КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА Факультет комп'ютерних наук та кібернетики Кафедра математичної інформатики

Дипломна робота

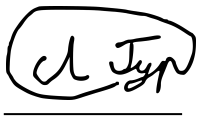
за спеціальністю 122 Комп'ютерні науки

на тему:

Розробка мобільного додатку для фітнесу під IOS

Виконав студент 4 курсу

Гуртовий Леонід Дмитрович



(підпис)

Науковий керівник:

доцент кафедри математичної інформатики, кандидат фіз.-мат. наук

Деревянченко Олександр Валерійович

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

(підпис)

КИЇВ 2021

РЕФЕРАТ

Робота складається зі вступу, 5 розділів, висновків, списку використаних джерел (7 найменувань) та додатку. Робота містить 24 рисунки, 2 таблиці. Загальний обсяг становить 52 сторінок, основний текст роботи викладено на 42 сторінках.

Темою роботи є “Розробка мобільного додатку для фітнесу під IOS” Метою роботи є розробка iOS додатку для підбору плану тренувать для користувача на основі його персональних даних.

Інструменти розроблення: безкоштовне, вільно поширюване інтегроване середовище розробки Xcode, що було розроблене Apple, написаний на мові Swift.

Результати роботи: розроблено додаток для фітнесу, що персонально підлаштовується до кожного користувача та надає йому програму для занять спортом, враховуючи персональні параметри. Також проведене Automation тестування.

Зміст

ВСТУП

Сьогодні у світі майже в кожної сучасної людини є смартфон. Кожен день люди все частіше роблять в телефоні те, що раніше робили в офлайні: замовляють їжу, роблять покупки в інтернет-магазинах, спілкуються по роботі тощо. Не виключенням стали і фітнес додатки. Оцінка сучасного стану об’єкта дослідження або розробки. На даний момент існує багато мобільних додатків для фітнесу. Найвідомішими є BetterMe, NTC(Nike training club), 8fit, AsanaRebel.

Актуальність роботи та підстави для її виконання. Сьогодні додатки для спорту є особливо актуальними через пандемію коронавіруса. Значний ріст їх популярності у всьому світі зумовлений періодичним закриттям всіх громадських місць, включаючи спортивні секції та фітнес центри. Людина, яка раніше купляла абонемент на рік до спортклубу, сьогодні більш ймовірно вибере тренуватися вдома, з декількох причин:

- Тренування вдома не залежить від ситуації в навколишньому світі, тобто такі аномалії як пандемія не можуть завадити людині проводити час вдома так, як вона хоче.
- Це дешевше, ніж відвідування спортклубу.
- Потребує мінімальних витрат часу (не потрібно витрачати час на те, щоб дістатися до спортивного клубу).

Мета роботи. Метою роботи є розробка iOS додатку для підбору плану тренувать для користувача на основі його персональних даних.

Об’єкт, методи й засоби дослідження або розроблення. Проект розроблено за допомогою нативної мови програмування Swift, розробленої компанією Apple. Було використано такі архітектурні патерни як Clean Architecture + MVVM.

Можливі сфери застосування. Практичною сферою застосування додатку є персональне користування будь-якою людиною, що має iPhone.

Взаємозв'язок з іншими роботами. При розробці проекту за основу деяких складових було взято найпопулярніший на ринку додаток BetterMe.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАДАЧІ

1.1 Загальні відомості про предметну область

Основною причиною потреби створення додатку на обрану тематику є різкий ріст людей, які займаються фітнесом вдома, у зв’язку із пандемією:

- Під час локдауну тренажерні зали та фітнес-студії були вимушені закрити або перейти в цифровий режим.
- Завантаження програм для здоров'я та фітнесу зросло на 46% у всьому світі.
- Найбільший приріст завантажень спостерігався в Індії - зріс на 156%.

Зростання популярності додатків для домашнього фітнесу.

Програми для домашнього фітнесу формували наше життя - задовго до COVID-19, але їхня популярність справді зросла в період пандемії.
Які регіони рухають це зростання? І чи залишиться домашній фітнес чи він зникне, коли пандемія закінчиться?
Далі будуть наведені дані за період з початку 2020 року (до пандемії) до осені 2020 року.

Зростання за кількістю завантажень.

З 1 по 2 квартал 2020 року кількість завантажень додатків для здоров’я та фітнесу зросла на 46% у всьому світі. Регіональний розподіл зростання кількості завантажень фітнес додатків зображено на рисунку 1.

Region	Download Growth
India	157%
MENA	55%
Europe	25%
Asia-Pacific	47%
Rest of the World	43%
Americas	21%

Рисунок 1 – Таблиця зростання популярності за кількістю завантажень фітнес додатків по різним частинам світу

Найбільший приріст завантажень спостерігався в Індії, який виріс на 156%. Це означає 58 мільйонів нових активних користувачів - майже все населення Італії.

Регіон Близького Сходу та Північної Африки (MENA) зафіксував друге місце за зростанням завантажень із зростанням на 55%, а потім Азіатсько-Тихоокеанський регіон із зростанням на 47%.

Зростання за щоденними активними користувачами.

Поряд зі зростанням кількості завантажень, додатки для фітнесу також збільшили кількість щоденних активних користувачів (DAU). Це важливо, оскільки показує, що люди не просто завантажували ці програми і забували про них - вони насправді їх використовували.

У всьому світі DAU для додатків для фітнесу з 1 до 2 кварталу зросли на 24%. Зростання популярності за щоденними активними користувачами в кожному регіоні видно з рисунку 2.

Region	DAU Growth
India	84%
MENA	28%
Europe	11%
Asia-Pacific	24%
Rest of the World	24%
Americas	8%

Рисунок 2 – Таблиця зростання популярності за щоденними активними користувачами по різним частинам світу

Як і у випадку із завантаженнями, в Індії спостерігається найбільший приріст DAU із збільшенням на 84%, або 12 млн нових користувачів. MENA зафіксував друге за зростанням зростання (яке також відповідає завантаженню) на рівні 28%.

На противагу цьому в Америці спостерігалося найменше зростання— лише 8% - але це також може бути функцією збільшення кількості користувачів для початку. Незважаючи на нижчий темп зростання, ніж в інших регіонах, можна зробити висновок, що люди цікавляться додатками для фітнесу, особливо в США.

Згідно з опитуванням OnePoll[1], проведеним в середині року, 74% американців користувалися принаймні одним додатком для фітнесу під час карантину, а 60% так сподобалися домашнім тренуванням, що зараз вони планують назавжди скасувати членство в спортзалі.[2]

1.2 Методологія створення

1) Мозковий штурм і планування

Все починається з крутої ідеї. Звичайно вона повинна бути продуманою, щоб потім реалізувати її. Ретельне планування є одним із

основних етапів розробки програмного забезпечення. Тільки після того як план розроблений можна рухатись далі.

2) Вимоги та аналіз

На цьому етапі розробки проект детально визначається та виконується аналіз. Спочатку треба зрозуміти мету для подальшого створення продукту та дизайну. На основі аналізу потреб визначити фактори, які будуть найкращим чином служити додатку. Передивитися тенденції на ринку, щоб переконатися, що до наявності новітньої технології, компонентів і елементів. Також треба виділити час для нестандартного мислення.

3) Дизайн

На етапі проектування створюємо актуальну концептуалізацію рішення, тобто детальну архітектуру програмного забезпечення. Цей етап включає в себе побудову структуру проекту з використанням остаточних прототипів, які на далі будуть використовуватись в проекті. Для мобільних додатків візуальне моделювання – це найголовніший компонент, починаючи від функціональності рішень і закінчуючи визначенням основних апаратних та програмних компонентів для подальшої реалізації цілей. Після цього можна приступити до самого розвитку проекту.

4) Інтеграція і тестування

Починається етап забезпечення інтеграції і тестування системи. Після реалізації коду проводиться тестування. Зазвичай використовується цілий ряд платформ для виконання модульних тестів, автоматизації компіляціїї та тестування. Це робиться для переконання в чистоті коду та досягнення цілей. Тепер, коли програмне забезпечення не містить помилок, починається фаза реалізації.

5)Впровадження

Це етап, коли відбувається установка створеного проекту. Це робиться крок за кроком відповідно до плану реалізації. Продукт переноситься у виробництво, впроваджуючи далі тільки певні зміни.

1.3 Вибір мови програмування для розробки продукту

Важливим етапом перед початком розробки є вибір мови програмування.

Існує два кардинально різні підходи до розробки мобільних додатків: кросплатформений та нативний.

- **кросплатформений підхід**

Кросплатформеність – здатність програмного забезпечення працювати більш, ніж на одній апаратній платформі і (або) операційній системі. Забезпечується завдяки використанню високорівневих мов програмування, середовищ розробки і виконання, що підтримують умовну компіляцію, компоновку і виконання коду для різних платформ. У нашому випадку розробки мобільного додатку кросплатформеність полягає у розробці спільного продукту під iOS та Android.

На сьогоднішній день є 2 популярних кросплатформених мови програмування для розробки мобільних додатків:

1) JavaScript – прототипно орієнтована мова програмування. Найбільш широке застосування знайшла в браузерах як мова сценаріїв для додання інтерактивності веб-сторінок. Для розробки мобільних додатків використовується фреймворк React Native
2) C# - об’єктно-орієнтована мова програмування. Найбільш широке застосування знайшла як мова розробки продуктів для платформи Microsoft на .NET Framework. Для розробки мобільних додатків використовується фреймворк Xamarin[3]

Порівняльна характеристика наведена у таблиці 1.1.

Мова програмування	Плюси	Мінуси
JavaScript React Native	1) Спрощений інтерфейс Фреймворк відмінно інтегрується з JavaScript, що робить його фреймворком з відкритим вихідним кодом. 2) Модульна архітектура React Native слід модульній архітектурі. Це забезпечує кращу стабільність усього процесу створення програми. 3) Підтримка сторонніх плагінів Це необхідно, оскільки деякі компоненти відсутні в реальній структурі ReactNative.	1) Складний користувацький інтерфейс Якщо ви плануєте розробляти програми з такими функціями, як переходи між екранами, складні жести і навіть анімації, React Native може бути не кращим варіантом для розгляду. Використання уніфікованого API здається складним завданням для React Native. 2) Управління пам’яттю React Native інтегрований з JavaScript. Так що це може бути не найкращий варіант.

C# Xamarin	1) Спільний код для iOS та Android Код, написаний на Xamarin, можуть використовуватися на різних платформах. Ця концепція «напиши один раз, використовуй всюди» допомагає заощадити багато часу, зусиль і грошей на проектах з розробки додатків. 2) Технічна підтримка Microsoft Створений компанією Microsoft, Xamarin забезпечує відмінну технічну підтримку і стабільне резервне копіювання. 3) Власний користувацький інтерфейс Xamarin забезпечує повний доступ до всіх власним інструментів і API-інтерфейсів.	1) Відставання оновлень Це серйозна суперечка для Xamarin. Всі ми знаємо, будь то iOS або Android; є поновлення через регулярні проміжки часу. Але розробники Xamarin не можуть використовувати ці оновлення відразу після їх випуску. Для інтеграції нових функцій в екосистему Xamarin потрібен якийсь час, і, отже, виникає затримка. 2) Важкі додатки Додатки, розроблені з використанням платформи Xamarin, здаються більше (в два рази більше власних файлів). Управління такими важкими додатками вимагає додаткових зусиль і часу на налаштування.

Таблиця 1.1. Порівняльна характеристика кросплатформених мов програмування

- **Нативний підхід**

Нативна розробка - це класичне рішення, яке вимагає писати програми під кожну платформу окремо, використовуючи різні мови і з огляду на особливості кожної платформи.

Процес розробки нативного додатку під iOS - це написання програми на одному з основних мов iOS-розробки: Objective-C або Swift. Цей шлях створений корпорацією Apple і передбачає дотримання всіх її ідеям. Apple забезпечує розробників останніми версіями SDK (software development kit - набір засобів розробки), документацією, а також середовищем розробки Xcode.

На сьогодняшній день доступні 2 мови для нативної розробки під платформу iOS:

1) Objective C - мова програмування для iOS додатків, створена на початку 1980-х років минулого століття шляхом схрещування C з Smalltalk (зв'язок з об'єктами через повідомлення). Objective-C спочатку сприймався, як проста надбудова над мовою C, що модифікує його деякі синтаксичні конструкції, але потім, Objective-C став одним з найбільш популярних мов для розробки додатків для iPhone і iPad. Це основна мова, що використовується компанією Apple, знання якої дозволяє писати під будь-які платформи Apple, в тому числі macOS.
2) Swift – Молода, могутня і відкрита мова програмування загального призначення. Поєднує в собі все краще від C і Objective-C, але позбавлена обмежень останнього. У Swift використовуються сувора типізація об'єктів, що зменшує кількість помилок ще на етапі написання коду. Також в Swift додані сучасні функції, такі як

дженерики, замикання, множинні повернені значення і багато іншого, що перетворюють створення додатка в більш гнучкий і захоплюючий процес.

Порівняльна характеристика нативних мов програмування наведена у таблиці 1.2.

Мова програмування	Плюси	Мінуси
Objective C	1) Високий рівень підтримки кода З кожним оновленням зміни в Objective-C мінімальні 2) Співтовариство Велика кількість документації, технічної літератури та величезне співтовариство. Apple надає і регулярно оновлює офіційні книги і ресурси. 3) Сумісність Сумісність Objective-C всередині проектів, написаних на Swift, дозволить	1) Типи даних Динамічна система типів даних, яка також є плюсом, передбачає можливість появи або пропуску помилок навіть під час компіляції. Зокрема, затягнути процес можуть помилки; 2) Продуктивність Низька в порівнянні з мовою Swift продуктивність; 3) Швидкість сборки при сумісності Взаємодія з файлами Swift відбувається за допомогою

	тобі застосовувати дві мови одночасно.	«моста», що сильно гальмує процес сборки.

Swift	1) Швидкість Зараз мова майже на одному рівні з C++ 2) Навігація в проекті Спрощена навігація по файлах проекту. На відміну від Objective-C, який створює два файли для оголошення і реалізації, Swift обходиться всього одним. Крім того, імена методів і коментарі між файлами синхронізуються автоматично 3) Простота синтаксису Легка читаємість, оскільки дана мова не побудована на основі C. 4) Підвищена безпека Swift, на відміну від Objective-C, строго типізований, що полегшує знайдення багів ще на етапі розробки	1) Швидкість сборки при сумісності Взаємодія з файлами Swift відбувається за допомогою «моста» (умовний адаптер, який переводить код на Swift в формат Objective-C), що сильно гальмує процес сборки. 2) Швидкість змін мови Swift постійно розвивається і змінюється. Наприклад, виклик методу може змінитися після оновлення.

Таблиця 1.2 . Порівняльна характеристика нативних мов програмування

Порівняння кросплатформеного та нативного підходів

Основною перевагою кросплатформеного підхода є те, що написаний проект можна використовувати як на платформі iOS, так і на платформі Android.

Переваги нативного підхода:

1) Додаток оптимізований саме під платформу iOS, а значить буде працювати швидко і коректно
2) Зрозумілий та простий доступ до апаратної частини девайсів – камери, мікрофону, геолокації і тд
3) Рідний для платформи інтерфейс. Нативні програми зазвичай оперують «платформеними» елементами інтерфейсу: меню, навігація, форми і всі інші елементи дизайну беруться від операційної системи і тому звичні і зрозумілі користувачеві.[4]

Остаточний вибір мови програмування

Кроссплатформену розробку можна порадити тоді, коли бюджет обмежений, потрібно дізнатися попит на мобільний додаток або якщо на стороні клієнта є свій веб-програміст. В інших випадках вибір краще робити на користь нативной розробки.

Нас цікавить тільки розробка під платформу iOS, отже, нативний підхід буде найдійнішим та більш якісний.

З нативних мов вибираємо Swift, адже в нього більше суттєвих над Objective C та він є набагато більш сучаснішим.

РОЗДІЛ 2. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

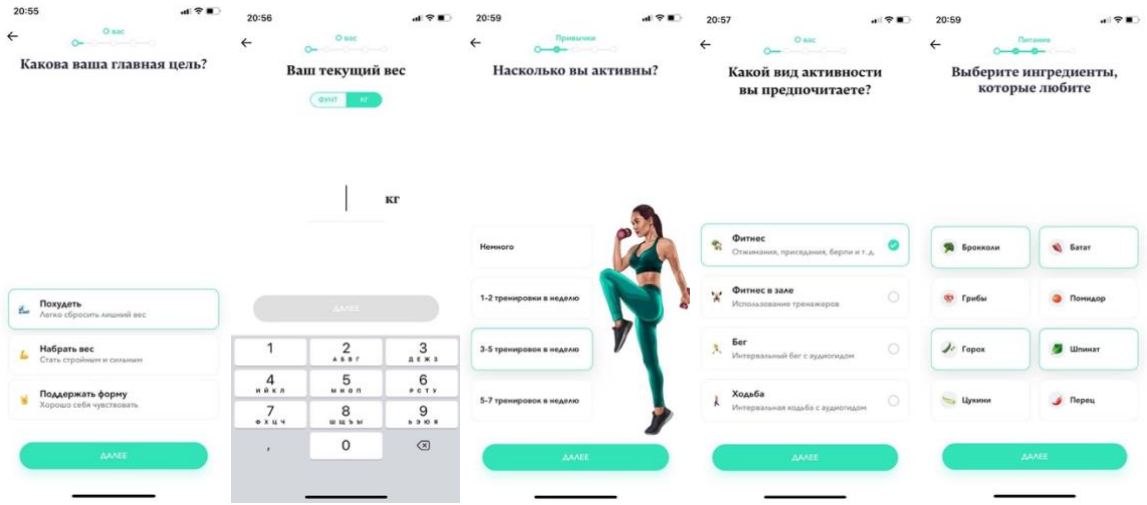
2.1 BetterMe

Один з найпопулярніших мобільних фітнес додатків на ринку. Був розроблений українською IT компанією Genesis у 2016 році. На даний момент зроблено близько 90 мільйонів встановлень додатку на платформах iOS та Android.

Цінова політика:

Безкоштовний функціонал	Платний функціонал	Ціна підписки
1. Логін/Реєстрація 2.Заповнення персональної анкети 3. Домашня сторінка: - календар занять - тренування - контроль споживання води - контроль голоду - лічильник кроків 4. Бібліотека тренувань	1.Програма харчування 2. Спортивні виклики 3. Чат з персональним тренером	$120/рік $20/місяць

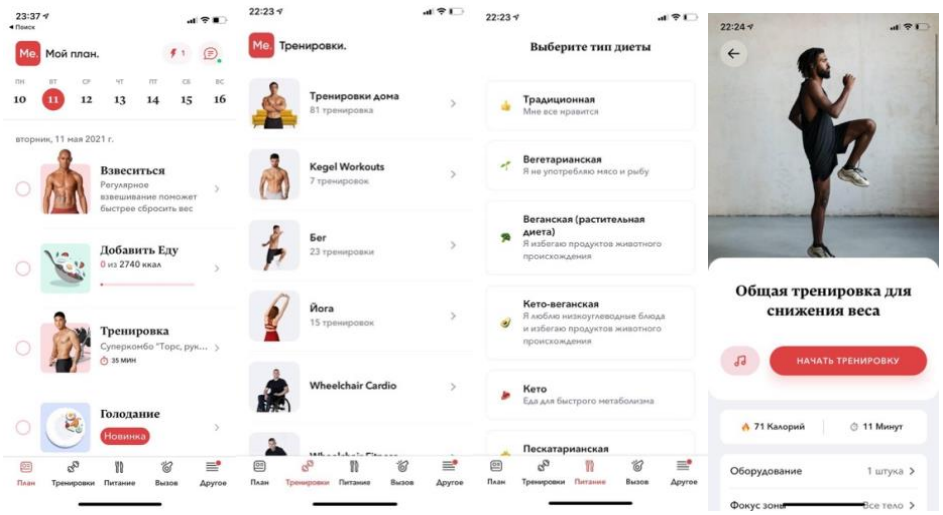
Заповнення персональної анкети (вона достатньо велика, наведено декілька прикладів)



Рисунки 3 – Заповнення персональної анкети в BetterMe

Основний функціонал

За персональними даними користувача надає йому план тренувань та харчування.



Рисунки 4 – Основний функціонал в BetterMe

Висновок: Як бачимо додаток робить персональний підхід до кожного юзера. Із мінусів: достатньо примітивний дизайн персональної анкети, цю частину бачить саме новий користувач додатку, а, отже, важливо захопити його увагу привабливим дизайном.

2.2 NTC (Nike training club)

Другий по популярності додаток для фітнесу. Був розроблений американською компанією Nike у 2020 році.

Цінова політика:

Безкоштовний функціонал	Платний функціонал	Ціна підписки
1. Логін/Реєстрація 2. Заповнення дуже короткої персональної анкети 3. Сторінка активності: - історія тренінгів та статистика - тренування - контроль споживання води - контроль голоду	Нема	Безкоштовна

- лічильник кроків 4. Бібліотека тренувань (приблизно 200 відео)		

Заповнення персональної анкети

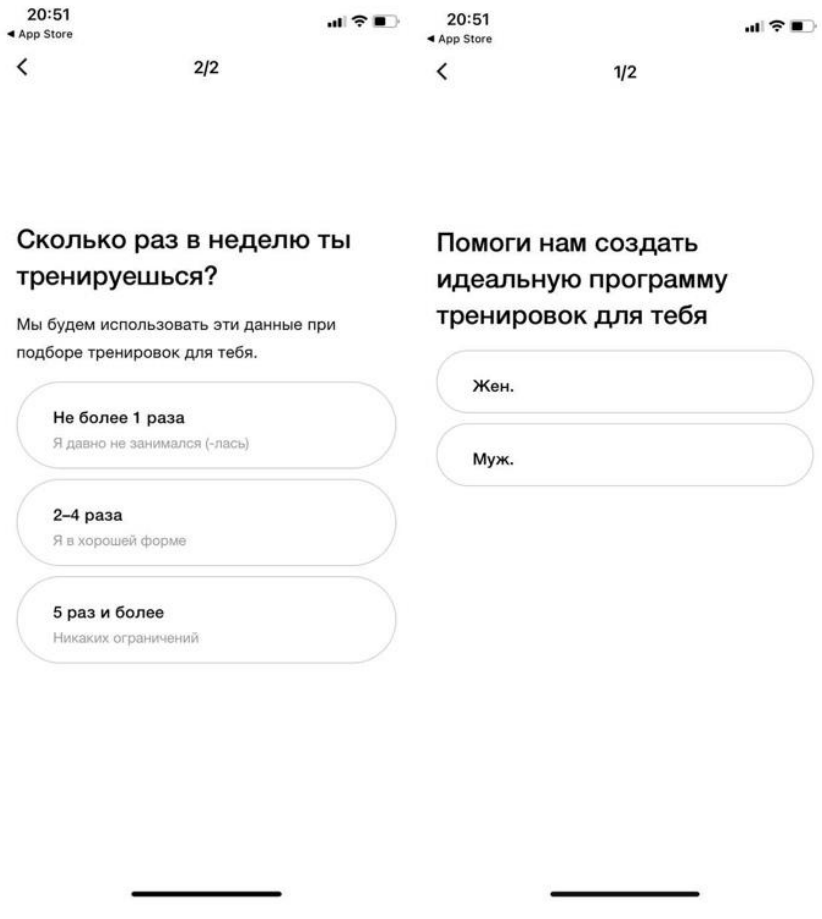
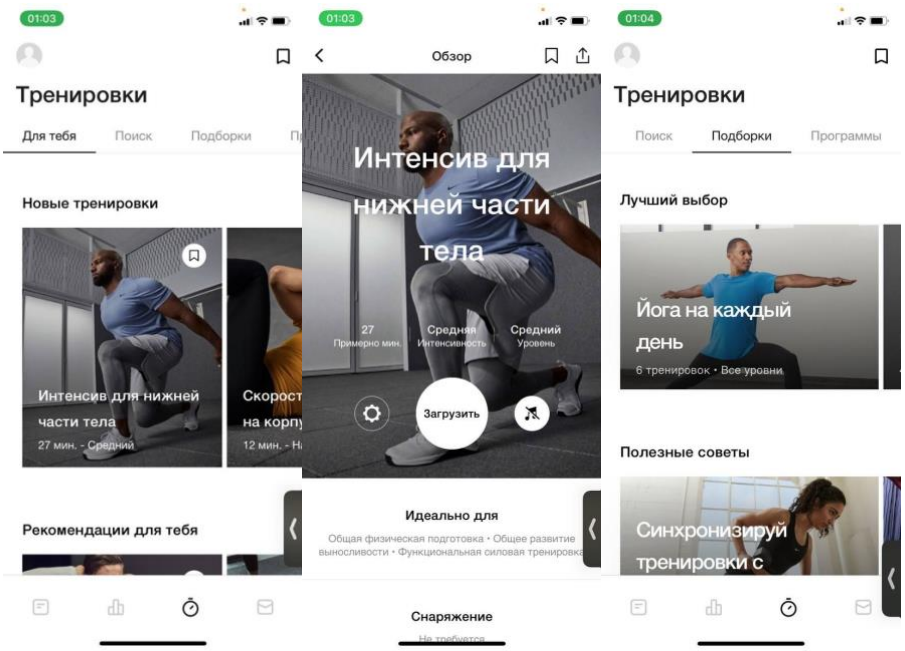


Рисунок 5 – Персональна анкета в NTC

Основний функціонал



Рисунки 5 – Основний функціонал NTC

Висновок: Як бачимо додаток збирає дуже мало персональних даних, а, отже не підлаштовується достатньо індивідуально до кожного користувача.

Отже, треба розробити додаток, що за функціоналом буде не гіршим за існуючих лідерів ринку, а дизайном буде навіть привабливішим, щоб захопити увагу користувача ще при першій спробі використання програми.

РОЗДІЛ 3. ВИКОРИСТАНІ ТЕХНОЛОГІЇ

3.1 Обрана мова програмування

Swift - багатопарадигмова компільована мова програмування, розроблена компанією Apple для того, щоб співіснувати з Objective C і бути стійкішою до помилкового коду. Swift була представлена на конференції розробників WWDC 2014. Мова побудована з LLVM компілятором, включеного у Xcode 6 beta [5].

Swift - це фантастичний спосіб написання програмного забезпечення, будь то для телефонів, настільних ПК, серверів чи чогось іншого, на якому працює код. Це безпечна, швидка та інтерактивна мова програмування, що поєднує в собі найкраще сучасне мовне мислення з мудрістю ширшої інженерної культури Apple та різноманітним внеском її спільноти з відкритим кодом. Компілятор оптимізований для продуктивності, а мова оптимізована для розвитку, без шкоди для жодного. Swift доброзичлива до нових програмістів. Це мова програмування індустріальної якості, яка така ж виразна і приємна. Також визначає великі класи поширених помилок програмування, використовуючи сучасні схеми програмування:

1) Змінні завжди ініціалізуються перед використанням.
2) Індекси масиву перевіряються на наявність помилок поза межами.
3) Масиви даних перевіряються на переповнення.
4) Необов’язково гарантувати, що значення нуля обробляються явно.
5) Пам'ять управляється автоматично.
6) Поводження з помилками дозволяє контролювати відновлення після несподіваних збоїв.

3.2 Архітектура та паттерни проекту

Уявіть, що нам потрібно внести невелику зміну в екран і спробувати, щоб перевірити, чи ця зміна працює. Якщо додаток, над яким ми працюємо, побудовано монолітно як єдиний продукт, нам потрібно буде перекомпілювати всю базу коду, щоб побачити результати. Цей процес трудомісткий і нудний. Ми повинні вміти компілювати та запускати частини самостійно, ізольовано.

Іноді частина програми є складною і досить великою, щоб її можна було розглядати як продукт самостійно. Ми можемо назвати це модулем. Модульність є ключовим фактором для досягнення програмного забезпечення, яке можна масштабувати та підтримувати з часом.

Незважаючи на те, що це може додати складності на початку проекту, в довгостроковій перспективі ми можемо знайти багато переваг у модульному рішенні: менший час нарощування при перекомпіляції після внесення змін, ізоляція змін можливість використовувати дитячі майданчики (playgrounds) для побудови та тестування інтерфейсу користувача тощо [6].

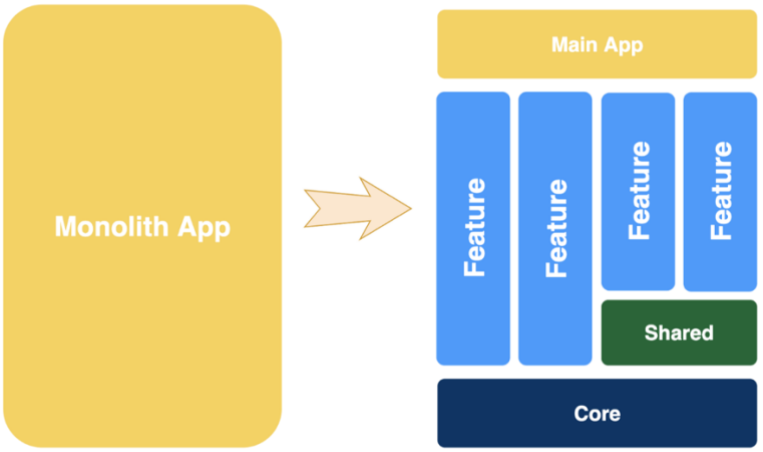


Рисунок 6 – схема розбиття монолітного додатку на модулі

Таким чином, проект було розбито на такі окремі проекти (модулі), які наведені на рисунку 7:

- FitnessApp - головний модуль, у якому містяться посилання на всі інші модулі та в якому прописана основна логіка взаємодії модулів між собою
- Networking - модуль, що містить абстракції запитів до сервера з використанням сторонньої бібліотеки Alamofire. Саме через нього відбувається вся взаємодія з backend.
- Services - модуль, що містить допоміжні класи та розширення додатку
- Authentication – модуль, що відповідає за аутентифікацію користувача
- Questionnaire – модуль, що відповідає за заповнення персональних параметрів користувачем
- Payment – модуль, що відповідає за оплату підписки на додаток
- Core – модуль, в якому зберігається основний функціонал додатку: training feed, trainings library, profile та інше.

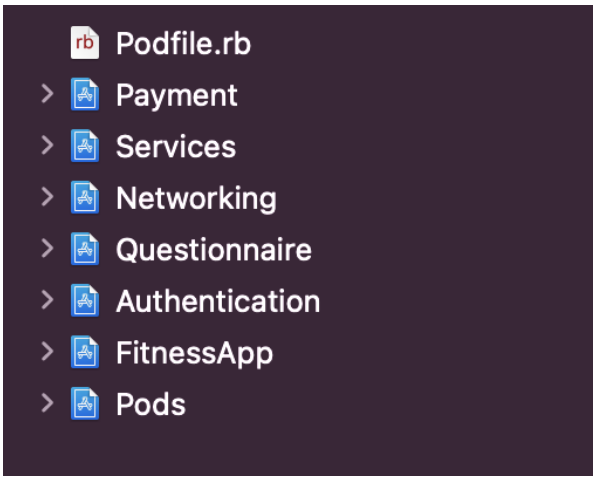


Рисунок 7 – Модулі всередині проекту

3.2.1 Приклад архітектури всередині модулю

Розглянемо архітектуру всередині модулів на прикладі модулю Authentication. Коротко архітектуру кожного модуля можна описати як Clean + MVVM

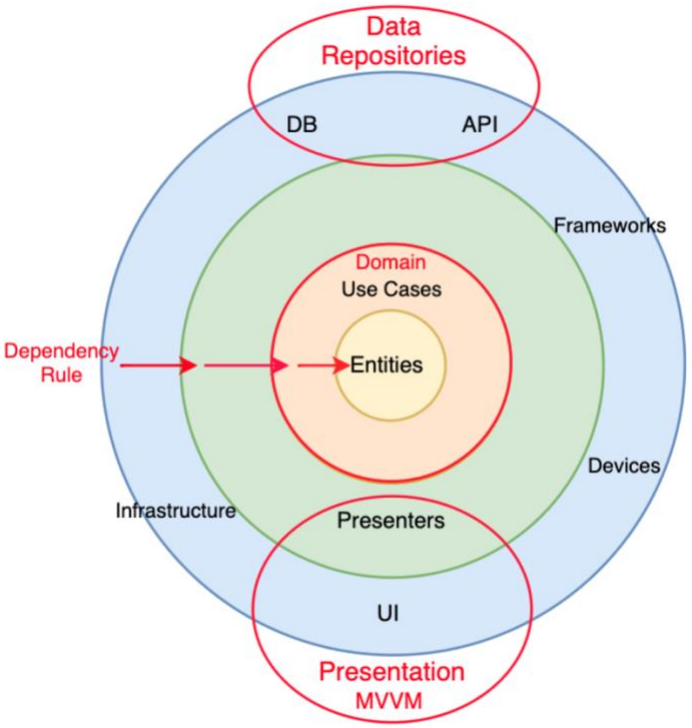


Рисунок 8 - Схема архітектури Clean + MVVM

Як видно з рисунку 8, на схемі Clean Architecture, ми маємо різні шари в додатку.

Головне правило - не мати залежностей від внутрішніх шарів до зовнішніх шарів. Стрілки, спрямовані ззовні всередину, є правилом залежності. Залежності можуть бути лише від зовнішнього шару всередину. Після групування отримуємо 3 шари: Presentation, Domain та Data.

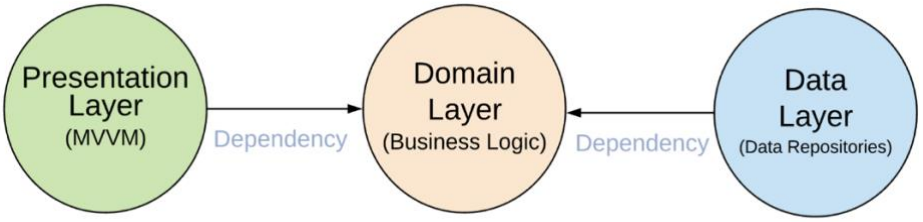


Рисунок 9 – Шари “Чистої” архітектури

1. Доменний рівень (бізнес логіка) є внутрішньою частиною цибулі (від зовнішніх залежностей він повністю ізольований). Він містить сутності, юз кейзи та репозиторії.
2. Презентаційний шар містить інтерфейс користувача. View координується тільки з View Models. Цей шар залежить тільки від рівня домену.
3. Дані містять все, що відповідає за роботу з сервером(Api Service) та сховища(Core Data, User Defaults).

На схемі нижче кожен компонент з кожного шару представлений разом із напрямком залежності, а також потоком даних (Запит / Відповідь). Ми можемо побачити точку інверсії залежностей, де ми використовуємо інтерфейси сховища (протоколи).

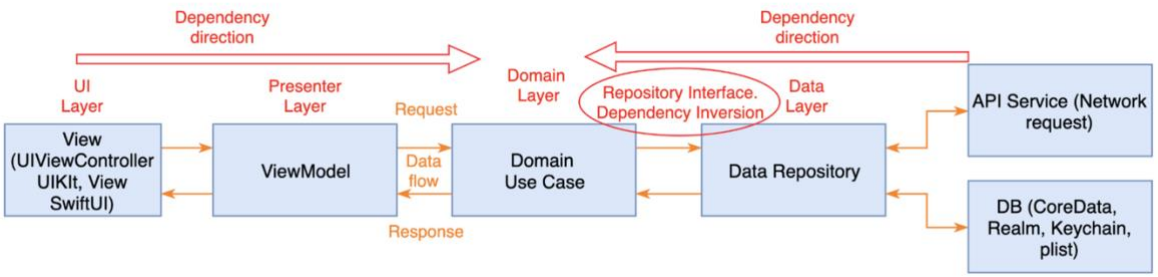


Рисунок 10 – Схема архітектури всередині модуля

Шаблон Model-View-ViewModel (MVVM) забезпечує чітке розділення проблем між інтерфейсом користувача та домейном.[7]

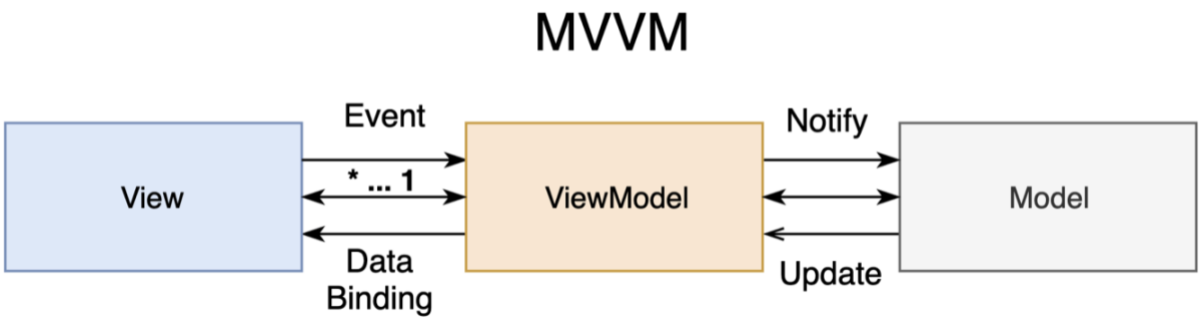


Рисунок 11 – Схема MVVM

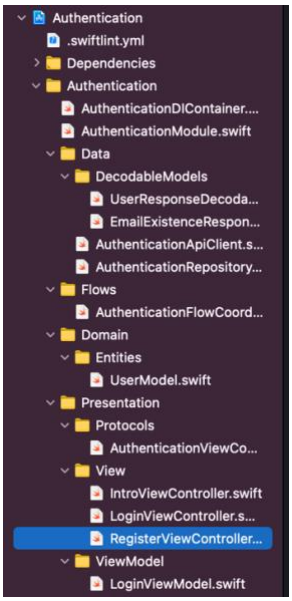
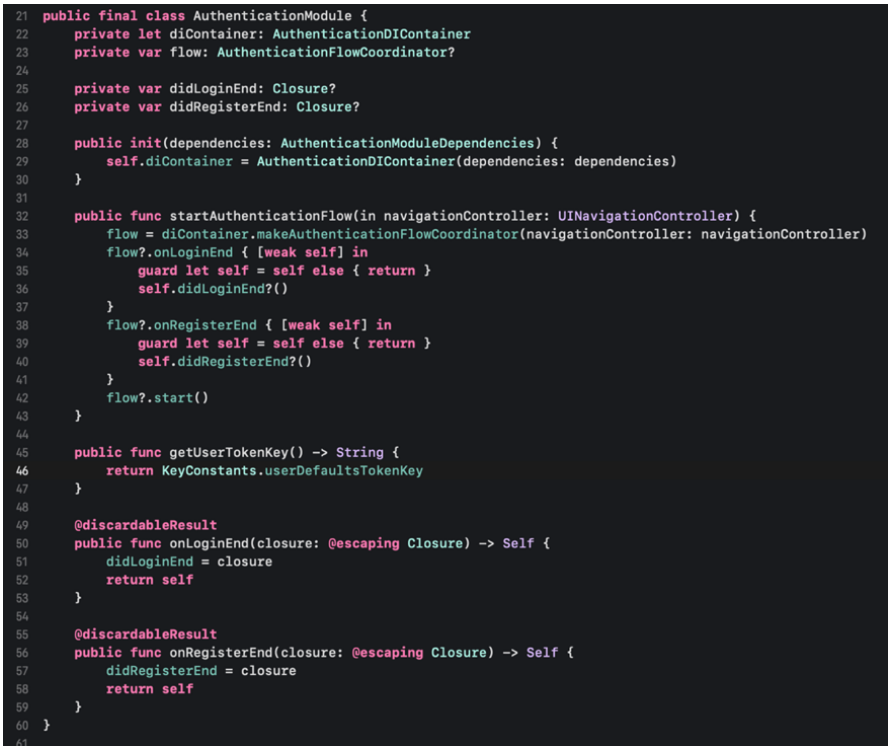


Рисунок 12 – Приклад архітектури всередині модуля

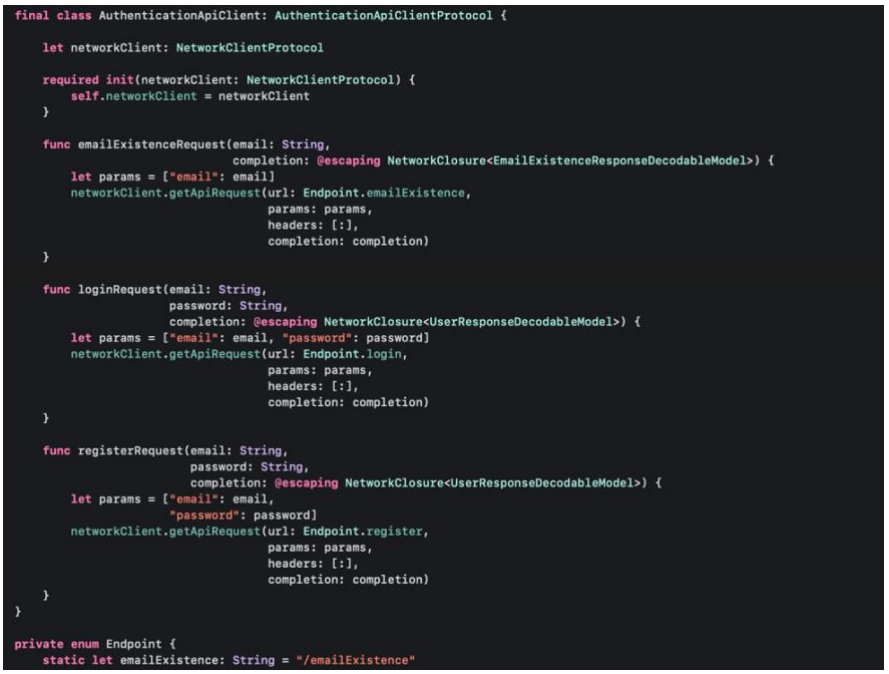
Кожен проект містить клас Module, з якого починається робота модуля, він створюється у головному FitnessApp.



```swift
public final class AuthenticationModule {
    private let diContainer: AuthenticationDIContainer
    private var flow: AuthenticationFlowCoordinator?

    private var didLoginEnd: Closure?
    private var didRegisterEnd: Closure?

    public init(dependencies: AuthenticationModuleDependencies) {
        self.diContainer = AuthenticationDIContainer(dependencies: dependencies)
    }

    public func startAuthenticationFlow(in navigationController: UINavigationController) {
        flow = diContainer.makeAuthenticationFlowCoordinator(navigationController: navigationController)
        flow?.onLoginEnd { [weak self] in
            guard let self = self else { return }
            self.didLoginEnd?()
        }
        flow?.onRegisterEnd { [weak self] in
            guard let self = self else { return }
            self.didRegisterEnd?()
        }
        flow?.start()
    }

    public func getUserTokenKey() -> String {
        return KeyConstants.userDefaultsTokenKey
    }

    @discardableResult
    public func onLoginEnd(closure: @escaping Closure) -> Self {
        didLoginEnd = closure
        return self
    }

    @discardableResult
    public func onRegisterEnd(closure: @escaping Closure) -> Self {
        didRegisterEnd = closure
        return self
    }
}
```

Рисунок 13 – Приклад класа AuthenticationModule

В кожному модулі є свій ApiClient, який через абстракцію Networking модуля звертається до сервера.



```swift
final class AuthenticationApiClient: AuthenticationApiClientProtocol {

    let networkClient: NetworkClientProtocol

    required init(networkClient: NetworkClientProtocol) {
        self.networkClient = networkClient
    }

    func emailExistenceRequest(email: String,
                               completion: @escaping NetworkClosure<EmailExistenceResponseDecodableModel>) {
        let params = ["email": email]
        networkClient.getApiRequest(url: Endpoint.emailExistence,
                                    params: params,
                                    headers: [:],
                                    completion: completion)
    }

    func loginRequest(email: String,
                      password: String,
                      completion: @escaping NetworkClosure<UserResponseDecodableModel>) {
        let params = ["email": email, "password": password]
        networkClient.getApiRequest(url: Endpoint.login,
                                    params: params,
                                    headers: [:],
                                    completion: completion)
    }

    func registerRequest(email: String,
                         password: String,
                         completion: @escaping NetworkClosure<UserResponseDecodableModel>) {
        let params = ["email": email,
                      "password": password]
        networkClient.getApiRequest(url: Endpoint.register,
                                    params: params,
                                    headers: [:],
                                    completion: completion)
    }
}

private enum Endpoint {
    static let emailExistence: String = "/emailExistence"
```

Рисунок 14 – Приклад ApiClient всередині Authentication модуля

Вся навігація всередині модуля відбувається завдяки класу Flow Coordinator, він же робить запит до сервера в потрібний момент за допомогою ApiClient.

```
    func checkEmailExistence() {
        guard let userEmail = self.loginViewModel.getUserEmail() else {
            return
        }
        apiClient.emailExistenceRequest(email: userEmail) { [weak self] result in
            guard let self = self else { return }
            switch result {
            case .success(let value):
                guard let exists = value.emailExists else {
                    return
                }
                if exists {
                    self.navigationController?.pushViewController(self.loginVC, animated: false)
                } else {
                    self.navigationController?.pushViewController(self.registerVC, animated: false)
                }
            case .failure(let error):
                print(error)
                return
            }
        }
    }

    func login() {
        guard let userEmail = self.loginViewModel.getUserEmail(),
              let password = self.loginViewModel.getUserPassword() else {
            return
        }
        apiClient.loginRequest(email: userEmail, password: password) { [weak self] result in
            guard let self = self else { return }
            switch result {
            case .success(let value):
                guard let token = value.token else {
                    return
                }
                self.repository.setUserToken(token: token)
                self.didLoginEnd?()
            case .failure(let error):
                print(error)
                return
            }
```

```
final class AuthenticationFlowCoordinator {
    private var navigationController: UINavigationController?
    private let dependencies: AuthenticationFlowCoordinatorDependencies
    private lazy var apiClient: AuthenticationApiClientProtocol = dependencies.makeAuthenticationApiClient()
    private lazy var repository: AuthenticationRepositoryProtocol = dependencies.makeAuthenticationRepository()
    private lazy var loginViewModel: LoginViewModelProtocol = dependencies.makeLoginViewModel()
    private lazy var introVC: AuthenticationViewControllerProtocol = makeIntroVC()
    private lazy var loginVC: AuthenticationViewControllerProtocol = makeLoginVC()
    private lazy var registerVC: AuthenticationViewControllerProtocol = makeRegisterVC()

    private var didLoginEnd: Closure?
    private var didRegisterEnd: Closure?

    init(navigationController: UINavigationController,
         dependencies: AuthenticationFlowCoordinatorDependencies) {
        self.navigationController = navigationController
        self.dependencies = dependencies
    }

    func start() {
        navigationController?.pushViewController(introVC, animated: false)
    }

    func makeIntroVC() -> AuthenticationViewControllerProtocol {
        let introVC = dependencies.makeIntroViewController(viewModel: loginViewModel)
        introVC.onTapLogInButton { [weak self] in
            guard let self = self else { return }
            self.checkEmailExistence()
        }
        return introVC
    }

    func makeLoginVC() -> AuthenticationViewControllerProtocol {
        let loginVC = dependencies.makeLoginViewController(viewModel: loginViewModel)
        loginVC.onTapLogInButton { [weak self] in
            guard let self = self else { return }
            self.login()
        }
        return loginVC
    }
```

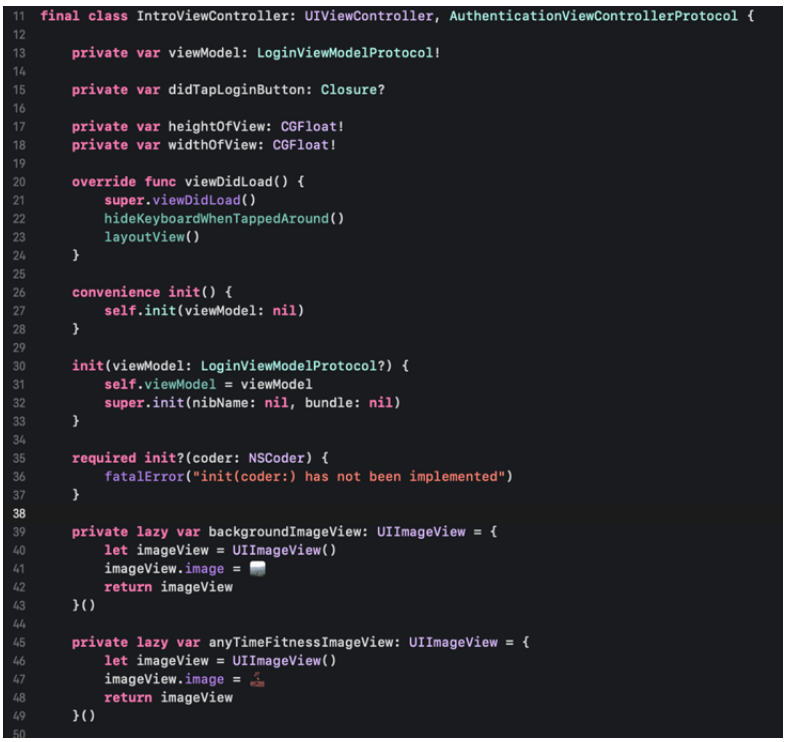
Рисунок 15 – Приклад Flow Coordinator всередині Authentication модуля

Всі дані необхідні для відображення користувачу обробляються у ViewModel. Зміни в цьому класі автоматично змінюють і View.

```swift
public final class LoginViewModel: LoginViewModelProtocol {
    private var user = UserModel()

    func setUserEmail(email: String) {
        user.email = email
    }

    func getUserEmail() -> String? {
        return user.email
    }

    func setUserPassword(password: String) {
        user.password = password
    }

    func getUserPassword() -> String? {
        return user.password
    }
}
```

Рисунок 16 – приклад ViewModel

Найнижчий у ієрархії класів – ViewController, що відповідає за View, тобто інтерфейс додатку.



```swift
final class IntroViewController: UIViewController, AuthenticationViewControllerProtocol {

    private var viewModel: LoginViewModelProtocol!

    private var didTapLoginButton: Closure?

    private var heightOfView: CGFloat!
    private var widthOfView: CGFloat!

    override func viewDidLoad() {
        super.viewDidLoad()
        hideKeyboardWhenTappedAround()
        layoutView()
    }

    convenience init() {
        self.init(viewModel: nil)
    }

    init(viewModel: LoginViewModelProtocol?) {
        self.viewModel = viewModel
        super.init(nibName: nil, bundle: nil)
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    private lazy var backgroundImageView: UIImageView = {
        let imageView = UIImageView()
        imageView.image =
        return imageView
    }()

    private lazy var anyTimeFitnessImageView: UIImageView = {
        let imageView = UIImageView()
        imageView.image =
        return imageView
    }()
```

Рисунок 17 – Приклад ViewController

3.2.2 Зв'язок з бек-ендом

Зв'язок з бек-ендом відбувається за архітектурою REST API за допомогою підключеної біблиотеки Alamofire. Alamofire - це HTTP мережева бібліотека, що дозволяє фронт-енду елегантно взаємодіяти з бек-ендом, на рисунку 5 зображена дана взаємодія. Ця бібліотека базується на NSURLSession, це звільняє від написання шаблонів, що спрощує написання коду. З ним досить просто доступ до даних в Інтернеті, а код буде набагато кращим і його буде легше читати. Є декілька основних функцій, доступних в Alamofire:

1) .upload - вивантаження (uploading) файлів за допомогою багатокомпонентних, поточних, файлових методів і методів даних.
2) .download - завантаження файлів або відновлення завантаження, що знаходиться в процесі.
3) .request - будь-який інший запит HTTP, непов'язаний з передачею файлів.

Ці функції Alamofire знаходяться в області видимості модуля, а не класу або структури. В основі Alamofire стоять класи і структури, такі як Manager, Request і Response; тим не менш, не потрібно повністю розуміти структуру Alamofire, щоб почати нею користуватися. На рисунку 6 зображений приклад коду проекту з використанням даної бібліотеки.

У проекті зв’язок з бекендом у проекті відбувається у модулі Networking. Реалізовано як абстракція для того, щоб доступ до Alamofire бібліотеки був тільки в одному місці, що не визиває протиріч з “Чистою” архітектурою.

Далі в ApiClient у кожному модулі викликається метод get чи post, передає туди адресу, параметри, хедери та отримує у відповідь response від сервера.

Адреса складається з basehost та endpoint. Basehost вказується у константах Networking модулю – це адреса сервера, де знаходиться сервер. Endpoint – це адреса конкретного реквесту, що передається з ApiClient.

У хедери в усіх реквестах після логіну чи реєстрації передається токен юзера, що зберігається в проекті до моменту, поки він не видалить додаток, чи не вийде зі свого акаунту.

```swift
import UIKit
import Alamofire
import Services

public protocol NetworkClientProtocol {
    func getApiRequest<T: Decodable>(url: String,
                                     params: [String: Any]?,
                                     headers: [String: String]?,
                                     completion: @escaping NetworkClosure<T>)

    func postApiRequest<T: Decodable>(url: String,
                                      params: [String: Any]?,
                                      headers: [String: String]?,
                                      completion: @escaping NetworkClosure<T>)
}

public final class NetworkClient: NetworkClientProtocol {

    public init() {}

    public func getApiRequest<T: Decodable>(url: String,
                                            params: [String: Any]?,
                                            headers: [String: String]?,
                                            completion:
                                                    @escaping NetworkClosure<T>) {
        let httpHeaders = getHTTPHeaders(headers: headers)

        AF.request(Constants.baseHost + url,
                   method: .get,
                   parameters: params,
                   headers: httpHeaders).responseDecodable(of: T.self) { afDataResponse in

                    switch afDataResponse.result {
                    case .success(let response):
                        completion(.success(response))
                    case .failure(let error):
                        completion(.failure(error))
                    }
                   }
    }
```

Рисунок 18 - Приклад коду проекту з використанням Alamofire

РОЗДІЛ 4. АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ

4.1 Аналіз існуючих підходів до тестування

Test – це перевірка коду або програми на правильність, за допомогою різноманітних перевірок. Тест може перевірити стан змінних екземплярів об'єкта після виконання деяких операцій, переконатися, що код показує певний exception, коли він піддається граничним умовам або перевірити візуальну правильність програми, сайту, додатку.

Тестування вручну(Manual QA) - це тестування програмного забезпечення, де тести виконуються Quality Assurance аналітикою вручну. Виконується для виявлення помилок у програмному забезпеченні, що розробляється.

Під час ручного тестування тестер перевіряє всі основні характеристики програми чи програмного забезпечення. У цьому процесі тестувальник програмного забезпечення виконує тестовівипадки та генерує тестові звіти без допомоги будь-яких інструментів тестування програмного забезпечення для автоматизації.

Під час автоматичного тестування програмного забезпечення тестери записують код або тестові сценарії для автоматизації виконання тесту. Тестери використовують відповідні засоби автоматизації для розробки тестових сценаріїв та перевірки програмного забезпечення. Мета цього - закінчити виконання тесту за менший проміжок часу без участі людини.

Автоматизоване тестування повністю покладається на попередньо розроблений тест, який запускається автоматично для порівняння фактичного результату з очікуваними результатами. Це допомагає тестеру визначити, чи працює програма так, як очікувалося.

Основними перевагами автоматичного тестування перед ручним тестуванням є:

- Легко проводити тестування у великих масштабах.
- Швидкий час виконання.
- Краща точність.

З огляду на ці переваги автоматичне тестування явно доцільніше використовувати, коли масштаб тестування великий, або коли на цикл розробки виділяється не так багато часу тоді і потрібно неодноразово виконувати коди, що мають більш високу частоту ітерацій.

4.2 Використані технології автоматизації в проекті

4.2.1 XCTest

UI Test(User interface test) дає змогу знаходити та взаємодіяти з інтерфейсом вашого додатка, щоб перевірити властивості та стан елементів інтерфейсу.

В проекті використовується технологія тестування XCTest, що є нативною для платформи IOS. XCTest Забезпечує можливість тестування користувацького інтерфейсу, інтегровану з Xcode. Є постійна інтеграція через Xcode Server та xcodebuild. XCTest Повністю Сумісний як з Objective-C, так і з Swift.

За допомогою тестування інтерфейсу можливо імітувати дії користувача на симуляторах чи пристроях, щоб дізнатись, чи працює наша програма з останніми змінами коду. За допомогою XCTest ми можемо взаємодіяти з елементом інтерфейсу та перевіряти його положення чи стан у наших додатках.

XCTest дозволяє писати тести на користувальницький інтерфейс прямо у Xcode з окремим тестуванням інтерфейсу користувача у додатку.

XCTest запускається в окремому процесі від основного додатка iOS і він не має доступу до внутрішніх методів, API чи даних програми. Він використовує технологію Accessibility для взаємодії з основним кодом програми, а це означає, що розробникам додатків необхідно надати інформацію про доступність елементів інтерфейсу, щоб зробити програмні продукти доступними, а також тестовими. На рисунку 7 можна побачити приклад тесту за проекта, що перевіряє правильність маршруту проекта протягом взаємодії інвестора з додатком.

```
func testInvestedProjectRoute() {
    setFakeLogin()
    let investorListOfProjectsScreen = InvestorListOfProjectsScreen()
    investorListOfProjectsScreen.tapViewStartupsButton()

    let startupsScreen = InvestorStartupsScreen()
    startupsScreen.tapFirstCell()

    let startupDetailedScreen = StartupDetailedScreen()
    startupDetailedScreen.tapInvestButton()
    startupDetailedScreen.tapOkButton()

    XCTAssertFalse(startupsScreen.projectCellExists(projectName: "Tinder"))

    startupsScreen.tapMyInvestmentsButton()

    XCTAssertTrue(investorListOfProjectsScreen.projectCellExists(projectName: "Tinder"))
}
```

Рисунок 19 – Автоматичний тест

4.2.2 MockServer

Mock server(або Stub server)- це “штучний” сервер, що підміняє responses від сервера на надіслані requests від додатку.

Переваги використання Mock server:

1) Незалежність від живого сервера
2) Швидкі та надійні відповіді сервера

3) Покриття тестами випадків з неправильними відповідями сервера

На рисунку 20 можна побачити принцип роботи MockServer.

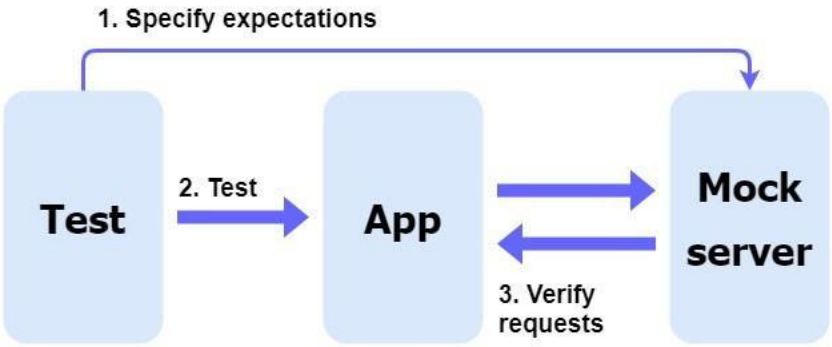


Рисунок 20 – Принцип роботи MockSever

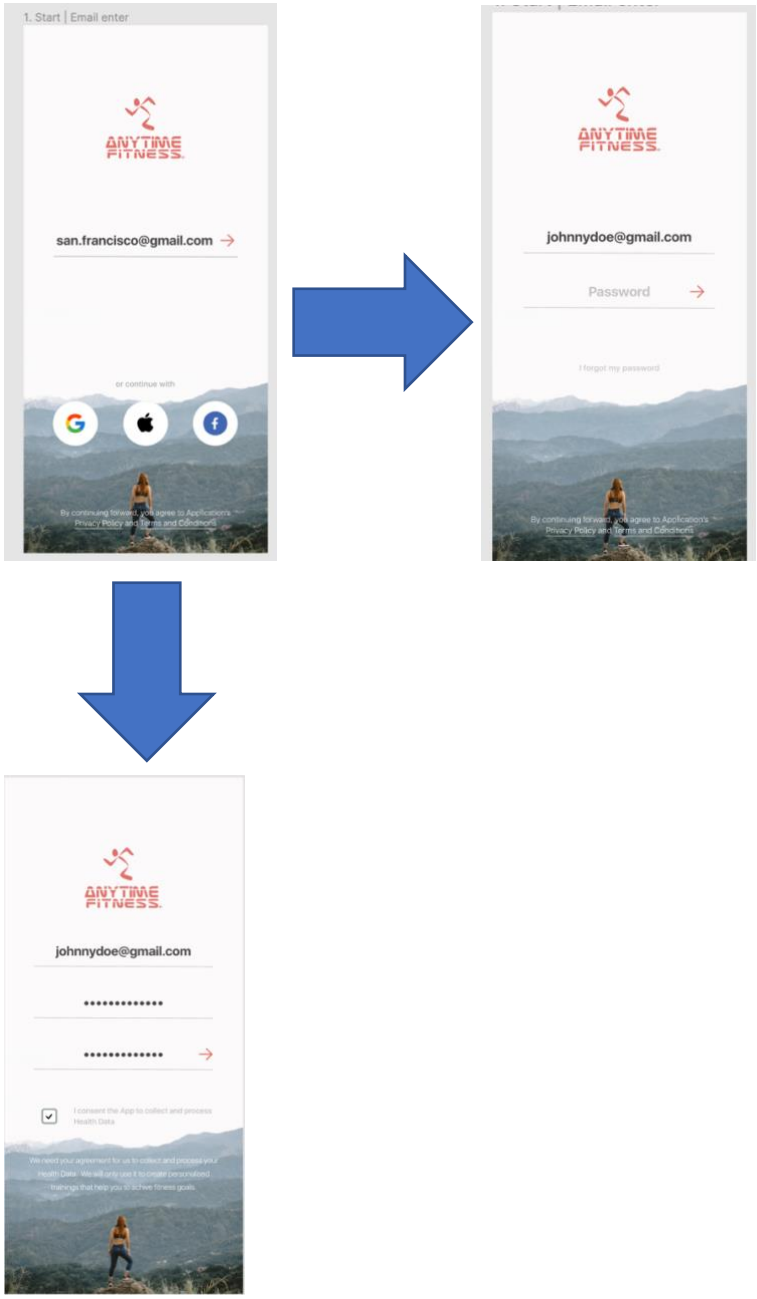
Розглянемо приклад. Автоматизатор пише тест на перевірку правильного відображення певного елемента у інтерфейсі, але через помилку back-end частини не може дійти до потрібного екрану. В цьому випадку тест повертає помилку, проте не через причину неправильно відображеного елемента. Для того щоб такої проблеми не виникало можна використовувати Mock server.

У проекті для Mock Server була використана бібліотека Ambassador. Таким чином можна, наприклад, в тесті взагалі пропустити етап реєстрації, якщо він нас не цікавить.

РОЗДІЛ 5. ДИЗАЙН ТА ІНСТРУКЦІЯ КОРИСТУВАЧА

5.1 Аутентифікація

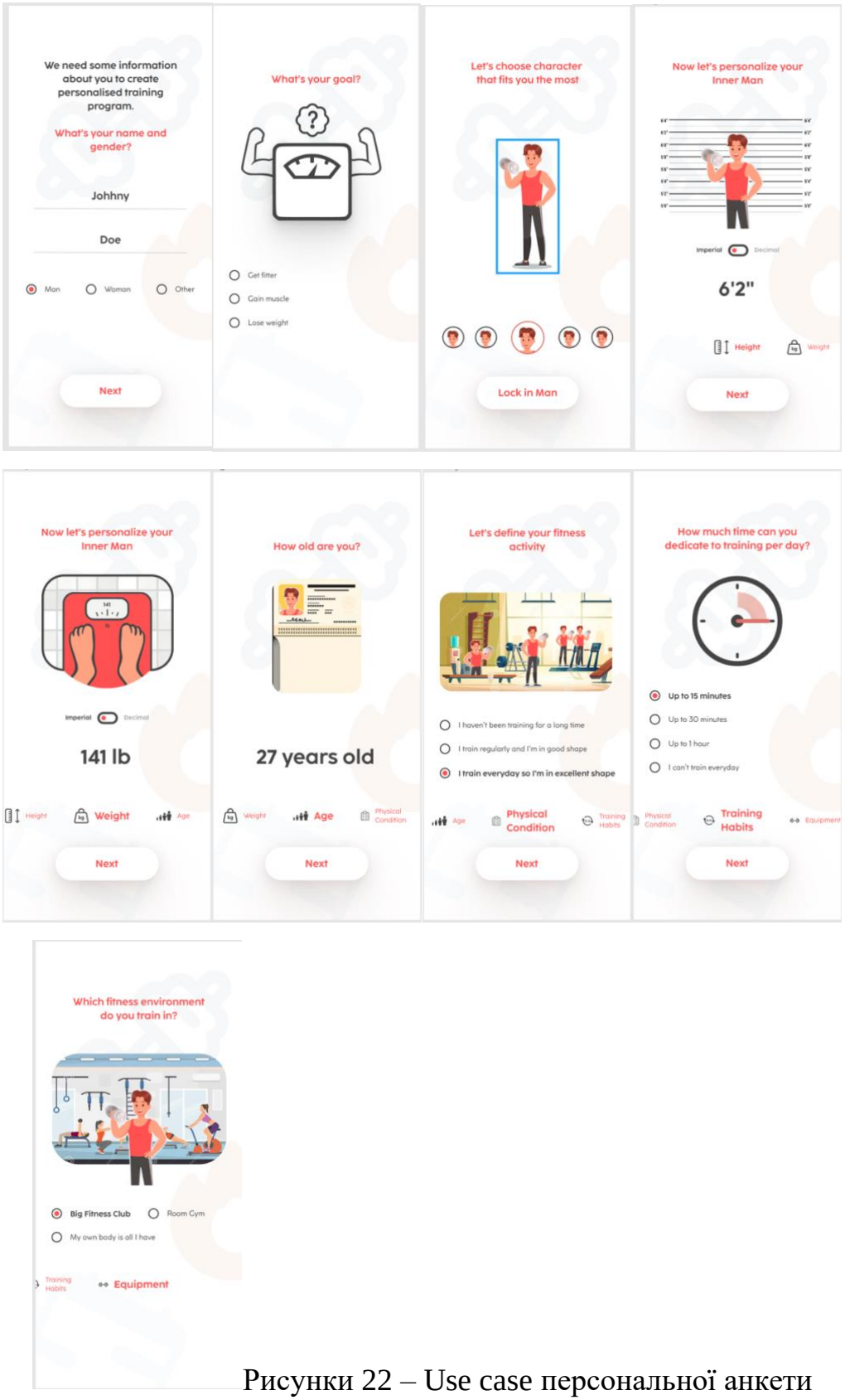
Спочатку користувач вводить свій мейл та натискає стрілку далі. Якщо мейл не було зареєстровано раніше – юзер переходить до реєстрації. Успішно створивши аккаунт, юзер відправляється до заповнення персональної анкети.



Рисунки 21 - Use case аутентифікації

5.2 Персональна анкета

В персональній анкеті юзер заповнює свої дані, параметри тіла та інше для подальшого оптимального підбору плану тренувань



Рисунки 22 – Use case персональної анкети

5.3 Плани підписки

Тут юзер отримує можливість купити підписку, що відкриває доступ до всіх функцій додатку, за різними планами. Він може купити підписку за допомогою Apple Pay або відмовитись від неї, натиснувши хрестик вгорі.

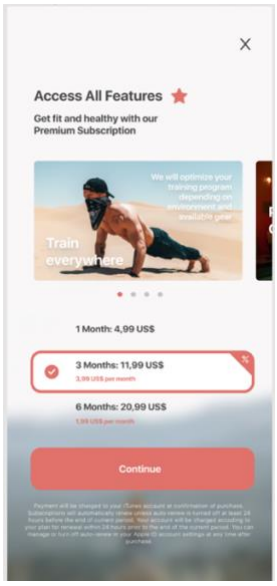


Рисунок 23 – Use case планів підписки

5.4 Основний функціонал

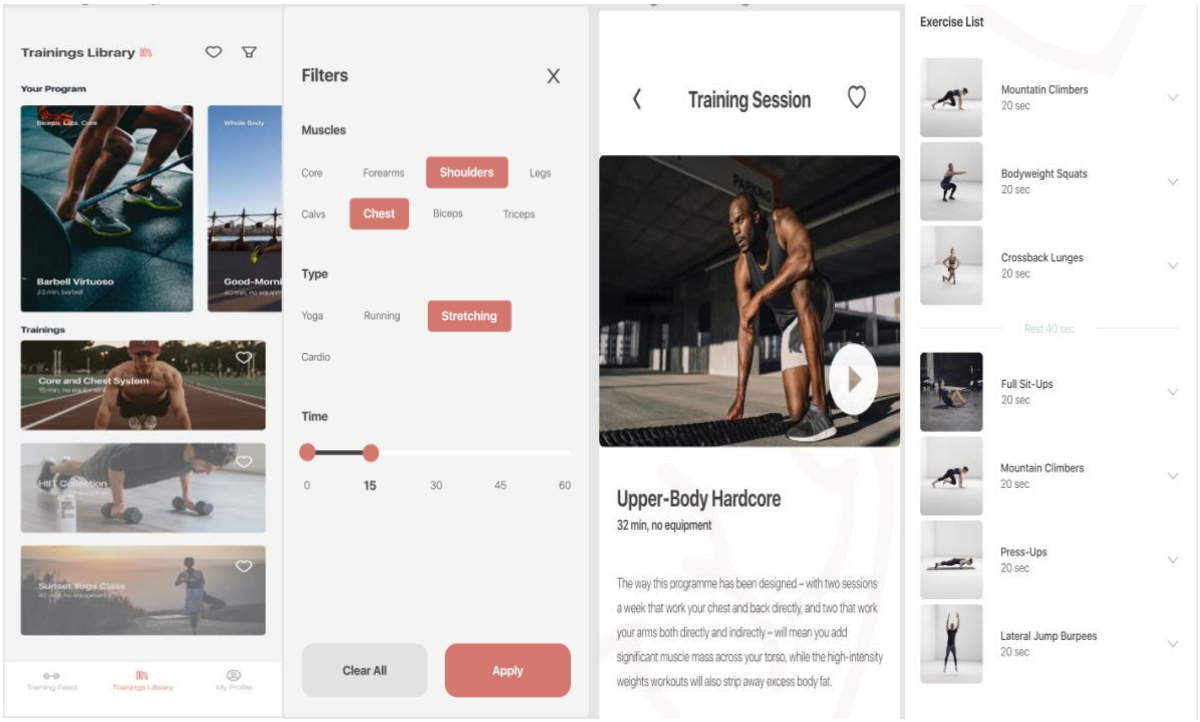
5.4.1 Розпорядок тренувань

Після успішного логіну або закінченої реєстрації користувач потрапляє на екран Training Feed. Тут йому надається його персональний календарний план тренувань, тижнева статистика та рекомендоване на сьогодні відео.

Рисунок 24 – Екран Training Feed

5.4.2 Бібліотека тренувань

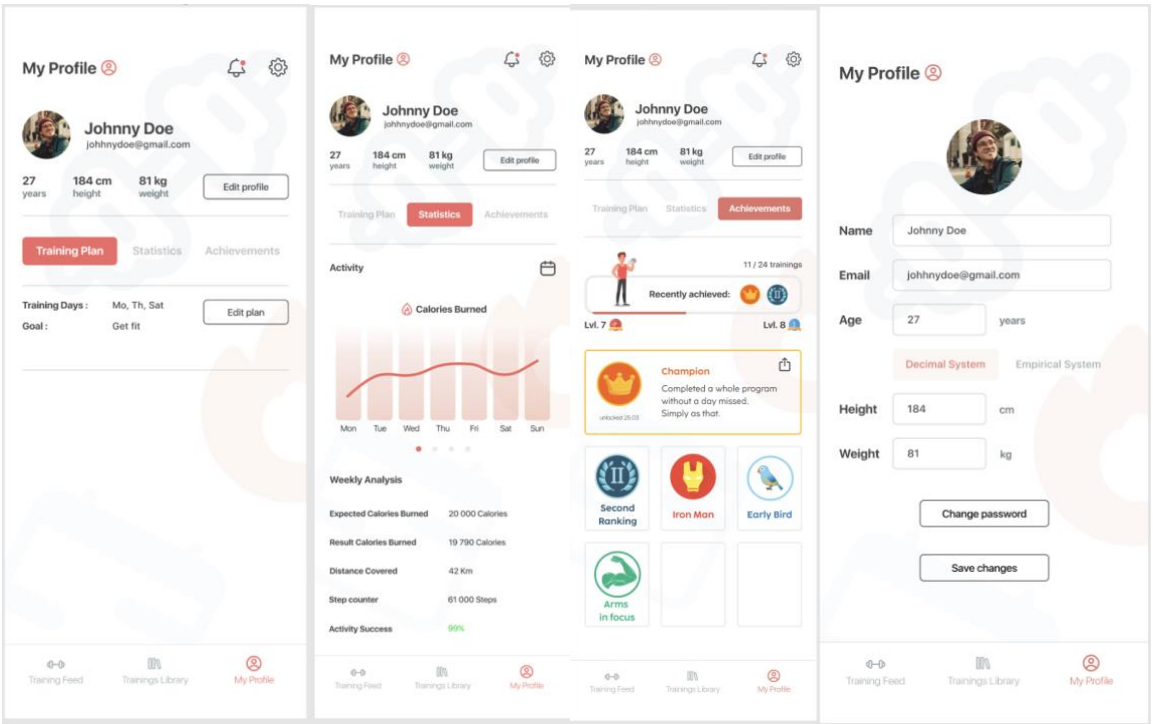
В цьому розділі юзер може знайти будь-які доступні тренування та зберігати їх собі, натиснувши на сердечко. Також може шукати тренування за особистими фільтрами, такими як: групи м’язів, тип тренування та час заняття. Кожне відео тренування поділяється на багато маленьких відео з вправами. Також між вправами користувач бачить довжину перерви на відпочинок.

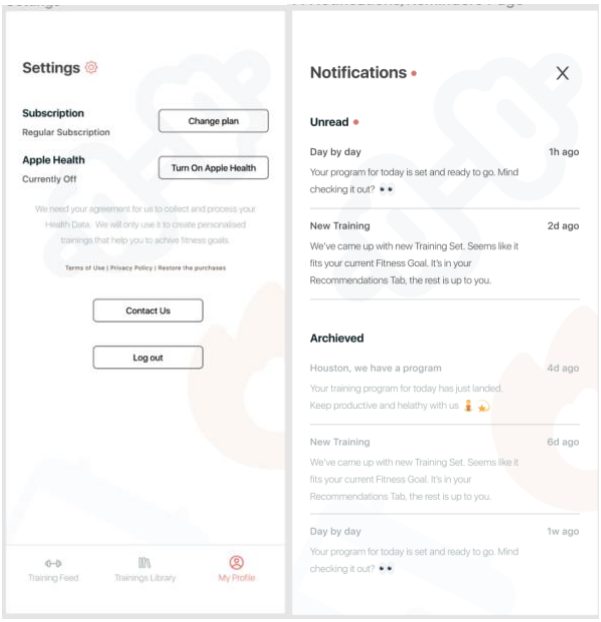


Рисунки 25 – Use case бібліотеки тренувань

5.4.3 Профіль

Тут користувач має можливість змінити персональні дані, скорегувати свій тренувальний план, змінити план підписки, налаштувати нотифікації, побачити статистику та досягнення у тренуваннях.





Рисунки 26 – Use case профілю

ВИСНОВКИ

Пандемія надала багато нових можливостей для розробників програмного софту, а особливо мобільних додатків. Значних обертів набрала і популярність мобільних додатків для фітнесу. В даний період часу, не дивлячись на те, що вже існує багато подібних програм, ситуація у світі дає можливість всім гідним додаткам у цій вертикалі зайняти свою нішу.

Проект був розроблений на нативній мові Swift, що дає можливість найраще адаптувати додаток саме під iOS. Було використано прогресивні паттерни програмування: Clean architecture + MVVM, це дозволяє грамотно розширювати проект не порушуючи правил SOLID та Dependency Injection.

Автоматичне тестування є однією з найголовніших технологій у світі тестування. Дана технологія допомагає компаніям швидше випустити нові функції на ринок програмних продуктів, забезпечивши користувача додатком без помилок. Перспективних шляхів в його розробці та використанні - безліч. Починаючи від інтеграції в розробки одно користувацьких додатків, закінчуючи інноваційними розробками програмного забезпечення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

[1] - Freeletics surveys Americans to uncover what is in store for the fitness industry post-COVID-19 [Електронний ресурс] – Режим доступу до ресурсу: https://www.freeletics.com/en/press/news/freeletics-surveys-americans-to-understand-what-is-in-store-for-the-fitness-industry-post-covid-19/.

[2] - Fitness apps grew by nearly 50% during the first half of 2020, study finds [Електронний ресурс] – Режим доступу до ресурсу: https://www.weforum.org/agenda/2020/09/fitness-apps-gym-health-downloads/.

[3] - Xamarin vs. React Native [Електронний ресурс] – Режим доступу до ресурсу: https://blog.logrocket.com/xamarin-vs-react-native/.

[4] - Native vs. Cross-Platform Apps – You'll Be the Winner [Електронний ресурс] – Режим доступу до ресурсу: https://www.zeolearn.com/magazine/native-vs-cross-platform-apps-youll-be-the-winner.

[5] - Swift The powerful programming language that is also easy to learn. [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/swift/.

[6] - Modular Architecture in iOS [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/flawless-app-stories/a-modular-architecture-in-swift-aafd9026aa99.

[7] - Clean Architecture and MVVM on iOS [Електронний ресурс] – Режим доступу до ресурсу: https://tech.olx.com/clean-architecture-and-mvvm-on-ios-c9d167d9f5b3.

ДОДАТОК А

Приклад коду з модуля аутентифікації

AuthenticationModule

```
Import
UIKit
        import Networking
        import Services

        public struct AuthenticationModuleDependencies {
            let networkClient: NetworkClientProtocol
            let userDefaults: UserDefaults

            public init(networkClient: NetworkClientProtocol, userDefaults:
        UserDefaults) {
                self.networkClient = networkClient
                self.userDefaults = userDefaults
            }
        }

        public final class AuthenticationModule {
            private let diContainer: AuthenticationDIContainer
            private var flow: AuthenticationFlowCoordinator?

            private var didLoginEnd: Closure?
            private var didRegisterEnd: Closure?

            public init(dependencies: AuthenticationModuleDependencies) {
                self.diContainer = AuthenticationDIContainer(dependencies:
        dependencies)
            }

            public func startAuthenticationFlow(in navigationController:
        UINavigationController) {
                flow =
        diContainer.makeAuthenticationFlowCoordinator(navigationController:
        navigationController)
                flow?.onLoginEnd { [weak self] in
                    guard let self = self else { return }
                    self.didLoginEnd?()
                }
                flow?.onRegisterEnd { [weak self] in
                    guard let self = self else { return }
```

```
                self.didRegisterEnd?()
            }
            flow?.start()
        }

        public func getUserTokenKey() -> String {
            return KeyConstants.userDefaultsTokenKey
        }

        @discardableResult
        public func onLoginEnd(closure: @escaping Closure) -> Self {
            didLoginEnd = closure
            return self
        }

        @discardableResult
        public func onRegisterEnd(closure: @escaping Closure) -> Self {
            didRegisterEnd = closure
            return self
        }
    }
```

AuthenticationDIContainer

```
import
UIKit

        final class AuthenticationDIContainer {

            private let dependencies: AuthenticationModuleDependencies

            init(dependencies: AuthenticationModuleDependencies) {
                self.dependencies = dependencies
            }

            func makeAuthenticationFlowCoordinator(navigationController:
        UINavigationController)
                -> AuthenticationFlowCoordinator {
                return AuthenticationFlowCoordinator(navigationController:
        navigationController, dependencies: self)
            }
        }

        extension AuthenticationDIContainer:
        AuthenticationFlowCoordinatorDependencies {
```

```
    func makeAuthenticationApiClient() -> AuthenticationApiClient {
        return AuthenticationApiClient(networkClient:
dependencies.networkClient)
    }

    func makeLoginViewModel() -> LoginViewModelProtocol {
        return LoginViewModel()
    }

    func makeIntroViewController(viewModel: LoginViewModelProtocol) ->
AuthenticationViewControllerProtocol {
        return IntroViewController(viewModel: viewModel)
    }

    func makeLoginViewController(viewModel: LoginViewModelProtocol) ->
AuthenticationViewControllerProtocol {
        return LoginViewController(viewModel: viewModel)
    }

    func makeRegisterViewController(viewModel: LoginViewModelProtocol)
-> AuthenticationViewControllerProtocol {
        return RegisterViewController(viewModel: viewModel)
    }

    func makeAuthenticationRepository() ->
AuthenticationRepositoryProtocol {
        return AuthenticationRepository(userDefaults:
dependencies.userDefaults)
    }
}
```

AuthenticationFlowCoordinator

```
import
UIKit
 import Services

 protocol AuthenticationFlowCoordinatorDependencies {
     func makeAuthenticationApiClient() -> AuthenticationApiClient
     func makeAuthenticationRepository() ->
 AuthenticationRepositoryProtocol
     func makeLoginViewModel() -> LoginViewModelProtocol
     func makeIntroViewController(viewModel: LoginViewModelProtocol) ->
 AuthenticationViewControllerProtocol
     func makeLoginViewController(viewModel: LoginViewModelProtocol) ->
 AuthenticationViewControllerProtocol
```

```
    func makeRegisterViewController(viewModel: LoginViewModelProtocol)
-> AuthenticationViewControllerProtocol
}

final class AuthenticationFlowCoordinator {
    private var navigationController: UINavigationController?
    private let dependencies:
AuthenticationFlowCoordinatorDependencies
    private lazy var apiClient: AuthenticationApiClientProtocol =
dependencies.makeAuthenticationApiClient()
    private lazy var repository: AuthenticationRepositoryProtocol =
dependencies.makeAuthenticationRepository()
    private lazy var loginViewModel: LoginViewModelProtocol =
dependencies.makeLoginViewModel()
    private lazy var introVC: AuthenticationViewControllerProtocol =
makeIntroVC()
    private lazy var loginVC: AuthenticationViewControllerProtocol =
makeLoginVC()
    private lazy var registerVC: AuthenticationViewControllerProtocol
= makeRegisterVC()

    private var didLoginEnd: Closure?
    private var didRegisterEnd: Closure?

    init(navigationController: UINavigationController,
         dependencies: AuthenticationFlowCoordinatorDependencies) {
        self.navigationController = navigationController
        self.dependencies = dependencies
    }

    func start() {
        navigationController?.pushViewController(introVC, animated:
false)
    }

    func makeIntroVC() -> AuthenticationViewControllerProtocol {
        let introVC = dependencies.makeIntroViewController(viewModel:
loginViewModel)
        introVC.onTapLogInButton { [weak self] in
            guard let self = self else { return }
            self.checkEmailExistence()
        }
        return introVC
    }

    func makeLoginVC() -> AuthenticationViewControllerProtocol {
```

```
        let loginVC = dependencies.makeLoginViewController(viewModel:
loginViewModel)
        loginVC.onTapLogInButton { [weak self] in
            guard let self = self else { return }
            self.login()
        }
        return loginVC
    }

    func makeRegisterVC() -> AuthenticationViewControllerProtocol {
        let registerVC =
dependencies.makeRegisterViewController(viewModel: loginViewModel)
        registerVC.onTapLogInButton { [weak self] in
            guard let self = self else { return }
            self.register()
        }
        return registerVC
    }

    func checkEmailExistence() {
        guard let userEmail = self.loginViewModel.getUserEmail() else
{
            return
        }
        apiClient.emailExistenceRequest(email: userEmail) { [weak
self] result in
            guard let self = self else { return }
            switch result {
            case .success(let value):
                guard let exists = value.emailExists else {
                    return
                }
                if exists {

self.navigationController?.pushViewController(self.loginVC, animated:
false)
                } else {

self.navigationController?.pushViewController(self.registerVC,
animated: false)
                }
            case .failure(let error):
                print(error)
                return
            }
        }
    }
```

```
    func login() {
        guard let userEmail = self.loginViewModel.getUserEmail(),
              let password = self.loginViewModel.getUserPassword()
else {
            return
        }
        apiClient.loginRequest(email: userEmail, password: password) {
[weak self] result in
            guard let self = self else { return }
            switch result {
            case .success(let value):
                guard let token = value.token else {
                    return
                }
                self.repository.setUserToken(token: token)
                self.didLoginEnd?()
            case .failure(let error):
                print(error)
                return
            }
        }
    }

    func register() {
        guard let userEmail = self.loginViewModel.getUserEmail(),
              let password = self.loginViewModel.getUserPassword()
else {
            return
        }
        apiClient.registerRequest(email: userEmail,
                                  password: password) { [weak self]
result in
            guard let self = self else { return }
            switch result {
            case .success(let value):
                guard let token = value.token else {
                    return
                }
                self.repository.setUserToken(token: token)
                self.didRegisterEnd?()
            case .failure(let error):
                print(error)
                return
            }
        }
    }
```

```
    @discardableResult
    func onLoginEnd(closure: @escaping Closure) -> Self {
        didLoginEnd = closure
        return self
    }

    @discardableResult
    func onRegisterEnd(closure: @escaping Closure) -> Self {
        didRegisterEnd = closure
        return self
    }
}
```