НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО
В.о. завідувача кафедри
_____Олександр КОВАЛЬ
«_____»_____2023р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій»
спеціальності 121 Інженерія програмного забезпечення
на тему: «Мобільний застосунок візуалізації даних Smart-теплиці»

Виконав:
студент IV курсу, групи ТІ-91
Панін Ілля Ігорович _____
(підпис)

Керівник:
проф. каф. ІПЗЕ, д.т.н Федорова Н.В. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент:

_____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2023

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Олександр КОВАЛЬ
(підпис)
«____»_____202_р.

ЗАВДАННЯ
на дипломну роботу студенту

_____Паніну Іллі Ігоровичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи
«Мобільний застосунок візуалізації даних Smart-теплиці» _____
керівник роботи Федорова Наталія Володимирівна д.т.н, проф. каф. ІПЗЕ _____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від "29" травня 2023 року № 2039-С
2. Строк подання студентом роботи __12 червня 2023_____
3. Вихідні дані до роботи: мова програмування Kotlin, середовище розробки Android Studio, операційна система Android.
4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно розробити): провести опис теплиць та їх показників, розробити мобільний застосунок для візуалізації даних, створити систему повідомлень, проаналізувати досвід використання системи.
5. Перелік ілюстративного матеріалу: існуючі застосунки-аналоги, візуалізація шарів архітектури, структура проекту, вигляд графіків, головний екран, екран додавання секції, детальний екран, система повідомлень.

6. Дата видачі завдання «30» вересня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Строки виконання етапів роботи	Примітка
1	Отримання завдання	30.09.2022	виконано
2	Дослідження предметної області	02.10.2022 – 20.02.2023	виконано
3	Постановка вимог до проектування системи	21.02.2023 - 28.02.2023	виконано
4	Дослідження існуючих рішень	29.02.2023 - 16.04.2023	виконано
5	Розробка програмного продукту	17.04.2023 - 10.05.2023	виконано
6	Тестування	11.05.2023 - 21.05.2023	виконано
7	Захист програмного продукту	17.05.2023	виконано
8	Оформлення дипломної роботи	22.05.2023 – 04.06.2023	виконано
9	Передзахист	05.06.2023	виконано
10	Захист	19.06.2023	виконано

Студент _____ Панін Ілля
(підпис) (ім'я, прізвище)

Керівник роботи _____ Наталія Федорова
(підпис) (ім'я, прізвище)

РЕФЕРАТ

Структура та обсяг дипломної роботи. Робота містить 50 сторінок, 14 рисунків, 1 додаток та 12 посилань.

Метою роботи є розробка мобільного застосунку візуалізації даних Smart-теплиці, що дозволить користувачам швидко та незалежно від місця отримувати інформацію про поточний стан теплиць, а також аналізувати статистичні дані про показники теплиць.

Для досягнення поставленої мети виконано такі завдання:

- описано теплиці та їх показники, проведений аналіз аналогічних мобільних застосунків;
- розроблено мобільний застосунок для візуалізації даних;
- проведено тестування застосунку та проаналізовано досвід використання системи.

Розроблено мобільний застосунок, що надає змогу користувачу отримувати інформацію про стан теплиць у режимі реального часу, візуалізувати показники теплиць у вигляді графіків.

Практичне значення одержаних результатів полягає в отриманні мобільного застосунку, що дозволяє власникам теплиць отримувати та аналізувати інформацію про теплиці в будь-якому місці, в будь-який час. Це дозволяє зробити процес слідкування за теплицями та рослинами в них більш простим, надійним та ефективним. Такий застосунок буде дуже корисним для людей, які працюють з теплицями в аграрній промисловості.

Ключові слова: теплиці, збір показників, візуалізація даних, режим реального часу, моніторинг стану.

ABSTRACT

Structure and scope of the thesis. The work contains 50 pages, 14 figures, 1 appendix and 12 references.

The purpose of the work is development of the mobile application for visualization of Smart-greenhouse data, which will allow users to quickly and regardless of location receive information about the current state of greenhouses, as well as analyze statistical data on greenhouse parameters.

To achieve the set goal, the following tasks were completed:

- greenhouses and their parameters are described, analysis of similar mobile applications is carried out;
- developed the mobile application for data visualization;
- the application was tested and the experience of using the system was analyzed.

The mobile application has been developed, which allows the user to receive information about the state of greenhouses in real time, to visualize parameters of greenhouses in the form of graphs.

The practical significance of the results obtained is to obtain the mobile application that allows greenhouse owners to receive and analyze greenhouse information anywhere, anytime. This allows you to make the process of monitoring greenhouses and plants in them simpler, more reliable and more efficient. Such an application will be very useful for people who work with greenhouses in the agricultural industry.

Keywords: greenhouses, collection of parameters, data visualization, real-time mode, condition monitoring.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

Android – операційна система створена та призначена для мобільних пристроїв.

SDK (Software Development Kit) – набір певних інструментів для розробки програмного забезпечення.

UI (User Interface) – інтерфейс користувача.

MVVM (Model-View-ViewModel) – шаблон проектування архітектури застосунка.

JVM (Java Virtual Machine) – віртуальна машина Java.

SQLite – компактна, вбудована система управління базами даних.

DSL (Domain Specific Language) – предметно-орієнтований язик програмування.

ВСТУП

Аграрна культура сьогодення ні яким чином не може обійтися без такої речі, як теплиці. Такий, начебто простий та легкий у створенні винахід дозволяє людям набагато покращити показники ефективності вирощування багатьох видів рослин, що потребують спеціальних умов.

Але такі теплиці мають декілька проблем, наприклад, кожний вид рослин потребує різних умов, тому потрібно створювати специфічні теплиці, проектуючи їх саме для цих умов, а також постійно слідкувати за їх станом, тому що погіршення умов може легко призвести до загибелі рослин.

Тому, зважаючи на розвиненість технологій була придумана така концепція як Smart-теплиці, вона дозволяє автоматизувати та тонко налаштувати процеси, які відбуваються у теплиці. Тобто автоматизація забирає на себе багато речей, які повинна робити людина, такі як: поливання, вентиляція, та, що є дуже важливим – знімання показників.

Навіть уявити собі сучасну систему, яка не має зручного інтерфейсу для слідкування за нею – неможливо. Саме тому, мобільний застосунок візуалізації даних Smart-теплиці, що був створений у цій роботі має на меті полегшити процес взаємодії людини з теплицею та надати їй змогу швидко та зручно слідкувати за показниками теплиці у режимі реального часу. При цьому, застосунок залишає в собі та використовує усі плюси саме мобільної платформи.

Застосунок має декілька варіантів слідкування за даними: текстовий варіант, графічний варіант та унікальний для мобільних застосунків – варіант з повідомленнями різних видів. Таким чином, користувач може отримувати інформацію навіть не відкриваючи застосунок.

Сама система зроблена з використанням принципів реактивності, тобто користувач не має робити ніяких оновлень даних вручну, усі оновлення даних відбуваються автоматично. Розроблявся застосунок на найпопулярнішій мобільній

платформі – Android, використовуючи мову програмування Kotlin, а також бібліотеку Room для створення бази даних та фреймворк Android SDK.

У зв’язку з тим, що тема Smart-теплиць є дуже перспективною у сучасній аграрній промисловості, результати розробки даної системи будуть корисними для усіх власників теплиць, які хочуть зробити свої володіння високотехнологічними та полегшити для себе процес використання таких теплиць.

Отже, розробка даного застосунку має високу важливість для покращення взаємодії системи Smart-теплиць з людиною, що дає позитивний ефект на загальний досвід використовування таких систем.

1 ПОСТАНОВКА ЗАДАЧІ

Дана бакалаврська дипломна робота має на меті розробку та створення мобільного застосунку для візуалізації даних Smart-теплиці, такий застосунок дозволяє власникам теплиць своєчасно отримувати інформацію про поточний стан теплиці, а також аналізувати дані показників.

При розробці даного програмного забезпечення будуть використані графіки для візуалізації даних показників, а також буде розроблена система повідомлень, яка слугує для своєчасного інформування користувача. Тому застосунок має мати декілька варіантів візуалізації інформації.

У створенні застосунку будуть використовуватися усі необхідні сучасні інструменти. Мовою програмування буде Kotlin, що працює за допомогою JVM, також буде використовуватись Android SDK – фреймворк для розробки мобільних застосунків під операційну систему Android. Дизайн застосунку буде розроблений за допомогою Material Design.

Робота буде вирішувати такі завдання:

- опис теплиці та показників які будуть зніматися з неї;
- створення мобільного застосунку для візуалізації даних теплиці та полегшення взаємодії користувача з нею;
- створення графіків для того, щоб користувач мав змогу провести аналіз показників;
- створення системи повідомлень для швидкого інформування користувача.

Мобільний застосунок має відповідати таким вимогам:

- використовувати принципи Clean Architecture;
- використовувати MVVM архітектуру для зв’язку між інтерфейсом та дата шаром застосунку;
- використовувати ін’єкцію залежностей та описувати інтерфейси, що не будуть залежати від своїх реалізацій;

- використовувати локальну базу даних для збереження певних даних користувача;
- бути незалежним від реалізації серверної частини;
- дизайн застосунку має відповідати сучасним стандартам;
- дизайн застосунку має бути зручним та нескладним;
- графіки для візуалізації даних кожного показника мають бути присутніми у застосунку;
- має бути наявною система повідомлень, яку користувач зможе налаштовувати під себе.

Висновки до розділу 1

У першому розділі була описана мета цієї дипломної роботи, також були описані основні завдання, які повинен виконувати мобільний застосунок, що розробляється у цій роботі. Ще був поданий список з основними вимогами до застосунку, були підняті як архітектурні, так і пов’язані з дизайном інтерфейсу рішення.

2 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛІЗ ІСНУЮЧИХ СИСТЕМ

Smart-теплиця – це автоматизована теплична система з цифровою аналітикою, яка потребує мінімальною участі людини для отримання максимальної кількості врожаю та мінімізує затрати матеріальних ресурсів для вирощування цього врожаю.

2.1 Інформація про Smart-теплиці та параметри системи

Почнемо з виділення характеристик теплиць з якими буде працювати наша система.

Для початку, це буде звичайна Smart-теплиця з можливістю контролювати освітлення та полив у ній, а також, звісно, вологість повітря. Теплиця може мати як один, так і бути поділена на декілька секторів, це залежить від її розмірів і призначення.

Якщо теплиця досить велика та використовується для вирощування декількох видів рослин, то кожен із цих секторів має мати свої власні вимірювальні прилади та працювати незалежно від інших. Навіть, якщо в теплиці вирощують тільки один вид рослин, але вона має досить великий розмір, то буде важливо мати декілька вимірювальних датчиків у різних її частинах, тому що проблеми можуть з’являтися та бути локальними.

Наведемо приклад теплиці з кількома секторами: в одному з них користувачу потрібно вирощувати помідори в повному циклі, а в іншому секторі вирощувати розсаду кількох видів. Усе це потребує настройки та контролю за середовищем вирощування, а також чіткого збору даних про стан секторів у вигляді декількох параметрів, які обговоримо нижче.

І так, ми маємо задачу виділити основні параметри, від яких залежить вирощування рослин та їх стан. Самих параметрів подібного роду можна нарахувати близько декількох десятків.

Але є декілька об’єктивних причин, чому нам не потрібно оперувати такою кількістю показників:

- кількість потрібного обладнання сильно зростає;
- необхідна кількість грошей на закупівлю підвищується;
- система приладів починає займати велику кількість місця та може заважати праці аграрія;
- складність системи теж зростає, через це потребується більша кількість зусиль на встановлення та налагодження обладнання, по тій же причині зростає можливість помилок у програмному забезпеченні та можливість виходу обладнання із ладу;
- так як для застосунку використовується мобільна платформа, то працювати з великою кількістю даних буде просто не зручно, зручність і простота стоїть на першому місці.

Беручи до уваги причини, описані вище, було прийнято взяти декілька критично важливих для вирощування рослин показників, показників, які не втрачають свого сенсу, незалежно від виду рослин.

Перерахуємо та опишемо ці параметри:

- освітленість – освітлення певної поверхні, що створюється світловим потоком, який падає на цю поверхню. Одиницею вимірювання освітленості є люкс, один люкс це освітленість світловим потоком один люмен, рівномірно розподіленим по поверхні площею один квадратний метр. Так як для нас важлива освітленість саме площі наших посадок, ця міра гарно підходить для нас. Саме від цього показника напряму залежить кількість енергії, яку будуть отримувати рослини;
- температура повітря – у цьому виборі немає ніяких складнощів, це найбільш зрозумілий для людини показник з усіх, який може нести в собі велику

кількість інформації про систему. Вимірювати температуру ми будемо у Цельсіях, так як ми в першу чергу подаємо інформацію для людини;

- вологість повітря – вміст водяної пари в повітрі, є одним з найважливіших параметрів середовища, що буде визначати наскільки комфортно почувають себе рослини. Вимірювати будемо саме відносну вологість, яка також залежить від температури, одиницею вимірювання є відсотки;
- вологість ґрунту - це кількість води в ньому, виражена у відсотках до маси абсолютно-сухого ґрунту. Вимірювати будемо у відсотках. Цей показник дозволяє слідкувати за роботою системи автоматичного поливу рослин, якщо він впаде надто низько, то рослини можуть зав’янути.

Вибравши параметри, які будуть використовуватись системою, ми маємо визначитися із частотою оновлення даних, тобто частотою з якою прилади будуть знімати показники. На щастя, це не є великою проблемою, тому оновлювати дані можна як раз у хвилину або дві хвилини, так і раз у п’ятнадцять хвилин.

Тепер, коли ми маємо на руках більшу частину інформації, яка необхідна для створення застосунку, потрібно розглянути проблеми, які наш застосунок має вирішувати, так само як і причини для його створення.

2.2 Опис проблеми

Спочатку потрібно зазначити, що середній користувач застосунку – це людина, яка розбирається у вирощуванні рослин та знає про потреби рослин, які вона вирощує.

У чому користувач не має бути сильно обізнаним – так це в приладах, що використовуються для замірів показників. Тобто користувач мусить мати інтерфейс, з якого він зможе отримувати всю необхідну інформацію, без необхідності взаємодіяти з приладами напряму.

Саме тому, створення нашого застосунку є необхідністю для комфортного використовування подібних систем та Smart-теплиць. Він дозволяє людині швидко отримувати повноту інформації відразу з декількох ділянок теплиці.

Також користувач без проблем може отримувати візуалізацію інформації у вигляді графіків. Ще, що є дуже важливим, людина може виконувати усі вищезазначені функції із майже будь-якого місцерозташування, єдиним обмежувачем є, звісно, наявність інтернет зв'язку.

Тоді виникає логічне питання, навіщо ми використовуємо саме мобільну платформу. Відповідь дуже проста – ми беремо усі плюси такого застосунку та робимо їх ще більшими. Тобто, кожна людина у сучасному світі має доступ до достатньо потужних смартфонів, при цьому зазвичай цей доступ є цілодобовим, бо телефон завжди є в твоїй кишені. Виходячи з цього, ми маємо покращену зручність та швидкість доступу до інформації.

Саме зручність і простота є найважливішими для таких систем та користувачів. Але, звісно, вибір мобільної платформи має свої проблеми та мінуси, наприклад, праця з великою кількістю інформації стає досить не зручною та складною.

Тут ми маємо розставляти пріоритети, у нашому випадку мобільна платформа буде більш підходящою. Причина проста – користувачу таких систем зазвичай потрібно слідкувати за поточним станом теплиці, щоб переконатися, що все працює справно та без помилок, якщо ж проблема виникає, то про неї потрібно швидко дізнатися.

У цьому застосунку людина зможе отримувати інформацію у режимі реального часу навіть не відкриваючи програму, а при виникненні проблем, отримувати повідомлення для найшвидшого інформування, та, як наслідок, найшвидшого усунення проблем. Також, якщо користувачу це потрібно, він має доступ до певної кількості статистичною інформації, що для зручності вивчення подана у вигляді графіків.

2.3 Аналіз існуючих рішень

На сучасному ринку існує не так багато застосунків, які взаємодіють зі Smart-теплицями, здебільшого це програми заточені під конкретні види датчиків, які мають сильно обмежений функціонал. В цілому це зрозуміло, бо зробити універсальний застосунок, який буде працювати з усіма приладами не представляється можливим.

Хоча ідея нашого застосунку намагається підійти до питання з більш широкої точки зору та зробити інтерфейс незалежним від приладів, але все ж є необхідним, як мінімум стандартизувати дані.

Роздивимось застосунок Memox Verdi, працює він виключно зі Smart-теплицею, яка має однойменну назву. Має досить простий дизайн, невелику кількість екранів та елементів з якими користувач може взаємодіяти, але свої функції застосунок виконує.

Побачивши головний екран цього застосунку відразу зрозуміло, які функції він виконує. Має список із теплиць та короткого опису їх показників, до якого можна додавати свої теплиці, та надавати їм назву. Далі можна перейти на детальний екран, де розміщена уся інформація про теплицю, її показники та налаштування.

Головну відмінність від нашого застосунку можна побачити саме на детальному екрані, цей додаток дозволяє в певній мірі керувати теплицею, у ньому можна включати та відключати освітлення та вентиляцію. Також, що є досить цікавим, застосунок на окремому екрані дозволяє задавати шаблони із налаштувань на декілька днів наперед. Тобто застосунок створює цикл із таких налаштувань і вони автоматично застосовуються кожен день без участі людини.

Але, цей застосунок не має ні статистичної інформації, ні графіків для її представлення. Також він не має функціоналу для сповіщень, що залишаються завжди активними та сповіщень про вихід показників за рамки, які задаються

користувачем. Подивитися на екран з детальною інформацією можна на рисунку 2.1.

Рисунок 2.1 – Детальний екран застосунку Memox Verdi

Також на рисунку 2.1 можна побачити, що програма дозволяє вибирати для кожної теплиці фото, яке користувач буде вважати потрібним. Досить непогана ідея, бо це дає людині можливість легко проводити асоціації та знаходити потрібну йому теплицю навіть не дивлячись на назви.

Далі роздивимось екран цього застосунку, що використовується для встановлення шаблони налаштувань. Можна побачити, що він складається із списку елементів, кожен з яких відповідає за конкретний день. Ці дні можна додавати та видаляти.

Для кожного дня можна налаштувати такі параметри як: вентиляція, освітлення та полив. При цьому кожен день може складатися із різних секцій, бо користувач може змінювати параметри в залежності від часу доби.

Також, що є гарним підходом до такого функціоналу, кожен шаблон можна зберігати та використовувати для різних теплиць, таким чином користувач не буде вводити однакові дані по декілька разів. Більше того, це дозволяє зменшити риск помилки вводу даних користувачем. Розробляючи наш застосунок, нам також варто пам’ятати, що одна помилка може поставити рослини в теплиці у небезпечне становище.

Побачити описаний вище екран застосунку можна на рисунку 2.2.

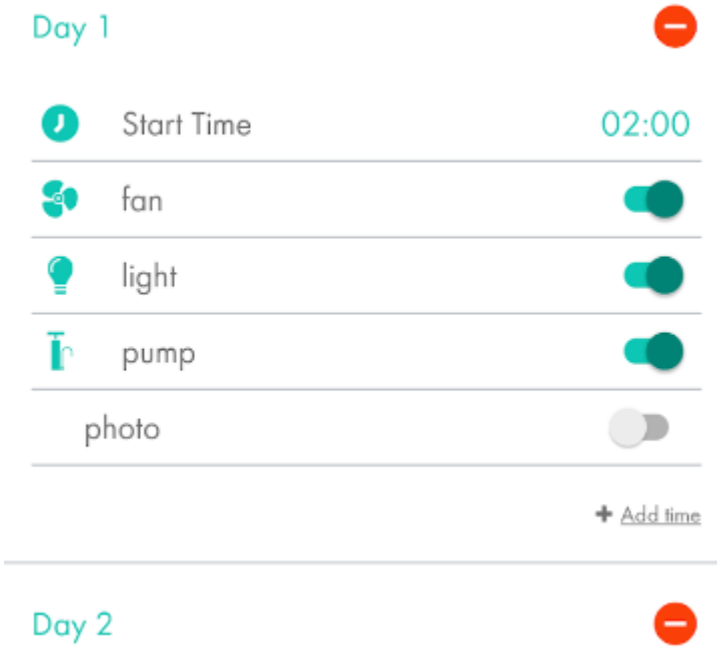

Рисунок 2.2 – Екран із шаблонами застосунку Memox Verdi

Але декілька речей у дизайні такого функціоналу є проблематичними та не інтуїтивними. По-перше не зрозумілою є функція додавання фото до налаштувань параметрів, додавання рисунку є гарною функцією в іншому екрані, але тут вона не виконує ніяких корисних дій.

Далі, користувачу не буде відразу зрозуміло, як працюють цикли цих шаблонів. Наприклад, маючи три елемента, що відображають три доби з різними налаштуваннями, що станеться коли настане четвертий день, чи почне шаблон виконуватись заново та як про це буде повідомлено користувачу. Також не

зрозумілим для користувача є те, що означає перший день у шаблоні. Він почне виконуватись у той же день, коли користувач його задіяв, чи на наступний день.

Але звісно, потрібно зазначити, що хоч тут і є певні незручності у використанні цього функціоналу, сама його наявність є великим плюсом цього застосунку.

Далі роздивимось другий застосунок, що виконує схожі функції з нашим застосунком, його назва – Balithi Smart Greenhouse.

Перш за все потрібно сказати, що застосунок має дуже простий дизайн, фактично це навіть надто простий дизайн, який складається лише з кількох елементів та двох екранів, один з яких це екран реєстрації.

Тож на головному екрані можемо побачити декілька елементів, кожен з яких складається з іконки та тексту. У цих елементах і відображається інформація про основні показники теплиці. Загалом використання іконок допомагає користувачу швидше орієнтуватися в інформації яку він бачить, також використання унікальних кольорів для відображення кожного елементу теж має сенс і покращує досвід користування застосунком.

Ще одним елементом на головному екрані є кнопка, що дозволяє відкрити відео наживо з теплиці, при умові, що така камера, звісно, є. Сама ідея є досить непоганою та може бути унікальною особливістю цього застосунку. Але на практиці, такий функціонал буде великою тратою зусиль, який потім може бути використаний лише з цілю перевірки, що в тебе в теплиці немає ніяких сторонніх істот.

Встановлення камери та її підтримка, кількість ресурсів яка буде потрібна для зберігання записів із цієї камери просто не можуть бути виправданими. Не кажучи вже про те, що передача потокового відео до застосунку та коректне його відображення є нетривіальною задачею та потребує великої кількості часу та знань для розробки.

Кажучи про наявний функціонал, на жаль, на цьому він був описаний повністю.

Таким чином, цей застосунок не має можливості слідкувати відразу за декількома теплицями, що є великим лімітом для користувача. Також застосунок не має жодних графіків для візуалізація даних. Зрозуміло, що і повідомлення про стан теплиці цей застосунок не відправляє. Ще варто зазначити, що дизайн мобільного застосунку не надто відповідає сучасним стандартам [7].

Головний екран застосунку Balithi Smart Greenhouse можна побачити на рисунку 2.3.

Рисунок 2.3 – Головний екран застосунку Balithi Smart Greenhouse

Тож, можемо зробити висновок, що цьому застосунку бракує функціоналу, але все ж таки основну свою функцію він виконує.

Загалом, роздивлятись більше застосунків не має великого сенсу, бо всі вони дуже подібні, є такі, що мають трохи більше або менше функцій, є такі, що мають складнішу систему взаємодії з апаратурою, та можуть підключатися до неї напряму. Але основна частина функцій у таких додатках подібна, тож більша різниця полягає у дизайні застосунку та приладами під які він заточений, деяка

частина застосунків використовують застарілий дизайн та його програмну реалізацію.

Також, варто зазначити, що на ринку, у відкритому доступі присутня досить невелика кількість мобільних застосунків подібних до того, що розробляється в цій роботі.

Висновки до розділу 2

У цьому розділі була описана предметна область системи, що розробляється у цій роботі. Також була подана інформація про Smart-теплиці та їх застосування. Були виділені основні параметри та показники теплиці з якими буде працювати мобільний застосунок, а також основні функції застосунка, які користувачі хочуть бачити у ньому.

Продовжуючи, був проведений аналіз існуючих рішень, подібних застосунків, що знаходяться у відкритому доступі. Були виділені переваги та недоліки тих чи інших технічних рішень у застосунках, а також їх дизайну. Був зроблений акцент на описі досвіду використання цих застосунків та їх унікальних функціях.

3 АНАЛІЗ ІНСТРУМЕНТІВ ДЛЯ РОЗРОБКИ ЗАСТОСУНКУ

Вибір засобів для реалізації програмних застосунків є дуже важливою частиною розробки. Для нашого застосунку при виборі інструментів потрібно сконцентруватись на нових технологіях, які вбирають в себе найкращі практики та дозволяють значно спростити розробку більшості функцій.

Вибір оптимальних технологій дає можливість знизити затрати часу на розробку, кількість зусиль на підтримку та оновлення програмного продукту у майбутньому, також деякі технології є критично важливими, без яких певний функціонал програми просто неможливо буде реалізувати.

3.1 Операційна система Android

Перш за все потрібно визначитись із операційною системою під яку буде розроблятися програма, так як ми робимо мобільний застосунок вибір лише з двох операційних систем: Android та IOS.

Операційною системою Android користуються більше 70 відсотків усіх мобільних пристроїв, тому перевага була віддана їй. Тут все просто, застосунок має охопити більшу кількість потенційних користувачів.

Також варто зазначити, що розробляти версії застосунку відразу для двох операційних систем потребувало б вдвічі більше зусиль та часу, тому цей варіант був відкинутий одразу.

Ще є певні технології, що дозволяють робити застосунки кросплатформними, але вони мають досить великі мінуси, все одно потребують тонкого налаштування під кожну із систем та не надають такого ж рівня контролю над застосунком та функціями пристрою, як нативні варіанти.

Операційна система Android створена на базі ядра Linux, при цьому деяка частина шарів системи була написана за допомогою мови програмування C++, але важливо те, що базовим елементом цієї ОС є реалізація віртуальної машини Java.

Тобто все програмне забезпечення для Android спирається на Java, що ми будемо далі використовувати. Одним із плюсів її використання є наявність Garbage Collector – це підпрограма, що автоматично керує оперативною пам'яттю комп'ютера, звільнюючи пам'ять від об'єктів, які більше не використовуються [1].

Сама ж платформа надає нам широкий спектр інструментів для розробки застосунків, які входять в Android SDK. Тому далі, ми повинні визначитися з версіями SDK під якими буде працювати застосунок. Було вирішено зробити мінімально потрібний SDK 28, на усіх пристроях з вищим SDK застосунок буде працювати так само.

3.2 Мова програмування Kotlin

Вище згадувалось, що для написання програмного забезпечення під Android ми маємо використовувати Java, але у написанні мобільних застосунків вона вважається застарілою, тому будемо використовувати Kotlin – мова, що працює на JVM, але при цьому виправляє більшість проблем Java [2]. Виділимо декілька переваг мови Kotlin:

- набагато краща реалізація функціональної парадигми, справжні лямбди та можливість робити з них ланцюги;
- наявність nullable та non-null типів;
- можливість створювати так звані extension функції, які дозволяють розширювати класи до яких у розробника не має доступу;
- наявність потужної системи для створення делегатів;
- наявність data класів та відмова від крапки з комою.

Усе це навіть не всі плюси, але вже їх достатньо, щоб зробити розробку застосунка набагато легшою.

Також, не рахуючи нативну Java реалізацію багатопоточності, Kotlin має свою реалізацію асинхронного програмування під назвою Coroutines. Це дуже потужний інструмент, який втілює в собі всі сучасні концепції асинхронності, так як це робить RxJava, але при цьому корутини повністю позбуваються необхідності робити колбеки. Сама ідея корутин полягає в написанні синхронного коду, що буде працювати як асинхронний [3].

Це велика перевага, бо у бібліотеках, які працюють на колбеках, при написанні складних модулів можуть бути по десять вкладених в один одного колбеків. Така ситуація робить код дуже складним для розуміння та змін у подальшому.

При цьому, реалізація корутин є набагато швидшою за звичайні інструменти для роботи з багатопоточністю. А сам фреймворк для написання застосунків Android є сумісним за багатьма концепціями корутин та реалізує велику кількість допоміжних функцій, які працюють завдяки корутинам.

3.3 Інструмент для ін’єкції залежностей Koin

Ін’єкція залежностей є важливою концепцією, що допомагає значно спрощувати створення залежностей, тобто об’єктів, у вашому коді. При цьому вона дозволяє збирати створення всіх залежностей в одному або кількох модулях та дозволяє повністю контролювати їх створення.

Правильно зроблена ін’єкція дозволяє зробити підміну залежностей або їх параметрів справою кількох секунд, що дуже позитивно відображається на тестуванні та створенні декількох реалізацій одного модуля, це може використовуватись наприклад у дебаг версіях програми.

При виборі Koin надавалася перевага в першу чергу простоті розробки на цій бібліотеці, бо в той час коли його аналог Dagger надає більший контроль для складних проектів, Koin набагато легше налаштувати та запустити.

Також, в Android одним з елементів, який є досить складним у створенні є ViewModel, детальніше про неї буде написано нижче. Koin дозволяє вирішити проблему створення цього елемента та передачі до нього залежностей за допомогою делегатів, що в свою чергу спрощує написання коду [6].

3.4 Бібліотека для створення SQLite баз даних

Для створення локальної бази даних ми будемо використовувати бібліотеку Room. Усі локальні бази даних у Android застосунках мають бути в форматі SQLite, що є зрозумілим вибором, бо вони зазвичай не повинні утримувати велику кількість інформації, а також їм не потрібно робити надто складні запити чи транзакції. Тому легкість бази даних є важливим фактором.

У цілому Room майже не має конкурентів, альтернативою є написання коду та праця напряму з SQLite, що не є зручним підходом. Інші ж бібліотеки працюють дуже подібним образом, але при цьому мають меншу кількість функціоналу, тому Room вважається стандартом.

Room дозволяє створювати таблиці та базу даних за допомогою простих класів та анотацій. При цьому інтерфейси взаємодії з таблицями, тобто написання запитів до них також робиться за допомогою анотацій [5].

З таким підходом розробники не мають знати багато інформації про SQLite, для того щоб створювати навіть складні бази даних. При цьому такий рівень абстракції дозволяє зберегти майже весь функціонал нативного SQLite, тож ситуації де бібліотека обмежує тебе є досить рідкісними.

3.5 Середовище розробки Android Studio

Між середовищами розробки вибору взагалі немає, єдине середовище, що має весь потрібний функціонал для розробки мобільних застосунків це Android Studio.

Вона була створена самими Google, для того щоб розробники не мали ніяких проблем у цьому питанні. Із гарного, це середовище розповсюджується повністю безоплатно, відразу з повним функціоналом.

Загалом, Android Studio має усі стандартні інструменти для допомоги розробнику, такі як: підсвітка коду, автозаповнення, автоматичний імпорт потрібних у файлі модулів, редагування коду із заміною по всьому проекту та ін.

Але це середовище також пропонує багато унікальних функцій: автоматичне завантаження на встановлення емулятору з потрібним рівнем SDK та його запуск прямо в Android Studio, едітор для xml файлів та Jetpack Compose разом із показом того, як код буде виглядати на екрані телефону, автоматичне підключення до пристрою та зчитування логів з нього. Також є можливість продивлятися локальні бази даних, які використовує застосунок прямо із середовища розробки.

Варто не забувати також про дуже корисні інструменти для виправлення багів у програмі. Ще, це середовище розробки надає зручні інструменти для перевірки кількості системних ресурсів, які використовує мобільний застосунок. Це дозволяє точно вимірювати ефективність застосунку, що є досить важливим для мобільної платформи.

Таким чином, Android Studio це повністю компетентне середовище розробки, яке надає велику кількість переваг та можливостей розробнику та значно спрощує процес розробки застосунків.

3.6 Система керування версіями Git

Будь-яка розробка хоча б скільки серйозного програмного забезпечення ніколи не проводиться без системи керування версіями. Такий інструмент є необхідним для проекту, так як зі збільшенням складності системи, зростає кількість проблем та багів, які потрібно виправляти, а також кількість нових функцій, які потрібно постійно додавати, створюючи нові версії.

Таким чином, через певну кількість версій слідкувати за усіма версіями без допомоги спеціалізованих інструментів стає неможливим. Без них виправлення помилок у нових версіях проекту та відкати до попередніх версій стають надто складними задачами.

Вибір Git є абсолютно очевидним, так як ця система керування версіями вважається золотим стандартом та фактично не має конкурентів. Також ця система має відкритий вихідний код та поширюється безоплатно.

Ще потрібно зазначити, що за довгий час свого існування для Git було створена велика кількість документації та інформації, що допомагає працювати з ним.

Виділимо декілька переваг, які надає ця система та які використовуються в цій роботі. По-перше Git дозволяє зберігати код застосунку у віддаленому репозиторії, що забезпечує можливість резервного копіювання та поширення коду проекту [8].

Далі, Git дозволяє уникати конфліктів між різними версіями коду а також має функціонал для їх об’єднання. Це може бути використано наприклад при розробці окремих функцій та особливостей застосунку, кінцева доля яких у проекті ще не вирішена.

Також Git дозволяє стежити за змінами коду у файлах, коли та ким вони були створені. Це особливо корисно для орієнтування в коді програми, зважаючи на те, що розробник може с часом забувати коли і для чого створювались певні частини проекту.

Можливість праці відразу з декількома гілками коду дозволяє одночасно розробляти декілька різних функцій застосунку, велика швидкість роботи алгоритмів цієї системи керування версіями та відносна простота її використання робить Git ще більш корисним вибором при розробці програмного забезпечення.

Підсумовуючи, можна дійти до висновку, що Git є показовою системою керування версіями, яка здатна значно спростити цикл розробки мобільного

застосунку. Таким чином, розробник застосунку буде мати можливість приділити більше часу та уваги розробці функціоналу мобільного застосунку.

3.7 Інструмент для створення графічних елементів

Для створення графічних елементів у коді в розробці мобільних застосунків під операційну систему Android існує два основних підходи.

Поговоримо про перший з них – Jetpack Compose. Цей варіант розробки інтерфейсу з'явився досить недавно та вважається більш прогресивним ніж інший підхід. Jetpack Compose пропонує розробникам перенести усю імплементацію інтерфейсу з xml файлів у звичайний код на Kotlin. Тобто відразу можна побачити основні плюси: можливість розробляти застосунки використовуючи виключно мову Kotlin, можливість використовувати усі можливості повноцінної мови програмування прямо в описані графічних елементів [10].

Наведемо простий приклад того, як це може працювати: розробнику потрібну зробити шість однакових елементів, за допомогою Jetpack Compose він може створити один елемент і шість разів відобразити його за допомогою циклу. Можна відразу побачити, чому це краще ніж використовувати xml для цього.

Сама бібліотека працює на такій можливості мови Kotlin, як створення власних DSL [9]. Також там використовуються функції, які позначаються анотацією @Composable, після цього всередині функції описується графічний елемент, а далі можна використовувати цю функцію де завгодно. Такий підхід сильно спрощує створення складних елементів на екрані користувача та підвищує читабельність коду.

Але не дивлячись на це, у цій роботі було вирішено використовувати стандартний підхід з використанням мови розмітки – xml. Таке рішення було прийнято тому, що наразі Jetpack Compose не є повністю безпечним для використання у серйозному проекті, також по роботі з цією бібліотекою поки не має достатньої кількості інформації.

Через деякий час коли бібліотеку розвинуть та виправлять помилки та незручності вона неодмінно буде новим стандартом у розробці графічного інтерфейсу.

Висновки до розділу 3

У цьому розділі були описані основні інструменти, які потрібні для створення нашого мобільного застосунку. Був проведений аналіз таких інструментів як: операційна система під якою буде працювати застосунок, мова програмування для його написання, фреймворк для створення самого застосунку, бібліотеки для створення бази даних та імплементації ін’єкції залежностей. Також було обговорено найкраще середовище для розробки застосунку та система керування версіями. При цьому, для кожного інструменту були приведені аналоги та був обґрунтований його вибір.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ ТА ЙОГО АРХІТЕКТУРИ

Реалізуючи застосунок було приділено особливу увагу таким факторам як: читабельність коду, його структурованість та модульність, також він має бути надійним та легко тестуватись. Важливим було використання різних кращих практик і стандартів при написанні мобільних застосунків.

Багато уваги було приділено архітектурі проекту та принципам за допомогою яких дата шар буде взаємодіяти та обмінюватись інформацією з шаром, що відповідає за інтерфейс користувача.

4.1 Архітектура застосунку

Застосунок проектувався відповідно до вимог Clean Architecture. Це паттерн, призваний максимально розділяти код та модулі програми на декілька шарів, кожен з яких повинен відповідати за свої функції та не залежати від інших. При цьому, такі шари мають виділити частину модулів, які можна буде використовувати в незалежності від фреймворків, що використовуються для створення застосунку. Візуалізацію цих шарів можна побачити на рисунку 4.1.

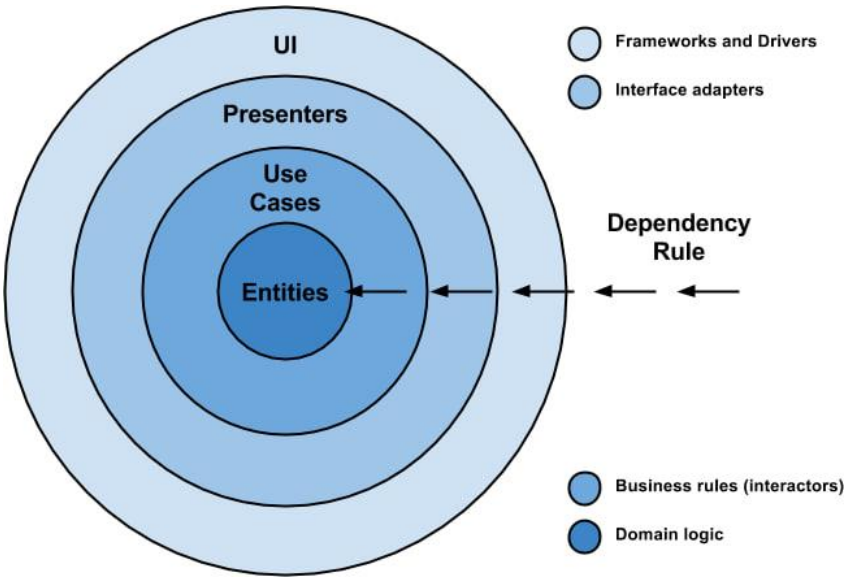

Рисунок 4.1 – Візуалізація шарів Clean Architecture

Як ми можемо побачити з рисунка вище, у центрі діаграми знаходиться шар з Entities. Переводячи, це наш найбільш незалежний шар коду, в якому знаходяться усі дата класи, тобто класи створені тільки для зберігання інформації, а також усі класи які ці дані передають в одну чи іншу сторону. Наприклад, класи та інтерфейси, що відповідають за отримання та передачу даних на сервер або на локальну базу даних.

Йдучи далі можна побачити шар під назвою Use Cases. Цей шар відповідає за усю бізнес логіку застосунка, тобто будь-яка обробка даних та приведення їх до потрібного виду, розрахунки, тощо. Також до цього шару можна віднести спеціальні модулі, що поєднують між собою локальну та дистанційну частину домену, наприклад визиваючи одну функцію з цього модуля будуть проводитися операції одночасно з сервером та локальним сховищем. Такі класи ми будемо застосовувати далі.

Наступний шар це Presenters. Це спеціальні класи, що використовуються для того щоб, зв’язати між собою дата шари та UI шар, це потрібно для того щоб шар інтерфейсу отримував відразу готові дані та не займався ніякою їх обробкою, він не повинен знати про дата шари та як з ними взаємодіяти, для цього в нього є презентери.

Ідея полягає в тому, що змінивши хоч усю реалізацію дата шару, нам не потрібно буде міняти шар з інтерфейсом, його логіка залишається незалежною. Але, для реалізації цього існує декілька різних підходів, найпопулярніші з яких: MVP, MVVM, MVI. Наш застосунок буде використовувати реалізацію MVVM, яку детальніше буде описано потім.

Розібравшись із теоретичною частиною, можемо подивитись на те, як ми структуруємо наш застосунок. Перш за все, потрібно сказати, що ми використовуємо трохи спрощену версію паттерна Clean Architecture, так як деяка бізнес логіка не виноситься у так звані Use Cases. У цілому, цей паттерн дуже гнучкий, тому часто його змінюють у конкретних реалізаціях.

У нашому проекті є декілька модулів, кожен з яких є відповідальним за різну роботу, цю структуру можна побачити на рисунку 4.2.

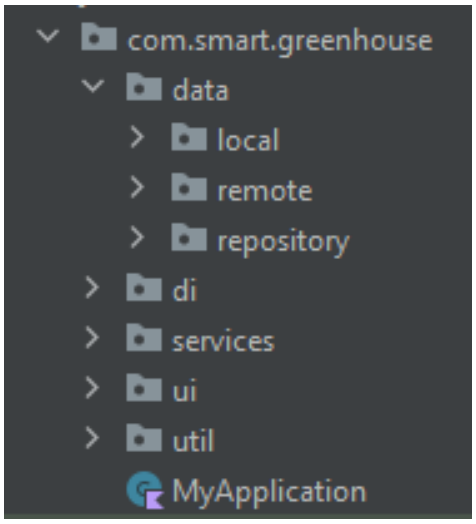

Рисунок 4.2 – Структура проекту

Перший з модулів – це local, у ньому створюється база даних та описується її логіка, а також там описуються дата класи для інформації, що вона зберігає. Варто зазначити, що там описується саме локальна версія дата класів.

Далі йде remote, модуль у якому ми описуємо будь яку взаємодію із сервером та віддаленими джерелами даних, а також відповідні дата класи для зберігання інформації, що надходить з них. Що є важливим, так це те, що спочатку ми описуємо інтерфейс, а вже потім реалізуємо його так як нам буде потрібно, при цьому далі модулі будуть використовувати лише цей інтерфейс. Такий підхід дозволяє нам швидко підміняти модулі, якщо це буде потрібно.

У модулі repository ми описуємо клас репозиторію, який поєднує між собою два модулі описані вище, саме через нього наші презентери будуть спілкуватися із дата шаром. Він також зроблений через інтерфейс, для того щоб презентери на залежали від конкретних реалізацій репозиторію.

Далі, останнє, що ми маємо, це ui модуль, він відповідає за програмну реалізацію інтерфейсу та взаємодію із ним. Цей модуль складається з декількох частин, кожна з яких відповідає за свій екран, який бачить користувач. Такі частини

складають з Fragment – клас, який працює напряму з елементами інтерфейсу та ViewModel – це і є наш презентер.

Так як ми використовуємо MVVM (Model-View-ViewModel), наш презентер має назву ViewModel. Сама реалізація цього паттерну виглядає так, передача даних між двома частинами відбувається за допомогою паттерну спостерігача. Працює це досить просто: фрагмент визиває функцію вьюмоделі, яка замість того щоб відразу повертати дані, поміщає їх у спостерігач на який, в свою чергу, підписується фрагмент. Таким чином фрагмент має можливість оперувати даними за допомогою лише кількох підписок, а фрагмент може запускати асинхронний код не заставляючи фрагмент очікувати результату функції.

4.2 Ін’єкція залежностей

Наш модуль для ін’єкції залежностей написано дуже стисло та лаконічно, але при цьому він дозволяє зберегти значну кількість часу при розробці. Тож, за допомогою бібліотеки Koin нам потрібно описати створення кількох залежностей. Сама бібліотека працює на Kotlin DSL, ця технологія дозволяє буквально описувати свою кастомну мову, тому опис залежностей виглядає дещо незвично, але насправді це дуже потужний інструмент для розробників бібліотек.

Для початку нам потрібно створити module, що робиться простим відкриттям блоку з однойменною назвою. Далі, нам потрібно описати створення таких залежностей як: об’єкт бази даних, реалізація інтерфейсу для взаємодії з базою даних, а також реалізації інтерфейсу репозиторія.

Так як усі ці залежності не повинні мати більше ніж один екземпляр, їх ми створюємо за допомогою блока single. Це дозволяє нам реалізувати паттерн Singleton.

Як можна побачити, тут ми також створюємо екземпляр ViewModel за допомогою спеціального блоку, це сильно допомагає, бо потім у коді фрагменту для того щоб її створити потрібно написати лише один рядок коду, який

використовує делегати. Проблема стандартної реалізації в тому, що екземпляр класу ViewModel не можна створювати напряму, тому приходиться вручну створювати фабрику для кожної із реалізацій.

4.3 Реалізація бази даних та потік даних у програмі

Локальна база даних була реалізована за допомогою бібліотеки Room. Роздивимось те, як вона працює. Перша річ, яку потрібно створити це таблиці, це робиться досить легко, за допомогою анотації @Entity та дата класу з назвою, яка буде назвою таблиці. Також за допомогою анотації @PrimaryKey потрібно вказати поле, що буде використовуватись як ключ.

Ми використовуємо локальну базу даних для зберігання секцій, які користувач створив та відслідковує, а саме для зберігання назв, які користувач може змінювати в будь-який момент.

Далі, нам потрібно створити інтерфейс для взаємодії з базою даних, робиться це за допомогою анотації @Dao та звичайного інтерфейсу. Такий інтерфейс має в собі декілька функцій, які позначені анотацією @Query. Як можна зрозуміти ця анотація приймає рядок як параметр, в якому можна описати запит до бази даних, але бібліотек також має декілька вже готових запитів, які можна застосувати за допомогою анотацій @Insert та @Delete [5].

Також варто зазначити, що одна із наших функцій повертає тип даних Flow, це тип даних взятий із корутин, сутність, яка відображає собою потік даних. Таким чином нам достатньо лише один раз визвати цю функцію та підписатись на потік даних, який вона повертає, після цього кожен раз коли дані в БД будуть змінюватись, ми будемо отримувати їх автоматично [4].

Далі, єдине що залишається зробити, це написати клас самої бази даних, робиться це за допомогою абстрактного класу та анотації @Database, яка приймає назву класів, що будуть використовуватись як таблиці, в параметри. Також

усередині класу потрібно створити абстрактні функції, що будуть повертати інтерфейси для взаємодією з БД.

Говорячи про потік даних, потрібно показати дані, які повертаються до нас із сервера, відповідний дані можна побачити на рисунку 4.3.

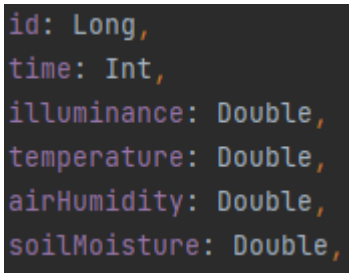

Рисунок 4.3 – Дані для SectionInfoRemote

Як ми можемо побачити саме об’єкти цього класу будуть нести основну частину інформації. Далі, ми переходимо до репозиторію, який при виклику функцій для отримання даних про секції, буде повертати тип даних Flow, про який написано вище. Функції репозиторію будуть із заданою періодичністю отримувати дані з сервера та об’єднувати їх із локальними даними, бо імена для секцій користувач задає локально.

Таким чином ми маємо безперервний потік даних, які будуть утримувати всю необхідну інформацію для використання в інтерфейсі та який оновлюється автоматично з певним періодом часу. Так ми досягаємо реактивності для нашого застосунку.

4.4 Реалізація графіків для показників

Створення та імплементація графіків у мобільних застосунках на операційній системі Android є досить складною задачею.

Для вирішення цієї проблеми існує декілька підходів. Один з них, який є найбільш базовим та контрольованим – це створення так званого CustomView.

Взагалі потрібно сказати, що View в Android розробці це деякий графічний елемент, який розробник описує в коді. Коли говорять про View, то зазвичай мають на увазі вже створений розробниками фреймворку елемент, який розробник застосунку може просто додати до коду та налаштувати під свою задачу, так як він того хоче.

Таким чином, CustomView – це створення та описання розробником застосунку свого власного елементу. І тут лежить досить велика проблема – така річ потребує великих знань та досвіду, тому що це, фактично, праця з фреймворком на досить низькому рівні.

Йдучи ще далі, побудова та відображення графіків потребує працю з елементами рисовки, які в свою чергу потребують працю з пікселями на екрані. А необхідність рахувати пікселі вручну, особливо враховуючи, що екрани пристроїв можуть бути дуже різними, це дійсно недобре.

Тому такий підхід може потенційно потребувати надто велику кількість часу та створити велику кількість багів та проблем у застосунку. Варто зазначити, що Android SDK не має вбудованих View та методів для роботи з графіками. Тож перший підхід можна не розглядати.

На щастя існує інший варіант вирішення проблеми – це знаходження відкритої для користування бібліотеки, що має в собі вже створені CustomView, які дозволяють будувати такі графіки.

Називається знайдене готове рішення MPAndroidChart – це досить непогана бібліотека, яка дає розробниками легко реалізувати велику кількість графіків різних видів. Самі графіки мають в собі декілька вбудованих функцій, які зазвичай вважаються стандартом для робити з такими графіками. Бібліотека є досить гнучкою та надає широкі можливості для кастомізації графіків.

Почнемо з того, що для створення найпростішого графіка нам потрібно лише створити елемент під назвою LineChart. Потім для створення масиву даних, що будуть відображатися на ньому, нам потрібно обгорнути дані застосунку в об’єкти класу Entry.

Клас Entry є досить простою обгорткою, яка зберігає в собі інформацію про координати точки на графіку у вигляді двох чисел з плаваючою точкою. Далі потрібно обгорнути список з наших точок в ще один тип даних під назвою LineDataSet.

Так це ще одна обгортка для даних, що не є гарною практикою написанню коду, але цей тип даних слугує для внутрішніх намірів бібліотеки, а також для деяких видів кастомізації відображення точок. Наприклад, тут за допомогою методів можна змінити колір точок, визначити розмір відображених точок, а також чи потрібно робити підписи із значенням кожною з них.

Але після відображення нашого графіку ми можемо відразу побачити проблеми, що потребують вирішення. По стандарту графік створюється за відображається з усіма можливими функціями ввімкненими. Тобто такий графік є абсолютно не читабельним, побачити його можна на рисунку 4.4.

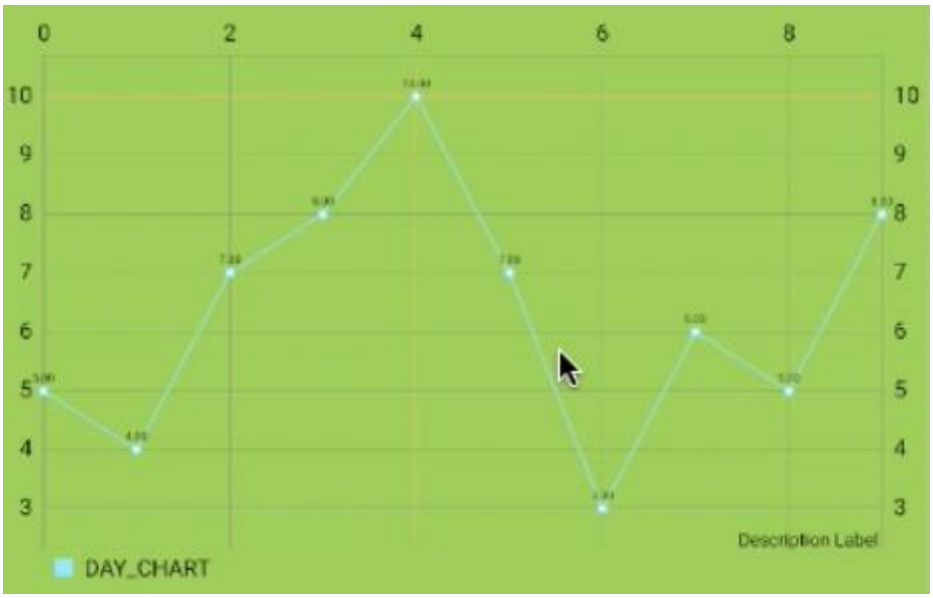

Рисунок 4.4 – Стандартний вид графіку

Як можна побачити з рисунку, ми маємо зайву ось у правій частині графіку та ось, що розташовується зверху графіка замість нормального нижнього положення. Також тут ми маємо дві зайві написи, які лише погіршують загальний вид графіку.

На щастя, розробник цієї бібліотеки зробив так, щоб усі ці параметри можна було легко змінити. Тому при старті нашого екрану ми конфігуруємо графік з потрібними нам параметрами.

4.5 Реалізація системи повідомлень

Для реалізації системи повідомлень у цьому мобільному застосунку ми будемо використовувати стандартні інструменти Android SDK.

Перш за все потрібно почати з визначення повідомлень, які ми маємо зробити та відобразити, бо вони поділяються на декілька видів, кожен з яких має свої обмеження.

Тому почнемо з головного типу повідомлень, які має відправляти наш застосунок, а саме постійного повідомлення. Постійне повідомлення – завжди залишаються активними та постійно оновлюються, виводячи нові дані для користувача. Таке повідомлення користувач не може відмінити чи закрити простим жестом руки, як це можна зробити з іншими повідомленнями. Зачинити ж його користувач може використовуючи застосунок напряму. Також це повідомлення може працювати навіть тоді, коли застосунок був повністю закритий.

Саме така поведінка нам потрібна для реалізації одного із завдань цієї роботи, а саме – можливість підписки користувача на оновлення даних для однієї конкретної теплиці, при цьому користувач має отримувати доступ до інформації навіть при закритому застосунку.

Важливо зазначити, що таке повідомлення завжди представляє собою деякі дії, що відбуваються на задньому фоні, доки користувач може використовувати будь-який інший застосунок. У нашому випадку виконується робота по запиту потрібних даних, їх зміна та відображення у вигляді повідомлення.

На цьому етапі ми стикаємося з обмеженнями операційної системи Android. Так як вона створена на мобільних пристроїв, вона завжди пильно слідкує за виділенням системних ресурсів.

Тобто операційна система не може собі дозволити виконувати дуже велику кількість операцій на задньому фоні. Її пріоритетом завжди буде те, на що користувач дивиться у даний момент часу. Саме тому ніякий застосунок не може просто так запустити процеси, що будуть виконуватись на задньому фоні без відома користувача [1].

Таким чином, операційна система змушує застосунки повідомляти користувачів про те, що певні процеси виконуються на задньому фоні. Розробник застосунку має завжди показувати постійне повідомлення, якщо він хоче виконувати певні дії навіть тоді, коли застосунок був повністю закритий. Саме це і є нашим випадком.

Також варто зазначити, що процеси можуть бути запущені на задньому фоні без відома користувача, але тільки якщо застосунок який їх запустив відчинений і використовується.

Розуміючи те як поводить себе операційна система розробити функціонал описаний вище стає набагато легше. Але навіть так це залишається складною задачею через обмеження системи.

Запускається процес на задньому фоні за допомогою сутності під назвою Service. А саме для нашого випадку використовується ForegroundService, цей компонент відрізняється тим, що він потребує наявність постійного повідомлення під час своєї роботи та може працювати незалежно від стану роботи застосунка, що запустив його. Якщо постійне повідомлення не буде відображене через декілька секунд після запуску цього компоненту, то застосунок просто буде закритий системою.

Для створення повідомлення потрібно спочатку створити канал для повідомлень через який буде проходити повідомлення. Це також одна з вимог операційної системи, потрібна для того, щоб користувач мав змогу завжди зайти у налаштування застосунку та відключити непотрібний йому канал повідомлень. Декілька каналів створюються якщо застосунок має декілька різних по сенсу видів повідомлень.

Самі повідомлення створюються досить легко, для цього потрібно лише використати клас Notification, який реалізує паттерн Builder. Використовуючи будівника потрібно задати необхідні розробнику параметри для повідомлення. Наш застосунок використовує такі параметри як: іконка повідомлення, заголовок для повідомлення, заголовок для його контенту, текст та стиль.

Один зі стилів, які використовує наш застосунок – це стиль великого тексту, він дозволяє створювати повідомлення, які тримають в собі велику кількість тексту. Спочатку такі повідомлення знаходяться у згорнутому стані, але користувач може розгорнути їх та побачити контент повністю.

Побачити повідомлення у згорнутому стані можна на рисунку 4.5.

Рисунок 4.5 – Повідомлення у згорнутому стані

Згадаємо, що наш застосунок має також другий вид повідомлень. Вони повинні відображатись коли користувач слідкує за певною секцією, тобто коли перший вид повідомлень вже є активним.

Ці повідомлення мають бути звичайними, тобто на відміну від попередніх, вони мають звукове сповіщення, а також можуть бути закриті без проблем для користувача.

Так як ці повідомлення використовують ту й саму інформацію, що і попередні, вони можуть бути створені в тому ж самому фоновому процесі. Для того щоб розрізняти їх між собою, для них буде створений інший канал повідомлень. Цей канал буде налаштований з більш високим пріоритетом, таким чином при надходженні повідомлення буде програний звуковий сигнал.

Самі ж повідомлення будуть створенні так само як і попередні, але вони матимуть іншу іконку та, звісно, інший контент.

Ще варто сказати, що усі процеси у нашому сервісі працюють на задньому фоні завдяки корутинам. Дані отримуються за допомогою Flow, тому сервіс не затримує та не перериває головний потік застосунку. Принцип не втручатися до потоку застосунку, що вирисовує графічний інтерфейс є дуже важливим, бо інакше застосунок буде працювати повільно.

4.6 Навігація у застосунку

Навігація у мобільних застосунках на Android завжди була не дуже зручною річчю. Розробнику потрібно переходити з одного логічного компонента, що представляє екран застосунку, на інший, при цьому історія переходів повинна зберігатись для того щоб користувач мав змогу повернутися назад будь-коли. Також потрібно враховувати, що дуже часто потрібно переходити на інший екран та передавати якісь дані, наприклад наш застосунок передає ідентифікатор секції при переході з головного екрану на детальний екран.

Раніше така навігація здійснювалась за допомогою за допомогою досить складних елементів логічних компонентів, часто проблемою було те, що зберігати історію переходів або модифікувати її ставало надто складно.

Тому сучасним стандартом, який ми використовуємо у цьому застосунку є Navigation component. Ця бібліотека дозволяє описувати графи для навігації по застосунку за допомогою або графічного інтерфейсу у середовищі розробки, або з використанням простого xml коду.

Бібліотека є досить гнучкою та дозволяє глибоко модифікувати поведінку переходів на інші екрани. Працює вона на основі генерації методів для компонентів, що представляють екран застосунку. Кожен метод відповідає за свій перехід на інші екрани. За допомогою тої ж генерації можна створювати безпечні для використання аргументи, які можна передавати з одного екрану в інший. Так як код генерується під час компіляції бібліотека завжди знає який вид даних ми передаємо.

Таким чином, використання бібліотеки зводиться до створення xml файлу, який описує граф та виклику декількох методів.

Висновки до розділу 4

У цьому розділі дипломної роботи була описана програмна реалізація застосунку. Була приділена особлива увага архітектурі застосунку та його структурі. Також були більш детально описані деякі інструменти розробки застосунку. Були описані проблеми, які виникали під час розробки та наведені їх рішення. Ще були проаналізовані особливості операційної системи Android та те, як вони впливали на розробку застосунка.

5 ВИКОРИСТАННЯ МОБІЛЬНОГО ЗАСТОСУНКУ

Відкриваючи мобільний застосунок користувач відразу попадає на головний екран, який має в собі список із секцій теплиці, які створюються користувачем за допомогою кнопки зі знаком плюс у нижній частині екрану. Кожний елемент списку тримає в собі інформацію про поточний стан секції з усіма даними потрібними для цього, а також показує останній час оновлення даних. Також на кожному елементі є кнопка у вигляді баку, що дозволяє видаляти елементи зі списку.

Сам список та дані в ньому оновлюються автоматично із заданим інтервалом, при цьому прогрес прокручування залишається на таким же. На рисунку 5.1 можна побачити головний екран.

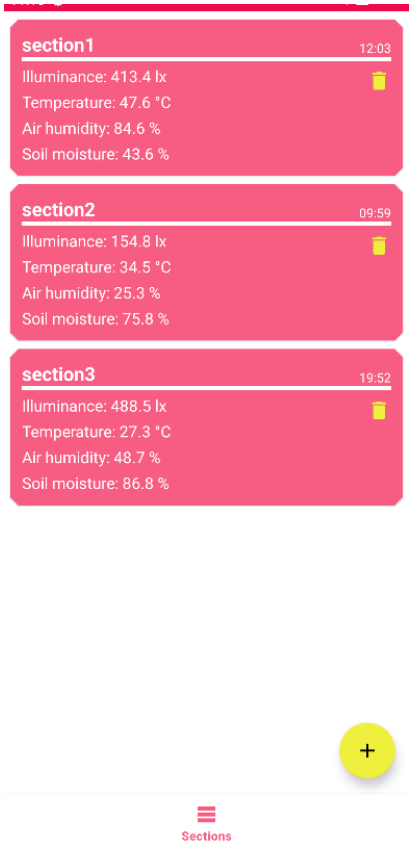

Рисунок 5.1 – Головний екран застосунку

Натискання на один із елементів списку відкриває екран із детальною інформацією про секцію теплиці. Але спочатку розглянемо екран для додавання

нового елементу до списку, що відкривається натисканням на кнопку зі знаком плюс.

Відкривається екран додавання нової секції, тут можна побачити два текстових поля, одне з них пропонує користувача ввести назву для секції, а інше ідентифікатор секції, якщо введені дані не відповідають вимогам програми або якщо поля залишаються порожніми, то застосунок показує користувачу його помилки у вигляді підписів під текстовими полями. При цьому, до поки дані введені неправильно кнопка підтвердження створення у нижній частині екрану залишається не активною.

Далі, у верхній частині екрану можна побачити кнопку навігації, що при натисненні поверне користувача назад на головний екран. Сам екран додавання секції можна побачити на рисунку 5.2.

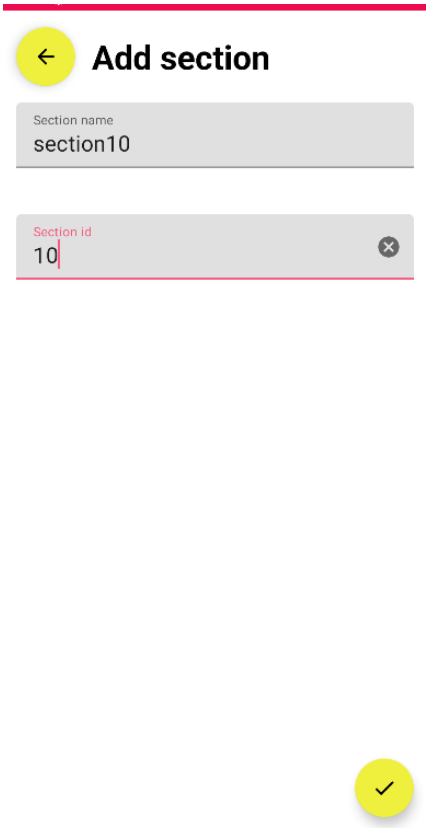

Рисунок 5.2 – Екран додавання секції

Роздивимось екран із детальною інформацією про секцію. На ньому можна побачити кнопку навігації для повернення назад, поруч з нею заголовок із назвою

секції. Далі знаходиться елемент із усією інформацією про секцію, подібний до елементів списку, він також автоматично оновлює інформацію.

Зупинимося на чотирьох кнопках овальної форми, по перше якщо вони не влізають в один ряд, його можна прокручувати. Далі, максимум одна з цих кнопок може бути активна, якщо була натиснута одна з кнопок, то вона відкриває графік нижче. При повторному натисканні, кнопка стає неактивною і графік знову пропадає. Сам графік по осі x показує час коли був знятий показник, що відображається на осі y, показник буде однойменним до натиснутої кнопки. Графік відображає інформацію на проміжку останніх 24 годин. Подивимось на екран з детальною інформацією на рисунку 5.3.

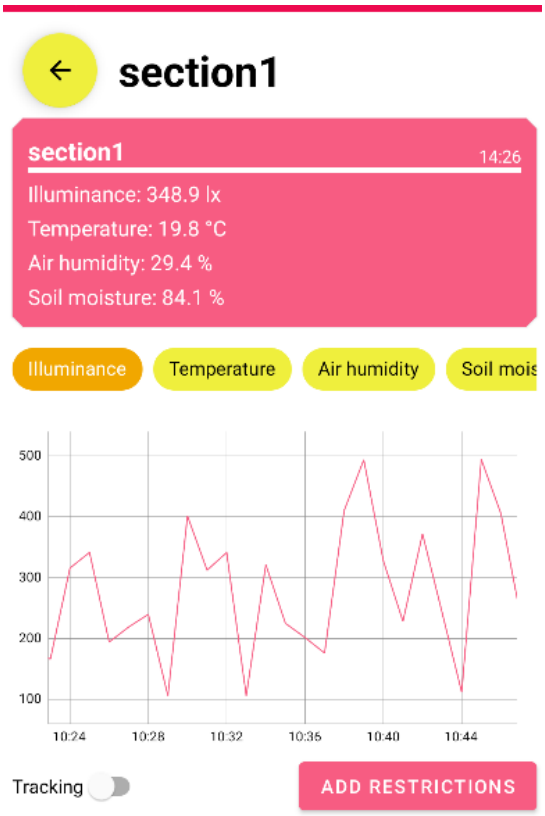

Рисунок 5.3 – Екран детальної інформації

Далі на екрані є перемикач, при його включені на телефоні буде запущене постійне, так зване висяче повідомлення, яке не можна відключати.

Відключається воно тільки вимкненням перемикача, саме повідомлення відображає всю інформацію про вибраний сектор теплиці, так само як і всюди дані

також автоматично оновлюються з певним інтервалом. Таке повідомлення не пропадає навіть при повному закритті застосунку, що дозволяє користувачу перевіряти інформацію у будь-який момент часу без запуску застосунку. Саме повідомлення можна побачити на рисунку 5.4.

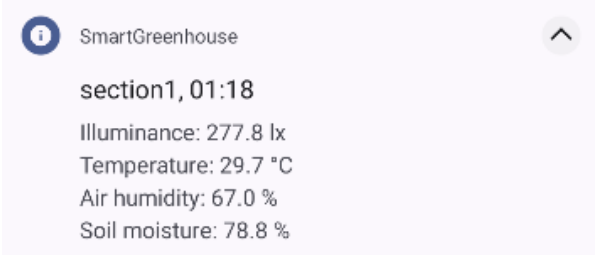

Рисунок 5.4 – Постійне повідомлення

Далі, натиснувши на кнопку ADD RESTRICTIONS користувач може перейти на екран із чотирма текстовими полями, кожне з яких відповідає за свої показник теплиці. На відміну від попередніх текстових полів ці не видають помилок, бо вони можуть бути заповнені лише натуральними числами. Тому підтвердити вибір за допомогою кнопки у нижній частині екрану можна навіть якщо деякі поля залишаються пустими. На рисунку 5.5 можна побачити екран з обмеженнями.

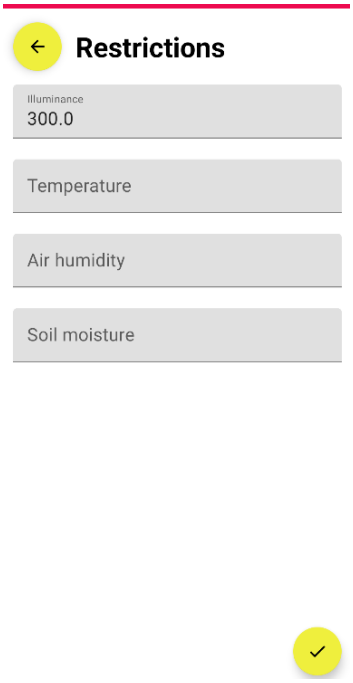

Рисунок 5.5 – Екран з обмеженнями

Таким чином, підтверджуючи числа, користувач встановлює критичні значення для потрібних йому показників, якщо поле залишається порожнім то критичне значення не встановлюється. Тоді, коли показники теплиці виходять за рамки критичних значень на телефон користувача приходить повідомлення із попередженням про цю подію. Повідомлення приходить навіть якщо застосунок був повністю закритий.

Побачити повідомлення про критичні показники теплиці можна на рисунку 5.6.

Рисунок 5.6 – Повідомлення про критичні показники

На рисунку вище можна побачити повідомлення про критичне значення одного показника, але якщо наприклад усі чотири показника вийдуть за межі одночасно, то повідомлення матиме інформацію про кожен із них.

Далі роздивимось декілька особливостей графіків у нашому застосунку. Варто сказати, що кожна шкала графіка може бути розширена чи навпаки зменшена. Під цим мається на увазі те, що наприклад шкала по осі x може бути зменшена з відображення двадцяти чотирьох годин до однієї години. При цьому зміна однієї шкали ніяк не впливає на іншу.

Також для збільшення видимості графіка з його стандартного виду були прибрані великі точки та підписи, які вказують значення. Це було зроблено тому, що графік може мати до тисячі чотирьохсот сорока точок одночасно, тому такі елементи просто зроблять неможливим нормальний його перегляд та аналіз.

Останнє, що варто зазначати це меню навігації, яке знаходиться у найнижчій частині екрану, там є лише один елемент при натисненні котрого користувач завжди буде відкривати головний екран застосунку.

Висновки до розділу 5

У цьому розділі роботи був описаний досвід використовування розробленого мобільного застосунку. Був проведений аналіз кожного екрану та його елементів, описані функції які вони виконують. Було показано як саме працює система повідомлень і як користувач може її налаштувати. Також була приділена увага візуалізації даних за допомогою графіків і те, як їх можна використовувати у застосунку.

ВИСНОВКИ

У ході цієї роботи було розроблено мобільний застосунок для візуалізації даних Smart-теплиці. Основною метою цієї роботи було створення якісного та зручного застосунку для того щоб користувачі були проінформовані про стан своєї теплиці у будь-який момент часу.

Було розглянуто теоретичні аспекти цієї предметної області та вирішено яким чином з нею працювати, також був проведений аналіз існуючих застосунків, які виконують подібні функції та проведена оцінка їх функціоналу та можливостей, деякі ідеї були перейняті та використанні у цьому проекті.

У ході роботи над системою було проведено аналіз інструментів, які існують для розробки мобільних застосунків. Був проведений вибір найбільш підходящих з них, тих які повністю відповідають сучасним стандартам та вимогам нашого застосунку.

Також у результаті розробки мобільного застосунку було зроблено зручний для користувачів інтерфейс та потрібний для предметної області функціонал. Частина функціоналу є досить унікальною і дозволить користувачу використовувати застосунок більш ефективно. Розроблена система є легко розширюваною, тобто вона завжди може бути оновлена з метою додавання нового функціоналу.

Таким чином мета дипломної роботи була досягнута, результати роботи показують актуальність розробленого застосунку та його практичне застосування. При необхідності застосунок легко може доповнюватись у відповідь на запити користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Android documentation. URL: https://developer.android.com/docs (дата звернення: 14.05.2023).

2. Kotlin documentation. URL: https://kotlinlang.org/docs/home.html (дата звернення: 14.05.2023).

3. Kotlin coroutines. URL: https://developer.android.com/kotlin/coroutines (дата звернення: 14.05.2023).

4. Kotlin flows on android. URL: https://developer.android.com/kotlin/flow (дата звернення: 14.05.2023).

5. Local database using Room. URL: https://developer.android.com/training/data-storage/room (дата звернення: 14.05.2023).

6. Koin dependency injection. URL: https://insert-koin.io/ (дата звернення: 14.05.2023).

7. Material design documentation. URL: https://m3.material.io/components (дата звернення: 14.05.2023).

8. Git documentation. URL: https://git-scm.com/doc (дата звернення: 14.05.2023).

9. Dawn Griffiths, David Griffiths Head First Kotlin: A Brain-Friendly Guide : O'Reilly Media, 2019, 480 с.

10. Jetpack Compose. URL: https://developer.android.com/jetpack/compose (дата звернення: 14.05.2023).

ДОДАТОК А

Мобільний застосунок візуалізації даних Smart-теплиці

Програмний код

Аркушів 10

2023

```
import android.app.Notification
import android.app.NotificationChannel
import android.app.NotificationManager
import android.app.Service
import android.content.Intent
import android.os.IBinder
import com.smart.greenhouse.R
import com.smart.greenhouse.data.repository.SectionsInfoRepository
import com.smart.greenhouse.util.convertNumberTimeToString
import com.smart.greenhouse.util.log
import kotlinx.coroutines.*
import org.koin.android.ext.android.inject

class SectionForegroundService : Service() {

    private val repository: SectionsInfoRepository by inject()

    private val job = SupervisorJob()
    private val scope = CoroutineScope(Dispatchers.IO + job)

    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        if (intent?.action == START_ACTION) {
            job.cancelChildren()

            val notificationChannel = NotificationChannel(CHANNEL_ID, CHANNEL_ID,
NotificationManager.IMPORTANCE_LOW)
            val restrictionNotificationChannel = NotificationChannel(RESTRICTION_CHANNEL_ID,
RESTRICTION_CHANNEL_ID, NotificationManager.IMPORTANCE_DEFAULT)
            val notificationManager = getSystemService(NotificationManager::class.java).apply {
                createNotificationChannel(notificationChannel)
                createNotificationChannel(restrictionNotificationChannel)
            }
            val defaultNotification = Notification.Builder(this@SectionForegroundService, CHANNEL_ID)
                .setSmallIcon(R.drawable.ic_info_black_24dp)
                .setContentTitle("Section name, time")
                .setStyle(Notification.BigTextStyle().bigText("Section info"))
                .build()
            startForeground(NOTIFICATION_ID, defaultNotification)

            scope.launch {
                val trackedSectionId = repository.getTrackedSectionId()!!
                repository.getLatestById(trackedSectionId).collect { sectionInfo ->
                    val contentString = StringBuilder().apply {
                        append(String.format(getString(R.string.illuminance), sectionInfo.illuminance) + "\n")
                        append(String.format(getString(R.string.temperature), sectionInfo.temperature) + "\n")
                        append(String.format(getString(R.string.air_humidity), sectionInfo.airHumidity, "%") + "\n")
                        append(String.format(getString(R.string.soil_moisture), sectionInfo.soilMoisture, "%") + "\n")
                    }.toString()
                    val notification = Notification.Builder(this@SectionForegroundService, CHANNEL_ID)
                        .setSmallIcon(R.drawable.ic_info_black_24dp)
                        .setContentTitle("${sectionInfo.name}, ${convertNumberTimeToString(sectionInfo.time)}")
                        .setStyle(Notification.BigTextStyle().bigText(contentString))
                        .setOngoing(true)
                        .build()
                    notificationManager.notify(NOTIFICATION_ID, notification)
                    log("trackedSectionId = $trackedSectionId")

                    val restriction = repository.getRestrictionById(trackedSectionId)
                    if (restriction != null) {
                        val restrictionContentString = StringBuilder().apply {
```

```
                    if (restriction.illuminance != null && restriction.illuminance < sectionInfo.illuminance)
                        append(String.format(getString(R.string.illuminance), sectionInfo.illuminance) + " is critically
high!\n")
                    if (restriction.temperature != null && restriction.temperature < sectionInfo.temperature)
                        append(String.format(getString(R.string.temperature), sectionInfo.temperature) + " is critically
high!\n")
                    if (restriction.airHumidity != null && restriction.airHumidity < sectionInfo.airHumidity)
                        append(String.format(getString(R.string.air_humidity), sectionInfo.airHumidity, "%") + " is critically
high!\n")
                    if (restriction.soilMoisture != null && restriction.soilMoisture < sectionInfo.soilMoisture)
                        append(String.format(getString(R.string.soil_moisture), sectionInfo.soilMoisture, "%") + " is critically
high!\n")
                }.toString()
                if (restrictionContentString.isNotEmpty()) {
                    val restrictionNotification = Notification.Builder(this@SectionForegroundService,
RESTRICTION_CHANNEL_ID)
                        .setSmallIcon(R.drawable.ic_error_black_24dp)
                        .setContentTitle("Attention! ${sectionInfo.name},
${convertNumberTimeToString(sectionInfo.time)}")
                        .setStyle(Notification.BigTextStyle().bigText(restrictionContentString))
                        .build()
                    notificationManager.notify(RESTRICTION_NOTIFICATION_ID, restrictionNotification)
                }
            }
        }
    } else if (intent?.action == STOP_ACTION) {
        stopForeground(STOP_FOREGROUND_REMOVE)
        stopSelf()
    }

    return super.onStartCommand(intent, flags, startId)
}

override fun onBind(intent: Intent?): IBinder? {
    return null
}

override fun onDestroy() {
    super.onDestroy()
    job.cancel()
}

companion object {
    private const val CHANNEL_ID: String = "SECTION_FOREGROUND_SERVICE"
    private const val RESTRICTION_CHANNEL_ID: String = "RESTRICTION_CHANNEL_ID"
    private const val NOTIFICATION_ID: Int = 1
    private const val RESTRICTION_NOTIFICATION_ID: Int = 2

    const val START_ACTION = "START_ACTION"
    const val STOP_ACTION = "STOP_ACTION"
}
}

import android.os.Bundle
import android.view.View
import androidx.navigation.fragment.findNavController
import androidx.navigation.fragment.navArgs
import com.github.mikephil.charting.components.XAxis
import com.github.mikephil.charting.data.LineData
```

```
import com.github.mikephil.charting.data.LineDataSet
import com.smart.greenhouse.R
import com.smart.greenhouse.databinding.FragmentSectionBinding
import com.smart.greenhouse.services.SectionForegroundService
import com.smart.greenhouse.ui.MainActivity
import com.smart.greenhouse.util.ViewBindingFragment
import com.smart.greenhouse.util.convertNumberTimeToString
import org.koin.androidx.viewmodel.ext.android.viewModel

class SectionFragment : ViewBindingFragment<FragmentSectionBinding>(FragmentSectionBinding::inflate) {

    private val vm: SectionViewModel by viewModel()

    private val args: SectionFragmentArgs by navArgs()

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

        bnd.itemSectionDetail.btnDelete.visibility = View.GONE
        vm.getSectionInfoById(args.sectionInfoId)
        configureObservers()
        configureLineChart()
        configureListeners()
        configureSwitch()
    }

    private fun configureSwitch() {
        if (args.sectionInfoId == vm.getTrackedSectionId()) bnd.switchService.isChecked = true
        bnd.switchService.setOnCheckedChangeListener { _, isChecked ->
            val activity = (requireActivity() as MainActivity)
            if (isChecked) {
                vm.setTrackedSectionId(args.sectionInfoId)
                activity.startSectionForegroundService(SectionForegroundService.START_ACTION)
            } else {
                vm.setTrackedSectionId(null)
                if (activity.isSectionForegroundServiceRunning()) {
                    activity.startSectionForegroundService(SectionForegroundService.STOP_ACTION)
                }
            }
        }
    }

    private fun configureListeners() {
        bnd.btnBackSection.setOnClickListener {
            requireActivity().onBackPressedDispatcher.onBackPressed()
        }
        bnd.btnAddRestrictions.setOnClickListener {

            val action = SectionFragmentDirections.actionNavigationSectionToRestrictionsFragment(args.sectionInfoId)
            findNavController().navigate(action)
        }
        bnd.chipGroupGraphs.setOnCheckedStateChangeListener { _, checkedIds ->
            when {
                checkedIds.isEmpty() -> {
                    bnd.chartLine.visibility = View.GONE
                }
                checkedIds[0] == bnd.chipIlluminance.id -> {
                    vm.getIlluminanceGraphInfoById(args.sectionInfoId, 1440)
                }
                checkedIds[0] == bnd.chipTemperature.id -> {
```

```
                vm.getTemperatureGraphInfoById(args.sectionInfoId, 1440)
            }
            checkedIds[0] == bnd.chipAirHumidity.id -> {
                vm.getAirHumidityGraphInfoById(args.sectionInfoId, 1440)
            }
            checkedIds[0] == bnd.chipSoilMoisture.id -> {
                vm.getSoilMoistureGraphInfoById(args.sectionInfoId, 1440)
            }
        }
    }
}

private fun configureLineChart() {
    bnd.chartLine.apply {
        axisRight.isEnabled = false
        description = null
        legend.isEnabled = false
        isDoubleTapToZoomEnabled = false
        isHighlightPerDragEnabled = false
        isHighlightPerTapEnabled = false

        xAxis.position = XAxis.XAxisPosition.BOTTOM
        xAxis.valueFormatter = TimeAxisFormatter()
        xAxis.granularity = 1f
        axisLeft.granularity = 0.1f
    }
}

private fun configureObservers() {
    vm.sectionInfo.observe(viewLifecycleOwner) { sectionInfo ->
        bnd.textTitleSectionName.text = sectionInfo.name
        bnd.itemSectionDetail.apply {
            textSection.text = sectionInfo.name
            textLastUpdateTime.text = convertNumberTimeToString(sectionInfo.time)
            textIlluminance.text = String.format(getString(R.string.illuminance), sectionInfo.illuminance)
            textTemperature.text = String.format(getString(R.string.temperature), sectionInfo.temperature)
            textAirHumidity.text = String.format(getString(R.string.air_humidity), sectionInfo.airHumidity, "%")
            textSoilMoisture.text = String.format(getString(R.string.soil_moisture), sectionInfo.soilMoisture, "%")
        }
    }
    vm.graphInfo.observe(viewLifecycleOwner) { entries ->
        val lineDataSet = LineDataSet(entries, "CHART_LINE").apply {
            setDrawValues(false)
            setDrawCircles(false)

            color = resources.getColor(R.color.pink_300, null)
        }
        bnd.chartLine.data = LineData(lineDataSet)
        bnd.chartLine.invalidate()
        bnd.chartLine.visibility = View.VISIBLE
    }
}
}

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.github.mikephil.charting.data.Entry
import com.smart.greenhouse.data.repository.SectionsInfoRepository
```

```
import com.smart.greenhouse.ui.data.SectionInfoUI
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class SectionViewModel(
    private val repository: SectionsInfoRepository,
) : ViewModel() {

    private val _sectionInfo = MutableLiveData<SectionInfoUI>()
    val sectionInfo: LiveData<SectionInfoUI> = _sectionInfo

    private val _graphInfo = MutableLiveData<List<Entry>>()
    val graphInfo: LiveData<List<Entry>> = _graphInfo

    fun getSectionInfoById(id: Long) {
        viewModelScope.launch(Dispatchers.IO) {
            repository.getLatestById(id).collect {
                _sectionInfo.postValue(it)
            }
        }
    }

    fun getTrackedSectionId(): Long? = repository.getTrackedSectionId()

    fun setTrackedSectionId(id: Long?) {
        repository.setTrackedSectionId(id)
    }

    fun getIlluminanceGraphInfoById(id: Long, amount: Int) {
        getGraphInfoById(id, amount) { Entry(it.time.toFloat(), it.illuminance.toFloat()) }
    }

    fun getTemperatureGraphInfoById(id: Long, amount: Int) {
        getGraphInfoById(id, amount) { Entry(it.time.toFloat(), it.temperature.toFloat()) }
    }

    fun getAirHumidityGraphInfoById(id: Long, amount: Int) {
        getGraphInfoById(id, amount) { Entry(it.time.toFloat(), it.airHumidity.toFloat()) }
    }

    fun getSoilMoistureGraphInfoById(id: Long, amount: Int) {
        getGraphInfoById(id, amount) { Entry(it.time.toFloat(), it.soilMoisture.toFloat()) }
    }

    private fun getGraphInfoById(id: Long, amount: Int, mapper: (SectionInfoUI) -> (Entry)) {
        viewModelScope.launch(Dispatchers.IO) {
            val entries = repository.getById(id, amount).map(mapper)
            _graphInfo.postValue(entries)
        }
    }
}

import com.smart.greenhouse.data.local.AppPreferences
import com.smart.greenhouse.data.local.SectionInfoLocal
import com.smart.greenhouse.data.local.SectionInfoLocalDao
import com.smart.greenhouse.data.remote.SectionInfoRemoteDataSource
import com.smart.greenhouse.ui.data.Restriction
import com.smart.greenhouse.ui.data.SectionInfoUI
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.Flow
```

```
import kotlinx.coroutines.flow.combine
import kotlinx.coroutines.flow.flow

class SectionsInfoRepositoryImpl(
    private val sectionInfoLocalDao: SectionInfoLocalDao,
    private val sectionInfoRemoteDataSource: SectionInfoRemoteDataSource,
    private val appPreferences: AppPreferences,
) : SectionsInfoRepository {

    override fun getAllLatest(): Flow<List<SectionInfoUI>> {
        val localFlow = sectionInfoLocalDao.getAll()
        val remoteFlow = flow {
            while (true) {
                emit(sectionInfoRemoteDataSource.getAllLatest())
                delay(10_000L)
            }
        }
        return localFlow.combine(remoteFlow) { local, remote ->
            val ui = mutableListOf<SectionInfoUI>()
            local.forEach { sectionInfoLocal ->
                val sectionInfoRemote = remote.find { sectionInfoLocal.id == it.id }
                if (sectionInfoRemote == null) {
                    ui.add(
                        SectionInfoUI(
                            id = sectionInfoLocal.id,
                            name = sectionInfoLocal.name,
                            time = 0,
                            illuminance = 0.0,
                            temperature = 0.0,
                            airHumidity = 0.0,
                            soilMoisture = 0.0
                        )
                    )
                    return@forEach
                }
                ui.add(
                    SectionInfoUI(
                        id = sectionInfoLocal.id,
                        name = sectionInfoLocal.name,
                        time = sectionInfoRemote.time,
                        illuminance = sectionInfoRemote.illuminance,
                        temperature = sectionInfoRemote.temperature,
                        airHumidity = sectionInfoRemote.airHumidity,
                        soilMoisture = sectionInfoRemote.soilMoisture
                    )
                )
            }
            return@combine ui
        }
    }

    override fun getLatestById(id: Long): Flow<SectionInfoUI> = flow {
        val local = sectionInfoLocalDao.getById(id)
        while (true) {
            val remote = sectionInfoRemoteDataSource.getLatestById(id)
            if (remote == null) {
                emit(
                    SectionInfoUI(
                        id = local.id,
                        name = local.name,
```

```
                    time = 0,
                    illuminance = 0.0,
                    temperature = 0.0,
                    airHumidity = 0.0,
                    soilMoisture = 0.0
                )
            )
        } else {
            emit(
                SectionInfoUI(
                    id = local.id,
                    name = local.name,
                    time = remote.time,
                    illuminance = remote.illuminance,
                    temperature = remote.temperature,
                    airHumidity = remote.airHumidity,
                    soilMoisture = remote.soilMoisture
                )
            )
        }
        delay(10_000L)
    }
}

override fun getTrackedSectionId(): Long? = appPreferences.trackedSectionId

override fun setTrackedSectionId(id: Long?) {
    appPreferences.trackedSectionId = id
}

override fun getRestrictionById(id: Long): Restriction? = appPreferences.getRestrictions().find { it.id == id }


override fun setRestrictions(restriction: Restriction) {
    appPreferences.setRestrictions(restriction)
}

override suspend fun getById(id: Long, amount: Int): List<SectionInfoUI> {
    val local = sectionInfoLocalDao.getById(id)
    return sectionInfoRemoteDataSource.getById(id, amount).map {
        SectionInfoUI(
            id = local.id,
            name = local.name,
            time = it.time,
            illuminance = it.illuminance,
            temperature = it.temperature,
            airHumidity = it.airHumidity,
            soilMoisture = it.soilMoisture
        )
    }
}

override suspend fun insertAll(sectionsInfoLocal: List<SectionInfoLocal>) =
    sectionInfoLocalDao.insertAll(sectionsInfoLocal)

override suspend fun deleteAll(sectionsInfoLocal: List<SectionInfoLocal>) =
    sectionInfoLocalDao.deleteAll(sectionsInfoLocal)
}

import android.os.Bundle
```

```
import android.view.View
import androidx.core.text.isDigitsOnly
import androidx.core.widget.doOnTextChanged
import com.smart.greenhouse.data.local.SectionInfoLocal
import com.smart.greenhouse.databinding.FragmentAddSectionBinding
import com.smart.greenhouse.util.ViewBindingFragment
import org.koin.androidx.viewmodel.ext.android.viewModel

class AddSectionFragment : ViewBindingFragment<FragmentAddSectionBinding>(FragmentAddSectionBinding::inflate)
{

    private val vm: AddSectionViewModel by viewModel()

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

        configureListeners()
    }

    private fun configureListeners() {
        bnd.apply {
            btnConfirmSection.isEnabled = false
            textFieldSectionName.error = "Must not be empty"
            textFieldSectionId.error = "Must not be empty"

            textFieldSectionName.editText!!.doOnTextChanged { _, _, _, _ -> textChanged() }
            textFieldSectionId.editText!!.doOnTextChanged { _, _, _, _ -> textChanged() }

            btnBackAddSection.setOnClickListener {
                requireActivity().onBackPressedDispatcher.onBackPressed()
            }
            btnConfirmSection.setOnClickListener {
                val sectionInfo = SectionInfoLocal(
                    id = bnd.textFieldSectionId.editText!!.text.toString().toLong(),
                    name = bnd.textFieldSectionName.editText!!.text.toString().trim()
                )
                vm.insertSectionInfo(sectionInfo)
                requireActivity().onBackPressedDispatcher.onBackPressed()
            }
        }
    }

    private fun textChanged() {
        val name = bnd.textFieldSectionName.editText!!.text.toString()
        val id = bnd.textFieldSectionId.editText!!.text.toString()
        if (name.isBlank()) {
            bnd.btnConfirmSection.isEnabled = false
            bnd.textFieldSectionName.error = "Must not be empty"
        } else {
            bnd.textFieldSectionName.error = null
        }
        if (id.isEmpty()) {
            bnd.btnConfirmSection.isEnabled = false
            bnd.textFieldSectionId.error = "Must not be empty"
            return
        } else if (!id.isDigitsOnly()) {
            bnd.btnConfirmSection.isEnabled = false
            bnd.textFieldSectionId.error = "Must be digits only"
            return
        } else {
```

```
            bnd.textFieldSectionId.error = null
        }
        if (name.isNotBlank()) {
            bnd.btnConfirmSection.isEnabled = true
        }
    }
}

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.smart.greenhouse.data.local.SectionInfoLocal
import com.smart.greenhouse.data.repository.SectionsInfoRepository
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

import android.app.ActivityManager
import android.content.Context
import android.content.Intent
import android.os.Bundle
import androidx.navigation.fragment.NavHostFragment
import androidx.navigation.ui.setupWithNavController
import com.smart.greenhouse.R
import com.smart.greenhouse.databinding.ActivityMainBinding
import com.smart.greenhouse.services.SectionForegroundService
import com.smart.greenhouse.util.ViewBindingActivity

class MainActivity : ViewBindingActivity<ActivityMainBinding>(ActivityMainBinding::inflate) {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val navHostFragment = supportFragmentManager.findFragmentById(R.id.nav_host_fragment_activity_main) as
NavHostFragment
        bnd.navView.setupWithNavController(navHostFragment.navController)
        bnd.navView.setOnItemReselectedListener {
            if (it.itemId == R.id.navigation_sections) {
                navHostFragment.navController.popBackStack(it.itemId, inclusive = false)
            }
        }
    }

    fun startSectionForegroundService(intentAction: String) {
        val serviceIntent = Intent(this, SectionForegroundService::class.java).apply {
            action = intentAction
        }
        startForegroundService(serviceIntent)
    }

    fun isSectionForegroundServiceRunning(): Boolean {
        return (getSystemService(Context.ACTIVITY_SERVICE) as ActivityManager)
            .getRunningServices(Int.MAX_VALUE)
            .any { it.service.className == SectionForegroundService::class.qualifiedName }
    }
}
```