НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Мобільний додаток для магазину інструментів та обладнання

Виконав : студент __4__ курсу, групи _ІП-05_
(шифр групи)

_____Кононенко Вадим Олександрович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник ___асистент кафедри, доктор філософії, Шульга М.В.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) ___доцент, к.т.н., Волокита А. М._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __асистент кафедри БМК, Матвійчук О.В._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2024 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Кононенко Вадима Олександровича

1. Тема проєкту *Мобільний додаток для магазину інструментів та обладнання*
керівник проєкту Шульга М.В., доктор філософії
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 27 травня 2024 року №2112-с

2. Термін здачі студентом закінченого проєкту *12 червня 2024 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

1. Огляд предметної області та існуючих рішень

2. Вибір та обґрунтування технології розробки

3. Огляд програмної реалізації застосунку

4. Демонстрація розробленого застосунку

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) діаграма класів (функціональна схема), алгоритм дії програмного забезпечення (принципова схема), структура системи (структурна схема), текст програмного коду.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А. М.		

7. Дата видачі завдання «19» січня 2024 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*10.12.2023-15.12.2023*	
2.	*Вивчення та аналіз завдання*	*15.12.2023-15.03.2024*	
3.	*Розробка архітектури та загальної структури системи*	*15.03.2024-25.03.2024*	
4.	*Розробка структур окремих підсистем*	*25.03.2024-5.04.2024*	
5.	*Програмна реалізація системи*	*5.04.2024- 15.04.2024*	
6.	*Оформлення пояснювальної записки*	*15.04.2024- 20.05.2024*	
7.	*Захист програмного продукту*	*25.04.2024*	
8.	*Передзахист*	*23.05.2024*	
9.	*Захист*	*19.06.2024*	

Студент-дипломник _____ Вадим КОНОНЕНКО
(підпис)

Керівник проєкту _____ Максим ШУЛЬГА
(підпис)

АНОТАЦІЯ

Дипломний проєкт описує розробку мобільного застосунку на платформі IOS для магазину інструментів та обладнання. Застосунок був розроблений на мові Swift та фреймворку SwiftUI. Дані в застосунку зберігаються та читаються, за допомогою фреймворку CoreData. Основні функції застосунку – зручний перегляд зареєстрованих інструментів, відстежування термінів гарантії та реєстрація купленого інструменту.

Ключові слова: IOS, Swift, SwiftUI, CoreData, мобільний застосунок.

ANNOTATION

The diploma project describes the development of a mobile application on the IOS platform for a tool and equipment store. The application was developed in the Swift language and the SwiftUI framework. Data in the application is stored and read using the CoreData framework. The main functions of the application are convenient viewing of registered tools, tracking of warranty terms and registration of the purchased tool.

Keywords: IOS, Swift, SwiftUI, CoreData, mobile application.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Мобільний додаток для	4		
			магазину інструментів та			
			обладнання			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Мобільний додаток для	56		
			магазину інструментів та			
			обладнання			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Мобільний додаток для	1		
			магазину інструментів та			
			обладнання			
			Діаграма класів			
			(Функціональна схема)			
	А4	***ІАЛЦ.4672008.005 Д2***	Мобільний додаток для	1		
			магазину інструментів та			
			обладнання			
			Алгоритм дії програмного забезпечення			
			(Принципова схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Мобільний додаток для	1		
			магазину інструментів та			
			обладнання			
			Структура системи			
			(Структурна схема)			
	А4	***ІАЛЦ.467200.007 Д4***	Мобільний додаток для	17		
			магазину інструментів та			
			обладнання			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА		
Зм	*Лист*	*№ докум.*	*Підп*	*Дата*			
Розроб		Кононенко В.О.			*Мобільний додаток для*	Літ.	Аркуш 1 / Аркушів 1
Перев		Волокита А.М.					

ТЕХНІЧНЕ ЗАВДАННЯ

ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Мобільний додаток для магазину інструментів та обладнання*»

Київ – 2024

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання спрямоване на створення та розробку IOS-застосунку для магазину інструментів.

Область застосування: використання на українському ринку користувачами на платформі IOS. Даний застосунок надає зручність для відслідковування останніх подій та гарантії інструментів.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даного застосунку є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначення цієї роботи є створення застосунку для магазину інструментів та обладнання для зручного відстеження власних інструментів на платформі IOS.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки цього проекту є технічна документація від розробників у вільному доступі у мережі Інтернет, статті та форуми на дану тематику.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

- Сучасний та зрозумілий інтерфейс
- Надати користувачу можливість відстежувати власні інструменти
- Надати користувачу можливість додавати новий інструмент
- Надати користувачу можливість бачити власні бонуси та бонусну ставку, а також загальну суму покупок

5.2. Вимоги до програмного забезпечення

- MacOS
- IOS версії 17 або вище
- Xcode версії 15.0 або вище

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3-2100T чи ARM Apple M1
- ROM не менше ніж 128 ГБ
- RAM не менше ніж 8 ГБ

6. ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.12.2023-15.12.2023
Вивчення та аналіз завдання	15.12.2023-15.03.2024
Розробка архітектури та загальної структури системи	15.03.2024-25.03.2024
Розробка структур окремих частин системи	25.03.2024-5.04.2024
Програмна реалізація системи	5.04.2024-15.04.2024
Виправлення помилок	15.04.2024-20.05.2024
Оформлення пояснювальної записки	23.05.2024

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Мобільний додаток для магазину інструментів та обладнання*»

Київ – 2024

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ПК: Персональний комп’ютер

UI: User Interface (Користувацький інтерфейс)

SMS: Short Message Service

IoT: Інтернет речей

Push-повідомлення: Коротке спливаюче повідомлення

Bluetooth: Технологія бездротового зв’язку

TabBar: Навігація застосунку

UX: User Experience (Користувацький досвід)

LLVM: універсальна система аналізу та оптимізації програм

IOS: операційна система

SDK: набір для розробки програмного забезпечення

ВСТУП

В сучасному світі технології розвиваються швидкими темпами, разом з ними електронна комерція. Зараз просто мати красивий та зручно розташований магазин не означає успіх серед конкурентів, тому мобільні додатки вже стали невід'ємною частиною кожного бізнесу, від кав'ярні до потужних підприємств.

Зараз кожен бренд та кожен магазин вже за замовчуванням має свій власний веб сайт, проте з розвитком смартфонів та пришвидшення темпу життя купівля та поглинання контенту через смартфон стала вже набагато зручнішою і швидшою, порівнюючи з необхідністю використання ПК.

Однак, сайти не надають ту зручність та адаптивність на смартфонах, як для користувачів комп'ютерів. Дослідження показали, що мобільні додатки можуть збільшити продажі від 10% до понад 300%, в порівнянні з сайтами, залежно від того, наскільки добре додаток відповідає потребам клієнтів і наскільки ефективно його рекламують. Враховуючи, що смартфон є практично в кожного, ця інвестиція є пріоритетом для магазинів, які хочуть підвищити лояльність клієнтів, підвищити рівень утримання порівняно з веб-сайтами, забезпечуючи більш привабливий, персоналізований і зручний досвід покупок.

Магазини, які не використовують власні мобільні додатки, зачасту стикаються з заниженою конкурентоспроможністю через недостатність залучення клієнтів, незручності в процесах продажу та низької швидкості реагування на запити клієнтів. Все це може сильно впливає на зниження доходу, уповільнення маштабування та заниження рівню лояльності клієнтів до бренду.

Мета даної роботи полягає у створенні застосунку, який підвищить продажі магазину, за рахунок: зручного відслідковування здійснених покупок, можливість сортувати інструмент по об'єктам будівництва, полегшений спосіб контролювати терміни гарантії інструменту та створення заявки на ремонт, у

випадку поломки або для проведення сервісної профілактики. Такий застосунок надасть клієнтам кращий UX, збільшить комфорт в співпраці та підвищить лояльність до бренду як побутових користувачів, так і професійних.

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Коротка історія застосунків для електронної комерції

Послуги мобільної комерції вперше з’явилися в 1997 році в Фінляндії, там були представлені перші два автомати Coca-Cola з підтримкою мобільних телефонів. Автомати підтверджували платіж за допомогою SMS-повідомлень, що було великим кроком для того часу. Потім ще декілька компаній реалізували даний механізм для клієнтів.

Однак, все розвивається і не стоїть на місці, тому через декілька років був випущений телефон від Blackberry. Його основною перевагою та революційною функцією стала інтегрована електронна пошта в реальному часі і саме це підштовхнуло компанії до створення мобільних застосунків для електронної пошти для комунікації.

В 2008 році, коли був представлений Apple App Store, кількість застосунків доступних для завантаження було лише 800. Однак після досягнення певного рівня розвитку мобільних телефонів, почався й активний розвиток мобільних застосунків. Було створено багато застосунків, які призначались для налаштування телефону, розваги, а деякі призначались для полегшення роботи бізнесу. Компанії створили свої власні застосунки з метою зручності для своїх клієнтів купувати товари та послуги лише в кілька натискань. Тому зараз безумовно час мобільних застосунків, навіть попри те, що існують мобільні версії сайтів, користуватися мобільним застосунком в рази зручніше та швидше.

1.2 Значення мобільних додатків для електронної комерції

Ринок інструментів є швидко зростаючим та дуже перспективним напрямом. Ріст підсилюється тим, що зараз дуже популярне бажання користувачів зробити роботу самому, не залучаючи для цього третіх осіб, тому люди купують інструмент для домашнього використання, щоб відчути себе майстром і прикласти руку до створення якогось виробу або ж ремонту у власній квартирі / будинку.

Ключовими гравцями на українському ринку інструментів є DeWalt, Bosch, Makita, Metabo, Dnipro-M які популярні серед користувачів та інноваційності в своєму широкому асортименті товарів. На ринку України та в цілому світу вони займають значну частину ринку та напряму впливають на вектор розвитку індустрії. Вони також активно вкладаються в розвиток та впровадження інноваційних, перспективних технологій, які надають клієнтам новий рівень взаємодії з інструментами, полегшуючи та пришвидшуючи процеси, що в свою чергу економлять час та витрати.

Нові тренди спонукають компанії вкладати значні кошти в вровадження технологічний рішень для задоволення попиту у користувачів, популярні технології зараз це: бездротовий інструмент, IoT технології та швидко зростаюча технологія штучного інтелекту, яка змінила та продовжує змінювати процеси в усіх сферах, для забезпечення нового користувацького досвіду у своїх клієнтів.

Варто зазначити, що продовжує активно розвиватись саме продаж товарів через інтернет. Це спонукає компанії створювати стильні, зручні та дружні до користувача додатки. Раніше можно було мати лише вебсайт і це влаштовувало клієнтів та закривало їх потребу, проте з розвитком мобільних телефонів та їхньої інтеграції в повсякденне життя кожної людини, адаптивність веб сайтів

для смартфонів сильно впливає на зручність в пошуку та обранні товарів для користувачів.

Одним з дійсно важливих факторів для компаній є постійне нагадування про себе та бути в полі зору клієнта - з цим мобільні застосунки справляються ідеально, порівнюючи з тими ж веб сайтами.

Наявність функції відправки Push-повідомлень компанії можуть сповіщати своїх клієнтів про нові пропозиції, оновлення асортименту та заходи, та маючи можливості для таргетованих сповіщень та анонсів, що зменшує витрати на рекламу та підвищує ефективність спрацювання даного інструменту, оскільки смс повідомлення або контекстна реклама може загубитись в потоці подій, а от сповіщення на мобільному телефоні буде набагато більш помітним та підвищить відгук та інфопривід.

1.3 Огляд успішних додатків в сфері продажу будівельного обладнання

Тепер розглянемо застосунки популярних та успішних компаній, продукцією яких користуються по всьому світу, таких як DeWalt та Bosch.

Порівнювати будемо функціональність та особливості кожного застосунку.

DeWalt - це американський бренд електроінструментів, який зарекомендував себе довговічним та надійним, тому як не їм зробити застосунок дійсно інноваційним.

Відкриваючи застосунок нам спершу запропоную зареєструватись, а потім відкривається головна сторінка - перелік інструментів користувача (рис. 1.1).

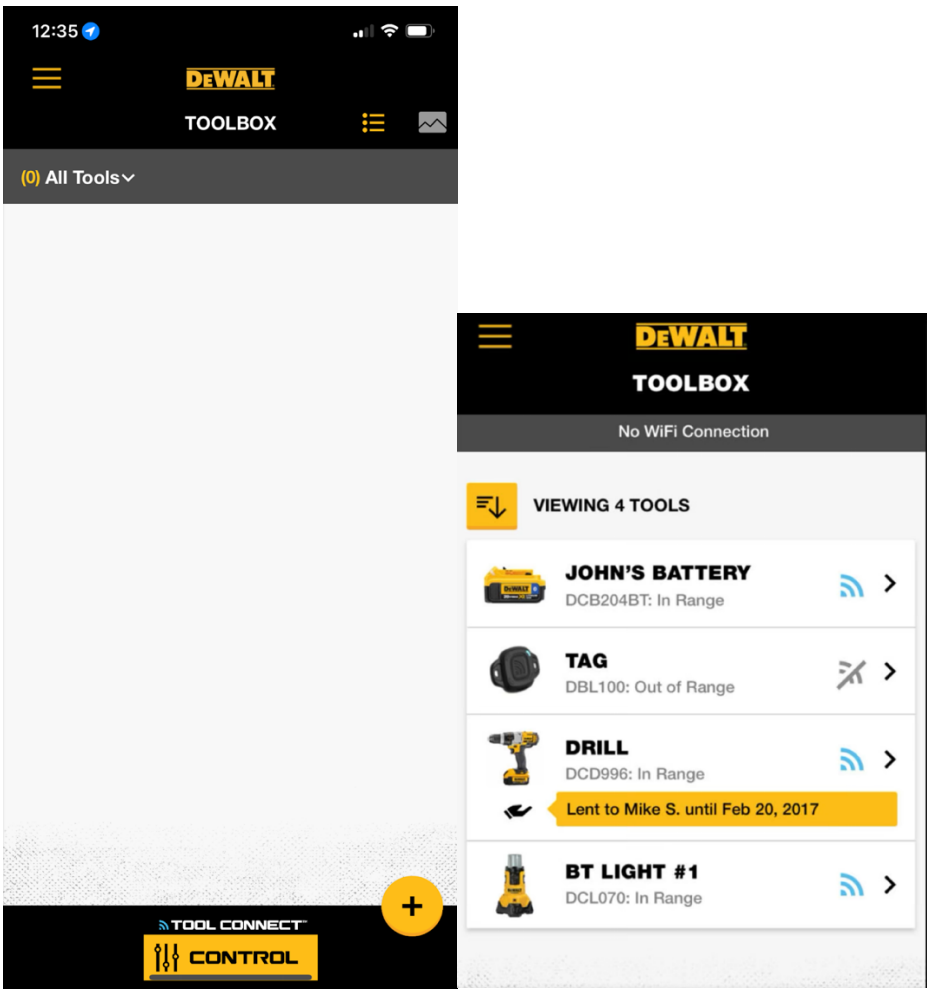


Рисунок 1.1. Головний екран застосунку

Також в застосунку реалізовано зручне додавання інструменту до списку власних девайсів. Натискаючи “+” справа знизу відкривається список усіх можливих типів інструменту, де вже можна з легкістю підключити власний інструмент до застосунку, використовуючи функцію Bluetooth (рис. 1.2). Для користувачів ця функція є ключовою в даному застосунку, тому розробники розробили зручний та простий спосіб для реалізації даної функції.

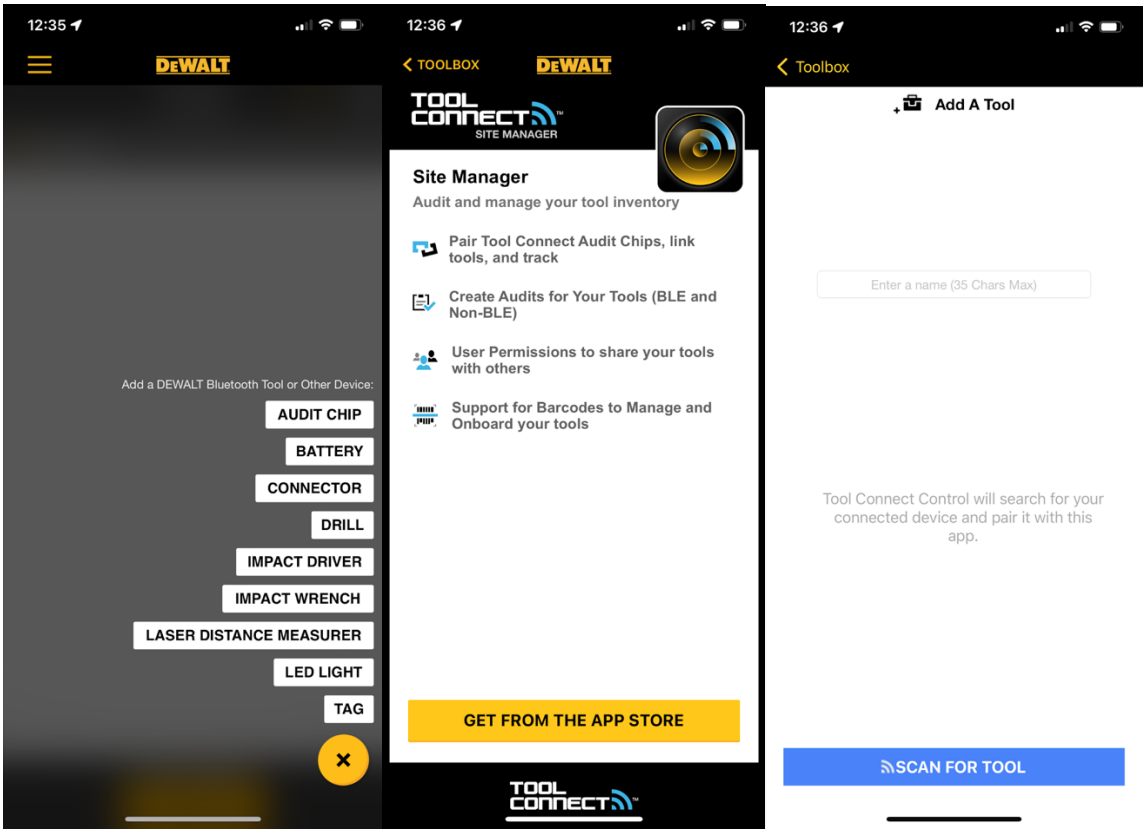


Рисунок 1.2. Додавання інструменту

Якщо перейти натисканням на окремий інструмент, то можно побачити детальну інформацію про нього, наприклад його стан, температуру та додаткову інформацію про версію програмного забезпечення, коли вперше інструмент був приєднаний до застосунку, також можна побачити додаткові дії та сповіщення стосовно інструменту (рис. 1.3). Дійсно зручним є те, що це все працює по технології Bluetooth, що в реалізовано в апаратній частині і для реалізації схожого функціоналу компанія додала додатковий модуль до інструменту, що підвищує його фінальну вартість. Однако таке рішення відкриває більше можливостей для реалізації додаткових крутих функцій та рішень для користувачів, що підвищить лояльність користувачів до бренду та створить потік нових, для кого такі функції несуть вирішальне значення для пришвидшення процесів та комфорту в роботі.

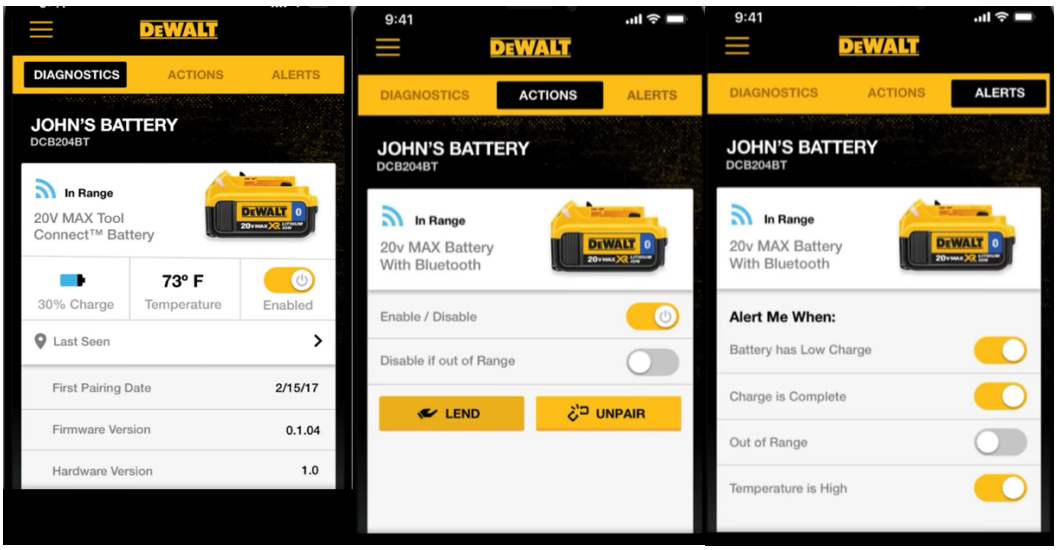


Рисунок 1.3 Детальний перегляд інструменту

А тепер можна розглянути переваги “DeWalt Tool Control”:

- Інноваційність є головною перевагою застосунку від DeWalt. Вона полягає у використанні Bluetooth та IoT для відстеження та управління інструментами, ця функція буде особливо корисна для професіональних користувачів, так званого професійного сегменту. Функція відстеження допомагає знайти інструмент на мапі, що буде доречним та корисним для користувачів.
- Взаємодія з інструментами: застосунок дійсно гарно інтегрований з великою кількістю інструментів DeWalt та забезпечує легкість в управлінні та обслуговуванні.
- Позика інструменту: за допомогою цієї функції можна позичати свої інструменти користувачам і бути спокійними, що вони автоматично вимикаються після закінчення періоду оренди.
- Виміри в реальному часі: під час роботи з інструментами ви можете через застосунок отримувати дані в реальному часі, що дозволяє зберігати вимірювання одразу на вашому пристрої.

- Мінімалістичність: застосунок виглядає стильним, зарахунок мінімалістичності та не перевантаженості застосунку, що дозволяє привертати увагу нових користувачів та утримувати існуючих.

Недоліки:

- Навігація реалізована у вигляді бокового меню, що у сучасних додатках є застарілим підходом та незручним в порівнянні з навігацією через TabBar.
- Відсутність єдиного простору для відслідковування термінів гарантії та в цілому відсутність вкладки “Сервіс”. Для багатьох клієнтів гарантія є великою перевагою бренду, та коли виникає потреба в ремонті, наявність даної функції облегчить цей процес своїм користувачам.
- Недоступність багатьох функцій у більшості країн, включаючи Україну, робить цей застосунок обмеженим в користувачах та незадоволеність у існуючих та потенційних клієнтів.

Наразі перейдемо до наступного застосунку популярного бренду Bosch, який має велику кількість шанувальників та користувачів і їх застосунок має назву “Bosch Toolbox”.

Переваги “Bosch Toolbox”:

- Багатофункціональність: даний застосунок пропонує багато дійсно корисних функцій, такі як калькулятор, конвертер одиниць вимірювання та наявність документації до інструментів, що робить його помічником для будь яких майстрів як домашніх, так і професійних (рис. 1.4).

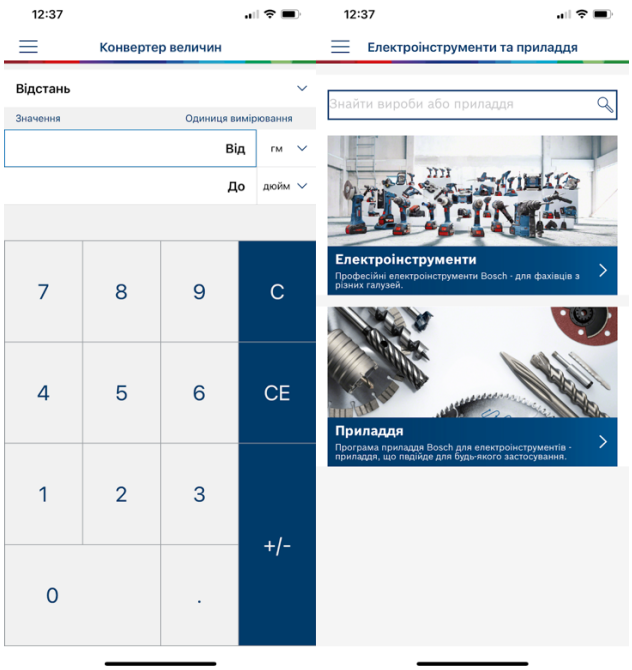


Рисунок 1.4. Корисні функції Bosch Toolbox

- Зручність інтерфейсу: головне меню надає можливості швидкого використання всіх функцій додатку, для швидкого старту роботи (рис. 1.5).

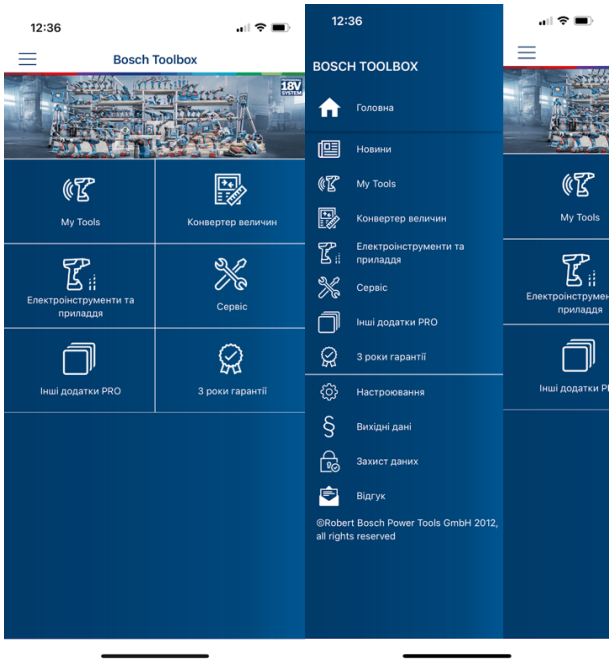


Рисунок 1.5. Головний екран Bosch Toolbox

- Каталог товарів: завдяки наявності одразу в застосунку можливості перегляду інструментів та витратних матеріалів, при потребі не потрібно буде додатково заходити на веб сайт, що зменшить час на отримання потрібої інформації.
- Зручність відстеження власних інструментів та можливість додавати інструменти в групи, для додаткових налаштувань розташування, категоризації (рис. 1.6).

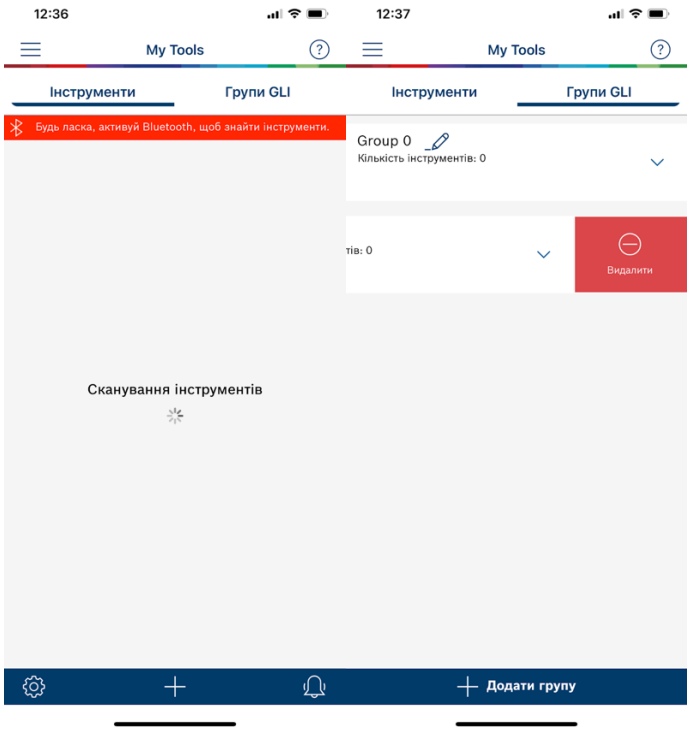


Рисунок 1.6. Екран доданих інструментів та груп

Недоліки “Bosch Toolbox”:

- Проблеми з навігацією: чим зручніше користувачу користуватись застосунком, тим це збільшує саме час проведений в застосунку і відповідно підвищує задоволення від використання. В застосунку компанії Bosch складний інтерфейс та обширний функціонал, проте через непродуману і не інтуїтивну навігацію важко, особливо

новому користувачу, знайти потрібну функцію та переміщатись між іншими вкладками та функціями. По-перше, навігація реалізована в боковому меню, це є основною незручністю. По-друге, стиль та логіка навігації зроблена без урахування платформи: користувачі IOS звикли до стандартної навігації і очікують цього в будь-якому застосунку, або хоча б дуже схожим до цього, однак функції «назад» та «вперед» по стеку навігації реалізовані знизу, в не інтуітивних та незручних місцях, що викликає дискомфорт та впливає на швидкість взаємодії з застосунком.

- Дизайн: одна з головних задач дизайну є створення першого враження. Користувачі переважно формують свою думку про застосунок після перших секунд взаємодії. Привабливий та стильний дизайн може допомогти користувачу захотіти дослідити та захотіти залишитись в даному застосунку. Гарний дизайн робить застосунок зручним та зрозумілим, допомагає користувачу без особливих проблем зрозуміти як користуватись застосунком. Одні з головних проблем в дизайні застосунку від Bosch:
    - Застарілий інтерфейс: застосунок досить функціональний, проте інтерфейс перевантажений, що додає складності користувачам в використанні. Також іконки, шрифти та в цілому UI/UI вже не відповідають сучасним трендам.
    - Палітра кольорів: обрана розробниками палітра є не досить яскравою та виглядає занадто монотонною. Комфорт в читанні та просто перегляді контенту на телефоні теж є вагомим фактором для задоволеності користувачів, але в застосунку бракує контрастності між елементами інтерфейсу.

ВИСНОВОК ДО РОЗДІЛУ 1

Сайти не надають такої адаптивності та інтеграції з функціями сматфонів, як мобільні додатки. Саме тому мати для компанії свій додаток це не тільки престижно, але й важливо для збільшення задоволеності клієнтів іх брендом та створити незабутні враження від продукту та взаємодії з брендом на інших етапах, таких як звернення на сервіс, зручна взаємодія з їх інструментом через додаток, який відкриває нові або розширює стандартні можливості інструменту.

В цьому розділі було розглянуто важливість для компаній створення власних застосунків, задля підвищення рівня задоволеності клієнтів та збільшення доходів через додаткову комунікацію через застосунок, у вигляді Push-повідомлень та бонусних програм.

Для кращого розуміння ринку застосунків компаній, які продають будівельний інструмент було проаналізовано переваги та недоліки застосунків від іменитих брендів. Дослідження показало, що основними проблемами є застарілий дизайн, не зовсім зручна навігація та низька швидкість відгуку інтерфейсу, що змушує користувача чекати через додаткові довгі анімації. Саме тому застосунок, який буде розроблятись в даній роботі буде включати в себе сучасний дизайн та простоту робочого простору, задля збереження уваги користувача і не втрачання фокусу, продуману навігацію для швидкості та зручності для користувача в використанні даним додатком.

РОЗДІЛ 2. ВИБІР І ОБГРУНТУВАННЯ ТЕХНОЛОГІЙ РОЗРОБКИ

2.1 Огляд мов програмування

Swift - це сучасна, високорівнева, універсальна мова програмування, розроблена Apple. Дана мова створена для швидкої, безпечної та інтерактивної роботи, що робить її придатною для системного програмування, мобільних та настільних додатків, а також серверних застосунків. Вперше представлена на WWDC Apple у 2014 році та виклала неоднозначну реакцію у розробників.

Однією з найважливіших переваг Swift насамперед є безпечність та надійність. Змінні завжди ініціалізуються перед використанням, масиви та цілі числа перевіряються на переповнення, пам’ять автоматично керується та захищає від багатьох програмних помилок. Ще важлива функція для безпеки полягає в тому, що за замовчуванням об’єкти Swift ніколи не можуть мати значення nil. Фактично, компілятор Swift не дозволить вам спробувати створити або використати нульовий об’єкт із помилкою під час компіляції. Такий синтаксис заохочує писати чистий та послідовний код, що в свою чергу покращує читабельність коду.

Висока ефективність має значення для розробки мобільних застосунків, оскільки ресурси мобільних пристроїв є обмеженими. Swift з самого початку задумувався та створювався для того, щоб бути швидким та продуктивним. Завдяки компіляції високопродуктивним компілятором LLVM, код на Swift перетворюється на гарно оптимізований машинний код, який ефективно використовує потужності сучасного обладнання та надає можливості створювати продуктивні застосунки.

Наразі існує 2 нативні мови програмування для розробки IOS додатків - Objective-C та Swift. Тож порівняймо основні відмінності цих мов.

Найпомітніша відмінність для кожного розробника це синтаксис. Код написаний на Swift чистий та зрозумілий та лаконічний, його легко читати та підтримувати. В свою чергу Objective-C базується на синтаксисі мови C, яку важко читати та набагато важче вчити самостійно. Код на Objective-C займає в 2 рази більше строк коду, ніж такий самий код на Swift.

Важливою перевагою Swift над Objective-C є безпечне управління пам'яттю, інтерференція типів, дженерики та опціональні типи, тому розробникам не потрібно витрачати свій час на роботу напряму з пам'яттю і дозволяє сфокусуватись на написанні ефективного коду.

Рішення зробити Swift проектом з відкритим кодом, призвело до прийняття її та дозволило створення ентузіастами багатьох відкритих бібліотек та фреймворків і дозволило цій мові розвиватись та розширюватись. Objective-C має закриту кодову базу і здебільшого залишається обмеженим лише для пристроїв Apple.

2.2 Огляд нативних фреймворків

2.2.1 UIKit

UIKit - це фреймворк, який дозволяє створювати користувацькі інтерфейси для IOS та tvOS, був представлений в 2008 році разом з першим iPhone та став основним фреймворком для створення користувацьких інтерфейсів в застосунках екосистеми Apple. Даний фреймворк був створений за допомогою мови Objective-C, яка була вдосконаленою версією мови C, яка надає можливості підтримки об’єктно-орієнтованого програмування (ООП) та в цілому створив значний поштовх для розробників та значно спростив процес створення застосунків, що дозволило створювати привабливі та інтуїтивно привабливі інтерфейси.

За цей час UIKit розвивався та став потужною та універсальною структурою для розробки інтерфейсу користувача. У результаті спільнота UIKit мала час, щоб створити величезну кількість крутезних застосунків, які дивують своїм функціоналом та дизайном. Розробникам вдалося створити елегантні та високопродуктивні програми, які користуються успіхом на ринку, що в котрий раз підтверджує потужність і гнучкість даного фреймворку.

Для створення продуктивних та адаптивних інтерфейсів (UI), розробники мають вже заготовлені компоненти, такі як кнопки, текстові поля, слайдери, перемикачі, колекції та навігація. Всі ці компоненти дозволяють розробникам не замислюватись про низькорівневу реалізацію і сфокусуватись на швидкому створенні зручних та дружніх для користувача інтерфейсів, що в свою чергу пришвидшує розробку та робить процес створення застосунку простішим та надає можливості для реалізації складних інтерфейсів з мінімальними витратами сил та часу.

З розвитком фреймворку до XCode (інтегроване середовище розробки для платформ Apple) додали інструмент, з назвою Interface Builder. Interface Builder - це візуальний редактор, який використовується для створення та розробки інтерфейсів користувача для платформи IOS. Даний інструмент дозволяє переглядати та редагувати так називаємі Storyboards, які включають в себе набір переглядів та екранів.

Важливою складовою розробки застосунків є адаптивність інтерфейсів під різні типи девайсів з різними діагоналями екранів. Завдяки технології AutoLayout - це ключовий інструмент для створення гнучких та адаптивних інтерфейсів в UIKit. Auto Layout дозволяє створювати динамічні інтерфейси, які відповідають на зміни розміру та орієнтації екрана, задавши обмеження для UI елементів, які автоматично адаптуються під різні умови відображення. Size Classes спрощують управління портретною та ландшафтною орієнтацією екранів.

Також ком'юніті розробників ентузіастів створили додаткові бібліотеки, які спрощують процес створення адаптивних екранів. Одними з них є бібліотеки: SnapKit, TinyConstraints.

SnapKit є потужною та популярною для роботи з AutoLayout. Ця бібліотека спрощує створення та управління обмеженнями представлень для реалізації адаптивного макету, що в свою чергу забезпечує кращу читаємість та зрозумілість коду. Обмеження задаються через блок коду (closure) ланцюговими викликами.

Схожа за функціоналом та можливостями бібліотека TinyConstraints є легкою та зручною для розробників, що надає можливості для написання читабельного та лаконічного коду без особливих проблем.

Не дивлячись на вік UIKit, він досі залишається досить популярним. Основними причинами цього є декілька факторів:

- Широка база розробників та екосистема: UIKit багато років був основним фреймворком для розробки інтерфейсів для IOS. Багато розробників та компаній мають чималий досвід в роботі з даним фреймворком, це в свою чергу означає наявність великої кількості ресурсів, навчальних курсів та спільноти, де можна знайти відповідь на багато питань, бо на ваше запитання вже, скоріш за все, є відповідь.
- Кількість існуючих додатків: за весь час існування UIKit на ньому було створено величезну кількість застосунків. Для переходу на нову технологію компаніям може знадобитись занадто багато ресурсів та часу, що не є дуже рентабельним рішенням.
- Стабільність та надійність роботи застосунка: для багатьох компаній питання стабільності і надійності є вирішальним при визначенні технології. Тому й досі більшість компаній обирає старий, але надійний та працюючий UIKit.

- Підтримка старих версій системи: так само для більшості компаній є важливим фактором саме охват більшості користувачів. Досі є значна кількість людей, які користуються IOS п’ятирічної давності, тому компанії обирають той інструмент, який задовільнить великий відсоток користувачів. Для таких компаній сфера діяльності яких, наприклад, банки дуже важливо дати можливість користування застосунком практично для всіх своїх клієнтів, оскільки це напряму буде впливати на їх дохід.

Враховуючи всі перелічені фактори, можна зробити висновок, що UIKit це фреймворк який ще надовго залишиться затребуваним і його вивчення є обов’язковим. UIKit надає гнучкість та контроль застосунком і без нього, використовуючи новий фреймворк SwiftUI, ви не зможете настільки детально на всіх етапах контролювати поведінку застосунка.

2.2.2 SwiftUI

В 2019 році був представлений інноваційний фреймворк SwiftUI, який змінив кардинально підхід до розробки та швидкості створення застосунків. Маючи за основу декларативний підхід до створення представлень та екранів, SwiftUI дозволяє розробникам ефективніше та легче створювати складні представлення та в цілому пришвидшує створення застосунків з 0. Така перевага інноваційного фреймворку від Apple допомогла значно спростити створення інтерфейсів, зменшити кількість коду та підвищити його читабельність.

Навіть з назви цього фреймворку зрозуміло, що SwiftUI написаний для мови Swift та використовує переваги функціоналу Swift, до прикладу, використання конструкції closure, щоб зробити декларативний код ще більш читабельним та зрозумілим. Розглянемо основні переваги нового фреймворку:

Основна перевага SwiftUI над UIKit - декларативний підхід до розробки представлень. Суть такого підходу полягає у тому, що розробник описує як він хоче, щоб виглядав інтерфейс, замість того, як цього досягти. Модифікатори - це потужний інструмент, який дозволяє модифікувати вигляд та поведінку в декларативному стилі, що є основним для створення UI.

Попередній перегляд дозволяє бачити зміни одразу під час написання коду. Ця функція пришвидшує процес розробки переглядів та надає можливість не запускаючи весь застосунок побачити зміни, та при необхідності одразу внести зміни.

Передбачуваність, завдяки односпрямованому потоку даних: компоненти інтерфейсу написані на SwiftUI більш передбачувані. В свою чергу в UIKit використовуються такі техніки, як делегація (Delegation) та оповіщення (Notification). Односпрямований потік даних може вплинути на зменшення помилок в інтерфейсі користувача, спричинених складністю керування змінним станом.

Прив’язка даних: функціонал, завдяки якому застосунки написані на SwiftUI оживають - прив’язка даних (Data Binding). Це дуже потужний механізм для забезпечення планого потоку даних між різними компонентами застосунку. Певні зміни в одному компоненті застосунку запускають зміни в інших компонентах, які покладаються на ці дані, завдяки все працює просто та без особливих налаштувань. Обгортки позначаються спеціальним символом @ на початку. В SwiftUI використовуються 3 основні обортки для зав’язування даних: @State, @Binding, @ObservableObject.

State - одна з найбільш часто використовуваних обгорток властивостей у SwiftUI. Призначення обгортки: змінення стану в одному представлені. Позначаючи змінну за допомогою State, ви дозволяєте SwiftUI відстежувати її зміни та автоматично оновлювати представлення щоразу, коли змінюється стан. Однак State працює лише в межах одного преставлення.

Використовуючи Binding, по суті, створюєте живий зв’язок між даними в батьківському представлені та даними в дочірньому представленні. Будь-які зміни, які ви вносите до даних у дочірньому представлені, миттєво відображатимуться в батьківському поданні і навпаки.

ObservableObject — це обгортка властивостей, розроблена для більш складного та спільного керування станом у різних представленнях. Це дозволяє вам створити об’єкт моделі, який зберігає стан вашої програми, і будь-яке представлення, що використовує цей об’єкт, автоматично оновлюватиметься, коли стан змінюватиметься.

Сумісність між платформами: SwiftUI підтримує кросплатформенну розробку, дозволяючи розробникам створювати застосунки для IOS, iPadOS, macOS, watchOS, tvOS та VisionOS використовуючи єдину кодову базу, що може значно знизити час на розробку декількох застосунків з використанням різних фреймворків та технологій.

Apple продовжує інвестувати в цю технологію, тому з упевненістю сказати, що SwiftUI це майбутнє розробки застосунків для Apple екосистеми. З виходом нового гаджета Apple Vision Pro з власною операційною системою VisionOS, один і єдиний спосіб створювати застосунки для нового гаджета є SwiftUI, компанія вже напряму показує вектор розвитку і те, що UIKit - це минуле і потрібно переходити на свіжий та перспективний SwiftUI. Однак, не дивлячись на те, що SwiftUI представлений 5 років тому, в загальному основною проблемою залишається незрілість фреймворку, а також розглянемо інші недоліки SwiftUI:

Мінімальна версія IOS: для багатьох компаній головним критерієм є мінімальна версія операційної системи, яку має підтримувати застосунок. SwiftUI доступний починаючи з 13 версії IOS, це означає, що користувачі з старими девайсами, або користувачі, які з різних причин не захотіли оновлюватись на останню версію системи, не зможуть завантажити даний додаток і в підсумку компанія втратить потенційних клієнтів та прибуток.

Недостатність сторонніх бібліотек: Екосистема сторонніх бібліотек для SwiftUI не така широка, як у UIKit. Розробникам може знадобитися створити власні рішення для певних функцій.

SwiftUI є чудовим наступником UIKit та є ідеальним інструментом для розробників, що дозволяє створювати кросплатформенні застосунки. Так, він ще незрілий, проте незважаючи на деякі виклики, SwiftUI точно стане основою майбутнього розроблення додатків у всьому портфоліо продуктів Apple.

2.3 Огляд альтернативних технологій

2.3.1 Flutter

Flutter - це фреймворк мобільної розробки з відкритим вихідним кодом, створений Google і випущений для спільноти відкритих програм у 2011 році.

Основною мовою програмування є Dart, розроблена так само Google і є об’єктно-орієнтованою мовою програмування, яка базується на класах. В синтаксисі Dart можна знайти багато схожого з Java або JavaScript. Dart оптимізований для UI, що дозволяє легко маніпулювати елементами інтерфейсу та управляти можливими станами застосунку.

Весь Flutter побудований за допомогою віджетів. Віджети представляють собою все, що ви бачите у себе на екрані - від кнопки до складних макетів. Для розробників існують велика кількість вже готових віджетів, які кожен може легко додати в свій проєкт, а також є можливість створювати власні віджети виходячи з потреб.

Головна перевага полягає в тому, що ви можете створити цю програму лише за допомогою однієї мови програмування та використовуючи єдину кодову базу, таким чином у вас буде лише одна кодова база, з якою буде простіше працювати. Flutter - це SDK (набір для розробки програмного

забезпечення), який дозволяє скомпілювати програму в машинний код, який працює на IOS, Android, Web, macOS. Однак, не дивлячись на переваги кросплатформеності, розробка може супроводжуватись викликами для забезпечення однакової функціональності та поведінки на різних платформах.

Швидкість розробки забезпечується функцією гарячого перезавантаження (hot reload), що дозволяє розробникам одразу побачити зміни в додатку без необхідності запуску застосунка повністю і пришвидшує, в цілому, процес створення готового застосунка.

Проте, як завжди існує своє “але”, тому розглянемо мінуси Flutter:

Складнощі з обслуговуванням: коли доводиться оновлювати віджети і матеріальні елементи Cupertino для Android та IOS, Flutter від розробників вимагає надзвичайної обережності, що ускладнює процес додавання нових функцій та в цілому процес обслуговування кодової бази.

Розмір застосунка: Flutter використовує свій власний механізм візуалізації та компоненти, що може збільшити розмір програми та може викликати проблеми з сумісністю.

Тож з упевненістю можна сказати, що Flutter інноваційний фреймворк та продовжує надавати можливості для швидкого створення кросплатформенних застосунків.

2.3.2 React Native

React Native був розроблений в компанією Facebook та випущений у 2015 році. Він є фреймворком з відкритим вихідним кодом, який дозволяє створювати кросплатформні мобільні додатки за допомогою JavaScript і React, бібліотеки JavaScript.

Почнемо одразу з головних переваг React Native для розробки мобільних додатків:

Повторне використання коду та єдина кодова база: швидкість створення застосунків, за рахунок можливості використання того ж самого коду для різних платформ.

Швидкість розробки: має функцію гарячого перезавантаження, що дозволяє бачити зміни одразу та вносити певні корективи навіть під час завантаження програми. Такоє є можливість перезавантажити окрему частину коду.

Сторонні плагіни: React Native з коробки пропонує ряд заготовлених плагінів від інших розробників та нативні модулі, це в свою чергу дозволяє економити час при розробці, взявши вже готовий компонент та не переписувати його з нуля.

Ком'юніті розробників: загалом React Native зародився після потреб спільноти розробників, тож він має обширне ком'юніті кваліфікованих розробників, що в свою чергу полегшує отримання підтримки за будь-якими питаннями.

Розглядаючи цей потужний фреймворк, не можна не зазначити й мінуси, що зачасту пов'язані з мовою JavaScript:

Перший та головний фактор, який впливає на досвід і задоволеність користувача - продуктивність. Хоча додатки, створені на React Native, відчуваються як нативні, проте продуктивність може бути значно нижчою в сценаріях, де потрібні інтенсивні обчислення.

Використання JavaScript моста робить пошук помилок та їх виправлення складнішим та довшим, порівнюючи з нативними додатками. Розробка зачасту супроводжується, в більшості, пошуком та лагодженням помилок, то цей фактор є досить вагомим при обранні фреймворку для розробки застосунку.

Проблеми з пам'яттю напряму впливають на перший пункт - продуктивність. Як вже було зазначено, на мобільних пристроях ресурси є обмеженими, тому витоки пам'яті призводять до уповільнення роботи застосунку або ж збільшення використання вільних ресурсів гаджету.

Підсумовуючи всі переваги та мінуси React Native слід зазначити, що він досить популярним, з великим ком’юніті шанувальників та з іменитими користувачами даної технології для створення своїх додатків та завдяки саме ним фреймворк продовжує розвиватись. Однозначно він гідний уваги і завдяки своїм перевагам, може легко конкурувати зі схожими фреймворками.

2.4 Технології управління даними

2.4.1 CoreData

CoreData - це фреймворк управління об'єктними графами та постійним зберіганням даних, розроблений Apple. Вперше представлений у 2005 році з виходом Mac OS X 10.4 Tiger, він був побудований як частина Cocoa API. Згодом CoreData став доступним і для iOS, починаючи з версії 3.0. Розробка CoreData була мотивована потребою у забезпеченні потужного, але при цьому простого у використанні рішення для управління моделлю даних у додатках для Apple пристроїв.

Технологія, яка є ідейним надихачем CoreData, була продуктом команди NeXT, Enterprise Objects Framework (EOF). Задум розробників EOF був доволі цікавий, вони реалізували підключення до бази даних з приховуванням деталей реалізації, а наступне це додали механізм, щоб при зчитуванні даних з реляційного формату, для зручності, вони були конвертовані у набір об’єктів. Зазвичай розробники взаємодіяли лише з об’єктами, що спрощувало розробку складних програм ціною певних налаштувань для відображення даних в об’єктах.

Система керування станом об’єкта в EOF в дійсності не мала абсолютно нічого спільного з реляційними базами даних. Один той самий код міг використовуватися і використовувався розробниками для керування графами

інших об’єктів. У цій ролі дійсно корисними частинами EOF були ті, які автоматично створювали набори об’єктів із необроблених даних, а потім відстежували їх. Саме ця концепція є основою Core Data.

У розробників була потреба в ефективному способі управління складними даними, такі як зберігання, керування версіями, кешування та оптимізація запитів, на тлі цього Apple і випустила CoreData, як потужний інструмент, що закриває потребу розробників та спрощує процес створення на налаштування застосунку.

Переваги CoreData:

- Ефективне керування даними: CoreData надає високий рівень абстракції. Це є величезним плюсом для розробників, високрівневість дозволяє зосередитись на логіці програми, а не на низькорівневих операціях.
- Автоматичне відстеження змін: CoreData сам автоматично відстежує зміни в об’єктах, якими він керує. Тож коли ви вносите певні зміни в об’єкті, CoreData оновлює основну базу даних, що дозволяє спростити процес збереження змін.
- Оптимізація пам’яті: CoreData використовує методи, такі як faulting для оптимізації використання пам’яті. Це означає, що об’єкти завантажуються в пам’ять тільки тоді, коли вони дійсно потрібні, зменшуючи обсяг споживаної пам’яті.
- Управління сутностями: ви можете створювати всі основні зв’язки з сутностями, такими як «один-до-одного», «один-до-багатьох», «багато-до-одного», «багато-до-багатьох» та за рахунок цього дозволяє представляти складні структури даних.

Тепер розглянемо недоліки CoreData:

- Комплексне налаштування: налаштування проєкту з CoreData може включає наступні етапи, такі як створення моделі, генерація підкласів та налаштування постійного координатора для сховища.

- Можливі проблеми з продуктивністю: розробники подбали про ефективність CoreData, проте неправильне використання або неправильне моделювання даних потенційно може призвести до проблем з продуктивністю.
- Проблеми в додатках з великою кількістю контекстів: в програмах, в яких є кілька контекстів, об’єднання змін може створити певні додаткові труднощі.

2.4.2 Realm

Realm - це система керування об’єктними базами даних з відкритим кодом. Розробка Realm розпочалася наприкінці 2010 року Олександром Стігсеном разом із Б’ярне Крістіансеном під назвою TightDB. Його рекламували як NoSQL із довговічністю, яку можна налаштовувати та можливістю спільного використання тих самих згрупованих даних між кількома процесами, а також навіть кількома пристроями та кластерами.
TightDB був перейменований на Realm у вересні 2014 року та випущений для публічного тестування.

Це дійсно потужний інструмент, який полюбляють розробники, і тепер розглянемо в чому саме його переваги.

Головною перевагою Realm є його швидкість. За рахунок того, що Realm є не реляційною базою даних і порівнюючи його з традиційними реляційними, то вони вимагають вимагають складних запитів та включають в себе кілька кроків для доступу до даних. На противагу, Realm використовує простішу модель даних та дозволяє забезпечити швидкий та ефективний доступ до даних - це є основною перевагою для мобільних додатків, де користувачі очікують від роботи застосунку швидкої роботи.

Гнучкість платформи та інтегрування Realm дозволяє працювати з ним на будь-яких платформах та з будь-якими мовами програмування, включаючи IOS, Android, React Native, Xamarin. Це робить його універсальним інструментом, який буде до нагоди для розробки на будь-якій платформі.

Теж немало важлива функція, яка полегшує життя розробникам - підтримка синхронізації даних у реальному часі між пристроями. Ще з приємних функцій в ньому реалізовано вбудоване шифрування для конфіденційності даних.

Тепер перейдемо до мінусів Realm:

- Керування міграціями: У той час як Realm дозволяє легко змінювати схеми даних, управління міграціями може бути складним, особливо в великих і складних базах даних.
- Витрати: Для використання деяких функцій Realm, таких як серверна синхронізація, можуть знадобитися додаткові витрати, особливо при масштабуванні додатків, що в свою чергу вплине на збільшення ціни застосунку для кінцевого користувача.

2.4.3 Firebase

Firebase - це набір серверних служб хмарних обчислень і платформ розробки додатків від Google. В собі він містить бази даних, служби, автентифікацію та інтеграцію для різноманітних програм, включаючи Android , iOS , JavaScript , Node.js , Java , Unity , PHP і C++ .

Трошки пірнемо в історію і розглянемо як виник нині дуже потужний та відомий інструмент для кожного розробника. Firebase розвинувся з Envolve, попереднього стартапу, заснованого Джеймсом Темпліном і Ендрю Лі в 2011 році. Envolve дав розробникам API для інтегрування онлайн-чату для їхніх веб сайтів. Розробники користувались можливостями Envolve для синхронізації

даних в реальному часі між своїми користувачами. В подальшому вони заснували компанію Firebase у 2011 році, а була запущена в 2012 році.

Цей сервіс має багато прихильників та користувачів, тому почнемо з його переваг.

Головною перевагою Firebase є те, що розробникам не потрібно реалізовувати підтримку серверної частини для створення застосунків, оскільки вона поставляється з SDK. Також важливим є синхронізація та зберігання в реальному часі. Це сильно полегшує роботу розробникам з даними з будь-якого пристрою.

Google Analytics дозволяє відстежувати шлях вашого користувача на пристроях. Це означає, що ви будете знати, чи використовує він смартфон, планшет чи ноутбук.

Firebase надає сервіс для швидкого і зручного виправлення помилок. Багато застосунків страждають від проблем з помилками, через що користувачі можуть відмовитись від нього і рейтинг програми падає. Firebase надає нам, як розробникам, можливість отримувати чіткі звіти про збої, що дозволяє швидше та легше виправляти помилки.

Firebase Authentication: даний сервіс полегшує створення етапу автентифікації для легкого входу в систему користувача. Можлива автентифікація за електронною поштою, Twitter, Facebook, Instagram.

Легкість збереження будь-якого контенту користувача, наприклад текстів, зображень та відео. Команда Firebase також надала SDK для хмарного сховища, щоб підключити мобільні пристрої для користувачів, які не в мережі. Таким чином, вони можуть продовжувати автоматичну передачу, щойно з’єднання буде встановлено.

Недоліки Firebase:

- Платформа з закритим кодом: закритість системи, надає дуже обмежений контроль розробників над додатком і розробники не можуть змінити код, навіть якщо реалізація Firebase не відповідає їх вимогам. Є певні SDK на

GitHub, які надають могу обходити дані обмеження. Однак, закритий код не дозволяє спільноті розробників давати суттєвий внесок у розвиток платформи.

- Ціна: безкоштовний тариф надає розробникам лише основні функції, але функції, які дійсно спрощують та прискорюють процес розробки є платними. Пропрієтарність Firebase, можливо, є головним чинником високої ціни на продукт. Якщо б це була платформа з відкритим вихідним кодом, то за рахунок сил розробки спільноти, ціна могла б бути нижчою, або ж навіть безкоштовною.
- Відсутність підтримки SQL бази даних: дві бази даних, які доступні на Firebase, є не реляційними - NoSQL. Навіть незважаючи на те, що Google додали кілька функцій до Firestore, виконання складних запитів на платформі є непростим завданням.
- Обмеженість лише Google Cloud: оскільки Firebase розміщено в Google Cloud, одному з найпотужніших хмарних сервісів у світі сьогодні. Однак ви не можете використовувати Firebase на інших платформах хмарних сервісів, як-от DigitalOcean, AWS або Azure. Фактично Google спеціально унеможливлює вибір конкурентів, які можуть бути економічно більш вигідних за Firebase.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі бакалаврської роботи було розглянуто найпопулярніші фреймворки для нативної розробки застосунків для платформи IOS, такі як UIKit та SwiftUI, а також фреймворки для кросплатформної розробки - Flutter та React Native.

Для реалізації застосунку було обрано SwiftUI, як основний фреймворк, оскільки він є нативним та новим фреймворком, який активно оновлюється та додаються функції та механізми, які в свою чергу полегшують та пришвидшують процес розробки застосунку. Немало важливим фактором стала швидкість розробки та простота реалізації функціональності застосунку, оскільки декларативний підхід дозволяє описувати, що я хочу побачити в результаті, а не вручну прописувати як його досягти.

Для роботи з Back-end частиною було прийнято рішення обрати CoreData, оскільки це нативне рішення, що додає швидкості і зручності в роботі з даними.

РОЗДІЛ 3. ОГЛЯД ПРОГРАМНОЇ РЕАЛІЗАЦІЇ МОБІЛЬНОГО ДОДАТКУ

Дослідивши детально тематику, мову програмування та додаткові фреймворки, які були обрані для даної роботи, було розглянуто достатню кількість переваг існуючих інструментів для розробки застосунку, тому можемо переходити до розробки продукту.

3.1 Створення проєкту

Для того, щоб створити проєкт для застосунку для платформи IOS потрібно обрати в меню вибору типу проєкту “App” (рис. 3.1).

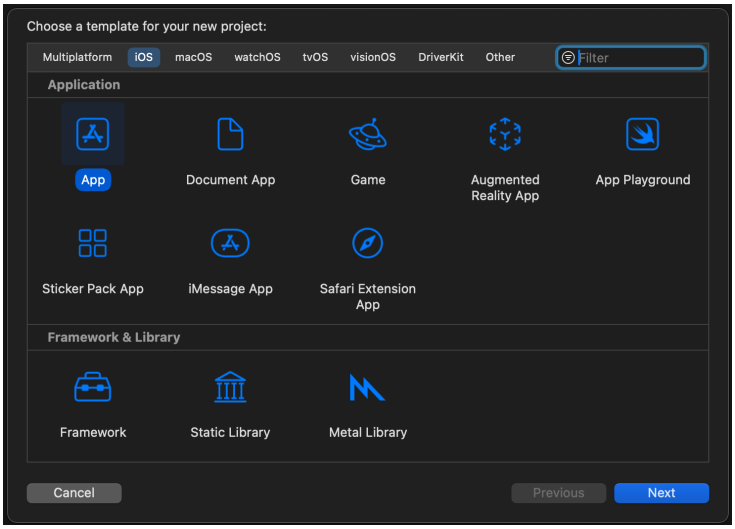


Рисунок 3.1. Вибір типу проєкту

Після обрання потрібного типу проєкту нам відкривається додаткових налаштувань проєкту (рис. 3.2), таких як:

- Назва проєкту
- Команда

- Ідентифікатор організації: тут потрібно вказати ідентифікатор, що в свою чергу є способом унікально ідентифікувати себе та свої застосунки в App Store
- Інтерфейс: тут потрібно обрати бажаний фреймворк для проєкту, на базі якого буде створений проєкт. В нашому випадку – SwiftUI
- Мова програмування: є можливість обрати Objective-C або Swift. Однак, для проєктів з використанням інтерфейсу SwiftUI обрати Objective-C буде неможливо – тільки для проєктів на UIKit
- Сховище
- Опція згенерувати файли для тестування

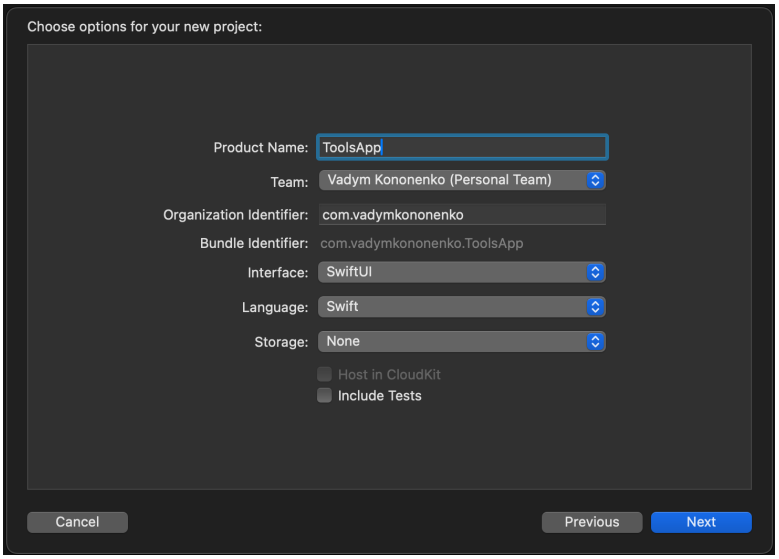


Рисунок 3.2. Налаштування проєкту

3.2 Налаштування бази даних

Після того як ми створили проєкт нам потрібно буде додати модель Core Data до проєкту. Для цього натискаємо «New File» -> в пошуку вводимо «Data» -> обираємо «Data Model» (рис. 3.3), потім відкривається вікно для налаштувань розташування файлу та його назви – вводимо назву «ToolsApp».

Після цього до проєкту додається файл ToolsApp.xcdatamodeld, що і буде зберігати наші моделі даних.

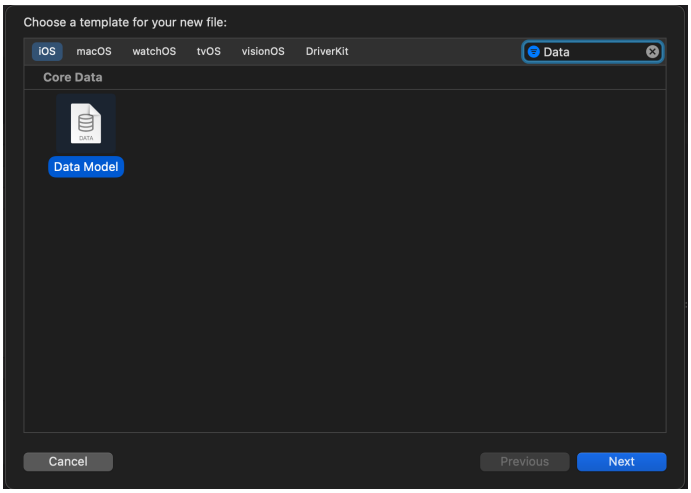


Рисунок 3.3. Пошук шаблону моделі Core Data

3.3 Структура моделей бази даних

Для даного проєкту були створені 3 моделі даних, такі як: Користувач (User), Інструмент (Tool), Подія (Event). Кожна модель має власні атрибути та залежності одна з одною, все це необхідно для зручного і зрозумілого запису і подальшого запису/зчитуванню потрібних даних про користувача.

Для сутності «User» в розділі «Attributes» було задано усі необхідні поля (рис. 3.4), в яких буде зберігатись інформація про конкретного користувача.

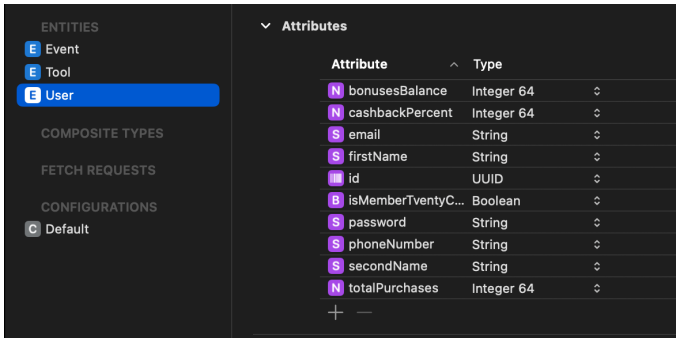


Рисунок 3.4. Сутність «User»

Наступним кроком було створення сутності інструменту «Tool» (рис. 3.5), яка відображає інструмент, яким володіє користувач. Атрибути id та serialNumber є ключовими для унікальності інструменту і унеможливлює реєстрацію інструменту, якщо такий вже зареєстрований у іншого користувача.

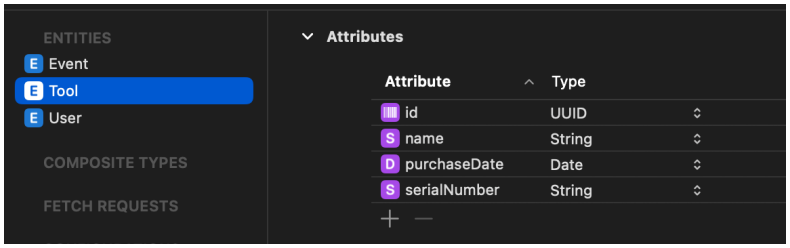


Рисунок 3.5. Сутність «Tool»

Остання сутність «Event» (рис. 3.6) потрібна для зберігання подій, пов’язаних з конкретним користувачем та зберігає необхідні дані про тип події, статус, місце та інструмент.

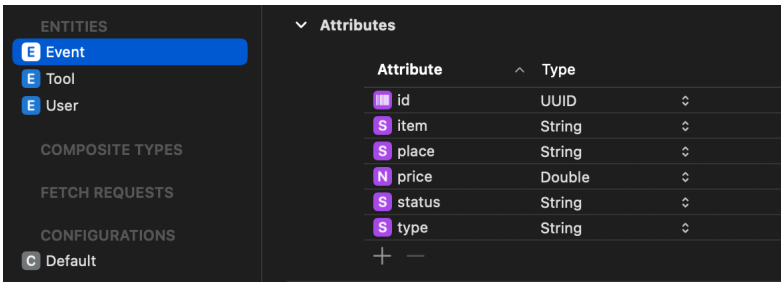


Рисунок 3.6. Сутність «Event»

Зважаючи на те, що сутності мають бути пов’язані з користувачем «User», було створення взаємозв’язки користувача з інструментами «Tool» з типом «To Many» та подіями «Event» з таким самим типом «To Many», що необхідно, щоб кожен користувач міг мати певну кількість інструменту та подій, які з ним пов’язані.

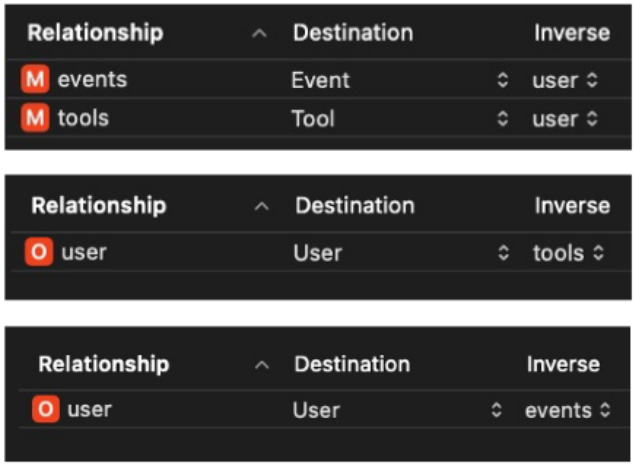


Рисунок 3.7. Взаємозв’язки між сутностями

3.3 Налаштування Core Data менеджера

Для коректної роботи з Core Data необхідно створити окремий клас, який буде відповідати за роботу з базою даних, тож створимо клас CoreDataManager.swift. Так, як CoreData є вбудованим фреймворком ми можемо не додавати залежності до проєкту, а можемо одразу прописати імпортування CoreData. Даний клас може бути використаний в різних класах одночасно, тому використовується шаблон проєктування Singleton, шляхом створення статичної змінної та приватного ініціалізатора. (рис. 3.8).

```
import Foundation
import CoreData

class CoreDataManager {

    static let shared = CoreDataManager()

    private init() {

    }
```

Рисунок 3.8. Використання шаблону проєктування Singleton

Тепер необхідно налаштувати Core Data контейнер та контекст, для цього потрібно створити змінні container та context (рис. 3.9).

```
class CoreDataManager {

    static let shared = CoreDataManager()

    let container: NSPersistentContainer
    let context: NSManagedObjectContext

    private init() {
        container = NSPersistentContainer(name: "ToolsApp")
        container.loadPersistentStores { description, error in
            if let error = error {
                print(error.localizedDescription)
            }
        }
        context = container.viewContext
    }
```

Рисунок 3.9. Створення та ініціалізація Core Data контейнера та контексту

В даному застосунку аутентифікація проводиться за електронною поштою та паролем. Виходячи з даної задачі була створена функція login, яка приймає параметром значення email, password та callback-функцію completion, яка повертає тип Result. Спершу створюємо request типу NSFetchRequest<User>, тож ми отримуємо сутність «User», а потім через функціонал предикатів отримуємо тільки того користувача, у якого логін та пароль співпадають з переданими в дану функцію. Операція з знаходження потрібного користувача може дати помилку, у випадку, якщо такого користувача не існує, тому викликаємо даний метод всередині конструкції do/catch, де в свою чергу оброблюємо успішне знаходження користувача або ж помилку (рис. 3.10).

```swift
//MARK: - Auth

func login(email: String, password: String, complition: @escaping (Result<User, Error>) -> ()) {
    let request = NSFetchRequest<User>(entityName: "User")
    request.predicate = NSPredicate(format: "email == %@ AND password == %@", email, password)

    do {
        let users = try context.fetch(request)
        if let user = users.first {
            complition(.success(user))
        }
    } catch {
        complition(.failure(error))
    }
}
```

Рисунок 3.10. Функція аутентифікації користувача

Створення нового користувача у базі даних відбувається в методі createUser. Він приймає параметри, такі як ім'я, прізвище, електронна пошта, пароль, телефонний номер, баланс бонусів, членство у клубі, відсоток кешбеку, та загальну суму покупок. За допомогою контексту CoreData створюється новий запис користувача, який потім зберігається у базі даних (рис. 3.11).

```swift
// MARK: - Creating

func createUser(firstName: String,
                secondName: String,
                email: String,
                password: String,
                phoneNumber: String,
                bonusesBalance: Int64,
                isMemberTventyClub: Bool,
                cashbackPercent: Int64,
                totalPurchases: Int64) {
    let user = User(context: context)

    user.id = UUID()
    user.firstName = firstName
    user.secondName = secondName
    user.email = email
    user.password = password
    user.phoneNumber = phoneNumber
    user.bonusesBalance = bonusesBalance
    user.isMemberTventyClub = isMemberTventyClub
    user.cashbackPercent = cashbackPercent
    user.totalPurchases = totalPurchases

    save()
}
```

Рисунок 3.11. Створення користувача

По аналогії метод createTool та createEvent використовуються для додавання нового інструменту у базу даних. Потім вони зберігаються у контексті CoreData (рис 3.12).

```swift
func createTool(name: String,
                serialNumber: String,
                purchaseDate: Date) -> Tool {
    let tool = Tool(context: context)

    tool.id = UUID()
    tool.name = name
    tool.serialNumber = serialNumber
    tool.purchaseDate = purchaseDate

    save()

    return tool
}

func createEvent(type: String,
                 status: String,
                 place: String,
                 price: Double,
                 item: String) -> Event {
    let event = Event(context: context)

    event.id = UUID()
    event.type = type
    event.status = status
    event.place = place
    event.price = price
    event.item = item

    save()

    return event
}
```

Рисунок 3.12. Створення інструменту та події

Методи updateUserBonuses, updateUserCashbackPercent, та updateUserTotalPurchase призначені для оновлення інформації про користувачів у базі даних. Вони шукають користувача за UUID, і якщо користувач знайдений, оновлюють відповідні дані. Виконується збереження змін у базі даних (рис. 3.13).

```swift
// MARK: - Update

func updateUserBonuses(userId: UUID, to bonuses: Int64) {
    let fetchRequest: NSFetchRequest<User> = User.fetchRequest()
    fetchRequest.predicate = NSPredicate(format: "id == %@", userId as CVarArg)

    do {
        let results = try context.fetch(fetchRequest)
        if let userToUpdate = results.first {
            userToUpdate.bonusesBalance = bonuses
            save()
        } else {
            print("User not found")
        }
    } catch {
        print("Failed to fetch user: \(error)")
    }
}
```

Рисунок 3.13. Оновлення кількості бонусів користувача

У методах updateUserCashbackPercent та updateUserTotalPurchase відмінності будуть лише у назві параметру функції percent та amount та у властивості, яку змінюємо cashbackPercent та totalPurchase.

Для видалення користувачів, інструментів, та подій з бази даних використовуються методи deleteUser, deleteTool, та deleteEvent. Вибрані об'єкти видаляються з контексту і зміни зберігаються.

```
//MARK: - Deleting

func deleteUser(_ user: User) {
    context.delete(user)
    save()
}

func deleteTool(_ tool: Tool) {
    context.delete(tool)
    save()
}

func deleteTool(_ event: Event) {
    context.delete(event)
    save()
}
```

Рисунок 3.14

Також були реалізовані методи для додання інструменту до користувача, додавання події до користувача, з використанням згенерованих функцій addToTool() та addToEvents(); методи для отримання даних з бази даних getAllUsers(), getAllTools(), getAllEvents(), getUserTools(), getUserEvents(), які необхідні для отримання потрібних для відображення даних користувачу.

3.4 Створення ToolViewModel

ViewModel у SwiftUI відіграє ключову роль у паттерні архітектури програмного забезпечення, відомому як Model-View-ViewModel (MVVM). Ця архітектура допомагає відділяти логіку управління даними від користувацького інтерфейсу, тим самим покращуючи читаність коду, його тестування та підтримку. Тому для роботи з Core Data та в цілому з CoreDataManager потрібно реалізувати використання логіки в ToolViewModel для всього проєкту.

```
import SwiftUI

class ToolsViewModel: ObservableObject {

    @AppStorage("loggedInUserID") private var loggedInUserID: String = ""

    private let manager = CoreDataManager.shared

    @Published var users: [User] = []
    @Published var tools: [Tool] = []
    @Published var events: [Event] = []
    @Published var loggedInUser: User?
    @Published var loggedInUserTools: [Tool]?
    @Published var loggedInUserEvents: [Event]?
```

Рисунок 3.15. ToolsViewModel головні властивості

У файлі ToolsViewModel.swift створюємо клас ToolsViewModel (рис. 3.15), який успадковується від протоколу ObservableObject. ObservableObject є одним з ключових протоколів, який дозволяє вашому класу оголошувати себе як джерело правди, стан якого може бути спостережуваним. Коли ви використовуєте клас, що підтримує ObservableObject, ви можете спостерігати за його властивостями і реагувати на їх зміни у вашому користувацькому інтерфейсі.

Також в даному класі реалізовані методи для роботи з CoreDataManager, які в свою чергу викликаються в ініціалізаторі ToolsViewModel та всі необхідні дані зберігаються так само в даному класі, що дає зручність в маніпулюванням

даними в представленнях. Для зручності відстеження користувача, який аутентифікований в застосунку використовується змінна loggedInUserId, що зберігає конкретний ідентифікатор аутентифікованого користувача.

3.5 Навігація застосунку

Інтерфейс застосунка був створений за допомогою фреймворку SwiftUI. Весь інтерфейс створюється в структурах, оскільки вони є набагато швидші, за класи. Великою перевагою саме структур є те, що вони є value type: якщо передати структуру в функцію або присвоїти її іншій змінній, то буде створена копія, на відміну від класу, де створиться посилання на клас.

Так як в застосунку присутній функціонал аутентифікації, то перед кожним запуском перевіряється аутентифікований користувач або ж ще ні, тому в головному класі перевіряється дана змінна і якщо користувач не аутентифікований – відкривається LoginView, якщо ж змінна loggedIn дорівнює true, то відкривається одразу головний екран (рис. 3.16).

```
import SwiftUI

struct ContentView: View {

    @AppStorage("loggedIn") private var loggedIn = false

    var body: some View {
        Group {
            loggedIn ? AnyView(TabScreenView()) : AnyView(LoginView())
        }
    }
}
```

Рисунок 3.16. Керування відображенням головного екрану

Перегляд TabScreenView (рис. 3.17) відкриється у випадку успішної аутентифікації користувача. Даний перегляд відповідає за навігацію застосунку

і за замовчуванням відкривається HomeView, а також є можливість переключатись між екранами, зокрема на ToolsView, ToolRegistrationView та ProfileView.

```
struct TabScreenView: View {

    @State private var selectedTab: Tab = .home

    var body: some View {
        TabView(selection: tabSelection()) {

            HomeView()
                .tabItem {
                    Image(systemName: "house")
                }
                .tag(Tab.home)

            ToolsView()
                .tabItem {
                    Image(systemName: "list.dash")
                }
                .tag(Tab.tools)

            ToolRegistrationView()
                .tabItem {
                    Image(systemName: "barcode.viewfinder")
                }
                .tag(Tab.toolRegistration)

            ProfileView()
                .tabItem {
                    Image(systemName: "person")
                }
                .tag(Tab.settings)
        }
    }
}
```

Рисунок 3.17. Навігація застосунку

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі було описано процес створення застосунку на базі фреймворку SwiftUI в інтегрованому середовищі розробки XCode та розібрані всі налаштування проєкту.

Також було детально розібрано процес створення та налаштування моделей в базі даних CoreData. Були задані всі необхідні поля для моделей та прописані взаємозв’язки між ними, для зручного використання в застосунку для відображення та зберігання даних.

Розглянутий в третьому розділі клас CoreDataManager був створений для налаштування контексту та контейнера CoreData і було реалізовано основні CRUD методи для роботи з базою даних в застосунку.

Наступним кроком була створена ViewModel для відокремлення логіки роботи з базою даних від створення користувацького інтерфейсу.

Додатково було розглянуто логіку навігації в застосунку, а саме як відбувається відслідковування статусу аутентифікованого користувача та навігація між екранами.

РОЗДІЛ 4. ОГЛЯД МОЖЛИВОСТЕЙ ЗАСТОСУНКУ

В додатку реалізовано функцію аутентифікації користувача за електронною поштою та паролем. На екрані реалізоване текстове поле для вводу електронної пошти та паролю, а також кнопка «Login», яка буде активна тільки після того, якщо поля «email» та «password» не пусті (рис. 4.1). Коли кнопка була натиснута і текстові поля заповнені, викликається метод loginUser у ViewModel, який повертає булеве значення і присвоюється змінній loggedIn. Змінна loggedIn використовується для відслідковування стану застосунку і в залежності від цієї змінної користувачу відображається екран авторіації, або ж головний екран застосунку.

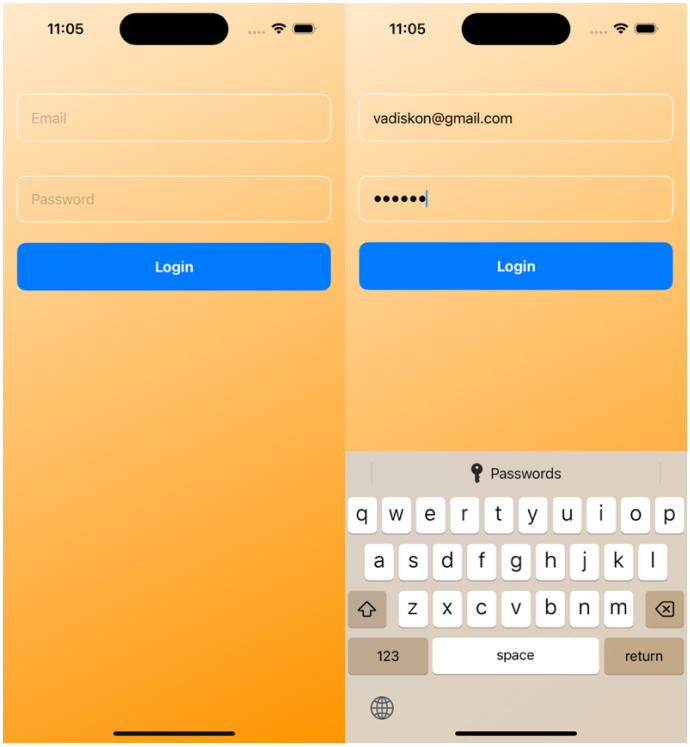


Рисунок 4.1. Екран авторизації

Коли користувач авторизувався, то відкривається головний екран застосунку. Згори відображається інформація про користувача, така як:

- Повне ім’я
- Шкала загальної суми покупок, яка відображається в відсотковому співвідношенні до доступної для максимальної бонусної ставки
- Відсоток бонусів – відсоток кєшбеку від покупок неакційних товарів
- На балансі – доступні бонуси на бонусному рахунку користувача, які він може використати при наступній покупці

Нижче розташований список з останніми подіями, які відображаються від нових до старих. В карточках відображається інформація про:

- Тип події – може бути «Купівля», «Сервіс», «Реєстрація інструменту»
- Статус – він відображається кольоровою вертикальною рискою зліва
- Назва інструменту
- Місце в якому сталась подія – може бути «Застосунок», адреса магазину
- Сума від покупки або ж за сервісне обслуговування

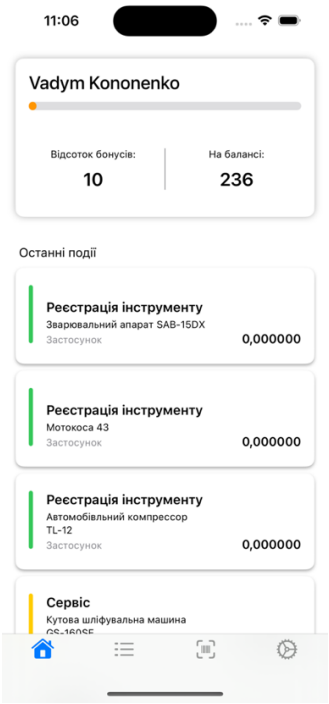


Рисунок 4.2. Головний екран

Наступна вкладка в TabBar – це екран відображення всіх інструментів, зареєстрованих за користувачем (рис. 4.3). Тут відображаються карточки всіх інструментів з додатковою інформацією про нього:

- Кількість днів до закінчення гарантії на інструмент, які для зручності користувача мають зелений або ж помаранчевий колір. Гарантія вже закінчилась – помаранчевий колір, якщо ще на гарантії – зелений. Гарантія складає 3 роки.
- Дата покупки
- Назва
- Серійний номер

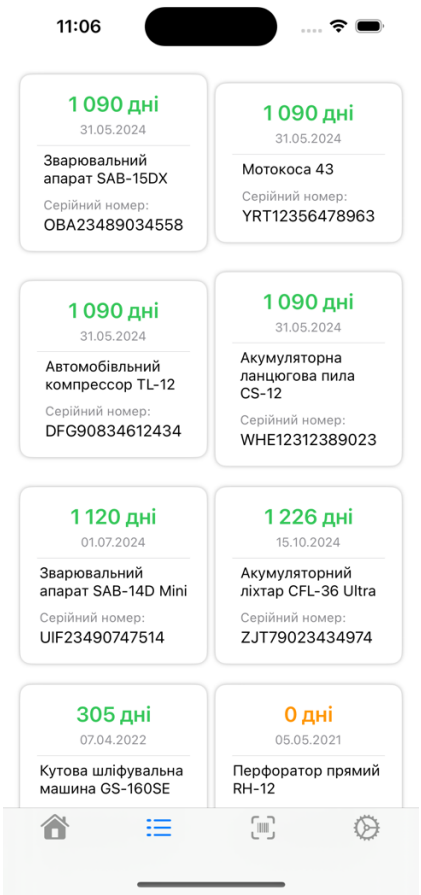


Рисунок 4.3. Екран з зареєстрованими інструментами

Екран реєстрації інструменту необхідний користувачу для того, щоб якщо при покупці інструмент не був зареєстрований за ним або ж якщо користувач придбав вживаний інструмент, то в нього є змога зареєструвати його за собою. Відкриваючи екран спочатку відображається поле вводу серійного номеру інструменту та кнопка «Далі». Коли користувач ввів серійний номер інструменту в текстове поле, натискаючи кнопку «Далі» спрацьовує скрипт який спершу перевіряє чи текстове поле заповнене та довжину серійного номеру, яка повинна бути 14 символів.

Після того, як пройшла успішна валідація, відкривається наступна частина, в якій користувач вводить (рис. 4.4):

- Назву інструменту та модель
- Дату придбання
- Кнопка «Зареєструвати»

Натискаючи на кнопку «Зареєструвати» спрацьовує скрипт валідації полів назви інструменту та серійного номеру. В разі успішної валідації виконується:

- Метод додавання інструменту користувачу (addToolToUser), який передаються значення полів назви, серійний номер та дата реєстрації.
- Метод створення та додавання події користувача (addEventToUser), який передаються значення типу події, статус, місце, ціна та інструмент.
- Присвоювання змінній showSuccessInfo позитивного значення
- Через 2 секунди виконується очищення полів та булевих значень на false

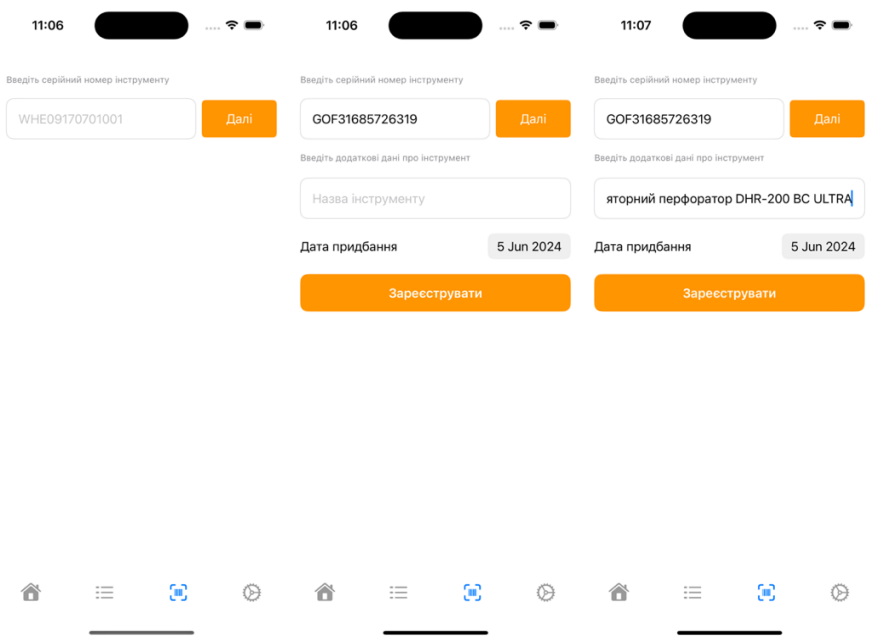


Рисунок 4.4. Екран реєстрації інструменту

Задля інформування користувача про успішність реєстрації додано символ «✓» зеленого кольору (рис. 4.5), який відображається коли змінна showSuccessInfo дорівнює true. Він відображається зліва від текстового поля з серійним номером, що дозволяє користувачу зрозуміти, що реєстрація пройшла успішно.

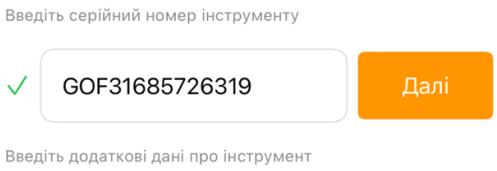


Рисунок 4.5. Індикатор успішної реєстрації

Екран профілю користувача є четвертим і останнім, він потрібен для користувача, щоб він міг легко і в одному місці побачити інформацію про себе та провести налаштування профілю (рис. 4.6). На даному екрані присутня кнопка виходу з аккаунту користувача, яка при натисканні викликає функцію у

ViewModel logoutUser() та присвоює змінній loggedIn значення false, після чого одразу відкривається екран аутентифікації.

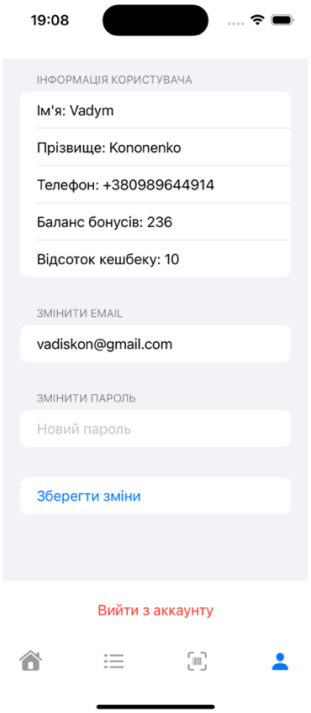


Рисунок 4.6. Екран профілю користувача

Тож розглянемо більш детально функціонал зміни паролю. В формі реалізована секція «Змінити пароль», яка має текстове поле, в яке користувач вводить новий пароль. Пароль зміниться, якщо поле не пусте, має більше 4 символів та не співпадає з старим паролем. Натиснувши кнопку «Зберегти зміни» проводиться валідація параметрів і якщо все успішно, то викликається функція оновлення змін в CoreData. Для інформування користувача про успішність операції було додано булеву змінну, яка відповідає за відображення тексту про успішність операції (рис. 4.7).

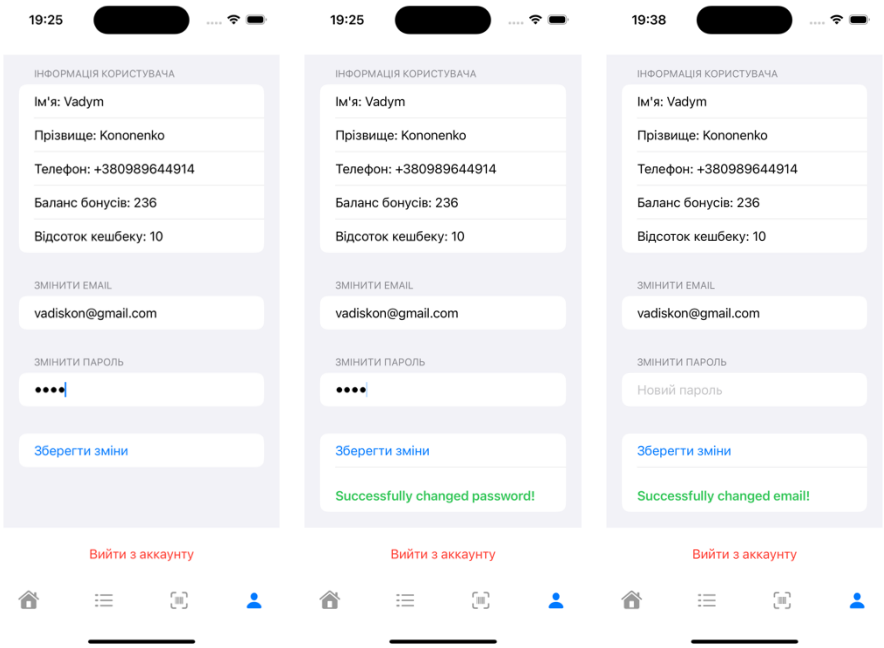


Рисунок 4.7. Зміна паролю та електронної адреси

По аналогії з зміною паролю змінюється і електронна адреса. При натисканні на кнопку «Зберегти зміни» проводиться валідація поля: поле не пусте і нова електронна адреса відрізняється від старої. У випадку успішності зміни – відображається текст для користувача, який інформує його про успішність даної операції (рис. 4.7).

ВИСНОВОК ДО РОЗДІЛУ 4

В цьому розділі була детально розглянута робота застосунку та протестовані всі головні переваги та функції застосунку. Відображення даних користувача з бази даних та функції запису і оновлення даних працюють правильно. Для користувача реалізована зручна навігація та окремі екрани, які є інтуїтивно зрозумілими, що робить застосунок комфортним для кінцевого споживача.

ВИСНОВКИ

У результаті виконання дипломної роботи було розроблено мобільний додаток для магазину інструментів та обладнання, який базується на фреймворку SwiftUI та використовує базу даних CoreData. Основною метою роботи було підвищення продажів магазину за допомогою надання зручного інструменту для відслідковування покупок, управління гарантійними зобов'язаннями та створення заявок на сервісне обслуговування.

У процесі роботи було виконано детальний аналіз предметної області, вивчено існуючі рішення на ринку, обрано оптимальні технології для реалізації проекту. Було розглянуто переваги та недоліки фреймворків UIKit, SwiftUI, Flutter, React Native та Xamarin, а також систем управління базами даних CoreData, Realm та Firebase.

Розробка додатку включала створення та налаштування моделей бази даних, розробку користувацького інтерфейсу, реалізацію основних функцій (аутентифікація, реєстрація інструментів, відслідковування гарантійних термінів, створення сервісних заявок) та налаштування навігації в додатку. Було реалізовано інтуїтивно зрозумілий інтерфейс, який забезпечує зручне користування додатком та підвищує задоволеність клієнтів.

Додаток успішно протестовано, перевірено коректність відображення та запису даних, зручність навігації та користування. Реалізовані функції відповідають вимогам до проекту та забезпечують необхідний функціонал для користувачів.

Таким чином, поставлені завдання дипломної роботи виконані, мета досягнута, розроблений додаток сприятиме підвищенню продажів та лояльності клієнтів магазину інструментів та обладнання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Офіційна документація мови програмування Swift [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/swift/
2. Огляд переваг та недоліків мови програмування Swift [Електронний ресурс] – Режим доступу до ресурсу: https://helpshift.com/glossary/messenger-app/
3. Порівняння мови програмування Swift та Objective-C [Електронний ресурс] – https://www.mindk.com/blog/swift-vs-objectivec/
4. Порівняння фреймворку SwiftUI та UIKit [Електронний ресурс] – https://www.mindk.com/blog/swift-vs-objectivec/
5. Огляд переваг та недоліків фреймворків SwiftUI та UIKit [Електронний ресурс] – https://sendbird.com/developer/tutorials/swiftui-vs-uikit
6. Огляд бібліотеки SnapKit [Електронний ресурс] – https://www.waldo.com/blog/snapkit-introductory-guide
7. Історія еволюції нового фреймворку SwiftUI [Електронний ресурс] – https://www.genui.com/resources/swiftui-and-apple-layout-paradigms
8. Переваги фреймворку SwiftUI [Електронний ресурс] – https://medium.com/@icodelabs/explore-the-advantages-and-disadvantages-of-swiftui-and-uikit-for-ios-app-development-215e5bb748fc
9. Переваги та недоліки фреймворку Flutter [Електронний ресурс] – https://www.intelivita.com/resources/what-is-flutter/
10. Огляд фреймворку Flutter [Електронний ресурс] – https://www.fullstack.com/labs/resources/blog/an-introduction-to-flutters-world
11. Переваги та недоліки використання фреймворку ReactNative [Електронний ресурс] – https://techexactly.com/blogs/advantages-and-disadvantages-of-using-react-native

12. Огляд фреймворку Flutter та його переваг [Електронний ресурс] – https://www.grazitti.com/blog/7-benefits-of-using-react-native-for-mobile-app-development/
13. Загальна інформація про фреймворк CoreData [Електронний ресурс] – https://en.wikipedia.org/wiki/Core_Data
14. Офіційна документація CoreData від Apple [Електронний ресурс] – https://developer.apple.com/documentation/coredata/creating_a_core_data_model

Додаток 1

Мобільний додаток для магазину інструментів та обладнання

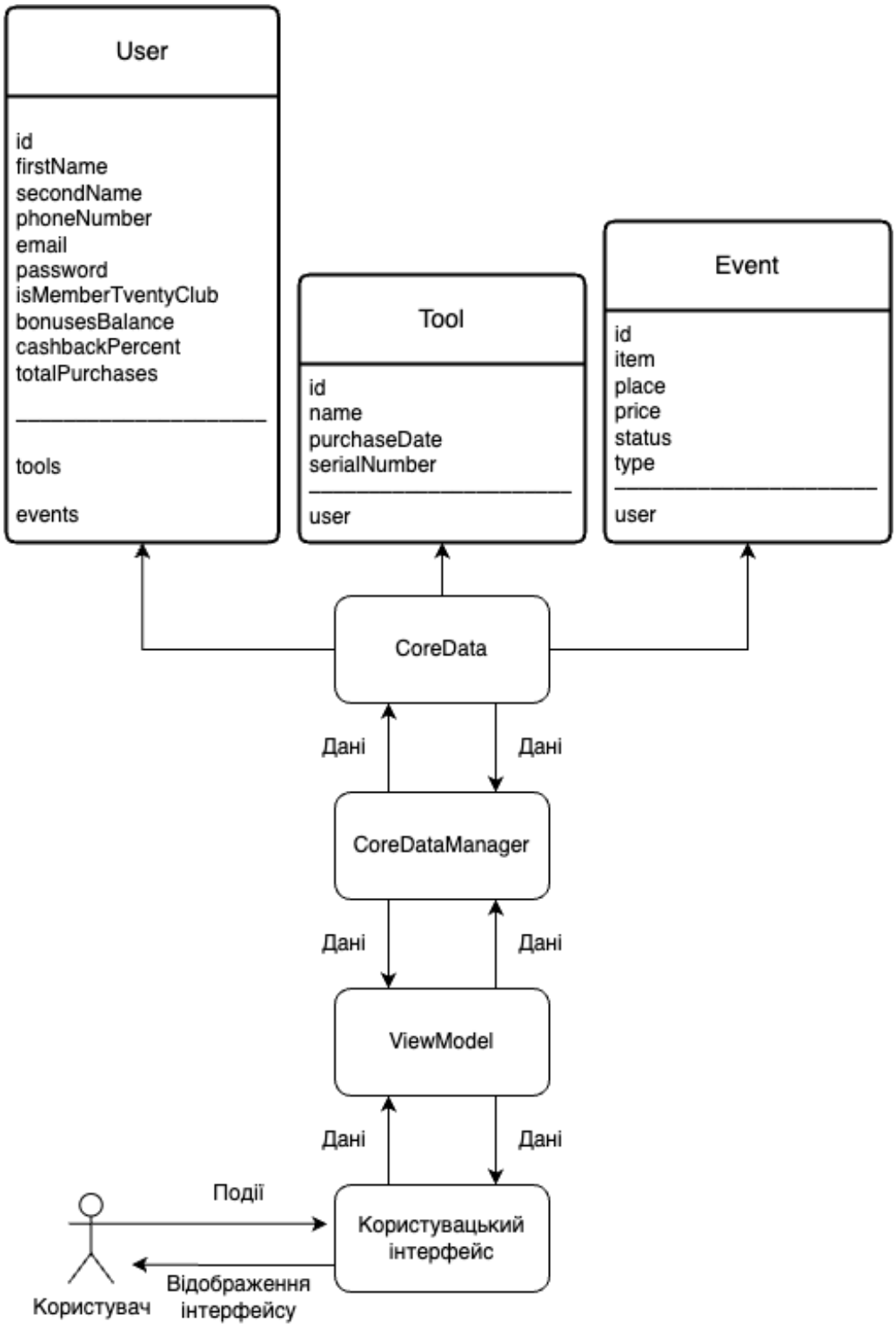
Діаграма класів

(Функціональна схема)

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2024 р



					ІАЛЦ.467200.004 Д1			
Зм.	Арк.	№ докум.	Підпис	Дата				
Розробив		Кононенко В.О.			*Мобільний додаток для магазину інструментів та обладнання* *Функціональна схема*	Літ.	Аркуш	Аркушів
Перевірив		Шульга М.В.					1	
Реценз.		Матвійчук О.В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІП-05		
Н. Контр.		Волокита А.М.						
Затвердив		Стіренко С.Г.						

Додаток 2

Мобільний додаток для магазину інструментів та обладнання

Алгоритм дії програмного забезпечення

(Принципова схема)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2024 р

					ІАЛЦ.467200.005 Д2		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Кононенко В.О.			*Мобільний додаток для магазину інструментів та обладнання* *Принципова схема*	Літ.	Аркуш 1 / Аркушів
Перевірив		Шульга М.В.					
Реценз.		Матвійчук О.В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІП-05	
Н. Контр.		Волокита А.М.					
Затвердив		Стіренко С.Г.					

Додаток 3

Мобільний додаток для магазину інструментів та обладнання

Структура системи

(Структурна схема)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2024 р

					ІАЛЦ.467200.006 Д3		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Кононенко В.О.			*Мобільний додаток для магазину інструментів та обладнання* *Структурна схема*	Літ.	Аркуш 1 / Аркушів
Перевірив		Шульга М.В.					
Реценз.		Матвійчук О.В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІП-05	
Н. Контр.		Волокита А.М.					
Затвердив		Стіренко С.Г.					

Додаток 4

Мобільний додаток для магазину інструментів та обладнання

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 9

Київ 2024 р

```
struct LoginView: View {
    @AppStorage("loggedIn") private var loggedIn = false
    @EnvironmentObject private var viewModel: ToolsViewModel
    @State private var email: String = ""
    @State private var password: String = ""

    var body: some View {
        ZStack {
            LinearGradient(colors: [Color.orange.opacity(0.2),
Color.orange.opacity(0.6) ,.orange], startPoint: .topLeading, endPoint:
.bottomTrailing)
                .ignoresSafeArea()

            VStack {
                TextField("Email", text: $email)
                    .background(Color.clear)
                    .padding()
                    .overlay {
                        RoundedRectangle(cornerRadius: 10)
                            .stroke(Color.white, lineWidth: 1)
                    }
                    .padding()
                    .autocorrectionDisabled(true)
                    .textInputAutocapitalization(.never)

                SecureField("Password", text: $password)
                    .background(Color.clear)
                    .padding()
                    .overlay {
                        RoundedRectangle(cornerRadius: 10)
```

```
                    .stroke(Color.white, lineWidth: 1)
                }
                .padding()
                .autocorrectionDisabled(true)
                .textInputAutocapitalization(.never)

            Button("Login") {
                guard !email.isEmpty, !password.isEmpty else { return }

                if viewModel.loginUser(email: email, password: password) {
                    loggedIn = viewModel.loginUser(email: email, password:
password)
                }
            }
            .foregroundColor(.white)
            .fontWeight(.bold)
            .frame(height: 55)
            .frame(maxWidth: .infinity)
            .background(Color.blue)
            .cornerRadius(10)
            .padding(.horizontal)
            Spacer()
        }
        .padding(.top, 30)
    }
  }
}
struct TabScreenView: View {
  @State private var selectedTab: Tab = .home
  var body: some View {
```

```
        TabView(selection: tabSelection()) {

            HomeView()
                .tabItem {
                    Image(systemName: "house")
                }
                .tag(Tab.home)

            ToolsView()
                .tabItem {
                    Image(systemName: "list.dash")
                }
                .tag(Tab.tools)

            ToolRegistrationView()
                .tabItem {
                    Image(systemName: "barcode.viewfinder")
                }
                .tag(Tab.toolRegistration)

            ProfileView()
                .tabItem {
                    Image(systemName: "person ")
                }
                .tag(Tab.settings)
        }
    }
}
struct ToolRegistrationView: View {
    @EnvironmentObject private var viewModel: ToolsViewModel
    @State private var serialNumber: String = ""
    @State private var toolName: String = ""
```

```
@State private var purchaseDate = Date()
@State private var showNextInfo: Bool = false
@State private var showSuccessInfo: Bool = false

var body: some View {
    VStack(alignment: .leading, spacing: 20) {
        Text("Введіть серійний номер інструменту")
            .font(.caption)
            .foregroundColor(.gray)

        HStack {
            if showSuccessInfo {
                Image(systemName: "checkmark")
                    .foregroundColor(.green)
            }

            TextField("WHE09170701001", text: $serialNumber)
                .padding()
                .background(Color.white)
                .cornerRadius(10)
                .overlay(
                    RoundedRectangle(cornerRadius: 10)
                        .stroke(Color.gray.opacity(0.3), lineWidth: 1)
                )

            Button(action: {
                if !serialNumber.isEmpty, serialNumber.count == 14 {
                    showNextInfo = true
                }
            }) {
                Text("Далі")
                    .frame(maxWidth: 100, maxHeight: .infinity)
```

```
                .foregroundColor(.white)
                .background(Color.orange)
                .cornerRadius(5)
            }
        }
        .frame(height: 50)

        if showNextInfo {
            Text("Введіть додаткові дані про інструмент")
                .font(.caption)
                .foregroundColor(.gray)

            TextField("Назва інструменту", text: $toolName)
                .padding()
                .background(Color.white)
                .cornerRadius(10)
                .overlay(
                    RoundedRectangle(cornerRadius: 10)
                        .stroke(Color.gray.opacity(0.3), lineWidth: 1)
                )

            DatePicker("Дата придбання", selection: $purchaseDate,
displayedComponents: .date)

            Button("Зареєструвати") {
                if !toolName.isEmpty, !serialNumber.isEmpty {
                    viewModel.addToolToUser(name: toolName, serialNumber:
serialNumber, purchaseDate: purchaseDate)
                    viewModel.addEventToUser(type: "Реєстрація інструменту", status:
"success", place: "Застосунок", price: 0, item: toolName)
                    showSuccessInfo = true
```

```
                    DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
                        self.showNextInfo = false
                        self.serialNumber = ""
                        self.showSuccessInfo = false
                    }
                }
            }
            .foregroundColor(.white)
            .font(.headline)
            .frame(maxWidth: .infinity, minHeight: 50)
            .background(Color.orange)
            .cornerRadius(10)
        }

        Spacer()
    }
    .navigationTitle("Реєстрація інструменту")
    .padding(.top, 20)
    .padding()
    .onAppear {
        showNextInfo = false
        serialNumber = ""
        showSuccessInfo = false
    }
  }
}
class CoreDataManager {

    static let shared = CoreDataManager()

    let container: NSPersistentContainer
    let context: NSManagedObjectContext
```

```
    private init() {
        container = NSPersistentContainer(name: "ToolsApp")
        container.loadPersistentStores { description, error in
            if let error = error {
                print(error.localizedDescription)
            }
        }
        context = container.viewContext
    }
    //MARK: - Auth
    func login(email: String, password: String, complition: @escaping (Result<User,
Error>) -> ()) {
        let request = NSFetchRequest<User>(entityName: "User")
        request.predicate = NSPredicate(format: "email == %@ AND password ==
%@", email, password)

        do {
            let users = try context.fetch(request)
            if let user = users.first {
                complition(.success(user))
            }
        } catch {
            complition(.failure(error))
        }
    }
    // MARK: - Creating
    func createUser(firstName: String,
                    secondName: String,
                    email: String,
                    password: String,
                    phoneNumber: String,
```

```
            bonusesBalance: Int64,
            isMemberTventyClub: Bool,
            cashbackPercent: Int64,
            totalPurchases: Int64) {
    let user = User(context: context)

    user.id = UUID()
    user.firstName = firstName
    user.secondName = secondName
    user.email = email
    user.password = password
    user.phoneNumber = phoneNumber
    user.bonusesBalance = bonusesBalance
    user.isMemberTventyClub = isMemberTventyClub
    user.cashbackPercent = cashbackPercent
    user.totalPurchases = totalPurchases

    save()
}

func createTool(name: String,
            serialNumber: String,
            purchaseDate: Date) -> Tool {
    let tool = Tool(context: context)

    tool.id = UUID()
    tool.name = name
    tool.serialNumber = serialNumber
    tool.purchaseDate = purchaseDate

    save()
```

```
    return tool
}

func createEvent(type: String,
                 status: String,
                 place: String,
                 price: Double,
                 item: String) -> Event {
    let event = Event(context: context)

    event.id = UUID()
    event.type = type
    event.status = status
    event.place = place
    event.price = price
    event.item = item

    save()

    return event
}
// MARK: - Update
func updateUserBonuses(userId: UUID, to bonuses: Int64) {
    let fetchRequest: NSFetchRequest<User> = User.fetchRequest()
    fetchRequest.predicate = NSPredicate(format: "id == %@", userId as CVarArg)

    do {
        let results = try context.fetch(fetchRequest)
        if let userToUpdate = results.first {
            userToUpdate.bonusesBalance = bonuses
            save()
        } else {
```

```
            print("User not found")
        }
    } catch {
        print("Failed to fetch user: \(error)")
    }
}

func updateUserCashbackPercent(userId: UUID, to percent: Int64) {
    let fetchRequest: NSFetchRequest<User> = User.fetchRequest()
    fetchRequest.predicate = NSPredicate(format: "id == %@", userId as CVarArg)

    do {
        let results = try context.fetch(fetchRequest)
        if let userToUpdate = results.first {
            userToUpdate.cashbackPercent = percent
            save()
        } else {
            print("User not found")
        }
    } catch {
        print("Failed to fetch user: \(error)")
    }
}

func updateUserTotalPurchase(userId: UUID, to amount: Int64) {
    let fetchRequest: NSFetchRequest<User> = User.fetchRequest()
    fetchRequest.predicate = NSPredicate(format: "id == %@", userId as CVarArg)

    do {
        let results = try context.fetch(fetchRequest)
        if let userToUpdate = results.first {
            userToUpdate.totalPurchases = amount
```

```
            save()
        } else {
            print("User not found")
        }
    } catch {
        print("Failed to fetch user: \(error)")
    }
}
// MARK: - Adding tool to user
func addToolToUser(_ user: User, name: String, serialNumber: String,
purchaseDate: Date) {
    let tool = createTool(name: name, serialNumber: serialNumber, purchaseDate:
purchaseDate)

    user.addToTools(tool)

    save()
}

func addEventToUser(_ user: User,
                    type: String,
                    status: String,
                    place: String,
                    price: Double,
                    item: String) {
    let event = createEvent(type: type, status: status, place: place, price: price, item:
item)

    user.addToEvents(event)

    save()
}
```

```
//MARK: - Fetching
func getAllUsers() -> [User]? {
    let request = NSFetchRequest<User>(entityName: "User")

    do {
        return try context.fetch(request)
    } catch {
        print("Error fetching: \(error.localizedDescription)")
        return nil
    }
}

func getAllTools() -> [Tool]? {
    let request = NSFetchRequest<Tool>(entityName: "Tool")

    do {
        return try context.fetch(request)
    } catch {
        print("Error fetching: \(error.localizedDescription)")
        return nil
    }
}

func getAllEvents() -> [Event]? {
    let request = NSFetchRequest<Event>(entityName: "Event")

    do {
        return try context.fetch(request)
    } catch {
        print("Error fetching: \(error.localizedDescription)")
        return nil
    }
```

```
    }

    func getUsersTools(userID: String) -> [Tool]? {
        let userRequest = NSFetchRequest<User>(entityName: "User")

        if let userUUID = UUID(uuidString: userID) {
            userRequest.predicate = NSPredicate(format: "id == %@", userUUID as
NSUUID)
        } else {
            print("Invalid UUID string: \(userID)")
            return nil
        }

        let toolRequest = NSFetchRequest<Tool>(entityName: "Tool")

        do {
            if let user = try context.fetch(userRequest).first {
                toolRequest.predicate = NSPredicate(format: "user == %@", user)

                return try context.fetch(toolRequest)
            }
            return nil
        } catch {
            print("Error fetching: \(error.localizedDescription)")
            return nil
        }
    }

    func getUsersEvents(userID: String) -> [Event]? {
        let userRequest = NSFetchRequest<User>(entityName: "User")

        if let userUUID = UUID(uuidString: userID) {
```

```
            userRequest.predicate = NSPredicate(format: "id == %@", userUUID as
NSUUID)
        } else {
            print("Invalid UUID string: \(userID)")
            return nil
        }

        let eventRequest = NSFetchRequest<Event>(entityName: "Event")

        do {
            if let user = try context.fetch(userRequest).first {
                eventRequest.predicate = NSPredicate(format: "user == %@", user)

                return try context.fetch(eventRequest)
            }
            return nil
        } catch {
            print("Error fetching: \(error.localizedDescription)")
            return nil
        }
    }
    //MARK: - Deleting
    func deleteUser(_ user: User) {
        context.delete(user)
        save()
    }

    func deleteTool(_ tool: Tool) {
        context.delete(tool)
        save()
    }
```

```
    func deleteTool(_ event: Event) {
        context.delete(event)
        save()
    }
    //MARK: - Saving
    func save() {
        if context.hasChanges {
            do {
                try context.save()
            } catch {
                print("Error with saving data: \(error)")
                context.rollback()
            }
        }
    }
}
```