СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки

Кафедра інформаційних технологій та програмування

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної випускної роботи

освітній ступінь бакалавр
спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)

спеціалізація „Інженерія програмного забезпечення”
(назва спеціалізації)

на тему „Мобільний додаток для кафе, орієнтованого на клієнта”

Виконав: студент групи ІПЗ-20д	_____ (підпис)	М.С. Тесля (ініціали і прізвище)
Керівник	_____ (підпис)	Д.М. Марченко (ініціали і прізвище)
Завідувач кафедри	_____ (підпис)	О.І. Захожай (ініціали і прізвище)

Рецензент_____

Київ – 2024

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
Кафедра інформаційних технологій та програмування

Освітній ступінь бакалавр
спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)
спеціалізація „Інженерія програмного забезпечення”
(назва спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Захожай О.І.
“___” _____ 2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ ВИПУСКНУ РОБОТУ СТУДЕНТУ

Тесля Михайло Сергійович
(прізвище, ім’я, по батькові)

1. Тема роботи: Мобільний додаток для кафе, орієнтованого на клієнта

Керівник роботи Марченко Дмитро Миколайович, професор, д.т.н.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
Затверджений наказом університету від “___”_____20__року №____

2. Строк подання студентом роботи _____
3. Вихідні дані до роботи Об’єктом даної роботи є процес створення мобільного додатку для кафе, орієнтованого на клієнта

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): Вступ. Аналітичний огляд, з висвітленням наступних питань: поширення мобільних операційних систем у світі, важливість для кафе мати мобільний додаток, порівняння мов програмування для розробки додатку. Основна частина, в якій висвітлити інформаційну та функціональну модель додатку, продемонструвати графічний вид додатку, етап створення додатку та тестування. Висновки. Перелік використаних джерел.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслеників) Додаток А. Рівень 1 функціональної діаграми IDEF0, Додаток B. Рівень 2 функціональної діаграми IDEF0, декомпозиція блоку A-1, Додаток C. Рівень 2 функціональної діаграми IDEF0, декомпозиція блоку A-3, Додаток D. Діаграма IDEF3 для додатка «Kind cafe»

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання кваліфікаційної випускної роботи	Строк виконання етапів	Примітка
1	Одержання завдання на виконання роботи	30.03.24	
2	Укладання і погодження з керівником плану і етапів виконання роботи	06.04.24	
3	Узагальнення даних літературних джерел, укладання розділу «Аналіз предметної галузі»	13.04.24	
4	Аналіз шляхів виконання завдання. Вибір і погодження з керівником оптимального шляху	20.04.24	
5	Укладання та тестування програмного продукту	27.04.24	
6	Укладання, оформлення та погодження пояснювальної записки з керівником	04.05.24	
7	Здача готової пояснювальної записки на кафедру	25.05.24	
8	Укладання доповіді і презентації	30.05.24	

Студент _____ М.С. Тесля
підпис (ініціали і прізвище)

Керівник роботи _____ Д.М. Марченко
підпис (ініціали і прізвище)

ЛИСТ ПОГОДЖЕННЯ І ОЦІНЮВАННЯ

дипломної роботи студента гр. ІПЗ-20д Тесля М.С.

Науковий керівник

Професор, д.т.н. _____ Марченко Д.М.

Оцінка наукового керівника: _____

Рецензент _____

ПІБ, місто роботи, посада

Оцінка рецензента: _____

Кінцева оцінка за результатами захисту:

_____

Голова ЕК
Професор кафедри ІТП
д.т.н. _____ Меняйленко О.С.

підпис

ЗМІСТ

ВСТУП

Актуальність. У сучасному світі неможливо уявити своє життя без телефону. Майже у кожної людини є телефон – невеликий пристрій завдяки якому втілена можливість комунікації на великих відстанях. Мобільні технології стали значно розвиненішими за майже два десятиріччя і зараз замість звичайного телефону більшість людей використовують смартфон – це пристрій що поєднує функціонал і потужність мобільного телефону та персонального комп'ютера. Це значить, що завдяки смартфону можна передивлятися веб-сторінки, виконувати банківські операції, навчатися, відстежувати певні аспекти свого здоров'я, спілкуватися з іншими людьми, та звісно оформлювати закази у магазинах та кафе [4].

З ростом популярності смартфонів, збільшується популярність і програм для них – мобільних додатків. Зараз більшості людей комфортніше виконувати багато своїх справ та операцій через мобільні додатки ніж через браузер або фізичний візит до закладу [19].

На даний час, більшість підприємств, які прагнуть збільшити кількість своїх клієнтів, та надати їм більш комфортні умови для взаємодії, розроблюють власний мобільний додаток. Це дозволяє їм більш наблизитися до своїх клієнтів, та надати їм більше можливостей для комунікації.

Вибір між веб-сторінкою та мобільним додатком у користь останнього може робитися через наступні фактори:

- швидкість – мобільні додатки швидше ніж веб-сторінки;

- автономність – мобільні додатки можуть зберігати останні актуальні дані та надають можливість використовувати додаток згідно цих даних;

- кращий вигляд та зрозуміліший інтерфейс - через те, що native мобільний додаток розроблюється для певної системи окремо, то підхід до побудови також буде виконаний із дотриманням вимог даної ОС та розроблений саме для використання на цій ОС;

- бренд – навіть якщо користувач не використовує певний час даний додаток, сама наявність його перед очами в момент користування смартфоном, все одно нагадує про бренди конкретних компаній або організацій за допомогою їх логотипів або кольорів у вигляді значка додатку у пристрої.

Таким чином наявність мобільного додатку для організацій, відкриває їм більш великі перспективи для ведення бізнесу, а користувачу надає більш гнучкі та комфортніші можливості для взаємодії із організацією.

Об’єкт дослідження: процес створення мобільного додатку на ОС Android.

Предмет дослідження: додаток для кафе на ОС Android, орієнтований на клієнта.

Мета дослідження: створення мобільного додатка для кафе на ОС Android, орієнтованого на клієнта.

Задачі дослідження:

1. Аналіз географічного розповсюдження найпоширеніших операційних систем для смартфонів;

2. Розгляд переваг наявності мобільного додатка для кафе;

3. Аналіз переваг між різними мовами програмування, сучасними засобами та інструментами, що можуть бути застосованими для розробки мобільного додатка;

4. Розробка дизайну додатку та його реалізація;

5. Створення комплексу об’єктів і функцій для можливості взаємодії між структурними елементами додатку і сервером.

1. АНАЛІТИЧНИЙ ОГЛЯД

1.1. Розповсюдження Android по світу та порівняння його з іншими ОС

Смартфон – це мобільний пристрій, що має потужну продуктивність та багатий функціонал. Даний пристрій поєднує у собі компактність мобільного телефону та розширені обчислювальні здатності, близько як ПК [17]. На даний час, поняття мобільного телефону і смартфону, у багатьох людей стало майже одним і тим самим, що каже про популярність даного пристрою.

Вже на 2019 рік, згідно зі статистикою Recearch.com [12], смартфони мали більш високі показники за долею веб-трафіку порівняно з ноутбуками та комп’ютерами.

Все більше користувачів звертаються до своїх мобільних пристроїв замість настільних комп'ютерів для таких завдань, як перегляд соціальних мереж, створення електронних листів, читання новин та покупок в Інтернеті.

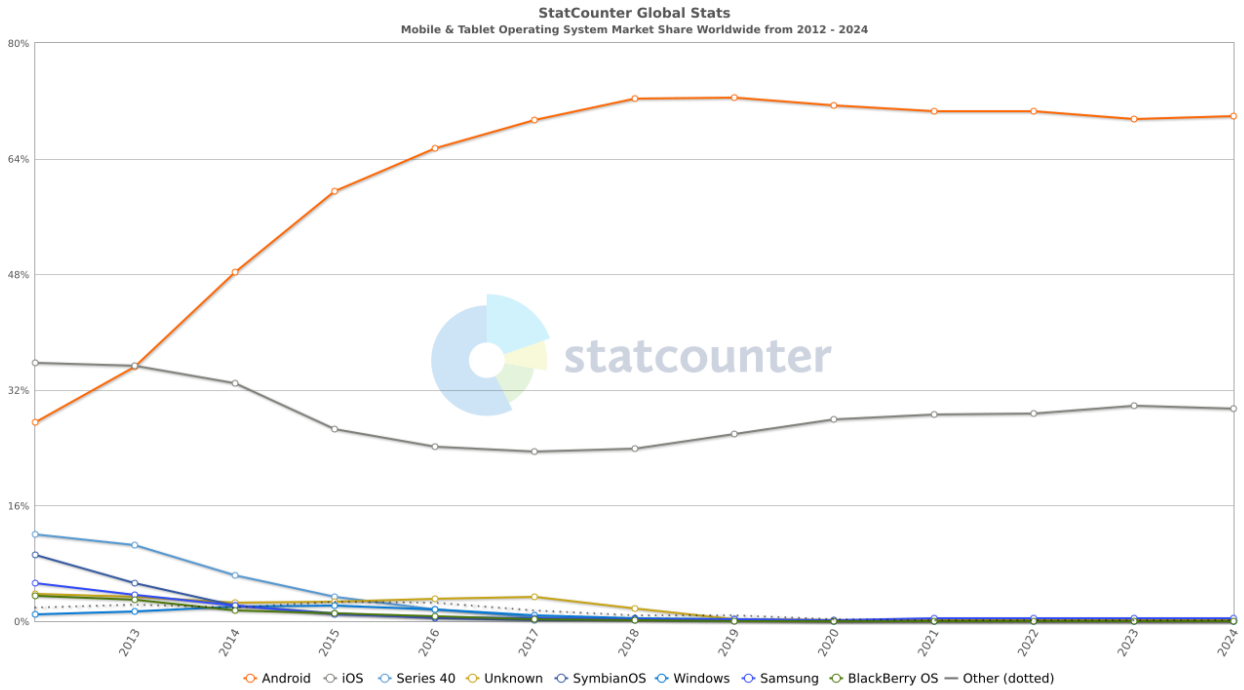


Рисунок 1.1.1 - Частка світового ринку операційних систем для мобільних пристроїв і планшетів

Смартфон став настільки популярний, що навряд чи минає день, щоб користувачі не використовували дані пристрої. Дослідження, проведене Dscout (Вінник, Північна Дакота), показало, що середньостатистична людина торкається свого смартфону не менше 2617 разів на день, доводячи, що мобільні пристрої стали ефективним інструментом, що дозволяє залишатися на зв'язку з іншими людьми і забезпечувати користувачам почуття безпеки [11].

Своє функціонування смартфон здійснює на основі повноцінної операційної системи. На даний момент, переважна кількість пристроїв працює на ОС Android та iOS.

Операційна система Android пройшла вже довгий шлях з листопада 2007 року по сьогоднішній час і зайняла тверду позицію на ринку мобільних пристроїв.

На ринку смартфонів, на зараз знаходяться два лідера, це iOS від Apple (сіра лінія на рис. 1.1.1), та Android від Google (червона лінія на рис. 1.1.1). Зараз, на травень 2024 року, їх розділення за поширенням та продажами (дані за весь світ) йде як: Android –70,87%, та iOS – 28.39%.

Таке розподілення визвано неоднорядною популярністю ОС у світі, а також віком користувачів та їх фінансовими можливостями. Так, вище наведені глобальні дані, але якщо взяти дані виключно, наприклад у США, то там ситуація протилежна: Android: 39%, і iOS: 60.58% (рис. 1.1.2). На даний момент показники України майже сходяться з глобальними даними (рис. 1.1.3), а саме, Android: 71.43%, iOS: 28.14%. Окрім цього, дані щодо Європи також майже сходяться з даними по Україні.

Основна причина через яку Android домінує на світовому ринку - це співвідношення вартість/якість. У багатьох країнах порівняно з США, люди мають набагато менший наявний дохід. Тому першим фактором при виборі смартфону є вартість. На цій арені Apple просто не може конкурувати з Google.

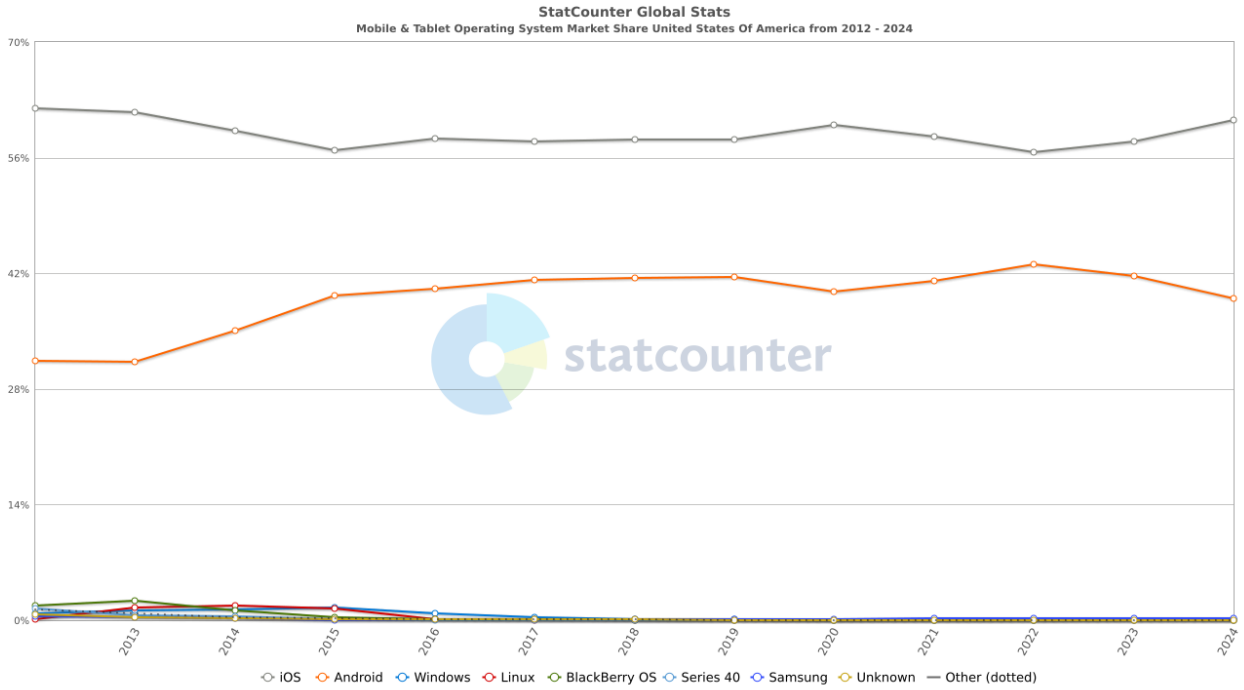


Рисунок 1.1.2 - Частка ринку мобільних і планшетних операційних систем у США

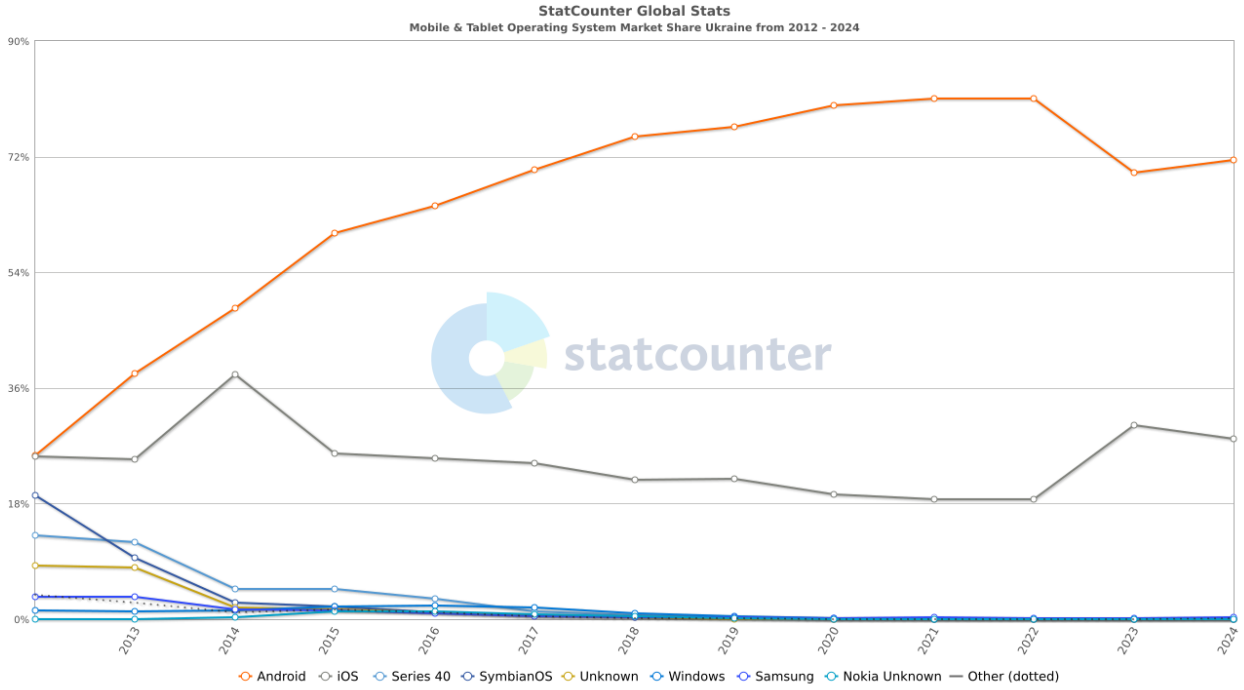


Рисунок 1.1.3 - Частка ринку мобільних і планшетних операційних систем у Україні

Таким чином можна сказати, що смартфон став невід'ємною частиною життя більшості людей. При розробці додатків для смартфонів, потрібно враховувати фактори розповсюдження основних ОС. Наприклад, для Європи і України найбільш поширеною ОС є Android. Це не значить що розробка під

iOS не важлива, а тільки те, що даним додатком буде користуватися більша частка споживачів.

1.2. Важливість для кафе мати додаток

Керування кафе, це складний процес постійної взаємодії із людьми.

Кожне кафе створене для того, щоб запропонувати клієнтам максимальне задоволення та зручність. Важливо справити гарне враження, щоб клієнт прийшов до закладу знову. Проте, нажаль багато людей – клієнтів закладу, скаржаться на недостатньо ефективне обслуговування і недостатню увага та швидкість роботи від працівників.

Зрозуміло, що зробити все ідеально, майже неможливо, особливо в умовах обмеженого фінансування, що актуально для початкового рівня кафе або кафе середнього рівня.

Розробка мобільного додатку для кафе, це сучасне рішення, яке дозволяє даним закладам йти у ногу із часом та вирішує такі проблеми:

- полегшення перегляду меню - дозволяє клієнтам вибрати продукт одним натисканням кнопки, переглядати та обирати страву у вільний час та у комфортній обстановці;

- полегшення формування замовлення – перекладення обов’язків на прийом замовлення на додаток, дозволяє клієнту більш детально переглянути своє замовлення або обрати додаткові опції. Це в свою чергу підвищує спокій клієнту та почуття фінансової безпеки;

- підтримка ланки людей з поганим зором – дані люди, у силу своїх природних особливостей, відчувають себе не дуже зручно коли їм потрібно робити заказ, а деякі ціни або товари вони не в змозі чітко побачити. Це накладає негативний відбиток на відвідування кафе. Проте, використовуючи додаток, дана людина може обійти неприємну для себе частину і продовжити насолоджуватися відпочинком [5];

- зменшення розміру черги без втрати по кількості клієнтів – оформлення замовлення через додаток дозволяє персоналу лише приймати оплату і не тратити зайвого часу на прийом та оформлення самого замовлення;

- покращення якості обслуговування – перекладення деяких обов'язків по роботі з клієнтами на додаток, збільшує час який персонал може приділити клієнту без втрати уваги до інших клієнтів. Окрім цього, деякі люди отримають більше задоволення при взаємодії з машиною аніж людиною;

- присутність у Інтернеті – додаток можна рекламувати у Інтернеті, що доказує рівень технологічного прогресу, який використовує дане кафе, і таким чином можна зробити додатковий висновок про більш якісне обслуговування яке користувач-клієнт може отримати;

- підвищення рівня впізнаваності бренду кафе – знаходження у Інтернеті та зокрема у PlayMarket дозволить показувати даний додаток серед рекомендацій і таким чином це буде відкладатися у пам'яті. Тепер, коли клієнт побачить кафе, це дасть додатковий стимул до створення замовлення саме там [14].

Високоорганізований мобільний додаток дозволяє створити вражаючу присутність в Інтернеті, щоб залучити велику кількість людей за короткий проміжок часу.

Найбільшою перевагою використання мобільного додатка для замовлення страв у кафе є те, що він зводить майже до нуля ймовірність ручних помилок, що у свою чергу підвищує задоволеність клієнтів.

Використання мобільних додатків для бізнесу різних категорій та розмірів, це розповсюджена практика серед країн Європи та США. Проте, в Україні даний підхід отримав широке розповсюдження здебільше серед великих міст або крупних компаній. У невеликих містах або маленьких закладах частіше не використовують сучасні технології у своєму бізнесі, що доволі часто приводить до невдоволення клієнтів які вже звикли до комфорту використання даних технологій, або навіть не знають про такі можливості. Також помітимо, що впровадження подібних додатків у бізнес, окрім явних

економічних та організаційних переваг, також допомагає інтеграції України у світову спільноту за рахунок зміцнення економіки та впровадження сучасних технологій у повсякденне життя громадян та сприяє цифровізації суспільства.

1.3. Порівняння мови Kotlin з мовою Java та React Native

Для розробки мобільного додатку було проведено аналіз кількох мов програмування, а саме:

- Kotlin;
- Java;
- framework ReactNative.

Почнемо з Kotlin. Kotlin – це мова програмування, розроблена компанією JetBrains. З'явилася вона у 2011 році, а у 2017 року мова Kotlin отримала офіційну підтримку для розробки Android-додатків. Як і Java, C і C++, Kotlin є типізованою мовою. Вона підтримує як об'єктно-орієнтоване, так і процедурне програмування [15].

Основні переваги мови Kotlin:

- короткий код – код написаний на Kotlin стислий і більш виразніший порівняно з багатослівністю Java. Це у свою чергу веде до чистішого і зрозумілішого коду і відповідно до цього означає швидший час розробки то поліпшення пошуку помилок;
- код Kotlin компілюється безпосередньо в байт-код, забезпечуючи роботу програм так само ефективно, як і їхні аналоги написані на Java;
- NullPointerException безпека – у мові реалізований захист від null exception, що потребує більш уважну та детальну перевірку типів даних. Така реалізація запобігає поширенню помилок пов’язаних із використанням null;
- підтримка функцій розширення – дані функції сприяють повторному використанню коду, та зменшують шаблонний код. Тобто це можливість розширити клас завдяки додаванню нового функціоналу не потребуючого успадкування або зміни цього класу [16];

- можливість інтеграції з Java - Kotlin легко інтегрується з існуючими кодовими базами Java, тобто існує можливість однаково легко використовувати код із інших проектів написаних на Java, у проекті з Kotlin. Ціль цієї можливості у забезпечені плавного процесу міграції для застарілих проектів або полегшуючи використання встановлених бібліотек Java у проекті Kotlin;

- офіційна підтримка від Google – Google офіційно визнала дану мову як пріоритетну для розробки Android додатків, та надає розширену документацію, інструменти та ресурси;

- підтримка coroutines – coroutines це особливість мови що надає можливість виконувати асинхронні операції, та допомагає у керуванні довгостроковими завданнями. Тобто завдяки цьому реалізується можливість призупиняти і продовжувати виконання блоку коду. Завдяки тому, що даний підхід дозволяє писати асинхронний код який виглядає майже як і синхронний, то це полегшує його читання та розуміння [2].

Наступна мова це Java. Java — це об’єктно-орієнтована мова програмування високого рівня, розроблена компанією Sun Microsystems (нині належить Oracle) і випущена в 1995 році.

Дана мова розроблена як платформа-незалежна, це означає, що програми Java можуть працювати на будь-якому пристрої що підтримує (має встановлену) віртуальну машину Java (JVM), незалежно від основного апаратного забезпечення чи операційної системи.

Основні переваги мови Java:

- незалежність від платформи;

- велика кількість матеріалів для знаходження необхідної інформації в процесі розробки;

- сильний рівень безпеки - Java містить кілька функцій безпеки, які допомагають захистити програми від зловмисних атак. Ці функції включають: безпеку типу, ізольоване програмне середовище та використання менеджера безпеки для застосування політик контролю доступу;

- велика кількість фахівців, що працюють з цією мовою програмування;

- керування пам’яттю - вбудований збирач сміття, автоматично керує виділенням і звільненням пам’яті, допомагаючи запобігти витоку пам’яті та іншим проблемам пов’язаним з пам’яттю;

- має більш велику швидкість чистих збірок (приблизно на 12-15%) порівняно з Kotlin;

- підтримка багатопоточності - Java має вбудовану підтримку для створення та керування кількома потоками виконання. Відповідно для Android додатку це особливо корисне, тому що багато функцій додатку вимагають одночасної обробки, наприклад, обробки фонових завдань, мережевих операцій або інших ресурсномістких дій;

- широка екосистема бібліотек, фреймворків і ресурсів [2].

Останній варіант, це React Native. Якщо Java і Kotlin, це повноцінні мови програмування, то React Native це framework з відкритим кодом для створення мобільних додатків за допомогою JavaScript і React, розроблений Facebook (Meta). React Native, це гарне рішення якщо потрібно створювати один і той же додаток для різних ОС (iOS та Android) використовуючи один і той же код [1].

Основні переваги React Native:

- можливість кросплатформної розробки – надає змогу написати код один раз і використовувати його для iOS та Android одночасно. Це надає як безумовні плюси описані вище, так і деякі обмеження у вигляді того, що деякі функції iOS не будуть діяти так як ми сподіваємося у Android, і навпаки;

- використання мови JavaScript замість інших мов, що робить даний framework дуже привабливим для веб-розробників, які хочуть почати розроблювати продукти для мобільних операційних систем;

- продуктивність – додатки написані із використанням React Native працюють близько за швидкістю до native додатків створених на Kotlin або Java [13].

1.4. Обґрунтування вибору мови Kotlin

Вибір найбільш оптимальної мови програмування для додатка на ОС Android залежить від кількох важливих факторів, зокрема вимог до проекту, досвіду команди, бажаних функцій, поширення даної ОС та бюджету замовника.

Ми дослідили, що переважний відсоток користувачів смартфонів в Україні та Європі користуються ОС Android. Таким чином, додаток має бути створений саме для цієї ОС (для даного проекту) або для обох найпопулярніших ОС.

Вибір між мовою Java і Kotlin у користь останній зумовлений тим, що Google активно просуває дану мову як основну для створення Android додатків, та стимулює Java розробників мобільних додатків переходити на Kotlin. Також, у Kotlin багато бібліотек та frameworks що дозволяють більш легко та швидко розробити мінімально життєздатний продукт. Окрім цього ця мова пропоную більшу швидкість розробки продукту через існування великої кількості бібліотек і лаконічності коду. Окрім цього:

- Kotlin має багато корисних мовних конструкцій і функцій, яких немає у Java (null-safety, покращені та скорочені лямбда-вирази, read-only колекції і т.д.);

- маючі нові функції, конструкції та доробки, також існує повна інтеграція з Java. Таким чином, ми можемо легко використовувати Java і Kotlin в одному проекті за необхідністю (тобто якщо необхідно використати існуючу бібліотеку або framework з Java, то це не становить труднощів);

- зменшення шаблонного коду порівняно з Java;

- існування data class - через те, що даний додаток оброблює велику кількість даних що надходять з сервера, то data class забезпечує дуже комфортну роботу із збереженням цих даних за певним шаблоном;

- існування деструктурізація – дозволяє створити більш стислий і читабельний код під час роботи з колекціями та іншими структурами даних;

- coroutines (співпрограми) – підтримка співпрограм забезпечує більш ефективний та простий спосіб асинхронного програмування порівняно з багатопоточністю Java, де потрібно було враховувати багато нюансів при роботі з потоками;

- Kotlin – це за заявою Google починаючи з 2019, є пріоритетною мовою для розробки мобільних додатків [7].

Відмова від кросплатформного рішення на користь native полягає у тому що:

- native додатки забезпечують більш плавну роботу та кращий користувацький досвід;

- кросплатформне рішення не забезпечує можливості у повній мірі використовувати апаратне забезпечення та можливості смартфону. Native додатки мають необмежений доступ до всіх функцій апаратного забезпечення пристрою, а це значить що вони можуть легко взаємодіяти з камерою, GPS, мікрофоном, датчиками та іншими компонентами, забезпечуючи більші можливості для користувача. Кросплатформні додатки навпаки, можуть мати обмеження в доступі до певних апаратних функцій, або вимагати додаткових обхідних шляхів;

- native додатки мають більшу гнучкість для створення дизайну відповідного до стандартів кожної з систем, в нашому випадку – *до* стандартів дизайну Android;

- при розташуванні мобільних додатків на площадках PlayMarket або AppStore, native додатки мають більший рейтинг для пріоритетного відображення. Це відбувається за рахунок того, що дані додатки з самого початку більш відповідають стандартам даних систем або площадок [13].

Так як ціль стоїть як створення мобільного додатка для кафе, орієнтованого на клієнта, то ми повинні сконцентруватися на задоволенні найбільш великої ланки клієнтів, та забезпечити їх кращим користувацьким досвідом. Наступним кроком, який повинен бути ініційований керівництвом кафе, може стати додаткова розробка додатку для iOS.

2. Інформаційна та функціональна модель додатка

2.1. Вимоги до додатка

Додаток складається з двох основних компонентів: клієнтської та серверної частин (роль якої виконує BaaS Firebase), між якими має бути налагоджена взаємодія.

На серверній частині має бути реалізована база даних для отримання і відправки списку страв та інформації про них. Кожна страва має бути представлена як структура даних, що складається з власного унікального по відношенню до всіх сутностей у базі даних ідентифікатора, а також основних і докладних для даної одиниці характеристик (опис, категорія страви, характеристика страви, адреси невеликого стилізованого зображення, а також адреси великого зображення).

Зображення що стосуються страв, повинні також розташовуватися у хмарі та діставатися за запитом.

Взаємодія між клієнтською та серверною частинами має здійснюватися за допомогою HTTP-запитів. При отриманні запитів, наприклад до бази даних, сервер повинен надати відповідь у вигляді JSON, що містить структуровану інформацію за даним запитом.

Користувач повинен мати можливість створити обліковий запис, із застосуванням свого ім’я (або nickname), своєї електронної пошти та пароля. Для підтвердження реєстрації, сервер має надіслати лист до пошти користувача із посиланням підтвердженням.

Крім того, на серверній частині має бути забезпечено прийом даних, що стосуються замовлення або персональних даних користувача, а також застосування пов'язаних із цим змін даних та іншого.

Клієнтська частина повинна бути реалізована у вигляді мобільного додатка, що запускається на мобільному пристрої з операційною системою Android, не нижче версії 7.0 (Nougat, API 24) і представлено у вигляді логічно

пов'язаних екранів, що надають докладну інформацію про питання пов'язані з наданими кафе послугах що цікавлять клієнта. Екрани програми повинні мати загальний стиль візуалізації.

Мобільний додаток повинен надавати такі можливості:

- реєстрація;

- авторизація (виконання входу в особистий кабінет);

- відобразити головне меню з можливістю вибору категорії, що цікавить;

- зберегти дані для входу до додатка, без необхідності додаткового підтвердження;

- змінити особисті дані у профілі;

- відобразити загальний список меню в реалізації lazy, для економії оперативної пам'яті, що споживається додатком;

- виконати пошук за назвою по елементах меню як серед усіх страв, так і по категоріях;

- переходити по натисканню на пункт меню, на наступний екран з можливістю детальніше ознайомитись із стравою,

- скасувати замовлення;

- переглянути кошик та отримання інформації про вартість покупок як окремо, так і сумарно з урахуванням знижок тощо.

- змінювати кількість одиниць товару, що додається в кошик;

У разі відмови роботи серверної частини та подальшої недоступності взаємодії з сервером, додаток повинен працювати у режимі offline (без інтернету), лише із використанням завантажених до локальної бази даних ресурсів, тобто даних отриманих при останній успішній взаємодії з сервером. Однак можливість замовлення та інших дій що потребують контакт з сервером повинні бути заборонені або просто не відправлятися до сервера. Якщо це перший запуск програми, і дані не могли завантажитися в локальну базу даних, тоді відобразити загальний інтерфейс програми (каркас).

2.2. Моделювання загальної структури

2.2.1. Single activity як патерн основної структури

Для створення даного додатка, був обраний патерн «Single activity». Але якщо розібратися, то він являє собою одну Activity і багато Fragments, які знаходяться на даному Activity.

Якщо взяти термінологію Android, то:

- Activity – це один із головних або фундаментальних блоків, з яких і буде будуватися додаток. Його саме головне призначення у тому, що даний компонент використовується як точка входу до додатку, для того щоб завантажилася логіка, підключився графічний інтерфейс і користувач зміг взаємодіяти з додатком [3]. Окрім цього, Activity відповідає за те, як користувач переходить з одних частин додатку до інших.

Якщо спростити пояснення, то Activity це таке вікно, з яким користувач може взаємодіяти. Дане вікно може відтворювати графічний інтерфейс. Воно може займати різну площу екрану. За загальним правилом, кожне Activity має реалізовувати лише одне вікно. Але будь який додаток може мати стільки Activity, скільки потребує його логіка. Саме головне це те, що для будь якого додатка необхідно мати як мінімум одну Activity, яка буде основною (за базовими налаштуваннями вона так і називається MainActivity), а від неї вже відходять інші Activity та елементи.

- Fragment – це модульна частина Activity, яка має свій lifecycle та свої обробники різних подій. Fragment це не заміна Activity, фрагмент не може існувати без неї, тільки разом. Кожне Activity може мати безліч фрагментів.

Ціль введення в Android була така: для розробки більш гнучких інтерфейсів на великих екранах (планшети). Зараз даний підхід також використовуються, але зараз також дуже зручно використовувати фрагменти для розбиття додатку на екрани, які базуються на одній Activity. Це полегшує взаємодію між різними частинами додатку, зменшую ймовірність «memory

leaks» (більш легше керувати життєвим циклом, та завжди можливо знищити даний фрагмент).

Таким чином, Activity являє немов би «контейнер», у якому можуть розташовуватися різні фрагменти.

У даному додатку, є основна MainActivity, у якій є багато фрагментів. Дана Activity має самий глибокий рівень керування. Навіть коли будуть запущені фрагменти і користувач буде взаємодіяти з ними, Activity усе одно буде продовжувати працювати одночасно, не переходячи у режим паузи (функція onPause() не буде запущена).

Для даного додатка взаємодія між Activity і Fragment може бути виражена як на рисунку 2.2.1.1. Назва окремих фрагментів, звісно може відрізнятися від кінцевого варіанту.

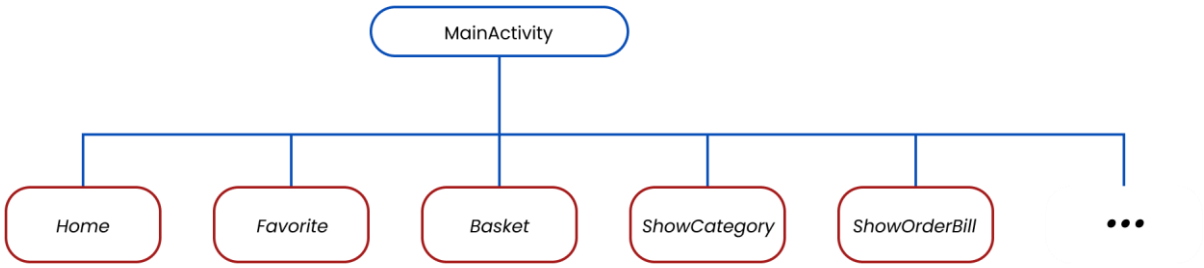


Рисунок 2.2.1.1. Принцип взаємодії *Activity* і *Fragments* даного додатка

2.2.2. Спільна ViewModel для всіх структурних елементів

Для поширення даних між окремими частинами додатку, буде застосовуватися підхід із використанням ViewModel.

Для даного додатка, ViewModel буде зберігати усі необхідні дані для:

- відображення на екрані;
- взаємодії із користувачем;
- взаємодії з локальною базою.

В даному випадку завдяки даному підходу ми отримаємо:

- покращене керування даними - ViewModel діє як центральне сховище даних;

- розділення інтересів – ViewModel відокремлює логіку інтерфейсу користувача (подання даних) нашої діяльності або логіки фрагментів (вибір даних, бізнес-логіка). Це робить код чистішим та зручнішим у обслуговуванні;

- обізнаність про життєвий цикл: ViewModel враховує життєвий цикл. Він автоматично дізнається, коли компонент інтерфейсу користувача (активність або фрагмент) проходить різні етапи життєвого циклу (наприклад, onCreate, onDestroy). Це дозволяє обробляти дані відповідним чином, наприклад, отримувати дані, коли інтерфейс користувача стає активним, і очищати ресурси, коли він знищується;

- обмін даними між фрагментами: існує можливість використовувати один екземпляр ViewModel, спільний для кількох фрагментів у межах додатку. Це дозволяє їм отримувати доступ до однакових даних і оновлювати їх, не потребуючи складних механізмів зв'язку [6].

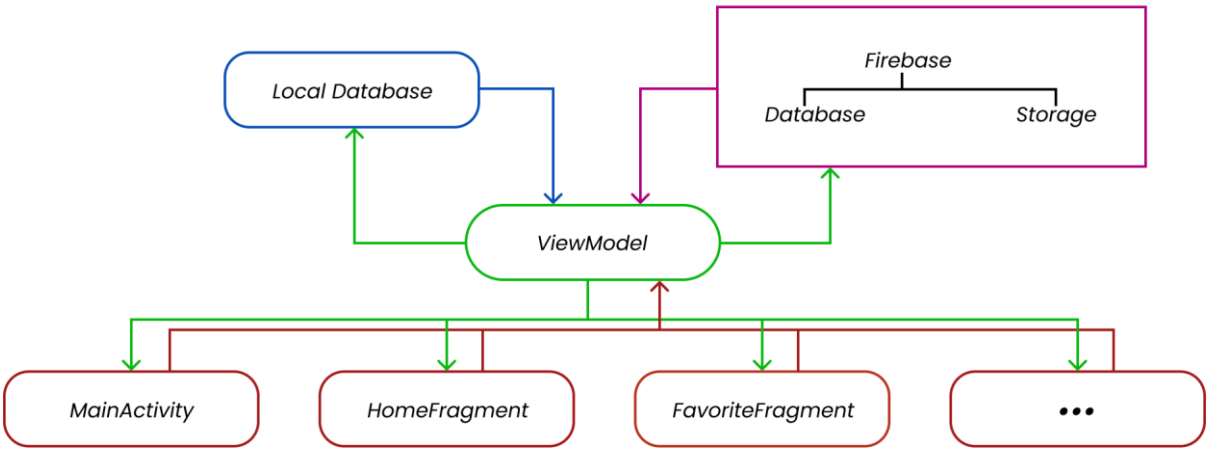


Рисунок 2.2.2.2 Структура взаємодії основних елементів додатку

На рисунку 2.2.2.2. зображено загальну структуру взаємодії локальної бази даних, серверу, ViewModel, та фрагментів. Уся інформація що надходить до додатка від локальної бази даних зберігається у ViewModel, і фрагменти можуть отримати доступ до неї лише через даний «шлюз». Дані з Firebase

також зберігаються у локальну базу даних через ViewModel. Важливим для даного додатка зауваженням є те, що запит на запис або отримання даних від серверу (бізнес-логіка), може, і в більшості випадків так і є, надходить від фрагментів. Але все одно, вона тут же зберігається саме у ViewModel. Нами було обрано не відображати дані трансферні дії у схемі, оскільки ціль все одно досягається завдяки ViewModel.

2.2.3 Хмарна база даних

Усі дані додатка мають зберігатися безпосередньо у базі даних на сервері. Для даного проекту, було обрано не звичайну БД, а Firebase Realtime Database. Суть даної «БД», що вона являє собою хмарну базу даних NoSQL, призначена для зберігання та синхронізації даних у режимі реального часу. Також важливо зазначити, що вона використовує структуру даних дуже схожу на JSON (використовуються одні і ті структурні принципи).

Дана модель хмарної бази даних було обрано через такі функції що вона надає:

- оновлення даних в режимі реального часу - будь-які зміни в даних відразу відображаються на всіх підключених клієнтах (пристроях) за лічені мілісекунди, таким чином, замовлення що буде створене або інформація що користувач змінить, будуть отримані майже одразу;

- підтримка автономності - пакети SDK бази даних у реальному часі дозволяють зберігати локальні дані. Це означає, що користувачі можуть отримувати доступ до даних і змінювати їх навіть у режимі offline. Коли вони відновлюють з’єднання, локальні зміни синхронізуються з хмарою. Це дуже важливо для даного додатка, через те що серед вимог була зазначена автономність (підтримка роботи без Інтернету);

- безпека даних - Firebase Realtime Database взаємодіє з Firebase Authentication, щоб забезпечити правила безпеки на основі перевірки ключових (секретних) полів у зареєстрованих користувачів. На основі цих

перевірок і правил, з’являється можливість налаштувати правила доступу до бази даних. Таким чином, ми можемо контролювати, хто може читати та записувати дані на основі статусу автентифікації;

- спрощення структури даних – оскільки у даному додатку використовуються Data classes у якості шаблона для entities, а у даних типах класів використовуються структура «поле та його значення» (що можна використовувати як ключ-значення), то запис і зчитування з подібної бази даних не становить складності.

Серед недоліків виділяють:

- неефективність (а іноді і неможливість) формування складних запитів – для даного додатка це нівелюється за рахунок того, що база даних не має складних структур. Нам потрібно лише отримати дані, інше виконується на пристрої;

- потенційність конфліктів даних: завдяки моделі кінцевої узгодженості (дані зрештою відображаються на всіх клієнтах) існує невелика ймовірність конфліктів даних під час редагування та синхронізації у режимі offline.

Структуру хмарної бази даних, наводимо на рисунках 2.2.3.1 та 2.2.3.2:

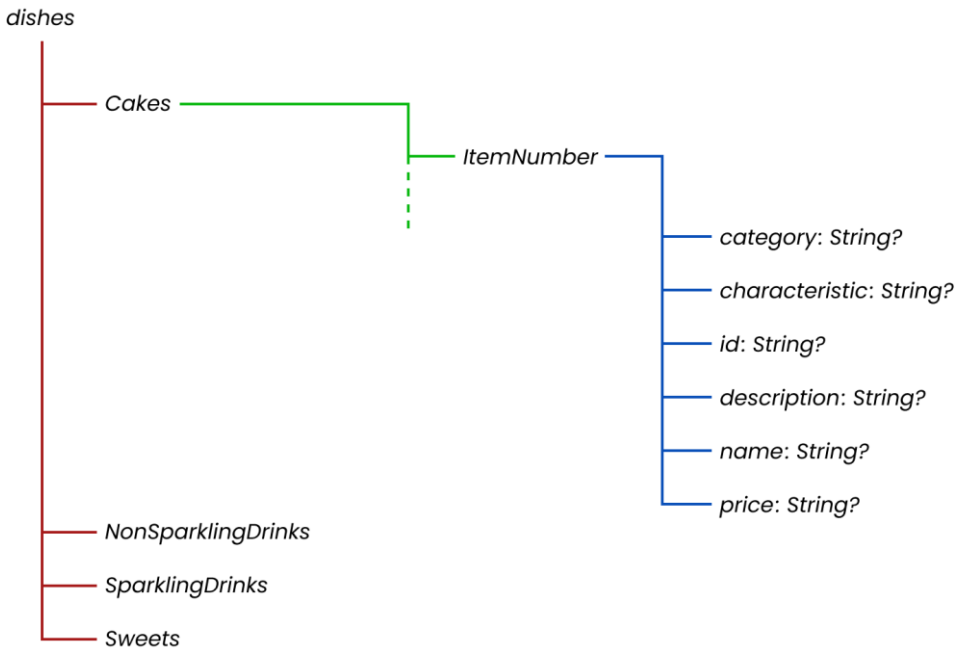


Рисунок 2.2.3.1 Структура моделі «меню» кафе на хмарній базі даних

Структура поділяється на дві основні гілки: страви та дані користувача. Страви у свою чергу поділяються на чотири основні категорії, газовані напої, негазовані напої, цукерки та тістечка. Кожна з категорій має свої власні страви, які мають характеристики id, назва категорії, опис, і вартість, тощо.

Кожна нова страва, повинна мати дані властивості.

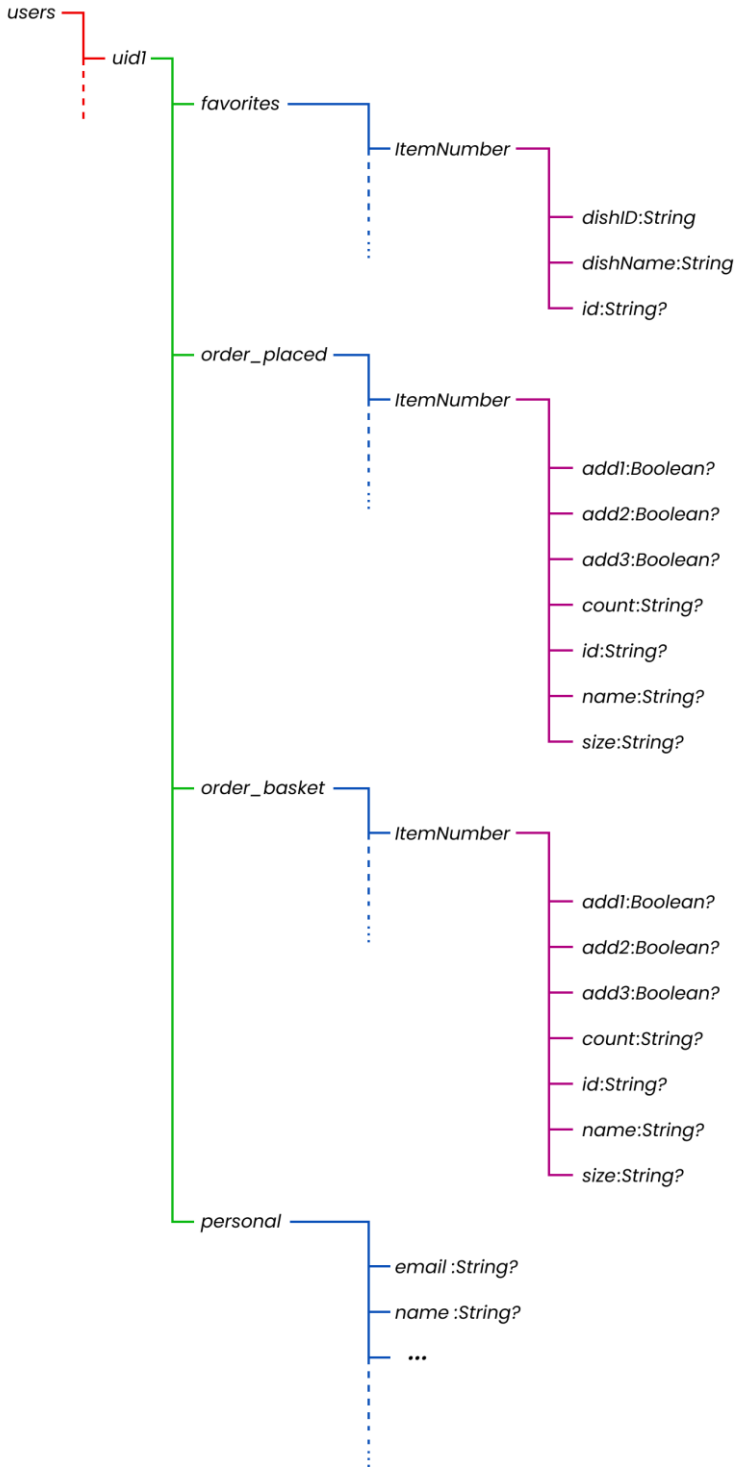


Рисунок 2.2.3.2 Структура моделі «користувач» на хмарній базі даних

Інша гілка – це користувачі. Кожен користувач повинен при реєстрації додаватись до даної гілки під своїм унікальним ідентифікатором (UID). Даний ідентифікатор створюється сервером та має високий рівень безпеки. На всьому протязі життя облікового запису користувача, даний ідентифікатор незмінний.

Далі, у кожного користувача є чотири основні розділи, це:

- обрані страви (favorites) – у даному розділі будуть розташовуватися елементи страв, що подобаються користувачу (він додав їх до улюбленого), та кожна з цих страв-елементів має таку структуру: ідентифікатор внутрішній, ідентифікатор страви та назва страви. Потім, ми можем шукати дану страву виходячи з даних параметрів за актуальною вартістю серед усіх страв;

- страви що йдуть у замовленні (order_placed), кожна зі страв повинна мати ідентифікатор, назву, кількість, розмір, та перелік добавок що обрав користувач;

- страви що додані у кошик (order_basket), але ще не додані у замовлення. Структура аналогічна замовленню;

- дані користувача (personal) – у даному розділі може бути лише один об’єкт. Він йде з такими параметрами як: ім’я, прізвище, електрона пошта, локація, телефон, знак зодіаку тощо. Звісно, обов’язкові це лише пошта та ім’я, а усе інше це на вибір користувача.

2.2.4 SQLite база даних, як локальне сховище

У даному додатку повинна бути реалізована локальна база даних, щоб додаток міг зберігати дані про користувача та елементи меню кафе, а користувач, відповідно, міг за відсутністю Інтернет з’єднання, передивлятися ті елементи, що були завантажені. Таким чином додаток буде вигідно відрізнятися від звичайних веб-додатків. Окрім цього це забезпечить збереження трафіку та можливість продивлятися меню без доступу до інтернету.

Android має за замовчуванням встановлену базу даних SQLite. Раніше, щоб користуватися базою даною, нам потрібно було б самостійно створювати і складні конструкції, забезпечувати безпеку операцій і писати важкі за наповненням коду запити, тобто виконувати SQLite API. На даний момент для реалізації локальної бази даних, офіційна документація рекомендує використовувати бібліотеку Room. Room, це можна сказати полегшуюча огортка, яка дозволяє використовувати усі можливості SQLite, але в той же час бере на себе обов'язки написання коду. Розробнику залишається зосередитися на написанні запитів [10].

Локальна база даних повинна ініціалізуватися навіть раніше чим Activity завантажиться, і на протязі усього додатку забезпечувати доступ до того ж самого синхронізованого об'єкту бази даних. Загальна структура виглядає наступним чином (рис. 2.2.4.1):

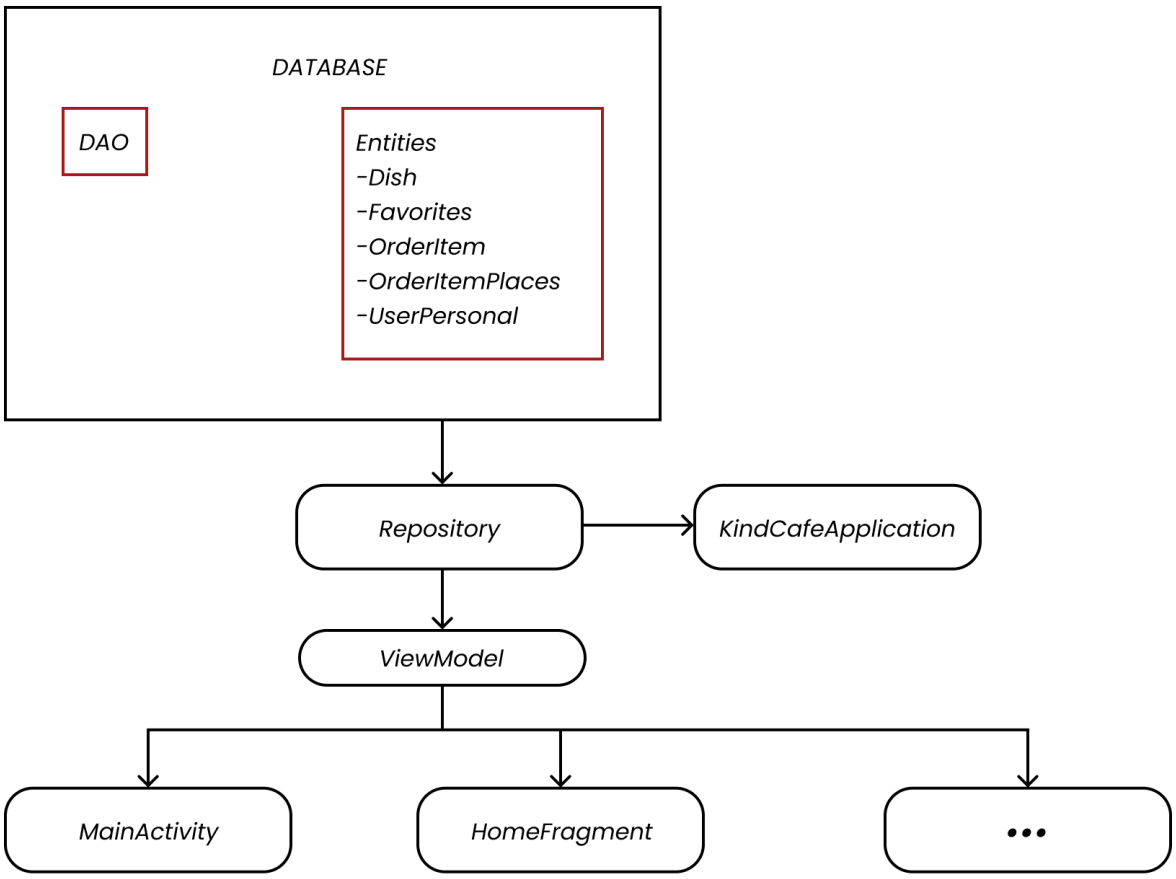


Рисунок 2.2.4.1 Модель зв'язку локальної бази даних і елементів додатка

На рисунку 2.2.4.1 зображено, що для бази даних ми повинні створити сутності (в даному випадку Data classes), які і будуть являти собою таблиці у базі даних. Для даного додатка, потребуються таблиці що будуть відображати страву, обране, страви у кошику (але ще не у замовленні), страви у замовленні, дані щодо поточного користувача.

Для того щоб таблиці могли бути використовувані у додатку, потрібно вказати які дії можна с ними виконувати. Для цього необхідно створити інтерфейс DAO (Data Access Object), та вказати у анотації команди SQLite, що потрібно виконати. Коли ми об’єднаємо всі частини, препроцесор Room сам створить необхідний код.

Для використання даної бібліотеки, в даному додатку буде використаний патерн Singleton. Даний патерн дозволяє гарантувати, що клас має тільки один екземпляр для додатка, і не може створювати більше. Також він забезпечує глобальну точку доступу до цього екземпляру. Таким чином ми зможемо контролювати доступ к певному ресурсу, у даному випадку до бази даних.

Саме за цим патерном, ми повинні створити Singleton Repository, у якому буде ініцілізуватися база даних. За даною логікою даний Singleton повинен створитися одразу як додаток запуститься, але до запуску MainActivity (у KindCafeApplication).

Потім єдиний доступ до функцій взаємодії з базою даних ми можем отримувати через зв’язку Repository - ViewModel.

2.2.5 Відстеження подій

Відстеження подій – це важлива частина роботи додатка, яка полягає у тому, що ми повинні спостерігати над станом інформації у базі даних або інших джерелах інформації, щоб при зміні цих даних, одночасно змінити стан графічного інтерфейсу або його елементів.

Для реалізації даної можливості, нами було обрано поведінковий шаблон проектування – спостерігач (Observer). Даний шаблон визначає залежність типу «один до багатьох» таким чином, що при зміні певного об’єкта, усі інші об’єкти що залежать від нього, отримують повідомлення про цю зміну (про цю подію).

Щоб скористатися цим патерном, у проекті було використано StateFlow. StateFlow – це особливий тим Flow, який базується на концепції потоків (які є асинхронними потоками даних), але додає спеціальні функції, корисні для керування станом даних.

Тобто, як і звичайні потоки, StateFlow видає потік значень протягом певного часу. Це дозволяє збирати ці значення та реагувати на зміни в компонентах UI.

Серед переваг даного Flow є:

- існування єдиного або центрального джерела істині для усіх елементів що підписані на нього. Він містить одне значення за раз і будь-які оновлення цього значення автоматично поширюються на всі збирачі, підписані на потік. Наприклад, якщо користувач видалить страву з обраного, то дана зміна торкнеться бази даних, і всі хто підписані на цей розділ, отримають інформацію що він був змінений і на даний момент має такий стан – StateFlow надсилає поточний лист страв що знаходяться у обраному. На основі цих даних, усі збирачі виконують запрограмовані дії;

- урахування життєвого циклу – при запуску у coroutine (наприклад у viewLifecycleOwner.lifecycleScope), відповідно в межах життєвого циклу пов’язаного з даним структурним елементом (фрагментом або Activity), StateFlow автоматично припиняє видавати оновлення, коли UI знищується, і відновлює, коли він знову стає активним. Це допомагає уникнути витоків пам’яті та несподіваної поведінки під час змін конфігурації;

- змішані оновлення - StateFlow зберігає лише останнє значення. Коли видається нове значення, попереднє значення відкидається і тільки нове

значення доступне для збирачів. Така поведінка спрощує керування станом порівняно з керуванням кількома змінними стану [9].

Таким чином, кожен з джерел інформації знаходиться під постійним спостереженням, і елементи поточного екрану які залежать від даної інформації, також постійно слухають про обновлення інформації та відновлюють ці дані відповідно з вкладеною логікою. Це забезпечує постійну актуальність даних і знімає проблему про відстеження змін у даних.

2.3. Функціональна модель додатка

Для моделювання і демонстрації функціональної моделі додатку у даному проекті будуть використовуватися нотації IDEF0 та IDEF3

2.3.1. IDEF0

IDEF0 – це методологія функціонального моделювання та графічна нотація, що використовується для створення функціональної моделі, що відображає структуру та функції системи, а також потоки інформації та матеріальні об'єкти, що пов'язують ці функції.

Нотацією називається формат опису бізнес-процесу, що є сукупністю графічних об'єктів що використовуються при моделюванні, а також правил моделювання. По суті та змісту – це особлива графічна мова, яка дозволяє описувати роботу підприємства, наочно демонструвати взаємодію між різними підрозділами, тобто. описувати бізнес-процеси (у нашому випадку для мобільного додатку).

Розроблена за IDEF0 функціональна діаграма буде мати:

- 0 рівень з одним функціональним блоком. Буде проведена декомпозиція даного блоку;

- 1 рівень з 5 функціональними блоками. Також буде проведена декомпозиція двох функціональних блоків;

- 2 рівень. Декомпозиція даного рівня буде проведена для одного блоку.

Розглянемо нульовий рівень моделі (рис. 2.3.1.1), він являє собою загальну структуру додатка та позначає головну мет, навіщо він створюється та які ресурси потребує.

Рисунок 2.3.1.1 Рівень 0 функціональної діаграми *IDEF0*

Розшифровуючи діаграму, дамо пояснення по кожному з елементів, що знаходяться на діаграмі:

- зліва позначені дані що надходять зовні та які будуть використані для успішного створення оформлення замовлення. Наприклад, ми не зможемо вірно це зробити, якщо користувач не зазначить своє ім’я (або псевдонім), адресу електронної пошти для входу до облікового запису (або створення останнього), інформацію про страву тощо;

- зверху знаходиться елемент керування – користувач. Користувач виступає у ролі керівника процесу через те, що саме він вирішує які дані представляти та які страви обирати тощо;

- знизу знаходиться елемент під назвою «Механізм». Суть цього елементу у представленні того, хто буде виконувати усі дії ініційовані користувачем (елементом управління) стосовно даних поданих на вхід (зовні) – тобто в нашому випадку це безпосередньо мобільний додаток.

І коли всі ці елементи зробили певну закладену роботу і виконали своє завдання, всі дані перетворені у ході роботи надходять на вихід, права частина діаграми. Тобто у ході виконання завдання усіма елементами буде створене оформлене замовлення.

Далі буде представлений перший рівень даної діаграми. Це більш докладна візуалізація процесу. Можна навіть сказати, що це декомпозиція 0-го рівня.

На даному рівні (рис. А.1), ми бачимо вже одразу п’ять функціональних блоків. Одразу слід помітити, що усі дані, що подаються на вхід: елемент керування та механізм виконування – залишилися тими ж. Це пояснюється дуже просто: виходячи з того, що ми просто більш докладніше розбираємо функціональний блок з нульового рівня, то тут ніяк не можуть з’явитися нові дані, бо це б суперечило логікі не тільки IDEF0, але й у якомусь сенсі закону збереження енергії, який говорить що: «Ніщо не виникає з нізвідки і не зникає в нікуди».

Зараз ми розглядуємо такий варіант розвитку подій, коли користувач хоче замовити страву, і він її замовляє. Для цього він повинен спочатку зареєструватися або якщо він вже має обліковий запис то авторизуватися. Далі будемо вважати що він ще не має облікового запису і повинен зареєструватися. Після цього йому необхідно здійснити пошук страв, провести налаштування страв, а саме обрати кількість страв, розмір бажаних страв, обрати особливість даних страв, наприклад газований напій може бути з лимонним смаком, апельсиновим тощо. Також якщо дозволяє страва, то можна обрати добавки,

наприклад сироп, горіхи тощо. Після підтвердження вибору, користувачу надається можливість перевірити докладно з чого складається вартість, тобто буде показаний віртуальний чек і якщо користувач з усім згоден, то він натискає «Send» і замовлення відправляється до сервера, де його потрібно вже оброблювати тощо. Подробиці, на кшталт що сформоване замовлення буде надіслане на сервер, оброблене та збережене у даній діаграмі ми не розглядаємо, наша задача закінчується на відправці замовлення на сервер.

Далі зробимо декомпозицію функціонального блоку «A-1» та «A-3» та розглянемо кожен з отриманих результатів.

У результаті декомпозиції блоку «A-1» (рис. В.1), це блок «Реєстрація», була отримана діаграма яка складається з чотирьох функціональних блоків, які також пов’язані один з одним. Тобто для виконання реєстрації, користувачу потрібно надати інформацію по кожному з цих блоків, щоб завершити реєстрацію. Не пройшовши по кожному з цих блоків, користувач не зможе отримати вихідний результат.

Тобто поточному блоку (наприклад «A-12») не важливо що користувач ввів на попередньому блоці («A-11»), але дані що надходять зовні, повністю повторюють ті дані що ми отримали на попередніх кроках, за тим виключенням, що на даному етапі ми потребуємо усіх тих даних. Зараз, для виконання функціонального блоку «A-1», нам достатньо знати про: ім’я, адресу електронної пошти, пароль. Маючи ці дані, додаток відсилає їх на сервер, де вони проходять перевірку та якщо усе гаразд, то реєструє даного користувача та надає йому права додавати страви до улюбленого, користуватися «кошиком» та відправляти замовлення до серверу (можна розглядати це як відправка замовлення до кафе).

Далі розглянемо декомпозицію блоку «A-3» - «Налаштування страви» (рис. С.1). Усі правила що були наведені вище, мають місце і у цьому разі, а також важлива послідовність наданих даних.

Наприклад, функціональний блок «A-32» – «Збільшити лічильник кількості одиниць страв до задовільного результату», не може бути виконаний,

поки користувач не виконає блок «A-31» – «Додати знайдену страву до кошика». Це дуже просто пояснюється тим, що неможливо змінити кількість страв, якщо ще не обрав цю страву і не додав її до кошику. У результаті, коли користувач послідовно виконує дії по обиранню страви, її кількості, добавок до неї, типу страви, її розміру, та натискає поле «Make order», то буде збережено інформацію про бажану страву та її характеристики, як у локальній базі даних, так і на сервері. Проте дана інформація це ще не замовлення, це лише запис про намір. І тільки після блоку «A-4» буде відправлена інформація як замовлення, а інформація про кошик видалиться через непотрібність.

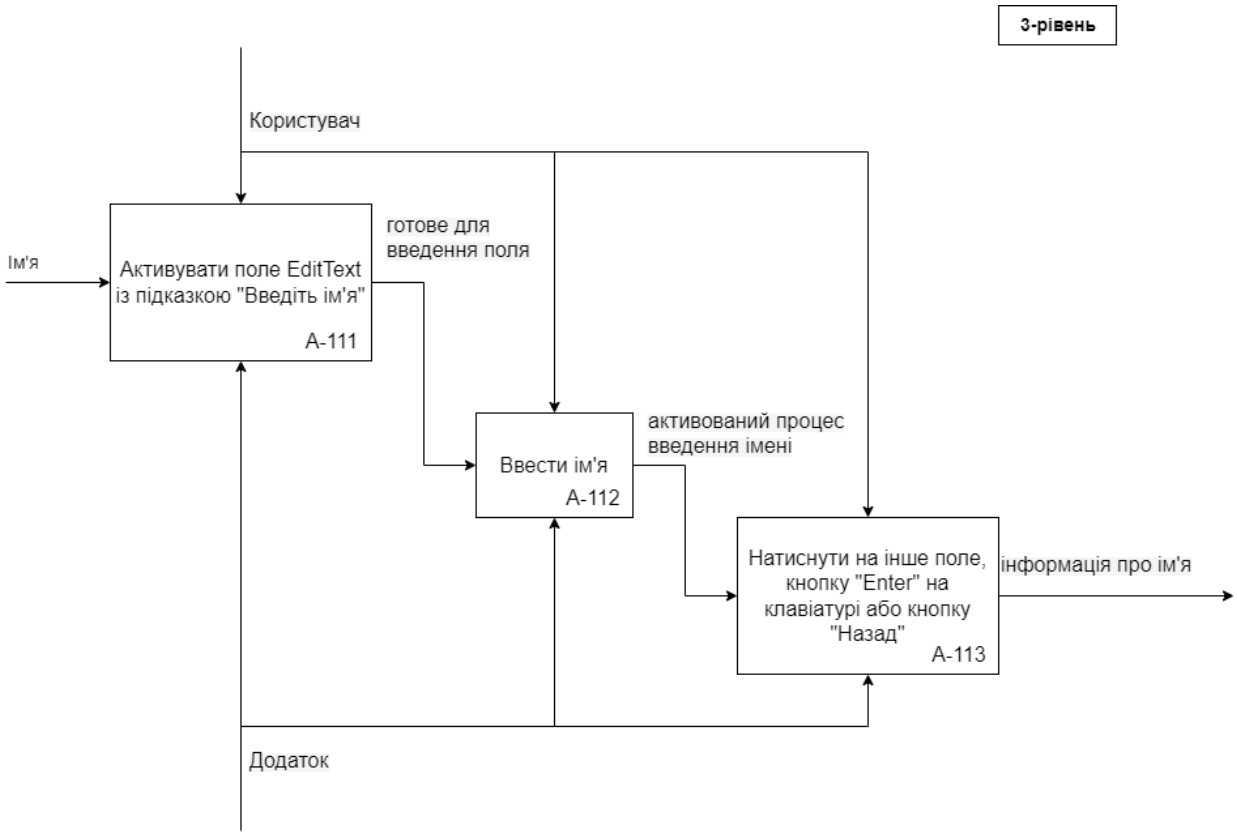


Рисунок 2.3.1.2. Рівень 3 функціональної діаграми *IDEF0*, декомпозиція блоку *A-11*

Наприкінці розглянемо найбільш глибокий рівень декомпозиції, зображений на рисунку 2.3.1.2.

У даному випадку, ми виконали декомпозицію блоку «Ввести ім’я», який є результатом іншої декомпозиції (такий ланцюжок продовжується до рівня 0 включно).

На рисунку 2.3.1.2 ми бачимо, що кожен функціональний блок пов’язаний з попереднім, а також потребує лише невеликої частини даних що ми затребували на нульовому рівні. Даний факт пояснюється глибиною декомпозиції. Користувачу потрібно послідовно виконати дії: торкнутися поля «Введіть ім’я», тобто активувати його, потім ввести своє ім’я та натиснути або кнопку «Enter» на програмній клавіатурі, або кнопку назад, щоб згорнути цю клавіатуру.

При декомпозиції важливо не зайти настільки глибоко, що втратиться сама сутність нотації IDEF0. Тобто, ми повинні загально відобразити бізнес процеси і не повинні торкатися саме програмних елементів продукту.

2.3.2. IDEF3

Нотація IDEF3 – це методологія моделювання та стандарт документування процесів (у т. ч. технологічних процесів), що відбуваються в системі, а також механізм збирання інформації про процеси. Це найважливіша нотація після IDEF0 і призначена для опису потоків робіт бізнес-процесу (WFD). Як правило, використовується спільно з нотацією IDEF0, але може і окремо.

IDEF3 показує причинно-наслідкові зв'язки між ситуаціями та подіями у зрозумілій експерту (аналітику) формі, використовуючи структурний метод вираження знань про те, як функціонує система, процес тощо.

Отже, розглянемо нотацію IDEF3 відповідно до нашого проекту (рис. D-1). На даній нотації зображені процеси що описують механізм замовлення страв у кафе. Перш за все, щоб почати взаємодіяти з додатком, користувач повинен відкрити його. Наступним кроком буде «Реєстрація» (пам’ятаємо, що користувач хоче саме заказати страву).

Даний крок позначений знаком «&» (логічне «І»), тобто він зобов’язує послідовно виконати усі дії що в ньому знаходяться. Так, користувач не зможе продовжити процес реєстрація, якщо не введе ім’я, електрону пошту і пароль.

Наступний блок 1.9 («Здійснити пошук страви»), використовує оператор логічної операції «Або» - «О». Тобто, щоб виконати цей блок, користувач повинен знайти бажану страву використовуючи поле для введення назви, або скористуватися пошуком по категоріям.

Далі, обираємо страву і переходимо до кошику, де знову використаємо логічне «І» щоб обрати кількість страв, тобто ми повинні спочатку обрати страву і потім обрати її кількість. Також після додавання до кошику, необхідно окрім кількості обрати розмір та тип страви або добавки. Для комфорту користувача, необхідні поля за замовчуванням обрані як найменші одиниці (тобто найменший розмір) і якщо користувач не хоче обирати їх, то будуть обрані саме ці дані.

Після налаштування страви, користувачу потрібно натиснути «Make order», перевірити вартість замовлення у віртуальному чеку і якщо все гаразд, то завершити замовлення натисканням кнопки «Send».

3. СТВОРЕННЯ МОБІЛЬНОГО ДОДАТКА

3.1. Розробка графічного інтерфейсу

Розробку будь якого графічного інтерфейсу слід проводити у найбільш підходящих для цього програмних засобах. Для проектування графічного інтерфейсу даного додатку, нами було обрана Figma.

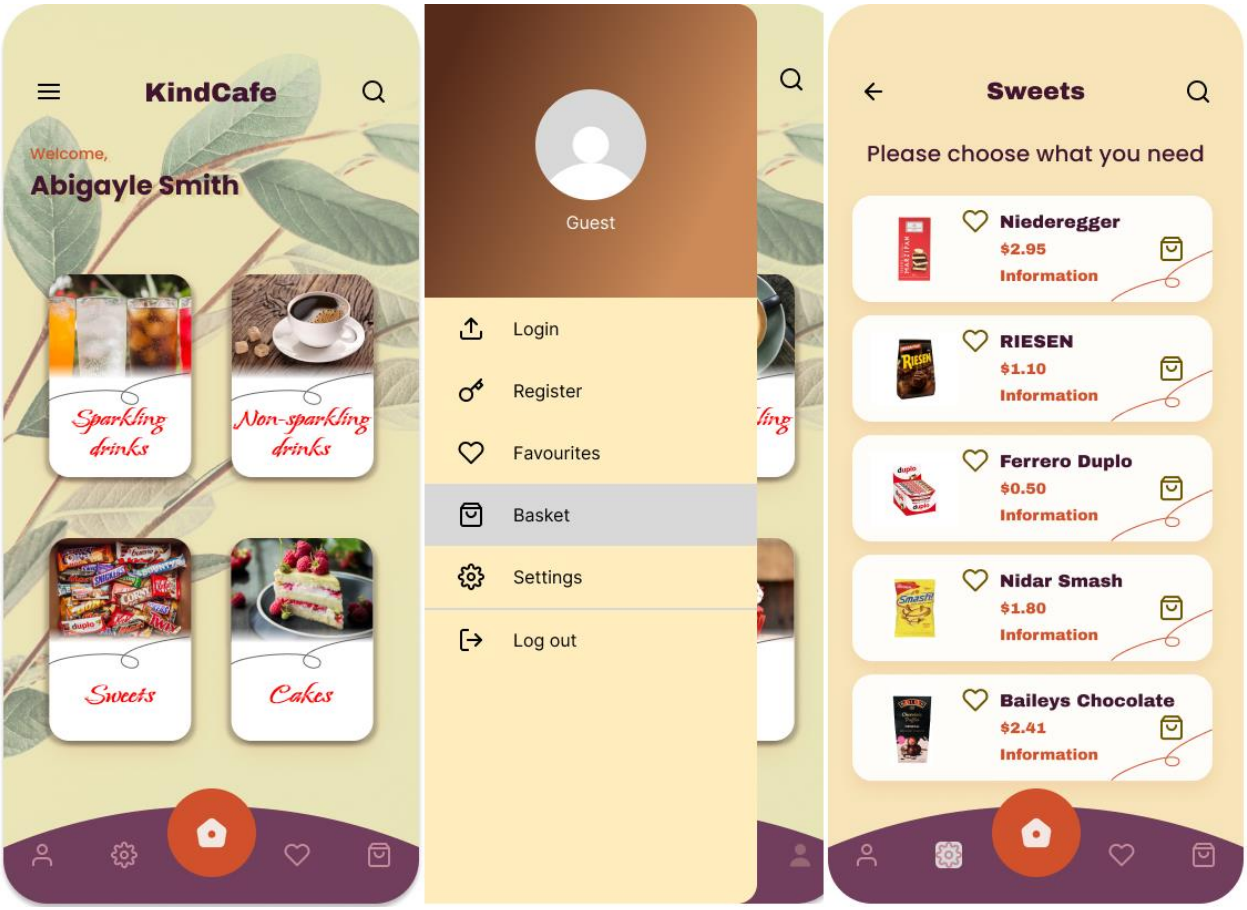


Рисунок 3.1.1. Головний екран, бокове меню та екран з відображенням цукерок

На рисунку 3.1.1. можна бачити, як має виглядати деякі з екранів додатку. Основуючись на даній візуальній інформації, нам необхідно перенести це до Android Studio. Під «перенести» мається на увазі, що нам потрібно програмними засобами реалізувати дані екрани у Android Studio.

Оскільки ми використовуємо звичайний підхід до програмування мобільних додатків, а не framework Jetpack Compose, то можемо зробити макети (layouts) прямо у xml файлах і додати ці макети як графічний інтерфейс до додатку. Подальші дії з цим інтерфейсом, це взаємодія останнього з інформацією і користувачем, які можуть змінювати дані структурні компоненти для досягнення своїх цілей.

Розглянемо найбільш важливі моменти при створенні даних екранів, та сконцентруємо увагу більш на підході який буде використаний для побудови.

Перш за все зазначимо, що в нас буде один головний екран і усі інші другорядні. На головному екрану ми розташуємо Toolbar який буде використаний далі який модифікація стандартного ActionBar (це стало можливо починаючі с API 21). Toolbar можна розглядати як Header для екрану (рис. 3.1.2).

Рисунок 3.1.2. *Toolbar* додатка

На даному toolbar буде розташовано три основних елемента, це значок «Гамбургер» (яка буде далі визивати бокове меню), він може змінюватися на значок «Стрілка направо» (його ціль – це повернення на попередній екран), текст що означає локацію (для екранів крім головного) або назву кафе (для головного екрану), та значок «Пошук», при натисканні на який буде запускатися механізм пошуку.

Для забезпечення більш витонченого відображення тексту на Toolbar, нами було змінена структура даного елементу за замовчуванням, а саме доданий всередину елемент TextView для відображення тексту (рис. 3.1.3).

TextView – це елемент розроблений для відображення тексту, це його основна ціль. У даному елементу ми можемо налаштувати текст із використанням стилів, розміру шрифту, положенню на екрані тощо [8].

```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/tb_main"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:elevation="9dp"
    android:background="@android:color/transparent"
    android:minHeight="?attr/actionBarSize"
    android:paddingStart="20dp"
    android:paddingTop="30dp"
    android:paddingEnd="20dp"
    android:theme="?attr/actionBarTheme"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:menu="@menu/toolbar_menu_general">

    <TextView
        android:id="@+id/tvToolbarTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        tools:text = "Kind cafe"
        android:textColor="@color/toolbar_title"
        android:textStyle="bold"
        android:textSize="24sp"
        android:layout_gravity="center_horizontal"
         />
</androidx.appcompat.widget.Toolbar>
```

Рисунок 3.1.3. Реалізація *Toolbar* у *xml*

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:id="@+id/clMainBottomMenu"
    android:layout_width="match_parent"
    android:layout_height="80dp"
    android:background="@drawable/ellipse_of_lower_menu"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent">

    <ImageButton
        android:id="@+id/ibProfile"
        android:background="@drawable/ripple_test"
        android:layout_width="30dp"
        android:layout_height="30dp"
        android:layout_marginTop="15dp"
        android:layout_marginStart="25dp"
        app:srcCompat="@drawable/ic_user"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent" />

    <ImageButton...>

    <ImageButton...>

    <ImageButton...>
</androidx.constraintlayout.widget.ConstraintLayout>
```

```
<ImageButton
    android:id="@+id/ibHome"
    android:layout_width="80dp"
    android:layout_height="80dp"
    android:background="@drawable/home_button_back"
    android:layout_marginBottom="15dp"
    android:src="@drawable/ic_home"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent" />
```

Рисунок 3.1.4. Реалізація *custom BottomBar* у *xml*

Аналогічно нам потрібно розробити нижнє меню. Через те що ми хочемо реалізовувати свою специфічну графічну структуру, нами було прийнято рішення про реалізацію даного елементу з нуля. Замість використання готового шаблона, ми створили окремий шар, в якому розташували чотири ImageButton, і одну зовні (рис. 3.1.4).

Окрім цього, для натискання на дані кнопки також була розроблена спеціальна анімація (рис. 3.1.5).

```
<ripple xmlns:android="http://schemas.android.com/apk/res/android"
    android:color="@color/white">

    <item>
        <shape android:shape="rectangle">
            <solid android:color="@color/bottom_menu_color"/>
            <corners android:radius="5dp"/>
        </shape>

    </item>
</ripple>
```

Рисунок 3.1.5 Імітація натискання на кнопки у *BottomBar*

При натисканні на кнопки у нижньому меню, буде плавний перехід з кольору кнопки до білого кольору і навпаки.

Простір, що залишився між верхнім та ніжнім меню, займає собою FragmentContainerView – елемент, що служить місцем де будуть розташовані абсолютно усі фрагменти для даного додатку. Таким чином ми забезпечуємо, що елементи верхнього та нижнього меню будуть завжди видні та незалежні від фрагментів.

Наступним реалізуємо фрагмент, який буде служити для відображення одної з категорії меню. Тут важливо зазначити, що ми намагалися де можливо уніфікувати структурні компоненти для можливості їх перевикористання із різними даними.

У фрагменті буде знаходитися TextView та вкладений елемент що може додатково прокручуватися з RecyclerView всередині, який і буде відображати список страв у меню (рис. 3.1.1).

Така складна структура необхідно для того, щоб картка зі стравою могла заходити за нижнє меню з кнопками, та виходити за межі його. Таким чином не буде рубленого виду. Окрім цього, для того щоб був вихід за межі нижнього меню, ми програмно указали відступ з низу, що дорівнює висоті даного меню (рис. 3.1.6).

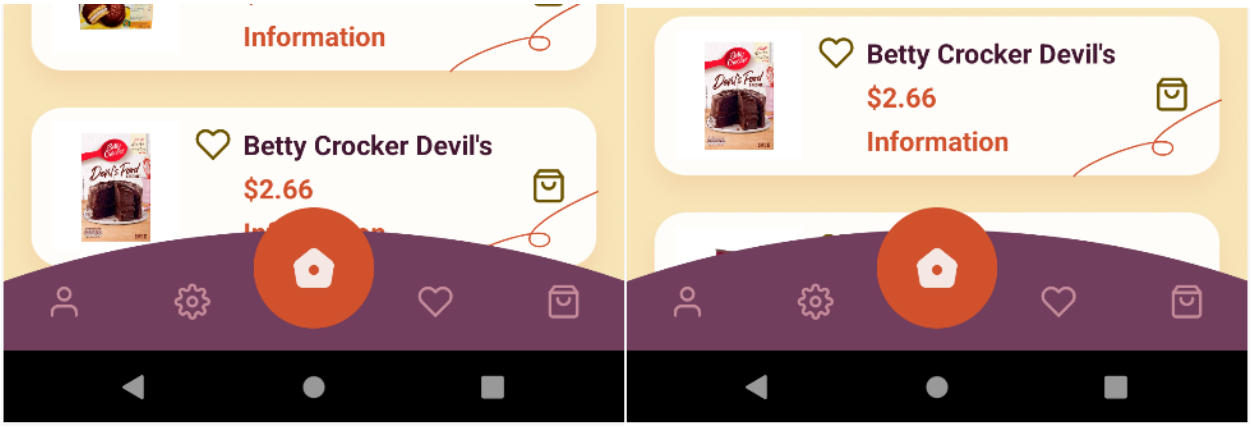


Рисунок 3.1.6 Відображення ходу картки за межі нижнього меню

Зазначимо, що для відображення меню у RecyclerView, нам необхідно розробити один шаблон картки і потім він буде використаний для відображення усіх страв, лише із зміненням інформації для кожної страви.

За цим же принципом були розроблені і інші фрагменти де необхідно відобразити страви: Favorites, Basket. Для Basket було лише перероблена структура картки та доданий додатковий елемент кнопки для продовження замовлення.

Окремо звернемо увагу на дизайн для фрагмента де буде відображений віртуальний чек.

Для даного layout використаний схожий підхід, але ми змінили вкладений елемент що прокручується на звичайний ScrollView. У даний елемент ми розташовуємо TextView(s), у які будуть відображені назва елементу, закладу і звісно сам чек (рис. 3.1.7).

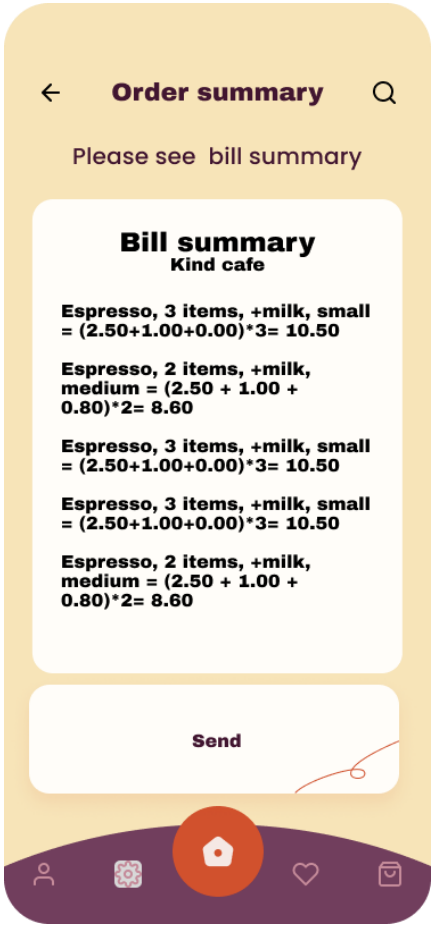


Рисунок 3.1.7 Відображення чеку

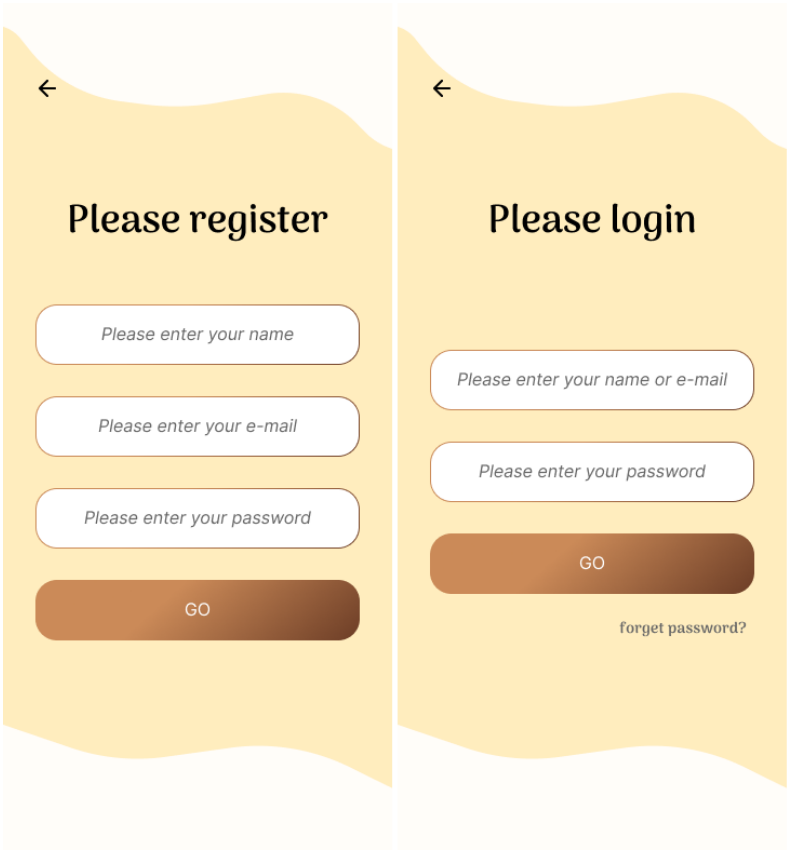


Рисунок 3.1.8 Зовнішній вид екранів *Login* і *Registration*

Екрани реєстрації виконані за схожими принципами із використанням компонентів PlainText (EditText), які запрограмовані на введення певних символів та правила вводу. Наприклад, для електронної пошти використаний алфавіт та значок «@» на першому плані, а кожен символ пароля буде прихований. Зовнішній вид даних екранів наведений на рисунку 3.1.8.

Розглянемо також екрани налаштувань користувача (рис. 3.1.9). Верхня частина даних екранів представляє собою елемент, що ми використовували у боковому меню, і у даному випадку ми використовуємо її знову. Далі йде комбінація значків і полів для вводу. Окремо звернемо увагу на вибір знаку зодіаку. При натисканні на дане поле, буде з'являтися додатковий перелік знаків зодіаку, і користувач зможе обрати той що йому потрібен.

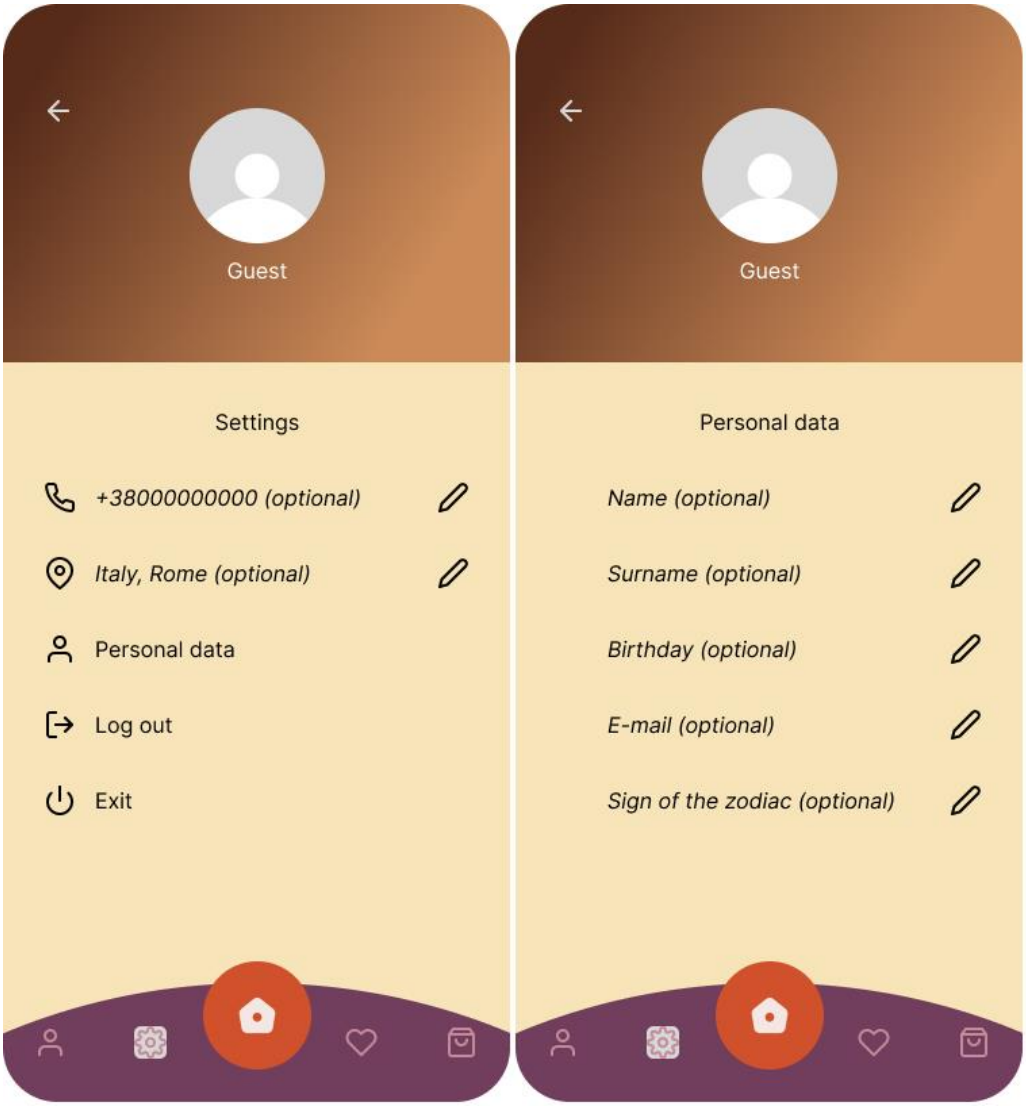


Рисунок 3.1.9 Зовнішній вид екранів *Setting General* і *Setting Personal*

Також зазначимо, що у даному додатку не планувалася функція зміни адреси електронної пошти, тому дане поле на layout буде TextView, тобто тільки для перегляду. Відображатися буде та пошта, яка була використана для реєстрації.

3.2. Програмна реалізація додатка

Для опису програмної реалізації додатка, опишімо лише основні, ключові функції та аспекти, а також елементи на які потрібно звернути увагу.

Опишемо процес реєстрації та авторизації користувача. Для реалізації даних функцій ми використовуємо сервіс Firebase Authentication та будемо використовувати ті API що він надає.

Для цього ми створимо окремий клас AccountManager, у якому буде знаходитися логіка реєстрації та авторизації користувача із використанням зазначених вище API.

Розглянемо процес реєстрації облікового запису (рис. 3.2.1). Для цього користувачу буде потрібно ввести ім’я, адресу електронної пошти та пароль. Далі, ми перш за все перевіряємо чи усі поля були заповнені, а потім звертаємося до API, куди і передаємо ці дані. API додатково перевіряє їх на коректність (наприклад, що у email немає пустого простору, а пароль складається не менш з 8 символів).

Якщо надані дані пройшли перевірку і все гаразд, тоді нам повернеться результат з кодом що сигналізує про успіх і на цьому можна закінчувати. Але для нашого проекту ми також реалізуємо надсилання посилання для верифікації облікового запису, а також більш гнучкий механізм закінчення реєстрації. Даний механізм полягає у тому, щоб при натисканні кнопки «GO» повернутися до головного екрану. Також потрібно розуміти, що відправка даних на сервер і отримування відповіді також не блискавичне.

Рішення полягає у тому, щоб забезпечити коректний перехід на головний екран коли користувач сам встиг натиснути на стрілку назад, або

коли відповідь повернулася, то анулювати back stack і повернутися на головний екран.

```
ionnomirai
fun signUpWithEmail(name: String, email: String, password: String, status: DefinitionOfStatus? = null) {
    if (email.isNotEmpty() && password.isNotEmpty() && name.isNotEmpty()) {
        myAuth
            .createUserWithEmailAndPassword(email, password)
            .addOnCompleteListener { task ->
                if (task.isSuccessful) {
                    /* if it is ok, then send verification to user */
                    sendEmailVerification(task.result?.user!!)
                    status?.onSuccess()
                    //activity.mainVM.setData(task.result?.user!!.email!!)
                    activity.mainVM.setData(name)
                    dbManager.setPrimaryData(myAuth.currentUser, UserPersonal(name = name, email = email))
                } else {
                    /* Get the current error */
                    Log.d(MY_TAG,  msg: "Global Exception: ${task.exception}")
                    AuxillaryFunctions.showSnackBar("Registration failed. Please check your internet conne
                    /* ... */
                }
            }
    } else {
        AuxillaryFunctions.showSnackBar("You have not filled in all fields.", activity)
    }
}
```

Рисунок 3.2.1. Механізм реєстрації користувача

Через те, що даний механізм буде використовуватися у декількох частинах додатка, було прийнято рішення зробити його у вигляді інтерфейсу. Реалізація даного інтерфейсу, тобто повернення об’єкту з готовим функціоналом, була винесена у окремий об’єкт Singleton, де ми і подальше будем створювати додаткові функції, для інших частин додатку.

Реалізація даного інтерфейсу виглядає наступним чином (рис. 3.2.2): для того щоб скористатися даною функцією створення об’єкту інтерфейсу, нам необхідно передати той фрагмент на якому користувач зараз знаходиться, напрям куди потрібно перейти та контейнер, у якому знаходиться даний фрагмент. Через те, що для даного проекту ми використали лише один такий контейнер, то можем вказати його за замовчуванням.

У даному інтерфейсі розташована лише одна функція для реалізації – onSuccess. Усю логіку даної функції ми розташували всередині блока try/catch. Так, якщо користувач натисне кнопку назад до автоматичного закриття

екрану, то коли дана функція буде відпрацьовувати, ми отримаємо виняток та зловимо його – робота додатка продовжиться правильно.

```
ionnomirai
fun defaultDefinitionOfStatusInterface(
    frag: Fragment,
    direction: SimplePopDirections,
    @IdRes fragContainer: Int = R.id.fcv_main): DefinitionOfStatus{
    return object : DefinitionOfStatus{
        override fun onSuccess() {
            try {
                (frag.activity as MainActivity).doWhenStartOrLogin()
                val curFrag = frag.parentFragmentManager.findFragmentById(fragContainer)
                when(direction){
                    SimplePopDirections.TOP_DESTINATION -> curFrag
                        ?.findNavController()?.popBackStack(R.id.homeFragment, inclusive: false)
                    SimplePopDirections.PREVIOUS_DESTINATION -> curFrag
                        ?.findNavController()?.popBackStack()
                }
            } catch (e: Exception){
                /* If user close screen earlier than it would auto*/
                Log.d(my_tag, msg: "LoginFrag exception: $e")
            }
        }
    }
}
```

Рисунок 3.2.2. Реалізація інтерфейсу повернення до головного екрану, при виконанні операцій з *Firebase*

Далі якщо функція почала відпрацьовувати, то вона визначає activity, напрям (в залежності від enum (перелік), який складається усього з двух значень) і переходить або на головний екран, або якщо обрано PREVIOUS_DESTINATION, то на попередній екран.

Далі розглянемо наступний, часто використовуваний механізм відображення інформації (списків страв) – RecyclerView. Для того щоб даний елемент працював, йому необхідно надати адаптер (інакше кажучи керуючий елемент).

У додатку, є дві різні реалізації даного адаптера. Тут також зазначимо, що взагалі для кожного випадка розроблюється окрема реалізація, але ми уніфікували даний механізм та зменшили кількість адаптерів до двох: для переліку страв взагалі і переліку страв у кошику.

```
20  class AdapterShowItems(
21      val itemMoveDirections: ItemMoveDirections
22  ): RecyclerView.Adapter<AdapterShowItems.ViewHolderMy>() {
23
24      private val my_tag = "AdapterShowItemsTag"
25      private var oldDishList = emptyList<Dish>()
26
        ionnomirai
27      override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolderMy {
28          val bindingOuter = ItemChooseRvBinding.inflate(LayoutInflater.from(parent.context), parent,
29          return ViewHolderMy(bindingOuter, itemMoveDirections)
30      }
31
        ionnomirai
32      override fun onBindViewHolder(holder: ViewHolderMy, position: Int) {
33          holder.setData(oldDishList[position])
34      }
35
        ionnomirai
36      override fun getItemCount(): Int {
37          return oldDishList.size
38      }
39
        ionnomirai
40      fun setNewData(newDishList: List<Dish>){...}
46
        ionnomirai *
47      class ViewHolderMy(val bindingInner: ItemChooseRvBinding,
48                         val itemMoveDirectionsInner: ItemMoveDirections)
49          : RecyclerView.ViewHolder(bindingInner.root) {
50          private val my_tag_inner = "ItemTag"
            ionnomirai
51          fun setData(data: Dish){...}
121
            ionnomirai
122         private fun setTintIcons(ic: IconsOnItem, isPress: Boolean){...}
133     }
```

Рисунок 3.2.3 Приклад програмний коду, для *RecyclerView adapter*

Загалом, кожний адаптер повинен мати у собі такі елементи (рис. 3.2.3):

- об’єкт спадкоємець класу ViewHolder– даний клас, який унаслідується від ViewHolder, ми повинні створити власноруч. Для кожного елементу списку, буде створений свій об’єкт даного класу. Відповідно до цього, кожен такий об’єкт може керувати відображенням даного елементу списку і налаштовувати його. Для графічної візуалізації ми розробили і продемонстрували шаблон у підрозділі 3.1 для відображення даних списків;

- функцію onCreateViewHolder() – на початку роботи, адаптер викликає дану функцію для створення об’єктів ViewHolder;

- onBindViewHolder() – викликається для налаштування об’єкта після того, як він стає у області доступній для перегляду користувачем;

- getItemCount() – викликається для того, щоб адаптер знав розмір списку даних з яким йому прийдеться працювати та для розрахунку індексів об’єктів.

Даний механізм дуже зручний тим, що він підтримує lazy варіант відображення. Даний варіант дозволяє відобразити на екрані лише ті дані, що необхідні, а усі інші додавати лише за необхідністю.

Наприклад, розглянемо варіант коли користувач захоче вручну знайти якусь конкретну страву у «пошуку» серед усіх страв. Допустимо, що кафе має усього 100 страв. За звичайним механізмом відображення, усі 100 елементів були б завантажені у пам’ять, що негативно відобразилось би на швидкодії.

RecyclerView пропонує інший варіант. Він створить близько 10 об’єктів ViewHolder (дана кількість залежить від багатьох факторів, наприклад: складність макету, розмір екрану тощо), потім заповнить певну кількість таких об’єктів даними, а інші переведе у схований пул. Коли користувач буде прокручувати екран, ті страви що зникають – очищаються, і чисті ViewHolder додаються до схованого пулу. З цього ж пулу дістаються інші ViewHolder для завантаження нових страв. Відбувається кругообіг об’єктів, і нові об’єкти створюються лише якщо дані об’єкти не справляються з напливом інформації.

Через те що ми отримуємо інформацію з сервера, то при відсутності інтернету ми не зможемо її отримувати. Для цього є сенс створити локальну базу даних, де ми і будемо зберігати дані і звідки будемо надавати інформацію до RecyclerView.

Для створення такої бази даних, буде використовуватися бібліотека Room. Для створення бази даних, дана бібліотека використовує data class як макет таблиці (рис. 3.2.4).

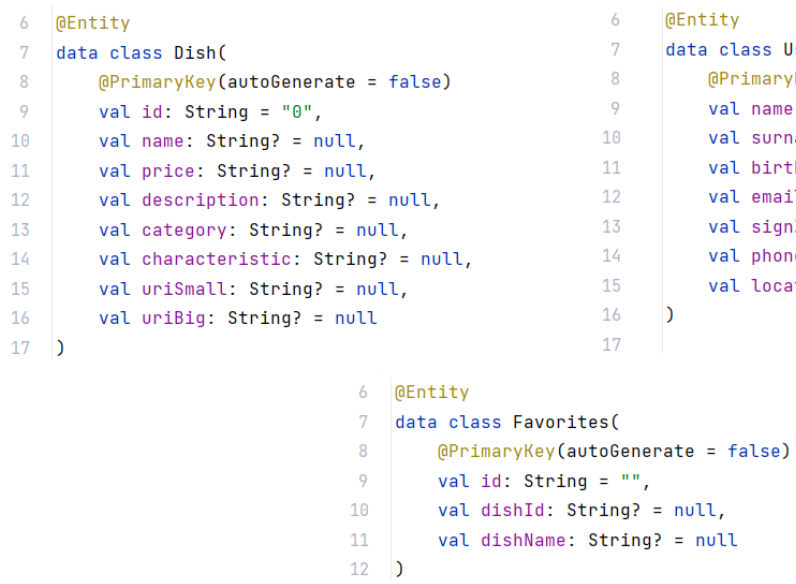


Рисунок 3.2.4 Деякі *Data class* що використовуються як *Entity*

Ми відмовляємося від використання авто генерації Primary Key через те, що усі дані ми отримуємо з сервера, де ці дані вже є. А ті дані що ми записуємо, використовують серверні дані. Таким чином умикається конфлікт. Також звернемо увагу, що для UserPersonal відсутній звичний Primary Key. Даний крок був створений через те, що для кожного користувача може бути лише один варіант облікового запису і налаштувань, збережений у базі даних – це його власний обліковий запис. Якщо користувач виходить з облікового запису, усі таблиці бази даних, що містять інформацію про користувача, також видаляються з пристрою. Вони будуть завантажені наступного разу, коли користувач увійде у свій обліковий запис.

Також зазначимо, що для реалізації на вподоби PrimaryKey, для бази даних на сервері використовується UID. Це унікальне значення яке присвоюється користувачу на момент створення облікового запису і більше не змінюється. Щоб не втручатися у приватну інформацію користувача, ми зробили додатковий рівень вкладеності (рис. 3.2.5). Тобто спочатку йде бар'єр у вигляді UID, а вже потім вся інформація по користувачу. На рисунку 3.2.5, даний бар'єр виглядає як наступний рівень після users і представляє собою набір символів у верхньому регістрі, латиницею.

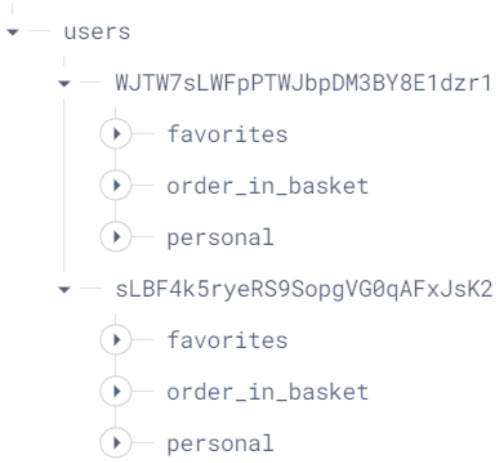


Рисунок 3.2.5 Приклад використання *UID* для збереження і відокремлення користувачів

```
@Dao
interface KindCafeDao {

    /*-----------------------------------Dish-----------------------------------*/
    ionnomirai
    @Upsert
    suspend fun insertDish(dish: List<Dish>)

    ionnomirai
    @Query("Select * FROM dish")
    fun getAllDishes(): Flow<List<Dish>>

    ionnomirai
    @Query("SELECT * FROM dish WHERE category = :categoryCur")
    fun getDishesByCategory(categoryCur: String): Flow<List<Dish>>

    ionnomirai
    @Query("SELECT * FROM dish WHERE id = :id AND name = :name")
    fun getDish(id: String, name: String): Flow<Dish>


    /*-----------------------------------Favorite-----------------------------------
    ionnomirai
    @Upsert
    suspend fun insertFavorites(favorites: Favorites)

    ionnomirai
    @Query("Select * FROM favorites")
    fun getAllFavorites(): Flow<List<Favorites>>

    ionnomirai
    @Query("SELECT * FROM dish WHERE id = :id AND name = :name")
    suspend fun getDishSimple(id: String, name: String): Dish
    ionnomirai
    @Delete
    suspend fun deleteFavDish(fav: Favorites)
```

Рисунок 3.2.6 Приклад інтерфейсу *DAO*, використаний у роботі додатка

Розглянемо, як можна взаємодіяти з базою даних. Для цього необхідно створити інтерфейс помічений анотацією @DAO, який буде містити перелік функцій з анотаціями, що містять або команду за замовчування (наприклад @INSERT, @DELETE тощо), або користувацьку інструкцію зрозумілу для SQLite (рис. 3.2.6.).

Далі, щоб працювати з базою даних необхідно створити екземпляр бази даних. Найкраще, якщо це буде Singleton. Для цього нам необхідно створити клас repository, який буде відповідати за створення екземпляра та зв'язку з функціями DAO. Це можна розглядати як допоміжний, проміжний шар між базою даних та місцем це буде використовуватися.

Singleton repository з базою даних створюється за допомогою використання companion object (це схоже на статичні функції з Java) (рис. 3.2.7).

```
ionnomirai
class KindCafeRepository private constructor(
    context: Context
){
  • • •
    ionnomirai
    companion object{
        private var INSTANCE: KindCafeRepository? = null

        ionnomirai
        fun initialize(context: Context){
            if(INSTANCE == null){
                INSTANCE = KindCafeRepository(context)
            }
        }

        ionnomirai
        fun get(): KindCafeRepository{
            return INSTANCE ?: throw IllegalStateException("The local KindCafeDatabase is not initialized.")
        }
    }
}
```

Рисунок 3.2.7 Логіка функцій для створення або отримання екземпляру локальної бази даних

Створення repository буде ініційоване до запуску основної Activity. Экземпляр даної бази даних буде створено одночасно з repository. Функції для взаємодії з базою даних можуть бути отримані зі зверненням до Singleton.

Оскільки ми будемо робити одну інформаційну ViewModel для зв’язку між усіма фрагментами, то раціонально виконувати запити до бази даних через дану модель. Тобто шлях буде такий: Фрагмент – ViewModel – Repository – Database.

Окрім зв’язку з базою даних, дана модель надає місце для розташування даних, яке буде їх зберігати незалежно від статусу життєвого циклу фрагменту. Тобто, якщо фрагмент зробив зміни в даних збережених у моделі, то навіть якщо він буде повністю знищений, ці дані будуть збережені.

Таким чином, дані що потребують різні фрагменти будуть завжди актуальні для усіх фрагментів.

Дані отримані з локальної бази даних або з сервера, будуть збережені у списках розташованих як StateFlow, що підтримують можливість постійного спостереження. Дані списки будуть розташовані у ViewModel. За допомогою StateFlow, при зміні стану змінної (наприклад списку), ті елементи що підписані на даний елемент будуть інформовані про дані зміни і повинні виконати дії що ми запрограмуємо.

Наведемо приклад отримання даних з бази даних стосовно усіх страв кафе (рис. 3.2.8).

Спочатку нам необхідно створити сховища для даних отриманих з бази даних. Для цього ми створюємо необхідні змінні StateFlow. Хочемо звернути увагу, що ми будемо зберігати спочатку усі страв, потім будемо в іншу змінну зберігати дані за певною вибіркою тощо. На наш погляд, краще одразу зробити необхідні вибірки з бази даних, які будуть збережені як об’єкти з текстовими даними, а ніж при кожній необхідності фільтрувати або робити нові вибірки з усіх страв, коли нам потрібна певна категорія. Наприклад, дані категорії нам будуть потрібні кожного разу, коли користувач захоче перейти до перегляду певної категорії.

Рисунок 3.2.8 *StateFlow* для збереження даних з локальної бази даних

Для кожного набору збережених даних ми робимо дві змінні. У одну, ми будемо зберігати дані і помітимо її як private – тобто, ніхто окрім даного класу не зможе до неї звертатися і змінювати. Друга буде виконувати роль доступу (дані read-only, тобто тільки для читання). За нею ми будемо отримувати доступ до цих даних, але ніяк не зможемо змінювати ці дані.

Якщо зазирнути глибше, то можна сказати, що хоч змінних і дві, але споживання даних у них не двійне. Справа у тому, що наприклад _allDishes – зберігає дані про всі страви, а allDishes є обчислюваною властивістю, тобто не має свого власного поля (комірки) у пам’яті. Обчислювальне вона є через те, що при кожному звертанні, воно «обчислює шлях» до змінної allDishes і повертає її, але як об’єкт доступний тільки для читання. Таким чином, ми забезпечуємо безпеку для даних від несанкціонованого чи необережного втручання у набір даних [18].

Для того що записати дані, нам потрібно створити функцію яка буде спостерігати за даними у базі даних, і при їх зміні записувати нові, актуальні дані до наших StateFlow (рис. 3.2.8). Дана особливість спеціально розроблена

і втілена у Room, та при роботі з StateFlow дозволяє не турбуватися про актуальність даних збережених з бази даних.

Також зазначимо що запуск даного спостереження необхідно робити або з контексту даної моделі, з фрагмента або Activity. Для даного додатка було вирішено розпочинати спостереження за базою даних у Activity при запуску додатка.

Дані функції позначені як suspend, що означає, що їх необхідно запускати не з основного потоку, а з додаткових за допомогою Coroutines. Це зроблено з тією метою, що якщо дані будуть довго надходити або зберігатися, то дані процеси не повинні заважити користувачу керувати додатком. У іншому випадку, екран додатку буде знаходиться у неактивному стані до тих пір, поки не будуть виконані операції з базою даних – що неприпустимо.

```
110     /*---------------------------------FUN dishes-----------------------------------------------*/
        ionnomirai
111     suspend fun getAllDishes(){
112         repository.getAllDishes().collect{ it: List<Dish>
113             _allDishes.value = it
114         }
115     }
116
        ionnomirai
117     suspend fun getDishesByCategory(category: Categories) {
118         when (category) {
119             Categories.SparklingDrinks -> repository.getDishByCategory(category).collect { it: List<Dish>
120                 _sparklingDrinks.value = it
121             }
122             Categories.Cakes -> repository.getDishByCategory(category).collect{ it: List<Dish>
123                 _cakes.value = it
124             }
125             • • •
126
127         }
128     }
129
```

Рисунок 3.2.9 Функції для отримання даних з локальної бази даних

Для отримання даних, початку спостерігання за базою даних, необхідно визвати наведені вище функції у Coroutine (рис. 3.2.9). Коли дана операція буде повністю виконана, то дані будуть збережені і готові для використання.

```
104        // retrieve data from Local DB
105        viewLifecycleOwner.lifecycleScope.launch { this: CoroutineScope
106            mainVM.getDishesByCategory(navArgs.category)
107        }
```

Рисунок 3.2.10 Отримання даних з локальної бази даних

За правилами, після виконання даних операцій і за відсутністю необхідності тримати у пам'яті даний Coroutine, його потрібно завершати. Проте, якщо Coroutine створений і прив'язаний до життєвого циклу фрагменту, то при видаленні його, також буде завершено і цей Coroutine (рис. 3.2.10).

Для того щоб почати спостереження за даними що ми зберегли у StateFlow, необхідно визвати у даній змінній функцію collect і у її тіло передати логіку яка буде виконуватися кожного разу, коли дані будуть змінені. Дане спостереження також необхідно проводити лише у Coroutine, щоб не навантажувати основний потік і не блокувати його. Наприклад, на рисунку 3.2.11 показано, що є деяка функція, яка потребує в себе параметр типу Categories (це enum з переліком основних категорій страв у кафе). Наприклад, коли користувач натискає на певну категорію починається перехід на інший фрагмент.

```
           ionnomirai
173        private suspend fun checkCategoryAndAction(categoryCur: Categories) {
174            when (categoryCur) {
175                Categories.SparklingDrinks -> mainVM.sparklingDrinks.collect { it: List<Dish>
176                    myAdapter.setNewData(it)
177                    Log.d(my_tag,  msg: "read when start: $it")
178                    mainVM.currentLocation = Locations.SHOW_SPARKLING_DRINKS.nameL
179                }
180            • • •
```

Рисунок 3.2.11 Спостерігання за даними отриманими з локальної бази даних

При старті фрагмента визивається дана функція і в залежності від того, яка категорія була передана, спостереження за тими даними і буде почато. В даному випадку, при кожній зміні даних стосовно газованих напоїв, буде

оновлюватися і адаптер, що керує відображенням на екрані даних що до страв. Зміни в даному випадку можуть бути я вигляді зміни ціни, опису або видалення страви ініційоване на сервері. При таких обставинах, зміни будуть відбуватися за «розумним» підходом із використанням DiffUtil, який буде за певним алгоритмом обраховувати зміни і оновлювати лише ті дані що потрібно. Даний випадок, гарно підходить для застосування даного інструмента.

Окрім цього, у описуваній функції додатково буде оновлено відстеження розташуванням користувача на фрагменті. Це потрібно для реалізацій інших функцій, пов'язаних з необхідністю чітко знати на якому фрагменті зараз знаходиться користувач.

Даний додаток отримує і записує дані не тільки до локальної бази даних, але й до бази даних серверу. Опишімо процеси зчитування і відправки даних, що використовуються у даному проекті.

Для зчитування даних перш за все потрібно встановити зв'язок із сервером, це робиться за допомогою API що надає Firebase (рис. 3.2.12). Далі потрібно вказати вузол, від якого і будемо розраховувати шлях до потрібних даних.

Після цього необхідно ініціювати зчитування усіх даних, починаючи з даного вузлу за допомогою listener addListenerForSingleValueEvent. Далі, нам необхідно пройти по всім даним і трансформувати їх до оговорених Data class. В даному випадку, ми додатково переходим безпосередньо до вузлів з категоріями, зчитуємо дані звідти і записуємо до нашого списку.

```
ionnomirai *
class DbManager {
    private val myTag = "DbManagerMy"
    private val myDatabase = Firebase.database.getReference( path: "dishes")
    private val myDatabaseUser = Firebase.database.getReference( path: "users")

    /*----------------------------------General-----------------------------------------*/
    ionnomirai *
    fun readAllDishDataFromDb(callbackRead: ReadAllData) {
        val result: MutableList<Dish> = mutableListOf()

        myDatabase.addListenerForSingleValueEvent(object : ValueEventListener {

            override fun onDataChange(snapshot: DataSnapshot) {
                // We will enter into each category
                for (i in Categories.entries) {
                    // move to category. Every iteration outer cycle, will be different category
                    val deeperSnapshot = snapshot.child(i.name).children

                    // collect each dish
                    for (item in deeperSnapshot) {
                        item.getValue(Dish::class.java)?.let { result.add(it) }
                    }
                }

                callbackRead.readAll(result)
                Log.d(myTag, result.toString())
            }

            override fun onCancelled(error: DatabaseError) {}
        })
    }
```

Рисунок 3.2.12 Зчитування даних з хмарної бази даних

Після цього нам необхідно записати ці дані до локальної бази даних. Для цього ми використовуємо інтерфейс, об’єкт якого буде створений у інших класах. Ціль даного інтерфейсу, за нашою задумкою, це записувати дані до локальної бази даних, та що головніше, вистроїти процеси зчитування, запису і подальшої дії додатку у зрозумілу лінію. Справа у тому, що всі API що пропонує Firebase, продумані до тонкощів і пропонують асинхронне виконання без ручного налаштування потоків. І це гарно, але робить відстеження закінчення операції доволі проблемним. Саме це і потрібно зробити для реалізації логіки додатку. Таким чином, ми встановлюємо ручні тригери в інтерфейсах, щоб зрозуміти, коли операція із сервером завершена.

Операція запису даних до серверу здебільшого схожа з читанням (рис. 3.2.13). Для виконання такої операції необхідно встановити початковий вузол, пройтись по кожному з внутрішніх вузлів і записати дані у необхідному місці.

Оскільки подібні операції може виконувати тільки зареєстрований користувач, то нам необхідно перевірити чи даний користувач увійшов у свій обліковий запис, чи ні, і продовжувати лише у тому випадку якщо увійшов.

Користувач може змінювати лише ті дані, що безпосередньо стосуються даного користувача і не чіпають інших. Для досягнення даної цілі, ми входимо до вузлу названого за унікальним для даного користувача ідентифікатором – UID, це гарантує безпеку даних. Даний вузол створюється у той момент коли користувач зареєстрував свій обліковий запис.

```
26    private val myDatabaseUser = Firebase.database.getReference( path: "users")

150   /*----------------------------------OrderBasket-----------------------
151
      ionnomirai
152   fun setOrderBasketToRDB(user: FirebaseUser?, data: List<OrderItem>){
153       user?.let {u ->
154           data.forEach {item ->
155               myDatabaseUser
156                   .child(u.uid)
157                   .child(CategoriesInUsers.ORDER_BASKET.cName)
158                   .child( pathString: "itemOrder${item.id}")
159                   .setValue(item)
160           }
161       }
162   }
```

Рисунок 3.2.13 Запис даних до хмарної бази даних

Для кожного типу інформації що необхідно записати, ми створили окрему функцію. Це гарантує що буде додаватись або перезаписуватись не увесь блок даних про користувача, а тільки окремий блок.

Для даного прикладу, функція зображена на рисунку 3.2.13, буде визиватися у тому випадку, коли користувач хоче додати щось до свого кошику. У користувача є декілька блоків що він зможе змінювати, це: кошик,

персональні дані, улюблене тощо. У даному випадку ми працюємо з кошиком. У хмарній базі даних буде створений вузол, ім’я якого буде розраховуватися як: «itemOrder» + «item.id». Таким чином, воно буде унікальним з одного боку, і легко знаходися з іншого боку. У даний вузол ми розташовуємо інформацію про страву. Якщо користувач захоче додати іншу страву, то буде створений інший блок для страви за формулою наведеною вище.

3.3. Демонстрація роботи додатку за одним з основних сценаріїв

У якості результату створення додатку для кафе, продемонструємо його роботу для виконання одного з основних сценаріїв. Оговорений сценарій складається з таких кроків: реєстрація, навігація по стравах, додавання до улюбленого, до кошику, налаштування, відправка результату до серверу. Також додатково до цього змінимо персональну інформацію про користувача.

Почнемо з обов’язкового кроку – це реєстрація користувача (створення облікового запису) (рис. 3.3.1). Для цього потрібно відкрити додаток, провести пальцем справа наліво, та обрати пункт «Register». Поки користувач не увійде у обліковий запис, інші кнопки у цьому меню не працюють.

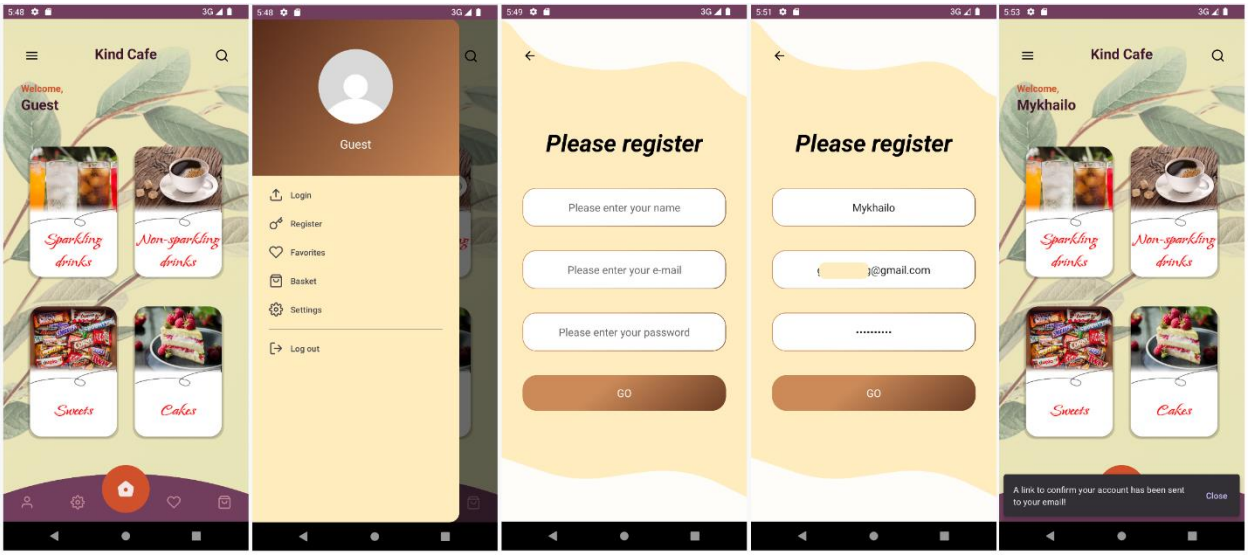


Рис 3.3.1 Процес реєстрації користувача у додатку

Далі, потрібно ввести ім’я, пошту, та пароль. Після цього, якщо дані введені вірно, то користувача перенаправить до головного екрану, якби він вже увійшов до облікового запису. Також на його пошту буде направлений лист з додатковою верифікацією. Але на даний момент, у даному додатку не використовується додаткова перевірка даних.

Наступним кроком буде виступати пошук страви (рис. 3.3.2). Пошук страви можна виконувати через пошук за назвою у глобальному пошуку чи у конкретній категорії, або через скролінг у певній категорії. Для того щоб перейти до пошуку за назвою, необхідно натиснути на значок пошуку.

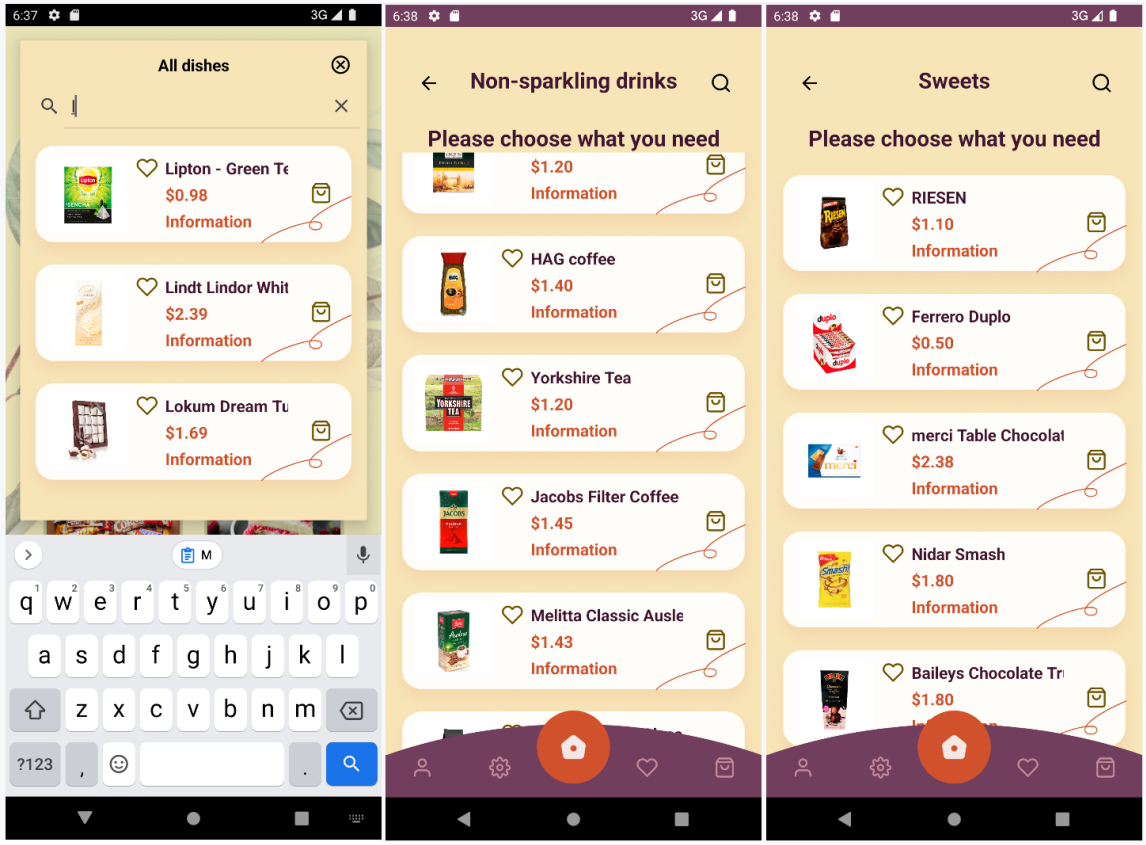


Рис 3.3.2 Процес вибору страви

Страву можна додати до обраного, тоді вони збережеться у окремій категорії для даного користувача. Її можна буде побачити якщо перейти на відповідний екран. Для того щоб додати страву до улюбленого або до кошику, необхідно натиснути на значок «серце» та «кошик» відповідно. Також

відмітимо, що є можливість передивитися більш детальну інформацію про страву, якщо натиснути на текст «Information». На даному екрані також можна додати в кошик або до улюбленого (рис. 3.3.3).

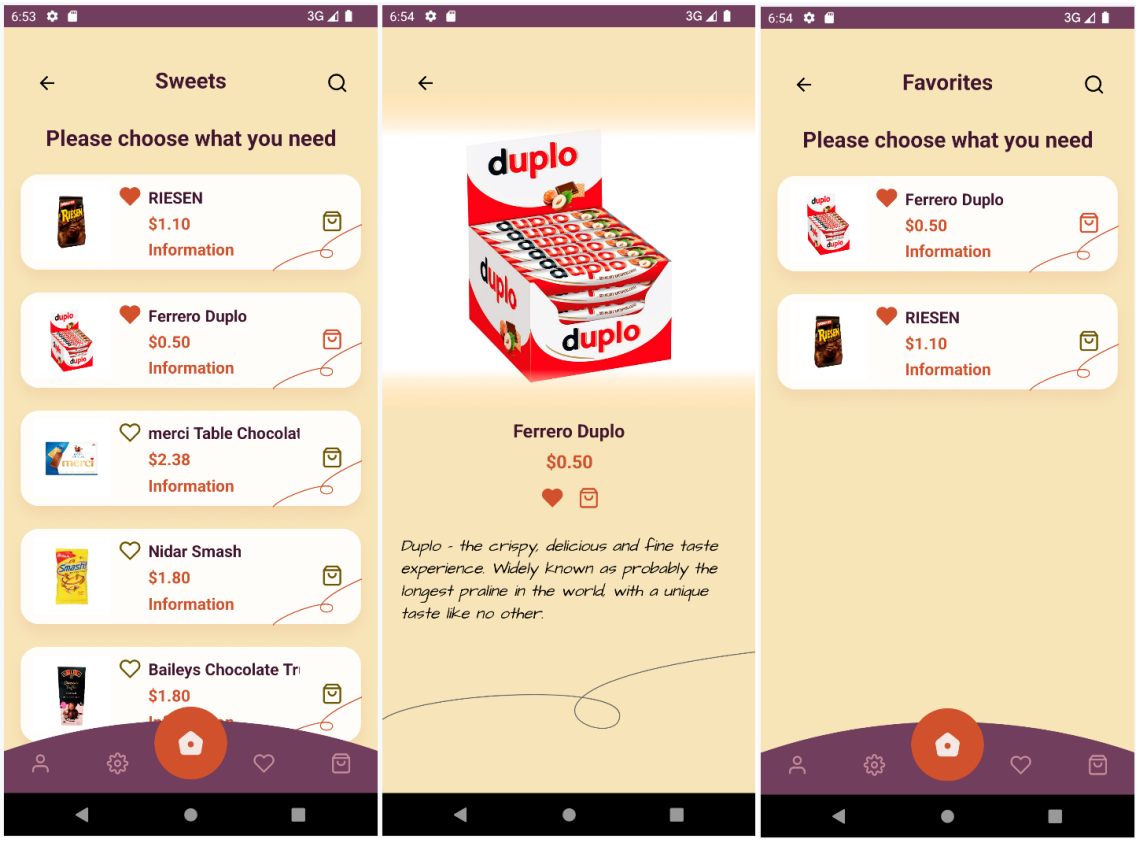


Рис 3.3.3 Процес додавання страви до улюбленого та до кошику

Розглянемо процес взаємодію з замовленням у кошику та у чеку (рис. 3.3.4). Після того, як користувач додав необхідні страви до кошику, він може змінювати їх кількість, розмір та змінювати тип або увімкнути добавки до страви. Якщо він змінив думку і зараз не бажає бачити страву у кошику, то він може видалити її прямо з кошику змахнувши пальцем вправо або вліво. Інший варіант, це знайти страву у пошуку і натиснути знов на значок кошику. Колір даного значка має змінитися з червоного на темно жовтий.

Коли налаштування страв закінчено, користувач має натиснути на кнопку «Make order» та перейти на екран з детальною інформацією про своє замовлення. Передивившись інформацію, перевіривши вартість замовлення та

інші нюанси, він повинен натиснути на кнопку «Send» – замовлення перейде до серверу.

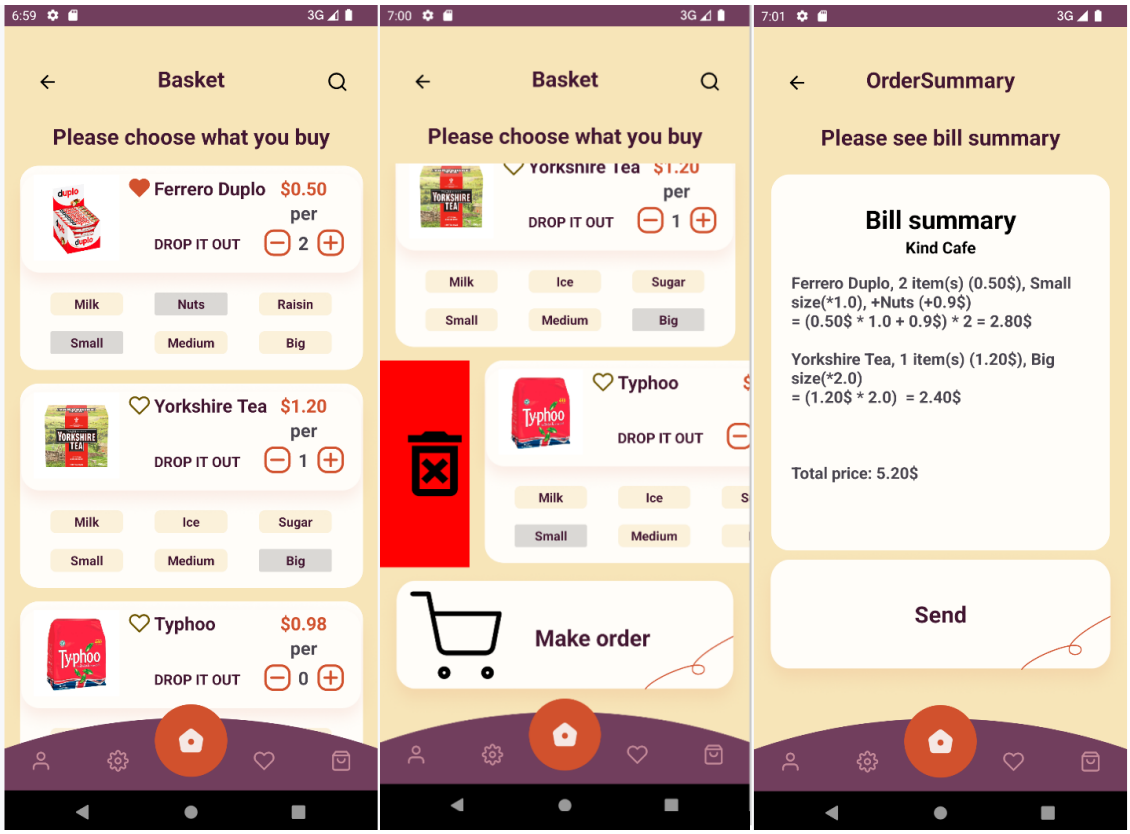


Рис 3.3.4 Процес налаштування страви у кошику

Щоб передивитися даний чек, користувач може знову зайти до кошику і якщо в нього вже є чек, він буде переправлений до нього. Якщо користувач хочу зробити інший заказ або видалити поточний, то після заказу кнопка «Send» буде змінена на кнопку відмови. Натиснувши на дану кнопку, замовлення видалиться і з локальної бази даних, і з бази даних сервера.

Користувач має можливість налаштувати додаткову інформацію про себе, яка може бути використана для надання додаткових послуг, наприклад знижка з нагоди дня народження. Для налаштування особистих даних, можна скористатися боковим або нижнім меню та вибрати пункт «Setting» або знизу натиснути на значки «Person» або «Шестерня». У персональних налаштуваннях можна обрати номер телефона, локацію, П.І.Б., дату народження та знак зодіаку (рис. 3.3.5). Адресу електронної пошти

використаної при реєстрації змінювати не можна, тому біля неї немає індикатора що позначає можливість редагування. Дані зберігаються автоматично, коли користувач покидає даний екран.

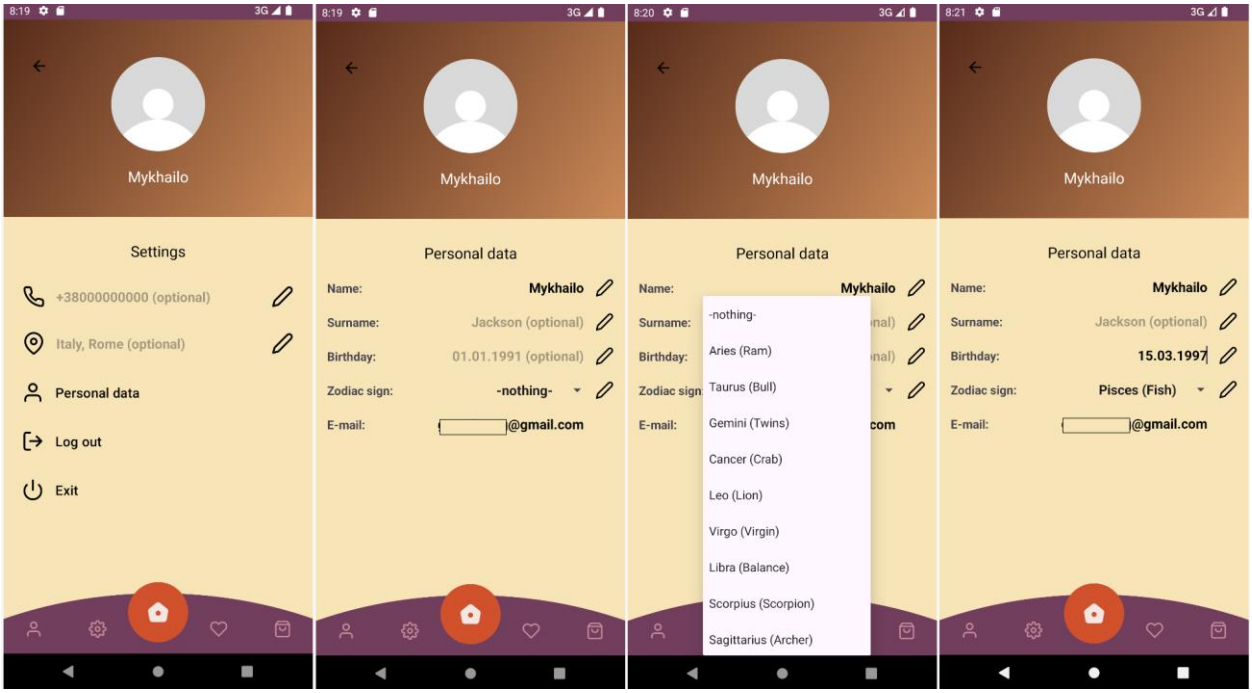


Рис 3.3.5 Процес налаштування особистих даних користувача

Таким чином, у даному розділі ми розглянули особливості графічної моделі додатка, а також основні механізми і структурні особливості додатка. Продемонстрували реальну роботу додатка за описаним вище сценарієм.

ВИСНОВКИ

При виконанні дипломної роботи було виконане дослідження розповсюдження мобільних операційних систем у різних країнах та за різним географічним положенням.

Розглянуті питання актуальності впровадження мобільних технологій у повсякденне життя громадян та окрему користь для бізнесу. Зокрема виконаний огляд роботи кафе і підприємницької діяльності пов’язаної з даною сферою та виділені проблеми, які можуть бути усунуті або полегшені у разі використання мобільних технологій, а саме мобільного додатка у повсякденній роботі кафе.

Проаналізовано переваги і недоліки, що пропонують різні мови програмування або frameworks для створення мобільного додатка. На основі даних що були отримані в результаті даного аналізу, для створення мобільного native додатку для ОС Android, була обрана мова Kotlin,

В результаті виконання дипломної роботи був створений мобільний додаток для кафе, який дозволяє користувачу створювати власний обліковий запис, передивлятися меню кафе, додавати страви що сподобались до улюбленого або до кошику. Існує можливість виконати пошук за назвою страви. Користувач може налаштовувати за власними уподобаннями страви у кошику, а саме: кількість, розмір, тип замовленого елементу меню або добавки до нього. Є можливість видалити елемент з кошику. Реалізована можливість додатково передивитись віртуальний чек і тільки після цього надсилати його до кафе. Усі дані щодо страв у улюбленому, кошику, чеці тощо, автоматично зберігаються для облікового запису користувача і можуть бути відкриті навіть без Інтернету. Також є можливість налаштувати дані у профілю клієнта.

Для демонстрації функціонування додатку, був розроблений і показаний сценарій одного із основних варіантів використовування додатку. Даний мобільний додаток успішно виконав поставлені перед ним цілі сценарію.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Brilworks Software, ReactNative vs Kotlin [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@Brilworks/react-native-vs-kotlin-ba9dbed2ecee
2. Comparison Kotlin to Java [Електронний ресурс] – Режим доступу до ресурсу: https://kotlinlang.org/docs/comparison-to-java.html
3. Forrester Alex, Boudjnah Eran, Dumbravan Alexandru – How to Build Android Apps with Kotlin, Second Edition – Birmengham, UK: Packt Publishing Ltd, 2023. – 704 с.
4. Global Mobile eCommerce Statistics [Електронний ресурс] – Режим доступу до ресурсу: https://www.merchantsavvy.co.uk/mobile-ecommerce-statistics/
5. Gonda Victoria, Android Accessibility by Tutorials, Second Edition – Virginia, USA: Razeware, 2022. – 217 с.
6. Griffths Dawn, Griffiths David, Head First, Android Development, Third Edition – USA: O'Reilly, 2023. – 905 с.
7. I/O 2019 [Електронний ресурс] – Режим доступу до ресурсу: https://io.google/2019/
8. Kumar Pankaj, Building Android Projects with Kotlin – London, UK: BPB, 2023. – 425 с.
9. Laurence Pierre-Olivar, Hinchman-Dominguez Amanda, Programming Android with Kotlin, Achieving Structured Concurrency with Coroutines, First Edition – USA: O'Reilly, 2024. – 338 с.
10. Lin Guo, The First Line of Code, Android Programming with Kotlin – Suzhou, Jiangsu, China: Posts & Telecom Press, 2022. – 714 с.
11. Mobile touches [Електронний ресурс] – Режим доступу до ресурсу: https://dscout.com/people-nerds/mobile-touches
12. Mobile vs desktop usage [Електронний ресурс] – Режим доступу до ресурсу: https://research.com/software/mobile-vs-desktop-usage

13. Native and cross-platform app development: how to choose? [Електронний ресурс] – Режим доступу до ресурсу: https://kotlinlang.org/docs/native-and-cross-platform.html

14. Neuroscientists Learn Why Some People Like Surprises [Електронний ресурс] – Режим доступу до ресурсу: https://www.scientificamerican.com/article/neuroscientists-learn-why/#

15. Sedunov Aleksei, Kotlin In-Depth, Second Edition – India: BPB Publications, 2022. – 674 с.

16. Skeen Josh, Greenhalgh David, Bailey Andrew, Kotlin Programming: The Big Nerd Ranch Guide, Second Edition – USA: Pearson Education, 2022. – 560 с.

17. Smartphone [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Smartphone

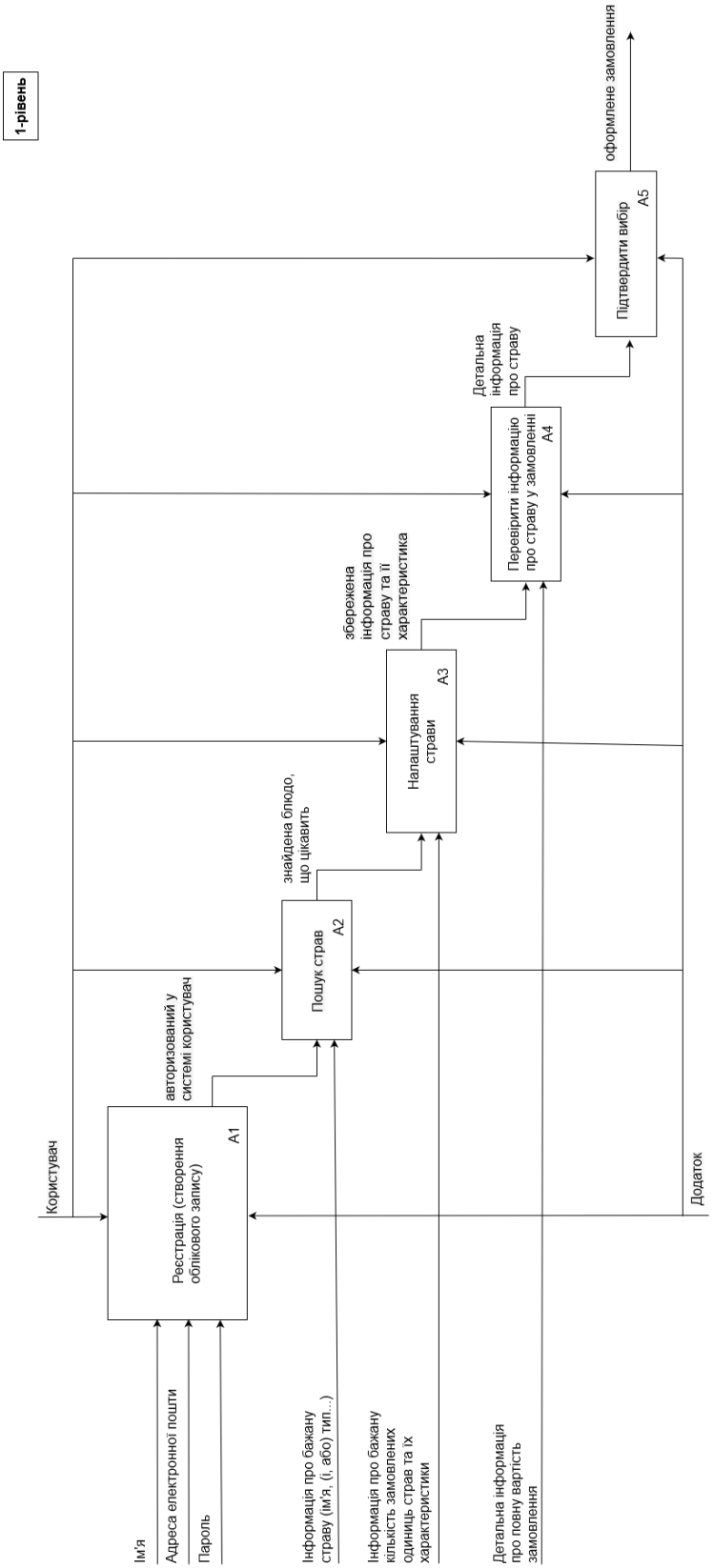
18. Spath Peter, Pro Android with Kotlin, Second Edition – Leipzig, Germany: Apress, 2022. – 881 с.

19. Збільшення кількості користувачів смартфонів у Україні [Електронний ресурс] – Режим доступу до ресурсу: https://ms.detector.media/mediadoslidzhennya/post/21573/2018-08-03-kilkist-korystuvachiv-smartfoniv-v-ukraini-zbilshylasya-do-85-doslidzhennya/

ДОДАТКИ

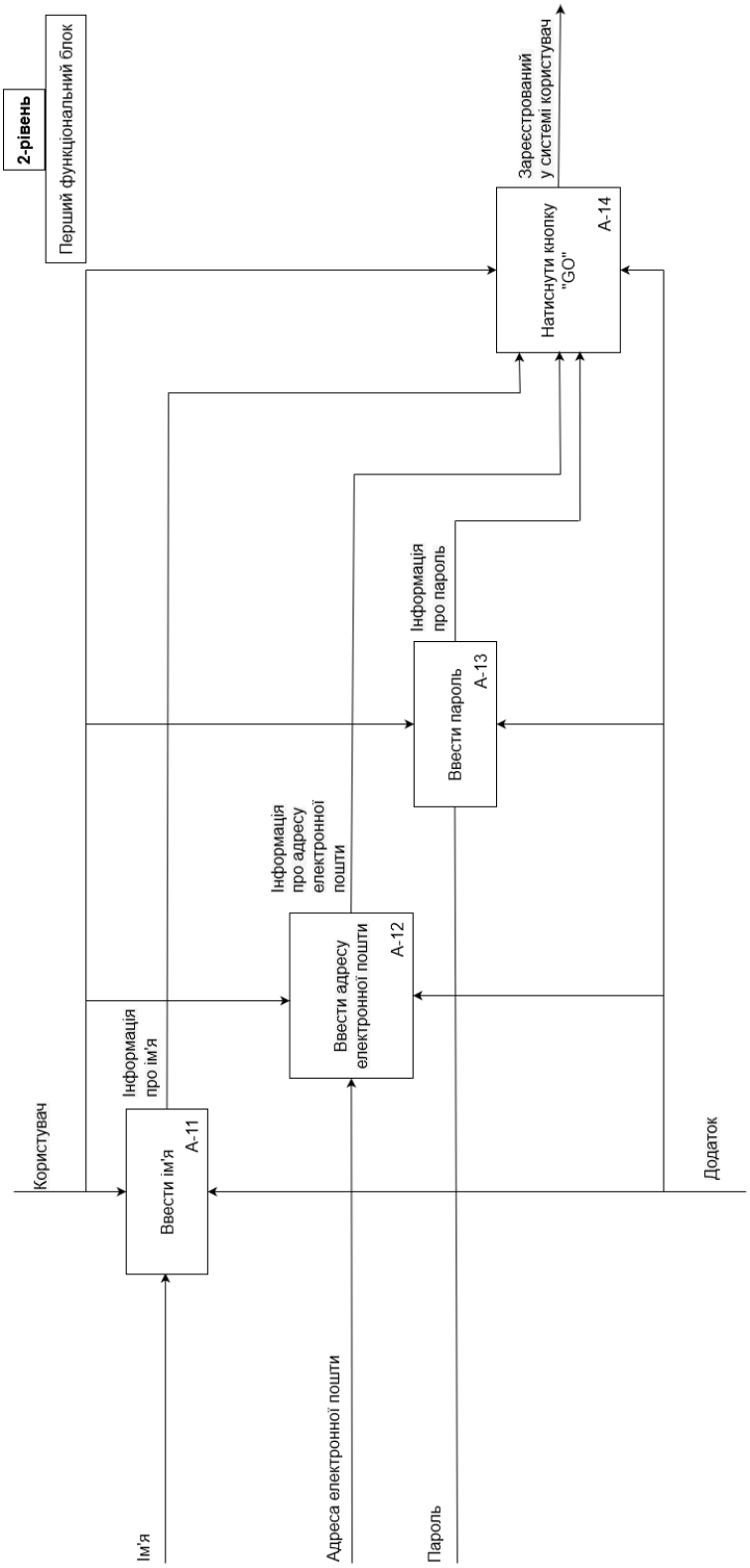
ДОДАТОК А. РІВЕНЬ 1 ФУНКЦІОНАЛЬНОЇ ДІАГРАМИ IDEF0

Рисунок А.1



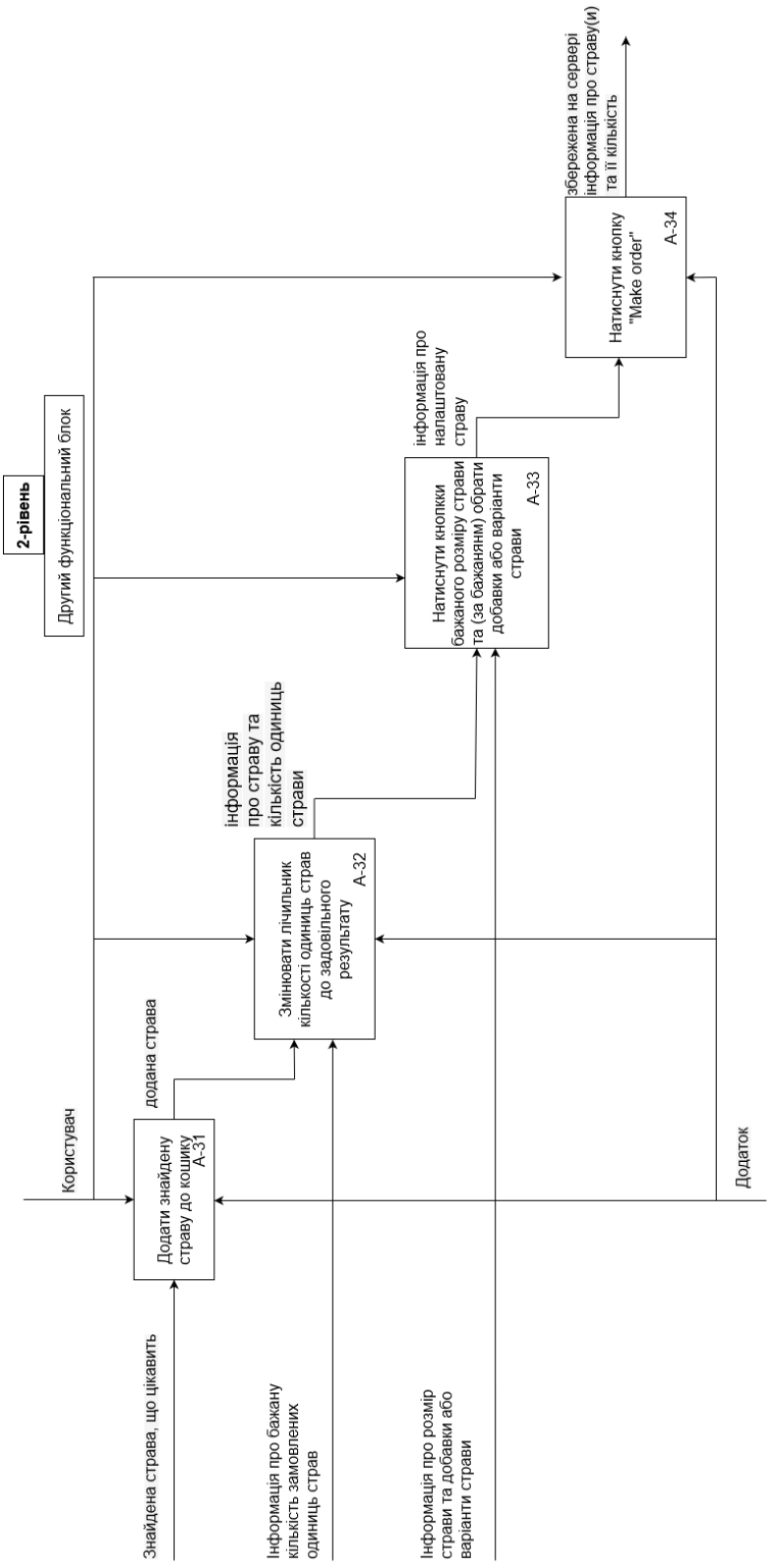
ДОДАТОК В. РІВЕНЬ 2 ФУНКЦІОНАЛЬНОЇ ДІАГРАМИ IDEF0, ДЕКОМПОЗИЦІЯ БЛОКУ А-1

Рисунок В.1



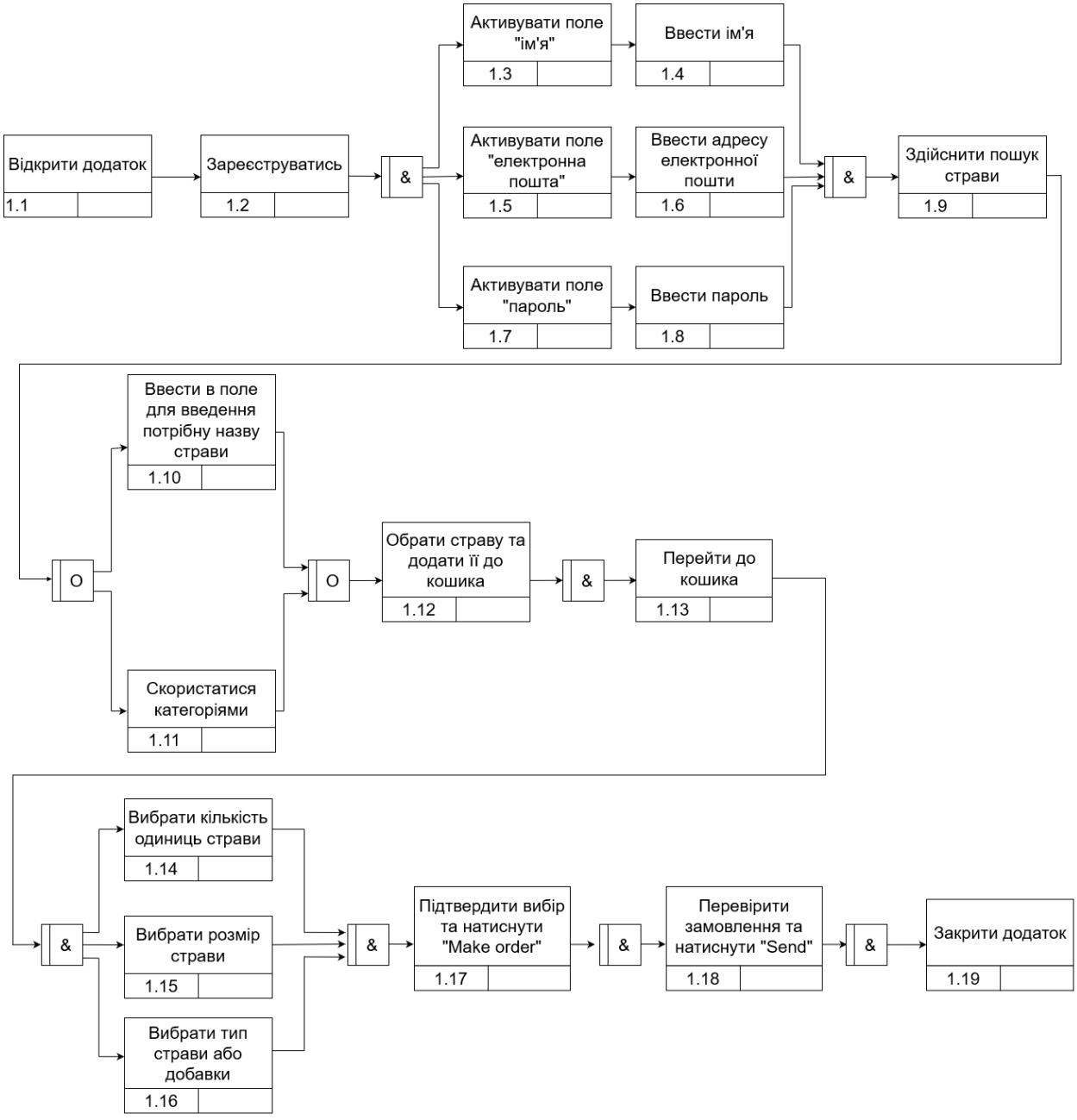
ДОДАТОК С. РІВЕНЬ 2 ФУНКЦІОНАЛЬНОЇ ДІАГРАМИ IDEF0, ДЕКОМПОЗИЦІЯ БЛОКУ А-3

Рисунок С.1



ДОДАТОК D. ДІАГРАМА IDEF3 ДЛЯ ДОДАТКА «KIND CAFE»

Рисунок D.1



ПРОГРАМНИЙ КОД

```
class MainActivity : AppCompatActivity() {

    /*------------------------------------- Properties ------------------
---------------------*/

    lateinit var binding: ActivityMainBinding

    /*  AppBarConfiguration:
    * - determining which drawerLayout to work with;
    * - definition of the "Burger" button;
    * - definition of top level destination;
    * - control of the back button for further transitions. */
    private lateinit var appBarConfiguration: AppBarConfiguration

    /* NavController - This will be the object that keeps track of the
current navigation position
    * among the navGraph. It also changes fragments in NavHostFrament*/
    lateinit var navController: NavController

    private val my_tag = "MainActivityTag"
    private val cacheSize: Long = 2048 * 2048 * 50 //+-209 MB
    private lateinit var picasso: Picasso

    private val accountHelper = AccountHelper(this, R.id.lDrawLayoutMain)

    /* Common viewModel to get data to fragment (for example Home fragment)
*/
    val mainVM: MainViewModel by viewModels()

    private val dbManager = DbManager()
    private val storageManager = StorageManager()

    private val listAllDishes = mutableListOf<Dish>()
    private val listSmallUris = mutableListOf<Dish>()
    private val listBigUris = mutableListOf<Dish>()
    private var listUserInfo = AllUserData()

    private val isDbServerDLDone = MutableStateFlow(false)
    private val isSmallUrisDone = MutableStateFlow(false)
    private val isBigUrisDone = MutableStateFlow(false)
    private val isUsersInfoDone = MutableStateFlow(false)

    private lateinit var pbUpdate: AlertDialog

    /*------------------------------------- Functions -------------------
---------------------*/

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        /* Connecting my custom Action Bar */
        setSupportActionBar(binding.tbMain)

        val navHostFragment =
            supportFragmentManager.findFragmentById(R.id.fcv_main) as
NavHostFragment

        navController = navHostFragment.navController
```

```
        appBarConfiguration = AppBarConfiguration(
            navGraph = navController.graph,
            drawerLayout = binding.lDrawLayoutMain
        )

        //setupNavigationMenu(navController)
        setupActionBar(navController, appBarConfiguration)

        /* When user name changed -- call this */
        mainVM.nameData.observe(this) {
            updateMainUI()
        }

        binding.apply {
            ibHome.setOnClickListener {
                navController.popBackStack(R.id.homeFragment, false)
            }

            ibBag.setOnClickListener {
                if (KindCafeApplication.myAuth.currentUser != null) {
                    Log.d(my_tag, "orderplaced:
${mainVM.orderPlaced.value.toString()}")
                    if(mainVM.orderPlaced.value.isEmpty()){
                        navController.popBackStack(R.id.homeFragment, false)

navController.navigate(R.id.action_homeFragment_to_basketFrag)
                    } else {
                        navController.popBackStack(R.id.homeFragment, false)

navController.navigate(R.id.action_homeFragment_to_orderSummaryFragment)
                    }
                } else {
                    Toast.makeText(this@MainActivity, "Please login",
Toast.LENGTH_SHORT).show()
                }
            }

            ibHeart.setOnClickListener {
                if (KindCafeApplication.myAuth.currentUser != null) {
                    navController.popBackStack(R.id.homeFragment, false)

navController.navigate(R.id.action_homeFragment_to_favoriteFragment)
                } else {
                    Toast.makeText(this@MainActivity, "Please login",
Toast.LENGTH_SHORT).show()
                }

            }

            ibSettings.setOnClickListener {
                if (KindCafeApplication.myAuth.currentUser != null){
                    navController.popBackStack(R.id.homeFragment, false)

navController.navigate(R.id.action_homeFragment_to_settingsGeneralFragment)
                }
            }

            ibProfile.setOnClickListener {
                if (KindCafeApplication.myAuth.currentUser != null){
                    navController.popBackStack(R.id.homeFragment, false)

navController.navigate(R.id.action_homeFragment_to_settingsPersonalFragment)
```

```
                }
            }
        }

        everyOpenHomeSettings()

        movingLogicN2()

        /* Set new cache size for Picasso */
        picasso = Picasso
            .Builder(this)
            .downloader(OkHttp3Downloader(this, cacheSize))
            .build()

        downloadDbWhenStart()

        doWhenStartOrLogin()

        downloadLocalDb()
    }

    /* Custom logic of moving between fragments */
    private fun movingLogicN2() {

        binding.nvLeft.setNavigationItemSelectedListener {
            when (it.itemId) {
                R.id.itemLogin -> {
                    if (!accountHelper.isUserLogin()) {
                        moveTo(R.id.action_homeFragment_to_loginFragment)
                    }
                }

                R.id.itemRegistrationFragment -> {
                    if (!accountHelper.isUserLogin()) {

moveTo(R.id.action_homeFragment_to_registrationFragment)
                    }
                }

                R.id.itemFavorites ->{
                    if (KindCafeApplication.myAuth.currentUser != null){
                        moveTo(R.id.action_homeFragment_to_favoriteFragment)
                    }
                }

                R.id.itemBasket ->{
                    if (KindCafeApplication.myAuth.currentUser != null){
                        moveTo(R.id.action_homeFragment_to_basketFrag)
                    }
                }

                R.id.itemSettings ->{
                    if (KindCafeApplication.myAuth.currentUser != null){

moveTo(R.id.action_homeFragment_to_settingsGeneralFragment)
                    }
                }

                R.id.itemLogout -> { logoutAction() }
            }
            true
        }
    }
```

```
    /* Simple wrap for fragment moving. Needed to reduce code. */
    private fun moveTo(@IdRes idDest: Int, needCloseSideMenu: Boolean = true)
{
        navController.navigate(idDest)
        if (needCloseSideMenu) {
            binding.lDrawLayoutMain.closeDrawer(GravityCompat.START)
        }
    }

    /* Update:
    * - toolbar header - change name;
    * - viewmodels.name -> give this name to fragment Home*/
    fun updateMainUI() {
        binding.apply {
            nvLeft
                .getHeaderView(0)
                .findViewById<TextView>(R.id.tvUserName).text =
mainVM.nameData.value
        }
    }

    /* Perform these settings every time the screen starts up */
    fun everyOpenHomeSettings() {
        binding.tbMain.title = ""
        binding.tvToolbarTitle.text = resources.getString(R.string.home_name)

        accessBottomPart(GeneralAccessTypes.OPEN)
        accessUpperPart(GeneralAccessTypes.OPEN)
    }

    /* Hide or show bottom menu */
    fun accessBottomPart(action: GeneralAccessTypes) {
        binding.apply {
            when (action) {
                GeneralAccessTypes.OPEN -> {
                    clMainBottomMenu.visibility = View.VISIBLE
                    ibHome.visibility = View.VISIBLE
                }

                GeneralAccessTypes.CLOSE -> {
                    clMainBottomMenu.visibility = View.GONE
                    ibHome.visibility = View.GONE
                }
            }
        }
    }

    fun accessUpperPart(action: GeneralAccessTypes) {
        binding.apply {
            when (action) {
                GeneralAccessTypes.OPEN -> {
                    tvToolbarTitle.isVisible = true
                    tbMain.menu.findItem(R.id.itbSearch).isVisible = true
                }

                GeneralAccessTypes.CLOSE -> {
                    tvToolbarTitle.isVisible = false
                    tbMain.menu.findItem(R.id.itbSearch).isVisible = false
                }
            }
        }
    }
```

```
    /* Connecting NavigationView (sliding panel) to toolbar (accessed via
appBarConfiguration). This does the following:
    *  - shows the current location in place title (toolbar);
    *  - shows the back button when we are not in the top destination;
    *  - shows the "burger" button when we are in the top destination. */
    private fun setupActionBar(
        navController: NavController,
        appBarConfiguration: AppBarConfiguration
    ) {
        setupActionBarWithNavController(navController, appBarConfiguration)
    }

    /* Handling back button click */
    override fun onSupportNavigateUp(): Boolean {
        return navController.navigateUp(appBarConfiguration)
    }

    override fun onCreateOptionsMenu(menu: Menu?): Boolean {
        menuInflater.inflate(R.menu.toolbar_menu_general, menu)
        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        if (item.itemId == R.id.itbSearch) {

DialogSearchGeneral(mainVM.currentLocation).show(supportFragmentManager,
null)
        }
        return super.onOptionsItemSelected(item)
    }

    private fun downloadDbWhenStart() {
        /* Get data from RealtimeDB*/
        dbManager.readAllDishDataFromDb(getCallbackReadAllData())

        /* Get Data about small uri */
        mainVM.viewModelScope.launch {
            isDbServerDLDone.collect {
                if (it) {
                    Log.d(my_tag, "downloadDbWhenStart: downloadDbWhenStart")
                    storageManager.readUri(
                        listAllDishes,
                        getUrisBySize(UriSize.Small, this),
                        UriSize.Small
                    )
                }
            }
        }

        /* Get Data about big uri */
        mainVM.viewModelScope.launch {
            isSmallUrisDone.collect {
                if (it) {
                    storageManager.readUri(
                        listSmallUris,
                        getUrisBySize(UriSize.Big, this),
                        UriSize.Big
                    )
                    Log.d(my_tag, "isSmallUrisDone")
                }
            }
        }
```

```
        /* Get Data about big uri */
        mainVM.viewModelScope.launch {
            isBigUrisDone.collect {
                if (it) {
                    mainVM.addDishLocal(listBigUris)
                    Log.d(my_tag, "Home-ViewModel added to local DB added")
                    pbUpdate.dismiss()
                    isDbServerDLDone.value = false
                    isSmallUrisDone.value = false
                    isBigUrisDone.value = false
                    Log.d(my_tag, "Home-ViewModel get to local DB added")
                    cancel()
                }
            }
        }
    }

    private fun getCallbackReadAllData(): ReadAllData {
        return object : ReadAllData {
            override fun readAll(data: List<Dish>) {
                listAllDishes.clear()
                listAllDishes.addAll(data)
                Log.d(my_tag, "Main-ViewModel data added")
                isDbServerDLDone.value = true
                pbUpdate =
ProgressUpdateMain.createProgressDialog(this@MainActivity)
            }
        }
    }

    private fun getUrisBySize(uriSize: UriSize, job: CoroutineScope):
GetUrisCallback {
        return object : GetUrisCallback {
            override fun getUris(newData: List<Dish>) {
                if (uriSize == UriSize.Small) {
                    listSmallUris.clear()
                    listSmallUris += newData
                    isSmallUrisDone.value = true
                    Log.d(my_tag, "Main-ViewModel Uri small added")
                } else {
                    listBigUris.clear()
                    listBigUris += newData
                    isBigUrisDone.value = true
                    Log.d(my_tag, "Main-ViewModel Uri big added")
                }
                job.cancel()
            }
        }
    }

    fun doWhenStartOrLogin() {

        val user = KindCafeApplication.myAuth.currentUser
        if (user != null) {
            // read data about personal
            dbManager.readUsersData(user, object : ReadUsersData {
                override fun readAllUserData(data: AllUserData) {
                    Log.d(my_tag, "Start or Login: in callback")
                    listUserInfo = data
                    isUsersInfoDone.value = true
                }
            })
```

```
            // write into local db
            lifecycleScope.launch {
                isUsersInfoDone.collect{
                    if(it){
                        Log.d(my_tag, "doWhenStartOrLogin:
doWhenStartOrLogin")
                        mainVM.deleteAllFavorites()
                        mainVM.deleteAllPersonal()

                        mainVM.deleteAllOrderItemsLocal()
                        mainVM.deleteAllOrderPlacedLocal()

                        listUserInfo.orderBasket?.let {orderBList ->
                            // if we downloaded from server, then delete
local (we will write further)
                            //mainVM.deleteAllOrderItemsLocal()
                            orderBList.forEach {
                                mainVM.addOrderItemsLocal(it)
                            }
                            Log.d(my_tag, "Start or Login: update basket from
server")
                        }

                        listUserInfo.orderPlaced?.let {orderPList ->
                            // if we downloaded from server, then delete
local (we will write further)
                            //mainVM.deleteAllOrderItemsLocal()
                            Log.d(my_tag, "Start or Login-- orderPList:
${orderPList}")
                            orderPList.forEach {
                                mainVM.addOrderPlacedLocal(it)
                            }
                            Log.d(my_tag, "Start or Login: update placed from
server")
                        }

                        listUserInfo.personal?.let {
                            mainVM.setPersonalDataLocal(it)
                            mainVM.setData(it.name)
                            Log.d(my_tag, "Start or Login: update personal")
                        }
                        listUserInfo.favorites?.let {
                            for (i in it){
                                mainVM.addFavoritesLocal(i)
                                Log.d(my_tag, "Start or Login: in favorite")
                            }
                        }
                        isUsersInfoDone.value = false
                        Log.d(my_tag, "Start or Login: person and fav done")
                    }
                }
            }



            // read data about order
            // write into local db
        } else {
            mainVM.setData(resources.getString(R.string.default_username))
        }
    }
```

```
    fun logoutAction(){
        if (accountHelper.signOut()) { // if we logout successfuly
            binding.lDrawLayoutMain.closeDrawer(GravityCompat.START)
            mainVM.setData(resources.getString(R.string.default_username))
            doWhenLogout()
        }
    }
    private fun doWhenLogout() {
        lifecycleScope.launch {
            mainVM.deleteAllLogout()
            cancel()
        }
    }

    private fun downloadLocalDb(){
        lifecycleScope.launch {
            Log.d(my_tag, "Local: getAllFavorites()")
            mainVM.getAllFavorites()
        }

        lifecycleScope.launch {
            Log.d(my_tag, "Local: getBasketLocal()")
            mainVM.getOrderItemsLocal()
        }

        lifecycleScope.launch {
            Log.d(my_tag, "Local: getOrderPlacedLocal()")
            mainVM.getOrderPlacedLocal()
        }

        lifecycleScope.launch {
            Log.d(my_tag, "Local: getAllDishes()")
            mainVM.getAllDishes()
        }

        lifecycleScope.launch {
            Log.d(my_tag, "Local: getOrderPlacedLocal()")
            mainVM.getOrderPlacedLocal()
        }

        lifecycleScope.launch {
            Log.d(my_tag, "Local: getPersonal()")
            mainVM.getPersonalDataLocal()
        }
    }
}
```

```
class ShowItemsFragment : Fragment() {
    /*----------------------------------------- Properties -------------------
---------------------*/
    private var _binding: FragItemsBinding? = null
    private val binding
        get() : FragItemsBinding {
            return checkNotNull(_binding) {
                "Cannot access binding because it is null. Is the view
visible"
            }
        }

    /* Common viewModel between activity and this fragment */
```

```
    private val mainVM: MainViewModel by activityViewModels()
    private lateinit var mainActivity: MainActivity

    //private val myAdapter = AdapterShowItems(clickItemElements())
    private lateinit var myAdapter: AdapterShowItems

    private val my_tag = "ShowItemsFragment"
    private val navArgs: ShowItemsFragmentArgs by navArgs()

    private val dbManager = DbManager()
    private val storageManager = StorageManager()

    val dList = mutableListOf<Dish>()
    val uSmallList = mutableListOf<Dish>()
    val uBigList = mutableListOf<Dish>()

    val goForwardMainData = MutableStateFlow(false)
    val goForwardUriSmallData = MutableStateFlow(false)
    val goForwardUriBigData = MutableStateFlow(false)

    /*-------------------------------------- Functions --------------------
----------------------*/

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragItemsBinding.inflate(layoutInflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        myAdapter = AdapterShowItems(
            AuxillaryFunctions.deafultItemMoveDirections(
                this,
                mainVM,
                { dish: Dish ->

ShowItemsFragmentDirections.actionShowItemsFragmentToDetailFragment(
                        dish.id,
                        dish.name!!
                    )
                }
            )
        )

        mainActivity = activity as MainActivity

        binding.rvShowItems.apply {
            layoutManager = LinearLayoutManager(requireContext())
            adapter = myAdapter
        }

        // retrieve data from Local DB
        viewLifecycleOwner.lifecycleScope.launch {
            mainVM.getDishesByCategory(navArgs.category)
        }

        // try to read data from Realtime DB
        dbManager.readDishDataFromDb(
```

```
            navArgs.category,
            clarificationGetDataFirebase()
        )

        // First try if we retrieve data from RDB, then retrieve uriSmall
        viewLifecycleOwner.lifecycleScope.launch {
            goForwardMainData.collect {
                if (it) {
                    Log.d(my_tag, "data added: Uri small")
                    storageManager.readUri(
                        dList,
                        clarificationGetUris(UriSize.Small),
                        UriSize.Small
                    )
                }
            }
        }

        // Next retrieve uriBig
        viewLifecycleOwner.lifecycleScope.launch {
            goForwardUriSmallData.collect {
                if (it) {
                    Log.d(my_tag, "data added: : Uri big")
                    storageManager.readUri(
                        uSmallList,
                        clarificationGetUris(UriSize.Big),
                        UriSize.Big
                    )
                }
            }
        }

        viewLifecycleOwner.lifecycleScope.launch {
            goForwardUriBigData.collect { third ->
                if (third) {
                    mainVM.addDishLocal(uBigList)
                    Log.d(my_tag, "all added")
                    goForwardMainData.value = false
                    goForwardUriSmallData.value = false
                    goForwardUriBigData.value = false
                    // we need get out there "Wait sign"
                }
            }
        }

        // Data we pass to adapter
        viewLifecycleOwner.lifecycleScope.launch {
            checkCategoryAndAction(navArgs.category)
        }

        viewLifecycleOwner.lifecycleScope.launch {
            mainVM.needUpdate.collect {
                if (it) {
                    myAdapter.notifyDataSetChanged()
                    mainVM.needUpdate.value = false
                }
            }
        }

    }

    private suspend fun checkCategoryAndAction(categoryCur: Categories) {
        when (categoryCur) {
```

```
            Categories.SparklingDrinks -> mainVM.sparklingDrinks.collect {
                myAdapter.setNewData(it)
                mainVM.currentLocation =
Locations.SHOW_SPARKLING_DRINKS.nameL
            }
            Categories.NonSparklingDrinks ->
mainVM.nonSparklingDrinks.collect{
                myAdapter.setNewData(it)
                mainVM.currentLocation =
Locations.SHOW_NON_SPARKLING_DRINKS.nameL
            }
            Categories.Sweets -> mainVM.sweets.collect{
                myAdapter.setNewData(it)
                mainVM.currentLocation = Locations.SHOW_SWEETS.nameL
            }
            Categories.Cakes -> mainVM.cakes.collect {
                myAdapter.setNewData(it)
                mainVM.currentLocation = Locations.SHOW_CAKES.nameL
            }
        }
    }

    private fun clarificationGetDataFirebase(): ReadAndSplitCategories {
        return object : ReadAndSplitCategories {
            override fun readAndSplit(data: List<Dish>) {
                dList.clear()
                dList += data
                goForwardMainData.value = true
            }
        }
    }

    private fun clarificationGetUris(uriSize: UriSize): GetUrisCallback {
        return object : GetUrisCallback {
            override fun getUris(newData: List<Dish>) {
                if (uriSize == UriSize.Small) {
                    uSmallList.clear()
                    uSmallList += newData
                    goForwardUriSmallData.value = true
                } else {
                    uBigList.clear()
                    uBigList += newData
                    goForwardUriBigData.value = true
                }

            }
        }
    }

    override fun onResume() {
        super.onResume()
        Log.d(my_tag, "onResume")
        mainActivity.everyOpenHomeSettings()
        mainActivity.supportActionBar?.title = ""
        mainActivity.binding.tvToolbarTitle.text =
navArgs.category.categoryName
    }

    override fun onPause() {
        super.onPause()
        Log.d(my_tag, "onPause")
    }
```

```
    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    override fun onDestroy() {
        super.onDestroy()
        Log.d(my_tag, "onDestroy")
    }
}


class BasketFrag: Fragment() {
    /*----------------------------------------- Properties -------------------
---------------------*/
    private var _binding: FragBasketBinding? = null
    private val binding
        get() : FragBasketBinding {
            return checkNotNull(_binding) {
                "Cannot access binding because it is null. Is the view
visible"
            }
        }

    /* Common viewModel between activity and this fragment */
    private val mainVM: MainViewModel by activityViewModels()

    private val my_tag = "BasketFragmentTag"
    private val currentFragmentName = "Basket"

    private lateinit var myAdapter : AdapterBasket
    private val dbManager = DbManager()

    private val detailedListI = mutableListOf<DetailedOrderItem>()

    private var swipeBackground: ColorDrawable =
ColorDrawable(Color.parseColor("#FF0000"))
    private lateinit var deleteIcon: Drawable

    /*----------------------------------------- Functions --------------------
---------------------*/

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragBasketBinding.inflate(layoutInflater, container,
false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        deleteIcon = ContextCompat.getDrawable(requireContext(),
R.drawable.ic_delete)!!

        myAdapter = AdapterBasket(
            AuxillaryFunctions.deafultItemMoveDirections(this, mainVM, null),
            AuxillaryFunctions.defaultClickSettingOrder(this, mainVM)
        )

        mainVM.currentLocation
```

```
        val mainAct = activity as? MainActivity

        mainAct?.let {
            it.binding.tvToolbarTitle.text = currentFragmentName
            it.supportActionBar?.title = ""
        }

        binding.rvItemsBuy.apply {
            layoutManager = LinearLayoutManager(requireContext())
            adapter = myAdapter
        }

        Log.d(my_tag, "basket: ${mainVM.orderBasket.value}")

        viewLifecycleOwner.lifecycleScope.launch{
            mainVM.getOrderItemsLocal()
        }

        viewLifecycleOwner.lifecycleScope.launch {
            mainVM.orderBasket.collect{

dbManager.setOrderBasketToRDB(KindCafeApplication.myAuth.currentUser, it)

                val detailedList =
AuxillaryFunctions.transformOrdItemToDishesDetailed(
                    it, mainVM.allDishes.value
                )
                detailedListI.clear()
                detailedListI.addAll(detailedList)

                Log.d(my_tag, "size detailed list: ${detailedList.size}")
                Log.d(my_tag, "list: ${detailedList}")
                myAdapter.setNewData(detailedList)
            }
        }

        viewLifecycleOwner.lifecycleScope.launch {
            mainVM.needUpdate.collect {
                if (it) {
                    myAdapter.notifyDataSetChanged()
                    mainVM.needUpdate.value = false
                }
            }
        }

        binding.clButtonMakeOrder.setOnClickListener {

            val someDishesHaveNotQuantity = detailedListI
                .filter { it.count == "0" || it.count == null}
                .isNotEmpty()
            /* if all dishes have quantity (count), then we can move further
*/
            if (!someDishesHaveNotQuantity){
                val action =
BasketFragDirections.actionBasketFragToOrderSummaryFragment(detailedListI.toT
ypedArray())
                findNavController().navigate(action)
            } else {
                Toast.makeText(context, R.string.quatity_zero,
Toast.LENGTH_SHORT).show()
            }
        }
```

```
        val itemTouchHelper =
ItemTouchHelper(AuxillaryFunctions.defaultSwipeDelBasketOrder(
            deleteIcon, swipeBackground, mainVM = mainVM
        ))
        itemTouchHelper.attachToRecyclerView(binding.rvItemsBuy)
    }

    override fun onResume() {
        super.onResume()
        mainVM.currentLocation = Locations.BASKET.nameL
        Log.d(my_tag, "onResume")
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

}
```