Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Мобільний застосунок для інтернет-магазину кави та чаю

(тема)

Виконав:
здобувач 4 року навчання,
групи КІУКІ-21-5
Артем НАУМОВ
(власне ім’я, прізвище)

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

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

Освітня програма
Комп’ютерна інженерія
(повна назва освітньої програми)

Керівник: ст. викл. Олена СЕВОСТЬЯНОВА
(посада, власне ім’я, прізвище)

Допускається до захисту

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

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

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

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

Освітня програма Комп’ютерна інженерія
(повна назва)

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

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Наумову Артему Валерійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Мобільний застосунок для інтернет-магазину кави та чаю

затверджена наказом по університету від “ 26 ” травня 2025 р. № 424 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 17 червня 2025 р.

3. Вхідні дані до роботи

1) поняття електронної комерції та її основні види
2) мова програмування Kotlin
3) використання сучасних архітектурних підходів
4) використання фреймворку Jetpack Compose для створення сучасного інтерфейсу
5) логіка взаємодії з сервісом Supabase

4. Перелік питань, що потрібно опрацювати у роботі

1) аналіз предметної області
2) аналіз технологій та підходів, які було використано у розробці застосунку
3) опис програмної реалізації застосунку
4) інструкція користувача
5) висновки

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

Слайд-презентація – 10 слайдів

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

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

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	27.05.25-30.05.25	
2	Вибір технологій та інструментів розробки	31.05.25-01.06.25	
3	Розробка основних функцій застосунку	02.06.25-06.06.25	
4	Тестування та відлагодження застосунку	07.06.25-09.06.25	
5	Оформлення матеріалів кваліфікаційної роботи	10.06.25-11.06.25	
6	Подання кваліфікаційної роботи керівникові та	12.06.25-13.06.25	
	її попередній захист		
7	Подання кваліфікаційної роботи на рецензу-	14.06.25-16.06.25	
	вання		

Дата видачі завдання " 26 " травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ст. викл. Олена СЕВОСТЬЯНОВА
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 74 с., 23 рис., 1 дод., 15 джерел.

KOTLIN, JETPACK COMPOSE, E-COMMERCE, ІНТЕРФЕЙС КОРИСТУВАЧА, НАВІГАЦІЯ, CLEAN ARCHITECTURE, DEPENDENCY INJECTION, SOLID, SUPABASE, CMS, БАЗА ДАНИХ.

Метою кваліфікаційної роботи є розробка мобільного застосунку для електронної комерції – інтернет-магазину, який спеціалізується на продажу кави та чаю.

У ході виконання кваліфікаційної роботи було створено працездатний нативний мобільний застосунок під операційну систему Android, який надає користувачам послуги електронної комерції, а саме продаж кави та чаю, а також пов’язаних з цією сферою товарів.

Проєкт створено на основі сучасних підходів до побудови архітектури програмного забезпечення, це надає можливості до легкої підтримки та розширення проєкту в майбутньому, а також забезпечує можливість легкої інтеграції нових людей до підтримки проєкту.

Застосунок створено у середовищі розробки Android Studio з використанням мови програмування Kotlin, яка є основною мовою розробки Android застосунків. Для забезпечення декларативного підходу в створенні інтерфейсу користувача було використано фреймворк Jetpack Compose.

ABSTRACT

Bachelor’s thesis: 74 pages, 23 figures, 1 appendice, 15 sources.

KOTLIN, JETPACK COMPOSE, E-COMMERCE, USER INTERFACE, NAVIGATION, CLEAN ARCHITECTURE, DEPENDENCY INJECTION, SOLID, SUPABASE, CMS, DATABASE.

The major goal of this thesis is to develop a mobile application for e-commerce – online store that specializes in selling coffee and tea.

In order to complete the qualification work, a functional native mobile application for the Android operating system was created, which provides users with e-commerce services, namely the sale of coffee and tea, as well as related products.

The project is based on modern approaches to building software architecture, which provides opportunities for easy support and expansion of the project in the future, as well as the ability to easily integrate new people into project support.

The application was created in the Android Studio development environment using the Kotlin programming language, which is the main language for developing Android applications. The Jetpack Compose framework was used to provide a declarative approach to creating the user interface.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 8
ВСТУП 9
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 10
1.1 Використання цифрових технологій для ведення бізнесу 10
1.2 Аналіз існуючих рішень 11
1.2.1 Інтернет-магазин «Kulturrra» 11
1.2.2 Інтернет-магазин «Coffeeok» 13
1.2.3 Мобільний застосунок «Мій чай» 14
1.3 Постановка задачі 15
2 АНАЛІЗ ТЕХНОЛОГІЙ ТА ПІДХОДІВ, ЯКІ БУЛО ВИКОРИСТАНО У РОЗРОБЦІ ЗАСТОСУНКУ 17
2.1 Мова програмування Kotlin 17
2.2 Операційна система Android 18
2.3 Фреймворк Jetpack Compose 20
2.4 Фреймворк Hilt 21
2.5 Система керування базами даних PostgreSQL 22
2.6 Інструмент для автоматизації збірки Gradle 23
2.7 Середовище розробки Android Studio 24
2.8 Інструмент для тестування мобільних застосунків Android Emulator 25
2.9 Система контролю версій Git 26
2.10 Концепція Clean Architecture 27
2.11 Архітектурний патерн MVVM 30
3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ 32
3.1 Опис структури застосунку 32
3.2 Каталог domain 33
3.3 Каталог di 35
3.4 Реалізація збереження та доступу до даних 36

3.5 Каталог data..... 39
3.6 Каталог presentation..... 43
4 ІНСТРУКЦІЯ КОРИСТУВАЧА 49
ВИСНОВКИ..... 65
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 67
ДОДАТОК А Графічний матеріал кваліфікаційної роботи..... 69

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

AAB – формат пакету Android для розповсюдження застосунків (англ., Android App Bundle).

ACID – набір властивостей, що гарантують надійну роботу транзакцій бази даних: атомарність, узгодженість, ізольованість, довговічність (англ., Atomicity, Consistency, Isolation, Durability).

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

APK – пакет Android (англ., Android Package).

AVD Manager – менеджер віртуальних пристроїв Android (англ., Android Virtual Device Manager).

CMS – система керуванням вмісту (англ., Content Management System).

CRUD – операції створення, читання, оновлення та видалення данних в базах даних (англ., Create Read Update Delete).

DRY – принцип програмування, уникай повторень (англ., Don't Repeat Yourself).

DTO – об’єкт передачі даних (англ., Data Transfer Object).

JVM – Java віртуальна машина (англ., Java Virtual Machine).

KISS – принцип програмування, роби коротше і простіше (англ., Keep It Short And Simple).

MVVM – архітектурний підхід (англ., Model-View-ViewModel).

SOLID – принципи об’єктно-орієнтованого програмування (англ., Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion).

SQL – мова структурованих запитів (англ., Structured Query Language).

XML – розширювана мова розмітки (англ., Extensible Markup Language).

ВСТУП

E-commerce – це одна з моделей бізнесу, основною відмінністю якої є те що продажі товарів та послуг відбуваються через інтернет або інші електронні мережі. Цей процес охоплює всі етапи онлайн-торгівлі: від реклами та просування товарів до оформлення замовлення, оплати та доставки. Усі послуги надаються через інтелектуальні пристрої, такі як комп’ютери, планшети, смартфони, для яких розробляються спеціальні програми або вебсайти [1].

Прикладом реалізації моделі e-commerce є мобільні застосунки, які дають змогу замовляти товари або послуги прямо зі смартфону. Такі програми можуть охоплювати як широкий спектр послуг, так і більш вузький, як-от застосунки для замовлення продуктів харчування, одягу, побутової техніки. Такі застосунки з кожним роком набирають популярності, так як надають зручний доступ до каталогів товарів, детальному опису продукції, персоналізованих рекомендацій, можливості швидко оформлювати замовлення та здійснювати оплату онлайн. І це все можна зробити в будь-якому місці та в будь-який час, після чого отримати своє замовлення на пошті, або навіть зробити доставку до свого дому.

Зважаючи на все це, мета даної кваліфікаційної роботи – створити нативний мобільний застосунок для ОС Android, який буде орієнтований на тих, хто хоче поринути у світ кави та чаю, надаючи можливість замовити різні сорти цих продуктів та пов’язаних з ним товарів [2].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Використання цифрових технологій для ведення бізнесу

E-commerce або електронна комерція – це процес купівлі-продажу товарів та послуг за допомогою інтернету. Цей термін охоплює всі види онлайн-торгівлі, що здійснюються за допомогою таких пристроїв як комп’ютери, смартфони, планшети. Тому з розвитком можливостей цих пристроїв, стає популярнішою і електронна комерція.

Історія електронної комерції починається ще в 60-х роках, але перша справжня онлайн-транзакція відбулась лише в 1994 році, тоді було продано компакт-диск через сайт NetMarket. На сьогоднішній день електронна комерція значно змінилась та розширилась, її використовують як великі компанії (наприклад, Amazon, яка надає багато послуг через інтернет), так і малі та середні бізнеси.

Електронна комерція змінила правила ринку торгівлі, надаючи навіть малим підприємствам можливість працювати на більш широкий ринок, продаючи свої послуги через інтернет з будь-якої точки світу, без необхідності будувати або орендувати приміщення. Це дало змогу знизити витрати та зосередитись на підвищенні якості послуг. Крім того це дозволило автоматизувати багато процесів, таких як обробка замовлень, взаємодія з постачальниками та підтримка клієнтів.

Так як електронна комерція – це процес, який відбувається в інтернеті, вебсайти та мобільні застосунки відіграють важливу роль у її розвитку. Вони забезпечують платформи для реклами, продажу та взаємодії з клієнтами. Завдяки широкому функціоналу таких платформ, стало легше залучати нових покупців, надаючи персоналізований досвід та зручні способи оплати та доставки.

Окрім основних функцій вебсайти та мобільні застосунки дозволяють проводити аналіз поведінки користувачів, отримувати дані про їх вподобання

та потреби. Це полегшує планування маркетингових стратегій, що в свою чергу допомагає створювати більш цільові пропозиції для користувачів.

Існують і недоліки електронної комерції, наприклад, покупці не можуть безпосередньо побачити та оцінити товар перед покупкою, що може призвести до розчарувань, якщо товар не відповідатиме очікуванням. Також до недоліків можна віднести те, що потрібно чекати на деякі послуги, наприклад, доставка замовлення, що може не підійти споживачу у випадку необхідності отримати товар без очікування його доставки. Ще одним недоліком є велика конкуренція. Електронна комерція спрощує створення бізнесу, що є великою перевагою, але це також означає що й конкуренти так само легко, можуть почати свій бізнес.

Електронна комерція ділиться на багато різних моделей. Найпоширеніша – це бізнес для споживача, коли компанія продає товар або послуги безпосередньо споживачу використовуючи мережу інтернет, наприклад, через онлайн-магазини чи мобільні застосунки. Також є модель, коли компанія продає послуги чи товари іншим компаніям. Окрім цього, існує модель, коли споживач продає товар чи послуги іншому споживачу, прикладом якої є сервіс OLX. Крім того, є модель, коли споживач надає свої послуги компанії [3].

1.2 Аналіз існуючих рішень

1.2.1 Інтернет-магазин «Kulturrra»

Kulturrra – це інтернет-магазин, який сконцентрован саме на продажі кави, аксесуарів та техніки для його приготування [4]. На головній сторінці магазину Kulturrra, можна одразу побачити всі категорії, які може запропонувати магазин (рисунок 1.1). Проаналізувавши його зміст можна зробити висновок, що інтернет-магазин має вузьку сферу продажу товарів, а саме продаж різних видів кави та аксесуарів. Також користувачам

пропонується купити брендові товари, наприклад світшоти, шкарпетки, термогорнятки з їх логотипом. Однак наразі в цій категорії товарів мало асортименту, пов’язаного зі сферою кави. Крім цього, користувачам пропонується купити кавову підписку на три місяці, яка дає змогу обрати сорт кави та отримувати свіжообсмажену каву на початку кожного місяця. Ще в магазині є можливість придбати подарункові сертифікати, які можна використати для повної або часткової оплати замовлених товарів.

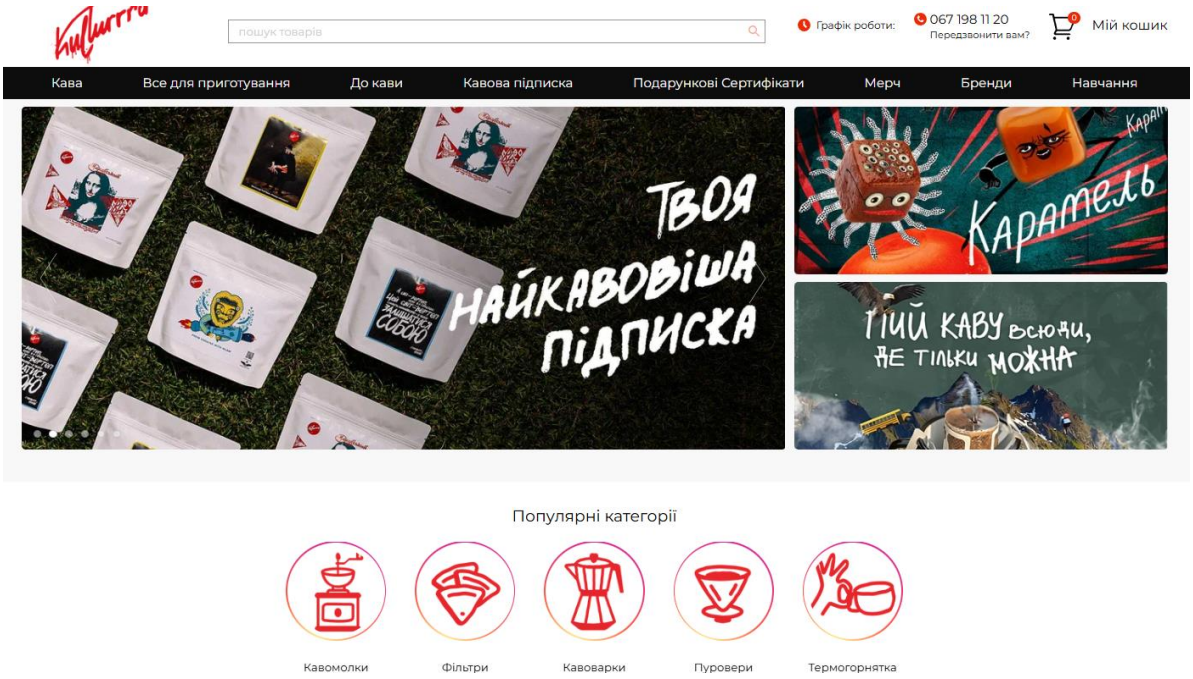


Рисунок 1.1 – Головна сторінка інтернет-магазину Kulturrra

Застосунок магазину Kulturrra пропонує користувачам простий, інтуїтивно зрозумілий інтерфейс, що дає змогу легко шукати необхідний товар, використовуючи категорії та пошук з необхідними фільтрами. Кожний товар має повний опис, який містить все необхідне для того щоб користувач міг сформувати правильне уявлення про товар. Якщо користувач хоче мати змогу зберігати свою адресу (для швидкого замовлення в майбутньому), улюблені товари та історію замовлень, йому пропонується створити аккаунт. В ньому будуть зберігатись усі ці данні, до яких можна буде отримати доступ з будь-якого пристрою, після авторизації.

Але є й не дуже вдалі рішення в дизайні вебсайту. Так, наприклад, кнопку видалення товару з корзини не так легко знайти, через те, що вона занадто мала та має сірий колір, що робить її непомітною. Ще одним недоліком є те що Kulturrra не має мобільного застосунку, оскільки користувачі все частіше віддають перевагу саме їм. Використовуючи мобільний застосунок, людина не витрачає свій час на пошук інтернет-магазину в браузері, що дає змогу заощадити свій час та швидко зробити замовлення.

1.2.2 Інтернет-магазин «Coffeeok»

Coffeeok (рисунок 1.2) – це інтернет-магазин, який спеціалізується на продажу кави, чаю та аксесуарів до них. Також є послуги оренди та ремонту кавомашин [5]. Інтернет-магазин Coffeeok має більш широкий асортимент та пропонує більше послуг ніж магазин Kulturrra. Інтерфейс приваблює своєю простотою: він лаконічний, зручний та має приємний для очей дизайн.

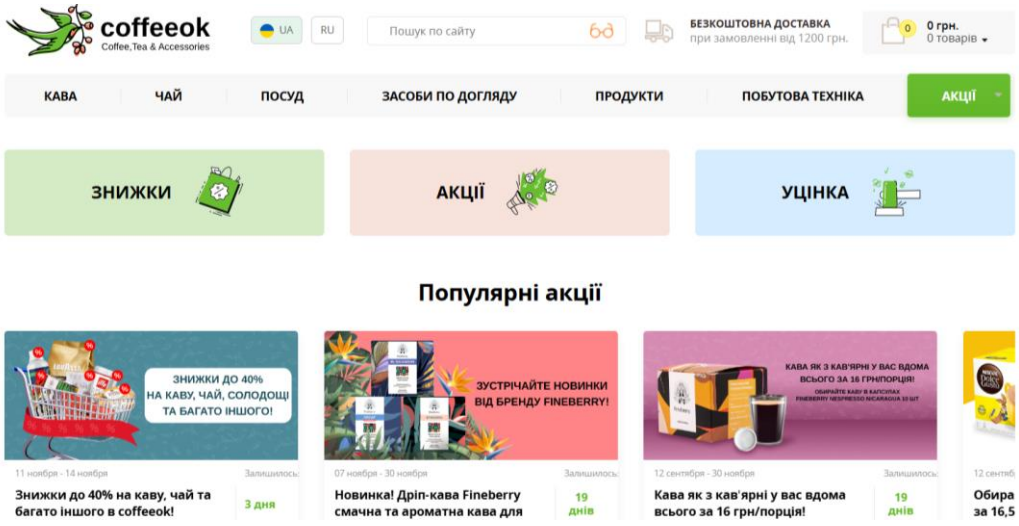


Рисунок 1.2 – Головна сторінка інтернет-магазину «Coffeeok»

Coffeeok пропонує багато різних видів кави та чаю, надаючи до кожного повний опис сорту та його специфічних смаків, що дає змогу користувачам отримати уявлення про товар, який вони збираються купити.

Також інтернет-магазин надає послуги оренди кавомашин, а також пропонує послуги їх ремонту, що розширює їх сферу діяльності.

Якихось підписок чи подарункових сертифікатів інтернет-магазин не пропонує, але на сайті є категорія товарів з уцінкою, де пропонуються товари з незначними пошкодженнями, які зазвичай мають лише візуальний характер та впливають тільки на їхній естетичний вигляд

З недоліків цього сайту можна виділити те, що він не має можливості створення аккаунту, тобто, користувач не може отримати доступ до історії замовлень (для функції збереження даних, для швидкого замовлення в майбутньому) та інших зручностей, які надає система створення профіля користувача. Також цей магазин не має мобільного застосунку, що позбавляє користувачів можливості швидкого доступу, зручних push-сповіщень та інших можливих його переваг.

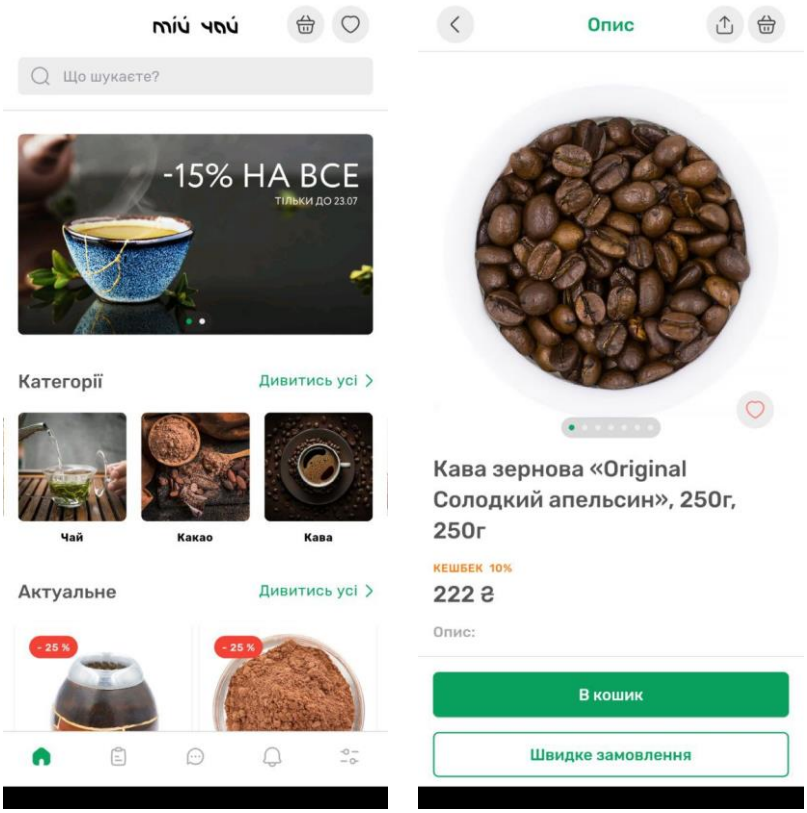
1.2.3 Мобільний застосунок «Мій чай»

Застосунок «Мій чай» – це магазин, який має як вебсайт, так і мобільний застосунок. Магазин спеціалізується на продажу чаю, какао, кави, десертів та аксесуарів до цих категорій товарів (рисунок 1.3 а) [6].

Інтерфейс мобільного застосунку має простий, інтуїтивно зрозумілий дизайн. Мінімалізм, який використовується, дає змогу акцентувати увагу саме на ключових моментах, не відволікаючи користувача від головної цілі застосунку, а саме продажу товарів.

Застосунок пропонує можливість швидкого замовлення без створення аккаунту, що полегшує досвід користувача, який не планує багато замовляти в цьому магазині. Але для того щоб отримати повний функціонал, який пропонується в застосунку, потрібно створити аккаунт. Це дасть змогу зберігати свою адресу (для майбутніх замовлень), отримати доступ до історії своїх замовлень, а також мати можливість користуватись бонусною програмою.

Але цей мобільний застосунок поступається своїм конкурентам у повноті опису товарів, так як більшість товару має дуже стислу характеристику, що може привести до того що користувач буде не задоволений отриманими послугами через неправильні очікування (рисунок 1.3 б). Також, кількість кожного товару сильно обмежена, що ускладнює можливість замовлення великих обсягів.



а) б)

Рисунок 1.3 – Інтерфейс мобільного застосунку «Мій чай»:
а) головний екран; б) детальний опис продукту

1.3 Постановка задачі

Проаналізувавши наведені інтернет-магазини та виділивши позитивні та негативні сторони кожного, було прийнято рішення розробки нативного застосунку для операційної системи Android. У застосунку буде реалізована

авторизація, для того щоб користувач мав змогу легко замовляти товар без введення даних кожен раз, коли він хоче замовити щось, що забезпечить можливість реалізувати інший функціонал для забезпечення комфорту користувачеві.

В застосунку також будуть реалізовані основні функції інтернет-магазину, а саме:

- перегляд товарів;
- пошук товарів та використання фільтрів;
- перегляд кошика;
- додавання товарів до списку обраних;
- додавання карток для оплати;
- додавання адреси користувача;
- створення замовлення;
- перегляд замовлень;
- отримання сповіщень з акціями та статусом замовлень.

Для розробки архітектури буде використана концепція Clean Architecture, яка забезпечує масштабованість та легку підтримку проєктів не зважаючи на їх розмір, розділяючи код на шари, кожен з яких виконує свою чітку задачу та мінімально пов’язаний з іншими. Але для забезпечення чистоти, організованості та ефективності коду все ще не достатньо використовувати тільки один принцип Clean Architecture, тому буде використано і інші концепції, які допоможуть писати простий та ефективний код. Концепція програмування KISS, закликає до спрощення коду, а принцип DRY націлений на мінімізацію дублювання коду, мотивуючи писати код розділяючи його на компоненти, які можна повторно використовувати. Також дуже популярні та необхідні принципи SOLID, які є базовими концепціями об’єктно-орієнтованого програмування. Вони сконцентровані на створенні коду, який буде стійкий до помилок.

Отже, дотримання цих принципів, необхідно для створення якісного застосунку, який буде легко підтримувати та додавати новий функціонал.

2 АНАЛІЗ ТЕХНОЛОГІЙ ТА ПІДХОДІВ, ЯКІ БУЛО ВИКОРИСТАНО У РОЗРОБЦІ ЗАСТОСУНКУ

2.1 Мова програмування Kotlin

Основна ідея створення мови Kotlin була в тому, щоб зробити таку мову програмування, яка б була практичною, а саме була корисною для щоденної розробки та допомагала її користувачам виконувати роботу за допомогою своїх функцій та інструментів.

Багато рішень, які були прийняті при розробці Kotlin, акцентували увагу на тому, наскільки вона є зручною для розробників, беручи натхнення та вдалі рішення з таких мов програмування, як Java, Scala, C# та Groovy. Все це призвело до того, що мова Kotlin почала та продовжує набирати популярності серед розробників, які цінують продуктивність і зручність [7].

Kotlin є мультиплатформною, статично типізованою мовою програмування загального призначення, тобто вона може використовуватись в багатьох сферах програмування: підтримує компіляцію для різних платформ, а саме для JVM, Android, JavaScript та Native. Крім цього Kotlin підтримує прозору взаємодію між різними платформами, використовуючи функції Kotlin Multiplatform Project.

Однією із ключових особливостей є те, що мова Kotlin відрізняє змінні, які можуть не містити ні якого значення, що в мові програмування значить null, від тих, які завжди мають значення (ця функція має назву null-safety). Її використання значно знижує ризик помилок програми і тому застосунки на Kotlin менш схильні до аварійного завершення, що свідчить про надійність мови. Ще однією потужною функцією є Kotlin Coroutines, яка дозволяє ефективно працювати з асинхронними операціями та багатопотоковістю. Це значно спрощує розробку програм, які виконують кілька різних задач одночасно (при цьому зберігаючи лаконічність та структурованість програмного коду).

Слід зазначити, що мова Kotlin повністю сумісна з Java, що стало однією з ключових причин, чому Kotlin було рекомендовано, як мову програмування для Android у 2019 році. Ця особливість, корисна та приваблива для проєктів, які послідовно оновлюються та переходять на Kotlin. Вона не вимагає значних зусиль на вивчення нового синтаксису, оскільки він схожий на Java.

2.2 Операційна система Android

ОС Android – це операційна система з відкритим вихідним кодом, яка заснована на ядрі Linux, що дало змогу забезпечити надійну та гнучку основу для розроблених функцій. Перш за все, вона була розроблена для використання в мобільних пристроях, таких як смартфони, планшети, смарт-годинники.

Історія Android почалась ще у 2003 році, коли була заснована компанія Android Inc. Спочатку вона планувалась як операційна система для цифрових камер, але зрозумівши потенціал проєкту, було вирішено змінити напрямок на створення операційної системи для мобільних пристроїв. У 2005 році компанію Android Inc викупила Google, що ознаменувало її вихід на ринок смартфонів. У 2007 році була представлена бета-версія Android, а 23 вересня 2008 року був комерційний реліз Android 1.0. Одразу ж був представлений перший смартфон з операційною системою Android, а саме HTC Dream, який став першим кроком до глобального поширення платформи.

На сьогоднішній день операційна система Android домінує на мобільному ринку та продовжує постійно розвиватися. Вона займає понад 71% світової частки ринку станом на листопад 2024 року, згідно з даними Statcounter [8], значно випереджуючи своїх конкурентів.

По суті операційна система Android слугує платформою, що дозволяє користувачам легко взаємодіяти зі своїми пристроями, отримувати доступ до багатьох сервісів та запускати безліч різноманітних застосунків.

Основою Android стала модульна архітектура (рисунок 2.1), яка дозволяє виробникам пристроїв та розробникам адаптувати досвід користувача до конкретних апаратних конфігурацій та вподобань користувачів.

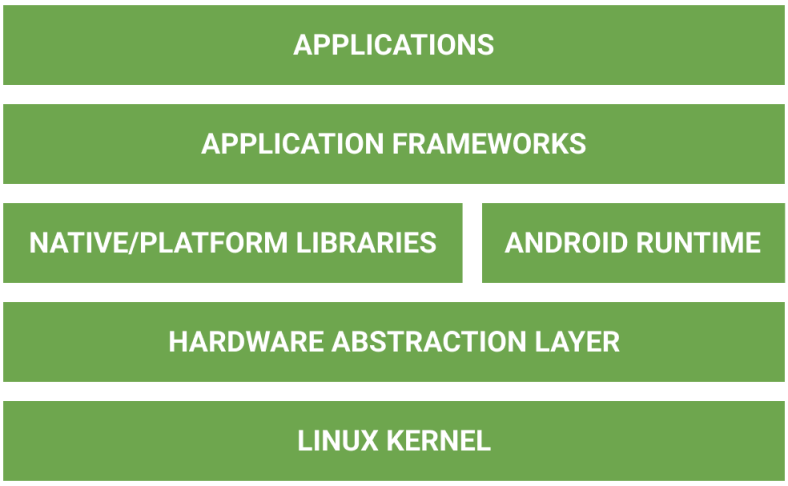


Рисунок 2.1 – Схема архітектури операційної системи Android

Як було зазначено раніше, в основі лежить ядро Linux, яке відповідає за обробку процесів, мережний зв'язок та оптимізацію енергоспоживання, а також за керування драйверами обладнання для основних функцій, таких як дисплей, камера, пам’ять, аудіо та інші.

Далі йде рівень абстракції апаратного забезпечення, який надає уніфікований інтерфейс для апаратних компонентів. Завдяки цьому, виробники апаратного забезпечення можуть реалізовувати специфічні для пристрою функції, при цьому не змінюючи код ядра системи Android, що в свою чергу забезпечує сумісність між різними пристроями.

Наступний рівень – це нативні бібліотеки, які надають основні системні функції операційної системи. Вони забезпечують підтримку мови програмування C, обробку аудіо та відео, управління дисплеями та взаємодію з базами даних та інше. Android Runtime потрібен для виконання та управління кодом застосунків, він компілює код програми у машинний код під час встановлення, що призводить до швидшого їх виконання.

Android фреймворк – це набір класів та бібліотек, які надають основні функції та компоненти для Android програм. Фреймворк дозволяє створювати власні застосунки, використовуючи вже готові компоненти.

На вершині знаходиться рівень застосунків, на якому користувачі безпосередньо взаємодіють зі своїми пристроями за допомогою встановлених програм для Android. Це можуть бути як стандартні застосунки, завантажені з Google Play Store, так і попередньо встановлені. Програми, які встановлюються відразу, часто мають вищий рівень доступу, що дозволяє їм надавати специфічні функції.

Отже, кожен рівень відіграє важливу роль, забезпечуючи безперебійну та ефективну роботу операційної системи Android.

2.3 Фреймворк Jetpack Compose

Jetpack Compose – це сучасний інструментарій, розроблений Google. Він має на меті спростити та покращити процес створення інтерфейсів користувача при розробці Android застосунків на мові програмування Kotlin.

Вперше Jetpack Compose було анонсовано у травні 2019 року і компанія Google почала впровадження нової парадигми розробки для Android, а саме перехід від підходу на основі XML, до більш сучасного способу побудови інтерфейсу користувача. Після анонсу було випущено декілька бета-версій, надаючи можливість розробникам протестувати та підготуватись до нового фреймворку, а розробники Jetpack Compose натомість отримували відгуки та повідомлення про помилки.

Реліз фреймворку відбувся в липні 2021 року. Це означало що розробники Android застосунків могли впевнено використовувати фреймворк у створенні своїх комерційних проєктів, знаючи, що це стабільна версія, яка має офіційну підтримку. У 2023 році, Jetpack Compose отримав велике оновлення, нова версія значно покращила продуктивність, розширила функціональність та почала інтеграцію в нові сфери. Окрім мобільної

розробки тепер підтримуються десктопні та вебзастосунки. Наразі фреймворк продовжує розвиватися, розширюється функціонал та покращується продуктивність. Фреймворк використовує декларативний підхід, який вже використовували фреймворки Flutter, Swift UI та React Native, що дозволяє розробникам описувати інтерфейс користувача як функції, які автоматично реагують на зміну даних [9]. Тобто, замість імперативного підходу, коли явно вказуються кроки, необхідні для оновлення інтерфейсу Jetpack Compose автоматизує процес відображення елементів на основі наданих даних, гарантуючи, що інтерфейс відображає поточний стан. Завдяки цьому розробники можуть писати передбачуваний та легший у налагоджені код, оскільки стан інтерфейсу тісно пов’язаний з його представленням.

2.4 Фреймворк Hilt

Hilt – це популярний фреймворк для інтеграції залежностей в застосунках для Android. Він значно спрощує процес керування ними та підвищує ефективність розробки. Основна ідея Hilt полягає в автоматизації створення та впроваджені залежностей, використовуючи анотації та аналітичний підхід до роботи з ними [10]. Завдяки цьому фреймворку, можна створювати більш організований та простий у підтримці код, без необхідності писати шаблонний код.

Однією з основних функцій фреймворку Hilt є управління життєвим циклом залежностей. Це означає, що контроль над створенням та знищенням залежностей відбувається автоматично, залежно від життєвого циклу компонентів застосунку. Наприклад, ті об’єкти, які мають обмежений термін існування, наприклад об’єкти ViewModel, автоматично будуть створені та знищені відповідно до цих обмежень, що дозволяє уникати проблем із витоками пам’яті та забезпечує коректну роботу застосунку у різних сценаріях.

Також, фреймворк Hilt, може працювати з іншими популярними бібліотеками, наприклад, такими як Room, ViewModel та LiveData. Завдяки цьому створення застосунків з багатошаровою архітектурою не потребує багато зусиль. Фреймворк автоматично генерує граф залежностей, який легко налаштовується, а також забезпечує можливість тестувати окремі компоненти, без необхідності глибоко занурюватися в деталі їх впровадження.

Отже, фреймворк Hilt значно полегшує процес розробки Android застосунків у випадках, коли необхідне впровадження зв'язків між компонентами. Він дозволяє ефективно розробляти застосунки, без необхідності ручного управління залежностями. Hilt підтримує як автогенерацію зв'язків, так і ручне налаштування, що спрощує впровадження різних архітектурних підходів, таких як Clean Architecture та MVVM. Завдяки цьому фреймворку можна легко забезпечити стабільність та продуктивність застосунку, не зважаючи на розмір проєкту.

2.5 Система керування базами даних PostgreSQL

PostgreSQL – це система керування базами даних з відкритим вихідним кодом. На відміну від інших систем, PostgreSQL, підтримує як нереляційні, так і реляційні типи даних, що робить її однією з найбільш сумісних та стабільних систем керувань базами даних.

Історія PostgreSQL почалась під керівництвом професора Майкла Стоунбрейкера в Каліфорнійському університеті Берклі. Спочатку система керування базами даних називалась POSTGRES та підтримувала кілька типів даних. У 1996 році проєкт перейменували на PostgreSQL, що стало відображати його підтримку мови SQL.

До основних переваг PostgreSQL включають підтримку складних запитів та велику гнучкість завдяки можливості розширення функціоналу через плагіни та розширення. А також до переваг можна віднести підтримку

високого рівня безпеки завдяки підтримки властивостей ACID, що робить транзакції бази даних надійними та вказує на те, що дані, які зберігаються точні, оскільки неповні зміни ніколи не зберігаються [11].

Оскільки система керування базами даних PostgreSQL є проєктом з відкритим вихідним кодом, вона не залежить від однієї комерційної організації, що дозволяє їй залишатись надійною та безкоштовною для використання у будь-яких типах проєктів, не залежно чи це комерційна, чи особиста розробка. А також завдяки своїй відкритості, система керування базами даних PostgreSQL, має велику спільноту розробників: вони постійно підтримують її розвиток як фінансово так і беручи пряму участь в її розробці, виправляючи помилки та додаючи функції.

Отже, завдяки багаторічному розвитку, система керування базами даних PostgreSQL має високий рівень стабільності та довговічності, що робить її однією з найкращих систем для управління даними у сучасних умовах.

2.6 Інструмент для автоматизації збірки Gradle

Gradle – це інструмент для автоматизації збірки, який використовується для створення, тестування та розгортання програм різних платформах. Він розповсюджується з відкритим вихідним кодом написаним на Java, Groovy та Kotlin. Gradle підтримує широкий спектр мов програмування, таких як Java, Kotlin, Groovy, C++, Scala та Swift.

Gradle забезпечує ефективність та гнучкість у процесах розробки, дозволяє автоматизувати ключові етапи, включаючи компіляцію, зв’язування, пакування та управління залежностями, що робить його особливо популярним серед розробників Java, Android, Scala.

Gradle вирішує недоліки інших інструментів збірки, таких як Maven та ANT, поєднуючи їхні переваги. Він дозволяє адаптувати збірку під будь-який тип проєкту, від Android до багатоплатформних рішень. Завдяки кешуванню

завдань Gradle виконує тільки ті дії, які потребують змін, що оптимізує та прискорює процес. Інструмент також аналізує виконання збірки, виявляючи проблеми продуктивності та помилки. Gradle може використовувати репозиторії Maven для управління залежностями та інтегрувати завдання з проєктів ANT.

Gradle використовує систему проєктів та завдань. Проєкти представляють компоненти програмного забезпечення, які необхідно створити або розгорнути. Завдання – це дії які виконує Gradle, наприклад, компіляція коду та запуск тестів. Завдання можуть бути динамічними та виконуватися одночасно. Всі аспекти збірки проєкту визначаються у файлах конфігурації.

Отже, завдяки своїй універсальності, продуктивності та гнучкості Gradle є популярним інструментом у розробці програмного забезпечення.

2.7 Середовище розробки Android Studio

Android Studio є офіційним середовищем розробки для створення застосунків на платформі Android [12]. Воно засноване на середовищі розробки IntelliJ IDEA, яке є популярним серед багатьох розробників, використовуючи основні його переваги та додаючи інші необхідні інструменти для створення, налагодження та тестування застосунків.

Середовище розробки було вперше анонсоване на конференції Google у травні 2013 року, а перша стабільна версія була випущена у грудні 2014 року, замінивши середовище розробки Eclipse, як основний інструмент для розробки Android застосунків.

Android Studio використовує систему збірки на основі Gradle, емулятор Android та різні шаблони коду. Проєкти у середовищі Android Studio можуть мати один або кілька модулів, які містять код та ресурси.

Також у середовищі є функція Apply Changes, яка дозволяє вносити зміни в код та ресурси, що застосовуються до запущеного застосунку в

реальному часі, без необхідності кожний раз перезапускати його. Вбудований редактор коду допомагає з написанням коду, пропонуючи автозавершення, рефакторинг та його аналіз. Застосунки, створені в Android Studio, можна компілювати у формат APK або AAB, за допомогою інтегрованого інструментарію, це забезпечує можливість подальшого розміщення на Google Play Store.

Отже, середовище розробки Android Studio дозволяє створювати високоякісні застосунки для Android та підтримує весь процес розробки – від кодування до тестування та розгортання програм.

2.8 Інструмент для тестування мобільних застосунків Android Emulator

Android Emulator у Android Studio – це інструмент для емуляції Android пристроїв, який інтегрований у середовищі розробки Android Studio. Він використовується для тестування та налагодження мобільних застосунків при їх розробці [13], що можливо завдяки точному моделюванню апаратного й програмного забезпечення Android пристрою.

Використання інструменту Android Emulator дозволяє тестувати застосунки на різних версіях ОС Android, екранах та конфігураціях без використання фізичних пристроїв. Можна моделювати різні сценарії роботи мобільного пристрою, такі як слабке інтернет-з’єднання, низький заряд батареї або при специфічних умовах роботи датчиків.

Для емуляції використовується технологія апаратної віртуалізації QEMU, що забезпечує ефективне імітування функцій операційної системи Android. Створення віртуальних пристроїв відбувається за допомогою AVD Manager: при створенні вказується параметри пристрою такі як, роздільна здатність екрану, обсяг пам’яті та інші.

Android Emulator підтримує інкрементну емуляцію, тобто виконуються тільки ті дії, які зазнали змін, уникаючи перезапуски чи перевстановлення, що сильно пришвидшує тестування. Також він легко інтегрується з іншими

інструментами Android Studio, а саме Logcat, Layout Inspector та Profiler, що дозволяє ефективно тестувати застосунки з мінімальними затратами часу та ресурсів.

Отже, Android Emulator є ключовим інструментом тестування Android застосунків, відтворюючи реальні сценарії використання, такі як вхідні дзвінки, зміна геолокації чи заряд батареї. Android Emulator дає змогу забезпечити високу якість та стабільність роботи застосунку на різних пристроях та версіях Android.

2.9 Система контролю версій Git

Git – це система контролю версій, яка необхідна для відстеження змін що вносяться до файлів, це дає змогу зберігати історію роботи та повертатись до попередній версій за потреби [14]. Крім того, Git полегшує роботу у команді, дозволяючи об’єднувати зміни від кількох людей у спільний проєкт, що робить її корисною як для індивідуального використання, так і для командної роботи.

Програмне забезпечення Git працює локально, тобто всі файли та історія змін зберігаються на комп’ютері, де знаходиться ця система. Однак, при бажанні застосовувати її в команді, можна використовувати онлайн-хостинги, такі як GitHub або GitLab. Вони дозволяють створити центральне сховище, куди учасники можуть завантажувати свої зміни, отримувати оновлення від інших, а також ефективно координувати спільну розробку.

З системою контролю версій Git, можна працювати як через командний рядок, так і використовуючи програми з графічним інтерфейсом, такі як GitHub Desktop.

Основним елементом роботи з Git є репозиторій, де містяться усі файли проєкту та повна історія змін, включно з даними про авторів, дати та коментарі до кожного коміту. Уся ця інформація, яка необхідна для відстеження змін, зберігається в окремій, прихованій папці в репозиторії.

Процес збереження змін у системі Git складається з двох етапів. Спочатку відбувається додавання файлів до індексації, після чого йде створення комітів. Таке розділення на етапи дає змогу обирати, які саме файли фіксувати, а які файли ще не готові до збереження, що забезпечує повний контроль над змінами, які мають потрапити до історії проєкту.

Окремою особливістю Git є робота з гілками, які дозволяють працювати над різними задачами незалежно одна від одної. Наприклад, можна створити гілку для розробки нового функціонала застосунку, а при необхідності, вносити інші виправлення в основну гілку. Після завершення створення нового функціоналу, можна об’єднати нову гілку з основною. При об’єднані гілок можуть виникнути конфлікти, якщо зміни торкаються одних і тих самих файлів, але Git надає інструменти для їх вирішення. Такий підхід забезпечує гнучкість та безпеку при паралельній роботі кількох розробників.

Отже, завдяки своїй гнучкості, можливості працювати локально та з віддаленими репозиторіями, а також підтримці гілок та об’єднання змін, Git став незамінним інструментом у розробці програм.

2.10 Концепція Clean Architecture

Clean Architecture – це концепція проєктування програмного забезпечення, яка направлена на чіткий поділ відповідальності між різними частинами програми. Головна ціль такого поділу – створити структуру, яка б легко могла адаптуватись до змін в програмі, а також без проблемно тестуватись та підтримуватись [15]. Для цього треба, щоб внутрішні шари були залежними лише від абстракцій, а не від реалізацій зовнішніх шарів. Залежність від абстракцій дозволяє зберігати стабільність та незалежність бізнес-логіки від деталей її реалізації (рисунок 2.2).

Перший шар Presentation відповідає за інтерфейс користувача, він займається обробкою введених даних, відображенням інформації та управлінням взаємодії користувача із застосунком. Presentation використовує

шар Use Cases для отримання даних та виклику відповідних дій, але не має прямого доступу до бізнес-логіки або деталей даних. Завдяки такому підходу зміни в дизайні інтерфейсу користувача можуть бути реалізовані без необхідності змінювати внутрішню логіку програми.

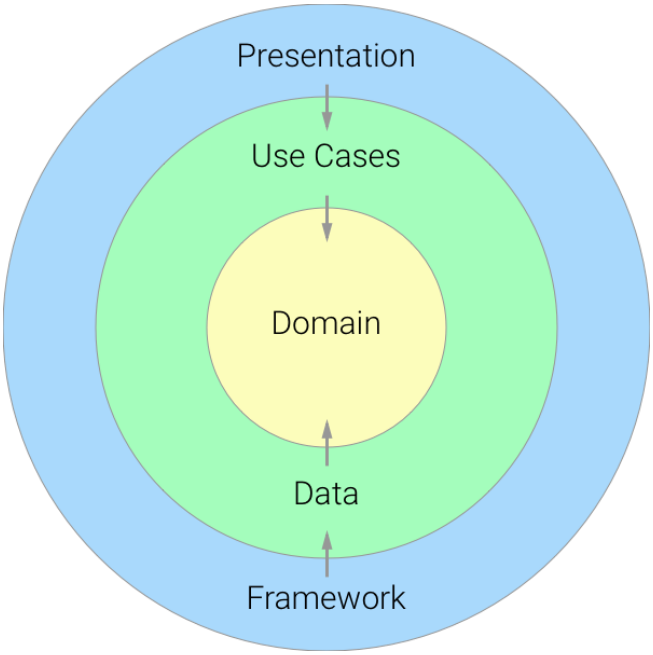


Рисунок 2.2 – Схема шарів концепції Clean Architecture

Шар Use Cases, використовується для взаємодії користувача із бізнес-логікою програми. У ньому реалізуються конкретні сценарії використання, що відображають бізнес-правила та потреби застосунку. Для доступу до даних чи зовнішніх ресурсів, використовуються абстракції, завдяки чому, шар Use Cases, не залежить від їхньої конкретної реалізації, що дозволяє легко адаптувати застосунок до нових вимог.

Третій шар Domain є ядром програми, який містить усі бізнес-моделі, правила та сутності. Цей шар незалежний від зовнішніх технологій чи інструментів, і є основою для інших частин застосунка. В цьому шарі визначаються ключові поняття та об'єкти, які є універсальними та залишаються актуальними незалежно від використаних технологій чи середовища виконання. Наприклад, тут створюються сутності, що описують користувачів, продукти, транзакції або інший контент.

Наступний шар Data займається отриманням, зберіганням та наданням даних іншим частинам застосунку. Він робить абстракцію над джерелами даних, таких як бази даних чи зовнішніх API.

У цьому шарі створюються репозиторії, які забезпечують доступ до даних через задані інтерфейси, які були оголошені в Domain шарі. Завдяки такому підходу, застосунок залишається гнучким до змін у способах отримання та зберігання даних, з можливістю легкої інтеграції нових джерел даних, незалежно від того, чи є вони локальними, чи віддаленими.

Останній шар – це Framework, який зазвичай охоплює інфраструктурні елементи застосунку. Це можуть бути зовнішні бібліотеки, сервіси або API, які необхідні для роботи застосунку. Цей шар може змінюватись в залежності від обраних технологій та бібліотек, але ці зміни не повинні викликати необхідність вносити глобальні правки в інші шари.

Отже, перевага чистої архітектури полягає в її гнучкості, що дозволяє легко адаптуватись до змін внесених у застосунок. Завдяки тому, що кожний шар не залежний від реалізації інших, можна легко проводити тестування кожного окремого компонента програми, а також розділення задач між шарами, знижує ризик виникнення помилок, що значно спрощує підтримку застосунку. Крім того, завдяки цьому підходу можна швидко інтегрувати нові функціональності без значного впливу на існуючий код.

Але існують певні недоліки чистої архітектури. Наприклад, для невеликих або простих застосунків її повне впровадження може виявитися зайвим, тому можна реалізувати окремі принципи цієї архітектури. Крім того, для новачків у програмуванні або тих хто тільки почав працювати з чистою архітектурою, може знадобитись більше часу на її освоєння та розуміння.

Таким чином, концепція Clean Architecture сприяє створенню стійких та ефективних застосунків. Її переваги стають найбільш помітними в масштабних проєктах, де кожен шар архітектури відповідає за певні завдання, забезпечуючи гнучкість та стабільність.

2.11 Архітектурний патерн MVVM

MVVM – це архітектурний патерн, який зазвичай використовується в розробці Android застосунків. Він допомагає ефективно організовувати взаємодію між різними складовими програми, відокремлюючи бізнес-логіку від інтерфейсу користувача. MVVM будується на наступних рівнях: Model, View та ViewModel (рисунок 2.3).

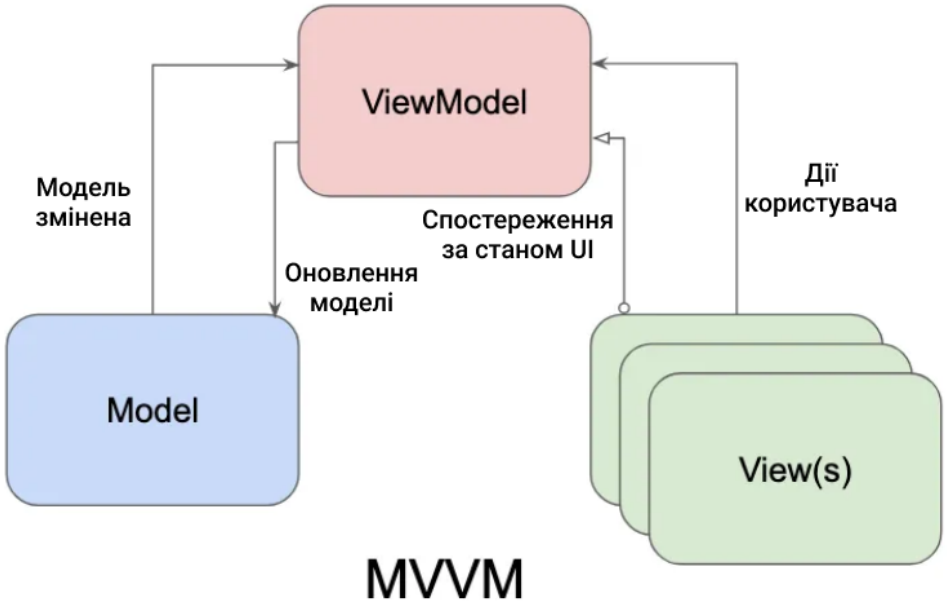


Рисунок 2.3 – Схема архітектурного патерну MVVM

Перший рівень (Model) відповідає за збереження та обробку даних застосунку, а також реалізацію його бізнес-логіки. Через нього відбувається взаємодія з базами даних, сервісами або іншими джерелами даних. Рівень моделі, реалізує правила за якими відбувається отримання, збереження, оновлення та видалення даних. Зазвичай цей рівень будується за принципами єдиної відповідальності для забезпечення чистого коду та зменшення залежностей. Тобто в модель можуть входити класи, що представляють дані користувачів або товарів з їх параметрами та методами для управління цими даними.

Наступний рівень (View) представляє собою інтерфейс користувача: елементи, які взаємодіють безпосередньо з користувачем, наприклад, кнопки, текстові поля та списки. Основна функція цього рівня полягає у відображенні даних, які користувач повинен бачити та з якими може взаємодіяти.

ViewModel – це останній рівень, який виступає посередником між Model та View. Він забезпечує доступ до даних, які підготовлені Model та надає їх у вигляді, зручному для відображення в View. Крім того, ViewModel обробляє взаємодії користувача, такі як натискання кнопок, введення даних чи перемикання станів і, якщо потрібно, повідомляє Model про необхідність оновлення даних. Завдяки тому що ViewModel не залежить від реалізації View, можна змінювати структуру та оформлення інтерфейсу без порушення бізнес-логіки або необхідності змінювати код в Model.

Отже, патерн MVVM дозволяє чітко розділити відповідальності між бізнес-логікою, інтерфейсом користувача та взаємодією між ними, що забезпечує кращу організацію коду, зменшення залежностей та повторюваності коду, спрощує підтримку та масштабування застосунку.

Але шаблон MVVM має і недоліки. Наприклад, у великих проєктах ViewModel може стати перенавантаженим надмірною кількістю бізнес-логіки, що може ускладнити його тестування та підтримку. В такому випадку можна дотримуватись принципів чистого коду, що допоможе виправити цей недолік.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ

3.1 Опис структури застосунку

Структура застосунку організована за принципами чистої архітектури, яка дозволяє чітко розділити логіку на окремі рівні, що відповідають різним аспектам функціональності застосунку. Тому кожен файл створюється та зберігається в підготовленому каталозі, що відповідає за той шар з яким працює цей компонент (рисунок 3.1). Такий підхід забезпечує модульність, легкість у тестуванні та розширюваність системи. Кожен рівень має мінімальну залежність від інших шарів, що сприяє незалежності та гнучкості компонентів.

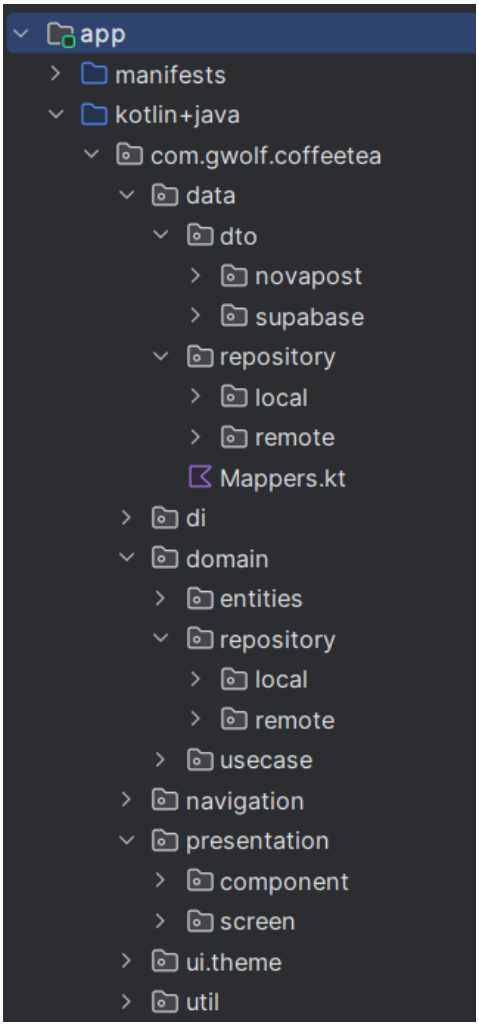


Рисунок 3.1 – Структура каталогів проєкту

3.2 Каталог domain

Каталог domain відповідає за реалізацію бізнес-логіки застосунку та взаємодію між різними об’єктами. На цьому рівні створюються інтерфейси репозиторіїв (лістинг 3.1), які визначають контракти для взаємодії з постачальниками даних, дозволяючи змінювати джерело даних без зміни бізнес-логіки додатку. Це означає, що бізнес-логіка не залежить від конкретної реалізації роботи з даними.

Лістинг 3.1 – Визначення контракту для постачальника даних

```
interface ProfileRepository {
    fun getProfile(): Flow<Profile?>
    fun uploadProfileImage(byteArray: ByteArray): Flow<String>
    fun updateProfileImagePath(imagePath: String): Flow<Unit>
    fun updateEmail(newEmail: String): Flow<Unit>
    fun updatePhone(newPhone: String): Flow<Unit>
    fun updatePassword(newPassword: String): Flow<Unit>
    fun updateNameInfo(firstName: String, lastName: String,
patronymic: String): Flow<Unit>
}
```

Бізнес-операції та обробка даних реалізуються у класах UseCase, які виконують конкретні дії або процеси. Такі класи приймають в свій конструктор необхідні інтерфейси, які впроваджуються завдяки фреймворку Hilt та його анотації @Inject (лістинг 3.2).

Лістинг 3.2 – Приклад конструктора UseCase класу

```
class AddOrderUseCase @Inject constructor(
    private val ordersRepository: OrdersRepository,
    private val productRepository: ProductRepository,
    private val cartRepository: CartRepository
)
```

Далі створюється метод invoke(), в якому викликаються функції інтерфейсів з конструктора (лістинг 3.3). Отримані дані проходять необхідні етапи обробки та конвертації. Після чого передаються далі для використання в ViewModel класах та відображення в інтерфейсі користувача.

Лістинг 3.3 – Метод UseCase класу

```
class GetPromotionsUseCase @Inject constructor(
    private val promotionRepository: PromotionRepository
) {
    operator fun invoke(): Flow<DataResult<List<Promotion>>> =
flow {
        try {
            promotionRepository.getPromotions().collect {
response ->
                emit(DataResult.Success(data = response))
            }
        } catch (e: Exception) {
            emit(DataResult.Error(exception = e))
        }
    }}
```

Методи, що взаємодіють із постачальниками даних, зазвичай є асинхронними, тому такі методи повертають Flow, що є асинхронним потоком, який може змінюватись з часом. Завдяки цьому можна забезпечити обробку станів та подій під час отримання даних.

Для обробки результатів запитів був створений клас, який повертає стан операції (лістинг 3.4). Цей клас описує можливі варіанти результатів, такі як успішний запит або помилка. Таким чином, при успішному виконанні операції, функція повертає дані, тоді як у випадку помилки або відсутності даних функція повертає інформацію про збій. Це дозволяє адекватно обробляти помилки та надавати відповідну інформацію користувачу в інтерфейсі, забезпечуючи тим самим зручність та інформативність при використанні застосунку.

Лістинг 3.4 – Клас-обгортка для обробки результатів операцій

```
sealed class DataResult<out R> {
    data class Success<out T>(val data: T) : DataResult<T>()
    data class Error(val exception: Throwable) :
DataResult<Nothing>()}
```

Також в цій папці знаходяться класи-сутності (лістинг 3.5), які описують основні характеристики об’єктів бізнес-логіки, наприклад, таких як користувачі або продукти.

Лістинг 3.5 – Приклад класу-сутності

```
data class OrderItem(
    val id: String = "",
    val orderId: String,
    val productId: String,
    val product: Product,
    val quantity: Int,
)
```

3.3 Каталог di

Першим розглянемо каталог di, в якому знаходяться модулі, необхідні для налаштування фреймворку Hilt. Класи, які позначені анотацією @Module забезпечують впровадження залежностей, автоматично створюючи та надаючи їх іншим компонентам застосунку. Це можливо завдяки анотаціям які надає фреймворк, і якими позначаються класи, об’єкти та змінні для яких необхідно впровадження зв’язків (лістинг 3.6).

Лістинг 3.6 – Оголошення модуля для Hilt

```
@Module
@InstallIn(SingletonComponent::class)
object RepositoryModule
```

В такому модулі знаходяться функції, які є постачальниками залежностей. Вони позначаються спеціальною анотацією @Provides, а також зазвичай анотацією @Singleton, яка оголошує їх як об’єкти Singleton, щоб мати лише один екземпляр під час роботи застосунку. Такі функції використовуються, наприклад, для створення зв’язку між інтерфейсом репозиторію та його реалізацією (лістинг 3.7).

Лістинг 3.7 – Приклад функції-постачальника залежностей

```
@Provides
@Singleton
fun providePromotionRepository(
    postgrest: Postgrest
): PromotionRepository = PromotionRepositoryImpl(postgrest)
```

3.4 Реалізація збереження та доступу до даних

Перед тим як перейти до наступного каталогу, що описує data шар, доцільно визначити засоби збереження даних. Оскільки основна частина інформації магазину має структурований характер, було обрано реляційну базу даних. Такий вибір зумовлений її здатністю забезпечувати цілісність, узгодженість та ефективну обробку даних. Тому після невеликого дослідження було вибрано сервіс Supabase, який надає доступ до створення та хостингу бази даних PostgreSQL.

Наступним кроком стало створення та налаштування таблиць бази даних (лістинг 3.8). Було визначено основні сутності, їх атрибути та зв’язки між таблицями, що забезпечило логічну структуру даних та ефективну взаємодію між компонентами застосунку (рисунок 3.2).

Лістинг 3.8 – Приклад створення таблиці для обраних товарів

```
create table public.favorites (
  created_at timestamp without time zone null default
CURRENT_TIMESTAMP,
  user_id uuid not null,
  id uuid not null default gen_random_uuid (),
  product_id uuid not null,
  constraint favorites_pkey primary key (id),
  constraint favorites_product_id_fkey foreign KEY (product_id)
references products (id) on update CASCADE on delete CASCADE,
  constraint favorites_user_id_fkey foreign KEY (user_id)
references users (id) on delete CASCADE
) TABLESPACE pg_default;
```

Також були створені тригери та написані відповідні функції (лістинг 3.9), що дозволило зменшити кількість запитів зі сторони застосунку, знизити навантаження на сервер та підвищити загальну продуктивність системи. Використання тригерів забезпечило автоматичне оновлення пов’язаних записів, що значно оптимізувало обробку даних, зменшивши ризик дублювання логіки на рівні застосунку та бази даних, а також покращило ефективність роботи всієї системи.

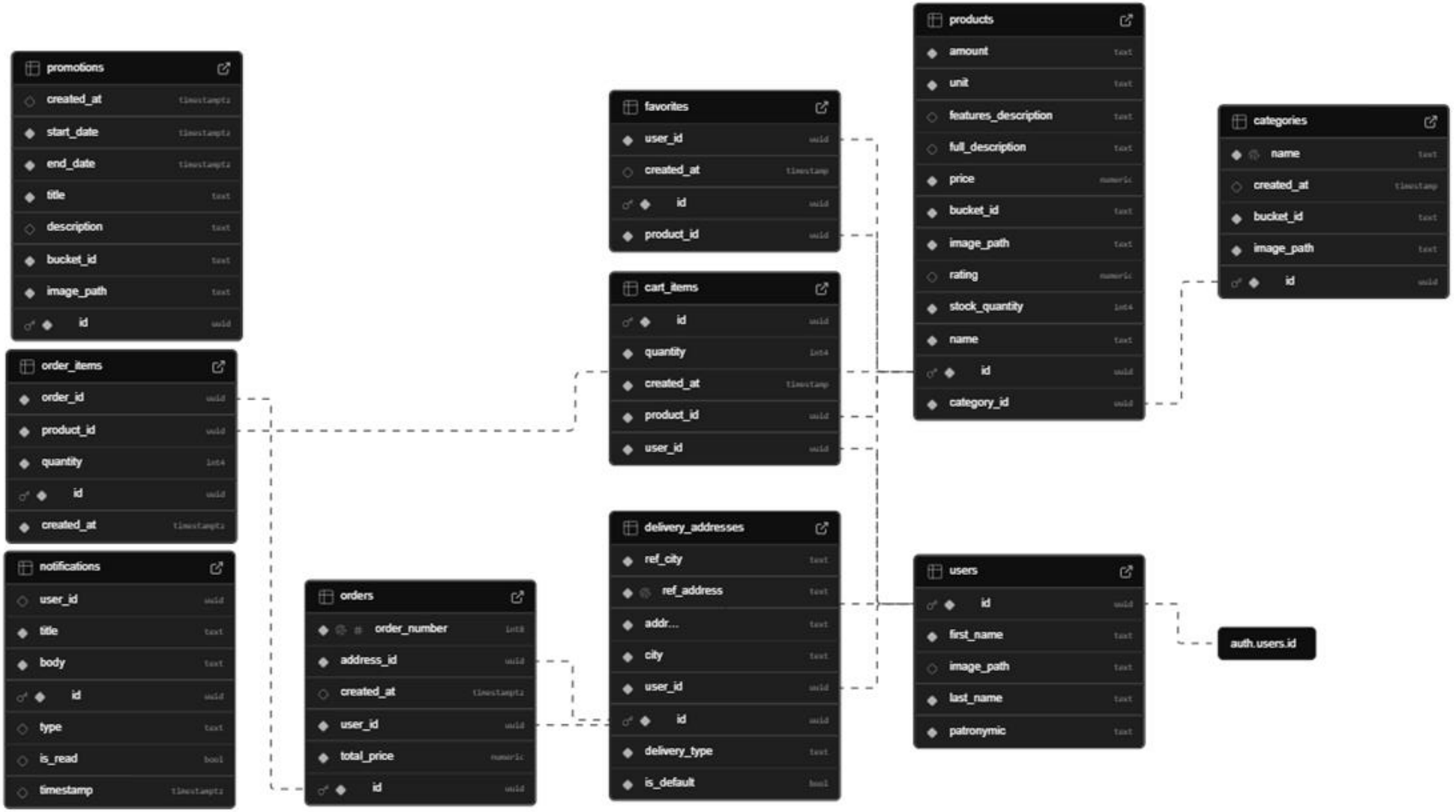


Рисунок 3.2 – Структура таблиць та зв`язки бази даних

Лістинг 3.9 – Приклад функції для тригеру

```
BEGIN
UPDATE public.cart
SET quantity = cart.quantity + NEW.quantity,
    added_at = NOW()
WHERE user_id = NEW.user_id AND product_id = NEW.product_id;
IF NOT FOUND THEN
    RETURN NEW;
END IF;
RETURN NULL;
END;
```

Для кожної таблиці були налаштовані правила доступу та використання CRUD-операцій (лістинг 3.10). Політики доступу були налаштовані таким чином, щоб обмежити доступ до даних лише для неавторизованих користувачів, що підвищує рівень захисту.

Використання цих правил дозволяє забезпечити цілісність та конфіденційність інформації, запобігаючи несанкціонованому доступу та операціям з даними, дозволяючи доступ лише тим користувачам, які успішно пройшли аутентифікацію.

Лістинг 3.10 – Правило доступу до обраних товарів користувача

```
CREATE POLICY "Users can select their own favorites"
ON favorites FOR SELECT TO authenticated
USING ( user_id = (select auth.uid()));
```

Крім того, Supabase надає можливість зберігання зображень у хмарному сховищі, що використовується для зберігання аватарів користувачів, зображень товарів магазину та промоакцій. Це дозволяє ефективно управляти медіа-контентом, забезпечуючи зручність доступу та масштабованість.

Для забезпечення конфіденційності та контролю доступу до цих зображень були налаштовані відповідні політики доступу (лістинг 3.11). Правила обмежують доступ до даних таким чином, що тільки авторизовані користувачі можуть переглядати або змінювати зображення, пов'язані з їх обліковими записами.

Лістинг 3.11 – Правило доступу до зображення профіля користувача

```
((bucket_id = 'profiles'::text) AND (( SELECT (auth.uid())::text
AS uid) = (storage.foldername(name))[1]))
```

3.5 Каталог data

В каталозі data знаходяться файли, які відповідають за взаємодію з джерелами даних, такими як база даних, віддалені API або локальні сховища.

Для зручної роботи з даними в межах застосунку та їх обміну із зовнішніми джерелами були створені моделі DTO (лістинг 3.12), які використовуються для стандартизації та структуризації даних.

Лістинг 3.12 – Приклад моделі DTO

```
@Serializable
data class NovaPostCityDto (
    @SerialName("Ref") val ref: String,
    @SerialName("Description") val name: String
)
```

Для моделей створюються функції конвертери (лістинг 3.13), які призначені для перетворення DTO з data рівня у класи-сутності з domain шару. Ці функції використовуються в класах репозиторіях.

Лістинг 3.13 – Функція для перетворення сутності з data в domain шар

```
fun OrderItemDto.toDomain(imageUrl: String) = OrderItem(
    id = id,
    orderId = orderId,
    productId = productId,
    product = product?.toDomain(imageUrl)!!,
    quantity = quantity
)
```

Локальне сховище використовується для збереження простих налаштувань користувача, наприклад, для збереження такої інформації, як стан параметра «запам’ятати користувача». Для цього було використано DataStore – рішення для зберігання даних, яке дозволяє зберігати пари ключ-значення або типізовані об’єкти.

DataStore забезпечує зручну та ефективну роботу з невеликими об’ємами даних, зберігаючи налаштування в асинхронному режимі без необхідності використовувати базу даних. Його можна зручно використовувати у будь-якому місці застосунку без додаткових налаштувань, оскільки екземпляр DataStore створюється як розширення контексту програми (лістинг 3.14).

Лістинг 3.14 – Створення екземпляру DataStore

```
val Context.dataStore: DataStore<Preferences> by
preferencesDataStore(name = "settings")
```

Як уже зазначалося вище, DataStore може зберігати пари ключ-значення, тому для зручного доступу до них, їх ключі можна зберігати у відкритому об’єкті, що буде містити усі екземпляри ключів (лістинг 3.15). Це дозволяє зберігати ключі в централізованому місці, що спрощує їх управління та доступ у різних частинах застосунку.

Зберігання ключів у окремому об’єкті також дозволяє уникнути дублювання коду та забезпечує зручну підтримку та розширення налаштувань в майбутньому, якщо потрібно буде додавати нові ключі.

Лістинг 3.15 – Об’єкт, що зберігає екземпляри ключів DataStore

```
object PreferencesKey {
    val onBoardingKey = booleanPreferencesKey(name =
"on_boarding_completed")
    val rememberUserKey = booleanPreferencesKey(name =
"remember_user")
}
```

Для доступу до DataStore, щоб зберігати та отримувати дані, в каталозі data розміщується репозиторій, який відповідає за роботу з локальним сховищем даних (лістинг 3.16). Завдяки використанню узагальнених типів, функції в репозиторії здатні працювати з різними типами даних без додаткових змін.

Лістинг 3.16 – Реалізація репозиторія для роботи з даними DataStore

```
class DataStoreRepositoryImpl(context: Context):
DataStoreRepository
{
private val dataStore = context.dataStore
    override suspend fun <T> saveState(key: Preferences.Key<T>,
value: T) {
        dataStore.edit {preferences ->
preferences[key] = value
}
    }
    override fun getDataStorePreferences(): Flow<Preferences> {
        return dataStore.data
    }
}
```

Наступним кроком після налаштування локального сховища стало інтегрування бази даних для зберігання більш складних та структурованих даних. Сервіс Supabase надає готову клієнтську бібліотеку для взаємодії з базою даних PostgreSQL. Для налаштування інтеграції потрібно створити екземпляр клієнта Supabase, який дозволяє підключатися до бази даних, виконувати запити та працювати з іншими компонентами (лістинг 3.17).

Лістинг 3.17 – Налаштування клієнта Supabase

```
@Provides
@Singleton
fun provideSupabaseClient(): SupabaseClient {
    return createSupabaseClient(
        supabaseUrl = BuildConfig.SUPABASE_URL,
        supabaseKey = BuildConfig.SUPABASE_ANON_KEY
    ) {
        install(Postgrest)
        install(Auth) {
            flowType = FlowType.PKCE
            scheme = "app"
            host = "supabase.com"
        }
        install(Storage)
    }
}
```

Після цього треба налаштувати доступ до даних через репозиторії щоб виконувати CRUD-операції в застосунку (лістинг 3.18).

Лістинг 3.18 – Реалізація репозиторія для роботи з таблицею пропозицій

```
class PromotionRepositoryImpl @Inject constructor(
    private val postgrest: Postgrest,
    private val storage: Storage
) : PromotionRepository {
    override fun getPromotions(): Flow<List<Promotion>> = flow {
        val response = withContext(Dispatchers.IO) {
            postgrest.from(PROMOTIONS_TABLE)
                .select()
                .decodeList<PromotionDto>()
        }
        val data = response.map { promotion ->
            val imageUrl = storage.from(promotion.bucketId)
                .createSignedUrl(promotion.imagePath,
HOURS_EXPIRES_IMAGE_URL.hours)
            return@map promotion.toDomain(imageUrl)
        }
        emit(data)
    }
}
```

У застосунку взаємодія із віддаленими API реалізована за допомогою фреймворку Ktor. Перед початком обміну даними із сервером необхідно створити екземпляр клієнту, який міститиме налаштування для виконання мережних запитів (лістинг 3.19). У застосунку це реалізовано через механізм впровадження залежностей фреймворку Hilt, що дозволяє забезпечити єдиний екземпляр клієнту. Такий підхід використовується для всіх об’єктів, які часто використовуються в застосунку. Наприклад, у проєкті реалізовано запити до API «Нової Пошти», що дозволяє отримувати інформацію про наявність відділень у різних містах (лістинг 3.20).

Лістинг 3.19 – Створення екземпляру клієнта Ktor

```
@Singleton
@Provides
fun provideHttpClient(): HttpClient {
    return HttpClient(Android) {
        install(ContentNegotiation) {
            json( Json { ignoreUnknownKeys = true } )
        }
        install(Logging) { level = LogLevel.ALL }
    }
}
```

Лістинг 3.20 – Запит для пошуку міст в яких є відділення Нової Пошти

```
override fun getCitiesBySearch(query: String): Flow<List<City>>
= flow {
    val novaPostProperties = NovaPostProperties.GetCity(
        findByString = query
    )
    val novaPostBody = NovaPostBody(
        apiKey = BuildConfig.NOVA_POST_API,
        modelName = NOVA_POST_ADDRESS_MODEL,
        calledMethod = NOVA_POST_GET_CITIES,
        methodProperties = novaPostProperties
    )
    val response = httpClient.post(NOVA_POST_API) {
        contentType(ContentType.Application.Json)
        setBody(novaPostBody)
    }.body<NovaPostResponse<NovaPostCityDto>>().data
    emit(response.toListCityDomain())
}
```

3.6 Каталог presentation

Останнім ключовим каталогом є presentation, який містить класи, що описують інтерфейс користувача та відповідні класи ViewModel. Як правило, кожен екран має свій індивідуальний клас ViewModel, що реалізує всю необхідну бізнес-логіку для отримання та відображення потрібної інформації. Для кращого розуміння того, як відбувається взаємодія між інтерфейсом та бізнес-логікою, розглянемо, як працює клас ViewModel.

Кожен клас ViewModel позначається спеціальною анотацією @HiltViewModel, яку надає фреймворк Hilt. Це забезпечує подальшу інтеграцію екземпляра ViewModel в функції, що описують інтерфейс користувача.

В конструктор класу вписуються відповідні UseCase класи для виконання необхідних дій (лістинг 3.21). Далі вказуються змінні, які зберігають стан даних інтерфейсу (лістинг 3.22). Одна з них є приватною змінною, яка містить поточний стан екрана, а інша публічна, що дозволяє читати стан ззовні, але не змінювати його, забезпечуючи таким чином інкапсуляцію і контроль доступу до даних.

Лістинг 3.21 – Приклад визначення класу ViewModel

```
@HiltViewModel
class SavedAddressesViewModel @Inject constructor(
    private val getAddressListUseCase: GetAddressListUseCase,
    private val updateSavedAddressUseCase:
UpdateSavedAddressUseCase,
    private val removeSavedDeliveryUseCase:
RemoveSavedDeliveryUseCase) : ViewModel()
```

Лістинг 3.22 – Приклад визначення змінних для збереження стану

```
private var _state =
MutableStateFlow(SavedAddressesScreenState())
val state: StateFlow<SavedAddressesScreenState> =
_state.stateIn(
    scope = viewModelScope,
    started = SharingStarted.WhileSubscribed(stopTimeoutMillis =
5000),
    initialValue = SavedAddressesScreenState()
)
```

Зазвичай вся інформація, яка використовується для відображення на екрані або за допомогою якої вирішується, що саме відображати, зберігається в дата класі, який і описує стан інтерфейсу (лістинг 3.23).

Лістинг 3.23 – Приклад дата класу який описує стан інтерфейсу

```
data class SavedAddressesScreenState(
    val listAddresses: List<Address> = listOf(),
    val isLoading: Boolean = false,
    val error: LocalizedText = LocalizedText.DynamicString(""),
)
```

Також створюється спеціальний клас, який необхідний щоб чітко визначити всі можливі взаємодії користувача з інтерфейсом экрана в одному місці.

Кожна подія в цьому класі є окремим підтипом, який містить необхідні дані для її обробки. Наприклад, якщо користувач обирає адресу за замовчуванням або видаляє її зі списку, відповідні події SelectDefaultAddress та RemoveAddress містять інформацію про обрану адресу або її ідентифікатор (лістинг 3.24).

Лістинг 3.24 – Клас який визначає всі можливі взаємодії з інтерфейсом

```
sealed class SavedAddressesIntent {
    data class SelectDefaultAddress(val address: Address) :
SavedAddressesIntent()
    data class RemoveAddress(val addressId: String) :
SavedAddressesIntent()
}
```

Щоб забезпечити реакцію на ці події у ViewModel реалізовано метод onEvent(). який приймає подію як параметр і визначає, яка бізнес-логіка має бути виконана за допомогою оператора when (лістинг 3.25). Такий підхід дозволяє чітко розділити рівень представлення та рівень логіки управління даними.

Лістинг 3.25 – Приклад метода що обробляє події

```
fun onIntent(intent: SavedAddressesIntent) {
    when (intent) {
        is SavedAddressesIntent.SelectDefaultAddress -> {
            updateAddresses(intent.address)
        }
        is SavedAddressesIntent.RemoveAddress -> {
            removeAddress(intent.addressId)
        }
    }
}
```

Перейдемо до опису того, як відбувається оновлення стану. Для прикладу розглянемо асинхронний метод getAddresses() (лістинг 3.26). Ця функція виконує запит на отримання списку адрес, використовуючи змінну з екземпляром UseCase класу. Вона викликає оператор invoke(), який ініціює запит до віддаленого API.

В результаті виконання отримуємо клас DataResult, який зберігає стан запиту. Після чого визначаємо за допомогою оператора when, що саме буде виконано та що буде оновлено в класі стану інтерфейсу. Таким чином забезпечується коректна обробка як успішного, так і помилкового результату, що підвищує надійність роботи застосунку та покращує взаємодію з користувачем.

Лістинг 3.26 – Метод getAddresses

```
private suspend fun getAddresses() {
    getAddressListUseCase.invoke().collect { response ->
        when (response) {
            is DataResult.Success -> {
                _state.update { it.copy(listAddresses =
response.data) }
            }
            is DataResult.Error -> {
                _state.update { it.copy(error =
LocalizedText.DynamicString(response.exception.message.orEmpty()
)) }
            }
        }
    }
}
```

Тепер можна перейти до розгляду того, як відбувається створення інтерфейсу користувача та його взаємодія з ViewModel. Отже UI застосунку складається з простих елементів, які надає фреймворк Jetpack Compose (лістинг 3.26).

Окрім простих елементів, таких як Text, Button або Image, у Jetpack Compose можна створювати більш складні компоненти, що об'єднують кілька базових елементів у єдину композицію (рисунок 3.3).

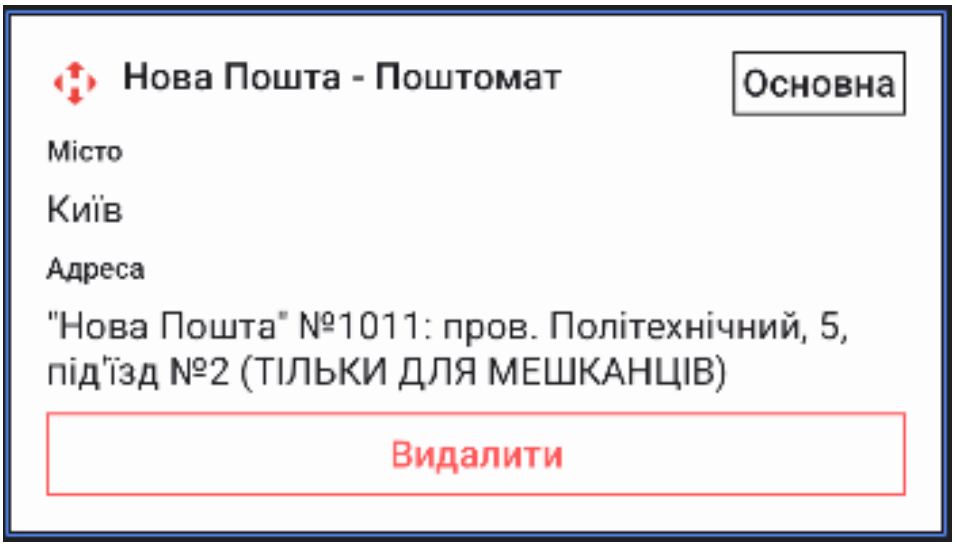


Рисунок 3.3 – Приклад композиції, що представляє компонент, який відображає збережену адресу доставки користувача

Лістинг 3.26 – Елемент Text

```
Text(
    modifier = Modifier
        .width(stepCircleSize + 48.dp),
    text = titleStep,
    fontFamily = robotoFontFamily,
    fontWeight = FontWeight.SemiBold,
    fontSize = 12.sp,
    textAlign = TextAlign.Center,
    overflow = TextOverflow.Visible,
    color = OnSurfaceColor
)
```

Компоненти інтерфейсу можуть відображати як статичну інформацію, яка залишається незмінною, так і динамічні дані, що змінюються з часом. Для того щоб інтерфейс реагував на зміни стану даних, використовується спеціальна змінна, яка слідкує за оновленнями та автоматично сповіщає всі залежні компоненти про зміни (лістинг 3.27).

Лістинг 3.27 – Приклад змінної що отримує стан даних в ViewModel

```
val state by viewModel.state.collectAsState()
```

Для використання даних з класу стану необхідно викликати змінну state та обрати яку саму інформацію необхідно отримати (лістинг 3.28).

Лістинг 3.28 – Приклад отримання даних з класу стану

```
CategoryContent(
    context = context,
    state = state,
    isNetworkConnected = isNetworkConnected,
    navigateToOtherScreen = { screen ->
        navController.navigate(screen)    },
    navigateBack = { navController.navigateUp() }
)
```

Для того щоб оповістити ViewModel про те, що відбулась якась подія, треба викликати публічний метод onEvent() (лістинг 3.29). У цьому прикладі зміна значення поля електронної пошти передається до ViewModel через подію EmailChanged.

Лістинг 3.29 – Приклад отримання даних з класу стану

```
CustomTextInput(
    icon = Icons.Outlined.MailOutline,
    placeholder = R.string.new_email_placeholder,
    text = state.email,
    onValueChange = { text ->
        viewModel.onEvent(ChangeEmailEvent.EmailChanged(text))
    },
    style = CustomTextInputStyle.EMAIL,
    imeAction = ImeAction.Done,
    isError = state.emailError != null,
    errorMessage = state.emailError
)
```

4 ІНСТРУКЦІЯ КОРИСТУВАЧА

Першим екраном, який бачить користувач, є вітальний. На ньому розташовані дві основні кнопки: перша (найбільш помітна) пропонує створити обліковий запис, а під нею розміщено текстове посилання, за допомогою якого можна перейти до екрану входу (рисунок 4.1).

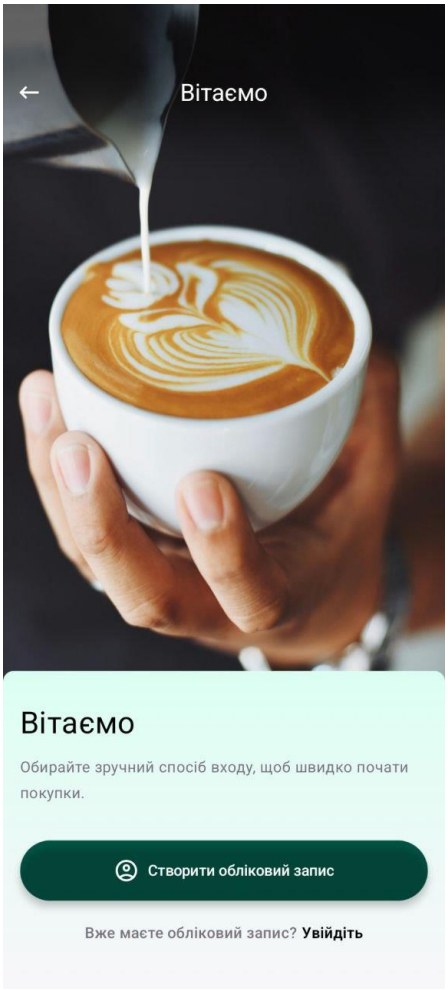


Рисунок 4.1 – Екран для вибору створення або входу в обліковий запис

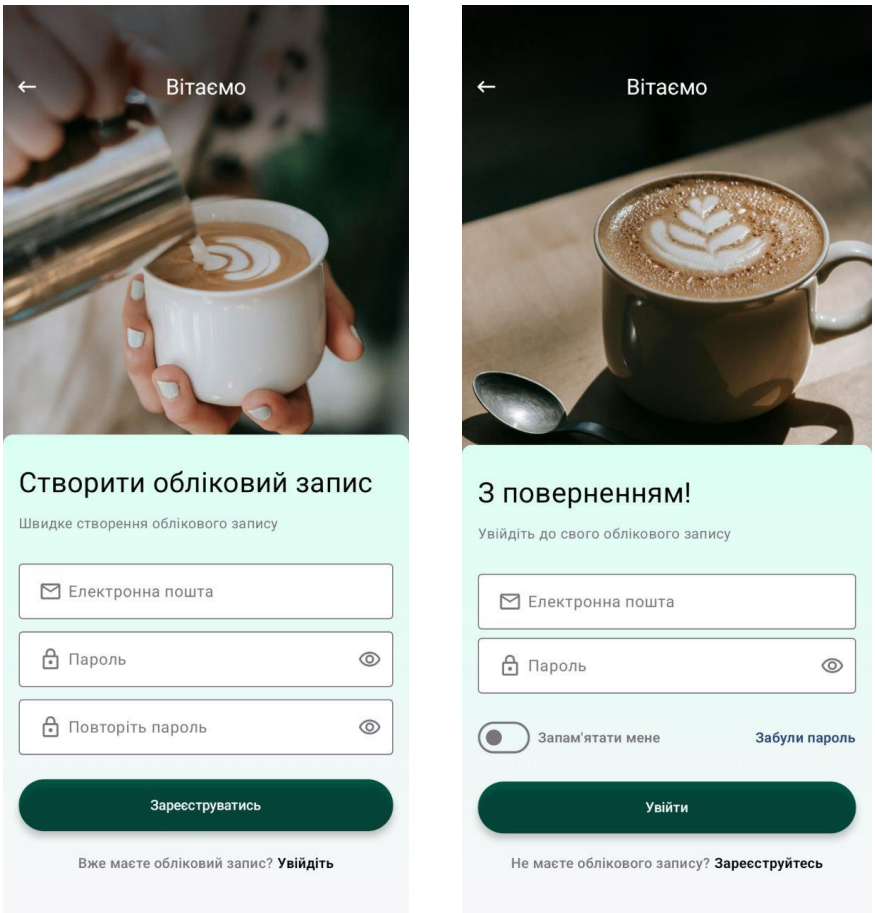
На екрані створення нового облікового запису (рисунок 4.2 а) розміщено три текстові поля. У перше з них користувач вводить електронну пошту, яка буде прив’язана до нового акаунта.

Наступні два поля призначені для введення пароля та його підтвердження. У кожному з полів для введення пароля, праворуч,

розташовано кнопку, що дозволяє перемикати режим відображення символів (показувати або приховувати введений пароль).

Після заповнення всіх полів стає доступною велика кнопка «Зареєструватись», натискання якої надсилає запит на реєстрацію. У разі правильного введення даних користувач автоматично переходить на головний екран застосунку. Якщо ж сталася помилка, наприклад, некоректно введено електронну адресу або паролі не збігаються, під формою з'являється відповідне повідомлення про помилку. Під основною кнопкою розташовано текстове посилання, яке дозволяє швидко перейти до екрана входу.

Екран входу має схожу структуру із сторінкою створення нового акаунта, проте містить деякі відмінності в елементах інтерфейсу (рисунок 4.2 б).



а) б)

Рисунок 4.2 – Інтерфейс мобільного застосунку:
а) екран створення обікового запису; б) екран входу

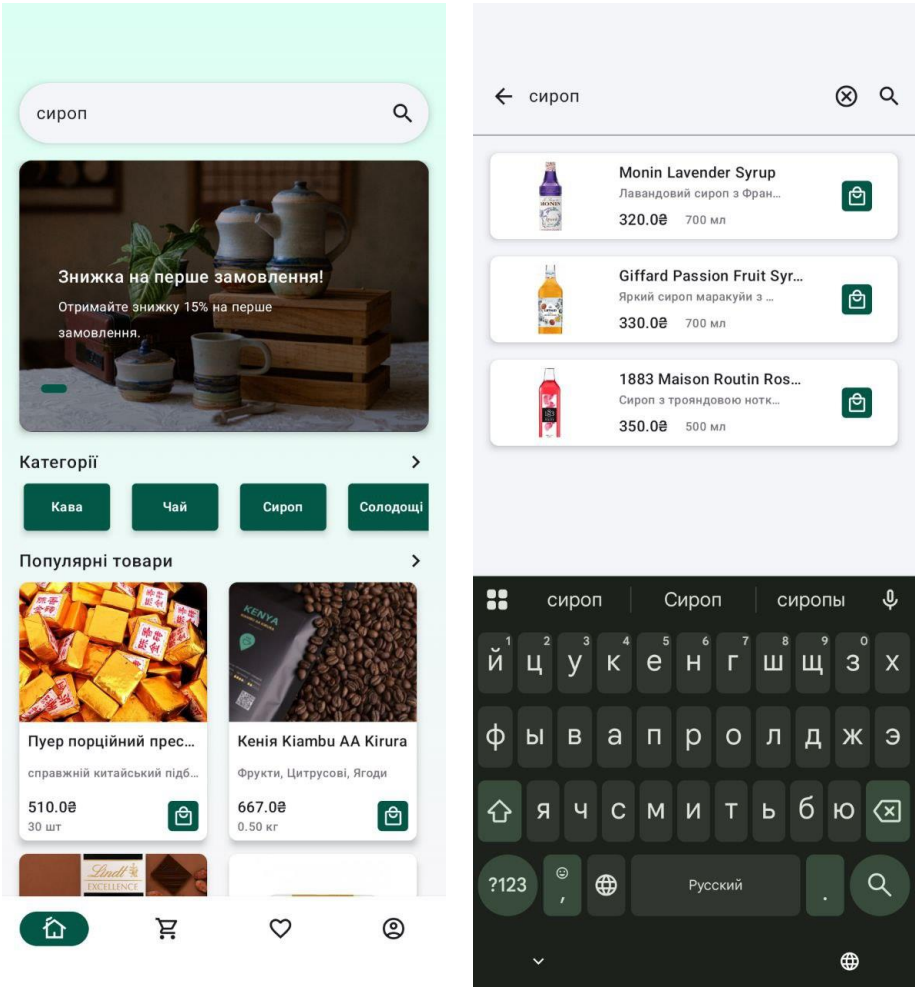
На цьому екрані присутні лише два текстові поля: перше призначене для введення електронної пошти, а друге для пароля. Підтвердження пароля на цьому етапі не вимагається. Як і у формі реєстрації, праворуч від поля введення пароля розташована кнопка, що дозволяє перемикати відображення символів пароля. Нижче форми входу розміщено два елементи на одній горизонтальній лінії, зліва знаходиться перемикач, що дозволяє обрати чи зберігати сесію після закриття застосунку, а справа – текстове посилання, яке веде на екран відновлення пароля.

Далі розміщується велика кнопка «Увійти», при натисканні якої надсилається запит на авторизацію користувача. У разі успішного входу здійснюється перехід на головний екран застосунку. Якщо ж дані введено некоректно або сталася помилка, під кнопкою відображається відповідне повідомлення про помилку. Під формою входу також міститься текстове посилання, за яким користувач може перейти на сторінку реєстрації, якщо ще не має облікового запису.

Головний екран застосунку (рисунок 4.3 а) можна умовно поділити на п’ять секцій. Перша секція розташована у верхній частині екрана та містить поле для пошуку продуктів. Друга секція – це горизонтальна карусель з акційними пропозиціями. Третя секція – також горизонтальна карусель, у якій представлено список категорій для швидкого вибору. Заголовок цієї секції виконано у вигляді текстового посилання, натиснувши на яке користувач переходить на сторінку з розгорнутим списком усіх доступних категорій (рисунок 4.4 а).

Четверта секція є найбільшою і займає більшу частину головного екрана. У ній розташовано список популярних товарів, які користувач найчастіше переглядає. Остання, п’ята секція – це нижнє меню навігації, яке також використовується на багатьох інших сторінках застосунку. У цьому меню розташовані кнопки для переходу між основними розділами. Це меню забезпечує швидкий доступ до основних функцій застосунку, підвищуючи зручність користування.

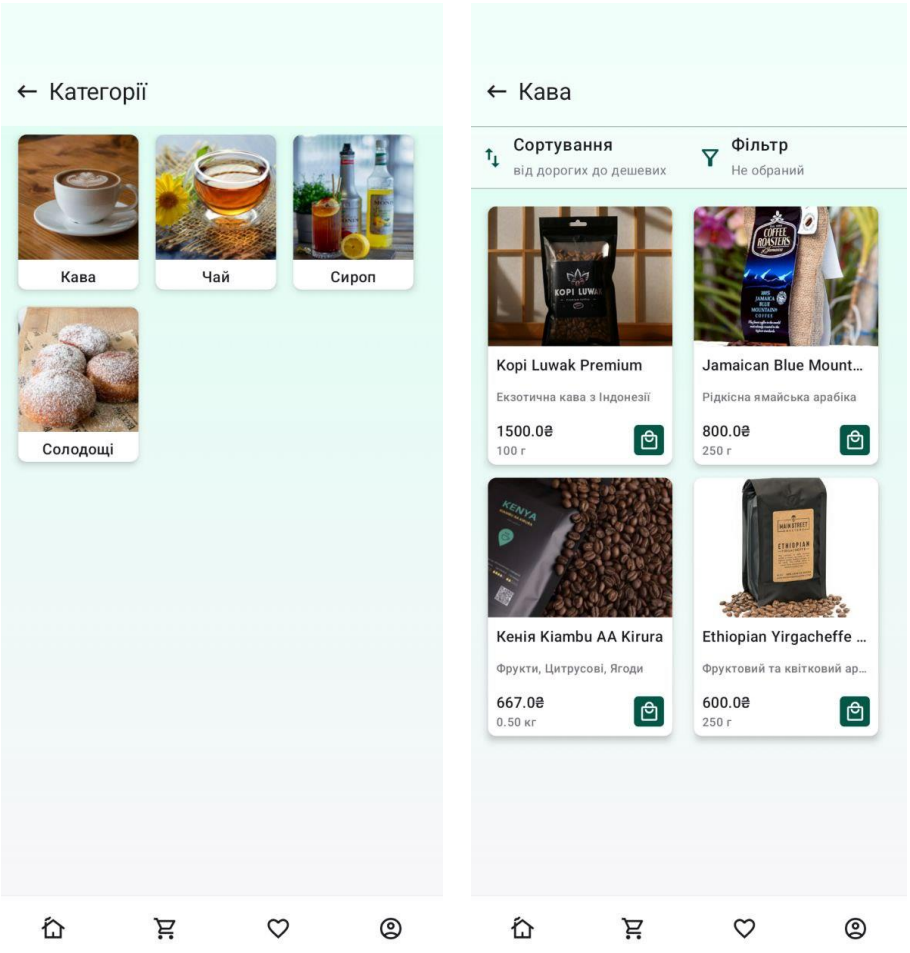
При натисканні на поле пошуку воно розширюється, дозволяючи користувачеві ввести пошуковий запит (рисунок 4.3 б). Якщо товари, що відповідають запиту, будуть знайдено, на екрані з’явиться їх список. Також, для зручності користувача передбачено додаткові елементи управління: кнопка для очищення введеного запиту та кнопка для виходу з режиму пошуку, яка повертає користувача до звичайного вигляду головного екрана.



а) б)

Рисунок 4.3 – Інтерфейс мобільного застосунку:
а) головний екран; б) пошук продуктів

При виборі категорії користувач переходить на екран зі списком товарів, що належать до обраної групи (рисунок 4.4 б). У верхній частині екрана розташовано кнопку для повернення на попередній екран, а поруч – назву вибраної категорії.



а) б)

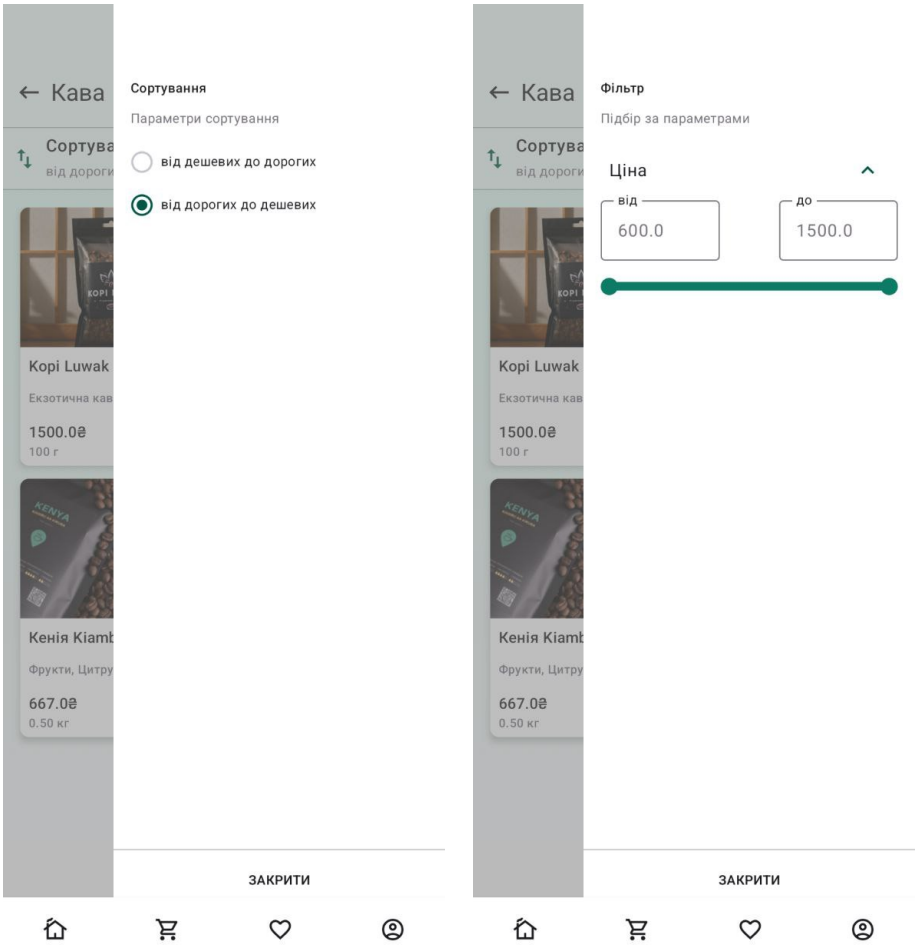
Рисунок 4.4 – Інтерфейс мобільного застосунку:
а) екран «Категорії»; б) список товарів за категорією

Нижче знаходяться два елементи керування:

- перший елемент відображає поточний тип сортування. При натисканні на нього відкривається меню вибору способу сортування товарів (рисунок 4.5 а);

- другий елемент показує чи застосовані до списку товарів додаткові фільтри пошуку. При натисканні на цей елемент відкривається меню для вибору або редагування фільтрів (рисунок 4.5 б).

Отже, ці елементи забезпечують швидкий доступ до налаштувань відображення товарів та допомагають користувачу зручно знаходити необхідні позиції.



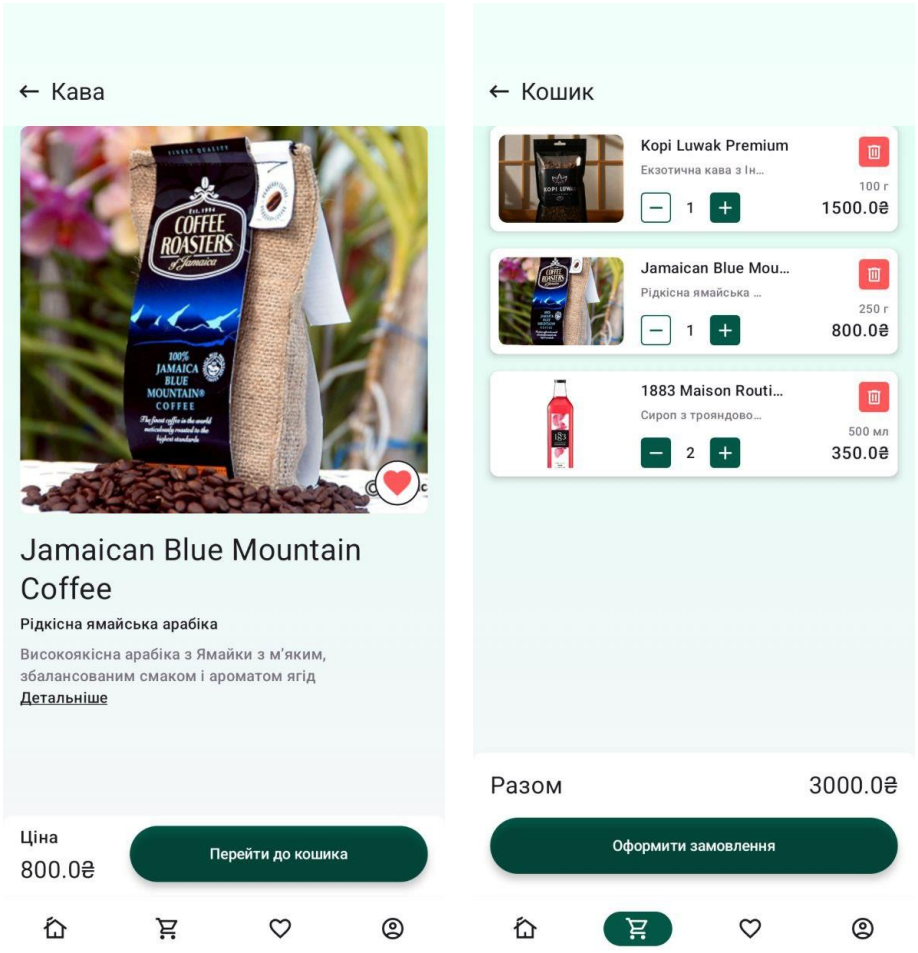
а) б)

Рисунок 4.5 – Інтерфейс мобільного застосунку:
а) меню вибору типу сортування; б) меню зі списком фільтрів

Якщо користувач натискає на картку товару, він переходить на сторінку з детальною інформацією про цей товар (рисунок 4.6 а). На цьому екрані розташоване велике зображення товару, вказано, до якої категорії він належить, відображено його ключові особливості, повний опис, а також ціна. Крім того, на сторінці розміщено дві кнопки:

- кнопка у вигляді серця, яка дозволяє додати товар до списку улюблених або видалити його звідти;

- кнопка з текстом «КУПИТИ», яка додає товар до кошика. Якщо товар вже у кошику, кнопка змінюється на «Перейти до кошика». При її натисканні здійснюється перехід на сторінку з кошиком (рисунок 4.6 б).



а) б)

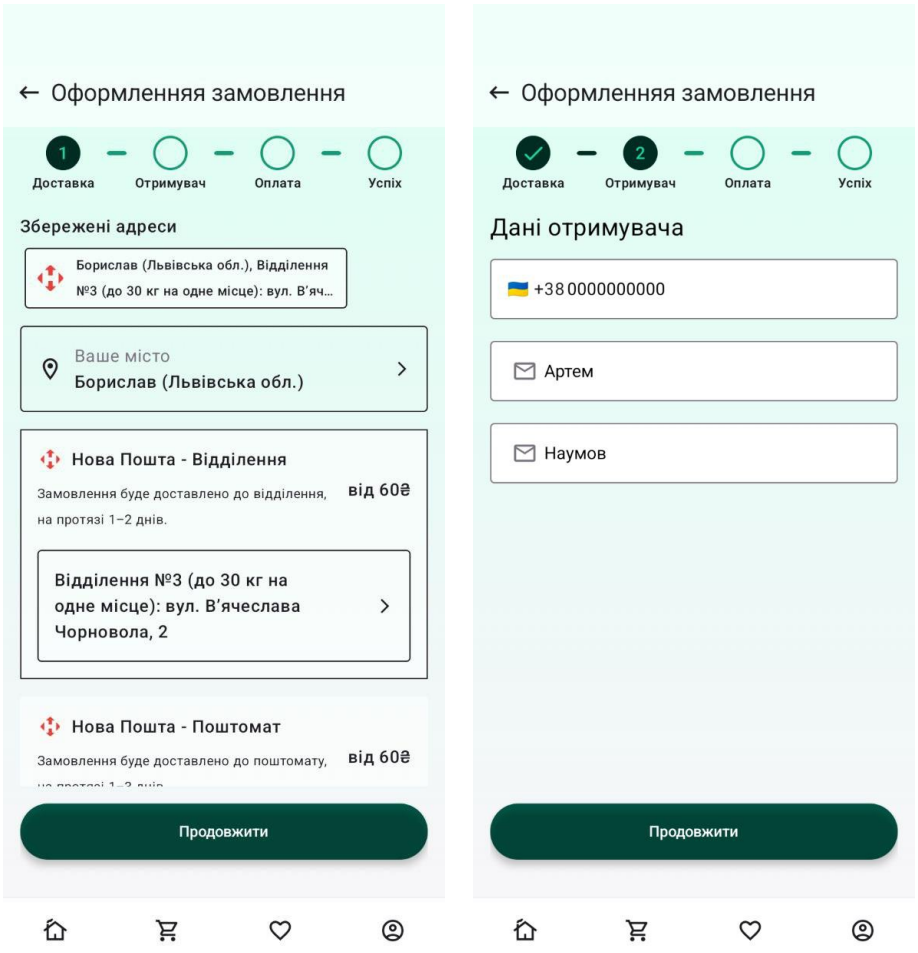
Рисунок 4.6 – Інтерфейс мобільного застосунку:
а) повна інформація про продукт; б) екран «Кошик»

На сторінку з кошиком можна перейти або через екран із повною інформацією про товар, або натисканням кнопки «До кошика», яка з'являється на картці продукту, якщо він вже доданий до переліку запланованих замовлень.

Основну частину екрана займає список покупок, кожен елемент якого містить коротку інформацію, ціну за одиницю товару, кнопки для збільшення або зменшення кількості продукту, а також кнопку для його видалення з кошика. Під списком відображається інформація про загальну суму за заказ, нижчі розташована велика кнопка з текстом «Оформити замовлення», при натисканні на неї відбувається перехід до підтвердження покупки.

Екран «Оформлення замовлення» складається з чотирьох послідовних кроків, які потрібно пройти для завершення оформлення.

Перший етап – вибір адреси доставки (рисунок 4.7 а). У верхній частині розташована горизонтальна карусель із збереженими адресами, що дозволяє швидко й зручно обрати потрібну. Нижче розміщений елемент вибору міста, при натисканні на нього відкривається пошук, де користувач вводить назву міста та обирає його зі списку запропонованих варіантів. Далі потрібно визначити спосіб доставки: на відділення або у поштомат «Нової пошти». Після того як спосіб доставки обрано, відповідний блок розгортається, і з’являється поле для вибору конкретної адреси.



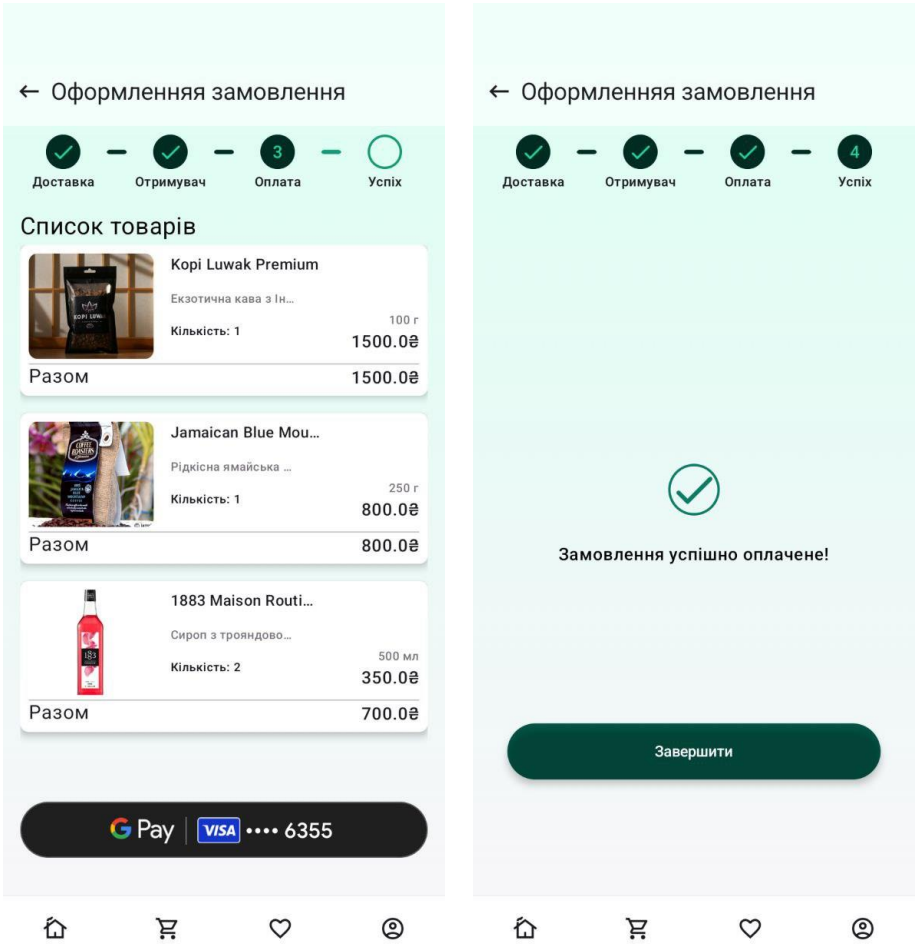
а) б)

Рисунок 4.7 – Інтерфейс мобільного застосунку:

а) вибір адреси для замовлення; б) вибір отримувача для замовлення

Наступний крок призначений для введення даних отримувача. Він містить три текстових поля для вводу імені, прізвища та номера телефону (рисунок 4.7 б). За замовчуванням ці поля заповнюються інформацією з профілю користувача, яку можна при необхідності змінити.

Далі йде крок оплати (рисунок 4.8 а), де відображається список товарів із кошика. Для кожного товару показана коротка інформація, кількість, ціна, а також загальна сума за цей продукт. Натискання на будь-який товар відкриває екран з повною інформацією про нього. Внизу екрану розташована кнопка для оплати через систему Google Pay.



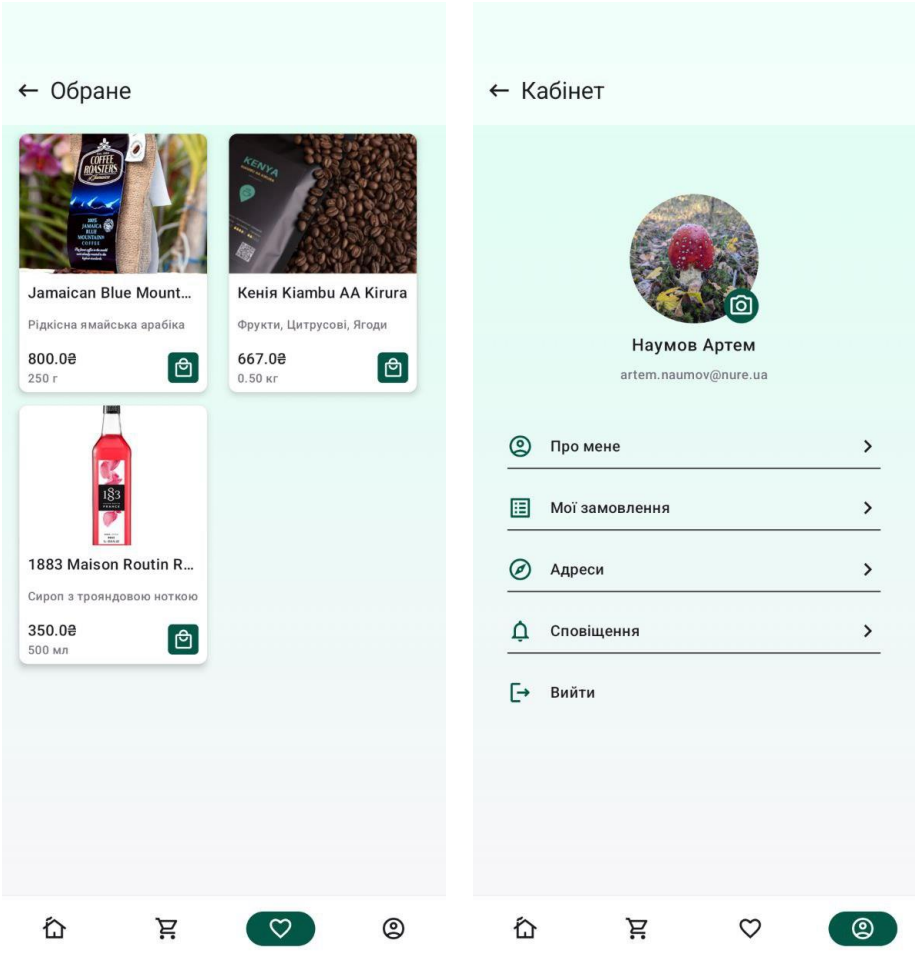
а) б)

Рисунок 4.8 – Інтерфейс мобільного застосунку:

а) оплата замовлення; б) повідомлення про успішну оплату

На останньому кроці користувачу надсилається повідомлення про успішне завершення оплати (рисунок 4.8 б). На екрані є кнопка «Завершити», яка повертає користувача на головний екран.

Екран обраного (рисунок 4.9 а) містить перелік товарів, які користувач додав до списку обраного. Додаткових функціональних елементів на цьому екрані немає. Останній екран, на який можна перейти за допомогою нижнього навігаційного меню – це сторінка кабінету користувача (рисунок 4.9 б). Перший елемент цього екрану це фотографія профілю, яку можна змінити, натиснувши на іконку у вигляді фотоапарату, яка знаходиться поруч. Нижче відображаються інформація про користувача.



а) б)

Рисунок 4.9 – Інтерфейс мобільного застосунку:

а) екран «Обрані»; б) екран «Кабінет»

Після наведеної інформації про користувача розташовано кнопки:

- «Про мене» – при натисканні відбувається перехід на екран редагування інформації профілю;
- «Мої замовлення» – відкриває історію замовлень;
- «Адреси» – при натисканні відбувається перехід на екран зі списком збережених адрес;
- «Сповіщення» – відкриває сторінку з нещодавніми та актуальними повідомлення з пропозиціями і важливою інформацією;
- «Вийти» – служить для завершення сесії користувача, щоб можна було увійти в інший акаунт або створити новий.

Екран «Про мене» (рисунок 4.10) можна розділити на три секції: персональні дані користувача, контактні дані, кнопка «Зберегти зміни».

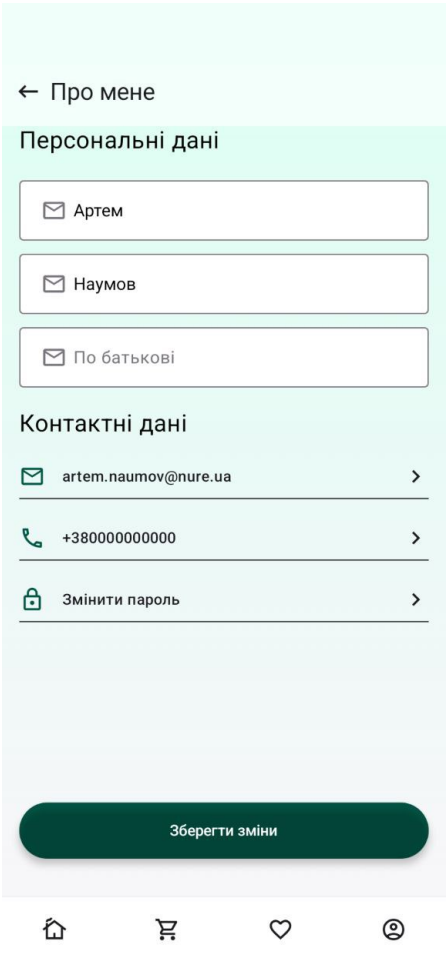
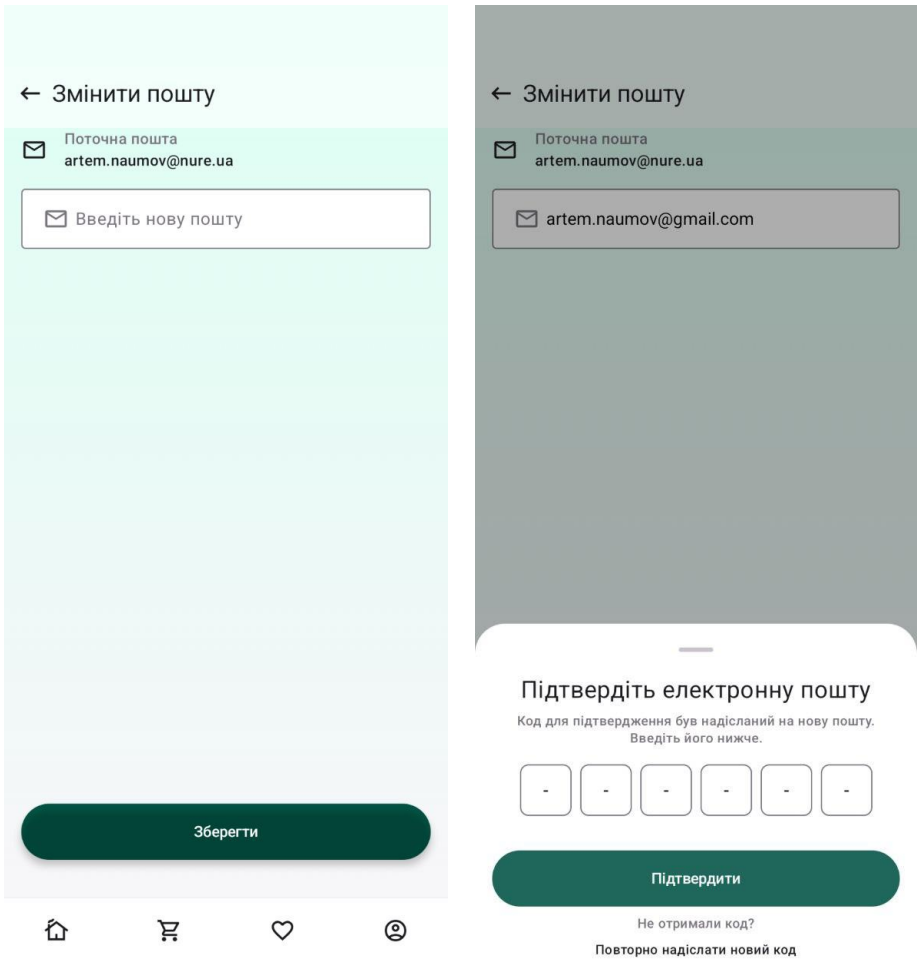


Рисунок 4.10 – Інтерфейс екрану «Про мене»

Перша секція відповідає за персональні дані користувача у ній розташовані три текстові поля для введення ПІБ. Друга секція «Контактні дані» – містить три кнопки, що дають змогу перейти до зміни електронної пошти, номера телефону та пароля. Третя секція містить кнопку «Зберегти зміни». При натисканні на неї відбувається збереження введеної інформації та повернення на попередню сторінку.

На екрані «Змінити пошту» (рисунок 4.11 а) знаходиться текстовий елемент із поточною електронною поштою, текстове поле для введення нової пошти та кнопка «Зберегти».

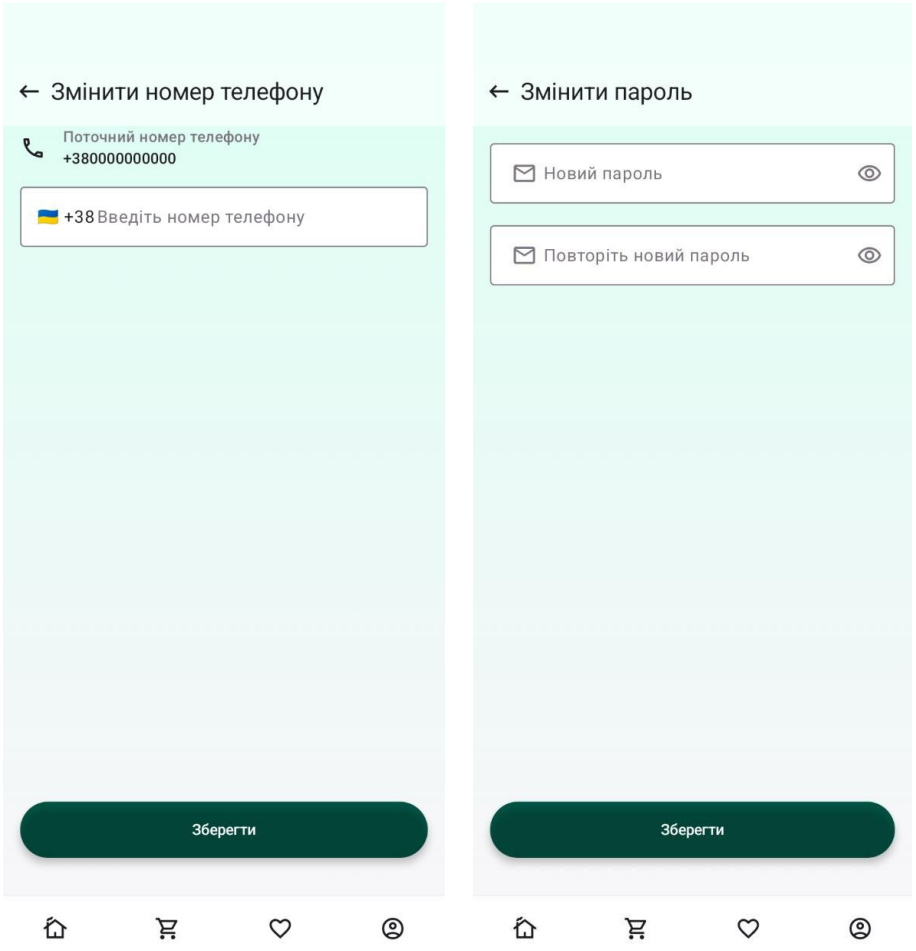


а) б)

Рисунок 4.11 – Інтерфейс мобільного застосунку:
а) екран зміни пошти; б) підтвердження нової пошти

Коли нова електронна пошта введена та натиснута кнопка, знизу з’являється меню з полем для введення коду підтвердження та кнопкою «Підтвердити» (рисунок 4.11 б). Якщо код введено правильно, пошта оновлюється та відбувається повернення на попередній екран. При ситуації, коли код підтвердження не надійшов, користувач може натиснути на текстову кнопку «Повторно надіслати новий код»

Екран для зміни номера телефону (рисунок 4.12 а) має таку ж саму структуру, що й екран зміни електронної пошти. А ось екран зміни пароля (рисунок 4.12 б) вже відрізняється від інших.

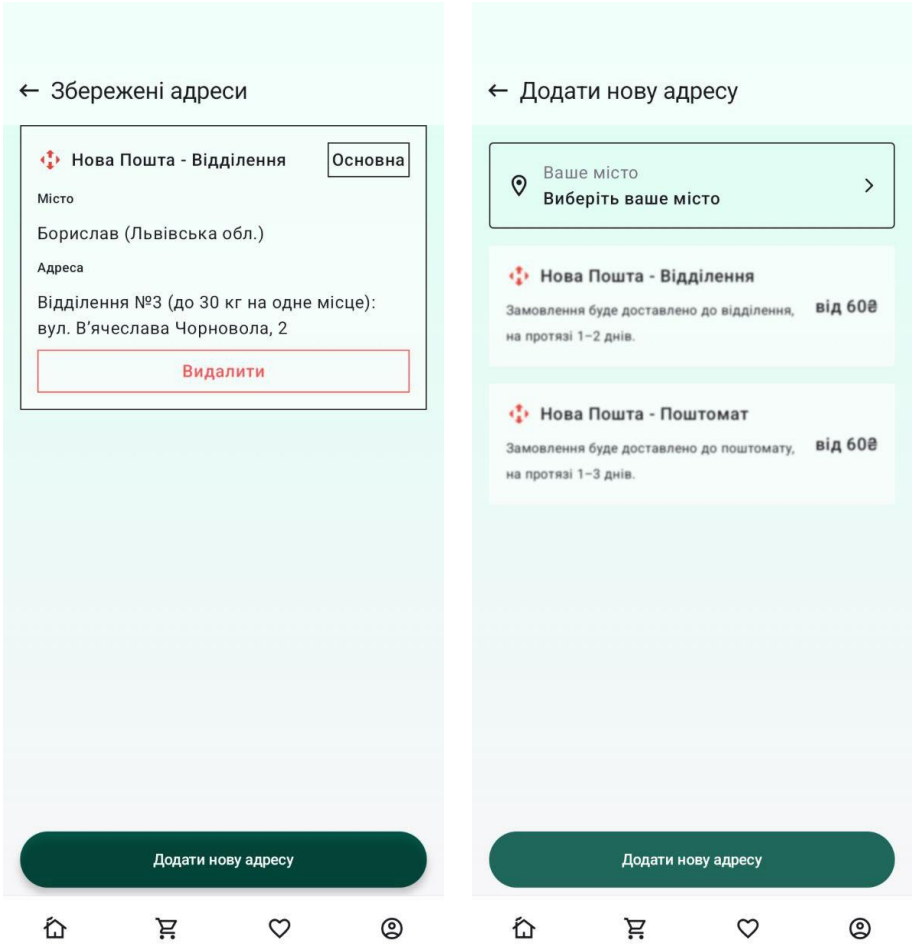


а) б)

Рисунок 4.12 – Інтерфейс мобільного застосунку:
а) екран для зміни номера телефону; б) екран зміни паролю

Він містить два текстових поля: перше для введення нового пароля, а друге – для повторного введення з метою підтвердження. У правій частині обох полів розташовані кнопки для перемикання режиму відображення символів. Після натискання кнопки «Зберегти», якщо всі дані введено правильно, пароль буде успішно змінено. Якщо виникне помилка то вона буде відображена під полями.

Екран зі збереженими адресами (рисунок 4.13 а) містить елементи які відображають інформацію про адреси збережені користувачем. Кожну адресу можна видалити за допомогою кнопки «Видалити», а також встановити її як основну – достатньо натиснути на потрібну адресу, після чого на ній з’явиться позначка «Основна».

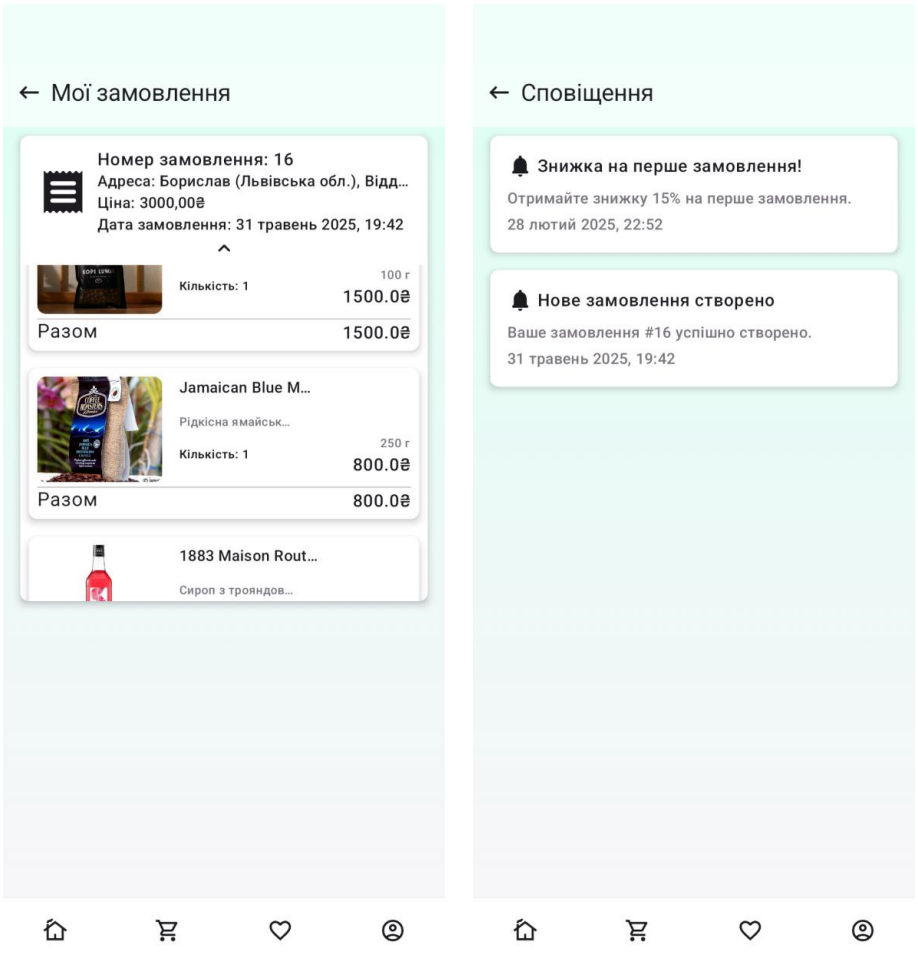


а) б)

Рисунок 4.13 – Інтерфейс мобільного застосунку:
а) екран «Адреси»; б) екран додавання нової адреси

Якщо користувач натисне на кнопку «Додати нову адресу», відкриється екран із формою для додавання нової адреси. Там необхідно вибрати місто та відділення або поштомат «Нової пошти» (рисунок 4.13 б).

Екран «Мої замовлення» відображає список історії замовлень користувача (рисунок 4.14 а). Кожен елемент списку містить коротку інформацію про замовлення: його номер, адресу доставки, загальну суму та дату створення замовлення. Якщо натиснути на елемент, розгорнеться перелік товарів, які входили до цього замовлення. Для кожного товару вказано коротку інформацію: зображення, кількість, ціну за одиницю та загальну суму.



а) б)

Рисунок 4.14 – Інтерфейс мобільного застосунку:
а) екран «Мої замовлення»; б) екран «Сповіщення»

Екран «Сповіщення» (рисунок 4.14 б) відображає зручний та інтуїтивно зрозумілий список, у якому кожен елемент містить детальну інформацію про отримані повідомлення. До цього списку автоматично додаються нові сповіщення про акційні пропозиції, створені замовлення, а також оновлення статусу вже існуючих замовлень.

Кожне сповіщення включає назву, короткий опис та точну дату й час його створення, що допомагає користувачу швидко зрозуміти, про що саме сповіщення. Цей екран забезпечує оперативне інформування користувача про важливі події та оновлення, допомагаючи не пропустити актуальні акції, новини або нагадування.

ВИСНОВКИ

В результаті виконання роботи було проведено детальний аналіз предметної області, що сприяло розумінню особливостей електронної комерції та способів її реалізації у вигляді мобільного застосунку інтернет-магазину. Також було проведено аналіз існуючих на даний час рішень. В ході аналізу було досліджено вебсайти та мобільні застосунки, які займаються онлайн-торгівлею, що дозволило визначити сучасні підходи до дизайну інтерфейсу користувача, способів побудови навігації та оформлення сторінок товарів мобільного застосунку.

На основі аналізу предметної області та існуючих рішень були визначені основні цілі та задачі кваліфікаційної роботи. Для їх реалізації було обрано технології та архітектурні підходи, які найкраще відповідають вимогам до створення мобільного застосунку інтернет-магазину. Оскільки програма розроблялась під нативну платформу Android, мовою програмування було обрано Kotlin. Для створення інтерфейсу застосунку був використаний фреймворк Jetpack Compose, який дозволяє гнучко будувати адаптивні інтерфейси з урахуванням вимог сучасного дизайну.

У якості серверної частини програми було обрано Supabase – сучасну сервісну платформу з відкритим кодом. Вона надала готовий інструментарій для взаємодії з базою даних PostgreSQL, а також хмарне сховище для зберігання зображень та інших файлів. Supabase також забезпечила просту інтеграцію аутентифікації користувачів у застосунку.

Під час проєктування архітектури застосунку були використані принципи Clean Architecture та патерн MVVM, що забезпечило чіткий розподіл відповідальності між шарами програми, спростило тестування та підтримку.

Також під час написання застосунку активно застосовувались сучасні принципи SOLID, DRY та KISS, які дозволили досягти високого рівня структурованості коду та спростити подальше масштабування проєкту.

В результаті виконання роботи було створено функціональний мобільний застосунок інтернет-магазину, що дозволяє переглядати каталог товарів, детальну інформацію про кожен продукт, реєструватися, входити до акаунту, додавати товари до кошика та оформляти замовлення.

Подальший розвиток проєкту передбачає реалізацію низки функціональних покращень, а також підвищення загальної стабільності роботи застосунку. Зокрема, планується:

- додавання розширеної системи фільтрів для зручнішого пошуку товарів, зокрема за типами товарів, країнами походження та методом обсмажування;

- реалізація промокодів та системи знижок, які дозволять користувачам отримувати вигідні пропозиції, а також підтримають маркетингові кампанії магазину;

- поліпшення роботи системи сповіщень, зокрема реалізація push-повідомлень про новинки, знижки, зміну статусу замовлення;

- інтеграція алгоритмів персоналізованих рекомендацій, які допоможуть користувачам швидше знаходити цікаві товари на основі їхніх попередніх дій;

- розширення асортименту продукції з урахуванням побажань користувачів та аналізу продажів;

- додавання нових способів оплати.

Отже, задачі кваліфікаційної роботи були успішно виконані, а також розроблений мобільний застосунок має потенціал для подальшого розвитку та вдосконалення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Tian Y., Stewart C. History of E-Commerce. Electronic Commerce. 2008. P. 1-8. doi.org/10.4018/978-1-59904-943-4.ch001.

2. Наумов А.В., Севостьянова О.М., Філімончук Т.В. Мобільний застосунок для інтернет-магазину. Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління: тези доповідей п’ятнадцятої міжнародної науково-технічної конференції. Т.2: секція 2. Баку: ІСУ АР; Харків: НТУ «ХПІ»;. Харків: ХНУРЕ; Харків: НАУ «ХАІ»; Жиліна: УмЖ. 2025. С. 31.

3. Santos V.F., Sabino L.R., Morais G.M., Goncalves C.A. E-Commerce: A Short History Follow-up on Possible Trends // International Journal of Business Administration. 2017. Vol. 8, №7. P. 130. doi.org/10.5430/ijba.v8n7p130.

4. Інтернет-магазин кави «Kulturrra». URL: https://kulturrra.coffee.

5. Інтернет-магазин кави, чаю та кавомашин «Coffeeok». URL: https://coffeeok.com.ua/ua.

6. LLC S.G. Мій Чай – Додатки в Google Play. Android Apps on Google Play. URL: https://play.google.com/store/apps/details?id=salesbox.me.miychay&hl=uk.

7. Kotlin language specification. Kotlin Programming Language. URL: https://kotlinlang.org/spec/introduction.html.

8. Mobile Operating System Market Share Worldwide | Statcounter Global Stats. StatCounter Global Stats. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide.

9. Get started with Jetpack Compose | Android Developers. Android Developers. URL: https://developer.android.com/develop/ui/compose/documentation.

10. Hilt. Dagger. URL: https://dagger.dev/hilt.

11. Ferrari L., Pirozzi E. Learn PostgreSQL – Second Edition: Use, manage and build secure and scalable databases with PostgreSQL. 2nd ed. Packt

Publishing, 2023. 744 p.

12. Meet Android Studio | Android Developers. Android Developers. URL: https://developer.android.com/studio/intro.

13. Run apps on the Android Emulator | Android Studio | Android Developers. Android Developers. URL: https://developer.android.com/studio/run/emulator.

14. Основи Git. URL: https://git-scm.com/book/uk/v2/Вступ-Основи-Git.

15. Martin R.C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall PTR, 2018. 420 p.