МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук та технологій
Кафедра комп'ютерних систем та мереж

Пояснювальна записка

до магістерської роботи
магістра
(ступінь вищої освіти)

на тему **КОМП'ЮТЕРНА СИСТЕМА АВТОМАТИЗАЦІЇ ОБЛІКУ ФАРМАЦЕВТИЧНОЇ ПРОДУКЦІЇ НА БАЗІ ANDROID**

Виконав: студент 2 курсу, групи КНТ-613м
спеціальності
123 «Комп'ютерна інженерія»
(код і найменування спеціальності)
Освітня програма (спеціалізація)
«Спеціалізовані комп'ютерні системи»
СЛОНЄВ М. А.
(ПРІЗВИЩЕ та ініціали)
Керівник СКРУПСЬКИЙ С. Ю.
(ПРІЗВИЩЕ та ініціали)
Рецензент СТЕПАНЕНКО О.О.
(ПРІЗВИЩЕ та ініціали)

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій

Кафедра «Комп'ютерні системи та мережі»

Ступінь вищої освіти магістерський

Спеціальність 123 Комп'ютерна інженерія

(код і найменування)

Освітня програма (спеціалізація) «Спеціалізовані комп'ютерні системи»

(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Зав. кафедри Кудерметов Р.К.

_____

"____" _____ 2024 року

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТА

СЛОНЄВ Максим Андрійович

(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Комп'ютерна система автоматизації обліку фармацевтичної продукції на базі Android.

керівник проєкту (роботи) к. т. н., доцент, СКРУПСЬКИЙ Степан Юрійович

(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом вищого навчального закладу від "18" жовтня 2024 року № 149

2. Строк подання студентом проєкту (роботи) 10 грудня 2024 року

3. Вихідні дані до проєкту (роботи) аналіз сучасних WMS-систем, Android, Kotlin, UML-моделювання, Jetpack Compose, Spring Boot, із фокусом на мобільні пристрої, звіти та складські дані.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) Аналіз існуючих рішень і технологій;
2) Розробка розподіленої системи;
3) Дослідження розподіленої системи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-3	СКРУПСЬКИЙ С.Ю., доцент		
нормоконтроль	ПОЛЬСЬКА О.В., ст. викл.		

7. Дата видачі завдання 01.10.2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз наявних методів та алгоритмів автоматизації обліку фармацевтичної продукції, перегляд отриманих результатів, виділення недоліків та переваг.	20.10.2024 р.	
2	Встановлення та налаштування програмного забезпечення	22.10.2024 р.	
3	Аналіз технологій Kotlin	25.10.2024 р.	
4	Розробка програмних компонентів системи	12.11.2024 р.	
5	Дослідження розробленої системи	24.11.2024 р.	
6	Оформлення отриманих результатів у ПЗ	02.12.2024 р.	
7	Оформлення графічного матеріалу	06.12.2024 р.	
8	Оформлення допоміжного матеріалу	09.12.2024 р.	

Студент(ка) _____ **Максим СЛОНЄВ**
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ **Степан СКРУПСЬКИЙ**
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

ПЗ: 100 с., 13 рис., 3 табл., 11 лістингів., 27 джерел, 1 додаток.

ANDROID, KOTLIN, MVVM, WMS-СИСТЕМИ, PDF-ЗВІТИ, QR-КОДИ

Метою магістерської роботи є дослідження мобільної системи для автоматизації обліку фармацевтичної продукції на базі Android, що забезпечить поліпшення ефективності складських операцій та точність обліку.

Предмет дослідження – технології автоматизації обліку та складських операцій із застосуванням мобільних платформ.

Об'єкт дослідження – процеси управління складськими операціями у фармацевтичній галузі.

Перший розділ магістерської роботи був аналізу існуючих WMS-систем, таких як NetSuite, Fishbowl, Odoo, їхні переваги та недоліки. Окрему увагу приділено обґрунтуванню вибору Android як основної платформи для розробки мобільного додатка.

Другий розділ присвячений проєктуванню та розробка мобільного додатка. Використано мову програмування Kotlin та інструмент Jetpack Compose. Розглянуто застосування архітектурних шаблонів, зокрема MVVM, та асинхронних підходів із використанням Kotlin Coroutines.

Третій розділ присвячений тестуванню працездатності системи та порівнянню отриманих результатів.

У результаті виконання дипломної роботи було проведено детальний аналіз, проектування та дослідження розробленої комп'ютерної системи.

ABSTRACT

Explanatory note to the master`s work: 100 pages, 13 figures, 3 tables, 11 listings, 27 sources, 1 addition.

ANDROID, KOTLIN, MVVM, WMS SYSTEMS, PDF REPORTS, QR CODES

The aim of the master’s thesis is to research a mobile system for automating the inventory management of pharmaceutical products based on Android, which will improve the efficiency of warehouse operations and inventory accuracy.

The subject of the research is the technologies for automating inventory management and warehouse operations using mobile platforms.

The object of the research is the processes of managing warehouse operations in the pharmaceutical industry.

The first chapter of the thesis analyzes existing WMS systems such as NetSuite, Fishbowl, Odoo, their advantages, and disadvantages. Particular attention is given to the justification for choosing Android as the main platform for developing the mobile application.

The second chapter is dedicated to the design and development of the mobile application. The Kotlin programming language and Jetpack Compose are used. The application of architectural patterns, particularly MVVM, and asynchronous approaches using Kotlin Coroutines is discussed.

The third chapter focuses on testing the functionality of the system and comparing the results obtained.

As a result of the thesis work, a detailed analysis, design, and research of the developed computer system were conducted.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ОС	–	операційна система
СКБД	–	система керування базами даних
ФІМД	–	функціональність інтерфейсу мобільного додатку
API	–	Application Programming Interface, інтерфейс програмування застосунків
CD	–	Continuous Deployment, безперервне розгортання
CI	–	Continuous Integration, безперервна інтеграція
CPU	–	Central Processing Unit, центральний процесор
CRUD	–	Create, Read, Update, Delete, створення, читання, оновлення, видалення
DI	–	Dependency Injection, інверсія залежностей
EAN	–	European Article Number, європейський артикульний номер
HTML	–	Hypertext Markup Language, мова розмітки гіпертексту
HTTP	–	Hypertext Transfer Protocol, протокол передачі гіпертексту
ID	–	identifier, ідентифікатор
IDE	–	Integrated Development Environment, інтегроване середовище розробки
JSON	–	Javascript Object Notation, нотація об’єктів JavaScript
JWT	–	Json Web Token, веб-токен у форматі JSON
MVP	–	Model-View-Presenter, модель-презентер-представлення
MVC	–	Model-View-Controller, модель-вид-контролер
MVVM	–	Model-View-Viewmodel, модель-вид-модель представлення
NFC	–	Near Field Communication, близька безконтактна комунікація
OLED	–	Organic Light Emitting Diode, органічний світлодіод
PHP	–	Hypertext Preprocessor, препроцесор гіпертексту
PDF	–	Portable Document Format, портативний формат документів
QR	–	Quick Response, швидка відповідь

RAM – Random Access Memory, оперативна пам’ять

REST – Representational State Transfer, репрезентативна передача стану

RXJAVA – Reactive Extensions For The Java Vm, реактивні розширення для Java Virtual Machine

SDK – Software Development Kit, набір засобів розробки програмного забезпечення

TCP/IP – Transmission Control Protocol / Internet Protocol, протокол управління передачею / інтернет-протокол

UI – User Interface, інтерфейс користувача

UML – Unified Modeling Language, уніфікована мова моделювання

WMS – Warehouse Management System, система управління складом

XML – Extensible Markup Language, розширювана мова розмітки

SQL – Structured Query Language, мова структурованих запитів

SSD – Solid State Drive, твердотільний накопичувач

ВСТУП

Фармацевтична продукція відіграє одну з ключових ролей у забезпеченні здоров'я та благополуччя населення в сучасному світі. Сьогодні ліки є невід'ємною частиною життя кожної людини, а попит на якісні медикаменти зростає щороку. Лікування захворювань, профілактика інфекцій, підтримка здоров'я – усе це залежить від надійного та безперебійного постачання лікарських засобів. Умови зберігання, швидкість доставки та точність обліку мають вирішальне значення для безпеки пацієнтів і якості медичних послуг.

Фармацевтичні склади виконують важливу функцію у забезпеченні своєчасного постачання лікарських засобів в аптеки, лікарні та медичні заклади. Водночас зростає потреба в ефективному управлінні такими складськими операціями. Оскільки більшість препаратів потребує суворого дотримання умов зберігання – температурного режиму, вологості та належної обробки – автоматизація процесів є невід'ємною складовою сучасної фармацевтичної логістики. Системи автоматизації допомагають мінімізувати помилки при управлінні запасами та забезпечують збереження якості медикаментів на всіх етапах логістичного ланцюга.

Сьогодні ми живемо у часи, коли розповсюдженість обчислюваних пристроїв та мережі інтернет досягли свого піку. Інформаційні технології використовуються майже у всіх сферах життя людини, починаючи від особистого блогу, і закінчуючи складними системами автоматизації виробничих та бізнес-процесів. Зі зростанням обсягу інформації та потреби в прискоренні процесів, фармацевтичні склади стають більш залежними від сучасних технологій для забезпечення безперервного постачання медикаментів. Від ефективного обліку та управління запасами залежить здоров'я та життя пацієнтів, що робить автоматизацію критично важливим інструментом у цій галузі.

Згідно з дослідженнями, автоматизація складів може суттєво знизити операційні витрати та підвищити продуктивність персоналу. Автоматизація

складів здатна скоротити витрати до 40%, водночас підвищуючи ефективність роботи складу та зменшуючи час обробки замовлень на 50%. У випадку фармацевтичної продукції, особливі умови зберігання та швидкість обробки замовлень є надзвичайно важливими, і впровадження автоматизованих систем дозволяє досягти цих цілей [1].

Одним із ключових підходів до оптимізації процесів є створення невеликих складів поблизу кінцевих споживачів, що дозволяє значно знизити витрати на доставку та прискорити процес постачання. Цей підхід є надзвичайно актуальним для аптечних мереж, де швидкість доставки безпосередньо впливає на доступність життєво важливих препаратів для споживачів. Невеликі модульні склади дозволяють зменшити витрати на логістику та забезпечити оперативність доставки, підвищуючи конкурентоспроможність мережі аптек.

Водночас, фармацевтична продукція вимагає дотримання суворих умов зберігання - від температурного режиму до вологості. Тому модульні склади, обладнані відповідно до потреб фармацевтичної продукції, стають важливим інструментом для забезпечення якості товару на всіх етапах постачання. Автоматизовані системи управління складами дозволяють контролювати всі параметри зберігання в реальному часі, що забезпечує відповідність нормативним вимогам і мінімізує ризики втрат через недотримання умов зберігання.

Впровадження автоматизованих систем може знизити кількість людських помилок на складах до 70%, що особливо актуально для фармацевтичної продукції, де кожна помилка може мати серйозні наслідки для здоров’я пацієнтів. Крім того, автоматизація дозволяє скоротити витрати на персонал і підвищити ефективність обробки замовлень, що робить її інструментом вибору для сучасних фармацевтичних компаній [2].

У світі складського обліку для точності та швидкості обробки інформації важливу роль відіграє звітування з використанням QR-кодів. Звіти дозволяють контролювати наявність товарів, а QR-коди, у свою чергу, забезпечують швидкий доступ до детальної інформації про кожен товар, що мінімізує ризик помилок, спрощує інвентаризацію та підвищує ефективність роботи на складах.

Одним із ключових завдань при розробці систем автоматизації є впровадження механізму генерування PDF-звітів, що містять інформацію про продукцію та QR-коди для кожного товару. Операція з генерування великої кількості таких документів на мобільному пристрої може суттєво навантажити апаратні ресурси, такі як процесор чи пам'ять.

Важливою складовою частиною дослідження в рамках даної дипломної роботи буде оцінка ефективності та продуктивності комп’ютерної системи на базі мобільних пристроїв у вирішенні задач генерації електронних звітів. Будуть проведені експерименти на пристроях різного цінового сегменту з метою визначення оптимальних технічних характеристик для обробки заданої кількості товарів.

Таким чином, розробки комп'ютерної системи автоматизації обліку фармацевтичної продукції дозволить не лише підвищити ефективність управління запасами, але й забезпечить надійне зберігання продукції та своєчасну доставку.

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ І ТЕХНОЛОГІЙ

1.1 Поняття складського обліку

Без налагодженого складського обліку підприємство не може функціонувати. Склади забезпечують не лише зберігання продукції, а й безперебійний товарообіг, що потребує комплексного підходу до організації інфраструктури.

Складський облік — це процес контролю за запасами товарів, що охоплює прийом, зберігання, видачу та облік. Ефективне управління складом оптимізує запаси, знижує витрати на зберігання та забезпечує своєчасне постачання. Сучасні WMS-системи інтегрують складський облік з іншими бізнес-процесами, підвищуючи точність і швидкість обробки замовлень.У рамках автоматизованих систем управління складом ключовими компонентами є:

Tote – це контейнер для зберігання товарів, що використовується для оптимізації простору на складі, дозволяючи зберігати окремі одиниці продукції або невеликі партії, а також спрощувати процеси інвентаризації та відбору товарів [3].

Pallet – це піддон, на який розміщуються великі партії товарів для зберігання або транспортування у контейнерах. Палети значно спрощують обробку великих обсягів продукції, полегшуючи її переміщення складом і між об'єктами логістики.

Station – це робоча зона на складі, де здійснюється обробка товарів, зокрема прийом палет, консолідація або відбір продукції. Станції можуть бути налаштовані під різні процеси, наприклад, під прийом товарів або виконання замовлень [3].

1.2 Визначення завдання

Завданням дипломної роботи є дослідження мобільної системи автоматизації обліку фармацевтичної продукції на базі Android, що включає генерацію звітів у форматі PDF з QR-кодами для кожного продукту. У межах дослідження будуть

проводитися експерименти з метою оцінки продуктивності мобільних пристроїв при виконанні таких завдань, а також аналіз споживання ресурсів, часу обробки і мінімальних технічних вимог для забезпечення комфортного користування.

Для того, щоб досягти визначеної мети, розробник має вирішити такі завдання:

- аналіз існуючих рішень для управління складом;
- вибір мобільної операційної системи для розробки;
- аналіз існуючих рішень побудови архітектури розподіленої системи;
- аналіз існуючих мов програмування для створення мобільних додатків;
- аналіз сучасних фреймворків для розробки мобільних застосунків;
- аналіз технологій для серверної частини системи;
- визначення та аналіз вимог до програмного забезпечення;
- проектування архітектури системи;
- розробка модулів клієнтської частини;
- розробка модулів серверної частини;
- розробка компонентів інтерфейсу користувача;
- розробка системи генерації та сканування звітів з QR-кодами;
- дослідження комп’ютерної системи;
- дослідження процесу генерації звітів з QR-кодами на різних пристроях;
- випробування комп’ютерної системи.

1.3 Огляд існуючих рішень для управління складом

На сьогоднішній день на ринку представлені різні системи управління складом. Найпоширеніші з них це NetSuite WMS, Fishbowl Inventory та Odoo WMS. Ці системи автоматизують процеси зберігання та обліку, дозволяючи знижувати витрати на зберігання і підвищувати ефективність обробки замовлень.

1.3.1 NetSuite WMS

NetSuite WMS є одним з лідерів ринку завдяки своїй інтеграції з ERP-системами та підтримці хмарного зберігання даних. Система пропонує високий рівень автоматизації процесів, таких як управління запасами та прийом товарів. Проте, для малого бізнесу ця система може бути занадто дорогою через високу вартість впровадження та технічної підтримки [4].

1.3.2 Fishbowl Inventory

Fishbowl Inventory – це рішення для малого та середнього бізнесу, яке пропонує потужний функціонал за доступною ціною. Воно інтегрується з QuickBooks для управління фінансами та дозволяє легко масштабувати бізнес. Основний недолік Fishbowl - це складний інтерфейс, що може ускладнити навчання нових користувачів [5].

1.3.3 Odoo WMS

Odoo WMS – це модульна система, яка дозволяє налаштовувати функції відповідно до потреб компанії. Вона є більш доступною порівняно з іншими рішеннями та пропонує широкий набір інструментів для автоматизації складських процесів. Проте, система вимагає глибокого налаштування та технічних знань, що може бути складним для малого бізнесу [6].

Порівнюючи ці системи, можна побачити, що кожна з них має свої певні сильні сторони. NetSuite WMS відрізняється своєю інтеграцією з ERP та можливістю обробки великого обсягу даних, Fishbowl Inventory є більш доступним варіантом для малого бізнесу, а Odoo WMS пропонує налаштування під специфічні потреби компанії. Незважаючи на переваги, які вони надають, усі ці рішення мають певні недоліки, такі як висока вартість NetSuite, складність інтерфейсу у Fishbowl, або потреба в поглиблених технічних знаннях Odoo.

Пропонована система для управління складом на базі Android з використанням генерації PDF-звітів та QR-кодів на мобільних пристроях вирішує ці проблеми. По-перше, використання мобільного пристрою як основного інструменту для управління складом дозволяє знизити витрати на інфраструктуру та забезпечити зручність використання для малого бізнесу. По-друге, можливість

швидкої генерації звітів та інтеграції QR-кодів спрощує процеси обліку та зменшує потребу в технічних навичках. Крім того, мобільний додаток дозволяє працювати автономно та забезпечує високу гнучкість у масштабуванні бізнесу.

1.4 Актуальність розробки під ОС Android

Сучасні мобільні пристрої на Android відіграють важливу роль у бізнесі, зокрема для складського обліку, завдяки можливостям інтеграції, персоналізації та масштабованості. Android став не тільки популярною операційною системою для смартфонів, але й основою для різних типів пристроїв, серед яких є планшети та розумні годинники, що дозволяє його використовувати на різних платформах.

Окрім цього, одним із визначальних факторів популярності цієї системи є те, що для Android-розробників створено безліч документації, яка спрощує комунікації між фахівцями по всьому світу та дозволяє новачкам швидше опанувати навички розробки. Завдяки цьому щодня у Google Play Market з’являється безліч нових застосунків, які спрощують життя мільйонів людей.

Розробка додатка для управління складом на базі Android пропонує низку переваг. Мобільний доступ до системи обліку дозволяє керівникам складу швидко отримувати дані про запаси, здійснювати інвентаризацію, контролювати процеси прийому та видачі товарів, а також миттєво генерувати PDF-звіти з QR-кодами для зручної обробки документації. Це вирішує проблему обмеженої мобільності та складнощів із ручним введенням даних у традиційних WMS-системах, які зазвичай використовують десктопні версії.

Крім того, можливість інтеграції мобільного додатку з існуючими WMS дозволяє покращити точність та зменшити людські помилки, що виникають під час обробки товарів. QR-коди, генеровані додатком, спрощують ідентифікацію товарів та відстеження їх руху, що значно підвищує ефективність операцій на складі. Оскільки Android підтримується різними виробниками пристроїв, це забезпечує

широкі можливості для використання системи в різних бізнес-сценаріях і галузях, включаючи малі та середні підприємства.

Особливу увагу варто звернути на можливість масштабування систем з використанням спеціалізованих пристроїв, таких як сканери компанії Zebra, що працюють на ОС Android. Ці промислові пристрої спеціально розроблені для швидкого та надійного сканування штрих-кодів, що дозволяє автоматизувати процеси інвентаризації та управління запасами. Zebra сканери використовують лазерні датчики для зчитування штрих-кодів на великих відстанях або при поганому освітленні, а бездротові модулі Bluetooth та NFC забезпечують інтеграцію та швидке налаштування через з'єднання з іншими пристроями.

Отже, використання мобільного Android-додатку забезпечує не лише гнучкість і доступність системи для різних типів підприємств, але й надає можливості для швидкого впровадження і масштабування залежно від потреб складу.

1.5 Аналіз наявних підходів до створення архітектури розподіленої системи

У галузі інформаційних технологій для побудови масштабних і складних систем використовуються спеціальні рекомендації та стандарти, що отримали назву архітектури.

У галузі інформаційних технологій для побудови масштабних і складних систем використовуються спеціальні рекомендації та стандарти, що отримали назву архітектури.

Існує чимало підходів до створення складних систем із якісною архітектурою. Незважаючи на певні відмінності, ці підходи мають багато спільного. Зазвичай, вони пропонують методи поділу програми на окремі модулі.

У кожній системі обов’язково присутні модулі, що відповідають за бізнес-логіку програми, та модулі, які забезпечують відображення даних.

Архітектурні підходи мають рекомендаційний характер, описуючи, як найкраще організувати систему для її зручної підтримки та масштабування. У цьому проєкті буде розглядатися багаторівнева монолітна архітектура.

Монолітна архітектура означає, що всі компоненти програмного додатка зібрані в одну цілісну структуру, яка працює на одній платформі. Такий підхід описує однорівневий додаток, де всі функціональні частини інтегровані в єдину програму.

Основними компонентами монолітної архітектури є:

– авторизація – відповідає за перевірку та ідентифікацію користувачів;

– презентація – забезпечує обробку HTTP-запитів і формування відповідей у форматах HTML, JSON або XML (для веб-API);

– API – набір чітко визначених методів і протоколів для взаємодії програмного забезпечення, що дозволяє викликати певні служби через функції або процедури;

– бізнес-логіка – реалізує ключові правила й алгоритми роботи додатка відповідно до бізнес-вимог;

– рівень бази даних – містить об’єкти доступу до даних, відповідальні за зберігання, читання і модифікацію інформації в базах даних;

– інтеграція додатків – забезпечує взаємодію з іншими сервісами або джерелами даних.

Переваги монолітної архітектури:

– проста розробка – сучасні інструменти розробки та середовища IDE створені для підтримки монолітних додатків, що робить їхню розробку зручною;

– просте розгортання – додаток можна легко розгорнути, передавши один файл або ієрархію каталогів у відповідне середовище виконання;

– просте масштабування – забезпечується шляхом запуску декількох копій програми з використанням балансування навантаження.

Недоліки монолітної архітектури:

– велика монолітна база коду – значний обсяг коду може бути складним для новачків у команді, що уповільнює розуміння та розвиток програми;

– перевантажена IDE – при значному обсязі коду IDE працює повільніше, що знижує продуктивність розробників;

– перевантажений веб-контейнер – великі програми потребують більше часу для запуску, що уповільнює робочий процес;

– складність у частому розгортанні – оновлення навіть одного компонента потребує повного повторного розгортання програми;

– залежність від обраного технологічного стека – складно поступово впроваджувати нові технології, оскільки архітектура прив’язує додаток до початково вибраного стека.

Враховуючи всі плюси і мінуси, монолітна архітектура є найкращим вибором для малих і середніх проектів, про які йдеться в цій статті. Програмні продукти відносно невеликі за складністю, знаходяться на ранніх стадіях розробки і не вимагають використання мікросервісних архітектур, оскільки мають обмежену кількість функцій.

Водночас архітектура має бути спроєктована з урахуванням можливості майбутнього масштабування, аби уникнути втрат продуктивності системи. Це дозволить забезпечити ефективну адаптацію до зростання функціональних вимог у перспективі.

Майбутня розподілена система будескладатисяз 3 рівнів: клієнтської частини, сервера додатків і сервера бази даних.

Клієнтська частина-це рівень відображення, який відповідає за взаємодію з системою користувача і відображення інформації. Він надає графічний інтерфейс, за допомогою якого користувач може надіслати запит на сервер і використовувати функціональність системи. Для передачі даних використовується протокол HTTP з архітектурою REST, а формат передачі даних – JSON [7].

Сервер додатків – серверна частина системи, яка забезпечує обробку користувацьких запитів і взаємодію з базою даних. Він виконує основну бізнес-логіку та відповідає за коректну роботу системи.

Сервер баз даних-це рівень, на якому розташовані система керування базами даних і сама база даних.

База даних-це структурований набір даних, організований відповідно до концептуальної моделі, яка визначає її властивості та взаємозв'язки. База даних підтримує 1 або більше областей застосування [8].

СКБД – це апаратно-програмний комплекс,який дозволяє визначати, створювати, маніпулювати, контролювати, керувати та використовувати бази даних [8].

Запити до СКБД приймаються з сервера через з’єднання TCP/IP. СКБД виконує ці запити та повертає дані відповідно до них.

1.6 Аналіз технологій системи на стороні клієнта

Існує багато мов програмування, які дозволяють створювати високоякісні програми. Деякі з них мають вбудовану середу розробки, яка значно спрощує процес програмування, в той час як інші вимагають додаткових інструментів, але їх функції не настільки великі.

1.6.1 Java

Java – одна з найстаріших і відомих мов програмування для створення високоякісних програм. Це популярна мова для розробки додатків для Android, і в 2019 році вона увійшла до ТОП-5 найкращих та функціональних мов. Однак Java дуже складна і може бути не найкращим вибором для початківців. Вивчення програмування на Java вимагає знань об'єктно-орієнтованого програмування, оскільки мова Працює з класами, винятками та конструкторами. Важливо ретельно продумати логіку роботи програми.

Однією з головних переваг Java є наявність власного середовища розробки - Android Studio. У 2014 році Google визнала це середовище офіційним для розробки Android-додатків, що значно полегшило роботу розробників. Процес створення

програм спрощується завдяки візуальному UI-редактору, функції автодоповнення коду та багатьом іншим можливостям [9].

Що стосується функціональності, Java є однією з найпотужніших мов програмування. Вона дозволяє реалізовувати складні завдання та проекти, що пояснює її велику популярність.

1.6.2 Kotlin

Kotlin – відносно нова мова програмування, випущена в 2017 році. Лише за короткий проміжок часу він зміг завоювати популярність серед розробників, а в 2019 році був визнаний Google кращою мовою для Android і обігнав Java.

Kotlin поєднав у собі найкращі риси інших мов. Мова має безліч переваг, таких як автоматичне визначення типів даних, функції-розширення та інші. Вона проста для розуміння і освоєння, доступна кожному. Оскільки Kotlin розроблений на основі Java, перехід з Java здійснюється без труднощів [10].

Kotlin легко інтегрується з багатьма фреймворками, є простою для вивчення та має відкритий код. Зрозумілий синтаксис сприяє підвищенню продуктивності при створенні додатків, а також робить код більш читабельним і лаконічним.

Проте у Kotlin є і недоліки. Швидкість компіляції коду може бути нестабільною: іноді процес проходить швидко, а іноді з затримками. Оскільки мова є новою, на неї є обмежена кількість навчальних матеріалів, тому розробники часто змушені самостійно шукати рішення у разі виникнення проблем. Попри це, Kotlin вважається одним з найкращих інструментів для розробки Android-додатків.

З огляду на зручність мови, масовий перехід розробників з Java на Kotlin, а також підтримку Google, для цієї роботи було обрано Kotlin як основну мову програмування.

1.6.3 Xml

Зазвичай для створення візуального інтерфейсу в Android-проектах використовують спеціальні XML-файли. Ці файли виступають як ресурси розмітки і містять визначення інтерфейсу у форматі XML-коду. Подібний підхід схожий на створення вебсайтів, де інтерфейс визначається в HTML-файлах, а логіка програми – в JavaScript.

Використання XML для оголошення інтерфейсу користувача, щоб відокремити розмітку від коду програми. Це означає, що ми можемо змінювати інтерфейс без необхідності редагувати код на Java чи Kotlin. Наприклад, розмітки в XML можуть бути визначені для різних орієнтацій екрану, розмірів пристроїв, мов тощо. Окрім того, використання XML для розмітки спрощує візуалізацію структури інтерфейсу та полегшує налагодження.

Для складної верстки, такої як створення звітів з QR-кодами, зазвичай застосовують XML, оскільки файли цього формату легше конвертувати у PDF.

Для створення UI компанія Google також рекомендує більш зручний інструмент – Jetpack Compose.

1.6.4 Jetpack Compose

Для спрощення розробки, прискорення процесу та полегшення підтримки, компанія Google анонсувала новий тулкіт – Jetpack Compose у травні 2019 року.

Jetpack Compose - це сучасний інструмент від Google для створення програм під Android за допомогою мови програмування Kotlin. Він спрощує процес написання та оновлення візуального інтерфейсу додатків, надаючи декларативний підхід до розробки.

Jetpack Compose сумісний з Android-бібліотеками для Java та Kotlin, пропонуючи новий підхід до розробки. Він потребує базових знань Kotlin, використовуючи її переваги. Compose зменшує кількість коду та забезпечує інтуїтивний декларативний API.

Важливою концепцією в Jetpack Compose є функція composable (функції анотовані за допомогою @Composable). Ці функції визначають окремі частини візуального інтерфейсу, з яких складається додаток. Це значно спрощує створення та оновлення складних інтерфейсів, а також полегшує тестування та підтримку компонентів.

1.6.5 JSON

JSON - це простий формат обміну даними, який легко читати та записувати як на людях, так і на комп'ютерами. Він базується на підмножині мови програмування JavaScript, визначеній у стандарті ECMA-262 3rd Edition від грудня

1999 року. JSON є текстовим форматом, який повністю незалежний від мови реалізації, але використовує правила синтаксису, знайомі програмістам мов типу C, таких як C, C++, C#, Java, JavaScript, Perl та Python та інших. Завдяки цим характеристикам, JSON є ідеальним форматом для обміну даними.

JSON має дві основні структури даних:

– колекція пар "ключ-значення". У різних мовах програмування це поняття застосовується у вигляді об'єкта, регістра, структури, словника, хешу, іменованого списку або реляційного масиву;

– упорядкований перелік значень. У більшості мов це реалізовано як масив, вектор, список або масив.

Ці структури є універсальними для майже всіх сучасних мов програмування. Логічно припустити, що формат даних, незалежний від мови програмування, повинен базуватися саме на цих структурах.

У цій дипломній роботі формат JSON використовується для обміну інформацією між клієнтською та серверною частинами розподіленого додатку.

1.6.6 MVVM

Важливо дотримуватися чіткої архітектури, щоб забезпечити стабільність, читабельність та ремонтопридатність додатків.

MVC та MVP - це архітектурні шаблони, які були популярні в розробці Android до випуску MVVM. Дек дек в шаблоні MVC модель містить бізнес-логіку і дані, контролер забезпечує взаємодію між моделлю і поданням, подання відображає дані користувачеві в MVP, ведучий діє як сполучна ланка між моделлю і поданням, а подання забезпечує відображення тільки даних.

MVVM – це архітектурний шаблон для клієнтських додатків, запропонований Джоном Госсманом як альтернатива шаблонам MVC і MVP, особливо при використанні технології зв'язування даних (Data Binding). Основна концепція MVVM полягає в розділенні логіки відображення даних від бізнес-логіки, що досягається через винесення їх у окремий клас для чіткого розмежування.

У MVVM процес розробки графічного інтерфейсу користувача розділений на 2 частини. У першій використовується мова розмітки або графічний інтерфейс користувача, а в другій розробляється бізнес-логіка або внутрішня логіка (модель даних). Частина ViewModel у MVVM діє як перетворювач значень, оскільки вона відповідає за перетворення з моделі у формат, зручний для роботи з об'єктами даних. З цієї точки зору, ViewModel є моделлю між поданнями і керує більшою частиною логіки відображення.

Уявлення (View) містить структуру, яку користувачі бачать на екрані. Це може включати як статичний, так і динамічний вміст, наприклад, анімації чи зміни станів. Важливо, що в уявленні не повинно бути логіки додатку. В Android уявленням може бути активність або фрагмент.

Модель-уявлення (ViewModel) з'єднує модель та уявлення. Вона відповідає за управління даними та їх перетвореннями (конверсіями). Важливим аспектом є використання біндінгу даних. В Android це зазвичай реалізується через класи, такі як AndroidViewModel або ViewModel, які дозволяють ефективно зв'язувати дані з уявленням без необхідності вручну керувати процесами.

Схематична взаємодія компонентів MVVM наведена на рис. 1.1.

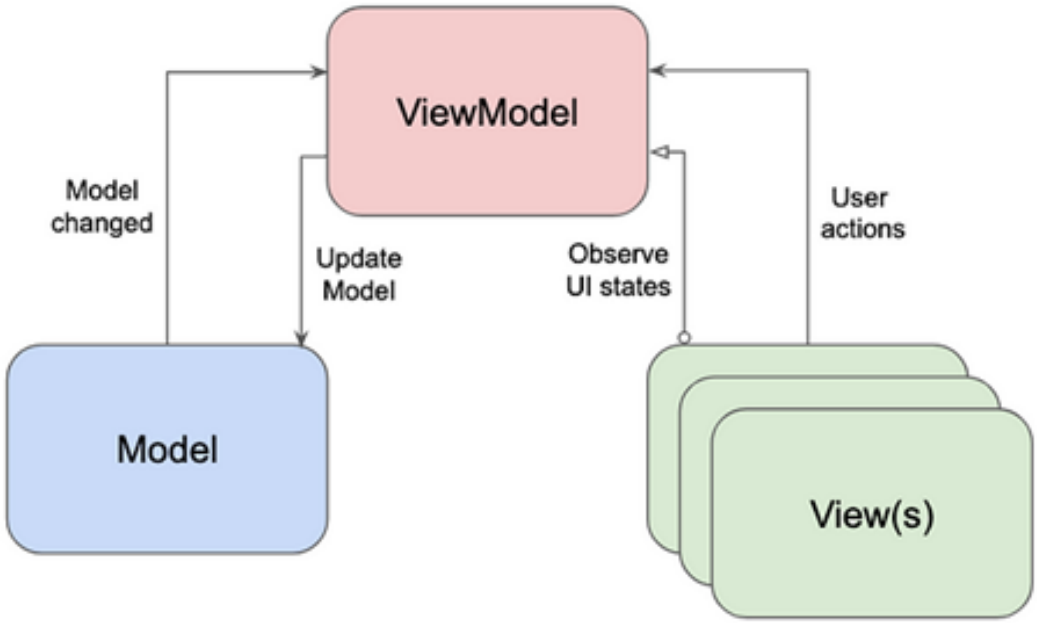

Рисунок 1.1 - Схематична взаємодія компонентів MVVM

Модель (Model) відповідає за рівень бізнес-даних і не залежить від конкретного графічного уявлення. В Android, відповідно до "чистої" архітектури, модель може включати базу даних, репозиторій і класи бізнес-логіки, що зберігають та обробляють дані програми.

1.6.7 Dependency injection

Dependency Injection – це процес надання зовнішніх залежностей програмному компоненту. Згідно з принципом єдиного обов'язку, об'єкт делегує створення своїх залежностей зовнішньому механізму, спеціально призначеному для цієї мети.

Ін’єкція залежностей є поширеним підходом у програмуванні і дуже добре підходить для розробки Android-додатків. Використовуючи DI, розробник створює основи для правильної архітектури програми, що забезпечує більшу гнучкість та розширюваність.

Реалізація DI надає кілька ключових переваг: можливість повторного використання коду, спрощення рефакторингу та полегшення тестування.

Популярним способом реалізації DI в Android-додатках є використання фреймворку Dagger, який, однак, вимагає значних зусиль для вивчення та налаштування. Альтернативою цьому є Koin – бібліотека, написана на чистому Kotlin, яка буде використовуватись у даній роботі, і є більш простою в освоєнні та використанні.

1.6.8 RxJava

Підтримка асинхронних і паралельних обчислень останнім часом стала важливою особливістю багатьох мов програмування. Паралельні обчислення дозволяють виконувати кілька завдань одночасно, а асинхронна обробка дозволяє уникнути блокування основного потоку програм під час тривалих операцій. Наприклад, якщо ви хочете створити графічне додаток для мобільного пристрою, вам необхідно відправити запит на інтернет-ресурс після натискання кнопки. Однак такий запит може зайняти багато часу, і запит повинен бути відправлений асинхронно, щоб не "зависати" під час роботи програми. В результаті користувач

може продовжити запуск програми, і після отримання відповіді від інтернет-джерела з'явиться відповідне повідомлення.

RxJava-це приклад реалізації шаблону спостерігача, який включає механізм асинхронної обробки. Це бібліотека для розробки реактивних програм на мові Java, які можуть асинхронно обробляти потоки даних та подій. RxJava реалізує модель ReactiveExtensions, яка дозволяє обробляти події та потоки даних асинхронним способом.

Основні переваги RxJava включають поліпшену продуктивність додатків, підвищену ефективність мережевих запитів і спрощену розробку асинхронного дека. Крім того, RxJava надає стандартні операції для обробки даних, такі як фільтрація, зіставлення і агрегація. Однак використання RxJava може мати деякі недоліки. Нерозуміння принципів реактивного програмування або неправильне використання потоків може призвести до проблем із продуктивністю та несподіваної поведінки програми. Крім того, освоєння RxJava для початківців жовтня - складне завдання, і на її освоєння йде чимало часу.Зважаючи на те, що Android-додаток у даному випадку буде розроблятися мовою Kotlin, доцільно розглянути вбудований в цю мову інструмент для асинхронної роботи – Kotlin Coroutines, який є спрощеним і більш природним способом роботи з асинхронністю в Kotlin.

1.6.9 Kotlin Coroutines

У мові Kotlin підтримка асинхронності та паралельних обчислень реалізована через корутини (coroutines). Корутини – це блоки коду, що виконуються асинхронно, тобто по черзі. В потрібний момент виконання корутини зупиняється, зберігаючи всі свої властивості, і дає можливість виконати інший код. Коли керування повертається до початкової корутини, вона продовжує свою роботу. Завдяки цьому програма може виконувати кілька функцій одночасно, що забезпечує високу ефективність і не блокує основний потік.

Окрім спільних програм, Kotlin також підтримує асинхронну передачу даних через потоки kotlin. жовтень. Потокове передавання-це асинхронний потік даних, який може надавати споживачам нульове або більше значення в будь-який момент

часу. Потоки Kotlin забезпечують зручний і безпечний спосіб передачі даних між компонентами програми, дозволяючи вам перемикати асинхронні деки без необхідності використання зовнішніх бібліотек.

Ключова відмінність між потоками Kotlin і RxJava: потоки Kotlin є частиною мови, тоді як RxJava – окрема бібліотека. Потоки Kotlin простіші, швидші та підтримують багатопоточність і асинхронність, що ідеально для серверних додатків. RxJava натомість пропонує ширший набір операторів для складних сценаріїв.

1.7 Аналіз технологій для серверної частини системи

Існує безліч варіантів створення серверної частини розподіленої прикладної системи. Ми рекомендуємо використовувати серверні мови програмування, такі як PHP, Java, Python і Node, щоб забезпечити гнучкість і масштабованість при зміні бізнес-вимог.js та інші. Кожна з цих мов має свої переваги та недоліки, і вибір конкретної мови залежить від вимог проекту, технічних характеристик та уподобань розробника.

1.7.1 PHP

PHP — це одна з найпопулярніших мов програмування загального призначення, розроблена спеціально для створення динамічних веб-сайтів і веб-додатків. PHP дозволяє вбудовувати код безпосередньо в HTML, що відрізняє його від JavaScript. PHP-скрипт запускається на сервері, генерує HTML-код і надсилається клієнту. PHP-код розміщується між спеціальними тегами <?php та ?>для ввімкнення/вимкнення режиму PHP всередині коду PHP широко використовується у веб-розробці для створення розподілених додатків з клієнтською частиною на мобільних платформах, але він підходить для таких сценаріїв. Варто розглянути інші альтернативні технології, які підходять для таких сценаріїв.

1.7.2 Java Servlets

Java Servlets – це технологія Java, яка дозволяє створювати веб-додатки для обробки HTTP-запитів і надання відповідей на них. За допомогою сервлетів можна реалізувати різноманітні функції, такі як формування відповіді на запит, обробка даних з форм, а також здійснення авторизації та аутентифікації користувачів. Крім того, серлети використовуються для зберігання і доступу до даних, а також для реалізації інших необхідних функцій, що забезпечують належну роботу веб-додатку. Java Servlets є однією з найпопулярніших технологій у сфері корпоративної веб-розробки.

1.7.3 Node.js

Node.js – це середовище виконання для JavaScript, яке дозволяє запускати JavaScript на серверній стороні. Цей метод є дуже ефективним, оскільки дозволяє писати код на стороні сервера тією ж мовою, що використовується на стороні клієнта. Це значно спрощує і покращує процес розробки веб-додатків, забезпечуючи зручність та підвищену продуктивність. Node.js також підтримує взаємодію між клієнтом і сервером в реальному часі, що сприяло його популярності в сфері веб-розробки.

1.7.4 Spring Boot

Раніше для розробки веб-застосунків програмісти активно використовували JavaBeans – класи в Java, створені за специфічними правилами. Вони спрощували розробку компонентів інтерфейсу користувача, але мали суттєві обмеження, такі як відсутність можливості управління транзакціями, безпекою та багатопоточністю, що критично важливо для великих програм.

Spring Boot – це фреймворк, який значно полегшує та прискорює розробку веб-застосунків і мікросервісів на основі Spring Framework. Він надає інструменти для швидкого старту проекту, зменшуючи рутинні налаштування й інтеграцію з основним Spring Framework. Головна мета Spring Boot – забезпечити простоту та ефективність при створенні додатків, усунувши необхідність вручну налаштовувати параметри через стандартні конфігурації та автоматичне налаштування, що значно пришвидшує розробку.

Цей фреймворк був розроблений компанією Pivotal Software (тепер частиною VMware) у 2014 році і з того часу став одним із найбільш популярних інструментів серед Java-розробників.

Однією з основних переваг Spring Boot є автоконфігурація, яка дозволяє автоматично налаштовувати додаток на основі його залежностей та структури без необхідності втручання з боку розробника. Фреймворк також включає вбудовані сервери, такі як Tomcat чи Jetty, що дозволяє легко розгорнути програму без додаткових налаштувань серверів. Крім того, Spring Boot має великий набір стандартних залежностей, що спрощує їх додавання та управління ними.

Завдяки підтримці Kotlin, використання Spring Boot є ідеальним вибором для серверної частини цього проекту. Оскільки мобільний застосунок також розробляється на Kotlin, це дозволяє забезпечити однорідність технологічного стека, полегшити розробку і підтримку коду, а також інтеграцію різних компонентів системи.

1.7.5 WebSocket

WebSocket – на відміну від автономного веб-протоколу HTTP, який дозволяє створювати інтерактивне з'єднання між сервером і клієнтом для обміну повідомленнями в реальному часі, WebSocket унікальний і дуже корисний для інтерактивних додатків, оскільки забезпечує двосторонній потік даних.

Першу версію протоколу було представлено в 2009 році, а вже в 2011 році він отримав статус RFC (Request for Comments), ставши стандартом, який широко використовується в Інтернеті. Зараз підтримка WebSocket доступна на всіх мобільних, десктопних та серверних пристроях.

Протокол WebSocket дозволяє створити постійне двонаправлене з'єднання. Сервер може не тільки відповідати на запити клієнта, але й ініціювати передачу даних у реальному часі без необхідності постійно чекати запити від клієнта. Обмін даними здійснюється в рамках одного з'єднання, що підтримує постійну взаємодію. Це з'єднання починається з процесу, відомого як «відкриваюче рукостискання». Клієнту достатньо просто прослуховувати канал для отримання оновлень від сервера.

Протокол WebSocket ідеально підходить для інтерактивних сервісів та веб-додатків, що потребують постійного оновлення даних у реальному часі. У рамках цього проекту він буде корисним для розробки системи завантаження товарів на склад, що дозволить працівникам підприємства оперативно відстежувати кількість продукції в контейнерах, забезпечуючи ефективність управління та обробки даних.

1.8 Висновки за розділом

У першому розділі розглянуто складський облік і визначено завдання для розробки системи автоматизації управління фармацевтичною продукцією. Проаналізовано сучасні WMS - системи, такі як NetSuite, Fishbowl та Odoo, що допомогло визначити підхід для створення системи на Android із використанням QR-кодів та генерації PDF-звітів для покращення ефективності управління складом.

Основними критеріями вибору технологій були доступність і вартість інтеграції. Використання Android дозволяє працювати з різними пристроями, підтримуючи функції автоматизації, такі як сканування штрих-кодів. Для архітектури обрано монолітний підхід через його простоту та швидкість розробки.

Для розробки застосунку обрана мова Kotlin, оскільки вона має переваги перед Java, зокрема зручну роботу з нульовими значеннями та безпеку виконання коду. Для верстки інтерфейсу використано Jetpack Compose, що забезпечує більшу гнучкість порівняно з XML. Архітектурний паттерн MVVM дозволяє чітко розділяти бізнес-логіку і UI, а Kotlin Coroutines забезпечують ефективну асинхронну роботу без додаткових бібліотек.

Для ін’єкції залежностей обрана бібліотека Koin, для серверної частини — Spring Boot. Для обміну даними в реальному часі використовується протокол WebSocket, що зменшує затримки та підвищує ефективність взаємодії.

2 РОЗРОБКА РОЗПОДІЛЕНОЇ СИСТЕМИ

2.1 Етапи розробки розподіленого додатку

Для створення масштабованих і зручних у підтримці розподілених додатків, основні етапи розробки, як правило, включають бекенд, фронтенд та розгортання, а також тестування та підтримку.

Розробка серверної частини охоплює кілька основних етапів:

– проєктування та реалізація базової архітектури, що включає базу даних, серверні API та бізнес-логіку;

– налаштування серверного середовища для обробки HTTP-запитів, що включає обробку CRUD операцій (створення, читання, оновлення та видалення даних);

– забезпечення масштабованості та стійкості через балансування навантаження, масштабування інфраструктури та впровадження CI/CD.

Створення клієнтської частини під ОС Android включають:

– дизайн інтерфейсу, що відповідає вимогам до юзабіліті та забезпечує зручний доступ до функціоналу додатку;

– розробка віджетів представлення з використанням XML або Jetpack Compose у зв’язку з мовою Kotlin;

– розробку та оптимізацію логіки відображення, що включає навігацію, анімації та адаптацію елементів для покращення користувацького досвіду.

Тестування програмного забезпечення охоплюють:

– функціональне тестування, щоб перевірити коректність виконання основних функцій додатку;

– інтеграційне тестування для оцінки сумісності між окремими компонентами системи та їх взаємодії;

– продуктивне тестування за допомогою CI/CD конвеєрів, таких як Jenkins чи Travis CI, для забезпечення стабільності та надійності всіх оновлень додатку.

Основні кроки розгортання та підтримки додатку включають:

– налаштування серверної інфраструктури для забезпечення безперервного та стійкого функціонування додатку;

– автоматизацію розгортання з використанням контейнеризації, наприклад, через Docker або Kubernetes;

– розміщення мобільного додатку в Google Play або AppStore після проходження модерації, яка враховує особливості проєкту, його історію та категорію.

2.2 Визначення технічних вимог до системи

При проектуванні розподілених систем важливо враховувати технічні вимоги, що визначають мінімальні ресурси для стабільної роботи. Вони залежать від масштабу, функцій, типу додатка та технологій. У розділі розглянуто фактори, що впливають на характеристики, та рекомендації щодо їх адаптації до проєкту.

2.2.1 Вимоги до робочих характеристик серверної частини

Для забезпечення стабільної та ефективної роботи серверної частини додатку, розробленої на базі Spring Framework, необхідно врахувати певні апаратні вимоги.

2.2.1.1 Процесорні ресурси

Серверу потрібен чотириядерний процесор із частотою 3.0 ГГц і вище для багатозадачності та обробки запитів. Рекомендовані Intel Xeon або AMD Ryzen, які забезпечують продуктивність для розподілених систем:

– Intel Xeon E-2286G: 8 ядер, тактова частота 3.8 ГГц (базова), 4.9 ГГц, що забезпечує високу продуктивність у робочих навантаженнях серверного типу;

– AMD Ryzen 7 5800X: 8 ядер, частота 3.8 ГГц (базова), до 4.7 ГГц (турбо), який є чудовим вибором для обробки високонавантажених завдань за розумною ціною;

– Intel Core i7-10700K: 8 ядер, тактова частота 3.8 ГГц (базова), 5.1 ГГц (турбо), добре підходить для серверів із середнім навантаженням і підтримує багатопоточність.

2.2.1.2 Оперативна пам'ять

Для стабільної роботи та високої пропускної здатності необхідно не менше 8 ГБ RAM, а для інтенсивних завдань - 16 ГБ або більше. Гарними рішеннями для побудови розподіленої системи будуть:

– Corsair Vengeance LPX DDR4 16 ГБ (2x8 ГБ) 3200 МГц: забезпечує високу продуктивність та стабільність;

– G.Skill Ripjaws V DDR4 16 ГБ (2x8 ГБ) 3600 МГц: відмінний вибір для серверів із навантаженнями середнього рівня;

– Crucial Ballistix DDR4 32 ГБ (2x16 ГБ) 3200 МГц: забезпечує значний обсяг пам’яті для сервісів, що вимагають обробки великих обсягів даних.

2.2.1.3 Сховище даних

Для забезпечення швидкого доступу до даних і надійності рекомендується SSD-накопичувачі типу NVMe:

– Samsung 970 EVO Plus NVMe M.2 500 ГБ: забезпечує високу швидкість зчитування і запису, оптимально підходить для додатків з інтенсивною обробкою даних;

– Western Digital Black SN750 NVMe M.2 500 ГБ: надійний і високопродуктивний накопичувач для серверних середовищ;

– Kingston A2000 NVMe PCIe M.2 1 ТБ: доступний за ціною та забезпечує достатній обсяг для великих баз даних і логів.

2.2.1.4 Мережеві інтерфейси

Сервер повинен мати інтерфейс, здатний забезпечувати швидкість передачі даних на рівні 1 Гбіт/с. У виборі мережевих можливостей достатньо наявності гігабітного Ethernet-порту для швидкого та стабільного з'єднання з мережею Інтернет:

– Intel Gigabit CT PCI-E Network Adapter: стабільний і продуктивний адаптер для серверних рішень;

– TP-Link TG-3468 PCI-E Network Adapter: бюджетний варіант, який забезпечує підтримку швидкості до 1 Гбіт/с.

2.2.1.5 Операційна система

Під час вибору операційної системи для розгортання розподіленого додатку важливо обрати операційну систему, яка може ефективно підтримувати високе навантаження, безпеку та зручність адміністрування. Зазвичай для серверних рішень використовуються Linux-дистрибутиви, оскільки вони надають низку переваг, зокрема стабільність, безпеку, гнучкість у налаштуванні та сумісність із сучасними технологіями. Розглянемо кілька популярних варіантів і їхні ключові особливості.

Ubuntu Server (версія 20.04 LTS) є одним із найпоширеніших варіантів завдяки підтримці довгострокових оновлень, широкому ком'юніті та простоті в налаштуванні. Це робить Ubuntu зручним вибором як для початківців, так і для досвідчених адміністраторів. Дистрибутив підтримує широке коло програм та інструментів, включаючи Docker і Kubernetes, що підвищує його сумісність із контейнерними додатками та дозволяє легко налаштовувати багатокористувацькі середовища.

CentOS (версії 7 та 8) більше підходить для інфраструктур, де критично важливі стабільність та захищеність. Як версія на базі Red Hat Enterprise Linux (RHEL), CentOS має тривалий цикл підтримки, що забезпечує його надійність і передбачуваність для систем із високими вимогами до безпеки та стабільності. Його чітка політика оновлень, які рідко змінюють системні компоненти, робить CentOS ідеальним вибором для великих корпоративних середовищ і тих, хто потребує безперебійного обслуговування та надійної інтеграції з різними корпоративними рішеннями.

Debian (версії 10 або 11) відома своєю стійкістю до збоїв і регулярно оновлюваними компонентами безпеки. Ця система є оптимальним вибором для серверів, що вимагають високої стабільності і забезпечують довготривалу підтримку. Debian має широке покриття архітектур, що дозволяє гнучко налаштовувати її під різні серверні середовища. Ця операційна система також

використовується як основа для багатьох інших дистрибутивів, зокрема Ubuntu, і підходить для конфігурацій, де потрібні особливо високі стандарти безпеки.

2.2.2 Вимоги до робочих характеристик клієнстської частини

Наступним етапом є вибір версії Android SDK на якій буде проводитись розробка додатка. Android SDK – це пакет розширень Android, який надає додаткові, зручні інструменти розробки. За допомогою цього пакету можна відслідковувати стан операційної системи, читати лог-файли, створювати віртуальні пристрої для тестування роботи додатка на різних версіях ОС і багато іншого. Від версії SDK залежить і версія операційної системи мобільного додатка.

Для мобільного додатку, оптимальною мінімальною версією Android є 9.0 (Pie), яка дає змогу скористатися сучасними можливостями та API. Android 9.0 включає функції, що значно покращують продуктивність і зручність: наприклад, Adaptive Battery та Adaptive Brightness, які інтелектуально керують ресурсами пристрою, продовжуючи час роботи від батареї. Окрім того, функція App Actions передбачає дії користувача на основі контексту, що полегшує роботу з додатком.

Ще одна перевага Android Pie – поліпшена підтримка безпеки, зокрема обмеження доступу до приватних даних, а також шифрування резервних копій у Google. За даними API version distribution від Android Studio, що приведені на рисунку 2.1, завдяки зростаючій популярності цієї версії, Android 9 дозволяє покрити близько 89,6% активних Android-пристроїв станом на 2024 рік, що забезпечує високу сумісність додатка з ринковими пристроями та дозволяє значно розширити його аудиторію користувачів.

Оскільки в даній роботі були проведені експерименти з метою оцінки продуктивності мобільних пристроїв під час виконання завдань генерації звітів з QR-кодами, а також аналізу споживання ресурсів, мінімальні апаратні вимоги для забезпечення комфортного користування були визначені у висновках дослідження.

Для встановлення додатку на пристрій необхідно було мати щонайменше 120 мегабайт вільного місця. Це дозволяло гарантувати, що додаток матиме достатньо простору для нормальної роботи, зокрема для зберігання даних, кешування та інших тимчасових файлів. Такі параметри забезпечували стабільну

та безперебійну роботу додатку на смартфонах з подібними характеристиками, без ризику виникнення проблем із продуктивністю чи збоями через нестачу пам’яті. Це також дозволяло користувачам з обмеженими ресурсами на пристрої ефективно користуватися додатком, не стикаючись із затримками або помилками під час роботи.

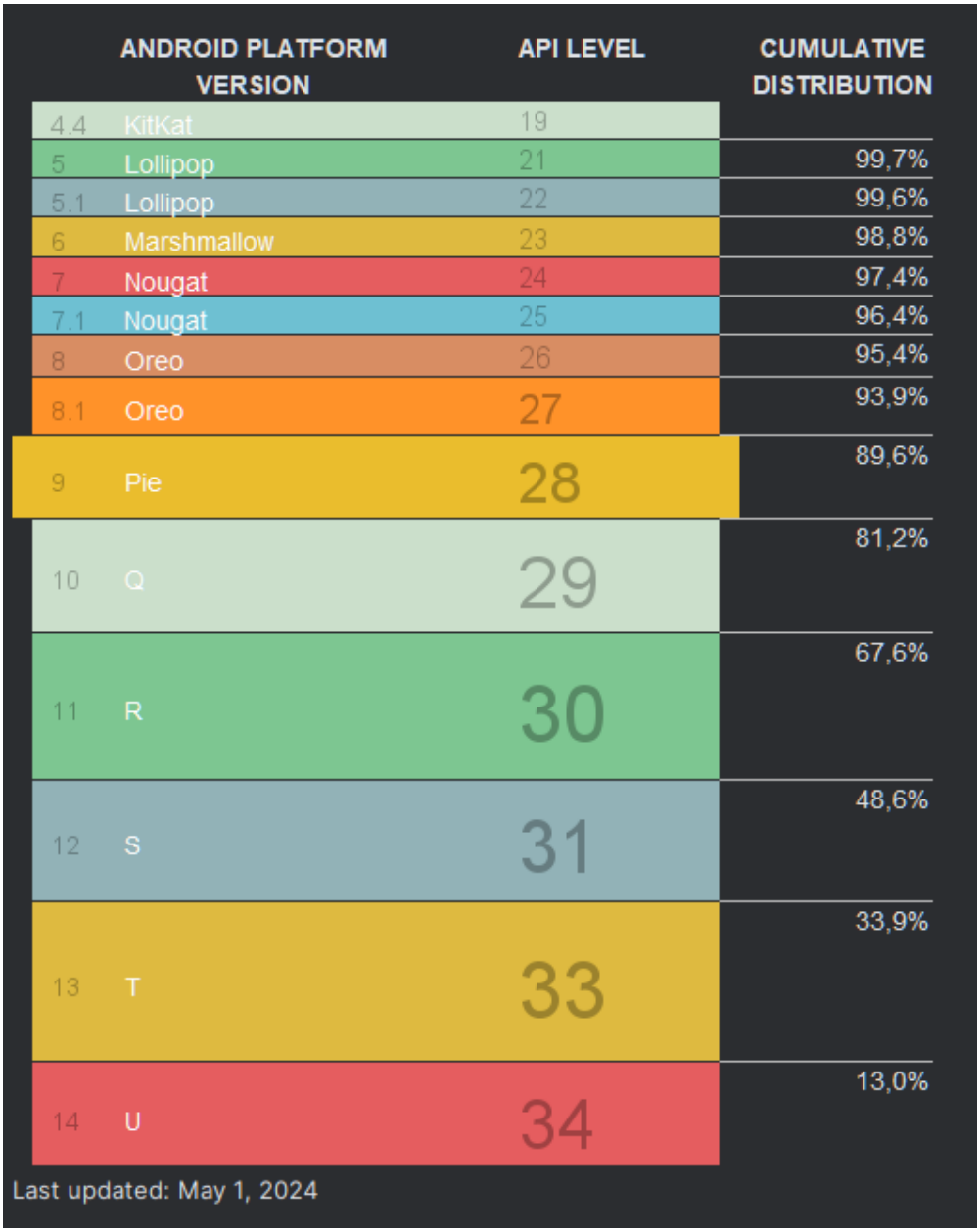

Рисунок 2.1 – Розповсюдження версій Android API

Зважаючи на особливості використання додатку, для його коректного функціонування було необхідне постійне підключення до інтернету. У випадку

поганого з’єднання додаток не міг працювати, оскільки доступ до мережі був важливий для завантаження даних із сервера та авторизації в системі.

2.2.3 Зручність супроводження

Кожен компонент системи являє собою окремий модуль, здатний працювати автономно. Він має бути реалізований відповідно до стандартних архітектурних шаблонів, що забезпечують простоту масштабування проекту, дозволяючи легко додавати новий функціонал або змінювати наявний.

2.3 Визначення програмних вимог до системи

Після визначення технічних характеристик системи необхідно встановити програмні вимоги, яким вона повинна відповідати. Серед таких вимог можуть бути складність і зручність користувацького інтерфейсу, швидкодія, стійкість до помилок і забезпечення безпеки та цілісності даних.

2.3.1 Визначення вимог до безпеки даних

У процесі розробки мобільного додатку важливо враховувати, що дані, з якими він працює, можуть зацікавити третіх осіб. Незалежно від варіації цінності цих даних, навіть базова приватна інформація, така як пароль для входу, потребує особливої уваги до забезпечення її захисту.

Одним із фундаментальних елементів безпеки мобільних додатків є правильна реалізація процесів аутентифікації та авторизації користувачів. Це системи, які не тільки перевіряють особистість користувача, але також визначають рівень доступу та дії, які користувачу дозволено виконувати в додатку. Щоб гарантувати відмовостійкість цієї системи, розробники додатків повинні підтримувати ряд найкращих практик для розробки мобільних додатків.

Захист користувацьких сесій у мобільному додатку є критичним для запобігання атакам перехоплення сесій. Для цього широко застосовуються токен-базовані платформи аутентифікації, такі як JWT. Токени створюються після

успішного входу в систему і забезпечують безпечне управління сеансами. Оскільки токени можуть містити важливу інформацію про користувача та дозволи, необхідно поводитися з ними обережно. Для цього в системі необхідно забезпечити зберігання чутливої інформації в локальному сховищі, що підтримує шифрування даних. Необхідно використовувати короткострокові токени доступу разом із довготривалими токенами для оновлення, що допоможе збалансувати безпеку та зручність користувача.

Окрім цього, додаток не повинен відображати приватну інформацію користувача великими, яскравими шрифтами без явної необхідності та окремого запиту від користувача, щоб уникнути можливості зчитування цих даних на відстані з екрану пристрою.

Також у релізній версії додатку слід відключити логування даних у системну консоль та незахищені файли. Логи, специфічні для розробників, можуть бути включені в систему, проте бажано у зашифрованому вигляді, щоб запобігти доступу сторонніх осіб до закритої службової інформації, яка може міститися у логах.

2.3.2 Визначення вимог до логічного устрою бази даних

Система керування базами даних повинна відповідати вимогам цілісності та безпеки даних. Усі сутності системи мають бути чітко взаємопов’язані, без логічних помилок.

СКБД має забезпечувати стабільну роботу під час виконання частих стандартних операцій, таких як перегляд, додавання, редагування та видалення даних, які стосуються предметної області системи.

2.3.3 Функціональність програмного забезпечення

Програмний продукт має виконувати наступні функції:

– виконання операцій завантаження товарів у коробки за допомогою станції приймання;

– генерація та відправлення звіту з інформацією про продукти та QR-кодами на принтер;

– сканування QR-кодів та перегляд закодованої інформації щодо сутності предметної галузі;

– пошук та перегляд інформації про існуючі продукти;

– пошук та перегляд інформації про існуючі контейнери;

– виконання операцій закріплення контейнерів до палет;

– виконання операцій закріплення палет до станцій.

2.3.4 Зовнішні інтерфейси

2.3.4.1 Визначення вимог до інтерфейсу програми

Проектування інтерфейсу є важливою складовою розробки мобільного додатка.

Інтерфейс – це інструмент взаємодії користувача з системою. Особливу увагу при розробці інтерфейсу слід приділити простоті і доступності системи для користувача. Функціонал має бути інтуїтивно зрозумілим, оформлення – приємним і витриманим в одному стилі, дизайн – адаптивним, для коректної роботи застосунка на різних пристроях.

Проектування UI мобільного додатку передбачає створення інтерфейсу, який буде одночасно унікальним і функціональним, а також відповідатиме користувацьким очікуванням. Основна увага приділяється мінімалізму та інтуїтивності, щоб уникнути перевантаження екрана складними елементами, які можуть вплинути на швидкість роботи програми. Важливі функції мають бути доступними з головного екрана, забезпечуючи простоту навігації та зручність взаємодії. Для досягнення адаптивності інтерфейс повинен динамічно підлаштовуватися під розміри різних екранів, зберігаючи доступність всіх елементів.

Для максимального охоплення аудиторії з різних країн важливим є впровадження підтримки кількох мов інтерфейсу, що дозволить зробити додаток орієнтованим не тільки на локальний, а й на міжнародний ринок. Наявність багатомовного інтерфейсу також підвищує довіру до додатка, демонструючи інклюзивність та турботу про зручність для всіх потенційних користувачів.

Дизайн, оптимізований для екранів OLED, повинен враховувати темну тему, яка покращує читабельність в умовах низького освітлення та допомагає економити заряд батареї. У ключових місцях, таких як кнопки «зберегти» і «видалити», елементи повинні розташовуватись зручно для користувача, а також на достатній відстані один від одного для уникнення помилок. Важливим аспектом є надання відгуку на дії користувача, наприклад, показ повідомлення про успішне збереження даних після натискання відповідної кнопки.

2.3.4.2 Функціональність інтерфейсу мобільного додатку

На підставі опису функціональності можна визначити функціональність інтерфейсу мобільного додатку.

ФІМД-1. Авторизація. Для того щоб авторизуватися у системі, під час першого відкриття додатку користувачу необхідно ввести дані облікового запису, що представлені у вигляді логіну та паролю. Після натискання на кнопку “Sign In” розпочнеться процес авторизації та з’явиться індикатор завантаження. У разі успішного входу, або автоматичної авторизації, користувач потрапляє на головний екран. Якщо дані невірні, додаток проінформує про помилку за допомогою повідомлення.

ФІМД-2. Відкриття бокового меню. Користувач на головному екрані натискає на “Гамбургер-меню”. “Гамбургер-меню” - це елемент інтерфейсу, зазвичай представлений у вигляді трьох горизонтальних смуг, який використовується для приховування пунктів навігації на мобільних пристроях або вузьких екранах. У боковому меню можна змінити установу, до якої прив’язані всі дані системи, обрати мову додатку, або за допомогою кнопки “Log out” вийти з облікового запису.

ФІМД-3. Підключення до станції приймання. Користувач на головному екрані, натискає на пункт “Почати приймання”. Відкривається екран з камерою для сканування станції приймання. У разі сканування іншої сутності, додаток сповістить про помилку. Після успішного сканування станції необхідно обрати товар на приймання за допомогою кнопки “Ввести вручну” або сканувати QR-код товару.

ФІМД-4. Вибір товару. Під час ручного введення товару відкривається модальне вікно з списком усіх наявних у системі товарів. За допомогою рядку пошуку передбачена можливість пошуку певних товарів за назвою. Натискання на кнопку “Додати” передає дані обраного товару на екран, що викликав модальне вікно.

ФІМД-5. Приймання товарів. Після успішного налаштування підключення до станції приймання на екрані приймання відображається обраний продукт та UI-елемент “Tab-Layout”, який забезпечує горизонтальний макет для відображення вкладок “Активні контейнери” та “Готові контейнери”. Перша вкладка містить у собі візуальну репрезентація реальної станції приймання та список рекомендованих для завантаження коробок. Репрезентація станції являє собою горизонтальний макет з шести слотів під контейнери. Кожен може включати інформацію про кількість продукта та можливі проблеми наприклад, невідповідність ваги. У разі проблем на слоті користувач бачить відповідний індикатор. Коли працівник складу поміщає справжній контейнер в один із слотів реальної станці, у додатку відповідний слот стає активним. Це дає змогу користувачу обрати кількість товару, що був завантажений, за допомогою кнопок “+” та “-“. Після підтвердження кількості товару, у разі зняття контейнера зі станції, цей контейнер переходить у категорію готових. Процесу завершення приймання товарів виглядає наступним чином: користувач сканує QR-код готового контейнеру, обраний контейнер підсвічується у списку, після чого відбувається сканування певної палети, до якої буде закріплений контейнер.

ФІМД-6. Сканування QR-кодів. На кожному екрані додатку у лівій нижній частині розташовується UI-елемент “Floating action button”, натиск на яку відкриває модальне вікно з камерою. Під час наведення камери на QR-код сутність ідентифікується системою, та у разі успішної валідації відкривається екран з деталями товару, контейнера, або палети. Виключення становить процес приймання товарів, де сканування сутностей призводить не до навігації, а до виконання певних дій, що описані у ФІМД-3 - ФІМД-5.

ФІМД-7. Перегляд існуючих контейнерів. На головному екрані користувач натискає на пункт “Список контейнерів”. Відкривається екран зі списком усіх наявних у системі контейнерів. У верхній частині екрану знаходиться рядок пошуку для швидкого знаходження контейнера за ідентифікатором. Під рядком пошуку представлені фільтри в вигляді UI-елементів “Chips”, які користувач може обирати або знімати для гнучкої фільтрації за станом контейнера.

ФІМД-8. Пошук товарів. На головному екрані користувач натискає на пункт “Список товарів”. Відкривається екран зі списком усіх наявних у системі ліків. У верхній частині екрану знаходиться рядок пошуку за назвою препарату. Для зручності навігації товари згруповані за першими літерами їх назв, і кожна група відділена заголовком, що позначає цю літеру.

ФІМД-9. Перегляд інформації про товар. Після сканування QR-коду товару відкривається екран з детальною інформацією про товар та його кількість на складі. Під карткою товару користувач може переглянути в яких контейнерах знаходиться поточний товар, та побачити список рекомендованих контейнерів для поповнення. Натискання на певний контейнер відкриває його екран з деталями з ФІМД-11. Для того щоб здійснити поповнення без спеціалізованої станції працівник натискає на кнопку “Додати цей товар до контейнеру”, що відкриває модальне вікно з камерою для сканування потрібного контейнера.

ФІМД-10. Додавання товару до контейнеру. Після сканування потрібного контейнера на екрані товару з’являється спеціалізований інтерфейс з кнопками для зміни кількості товару та підтвердження поповнення.

ФІМД-11. Перегляд інформації про контейнер. Після сканування QR-коду контейнера відкривається екран із детальною інформацією про нього. Під карткою контейнера відображається список товарів із зазначенням кількості кожного з них, а також спеціалізована кнопка, що дозволяє відкріпити або закріпити контейнер до палети за допомогою модального вікна з камерою. Натискання на певний товар відкриває його екран з деталями з ФІМД-9.

ФІМД-12. Перегляд сканованої палети. Після сканування QR-коду палети відкривається екран із детальною інформацією про неї. Під карткою палети

відображається список контейнерів та товарів, що містяться в ній, а також спеціалізована кнопка, яка дозволяє відкріпити або закріпити палету до станції за допомогою модального вікна з камерою. Натискання на певний товар або контейнер відкриває екран з деталями про сутність з ФІМД-9 чи ФІМД-11 відповідно.

ФІМД-13. Генерація PDF-звіту з QR-кодами. Користувач на головному екрані натискає на пункт меню “Створити звіт”, та потрапляє на екран з визначенням кількості товарів, що буде фігурувати у PDF-документі. Натиск на кнопку “Згенерувати” запускає процес генерації, про який юзер дізнається за допомогою індикатору завантаження. У разі успіху додаток автоматично відкриє згенерований документ за допомогою існуючого засобу перегляду PDF-файлів на девайсі.

2.4 Проектування архітектури системи

Схематичне спрощене відображення архітектури розроблюваного додатку, який активно просувається компанією Google, представлено на рисунку 2.2.

Архітектурні шари Android – це організація коду та відповідальностей в процесі розробки додатків. Зазвичай вони включають презентаційний (presentation), доменний (domain) та шар даних (data). Кожен з цих шарів має свої функції та обов'язки, а їх взаємодія дозволяє побудувати ефективну та масштабовану архітектуру.

Презентаційний шар (presentation) – цей шар відповідає за представлення даних користувачу та взаємодію з ним через інтерфейс користувача. Він включає активності, фрагменти, адаптери, представлення та інші компоненти, які відображають дані користувачу та обробляють його дії. Презентаційний шар не повинен містити бізнес-логіки, його основна відповідальність - це перетворення та відображення даних, отриманих з доменного шару.

Розглянемо компоненти з схематичного представлення архітектури Android додатку, що представлені на рисунку 2.2, детальніше.

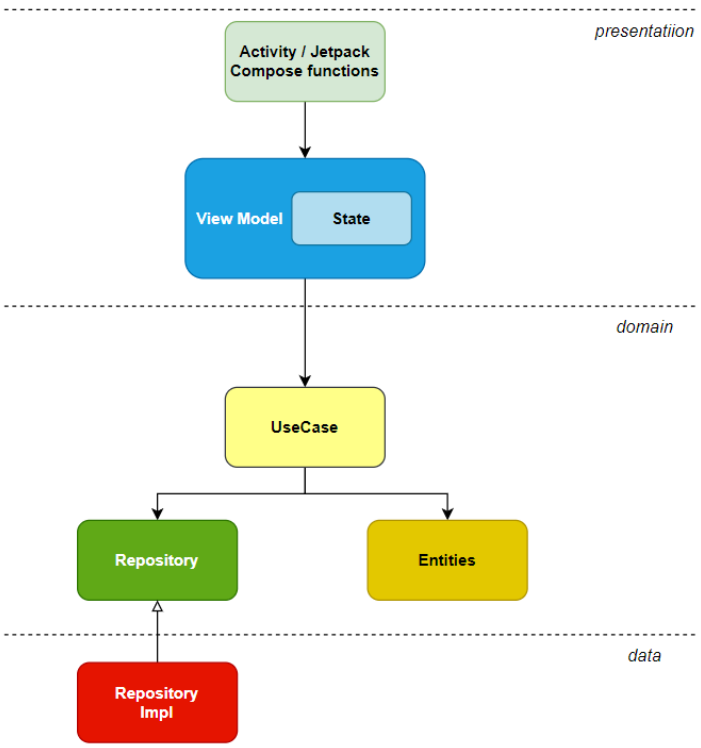

Рисунок 2.2 - Схема архітектури Android додатку

Доменний шар (domain): цей шар відповідає за бізнес-логіку додатку та визначає правила його роботи. Він включає моделі даних (Entities), інтерфейси репозиторіїв і забезпечує зв’язок між презентаційним шаром і шаром даних. Доменний шар є незалежним від платформи та джерел даних, що дозволяє забезпечити універсальність і цілісність бізнес-логіки, не пов'язаної з конкретною інфраструктурою.

Шар даних (data): цей шар відповідає за управління даними, включаючи їх отримання, збереження та обробку з різних джерел, таких як бази даних, веб-сервіси чи кеш. До його складу входять реалізації репозиторіїв, локальні та віддалені сховища, API-клієнти та інші компоненти, які забезпечують доступ до даних. Шар даних взаємодіє з доменним шаром, надаючи йому необхідні дані для роботи та обробляючи запити на збереження чи модифікацію даних, наприклад, виконуючи запити до бази даних або обмінюючись інформацією з веб-сервісами.

2.4.1 Activity та Jetpack Compose functions

Activity та функції Jetpack Compose є складовими шару View (представлення), і їхня основна роль полягає у забезпеченні взаємодії між користувачем і додатком. Ці компоненти не повинні містити бізнес-логіку або складних процесів, оскільки їхнє завдання – лише відображати дані та реагувати на дії користувача. Вони спостерігають за поведінкою користувача, передають відповідні події у вищі шари архітектури, і відображають інформацію, надану доменним шаром або шаром даних, у зрозумілому та зручному вигляді.

2.4.2 ViewModel

ViewModel зберігає стан інтерфейсу та забезпечує його цілісність під час змін конфігурації, виступаючи посередником між View і Model. Він взаємодіє з даними через UseCase, залишаючись незалежним від джерел. Кожен фрагмент має свій ViewModel для збереження стану екрана.

Compose State пропонує розробникам зручний спосіб оновлення та спостереження за станом додатку.

У контексті ViewModel State використовується для представлення стану користувацького інтерфейсу та передачі змін у реактивному стилі. ViewModel може містити кілька полів State, кожне з яких відповідає за різні аспекти стану додатку. Коли стан змінюється, ViewModel передає нове значення через StateFlow, і підписані на нього спостерігачі (наприклад, фрагменти або активності) отримують оновлення та оновлюють відповідні частини інтерфейсу користувача (UI) з урахуванням нових даних.

2.4.3 UseCase

UseCase представляє конкретний сценарій використання або операцію, яку виконує додаток. Він містить бізнес-логіку і здійснює запити до репозиторіїв або джерел даних для отримання та обробки необхідної інформації. UseCase може включати такі операції, як отримання списку елементів, додавання нового елемента, оновлення даних тощо. Він має бути незалежним від фреймворків та зовнішніх ресурсів, виконуючи лише бізнес-логіку, без залежності від конкретних технологій чи інфраструктури.

2.4.4 Repository

Repository є важливим компонентом в архітектурі Clean Architecture і служить для розділення бізнес-логіки додатку від джерел даних. У доменному шарі Repository визначає контракт (інтерфейс) для доступу до даних, описуючи необхідні операції для отримання, збереження та оновлення інформації. Це дозволяє зберігати чистоту бізнес-логіки, оскільки Repository абстрагує джерела даних від решти частин системи, надаючи єдиний доступ до даних через визначений інтерфейс.

У шарі data, реалізація Repository (Repository Implementation) забезпечує конкретну реалізацію методів, описаних в контракті Repository. Вона взаємодіє з різними джерелами даних, такими як локальна база даних, веб-сервер або зовнішні API, і виконує конкретні операції з даними, необхідні для виконання бізнес-логіки.

Використання Repository дозволяє зберегти чистоту доменної логіки, роблячи її незалежною від конкретних джерел даних, а також спрощує тестування та обслуговування додатку. Він створює абстракцію, яка відокремлює доменну логіку від деталей реалізації, що дозволяє змінювати або додавати нові джерела даних без необхідності змінювати саму логіку додатку.

2.5 Огляд бібліотек для інтеграції системи генерації та сканування QR-кодів

Зважаючи на потребу автоматизації та покращення облікових процесів, інтеграція функцій для генерації й сканування QR-кодів є ключовою для забезпечення ефективності та зручності користувачів. QR-коди слугують універсальним рішенням для швидкого доступу до даних, а також для їх миттєвого зчитування, що робить їх особливо корисними у мобільних додатках. Для реалізації таких функцій було обрано дві відомі бібліотеки: ZXing та ML Kit від Google. Вибір

цих бібліотек зумовлений їх високою продуктивністю, підтримкою актуальних стандартів кодування та стабільною підтримкою з боку розробників і спільноти.

2.5.1 Огляд бібліотеки Zxing

Бібліотека ZXing Zebra Crossing є відомим інструментом для обробки зображень штрих- та QR-кодів, розробленим на основі Java. Вона підтримує розпізнавання як 1D, так і 2D кодів, зокрема формати QR, PDF417, Aztec, UPC, EAN, Code 39, і Code 128. ZXing широко використовується в Android-додатках, що потребують сканування або генерації QR-кодів, оскільки є ефективною для реалізації зчитування кодів за допомогою камери на мобільних пристроях [27] .

ZXing включає модулі для обробки зображень, клієнтський код для Java SE, а також спеціалізований Android Barcode Scanner. Ця бібліотека є базовою для багатьох сторонніх проєктів та пропонує просте API для інтеграції з додатками. Серед інших функцій – можливість розпізнавати коди в реальному часі завдяки обробці зображень і оптимізації під мобільні пристрої. ZXing має активну спільноту, що надає документацію та підтримку для інтеграції в різні проєкти.

2.5.2 Огляд бібліотеки ML Kit

ML Kit – це бібліотека для мобільних пристроїв, розроблена компанією Google, що дозволяє легко інтегрувати функції машинного навчання в Android- та iOS-додатки. Використовуючи ML Kit, розробники можуть додавати до додатків різноманітні можливості, як-от розпізнавання тексту, зчитування штрих- і QR-кодів, обробка зображень, розпізнавання облич, та інші можливості комп’ютерного зору. Оскільки бібліотека працює на основі Google Machine Learning та TensorFlow Lite, її алгоритми оптимізовані для ефективного використання на мобільних пристроях, що робить її використання доцільним у даній роботі.

Для сканування QR-кодів ML Kit пропонує окремий модуль – Barcode Scanning API, який підтримує зчитування різноманітних кодів, включно з QR-кодами, штрихкодами формату EAN, UPC, і PDF417. Цей API дозволяє швидко обробляти коди навіть на середньопродуктивних пристроях завдяки вбудованим моделям, що працюють безпосередньо на пристрої, що особливо важливо для роботи в режимі офлайн. До переваг ML Kit належить його здатність ефективно

зчитувати коди під різними кутами, навіть якщо зображення неідеально сфокусоване при недостатньому освітленні. Окрім цього, ML Kit підтримує розробників і надає детальну документацію для швидкої інтеграції API [28].

2.6 Розробка серверної частини застосунку

2.6.1 Розробка бази даних

Розробка серверної частини застосунку зазвичай починається зі створення надійного та організованого сховища даних, яке забезпечує зберігання й обробку інформації системи. Це сховище об'єднує й структурує дані, що дозволяє виконувати операції запису, читання та обробки, критично важливі для забезпечення функціональності мобільного додатка.

Схему розробленої бази наведено на рис. 2.3. Поля, за допомогою яких створюється зв’язок між документами різних коллекцій, на схемі виділено жирним шрифтом. Самі зв’язки позначені за допомогою кольорових ліній. Тип зв’язку “багато до багатьох” позначений за допомогою пунктиру.

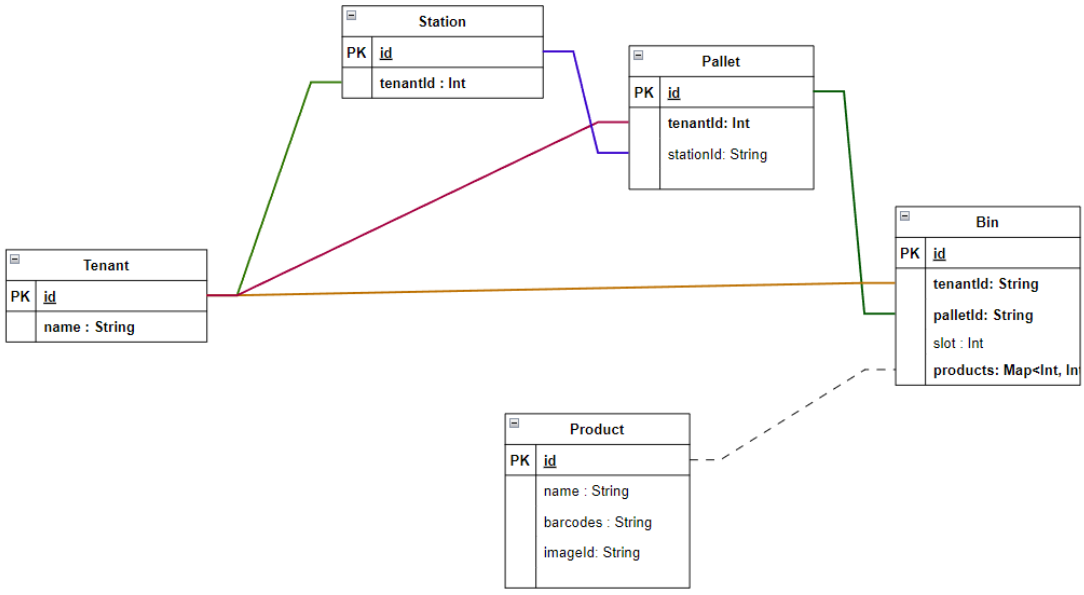

Рисунок 2.3 – Схема бази даних

На схемі представлено наступні сутності.

Tenant. Містить основну інформацію про установу, включаючи унікальний ідентифікатор (id) та назву (name). Сутність є базовою і використовується для прив’язки інших елементів до конкретної установи.

Product. Містить інформацію про товари, унікальний ідентифікатор (id), назву (name), штрихкод, що ідентифікує товар, та ідентифікатор зображення (imageId).

Bin. Представляє контейнери для зберігання продукції. Містить унікальний ідентифікатор (id), ідентифікатор орендаря (tenantId), ідентифікатор піддона (palletId), якщо контейнер прив’язаний до піддона, номер слоту (slot) і асоціативний масив товарів (products), де ключем є id товару, а значенням – кількість одиниць.

Pallet. Містить інформацію про піддони. Включає унікальний ідентифікатор (id), ідентифікатор орендаря (tenantId), а також ідентифікатор станції (stationId), якщо піддон прив’язаний до станції. Сутність використовується для обліку піддонів у системі.

Station. Представляє робочі станції, на яких проводяться операції. Містить унікальний ідентифікатор (id) та ідентифікатор орендаря (tenantId). Ця сутність забезпечує зв’язок між фізичними станціями та логічними операціями у системі.

Для розробки сховища був обраний об'єктний підхід замість використання традиційних SQL-запитів. Це забезпечує більшу гнучкість, зручність у використанні та спрощує підтримку коду. Завдяки інкапсуляції даних та логіки у вигляді класів і об'єктів, такий підхід дозволяє легко керувати й маніпулювати інформацією безпосередньо в програмному коді, не звертаючись до бази даних при кожній операції. Об'єктно-орієнтований підхід також спрощує тестування та модифікацію коду, оскільки дозволяє працювати з об'єктами як з цілими сутностями, зберігаючи зв'язок між даними та поведінкою в одному місці.

Клас Store, код якого приведений у лістингу 2.1, слугує для зберігання та управління списками різних об'єктів, пов'язаних із внутрішніми ресурсами додатку, таких як продукти, установи, контейнери, палети, станції, задачі та станції приймання. Він представляє кожен тип сутності (наприклад, Product, Tenant, Bin)

як окремий список. Це дозволяє централізовано керувати інформацією та полегшує доступ до даних, необхідних для бізнес-логіки програми.

Лістинг 2.1 – Класс Store

```
class Store {
    companion object {
        var products = mutableListOf<Product>()
        var tenants = mutableListOf<Tenant>()
        var bins = mutableListOf<Bin>()
        var pallets = mutableListOf<Pallet>()
        var stations = mutableListOf<Station>()
        var issues = mutableListOf<Issue>()
        val tasks = mutableListOf<Task>()
        var inboundStations = mutableListOf<IbsContext>()

        fun getOpenIssues(): List<Issue> {
            val openIssues = mutableListOf<Issue>()
            for (issue in issues) {
                openIssues.removeIf { it.tenantId == issue.tenantId
&& it.entityType == issue.entityType && it.entityId ==
issue.entityId }
                if (issue.operation == IssueOperation.Report) {
                    openIssues.add(issue)
                }
            }  return openIssues  }  }}
```

Метод getOpenIssues() визначає всі "відкриті" проблеми з колекції issues, усуваючи дублікати на основі параметрів tenantId, entityType та entityId, та залишає лише ті, що мають статус Report.

2.6.2 Розробка контролерів

Головна логіка програми реалізована з використанням контролерів в архітектурі Spring MVC. Контролери, що позначаються аннотацією @RestController в Spring Framework, реалізують обробку HTTP-запитів (GET,

POST, PUT, DELETE) та забезпечують інтерфейс для взаємодії з клієнтами (наприклад, мобільними додатками або веб-інтерфейсом). Вони виконують функцію “посередника” між клієнтською стороною і логікою програми, обробляючи запити, виконуючи необхідні операції та повертаючи відповідні відповіді, зазвичай у форматі JSON чи XML. У якості прикладу розглянемо клас ProductController, приведений у лістингу 2.2.

Лістинг 2.2 - Код класу ProductController

```
@RestController
@RequestMapping("/products")
class ProductController {
    @GetMapping
    fun getProducts(
        @RequestParam(required = false) name: String?,
        @RequestParam pageSize: Int,
        @RequestParam page: Int,
        ): PageResponse<Product> {
        var products = Store.products.toMutableList()
        if(!name.isNullOrBlank()) {
            products = products.filter { it.name.startsWith(name)
}.toMutableList()
        }
        val productsPage = products.sortedBy { it.name
}.drop((page - 1) * pageSize).take(pageSize)
        return PageResponse(products.count(), productsPage)
    }

    @GetMapping("/{productId}")
    fun getProduct(@PathVariable productId: String): Product? {
    return Store.products.find { it.id == productId.toInt() }}
}
```

Клас ProductController надає інтерфейс для доступу до ресурсів продуктів, а також виконує функції фільтрації, сортування та пагінації. Перший метод,

getProducts, дозволяє отримати перелік продуктів з можливістю фільтрації за параметром name, сортуванням за назвою та пагінацією, яка визначається параметрами pageSize та page. Метод повертає результат у вигляді об'єкта PageResponse, що містить як загальну кількість продуктів, так і підмножину даних, відповідно до заданих параметрів. Другий метод, getProduct, дозволяє отримати інформацію про конкретний продукт за допомогою унікального ідентифікатора productId.

Клас OperationController реалізовує основний функціонал, пов’язаний зі скануванням об’єктів у системі, та надає методи доступу й оновлення таких сутностей, як контейнери, станції, палети, слоти. У лістингу 2.3 наведений метод scan, який був розроблений для ідентифікації типу об’єкта на основі унікального тега, що передається в параметрах запиту. У додатку А.1 наведений повний код цього класу.

Лістинг 2.3 - Код методу scan у OperationController

```
@GetMapping("/scan")
fun scan(@RequestParam tenantId: Int, @RequestParam tag: String): 
ScanDto {
    if (tag.isBlank()) {
        throw BadRequestException("Tag cannot be empty")
   } when {
        tag.startsWith(binTagPrefix, ignoreCase = true) -> {
            val bin = bins.find { it.tenantId == tenantId && it.id 
== tag.substring(binTagPrefix.length) }
            return ScanDto(EntityType.Bin, bin?.toDto())
        }
        tag.startsWith(stationTagPrefix, ignoreCase = true) -> 
{
            val station =
                stations.find { it.tenantId == tenantId && it.id == 
tag.substring(stationTagPrefix.length) }
            return ScanDto(EntityType.Station, station?.toDto())
```

```
        }        tag.startsWith(palletTagPrefix, ignoreCase = true)
-> {
            val pallet = pallets.find { it.tenantId == tenantId &&
it.id == tag.substring(palletTagPrefix.length) }
            return ScanDto(EntityType.Pallet, pallet?.toDto())
        }

        tag.startsWith(slotTagPrefix, ignoreCase = true) -> {
            val regexMatches = Regex("^([^.]+)\\.([0-5])$")
                .find(tag.substring(slotTagPrefix.length))
                ?: throw BadRequestException("Invalid slot format")
            val palletId = regexMatches.groups[1]?.value
            val slot = regexMatches.groups[2]?.value!!.toInt()
            val pallet = pallets.find { it.tenantId == tenantId &&
it.id == palletId }
            return ScanDto(EntityType.Slot, SlotDto(slot,
pallet?.toDto()))
        }

        else -> {
            val product = Store.products.find { product ->
product.barcodes.any { barcode -> barcode == tag } }
            return ScanDto(EntityType.Product, product)
        }
    }
}
```

Метод scan приймає два параметри – tenantId та tag, що дозволяє визначити, до якої установи та типу об’єкта належить відсканований тег. Тег розпізнається за префіксами, і, в залежності від типу, здійснюється пошук відповідного об’єкта в локальних списках об’єктів bins, stations, pallets, або ж об'єктів з індивідуальними кодами продукції у Store.products. Наприклад, якщо тег починається з префікса контейнера, метод scan шукає об’єкт контейнера у відповідному списку, створюючи об'єкт типу ScanDto для подальшого використання в програмі.

Окрім цього в класі OperationController представлені наступні методи:

– updateBin оновлює інформацію про конкретний контейнер (bin) на основі його ID, параметрів користувача (tenantId), і нових даних, що передаються у запиті. Перевіряє наявність контейнера, валідність переданих параметрів, та оновлює інформацію про палету і слот контейнера. Якщо контейнер додається до палети, видаляє його зі станції приймання;

– getBinProduct повертає інформацію про продукт у контейнері за ID контейнера (binId) і продукту (productId). Перевіряє наявність контейнера та продукту в системі, а також чи міститься продукт у контейнері;

– createOrUpdateBinProduct додає або оновлює кількість конкретного продукту у контейнері. Якщо кількість продукту дорівнює нулю, продукт видаляється з контейнера;

– getPallets повертає список палет із врахуванням параметрів фільтрації: сторінки, розміру сторінки, чи палета вільна (floating), та чи має активні проблеми (flagged);

– getPallet повертає детальну інформацію про конкретну палету за її ID, з можливістю включення деталей про контейнери і продукти;

– updatePalletBins оновлює зв'язки між палетою та контейнерами, перевіряючи валідність контейнерів і кількість контейнерів на кожному слоті;

– updatePalletSlotBins оновлює контейнери для конкретного слоту палети, перевіряючи кількість контейнерів (1 або 2) та валідність слоту.

Кожен з цих методів забезпечує обробку помилок, що включає перевірку вхідних параметрів, наявність ресурсів та відповідність бізнес-логіці.

Для реалізації функціоналу станції приймання був розроблений клас HardwareController повний код якого наведений у лістингу А.2. Цей клас відповідає за інтеграцію з даними станцій приймання та дозволяє управління цими даними через різні HTTP-запити, зокрема PATCH, GET і PUT.

Крім того, він інтегрується з сервісом IbsSocketMessageService, що дозволяє синхронізувати дані між сервером та станціями. Враховуючи об’єм коду, у лістингу 2.4 як приклад наведено лише декілька методів контролеру.

Метод updateIbsContext використовує HTTP-запит PATCH і дозволяє оновити контекст станції приймання, ідентифікованої через ibsId, а також обробляти продукт, пов’язаний із запитом. Метод здійснює пошук станції в Store.inboundStations, а якщо така станція не знайдена, викидається помилка BadRequestException. Якщо переданий productId є валідним і відповідає існуючому продукту, станції присвоюється цей продукт. Метод повертає оновлений контекст IbsContext, відображаючи останні зміни на станції.

Лістинг 2.4 - Методи класу HardwareController

```
@RestController
@RequestMapping("/origin/hardware")
class HardwareController(@Autowired private val socket:
IbsSocketMessageService) {

    @PatchMapping("/inbound/ibs/{ibsId}")
    fun updateIbsContext(
        @RequestParam tenantId: Int,
        @PathVariable ibsId: String,
        @RequestBody update: InboundContextUpdate): IbsContext {
        val ibsContext = Store.inboundStations.find { it.tenantId ==
tenantId && it.id == ibsId }
        if(ibsContext == null) {
            throw BadRequestException("Inbound station not found")
        }
        if(update.productId != null) {
            val product = Store.products.find { product ->
product.id == update.productId }
            if(product == null) {
                throw BadRequestException("Product not found")
            }
            ibsContext.product = product
        }
        return ibsContext
    }
```

```
    @GetMapping("/inbound/ibs/{ibsId}")
    fun getIbsContext(
        @RequestParam tenantId: Int,
        @PathVariable ibsId: String): IbsContext {
        val ibsContext = Store.inboundStations.find { it.tenantId ==
tenantId && it.id == ibsId }
        if(ibsContext == null) {
            throw BadRequestException("Inbound station not found")
        }
        return ibsContext
    }
}
```

Функція getIbsContext реалізує HTTP-запит GET для отримання актуального контексту конкретної станції приймання, визначеної параметрами tenantId та ibsId. Якщо відповідна станція не знайдена, викидається виняток BadRequestException, який повідомляє про те, що станцію не знайдено. Повертається об’єкт IbsContext, що містить актуальну інформацію про станцію.

PUT-метод createOrUpdateIbsBinProduct відповідає за створення або оновлення даних продуктів у контейнері станції приймання. Він перевіряє наявність продукту та контейнера, генерує виняток у разі їх відсутності, оновлює кількість продукту та синхронізує дані через IbsSocketMessageService.

IbsSocketMessageService – це сервіс, створений засобами Spring, який використовує SimpMessagingTemplate для реалізації обміну повідомленнями в реальному часі. Він відповідає за надсилання повідомлень через сокет, забезпечуючи інтерактивну комунікацію між сервером і клієнтами, які підписані на певні теми WebSocket. Код IbsSocketMessageService можна побачити у лістингу 2.5.

Лістинг 2.5 – Код IbsSocketMessageService

```
@Service
class IbsSocketMessageService(@Autowired private val template:
SimpMessagingTemplate) {
```

```
    fun sendContextToIbsTopic(ibsContext: IbsContext) {

template.convertAndSend("/topic/ibs/${ibsContext.tenantId}/${ibsCont
ext.id}/context", ibsContext)
    }
    fun sendStatusToIbsTopic(tenantId: Int, ibsId: String,
connected: Boolean) {
    template.convertAndSend("/topic/ibs/$tenantId/$ibsId/connected",
connected)
    }
}
```

Основна функція цього сервісу полягає в тому, щоб підтримувати актуальність інформації, передаючи зміни даних станції приймання у режимі реального часу. Метод sendContextToIbsTopic надсилає актуальний контекст конкретної станції ibsContext у WebSocket-тему /topic/ibs/{tenantId}/{ibsId}/context, яку клієнти можуть відстежувати. Це дозволяє швидко синхронізувати дані між сервером і клієнтами, передаючи оновлену інформацію про поточний стан станції приймання.

Метод sendStatusToIbsTopic використовується для відправлення оновлень статусу підключення станції. Цей метод передає дані про підключення або відключення станції, що зберігає поточний стан у WebSocket-темі /topic/ibs/{tenantId}/{ibsId}/connected. Завдяки такій архітектурі, клієнти, підписані на цю тему, можуть отримувати миттєві повідомлення про зміни через механізм оновлення даних станції приймання в реальному часі.

2.7 Розробка модулів клієнтської частини

Розробка комп’ютерної системи складається з розробки мобільного застосунку в IDE Android Studio за допомогою UI-фреймворку Jetpack Compose та

мови програмування Kotlin, а також впровадження системи генерації та сканування QR-кодів на базі бібліотек Zxing і ML Kit.

2.7.1 Опис основних класів системи

Клас MainActivity, код якого приведений у лістингу 2.6, є центральним компонентом додатку і відповідає за управління інтерфейсом користувача та його основною логікою. Цей клас ініціалізує ключові сервіси, необхідні для коректної роботи додатку, включаючи ViewModel та різні залежності для забезпечення локалізації, сканування штрихкодів та обробки дозволів.

У методі onCreate, MainActivity виконує налаштування основних компонентів, зокрема, налаштовує заставку через метод installSplashScreen, прив'язує об'єкт дозволів до поточної активності та ініціалізує локалізацію. Далі активується сканер QR-кодів zebraScanner, який є частиною інтегрованого модуля для зчитування кодів.

Лістинг 2.6 - Код MainActivity.kt

```
class MainActivity : ComponentActivity() {

    private val mainViewModel: MainViewModel by viewModel()
    private var splashScreen: SplashScreen? = null

    private val zebraScanner: ZebraScanner by inject()
    private val permissions: Permissions by inject<Permissions>()
    private val localization: Localization by inject()

    private val strings: Strings by inject<Strings>()
    override fun onCreate(savedInstanceState: Bundle?) {
        splashScreen = installSplashScreen()
        super.onCreate(savedInstanceState)
        (permissions as? AndroidPermissions)?.attachActivity(this)
        localization.attach(this)
        zebraScanner.startScan()
        splashScreen?.setKeepOnScreenCondition { false }
        setContent {
```

```
            App(startDestination = Destination.Splash)
        }
    }

    override fun onDestroy() {
        zebraScanner.release()
        (permissions as? AndroidPermissions)?.detach()
        super.onDestroy()
    }

    @Composable
    fun App(
        navController: NavHostController = rememberNavController(),
        drawerState: DrawerState = rememberDrawerState(initialValue
= DrawerValue.Closed),
        startDestination: Destination
    )
```

За допомогою Dependency Injection бібліотеки Koin, MainActivity отримує об'єкти, необхідні для функціонування, зокрема ZebraScanner, Permissions, Localization та Strings. Це дозволяє легко керувати залежностями та робити код більш гнучким і тестованим. Метод setContent викликає App, який виконує відображення навігаційної структури додатку. App приймає параметри, що містять контролер навігації NavHostController та стан бокового меню DrawerState. ZebraScanner активно зчитує штрихкоди, ініціюючи сканер при запуску активності. У методі onDestroy цей компонент звільняється, щоб оптимізувати використання ресурсів. Локалізація інтегрована через потоки contextFlow, що забезпечує зміну мови інтерфейсу в режимі реального часу. В кінці класу MainActivity можна побачити секцію з навігаційними ефектами, код якої приведений у лістингу 2.7.

Лістинг 2.7 - Код Navigation Effects

```
@Composable
fun NavigationEffects(
```

```
    navigationChannel: Channel<NavigationIntent>,
    navHostController: NavHostController
) {
    val activity = (LocalContext.current as? Activity)
    LaunchedEffect(activity, navHostController, navigationChannel) {
        navigationChannel.receiveAsFlow().collect { intent ->
            if (activity?.isFinishing == true) {
                return@collect
            }
            when (intent) {
                is NavigationIntent.NavigateBack -> {
                    if (intent.route != null) {
                        navHostController.popBackStack(intent.route,
intent.inclusive)
                    } else {
                        navHostController.popBackStack()
                    }
                }
                is NavigationIntent.NavigateTo -> {
                    navHostController.navigate(intent.route) {
                        launchSingleTop = intent.isSingleTop
                        intent.popUpToRoute?.let { popUpToRoute ->
                            popUpTo(popUpToRoute) { inclusive =
intent.inclusive }
                        }
                    }}
                is NavigationIntent.NavigateToWithClearBackStack ->
{
                    navHostController.navigate(intent.route) {
                        launchSingleTop = intent.isSingleTop
                        popUpTo(navHostController.graph.id) {
                            inclusive = true
                        }
                    }}
            } }
}
```

NavigationEffects використовує канал navigationChannel із mainViewModel для обробки навігаційних подій. Це дозволяє MainActivity керувати навігацією між екранами, реагуючи на події типу NavigateBack, NavigateTo, і NavigateToWithClearBackStack, та надає користувачу інтуїтивний та швидкий доступ до різних розділів додатку

Для підтримки декількох мов в додатку був розроблений клас LocalizationImpl, код якого приведений у лістингу 2.8. Він реалізує інтерфейс Localization та забезпечує управління мовними налаштуваннями. Його основна функція – динамічна зміна мови інтерфейсу. Клас містить потік contextFlow, який передає поточний контекст додатку через MutableStateFlow, забезпечуючи актуальний контекст у компонентах, що відображають інтерфейс користувача.

Лістинг 2.8 – Клас LocalizationImpl

```
class LocalizationImpl : Localization {

    private var context: Context? = null
    private var activity: Context? = null

    private val _contextFlow = MutableStateFlow<Context?>(null)
    override val contextFlow: Flow<Context?> =
_contextFlow.asStateFlow()
    override fun attach(context: Context) {
        this.context = context
        activity = context
        _contextFlow.value = context
    }

    @Suppress("DEPRECATION")
    override fun setLocale(locale: String) {
        val context = context ?: throw
IllegalStateException("Context is not attached")
        val resources = context.resources
        if (resources.configuration.locales[0].language == locale) {
```

```
            return
        }
        val configuration = resources.configuration
        val newLocale = Locale(locale)
        configuration.setLocale(newLocale)
        configuration.setLayoutDirection(newLocale)
        val newContext =
context.createConfigurationContext(configuration)
        newContext.resources.updateConfiguration(configuration,
resources.displayMetrics)

        activity?.resources?.updateConfiguration(configuration,
resources.displayMetrics)
        this.context = newContext
        _contextFlow.value = newContext
    }
}
```

Метод attach встановлює контекст, зберігаючи його та оновлюючи contextFlow, що дозволяє змінювати контекст додатку в процесі роботи. Метод setLocale відповідає за зміну мови, перевіряючи, чи збігається поточна мова інтерфейсу з обраною, і за необхідності оновлює Locale та ресурси, щоб зміни відобразились у користувацькому інтерфейсі.

Для реалізації класу що сканує штрих-коди у реальному часі було створено ZebraScannerImpl, код якого приведений у лістингу 2.9.

Лістинг 2.9 – Клас ZebraScannerImpl

```
class ZebraScannerImpl(context: Context) : ZebraScanner,
ImageAnalysis.Analyzer {

    private val _scans = MutableSharedFlow<String>(
        extraBufferCapacity = 1,
        onBufferOverflow = BufferOverflow.DROP_OLDEST
    )
```

```
    override val scans: Flow<String> = _scans.asSharedFlow()
   private var lastScanTimestamp = 0L
        @SuppressLint("UnsafeOptInUsageError")
    override fun analyze(imageProxy: ImageProxy) {
        if (lastScanTimestamp + SCAN_TIMEOUT >
System.currentTimeMillis()) {   imageProxy.close()
            return
        }
        try {
            val options = BarcodeScannerOptions.Builder()
                .setBarcodeFormats(
                    Barcode.FORMAT_QR_CODE)
                .build()
            val scanner = BarcodeScanning.getClient(options)
            val image = imageProxy.image

            if (image?.format == ImageFormat.YUV_420_888) {
                val inputImage =
                    InputImage.fromMediaImage(image,
imageProxy.imageInfo.rotationDegrees)
                scanner.process(inputImage)
                    .addOnSuccessListener { barcodes ->
                    barcodes.firstOrNull()?.rawValue?.also {
         lastScanTimestamp = System.currentTimeMillis()
              _scans.tryEmit(it)}}.addOnFailureListener {
                        it.printStackTrace()  }
                    .addOnCompleteListener {
                        imageProxy.close()} } else {
                imageProxy.close()}
        } catch (exc: Exception) {
            exc.printStackTrace()
            imageProxy.close()
        } }
    companion object {
        private const val SCAN_TIMEOUT = 3000L } }
```

ZebraScannerImpl реалізує інтерфейс ZebraScanner і виконує аналіз зображення на основі API CameraX. Основним завданням цього класу є обробка зображень, отриманих з камери, та зчитування штрих-кодів із застосуванням бібліотеки MLKit для подальшої обробки сканованих даних.

Клас має приватне поле _scans, що є об’єктом MutableSharedFlow, який використовує механізм реактивного програмування для поширення результатів сканування серед підписників. Це поле зберігає останні зчитані дані, дозволяючи отримувати їх навіть у разі переповнення буферу. Основний метод analyze, який отримує зображення через інтерфейс ImageAnalysis.Analyzer, виконує перевірку часу останнього сканування для запобігання повторного зчитування штрих-коду занадто часто.

Метод analyze перевіряє, чи передане зображення відповідає формату YUV_420_888. Якщо так, створюється InputImage із заданим обертанням зображення, що забезпечує коректну орієнтацію штрих-коду для сканера. Далі налаштовується об’єкт BarcodeScannerOptions, який визначає формат штрих-кодів для зчитування. MLKit-сканер обробляє inputImage та викликає відповідні методи на випадок успіху, невдачі або завершення обробки.

Метод onSuccessListener зчитує перший знайдений штрих-код та записує його значення у MutableSharedFlow _scans, яке можна отримати через scans. Завдяки цьому, всі підписники зможуть отримати результат сканування, що дозволяє обробляти зчитані штрих-коди у реальному часі без необхідності постійного зберігання результатів у змінних.

2.7.2 Огляд класів для генерації PDF-звітів з QR-кодами

Важливою частиною функціоналу додатку є генерація PDF-звітів які містять QR-коди. За генерацію саме QR-кодів у проекті відповідальний клас QRCodeRepositoryImpl, код якого приведений у лістингу 2.10.

Лістинг 2.10 - Код класу QRCodeRepositoryImpl

```
class QRCodeRepositoryImpl : QRCodeRepository {
```

```
    override fun generateProductsQRCodes(codeData:
Array<ProductPrintLabel?>, out: Array<Bitmap?>) {

for (index in codeData.indices) {
            codeData[index]?.also { data ->
                out[index] = encodeAsBitmap(data.codeData, 220)
  }
        }
    }
    @Throws(WriterException::class)
    private fun encodeAsBitmap(str: String, codeWidth: Int): Bitmap
{
        val writer = QRCodeWriter()
        val bitMatrix = writer.encode(
            str,
            BarcodeFormat.QR_CODE,
            codeWidth,
            codeWidth,
            mapOf<EncodeHintType, Int>(EncodeHintType.MARGIN to 0)
        )
        val w = bitMatrix.width
        val h = bitMatrix.height
        val pixels = IntArray(w * h)

for (y in 0 until h) {
            for (x in 0 until w) {
                pixels[y * w + x] = if (bitMatrix[x, y]) Color.BLACK
else Color.WHITE
            }
        }
        val bitmap = Bitmap.createBitmap(w, h,
Bitmap.Config.ARGB_8888)
        bitmap.setPixels(pixels, 0, w, 0, 0, w, h)
        return bitmap
    }
}
```

Метод generateProductQRCodes генерує QR-коди для масиву продукції на основі переданого масиву ProductPrintLabel. Використовуючи цикл, метод кодує дані для кожного товару та зберігає відповідні QR-коди у вихідному масиві.

Приватний метод encodeAsBitmap використовуючи бібліотеку ZXing (QRCodeWriter), метод генерує BitMatrix, представляючи QR-код. Далі, для кожного пікселя у BitMatrix, створюється масив pixels, де кожному пікселю відповідає колір чорний або білий в залежності від значення в BitMatrix. З отриманих даних створюється об'єкт Bitmap, який і повертається як результат методу.

Інтерфейс QRCodeRepository є лише частиною процесу випадку використання GenerateProductLabelsPdfUseCase, код якого приведений у лістингу 2.11.

Клас GenerateProductLabelsPdfUseCase відповідає за створення PDF-документу з етикетками продукції. У процесі роботи він генерує QR-коди, додає сторінки з інформацією про товари до PDF-файлу та завершує запис при необхідності.

Лістинг 2.11 - Код класу GenerateProductLabelsPdfUseCase

```
class GenerateProductLabelsPdfUseCase(
    coroutineContext: CoroutineContext,
    errorHandler: ErrorHandler,
    private val repository: QRCodeRepository,
    private val pdfConverter: PDFConverter
) : UseCase<GenerateProductLabelsPdfUseCase.Param, Unit>(
    coroutineContext = coroutineContext,
    errorHandler = errorHandler
) {
    override suspend fun executeActual(param: Param) {
        val pageSize = 4
        param.apply {
           val mod = plantsPrintLabels.size % pageSize
           val sizeWithoutMod = plantsPrintLabels.size - mod
```

```
            val subdata = Array<ProductPrintLabel?>(pageSize) { null
}
             val codes = Array<Bitmap?>(pageSize) { null }
             for (i in 0 until plantsPrintLabels.size step pageSize)
{
                 if (i < sizeWithoutMod) {
                     for (j in 0 until pageSize) {
                         subdata[j] = plantsPrintLabels[i + j]
                     }
                 } else {
                     for (j in 0 until mod) {
                         subdata[j] = plantsPrintLabels[i + j]
                     }
                 }
                 repository.generatePlantsQRCodes(codeData = subdata,
out = codes)
                 pdfConverter.addProductLabelPage(
                     data = subdata,
                     codeImages = codes,
                     pdfDocument = pdfDocument
                 )
                 for (k in 0 until pageSize) {
                     codes[k] = null
                     subdata[k] = null }}
             if (finishWriting) {
                 pdfConverter.finishWriting(pdfDocument)
             }
     }
 }
    data class Param(
        val plantsPrintLabels: MutableList<ProductPrintLabel>,
        val pdfDocument: PdfDocument,
        val finishWriting: Boolean
    )
}
```

GenerateProductLabelsPdfUseCase генерує QR-код через QRCodeRepository, отримує його зображення та передає разом із даними етикетки в pdfConverter. Сторінки додаються методом pdfConverter.addProductLabelPage, а завершення запису PDF здійснюється через pdfConverter.finishWriting, якщо finishWriting = true.

Клас GenerateProductLabelsPdfUseCase виступає посередником між генерацією QR-кодів, формуванням вмісту PDF і записом у файл, завершуючи процес через PDFConverter код якого можна побачити у лістингу А.3.

Клас PDFConverter відповідає за створення PDF-файлів з етикетками для продуктів, використовуючи надані дані та QR-коди.

– createBitmapFromProductLabelView: Створює зображення (Bitmap) з етикеткою для продукту на основі переданої макетної (layout) View, даних продукту та зображення QR-коду;

– createBitmap: Створює зображення (Bitmap) з переданої макетної View та розмірами;

– addProducttLabelPage: Додає сторінки до PDF-документу для етикеток продуктів, створюючи зображення для кожного продукту та розташовуючи його на сторінці PDF;

– finishWriting: Завершує запис у PDF-документ, записує його у файл та відображає результат;

– renderPdf: Відображає PDF-файл за допомогою зовнішнього переглядача;

– rotateBitmap: Обертає зображення на 270 градусів.

2.8 Висновки щодо розробки розподіленої системи

У другому розділі при розробці розподіленої системи було здійснено комплексний підхід до планування, проектування та реалізації системи. Ключовою метою стало створення функціонального рішення, яке забезпечить інтеграцію

бізнес-процесів, зручний користувацький досвід і високу продуктивність. Проект складався з декількох взаємопов'язаних етапів, що охоплюють різні аспекти розробки програмного забезпечення.

У першу чергу були визначені технічні характеристики системи, зокрема вимоги до продуктивності серверного та клієнтського компонентів. Це допомогло підібрати необхідні інструменти та ресурси, що забезпечать стабільну роботу додатку навіть за високих навантажень. Також були враховані вимоги до захисту даних, що дозволило створити безпечне середовище для зберігання та обробки інформації.

На підставі опису функціональності програмного забезпечення було визначено функціональність інтерфейсу мобільного додатку та виконано моделювання ключових сутностей системи, таких як установи, продукти, контейнери, палети, а також елементів, пов'язаних зі станціями та слотами. Це дозволило створити гнучкий та багатофункціональний інтерфейс, який відповідає запитам кінцевих користувачів.

Архітектура системи була ретельно пророблена з метою забезпечення її стабільності та масштабованості. Проведений огляд бібліотеки для імплементації системи сканування та генерації QR-кодів дозволив визначити найкращий варіант для використання в системі з урахуванням функціональних потреб.

У процесі розробки серверної частини було створено централізоване управління різними типами даних, такими як продукти, установи, контейнери, палети за допомогою об'єктно-орієнтованого підходу. Це дозволило зручно організувати дані, спростити доступ до них та полегшити підтримку коду. Методи контролерів були розроблені з урахуванням вимог безпеки, перевірки валідності даних та обробки помилок, що сприяло створенню стабільної та безпечної системи.

Розробка клієнтської частини велася із застосуванням сучасних інструментів і технологій, які дозволили ефективно забезпечити функціонал генерації PDF-звітів з QR-кодами на стороні мобільного додатку.

У результаті виконання поставлених завдань було створено надійну систему, яка відповідає сучасним вимогам до продуктивності, безпеки та зручності

використання. Усі етапи розробки були реалізовані з урахуванням перспектив розвитку та інтеграції системи в реальні умови експлуатації.

3 ДОСЛІДЖЕННЯ РОЗПОДІЛЕНОЇ СИСТЕМИ

3.1 Дослідження споживання мобільних ресурсів під час генерації звіту

Для успішної реалізації вимоги щодо генерації PDF-звіту з QR-кодами на мобільному пристрої та оцінки продуктивності на трьох моделях смартфонів (Google Pixel 4a, Redmi 9A, Samsung Galaxy S21), важливо дослідити наступні параметри:

- час обробки, необхідний для генерації звіту;
- використання ресурсів процесора;
- обсяг оперативної пам’яті, який споживає процес;
- визначення меж продуктивності пристроїв для забезпечення комфортного використання програми.

Для реалізації завдання буде використано Android Profiler в Android Studio. Цей інструмент дозволяє аналізувати використання CPU, пам'яті та інших ресурсів, а також відстежувати поведінку додатку в реальному часі.

Google Pixel 4a оснащений процесором Qualcomm Snapdragon 730G із 64-бітною архітектурою Kryo 470. Він має два продуктивні ядра Cortex-A76 (до 2,2 ГГц) і шість енергоефективних Cortex-A55 (до 1,8 ГГц). Чип виготовлений за 8-нм техпроцесом, що забезпечує енергоефективність. Графіку обробляє Adreno 618, підтримуючи API OpenGL ES 3.2, Vulkan 1.1 і DirectX 12 для плавної роботи та якісної HDR-графіки. Результати роботи наведені в таблиці 3.1, та на рисунках 3.1, 3.2 та 3.3. Пристрій оснащений 128 ГБ вбудованої пам’яті UFS 2.1 для зберігання даних, хоча можливості розширення через microSD немає. ОЗП 6 ГБ (LPDDR4x) із частотою до 2133 МГц сприяє швидкій роботі в багатозадачному режимі.

Його 5,81-дюймовий OLED-дисплей із роздільною здатністю 2340 x 1080 пікселів підтримує HDR10, забезпечуючи яскраву і чітку картинку із щільністю ~443 PPI.

Таблиця 3.1 – Результати вимірювання продуктивності системи залежно від кількості оброблюваних продуктів при використанні Google Pixel 4a

Кількість продуктів	CPU, %	RAM, мб	Час, сек
50	17	220	1.2
100	18	235	2.0
200	16	308	4.0
300	17	394	5.9
400	17	373	7.8
500	19	453	9.6
600	18	528	11.4
1000	17	777	19.2
2000	21	1432	39
2000	21	1432	39
3000	21	1910	57.9
4000	24	2435	77.8
5000	24	3008	91.1
6000	23	3678	119.2
7000	25	4210	164.9
7500	19	4500	167.6
8000	24	4784	crash

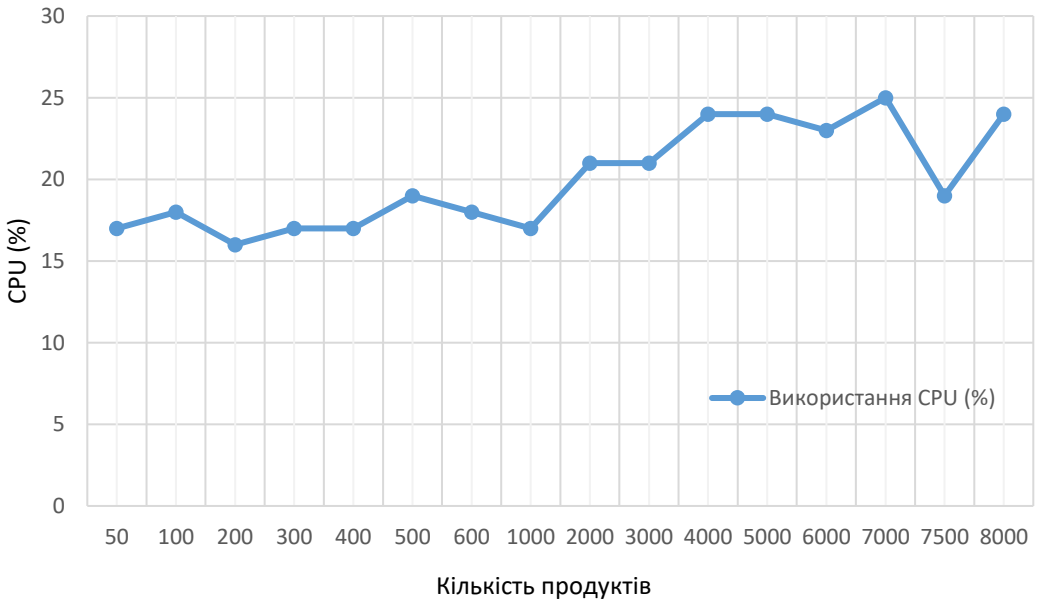

Рисунок 3.1 – Залежність CPU від кількості продуктів на Google Pixel 4a

Рисунок 3.2 – Залежність RAM від кількості продуктів на Google Pixel 4a

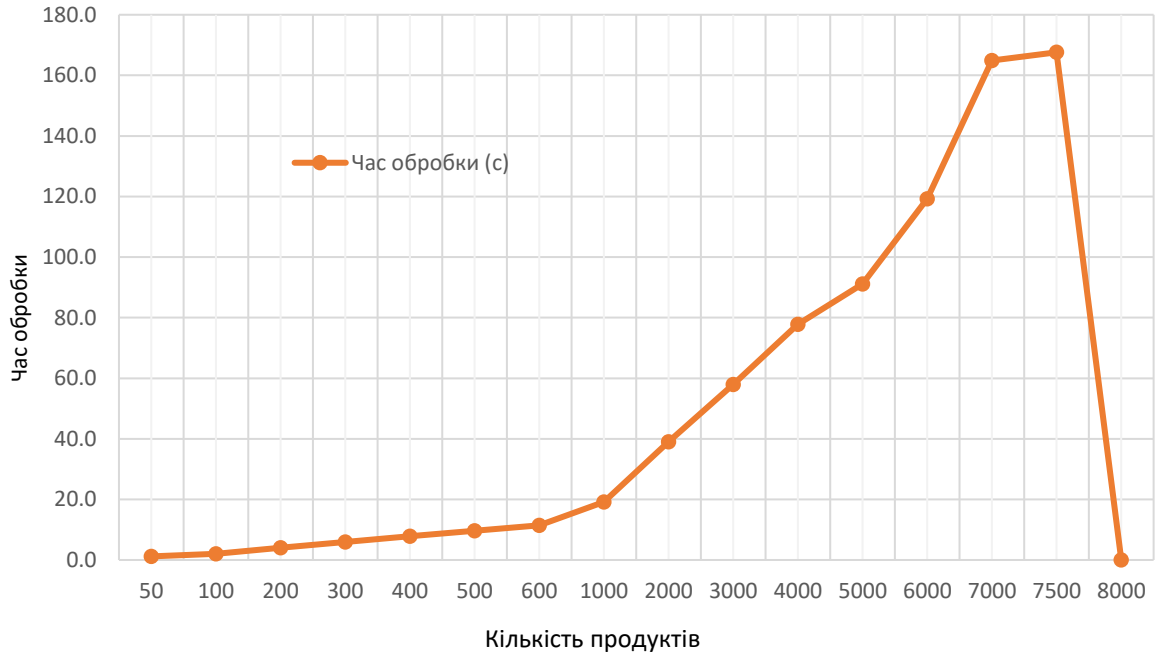

Рисунок 3.3 – Залежність часу обробки від кількості продуктів на Google Pixel 4a

Завантаження CPU під час дослідження залишалося стабільним, у межах 17–25%, навіть при обробці 7500 продуктів. Це свідчить, що продуктивність процесора Qualcomm Snapdragon 730G достатня для виконання цих задач. Google Pixel 4a має 6 ГБ RAM, але програма крашилася при споживанні пам’яті на рівні 4784 МБ, що становить приблизно 80% доступного обсягу. Це вказує на

критичну роль оперативної пам’яті для стабільної роботи програми. Рекомендовано обмежити використання пам’яті на цьому пристрої до 70–75% (4200–4500 МБ).

Час генерації звітів зростав із кількістю продуктів:

– для 3000 продуктів час складав 57.9 секунд, що вже на межі допустимого очікування;

– для 7500 продуктів час виконання досяг 167.6 секунд, що значно погіршує користувацький досвід.

Для цього пристрою рекомендується обмежувати кількість продуктів до 3000–5000, залежно від пріоритетів користувача.

Samsung Galaxy S21 FE 5G побудований на базі продуктивного процесора Qualcomm Snapdragon 888 (або Exynos 2100 для окремих регіонів). Він має 8 ядер: одне високопродуктивне ядро Cortex-X1 із частотою до 2,84 ГГц, три ядра Cortex-A78 із частотою до 2,42 ГГц і чотири енергоефективних ядра Cortex-A55 із частотою до 1,8 ГГц. Завдяки 5-нм техпроцесу забезпечується баланс між продуктивністю і енергоощадністю. Результати роботи наведені в таблиці 3.2, та на рисунках 3.4, 3.5 та 3.6.

Графічна підсистема представлена Adreno 660 (Snapdragon) або Mali-G78 MP14 (Exynos), що дозволяє запускати ресурсоємні ігри та програми з підтримкою HDR і сучасних API, таких як Vulkan 1.1.

Оперативна пам’ять представлена варіантами 6 ГБ або 8 ГБ стандарту LPDDR5, що гарантує швидку роботу системи. Обсяг вбудованого сховища становить 128 ГБ або 256 ГБ типу UFS 3.1, яке забезпечує високу швидкість доступу до даних, але можливості розширення через microSD немає.

Galaxy S21 FE працює на базі Android 12, з фірмовою оболонкою One UI 4.0, яка надає користувачеві зручний інтерфейс і широкі можливості кастомізації.

Смартфон оснащений 6,4-дюймовим Dynamic AMOLED 2X екраном із роздільною здатністю 2400 x 1080 пікселів, частотою оновлення 120 Гц і підтримкою HDR10+. Щільність пікселів становить ~411 PPI, що забезпечує чітке і яскраве зображення.

Таблиця 3.2 – Результати вимірювання продуктивності системи залежно від кількості оброблюваних продуктів при використанні Samsung Galaxy S21 FE 5G

Кількість продуктів	CPU, %	RAM, мб	Час, сек
50	20	227	0.7
100	23	251	1.0
200	24	313	2.0
300	21	349	2.9
400	21	411	3.8
500	21	479	4.7
600	22	533	5.7
1000	22	765	9.5
2000	21	1371	19.1
3000	22	1863	28.8
4000	23	2411	43.2
5000	21	3053	52.0
6000	22	3522	70.1
7000	22	4731	89.4
7500	22	4500	167.6
8000	22	4754	98.2
9000	19	5236	114.7
10000	22	5898	crash

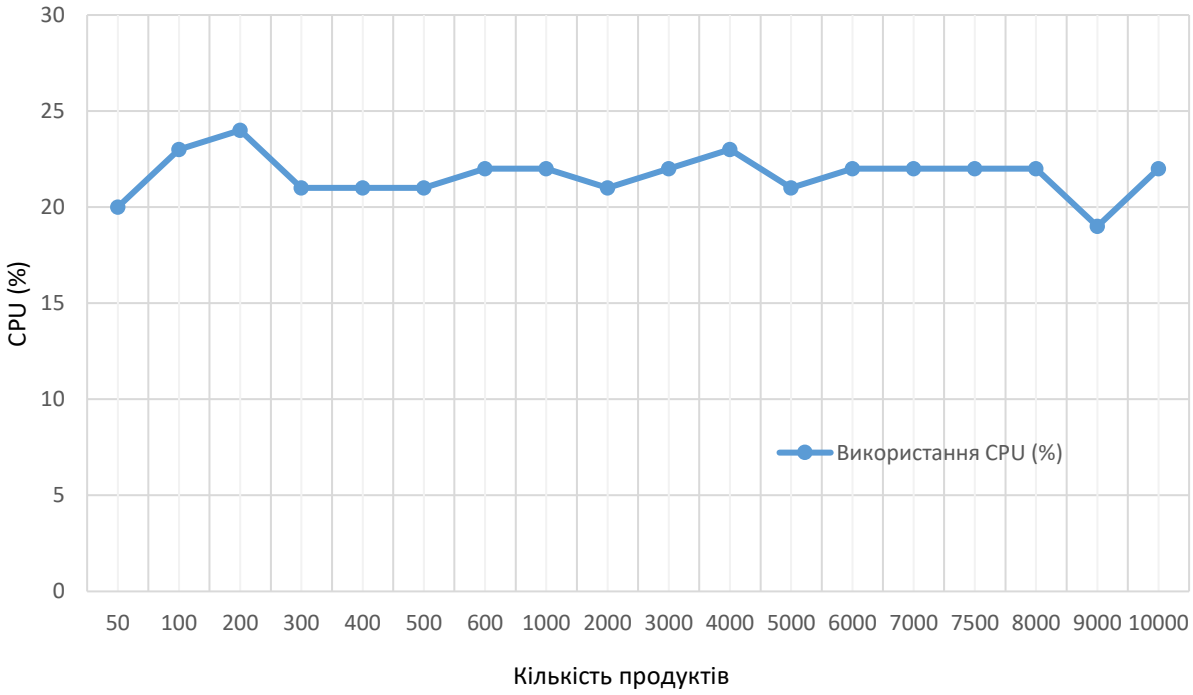

Рисунок 3.4 – Залежність CPU від кількості продуктів на Samsung Galaxy S21 FE 5G

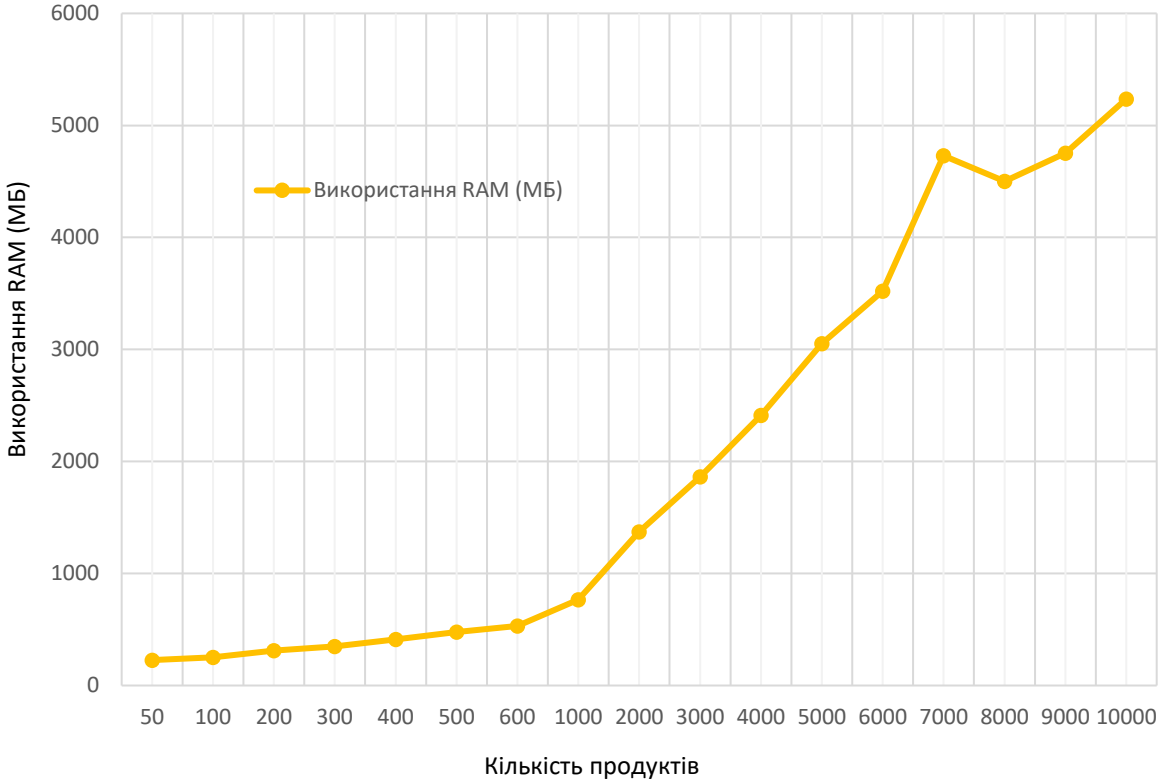

Рисунок 3.5 – Залежність RAM від кількості продуктів на Samsung Galaxy S21 FE 5G

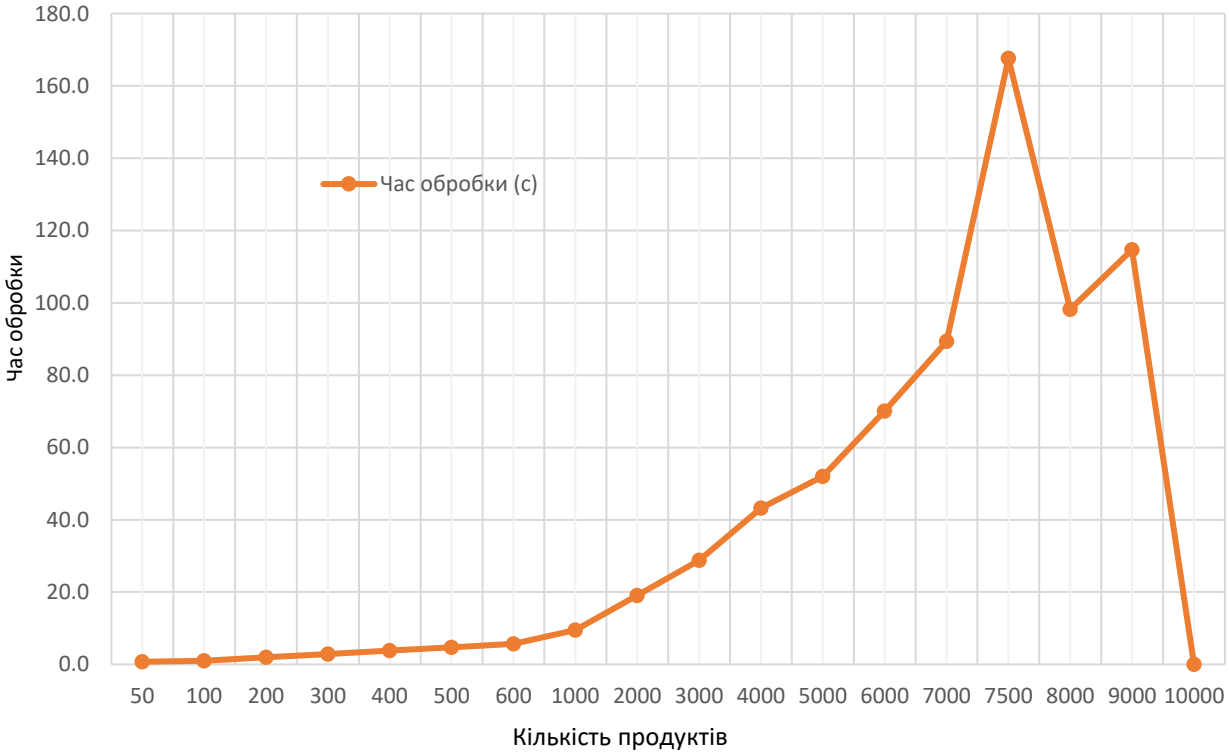

Рисунок 3.6 – Залежність часу обробки від кількості продуктів на Samsung Galaxy S21 FE 5G

Galaxy S21 FE має потужний процесор Qualcomm Snapdragon 888, що забезпечує стабільну роботу навіть при обробці 10000 продуктів. Завантаження CPU не перевищувало 22–23%, свідчачи про те, що обчислювальна потужність не є обмежувальним фактором.

На цьому пристрої також відзначено, що RAM є основним обмежувачем. При споживанні пам’яті на рівні 5898 МБ (приблизно 75% від 8 ГБ), відбувся краш програми. Це підтверджує, що навіть на пристроях із великою кількістю оперативної пам’яті слід залишати запас для системи.

Час виконання Galaxy S21 FE демонструє значно кращий час виконання:

– для 3000 продуктів – 28.8 секунд;

– для 7500 продуктів – 98.2 секунд.

Це дозволяє рекомендувати обробку до 6000 продуктів із допустимим часом очікування для користувача.

Redmi 9A – це доступний смартфон початкового рівня, побудований на базі енергоефективного процесора MediaTek Helio G25. Чип включає 8 ядер ARM Cortex-A53 із частотою до 2,0 ГГц, що забезпечує плавну роботу базових додатків і щоденних задач. Виготовлений за 12-нм техпроцесом, Helio G25 гарантує низьке енергоспоживання. Результати роботи наведені в таблиці 3.3, та на рисунках 3.7, 3.8 та 3.9.

Графічна підсистема представлена GPU PowerVR GE8320 із частотою до 650 МГц, яка підходить для виконання стандартних графічних завдань і запуску невимогливих ігор.

Смартфон оснащений оперативною пам’яттю обсягом 2 або 3 ГБ стандарту LPDDR4X, що забезпечує швидку роботу з базовими програмами та комфортне виконання повсякденних задач. Для зберігання даних доступно 32 ГБ вбудованої пам’яті типу eMMC 5.1, яка підтримує розширення за допомогою карти microSD обсягом до 512 ГБ, що дає можливість зберігати значний обсяг файлів, фото та відео.

Екран смартфона — 6,53-дюймовий IPS-дисплей із роздільною здатністю HD+ (1600 x 720 пікселів), співвідношенням сторін 20:9 і гарною деталізацією.

Завдяки сертифікації TÜV Rheinland дисплей знижує вплив синього світла, що допомагає зменшити втому очей при тривалому користуванні.

Таблиця 3.3 – Результати вимірювання продуктивності системи залежно від кількості оброблюваних продуктів при використанні Redmi 9A

Кількість продуктів	CPU, %	RAM, мб	Час, сек
50	33	157	7
100	35	156	8.3
200	31	123	15.6
300	30	279	22.3
400	30	353	29.4
500	32	408	37.0
600	35	460	44.0
1000	26	722	76.1
2000	31	1371	164.7
3000	33	1643	crash

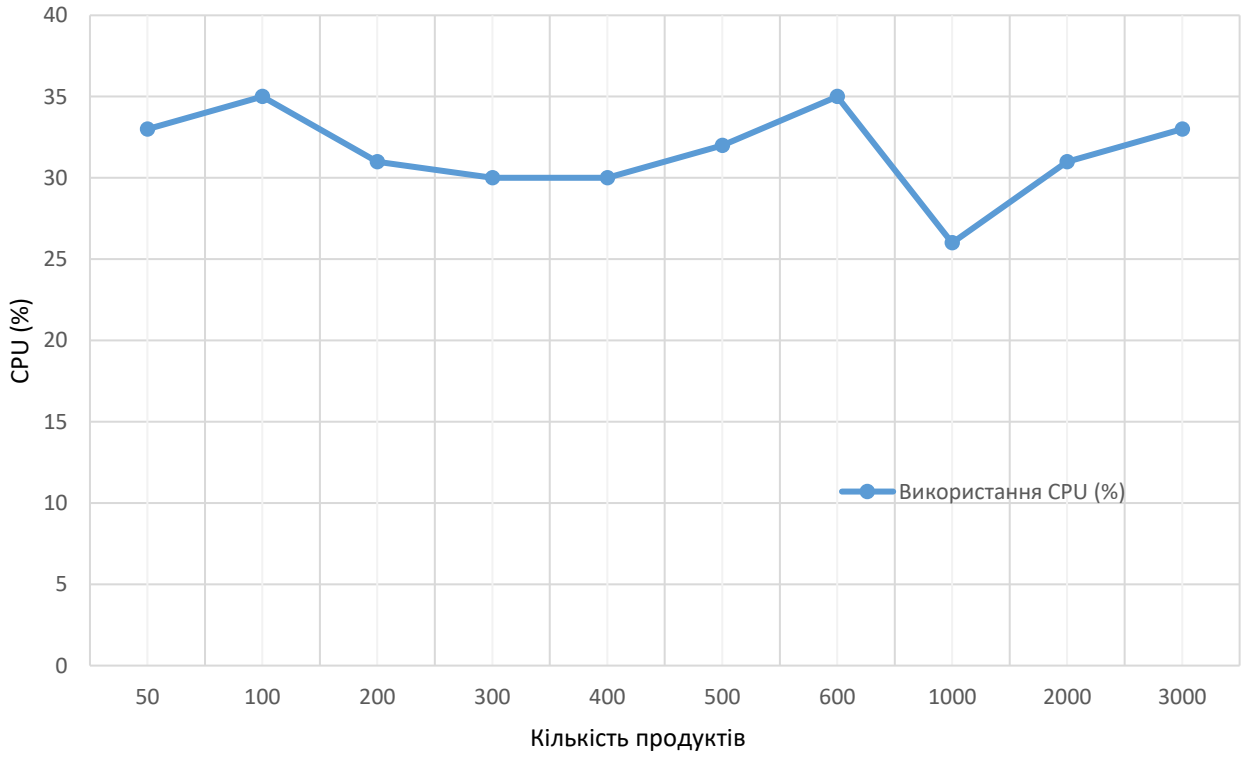

Рисунок 3.7 – Залежність CPU від кількості продуктів на Redmi 9A

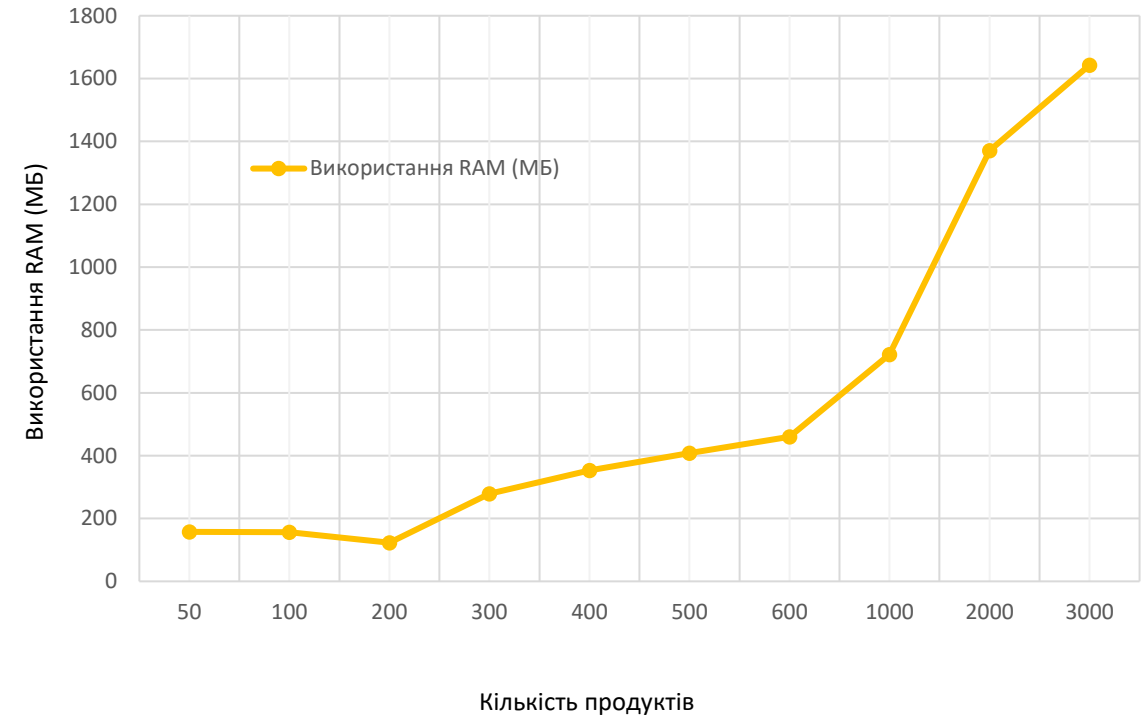

Рисунок 3.8 – Залежність RAM від кількості продуктів на Redmi 9A

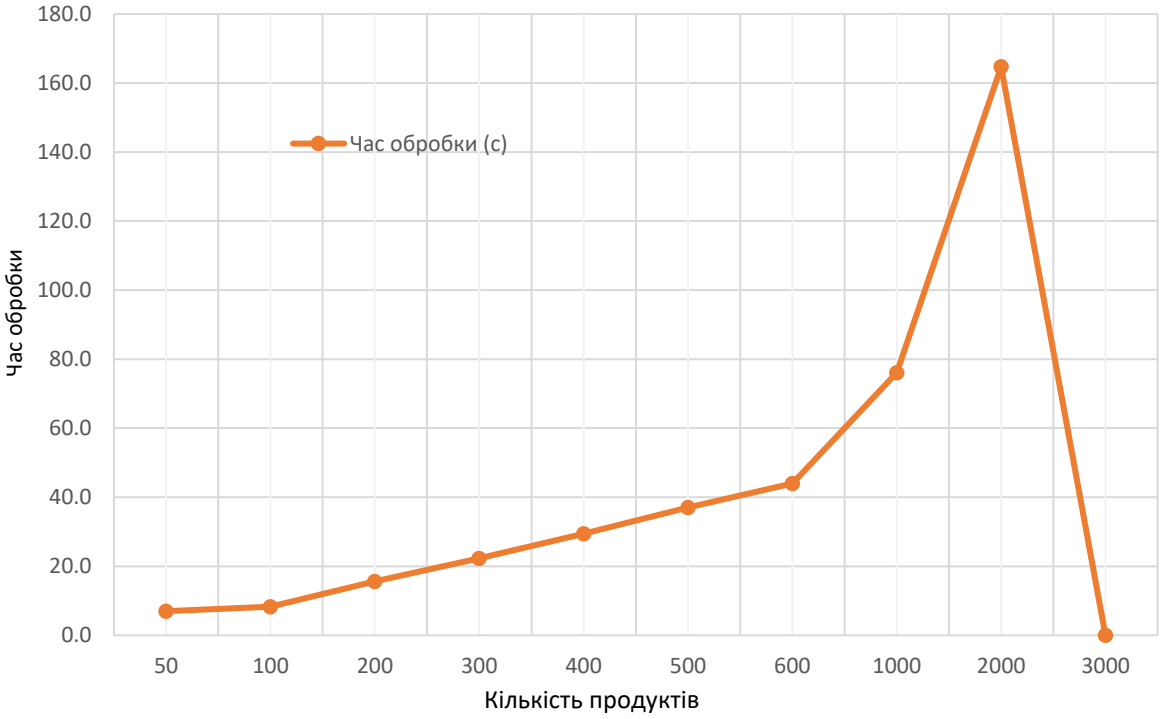

Рисунок 3.9 – Залежність часу обробки від кількості продуктів на Redmi 9A

Процесор MediaTek Helio G25 виявився найслабшим із трьох тестованих пристроїв, що відобразилося у більшому завантаженні CPU, досягаючи 35% при

обробці 7500 продуктів. Проте навіть цей рівень є прийнятним, що вказує на те, що обчислювальна потужність залишається достатньою для цієї задачі.

Redmi 9A оснащений 2–3 ГБ оперативної пам’яті, що стало основною причиною крашу програми вже при обробці 3000 продуктів. Наприклад, при споживанні 1643 МБ (приблизно 70% доступної пам’яті), програма втрачала стабільність. Це вказує на обмеження для пристроїв початкового рівня.

Для цього пристрою час генерації значно зростав:

– для 3000 продуктів – 164.7 секунд.

– краш відбувся на 3000 продуктах через недостатню оперативну пам’ять.

Рекомендовано обмежити обробку до 1000 продуктів, щоб забезпечити комфортний досвід.

3.2 Висновки за розділом

У розділі дипломної роботи було проведено дослідження розробленої розподіленої системи автоматизації обліку фармацевтичної продукції. Основна увага приділялася оцінці ефективності, продуктивності та надійності запропонованого рішення.

Здійснено аналіз споживання ресурсів мобільними пристроями під час виконання основних функцій системи, таких як генерація звітів у форматі PDF із використанням QR-кодів, інтеграція з серверною частиною через Spring Boot та виконання операцій обліку. Результати досліджень підтвердили, що система забезпечує високу швидкість обробки даних навіть на пристроях середнього цінового сегмента, що робить її доступною для широкого кола користувачів.

Дослідження показали, що використання Kotlin Coroutines для асинхронних операцій дозволяє мінімізувати затримки в роботі додатка, забезпечуючи безперебійну взаємодію між клієнтською та серверною частинами системи. QR-

коди продемонстрували високу ефективність для ідентифікації товарів, а звіти у форматі PDF забезпечили зручність у веденні складського обліку та наданні документації.

Виявлено, що найбільше навантаження на систему припадає на процеси генерації звітів та обробки великого обсягу даних. Для підвищення продуктивності системи в подальшому рекомендовано оптимізувати алгоритми роботи з базою даних та застосовувати стиснення файлів PDF для економії пам’яті.

Процесорні ресурси показали стабільність і не є обмежувальним фактором, у той час як оперативна пам'ять виступає ключовим елементом, який визначає межі продуктивності.

Основними рекомендаціями забезпечення стабільної роботи програми є:

– оптимізація використання пам’яті. Зменшити споживання оперативної пам’яті за рахунок компресії даних та потокової обробки;

– обмеження кількості оброблюваних продуктів. Для пристроїв із 2–3 ГБ RAM обмежити до 1000 продуктів, для 6 ГБ RAM – до 3000–5000, а для 8 ГБ RAM – до 6000–7500;

– покращення користувацького досвіду. Впровадити індикатори прогресу та попередження про тривалий час виконання операцій;

– залишення запасу пам’яті. Застосунок не повинен використовувати більше ніж 70–75% оперативної пам’яті для забезпечення стабільності системи.

Таким чином, проведене дослідження підтвердило, що розроблена система є ефективною, надійною та готовою до впровадження у фармацевтичну галузь. Вона дозволяє значно підвищити точність та швидкість складських операцій, що сприяє оптимізації процесів обліку та зменшенню людських помилок.

ВИСНОВКИ

У процесі виконання дипломної роботи було розроблено комп’ютерну систему автоматизації обліку фармацевтичної продукції на базі Android, яка поєднує сучасні технології та методи розробки. Використання мови програмування Kotlin, архітектурного патерна MVVM та інструменту Jetpack Compose забезпечило простоту та ефективність реалізації користувацького інтерфейсу. Для серверної частини системи було обрано Spring Boot, що дозволило досягти високої продуктивності й масштабованості.

У ході дослідження:

– проведено аналіз сучасних WMS-систем і визначено переваги використання мобільних платформ для автоматизації складських процесів;

– розроблено проєкт клієнтської частини з урахуванням вимог до зручності та адаптивності користувацького інтерфейсу;

– впроваджено механізм генерації PDF-звітів із QR-кодами, який забезпечує зручне управління складською документацією;

– проведено тестування розробленої системи на мобільних пристроях із різними технічними характеристиками для визначення мінімальних вимог до обладнання.

Результати роботи підтверджують, що впровадження розробленої системи дозволяє значно підвищити ефективність управління складськими операціями, мінімізувати помилки в обліку фармацевтичної продукції та оптимізувати роботу персоналу. Система також створює передумови для подальшого розвитку і масштабування, включаючи інтеграцію з іншими інформаційними системами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Automation has reached its tipping point for omnichannel warehouses. URL: https://www.mckinsey.com/industries/retail/our-insights/automation-has-reached-its-tipping-point-for-omnichannel-warehouses.

2 32+ Warehouse Automation Statistics You Need to Know in 2024. URL: https://warehousewiz.com/blogs/news/warehouse-automation-statistics.

3 What is a warehouse management system (WMS). URL: https://www.sap.com/products/scm/extended-warehouse-management/what-is-a-wms.html.

4 What is NetSuite Warehouse Management System? URL: https://scalenorth.com/insights/netsuite-wms-intro.

5 Fishbowl. URL: https://www.fishbowlinventory.com/

6 Oddo Inventory. URL: https://www.odoo.com/uk_UA/app/inventory

7 Ramadan M. Master dependency injection for android using dagger: learn dagger 2 with kotlin step by step. Independently Published, 2019.

8 Jake Wharton. Android App Development with Kotlin. 1st ed. – Addison-Wesley Professional, 2019. – 760 p.

9 Guide to app architecture - Android Developers. URL: https://developer.android.com/topic/architecture.

10 Horstmann C. S., Cornell G. Core Java Volume I – Fundamentals. 11th ed. – Upper Saddle River, NJ: Pearson Education, Inc., 2018. – 928 p.

11 Monolithic Architecture pattern. URL: https://microservices.io/patterns/monolithic.html.

12 MVVM – Model View ViewModel. URL: https://medium.com/nuances-of-programming/mvvm-на-android-с-компонентами-архитектуры-библиотека-koin-e2e77b77950e.

13 Jemerov D., Isakova S. Kotlin in action. Manning Publications Co. LLC, 2017. 360 p.

14 Head first design patterns: a brain-friendly guide / K. Sierra et al. O'Reilly Media, Incorporated, 2004. 694 p.

15 Griffiths D., David G. Head first android development. O'Reilly Media, Incorporated, 2021.

16 Gookin D. Android phones and tablets for dummies. Wiley & Sons, Incorporated, John, 2017. 384 p.

17 Nurkiewicz T. Reactive Programming with RxJava. O'Reilly Media, 2016. 372 p.

18 Kotlin coroutines by tutorials: mastering coroutines in kotlin and android. Razeware LLC, 2019.

19 David Sklar, Adam Trachtenberg. PHP Cookbook: Solutions and Examples for PHP Programmers. 3rd ed. – O'Reilly Media, Inc., 2014. – 820 p. ISBN 978-1-491-90335-9.

20 Cantelon M., Harter T., Holowaychuk T. Node.js in Action. 2nd ed. – Shelter Island, NY: Manning Publications Co., 2017. – 432 p.

21 Introduction to Monolithic Architecture and MicroServices Architecture. URL: https://medium.com/koderlabs/introduction-to-monolithic-architecture andmicroservices-architecture-b211a5955c63.

22 ISO/IEC/IEEE 2382:2015 Information technology – Vocabulary. URL: https://www.iso.org/standard/63598.html.

23 SpringBoot Overview. URL: https://spring.io/projects/spring-boot#overview.

24 WebSockets for fun and profit. URL: https://stackoverflow.blog/2019/12/18/websockets-for-fun-and-profit/

25 Demystifying Application Deployment: A Comprehensive Guide. URL: https://www.wgu.edu/blog/demystifying-application-deployment-comprehensive-guide2311.html

26 Android Developers: Android 9 features and APIs. URL: https://developer.android.com/about/versions/pie/android-9.0

27 ZXing Github Repository. URL: https://github.com/zxing/zxing

ДОДАТОК А
ТЕКСТИ ПРОГРАМ

Лістинг А.1 – Код класу OperationController

```
@RestController
@RequestMapping("/origin/operations")
class OperationController(@Autowired private val socket:
IbsSocketMessageService) {

    @GetMapping("/scan")
    fun scan(@RequestParam tenantId: Int, @RequestParam tag:
String): ScanDto {
        if (tag.isBlank()) {
            throw BadRequestException("Tag cannot be empty")
        }
        when {
            tag.startsWith(binTagPrefix, ignoreCase = true) -> {
                val bin = bins.find { it.tenantId == tenantId &&
it.id == tag.substring(binTagPrefix.length) }
                return ScanDto(EntityType.Bin, bin?.toDto())
            }

            tag.startsWith(stationTagPrefix, ignoreCase = true) -> {
                val station =
                    stations.find { it.tenantId == tenantId && it.id
== tag.substring(stationTagPrefix.length) }
                return ScanDto(EntityType.Station, station?.toDto())
            }

            tag.startsWith(palletTagPrefix, ignoreCase = true) -> {
                val pallet = pallets.find { it.tenantId == tenantId
&& it.id == tag.substring(palletTagPrefix.length) }
                return ScanDto(EntityType.Pallet, pallet?.toDto())
```

```
            }
            tag.startsWith(slotTagPrefix, ignoreCase = true) -> {
                val regexMatches = Regex("^([^.]+)\\.([0-5])$")
                    .find(tag.substring(slotTagPrefix.length))
                    ?: throw BadRequestException("Invalid slot
format")

                val palletId = regexMatches.groups[1]?.value
                val slot = regexMatches.groups[2]?.value!!.toInt()
                val pallet = pallets.find { it.tenantId == tenantId
&& it.id == palletId }
                return ScanDto(EntityType.Slot, SlotDto(slot,
pallet?.toDto()))
            }

        else -> {
                val product = Store.products.find { product ->
product.barcodes.any { barcode -> barcode == tag } }
                return ScanDto(EntityType.Product, product)
            }
        }
    }

    @PutMapping("/bins/{binId}")
    fun updateBin(
        @PathVariable binId: String,
        @RequestParam tenantId: Int,
        @RequestBody body: UpdateBin,
        @RequestParam(required = false) withProducts: Boolean?
    ): BinDto {
        val bin = bins.filter { it.tenantId == tenantId }.find {
it.id == binId }
        if (bin == null) {
            throw BadRequestException("Bin not found")
        }
        if (!body.palletId.isNullOrEmpty() && body.slot != null) {
```

```
            if (body.slot < 0 || body.slot > 5) {
                throw BadRequestException("Invalid slot value")
            }
            if (pallets.none { it.tenantId == tenantId && it.id ==
body.palletId }) {
                throw BadRequestException("Pallet not found")
            }
            if (bins.count { it.tenantId == tenantId && it.palletId
== body.palletId && it.slot == body.slot } == 2) {
                throw BadRequestException("Slot is full")
            }
        }
        if(body.palletId.isNullOrEmpty()) {
            bin.apply {
                palletId = null
                slot = null
            }
        } else {
            bin.apply {
                palletId = body.palletId
                slot = body.slot
           }
            // clear ready bin if it was inserted to a pallet
            val ibsContext = Store.inboundStations
                .filter { it.tenantId == tenantId }
                .find { it.readyBin.any { readyBin ->
readyBin.bin.id == bin.id } }
            if(ibsContext != null) {
                ibsContext.readyBin.removeIf { readyBin ->
readyBin.bin.id == bin.id }
                socket.sendContextToIbsTopic(ibsContext)
            }
        }
        return getBinDto(tenantId, binId, withProducts)!!
    }
```

```
    @GetMapping("/bins/{binId}/products/{productId}")
    fun getBinProduct(
        @PathVariable binId: String,
        @PathVariable productId: Int,
        @RequestParam tenantId: Int
    ): BinProductDto {
        val bin = bins.filter { it.tenantId == tenantId }.find {
it.id == binId }
        if (bin == null) {
            throw BadRequestException("Bin not found")
        }
        val product = Store.products.find { it.id == productId }
        if (product == null) {
            throw BadRequestException("Product not found")
        }
        return bin.products[productId]?.let {
            BinProductDto(productId, product.name, it)
        } ?: throw BadRequestException("Product not found in bin")
    }

    @PutMapping("/bins/{binId}/products/{productId}")
    fun createOrUpdateBinProduct(
        @PathVariable binId: String,
        @PathVariable productId: Int,
        @RequestParam tenantId: Int,
        @RequestBody body: UpdateBinProduct
    ): BinProductDto {
        val product = Store.products.find { it.id == productId }
        if (product == null) {
            throw BadRequestException("Product not found")
        }
        val bin = bins.filter { it.tenantId == tenantId }.find {
it.id == binId }
        if (bin == null) {
            throw BadRequestException("Bin not found")
        }
```

```
        if (body.quantity == 0) {
            if (!bin.products.containsKey(productId)) {
                throw BadRequestException("The product is not in the
bin")
            }
            val quantity = bin.products.remove(productId)!!
            return BinProductDto(productId, product.name, quantity)
        } else {
            bin.products[productId] = body.quantity
            return BinProductDto(productId, product.name,
body.quantity)
        }
    }

    @GetMapping("/pallets")
    fun getPallets(
        @RequestParam tenantId: Int,
        @RequestParam pageSize: Int,
        @RequestParam page: Int,
        @RequestParam(required = false) floating: Boolean?,
        @RequestParam(required = false) flagged: Boolean?
    ): PageResponse<PalletDto> {
        var pallets = Store.pallets.filter { it.tenantId == tenantId
}.toMutableList()
        if (floating != null) {
            pallets = if (floating) {
                pallets.filter { it.stationId == null
}.toMutableList()
            } else {
                pallets.filter { it.stationId != null
}.toMutableList()
            }
        }
        if (flagged != null) {
            pallets.filter { bin ->
                val lastIssue =
```

```
                    Store.issues.findLast { it.tenantId == tenantId
&& it.entityType == EntityType.Pallet && it.entityId == bin.id }
                if (flagged) {
                    lastIssue != null && lastIssue.operation ==
IssueOperation.Report
                } else {
                    lastIssue == null || lastIssue.operation ==
IssueOperation.Clear
                }
            }
        }
        val palletsPage = pallets.drop((page - 1) *
pageSize).take(pageSize)
            .map { pallet -> pallet.toDto() }
            .toMutableList()
         return PageResponse(pallets.count(), palletsPage)
    }

    @GetMapping("/pallets/{palletId}")
    fun getPallet(
        @PathVariable palletId: String,
        @RequestParam tenantId: Int,
        @RequestParam(required = false) withBins: Boolean?,
        @RequestParam(required = false) withProducts: Boolean?
    ): PalletDto? {
        return getPalletDto(tenantId, palletId, withBins,
withProducts)
    }

    private fun getPalletDto(
        tenantId: Int,
        palletId: String,
        withBins: Boolean?,
        withProducts: Boolean?
    ): PalletDto? {
        val pallet = pallets.filter { it.tenantId == tenantId }.find
```

```
{ it.id == palletId }
        val palletDto = pallet?.toDto() ?: return null
        if (withBins != null && withBins) {
            palletDto.bins = bins
                .filter { it.tenantId == tenantId && it.palletId ==
palletId }
                .map { bin ->
                    val binDto = bin.toDto()
                    if (withProducts != null && withProducts) {
                        binDto.products = bin.products.map {
(productId, quantity) ->
                            val product = Store.products.find {
it.id == productId }
                            BinProductDto(productId, product?.name
?: "Unknown", quantity)
                        }
                    }
                    binDto
                }
        }
        return palletDto
    }

    @PutMapping("/pallets/{palletId}/bins")
    fun updatePalletBins(
        @PathVariable palletId: String,
        @RequestParam tenantId: Int,
        @RequestBody body: List<UpdatePalletBin>,
        @RequestParam(required = false) withBins: Boolean?,
        @RequestParam(required = false) withProducts: Boolean?
    ): PalletDto {
        val pallet = pallets.filter { it.tenantId == tenantId }.find
{ it.id == palletId }
        if (pallet == null) {
            throw BadRequestException("Pallet not found")
        }
```

```
        // validate all bins found
        body.forEach { update ->
            if (!bins.any { it.tenantId == tenantId && it.id ==
update.binId }) {
                throw BadRequestException("Bin ${update.binId} not
found")
            }
        }

        // check there is not more than 2 bins with the same slot
        body.groupingBy { it.slot }.eachCount().forEach { (slot,
count) ->
            if (count > 2) {
                throw BadRequestException("Slot $slot has more than
two occurrences")
            }
        }

        // float bins that are not in the request
        bins.filter { it.tenantId == tenantId && it.palletId ==
palletId }
            .filter { x -> body.none { it.binId == x.id } }
            .forEach { it.palletId = null }

        // assign bins that are in the request
        body.forEach { update ->
            val bin = bins.find { it.tenantId == tenantId && it.id
== update.binId }!!
            bin.palletId = palletId
            bin.slot = update.slot
        }

        return getPalletDto(tenantId, palletId, withBins,
withProducts)!!
    }
```

```
    @PutMapping("/pallets/{palletId}/slots/{slotId}/bins")
    fun updatePalletSlotBins(
        @PathVariable palletId: String,
        @PathVariable slotId: Int,
        @RequestParam tenantId: Int,
        @RequestBody body: List<String>
    ) {
        val pallet = pallets.filter { it.tenantId == tenantId }.find
{ it.id == palletId }
        if (pallet == null) {
            throw BadRequestException("Pallet not found")
        }
        if (slotId < 0 || slotId > 5) {
            throw BadRequestException("Invalid slot value")
        }
        // check there is one or two bins to update
        body.size.let {
            if (it < 1 || it > 2) {
                throw BadRequestException("Invalid number of bins to
update. Need 1 or 2")
            }
        }
        // float bins that are not in the request
        bins.filter { it.tenantId == tenantId && it.palletId ==
palletId && it.slot == slotId }
            .forEach { it.palletId = null }

        // assign bins that are in the request
        bins.filter { it.tenantId == tenantId &&
body.contains(it.id) }
            .forEach {
                it.palletId = palletId
                it.slot = slotId
            }
    }
```

Лістинг А.2 – Код класу HardwareController

```
@RestController
@RequestMapping("/origin/hardware")
class HardwareController(@Autowired private val socket:
IbsSocketMessageService) {

    @PatchMapping("/inbound/ibs/{ibsId}")
    fun updateIbsContext(
        @RequestParam tenantId: Int,
        @PathVariable ibsId: String,
        @RequestBody update: InboundContextUpdate): IbsContext {
        val ibsContext = Store.inboundStations.find { it.tenantId ==
tenantId && it.id == ibsId }
        if(ibsContext == null) {
            throw BadRequestException("Inbound station not found")
        }
        if(update.productId != null) {
            val product = Store.products.find { product ->
product.id == update.productId }
            if(product == null) {
                throw BadRequestException("Product not found")
            }
            ibsContext.product = product
        }
        return ibsContext
    }
    @GetMapping("/inbound/ibs/{ibsId}")
    fun getIbsContext(
        @RequestParam tenantId: Int,
        @PathVariable ibsId: String): IbsContext {
        val ibsContext = Store.inboundStations.find { it.tenantId ==
tenantId && it.id == ibsId }
        if(ibsContext == null) {
            throw BadRequestException("Inbound station not found")
        }
```

```
        return ibsContext
    }

PutMapping("/inbound/ibs/{ibsId}/bins/{binId}/products/{productId}")
    fun createOrUpdateIbsBinProduct(
        @PathVariable ibsId: String,
        @PathVariable binId: String,
        @PathVariable productId: Int,
        @RequestParam tenantId: Int,
        @Valid @RequestBody body: UpdateIbsBinProduct
    ): IbsContext {
        val product = Store.products.find { it.id == productId }
        if (product == null) {
            throw BadRequestException("Product not found")
        }
        val ibsContext = Store.inboundStations.find { it.tenantId ==
tenantId && it.id == ibsId }
        if (ibsContext == null) {
            throw BadRequestException("Inbound station not found")
        }

        val bin = ibsContext.slots.values.map { v -> v.bin }.find {
it.id == binId }
        if (bin == null) {
            throw BadRequestException("Bin not found")
        }
        if (bin.products.containsKey(productId)) {
            bin.products[productId] = bin.products[productId]!! +
body.quantity
        } else {
            bin.products[productId] = body.quantity
        }
        socket.sendContextToIbsTopic(ibsContext)
        return ibsContext
    }
}
```

Лістинг А.3 – Код класу PdfConverter

```
class PDFConverter(val context: Context) {

    private fun createBitmapFromPlantLabelView(
        view: View,
        data: ProductPrintLabelData,
        codeImage: Bitmap
    ): Bitmap {
        val labelCodeImage: ImageView =
view.findViewById(R.id.labelCode)
        val productIdTv: TextView =
view.findViewById(R.id.productId)
        val productNameTv: TextView =
view.findViewById(R.id.productName)
        val quantityTv: TextView = view.findViewById(R.id.quantity)
        labelCodeImage.setImageBitmap(codeImage)
        data.apply {
            productIdTv.text = id
            productNameTv.text = name
            quantityTv.text = quantity
        }
        return createBitmap(
            view = view,
            labelHeight = PLANT_LABEL_HEIGHT,
            labelWidth = PLANT_LABEL_WIDTH
        ).rotateBitmap()
    }

    private fun createBitmap(
        view: View,
        labelWidth: Int,
        labelHeight: Int,
    ): Bitmap {
        view.measure(
            View.MeasureSpec.makeMeasureSpec(
```

```
                labelWidth, View.MeasureSpec.EXACTLY
            ),
            View.MeasureSpec.makeMeasureSpec(
                labelHeight, View.MeasureSpec.EXACTLY
            )
        )
        view.layout(0, 0, labelWidth, labelHeight)

        val bitmap = Bitmap.createBitmap(
            labelWidth,
            labelHeight,
            Bitmap.Config.ARGB_8888
        )
        val canvas = Canvas(bitmap)
        view.draw(canvas)

        return bitmap
    }

    fun finishWriting(pdfDocument: PdfDocument) {
        val filePath = File(context.getExternalFilesDir(null),
"batchLabelsPdf.pdf")
        pdfDocument.writeTo(FileOutputStream(filePath))
        pdfDocument.close()
        renderPdf(context, filePath)
    }

    fun addProductLabelPage(
        data: Array<ProductPrintLabel?>,
        codeImages: Array<Bitmap?>,
        pdfDocument: PdfDocument
    ) {
        val inflater = LayoutInflater.from(context)
        val view = inflater.inflate(R.layout.product_print_label,
null)
```

```
        val demoBitmap = createBitmapFromPlantLabelView(
            view,
            data[0]!!.labelData,
            codeImages[0]!!
        )

        val pageInfo = PdfDocument.PageInfo.Builder(
            demoBitmap.width * 4,
//              demoBitmap.height + EMPTY_WHITE_ZONE_HEIGHT,
            demoBitmap.height + 0,
            1
        ).create()
        val page = pdfDocument.startPage(pageInfo)
        var left = 0F

        for (i in data.indices) {
            if (data[i] != null && codeImages[i] != null) {
                page.canvas.drawBitmap(
                    createBitmapFromPlantLabelView(
                        view,
                        data[i]!!.labelData,
                        codeImages[i]!!
                    ),
                    left,
//                    EMPTY_WHITE_ZONE_HEIGHT.toFloat(),
                    0F,
                    null
                )
                left += demoBitmap.width
            }

        }
        pdfDocument.finishPage(page)
    }

    private fun renderPdf(context: Context, filePath: File) {
```

```
    val uri = FileProvider.getUriForFile(
        context,
        context.applicationContext.packageName + ".provider",
        filePath
    )
    val intent = Intent(Intent.ACTION_VIEW)
    intent.flags = Intent.FLAG_ACTIVITY_CLEAR_TOP
    intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK
    intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
    intent.setDataAndType(uri, "application/pdf")

    try {
        context.startActivity(intent)
    } catch (e: ActivityNotFoundException) {

    }
}

private fun Bitmap.rotateBitmap(): Bitmap {
    return Bitmap.createBitmap(
        this,
        0,
        0,
        this.width,
        this.height,
        Matrix().apply {
            postRotate(
                LABEL_ROTATION_DEGREES,
                this@rotateBitmap.height / 2f,
                this@rotateBitmap.height / 2f
            )
        },
        true
    )
}
```

```
    companion object {
        const val PLANT_LABEL_WIDTH = 600
        const val PLANT_LABEL_HEIGHT = 254
        const val BATCH_LABEL_WIDTH = 1190
        const val BATCH_LABEL_HEIGHT = 1684
        const val EMPTY_WHITE_ZONE_HEIGHT = 715
        const val LABEL_ROTATION_DEGREES = 270f
    }
}
```