Medicare

Medicine Identification App

By

GREESHMA GIRISH C (2241130)

SANDEEP MATHEW (2241155)

Under the supervision of

Dr. Sreeja C.S

Project Report submitted in partial fulfilment of the requirements of

VI semester BCA, CHRIST (Deemed to Be University)

March - 2025

CERTIFICATE

This is to certify that the report titled ***Medicare*** *is a bona fide record of work done by* ***Greeshma Girish C (2241130), Sandeep Mathew (2241155)*** *of Christ University, Bengaluru, in partial fulfillment of the requirements of 6th Semester BCA during the year 2025.*

Head of the Department **Project Guide**

Valued-by:

1.

2.

Name	:	Greeshma Girish C
Register Number	:	2241130
Examination Centre	:	Christ University
Date of Exam	:	

ACKNOWLEDGEMENTS

First of all, we thank God almighty for his immense grace and blessings showered on us at everystage of this work. We are grateful to our respectable Head, Department of Computer Science, CHRIST (Deemed to be University), **Dr. Ashok Immanuel V**, for providing the opportunity to take up this project as part of my curriculum.

We also pay our gratitude to the Coordinator, Department of Computer Science, CHRIST (Deemedto be University) **Dr. Sagaya Aurelia P** for their support throughout.

We are grateful to our guide, Assistant Professor, Department of Computer Science, CHRIST (Deemed to be University), **Dr. Sreeja C.S,** whose insightful leadership and knowledge benefitedus to complete this project successfully. Thank you so much for your continuous support and presence whenever needed.

We would also like to thank our Alumni evaluator**,** whose knowledge and guidance benefited us in making the changes as per the industry requirement. Thank you so much for your support and presence whenever needed.

We express our sincere thanks to all faculty members and staff of the Department of Computer Science, CHRIST (Deemed to be University), for their valuable suggestions during the course ofthis project. Their critical suggestions helped us to improve the project work.

Last but not the least, we would like to thank everyone who is involved in the project directly or indirectly.

ABSTRACT

Medicare aims to address the critical challenges of counterfeit medicines and healthcare accessibility by providing an advanced medicine verification and information platform. The initiative focuses on empowering users with reliable tools to authenticate medicines, ensuring patient safety, and enhancing trust in pharmaceutical products. By integrating machine learning and real-time data retrieval, Medicare enables users to scan, verify, and access detailed medicine information, reducing the risks associated with counterfeit drugs.

Beyond verification, Medicare fosters a user-centric approach by improving accessibility to essential healthcare information. The platform is designed to assist consumers, healthcare professionals, and regulatory bodies in making informed decisions, ultimately contributing to a safer and more transparent medical ecosystem.

Aligned with the United Nations Sustainable Development Goals, Medicare supports SDG Goal 3 (Good Health and Well-being) by enhancing medication safety and SDG Goal 9 (Industry, Innovation, and Infrastructure) by leveraging innovative technology for secure healthcare solutions. The initiative envisions a future where patients can confidently access genuine medicines, ensuring better health outcomes for all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 BACKGROUND OF THE PROJECT

The Medicare app is a healthcare solution designed to address the growing issue of counterfeit medicines in India. Counterfeit drugs pose serious health risks, leading to ineffective treatments and adverse health effects. To combat this, the Medicare app offers a medicine verification system that enables users to scan medicines using their mobile device cameras and access crucial details instantly. This app serves as a comprehensive platform for verifying medicine authenticity, checking manufacturer details, and reviewing important information such as composition, usage, and potential side effects. By scanning medicine packages, users can ensure that they are consuming genuine medicines, thereby enhancing patient safety and healthcare trust.

1.2 OBJECTIVE

The primary objective of the Medicare app is to provide an intelligent and user-friendly solution for medicine verification and management, ensuring public safety and reducing medical errors. By enabling users to scan medicines and instantly access detailed information, the app helps prevent counterfeit drug consumption and enhances medication adherence. It streamlines workflows for both individuals and healthcare providers by offering features like prescription tracking, medication reminders, and stock alerts.

1.3 PURPOSE, SCOPE AND APPLICABILITY

Purpose

- A medicine verification system that enables users to scan medicines and access details like composition, manufacturer information, and side effects.
- To help users make informed healthcare decisions by providing accurate and reliable drug-related information.
- To enhance medication adherence by offering prescription tracking, reminders, and stock.

Scope

- The app can be used by patients, healthcare professionals, elderly individuals, and caregivers to ensure safe and informed medication use.
- It is designed as a mobile application accessible on smartphones, making it convenient for users to verify medicines anytime, anywhere.
- The app helps in reducing the prevalence of counterfeit drugs by promoting awareness and trust in authentic medicines.

Applicability

- The Medicare app is beneficial for anyone who consumes medication, especially individuals managing chronic conditions, elderly patients, and caregivers.
- It helps users verify the authenticity of medicines, understand their effects, and maintain proper intake schedules

1.4 MODULES

- **Medicine Scan & Counterfeit Detection:** This module enables users to scan medicine packages to verify authenticity and detect counterfeit drugs, ensuring safe consumption.
- **Prescription Tracking:** Users can digitally store and track their prescriptions, helping them manage ongoing treatments and medication history efficiently.
- **Stock Alert:** This feature notifies users when their medicine stock is running low, preventing missed doses and ensuring timely refills.
- **Medicine Reminder:** Users can set up reminders for their medication schedules, ensuring adherence to prescribed treatments.
- **Store Health Vitals:** This module allows users to log and track essential health parameters such as blood pressure, sugar levels, and other vital signs for better health management.
- **Edit Profile:** Users can manage and update their personal information, medical history, and preferences for a personalized experience.

1.5 MAJOR OUTCOME

The Medicare app aims to enhance healthcare accessibility and safety by providing a reliable solution for verifying medicines, tracking prescriptions, and managing medication schedules. With real-time scanning capabilities, users can quickly identify counterfeit medicines, reducing the risk of consuming harmful drugs. The app empowers users to take control of their health by offering features such as stock alerts, reminders, and vital health tracking, ensuring better adherence to prescribed treatments and overall well-being.

1.6 HIGHLIGHT OF THE PROJECT

- Users can scan medicines to verify authenticity and detect counterfeit drugs.
- User-friendly interface for seamless navigation and accessibility.
- Prescription tracking to manage ongoing treatments efficiently.
- Stock alerts to notify users when medication supplies are running low.
- Medicine reminders to ensure timely intake of prescribed drugs.
- Storage of health vitals for continuous health monitoring.

1.7 TOOLS USED

- React Native
- Node JS
- Express JS
- MongoDB
- Roboflow (YOLO)

2. SYSTEM ANALYSIS AND REQUIREMENTS

This chapter describes the system requirements and its analysis. It includes hardware and software requirements as well as the functional and non-functional requirements.

2.1 EXISTING SYSTEM

Several apps currently offer medication identification and management, each with unique features. The MedSnap App utilizes a smartphone camera to identify pills, providing details about their name, dosage, and potential drug interactions. However, it is primarily designed for healthcare providers and requires users to place pills on a special MedSnap tray for accurate identification. The MedHelper App focuses on medication reminders and prescription tracking, helping users manage their schedules effectively. Meanwhile, the Pill Identifier App helps users identify medicines based on their physical attributes, such as shape, color, and imprint, making it easier for users to recognize their medications.

2.2 LIMITATIONS OF EXISTING SYSTEM

The MedSnap App requires a specialized tray for pill identification, which limits accessibility for general users. The MedHelper App, although helpful in tracking prescriptions, lacks a scanning feature for medicine verification and has a limited database, making it less effective for identifying lesser-known drugs. The Pill Identifier App requires manual input and does not support physical packaging scans, preventing it from detecting counterfeit medicines. These shortcomings highlight the need for a more comprehensive solution like the Medicare app, which integrates instant medicine scanning, detailed verification, counterfeit detection, and medication management into a single, user-friendly platform.

2.3 PROPOSED SYSTEM

The Medicare app is a comprehensive healthcare solution designed to streamline medicine identification, counterfeit detection, prescription tracking, and overall medication management. It provides users with an efficient way to verify medicines, receive reminders, track stock levels, and store health vitals in a single, user-friendly platform. By integrating advanced scanning technology, the app allows users to instantly identify medicines and detect counterfeit drugs, ensuring safety and reliability. Additionally, the app enables users to maintain a digital record of their prescriptions, receive timely alerts for medication intake, and track their health vitals for better healthcare management.

2.4 BENEFITS OF THE PROPOSED SYSTEM

Many existing healthcare apps focus only on specific functionalities, such as medicine reminders or prescription tracking, but lack an integrated system for medicine verification and counterfeit detection. The Medicare app bridges this gap by offering a holistic solution that ensures users not only manage their medications effectively but also verify their authenticity. This helps in preventing the risks associated with counterfeit drugs while promoting adherence to prescribed medications. The app is designed to be user-friendly and accessible to all, from general consumers to elderly users who may need assistance in tracking their medications.

2.5 FEATURES OF THE PROPOSED SYSTEM

- **Comprehensive Medicine Verification**: The app allows users to scan medicines using their smartphone camera, instantly verifying their authenticity and detecting counterfeit drugs to ensure safe medication use.
- **Prescription Tracking:** Users can digitally store and track their prescriptions, reducing the risk of missing doses or misplacing important medication details.
- **User-Friendly Interface**: Designed for ease of use, the app provides a seamless experience for all users, including elderly individuals and those with minimal technical knowledge.
- **Secure Data Management**: User health records, prescriptions, and medical history are securely stored, ensuring privacy and confidentiality.

- **Security & Privacy:** All medical records and personal data are securely stored, ensuring user confidentiality.

2.6 SYSTEM REQUIREMENTS SPECIFICATION

2.6.1 Requirements/User Stories

The high-level client requirements in terms of user stories are given as below:

- The system should allow users to scan and identify medicines using their camera.
- The app should provide detailed medicine information, including name, dosage, and potential side effects.
- The system should verify the authenticity of scanned medicines and detect fake drugs.
- Users should receive medicine intake reminders to ensure timely consumption.
- Users should be able to store and access their medication history for future reference.
- The system should maintain a secure and reliable database to ensure accurate medicine verification.

2.6.2 User Characters

- **Users:** The primary actor who scans medicines to verify their authenticity, retrieve detailed information, and receive intake reminders.
- **Admin/App:** Processes the scanned medicine data, identifies the drug, checks for counterfeits, provides interaction warnings, and sends reminders to users.

2.6.3 Use Case Diagram

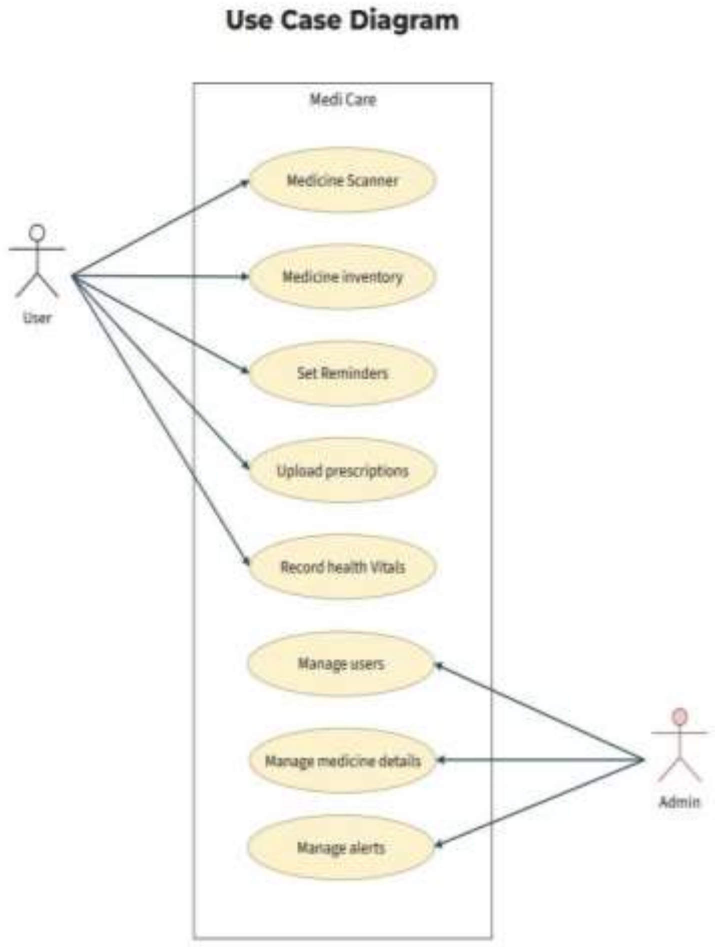


Fig. 2.1 Use case diagram

The above use case diagram depicts the system from the user's point of view. The user interacts with the app to access various functionalities, and the system responds accordingly. The app allows users to scan medicines for identification, detect counterfeit drugs, access detailed medicine information, and set reminder alerts for timely intake. Users can verify the authenticity of their medication, receive accurate dosage details, and ensure they take the right medicine at the right time, improving overall safety and health management.

2.6.4 Software and Hardware Requirements

Software Requirements

- React Native
- Node JS
- Express JS
- Roboflow
- MongoDB

Hardware Requirements

- Processor: Quad-core (1.8 GHz or higher)
- Operating System: Android 7.0 or higher
- RAM: 3 GB
- Storage: 32 GB (at least 500 MB free for app installation)
- Camera: 8 MP or higher (for scanning medicine details)

2.6.5 Constraints

Medicine flows, including chemical compositions, spectroscopy data, and other pharmaceutical details, are regulated by pharmacists due to their direct connection with clinicians. Non-medicinal items such as X-ray, radiology, and laboratory articles are not managed by the app, as they follow their own standards and do not impact patient safety. The application is only compatible with Android-based systems

2.6.6 Functional Requirements

Requirement ID	Requirement Name	Description
C_FR1	Access mode	Users (e.g., General Users, Physicians, Nurses, and Pharmacy Personnel) shall access the system with dedicated privilege sets, ensuring personalized functionality and interface.
C_FR2	Dashboard display	The system shall display a personalized dashboard for users, including medication schedules, stock alerts, prescription tracking, and patient-specific updates.
C_FR3	Medicine Scanning	The system shall enable users to scan medicine packaging or pills using the device camera for identification and detailed information retrieval.
C_FR4	Information Display	Users shall access detailed medicine information, including name, composition, uses, dosage, side effects, manufacturer, and other relevant data.
C_FR5	Prescription Management	The system shall allow users to upload, track, and manage prescriptions for streamlined medication management.
C_FR6	Medicine Reminder Alerts	Users shall set up and receive reminders for medication schedules to ensure timely consumption.
C_FR7	Stock and Refill Alerts	The system shall track medicine stock levels and notify users to restock when supplies are low.

Table 2.1 Functional Requirements

2.6.7 Non-Functional Requirements

Requirement ID	**Requirement Name**	**Description**
C_NF_R1	Performance	As it's a Mobile application, the network, hardware and other related infrastructure plays a vital role in determining the application performance. • **Data Compression**: Efficient formats and caching reduce network burden. • **Navigation**: Minimal screens and gestures streamline user interaction. • **Graphics**: Lightweight visuals and optimized color schemes enhance performance.
C_NF_R2	Safety/Security	Performing frequent backup can reduces the data loss due to sudden server or the system crash. Being a healthcare workflow system, its primary character should be security, thus providing secure environment for the app flow process. Users can confidently verify the medicines they consume, reducing health risks associated with fake or incorrect drugs.
C_NF_R3	Quality Requirements ● Usability	The Medicare app is designed for users of all backgrounds, ensuring an intuitive and easy-to-use interface. Clear navigation, simple instructions, and accessible features make it user-friendly without requiring technical expertise. Additionally, in-app guides and tooltips enhance user experience and understanding.

	● Reliability	The app ensures accurate medicine identification and counterfeit detection without delays or incomplete processing. Each scan and search is performed thoroughly to provide reliable results, minimizing errors in medicine verification.
	● Availability	The Medicare app is designed to function seamlessly, handling multiple user requests efficiently. The system ensures quick responses and uninterrupted access, allowing users to check medicine authenticity and receive reminders anytime.

Table 2.2 Non-Functional Requirements

2.7 BLOCK DIAGRAM

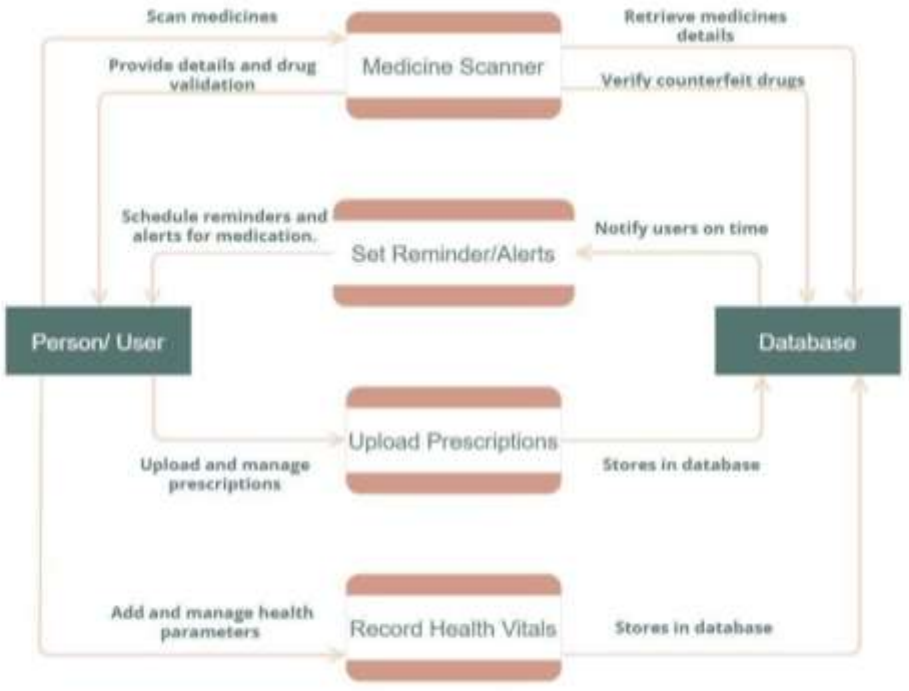


Fig 2.2 Block Diagram

The above diagram illustrates the overall system design and its key functions along with associated components. The Medicare app enables users to scan and identify medicines, verifying their authenticity and retrieving essential drug information. When a user scans a medicine, the app processes the request, cross-referencing it with a comprehensive database to detect counterfeit drugs, provide dosage details, and highlight potential interactions. The system ensures accurate and real-time responses, helping users make informed decisions about their medication. This depiction offers a high-level overview of the application, allowing the team to analyse and refine system requirements effectively.

3. SYSTEM DESIGN

This Document shows the list of Modules that are there on the Medicare app along with the System Architecture. The Data Flow Diagram and the ER Diagram are also mentioned so that the application of the site can be easily understood. The Sample UI Screen and the tables in the database have been listed out.

3.1 SYSTEM ARCHITECTURE

Medicare is a mobile application designed to help users manage their medications effectively. This system was developed to address the limitations of existing solutions by integrating medicine scanning, counterfeit detection, prescription tracking, and health monitoring into a single platform. Users can scan medicines to verify authenticity, track prescriptions, receive medication reminders, and monitor their health vitals.

All user data, including prescription details and health records, is securely stored, retrieved, and updated through a database. Since the system follows a client-server architecture, the application logic, data storage, and user interface are developed as independent modules. This three-tier architecture ensures efficient data processing, secure transactions, and a seamless user experience across multiple devices.

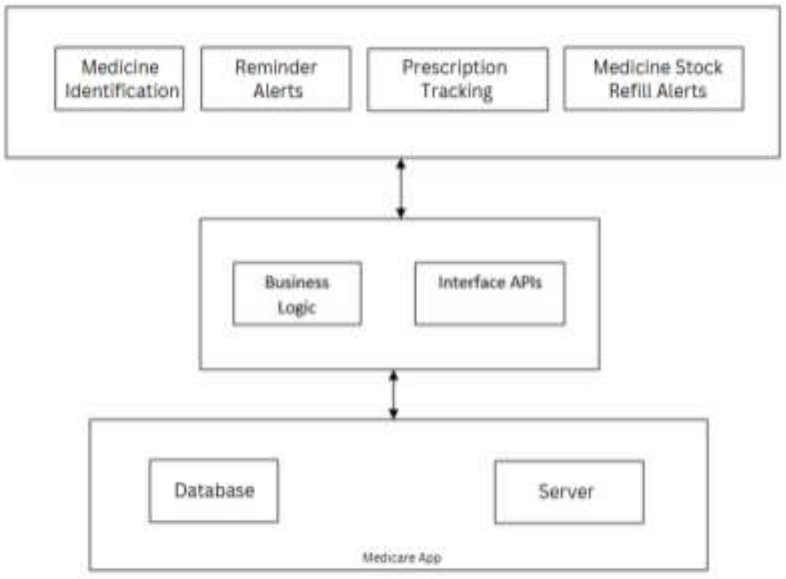


Fig 3.1 Medicare Management System

The figure depicts a very high-level architecture of the system. The first level holds four different modules that provides set of functionalities to its user at the front. The middle layer which is the business layer makes an interface connection between the top and bottom layer in serving and processing the user request. The bottom most layer holds the backend database, webserver and drug repository that provides the required information to all the above layers.

3.2 MODULE DESIGN

This system basically composed of four main high-level modules as mentioned below. Each of these modules in turn broke down into multiple sub modules.

a) Scanning and Identification Module:

The Scanning and Identification module leverages advanced image recognition algorithms to analyze medicine packaging, pills, or barcodes using the device camera. It identifies the medicine and retrieves comprehensive information, including name, composition, uses, dosage, side effects, and manufacturer details, ensuring users receive accurate and quick access to essential medication data.

b) Prescription Management Module:

The Prescription Management module enables users to upload digital copies of prescriptions through images or PDFs. Users can view and organize their prescriptions, track prescribed medication schedules, and maintain a history of medical records. This ensures better medication adherence and simplifies the management of prescriptions for ongoing treatments or consultations.

c) Reminder Module:

The Reminder and Alert module help users stay on top of their medication regimen by enabling custom schedules with timely notifications. It provides alerts for upcoming doses, tracks medicine stock levels, and sends refill reminders when supplies are low, ensuring consistent and efficient medication management for better health outcomes. These main modules are further composed of following sub modules.

d) User Profile Module:

The User Profile module stores personalized user data, including medicine history, health preferences, and prescription records. It enables tailored recommendations, efficient tracking of past medications, and easy retrieval of health information, offering a customized experience to improve medication adherence and support individual health management goals.

e) Admin Module:

The Admin Module handles backend operations, enabling administrators to manage drug databases, including adding, updating, or removing medicine information. It oversees app functionalities, user data management, and system performance, ensuring accurate information delivery, seamless user experience, and compliance with regulatory standards for efficient app operation.

These main modules are further composed of following sub modules

i Login Module

This module provides all access definitions to its users at various level.

ii Medicine Scanning

The module uses the device camera and image recognition to identify medicines, with a manual search option for added reliability.

iii Stock Management Module

The inventory feature tracks medicine stock, sends low stock alerts, and provides refill suggestions via nearby pharmacies or online options.

3.3 DATA FLOW DIAGRAM

3.3.1 DFD Level 0

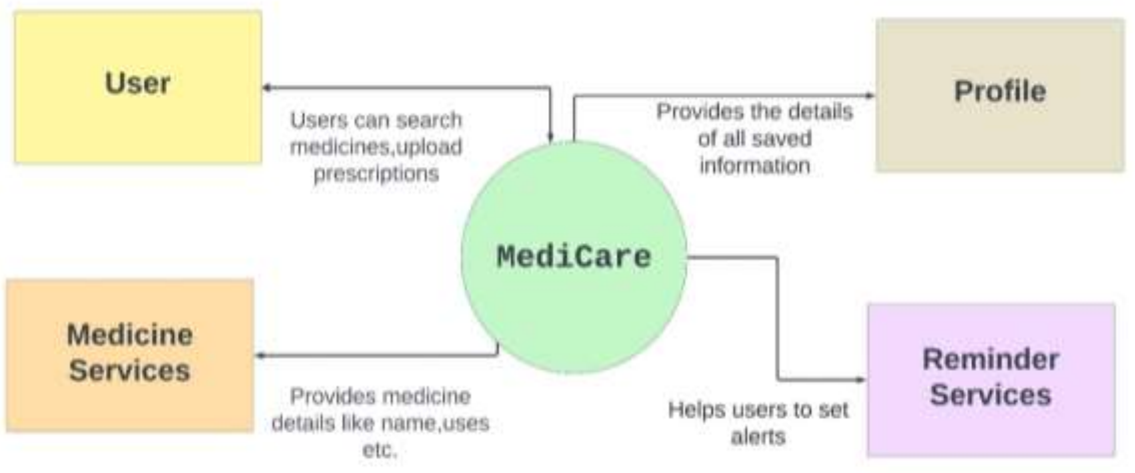

User
Users can search medicines,upload prescriptions
Provides the details of all saved information
Profile
MediCare
Medicine Services
Provides medicine details like name,uses etc.
Helps users to set alerts
Reminder Services

Fig. 3.2 DFD-0

3.3.2 DFD Level 1

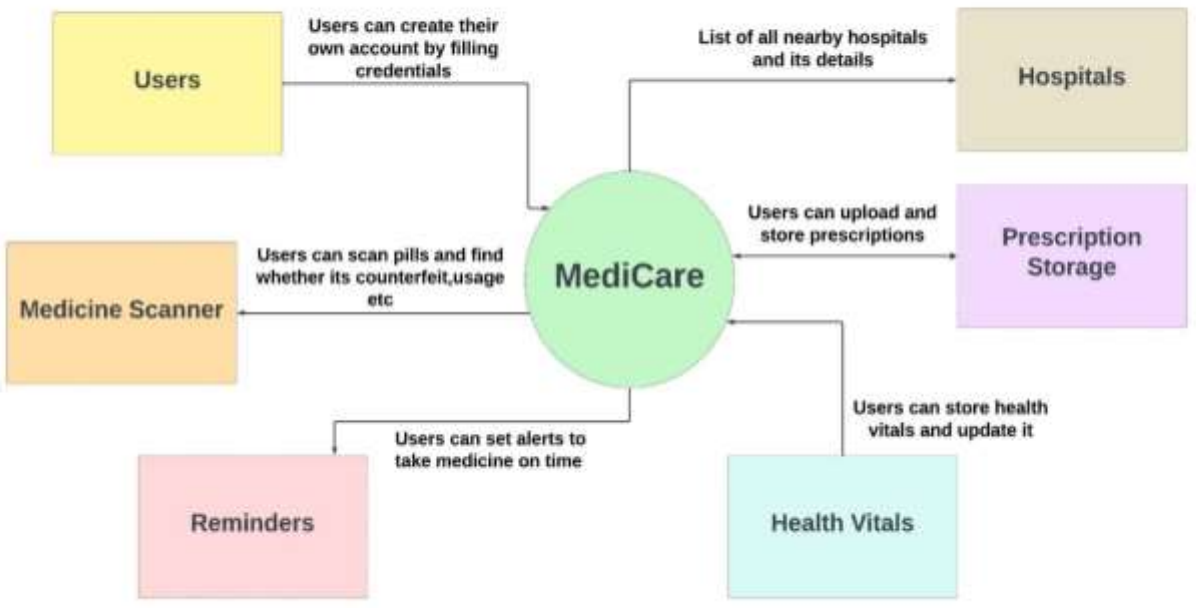

Users
Users can create their own account by filling credentials
List of all nearby hospitals and its details
Hospitals
Users can upload and store prescriptions
Prescription Storage
Users can scan pills and find whether its counterfeit,usage etc
Medicine Scanner
MediCare
Users can store health vitals and update it
Users can set alerts to take medicine on time
Reminders
Health Vitals

Fig. 3.3 DFD-1

3.3.3 DFD Level 2

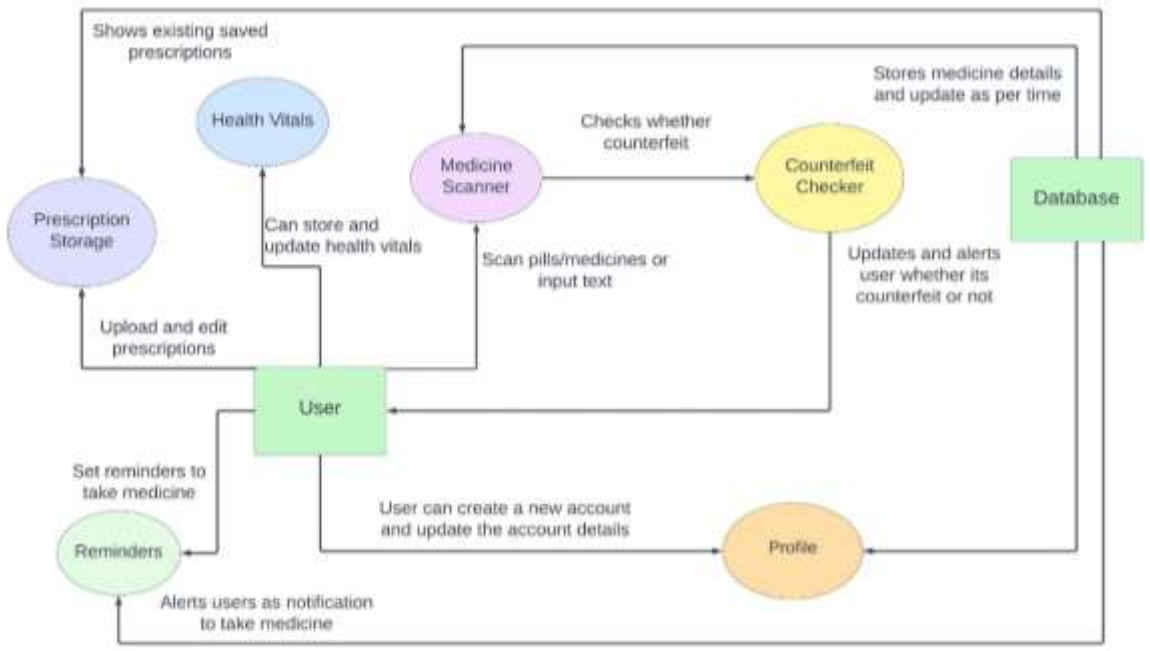

Shows existing saved prescriptions
Health Vitals
Medicine Scanner
Checks whether counterfeit
Counterfeit Checker
Stores medicine details and update as per time
Database
Prescription Storage
Can store and update health vitals
Scan pills/medicines or input text
Updates and alerts user whether its counterfeit or not
Upload and edit prescriptions
User
Set reminders to take medicine
User can create a new account and update the account details
Reminders
Profile
Alerts users as notification to take medicine

Fig. 3.4 DFD-2

The above diagram illustrates the data flow within the Medicare system, showing how different modules interact and exchange information. Users create accounts to access features such as prescription storage, health vitals tracking, and medicine scanning. The system processes user inputs, allowing them to upload and retrieve prescriptions, scan medicines for authenticity, and receive medication reminders. It also fetches hospital details based on location and enables users to store and update their health data. Each module communicates with the central Medicare system, ensuring seamless healthcare management and data processing.

3.4 ER DIAGRAM

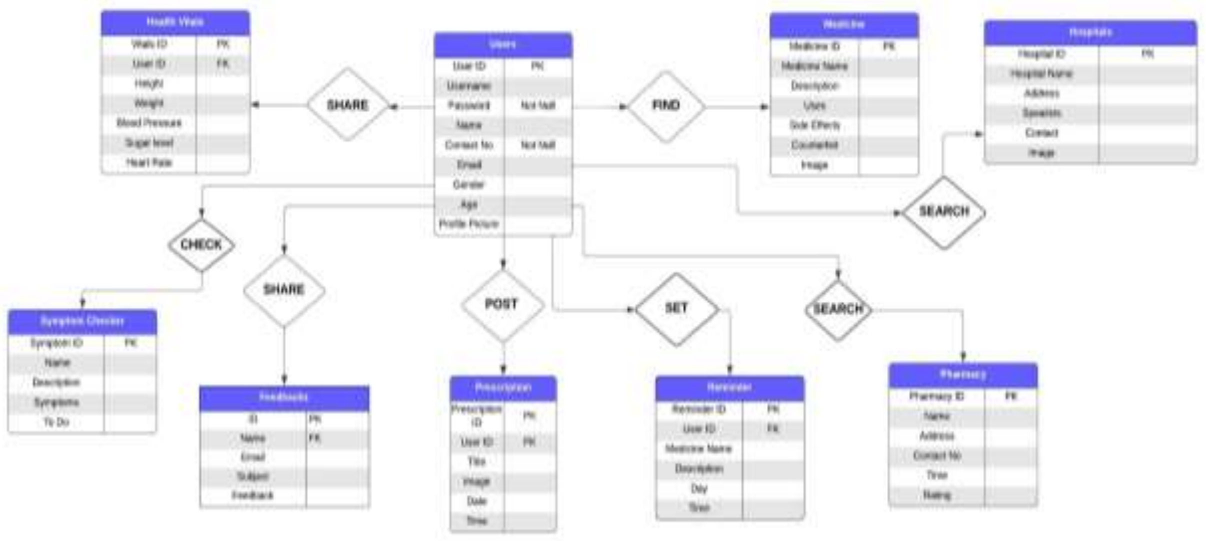


Fig 3.5 Entity Relationship Diagram

The above ER diagram depicts the entities involved in the proposed system and the relationship that they share between each of the other entities. The association links describes these relationships and the cardinality represents the number of instances of the entities and the participation tells what level the entity participation is important.

3.5 DATABASE DESIGN

3.5.1 Tables and Relationship

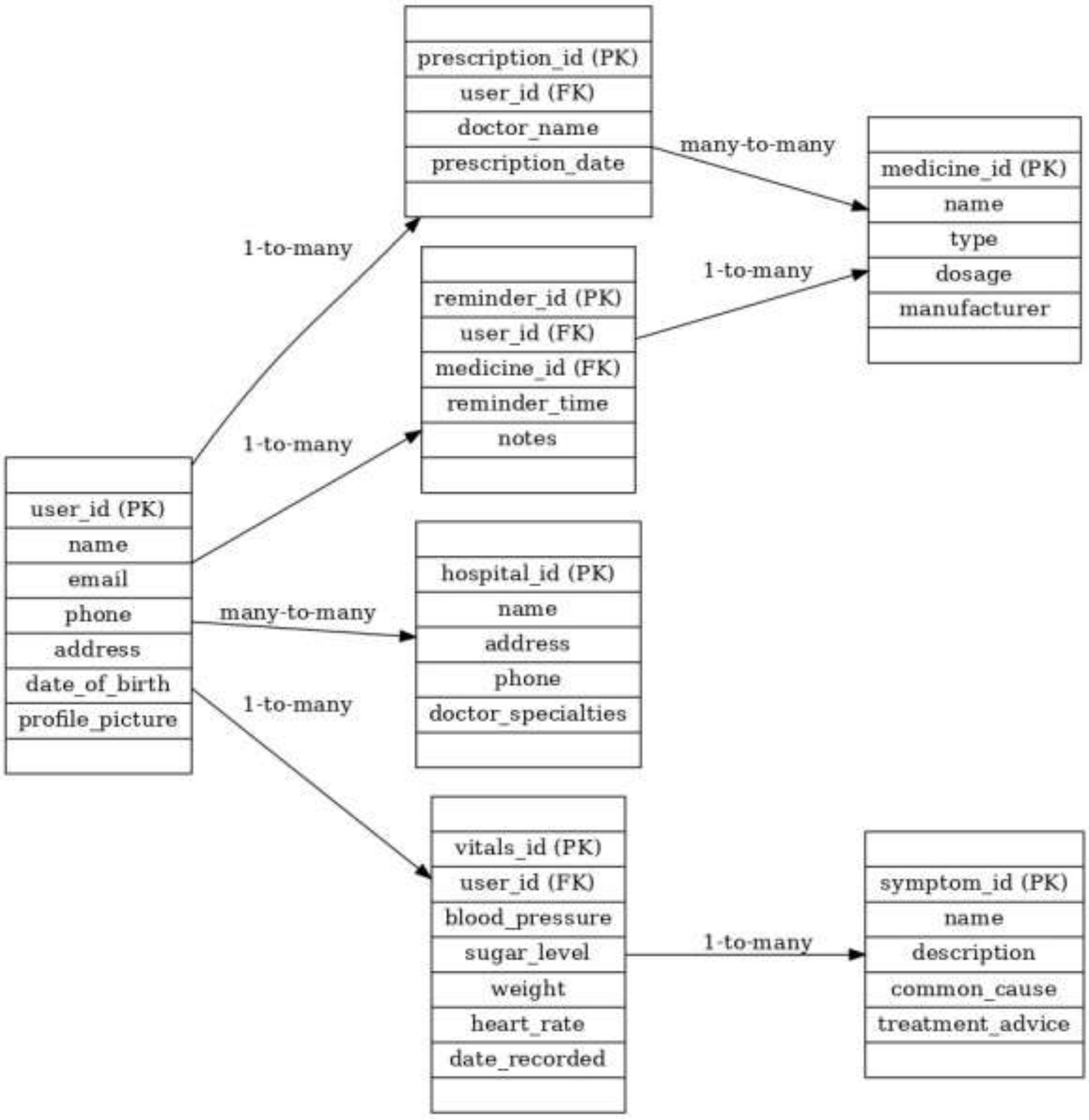


Fig 3.6 Table Relationship Diagram

The above diagram clearly shows the relationship that exists between the system database tables. Cardinality and the Participation links tells what number of instances and what level of participation is required by each entity. The primary key, foreign key links describes how each table are interconnected together in accessing common information.

3.5.2 Table Design

Field	Data Type	Length	Constraint
username	Varchar	16	FOREIGN KEY
password	Varchar	16	Not Null

Table 3.1 Login Table

Field	Data Type	Length	Constraint
Medicine_ID	Int	250	PRIMARY KEY
Medicine_Name	Varchar	50	Not Null
Medicine_Imprint	Varchar	50	Not Null
Description	Varchar	255	Default
Uses	Varchar	255	Default
Side_Effects	Varchar	255	Default
Counterfeit	Boolean	-	Default FALSE
Image	Image	10	Default

Table 3.2 Medicine Table

Field	Data Type	Length	Constraint
Reminder_ID	Int	250	PRIMARY KEY
User_ID	Int	250	FOREIGN KEY
Medicine_Name	Varchar	255	Not Null
Description	Varchar	255	Default
Day	Varchar	50	Default
Time	Time	-	Default

Table 3.3 Reminders Table

Field	Data Type	Length	Constraint
Prescription_ID	Int	250	PRIMARY KEY
User_ID	Int	250	FOREIGN KEY
Title	Varchar	255	Not Null
Image	Varchar	255	Default
Date	Date	-	Not Null
Time	Time	-	Not Null

Table 3.4 Prescriptions Table

Field	Data Type	Length	Constraint
User_ID	Int	250	PRIMARY KEY
Username	Varchar	100	Not Null
Email	Varchar	255	Default
Password	Varchar	255	Default
Contact	Int	255	Default
Age	Int	255	Default

Table 3.5 Users Table

Field	Data Type	Length	Constraint
Vitals_ID	Int	250	PRIMARY KEY
User_ID	Int	250	FOREIGN KEY
Height	Float	-	Default
Weight	Float	-	Default
Blood_Pressure	Varchar	255	Default
Sugar_Level	Varchar	255	Default
Heart_Rate	Int	12	Default

Table 3.6 Health Vitals Table

3.5.3 Data Integrity and Constraints

Data integrity constraints are essential rules that ensure the quality and accuracy of information stored in the database. Several measures have been implemented to protect sensitive data from unauthorized access. Additionally, data processed within the Medicare app undergoes Data Standardization, which converts raw data into a structured format to facilitate accurate processing and analysis. Standardized data ensures consistency, making it more relevant for medical evaluations. The database also follows Normalization principles, organizing data efficiently to eliminate redundancy and prevent duplication. This enhances data integrity and optimizes database performance.

With a strong focus on usability, the app is designed for seamless navigation, making it user-friendly and accessible.

1. User Authentication

- Users must create an account to access Medicare services.
- A secure digital wallet is integrated to store health records and manage transactions.
- User identity verification ensures secure access to medical services.

2. Medical Record Management

- **Patient ID**: Unique identifier for each patient, limited to 20 characters.
- **Prescription Records**: Standardized format to ensure consistency and accuracy

3.6 INTERFACE DESIGN AND PROCEDURAL DESIGN

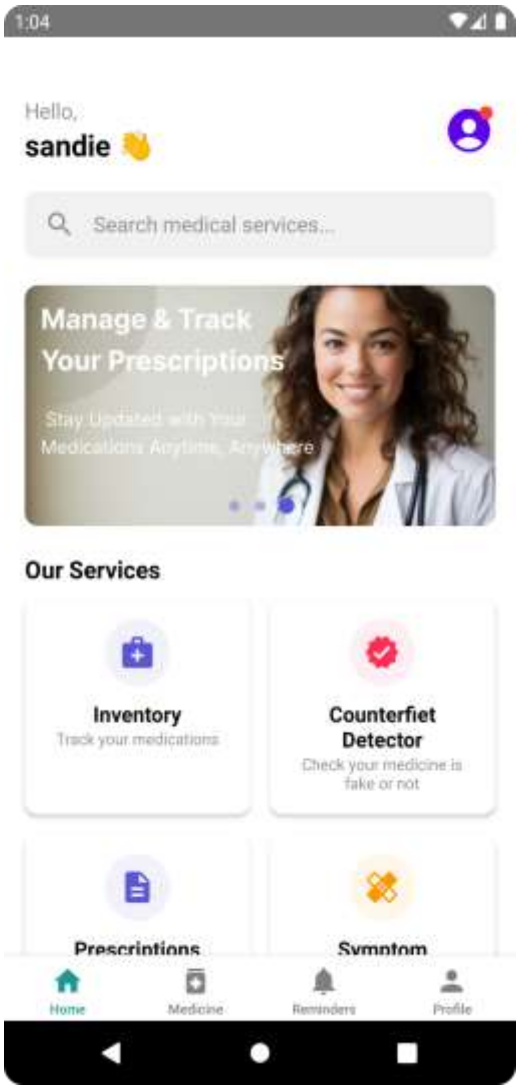


Fig 3.7 Home Page

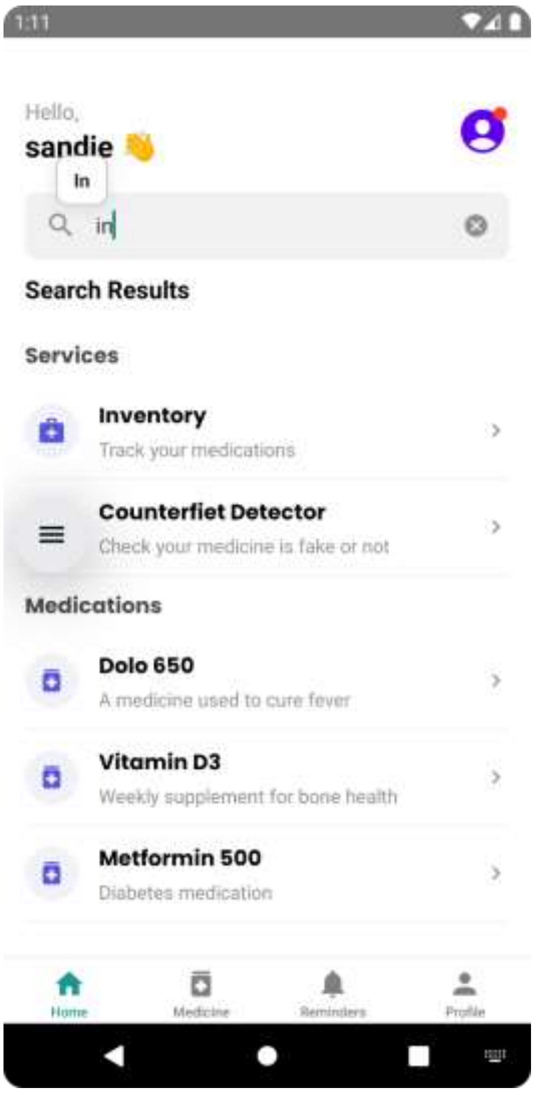


Fig 3.8 Search module Page

3.6.1 User Interface Design

The Medicare app is being designed for the following healthcare users, which provides user specific interface feature to perform role-based tasks.

a) **Elderly Users:** Older adults often struggle with managing medications due to memory decline, chronic illnesses, and complex regimens. Challenges like forgetting doses, vision impairments, or handling pills can affect adherence. Medicare's reminders, health tracking, and easy-to-use design empower older adults to manage medications independently, improving health outcomes and reducing stress.

b) **Patients with Chronic Conditions:** Patients with chronic conditions like diabetes, hypertension, or heart disease often require strict adherence to medication schedules to manage their health effectively. Missing doses or inconsistent tracking can lead to complications. Medicare provides medication reminders, tracking features, and personalized health insights, ensuring these patients stay on track and improve their long-term health outcomes.

c) **Caregivers:** Family members or professional caregivers responsible for managing the medication and health of others, such as elderly parents or patients with disabilities.

d) **Pharmacists and Healthcare Providers**: Professionals who may recommend the app to their patients for medication adherence and prescription tracking.

4. IMPLEMENTATION

This chapter consists of the coding standards used, the coding details, the code as well as the output of the desired implementation.

4.1 CODING STANDARDS

A well-defined coding standard ensures consistency, improves readability, enhances maintainability, and reduces complexity. It also facilitates code reuse and makes debugging easier.

The following coding standards are followed :

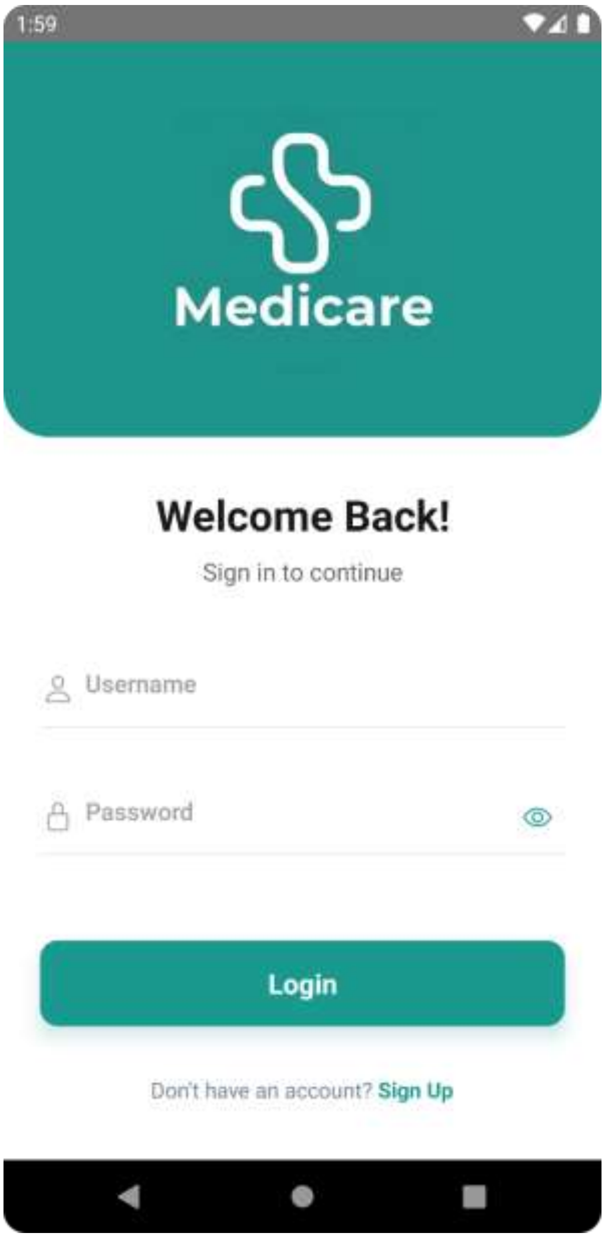
- **Indentation:** Used an indent of 4 spaces instead of tabs to maintain consistency across different systems.
- **Variable Naming:** Used lowercase letters with underscores (_) as word separators for better readability.
- **Control Structures:** Control statements (if, for, while, switch, etc.) include one space after the keyword for better readability.
- **Function Calls:** Functions are called with no spaces between the function name, opening parenthesis, and the first parameter. However, there are spaces between parameters separated by commas.
- **Docstrings:** Both single-line and multi-line docstrings are used in Python. Triple quotes are used for defining functions and program descriptions.
- **Comments:**
    - Short comments should form complete sentences with capitalized first words (unless starting with an identifier).
    - Block comments consist of multiple sentences, with each sentence ending with a period.
    - Both block comments and inline comments begin with #.
- **Declaration Block Alignment:** Blocks of declarations are properly aligned for better readability.
- **One Statement Per Line:** Each line contains only one statement, unless two statements are closely related.

4.2 CODING DETAILS

Server.js

```
require('dotenv').config({ path: 'E:\\Medi_Care\\.env' });
const express = require('express');
require('dotenv').config();
const { MongoClient } = require('mongodb');
const { ObjectId } = require('mongodb');

const app = express();


app.use(express.json());

// MongoDB connection function
const connectDB = async () => {
  try {
    const client = new MongoClient(process.env.MONGO_URI);
    await client.connect();
    console.log('MongoDB connected successfully');
    return client;
  } catch (error) {
    console.error('MongoDB connection failed:', error.message);
    process.exit(1);  // Exit process with failure on connection failure
  }
};

// API Routes
const main = async () => {
  const client = await connectDB();
  const db = client.db();
  const remindersCollection = db.collection('reminders');
  const usersCollection = db.collection('users');  // Specify collection name
  const healthCollection = db.collection('healthvitals');  // Specify collection
name
  const prescriptionsCollection = db.collection('prescriptions');  // Specify
collection name
  const inventoryCollection=db.collection('inventory');
  const medicationCollecton=db.collection('medicines')

  // Default route
  app.get('/', (req, res) => {
    res.send('API is running...');
  });

// User login endpoint
app.post('/login', async (req, res) => {
  try {
    const { username, password } = req.body;
    const user = await usersCollection.findOne({ username });

    if (!user) {return res.status(404).json({ error: 'User not found' });}

    // You should ideally use bcrypt to compare the password securely
    if (user.password !== password) {
      return res.status(404).json({ error: 'Incorrect Password' });
    }

    // Include the username in the response
    res.status(200).json({
      message: 'Login successful',
      isAdmin: user.isadmin,
      username: user.username, // Include username
    });
  } catch (error) {
    console.error('Error during login:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});

// User registration endpoint
app.post('/register', async (req, res) => {
  try {
    const { username, email, contact, age, gender, password } = req.body;
    const existingUser = await usersCollection.findOne({ username });
    if (existingUser) return res.status(400).json({ error: 'User already exists'
});

    // Insert new user
    const result = await usersCollection.insertOne({
      username, email, contact, age, gender, password
    });
    res.status(201).json({ message: 'User registered successfully', userId:
result.insertedId });
  } catch (error) {
    console.error('Error during registration:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});

app.get('/users/:username', async (req, res) => {
  try {
    const username = req.params.username;
    const user = await usersCollection.findOne({ username });
    if (user) {
      res.json(user);
    } else {
      res.status(404).json({ error: 'User not found' });
    }
  } catch (error) {
    res.status(500).json({ error: 'Server error' });
  }
});
// Update user profile endpoint
// BACKEND - server.js or routes/users.js
app.put('/users/:username', async (req, res) => {
  try {
    const { username } = req.params;
    const updates = req.body;

    // Remove any undefined or empty fields
    Object.keys(updates).forEach(key => {
      if (updates[key] === undefined || updates[key] === '') {
        delete updates[key];
      }
    });

    // Only update if there are valid fields to update
    if (Object.keys(updates).length === 0) {
      return res.status(400).json({
        success: false,
        message: 'No valid fields to update'
      });
    }

    const updatedUser = await usersCollection.findOneAndUpdate(
      { username: username },
      { $set: updates },
      { new: true }
    );

    if (!updatedUser) {
      return res.status(404).json({
        success: false,
        message: 'User not found'
      });
    }

    return res.status(200).json({
      success: true,
      message: 'Profile updated successfully',
      user: updatedUser
    });
  } catch (error) {
    console.error('Server error:', error);
    return res.status(500).json({
      success: false,
      message: 'Error updating profile'
    });
  }
});

// Fetch health vitals for a user
app.get('/healthvitals/:username', async (req, res) => {
  try {
    const username = req.params.username;
    // Fetch health vitals for the user from the healthvitals collection
    const healthVitals = await healthCollection.findOne({ username });

    if (healthVitals) {
      res.status(200).json(healthVitals);
    } else {
      res.status(404).json({ error: 'Health vitals not found for this user' });
    }
  } catch (error) {
    console.error('Error fetching health vitals:', error);
    res.status(500).json({ error: 'Server error' });
  }
});

app.post('/healthvitals/:username', async (req, res) => {
  try {
    const username = req.params.username;
    const healthVitals = req.body;

    // Use findOneAndUpdate to update or insert the health vitals
    const updatedVitals = await healthCollection.findOneAndUpdate(
      { username }, // Find by username
      { $set: healthVitals }, // Update with new health vitals
      { returnDocument: 'after', upsert: true } // Return the updated document and
create if it doesn't exist
    );

    res.status(200).send({
      message: 'Health vitals saved successfully!',
      data: updatedVitals.value // Return the updated or inserted document
    });
  } catch (error) {
    console.error('Error saving health vitals:', error);
    res.status(500).json({ error: 'Failed to save health vitals' });
  }
});


  // Add Reminder API
app.post('/addReminder', async (req, res) => {
  try {
    const { username, name, description, days, times, totalDoses } = req.body;

    // Add username to required fields validation
    if (!username || !name || !description || !days || !times || totalDoses ===
undefined) {
      return res.status(400).json({
        message: 'All fields are required including username'
      });
    }

    const updatedTimes = times.map(time => {
      const completedForDays = {};
      days.forEach(day => {
        completedForDays[day] = false;
      });
      return { ...time, completed: completedForDays };
    });

    const completed = {};
    days.forEach(day => {
      completed[day] = false;
    });

    const newReminder = {
      username, // Store the username with the reminder
      name,
      description,
      days,
      times: updatedTimes,
      totalDoses,
      completed,
      createdAt: new Date(),
    };

    const result = await remindersCollection.insertOne(newReminder);

    res.status(201).json({
      message: 'Reminder added successfully',
      reminderId: result.insertedId,
    });
  } catch (error) {
    console.error('Error adding reminder:', error.message);
    res.status(500).json({ message: 'Internal server error' });
  }
});

// Add a new endpoint to get reminders for a specific user
app.get('/reminders/:username', async (req, res) => {
  try {
    const { username } = req.params;

    if (!username) {
      return res.status(400).json({ message: 'Username is required' });
    }

    const reminders = await remindersCollection.find({ username }).toArray();

    res.status(200).json(reminders);
  } catch (error) {
    console.error('Error fetching reminders:', error.message);
    res.status(500).json({ message: 'Internal server error' });
  }
});

  // Get Reminders API
  app.get('/reminders', async (req, res) => {
    try {
      const reminders = await remindersCollection.find().toArray();
      const currentDay = new Date().toLocaleDateString('en-US', { weekday: 'short'
});

      const updatedReminders = reminders.map((reminder) => {
        reminder.times = reminder.times.map(timeObj => {
          if (timeObj.completed && timeObj.completed[currentDay]) {
            timeObj.completed[currentDay] = true;
          }
          return timeObj;
        });

        const allTimesCompletedForDay = reminder.times.every(timeObj =>
timeObj.completed[currentDay] === true);
        reminder.completed = allTimesCompletedForDay ? { [currentDay]: true } :
reminder.completed;

        return reminder;
      });

      res.status(200).json(updatedReminders);
    } catch (error) {
      console.error('Error fetching reminders:', error.message);
      res.status(500).json({ message: 'Internal server error' });
    }
  });

  // Update Reminder API
  app.patch('/reminders/:id', async (req, res) => {
    try {
      const { id } = req.params;
      const { time, days } = req.body;

      const day = Array.isArray(days) ? days[0] : days;

      const reminder = await remindersCollection.findOne({ _id: new ObjectId(id)
});

      if (!reminder) {
        return res.status(404).json({ message: 'Reminder not found' });
      }

      const timeObjIndex = reminder.times.findIndex(t => t.time === time);
      if (timeObjIndex === -1) {
        return res.status(404).json({ message: 'Time not found' });
      }

      reminder.times[timeObjIndex].completed[day] = true;

      const allTimesCompletedForDay = reminder.times.every(t => t.completed[day]
=== true);

      const result = await remindersCollection.updateOne(
        { _id: new ObjectId(id) },
        {
          $set: {
            [`times.${timeObjIndex}.completed.${day}`]: true,
            [`completed.${day}`]: allTimesCompletedForDay ? true :
reminder.completed[day],
          }
        }
      );

      if (result.modifiedCount > 0) {
        res.status(200).json({
          message: `Reminder marked as completed for the day (${day}).`,
          reminder,
          allTimesCompletedForDay
        });
      } else {
        res.status(404).json({ message: 'Failed to update reminder. No
modification occurred.' });
      }
    } catch (error) {
      console.error('Error updating reminder:', error.message);
      res.status(500).json({ message: 'Internal server error' });
    }
  });


  // Inventory API
  // API to fetch inventory items
  app.get('/inventory', async (req, res) => {
    try {
      const { username } = req.query;

      if (!username) {
        return res.status(400).json({ message: 'Username is required' });
      }

      // Find only inventory items created by this user
      const items = await inventoryCollection.find({ createdBy: username
}).toArray();

      res.status(200).json(items);
    } catch (error) {
      console.error('Error fetching inventory items:', error.message);
      res.status(500).json({ message: 'Internal server error' });
    }
  });



app.post('/inventory', async (req, res) => {
  try {
    const { name, price, stock, type, username } = req.body;

    // Validate required fields
    if (!name || !price || !stock || !type) {
      return res.status(400).json({
        message: 'All fields are required (name, price, stock, type)'
      });
    }

    // Validate user is logged in
    if (!username) {
      return res.status(401).json({
        message: 'Authentication required'
      });
    }

    // Create new inventory item with username
    const newItem = {
      name,
      price: parseFloat(price),
      inStock: parseInt(stock),
      type,
      createdBy: username,
      createdAt: new Date()
    };

    const result = await inventoryCollection.insertOne(newItem);

    res.status(201).json({
      message: 'Inventory item added successfully',
      itemId: result.insertedId,
      item: newItem
    });
  } catch (error) {
    console.error('Error adding inventory item:', error.message);
    res.status(500).json({ message: 'Internal server error' });
  }
});


app.get('/stats', async (req, res) => {
  try {
    // Fetch total items in the database
    const totalItems = await inventoryCollection.countDocuments();

    // Count items with low stock (inStock < 5 but > 0)
    const lowStock = await inventoryCollection.countDocuments({ inStock: { $lt: 5,
$gt: 0 } });

    // Count items that are out of stock (inStock === 0)
    const outOfStock = await inventoryCollection.countDocuments({ inStock: 0 });

    // Return statistics as an object
    res.status(200).json({
      totalItems,
      lowStock,
      outOfStock,
    });
  } catch (error) {
    console.error('Error fetching inventory stats:', error);
    res.status(500).json({ message: 'Error fetching inventory stats', error });
  }
});

// If you're sending base64 image data
app.use(express.json({limit: '50mb'}));
app.use(express.urlencoded({
  extended: true,
  limit: '50mb',
  parameterLimit: 100000
}));

app.post('/prescriptions', async (req, res) => {
  try {
    const prescription = {
      username: req.body.username,
      name: req.body.name,
      medication: req.body.medication,
      doctor: req.body.doctor,
      hospital: req.body.hospital,
      description: req.body.description,
      date: req.body.date,
      image: req.body.image,
      createdAt: new Date()
    };

    const result = await db.collection('prescriptions').insertOne(prescription);
    res.status(200).json({
      message: 'Prescription saved successfully',
      _id: result.insertedId
    });
  } catch (error) {
    console.error('Error saving prescription:', error);
    res.status(500).json({ error: error.message });
  }
});

// GET prescriptions by username
app.get('/prescriptions/:username', async (req, res) => {
  try {
    const prescriptions = await db
      .collection('prescriptions')
      .find({ username: req.params.username })
      .sort({ createdAt: -1 })
      .toArray();

    res.json(prescriptions);
  } catch (error) {
    console.error('Error fetching prescriptions:', error);
    res.status(500).json({ error: error.message });
  }
});

// DELETE prescription
app.delete('/prescriptions/:id', async (req, res) => {
  try {
    const result = await db
      .collection('prescriptions')
      .deleteOne({ _id: new ObjectId(req.params.id) });

    if (result.deletedCount === 0) {
      return res.status(404).json({ message: 'Prescription not found' });
    }

    res.status(200).json({ message: 'Prescription deleted successfully' });
  } catch (error) {
    console.error('Error deleting prescription:', error);
    res.status(500).json({ error: error.message });
  }
});

// GET single prescription
app.get('/prescriptions/detail/:id', async (req, res) => {
  try {
    const prescription = await db
      .collection('prescriptions')
      .findOne({ _id: new ObjectId(req.params.id) });

    if (!prescription) {
      return res.status(404).json({ message: 'Prescription not found' });
    }

    res.json(prescription);
  } catch (error) {
    console.error('Error fetching prescription:', error);
    res.status(500).json({ error: error.message });
  }
});




// Search endpoint
app.get('/search', async (req, res) => {
  const { query } = req.query;  // Expecting 'query' parameter from the request

  if (!query) {
    return res.status(400).json({ message: 'Query parameter is required.' });
  }

  try {
    // Search the database for medicines matching the query
    const medicines = await medicationCollecton.find({
      $or: [
        { name: { $regex: query, $options: 'i' } }, // Case-insensitive match for
name
        { imprint: { $regex: query, $options: 'i' } }, // Case-insensitive match
for imprint
      ]
    }).toArray(); // Use toArray() instead of lean()

    // Convert _id to string
    const formattedMedicines = medicines.map(medicine => ({
      ...medicine,
      _id: medicine._id.toString()
    }));

    // If no results found
    if (formattedMedicines.length === 0) {
      return res.status(404).json({ message: 'No medicines found.' });
    }

    // Return the list of matching medicines
    res.json(formattedMedicines);
  } catch (error) {
    console.error('Error searching for medicine:', error);
    res.status(500).json({ message: 'Server error while searching for medicine.'
});
  }
});

app.get('/api/medicine', async (req, res) => {
  try {
      const { query } = req.query;

      // Basic input validation
      if (!query || typeof query !== 'string') {
          return res.status(400).json({
              message: 'Invalid query parameter. Query must be a non-empty
string.'
          });
      }

      const sanitizedQuery = query.trim();
      // Search in both name and imprint fields
      const medicine = await medicationCollecton.findOne({
          $or: [
              { name: { $regex: sanitizedQuery, $options: 'i' } },
              { imprint: { $regex: sanitizedQuery, $options: 'i' } }
          ]
      });

      if (!medicine) {
          return res.status(404).json({
              message: `No medicine found matching "${sanitizedQuery}"`
          });
      }

      // Format the response
      const formattedMedicine = {
          ...medicine,
          _id: medicine._id.toString(),
          'image ': medicine['image '] ? medicine['image '].trim() : null
      };

      res.json(formattedMedicine);

  } catch (error) {
      console.error('Error fetching medicine:', error);
      res.status(500).json({
          message: 'Server error occurred while fetching medicine data'
      });
  }
});

 // Add this new endpoint to get current user data
 app.get('/api/current-user/:username', async (req, res) => {
  try {
    const { username } = req.params;

    if (!username) {
      return res.status(400).json({
        success: false,
        message: 'Username is required'
      });
    }

    const user = await usersCollection.findOne(
      { username },
      { projection: { password: 0 } } // Exclude password from the response
    );
    if (!user) {
      return res.status(404).json({
        success: false,
        message: 'User not found'
      });
    }

    return res.status(200).json({
      success: true,
      user: {
        _id: user._id,
        username: user.username,
        email: user.email,
        contact: user.contact,
        age: user.age,
        gender: user.gender,
        isAdmin: user.isadmin
      }
    });

  } catch (error) {
    console.error('Error fetching current user:', error);
    return res.status(500).json({
      success: false,
      message: 'Error fetching user data'
    });
  }
});

app.get('/api/remind/:username', async (req, res) => {
  try {
    const { username } = req.params;
    const reminders = await remindersCollection.find({ username }).toArray();
    res.json({
      success: true,
      reminders: reminders
    });
  } catch (error) {
    console.error('Error fetching reminders:', error);
    res.status(500).json({
      success: false,
      error: 'Failed to fetch reminders'
    });
  }
});

//main
const multer = require("multer");
const fs = require("fs");
const axios = require("axios");;

// Configure multer for handling file uploads
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, "uploads/");
  },
  filename: function (req, file, cb) {
    cb(null, Date.now() + "-" + file.originalname);
  }
});

// Set up multer with the storage configuration
const upload = multer({
  storage: storage,
  limits: {
    fileSize: 5 * 1024 * 1024 // 5MB limit
  }
});

// Make sure upload directory exists
if (!fs.existsSync("uploads/")) {
  fs.mkdirSync("uploads/");
}

app.post("/detect-counterfeit", upload.single("image"), async (req, res) => {
  try {
    if (!req.file) {
      return res.status(400).json({ error: "No image uploaded" });
    }

    console.log("File received:", req.file);

    // Read the uploaded image file and convert it to base64
    const imageBase64 = fs.readFileSync(req.file.path, {
      encoding: "base64"
    });

    // Send request to Roboflow API
    const response = await axios({
      method: "POST",
      url: "https://detect.roboflow.com/fake-med/1",
      params: {
        api_key: "WRHcEkNdWcTa1wqzfGQs"
      },
      data: imageBase64,
      headers: {
        "Content-Type": "application/x-www-form-urlencoded"
      }
    });

    // Log the full API response for debugging
    console.log("Full API response:", JSON.stringify(response.data, null, 2));

    // Initialize result with default values - EXPLICITLY set isCounterfeit to
false
    let result = {
      isCounterfeit: false,
      confidence: 0,
      currencyType: null,
      denomination: null,
      features: []
    };

    const predictions = response.data.predictions || [];

    if (predictions.length > 0) {
      // Get highest confidence prediction
      const topPrediction = predictions.reduce((prev, current) =>
        (prev.confidence > current.confidence) ? prev : current
      );

      console.log("Top prediction class:", topPrediction.class);
      console.log("Top prediction confidence:", topPrediction.confidence);

      // Store confidence regardless of class
      result.confidence = topPrediction.confidence;

      // SIMPLE LOGIC: Only mark as counterfeit if class is EXACTLY "counterfeit"
      // For any other class (including "authentic"), keep isCounterfeit as false
      if (topPrediction.class === "counterfeit") {
        result.isCounterfeit = true;
        console.log("DETECTED AS COUNTERFEIT");
      } else {
        result.isCounterfeit = false;
        console.log("NOT DETECTED AS COUNTERFEIT - class is:",
topPrediction.class);
      }

      // Add any feature points if available
      if (topPrediction.points) {
        result.features = topPrediction.points.map(point => point.class);
      }
    }
    console.log("Final result being sent:", result);

    // Delete temp file
    fs.unlinkSync(req.file.path);

    // Send result
    return res.json(result);

  } catch (error) {
    console.error("Error in counterfeit detection:", error);

    if (error.response) {
      console.error("API error details:", {
        data: error.response.data,
        status: error.response.status
      });
    }

    return res.status(500).json({
      error: "Failed to process image",
      message: error.message
    });
  }
});


app.post('/logout', (req, res) => {
    // For token-based auth, you might blacklist the token (using a database or
in-memory storage)
    res.status(200).json({ message: 'User logged out successfully' });
  });

  // Start the server
  const PORT = process.env.PORT || 5000;
  app.listen(PORT, '0.0.0.0', () => {
  console.log(`Server running on http://20.193.156.237:${PORT}`);
});
};

// Start the main function
main().catch((err) => console.error(err));
```

4.3 SCREENSHOTS


1:59

Medicare

Welcome Back!

Sign in to continue

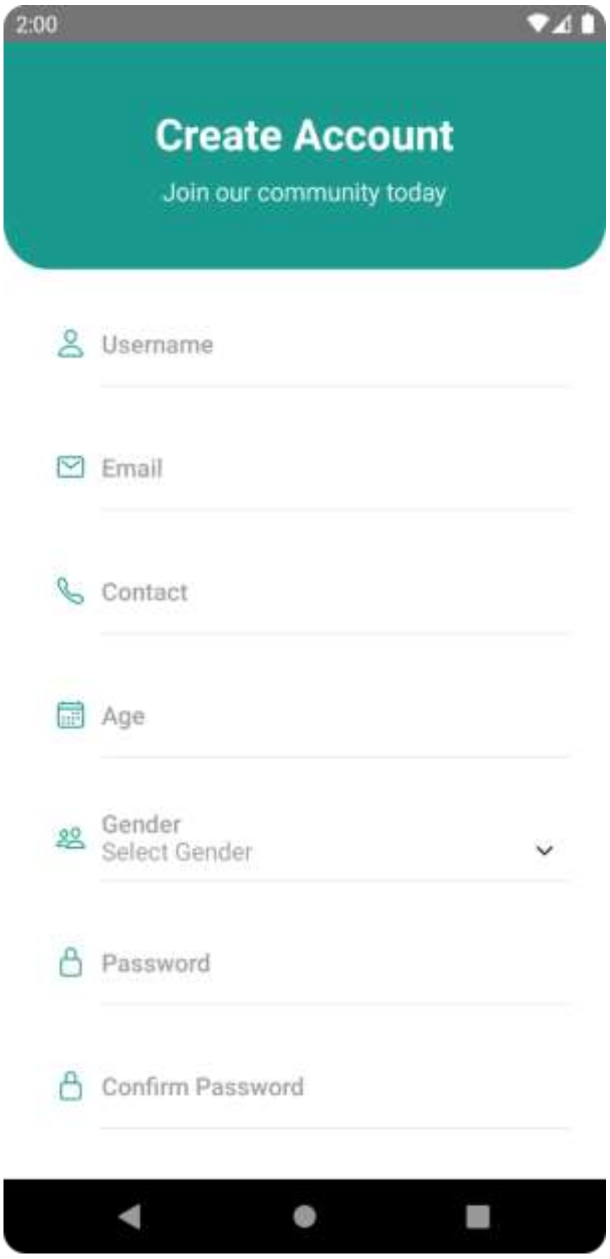
Username

Password

Login

Don't have an account? Sign Up

Fig 4.1 Login Page


2:00

Create Account

Join our community today

Username

Email

Contact

Age

Gender
Select Gender

Password

Confirm Password

Fig 4.2 Sign Up Page

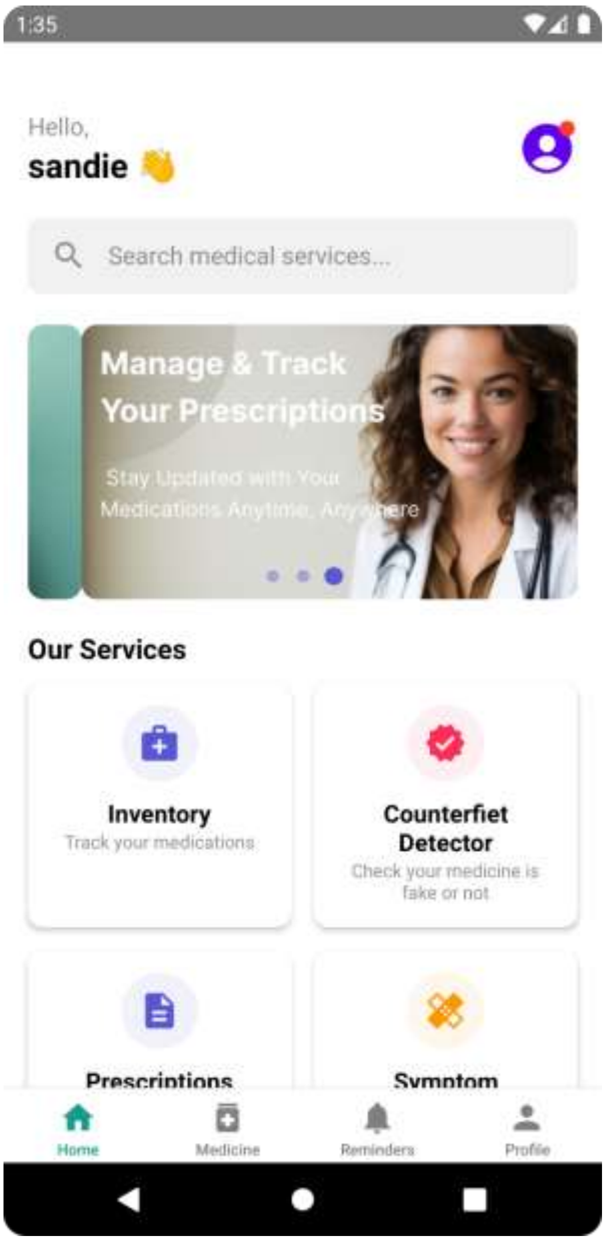


Fig 4.3 Home Page

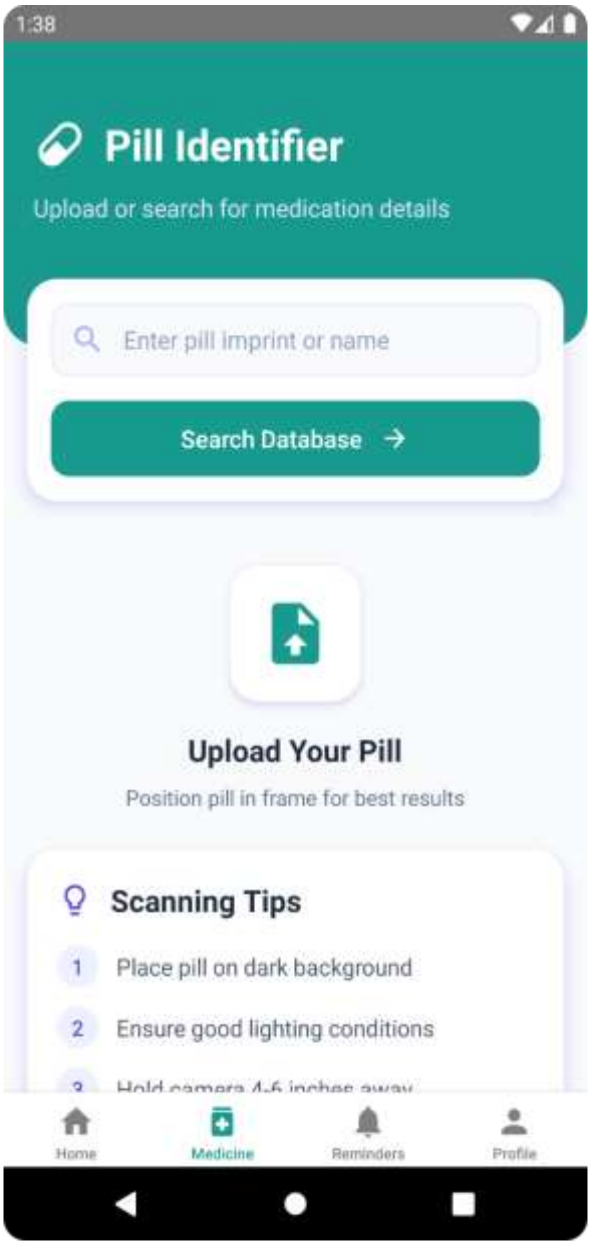
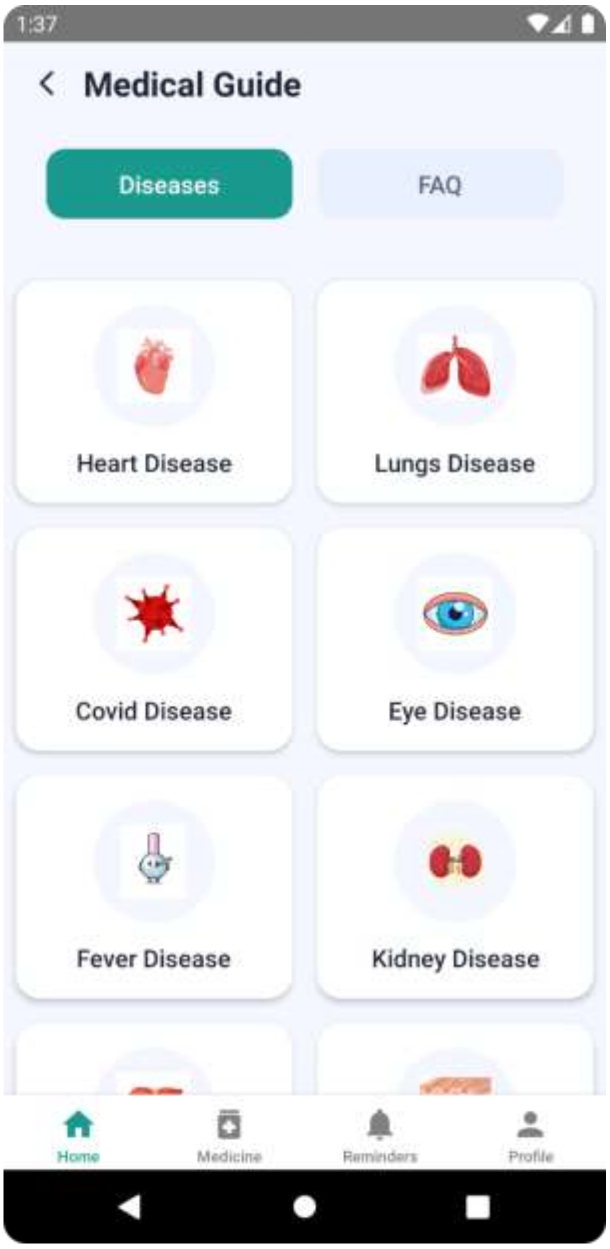

1:38
Pill Identifier
Upload or search for medication details
Enter pill imprint or name
Search Database →
Upload Your Pill
Position pill in frame for best results
Scanning Tips
1 Place pill on dark background
2 Ensure good lighting conditions
3 Hold camera 4-6 inches away
Home
Medicine
Reminders
Profile

Fig 4.4 Pill Identifier Page



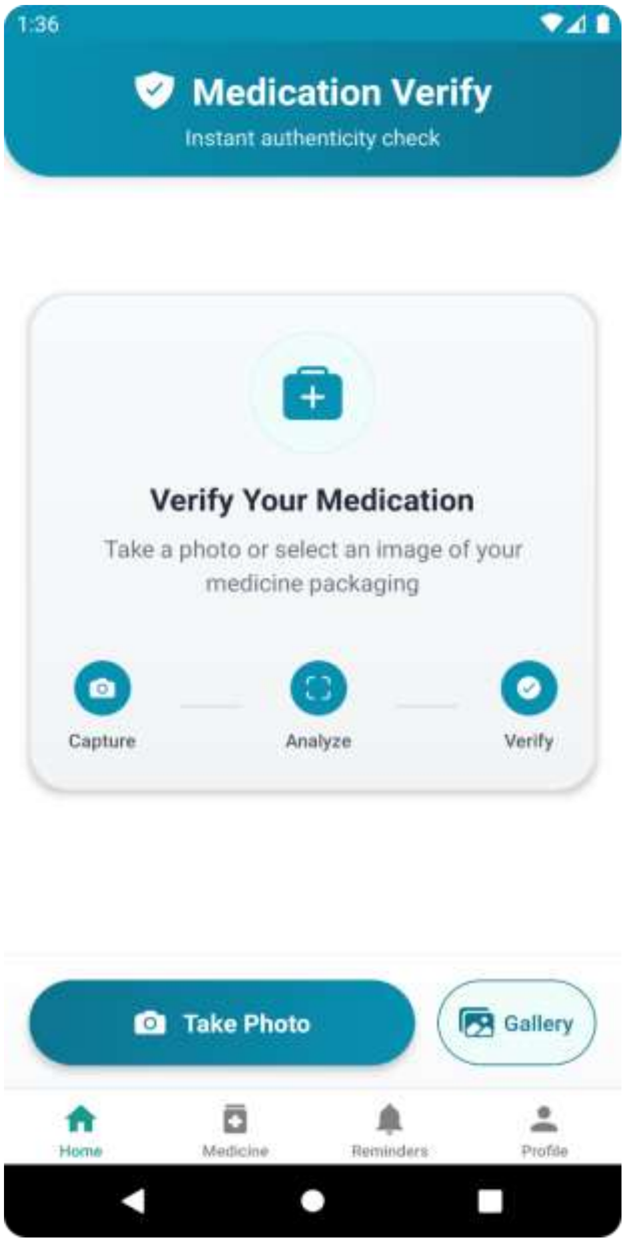
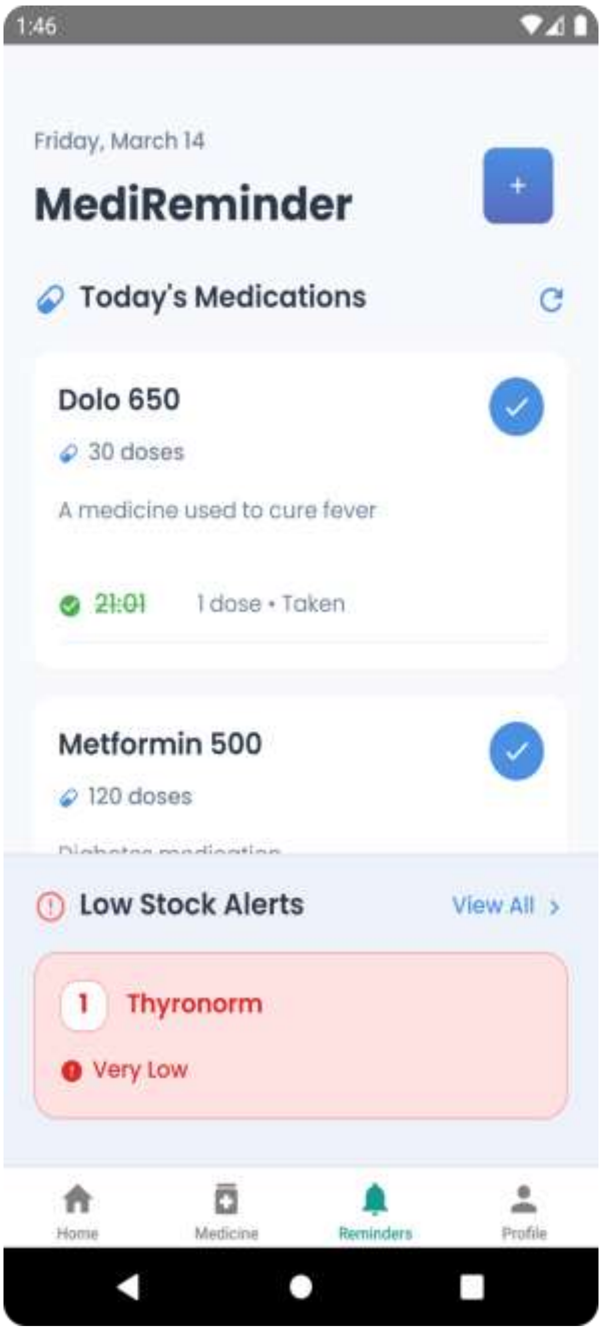


Fig 4.5 Symptom Checker Page

Fig 4.6 Counterfeit Detector Page


1:46

Friday, March 14

MediReminder

Today's Medications

Dolo 650

30 doses

A medicine used to cure fever

~~21:01~~ 1 dose • Taken

Metformin 500

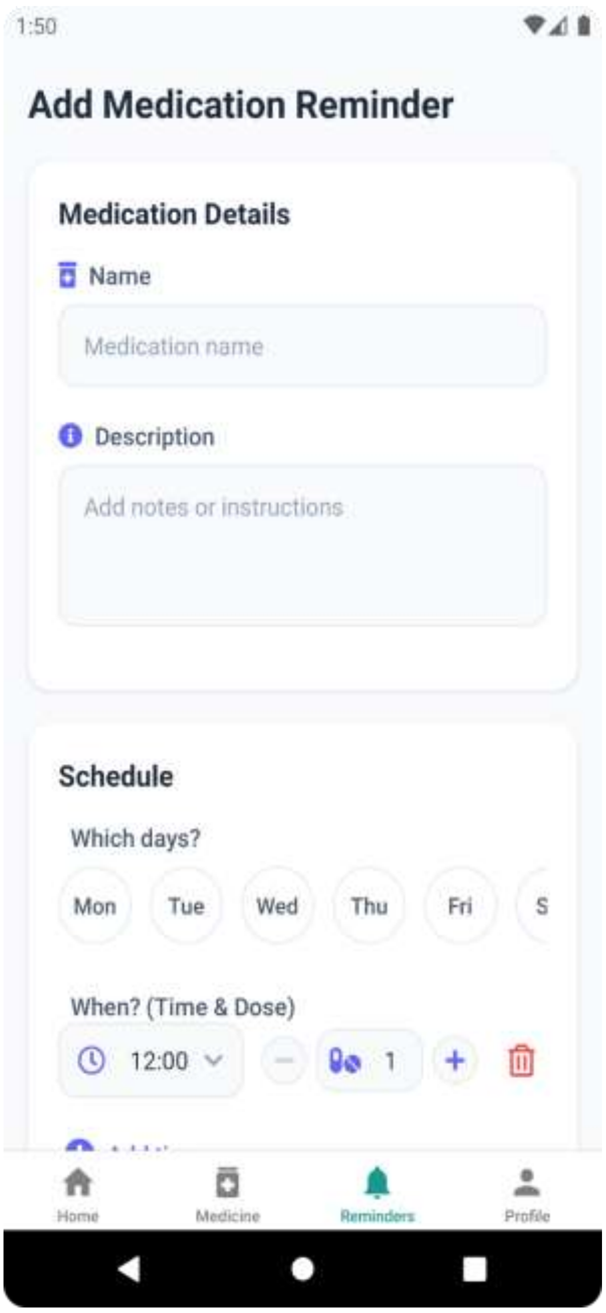
120 doses

Low Stock Alerts View All

1 Thyronorm

Very Low

Fig 4.7 Reminder Page


1:50

Add Medication Reminder

Medication Details

Name

Medication name

Description

Add notes or instructions

Schedule

Which days?

Mon Tue Wed Thu Fri S

When? (Time & Dose)

12:00 1

Fig 4.8 Add Reminder Page

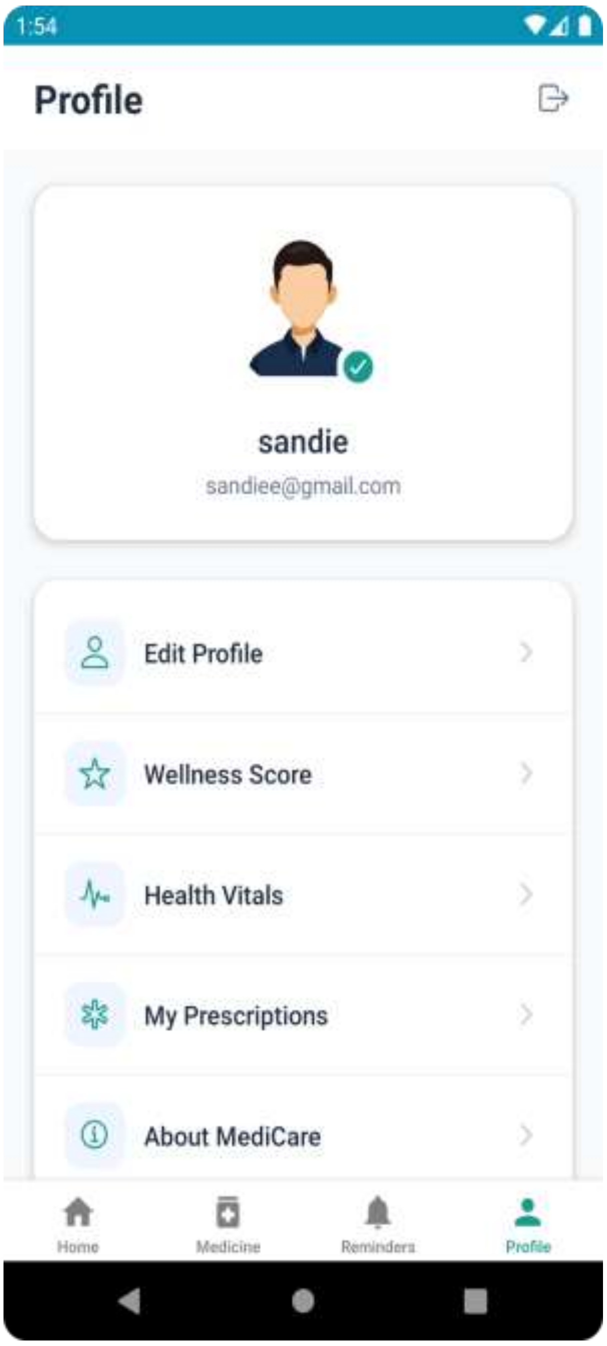


Fig 4.9 Profile Page

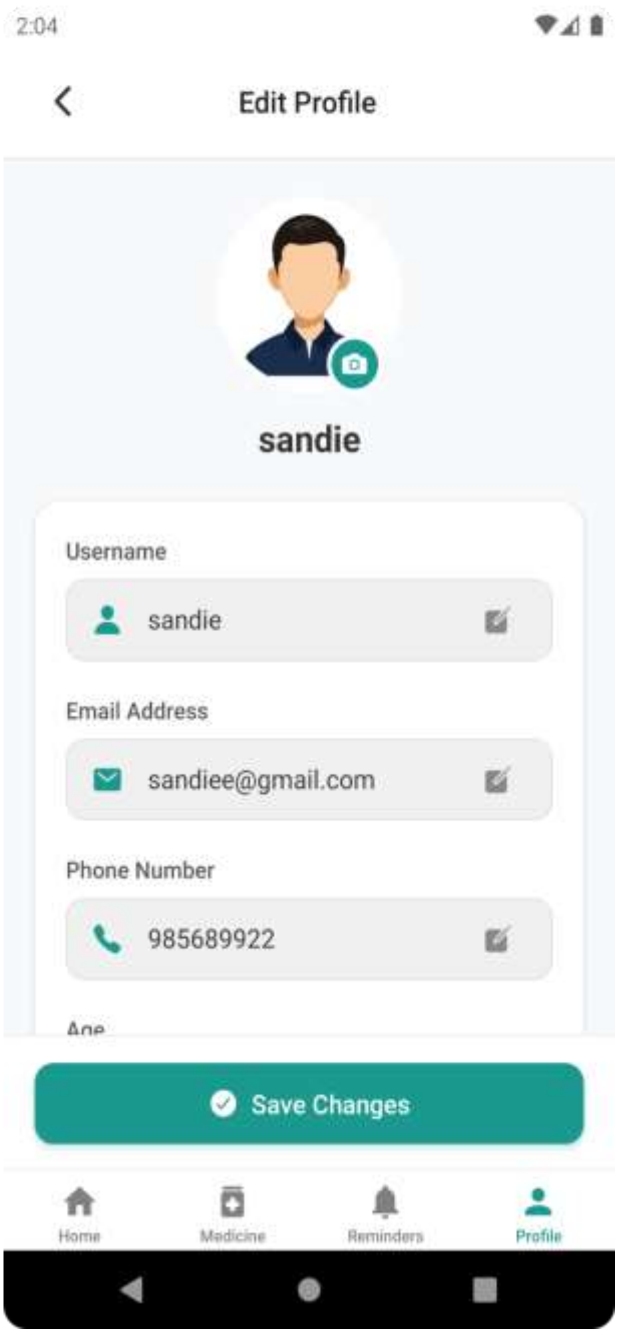
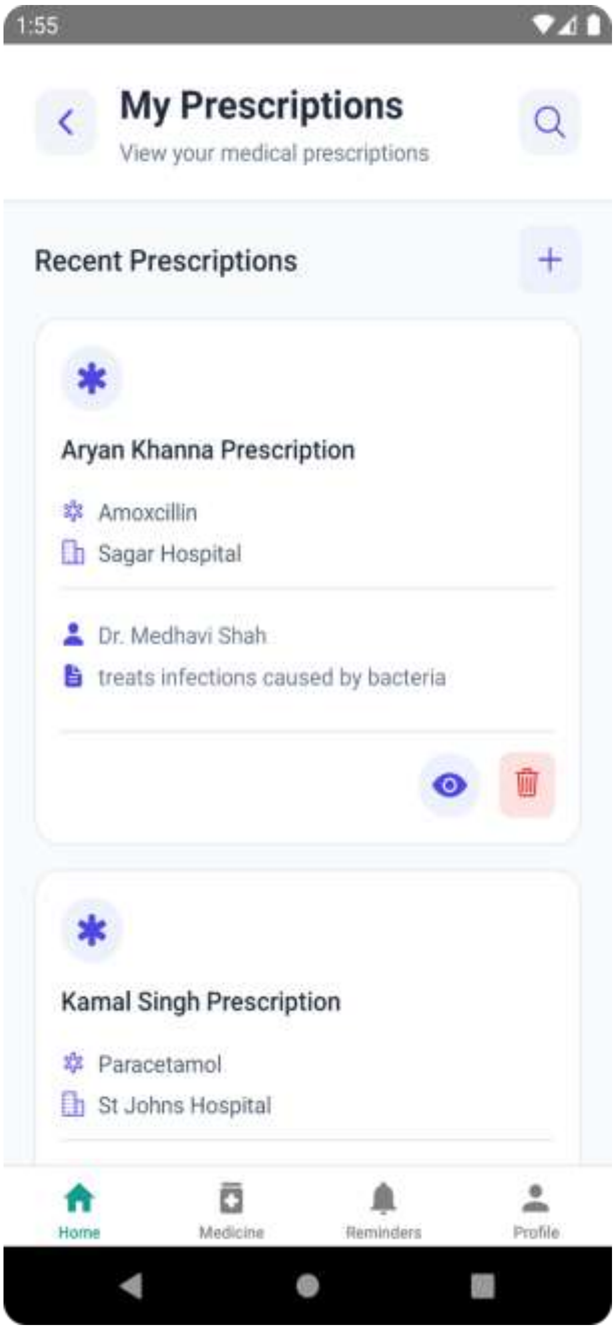


Fig 4.10 Edit Profile Page



1:55

My Prescriptions

View your medical prescriptions

Recent Prescriptions

Aryan Khanna Prescription

Amoxcillin

Sagar Hospital

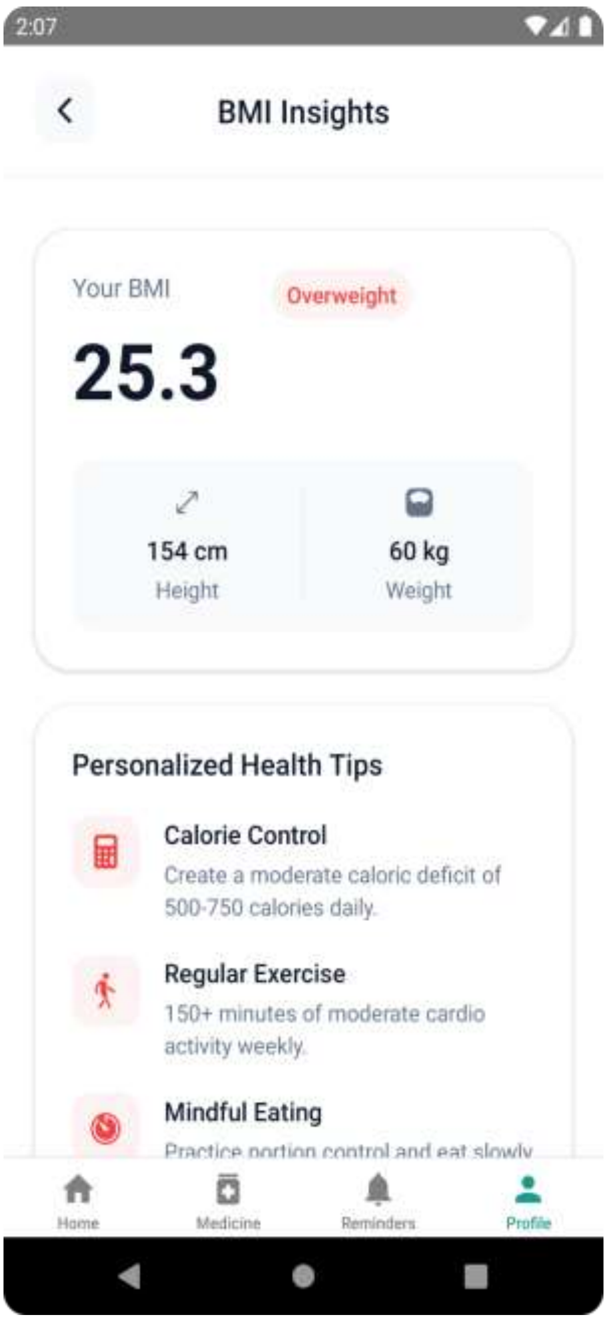
Dr. Medhavi Shah

treats infections caused by bacteria

Kamal Singh Prescription

Paracetamol

St Johns Hospital



2:07

BMI Insights

Your BMI Overweight

25.3

154 cm Height

60 kg Weight

Personalized Health Tips

Calorie Control

Create a moderate caloric deficit of 500-750 calories daily.

Regular Exercise

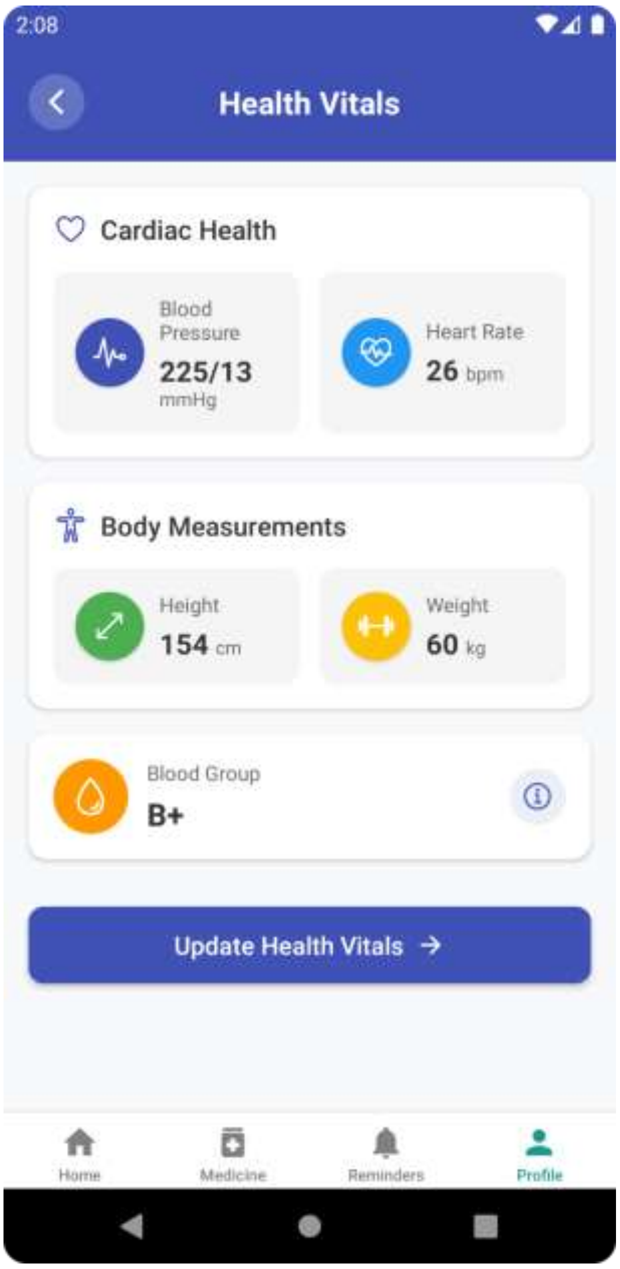
150+ minutes of moderate cardio activity weekly.

Mindful Eating

Practice portion control and eat slowly

Fig 4.11 Prescription Page

Fig 4.12 BMI Insights Page


2:08

Health Vitals

Cardiac Health

Blood Pressure 225/13 mmHg

Heart Rate 26 bpm

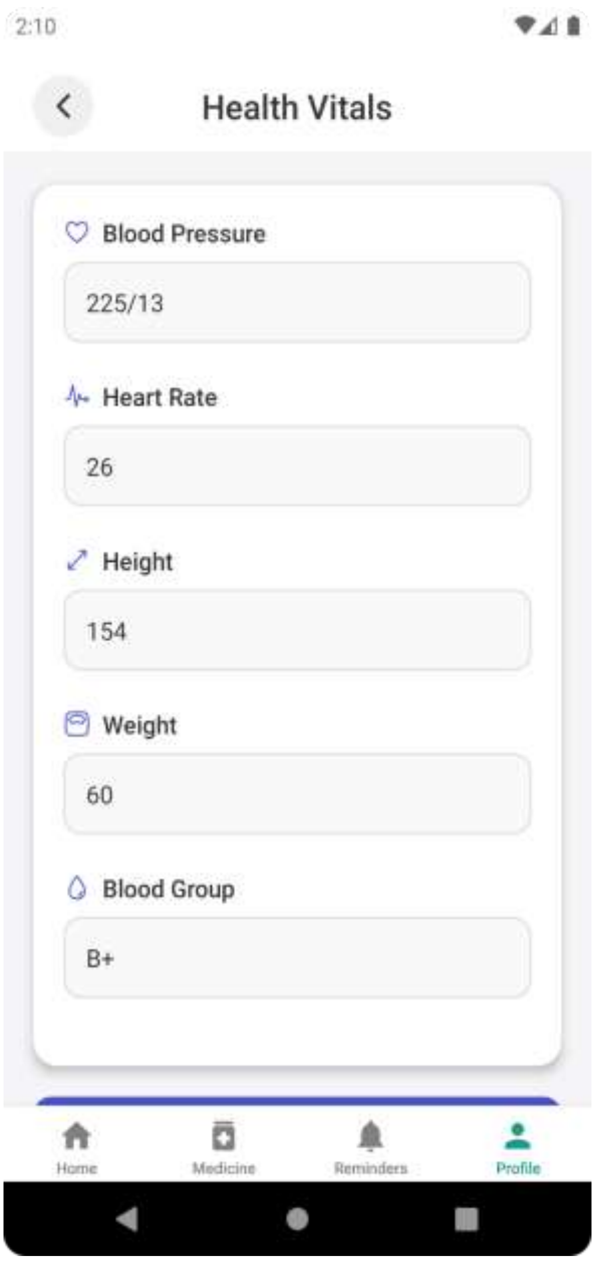
Body Measurements

Height 154 cm

Weight 60 kg

Blood Group B+

Update Health Vitals →


2:10

Health Vitals

Blood Pressure: 225/13

Heart Rate: 26

Height: 154

Weight: 60

Blood Group: B+

Fig 4.13 Health Vitals Page

Fig 4.14 Edit Health Vitals Page

5. TESTING

5.1 TEST STRATEGIES

5.1.1 System Testing

System testing is a critical element of quality assurance and represents the ultimate review of analysis, design and coding. Test case design focuses on set of techniques for the creation of test because that meet overall testing objectives. When a system is developed it is hoped that it performs properly. The main purpose of testing an information system Is to find the errors and correct them. The scope of system testing should include both manual and computerized operations. System testing is a comprehensive evaluation of the programs, manual procedures, computer operations and controls.

System testing is the process of checking whether the developed system is working according to the objective and requirement. All testing is to be conducted in accordance to the test conditions specified earlier. This will ensure that the test coverage meets the requirements and that testing is done in a systematic manner.

5.1.2 Functionality Testing

The functionality testing ensured that the medicine scanner accurately detects and identifies medicines. The counterfeit detection system was tested to verify that it provides correct results in distinguishing genuine medicines from counterfeit ones. Additionally, the global medicine checker was evaluated to confirm that it retrieves accurate and relevant medicine information.

5.1.3 Integration Testing

Integration testing focused on ensuring seamless interaction between different components of the system, including the camera module, ML model, and database. The API requests for medicine data were also tested to confirm that they function correctly, retrieving real-time information without errors.

5.1.4 Performance Testing

Performance testing was conducted to measure the response time of the medicine scanning feature and result retrieval. The app’s performance was evaluated across different Android versions and devices to ensure consistent functionality and efficiency.

5.1.5 Usability Testing

Usability testing aimed to ensure that the app's user interface is intuitive and user-friendly. The navigation and accessibility of the app were assessed to confirm that they meet usability standards, providing a smooth experience for users.

5.1.6 Test Data Implementation

Test data was created and implemented to evaluate the performance and accuracy of the Medicare application. The test data included both real and synthetic medicine details to ensure the robustness of the system.

The implementation process of test data involved several key steps to ensure the accuracy and efficiency of the system. Dataset preparation began with collecting medicine images from various sources, including online databases and manually captured photos. Each medicine was labelled as either genuine or counterfeit for ML model training and testing, with additional metadata such as medicine name, manufacturer, batch number, and expiry date included for better identification. Data integration involved uploading the test data into the application's database, ensuring seamless retrieval during testing, and using APIs to fetch real-time medicine data for verification.

The system was then tested with various inputs, including valid data, where real medicine images and details were used to verify correct identification; invalid data, consisting of counterfeit or altered medicine images to assess detection accuracy; and edge cases, such as low-quality images, blurred labels, and incomplete medicine details, to evaluate the system's robustness in real-world scenarios.

5.1.7 Test Characteristics

1. A good test should effectively identify counterfeit medicines with high accuracy.
2. A good test should avoid redundant checks and focus on unique detection scenarios.

3. A good test should use the most reliable and efficient methods for medicine verification.
4. A good test should balance complexity, ensuring it is neither too simplistic nor overly intricate for practical use.

5.2 TEST CASES

Sl. No.	Module Name	Test Case No.	Test Case Description	Expected Result
1	User Authentication	TC1	Verify if the user can register successfully.	User should be able to register with valid details.
2	Login Module	TC2	Verify if the user can log in with credentials.	User should access the system with correct login details.
3	Medicine Scanner	TC3	Verify if the app scans medicine labels correctly.	The app should correctly recognize medicine details.
4	Prescription Management	TC4	Verify if the user can upload prescriptions.	The user should be able to upload a valid prescription.

5	Appointment Booking	TC5	Verify if users can book doctor appointments.	The user should successfully book an available slot.
6	Contact Us Module	TC6	Verify if the user enters their full name.	The user must enter their full name.
7	Contact Us Module	TC7	Verify if the user enters a valid email.	The user must enter a valid email.
8	Contact Us Module	TC8	Verify if the user enters a message before submission.	The user must enter a message.

Table 5.1 Test Case Table

5.3 TEST REPORTS

Sl. No.	Test Case No.	Test Status	Test Result
1	TC1	Successful	Users can register with valid details.
2	TC2	Successful	Users can log in with correct credentials.
3	TC3	Successful	Medicine scanning functions correctly.
4	TC4	Successful	Users can upload prescriptions successfully.
5	TC5	Successful	Appointment booking works as expected.
6	TC6	Successful	Name field validation works correctly.
7	TC7	Successful	Email validation is implemented correctly.
8	TC8	Successful	Message submission works as expected.

Table 5.2 Test Report Table

6.CONCLUSION

The major implementation and design issues along with advantages and disadvantages of the project is properly mentioned. The future scope of the project is also mentioned in this chapter.

6.1 DESIGN AND IMPLEMENTATION ISSUES

Poor design and implementation can impact the effectiveness and acceptance of a software system.

6.1.1 Design Issues

During the design phase, several challenges were encountered. Integrating the machine learning model with the mobile application required careful optimization to ensure efficient processing and accurate medicine detection. Designing a user-friendly interface that accommodates complex functionalities, such as medicine scanning and counterfeit detection, posed challenges in layout alignment and user accessibility. Additionally, ensuring seamless interaction between the camera module and the backend system required multiple design iterations.

6.1.2 Implementation Issues

The implementation phase also presented various difficulties. Initially, integrating external APIs for real-time medicine verification required handling different data formats and ensuring consistent responses. The model's accuracy had to be fine-tuned to correctly classify medicines, especially when dealing with low-quality images or incomplete medicine details. Furthermore, optimizing the application for different Android devices to ensure smooth performance across various screen sizes and operating system versions required extensive testing and debugging.

6.2 ADVANTAGES AND LIMITATIONS

6.2.1 Advantages

- The application enables users to scan and verify medicines instantly.
- The counterfeit detection feature helps users distinguish between genuine and fake medicines.
- A global medicine checker provides users with essential drug-related information.
- The user-friendly interface ensures easy navigation and accessibility.
- The app supports real-time API integration for accurate medicine verification.

6.2.2 Limitations

- The application requires an internet connection to fetch real-time medicine data.
- The accuracy of counterfeit detection depends on the quality and availability of medicine images in the database.
- The app currently supports only a limited number of pharmaceutical databases.

6.3 FUTURE SCOPE OF THE PROJECT

With the increasing concern over counterfeit medicines, there is significant potential for expanding the Medicare app. Future enhancements could include integrating AI-driven prescription validation to ensure medication safety. Expanding the database to include a wider range of pharmaceutical products and international medicines will improve detection accuracy. Implementing blockchain technology for secure and tamper-proof medicine verification can further enhance reliability. Additionally, the application can be extended to include patient health tracking features, such as dosage reminders and interactions with healthcare providers. These improvements will strengthen the app's role in ensuring medicine authenticity and user safety in the healthcare sector.

REFERENCES

[1] P. Gupta, K. Singhal, and A. Pandey, “Counterfeit (Fake) Drugs & New Technologies to Identify it in India,” vol. 3, no. 11, pp. 4057 4064, Oct. 2012.

[2] R. Kumar and R. Tripathi, “Traceability of counterfeit medicine supply chain through Blockchain,” 2019, pp. 568–570. doi: 10.1109/COMSNETS.2019.8711418.

[3] Big Problem, Smart Solution: An App That IDs the Pills You're Taking, Aug 8,2013. https://www.wired.com/2013/08/an-app-thats-looking-to-document-the-worlds-pill-supply/

[4] “DNA,” How to differentiate counterfeit medicines from genuine ones, Jun. 24, 2021. https://www.dnaindia.com/analysis/report-dna special-how-to-differentiate-counterfeit medicines-from-genuine-ones-2897092

.