робота з користувацьким інтерфейсом, доступ до баз даних, робота з API та інтеграція з нативними можливостями платформи Android. Основна мета фреймворку — спростити і прискорити розробку додатків, забезпечуючи розробників вже готовими компонентами та архітектурними патернами.

Фреймворк зазвичай включає:

- набір готових бібліотек та компонентів: це можуть бути UI-елементи (кнопки, списки, навігаційні панелі), інструменти для обробки даних, робота з мережевими запитами, картами, GPS тощо;

- API для доступу до нативних функцій Android: фреймворк надає зручні методи для використання камер, сенсорів, систем сповіщень, GPS та інших функцій Android, що полегшує доступ до них без необхідності писати низькорівневий код;

- шаблони архітектури: багато фреймворків включають в себе архітектурні рішення (наприклад, MVVM або MVP), які допомагають структурувати додаток і забезпечують кращу підтримку і масштабованість проєкту;

- інструменти для кросплатформенності: деякі фреймворки (наприклад, Flutter, React Native) дозволяють створювати один код для додатків як на Android, так і на інших платформах (наприклад, iOS), що значно знижує витрати часу і ресурсів.

Фреймворк працює як основа, на якій будується додаток, надаючи розробникам структуру і спрощуючи взаємодію з платформою Android. Кожен фреймворк має свої особливості, переваги та обмеження, що робить його підходящим для певних типів проєктів.

Flutter є одним із найпопулярніших кросплатформенних фреймворків для розробки додатків на Android та iOS. Він був створений Google і дозволяє розробляти додатки на мові програмування Dart. Основна перевага Flutter полягає в тому, що він надає єдину кодову базу для обох платформ, що значно

скорочує час розробки та підтримки проєктів. Flutter використовує власний графічний движок для рендерингу інтерфейсів, завдяки чому забезпечується висока продуктивність і гнучкість в налаштуванні дизайну. Однією з головних особливостей фреймворку є інструмент Hot Reload, що дозволяє вносити зміни в код і миттєво бачити результат без перезавантаження програми.

React Native від Facebook також є потужним кросплатформенним фреймворком, який дозволяє створювати додатки для Android і iOS з використанням JavaScript та React. Основною ідеєю React Native є створення нативних компонентів, що дозволяє додаткам працювати швидко і без значних втрат продуктивності. Розробники можуть використовувати відому їм екосистему JavaScript для створення мобільних додатків, що робить цей фреймворк популярним серед веб-розробників, які хочуть розширити свою діяльність на мобільну платформу. React Native дозволяє використовувати нативні API Android через спеціальні містки (bridges), що забезпечує доступ до нативних функцій, таких як камера або геолокація.

Apache Cordova (раніше PhoneGap) є фреймворком для створення мобільних додатків за допомогою веб-технологій, таких як HTML, CSS і JavaScript. Він надає інструменти для створення кросплатформенних додатків шляхом обгортання веб-додатків у нативний контейнер, що дозволяє їм працювати на Android та інших платформах. Однією з основних переваг Cordova є простота у використанні для тих, хто вже знайомий з веб-розробкою, що дозволяє швидко створювати мобільні додатки. Cordova підтримує різні плагіни для доступу до нативних функцій Android, таких як файловий доступ, робота з камерою або сенсорами.

Ionic є ще одним фреймворком для кросплатформенної розробки на основі веб-технологій, але на відміну від Cordova, він має власний набір UI-компонентів і бібліотек для створення сучасних мобільних інтерфейсів. Ionic використовує Angular або інші популярні фреймворки JavaScript для створення додатків і надає компоненти, які легко адаптуються до зовнішнього вигляду нативних додатків Android та iOS. Завдяки цьому розробники можуть створювати додатки з

приємним і сучасним інтерфейсом, використовуючи вже знайомі їм технології веб-розробки. Ionic також дозволяє використовувати ті ж плагіни Cordova для доступу до нативних функцій платформи Android.

NativeScript є фреймворком, який дозволяє створювати додатки для Android і iOS на JavaScript або TypeScript з повним доступом до нативних API обох платформ. NativeScript забезпечує можливість безпосередньо працювати з нативними елементами Android, використовуючи JavaScript-код, що дозволяє досягати високої продуктивності додатків. Одна з ключових особливостей NativeScript полягає в тому, що він дозволяє писати єдиний код для логіки програми, але водночас забезпечує нативні інтерфейси для кожної платформи, що робить його цікавим вибором для тих, хто хоче створювати високоякісні кросплатформенні додатки.

Jetpack Compose є фреймворком, розробленим Google для спрощення процесу створення користувацьких інтерфейсів у додатках Android. Це декларативний UI-фреймворк, що дозволяє розробникам описувати інтерфейси користувача через функції, а не через традиційний XML. Jetpack Compose значно спрощує процес побудови інтерфейсів завдяки декларативному підходу, що дає змогу швидко і ефективно змінювати компоненти на екрані залежно від стану додатка. Compose інтегрується з іншими інструментами Jetpack, такими як ViewModel та LiveData, що робить його частиною єдиної архітектури для сучасних Android-додатків [6].

Кожен із цих фреймворків має свої сильні сторони і призначений для різних підходів до розробки додатків на Android. Для створення додатку на Android для магазину з доставкою, найкраще обрати три фреймворки, які забезпечать високу продуктивність, зручність у розробці, а також можливості для масштабування та кросплатформенності.

Першим фреймворком, який варто обрати, є Flutter. Він дозволяє створювати додатки як для Android, так і для iOS на основі єдиного коду. Для магазину з доставкою це ідеальний варіант, оскільки Flutter має відмінну продуктивність завдяки власному графічному движку. Це дозволяє створювати

швидкі і плавні користувацькі інтерфейси, що особливо важливо для додатків з великими обсягами даних, такими як каталоги товарів. Крім того, Hot Reload прискорить процес розробки, оскільки зміни в інтерфейсі або логіці можуть миттєво відображатися на симуляторі без перезавантаження додатка. Flutter також має добре розвинену екосистему плагінів для інтеграції з платіжними системами, картами і GPS, що важливо для реалізації функцій оплати та доставки.

Другий фреймворк, який варто розглянути, — це React Native. Цей фреймворк на основі JavaScript також дозволяє створювати кросплатформенні додатки для Android і iOS. Його перевага в тому, що він використовує нативні компоненти Android, що забезпечує швидку і плавну роботу інтерфейсу. React Native стане відмінним вибором для магазину з доставкою, оскільки має багату бібліотеку компонентів та інтеграцію з такими сервісами, як картографічні сервіси (Google Maps), пуш-сповіщення та нативні API для роботи з GPS, що важливо для функції відстеження замовлень у реальному часі.

Третім фреймворком, який буде корисним для цього типу додатку, є Jetpack Compose. Це сучасний Android-фреймворк від Google для створення нативних інтерфейсів користувача. Він базується на декларативному підході і дозволяє легко створювати складні інтерфейси з мінімальними зусиллями. Для магазину з доставкою Jetpack Compose є відмінним варіантом завдяки своїй гнучкості та інтеграції з іншими компонентами Jetpack, такими як Room (для баз даних) і Navigation (для керування переходами між екранами). Це дозволить створити додаток з високою продуктивністю та нативними функціями Android, особливо якщо ваша команда зосереджена на створенні саме Android-додатків і не потребує кросплатформенного рішення.

Визначившись з мовою програмування, IDE та фреймворками для зручності створення додатку, варто перейти до обрання архітектури, щоб остаточно визначитися зі структурою застосунку.

Перша архітектура — це MVC (Model-View-Controller). В цій архітектурі додаток поділяється на три основні компоненти: модель (Model), що відповідає

за роботу з даними; представлення (View), що відображає інтерфейс користувача; та контролер (Controller), який керує взаємодією між моделлю і представленням. У такій архітектурі представлення (інтерфейс користувача) не має прямого доступу до даних, тому контролер виступає посередником, реагуючи на дії користувача і змінюючи модель або представлення. В Android MVC може бути застосовано через фрагменти (Fragments) і активності (Activities) як частини View, тоді як моделі можуть представляти дані, отримані через базу даних або API.

Друга архітектура — MVP (Model-View-Presenter). В MVP також використовується розподіл на три основні компоненти: модель (Model), представлення (View) і презентер (Presenter). Однак тут презентер займає активнішу роль, керуючи всією логікою інтерфейсу і відділяючи її від View. View стає більш "пасивним", відображаючи тільки те, що йому передає презентер. Це дозволяє легше тестувати логіку додатка, оскільки View і Presenter чітко відокремлені один від одного. В Android MVP легко реалізується за допомогою активностей, фрагментів і спеціальних інтерфейсів для взаємодії з презентером, що дозволяє зменшити залежність між компонентами.

Третя архітектура — MVVM (Model-View-ViewModel). Ця архітектура є однією з найпопулярніших для Android-додатків завдяки підтримці Jetpack-компонентів. У MVVM модель (Model) містить дані та логіку бізнесу, представлення (View) відображає дані і реагує на взаємодії користувача, а ViewModel — це посередник, який керує станом інтерфейсу і логікою додатка. Однією з ключових особливостей MVVM є двостороння прив'язка даних (data binding), коли зміни в даних автоматично оновлюють інтерфейс, і навпаки. Це робить розробку швидшою та дозволяє легко масштабувати додаток. В Android Jetpack пропонує такі компоненти, як LiveData та ViewModel, які забезпечують чисту реалізацію цієї архітектури.

Існує також архітектура Clean Architecture, яка була запропонована Робертом Мартіном. Clean Architecture базується на принципах розділення відповідальностей і концентрується на тому, щоб кожен шар додатка мав чітко

визначені обов'язки. Основна ідея полягає в тому, що логіка додатка відокремлена від зовнішніх залежностей, таких як робота з базами даних або мережами. Зазвичай Clean Architecture складається з кількох рівнів: зовнішній шар відповідає за інтерфейси (UI та нативні компоненти Android), середній шар — за бізнес-логіку, а внутрішній шар містить моделі даних та логіку, незалежну від платформи. Це дозволяє легко модифікувати і тестувати додаток, оскільки кожен шар ізольований від іншого. Clean Architecture може використовуватися разом з іншими архітектурними підходами, наприклад, MVVM або MVP, що надає гнучкості при проєктуванні додатку.

Ще однією цікавою архітектурою є Unidirectional Data Flow (однобічний потік даних), який використовується у фреймворках, як-от Jetpack Compose або Redux. В цьому підході всі зміни в стані додатка проходять через один єдиний канал, що робить поведінку додатка передбачуваною і легкою для налагодження. Дані передаються вниз (від бізнес-логіки до інтерфейсу), а дії користувача піднімаються вгору, створюючи новий стан додатка. Такий підхід забезпечує стабільність і полегшує масштабування додатка, особливо коли він містить складні взаємодії та бізнес-логіку.

Для створення додатку для магазину з доставкою на платформі Android найкраще підходить архітектура MVVM (Model-View-ViewModel). Вона пропонує оптимальне рішення для побудови масштабованих, підтримуваних і реактивних додатків, що є ключовими вимогами для застосунку з великою кількістю даних, функціями замовлення, доставки та взаємодії з користувачем.

Ось основні аргументи, чому MVVM є найкращим варіантом:

- чітке розділення відповідальностей;
- прив'язка даних (Data Binding);
- модульність і масштабованість;
- підтримка асинхронних операцій;
- тестованість;
- підтримка Jetpack-компонентів.

MVVM розділяє логіку додатка на три окремі компоненти: модель, представлення та ViewModel. Модель відповідає за обробку даних (товари, замовлення, доставки), представлення відповідає за користувацький інтерфейс, а ViewModel управляє логікою і взаємодією між інтерфейсом та даними. Це допомагає підтримувати чистий код і спрощує підтримку та розвиток додатку.

Однією з основних переваг MVVM є можливість двосторонньої прив'язки даних. Це означає, що будь-які зміни в даних (нові товари, оновлення статусу замовлення) автоматично відображаються в інтерфейсі користувача, а дії користувача (зміна адреси доставки або вибір продукту) миттєво передаються назад в ViewModel. Такий підхід ідеально підходить для додатків з реальними даними, що часто змінюються.

MVVM легко масштабується, оскільки кожен компонент є окремою одиницею з чітко визначеними завданнями. Це дозволяє легко додавати нові функції або змінювати існуючі без ризику порушити загальну структуру додатку. У додатку для магазину з доставкою можна просто додати нові екрани для акцій чи інтегрувати нові методи оплати.

Магазин з доставкою передбачає взаємодію з сервером для отримання даних про товари, відстеження замовлень, сповіщення про статус доставки тощо. У MVVM використання таких компонентів, як LiveData та Coroutines, дозволяє ефективно обробляти асинхронні запити і оновлювати інтерфейс у відповідь на зміни в реальному часі.

Завдяки тому, що логіка додатку сконцентрована в ViewModel, її легко тестувати окремо від інтерфейсу. Це спрощує розробку автоматизованих тестів для бізнес-логіки, що є важливим для додатків з складними процесами замовлення, оплати і доставки.

MVVM інтегрується з основними Android-компонентами Jetpack, такими як ViewModel, LiveData, Room, Navigation. Це спрощує роботу з базами даних, управління навігацією між екранами, зберігання і відновлення стану додатку після зміни орієнтації або фонових операцій.

Зважаючи на ці переваги, архітектура MVVM ідеально підходить для додатку магазину з доставкою, оскільки вона забезпечує високу продуктивність, легкість в обслуговуванні та масштабованість. Це дозволяє легко інтегрувати функції, які важливі для такого типу додатків, як реальний час оновлень доставки, пошук товарів, фільтри, пуш-сповіщення та обробка замовлень.

Останній пункт який варто розглянути це база даних, що буде використовуватися, оскільки додаток для магазину з доставкою має взаємодіяти як з інформацією з акаунту користувача, так і з даними щодо товарів, виконаних та прийнятих замовлень, тощо.

Для написання додатків на платформі Android можна використовувати різні бази даних, залежно від специфіки проєкту. Одним із найбільш популярних варіантів для цього є база даних SQLite. Це легка реляційна база даних, яка вбудована в Android SDK і надає зручні інструменти для локального зберігання даних у додатку. SQLite є чудовим вибором для застосунків, які потребують офлайн-доступу до даних або зберігання інформації локально, наприклад, для кешування товарів, історії замовлень чи налаштувань користувача.

SQLite не вимагає окремого сервера, оскільки це файлова система баз даних, яка зберігає всю інформацію в одному файлі на пристрої. Вона забезпечує повний набір SQL-запитів для створення, читання, оновлення та видалення (CRUD) даних. SQLite підтримує транзакції та індекси, що робить її достатньо потужною для багатьох типів додатків. Важливо зазначити, що робота з SQLite на Android може бути оптимізована за допомогою використання Android Jetpack-компонента Room, який додає додатковий рівень абстракції і спрощує роботу з базою даних через об'єктно-реляційне відображення (ORM). Room також забезпечує кращу інтеграцію з LiveData і ViewModel, що робить його ідеальним рішенням для додатків, побудованих на архітектурі MVVM.

Ще однією перевагою Room є його можливість автоматично виконувати валідацію SQL-запитів на етапі компіляції, що знижує кількість помилок. Крім того, Room дозволяє легко мігрувати базу даних, що є важливим, коли потрібно оновити структуру бази даних після випуску додатка.

Для додатків, що використовують великі обсяги даних або потребують синхронізації з хмарою, можна використовувати Firebase Realtime Database або Cloud Firestore. Ці сервіси від Google забезпечують хмарне зберігання даних і дозволяють синхронізувати дані між користувачами в реальному часі. Firebase надає готове рішення для зберігання структурованих даних, що дозволяє уникнути написання власної серверної частини. Це корисно для додатків, які потребують роботи з інформацією в реальному часі, наприклад, для відстеження статусу замовлення в додатку магазину з доставкою [7].

Таким чином, для додатку на Android, який потребує локального зберігання даних, найкращим вибором буде SQLite разом із Room для зручної інтеграції та простоти роботи. Якщо ж потрібно забезпечити синхронізацію даних між кількома користувачами чи пристроями або підтримку реального часу, можна використовувати Firebase або Cloud Firestore.

Firebase є найкращим варіантом для бази даних у додатку для магазину з доставкою через кілька ключових переваг, які роблять його ідеальним для цієї категорії. Перш за все, Firebase забезпечує хмарне зберігання даних та синхронізацію в реальному часі, що є критично важливим для додатків з доставкою. Користувачі можуть відстежувати свої замовлення, перевіряти доступність товарів, змінювати замовлення чи адреси, і все це оновлюється автоматично і миттєво на всіх пристроях. Firebase також пропонує надійну інфраструктуру від Google, що гарантує високу продуктивність та стабільність навіть при великій кількості одночасних користувачів.

Однією з головних причин, чому Firebase є ідеальним вибором, є його безсерверна архітектура. Це означає, що тут не потрібно займатися налаштуванням і управлінням сервером або базою даних вручну. Firebase бере на себе всю серверну частину, включаючи масштабування і захист даних, що значно спрощує розробку додатка. Крім того, Firebase інтегрується з іншими сервісами Google, такими як Google Analytics, що дозволяє отримувати детальну інформацію про поведінку користувачів і оптимізувати додаток на основі цієї аналітики.

Firebase пропонує також широкий набір сервісів, які роблять розробку більш гнучкою і зручною для додатків магазину з доставкою. До них входять:

Cloud Firestore — більш потужний варіант для зберігання структурованих даних. Він забезпечує гнучкіший механізм для запитів та масштабування даних у великих проєктах. Це підходить для додатків з великими обсягами даних, які можуть зростати з часом, або для тих, хто хоче працювати з більш складними структурованими даними, такими як категорії товарів або історія замовлень.

Firebase Authentication — цей сервіс забезпечує просту інтеграцію автентифікації користувачів через різні методи, такі як електронна пошта, Google, Facebook, або інші соціальні мережі. Це значно спрощує процес входу для користувачів, що важливо для додатків з доставкою, оскільки дозволяє зберігати їх дані для швидких замовлень і перевірки статусу доставки.

Firebase Cloud Messaging (FCM) — сервіс для надсилання пуш-сповіщень, що є важливим для інформування користувачів про статус їхніх замовлень, спеціальні пропозиції, або оновлення доставки. Це дозволяє додатку швидко і ефективно підтримувати зв’язок з користувачами, навіть коли додаток неактивний.

Firebase Analytics — надає детальну аналітику про поведінку користувачів у додатку. Це дозволяє відслідковувати, які продукти користувачі переглядають найчастіше, які кроки вони виконують у процесі замовлення, і як можна покращити їхній досвід. Такі дані допомагають оптимізувати взаємодію з додатком і підвищити конверсію.

Firebase Crashlytics — сервіс для відстеження помилок і збоїв у додатку. Він автоматично збирає дані про помилки, що виникають на різних пристроях і платформах, дозволяючи швидко вирішувати технічні проблеми та поліпшувати стабільність додатку. Це особливо важливо для додатків з доставкою, де стабільність є критичним фактором.

З огляду на вищезгадані сервіси, Firebase пропонує комплексне рішення для магазину з доставкою. Воно забезпечує реальний час синхронізації даних, просту автентифікацію, безсерверне виконання логіки і надійне управління

сповіщеннями. Це робить його ідеальним варіантом для додатка, який потребує стабільної та масштабованої інфраструктури для обслуговування користувачів і ефективної роботи з доставками та замовленнями.

1.3 Визначення вимог до додатку

Цільова аудиторія додатку для магазину з доставкою має чіткі потреби, які визначають очікувані функції, зручність використання та загальну продуктивність додатку. Основні потреби включають простоту використання, швидкість обслуговування, широкий вибір товарів, надійність доставки та можливість відстеження замовлень у реальному часі. Детальний розгляд цих потреб допомагає краще зрозуміти, що саме має забезпечити додаток для магазину з доставкою:

Користувачі очікують, що додаток буде інтуїтивно зрозумілим, з мінімальними труднощами при навігації. Важливо забезпечити просту реєстрацію та авторизацію, швидкий доступ до продуктів, чітко структуровані категорії товарів і легку систему пошуку.

Цільова аудиторія хоче мати доступ до різноманітного асортименту товарів, включаючи свіжі продукти, техніку, одяг та інші категорії. Важливим аспектом є точна інформація про наявність товарів на складі. Вони не хочуть витрачати час на замовлення товарів, які не доступні для доставки. Тому інтеграція в реальному часі з базою даних щодо наявності продукції є ключовою потребою.

Користувачі бажають отримати свої товари якомога швидше і в узгоджений час. Тому важливим є розумний підхід до логістики, з поділом доставки на три рівні (близька, середня та далека). Можливість вибору часу доставки (як термінової, так і запланованої) дозволяє користувачам планувати свої замовлення відповідно до власного розкладу.

Сучасні користувачі цінують можливість контролювати процес доставки від початку до кінця. Вони хочуть бачити інформацію про те, коли замовлення було оформлене, передане на склад, у дорозі, і скільки часу залишилося до прибуття. Ця функція створює відчуття контролю і впевненості у своєму замовленні. Інтеграція з картами, які показують маршрут кур’єра, та пуш-сповіщення щодо статусу доставки значно покращують користувацький досвід.

Користувачі очікують, що додаток буде підлаштовуватися під їхні інтереси, пропонуючи персоналізовані рекомендації, знижки чи акції на основі їхніх попередніх замовлень або вподобань. Система повинна аналізувати покупки користувачів і пропонувати відповідні продукти або знижки.

Користувачі хочуть мати можливість швидко залишати відгуки про товари або доставку, а також отримувати своєчасну допомогу від служби підтримки у разі виникнення проблем. Інтеграція чату з підтримкою або можливість швидко зв'язатися з оператором допоможе вирішувати будь-які питання, пов'язані із замовленням, що підвищить лояльність.

Офлайн-доступ до історії замовлень та інформації про продукти. Часто користувачі хочуть мати можливість переглянути попередні замовлення або інформацію про продукти навіть без підключення до Інтернету.

Для користувачів важливо, щоб їх особисті дані та інформація про платежі були захищені. Інтеграція надійних технологій шифрування та відповідність стандартам безпеки допоможе запевнити користувачів у безпечності додатку.

2 ПРОЄКТУВАННЯ ДИЗАЙНУ ТА ФУНКЦІОНАЛУ ЗАСТОСУНКУ

Важливо розуміти, що жоден застосунок не може нормально функціонувати без продуманої архітектури та функціоналу. Водночас, не можна допустити складності в навігації для покупців, адже мало хто захоче користуватись додатком, в якому він нічого не розуміє. Не варто нехтувати і

дизайном, оскільки він відіграє важливу роль у привабленні потенційних клієнтів. Функціонал застосунку передбачає створення аккаунту користувача, після чого взаємодія з додатком поділяється на чотири різні ролі – «Покупець», «Адміністратор», «Кур’єр» та «Підтримка».

У першому випадку користувач матиме доступ до перегляду асортиментів товарів, їх замовлення, перегляду меню з гарячими стравами, відстеження кур’єру, налаштувань акаунту та окремого вікна для спілкування з представником служби підтримки.

У разі ролі «Адміністратор» користувач матиме доступ до редагування наявного списку товарів, додавання нової продукції, редагування бази даних, зміни ролі користувачів «Кур’єр» та «Підтримка». Для того, щоб захистити доступ до такого функціоналу від небажаних дій з боку звичайних покупців адміністратору потрібно ввести спеціальний код, який надається магазином.

Функціонал кур’єра – це карта з помітками найближчих до нього замовлень та точні дані про замовлення. Йому надається точка, до якої буде проводитися доставка продукції, після чого будується маршрут. В залежності від типу транспорту кур’єра йому буде доступніша ширша, або ж вужча зона міста з наявними замовленнями.

«Підтримка» матиме доступ до списку з діалогами користувачів, які очікують відповіді, при обранні вона отримає доступ до імені користувача, та замовлення за яким у нього виникло питання чи скарга.

У всіх випадках користувач матиме можливість перемикатися на режим «Покупець» для замовлення продукції, що є доволі зручним.

2.1 Вимоги до функціоналу додатку на основі потреб користувачів

Для того щоб створити зручний додаток, який задовольнить потреби користувачів потрібно враховувати їхні інтереси. Основним функціоналом додатку для покупців є:

- реєстрація нових користувачів та зручний вхід для вже зареєстрованих клієнтів;

- збереження інформації та налаштувань користувача у хмарній базі даних, можливість увімкнення зручних сповіщень;

- зручний перегляд каталогу товарів з можливістю зміни представлення як у вигляді списку, так і у вигляді плиток, сортування за певними критеріями;

- можливість перегляду фото товару, детальної інформації про нього з урахуванням актуальної ціни, наявності та акційних пропозицій;

- зручне замовлення в один клік;

- можливість написання відгуку про товар;

- бокова панель з наявними вкладками, анімований перехід між ними;

- самостійне обрання точок видачі або доставки;

- можливість замовлення певних гарячих страв, які готуються на кухні в точках видачі. Альтернатива – кнопка, яка додає у кошик інгредієнти у необхідних пропорціях для самостійного приготування за бажанням клієнта зі вказання рецепту;

- відстеження кур’єру та можливість спілкування з ним за допомогою внутрішнього чату;

- вкладка з історією покупок для правильного розподілення коштів та покращення фінансової грамотності споживача;

- вкладка з актуальними акційними пропозиціями;

- наявність закріпленої кнопки виклику чату зі службою підтримки для надання скарг, пропозицій або ж запитань.

Варто не забувати, що користувачі додатку мають різні ролі, тому важливо розглянути функціонал і в контексті кур’єрів. Основним функціоналом застосунку для кур’єрів є:

- збереження функціоналу для звичайного покупця, з можливістю перемикання між ролями;

- при переході на роль «Кур’єр» залишаються лише дві вкладки. Перша – карта з замовленнями. Друга – налаштування та зміна інформації;

- на карті в залежності від обраного при реєстрації виду транспорту відображаються наявні замовлення. При натисканні на точку кур’єр отримує інформацію про замовлення та підтверджує свій намір доставити товари, після чого на карті відображається найбільш оптимальний шлях або ж відхиляє його. У разі підтвердження точка зникає у інших кур’єрів;

- після підтвердження початку доставки між клієнтом та кур’єром з’являється можливість ведення чату для уточнення запитань;

- на вкладці налаштування кур’єр може змінити інформацію щодо свого транспортного засобу, а отже збільшити або скоротити можливий радіус пошуку замовлень;

- після доставки кур’єр та клієнт мають підтвердити успішну доставку товарів, про що надійде сповіщення на телефон.

Для представників служби підтримки також передбачений певний особливий функціонал який включає в себе:

- функції, доступні звичайним покупцям та можливість переключання на роль «Служби підтримки», яка має одну вкладку зі зверненнями покупців;

- обов’язково зробити інформацію про актуальний статус діалогу, тобто відмітки про прочитання повідомлення, наявність в мережі користувачів та можливість завершення діалогу з обох сторін;

- при обранні діалогу з користувачем представник служби підтримки отримує інформацію про товар або замовлення, з яким виникла проблема якщо така є.

Врешті решт користувачі з найбільшими повноваженнями мають статус «Адміністратор». Для них передбачений такий функціонал:

- редагування інформації про категорію, товар, характеристики, можливість повного видалення та додавання даних, категорій або товарів;

- налаштування акційних програм, додавання до них певних товарів, можливість редагування ціни та зміни умов акції;

- перегляд бази даних про представників служби підтримки та кур’єрів, перегляд відгуків про персонал з можливістю зміни ролі для певного користувача, розгляд заяв на отримання роботи за вищевказаними вакансіями.

Реалізація зберігання інформації та її правильне використання для задоволення вимог користувачів здійснюватиметься завдяки хмарній базі даних Firebase Realtime Database.

2.2 Архітектурний паттерн додатку MonoSell

Архітектура MVVM (Model-View-ViewModel) розділятиме код на три основні компоненти: Model, View та ViewModel. Кожен з них має свої обов’язки та взаємодіє із іншими компонентами для забезпечення чіткого розподілу відповідальності та полегшення підтримки додатку.

Model відповідатиме за управління даними, бізнес-логікою та взаємодією із зовнішніми джерелами (базами даних або API). У випадку Monosell класи Model включатимуть моделі товарів, користувачів, замовлень та доставки, тощо. Наприклад, клас Product буде містити інформацію про товар (назва, ціна, зображення), а Order – про замовлення (товари, статус доставки, клієнт). Ці класи є "чистими" і не мають жодної залежності від UI.

View відповідатиме за відображення даних користувачу та отримання вводу від нього. У Monosell це інтерфейси екранів, які будуть створені за допомогою Jetpack Compose або інших інструментів. Наприклад, екран кошика або профілю користувача реалізуватиметься через відповідні UI-компоненти. View лише відображатиме дані, які їй надаватиме ViewModel, і повідомлятиме про дії користувача.

ViewModel буде сполучною ланкою між Model та View. Вона міститиме логіку управління станом View та отримуватиме дані від Model, щоб передати їх у зручному для відображення форматі. Наприклад, клас CartViewModel

об’єднуватиме інформацію про додані в кошик товари, їх підсумкову вартість і статус готовності до оформлення замовлення.

Коли користувач виконуватиме дію у View, View передаватиме цю дію у ViewModel через виклик певного методу addProductToCart(). ViewModel оброблятиме дію, взаємодіятиме з Model для оновлення даних тобто додаватиме товар у список замовлення та оновлюватиме стан. Після цього ViewModel повідомлятиме View про зміну стану, що змушуватиме View оновити інтерфейс, тобто перерахувати товари в кошику.

Такий розподіл зумовлений потребою зробити код модульним, простим у тестуванні та підтримці. ViewModel буде ізольована від деталей відображення (UI) і може бути протестована без залежності від View, а Model зберігатиме логіку роботи з даними, незалежно від того, як вони використовуються. Модель архітектурного паттерну, яка проєктуватиметься в додатку MonoSell відображена на рисунку 2.1.

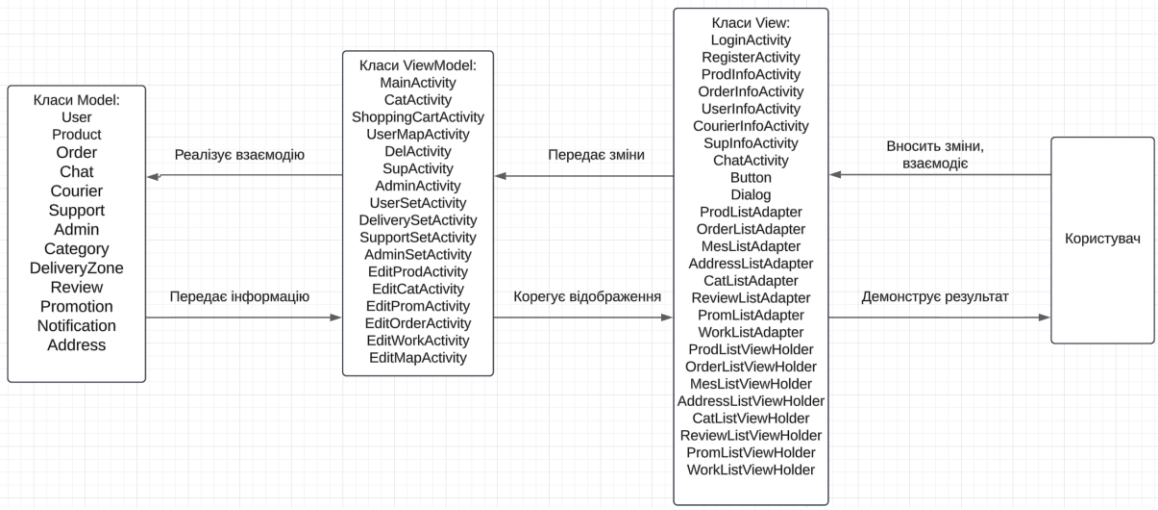

Рисунок 2.1 – Розподіл та взаємодія між класами за архітектури MVVM

Після визначення з основною архітектурою варто правильно розподілити класи. Отже, до класів Model відноситимуться Product, Order, User, Courier, Category, DeliveryZone, Review, Promotion, Notification, Address, Chat, Support, Admin.

Для прикладу клас Product буде моделлю будь-якого продукту, яка зберігатиме в собі інформацію щодо продукту, тобто його назву, ціну, категорію,

опис, зображення, категорію та ID, завдяки якому цей об’єкт буде зв’язуватися з іншими представниками Model.

Ця структура буде базовою для більшості класів Model, адже вона зберігатиме дані про той чи інший об’єкт у кількох змінних різного типу, в залежності від значення, ініціалізуватиме геттери та сеттери, які дозволять зручно змінювати стандартну інформацію про об’єкт через методи get() та set().

Незалежно від того, які дії робитиме користувач, додаватиме у кошик, робитиме замовлення, видалятиме з кошику чи щось подібне, інформація про товар не змінюватиметься, модель залишатиметься сталою та буде використовуватися і надалі, наприклад, для передачі продукції до списку замовлень. Аналогічно і модель замовлення також буде яскравим представником класу Model. Неважливо, чи буде користувач купувати товар, чи відмінить замовлення в останню мить, він в жодному разі не зможе додати характеристику до нього, окрім тієї яка була в нього від самого початку.

Взаємодія з класами Model полягатиме в тому, що коли користувач додаватиме товар до кошика, натискаючи при цьому кнопку, описану у класі ProdListAdapter, що буде відноситися до View, то вона оброблятиме цю дію і відправлятиме інформацію до ViewModel. ViewModel оброблятиме запит за допомогою методів, які будуть описані всередині Activity, на якому відбувається взаємодія, наприклад, методу addProductToCart(), описаному в класі CatActivity. Це дозволятиме об’єкту класу Product, що зберігатиме в собі інформацію про обрану продукцію ставати частиною списку об’єктів в класі Order, які зберігатимуть дані про замовлення і т.д.

Таке розмежування дозволить легко змінювати або розширювати функціонал без ризику порушити роботу всього додатку. Класи Model будуть ізольовані від UI та логіки ViewModel, що робить їх незалежними, багаторазово використовуваними та зручними для тестування.

Класи View в архітектурі MVVM відповідатимуть за відображення даних користувачу та отримання вводу від нього. Це UI-компоненти, які реалізуватимуться з використанням інструментів для створення інтерфейсу,

тобто, Jetpack Compose, XML макетів або бібліотек на кшталт Flutter. Вони будуть "тонкими" і не міститимуть бізнес-логіки — лише реагуватимуть на зміни стану, які передаватимуться з ViewModel, і транслюватимуть дії користувача назад у ViewModel. Яскравим прикладом класу View буде LoginActivity, функціонал якого передбачатиме створення акаунту користувача. Користувач вводитиме свої дані у поля класу EditText. У випадку якщо він ввів неправильні значення, то в спеціальне текстове поле tvErrorMessage виводитиметься інформація про помилку і її виправлення. Якщо ж все добре, то інформація з полів переводитиметься у String та передаватиметься в Firebase Authentication.

Взаємодія з класами Model полягатиме в тому, що до бази даних передаватиметься інформація про користувача User у вигляді рядків з його електронною поштою та паролем. Ці рядки будуть перевірятися у всіх зареєстрованих користувачів. У випадку якщо всі дані вірні, проводитиметься взаємодія з класом ViewModel – MainActivity, що відповідатиме за логіку входу до додатку. Якщо буде знайдена помилка, ViewModel оновлюватиме errorMessage, і TextView автоматично відображатиме повідомлення.

Аналогічно варто відзначити і те, що всі класи Adapter та ViewHolder також відноситимуться до View, адже вони забезпечуватимуть відображення інформації у вигляді списку, наприклад, список продукції на сторінці каталогу. Можливий варіант реалізації адаптеру представлено у додатку А.1.

Класи ViewModel в архітектурі MVVM виконуватимуть роль "посередника" між View (UI) та Model (дані). Вони відповідатимуть за бізнес-логіку, отримання даних з Model та надання їх View. ViewModel також зберігатиме стан екрану під час зміни конфігурації, наприклад, при повороті екрана [10].

У додатку MonoSell класи ViewModel міститимуть в собі логіку роботи додатку. Найпростіший приклад – користувач бачитиме перед собою список товарів, доданих у кошик, при цьому інформація про них зберігатиметься у базі даних і братиметься звідти ж. Буде прописана логіка відображення товарів у кошику, наприклад, за допомогою методу loadShoppingCart(), логіка

відображення загальної вартості за допомогою методу updateTotals() та оформлення замовлення і розташування його на мапі за допомогою методу placeOrder(). В першому методі буде проводитися очищення списку продукції методом clear() для того, щоб при кожному виклику екрану не відбувалося дублювання. Далі за допомогою циклу перевірятимуться всі продукти, які передаватимуться з гілки бази даних new, що належить гілці orders, яка пов’язана з кожним користувачем і якщо об’єкт класу Product не пустий, він додаватиметься до списку productsList методом add(), після чого ініціюватиметься оновлення адаптеру для коректного відображення та подальше оновлення остаточної ціни методом updateTotals(). В ньому підраховуватиметься кількість продукції, вона множитиметься на вартість за одиницю і заноситиметься у певну змінну totalAmount.

Як працюватиме цей ViewModel:

- інформація про кошик завантажуватиметься методом loadShoppingCart();
- загальна вартість автоматично обчислюватиметься через метод updateTotals();
- дані передаватимуться у View через об’єкт класу Adapter та RecyclerView.

2.3 Проєктування структури бази даних додатку MonoSell

Firebase Realtime Database – це хмарна база даних, яка дозволяє розробникам зберігати й синхронізувати дані між клієнтськими додатками в режимі реального часу. Вона особливо популярна в мобільних додатках, де потрібне швидке оновлення даних та злагоджена робота між різними користувачами. Основні особливості Realtime Database полягають в тому, що ця

БД зберігає дані в форматі JSON. JSON-структура має вигляд ієрархічного дерева, подібного до XML, і дозволяє зберігати як прості ключ-значення, так і вкладені об'єкти [13].

Firebase Realtime Database буде використовуватись в якості бази даних оскільки вона має декілька переваг над іншими БД, а саме:

- висока зручність та простота оформлення дозволяє прискорити процес адаптації даних для зберігання;

- дані автоматично синхронізуються з усіма підключеними клієнтами. Це означає, що при зміні даних один користувач може моментально бачити зміни, зроблені іншим, що робить зручною взаємодію між користувачами з різними ролями;

- Firebase Realtime Database дозволяє кешувати дані на пристрої користувача, що допомагає додатку працювати офлайн. Після відновлення з'єднання база автоматично синхронізує всі зміни;

- дані захищені правилами безпеки, які можна налаштовувати для кожного вузла бази даних. Це дозволяє обмежити доступ або керувати правами на читання й запис даних залежно від автентифікації користувача;

- Firebase Realtime Database добре працює для додатків невеликого та середнього масштабу, де потрібні часті оновлення даних;

- у Firebase Realtime Database підтримується проста фільтрація даних за допомогою методів для сортування, пошуку і фільтрації.

Також вибір саме цієї бази даних зумовлений тим, що такі додатки як Instagram Lite, Shazam, Uber Eats, Gameloft та Duolingo використовують її для збереження даних, чатів, профілю, коментарів і т.д.

Для правильної взаємодії між різними класами та методами потрібно правильно вибудувати зв’язок в межах бази даних.

Модель користувачу включає в себе такі поля, як «ID», «Ім’я», «Прізвище», «Електронна пошта», «Номер телефону», «Адреса», «Замовлення», «Чати» та «Налаштування». До перших п’ятьох полів заноситиметься дані в форматі String, які містять в собі текст.

До «Адреси» заноситиметься об’єкт класу Address, який включатиме в себе поля з вулицею, номером будинку та квартири для найбільш точної доставки.

Комірка «Замовлення» прийматиме в себе ID, яке пов’язуватиметься з певним об’єктом класу Order. Цей об’єкт включатиме в себе ID.

Воно необхідне для зв’язку з користувачем, дату створення, дату завершення в форматі String, та поле «Кур’єр» в яке заноситиметься ID кур’єра який працюватиме з замовленням.

Також є об’єкт класу Address, кількість продуктів, в який передаватиметься список з кількістю замовлених продуктів, та саме поле «Список продуктів», в яке зноситимуться ID наявних товарів у магазині MonoSell, які замовив користувач.

За цим ID можна буде знайти відповідний товар, інформація про якого міститиме у собі назву, опис, категорію у форматі String, а також ціну і вагу у вигляді змінної типу float. Все це дозволить структурувати підхід та добитися правильної взаємодії між даними, які будуть використовуватись в додатку MonoSell (рисунок 2.2).

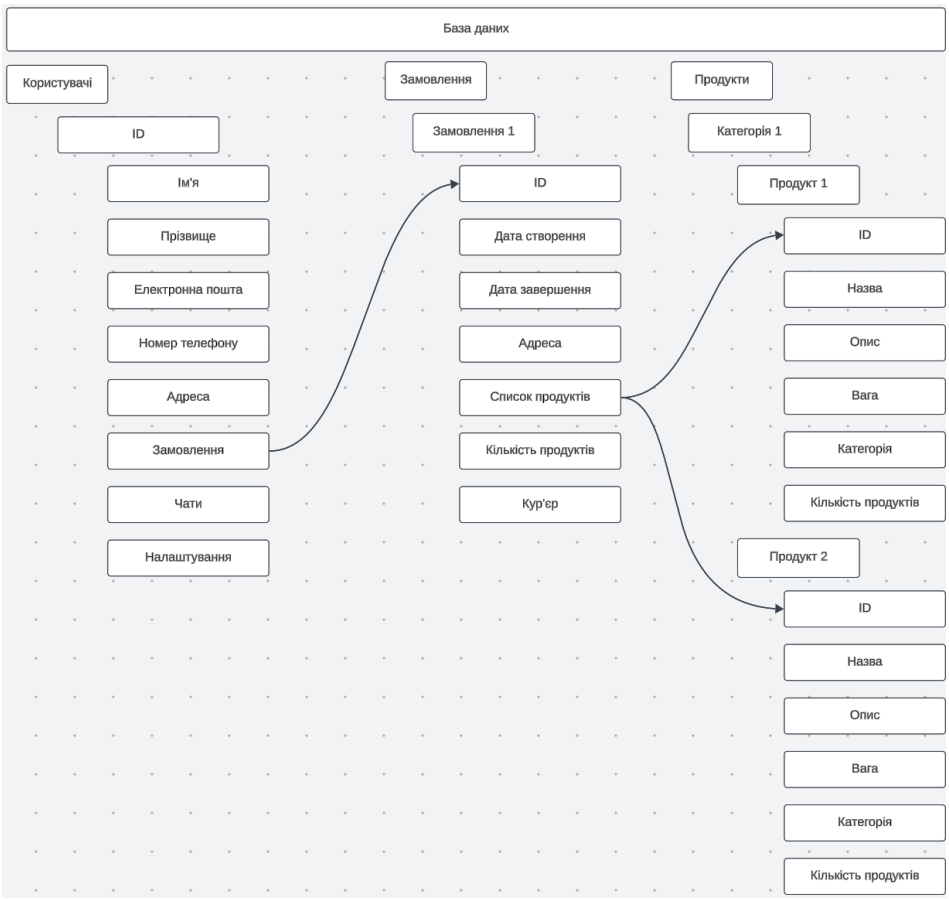

Рисунок 2.2 – Взаємозв’язок між користувачем, замовленням та продукцією

Варто зазначити, що в базі даних зберігатиметься також і кожен чат, який відбуватиметься між користувачем та кур’єром, або між користувачем та сапортом. Це дозволить у випадку появи негативних оцінок відстежити причину та належне виконання службових обов’язків. Кожному чату надаватиметься певне ID. Надалі це ID заноситиметься у список чатів для користувача та підтримки або кур’єру. В свою чергу до кожного ID чату заноситиметься користувач та співрозмовник а зв’язок вибудовуватиметься за допомогою ID. Загалом кожне повідомлення матиме ID для відстеження, і включатиме в себе текст та дату відправки. Клас Support включатиме в себе інформацію про ім’я, прізвище, електронну пошту, номер телефону, зберігатимуться чати та оцінки. Клас Courier відрізнятиметься лише полями «Транспортний засіб» та «Замовлення». Відображення зв’язку зображено на рисунку 2.3 та рисунку 2.4.

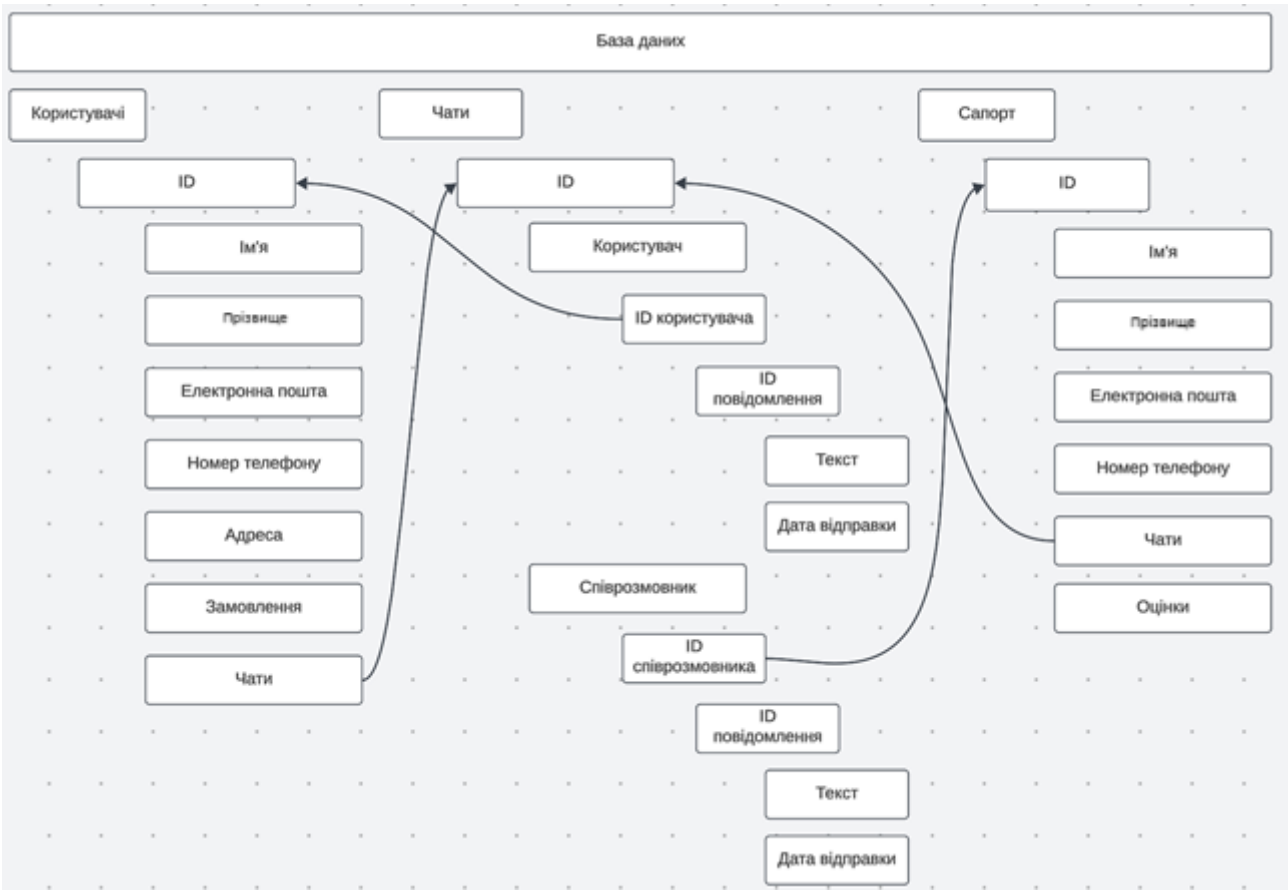

Рисунок 2.3 – Зв’язок чату з користувачем та сапортом

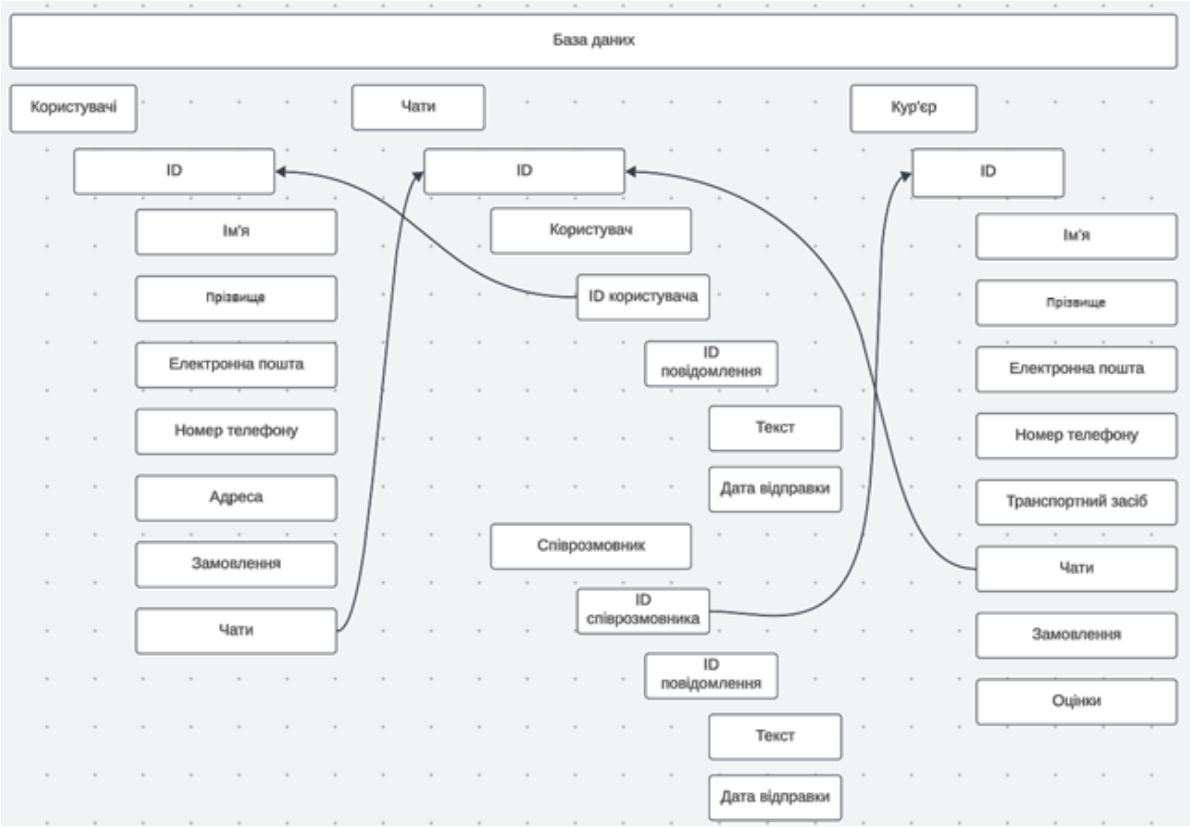

Рисунок 2.4 – Зв’язок чату з користувачем та кур’єром

У кожного кур’єру буде поле «Замовлення», в яке заноситиметься ID замовлення через яке будуватиметься зв’язок з певним об’єктом класу Order. Варто зазначити що ці два класи взаємопов’язані, адже кожен об’єкт Order має в собі ID кур’єра, який доставлятиме це замовлення. Взаємозв’язок можна побачити на рисунку 2.5.

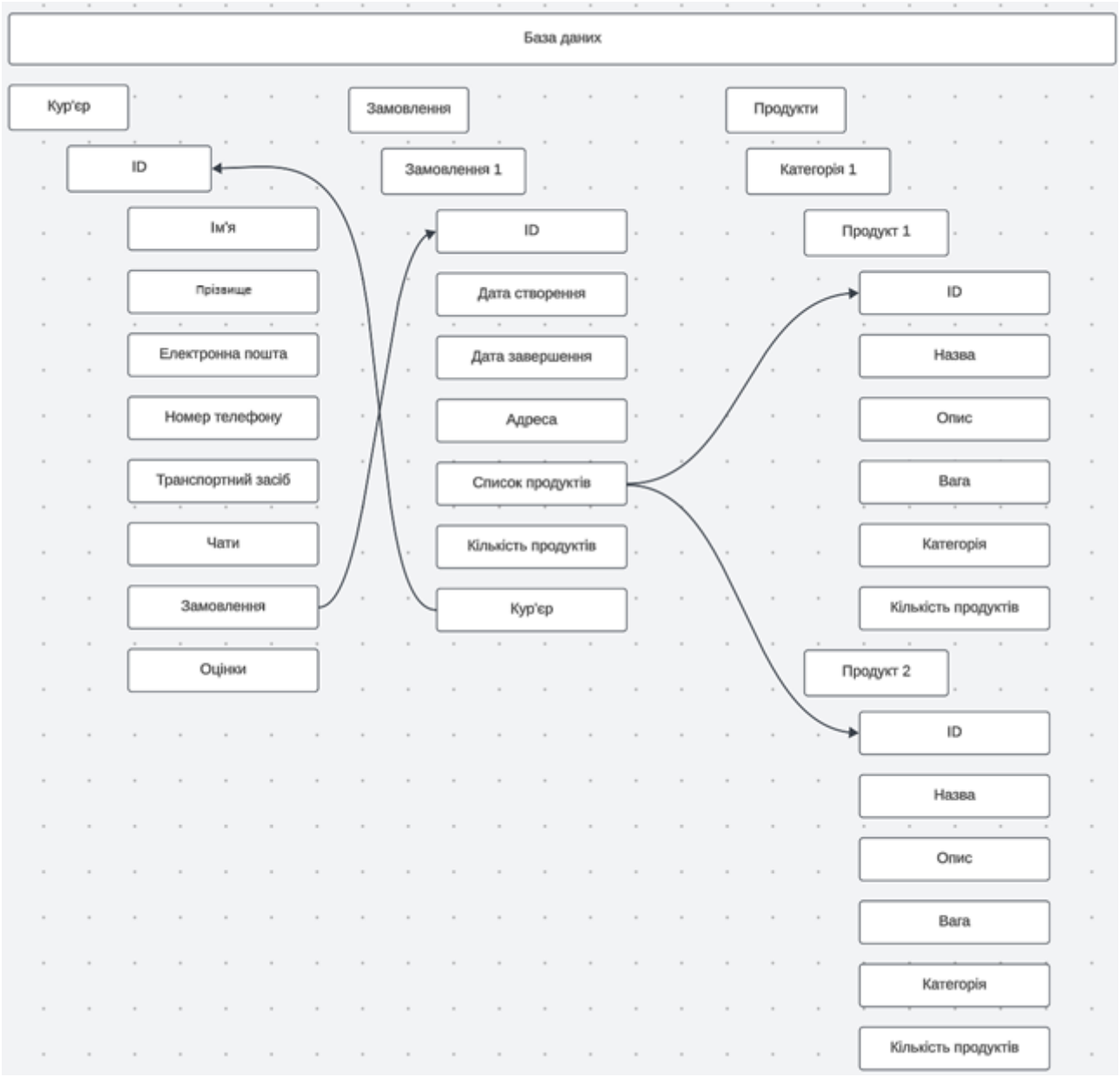

Рисунок 2.5 – Взаємозв’язок між кур’єром та замовленням у базі даних Firebase Realtime Database

Також в базу даних заноситиметься окремий клас Review, що включатиме в себе текстове поле з відгуком та оцінку в числовому форматі. Це дозволить представнику ролі «Адміністратор» сортувати відгуки за якістю. В разі наявності негативних оцінок проводиться пошук за ID, після чого знаходиться сапорт або кур’єр який отримав низьку оцінку. Надалі проводиться перевірка всіх облікових записів з ролями «Кур’єр» або «Сапорт» після чого порівнюється ID відгуку. Це дозволить швидко знаходити необхідного працівника (рисунок 2.6).

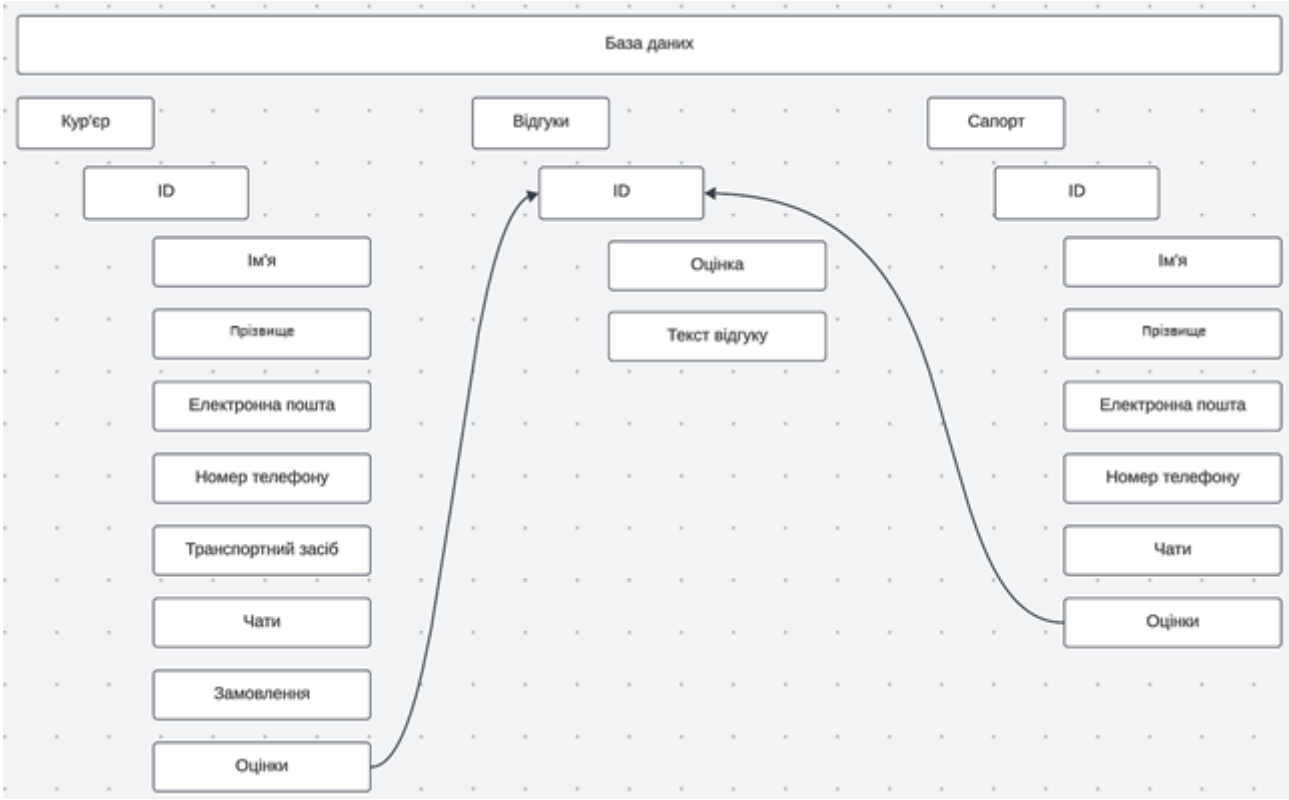

Рисунок 2.6 – Взаємозв’язок між комірками відгуків, кур’єрів та сапортів

Для приваблення покупців будуть впроваджуватися акційні пропозиції та знижки. Все це заноситиметься до окремого класу Promotion, який включатиме в себе два поля – «Список продуктів» та «Знижки». В обидві комірки буде заноситися список з даними. У першому полі буде список ID продукції, яка занесена до цієї акційної пропозиції, до другої – список з числовими значеннями знижок на товари. Такий підхід дозволить правильно налаштовувати та сортувати акційні пропозиції. Взаємозв’язок показано на рисунку 2.7.

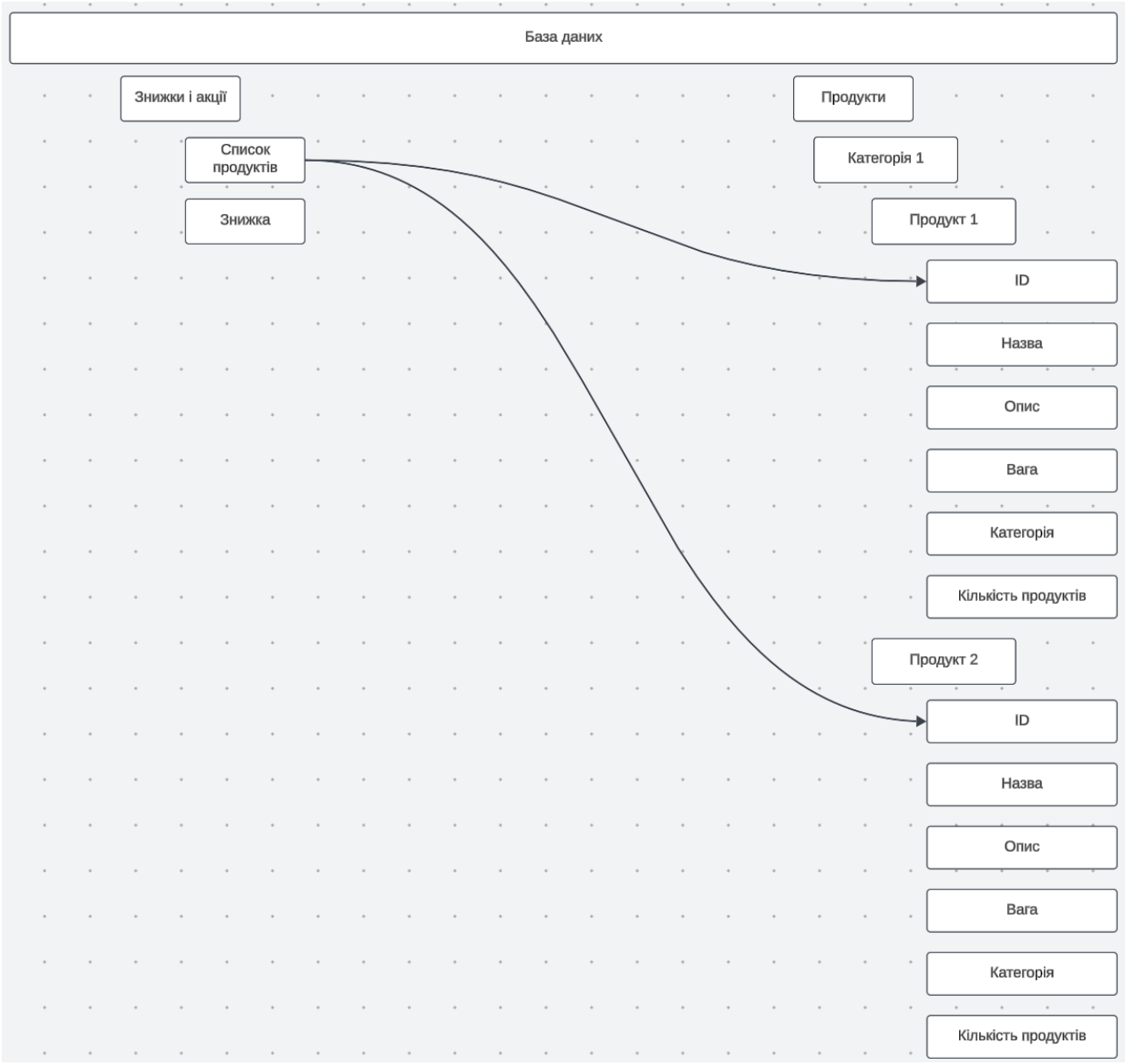

Рисунок 2.7 – База даних акційних пропозицій і знижок

2.4 Проєктування дизайну та інтерфейсу додатку

Дизайн додатку Monosell має бути мінімалістичним, сучасним і орієнтованим на зручність користувачів. Основним завданням буде створити інтерфейс, який буде інтуїтивно зрозумілим, естетично привабливим і практичним для виконання дій, пов’язаних із замовленням товарів.

Кольорова гамма повинна передавати професіоналізм і надійність. Основним кольором обирається синій у різних відтінках, адже він викликає довіру і асоціюється зі стабільністю. Для акцентів використовуватиметься зелений, який асоціюється з успіхом та природністю. Нейтральні кольори, такі як білий і світло-сірий, застосовуватимуться для фону, створюючи простір і полегшуючи сприйняття основного контенту. Для позначення помилок або попереджень використовуватиметься червоний або жовтий відповідно, щоб привернути увагу користувача до важливих повідомлень.

Шрифти є ключовим елементом дизайну, оскільки вони впливають на зручність читання. Шрифт Segoe Print та Roboto стане ідеальним вибором, оскільки він добре виглядатиме на мобільних пристроях, підтримуватиме кирилицю і забезпечить сучасний вигляд додатку. Різні стилі шрифтів – від жирного для заголовків до звичайного для тексту – дозволять забезпечити візуальну ієрархію.

Загальна кольорова гамма включатиме яскраво-червоні та жовті акценти для привертання уваги до знижок, при цьому основний фон залишиться світлим і нейтральним, щоб не відволікати. Завдяки сучасній візуалізації та продуманій структурі сторінка легко сприйматиметься і сприятиме швидкому вибору потрібних пропозицій. Розташування елементів інтерфейсу повинно бути логічним і забезпечувати швидкий доступ до основних функцій. Головний екран буде побудований з акцентом на категорії товарів у вигляді інтерактивних карток із зображеннями та назвами. У верхній частині розміщуватиметься пошуковий рядок, який завжди доступний для швидкого пошуку. Навігаційна панель із піктограмами, розташовуватиметься з лівого боку екрану. Це забезпечить легкий доступ навіть на великих екранах.

У списку товари відображатимуться у вигляді карток із зображеннями та назвами. Таке компонування дозволить швидко переглядати товари і взаємодіяти з ними. В кошику товари відображатимуться компактно зі зручним доступом до функцій редагування кількості або видалення. Загальна сума замовлення завжди буде видима внизу екрана разом із кнопкою "Оформити замовлення".

Дизайн також передбачатиме можливість анімацій, наприклад, при додаванні товару в кошик, щоб покращити враження від взаємодії. Темна тема стане додатковою перевагою, адже вона покращує доступність та комфорт для користувачів у нічний час.

Вибір кольорової гами, шрифту та компонування елементів базуватиметься на принципах юзабіліті та естетики. Ці рішення дозволять створити приємний досвід користувача, де акцент зроблено на простоті і функціональності, що відповідає концепції "магазину в телефоні" [14].

2.4.1 Проєктування форми реєстрації

Спочатку треба змоделювати дизайн форми реєстрації. Її головна мета — швидко і без зайвих зусиль провести майбутніх клієнтів через процес створення облікового запису.

Верхня частина міститиме спрощений варіант логотипа Monosell, що підкреслює бренд.

Сама форма для введення даних передбачає:

- ім'я: текстове поле з підказкою "Ваше ім'я";

- електронна пошта: текстове поле з підказкою "Електронна пошта";

- пароль: текстове поле для пароля з підказкою "Введіть пароль" і можливістю відображення/приховування символів;

- підтвердження пароля: текстове поле для підтвердження пароля.

Кнопка реєстрації розташовуватиметься під формою, на всю ширину та матиме легкий ефект натискання (зміна відтінку при кліку). Під кнопкою розташовуватиметься текст для навігації: "Вже маєте акаунт? Увійдіть" із посиланням на сторінку входу (синього кольору). Оскільки додаток використовуватиме Firebase Authentication, є можливість додаткової реєстрації через кнопку "Продовжити з Google" із логотипом Google.

Якщо користувач забув заповнити поле або ввів неправильні дані, під цим полем з’являтиметься текст помилки, наприклад, "Це поле є обов'язковим".

Всі кнопки та поля реагуватимуть на дії користувача. Кнопка реєстрації стане активною лише тоді, коли всі обов’язкові поля заповнені. Всі елементи

вирівнюватимуться по центру екрану, що полегшує доступ до них однією рукою. Кнопка реєстрації розташовуватиметься достатньо низько, щоб її було легко натиснути великим пальцем на мобільних пристроях. На схематичному зображенні поле з номером 1 відображає розташування логотипу, поля з номерами 2,3 – це EditText, в який користувач вводить свої дані для реєстрації, поля 4 – EditText для введення паролю. Під ним розташовуватиметься об’єкт класу CheckBox, при натисканні на який можна буде сховати, а бо показати введений пароль. Це буде зроблено для підвищення зручності. Номер 5 означає кнопку Button, при натисканні на яку відбуватиметься реєстрація, якщо всі поля заповнені правильно. Об’єкт під номером 6 – це текст в полі TextView, розрахований на те, що користувач вже має акаунт і бажає перейти до логіну. Під номером 7 кнопка Button, яка переводитиме користувача до форми реєстрації через акаунт Google. Результат моделювання представлено на рисунку 2.8.

Рисунок 2.8 – Модель форми реєстрації

2.4.2 Модель сторінки для логіну користувача

Після проходження реєстрації майбутній покупець переходитиме на сторінку для входу. Як і у форми для реєстрації тут поставлена проста задача – зробити вхід максимально зручним та швидким.

Верхня частина міститиме логотип Monosell для брендингу та забезпечення впізнавання, що відображається пунктом 1 на рисунку 2.9.

У центрі сторінки знаходитиметься форма, яка складатиметься з двох основних полів:

- електронна пошта: текстове поле EditText з підказкою "Введіть вашу електронну пошту", що помічено номером 2 на схематичному зображенні;

- пароль: текстове поле EditText з підказкою "Введіть ваш пароль" із кнопкою відображення/приховування пароля (іконка ока), що відображається номером 3;

- під полями форми буде розташована кнопка Button "Увійти" під номером 5, а також допоміжний текст TextView "Забули пароль?" під номером 4 із посиланням на сторінку відновлення пароля (синій текст).

Під кнопкою "Увійти" розміщуватиметься допоміжний текст TextView для навігації "Ще не маєте акаунта MonoSell? Зареєструйтесь" із посиланням на сторінку реєстрації, що помічений на рисунку 2.9 номером 6.

Оскільки використовуватиметься Firebase Authentication, покупцям буде доступний вхід через соціальну мережу Google завдяки кнопці Button під номером 7.

У разі невірного введення даних або інших проблем (наприклад, невірний пароль), під формою з'являтиметься червоний текст із повідомленням.

Кнопка "Увійти" розташовуватиметься на зручній висоті для взаємодії великим пальцем на мобільному пристрої. Посилання "Забули пароль?" та "Зареєструйтесь" матимуть чіткий відступ від кнопки входу, щоб уникнути випадкового натискання.

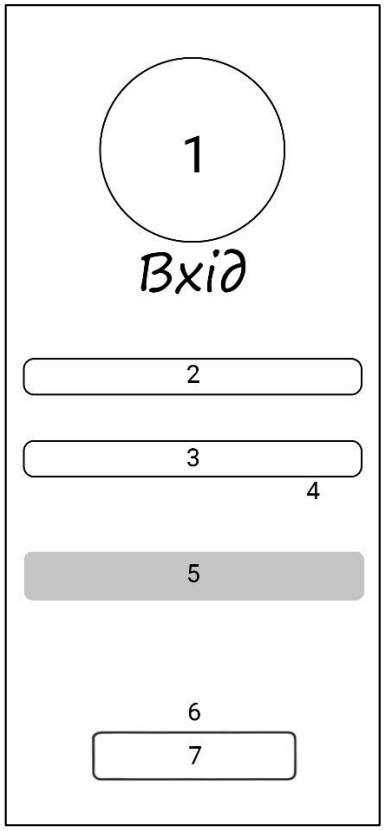

Рисунок 2.9 – Модель форми логіну

2.4.3 Модель головної сторінки додатку MonoSell

Головна сторінка додатку має бути сучасною, зручною та інтуїтивно зрозумілою, щоб створити позитивний досвід для користувачів. У верхній частині екрана розташовуватиметься логотип магазину, що відображений номером 3 на рисунку 2.10 та кнопка NavigationView для відкриття бокового меню під номером 2, що надасть доступ до основних вкладок. Центральна частина займатиметься пошуковим рядком SearchView для швидкого знаходження товарів, що відображено поміткою 1 на схематичному зображенні, а під ним розташовуватиметься новорічне оформлення дизайну під номером 4. Динамічний банер RecyclerView з акційними пропозиціями відображається номером 5, в нього передаватиметься список з зображеннями, при натисканні на які користувачеві демонструватиметься детальна інформація щодо акцій. Номер 6 відображатиме який за порядковим номером банер зображено на екрані

Основна частина сторінки буде заповнена асортиментом товарів, який представлений у вигляді зручної сітки. Це можливе завдяки об’єкту

RecyclerView. Кожен товар матиме фото та назву. Для категоризації товарів буде передбачена прокрутка з іконками: свіжі фрукти, овочі, зелень, м’ясні вироби, молочні вироби тощо, відображена цифрою 7.

На нижній панелі буде розташована кнопка Button з логотипом кошику, що переводитиме користувача на вкладку з обраними для покупки товарами. Вона позначена цифрою 8 на рисунку 2.10. Номер 9 – кнопка Button для налаштування сповіщень, а номер 10 – це кнопка Button, що переводить користувача до загальних налаштувань акаунту.

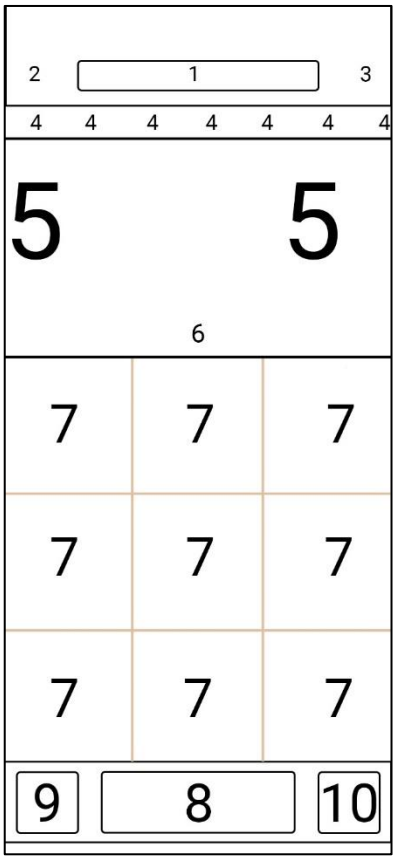

Рисунок 2.10 – Головна сторінка додатку

В лівому верхньому куті буде доступна кнопка для відкриття бокового меню, відображена цифрою 2 на рисунку 2.11. Бокове меню передбачатиме можливість навігації між сторінками. Позначка 11 – пункт для переходу на Activity головної сторінки, позначка 12 – пункт для переходу на Activity з акційними пропозиціями, 13 – Activity з гарячими стравами, 14 – Activity зі

списками обраних товарів, 15 – перехід на Activity з доставкою замовлення. Цифра 16 – діалогове вікно Dialog для обрання вашої адреси.

Рисунок 2.11 – Бокове меню для навігації

2.4.4 Модель сторінки з картою для відстеження замовлень

Сторінка з мапою для відстеження доставки замовлень буде інформативною та інтуїтивно зрозумілою, щоб забезпечити користувачам комфортний досвід. Центральне місце займатиме інтерактивна карта в реальному часі. Вона матиме чітке маркування: точка, що відображатиме місцезнаходження користувача, позначена синім маркером, а місцезнаходження кур'єра буде виділено зеленим маркером з іконкою мотоцикла, авто або людини з сумкою.

У верхній частині екрана буде блок із коротким статусом замовлення. Наприклад, статус показуватиме повідомлення на кшталт "Ваше замовлення готується" або "Кур'єр у дорозі". Поруч відображатиметься орієнтовний час прибуття, який оновлюватиметься в реальному часі.

Користувач зможе взаємодіяти з картою, масштабуючи її або переміщаючи, щоб переглянути деталі маршруту кур’єра. Маршрут кур'єра до місця доставки підкреслюватиметься яскравою лінією на карті. Нижче карти може бути доступний список із деталями замовлення: орієнтований час доставки, ім’я і контакт кур'єра з кнопкою швидкого дзвінка або повідомлення.

Для більшого залучення буде передбачено push-сповіщення або з'являються поп-апи з оновленнями статусу доставки. Наприклад: "Кур'єр уже поруч!".

Пункт 1– SearchView для пошуку товарів, 2 – бокове меню NavigationView, 3 – логотип додатку, 4 – новорічне оформлення, 5 – TextView з оновленим статусом доставки замовлення, пунктом 6 відображено MapFragment, де демонструватиметься мапа, а 7 – це оновлене відображення місцезнаходження кур’єра. Пункти 8, 9 та 10 – це TextView з даними про кур’єра, а пунктом 11 помічено кнопку Button для переходу на Activity з чатом (рисунок 2.12).

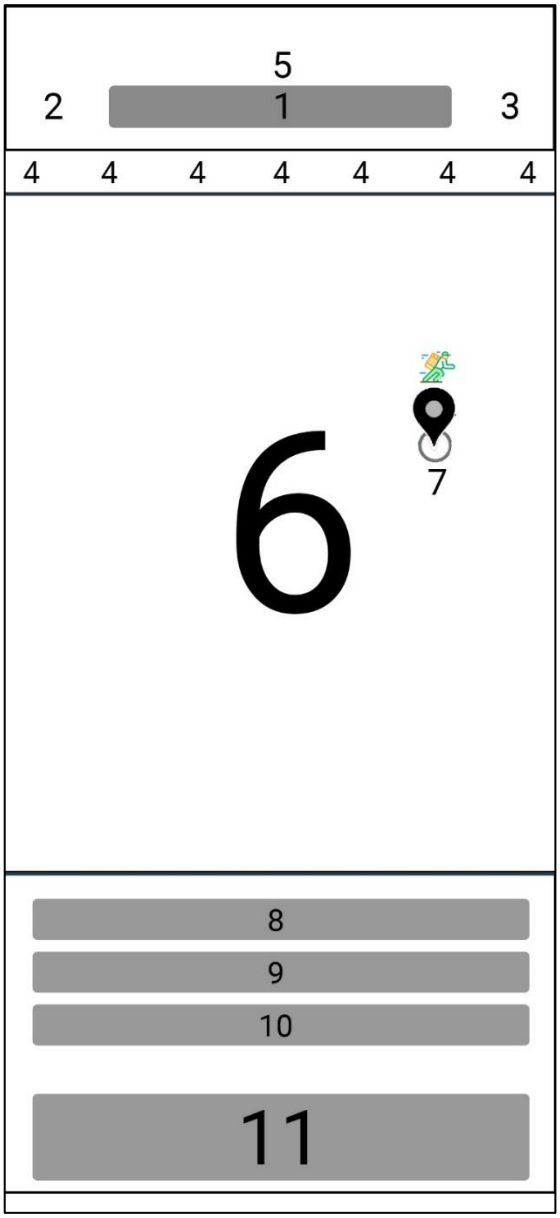

Рисунок 2.12 – Модель мапи користувача

2.4.5 Модель сторінки з кошиком покупця

Сторінка кошика замовлень має бути структурованою та інтуїтивно зрозумілою, створюючи комфортний досвід перед фіналізацією покупки. У верхній частині сторінки розміщуватиметься заголовок "Кошик", щоб користувач чітко розумів, де він знаходиться. Розташування позначене цифрою 5 на рисунку 2.13.

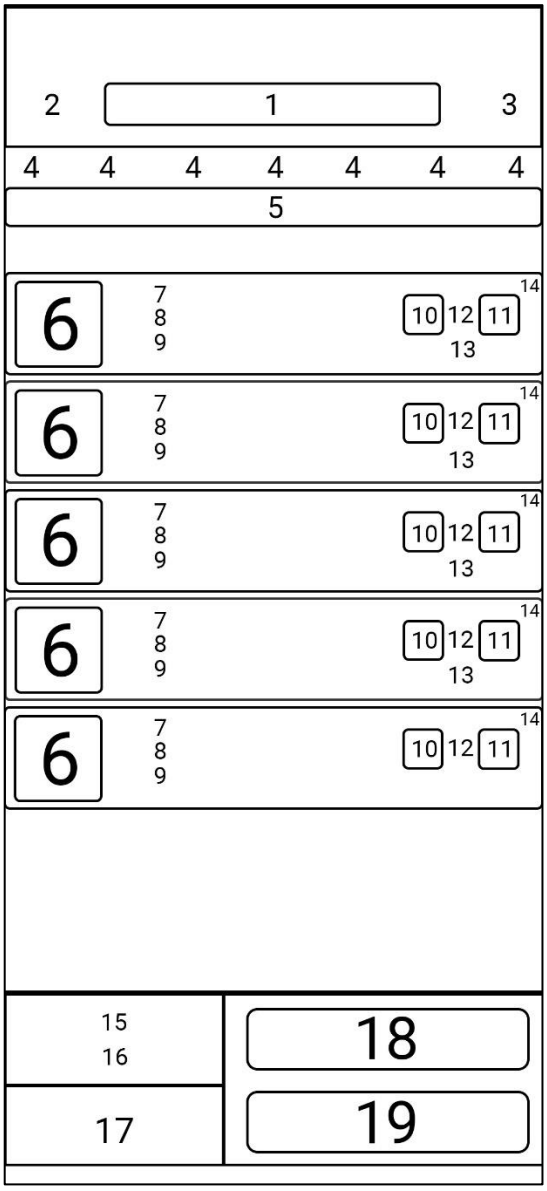

Рисунок 2.13 – Схема кошику користувача

Основна частина сторінки буде присвячена списку товарів, доданих у кошик. Кожна позиція відображатиметься у вигляді картки з мініатюрним зображенням товару (елемент 6), його назвою (елемент 7), ціною за одиницю (елемент 8), вагою (елемент 9) та загальною сумою для цієї позиції (елемент 13). Користувач може змінити кількість товарів (елемент 12) за допомогою кнопки Button з плюсом (елемент 11) і мінусом (елемент 10) або видалити товар з кошика через зручну іконку (елемент 14). Ці кнопки реагуватимуть миттєво, оновлюючи підсумкову суму.

Під списком товарів розташовуватиметься блок із розрахунком загальної вартості замовлення, який включає підсумок TextView, що позначений цифрою 15, знижки (якщо є), вартість доставки, позначена цифрою 16, і підсумкову суму, позначена цифрою 17. Поруч буде розташована кнопка Button переходу до оформлення замовлення, яка яскраво виділятиметься зеленим кольором, на рисунку 2.13 ця кнопка відображається цифрою 18.

У нижній частині сторінки буде кнопка "Повернутися до покупок", що стимулює додавати більше товарів. Її розташування позначене цифрою 19. Кольорова гама залишиться світлою, із акцентами на ключових діях і даних, таких як ціни та кнопка оформлення замовлення.

2.4.6 Проєктування інтерфейсу для кур’єра

Інтерфейс для кур’єрів буде максимально простий, функціональний і зручний, адже основна мета — допомогти кур’єру швидко орієнтуватися у замовленнях та забезпечити якісну доставку.

На головному екрані в ролі кур’єра відображатиметься інтерактивна карта, яка займатиме більшу частину сторінки. Вона показуватиме доступні замовлення у вигляді маркерів, що відображають адресу замовлення і інформацію про вагу чи тип товарів. Вид транспорту (наприклад, велосипед, мотоцикл чи авто), який кур’єр обрав у налаштуваннях, впливатиме на радіус доступних замовлень, і ця зона також буде позначена на карті для візуального розуміння.

У верхній частині екрана буде розташовано статусний рядок, де вказано поточний стан кур’єра: "Вільний", "Зайнятий" чи "Доставка в процесі". Поруч знаходиться кнопка для зміни статусу "Перейти в офлайн".

При натисканні на маркер замовлення на карті з’являтиметься спливаюче вікно з деталями: адреса, опис товарів, приблизна вага, відстань до клієнта та орієнтовний дохід від доставки. У цьому вікні буде дві кнопки: "Прийняти" і "Відхилити". Якщо кур’єр приймає замовлення, точка автоматично зникає з карт інших кур’єрів, а додаток розраховує та показує оптимальний маршрут на карті.

У процесі доставки буде доступний чат із клієнтом. Інтерфейс чату спрощений, із можливістю надсилати текстові повідомлення чи швидкі відповіді

на зразок "Я вже біля вас", "Пробки на дорозі". Чат можна буде швидко викликати через кнопку, розташовану на карті або у спливаючому вікні замовлення.

Друга вкладка — налаштування. Тут кур’єр може оновити інформацію про свій транспорт, наприклад, змінити тип або вказати доступність в інші години. Всі ці параметри впливають на пошук замовлень. Також у цьому розділі є кнопка "Переключитися на клієнта", що повертає користувача до інтерфейсу для покупців.

Загалом інтерфейс мало чим відрізнятиметься від аналогічної вкладки у користувача, проте на боковому меню, що позначене цифрою 2 на рисунку 2.14 буде всього лише дві вкладки з налаштуваннями та картою. Замість логотипу цифра 3 відображає кнопку для перемикання в режим користувача. Цифрою 5 позначене майбутнє розташування статусного рядка. Цифра 12 – коло, яке відображатиме радіус в межах якого кур’єр може приймати замовлення.

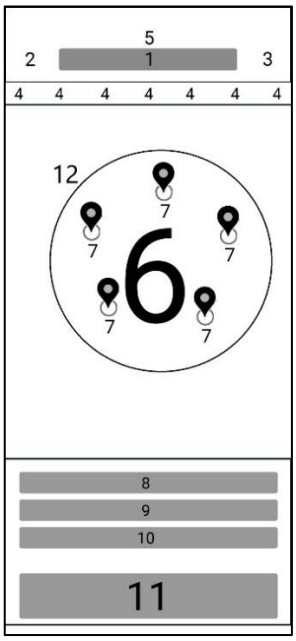

Рисунок 2.14 – Схема інтерфейсу для кур’єра

Загальний стиль інтерфейсу залишиться мінімалістичним, з акцентом на функціональність. Кольорові акценти використовуватимуться для важливих елементів, таких як маркери замовлень чи статус доставки, щоб швидко

привертати увагу. Чітка навігація та адаптивний дизайн забезпечуватимуть кур’єру простоту користування навіть у динамічних умовах роботи.

2.4.7 Проєктування інтерфейсу для представників служби підтримки

Інтерфейс для представників служби підтримки буде продуманим, простим у використанні та орієнтованим на швидке вирішення запитів користувачів. Головний екран у режимі «Служба підтримки» буде представлений списком звернень покупців. У верхній частині екрана знаходитиметься заголовок «Заявки від користувачів».

Кожне звернення у списку відображатиметься у вигляді окремого блоку з короткою інформацією: ім'я користувача, текст останнього повідомлення, час останньої активності та індикатор статусу (наприклад, зеленим позначено «В мережі», сірим — «Офлайн»).

При виборі конкретного звернення відкриватиметься вікно діалогу. Верхня частина цього вікна міститиме інформацію про користувача, включаючи ім'я, контактні дані (якщо вони доступні), статус у мережі та відомості про замовлення або товар, який став причиною звернення. Наприклад, відображатимуться назва товару, номер замовлення та деталі проблеми, якщо це було вказано клієнтом.

У самому чаті передбачено зрозумілий і функціональний інтерфейс для листування. Повідомлення відображатимуться у форматі діалогу з позначками про прочитання, часом відправлення та статусом відповіді. Для зручності служби підтримки передбачені шаблонні відповіді та кнопка «Вирішити проблему», яка автоматично фіксуватиме успішне завершення діалогу.

Поруч із полем введення тексту знаходитимуться кнопки для прикріплення файлів (наприклад, зображень чи документів) або швидкого переходу до історії замовлення клієнта. Після завершення діалогу користувач отримуватиме повідомлення із пропозицією оцінити якість підтримки.

Верхня частина екрану не зміниться, на рисунку 2.15 зображені елементи, де цифра 1 – пошуковий рядок, 2 – кнопка для виклику бокового меню, 3 – логотип додатку, , 4 – новорічне оформлення додатку, 5 – це заголовок TextView

«Заявки від користувачів», весь список буде створено за допомогою RecyclerView,, де в кожному елементі цифрою 6 показане майбутнє розташування фото користувача, 7 - його статус відображений у TextView, 8 – прізвище та ім’я користувача TextView, 9 – текстове поле зі скаргою або пропозицією, 10 – час останнього заходу в мережу, а 11 – кількість повідомлень.

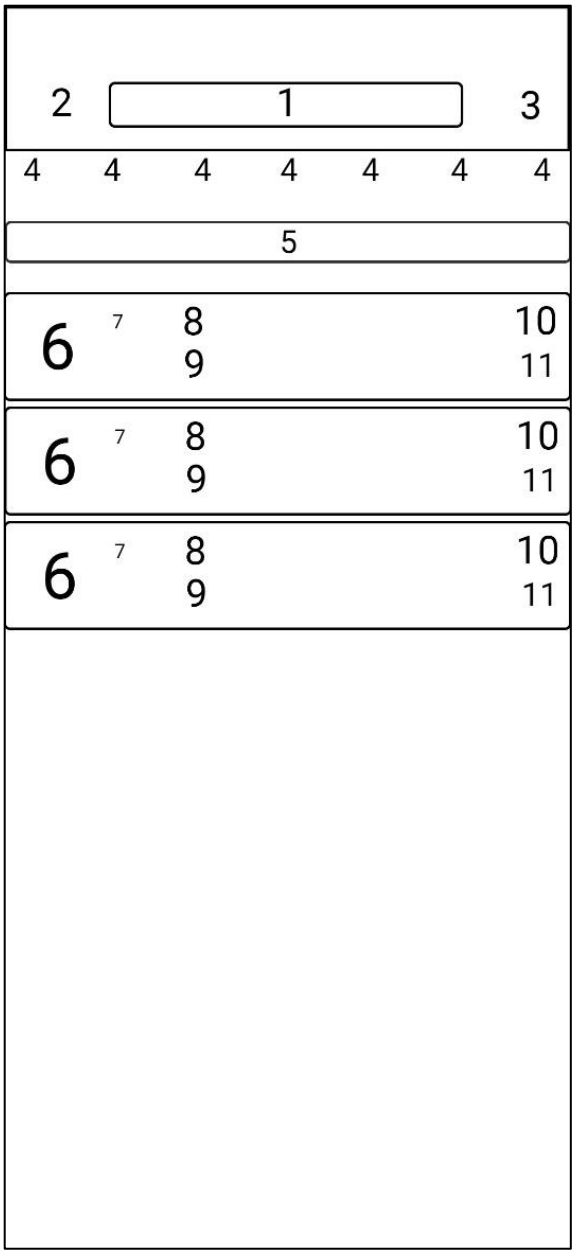

Рисунок 2.15 – Модель з чатами у сапорта

2.4.8 Моделювання інтерфейсу для представників ролі «Адміністратор»

У верхній частині сторінки буде розташований пошуковий рядок для швидкого доступу до категорій, товарів або користувачів. Центральна частина інтерфейсу буде поділена на кілька секцій. Перший блок присвячений управлінню налаштуванням додатку. Тут адміністратор бачитиме можливі налаштування додатку, та матиме можливість редагувати деякі елементи. За допомогою другого блоку адміністратор зможе налаштовувати акційні пропозиції, додавати товари або збільшувати знижки. Третій блок це управління персоналом, де викликатиметься список всіх користувачів з ролями «Кур'єр» чи «Сапорт», а також буде можливість сортування за наданими оцінками їх діяльності. Звичайно, адміністратор зможе змінювати ролі та позбавляти доступу до функціоналу у випадку неналежного виконання своїх обов'язків. Четвертий блок – це управління категоріями та товарами. Тут адміністратор зможе додавати нові та видаляти існуючі категорії, редагувати їх фото та працювати з товарами. Налаштування мапи дозволить адміністратору змінювати зовнішній вигляд MapFragment. Загальні налаштування – можливість налаштування у самого адміністратора.

На рисунку 2.16 відображений проєктований інтерфейс редагування категорій та продукції. Позначки 1,2,3 та 4 не змінили свого значення. Під цифрою 5 позначене можливе розташування випадаючих списків з обранням категорії налаштувань, необхідних адміністратору. У даному випадку зображений макет з обраним налаштуванням категорій та продукції. Цифра 6 – позначення розташування фото з підписами різних категорій, цифра 7 – розташування кнопки для роботи з товарами, які належать до обраної категорії, цифра 8 – видалення категорії. Цифра 9 позначатиме кнопку для переходу в режим покупця з можливістю купувати товари. 10 – налаштування сповіщень, 11 – налаштування додатку.

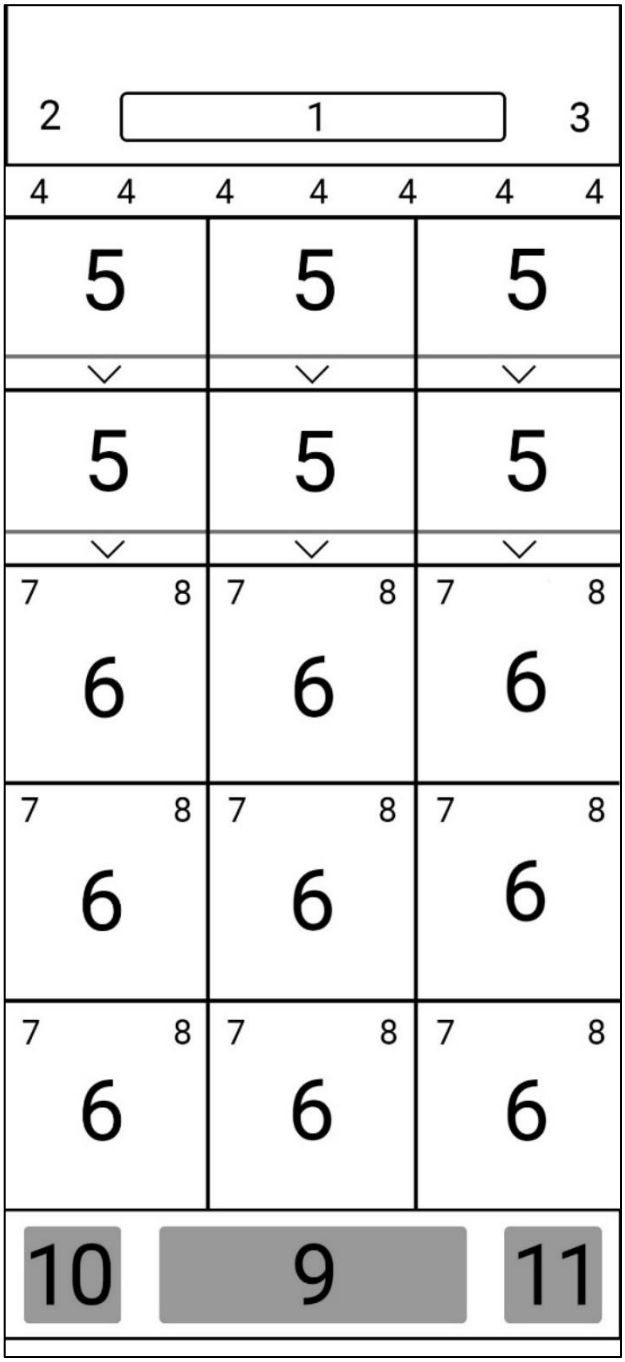

Рисунок 2.16 – Макет вікна редагування категорій та продукції у адміністратора

2.5 Розробка алгоритмів роботи з додатком

Загальний алгоритм роботи додатку виглядатиме наступним чином. Коли користувач буде заходити в додаток, викликатиметься MainActivity, де буде

проводитися першочергова перевірка, чи зареєстрований користувач. Якщо він не зареєстрований, то клієнт відправлятиметься на сторінку для реєстрації, а після заповнення полів – до LoginActivity. Якщо ж користувач вже матиме акаунт, то проводитиметься перевірка на час останньої активності. Якщо вона була більше ніж тиждень тому, то його відправлятимуть до LoginActivity для повторного заповнення даних. Після заповнення відбуватиметься перевірка правильності введених даних. Якщо все правильно, проводитиметься перевірка ролі, аналогічно буде і у випадку, якщо користувач заходить частіше ніж раз в 7 днів якщо ні – клієнт вводитиме дані повторно.

Спочатку проводитиметься перевірка, чи має користувач роль «Покупець»? Якщо так, його відправлятимуть до CatalogActivity, де він зможе придбати необхідні товари, провести налаштування, замовити доставку, передивитися акційні пропозиції, тощо.

Далі проводитиметься перевірка, чи має користувач роль «Адмін»? Якщо так, його відправлятимуть до AdminActivity, де він зможе проводити налаштування додатку, зміну каталогу, товарів, налаштовувати мапу, обробляти відгуки та змінювати ролі у користувачів.

Далі проводитиметься перевірка, чи має користувач роль «Кур’єр»? Якщо так, його відправлятимуть до DeliveryMapActivity, де він зможе приймати замовлення, читати інформацію про замовлення, спілкуватися з клієнтом та прокладати найбільш оптимальний маршрут в залежності від обраного типу транспорту. Також на ньому оброблятиметься логіка, за якою доставка не може відмовитися від замовлення багато разів.

Якщо ж жодна з вищевказаних ролей не підходить, то користувач має роль «Сапорт» і направлятиметься до SupportActivity, де представлений список чатів з клієнтами, які мають питання. Алгоритм переходу до правильного Activity зображено на рисунку 2.17.

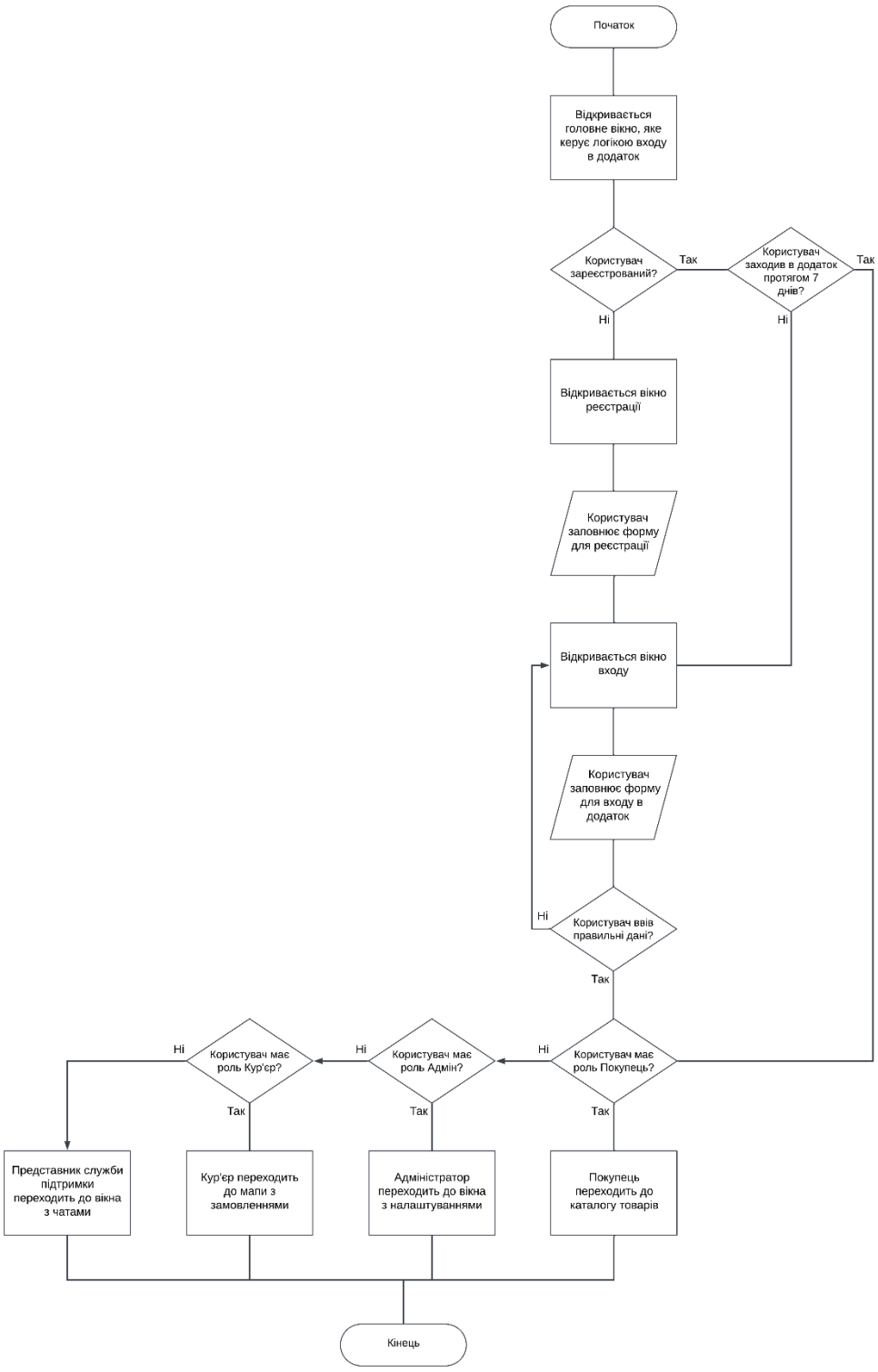

Рисунок 2.17 – Алгоритм розподілення користувача між різними Activity

Загальна логіка роботи додатку з покупцем виглядатиме наступним чином. Покупець переходитиме до CatalogActivity. Якщо клієнт бажатиме оформити певне замовлення, то у нього буде два варіанти: або він шукатиме певний продукт, який йому найбільше потрібен, або ж перейде до перегляду каталогу. У

першому випадку користувач скористається SearchView, де буде вводити назву потрібного товару.

У другому випадку покупець просто переглядатиме каталог та шукатиме необхідний товар по категоріях. Коли клієнт натискатиме на товар, то переходитиме до нього та додаватиме у кошик. Якщо він забажає продовжити покупки, то повертатиметься до CatalogActivity. Якщо ні – покупець натискатиме на кнопку кошику, переходитиме до ShoppingCartActivity, де проводитиме редагування замовлення.

Якщо він забажає переглянути каталог ще раз, то зможе повернутися до нього, або ж оформити замовлення, після чого переходитиме до CustomerMapActivity, далі отримуватиме замовлення і оцінюватиме якість доставки.

Якщо ж покупець з самого початку не буде бажати оформити замовлення, то зможе перейти або до перегляду акційних пропозицій, або до перегляду списку замовлень, до замовлення гарячих страв, спілкування з представниками служби підтримки та проведення налаштувань.

В першому випадку користувач переходитиме до PromotionListActivity, де переглядатиме акційні пропозиції, обиратиме необхідні товари та переходитиме до алгоритму їх придбання.

В другому випадку користувач переходитиме до ShoppingListActivity.

В третьому випадку – до HotActivity, де замовлятиме гарячі страви або інгредієнти до них, переходячи до ShoppingCartActivity.

Для спілкування з сапортом відбудеться перехід до ChatWithSupportActivity, де відображатиметься чат та статус обох учасників діалогу.

Для налаштування використовуватиметься UserSettingsActivity. Алгоритм роботи додатку за покупця продемонстровано на рисунку 2.18.

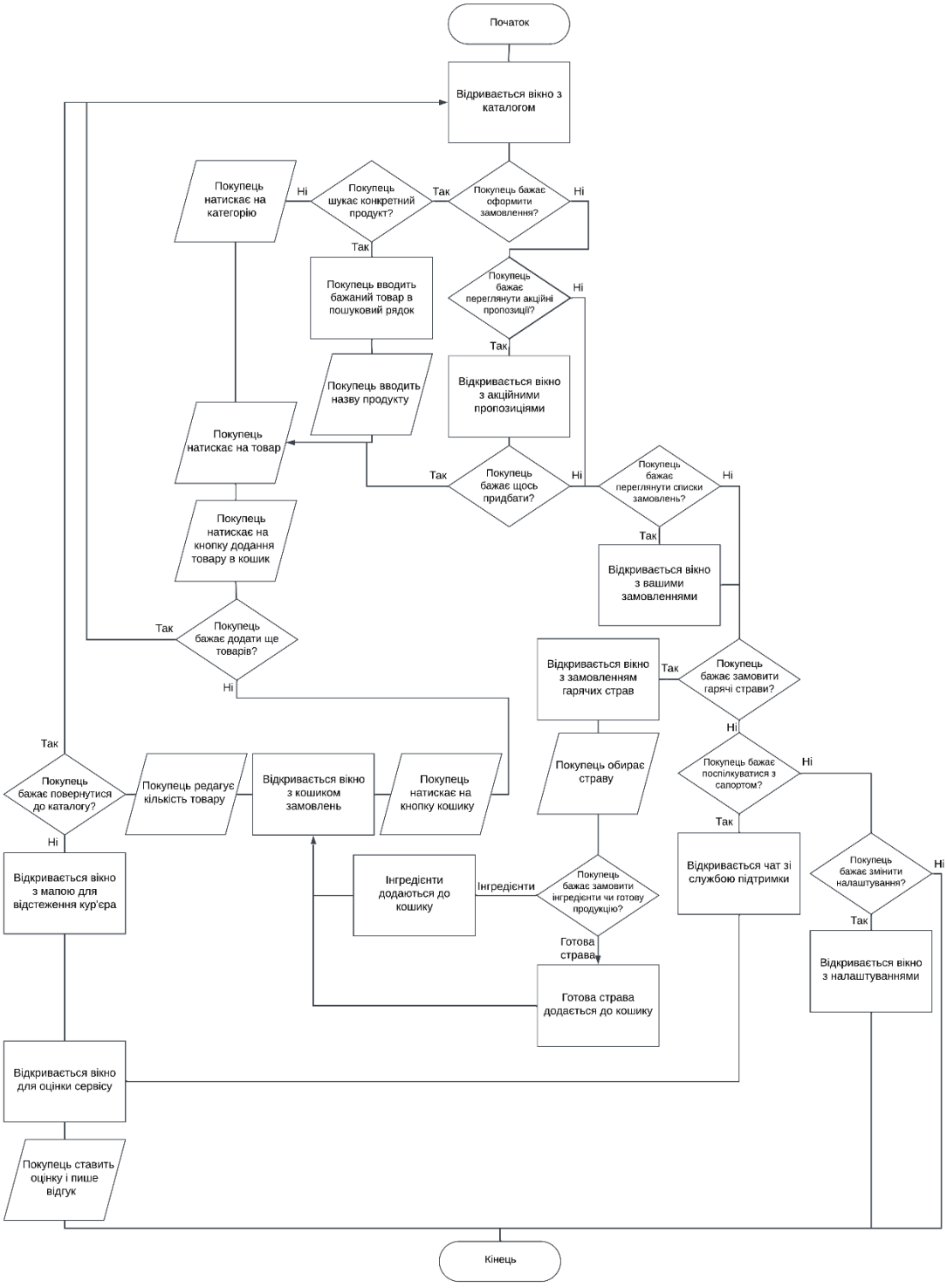

Рисунок 2.18 – Загальний алгоритм роботи з додатком для користувача

Варто розглянути і алгоритм роботи додатку з боку кур’єра. Початок роботи відбуватиметься на DeliveryMapActivity. В залежності від того, який транспорт буде обраний для доставки, на карті відображатимуться замовлення в

певному радіусі. Власним ходом – 2 км, на двоколісному транспорті – 6 км, на автомобілі – 10 км.

Далі кур’єр або підтверджуватиме, або відмовлятиметься від доставки замовлення. У разі відмови проводитиметься перевірка, чи було відмовлено більш ніж 10 покупцям за тиждень.

Якщо ні – проводитиметься перевірка, чи відхилив кур’єр більше 2 замовлень за день.

Коли кур’єр підтверджуватиме замовлення, на карті прокладатиметься маршрут. Якщо кур’єр бажатиме продовжити доставку, то алгоритм повторюватиметься з DeliveryMapActivity (рисунок 2.19).

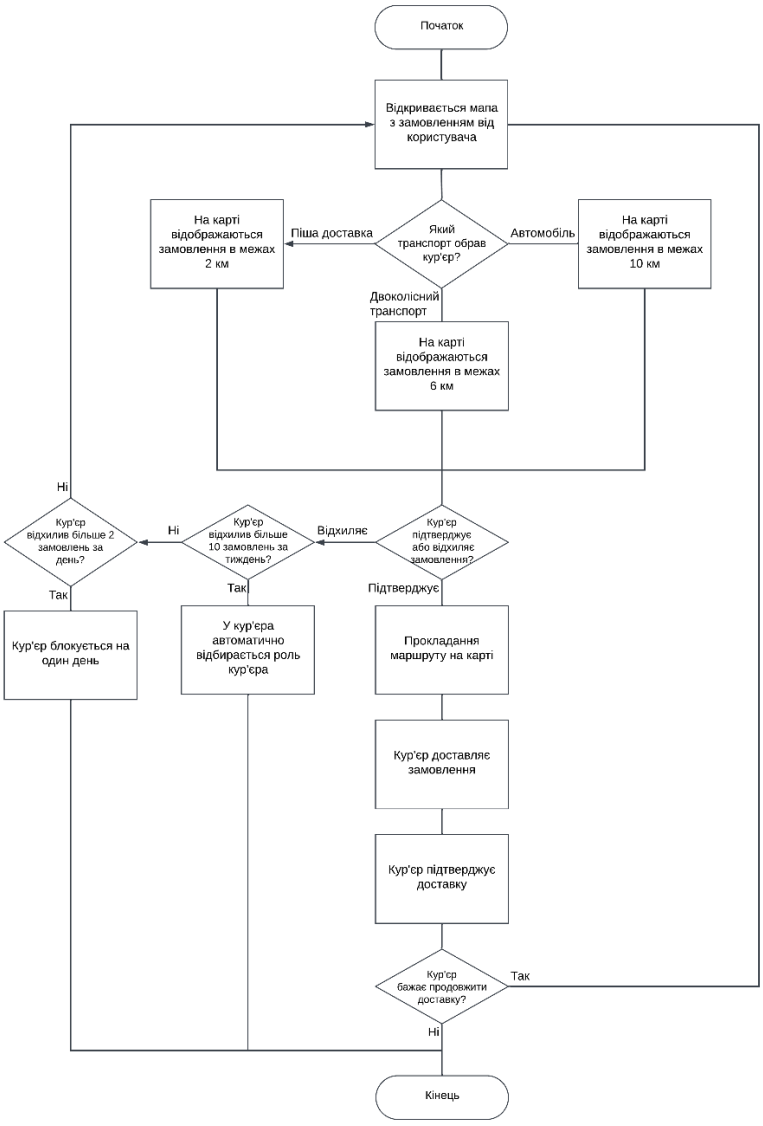

Рисунок 2.19 – Загальний алгоритм роботи з додатком для кур’єра

Якщо розглядати алгоритм роботи представника служби підтримки, то можна зрозуміти, що процес починатиметься з SupportActivity, де користувач обиратиме чат з клієнтом, відкриватиметься SupportChatActivity, а далі під час діалогу вирішуватиметься проблема покупця. Якщо сапорт бажатиме обробити наступне питання, він повертатиметься до SupportActivity. Якщо ні, то він зможе або змінити налаштування, або завершити алгоритм роботи в якості підтримки, що зображено на рисунку 2.20.

Рисунок 2.20 – Загальний алгоритм роботи з додатком для представника служби підтримки

Початок роботи в якості адміністратору починатиметься з AdminActivity. В залежного від того, з якими саме параметрами бажає працювати адміністратор, відбуватиметься обрання певної категорії налаштувань. Далі до екрану зі списком можливих налаштувань передаватиметься певний адаптер – AppSettingsAdapter, PromotionSettingsAdapter, StaffSettingsAdapter, CatalogSettingsAdapter, MapSettingsAdapter, AdminSettinsAdapter, призначених для налаштування дизайну застосунку, акційних пропозицій, роботи з персоналом, редагування каталогу, налаштування мапи та налаштувань для адміністратору відповідно. Алгоритм зображено на рисунку 2.21.

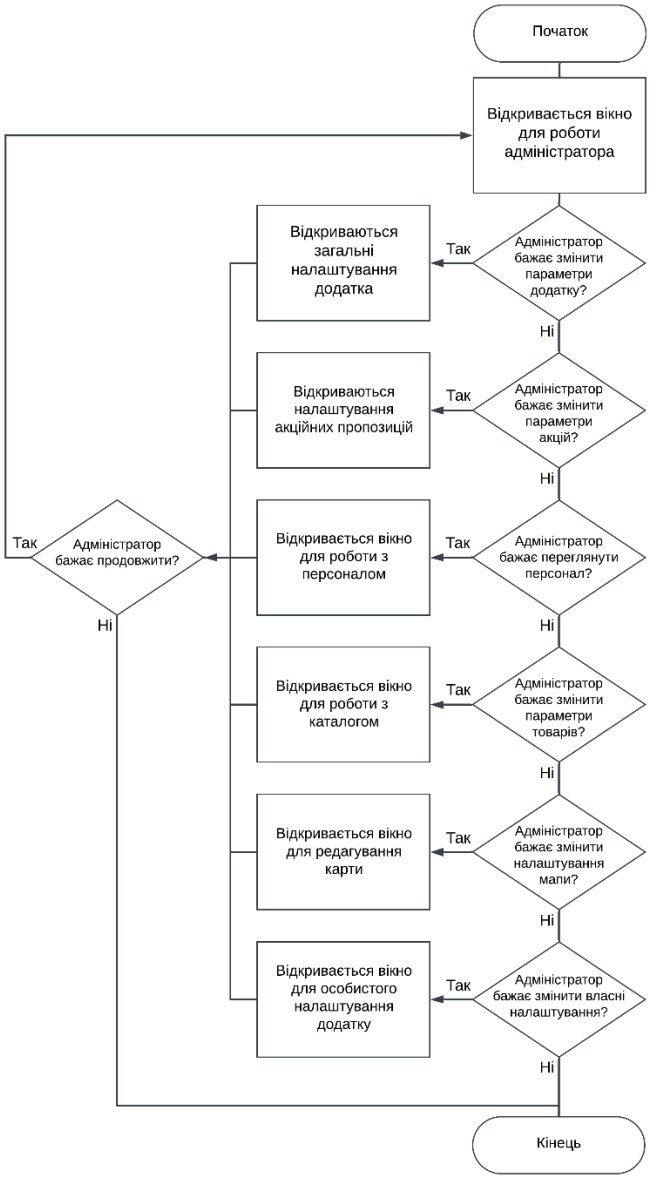

Рисунок 2.21 – Загальний алгоритм роботи з додатком для адміністратору

3 ДОСЛІДЖЕННЯ ЕФЕКТИВНОГО РОЗПОДІЛУ РЕСУРСІВ

Розрахунок навантаження на кур'єрів для служби доставки є важливим завданням, оскільки він впливає на ефективність роботи компанії, задоволення клієнтів та витрати на обслуговування. Рівномірний розподіл завдань дозволяє уникнути ситуацій, коли одні кур'єри перевантажені, а інші мають мало роботи. Це сприяє більш ефективному використанню людських ресурсів, а знаючи кількість замовлень та їх географічне розташування, можна створити оптимальні маршрути, зменшуючи час доставки.

3.1 Дослідження методів розподілу замовлень між кур'єрами

Варто зазначити, що для високої якості наданих послуг та забезпечення рівноцінного розподілу коштів серед кур’єрів потрібно використати найбільш оптимальний алгоритм, який назначатиме замовлення між різними кур’єрами. В ході дослідження було проведено тестування двох алгоритмів, за якими проводитиметься розподіл замовлень – це угорський алгоритм та жадібний алгоритм.

Жадібний алгоритм працює за принципом вибору локально найкращого рішення на кожному етапі. Замовлення призначається тому кур’єру, у якого на цей момент найменша відстань до точки доставки або найменша "ціна" виконання. Хоча він швидкий і простий у реалізації, жадібний алгоритм часто ігнорує загальну картину, що може призводити до нерівномірного розподілу завдань між кур’єрами [11].

Формула жадібного алгоритму:

$\forall j \in \{1,2,\dots,n\}, i^* = \arg\min c_{ij}, x_{i^*j} = 1$, де (3.1)

j — індекс замовлення (точка роздачі).

$j \in \{1,2,\dots,n\}$, де n — загальна кількість замовлень;

i — індекс кур’єра.

$i \in \{1,2,\dots,m\}$, де m — кількість доступних кур’єрів;

c_{ij} — "вартість" виконання замовлення j кур’єром i;

$i^* = \arg min_i\, c_{ij}$ — вибір кур’єра i^* для якого вартість c_{ij} мінімальна. Іншими словами, для кожного замовлення j вибирається кур'єр, який зможе виконати це завдання з найменшими витратами;

$x_{i^*j} = 1$ — після визначення кур'єра i^*, йому призначається замовлення j. Змінна x_{i^*j} приймає значення 1 для вибраної пари (i^*, j), що означає, що це замовлення виконує цей кур'єр.

Тоді загальна вартість всіх замовлень виглядає так:

$$C = \sum_{j=1}^{n} c_{i^*j}, x_{i^*j} \tag{3.2}$$

Угорський алгоритм базується на мінімізації сумарної "ціни" для відповідності замовлень кур’єрам. Він використовує матрицю вартості, яка відображає "ціну" виконання замовлення кожним кур’єром. За допомогою перетворень матриці та покриття нулів мінімальною кількістю ліній, угорський алгоритм знаходить оптимальний розподіл замовлень, що дозволяє рівномірно розподілити навантаження [12].

Математична модель угорського алгоритму виглядає наступним чином:

$$min \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{3.3}$$

Тут c_{ij} — вартість призначення j-го замовлення i-му кур’єру;

x_{ij} — бінарна змінна, яка визначає, чи було призначено j-те замовлення i-му кур’єру;

$\sum_{i=1}^{m} \sum_{j=1}^{n}$ — загальна сума всіх витрат, яку потрібно мінімізувати.

Для проведення дослідження було імітовано створення спочатку 9 замовлень в різних радіусах від точки видачі, потім 15, потім 21, потім 30. При цьому обробляли їх 3 кур’єри. На рисунку 3.1 представлені різниці між максимальною ціною для шляху кожного кур’єру в залежності від використаного методу та кількості замовлень.

Як можна помітити, якщо для розподілення замовлень між кур’єрами буде використовуватися жадібний алгоритм та угорський алгоритм, то при меншій кількості замовлень різниця між ефективністю невелика, що відображається на «ціні» для шляху, яка включає в себе суму дистанцій, які подолає кур’єр для доставки замовлення та кількості вже виконаних замовлень, помножених на коефіцієнт.

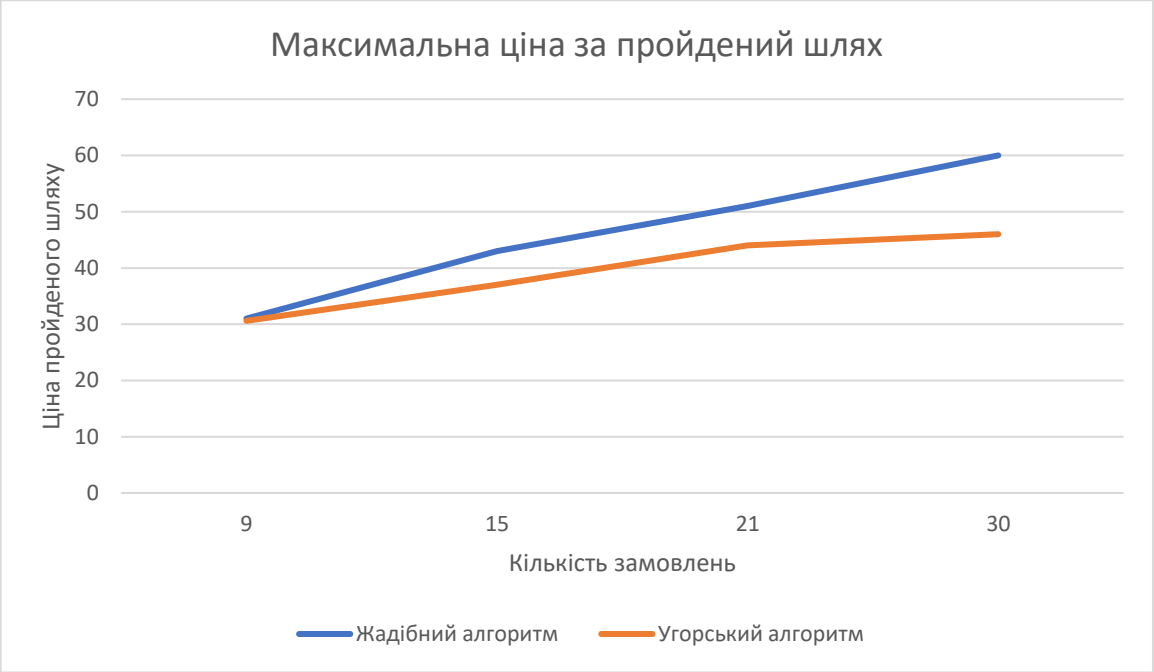

Рисунок 3.1 – Графік витрат при використанні жадібного та угорського алгоритмів

Варто зазначити, що у випадку з 9 замовлення і угорський алгоритм і жадібний алгоритм рівномірно розподілили замовлення. Проте, з появою 15 замовлень розподілення жадібним алгоритмом показало меншу ефективність, адже стало перевантажувати певного кур’єра, що погіршилося з підвищенням

кількості замовлень. Наостанок жадібний алгоритм розподілив 30 замовлень між трьома кур’єрами таким чином, що кожен з них мав на два виконаних замовлення більше ніж попередній.

Проблема полягає в тому, що жадібний алгоритм не підходить для рівномірного розподілення замовлень між трьома кур’єрами, тому що не враховує загальної картини. При цьому угорський алгоритм зіставляє замовлення і кур'єрів, мінімізуючи сумарні витрати, навіть якщо на першому кроці рішення здається неочевидним.

Якщо кур’єр буде доставляти замовлення, які між собою знаходяться на близькій відстані, то логічним рішенням буде розподілити їх одночасно для цього кур’єра, бо це збільшить ефективність. Варто також зазначити, що у випадку коли три кур’єра мали однакову «ціну» доставки одного замовлення, але при цьому різну «ціну» для інших, жадібний алгоритм був неефективним, адже враховував лише перше замовлення, в той час як угорський алгоритм дозволив мінімізувати відстань, яку кур’єрам довелося проїхати для імітації доставки замовлення.

Дослідивши обидва алгоритми можна прийти висновку, що для вирішення задачі не просто вигідного, але і рівномірного розподілу замовлень між кур’єрами краще обрати угорський алгоритм, адже він забезпечить рівномірний розподіл завдань між персоналом і зменшить кількість використаних ресурсів, тобто оптимізує доставку. Жадібний алгоритм хоча і швидший, він не здатен забезпечити надійності при виконанні великої кількості замовлень.

3.2 Визначення розташування точок видачі

З огляду на те, що ціллю дослідження є створення унікального досвіду користувача за допомогою додатку, завдяки якому доставка продукції стане

більш доступною, варто розглянути економічну вигоду та найбільш оптимальне місцерозташування точок видачі.

На рисунку 3.2 зображена мапа Запоріжжя, де чорними точками позначене приблизне розташування точок видачі, з яких буде відбуватися доставка. Зона радіусом 2 км позначена зеленим кольором і відображає радіус доставки, яка відбуватиметься пішки. Зона радіусом у 6 км позначена блакитним кольором і відображає доставку за допомогою двоколісних транспортних засобів. Найбільш дальній радіус у 10 км відображає зону доставки за допомогою автомобілю. Максимальна розрахована тривалість доставки складає 30 хвилин. Таким чином можна зробити висновок, що для оптимальної роботи та реалізації проєкту вистачить всього дві точки видачі.

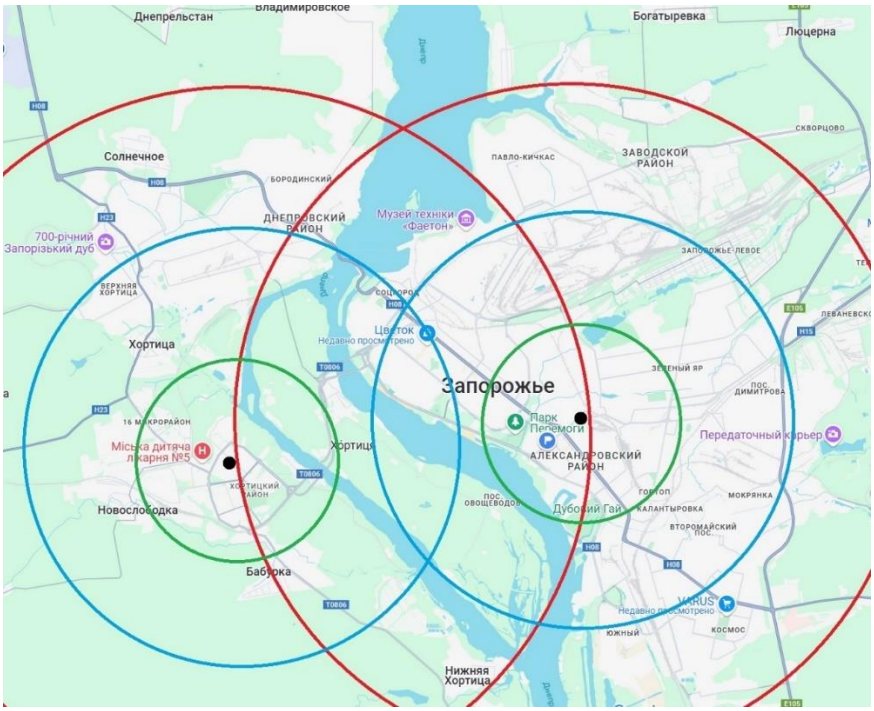

Рисунок 3.2 – Мапа розташування точок видачі та радіус доставки для різних видів транспорту

В свою чергу це позитивно вплине на собівартість послуг, адже плата за оренду буде значно меншою. За приблизними підрахунками оренда складських приміщень на території Хортицького району обійдеться у 5 тисяч доларів на місяць (з огляду на середню вартість 5 доларів за квадратний метр, та площу у

1000 квадратів), на території Олександрівського району – 8 тисяч доларів на місяць (з огляду на середню вартість 8 доларів за квадратний метр, та площу у 1000 квадратів). Загальна вартість оренди становить 13 тисяч доларів на місяць. Першочергово кошти підуть на проведення рекламної кампанії, закупівлю необхідного обладнання, транспортних засобів та пошук найбільш вигідних пропозицій транспортування продукції. З огляду на вищевказані аспекти очікується що собівартість продукції знизиться на 20-30%, що дозволить зробити процес доставки товарів більш вигідним та доступним для різних категорій населення.

4 РЕАЛІЗАЦІЯ ДОДАТКУ

Після проходження всіх підготовчих етапів, побудови алгоритмів роботи для кожної ролі та проєктування структури і дизайну додатку було прийнято рішення про реалізацію функціоналу за допомогою IDE Android Studio, адже це дуже популярне та актуальне середовище розробки, що офіційно підтримується Google. Вбудована інтеграція з Android Software Development Kit (SDK) спрощує процес налаштування середовища розробки. Потужні інструменти дозволяють швидко знаходити помилки, адже Android Studio створено на базі IntelliJ IDEA, що забезпечує інтелектуальне автодоповнення коду, рефакторинг і аналіз помилок у реальному часі. Варто зазначити також доволі зручний візуальний редактор макетів, який дозволяє проєктувати інтерфейс за допомогою drag-and-drop, з можливістю перегляду результату на різних пристроях та екранах. Тестування додатку можна проводити не тільки на фізичних пристроях, а і за допомогою внутрішнього емулятору з можливістю налаштування різних версій Android, розмірів екранів і конфігурацій. Android Studio пропонує вбудовані можливості для налагодження, включаючи:

- візуалізацію використання пам'яті та процесора;

- інструменти для аналізу продуктивності додатку (CPU Profiler, Network Profiler, Memory Profiler);

- інтеграцію з тестувальними бібліотеками, такими як Espresso та JUnit.

Ще одним з факторів, чому в якості середовища розробки було обрано саме Android Studio є те, що ця IDE має велику спільноту, отже можна легко знайти відповіді на ті питання, які вас цікавлять. Мова розробки – Java.

4.1 Реалізація екранів для користувачів

Будь-яка сторінка, будь-який додаток має процес завантаження, який в залежності від потужності мобільного пристрою триває декілька мілісекунд, або ж затримується на деякий час. Важливо зробити так, щоб цей процес не відлякував клієнтів, а мав позитивний ефект на потенційних покупців. Саме для цього в додатку MonoSell повністю перероблена анімація для завантаження сторінки. Це створює плавний і привабливий досвід для користувачів додатку.

Файл XML передбачає зовнішній вигляд екрану під час завантаження сторінки. Фон – це контейнер RelativeLayout., який займає весь екран завдяки властивостям match_parent для вказання ширини та довжини контейнеру. В якості фону використовується зображення background_loading_image, яке до цього було завантажено до теки drawable, у верхній частині екрану відображається логотип MonoSell, реалізований за допомогою ImageView, якому присвоюється ID logoLoading . Зручно те, що розмір цього зображення залежить від розміру завантажуваної картинки за рахунок параметрів wrap_content для ширини та довжини. Саме зображення називається logoLoading, а для відображення реального прогресу завантаження сторінки використовується ProgressBar з ID customProgressBar, шириною та довжиною в 100dp. Важливо розуміти, що цей ProgressBar повинен відображати конкретне значення прогресу, тому використовується параметр android:indeterminate="false", він розташований

по центру екрану а для відображення самого прогресу використовується файл custom_progress_bar з теки drawable, що відображено в додатку А.1.

Суть кастомного завантаження полягає в тому, що створюється клас CustomLoadingActivity, який унаслідує AppCompatActivity. Проводиться ініціалізація ProgressBar customProgressBar, а також створення двох масивів з різними кольорами (відтінки червоного, помаранчевого, зеленого, синього тощо) colors та формами (коло, прямокутник, квадрат, прямокутник з круглими углами) shapes. Надалі для змінної customProgressBar присвоюється XML-файл з аналогічною назвою customProgressBar, потім ініціалізуються дві змінні типу int – colorIndex та shapeIndex, в які записується випадкове значення за допомогою метода nextInt, об’єкта класу Random, в яке передається значення розміру кожного з масивів з кольорами та формами за допомогою методу length. Потім для customProgressBar встановлюється випадкова форма за допомогою методу setProgressDrawable(), після чого програється анімація заповнення фігури, яка залежить від прогресу завантаження за допомогою методу animateProgressBar(), що зображено в додатку А.2. Варто не забувати, що анімація програється при переході з одного Activity на інше, тому під час виклику CustomLoadingActivity до нього передається значення цільового Activity, яке присвоюється у змінну типу String targetActivity за допомогою getIntent().getStringExtra("targetActivity"). Для переходу проводиться присвоєння Class<?> targetClass значення Class.forName(targetActivity), що приймає в якості аргументу назву класу, а повертає об’єкт. Далі все по стандарту: створюється Intent, викликається метод startActivity(), а CustomLoadingActivity припиняє роботу завдяки методу finish(). Важливо зазначити, що для уникання помилки використовується блок try{} catch {} [15]. Приклад переходу між Activity з анімацією на лістингу 4.1.

Лістинг 4.1 – Приклад переходу між Activity з викликом нової анімації

```
Intent intent = new Intent(RegisterActivity.this,
CustomLoadingActivity.class);
intent.putExtra("targetActivity", LoginActivity.class.getName());
startActivity(intent);
```

```
overridePendingTransition(android.R.anim.fade_in,
android.R.anim.fade_out);
```

Після налаштування анімації варто переходити до реєстрації нового користувача. Для цього створюється інтерфейс, описаний в XML файлі. Оголошується клас RegistrationActivity, який наслідує AppCompatActivity, що надає підтримку сучасних елементів інтерфейсу. Створюються змінні для зберігання посилань на елементи інтерфейсу (EditText, Button, TextView), firebaseAuth — для аутентифікації користувачів через Firebase, databaseReference — для збереження профілів у базу Firebase Realtime Database, googleSignInClient — об'єкт для роботи з Google Sign-In. Далі ініціалізується Firebase Authentication та посилання на вузол users у Firebase Realtime Database для збереження профілів, що відображено на лістингу 4.2.

Лістинг 4.2 – Ініціалізація змінних для роботи з базою даних Firebase Realtime Database

```
firebaseAuth = FirebaseAuth.getInstance();
databaseReference =
FirebaseDatabase.getInstance().getReference("users");
```

Наступний етап - налаштування Google Sign-In, де GoogleSignInOptions — налаштовує запит на вхід через Google, DEFAULT_SIGN_IN — стандартний режим входу, requestIdToken — потрібний для Firebase аутентифікації, requestEmail — запит електронної пошти користувача. Потім створюється об'єкт googleSignInClient, що відображається на лістингу 4.3.

Лістинг 4.3 - Налаштування Google Sign-In

```
GoogleSignInOptions gso = new
GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
.requestIdToken(getString(R.string.default_web_client_id)).request
Email().build();
googleSignInClient = GoogleSignIn.getClient(this, gso);
```

Наступний крок - ініціалізація елементів інтерфейсу, тобто прив'язка змінних до елементів із XML за допомогою findViewById(). Оскільки в додатку передбачена аутентифікація через Google, потрібно обробляти натискання на кнопку btnGoogleSignIn, що викликає метод signInWithGoogle(), що зображено на лістингу 4.4.

Лістинг 4.4 - Обробник для Google Sign-In та метод signInWithGoogle()

```
btnGoogleSignIn.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
signInWithGoogle();}});
…
private void signInWithGoogle() {
Intent signInIntent = googleSignInClient.getSignInIntent();
startActivityForResult(signInIntent, RC_SIGN_IN);}
```

У разі якщо користувач вже зареєстрований, є можливість входу після натискання на текст TextView завдяки обробнику натискання setOnClickListener() (лістинг 4.5). Коли користувач заповнює всі поля правильно, активується кнопка реєстрації, після натискання на яку клієнт заноситься в базу даних.

Лістинг 4.5 - Обробник переходу на сторінку входу

```
tvLogin.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
startActivity(new Intent(RegistrationActivity.this,
LoginActivity.class));}});
```

Метод signInWithGoogle() створює Intent для Google Sign-In та запускає його. Також проводиться перевизначення оголошення методу onActivityResult () в базовому класі. Спочатку перевіряється, чи requestCode відповідає коду, визначеному для Google Sign-In (RC_SIGN_IN). Це гарантує, що обробляється тільки результат входу через Google. Отримується об'єкт Task для обробки результату входу через Google. Метод

GoogleSignIn.getSignedInAccountFromIntent(data) отримує дані облікового запису з інтенту. Використовується try-блок для спроби отримати успішний результат із об'єкта task. getResult(Exception.class) витягує обліковий запис GoogleSignInAccount, якщо вхід був успішним. Якщо вхід був невдалим, викликається виняток, і програма переходить у блок catch. Перевіряється, чи отриманий обліковий запис не null. Якщо він існує, викликається метод firebaseAuthWithGoogle(account) для продовження аутентифікації користувача через Firebase. Блок catch (Exception e) обробляє будь-який виняток, що виник під час отримання результату Google Sign-In (лістинг 4.6).

Лістинг 4.6 – Перевизначений метод onActivityResult ()

```
if (requestCode == RC_SIGN_IN) {
Task<GoogleSignInAccount> task =
GoogleSignIn.getSignedInAccountFromIntent(data);
try {GoogleSignInAccount account = ask.getResult(Exception.class);
if (account != null) {firebaseAuthWithGoogle(account);}} catch
(Exception e) {…}}}
```

Метод firebaseAuthWithGoogle авторизує користувача в Firebase за допомогою облікового запису Google, якщо успішно, зберігає дані клієнта в базу. Спочатку він який приймає об'єкт GoogleSignInAccount, який містить інформацію про обліковий запис Google. Потім створюється об'єкт AuthCredential для Firebase аутентифікації. Далі викликається метод signInWithCredential, щоб виконати вхід у Firebase з використанням облікових даних, отриманих від Google. Додається слухач (OnCompleteListener), який спрацьовує після завершення процесу аутентифікації. Перевіряється, чи виконання аутентифікації було успішним. Якщо все добре, отримується поточний користувач (FirebaseUser), який щойно увійшов до Firebase. Якщо користувач успішно аутентифікований, тобто firebaseUser != null, викликається метод saveUserProfile, щоб зберегти його дані в базу, де firebaseUser.getUid() — зберігає унікальний ідентифікатор користувача в Firebase, firebaseUser.getDisplayName() — зберігає ім'я користувача (яке вказано у

Google), а firebaseUser.getEmail() — зберігає електронну пошту користувача (лістинг 4.7).

Лістинг 4.7 - Метод firebaseAuthWithGoogle()

```
private void firebaseAuthWithGoogle(GoogleSignInAccount account) {
AuthCredential credential =
GoogleAuthProvider.getCredential(account.getIdToken(), null);
firebaseAuth.signInWithCredential(credential).addOnCompleteListene
r(this, task -> {if (task.isSuccessful()) {
FirebaseUser firebaseUser = firebaseAuth.getCurrentUser();
if (firebaseUser != null) {
saveUserProfile(firebaseUser.getUid(),
firebaseUser.getDisplayName(), firebaseUser.getEmail());}}
…
```

Якщо ж користувач бажає зареєструватися за допомогою електронної пошти та паролю, то викликається метод registerUser() , де отримується текст із поля введення EditText etName (ім'я користувача), конвертується в рядок toString() і обрізається від зайвих пробілів на початку та в кінці trim(). Так відбувається з усіма полями etEmail з електронною поштою та etPassword з паролем. Потім викликається метод Firebase Authentication для створення нового користувача з використанням введених email та пароля – createUserWithEmailAndPassword(email, password). Додається слухач OnCompleteListener, який буде виконуватись після завершення спроби створення користувача. Перевіряється, чи реєстрація користувача була успішною. Якщо task.isSuccessful() повертає true, продовжується виконання в блоці. Отримується об'єкт поточного користувача FirebaseUser, який був створений і аутентифікований у Firebase після успішної реєстрації. Перевіряється, чи об'єкт firebaseUser не є null. Якщо користувач існує, викликається метод saveUserProfile, який зберігає дані користувача (унікальний ідентифікатор firebaseUser.getUid(), ім'я name і електронну пошту email) у базу даних Firebase (лістинг 4.8).

Лістинг 4.8 - Метод registerUser () для реєстрації користувача

```
private void registerUser() {
…
firebaseAuth.createUserWithEmailAndPassword(email,
password).addOnCompleteListener(task -> {
if (task.isSuccessful()) {
FirebaseUser firebaseUser = firebaseAuth.getCurrentUser();
if (firebaseUser != null) {
saveUserProfile(firebaseUser.getUid(), name, email);}}}
```

Метод saveUserProfile() зберігає профіль нового користувача в базу даних Firebase Realtime Database і після успіху переходить на сторінку для входу (LoginActivity). Спочатку створюється новий об'єкт класу User, який є моделлю для збереження даних користувача. Проводиться посилання на вузол бази даних databaseReference, де child(userId) створює підвузол із ім'ям, рівним userId. Це гарантує, що дані кожного користувача зберігаються в окремому вузлі. Далі метод setValue(user) зберігає об'єкт user у цьому вузлі бази даних . Також додається слухач OnCompleteListener, який спрацьовує після завершення спроби зберегти дані в базу. Вікно реєстрації зображено на рисунку 4.1.

Рисунок 4.1 – Вікно реєстрації

Якщо task.isSuccessful() повертає true, викликається новий Intent, щоб перейти до екрана входу LoginActivity, куди передається RegistrationActivity.this, тобто поточний контекст активності реєстрації та LoginActivity.class клас активності входу. Обов’язково поточна активність завершується методом finish(), щоб користувач не міг повернутись до неї натисканням кнопки "Назад". Весь процес відображено на лістингу 4.9.

Лістинг 4.9 - Метод saveUserProfile()

```
private void saveUserProfile(String userId, String name, String
email) {
User user = new User();
databaseReference.child(userId).setValue(user)
.addOnCompleteListener(task -> {
if (task.isSuccessful()) {startActivity(new
Intent(RegistrationActivity.this, LoginActivity.class));
finish();}}}
```

Процес логіну мало чим відрізняється від процесу реєстрації. Проводиться ініціалізація всіх елементів інтерфейсу, Firebase, налаштування Google Sign-In, далі вішається обробник кнопки "Увійти", кнопки Google Sign-In, тексту "Забули пароль?" та "Зареєструйтесь". Метод loginUser виконує авторизацію користувача через Firebase Authentication. Спочатку отримуються введені email та пароль із відповідних текстових полів EditText etEmail та etPassword. Введені дані обрізаються від зайвих пробілів методом trim(). Потім перевіряється, чи ці поля заповнені. Якщо хоча б одне з них порожнє, користувач отримує повідомлення через Toast із текстом "Будь ласка, заповніть всі поля", і виконання методу завершується командою return (лістинг 4.10).

Лістинг 4.10 – Отримання даних та перевірка полів на порожність у методі loginUser()

```
private void loginUser() {
String email = etEmail.getText().toString().trim();
String password = etPassword.getText().toString().trim();
if (email.isEmpty() || password.isEmpty()) {
Toast.makeText(this, "Будь ласка, заповніть всі поля",
Toast.LENGTH_SHORT).show();
```

```
return;}
```

Далі використовується метод Firebase signInWithEmailAndPassword, який приймає email і пароль для спроби входу. Після завершення операції додається слухач OnCompleteListener, який перевіряє, чи вхід був успішним. У разі успіху показується повідомлення "Вхід успішний!". Потім отримується поточний користувач через firebaseAuth.getCurrentUser(). Якщо користувач існує (user != null), у локальних налаштуваннях додатка SharedPreferences зберігається час останнього входу користувача в мілісекундах під ключем lastLoginTime. Це здійснюється за допомогою SharedPreferences.Editor і методу apply(). Після цього виконується перехід на головний екран додатка CatActivity за допомогою Intent, а поточна активність LoginActivity завершується методом finish(), щоб виключити можливість повернення на екран входу (лістинг 4.11).

Лістинг 4.11 - Метод для входу користувача через електронну пошту та пароль loginUser()

```
firebaseAuth.signInWithEmailAndPassword(email, password)
.addOnCompleteListener(task -> {if (task.isSuccessful()) {
Toast.makeText(this, "Вхід успішний!", Toast.LENGTH_SHORT).show();
FirebaseUser user = firebaseAuth.getCurrentUser();
if (user != null) {SharedPreferences prefs =
getSharedPreferences("UserPrefs", MODE_PRIVATE);
SharedPreferences.Editor editor = prefs.edit();
editor.putLong("lastLoginTime", System.currentTimeMillis());
editor.apply();
startActivity(new Intent(LoginActivity.this, CatActivity.class));
finish();}}});}
```

Важливо враховувати, що користувачеві буде незручно постійно вводити свої дані для подальшого входу на сторінку з асортиментом товарів, тому під час логіну додаток записує дату та час останнього входу. При запуску додатку проходить перевірка, чи користувач авторизований у Firebase, а також визначається, чи минув тиждень з моменту останнього входу, щоб вирішити, на який екран перенаправити користувача: екран входу чи головний екран додатку. Спочатку отримується поточний авторизований користувач через

firebaseAuth.getCurrentUser(). Якщо користувач існує за умови currentUser != null, витягується значення часу останнього входу з локальних налаштувань SharedPreferences під ключем "lastLoginTime". Якщо це значення дорівнює 0 відсутня інформація про час входу або з моменту останнього входу минув більше тижня, метод isMoreThanAWeek повертає true. У цьому випадку користувач перенаправляється на екран входу LoginActivity за допомогою startActivity і Intent. Вікно для логіну зображене на рисунку 4.2.

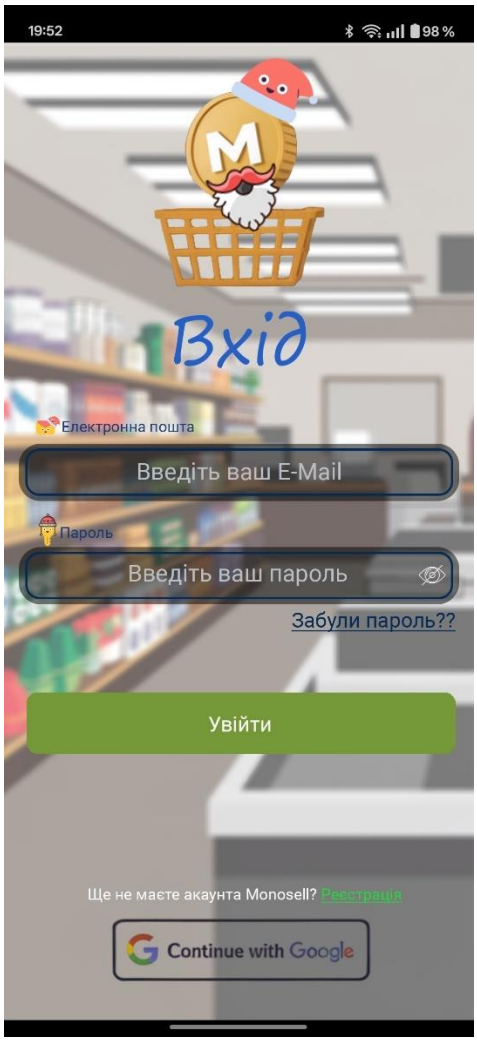

Рисунок 4.2 – LoginActivity

Якщо ж час останнього входу є і з моменту входу не минув тиждень, користувача направляють на головний екран додатку CatActivity. Якщо користувач не авторизований, тобто currentUser == null, також відбувається

перенаправлення на екран входу. Після визначення потрібного екрану завершується поточна активність викликом finish(), щоб користувач не міг повернутися на стартовий екран кнопкою "Назад". Реалізація цієї функції описана на лістингу 4.12.

Лістинг 4.12 - перевірка авторизації та часу останнього входу

```
FirebaseUser currentUser = firebaseAuth.getCurrentUser();
SharedPreferences prefs = getSharedPreferences("UserPrefs",
MODE_PRIVATE);
if (currentUser != null) {long lastLoginTime =
prefs.getLong("lastLoginTime", 0);
if (lastLoginTime == 0 || isMoreThanAWeek(lastLoginTime)) {
startActivity(new Intent(MainActivity.this, LoginActivity.class));
} else {startActivity(new Intent(MainActivity.this,
CatActivity.class));}}
finish(); }
```

Метод isMoreThanAWeek визначає, чи пройшло більше тижня з моменту останнього входу. Спочатку отримується поточний час у мілісекундах через Calendar.getInstance().getTimeInMillis(). Далі обчислюється кількість мілісекунд у тижні (7 * 24 * 60 * 60 * 1000). Повертається результат перевірки, чи різниця між поточним часом і часом останнього входу перевищує тривалість тижня. Реалізація методу відображена на лістингу 4.13.

Лістинг 4.13 - Метод isMoreThanAWeek() для перевірки часу з моменту останнього логіну

```
private boolean isMoreThanAWeek(long lastLoginTime) {
long currentTime = Calendar.getInstance().getTimeInMillis();
long oneWeekInMillis = 7 * 24 * 60 * 60 * 1000;
return currentTime - lastLoginTime > oneWeekInMillis;}
```

4.2 Реалізація екранів каталогу для користувачів

На кожній вкладці застосунку користувачеві доступне бокове меню, яке спрощує навігацію та дозволяє переміщатися між Activity. Це меню відкривається за допомогою кнопки, яка розташована у верхньому лівому кутку, а закривається при свайпі вліво. Спочатку отримуються посилання на відповідні елементи інтерфейсу: drawerLayout (головний контейнер для бічного меню), menuButton (кнопка для відкриття меню), і navigationView (саме бічне меню, що відображає елементи навігації).

Далі додається обробник кліку для кнопки menuButton. При натисканні викликається метод openDrawer(), щоб відкрити меню navigationView, і одночасно приховується кнопка за допомгою методу menuButton.setVisibility(View.GONE), оскільки вона стає неактуальною, поки меню відкрите.

Щоб відновити видимість кнопки після закриття меню, додається слухач DrawerLayout.SimpleDrawerListener. У методі onDrawerClosed викликається menuButton.setVisibility(View.VISIBLE), коли меню закривається. Це забезпечує повернення кнопки на екран для повторного відкриття меню.

Бічне меню navigationView також налаштовується для обробки вибору пунктів. Використовується метод setNavigationItemSelectedListener, де обробляються дії на основі ідентифікатора обраного пункту item.getItemId(). Наприклад, якщо обраний пункт nav_promotions, викликається нова активність PromotionsActivity через startActivity. Після обробки дії меню закривається методом drawerLayout.closeDrawer(navigationView). Бічне меню на головній сторінці зображене на рисунку 4.3.

Рисунок 4.3 – Головна сторінка додатку

Крім того, додається обробник жестів за допомогою setOnTouchListener. Якщо користувач проводить пальцем по екрану MotionEvent.ACTION_MOVE і меню відкрите drawerLayout.isDrawerOpen(navigationView), воно закривається

викликом drawerLayout.closeDrawer(navigationView). Реалізація бокового меню для навігації відображена в додатку А.3.

Для того, щоб користувач міг легко знайти необхідний йому товар на головній сторінці магазину реалізований пошуковий рядок. Спочатку оголошується змінна для пошукового рядка, в який користувач вводить текст, далі змінна для RecyclerView, що відповідає за відображення списку знайдених товарів, потім адаптер для RecyclerView, який заповнює його елементами Product, а також локальний список, який зберігає товари, що відповідають введеному тексту.

Наступний крок це встановлення менеджеру компоновки для списку resultsList. LinearLayoutManager організовує елементи списку у вигляді вертикального списку. Далі створюється новий адаптер ProductAdapter, який буде відповідати за відображення елементів списку. productList — це дані (масив), які будуть використовуватися для наповнення елементів списку. За допомогою методу resultsList.setAdapter(productAdapter), адаптер productAdapter прив'язується до resultsList, забезпечуючи механізм відображення даних зі списку productList у візуальних елементах списку. Далі в змінну DatabaseReference databaseReference за допомогою методу FirebaseDatabase.getInstance().getReference("products") заноситься посилання на вузол "products" у базі даних Firebase. За допомогою методу addTextChangedListener() текстового поля searchBar додається обробник подій TextWatcher. Він слідкує за змінами тексту в полі і виконує певний код при цих змінах. В даному випадку відбувається перевизначення методу onTextChanged(), який викликається під час зміни тексту. Змінна query типу String за допомогою методів toString().toLowerCase().trim() отримує текст із searchBar, переведений у нижній регістр і обрізаний від пробілів trim(). Далі проводиться перевірка if (!query.isEmpty()), якщо рядок query не порожній, виконується пошук. Метод resultsList.setVisibility(View.VISIBLE) робить список видимим, коли є текст для пошуку. Далі на змінну бази даних databaseReference методом addListenerForSingleValueEvent() додається одноразовий обробник подій, який

виконується при завантаженні даних з Firebase. При цьому в методі onDataChange(), який викликається, коли дані отримано, очищається список productList, проходить ітерація через всі дочірні елементи вузла products, кожен елемент конвертується у об'єкт класу Product і якщо назва продукту містить текст із query, продукт додається до списку. Метод productAdapter.notifyDataSetChanged() оновлює адаптер, щоб список відобразив нові дані. Реалізація пошуку відображається на лістингу 4.14.

Лістинг 4.14 – пошуковий рядок для покупця

```
searchBar.addTextChangedListener(new TextWatcher() {
@Override
public void onTextChanged(CharSequence s, int start, int before,
int count) {String query = s.toString().toLowerCase().trim();
if (!query.isEmpty()) {resultsList.setVisibility(View.VISIBLE);
databaseReference.addListenerForSingleValueEvent(new
ValueEventListener() {@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
productList.clear();
for (DataSnapshot productSnapshot : snapshot.getChildren()) {
Product product = productSnapshot.getValue(Product.class);
if (product != null &&
product.getName().toLowerCase().contains(query)) {
productList.add(product);}}
productAdapter.notifyDataSetChanged();}
```

Щоб привабити покупця використовується динамічна полоса з банерами, де відображаються актуальні акційні пропозиції. Це реалізується завдяки RecyclerView, спеціальному адаптеру BannerAdapter та списку List<String> bannerImages. Ініціалізуються всі елементи, надалі завантажуються актуальні картинки з хмарної бази даних Firebase завдяки методу loadBannersFromFirebase() та відбувається автоматична прокрутка завдяки startAutoScroll(). Метод loadBannersFromFirebase() завантажує банерні зображення з бази даних Firebase у список для подальшого відображення. Спочатку створюється посилання databaseReference на вузол banners за допомогою методу FirebaseDatabase.getInstance().getReference("banners") у базі

Firebase, який містить необхідні дані. Потім для цього посилання додається обробник події addListenerForSingleValueEvent.

У методі onDataChange, який викликається після успішного отримання даних, відбувається ітерація через всі дочірні елементи вузла banners через цикл for (DataSnapshot data : snapshot.getChildren()). Кожен дочірній елемент інтерпретується як рядок (тип String), що є URL-адресою банерного зображення. Ці URL-адреси додаються до списку bannerImages. Після заповнення списку викликається bannerAdapter.notifyDataSetChanged(), щоб повідомити адаптер про оновлення даних і забезпечити їх відображення у відповідному компоненті інтерфейсу. Реалізація відображена на лістингу 4.15.

Лістинг 4.15 – Завантаження банерів з Firebase

```
private void loadBannersFromFirebase() {
DatabaseReference databaseReference =
FirebaseDatabase.getInstance().getReference("banners");
databaseReference.addListenerForSingleValueEvent(new
ValueEventListener() {@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
for (DataSnapshot data : snapshot.getChildren()) {
String imageUrl = data.getValue(String.class);
bannerImages.add(imageUrl);}
bannerAdapter.notifyDataSetChanged();}});}
```

В методі startAutoScroll() для прокрутки банеру спочатку створюється об'єкт Runnable під назвою scrollRunnable, що містить код для виконання повторюваної задачі. У методі run цього об’єкта за допомогою конструкції if (bannerImages != null && !bannerImages.isEmpty()) перевіряється, чи список bannerImages не є порожнім і чи він ініціалізований. Якщо умова виконується, то залежно від поточної позиції банера визначається наступний індекс для прокрутки.

Далі проводиться перевірка if (currentIndex == bannerImages.size() - 1) , тобто якщо currentIndex (індекс поточного банера) дорівнює останньому індексу у списку, то він скидається до нуля currentIndex = 0, і bannerRecyclerView прокручується до початку методом scrollToPosition. В іншому випадку

currentIndex збільшується на одиницю, і відбувається плавна прокрутка до наступного елемента за допомогою smoothScrollToPosition. Після виконання прокрутки Runnable планується повторно через 5000 мілісекунд за допомогою handler.postDelayed, забезпечуючи періодичність завдання. Метод handler.postDelayed(scrollRunnable, 5000) у самому кінці планує початковий запуск задачі через 5 секунд після виклику startAutoScroll (лістинг 4.16).

Лістинг 4.16 – Автоматична прокрутка банерів

```
private void startAutoScroll() {
Runnable scrollRunnable = new Runnable() {@Override
public void run() {if (bannerImages != null &&
!bannerImages.isEmpty()) {if (currentIndex == bannerImages.size()
- 1) {currentIndex = 0;
bannerRecyclerView.scrollToPosition(currentIndex);
} else {currentIndex++;
bannerRecyclerView.smoothScrollToPosition(currentIndex);}}
handler.postDelayed(this, 5000);} };
handler.postDelayed(scrollRunnable, 5000);}
```

Найважливішим елементом магазину є каталог продукції. Тут він представлений у вигляді RecyclerView, адаптеру. Для того, щоб не займати зайве місце у пам’яті смартфону користувача вся інформація про продукти береться з хмарної бази даних за допомогою методу loadProductsFromFirebase(). Він завантажує список продуктів із бази даних Firebase і оновлює інтерфейс для їх відображення. Спочатку створюється посилання на вузол products у Firebase за допомогою об’єкту класу DatabaseReference та методу FirebaseDatabase.getInstance().getReference("products"). Потім для цього посилання додається обробник події addListenerForSingleValueEvent, який дозволяє одноразово отримати дані з бази. Інтерфейс каталогу представлено на рисунку 4.4.

Рисунок 4.4 – Інтерфейс головної сторінки додатку

У методі onDataChange (), який викликається після успішного отримання даних, виконується очищення списку allProducts для запобігання дублюванню даних. Далі відбувається ітерація через всі дочірні елементи вузла products у базі за допомогою циклу for (DataSnapshot data : snapshot.getChildren()). Кожен елемент перетворюється у об'єкт класу Product за допомогою методу getValue. Якщо об’єкт product не є порожнім, він додається до списку allProducts.

Після завершення заповнення списку викликається productAdapter.notifyDataSetChanged(), що сигналізує адаптеру про оновлення даних і забезпечує їх відображення в інтерфейсі RecyclerView. Реалізація представлена на лістингу 4.17.

Лістинг 4.17 - Завантаження даних з Firebase

```
private void loadProductsFromFirebase() {
```

```
DatabaseReference databaseReference =
FirebaseDatabase.getInstance().getReference("products");
databaseReference.addListenerForSingleValueEvent(new
ValueEventListener() {@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
allProducts.clear();
for (DataSnapshot data : snapshot.getChildren()) {
Product product = data.getValue(Product.class);
if (product != null) allProducts.add(product);}
productAdapter.notifyDataSetChanged();}});}
```

Після того як користувач обрав бажані товари та додав їх у кошик, він може натиснути на кнопку та перейти до оформлення замовлення. Коли користувач натискає кнопку, викликається метод setOnClickListener, що задає обробник події для кліку. Всередині обробника спочатку створюється посилання cartRef на вузол currentOrder у Firebase Realtime Database, де зберігається поточне замовлення. Далі отримується унікальний ідентифікатор товару productId за допомогою методу product.getId(). Початкова кількість товару в замовленні встановлюється як 1, і для передачі даних створюється об’єкт типу Map, який зберігає ключі та значення. У цьому випадку в cartItem додаються чотири поля: productId (ідентифікатор товару), weight (вага товару), quantity (кількість) та price (ціна товару), яку отримано з об’єкта product.

Потім для збереження цього товару у базі даних використовується метод setValue із посиланням на вузол, визначений за ідентифікатором товару методом cartRef.child(productId). Після виконання операції додається слухач addOnCompleteListener, який перевіряє, чи операція була успішною. Якщо збереження завершилося успішно, викликається метод Toast.makeText для виведення повідомлення "Товар додано до кошика". У разі помилки аналогічним чином виводиться повідомлення "Помилка додавання товару" (лістинг 4.18).

Лістинг 4.18 – Процес додавання товару до кошику

```
btnAddToCart.setOnClickListener(v -> {
DatabaseReference cartRef =
FirebaseDatabase.getInstance().getReference("currentOrder");
String productId = product.getId();
int quantity = 1;
```

```
Map<String, Object> cartItem = new HashMap<>();
cartItem.put("productId", productId);
cartItem.put("quantity", quantity);
cartItem.put("weight", weight);
cartItem.put("price", product.getPrice());
cartRef.child(productId).setValue(cartItem).addOnCompleteListener(
task -> {
if (task.isSuccessful()) {
Toast.makeText(context, "Товар додано до кошика",
Toast.LENGTH_SHORT).show();
} else {Toast.makeText(context, "Помилка додавання товару",
Toast.LENGTH_SHORT).show();}});});
```

Для відображення продукції використовується знайомий RecyclerView, але цього разу до адаптеру додаються методи для обробки натискання на кнопку "+", обробки натискання на кнопку "-", та обробки натискання на кнопку видалення. Обов’язково проводиться перевірка на те, чи пустий список обраних товарів, чи ні, та в залежності від цього або виводиться повідомлення про помилку, або ж ініціалізується адаптер, та проводяться остаточні обчислення суми за допомогою методу calculateTotals().

4.3 Реалізація екрану з мапою для користувачів

Коли користувач підтверджує замовлення, його перенаправляє на сторінку з мапою, де чітко відображається позиція покупця, позиція кур’єра та маршрут, за яким їде або йде кур’єр. Для більш приємної кастомізації використовуються іконки зі вказаним транспортним засобом, яким рухається доставка. Для реалізації цієї функції спочатку завантажуються іконки у проєкт. Також для роботи з актуальною геолокацією до AndroidManifest.xml додаються запити на дозвіл android:name="android.permission.ACCESS_FINE_LOCATION" та android:name="android.permission.ACCESS_COARSE_LOCATION".

Позиція кур’єра та клієнта зберігається в режимі реального часу у базу даних Firebase Database. Реалізується ця функція за допомогою методу

startLocationUpdates(). Спочатку створюється об'єкт FusedLocationProviderClient за допомогою класу LocationServices, який є частиною Google Play Services. Цей клієнт надає API для роботи з геолокацією, підтримуючи високу точність і оптимізацію використання батареї.

Далі створюється об'єкт LocationRequest, який налаштовує параметри запиту місцезнаходження. Викликається метод create() для створення нового екземпляра. Встановлюється інтервал запитів у 5000 мілісекунд (5 секунд), який визначає бажану частоту оновлення координат. Вказується також найшвидший можливий інтервал в 2000 мілісекунд (2 секунди), щоб дозволити більш часті оновлення, якщо інші додатки також отримують дані про місцезнаходження. Метод setPriority() встановлює пріоритет точності на PRIORITY_HIGH_ACCURACY, що дозволяє використовувати GPS для визначення координат.

Метод requestLocationUpdates підписується на оновлення місцезнаходження відповідно до заданих параметрів. Цей метод приймає об'єкт LocationRequest, екземпляр LocationCallback, і Looper. У LocationCallback реалізується метод onLocationResult, який викликається при кожному оновленні координат. Перевіряється, чи результат не є null, щоб уникнути помилок. Отримані місцезнаходження обробляються циклом for, де з об'єкта Location витягуються широта (latitude) та довгота (longitude).

Отримані координати передаються у метод updateLocationToDatabase, який оновлює їх у Firebase Realtime Database. Цей механізм дозволяє постійно зберігати актуальне місцезнаходження пристрою в базі даних. Нарешті, передається Looper.getMainLooper(), щоб операції викликалися в основному потоці програми.

В UserMapActivity імплементується інтерфейс OnMapReadyCallback, який необхідний для взаємодії з картою після її ініціалізації. Далі ініціалізуються всі необхідні змінні GoogleMap mMap, Marker userMarker, Marker courierMarker, Polyline courierPath та база даних DatabaseReference databaseReference. Перша змінна відповідає за мапу, далі йдуть два маркери для користувача та кур’єра,

шлях кур’єра до користувача. Отримується карта через фрагмент SupportMapFragment, який вказаний у макеті через ID map. Потім викликається метод getMapAsync(), щоб підготувати карту, після чого onMapReady() буде викликаний автоматично.

Цей метод викликається, коли карта готова до використання. У ньому зберігається об'єкт карти в змінній mMap та викликаються методи для відстеження місцезнаходження користувача (trackUserLocation) і кур’єра (trackCourierLocation). В методі trackUserLocation() додається слухач до вузла usersLocation у Firebase. Кожного разу, коли дані змінюються, викликається onDataChange, далі зчитуються координати latitude і longitude з бази даних і створюється об'єкт LatLng, який зберігає ці координати. Якщо маркера ще немає, він створюється з заданою позицією, назвою ("Ваша локація") та синім кольором. Якщо маркер вже є, оновлюється його позиція. Камера карти переміщується до координат користувача з масштабом 15.

В методі trackUserLocation() додається слухач до вузла couriersLocation у Firebase, зчитуються координати кур’єра та його тип транспорту (transportMode), далі Маркер кур’єра створюється, якщо його ще немає, або оновлюється позиція та іконка, яка залежить від способу транспорту. До списку courierPathPoints додається нова точка маршруту. Якщо лінія маршруту вже існує, вона видаляється, і створюється нова з оновленими точками. Для зміни іконки в залежності від типу транспорту використовується метод getTransportIcon(). Реалізація роботи з картою відображена в додатку А.4.

Для найбільш зручного та приємного досвіду користування додатком MonoSell, час доставки розраховується сервісами Google з урахуванням завантаженості трафіку. Для цього використовується метод calculateTime, в параметри якого входять чотири змінні double, в які заносяться довгота та ширина початкової та кінцевої точки. В самому методі створюється дві змінні типу String, це String apiKey та String url для зберігання API додатку і тексту запиту HTTP, що відображено на лістингу 4.19.

Лістинг 4.19 – Ініціалізація змінних у методі для визначення часу доставки

```
private void calculateTime(double startLat, double startLng,
double endLat, double endLng) {
String apiKey = " AIzaSyD-xKM2Bzmt4jG6kT1p8z8krQs7WoVrFq8 ";
String url =
"https://maps.googleapis.com/maps/api/directions/json?" +
"origin=" + startLat + "," + startLng +
"&destination=" + endLat + "," + endLng +
"&departure_time=now" +
"&key=" + apiKey;
…}
```

Для керування чергою запитів використовується бібліотека Volley. Загалом, спочатку створюється об’єкт RequestQueue для відправлення HTTP-запитів. Потім створюється JsonObjectRequest, щоб надіслати GET-запит до API, в якому використовуються такі параметри, як Request.Method.GET — вказує, що це GET-запит, url — сформована URL-адреса, null — вказує, що тіло запиту відсутнє (для GET-запитів воно не потрібне). В блоці response обробляється відповідь API, де routes — масив маршрутів, а leg — перший елемент масиву "legs", який описує окремий сегмент маршруту. Далі Отримується час подорожі з урахуванням дорожнього руху. Поле duration_in_traffic містить час у текстовому форматі (наприклад, "25 mins"). Знаходиться елемент TextView із id і встановлюється текст, що показує шлях доставки з урахуванням навантаженості трафіку. Наостанок додається створений запит до черги requestQueue, щоб він виконався (лістинг 4.20).

Лістинг 4.20 – Метод для розрахування часу доставки та виведення у текстовому форматі

```
private void calculateTime(double startLat, double startLng,
double endLat, double endLng) {
…
RequestQueue requestQueue = Volley.newRequestQueue(this);
JsonObjectRequest jsonObjectRequest = new
JsonObjectRequest(Request.Method.GET, url, null,
response -> {
try {JSONArray routes = response.getJSONArray("routes");
JSONObject leg =
routes.getJSONObject(0).getJSONArray("legs").getJSONObject(0);
```

```
String durationInTraffic =
leg.getJSONObject("duration_in_traffic").getString("text");
TextView timeToDelivery = findViewById(R.id. timeToDelivery);
timeToDelivery.setText("Час доставки: " + durationInTraffic);}
requestQueue.add(jsonObjectRequest);
}}
```

Інтерфейс карти користувача представлено на рисунку 4.5.

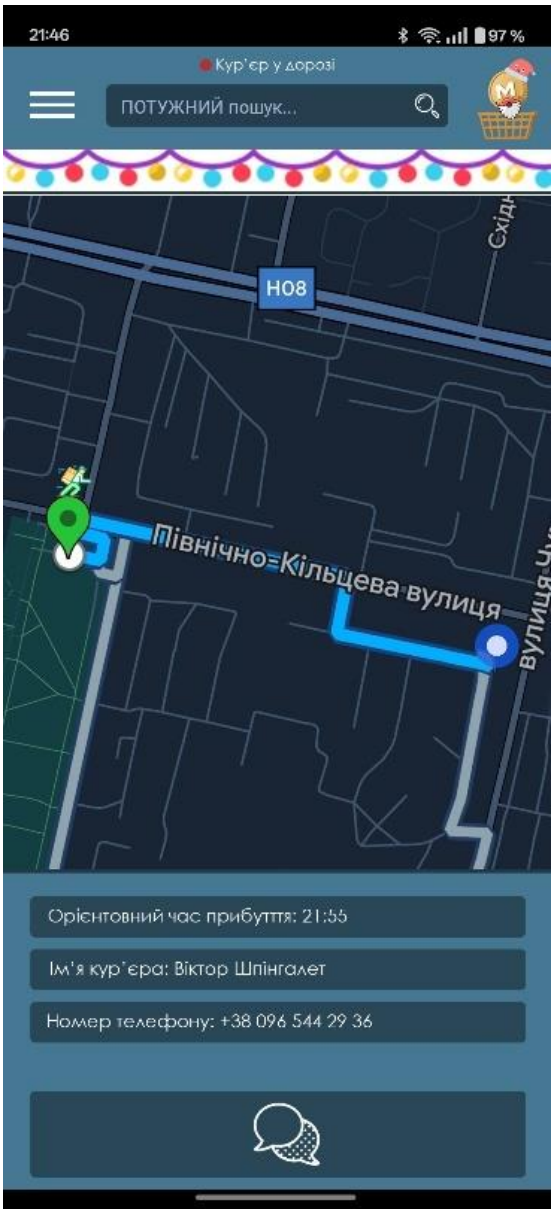

Рисунок 4.5 – Мапа користувача

В кожного користувача є можливість переглянути історію своїх замовлень для підвищення фінансової обізнаності. Для цього метод saveOrderToHistory()

зберігає інформацію про кожне замовлення у базі даних, а метод loadOrderHistory() відображає історію замовлень у зручному RecyclerView з прив’язкою до ID користувача. Метод saveOrderToHistory відповідає за перенесення активного замовлення з вузла currentOrder до вузла orderHistory в Firebase Realtime Database для вказаного користувача. Спочатку створюється посилання на вузол користувача у базі даних за допомогою DatabaseReference. Далі здійснюється разове зчитування даних з вузла currentOrder за допомогою методу addListenerForSingleValueEvent. У разі наявності активного замовлення (snapshot.exists() повертає істину) зчитується поточний об'єкт замовлення як Map<String, Object>. Потім генерується унікальний ідентифікатор для нового запису в orderHistory за допомогою push().getKey(). Якщо ідентифікатор успішно створено, то активне замовлення копіюється у вузол orderHistory із цим ідентифікатором. Після успішного збереження викликається метод removeValue, який видаляє активне замовлення з вузла currentOrder. У разі невдачі при збереженні відображається повідомлення про помилку за допомогою Toast. Якщо активного замовлення немає, користувачу виводиться відповідне повідомлення.

Метод loadOrderHistory відповідає за зчитування історії замовлень з вузла orderHistory у Firebase для заданого користувача. Створюється посилання на відповідний вузол у базі даних за допомогою DatabaseReference. Далі метод addValueEventListener додає слухач змін у цьому вузлі. У методі onDataChange зчитуються всі дочірні елементи orderHistory. Для кожного дочірнього елемента створюється об'єкт класу Order шляхом перетворення отриманих даних за допомогою методу getValue(Order.class). Потім ці об'єкти додаються до списку orderList. Після завершення зчитування список замовлень передається адаптеру orderHistoryAdapter для відображення у відповідному інтерфейсі. У разі помилки при зчитуванні даних викликається метод onCancelled, де через Toast відображається повідомлення про помилку завантаження. Код реалізації написаний в додатку А.5.

4.4 Реалізація екрану підтримки для користувачів

Якщо у користувача виникнуть питання, або він має свої пропозиції чи скарги, то він завжди може легко звернутися до служби підтримки за допомогою кнопки виклику чату, яка відкриває екран ChatActivity. На цьому етапі користувач може з легкістю поспілкуватися з представником служби підтримки та отримати відповіді на вся питання, які його цікавлять. Для того, щоб спілкування було прозорим і не було питань щодо неякісного обслуговування або некоректної відповіді користувач та представник служби підтримки бачать статус діалогу один у одного. Реалізується це за рахунок хмарної бази даних Firebase. Коли представник служби підтримки заходить у додаток, то його статус автоматично змінюється на «онлайн», коли відкриває чат, змінюється на «у чаті», а коли він не у додатку, то його статус відображається як «офлайн». Щоб забезпечити подібний функціонал потрібно враховувати життєвий цикл додатку та чату. Для цього проводиться перевизначення методів onStart() та onStop() на головному екрані та на ChatActivity (лістинг 4.21).

Лістинг 4.21 – Реалізація відображення актуального статусу для користувача

```
@Override
protected void onStart() {
super.onStart();
setOnlineStatus();}
@Override
protected void onStop() {
super.onStop();
setOfflineStatus();}
private void setOnlineStatus() {
supportRef.child("status").setValue("online");
supportRef.child("currentChat").setValue(null);}
private void setOfflineStatus() {
supportRef.child("status").setValue("offline");
supportRef.child("currentChat").setValue(null);}}
```

На самому ChatActivity користувач та представник служби підтримки спілкуються завдяки трьом методам - loadMessages(), monitorSupportStatus() та sendButton.setOnClickListener(v -> sendMessage()). Тобто повідомлення завантажується для перегляду чату, плюс зберігається історія у режимі реального часу, проводиться моніторинг статусу підтримки та додається можливість надсилання повідомлення.

Метод loadMessages відповідає за завантаження повідомлень із Firebase Realtime Database. Алгоритм його роботи доволі простий.

- chatRef.addValueEventListener додає слухач змін до вузла чату в базі даних;

- у методі onDataChange здійснюється очистка списку messages (щоб уникнути дублювання);

- далі за допомогою циклу for зчитуються всі дочірні вузли з даними повідомлень у форматі DataSnapshot;

- для кожного з них створюється об’єкт класу Message, і цей об’єкт додається до списку messages;

- chatAdapter.notifyDataSetChanged повідомляє адаптер, що дані змінились, і список потрібно оновити;

- messagesRecyclerView.scrollToPosition(messages.size() - 1) автоматично прокручує список до останнього повідомлення;

- якщо зчитування даних не вдалось, у onCancelled через Toast відображається повідомлення про помилку.

Метод monitorSupportStatus відповідає за моніторинг статусу служби підтримки. Метод supportStatusRef.child("status").addValueEventListener додає слухач до вузла зі статусом підтримки в базі даних, у onDataChange зчитується значення вузла status, перетворюється на рядок (String status) і відображається у текстовому полі supportStatus. Якщо статус є null, замість нього відображається текст "Офлайн". Якщо виникає помилка, метод onCancelled також встановлює статус "Офлайн".

Метод sendMessage реалізує відправку повідомлення. Спершу з текстового поля messageInput зчитується введений текст і видаляються зайві пробіли на початку та в кінці (trim), якщо текст не порожній, генерується унікальний ідентифікатор для повідомлення через chatRef.push().getKey().Далі створюється об’єкт Message, що включає текст, автора ("user"), статус повідомлення ("sent"), і часову мітку (System.currentTimeMillis). Якщо messageId успішно створений, повідомлення додається до вузла чату у Firebase через chatRef.child(messageId).setValue(message). У разі успіху вміст текстового поля очищується (messageInput.setText("")). У разі невдачі користувач отримує повідомлення про помилку через Toast.

Загалом, зовнішній вигляд екрану з чатом для користувачів представлено на рисунку 4.6.

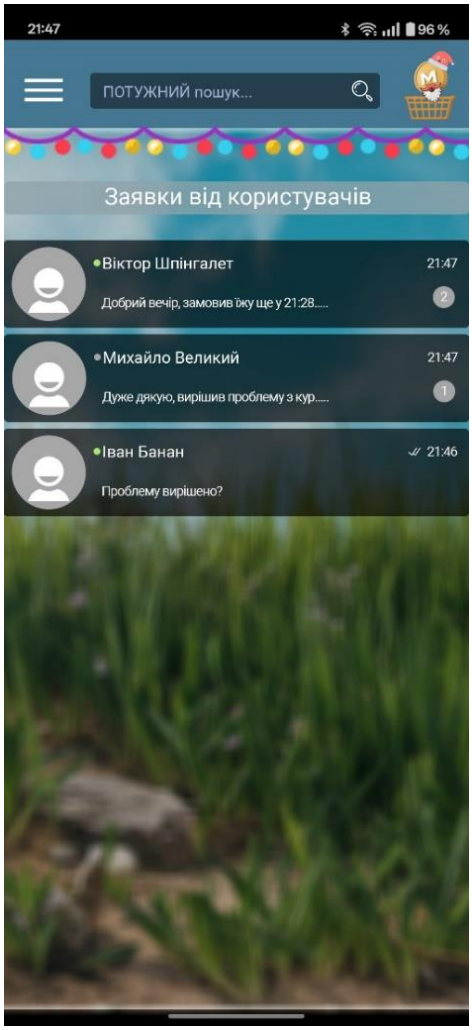

Рисунок 4.6 – Екран з чатами для представників служби підтримки

4.5 Реалізація екранів додатку для кур’єра

Функціонал для кур’єра передбачає аналогічну роботу з мапами від Google, проте має декілька особливостей. В залежності від налаштувань, де кур’єр вказує свій вид транспорту, йому на карті відмічаються лише ті маркери, які знаходяться в певному радіусі (для автомобілістів радіус становить 10 км, для володарів велосипедів 5-6 кілометрів, для пішої доставки до 2 км). Також особливістю є те, що при підтвердженні доставки замовлення воно зникає у інших кур’єрів. Для реалізації такого функціоналу до кожного замовлення додається його статус, а ID замовлення пов’язується з ID кур’єра.

Для початку створюється метод private double getRadius(String transportType), який визначає радіус в залежності від обраного транспортного засобу. Код цього методу відображено на лістингу 4.22.

Лістинг 4.22 – Метод визначення допустимого радіусу доставки

```
private double getRadius(String transportType) {
switch (transportType) {
case "foot": return 2000; // 2 км
case "bike": return 6000; // 6 км
case "car": return 10000; // 10 км
default: return 1000;}}
```

Наступний крок – створення методу calculateDistance(), який може розрахувати відстань між двома точками за формулою гаверсінуса (лістинг 4.23).

Лістинг 4.23 – Метод визначення відстані між двома точками

```
private double calculateDistance(double lat1, double lon1, double
lat2, double lon2) {
final int EARTH_RADIUS = 6371000; // Радіус Землі в метрах
double latDistance = Math.toRadians(lat2 - lat1);
double lonDistance = Math.toRadians(lon2 - lon1);
double a = Math.sin(latDistance / 2) * Math.sin(latDistance / 2) +
Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2)) *
Math.sin(lonDistance / 2) * Math.sin(lonDistance / 2);
double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
```

```
return EARTH_RADIUS * c;}
```

Після створення цих допоміжних методів можна переходити до реалізації особливого функціоналу для кур’єрів – loadOrdersWithinRadius(), що дозволить відображати точки замовлень лише в певному радіусі та loadOrdersOnMap() що фільтрує замовлення та видаляє їх у випадку, якщо хтось інший прийняв його.

Метод loadOrdersWithinRadius приймає courierId (ідентифікатор кур'єра), щоб визначити його поточне місцезнаходження і завантажити відповідні замовлення. Далі створюється посилання на вузол бази даних FirebaseDatabase.getInstance().getReference("couriers").child(courierId), що відповідає конкретному кур'єру, використовуючи його ідентифікатор, потім створюється посилання на вузол бази даних із замовленнями. Після цього додається однократний слухач addListenerForSingleValueEvent для отримання даних про кур'єра. У методі onDataChange перевіряється, чи існують дані для кур'єра: if (!snapshot.exists()) return;. Якщо дані відсутні, метод завершиться. Далі для повного функціоналу зчитується тип транспорту кур'єра (наприклад, "пішки", "автомобіль"). Після цього зчитуються координати місцезнаходження кур'єра (широта і довгота). Визначається радіус доступності замовлень залежно від типу транспорту (функція getRadius повертає значення у метрах). Додається слухач змін до вузла замовлень, щоб відслідковувати всі доступні замовлення в реальному часі. У методі onDataChange для кожного замовлення виконується цикл for (DataSnapshot orderSnapshot : ordersSnapshot.getChildren()) { ... }, де перебираються всі замовлення у вузлі бази даних. Надалі кожен дочірній вузол перетворюється на об'єкт класу Order. Перевіряється if (order != null && "available".equals(order.getStatus())) { ... }, чи замовлення існує і чи його статус дорівнює "available" (доступне для доставки). Потім розраховується відстань між кур'єром і замовленням за допомогою функції calculateDistance і результат заносить до змінної double distance. Перевіряється if (distance <= radius) { ... } чи знаходиться замовлення в межах радіусу доставки. Створюється об'єкт LatLng location = new LatLng(orderLat, orderLon) для представлення координат

замовлення. Фінальний етап - googleMap.addMarker(new MarkerOptions()...), тобто на карті додається маркер для замовлення, встановлюється позиція маркера, вказується адреса замовлення як заголовок, в описі (snippet) відображається вага замовлення і відстань до нього та до маркера додається тег із ідентифікатором замовлення order.getOrderId() (Додаток А.7).

Наступний крок – створення методу acceptOrder() для прийняття замовлення, яке пов’язує кур’єра та оновлення статусу замовлення (лістинг 4.24).

Лістинг 4.24 – Метод для прийняття замовлення

```
private void acceptOrder(String orderId, Marker marker) {
String courierId = "12";
ordersRef.child(orderId).updateChildren(new HashMap<String,
Object>() {{
put("status", "accepted");
put("courierId", courierId);
}}).addOnSuccessListener(aVoid -> {
Toast.makeText(getContext(), "Замовлення прийнято",
Toast.LENGTH_SHORT).show();
}).addOnFailureListener(e -> {
Toast.makeText(getContext(), "Помилка прийняття замовлення",
Toast.LENGTH_SHORT).show();});}
```

Далі відбувається фільтрація замовлень для кожного кур’єра, що реалізується за допомогою методу loadOrdersOnMap(). Метод loadOrdersOnMap приймає параметр courierId, який ідентифікує кур'єра. На основі цього ідентифікатора метод визначає, які замовлення показати на карті. Додається слухач змін до вузла замовлень у Firebase ordersRef.addValueEventListener(new ValueEventListener() { ... }), який буде реагувати на будь-які оновлення замовлень у реальному часі.

У методі onDataChange виконується очищення карти за допомогою googleMap.clear();, щоб прибрати попередні маркери перед завантаженням нових. Цикл for (DataSnapshot orderSnapshot : snapshot.getChildren()) { ... } перебирає всі замовлення, отримані з бази даних. Кожен дочірній вузол замовлення зчитується у змінну order. Order order =

orderSnapshot.getValue(Order.class), що перетворює об'єкт замовлення з формату Firebase у клас Order, щоб зручно працювати з його властивостями.

Перевіряється, чи об'єкт order не є порожнім, за допомогою if (order != null). Якщо дані замовлення існують, код продовжує обробку. Потім зчитуються статус замовлення (available, accepted тощо) і ідентифікатор призначеного кур'єра. Далі перевіряється, чи замовлення доступне: if ("available".equals(status)). Якщо так, додається маркер на карту:

- координати замовлення створюються як об'єкт LatLng: LatLng location = new LatLng(order.getLatitude(), order.getLongitude());

- на карту додається маркер за допомогою googleMap.addMarker(new MarkerOptions().position(location).title(order.getAddress()).snippet("Вага: " + order.getWeight() + " кг"));

- замовленню додається унікальний тег, його ідентифікатор: .setTag(order.getOrderId());

- якщо замовлення має статус accepted і призначене цьому кур'єру (courierId.equals(assignedCourier)), також додається маркер;

- координати обчислюються аналогічно;

- у заголовку маркера вказується адреса, а у підказці – текст "Ваше замовлення".

Метод реалізації відображено в додатку А.8.

Для зручності відображення інформації про замовлення передбачається візуалізація зони доступності для кур’єра у вигляді кола на мапі (лістинг 4.25).

Лістинг 4.25 - Візуалізація зони доступності

```
private void displayAvailabilityZone(double latitude, double
longitude, double radius) {
if (availabilityZone != null) {
availabilityZone.remove();}
availabilityZone = googleMap.addCircle(new
CircleOptions().center(new LatLng(latitude,
longitude)).radius(radius).strokeColor(Color.BLUE).fillColor(Color
.argb(50, 0, 0, 255)));}
```

Окремо варто відзначити Activity для представника підтримки, яке включає в себе список всіх доступних запитань, при цьому, натискаючи на чат відкривається діалогове вікно, яке після завершення діалогу зникає у інших представників служби підтримки (додаток А.9). Прокрутка можлива завдяки класу RecyclerView. Для більшої зручності створюється модель як повідомлення, так і чату загалом.

Вся інформація зберігається у хмарному сховищі (додаток А.10). При цьому у випадку виникнення скарг можна легко відстежити неналежне виконання обов'язків, адже всі чати зберігають ID користувача та представника служби підтримки. Важливо також розуміти, що для спілкування в чаті потрібно щоб обидва користувачі пройшли автентифікацію. Для цього робиться перевірка у методі onStart() для кожного Activity, пов'язаного з чатом (додаток А.11). Також після закінчення чату користувачу надається можливість оцінити ефективність та написати відгук за допомогою Dialog (додаток А.12).

4.6 Реалізація екранів додатку для адміністратора

В обов'язки адміністратору входить редагування інформації про товар, тож при вході в режимі адміністратору користувач отримує доступ до перегляду списку товарів за категоріями, при цьому натискаючи на товар відкривається Dialog, в якому адміністратор може вносити зміни в характеристиках, які автоматично будуть відправлятися в базу даних (додаток А.13).

Також адміністратор може додавати категорії, видаляти їх, додавати продукцію та видаляти її. Для цього біля кожної категорії є кнопка для додавання продукції, що викликає метод openAddProductDialog(), відображений в додатку А.14, а також кнопка для видалення категорії, що викликає метод deleteCategory(), відображений в додатку А.15. А біля кожного продукту є кнопка для його видалення. В самому кінці є кнопка для додавання категорії, що

викликає метод openAddCategoryDialog() (додаток А.16). Оновлення бази даних відбувається після внесення будь-яких змін за допомогою методу loadCategories() (додаток А.17).

В фінальній версії застосунку адміністратор може працювати зі списком працівників, їх ролями, доступом до функціоналу (додаток А.18). Також йому надається можливість змінювати мапу і при цьому додавати нові точки видачі на карту. Якщо ж адміністратор бажає внести корективи в акційні пропозиції, тобто додати продукцію, зменшити або збільшити відсоток знижки, тощо, то він зможе легко це зробити за допомогою вкладки з налаштуваннями акцій. За допомогою кнопки з налаштуваннями додатку адміністратор може внести певні корективи у відображення інформації, зменшити чи збільшити розмір елементів інтерфейсу. Зовнішній вигляд екрану для адміну представлено на рисунку 4.7.

Рисунок 4.7 – Екран налаштувань для адміністратора

ВИСНОВКИ

У сучасному світі жоден магазин не буде конкурентоспроможним, якщо не матиме власного додатку. Оскільки комерційна сфера постійно розвивається, важливо створювати сучасні та актуальні застосунки для найбільш популярної в Україні ОС Android, які будуть відповідати вимогам користувачів.

Для правильного розподілення ресурсів і створення найбільш унікального та зручного додатку було досліджено сучасні тенденції застосунків для магазинів з доставкою, проведено огляд технологій, необхідних для розробки мобільного додатку та розглянуто потреби цільової аудиторії.

Під час проведення дослідження було розглянуто декілька додатків для магазинів з функцією доставки і було виявлено, що вони пропонують багато різних способів замовлення, оплати товару, відслідковування кур’єру та інших елементів функціоналу. Поєднавши всі переваги цих додатків можна створити один застосунок, який задовольнить потреби всіх користувачів.

Не можна проходити повз того факту, що розглянуті додатки мають і свої недоліки, серед яких можна зазначити складну навігацію між різними вкладками, неможливість замовлення доставки за певною адресою, або ж навпаки, неможливість самовивозу.

Також було проведення дослідження наявних технологій для розробки мобільного додатку. Було виявлено що найбільш популярними та простими мовами для програмування на ОС Android є Kotlin та Java. Визначившись з мовою програмування актуальним питання стало обрання найпродуктивнішого та найсучаснішого IDE, яким стала Android Studio, яка включає в себе всі необхідні для розробки інструменти та тестувальний майданчик. До трьох потужних та корисних фреймворків можна віднести Flutter, React Native та Jetpack Compose які забезпечують високу продуктивність роботи застосунку, його швидкодію, цікаві анімації та приємний на вигляд інтерфейс.

Серед розглянутих архітектур найбільш актуальною є MVVM, оскільки вона задовольняє кільком параметрам:

- чітке розділення відповідальностей;
- прив'язка даних (Data Binding);
- модульність і масштабованість;
- підтримка асинхронних операцій;
- тестованість.

Найбільш актуальною базою даних, яку варто використати при розробці додатку на Android це Firebase яка підтримується Google і дає доступ до кількох сервісів, які здатні облегшити авторизацію, заповнення бази даних, створити запити і т.д.

Під час дослідження було виявлено що основні вимоги користувачів до додатку для магазину з доставкою полягають у зрозумілості та простій навігації, широкому асортименту з можливістю зручної фільтрації, швидкій та своєчасній доставці, контролі над процесом доставки та безпеці особистих даних.

Було проведено повне проєктування додатку для магазину з доставкою, моделювання інтерфейсу для представників кожної ролі з урахуванням вимог до зручності та мінімалізму, через що додаток забезпечує зручність у навігації та керуванні, інтуїтивно-зрозумілий та простий дизайн та функціонал.

Надалі проводилася реалізація функціоналу та взаємодії між різними елементами екрану в додатку, де основна ціль полягала у:

- реалізації можливості зручної купівлі продукції для користувачів;
- можливості перегляду всіх звернень з боку представника служби підтримки;
- можливості розрахунку оптимального маршруту для кур’єра для найшвидшої доставки замовлення споживачам;
- можливості реалізації всіх функцій адміністрування.

Також проведений розрахунок найбільш оптимального розташування складів магазину, з яких буде проводитися найшвидша доставка у бідь-яку точку міста, при цьому в залежності від відстані обирається транспорт для доставки.

Для оптимізації швидкості доставки до клієнтів було проведення дослідження двох алгоритмів розподілу замовлень між кур’єрами – угорського алгоритму та жадібного алгоритму. В результаті дослідження було виявлено що угорський алгоритм працює набагато краще, адже він враховує декілька факторів таких як завантаженість кур’єра, оптимальний шлях, найбільш вигідний розподіл замовлень для найшвидшого результату, що робить його використання обов’язковою умовою для покращення вражень клієнта.

Для подальшого розвитку та вдосконалення додатку потрібно:

- покращення дизайну;
- додавання фільтрації за алергенами, наявними в продукції;
- додавання сповіщень про акційні пропозиції, які можуть зацікавити користувача;
- реалізація більшого функціоналу для адміністратору;
- портування додатку на ОС iOS.

Загалом додаток здатен забезпечити користувачів всім необхідним функціоналом та дати можливість кожній ролі виконувати свої функції. Єдине, що лишилось – інтеграція методів оплати для повноцінного замовлення продукції. Додаток є унікальним, адже пропонує користувачам зручний інтерфейс, дозволяє відстежувати замовлення в реальному часі, при цьому маючи можливість спілкування з кур’єром, дає змогу замовити доставку гарячих страв, або ж додавання в один клік всіх необхідних інгредієнтів для людей, які мають бажання самостійно приготувати ту чи іншу страву. Також він реалізує весь необхідний функціонал для персоналу в смартфоні, що робить його більш привабливим для користувачів з огляду на економічну доцільність.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1) Android studio: app development on android 6. Createspace Independent Publishing Platform, 2016. 116 p.

2) Android programming: the big nerd ranch guide / C. Stewart et al. Pearson Education, Limited, 2021.

3) Android studio: app development on android 6. Createspace Independent Publishing Platform, 2016. 116 p.

4) Griffiths D., David G. Head first android development: a brain-friendly guide. O'Reilly Media, 2017. 928 p.

5) John H. Android programming for beginners: learn all the java and android skills you need to start making powerful mobile applications. Packt Publishing, Limited, 2015.

6) Smyth N. Jetpack compose essentials: developing android apps with jetpack compose, android studio, and kotlin. Payload Media, 2022. 502 p.

7) Yahiaoui H. Firebase Cookbook: Over 70 recipes to help you create real-time web and mobile applications with Firebase. Packt Publishing, 2017. 288 p.

8) Nichols C., Klabnik S. Rust programming language, 2nd edition. No Starch Press, Incorporated, 2023.

9) Bruce J. Visual studio code: end-to-end editing and debugging tools for web developers. Wiley & Sons, Incorporated, John, 2019.

10) Advanced MVVM (hard Copy). Lulu Press, Inc., 2010.

11) Bhargava A. Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People. Manning Publications Co. LLC, 2016.

12) Atallah M. Algorithms and Theory of Computation Handbook. CRC Press, 1998.

13) Sadalage P., Fowler M. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Pearson Education, Limited.

14) Clifton I. G. Android User Interface Design: Implementing Material Design for Developers. Pearson Education, Limited.

15) Darwin I. F. Java Cookbook: Problems and Solutions for Java Developers. O'Reilly Media, Incorporated, 2020. 638 p.

16) Соколов М.О., Хохлов М.М. Android-застосунок магазину з доставкою MonoSell. Технологія-2024 : XXVII міжнар. науково-техн. конф., м. Київ, 24 трав. 2024 р., 2024. – С. 181-182.

ДОДАТОК А

Лістинг А.1 - XML для завантаження

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:background="@drawable/background_loading_image"
android:gravity="center">
<ImageView
android:id="@+id/logoLoading"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:src="@drawable/ logoLoading "
android:layout_marginTop="40dp"
android:layout_centerHorizontal="true" />
<ProgressBar
android:id="@+id/customProgressBar"
android:layout_width="100dp"
android:layout_height="100dp"
android:indeterminate="false"
android:progressDrawable="@drawable/custom_progress_bar"
android:layout_centerInParent="true" />
</RelativeLayout>
```

Лістинг А.2 – Код для створення анімації переходу між Activity

```
public class CustomLoadingActivity extends AppCompatActivity {
private ProgressBar customProgressBar;
private final String[] colors = {"#FF5722", "#4CAF50", "#3F51B5",
"#FFC107", "#E91E63"};
private final int[] shapes = {R.drawable.custom_progress_bar,
R.drawable.circle_shape, R.drawable.rectangle_shape,
R.drawable.circle_rectangle_shape, R.drawable. square_shape };
…
customProgressBar = findViewById(R.id.customProgressBar);
Random random = new Random();
int colorIndex = random.nextInt(colors.length);
int shapeIndex = random.nextInt(shapes.length);
customProgressBar.setProgressDrawable(getDrawable(shapes[shapeInde
x]));
animateProgressBar();
String targetActivity =
getIntent().getStringExtra("targetActivity");
new Handler().postDelayed(() -> {
try {Class<?> targetClass = Class.forName(targetActivity);
Intent intent = new Intent(CustomLoadingActivity.this,
targetClass);
startActivity(intent);
finish();} catch (ClassNotFoundException e) {
```

```
e.printStackTrace();}}, 3000);}
private void animateProgressBar() {
new Thread(() -> {
for (int progress = 0; progress <= 100; progress += 5) {
int finalProgress = progress;
runOnUiThread(() -> customProgressBar.setProgress(finalProgress));
try {Thread.sleep(150); } catch (InterruptedException e) {
e.printStackTrace();}}}).start();}}
```

Лістинг А.3– Бокове меню та взаємодія з ним

```
drawerLayout = findViewById(R.id.drawer_layout);
menuButton = findViewById(R.id.menu_button);
NavigationView navigationView =
findViewById(R.id.navigation_view);
menuButton.setOnClickListener(v -> {
drawerLayout.openDrawer(navigationView);
menuButton.setVisibility(View.GONE);});
drawerLayout.addDrawerListener(new
DrawerLayout.SimpleDrawerListener() {
@Override
public void onDrawerClosed(View drawerView) {
super.onDrawerClosed(drawerView);
menuButton.setVisibility(View.VISIBLE); }});
navigationView.setNavigationItemSelectedListener(item -> {
switch (item.getItemId()) {
case R.id.nav_promotions:
startActivity(new Intent(MainShopActivity.this,
PromotionsActivity.class));
break;
case R.id.nav_map:
startActivity(new Intent(MainShopActivity.this,
MapActivity.class));
break;
…
}
drawerLayout.closeDrawer(navigationView);
return true;});
drawerLayout.setOnTouchListener((v, event) -> {
if (event.getAction() == MotionEvent.ACTION_MOVE &&
drawerLayout.isDrawerOpen(navigationView)) {
drawerLayout.closeDrawer(navigationView);
return true;}
return false;});}
```

Лістинг А.4 – Мапа користувача

```
public class RealTimeTrackingActivity extends AppCompatActivity
implements OnMapReadyCallback {
private GoogleMap mMap;
private Marker userMarker;
private Marker courierMarker;
```

```
private Polyline courierPath;
private DatabaseReference databaseReference;
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_maps);
databaseReference = FirebaseDatabase.getInstance().getReference();
SupportMapFragment     mapFragment     =     (SupportMapFragment)
getSupportFragmentManager().findFragmentById(R.id.map);
assert mapFragment != null;
mapFragment.getMapAsync(this);}
@Override
public void onMapReady(@NonNull GoogleMap googleMap) {
mMap = googleMap;
startLocationUpdates();
trackUserLocation();
trackCourierLocation();}
private void startLocationUpdates() {
FusedLocationProviderClient      fusedLocationProviderClient      =
LocationServices.getFusedLocationProviderClient(this);
LocationRequest locationRequest = LocationRequest.create();
locationRequest.setInterval(5000);
locationRequest.setFastestInterval(2000);
locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY
);
fusedLocationProviderClient.requestLocationUpdates(locationRequest
, new LocationCallback() {
@Override
public void onLocationResult(LocationResult locationResult) {
if (locationResult == null) return;
for (Location location : locationResult.getLocations()) {
double latitude = location.getLatitude();
double longitude = location.getLongitude();
updateLocationToDatabase(latitude,                  longitude);}}},
Looper.getMainLooper());}
private void trackUserLocation() {
databaseReference.child("usersLocation").addValueEventListener(new
ValueEventListener() {
@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
if (snapshot.exists()) {
double lat = snapshot.child("latitude").getValue(Double.class);
double lng = snapshot.child("longitude").getValue(Double.class);
LatLng userLatLng = new LatLng(lat, lng);
if (userMarker == null) {
userMarker                  =                  mMap.addMarker(new
MarkerOptions().position(userLatLng).title("Your Location")
.icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactor
y.HUE_BLUE)));} else {userMarker.setPosition(userLatLng);}
mMap.animateCamera(CameraUpdateFactory.newLatLngZoom(userLatLng,
15));}}
@Override
public void onCancelled(@NonNull DatabaseError error) {}});}
```

```
private void trackCourierLocation() {
databaseReference.child("couriersLocation").addValueEventListener(
new ValueEventListener() {
private final List<LatLng> courierPathPoints = new ArrayList<>();
@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
if (snapshot.exists()) {
double lat = snapshot.child("latitude").getValue(Double.class);
double lng = snapshot.child("longitude").getValue(Double.class);
String                        transportMode                        =
snapshot.child("transportMode").getValue(String.class);
LatLng courierLatLng = new LatLng(lat, lng);
if (courierMarker == null) {
courierMarker = mMap.addMarker(new MarkerOptions()
.position(courierLatLng).title("Courier Location")
.icon(getTransportIcon(transportMode)));
} else {courierMarker.setPosition(courierLatLng);
courierMarker.setIcon(getTransportIcon(transportMode));}
courierPathPoints.add(courierLatLng);
if (courierPath != null) {
courierPath.remove();}
courierPath = mMap.addPolyline(new PolylineOptions()
.addAll(courierPathPoints).width(5).color(0xFF00FF00));}}
@Override
public void onCancelled(@NonNull DatabaseError error) {}});}
private       BitmapDescriptorFactory       getTransportIcon(String
transportMode) {
if          (transportMode          ==          null)          return
BitmapDescriptorFactory.defaultMarker();
switch (transportMode) {
case "walking":
return
BitmapDescriptorFactory.fromResource(R.drawable.icon_walking);
case "motorcycle":
return
BitmapDescriptorFactory.fromResource(R.drawable.icon_motorcycle);
case "car":
return BitmapDescriptorFactory.fromResource(R.drawable.icon_car);
default:
return BitmapDescriptorFactory.defaultMarker();}}}
```

Лістинг А.5 – Збереження замовлення у базі даних та реалізація перегляду історії замовлень

```
private void saveOrderToHistory(String userId) {
DatabaseReference userRef =
FirebaseDatabase.getInstance().getReference("users").child(userId)
;
userRef.child("currentOrder").addListenerForSingleValueEvent(new
ValueEventListener() {
@Override
```

```
public void onDataChange(@NonNull DataSnapshot snapshot) {
if (snapshot.exists()) {
Map<String, Object> currentOrder = (Map<String, Object>)
snapshot.getValue();
String orderId = userRef.child("orderHistory").push().getKey();
if (orderId != null) {
userRef.child("orderHistory").child(orderId).setValue(currentOrder
).addOnSuccessListener(aVoid -> {
userRef.child("currentOrder").removeValue();})
.addOnFailureListener(e -> {
Toast.makeText(getApplicationContext(), "Помилка збереження
замовлення", Toast.LENGTH_SHORT).show();
});}} else {Toast.makeText(getApplicationContext(), "Немає
активного замовлення", Toast.LENGTH_SHORT).show();}}
@Override
public void onCancelled(@NonNull DatabaseError error) {
Toast.makeText(getApplicationContext(), "Помилка доступу до бази
даних", Toast.LENGTH_SHORT).show();
}});}
private void loadOrderHistory(String userId) {
DatabaseReference                    userRef                    =
FirebaseDatabase.getInstance().getReference("users").child(userId)
.child("orderHistory");
userRef.addValueEventListener(new ValueEventListener() {
@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
List<Order> orderList = new ArrayList<>();
for (DataSnapshot orderSnapshot : snapshot.getChildren()) {
Order order = orderSnapshot.getValue(Order.class);
orderList.add(order);}
orderHistoryAdapter.setOrders(orderList);
}
@Override
public void onCancelled(@NonNull DatabaseError error) {
Toast.makeText(getApplicationContext(),    "Помилка    завантаження
історії", Toast.LENGTH_SHORT).show();
}});}
```

Лістинг А.6 – Основні методи роботи з чатом

```
private void loadMessages() {
chatRef.addValueEventListener(new ValueEventListener() {
@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
messages.clear();
for (DataSnapshot messageSnapshot : snapshot.getChildren()) {
Message message = messageSnapshot.getValue(Message.class);
messages.add(message);}
chatAdapter.notifyDataSetChanged();
messagesRecyclerView.scrollToPosition(messages.size() - 1);}
@Override
public void onCancelled(@NonNull DatabaseError error) {
```

```
Toast.makeText(ChatActivity.this, "Помилка завантаження
повідомлень", Toast.LENGTH_SHORT).show();}});}
private void monitorSupportStatus() {
supportStatusRef.child("status").addValueEventListener(new
ValueEventListener() {
@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
String status = snapshot.getValue(String.class);
supportStatus.setText(status != null ? status : "Офлайн");}
@Override
public void onCancelled(@NonNull DatabaseError error) {
supportStatus.setText("Офлайн");}});}
private void sendMessage() {
String text = messageInput.getText().toString().trim();
if (!text.isEmpty()) {
String messageId = chatRef.push().getKey();
Message message = new Message(text, "user", "sent",
System.currentTimeMillis());
if (messageId != null) {
chatRef.child(messageId).setValue(message).addOnSuccessListener(aV
oid -> {
messageInput.setText("");
}).addOnFailureListener(e -> {
Toast.makeText(ChatActivity.this, "Помилка надсилання
повідомлення", Toast.LENGTH_SHORT).show();});}}}
```

Лістинг А.7 – Метод завантаження замовлень із урахуванням радіуса

```
private void loadOrdersWithinRadius(String courierId) {
DatabaseReference courierRef =
FirebaseDatabase.getInstance().getReference("couriers").child(cour
ierId);
DatabaseReference ordersRef =
FirebaseDatabase.getInstance().getReference("orders");
courierRef.addListenerForSingleValueEvent(new ValueEventListener()
{@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
if (!snapshot.exists()) return;
String transportType =
snapshot.child("transportType").getValue(String.class);
double courierLat =
snapshot.child("latitude").getValue(Double.class);
double courierLon =
snapshot.child("longitude").getValue(Double.class);
double radius = getRadius(transportType);
ordersRef.addValueEventListener(new ValueEventListener() {
@Override
public void onDataChange(@NonNull DataSnapshot ordersSnapshot) {
googleMap.clear();
for (DataSnapshot orderSnapshot : ordersSnapshot.getChildren()) {
Order order = orderSnapshot.getValue(Order.class);
if (order != null && "available".equals(order.getStatus())) {
```

```
double orderLat = order.getLatitude();
double orderLon = order.getLongitude();
double distance = calculateDistance(courierLat, courierLon,
orderLat, orderLon);
if (distance <= radius) {
LatLng location = new LatLng(orderLat, orderLon);
googleMap.addMarker(new
MarkerOptions().position(location).title(order.getAddress()).snipp
et("Вага: " + order.getWeight() + " кг\nВідстань: " + (distance /
1000) + " км"))
.setTag(order.getOrderId());}}}}
@Override
public void onCancelled(@NonNull DatabaseError error) {
Toast.makeText(getContext(), "Помилка завантаження замовлень",
Toast.LENGTH_SHORT).show();}});}
@Override
public void onCancelled(@NonNull DatabaseError error) {
Toast.makeText(getContext(), "Помилка доступу до даних кур’єра",
Toast.LENGTH_SHORT).show();}});}
```

Лістинг А.8 – Фільтрація замовлень для кожного кур’єра

```
private void loadOrdersOnMap(String courierId) {
ordersRef.addValueEventListener(new ValueEventListener() {
@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
googleMap.clear();
for (DataSnapshot orderSnapshot : snapshot.getChildren()) {
Order order = orderSnapshot.getValue(Order.class);
if (order != null) {
String status = order.getStatus();
String assignedCourier = order.getCourierId();
if ("available".equals(status)) {
LatLng location = new LatLng(order.getLatitude(),
order.getLongitude());
googleMap.addMarker(new
MarkerOptions().position(location).title(order.getAddress()).snipp
et("Вага: " + order.getWeight() + " кг"))
.setTag(order.getOrderId());} else if ("accepted".equals(status)
&& courierId.equals(assignedCourier)) {
LatLng location = new LatLng(order.getLatitude(),
order.getLongitude());
googleMap.addMarker(new
MarkerOptions().position(location).title(order.getAddress()).snipp
et("Ваше замовлення"));}}}}
@Override
public void onCancelled(@NonNull DatabaseError error) {
Toast.makeText(getContext(), "Помилка завантаження замовлень",
Toast.LENGTH_SHORT).show();}});}
```

Лістинг А.9 – Activity для списку чатів з методом loadChats()

```
…
chatList = new ArrayList<>();
chatAdapter = new ChatAdapter(chatList, chat -> {
Intent intent = new Intent(ChatListActivity.this,
ChatDetailActivity.class);
intent.putExtra("chatId", chat.getChatId());
intent.putExtra("clientId", chat.getClientId());
startActivity(intent);});
recyclerViewChats.setAdapter(chatAdapter);
db = FirebaseFirestore.getInstance();
loadChats();}
private void loadChats() {
db.collection("chats").whereEqualTo("supportId",
currentSupportId).addSnapshotListener(new EventListener<>() {
@Override
public void onEvent(@Nullable
com.google.firebase.firestore.QuerySnapshot value,
@Nullable FirebaseFirestoreException error) {
if (error != null) {Log.w("ChatListActivity", "Listen failed.",
error);
return; }
List<Chat> chats = new ArrayList<>();
for (DocumentSnapshot doc : value) {
Chat chat = doc.toObject(Chat.class);
if (chat != null) {chat.setChatId(doc.getId());
chats.add(chat); }}
chatAdapter.setChatList(chats);}});}}
```

Лістинг А.10 – Методи loadMessages() та sendMessage() для завантаження та відправки повідомлення в чаті з представником підтримки

```
private void loadMessages() {
CollectionReference messagesRef =
db.collection("chats").document(chatId).collection("messages");
messagesRef.orderBy("timestamp").addSnapshotListener(new
EventListener<>() {
@Override
public void onEvent(@Nullable QuerySnapshot value,
@Nullable FirebaseFirestoreException error) {
if (error != null) {
Log.w("ChatDetailActivity", "Listen failed.", error);
return;}
List<Message> messages = new ArrayList<>();
for (DocumentSnapshot doc : value) {
Message message = doc.toObject(Message.class);
if (message != null) {messages.add(message);}}
messageAdapter.setMessageList(messages);
recyclerViewMessages.scrollToPosition(messages.size() - 1);}});}
private void sendMessage() {
String messageText = editTextMessage.getText().toString().trim();
```

```
if (messageText.isEmpty()) {
return;}
Message message = new Message();
message.setSenderId(currentSupportId);
message.setMessage(messageText);
message.setTimestamp(System.currentTimeMillis());
CollectionReference messagesRef =
db.collection("chats").document(chatId).collection("messages");
messagesRef.add(message).addOnSuccessListener(documentReference ->
{DocumentReference chatRef =
db.collection("chats").document(chatId);
chatRef.update("lastMessage", messageText, "timestamp",
message.getTimestamp())
.addOnSuccessListener(aVoid -> {
editTextMessage.setText("");})
.addOnFailureListener(e -> {
Log.w("ChatDetailActivity", "Error updating chat.", e);});});})
.addOnFailureListener(e -> {Log.w("ChatDetailActivity", "Error
sending message.", e);});}}
```

Лістинг А.11 – Перевизначення методу onStart()

```
@Override
protected void onStart() {
super.onStart();
if (mAuth.getCurrentUser() == null) {
Intent intent = new Intent(ChatListActivity.this,
LoginActivity.class);
startActivity(intent);
finish();}}
```

Лістинг А.12 – Можливість отримання відгуку про чат між користувачем та представником служби підтримки

```
private void showFeedbackDialog(String chatId, String userId) {
Dialog feedbackDialog = new Dialog(this);
feedbackDialog.setContentView(R.layout.feedback_dialog);
RatingBar ratingBar = feedbackDialog.findViewById(R.id.ratingBar);
EditText reviewInput =
feedbackDialog.findViewById(R.id.reviewInput);
Button submitButton =
feedbackDialog.findViewById(R.id.submitFeedbackButton);
submitButton.setOnClickListener(v -> {
float rating = ratingBar.getRating();
String review = reviewInput.getText().toString();
if (rating == 0) {
Toast.makeText(this, "Будь ласка, поставте оцінку",
Toast.LENGTH_SHORT).show();
return;}
Map<String, Object> feedback = new HashMap<>();
feedback.put("chatId", chatId);
```

```
feedback.put("userId", userId);
feedback.put("rating", rating);
feedback.put("review", review);
feedback.put("timestamp", System.currentTimeMillis());
FirebaseFirestore db = FirebaseFirestore.getInstance();
db.collection("feedback").add(feedback).addOnSuccessListener(docum
entReference -> {
Toast.makeText(this, "Дякуємо за ваш відгук!",
Toast.LENGTH_SHORT).show();
feedbackDialog.dismiss();}).addOnFailureListener(e -> {
Toast.makeText(this, "Помилка при відправці відгуку",
Toast.LENGTH_SHORT).show();});});
feedbackDialog.show();}
```

Лістинг А.13 – Відкриття Dialog для зміни характеристик товару

```
private void openEditProductDialog(Product product) {
EditProductDialog dialog = new EditProductDialog(product);
dialog.show(getSupportFragmentManager(), "EditProductDialog");}
…
EditText nameInput = view.findViewById(R.id.nameInput);
EditText priceInput = view.findViewById(R.id.priceInput);
EditText stockInput = view.findViewById(R.id.stockInput);
…
Button saveButton = view.findViewById(R.id.saveButton);
nameInput.setText(product.getName());
priceInput.setText(String.valueOf(product.getPrice()));
stockInput.setText(String.valueOf(product.getStock()));
…
db = FirebaseFirestore.getInstance();
saveButton.setOnClickListener(v -> {
String newName = nameInput.getText().toString();
double newPrice =
Double.parseDouble(priceInput.getText().toString());
int newStock = Integer.parseInt(stockInput.getText().toString());
…
db.collection("products").document(product.getId())
.update("name", newName, "price", newPrice, "stock", newStock…)
.addOnSuccessListener(aVoid -> {
Toast.makeText(getContext(), "Товар оновлено!",
Toast.LENGTH_SHORT).show();
dismiss();})
.addOnFailureListener(e -> Toast.makeText(getContext(),
"Помилка!", Toast.LENGTH_SHORT).show());});
return view;}
```

Лістинг А.14 – Метод openAddProductDialog() для додавання продукції

```
private void openAddProductDialog(Category category) {
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Додати товар до категорії: " +
category.getName());
```

```
View dialogView =
LayoutInflater.from(this).inflate(R.layout.dialog_add_product,
null);
builder.setView(dialogView);
EditText productNameInput =
dialogView.findViewById(R.id.productNameInput);
EditText productPriceInput =
dialogView.findViewById(R.id.productPriceInput);
EditText productStockInput =
dialogView.findViewById(R.id.productStockInput);
…
builder.setPositiveButton("Додати", (dialog, which) -> {
String productName = productNameInput.getText().toString().trim();
String productPriceStr =
productPriceInput.getText().toString().trim();
String productStockStr =
productStockInput.getText().toString().trim();
if (!productName.isEmpty() && !productPriceStr.isEmpty() &&
!productStockStr.isEmpty()) {
try {double productPrice = Double.parseDouble(productPriceStr);
int productStock = Integer.parseInt(productStockStr);
Product newProduct = new Product();
newProduct.setName(productName);
newProduct.setPrice(productPrice);
newProduct.setStock(productStock);
…
category.getProducts().add(newProduct);
db.collection("categories").document(category.getId())
.update("products", category.getProducts())
.addOnSuccessListener(aVoid -> {
Toast.makeText(this, "Товар додано успішно",
Toast.LENGTH_SHORT).show();
loadCategories();})
.addOnFailureListener(e ->
Toast.makeText(this, "Помилка при додаванні товару",
Toast.LENGTH_SHORT).show());} catch (NumberFormatException e) {
Toast.makeText(this, "Ціна та кількість повинні бути числовими
значеннями", Toast.LENGTH_SHORT).show();}} else {
Toast.makeText(this, "Всі поля мають бути заповнені",
Toast.LENGTH_SHORT).show();}});
builder.setNegativeButton("Скасувати", (dialog, which) ->
dialog.dismiss());
builder.show();}
```

Лістинг А.15 – Метод deleteCategory () для видалення категорії

```
private void deleteCategory(Category category) {
db.collection("categories").document(category.getId())
.delete().addOnSuccessListener(aVoid -> {
Toast.makeText(this, "Категорію видалено",
Toast.LENGTH_SHORT).show();
loadCategories();
```

```
}).addOnFailureListener(e -> Toast.makeText(this, "Помилка
видалення", Toast.LENGTH_SHORT).show());}
```

Лістинг А.16 – Метод openAddCategoryDialog() для додавання категорії

```
private void openAddCategoryDialog() {
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Додати нову категорію");
final EditText input = new EditText(this);
input.setHint("Введіть назву категорії");
builder.setView(input);
builder.setPositiveButton("Додати", (dialog, which) -> {
String categoryName = input.getText().toString().trim();
if (!categoryName.isEmpty()) {
Category newCategory = new Category();
newCategory.setName(categoryName);
newCategory.setProducts(new ArrayList<>());
db.collection("categories").add(newCategory)
.addOnSuccessListener(documentReference -> {
Toast.makeText(this, "Категорія додана успішно",
Toast.LENGTH_SHORT).show();
loadCategories(); }).addOnFailureListener(e ->
Toast.makeText(this, "Помилка при додаванні категорії",
Toast.LENGTH_SHORT).show());} else {
Toast.makeText(this, "Назва категорії не може бути пустою",
Toast.LENGTH_SHORT).show();}});
builder.setNegativeButton("Скасувати", (dialog, which) ->
dialog.dismiss());
builder.show();}
```

Лістинг А.17 – Метод loadCategories() для оновлення бази даних

```
private void loadCategories() {
db.collection("categories").get().addOnCompleteListener(task -> {
if (task.isSuccessful() && task.getResult() != null) {
categoryList.clear();
for (DocumentSnapshot document : task.getResult()) {
Category category = document.toObject(Category.class);
category.setId(document.getId());
categoryList.add(category);}
categoryAdapter.notifyDataSetChanged();}});}
```

Лістинг А.18 – Реалізація роботи з персоналом

```
private void fetchUsers() {
dbRef.orderByChild("role").addListenerForSingleValueEvent(new
ValueEventListener() {
@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
userList.clear();
for (DataSnapshot userSnapshot : snapshot.getChildren()) {
```

```
String role = userSnapshot.child("role").getValue(String.class);
String userData = "ID: " + userSnapshot.getKey() + ", Role: " + role;
userList.add(userData);
}
adapter.notifyDataSetChanged();
}
@Override
public void onCancelled(@NonNull DatabaseError error) {
Toast.makeText(AdminActivity.this, "Помилка завантаження даних",
Toast.LENGTH_SHORT).show();
}});}
private void showUserOptions(String userData) {
String userId = userData.split(",")[0].replace("ID: ", "").trim();
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Оберіть дію");
String[] options = {"Змінити роль", "Видалити"};
builder.setItems(options, (dialog, which) -> {
if (which == 0) {
showRoleChangeDialog(userId);
} else if (which == 1) {
deleteUser(userId);
}});
builder.show();}
private void showRoleChangeDialog(String userId) {
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Оберіть нову роль");
String[] roles = {"адмін", "кур'єр", "сапорт", "покупець"};
builder.setItems(roles, (dialog, which) -> {
String newRole = roles[which];
updateUserRole(userId, newRole);});
builder.show();}
private void updateUserRole(String userId, String newRole) {
Map<String, Object> updates = new HashMap<>();
updates.put("role", newRole);
dbRef.child(userId).updateChildren(updates)
.addOnSuccessListener(aVoid -> Toast.makeText(AdminActivity.this,
"Роль оновлена на " + newRole, Toast.LENGTH_SHORT).show())
.addOnFailureListener(e -> Toast.makeText(AdminActivity.this,
"Плмилка оновлення", Toast.LENGTH_SHORT).show());
}
private void deleteUser(String userId) {
dbRef.child(userId).removeValue()
.addOnSuccessListener(aVoid -> {
Toast.makeText(AdminActivity.this, "Користувача видалено",
Toast.LENGTH_SHORT).show();
fetchUsers();}).addOnFailureListener(e ->
Toast.makeText(AdminActivity.this, "Помилка видалення",
Toast.LENGTH_SHORT).show());}
```