Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»
_______________Інститут електроенергетики ____________
(інститут)
________________факультет інформаційних технологій _____________
(факультет)
Кафедра __________інформаційних технологій та комп'ютерної інженерії
____
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня ____
бакалавра___
(бакалавра, спеціаліста, магістра)

студента _________Приймакова Родіона Віталійовича_______________________

(ПІБ)
академічної групи _____________126-21-2____________________________
(шифр)
спеціальності _____126 «Інформаційні системи та технології _______________
(код і назва спеціальності)
за освітньо-професійною програмою ___________________________________
(за наявності)
_______________________«Інформаційні системи та технології»______________
(офіційна назва)

на тему __Розробка мобільного застосунку для моніторингу здоров'я користувачів__
____

(назва за наказом ректора)

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
кваліфікаційної роботи	проф. Коротенко Г.М.			
розділів:				

Рецензент	Доцент Ширін А. Л.			

Нормоконтролер	проф. Коротенко Г.М.			

Дніпро 2025

ЗАТВЕРДЖЕНО:
завідувач кафедри
______інформаційних технологій _____
_____та комп'ютерної інженерії______
(повна назва)
_______________ _Гнатушенко В.В._
(підпис) (прізвище, ініціали)

«_____»_____________2025_року

ЗАВДАННЯ на кваліфікаційну роботу
ступеня ____бакалавра _______
(бакалавра, спеціаліста, магістра)

студенту______Приймакову Р. В.________ **академічної групи** _____126-21-2__
(прізвище та ініціали) (шифр)

спеціальності ______126 «Інформаційні системи та технології»______________

за освітньою-професійною програмою ______________________________

(за наявності)

______«Інформаційні системи та технології»_____________________ **на тему** ___
Розробка мобільного застосунку для моніторингу здоров'я користувачів
____ затверджену
наказом ректора НТУ «Дніпровська політехніка» від 05.05.2025 р. .№ 336-с

Розділ	Зміст	Термін виконання
Розділ 1	Теоретичні аналіз стану області рішення завдання	01.04.2025 – 28.04.2025
Розділ 2	Проєктні рішення	01.05.2025 – 30.05.2025

Завдання видано __________________ ___Коротенко Г. М.___
(підпис керівника) (прізвище, ініціали)

Дата видачі 01.02.2025 р

Дата подання до екзаменаційної комісії ____17.06.2025.___

Прийнято до виконання _____________ ____________Приймаков Р. В._____
(підпис студента) (прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 63 стор., 12 рис., 1 табл., 25 джерел.

Об’єкт розробки: мобільний застосунок для моніторингу здоров’я користувачів.

Мета кваліфікаційної роботи: розробка мобільного застосунку для збору, обробки та аналізу даних про стан здоров’я користувача з використанням мови програмування Kotlin.

У вступі розглянуто актуальність проблеми контролю здоров’я за допомогою цифрових технологій та обґрунтовано потребу у створенні зручного мобільного застосунку для персонального моніторингу фізичного стану.

У першому розділі кваліфікаційної роботи проведено аналіз існуючих мобільних сервісів, визначено їхні функціональні можливості, а також сформовано вимоги користувачів до таких застосунків. Окрім цього, обґрунтовано вибір платформи та мови розробки — Kotlin для Android.

У другому розділі описано проектні рішення: архітектуру застосунку, структуру бази даних, механізми авторизації, особливості реалізації інтерфейсу користувача та тестування працездатності програмного продукту.

Практичне значення роботи полягає у створенні мобільного інструменту, який дозволяє користувачам здійснювати самостійний контроль ключових показників здоров’я, що сприяє підвищенню обізнаності про свій фізичний стан та покращенню якості життя.

Список ключових слів: МОБІЛЬНИЙ ЗАСТОСУНОК, ЗДОРОВ’Я КОРИСТУВАЧА, МОНИТОРИНГ, ANDROID, KOTLIN, SQLITE, UI/UX, API, АВТОРИЗАЦІЯ, МОБІЛЬНА РОЗРОБКА

ABSTRACT

Explanatory note: 63 pages, 12 figures, 1 table, 25 sources.

Objective of the qualification work: development of a mobile application for collecting, processing, and analyzing data about the user's health status using the Kotlin programming language.

Purpose of the qualification work: to develop a mobile application for collecting, processing, and analyzing data on the user's health status using the Kotlin programming language.

The introduction outlines the relevance of the problem of health monitoring using digital technologies and justifies the need to create a convenient mobile application for personal monitoring of physical condition.

The first chapter of the qualification work analyzes existing mobile services, identifies their functional capabilities, and defines user requirements for such applications. Additionally, it justifies the choice of the development platform and programming language — Kotlin for Android.

The second chapter describes the design decisions: the application architecture, the database structure, authorization mechanisms, features of the user interface implementation, and functionality testing of the software product.

The practical value of this work lies in the creation of a mobile tool that enables users to independently monitor key health indicators, thereby increasing awareness of their physical condition and improving their quality of life.

Keywords: MOBILE APPLICATION, USER HEALTH, MONITORING, ANDROID, KOTLIN, SQLITE, UI/UX, API, AUTHORIZATION, MOBILE DEVELOPMENT

ЗМІСТ

ВСТУП **6**

РОЗДІЛ І ТЕОРЕТИЧНІ АНАЛІЗ СТАНУ ОБЛАСТІ РІШЕННЯ ЗАВДАННЯ **8**

1.1 Огляд існуючих мобільних застосунків для моніторингу здоров’я . 8

1.2 Основні функції та можливості аналогічних сервісів........ 12

1.3 Аналіз вимог користувачів до таких застосунків........ 14

1.4 Вибір платформи та мови програмування 16

1.5 Постановка завдання на розробку........ 17

РОЗДІЛ ІІ ПРОЄКТНІ РІШЕННЯ **20**

2.1 Архітектура мобільного застосунку 20

2.2 Підключення бази даних........ 22

2.3 Налаштування авторизації........ 29

2.4 Створення інтерфейсу користувача........ 37

2.5 Тестування застосунку........ 51

ВИСНОВКИ........ **59**

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ **61**

ВСТУП

У сучасному світі питання здоров'я та добробуту людини набувають особливого значення. Зростаюча популярність здорового способу життя, а також підвищена увага до профілактики та контролю фізичного стану організму сприяли розвитку цифрових інструментів, що дозволяють користувачам ефективно стежити за власним здоров'ям. Одним із найпоширеніших рішень у цій сфері є мобільні застосунки для моніторингу здоров'я, які забезпечують доступ до ключових даних про фізичну активність, сон, харчування, рівень стресу, частоту серцебиття тощо.

Світовий ринок мобільних додатків для здоров'я активно зростає. Такі компанії, як Google (Google Fit), Apple (Apple Health), Samsung (Samsung Health) та інші розробляють багатофункціональні рішення, орієнтовані на інтеграцію з носимими пристроями, що відкриває нові можливості для безперервного моніторингу фізіологічних параметрів. Однак попри широке розповсюдження подібних сервісів, існує потреба у створенні адаптованих та доступних рішень, які враховують локальні особливості, мовну підтримку, специфіку користувацьких запитів і можливість гнучкого налаштування функцій відповідно до індивідуальних потреб.

Аналіз науково-технічної літератури, присвяченої темі моніторингу здоров'я за допомогою цифрових технологій, свідчить про те, що значна частина досліджень зосереджена на аналізі біометричних даних, візуалізації інформації, забезпеченні конфіденційності персональних даних користувача та інтеграції застосунків з хмарними сервісами. Утім, на рівні індивідуальних користувачів існує недостатня кількість гнучких, простих у користуванні мобільних рішень, які дозволяли б контролювати власне самопочуття, вести щоденник симптомів або звичок, отримувати рекомендації та нагадування.

Актуальність теми цієї атестаційно роботи обумовлена потребою у створенні зручного мобільного застосунку, який би дозволяв користувачам вести облік особистих даних щодо здоров'я, отримувати зворотний зв'язок у

вигляді графіків та інтерпретацій, а також інтегруватися з іншими пристроями або сервісами. Такий інструмент має потенціал як для особистого використання, так і для застосування у рамках телемедицини або віддалених консультацій з лікарями.

Метою цієї роботи є розробка мобільного застосунку для моніторингу здоров'я користувачів на платформі Android із використанням мови Kotlin, що дозволяє забезпечити зручність, швидкодію та інтеграцію з сучасними технологіями мобільної розробки.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Провести огляд існуючих мобільних застосунків для моніторингу здоров'я та визначити їхні функціональні особливості.
2. Визначити основні потреби користувачів у контексті функціональності подібних застосунків.
3. Обґрунтувати вибір мови програмування Kotlin та програмних засобів для реалізації проєкту.
4. Розробити архітектуру мобільного застосунку з урахуванням вимог до безпеки, гнучкості та масштабованості.
5. Реалізувати функціонал бази даних, авторизації та обміну даними.
6. Створити зручний і зрозумілий користувацький інтерфейс.
7. Провести тестування застосунку на відповідність функціональним та нефункціональним вимогам.

Об'єктом дослідження є процеси моніторингу фізичного стану людини з використанням мобільних технологій. Предметом дослідження є методи та інструменти розробки мобільних застосунків, які забезпечують ефективну фіксацію та аналіз даних про здоров'я користувача.

Методи дослідження включають аналіз літературних джерел, вивчення існуючих застосунків, застосування принципів об'єктно-орієнтованого програмування, розробку архітектури програмного забезпечення, реалізацію інтерфейсу та тестування програмного продукту в реальних умовах.

РОЗДІЛ І
ТЕОРЕТИЧНІ АНАЛІЗ СТАНУ ОБЛАСТІ РІШЕННЯ ЗАВДАННЯ

1.1 Огляд існуючих мобільних застосунків для моніторингу здоров’я

У сучасному цифровому середовищі мобільні застосунки для моніторингу здоров’я стали важливими інструментами, що дозволяють користувачам слідкувати за фізичним та емоційним станом, вести облік щоденної активності, контролювати сон, пульс, споживання калорій, рівень стресу, а також отримувати персоналізовані рекомендації щодо покращення способу життя. На ринку існує безліч таких додатків, однак серед них особливо виділяються три найпопулярніші: Apple Health, Google Fit та Samsung Health, кожен з яких має свої переваги та недоліки.

Apple Health – це офіційний додаток компанії Apple, вперше представлений у 2014 році як частина iOS 8 [1]. Його головне призначення – централізовано збирати, зберігати й аналізувати дані про фізичну активність, медичні показники та загальний стан здоров’я користувача (рис. 1.1).

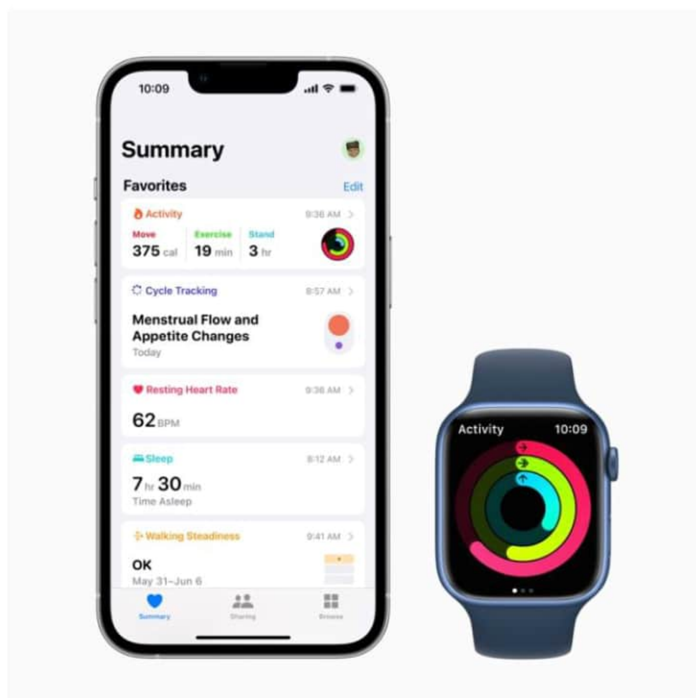

Рисунок 1.1 – Apple Health

Apple Health автоматично фіксує кількість кроків, пройдену відстань, сходи, що були подолані, і спалені калорії завдяки вбудованим сенсорам iPhone. З Apple Watch функціональність суттєво розширюється — додаток може відстежувати серцевий ритм, рівень кисню в крові, ЕКГ, фазу сну, а також фіксувати падіння чи аномальні зміни пульсу.

Крім того, у додатку є окремий розділ «Medical ID», де можна зберігати життєво важливу інформацію (група крові, алергії, хронічні хвороби, контактні дані для екстрених випадків). Apple Health також підтримує інтеграцію з понад 1000 сторонніми застосунками — від трекерів харчування і води до фітнес-додатків та медичних сервісів. Унікальною особливістю є функція «Health Records» (в деяких країнах), яка дозволяє пацієнтам автоматично отримувати електронні медичні записи зі своїх лікарень або клінік.

Перевагами Apple Health є глибока інтеграція з апаратною частиною Apple, високий рівень захисту даних (використовується наскрізне шифрування), а також велика гнучкість у роботі з даними. Проте доступність лише для пристроїв Apple, залежність від Apple Watch для збору багатьох показників та складність у взаємодії з лікарями в країнах, де не підтримуються «Health Records», становлять помітні недоліки.

Google Fit — це фірмовий продукт Google, орієнтований на Android-користувачів, створений у партнерстві з Американською кардіологічною асоціацією та Всесвітньою організацією охорони здоров'я [2]. Його головною метою є сприяння фізичній активності за допомогою простої системи моніторингу. Основними показниками, які фіксує Google Fit, є кількість кроків, хвилини активності, калорії, серцевий ритм, відстань, сон. Одна з головних особливостей — система «Heart Points» і «Move Minutes», що мотивують користувача бути активним за рекомендаціями ВООЗ (рис. 1.2).

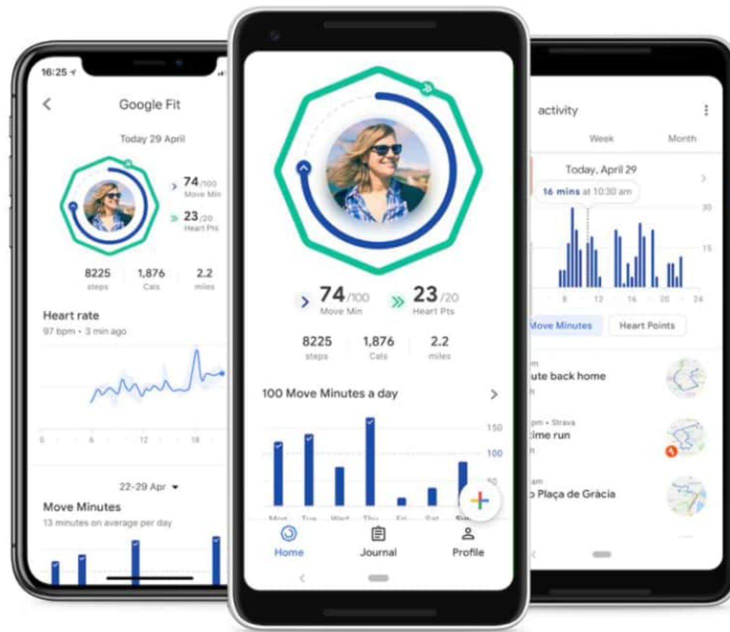

Рисунок 1.2 – Google Fit

Додаток підтримує синхронізацію з численними фітнес-трекерами (Fitbit, Xiaomi Mi Band, Garmin, Polar), додатками (Strava, Runkeeper, Nike Training Club, MyFitnessPal), а також із пристроями Wear OS. Користувачі можуть переглядати статистику у вигляді днів, тижнів або місяців, ставити цілі та отримувати прості поради.

Проте Google Fit має менш глибокий функціонал у порівнянні з конкурентами: відсутні вбудовані медичні карти, відстеження менструального циклу, рівня кисню в крові, настрою або докладного сну без сторонніх додатків. Деякі користувачі також вказують на нестабільну синхронізацію з певними пристроями та неточності в підрахунку активності. Однак загалом Google Fit залишається популярним через свою простоту, кросплатформеність та відкриту архітектуру.

Samsung Health — це розвинений фітнес і медичний застосунок компанії Samsung, вперше представлений у 2012 році (спочатку як S Health). Він пропонує один із найширших спектрів функцій серед подібних продуктів. Додаток дозволяє не лише рахувати кроки, відстань, спалені калорії, а й відстежувати пульс, рівень стресу, якість сну, кількість випитої води,

харчування, масу тіла, менструальний цикл, артеріальний тиск. У Samsung Health є режим «Together», що дозволяє змагатися з друзями, проходити фітнес-челенджі, а також переглядати глобальні лідерборди [3].

Рисунок 1.3 – Samsung Health

Завдяки тісній інтеграції з пристроями Samsung, такими як Galaxy Watch, додаток може виконувати ЕКГ, замір рівня SpO2 та навіть аналізувати фази сну з точністю до фаз REM. Крім того, у застосунку є модуль «Mindfulness» з медитаціями, дихальними вправами, а також підтримка зовнішніх додатків типу Calm або Fitbit. Важливим є й модуль безпеки: додаток зберігає «екстрену інформацію» та може бути використаний у надзвичайних ситуаціях.

Однак Samsung Health має складний інтерфейс із великою кількістю меню та налаштувань, що може бути перевантажуючим для новачків. Крім того, не всі функції доступні на смартфонах не від Samsung, і на деяких моделях обмежено підтримку навіть базових параметрів (наприклад, вимірювання пульсу або рівня кисню). Також певні функції, зокрема ЕКГ, можуть бути доступні лише в обмежених регіонах.

Тепер на основі даного аналізу можемо побудувати порівняльну таблицю для узагальнення (табл. 1.1).

Таблиця 1.1 – Порівняння Apple Health, Google Fit та Samsung Health

Характеристика	Apple Health	Google Fit	Samsung Health
Платформа	iOS, watchOS	Android, iOS	Android, iOS
Основні функції	Кроки, сон, пульс, медичні дані	Кроки, калорії, активність	Кроки, сон, пульс, їжа, вода
Сумісність із пристроями	Apple Watch	Wear OS, інші	Galaxy Watch, інші
Відстеження сну	Так (із Apple Watch)	Через сторонні додатки	Так (вбудовано)
Пульс / SpO_2 / ЕКГ	Так / Так / Так	Через інші пристрої	Так / Так / Так
Менструальний цикл	Через сторонні додатки	Ні	Так (вбудовано)
Інтеграція з іншими сервісами	Висока	Висока	Обмежена
Захист даних	Високий	Високий	Високий
Інтерфейс	Простий	Зручний	Насичений, менш інтуїтивний

Таким чином, усі три платформи мають свою унікальну цінність. Apple Health зосереджений на медичній точності й екосистемній інтеграції, Google Fit — на універсальності, простоті та підтримці сторонніх сервісів, а Samsung Health — на багатофункціональності, інтенсивному відстеженні стану тіла та духу, а також гейміфікації здоров’я. Вибір між ними багато в чому залежить від типу пристрою, особистих пріоритетів користувача та його потреб у зборі й обробці інформації про здоров’я.

1.2 Основні функції та можливості аналогічних сервісів

Застосунки для моніторингу здоров'я виконують широкий спектр функцій, спрямованих на забезпечення користувачам постійного контролю за фізичним і частково психоемоційним станом. Їх основна мета — надати користувачеві інструменти для ведення здорового способу життя, раннього виявлення потенційних проблем зі здоров'ям та стимулювання до активності. Основні функції таких сервісів умовно можна поділити на кілька категорій: фізична активність, фізіологічні показники, харчування, сон, ментальне здоров'я та інтеграція з медичними системами [4].

Найпоширенішою функцією є підрахунок кроків. Застосунки, як правило, використовують дані акселерометра або синхронізуються зі смарт-годинниками для точнішого відстеження. На основі кількості пройдених кроків формується інформація про витрачені калорії та загальний рівень активності протягом дня. Крім того, більшість сервісів пропонують можливість встановлення цілей активності, які мотивують користувачів більше рухатися.

Моніторинг фізіологічних показників — ще один важливий напрямок. Додатки можуть відстежувати частоту серцевих скорочень, насичення крові киснем (SpO_2), артеріальний тиск, а також рівень стресу. Ці дані особливо корисні для користувачів із хронічними захворюваннями або тих, хто веде активний спосіб життя. В деяких випадках, наприклад у Apple Health чи Samsung Health, передбачено навіть функціональність для запису ЕКГ, що дозволяє отримати попереднє уявлення про серцевий ритм.

Важливим напрямом є контроль сну. Застосунки дозволяють відстежувати тривалість сну, його фази (глибокий, легкий, REM), визначати якість відпочинку та рекомендувати оптимальні години для засинання. Деякі програми інтегрують функції “розумного будильника”, який будить людину в оптимальну фазу сну, щоб уникнути відчуття розбитості після пробудження.

Багато мобільних сервісів також приділяють увагу харчуванню. Користувач може фіксувати прийоми їжі, підраховувати калорії, контролювати споживання води, білків, жирів, вуглеводів та інших нутрієнтів.

Завдяки цьому формується більш повна картина здоров’я користувача і впливу способу харчування на його стан.

Ще однією корисною функцією є ведення щоденника самопочуття або рівня стресу, що дозволяє користувачеві відстежувати зміни в емоційному стані, корелювати їх із фізичною активністю та якістю сну. У деяких застосунках доступна інтеграція з медичними картками, аналізами та можливість додавання медичних даних, що може бути корисним під час візиту до лікаря [5].

Крім того, важливою є синхронізація з іншими пристроями та платформами: смарт-годинниками, фітнес-браслетами, глюкометрами, тонометрами. Сучасні застосунки часто дозволяють передавати зібрані дані до хмарних сервісів, ділитися ними з лікарями або членами родини.

1.3 Аналіз вимог користувачів до таких застосунків

Мобільні застосунки для моніторингу здоров’я користувачів займають важливе місце в повсякденному житті, адже вони дозволяють контролювати фізичну активність, здоров’я та загальний стан організму. Тому вимоги до таких додатків є дуже різноманітними та стосуються не тільки їх функціональності, а й зручності користування, безпеки даних та естетики інтерфейсу.

Однією з основних вимог є можливість відстеження фізичної активності, що включає моніторинг кількості кроків, відстані, витрачених калорій та часу активності. Користувачі також бажають мати можливість синхронізувати застосунок з фітнес-трекерами або смарт-годинниками для більш точного збору даних. Це дозволяє не тільки отримувати детальну інформацію про фізичну активність, а й створювати персоналізовані плани тренувань або відпочинку.

Ще однією важливою вимогою є моніторинг сну. Користувачі хочуть отримувати точні дані про тривалість сну, його фази, а також мати можливість

отримувати рекомендації для покращення якості сну. Це може включати аналіз часу засинання, частоти пробуджень і загального часу сну, що дає змогу користувачам вчасно коригувати свої звички для поліпшення самопочуття.

Користувачі також зацікавлені у веденні особистих записів щодо свого стану здоров'я, прийому ліків, харчування та інших важливих аспектів. Це дозволяє не лише стежити за процесом, а й отримувати нагадування про необхідні дії, такі як прийом ліків або достатню кількість води протягом дня. Для багатьох користувачів дуже важливо мати можливість самостійно вводити дані, щоб точно контролювати різні аспекти здоров'я.

Особливо важливим є наявність інструментів для аналітики та створення звітів. Користувачі хочуть бачити результати своїх зусиль у вигляді наочних графіків і статистики, що допомагає їм відслідковувати динаміку змін, оцінювати прогрес і вчасно коригувати свій підхід до здоров'я. Такі звіти можуть бути зібрані за певний період часу та допомогти користувачам краще розуміти, як їхня діяльність впливає на здоров'я [6].

Не менш важливими є вимоги до зручності використання та інтерфейсу. Користувачі віддають перевагу простому та інтуїтивно зрозумілому інтерфейсу, який дозволяє швидко знайти необхідну інформацію або виконати певну дію. Додатки повинні бути адаптовані під різні розміри екранів, мати чітке розподілення функцій і доступ до основних можливостей без зайвих кроків.

Окрім того, безпека персональних даних є одним з основних аспектів. Оскільки багато користувачів діляться своїми особистими даними через додатки для моніторингу здоров'я, важливо забезпечити надійний захист цієї інформації. Це включає використання шифрування даних, надійних систем автентифікації та захисту від несанкціонованого доступу.

1.4 Вибір платформи та мови програмування

При розробці мобільного застосунку для моніторингу здоров'я важливим етапом є вибір платформи та мови програмування, оскільки це безпосередньо впливає на ефективність розробки, функціональність і майбутню підтримку застосунку. Для цього проекту було обрано платформу Android та мову програмування Kotlin.

Android є однією з найбільш популярних мобільних платформ у світі, що має величезну кількість користувачів. Це дає змогу забезпечити широкий охоплення аудиторії. Крім того, Android підтримує безліч пристроїв з різними характеристиками, що є важливим для застосунків, орієнтованих на здоров'я, оскільки користувачі можуть мати різні смартфони. Платформа надає багатий набір інструментів для розробки мобільних застосунків, включаючи можливість інтеграції з різноманітними сенсорами, такими як пульсометри, акселерометри та GPS, що дозволяє зібрати необхідні дані для моніторингу фізичного стану користувачів. Android також забезпечує можливості для збереження даних, що є важливим для реалізації функціоналу історії стану здоров'я користувача [7].

Мова програмування Kotlin була обрана завдяки її багатьом перевагам. Kotlin є офіційно підтримуваною мовою для розробки на платформі Android. Це сучасна мова, яка поєднує функціональність, безпеку та простоту. Однією з основних переваг Kotlin є його сумісність з Java, що дає можливість інтегрувати існуючі бібліотеки і рішення, розроблені на Java, в новий проект. Крім того, Kotlin має більш лаконічний синтаксис порівняно з Java, що дозволяє зменшити кількість коду та підвищити його читаємість. Це також знижує ймовірність помилок під час розробки та спрощує подальше обслуговування застосунку [8].

Kotlin також дозволяє уникати багатьох поширених помилок програмування завдяки вбудованим засобам безпеки, зокрема безпечному управлінню нульовими значеннями. Це є важливим аспектом для проекту,

пов'язаного з даними користувачів, де точність та безпека даних є критичними. Мова також підтримує функціональні можливості, такі як лямбда-вирази, що дозволяють легко працювати з асинхронними задачами та колекціями, що робить розробку ефективнішою.

Що стосується середовища розробки, було вибрано Android Studio. Це офіційне середовище для розробки застосунків на платформі Android, яке пропонує всі необхідні інструменти для створення, налагодження та тестування мобільних додатків. Android Studio має інтеграцію з Kotlin, що робить розробку ще зручнішою, а також включає потужний графічний редактор для створення інтерфейсу користувача. Середовище також надає інструменти для профілювання застосунку, тестування на емуляторах різних пристроїв і аналізу продуктивності, що особливо важливо для створення застосунку для моніторингу здоров'я, де важливі швидка реакція та точність [9].

Таким чином, вибір платформи Android і мови програмування Kotlin, разом з використанням Android Studio, забезпечує оптимальне середовище для розробки мобільного застосунку для моніторингу здоров'я, зручне та ефективне управління даними користувачів, а також швидке реагування на запити та можливість тестувати і оптимізувати застосунок для різних пристроїв.

1.5 Постановка завдання на розробку

Завдання на розробку мобільного застосунку для моніторингу здоров'я користувачів полягає у створенні функціонального та надійного додатка зі зручним інтерфейсом для відслідковування фізичної активності та здоров'я. Застосунок буде призначений для платформи Android, з використанням Kotlin як основної мови програмування та Android Studio для розробки. Основна мета полягає в тому, щоб надати користувачам можливість слідкувати за їхнім здоров'ям, контролювати рівень активності, харчування та гідратації [10].

Застосунок дозволить відслідковувати кількість зроблених користувачем кроків протягом дня. Це буде реалізовано за допомогою вбудованого сенсора крокоміра, що автоматично підраховує кроки, а також дає можливість встановлювати індивідуальні цілі по кількості кроків. Крім того, користувачі зможуть контролювати споживання води, вводити інформацію про кількість випитої води та отримувати нагадування для підтримки рівня гідратації.

Іншим важливим аспектом буде можливість реєстрації калорій, як спожитих, так і витрачених. Користувачі зможуть записувати їжу, що вони споживають, а також враховувати витрачені калорії в процесі фізичної активності. Цей функціонал дасть можливість слідкувати за енергетичним балансом і підтримувати здорове харчування. Окрім цього, застосунок надасть можливість записувати прийоми їжі, що допоможе користувачам слідкувати за своїм раціоном та зробить управління харчуванням простим і зручним [11].

Для забезпечення безпеки даних користувачів буде використано Firebase Authentication. Це дозволить користувачам безпечно реєструватися, входити в систему та зберігати свої особисті дані. За допомогою цього інструменту буде забезпечено належне управління акаунтами, з можливістю автентифікації через електронну пошту або соціальні мережі, такі як Google чи Facebook.

Застосунок також передбачає інтеграцію з різними сервісами для відправлення сповіщень та нагадувань. Користувачі отримуватимуть повідомлення щодо важливих подій, таких як необхідність виконати фізичну вправу, випити воду чи їсти. Ці нагадування допоможуть підтримувати здорові звички.

Для зручності користувачів буде доступна функція встановлення будильників для нагадування про різноманітні здоров’я-орієнтовані активності. Це може бути зручний спосіб нагадування про важливі завдання, такі як виконання вправ або прийом їжі [12].

Щоб забезпечити безперервний моніторинг здоров’я, застосунок працюватиме у фоновому режимі, що дозволить відстежувати активність

користувачів без необхідності постійно тримати застосунок відкритим. Використання таких інструментів, як Fused Location Provider для точного відслідковування локації під час outdoor активностей, а також Firestore для зберігання даних про користувачів і Room Database для локального зберігання, дозволить забезпечити ефективне та безпечне управління даними.

РОЗДІЛ II
ПРОЄКТНІ РІШЕННЯ

2.1 Архітектура мобільного застосунку

Архітектура мобільного застосунку визначає його логічну структуру, взаємодію між компонентами та рівень масштабованості, підтримуваності й безпеки. Для реалізації застосунку моніторингу здоров'я користувача було обрано багаторівневу архітектуру, яка забезпечує чіткий поділ обов'язків між компонентами (рис. 2.1).

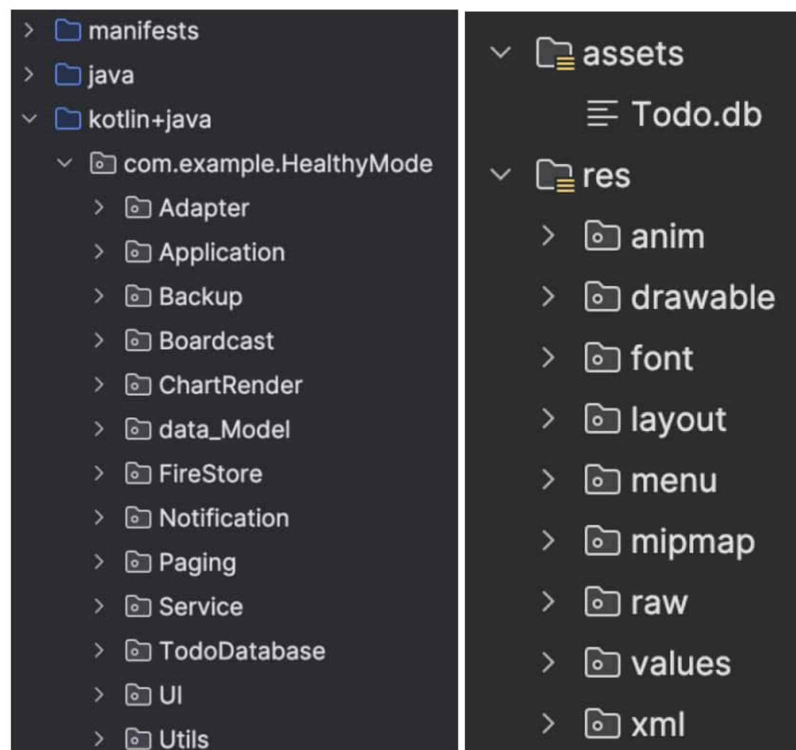

Рисунок 2.1 – Архітектура додатку

У проєкті реалізовано шаблон MVVM (Model-View-ViewModel), що дозволяє ефективно розділити логіку представлення, бізнес-логіку та доступ до даних. Такий підхід покращує тестованість, повторне використання коду та підтримку проєкту:

- Model — містить класи даних (data_Model), а також роботу з локальною базою даних (TodoDatabase) та хмарною базою Firestore (FireStore). Також до моделі входять сервіси, які відповідають за отримання, обробку та збереження даних (Service, Backup).
- View — екранні компоненти користувацького інтерфейсу (UI, layout, drawable, menu). Реалізовані за допомогою фрагментів та активностей.
- ViewModel — посередник між View та Model, що забезпечує зв'язок між даними та інтерфейсом, використовуючи LiveData та корутини.

Структура проєкту передбачає такі основні модулі:

- Adapter — містить адаптери для відображення списків (наприклад, історія активності, споживання води тощо).
- Application — базовий клас застосунку, де ініціалізуються глобальні сервіси.
- Boardcast — реалізує логіку обробки системних подій (наприклад, будильники, нагадування).
- Notification — реалізує систему локальних нагадувань і повідомлень.
- Paging — підтримує роботу з великими обсягами даних, реалізуючи посторінкове завантаження.
- ChartRender — відповідальний за побудову графіків стану здоров'я (напр., фізична активність, сон).
- Utils — містить допоміжні класи (форматування дат, конвертація даних тощо).

Для зберігання даних використовується подвійний підхід:

- Локальна база даних SQLite — розміщена в assets/Todo.db та доступна через пакет TodoDatabase.
- Хмарна база Firestore — забезпечує синхронізацію між пристроями та резервне зберігання в FireStore.

Застосунок інтегрований з Firebase, використовуючи:

- Firestore — для зберігання даних користувача;

- Firebase Authentication — для авторизації (буде описано у розділі 2.3);
- Firebase Cloud Messaging — для реалізації push-повідомлень (через Notification).

Використання Firebase дозволяє гнучко керувати правилами доступу до даних, забезпечуючи безпеку персональної інформації користувачів. Додатково реалізовано локальне шифрування резервних копій (Backup).

Компоненти взаємодіють між собою через:

- Інтерфейси та абстракції;
- LiveData та Flow (для реактивного оновлення інтерфейсу);
- Room (для доступу до SQLite).

Розроблена архітектура є гнучкою, масштабованою та забезпечує ефективне розділення обов’язків. Це дозволяє легко впроваджувати нові функції, проводити тестування та підтримувати застосунок у майбутньому [13].

2.2 Підключення бази даних

У застосунку для моніторингу здоров’я користувача реалізовано підключення до хмарної бази даних Firebase Firestore за допомогою бібліотеки Dagger Hilt для впровадження залежностей (Dependency Injection). Це дозволяє зручно й централізовано керувати екземплярами Firestore та іншими пов’язаними сервісами.

Файл FirebaseModule.kt відповідає за підключення мобільного застосунку до хмарної бази даних Firebase Firestore. Він реалізований як модуль Dagger Hilt, що дозволяє зручно управляти залежностями в межах усього проєкту. Клас FirebaseModule оголошено як об’єкт із допомогою анотації @Module, а завдяки анотації @InstallIn(SingletonComponent::class) він підключається до глобального скоупу застосунку, тобто залежності, створені тут, діятимуть протягом усього життєвого циклу застосунку [14].

```
@InstallIn(SingletonComponent::class)
@Module
object FirebaseModule {
```

Усередині модуля оголошується змінна userDitails, яка містить посилання на конкретний документ користувача в колекції user. UID користувача отримується за допомогою аутентифікації через FirebaseAuth, тому цей фрагмент передбачає, що користувач вже авторизований.

```
var userDitails: DocumentReference =
Firebase.firestore.collection("user").document(
FirebaseAuth.getInstance().currentUser!!.uid)
```

Крім того, в модулі визначено функцію provideFireStoreInstance(), яка за допомогою анотацій @Provides та @Singleton повертає екземпляр колекції user у Firestore. Ця функція використовується для ін'єкції залежності на рівні всього застосунку. Це дає змогу іншим компонентам отримувати доступ до бази даних без необхідності кожного разу вручну створювати екземпляри Firestore або колекцій — все це виконується автоматично через Hilt.

```
@Provides
@Singleton
fun provideFireStoreInstance(): CollectionReference {
    return
FirebaseFirestore.getInstance().collection("user")
}
```

RepositoryModule.kt відповідає за налаштування залежності для репозиторію, який забезпечує взаємодію з базою даних Firestore у мобільному застосунку. Він є частиною архітектури проєкту та використовує Dagger Hilt для впровадження залежностей у скоупі всього застосунку.

Файл описує об'єкт RepositoryModule, який є модулем Dagger Hilt, позначеним анотаціями @Module та @InstallIn(SingletonComponent::class). Це означає, що залежності, визначені в цьому модулі, діятимуть на рівні всього застосунку (тобто створюються один раз і використовуються повторно).

```
@RequiresApi(Build.VERSION_CODES.O)
@InstallIn(SingletonComponent::class)
@Module
object RepositoryModule {
```

Модуль містить одну функцію provideRepository(), яка позначена анотаціями @Provides і @Singleton. Вона приймає як параметр посилання на колекцію Firestore (CollectionReference), яке надається іншим модулем (наприклад, FirebaseModule), і на його основі створює екземпляр класу RepositoryImp. Цей клас є реалізацією інтерфейсу Repository і містить логіку роботи з базою даних, наприклад: зчитування, додавання чи оновлення документів.

```
@Provides
@Singleton
fun provideRepository(
    database: CollectionReference,
): Repository {
    return RepositoryImp(database)
}
```

Також файл позначений анотацією @RequiresApi (Build.VERSION_CODES.O), що вказує: для його використання необхідна мінімальна версія Android 8.0 (API 26), найімовірніше через використання функцій Java 8 у реалізації репозиторію.

Файл Repository.kt описує інтерфейс Repository, який є частиною архітектури застосунку і визначає контракт (тобто набір методів), через який інші компоненти програми можуть взаємодіяти з джерелами даних — зокрема з Firebase Firestore.

Інтерфейс визначає низку абстрактних функцій, які повинні бути реалізовані в класі-реалізації (наприклад, у RepositoryImp). Основна мета цього інтерфейсу — ізолювати бізнес-логіку від конкретної реалізації зберігання чи обробки даних. Це полегшує тестування, супровід та масштабування коду [15].

```
interface Repository {
    fun getWeight(result: (UIstate<ArrayList<Entry>>) ->
Unit)
    fun changedWeight(weight: weight, result:
(UIstate<String>) -> Unit)
    fun addFood(nutrient: Nutrient, result:
(UIstate<Nutrient>) -> Unit)
    fun getNutrients(mealTimes: List<String>):
LiveData<ArrayList<Float>>
    fun getcalories(mealTimes: List<String>):
LiveData<ArrayList<Float>>
    fun changeHeight(height: String,context:Context)
    fun getheight(result: (UIstate<String>) -> Unit)
}
```

Методи інтерфейсу охоплюють наступну функціональність:

- getWeight() — отримання даних про вагу користувача у вигляді масиву точок (Entry) для побудови графіка, з використанням замикання з результатом типу UIstate.
- changedWeight() — зміна або оновлення даних про вагу користувача, зворотній результат повертається також через UIstate.
- addFood() — додавання нової інформації про спожиту їжу (поживні речовини) до бази даних.
- getNutrients() — отримання масиву значень поживних речовин (Float) для заданих прийомів їжі у вигляді LiveData, що дозволяє автоматично оновлювати UI при зміні даних.
- getcalories() — подібно до попереднього методу, але повертає кількість калорій для кожного прийому їжі.
- changeHeight() — оновлення зросту користувача, при цьому контекст Android передається для можливого використання ресурсів або відображення повідомлень.
- getheight() — отримання поточного значення зросту користувача, результат повертається через замикання з обгорткою UIstate.

Використання обгортки UIstate дозволяє зручно обробляти різні стани запитів до бази даних (наприклад, завантаження, успіх, помилка) в єдиному

стилі, що відповідає сучасному підходу до асинхронної обробки даних в Android.

Клас RepositoryImp є реалізацією інтерфейсу Repository і виступає в ролі посередника між Firebase Firestore і логікою програми. Цей клас відповідає за збереження, оновлення та отримання даних, що стосуються харчування користувача, його ваги та зросту. Для роботи він використовує Firebase Authentication для отримання ідентифікатора поточного користувача, а також Firestore як джерело даних. Більшість операцій є асинхронними, і результати повертаються через LiveData або через функції зворотного виклику (callback), обгорнуті у UIstate, що дозволяє зручно обробляти помилки й стани завантаження.

```
@RequiresApi(Build.VERSION_CODES.O)
class RepositoryImp(
    private val database: CollectionReference
) : Repository {
```

Метод changeHeight() оновлює значення зросту користувача в документі Firestore, який відповідає UID користувача. При успішному збереженні показується повідомлення про успіх, у протилежному випадку — повідомлення про помилку. Метод getheight() підписується на зміну значення зросту в базі даних у реальному часі, і якщо воно існує, повертає його у вигляді рядка через UIstate.

```
     override fun changeHeight(height: String,context: 
Context){
        val ref=database.document(
            FirebaseAuth.getInstance().currentUser!!.uid)
        ref.update("height",height).addOnSuccessListener { 
...
            return@addOnFailureListener
        }
    }
```

Функція getWeight() витягує дані про вагу користувача з підколекції Weight track і трансформує їх у формат Entry, який зручно використовувати для побудови графіків. Кожен запис містить дату і значення ваги. Для перетворення дати у часову мітку використовується SimpleDateFormat. У разі успіху повертається список об'єктів Entry; у разі невдачі — повідомлення про помилку.

```
     override fun getWeight(result: (UIstate<ArrayList<Entry>>)
-> Unit) {
             database.document(

FirebaseAuth.getInstance().currentUser!!.uid).collection("Weight
track").get().addOnSuccessListener {
                 val entries = ArrayList<Entry>()
                 for (document in it) {
                     ...
                     val milliseconds = _date.time
                     if (weight != null) {
                         entries.add(
                             Entry(
                                 milliseconds.toFloat(),
                                 weight.toFloat()
     ...
```

Метод changedWeight() зберігає нове значення ваги в колекції Weight track, використовуючи поточну дату як ідентифікатор документа. Після запису викликається відповідна функція зворотного виклику із повідомленням про успішне або неуспішне завершення операції.

```
     override fun changedWeight(weight: weight, result:
(UIstate<String>) -> Unit){
             database.document(

FirebaseAuth.getInstance().currentUser!!.uid).collection("Weight
track").document(LocalDate.now().toString())
                 .set(weight)
                 .addOnSuccessListener {
                     result.invoke(
                         UIstate.Success("weight changed
successfully")
                     )
```

```
                }
                .addOnFailureListener {
        …
```

Метод addFood() додає інформацію про страву до колекції Meals, в яку вбудовано ще одну колекцію, що відповідає часу прийому їжі (наприклад, сніданок чи обід). У межах цієї колекції створюється документ з назвою страви. Після успішного додавання викликається відповідний callback, який повертає об'єкт Nutrient або помилку.

```
     override fun addFood(nutrient: Nutrient, result:
(UIstate<Nutrient>) -> Unit) {
                 database.document(

FirebaseAuth.getInstance().currentUser!!.uid).collection("Meals"
).document(LocalDate.now().toString()).collection(nutrient.time!
!).document(nutrient.foodName!!).set(nutrient).addOnSuccessListe
ner {
     result.invoke(UIstate.Success(nutrient))
                     }.addOnFailureListener {
     ...
```

Метод getcalories() працює асинхронно через корутини й дозволяє отримати загальну кількість калорій для кожного прийому їжі, переліченого в списку mealTimes. Для кожного з них запускається окремий запит, а отримані значення додаються до загального списку. Коли всі дані отримано, вони публікуються в LiveData, яку потім можна спостерігати в UI.

```
     override fun getcalories(mealTimes: List<String>):
LiveData<ArrayList<Float>> {
         val nutrientsLiveData =
MutableLiveData<ArrayList<Float>>()
         val nutrients = ArrayList<Float>()
         CoroutineScope(Dispatchers.IO).launch {
             val deferredList = mealTimes.map { mealTime ->
                 async {
                     val query = database.document(

FirebaseAuth.getInstance().currentUser!!.uid).collection("Meals"
).document(LocalDate.now().toString()).collection(mealTime)
     …
```

Метод getNutrients() подібний до попереднього, але він повертає не тільки калорії, а й вміст вуглеводів, цукру, білків і жирів. Для цього створюється масив з п’яти чисел, який поступово заповнюється на основі даних, витягнутих з бази. Після завершення обробки результат передається у LiveData.

```
     override fun getNutrients(mealTimes: List<String>):
LiveData<ArrayList<Float>> {
             val nutrientsLiveData =
MutableLiveData<ArrayList<Float>>()
             val nutrients = arrayListOf(0f, 0f, 0f, 0f , 0f)
             CoroutineScope(Dispatchers.IO).launch {
                 mealTimes.forEach { mealTime ->
                     val query = database.document(
FirebaseAuth.getInstance().currentUser!!.uid).collection("Meals"
).document(LocalDate.now().toString()).collection(mealTime)
```

2.3 Налаштування авторизації

Компонент Login_fragment реалізує інтерфейс авторизації користувача в Android-додатку за допомогою Firebase Authentication. Після створення представлення фрагмента у методі onCreateView, з XML-файлу activity_login_fragment завантажується розмітка, а також викликаються два методи: userlog() та forget(). Перший відповідає за обробку натискання кнопки входу, другий — за логіку відновлення пароля [16].

Метод userlog() прикріплює обробник події до кнопки входу. У ньому використовується захист від багаторазового кліку — одразу після натискання змінна btnEnable встановлюється в false, і повторно активується тільки через 1 секунду, використовуючи Handler().postDelayed(...). Коли кнопка стає неактивною, викликається метод createuserlog().

```
     fun userlog(){
        var btnEnable=true
        val logButton: Button =root!!.findViewById(R.id.loginbtn)
        logButton.setOnClickListener(View.OnClickListener {
            if(btnEnable)
            {
```

```
                btnEnable=false
                createuserlog()
            }
            Handler().postDelayed({
                btnEnable=true
            },1000)
        })
    }
```

У createuserlog() реалізовано логіку перевірки введених даних і спробу входу. Спочатку з розмітки отримуються поля введення email та пароля, текстове поле для відображення помилок, а також TextInputLayout, які містять ці поля — вони використовуються для застосування анімацій.

```
     val memail: EditText = root!!.findViewById(R.id.email)
             val mpassword: EditText
=root!!.findViewById(R.id.password)
             val
pass_error:TextView=root!!.findViewById(R.id.password_error)
             val rightAnimation: Animation? =
AnimationUtils.loadAnimation(activity, R.anim.rightt_left)
             val leftAnimation: Animation? =
AnimationUtils.loadAnimation(activity, R.anim.left_right)
```

Анімації завантажуються з ресурсів right_left і left_right і встановлюються для відповідних елементів інтерфейсу, щоб надати візуальний ефект під час помилок. Далі виконується перевірка: якщо поле email порожнє — відображається помилка; якщо поле пароля порожнє або коротше за 6 символів — виводиться відповідне повідомлення про помилку [17].

```
     if (memail.text.toString().isEmpty()) {
                memail.error = "Email is Required."
                return
            }
            if (mpassword.text.toString().isEmpty()) {
                pass_error.text="Password is Required *"
                return
            }
            if (mpassword.text.toString().length < 6) {
                pass_error.text="Password Must be greater than 6
Characters *"
```

```
            return
        }
```

Якщо ж валідацію пройдено успішно, виконується вхід до Firebase через метод signInWithEmailAndPassword(...), передаючи email і пароль користувача.

```
fAuth.signInWithEmailAndPassword(memail.text.toString(),mpa
ssword.text.toString()).addOnCompleteListener { task ->
```

Після спроби входу перевіряється результат виконання. Якщо вхід був успішним, тоді перевіряється, чи підтверджено електронну пошту користувача. Якщо підтвердження є, відображається повідомлення про успішний вхід, запускається головна активність додатку (Home_screen), і поточна активність закривається. Якщо ж пошта не підтверджена, користувачеві повідомляють про це через Toast-повідомлення. У разі виникнення помилки входу (наприклад, невірний пароль або неіснуючий обліковий запис), ця помилка виводиться також у Toast-повідомленні.

```
if (task.isSuccessful) {
     if(fAuth.currentUser!!.isEmailVerified){
          Toast.makeText(activity, "Logged in
          Successfully", Toast.LENGTH_SHORT)
.show()
startActivity(Intent(activity, Home_screen::class.java))
     activity!!.finish()
}
```

Метод forget() відповідає за логіку скидання пароля. Коли користувач натискає на напис “Forgot Password”, відкривається діалогове вікно, в якому потрібно ввести email. Після підтвердження (натискання кнопки “Yes”) виконується перевірка: якщо поле порожнє — виводиться помилка. Якщо email заповнено, тоді через sendPasswordResetEmail(...) надсилається лист з посиланням на зміну пароля. У разі успіху виводиться повідомлення, що посилання було надіслано, в іншому випадку — повідомлення про помилку [18].

```
forgotTextLink.setOnClickListener(View.OnClickListener { v ->
            val resetMail = EditText(v.context)
            val passwordResetDialog =
AlertDialog.Builder(v.context)
            passwordResetDialog.setTitle("Reset Password ?")
            passwordResetDialog.setMessage("Enter Your Email To
Received Reset Link.")
            passwordResetDialog.setView(resetMail)
            passwordResetDialog.setPositiveButton(
                "Yes"
            ) { dialog, which ->
```

Далі розглянемо фрагмент Android-додатку для реєстрації користувачів. Клас signup_fragment наслідується від Fragment і містить кілька важливих змінних. Змінна root зберігає посилання на кореневий view фрагменту, ref призначена для роботи з Firebase Realtime Database (хоча фактично не використовується), progressBarDialog для відображення індикатора завантаження, chn для зберігання обраної статі користувача, та db для роботи з Firebase Firestore.

```
class signup_fragment(): Fragment() {
    var root:View?=null
    private lateinit var ref:DatabaseReference
    private var progressBarDialog:ProgressDialog?=null
    private var chn:String?=null
    private var db =Firebase.firestore
    companion object{
        const val TAG="TAG"
    }
```

Метод onCreateView частиною життєвого циклу фрагменту і викликається при створенні UI. Він інфлейтить layout файл signup_fragment і викликає метод register() для ініціалізації компонентів інтерфейсу. Метод повертає створений view.

```
override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
```

```
            root = inflater.inflate(R.layout.signup_fragment, 
container, false)

            register()
            return root
        }
```

Функція createuser відповідає за створення нового користувача в системі Firebase Authentication. Спочатку вона отримує посилання на всі поля введення: повне ім'я, телефон, електронну пошту та пароль. Далі проводиться валідація введених даних - перевіряється, чи всі обов'язкові поля заповнені і чи пароль містить принаймні 6 символів [19].

```
     fun createuser(){
            var 
ffullname:EditText=root!!.findViewById(R.id.fullname)
     ...
            val fAuth:FirebaseAuth= FirebaseAuth.getInstance()
     ...
     if (password.text.toString().length<6)
         {
             password.error="password must be greater then 6 
                  characters"
             return
          }
```

Після успішної валідації відображається діалог прогресу і викликається метод Firebase для створення користувача з електронною поштою та паролем. У разі успішного створення облікового запису автоматично відправляється лист для верифікації електронної пошти.

```
     fAuth.createUserWithEmailAndPassword(memail, 
mpassword).addOnCompleteListener { task ->
          Log.d("MyApp", task.isSuccessful.toString())
          Log.e("MyApp", "Error: ${task.exception?.message}", 
task.exception)
          if (task.isSuccessful) {
```

Після створення облікового запису дані користувача зберігаються у Firebase Firestore. Створюється HashMap з інформацією про користувача:

повне ім'я, електронна пошта, пароль, телефон, дата народження та стать. Ці дані записуються в колекцію "user" під унікальним ідентифікатором користувача.

```
     val usermap= hashMapOf(
          "fullname" to ffullname.text.toString().trim(),
          "email" to
     ...
          dob.text.toString().trim(),
          "gender" to chn.toString().trim(),
     )
     db.collection("user").document(currenuser.toString()).set(u
sermap)
     .addOnSuccessListener {
          ffullname.text.clear()
          email.text.clear()
          password.text.clear()
          phone.text.clear()
     }
```

Функція register ініціалізує UI компоненти фрагменту. Вона налаштовує обробник кліків для поля дати народження, який відкриває DatePicker. Також створюється адаптер для випадаючого списку вибору статі, використовуючи масив з ресурсів додатку. Встановлюється обробник для кнопки реєстрації, який викликає функцію createuser.

```
     fun register(){
         val dob:TextView=root!!.findViewById(R.id.dob)
         dob.setOnClickListener {
             clickDataPicker()
         }
         val gender=resources.getStringArray(R.array.gender)
         val
arrayAdapter=ArrayAdapter(requireContext(),R.layout.dropdown,gen
der)
          val
chgen=root!!.findViewById<AutoCompleteTextView>(R.id.gender)
          chgen.setAdapter(arrayAdapter)
```

Функція clickDataPicker відповідає за відображення діалогу вибору дати. Вона створює DatePickerDialog з поточною датою як початковою. Після вибору дати вона форматується у вигляді "день/місяць/рік" і відображається в

відповідному полі. Також встановлюється обмеження, що не можна вибрати дату пізніше вчорашнього дня, що логічно для дати народження.

```
fun clickDataPicker() {
        val c = Calendar.getInstance()
...
        val dpd = DatePickerDialog(
            requireContext(),
            { view, year, monthOfYear, dayOfMonth ->
                val selectedDate = "$dayOfMonth/${monthOfYear +
1}/$year"
                val dob:TextView=root!!.findViewById(R.id.dob)
                dob.text=(selectedDate.toString())
                val sdf = SimpleDateFormat("dd/MM/yyyy",
Locale.ENGLISH)
```

MainAuthentication представляє головну активність для аутентифікації користувачів у нашому додатку.

У методі onCreate використовується ViewBinding для зв'язування з layout файлом активності. Створюється об'єкт ActivityMainAuthenticationBinding, який забезпечує типобезпечний доступ до всіх елементів інтерфейсу. Цей підхід є більш сучасним та безпечним порівняно зі старим методом findViewById.

```
class MainAuthentication : AppCompatActivity() {
    @SuppressLint("MissingInflatedId")
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding
=ActivityMainAuthenticationBinding.inflate(layoutInflater)
```

Активність завантажує дві анімації з ресурсів: left_right для верхнього елементу та bottom_animation для нижнього. Анімація topAnimation застосовується до заголовку titleText, що створює ефект руху зліва направо. Для елементу animationView2 викликається метод playAnimation(), що вказує на використання Lottie анімації. Нижня частина інтерфейсу logindetails отримує анімацію появи знизу.

```
    val topAnimation: Animation? =
AnimationUtils.loadAnimation(this, R.anim.left_right)
            val bottomanim: Animation? =
AnimationUtils.loadAnimation(this, R.anim.bottom_animation)
            binding.titleText.animation=topAnimation
            binding.animationView2.playAnimation()
            binding.logindetails.animation=bottomanim
```

Код створює дві вкладки: "Login" та "Signup" за допомогою TabLayout. Встановлюється гравітація GRAVITY_FILL, що означає, що таби займуть всю доступну ширину екрану рівномірно. Цей підхід забезпечує зручну навігацію між формами входу та реєстрації.

```
    val tabLayout: TabLayout =findViewById(R.id.tab_layout)
    tabLayout.addTab(tabLayout.newTab().setText("Login"))
    tabLayout.addTab(tabLayout.newTab().setText("signup"))
    tabLayout.tabGravity= TabLayout.GRAVITY_FILL
```

ViewPager використовується для забезпечення можливості свайпу між різними фрагментами. Викликається метод setupViewPager, який налаштовує адаптер для ViewPager. Після цього TabLayout синхронізується з ViewPager за допомогою методу setupWithViewPager, що дозволяє переключатися між табами як дотиком, так і свайпом.

```
    val viewpager: ViewPager =findViewById(R.id.view_pager)
    setupViewPager(viewpager)
            tabLayout.setupWithViewPager(viewpager)
```

Метод setupViewPager створює та налаштовує ViewPagerAdapter, який управляє фрагментами. До адаптера додаються два фрагменти: Login_fragment з назвою "Login" та signup_fragment з назвою "Signup". Адаптер використовує supportFragmentManager для управління життєвим циклом фрагментів.

```
    private fun setupViewPager(viewpager: ViewPager) {
            var adapter: ViewPagerAdapter =
ViewPagerAdapter(supportFragmentManager)
            adapter.addFragment(Login_fragment(), "Login")
            adapter.addFragment(signup_fragment(), "Signup")
```

```
        viewpager.setAdapter(adapter)
    }
```

2.4 Створення інтерфейсу користувача

Клас Home_screen є головною активністю застосунку після входу користувача, і його основне завдання — організувати інтерфейс користувача з нижньою навігацією, обробкою дозволів, запуском фонових сервісів та завантаженням відповідних фрагментів залежно від стану користувача [20].

На початку класу оголошується масив permissions, який містить усі необхідні дозволи, які додаток повинен отримати. До них входять доступ до геолокації, сенсорів тіла, розпізнавання фізичної активності та дозвіл на сповіщення. Ці дозволи є критично важливими для роботи з функціоналом здоров'я, трекінгу активності та надсилання push-повідомлень.

```
    private var permissions = arrayOf(
        Manifest.permission.ACCESS_COARSE_LOCATION,
        Manifest.permission.ACCESS_FINE_LOCATION,
        Manifest.permission.BODY_SENSORS,
        Manifest.permission.ACTIVITY_RECOGNITION,
        Manifest.permission.POST_NOTIFICATIONS
    )
```

Змінна REQUEST_CODE_PERMISSIONS використовується як ідентифікатор запиту дозволів. Далі оголошено змінну buttonnav, яка відповідає за нижнє меню навігації. Змінна userDitails містить посилання на документ поточного користувача в Firestore, де зберігаються персональні дані.

```
    private val REQUEST_CODE_PERMISSIONS = 11;
    private var userDitails: DocumentReference =
Firebase.firestore.collection("user").document(
        FirebaseAuth.getInstance().currentUser!!.uid.toString()
    )
```

```
private lateinit var binding: ActivityHomeScreenBinding
```

У методі onCreate, який викликається при запуску активності, виконується прив’язка до layout-файлу через ActivityHomeScreenBinding. Встановлюється світла тема статусбару. Викликається метод checkpermission, щоб перевірити і запросити дозволи, якщо їх ще не надано.

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding =
ActivityHomeScreenBinding.inflate(layoutInflater)
        setContentView(binding.root)
        window.decorView.systemUiVisibility =
View.SYSTEM_UI_FLAG_LIGHT_STATUS_BAR
        checkpermission()
```

Нижня навігація спочатку робиться видимою. Створюються об’єкти фрагментів: головний фрагмент (Home), плани, профіль, додавання ваги та статистика. Далі виконується запит до Firestore, щоб перевірити, чи є у користувача хоча б один запис ваги [21]. Якщо таких записів немає, тобто користувач ще не вводив свою вагу, навігаційна панель і додаткові елементи приховуються, а на екран виводиться фрагмент додавання ваги (AddWeight). Якщо записи присутні, активується головний фрагмент і викликається метод Getlatestweight, який зберігає останнє значення ваги в локальні налаштування.

```
binding.buttonnav.visibility = View.VISIBLE
val Home_fragment = Home_fragment()
val plansFragment = Plans_fragment()
val ProfileFragment = profile_fragment()
val Weight = AddWeight()
val Stats=Stats()
```

Підключається обробка події відкриття клавіатури через KeyboardVisibilityEvent. Коли клавіатура з’являється, навігація приховується, щоб не заважати введенню тексту. Коли клавіатура закривається, навігація знову показується. Далі встановлюється обробник натискання на нижню

навігацію — кожен пункт меню відповідає певному фрагменту: головна сторінка, плани, профіль, статистика. При натисканні викликається функція makeCurrentFrag, яка змінює відображуваний фрагмент із використанням анімації.

```
KeyboardVisibilityEvent.setEventListener(
     this
) { isOpen ->
    if (isOpen) {
        binding.buttonnav.visibility = View.INVISIBLE
            } else {
                binding.buttonnav.visibility = View.VISIBLE
            }
    }
```

Після цього йде ще один запит до Firestore: отримуються збережені дані про здоров'я з документа health_report. Якщо такі дані існують, то вони зберігаються локально в SharedPreferences.

```
userDitails.collection("fitness").document("health_report")
            .get().addOnSuccessListener { value ->
                if (value != null && value.exists()) {
                    val stat =
value.data?.get("data").toString()

Constant.savedata(this,"health","stats",stat.toString())
                }
            }.addOnFailureListener {
            }
```

Також викликається метод Reset_AT_12_AM.setNotification, який встановлює повторюване сповіщення на північ кожного дня — воно скидає кількість кроків через відповідний Broadcast.

```
Reset_AT_12_AM.setNotification(this)
```

У методі onStart перевіряється, чи запущений сервіс MyService. Якщо ні — він запускається. Це необхідно, щоб збирати дані у фоновому режимі, наприклад, кроки користувача.

```
    override fun onStart() {
            super.onStart()
            if
(!Constant.isServiceRunning(MyService::class.java,
applicationContext)) {
                val intent = Intent(this,
MyService::class.java).apply {
                    action = MyService.ACTION_START
                }
                startService(intent)
            }
        }
```

Метод makeCurrentFrag використовується для підміни поточного фрагмента іншим із застосуванням анімацій. Це забезпечує плавний перехід між фрагментами в інтерфейсі.

```
        private fun makeCurrentFrag(Fragment: Fragment) {
            supportFragmentManager.beginTransaction().apply {
                setCustomAnimations(R.anim.left_center,
R.anim.right_center);
                replace(R.id.frg, Fragment)
                commit()
            }
        }
```

Метод checkpermission перевіряє, чи всі потрібні дозволи вже надано. Якщо ні — відбувається запит дозволів через стандартний механізм Android.

```
private fun checkpermission() {
        val allPermissionGranted = permissions.all {
            ContextCompat.checkSelfPermission(
                this, it
            ) == PackageManager.PERMISSION_GRANTED
        }
        if (!allPermissionGranted) {
            requestPermissions(permissions,
REQUEST_CODE_PERMISSIONS)
        }
    }
```

Метод onRequestPermissionsResult обробляє результат надання дозволів. Якщо користувач погодився на всі — можна безпечно виконувати функціональність, яка потребує цих дозволів. Якщо хоча б один дозвіл не надано — додаток може працювати з обмеженим функціоналом [22].

Метод Getlatestweight виконує ще один запит до Firestore, де зберігаються записи про відстеження ваги. Він отримує останній запис і зберігає його у локальні налаштування, щоб ця інформація була доступна і поза мережею.

```
         fun Getlatestweight() {
             userDitails.collection("Weight
track").get().addOnSuccessListener { result ->
                 val size = result.documents.size
                 val curr_weight = result.documents[size -
1].get("weight").toString().toFloat()
                 Constant.savedata(this, "weight", "curr_w",
curr_weight.toString())
             }
         }
```

Наприкінці оголошується об’єкт Reset_AT_12_AM. Він містить функцію setNotification, яка встановлює таймер за допомогою AlarmManager. Цей таймер спрацьовує щодня опівночі, щоб відправити broadcast через клас Step_reset_BoardCast. Це використовується, наприклад, для скидання денного лічильника кроків або інших щоденних метрик.

```
object Reset_AT_12_AM {
    @RequiresApi(VERSION_CODES.M)
    @SuppressLint("ServiceCast", "UnspecifiedImmutableFlag")
    fun setNotification(context: Context) {
        ...
        val calendar = Calendar.getInstance()
        calendar.set(Calendar.HOUR_OF_DAY, 0)
```

Клас AddWeight є фрагментом у застосунку, що відповідає за початкове введення ваги користувачем, а за потреби — ще й зросту. Його основне

завдання — зібрати ці дані, зберегти їх у Firestore та локально, а потім передати користувача до головного екрану застосунку [23].

Клас наслідує Fragment і працює з view binding'ом через об'єкт FragmentAddWeightBinding, який дає зручний доступ до елементів макету XML-файлу фрагмента. У полі userDitails зберігається посилання на документ поточного користувача у Firestore. Це потрібно, щоб записати дані про вагу та зчитати/оновити інформацію про зріст.

```
class AddWeight : Fragment() {
    private lateinit var binding: FragmentAddWeightBinding
    private var userDitails: DocumentReference =
Firebase.firestore.collection("user").document(

FirebaseAuth.getInstance().currentUser!!.uid.toString()
    )
```

У методі onCreateView, що викликається під час створення фрагмента, виконується ініціалізація binding'а, а також налаштовуються числові селектори (NumberPicker): для ваги встановлюється діапазон від 1 до 200, для зросту — окремо в футах (від 3 до 8) і дюймах (від 1 до 12), з можливістю циклічного прокручування.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    binding = FragmentAddWeightBinding.inflate(inflater,
container, false)
    binding.apply {
        weight.minValue = 1
        weight.maxValue = 200
        ft.minValue = 3
        ft.maxValue = 8
```

Обробка натискання кнопки add є центральною логікою цього фрагмента. Спочатку перевіряється наявність інтернет-з'єднання за допомогою функції isInternetOn. Якщо з'єднання є, викликається функція addweight, яка зберігає введену вагу в базу даних Firestore, а також у локальне

сховище через метод Constant.savedata. Після збереження ваги приховується елемент weightv, який містить picker для вибору ваги.

```
add.setOnClickListener {
    if (isInternetOn(requireContext())) {
         addweight()
         weightv.visibility = View.GONE
```

Далі створюється слухач змін (addSnapshotListener) для документа користувача. Метою є перевірка, чи поле height (зріст) уже збережене. Якщо такого поля ще немає, стає видимим блок height, який дозволяє користувачу ввести зріст у футах і дюймах. Значення дюймів автоматично конвертуються у частину фута шляхом ділення на 12, і підсумовуються.

```
userDitails.addSnapshotListener { value, error ->
     if (value!!.exists() && !value.contains("height")) {
          height.visibility = View.VISIBLE
          val height = ft.value + (inch.value / 12)
```

Коли користувач натискає кнопку addh, знову перевіряється наявність інтернету. Якщо з’єднання активне, значення зросту зберігається у полі height документа користувача у Firestore. Після цього запускається нова активність Home_screen, а поточна активність завершується (finish), щоб не повертатися назад.

```
addh.setOnClickListener {
    if (isInternetOn(requireContext())) {
        userDitails.update("height", height.toString())
        startActivity(
            Intent(
                requireActivity(),
                Home_screen::class.java
            )
        )
        requireActivity().finish()
```

Якщо ж інтернет-з’єднання відсутнє на будь-якому з етапів, показується повідомлення через Toast, яке інформує користувача про необхідність увімкнути підключення до Інтернету.

```
Toast.makeText(
    requireContext(),
    "Please Turn on your internet connection",
    Toast.LENGTH_SHORT
).show()
```

Метод addweight, який викликається окремо, зчитує значення ваги з NumberPicker, отримує поточну дату у форматі ISO (через LocalDate.now()), зберігає вагу локально і створює словник map, що містить вагу та дату [24]. Потім ці дані записуються у підколекцію Weight track користувача, у документ, який має назву поточної дати. Таким чином, відстежується історія змін ваги за датами.

```
fun addweight() {
    val weight = binding.weight.value.toString()
    val curr_date = LocalDate.now().toString()
    Constant.savedata(requireContext(), "weight", "curr_w",
weight.toString())
    val map = hashMapOf("weight" to weight, "date" to
curr_date)
    userDitails.collection("Weight
track").document(curr_date).set(map)
}
```

Клас Home_fragment в нашому додатку відповідає за основну домашню сторінку користувача. Цей фрагмент має функціональність, пов’язану з контролем здоров’я користувача: кількість кроків, споживання води, привітання залежно від часу доби, перехід до обліку ваги, їжі, нагадувань тощо.

У методі onCreateView відбувається ініціалізація основних елементів інтерфейсу, прив’язка до layout через view binding, запуск циклічного Runnable для оновлення UI щосекунди та налаштування слухачів на кнопки. Наприклад, при натисканні на кнопку “додати воду” викликається метод addwater(), що

оновлює інформацію у Firebase і UI. Якщо кількість склянок досягає 10, запускається анімація як заохочення.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    binding = FragmentHomeFragmentBinding.inflate(inflater,
container, false)
    handler.post(updatetimeRunnable)
        dialog = Dialog(requireActivity())
```

Слухач змін Firebase підписується на колекцію “user”, зокрема на документ користувача за UID, щоб отримати повне ім’я для привітання. Це дозволяє динамічно показувати ім’я на головному екрані.

```
userDitails.addSnapshotListener { value, error ->
    if (error != null) {
        return@addSnapshotListener
    }
    if (value != null && value.exists()) {
        binding.name.text =
value.data!!["fullname"].toString()
    }
}
```

Окремий метод greeting_class() формує привітання залежно від поточного часу, а також змінює іконку (сонце чи місяць) відповідно до частини дня. Метод existwater() перевіряє, чи існує документ щодо вживання води на сьогодні. Якщо ні — створює його з початковим значенням “0 склянок”.

```
private fun greeting_class() {
    val time: ImageView = binding.weather
    val greeting: TextView = binding.greeting
    val calendar = Calendar.getInstance()
    val hour = calendar.get(Calendar.HOUR_OF_DAY)
```

Метод stepCounter() відповідає за підрахунок кроків. Він читає збережені в SharedPreferences загальні та попередні кроки, обчислює пройдений шлях за день, оновлює інтерфейс (лічильник кроків, калорії) та прогрес-бар. Також передбачена можливість скидання лічильника довгим натисканням на прогрес-бар — при цьому зберігаються поточні дані як попередні, і прогрес починається з нуля.

```
fun stepCounter() {
    val t_step = Constant.loadData(requireContext(),
"step_count", "total_step", "0").toString()
    val pre_step =
        Constant.loadData(requireContext(), "step_count",
"previous_step", "0").toString()
    val curr_step = abs(t_step.toInt() -
pre_step.toInt()).toString()
    binding.walk.text = curr_step.toString()
```

Метод set_target() завантажує збережене значення цільової кількості кроків з SharedPreferences і оновлює ціль на екрані та в прогрес-барі. Це дозволяє користувачеві мати персоналізовану мету.

```
fun set_target() {
    val sharedPreferences: SharedPreferences =
        requireActivity().getSharedPreferences("myPrefs",
Context.MODE_PRIVATE)
    val target = sharedPreferences.getString("target",
"1000")
    val cpbar = binding.circularProgressBar
    cpbar.progressMax = target!!.toFloat()
```

Використовуються сучасні підходи, як-от LiveData, ViewModel, Firebase Firestore для зберігання та отримання даних у реальному часі, а також Lottie для анімацій.

Клас Stats є частиною інтерфейсу користувача, який призначений для роботи з медичними статистичними даними користувача. Він реалізує фрагмент, що дозволяє переглядати та редагувати чотири ключові показники здоров’я: артеріальний тиск, рівень цукру натще, рівень цукру після їжі та

рівень холестерину. Для кожного з цих параметрів виводиться останні п’ять значень, які зберігаються у вигляді рядка, розділеного символами : та ?.

Після створення фрагмента, в методі onViewCreated, ініціалізується основна логіка взаємодії з користувачем.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {

_binding=FragmentStatsBinding.inflate(inflater,container,fa
lse)
    return binding.root
}
```

Метод ui() викликається для налаштування елементів інтерфейсу, а також для підписки на оновлення з Firestore у документі health_report, що зберігається в колекції fitness користувача. Якщо документ оновлюється, нові дані зберігаються в локальній пам’яті за допомогою методу Constant.savedata.

```
private fun ui(){
userDitails.collection("fitness").document("health_report")
.addSnapshotListener { value, error ->
            if (value!!.exists())
            {
Constant.savedata(requireContext(),"health","stats",value.g
et("data").toString())
            }
        }
```

Фактичні значення статистики завантажуються з локального сховища у методі getDataFromSharedPreference, а потім розділяються на окремі блоки для кожного параметра. Кожен блок передається у відповідний метод відображення: setBloodPressure, setSugarFasting, setSugarPP та setCholesterol. У цих методах значення перетворюються у список цілих чисел, виводяться мінімальні та максимальні значення, а також графічно відображаються через спеціалізовані компоненти SparkLineLayout.

```
setBloodPressure(splitParts[0], splitParts)
setSugarFasting(splitParts[1], splitParts)
setSugarPP(splitParts[2], splitParts)
setCholesterol(splitParts[3], splitParts)
```

Для кожного параметра передбачено кнопку, яка дозволяє користувачу додати нове значення. При натисканні відкривається діалогове вікно з полем вводу, в яке користувач може ввести нове числове значення. Після підтвердження нове значення додається до черги, найстаріше значення видаляється, і оновлений список перетворюється назад у строку, яку потім зберігають як у Firestore, так і локально через SharedPreferences. Графік після цього перерисовується, щоб відобразити зміни [25].

Метод mergedData відповідає за формування нового рядка зі статистикою після оновлення одного з параметрів. Він вставляє новий список значень на відповідне місце у загальній структурі рядка статистики, зберігаючи решту даних без змін.

```
private fun mergedData(i: Int, List: ArrayList<Int>,
splitParts: List<String>): String? {
    val tempString: StringBuilder = StringBuilder()
    for (j in 0..4) {
        tempString.append(List[j]).append(":")
    }
    val finalStats: StringBuilder = StringBuilder()
```

Клас Reminder є активністю, яка відповідає за інтерфейс і функціональність нагадування про споживання води. Він використовує FragmentReminderBinding для прив’язки UI-компонентів і керує збереженням та зчитуванням налаштувань користувача через SharedPreferences (реалізовано в класі Constant). Основна мета цієї активності — дозволити користувачеві встановити інтервал нагадування, час початку та завершення сповіщень, а також активувати або деактивувати функцію нагадувань.

```
class Reminder : AppCompatActivity() {
    private lateinit var binding: FragmentReminderBinding
    private  var startTime:Long=32400000
```

```
    private  var endTime:Long=79200000
    private  var start="9:00 AM"
    private  var end="10:00 PM"
```

У методі onCreate ініціалізується прив’язка до макету, створюється канал сповіщень (для Android 8+), а також зчитуються збережені значення часу початку і завершення в мілісекундах. Встановлюється мінімальне і максимальне значення для NumberPicker, який задає інтервал у годинах. Після цього викликається метод UI, що налаштовує користувацький інтерфейс.

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

Метод createNotification створює канал сповіщень із заданим ID, ім’ям і рівнем важливості. Це потрібно для того, щоб надсилати сповіщення про споживання води, і працює лише на Android 8.0 і вище.

```
private fun createNotification() {
    val name = "channel"
    val message = "message"
    val important = NotificationManager.IMPORTANCE_DEFAULT
    val channel = NotificationChannel(channel_ID, name,
important)
```

Метод UI відповідає за налаштування всіх візуальних елементів та взаємодію з користувачем. Встановлюється світла тема статус-бару, обробляється натискання кнопки “назад”, зчитуються і встановлюються збережені значення інтервалу, часу початку та завершення. Перевіряється стан перемикача (switchMaterial) і відповідно змінюється його колір: якщо активований — помаранчевий, якщо ні — сірий.

```
private fun UI() {
    window.decorView.systemUiVisibility =
View.SYSTEM_UI_FLAG_LIGHT_STATUS_BAR
```

```
    binding.backward.setOnClickListener {
        finish()
    }
    val Interval = Constant.loadData(this, "reminder",
"interval", "1")
```

Обробник подій перемикача зберігає новий стан перемикача у змінну check та оновлює його зовнішній вигляд. Кнопки вибору часу для початку та завершення викликають діалогове вікно вибору часу (реалізовано у Constant.showTimePicker). Вибраний час зберігається у змінні startTime або endTime у мілісекундах, а також у форматованому вигляді (наприклад, “09:00 AM”) для подальшого виводу у відповідні поля вводу.

```
binding.interval.value = Interval!!.toInt()
binding.start.setText(Constant.loadData(this,"reminder","st
art","09:00AM"))
binding.end.setText(Constant.loadData(this,"reminder","end"
,"10:00 PM"))
var check = Constant.loadData(this, "reminder", "check",
"0")
```

Кнопка збереження (binding.save) реагує на натискання, перевіряючи, чи активовано нагадування (check == "1"). Якщо так, зберігає всі параметри (інтервал, час початку і завершення, статус) у SharedPreferences та відображає повідомлення про успішне збереження. Потім створюється JobInfo із використанням JobScheduler і класу WaterService, який встановлює мінімальний інтервал запуску служби згідно з вибраним інтервалом. Ця служба відповідальна за генерацію сповіщень про необхідність випити воду.

```
binding.save.setOnClickListener{
    if(check=="1")
    {
        Constant.savedata(this, "reminder", "check", "1")
        Constant.savedata(applicationContext, "reminder",
"start", start)
        Constant.savedata(applicationContext, "reminder",
"end",end)
```

Якщо перемикач деактивований (check == "0"), зберігається відповідне значення в налаштуваннях, відображається повідомлення про видалення нагадування, і JobScheduler скасовує задачу з ідентифікатором 2.

```
else{
    Toast.makeText(this, "Water Reminder removed
successfully", Toast.LENGTH_SHORT).show()
    Constant.savedata(this, "reminder", "check", "0")
    val
jobScheduler=this.getSystemService(Context.JOB_SCHEDULER_SE
RVICE) as JobScheduler
    jobScheduler.cancel(2)
}
```

2.5 Тестування застосунку

При відкритті застосунку нас зустрічає форма для входу в акаунт та створення акаунту на двох різних вкладках. Створимо новий акаунт (рис. 2.1).

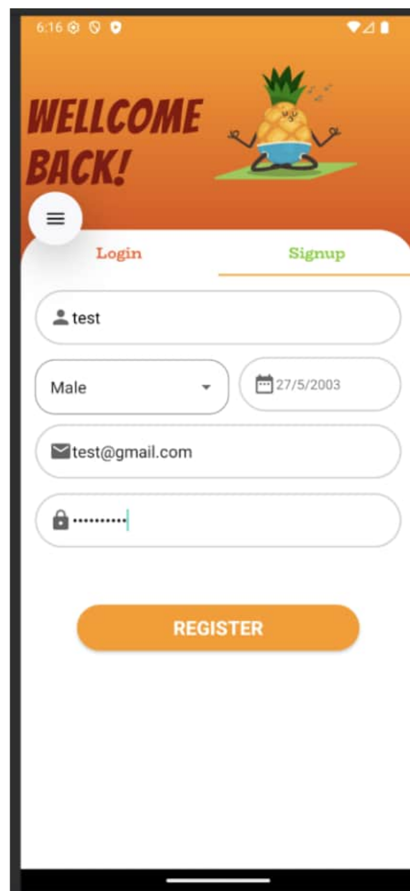

Рисунок 2.1 – Створення акаунту

Далі нам прийде лист на пошту для активації акаунту. Після підтвердження перейдемо на вкладку з входом в акаунт (рис. 2.2).

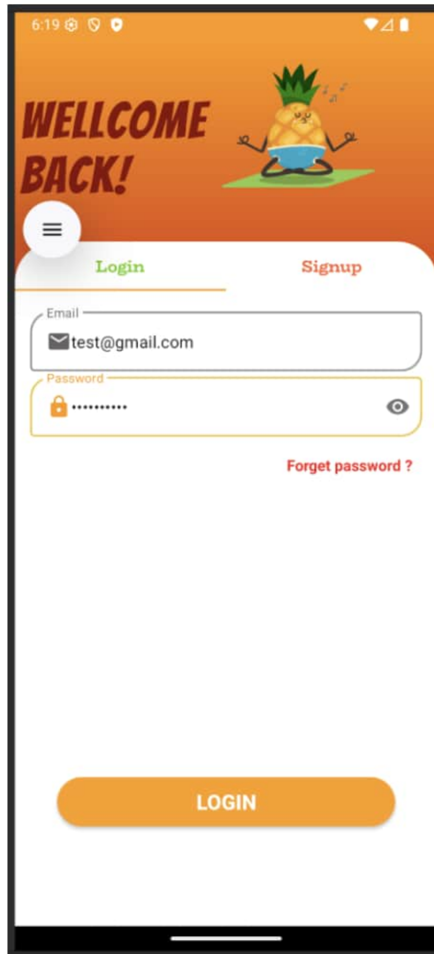

Рисунок 2.2 – Вхід в акаунт

Після чого ми перейдемо на стартовий екран, де треба вибрати вагу користувача (рис. 2.3).

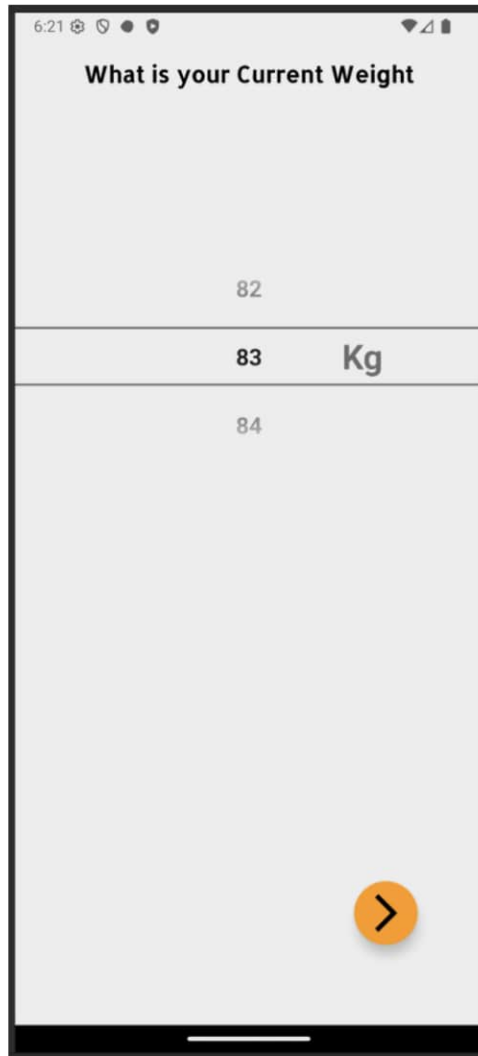

Рисунок 2.3 – Вибір ваги

Наступним кроком буде вибрати зріст (рис. 2.4).

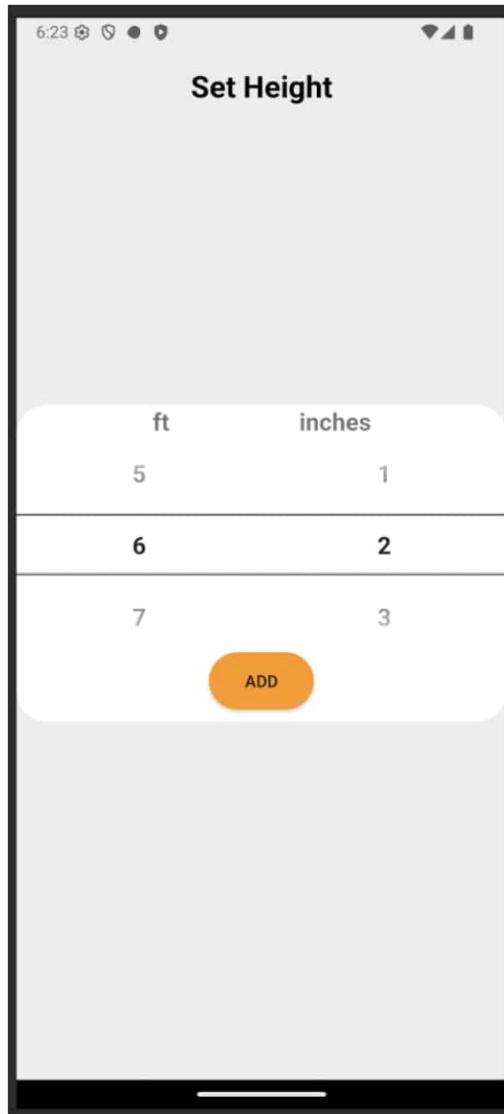

Рисунок 2.4 – Вибір зросту

Після чого нас перекине на головну сторінку застосунку де відображається кількість випитої води, кількість кроків, можливість нагадування про прийом їжі, кількість витрачених калорій за день та кількість годин які користувач спав (рис. 2.5).

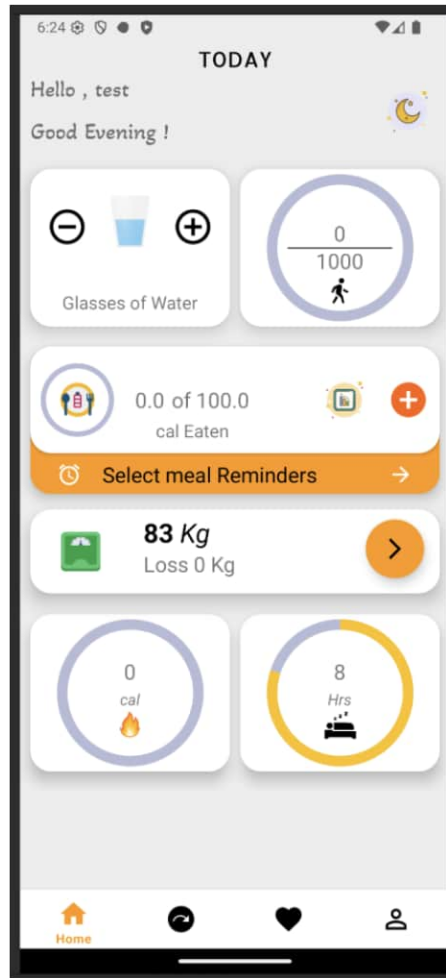

Рисунок 2.5 – Головна сторінка застосунку

Перейдемо на вкладку з нагадуваннями, де як ми можемо побачити можна додати нагадування на прийом їжі, пиття води, приймання ліків та миття рук.

Рисунок 2.6 – Нагадування

Наступною є вкладка зі статистикою, де можна переглянути різні хакартеристики здоров’я, такі як тиск, рівень цукру та коластеролу. Дані потрібно додавати вручну, після виміру цих показників відповідними приалдами.

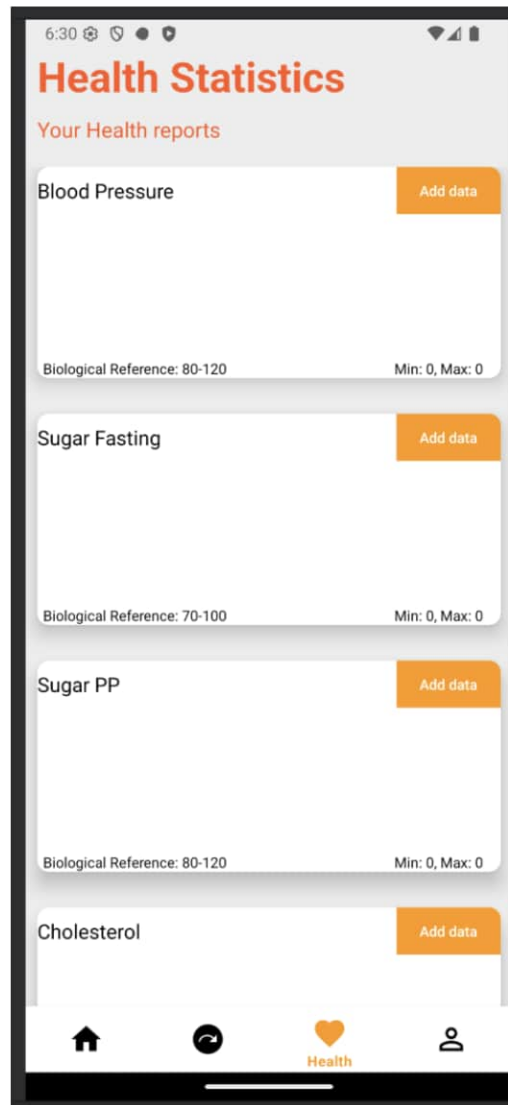

Рисунок 2.7 – Сторінка з показниками здоров’я

Також можна перейти на сторінку профіля користувача де можна переглянути основну інформацію користувача, таку як вік, стать, ріст, вагу та показник співвідношення ваги до зросту, і відповідне повідомлення в якій формі знаходиться користувач (рис. 2.8). Також є можливість вийти з акаунту.

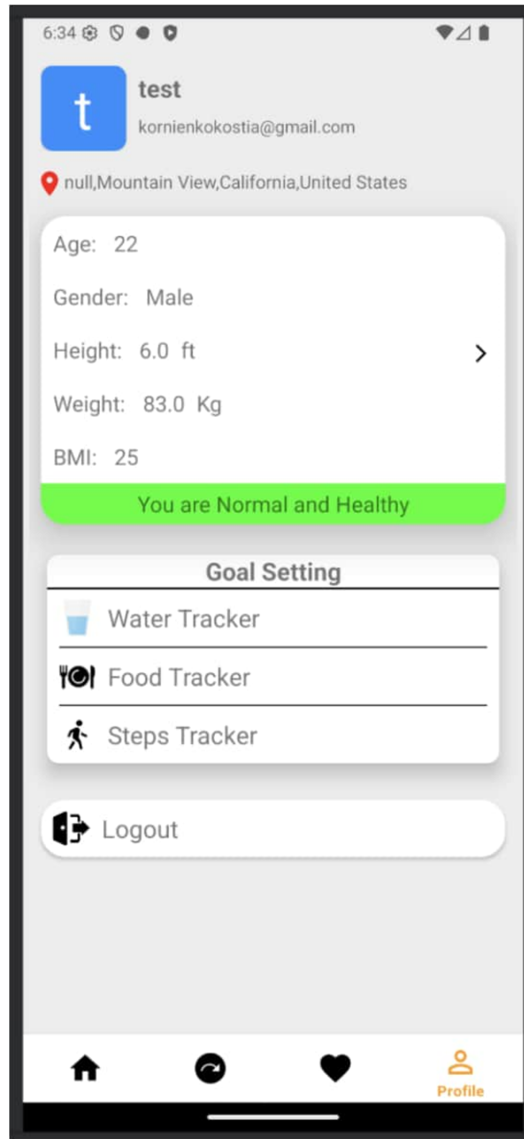

Рисунок 2.8 – Сторінка профілю

ВИСНОВКИ

У процесі виконання дипломної роботи на тему «Розробка мобільного застосунку для моніторингу здоров’я користувачів» було здійснено повний цикл проєктування та реалізації мобільного програмного забезпечення, призначеного для допомоги користувачам у відстеженні власного фізичного стану, дотриманні здорового способу життя та формуванні корисних звичок.

На першому етапі було проведено аналіз предметної області, зокрема здійснено огляд існуючих мобільних застосунків для моніторингу здоров’я. Було виділено найпоширеніші функції таких сервісів, серед яких: облік кількості випитої води, відстеження сну, нагадування про прийом ліків, фіксація фізичної активності, вимірювання пульсу та тиску, ведення особистого щоденника самопочуття тощо. Проведене дослідження дало змогу окреслити сильні та слабкі сторони аналогічних рішень, а також виявити вимоги кінцевих користувачів до подібних застосунків, зокрема: зручність інтерфейсу, простота реєстрації, висока швидкодія, безпека персональних даних, підтримка офлайн-режиму.

З урахуванням сучасних тенденцій розробки мобільного ПЗ, було обґрунтовано вибір мови програмування Kotlin, яка є офіційно підтримуваною Google для Android-розробки. Обрано платформу Android через її широке поширення серед користувачів та відкриту екосистему. Постановка завдання полягала у створенні зручного, функціонального та адаптивного застосунку, який би дозволяв контролювати здоров’я користувача у зручний для нього спосіб.

У другому розділі роботи було розроблено архітектуру застосунку, яка базується на принципах модульності, розширюваності та підтримки. Реалізовано структурування коду за архітектурним патерном MVVM (Model-View-ViewModel), що забезпечило чіткий поділ відповідальності між компонентами та покращило тестованість проєкту.

У процесі розробки було підключено локальну базу даних (SQLite), яка дозволяє зберігати дані користувача — такі як журнал споживання води, час сну, стан здоров’я тощо. Для забезпечення входу в систему реалізовано авторизацію, яка гарантує захист персональної інформації. Окрему увагу приділено створенню інтерфейсу користувача — він розроблений із дотриманням принципів UX/UI-дизайну: простота, доступність, мінімалізм, логічна структура елементів, підтримка різних розмірів екранів.

У рамках завершального етапу було здійснено тестування застосунку. Перевірено коректність функціоналу, стабільність роботи, обробку виняткових ситуацій та зручність користування. За результатами тестування застосунок продемонстрував високу стабільність та готовність до практичного використання.

Таким чином, у межах даної дипломної роботи було досягнуто поставленої мети: спроєктовано та реалізовано мобільний застосунок на платформі Android з використанням мови Kotlin, який забезпечує базовий функціонал моніторингу стану здоров’я користувача. Розроблений застосунок має потенціал для подальшого розширення, зокрема: підключення зовнішніх пристроїв (фітнес-браслетів), інтеграція з сервісами Google Fit або Apple Health, реалізація хмарної синхронізації та персоналізованої аналітики на базі штучного інтелекту.

Отже, розроблений застосунок може стати надійним інструментом для щоденного контролю здоров’я, сприяти формуванню здорових звичок і підвищенню якості життя користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Apple Health. URL: https://www.apple.com/health/ (дата звернення: 20.05.2025).
2. Google Fit. URL: https://www.google.com/fit/ (дата звернення: 20.05.2025).
3. Samsung Health. URL: https://www.samsung.com/ua/apps/samsung-health/ (дата звернення: 20.05.2025).
4. Are health apps actually bad for your health?. URL: https://www.bbc.com/bbcthree/article/9fe47476-ad1f-414c-a925-cf078a2145a8 (дата звернення: 20.05.2025).
5. 5 Symptom Diary & Health Journal Apps. URL: https://www.teamscopeapp.com/blog/5-diary-apps-for-tracking-symptoms (дата звернення: 20.05.2025).
6. Fuselab Creative. Healthcare Data Visualization: Insights to Diagnosis. URL: https://fuselabcreative.com/the-role-of-healthcare-data-visualization/ (дата звернення: 20.05.2025).
7. Android Developers. Health Platform API | Android health & fitness. URL: https://developer.android.com/health-and-fitness/guides/health-services/health-platform (дата звернення: 20.05.2025).
8. A large scale analysis of mHealth app user reviews. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC9553639/ (дата звернення: 28.05.2025).
9. A Review of the Quality and Impact of Mobile Health Apps. URL: https://www.annualreviews.org/content/journals/10.1146/annurev-publhealth-052020-103738 (дата звернення: 28.05.2025).
10. Quality, Usability, and Effectiveness of mHealth Apps and the Role of User Reviews. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC10196903/ (дата звернення: 28.05.2025).

11. Exploring User Requirements for Mobile Health Service Systems. URL: https://michelangelo-scholar.com/upload/JDSSI-NO_00021A-20241009.pdf (дата звернення: 28.05.2025).
12. Understanding User Requirements for a Senior-Friendly Mobile Health App. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC9602267/ (дата звернення: 28.05.2025).
13. What are the Healthcare App Development Requirements? URL: https://www.addevice.io/blog/healthcare-app-development-requirements (дата звернення: 28.05.2025).
14. Requirements and Cost of a Healthcare Mobile App. URL: https://nordstone.co.uk/blog/requirements-and-cost-of-a-healthcare-mobile-app (дата звернення: 28.05.2025).
15. Best Practices for Secure Healthcare Mobile App Development. URL: https://mirrahealthcare.com/insights/8-strategies-to-ensure-data-privacy-and-security-in-healthcare-mobile-app-development (дата звернення: 28.05.2025).
16. What Regulations Must Your Mobile Health App Comply With? URL: https://thisisglance.com/learning-centre/what-regulations-must-your-mobile-health-app-comply-with (дата звернення: 28.05.2025).
17. Healthcare Mobile App Development: The 9-Step Guide. URL: https://kms-healthcare.com/blog/healthcare-mobile-app-development/ (дата звернення: 28.05.2025).
18. How to Develop a Top Health Monitoring App: PixelPlex Guide. URL: https://pixelplex.io/blog/health-monitoring-app-development-guide/ (дата звернення: 28.05.2025).
19. Effectiveness of Mobile Apps to Promote Health and Manage Disease. URL: https://mhealth.jmir.org/2021/1/e21563/ (дата звернення: 28.05.2025).
20. Development features and study characteristics of mobile health apps. URL: https://www.nature.com/articles/s41746-021-00517-1 (дата звернення: 28.05.2025).

21. Effectiveness of mobile health applications on clinical outcomes and patient engagement. URL: https://www.sciencedirect.com/science/article/pii/S2352013224000292 (дата звернення: 28.05.2025).
22. Health Behaviors and Outcomes of Mobile Health Apps and Patient Engagement. URL: https://www.scirp.org/journal/paperinformation?paperid=136800 (дата звернення: 28.05.2025).
23. Monitoring Symptoms of COVID-19: Review of Mobile Apps. URL: https://mhealth.jmir.org/2022/6/e36065 (дата звернення: 28.05.2025).
24. Use the Health app on your iPhone or iPad. URL: https://support.apple.com/en-us/104997 (дата звернення: 28.05.2025).
25. Google Fit guide: Everything you need to know. URL: https://www.androidauthority.com/google-fit-393110/ (дата звернення: 28.05.2025).