# MediPulse Project Documentation

**Version:** 1.0 **Date:** May 04, 2025

**Table of Contents:**

1. [Project Overview](#section-1-project-overview)
   * [1.1 Introduction](#X95cfb64b4e7a3c68f41b7aea44ce2e38e5bbf88)
   * [1.2 Problem Statement](#Xb3d3cf96436a36cadb11a8579eb11996681af9c)
   * [1.3 Project Goals and Objectives](#X48f6ae4e1317448871ce7edd98032abf866eef2)
   * [1.4 Scope and Limitations](#Xffbffe24e84596729c20914de87fed5cb382607)
   * [1.5 Target Audience (Application Users)](#X2ee7f74139a5d1bfa06df1f2f373873f3c1d34a)
   * [1.6 System Components Overview](#Xf2669ee89a33acfb207d0d4f28c6dc19373c7df)
   * [1.7 Technology Stack Overview](#Xf75c500c02d5a4e676a74aae521aad5bca01f05)
   * [1.8 Project Significance](#X3b57af56eb7dc2fc4b7b07aceff888e71c3dd61)
2. [System Architecture and Design](#section-2-system-architecture-and-design)
   * [2.1 Introduction](#X71579871a0dd0939e4cd6f75756a6d3c68eb32f)
   * [2.2 Overall Architecture](#Xa4b32a86b2009daec3e709dc407dfb0888f69af)
   * [2.3 Backend API (MediPulse Core)](#X088a3bc2cd0a3ec960ece4b12e69c5e5f1b913e)
   * [2.4 Frontend (Web Application)](#X0e75f7601a3b4074acb2384ded22943971f47df)
   * [2.5 External Services](#Xd55705dd402f5464960c75a3af75d85ba2249e9)
   * [2.6 Data Flow and Sequence](#X484907708a24bdbc6d11ec99cbcd7f72fbf22ee)
   * [2.7 Design Principles and Considerations](#X64a5634bff90a3f0e25c741e4efb781412f0358)
   * [2.8 Anticipated Android Integration](#X2fe30f204286c4800ebb86fd5442a0b04635d5f)
3. [Implementation Details](#section-3-implementation-details)
   * [3.1 Introduction](#X62a86e61755f82f330fe6fe2c436cd5f645a3df)
   * [3.2 Backend Implementation (medipulseGROQ.py)](#Xf1f4a86c04cac330261f93c5fd10c16c930a781)
   * [3.3 Frontend Implementation](#Xfe61bb2e82b422f32477e181a2101df031c86cd)
   * [3.4 Data Structures](#X4b27492f93ba278f666fd8e781f1bc7da92dcf5)
   * [3.5 Error Handling](#Xfd90035918d1729502f5290ce94946730ff9d36)
4. [Features and Functionality](#section-4-features-and-functionality)
   * [4.1 Introduction](#X3d12c86abcf5b4dbb850c0bb314c2783e782c05)
   * [4.2 Core Features](#Xd1d91ac18a9535dbe7864cb1363b2009a45c8ea)
   * [4.3 User Workflow](#X608bd8bd04b5631298e7cb507c739eba89bce8c)
   * [4.4 Limitations (Recap)](#Xc65893339e82c38a04b85a1955cc08c16a75223)
5. [Frontend Integration (Website and Android)](#X781c595dc31cc92d057276e72ca706d8156f14e)
   * [5.1 Introduction](#X2044758a0bc63bb0dcd425e1bab7ef72e381a6e)
   * [5.2 Web Frontend Integration](#Xc2905b9d664f07da481ac54af6b219c30f3d7a5)
   * [5.3 Android Application Integration (Anticipated)](#X87700dae66407c0f4f26c44eedc19b4e9da12ef)
   * [5.4 Unified API Interface](#X07ae6e25f0f342d04e362ffa4859b4a148c4bd3)
6. [Appendices](#section-6-appendices)
   * [6.1 Sequence Diagram (PlantUML)](#X80cded7a0bc64606dd9a49a28a5afdf0518f34b)

## MediPulse Project Documentation

### Section 1: Project Overview

#### 1.1 Introduction

MediPulse represents a significant endeavor within the domain of digital health, conceived as a capstone project to explore the application of advanced Artificial Intelligence (AI) in providing accessible, preliminary health information. In an era where individuals increasingly turn to online resources for health-related queries, MediPulse aims to offer a structured, responsible, and informative initial point of contact. The system is designed as an AI-powered medical assistant capable of engaging users in a conversational manner, gathering information about their symptoms and relevant context (such as age, smoking habits, and pre-existing conditions), and subsequently providing general health guidance based on established medical knowledge patterns.

The core motivation behind MediPulse stems from the need to bridge the gap between the public’s demand for readily available health information and the critical importance of seeking professional medical advice for diagnosis and treatment. While numerous online resources exist, they often lack personalization, context-awareness, or the necessary safeguards against misinterpretation. MediPulse differentiates itself by employing a sophisticated Large Language Model (LLM), specifically Groq’s Llama 3 70B model, fine-tuned with a carefully crafted medical protocol (MediPulse Protocol v3.5). This protocol enforces strict operational boundaries, prioritizing user safety, emphasizing the preliminary nature of the information provided, and consistently directing users towards qualified healthcare professionals for definitive medical assessment.

The project encompasses the development of a robust backend API, built using the Flask framework in Python, which handles user interactions, session management, language processing (supporting both English and Arabic), and communication with the Groq LLM. Furthermore, the system architecture anticipates user access through multiple frontend interfaces: a dedicated website and an Android mobile application. These interfaces are designed to provide a seamless and intuitive user experience for interacting with the MediPulse assistant.

This document serves as comprehensive technical documentation for the MediPulse project, primarily intended for project supervisors and potentially future developers. It details the project’s objectives, architectural design, implementation specifics, features, and the integration of its various components. The subsequent sections will delve into the technical intricacies, design choices, and operational flow of the MediPulse system.

#### 1.2 Problem Statement

The digital age has democratized access to information, including health-related knowledge. However, this accessibility presents challenges. Individuals experiencing health concerns often resort to searching symptoms online, leading them to a vast, unregulated sea of information that can range from helpful to dangerously inaccurate. This self-diagnosis attempt, often fueled by anxiety, can lead to several negative outcomes:

1. **Misinformation and Anxiety:** Unverified sources or poorly interpreted information can cause unnecessary panic or, conversely, a false sense of security, delaying necessary medical attention.
2. **Lack of Contextual Understanding:** Generic online information rarely accounts for individual factors like age, medical history, lifestyle, or the specific nuances of symptoms, which are crucial for any meaningful health assessment.
3. **Overburdening Healthcare Systems:** While seeking professional help is vital, preliminary self-triage based on unreliable information can sometimes lead to unnecessary visits for minor issues or delayed visits for serious ones.
4. **Accessibility Barriers:** Not everyone has immediate access to healthcare professionals due to cost, location, or time constraints. While online tools cannot replace doctors, a reliable first point of information can be valuable.

MediPulse addresses these challenges by providing a controlled, AI-driven environment for preliminary health inquiries. It aims to offer users a more structured and responsible way to understand potential implications of their symptoms *before* or *while* seeking professional consultation, without claiming to diagnose or treat.

#### 1.3 Project Goals and Objectives

The primary goal of the MediPulse project is to develop and deploy an AI-powered medical assistant that provides safe, reliable, and preliminary health information to users based on their reported symptoms and context.

Specific objectives include:

1. **Develop a Secure and Scalable Backend API:** Create a robust backend service using Flask that manages user sessions, processes requests, interacts with the LLM, and enforces the defined medical protocol.
2. **Implement an Advanced AI Interaction Model:** Utilize the Groq Llama 3 70B LLM, guided by a custom-designed prompt (MediPulse Protocol v3.5), to handle conversations, understand user input, and generate informative yet cautious responses.
3. **Prioritize User Safety:** Embed non-negotiable safety protocols within the AI’s operational logic, including immediate identification of potential emergencies, strict avoidance of diagnoses and prescriptions, and clear disclaimers about the information’s nature.
4. **Support Multilingual Interaction:** Enable the system to communicate effectively in both English and Arabic, catering to a broader user base.
5. **Gather Essential Context:** Implement a structured initial questioning phase to collect relevant user information (age, smoking status, existing conditions) to provide more contextually relevant guidance.
6. **Develop User-Friendly Frontend Interfaces:** Design and implement (or facilitate the implementation of) intuitive interfaces for web and Android platforms, allowing easy interaction with the backend API.
7. **Ensure Responsible Information Delivery:** Structure AI responses to include potential condition categories (without diagnosing), safe OTC suggestions (with disclaimers, used sparingly), guidance on when to seek professional help, and consistent final disclaimers.
8. **Maintain User Privacy:** Limit data collection to essential information and manage sessions securely.
9. **Produce Comprehensive Documentation:** Create detailed technical documentation suitable for project evaluation and future development.

#### 1.4 Scope and Limitations

It is crucial to define the scope and limitations of the MediPulse system to manage expectations and ensure responsible usage.

**In Scope:**

* Providing preliminary, general health information based on user-reported symptoms and basic context (age, smoker status, existing conditions).
* Engaging in conversational interaction in English and Arabic.
* Identifying and flagging potentially serious symptoms requiring immediate emergency care.
* Suggesting general categories of conditions potentially related to symptoms.
* Offering guidance on generally safe Over-The-Counter (OTC) options for mild, common symptoms, with appropriate disclaimers and dosage information.
* Advising users on when it might be appropriate to seek professional medical attention based on symptom severity and duration.
* Operating via a backend API accessible by designated frontend applications (website, Android app).
* Maintaining conversation history within a limited session timeframe for contextual understanding.

**Out of Scope (Limitations):**

* **Diagnosis:** MediPulse **does not** provide medical diagnoses. It is not a substitute for a qualified healthcare professional.
* **Treatment Plans:** The system **does not** create or recommend specific medical treatment plans.
* **Prescriptions:** MediPulse **will never** suggest or recommend prescription medications.
* **Medical Emergencies:** While it attempts to identify potential emergencies and advise seeking immediate care, it **cannot** manage or treat medical emergencies.
* **Replacing Healthcare Professionals:** The system is designed as an informational tool and **does not** replace consultation, examination, or diagnosis by a doctor, nurse, or other qualified provider.
* **Handling Complex Cases:** The system’s knowledge is based on general patterns and may not be suitable for complex, chronic, or rare conditions.
* **Mental Health Crises:** MediPulse is not equipped to handle mental health crises. Users experiencing such crises should contact appropriate emergency services or mental health professionals.
* **Veterinary Advice:** The system is designed for human health information only.
* **Guaranteed Accuracy:** While based on the LLM’s training and the defined protocol, the information provided is general and cannot be guaranteed to be accurate or applicable in every individual case.

#### 1.5 Target Audience (Application Users)

The intended users of the MediPulse application (website and Android app) are general members of the public who are seeking preliminary information about non-emergency health symptoms they may be experiencing. This includes individuals who:

* Are unsure about the potential significance of their symptoms.
* Want to understand possible causes before consulting a doctor.
* Need guidance on basic self-care measures or appropriate OTC options for mild conditions.
* Prefer to interact in either English or Arabic.

Users are expected to understand that MediPulse is an informational tool only and that professional medical advice is necessary for diagnosis and treatment.

#### 1.6 System Components Overview

The MediPulse system comprises three main components:

1. **Backend API (MediPulse Core):** This is the central processing unit of the system, developed using Python and the Flask web framework. It hosts the core logic, including:
   * API endpoints for communication with frontends (/api/start, /api/chat).
   * Session management (using Flask-Session with filesystem storage).
   * Interaction logic with the Groq Llama 3 LLM.
   * Implementation of the MediPulse Medical Protocol v3.5.
   * Language handling (English/Arabic).
   * Structured question flow and context gathering.
2. **Web Frontend:** A web-based interface (details to be elaborated in later sections) allowing users to access MediPulse via a standard web browser. This component interacts with the Backend API to send user input and display responses.
3. **Android Mobile Application:** A native Android application (details to be elaborated in later sections) providing a mobile-optimized experience for interacting with the MediPulse assistant. This app also communicates directly with the Backend API.

#### 1.7 Technology Stack Overview

* **Backend:** Python 3, Flask, Flask-CORS, Flask-Session
* **AI/LLM:** Groq API, Llama 3 70B Model, Langchain (for schema and memory potentially, though memory seems custom-managed in session)
* **Environment:** Dotenv (for environment variable management)
* **Frontend (Anticipated):** HTML, CSS, JavaScript (for Web), Kotlin/Java (for Android)
* **Deployment (Potential):** Standard web hosting/cloud platform for Flask API, Google Play Store for Android App.

#### 1.8 Project Significance

MediPulse holds significance in demonstrating a responsible approach to leveraging AI for public health information. By integrating strict safety protocols and clear limitations directly into the AI’s operational framework, it sets a precedent for developing trustworthy digital health tools. Its multilingual capability enhances accessibility, and its focus on preliminary guidance aims to empower users with information while reinforcing the irreplaceable value of professional medical consultation. For the development team, it serves as a valuable exercise in API design, AI integration, frontend-backend communication, and the ethical considerations inherent in health technology.

## MediPulse Project Documentation

### Section 2: System Architecture and Design

#### 2.1 Introduction

This section provides a detailed description of the architectural design of the MediPulse system. MediPulse is architected as a web-based application employing a client-server model, augmented by a powerful Large Language Model (LLM) for its core conversational AI capabilities. The architecture prioritizes modularity, safety, user experience, and scalability. It comprises three primary logical components: the Backend API (MediPulse Core), the Frontend Interfaces (Web and anticipated Android), and the External LLM Service (Groq API).

#### 2.2 Overall Architecture

The system follows a classic three-tier architecture:

1. **Presentation Tier (Frontend):** This tier is responsible for user interaction. It includes a web application built with HTML, CSS, and JavaScript, providing a chat interface within a web browser. An Android mobile application is also envisioned as part of this tier, offering a native mobile experience. Both frontend clients communicate with the backend via HTTP requests.
2. **Application Tier (Backend API):** This is the core logic layer, implemented as a Flask web application in Python. It handles incoming requests from the frontends, manages user sessions and conversation state, enforces the medical safety protocol, interacts with the LLM service, and sends responses back to the clients. It acts as the central orchestrator of the system.
3. **Data/Service Tier:** This tier primarily consists of the external Groq API service, which provides the Llama 3 70B LLM used for generating health-related information. Session data is managed within the Application Tier using server-side filesystem storage.

Communication between the Frontend and Backend occurs via a RESTful API exposed by the Flask application. The Backend communicates with the Groq API using HTTPS requests, sending prompts and receiving generated text.

graph LR  
 User -- Interacts --> FE(Frontend: Web/Android)  
 FE -- HTTP Requests (JSON) --> BE(Backend API: Flask)  
 BE -- Stores/Retrieves --> FS(Filesystem Session Cache)  
 BE -- API Call (Prompt) --> LLM(Groq API: Llama 3)  
 LLM -- API Response (Text) --> BE  
 BE -- HTTP Responses (JSON) --> FE  
 FE -- Displays Info --> User

*Figure 2.1: High-Level System Architecture Diagram*

#### 2.3 Backend API (MediPulse Core)

The backend is the heart of the MediPulse system, developed using the Flask microframework in Python. It exposes API endpoints, manages conversation logic, and integrates with the LLM.

**2.3.1 Technology Stack:**

* **Framework:** Flask (v2.x or later assumed based on modern practices)
* **Language:** Python 3
* **Key Libraries:**
  + Flask: Core web framework.
  + Flask-CORS: Handles Cross-Origin Resource Sharing, allowing the frontend (potentially served from a different domain/port during development) to interact with the API.
  + Flask-Session: Manages server-side sessions, configured to use the filesystem (SESSION\_TYPE = "filesystem") for storing session data persistently within a defined lifetime (PERMANENT\_SESSION\_LIFETIME = timedelta(hours=1)). Session data is stored in the ./session\_cache directory.
  + python-dotenv: Loads environment variables from a .env file (though .env itself is rightly excluded from the repository via .gitignore), used for sensitive information like API keys and secret keys.
  + langchain: Although ConversationBufferMemory is imported, the actual memory management appears to be custom-built using the Flask session (session["memory\_history"]). Langchain schemas (SystemMessage, HumanMessage) are used to structure the interaction with the LLM.
  + langchain-groq: Provides the ChatGroq class for seamless integration with the Groq API, specifically configured to use the llama3-70b-8192 model.
  + secrets: Used to generate a default Flask secret key if one isn’t provided via environment variables.
  + logging: Standard Python logging library configured for basic informational logging.

**2.3.2 Core Components:**

* **Flask Application (app):** The central Flask application instance is initialized, configured with static/template folder paths, a secret key (crucial for session security), and session parameters. CORS is enabled for all routes.
* **LLM Initialization (llm):** An instance of ChatGroq is created, configured with the specific model name (llama3-70b-8192), a low temperature (0.1) for more deterministic responses, and the Groq API key loaded from environment variables.
* **Medical Prompt Configuration:** This is a critical design element ensuring safe and responsible AI behavior.
  + MEDICAL\_PROMPT\_BASE: A detailed multi-part string defining the AI’s role, sole purpose (preliminary info, not diagnosis), core requirements (safety, emergency identification, no prescriptions, limited OTC guidance, no diagnostic decisions), response guidelines (clarification, conciseness, language, tone), assessment format, and boundaries (scope, off-topic deflection, privacy).
  + PROMPT\_COMPONENTS: A dictionary containing language-specific text (English and Arabic) for various parts of the prompt (instructions, emergency warnings, disclaimers, etc.).
  + get\_dynamic\_prompt(language): A function that dynamically constructs the final system prompt by injecting the appropriate language components from PROMPT\_COMPONENTS into the MEDICAL\_PROMPT\_BASE template based on the user’s selected language.
* **Structured Questions (questions):** A list of dictionaries defining the initial questions asked to the user (Language, Age, Smoker, Existing Conditions). Each dictionary specifies an id, English text (text), Arabic text (text\_ar), and input type (language, number, boolean). This structured approach ensures necessary context is gathered before assessment.
* **Session Management:** Flask-Session is used to maintain user state across requests. Key session variables include:
  + session["memory\_history"]: Stores the conversation history as a list of (input, output) tuples.
  + session["current\_index"]: Tracks the current question index during the initial questioning phase.
  + session["stage"]: Indicates the current phase of the conversation (‘questions’ or ‘assessment’).
  + session["language"]: Stores the user’s selected language (‘en’ or ‘ar’).
  + session["created\_at"]: Timestamp for session creation. The session data is stored on the server’s filesystem in the session\_cache directory, identified by a session ID cookie sent to the client.

**2.3.3 API Endpoints:**

The backend exposes the following HTTP endpoints:

* **GET /:**
  + **Purpose:** Serves the main HTML page for the web frontend.
  + **Implementation:** Uses Flask’s render\_template("index.html").
* **POST /api/start:**
  + **Purpose:** Initializes a new chat session.
  + **Implementation:** Clears any existing session data for the user, initializes session variables (memory\_history, current\_index=0, stage='questions', language='en', created\_at), retrieves the first question (language selection), and returns a JSON response containing the new session\_id and the first question details (text in both languages, type).
* **POST /api/chat:**
  + **Purpose:** Handles all subsequent user messages during the chat.
  + **Implementation:**
    1. Retrieves the user’s message and session state (stage, index, language) from the request and session.
    2. Performs session validation (checks if essential session variables exist).
    3. **Language Selection Handling:** If it’s the first question (index 0), determines and sets the session language based on user input, increments the index, and returns the next question in the selected language.
    4. **Question Stage Handling:** If in the ‘questions’ stage (and not the first question):
       - Retrieves the current question text in the correct language.
       - Performs input validation (specifically for boolean types, checking against English/Arabic ‘yes’/‘no’ variations).
       - Saves the question and the processed user answer to the session["memory\_history"].
       - Increments session["current\_index"].
       - If more questions remain, returns the next question.
       - If all questions are answered, transitions session["stage"] to ‘assessment’ and returns a prompt asking the user to describe their symptoms.
    5. **Assessment Stage Handling:** If in the ‘assessment’ stage:
       - Rebuilds the conversation history from session["memory\_history"] into a format suitable for the LLM (using ConversationBufferMemory temporarily, though the primary storage is the session).
       - Retrieves the dynamically generated system prompt for the selected language using get\_dynamic\_prompt().
       - Constructs the message list for the LLM: [SystemMessage(prompt), \*history, HumanMessage(user\_input)].
       - Invokes the llm (Groq API) with the messages.
       - Saves the user input and the LLM’s response to session["memory\_history"].
       - Returns the LLM’s content as the response.
    6. **Error Handling:** Includes try-except blocks to catch potential errors during processing or LLM interaction, returning JSON error messages and appropriate HTTP status codes (e.g., 400 for bad input, 401 for invalid session, 500 for internal errors).

**2.3.4 Deployment Configuration:**

* **Procfile:** Contains web: gunicorn medipulseGROQ:app. This indicates the application is intended for deployment using the Gunicorn WSGI server, a common practice for production Flask applications on platforms like Heroku or Railway.
* **requirements.txt:** Lists all necessary Python dependencies, crucial for recreating the environment during deployment.

#### 2.4 Frontend (Web Application)

The web frontend provides the user interface for interacting with the MediPulse assistant.

**2.4.1 Technology Stack:**

* **Structure:** HTML (templates/index.html)
* **Styling:** CSS (static/style.css - content not analyzed but assumed to style the chat interface)
* **Logic:** JavaScript (static/script.js)

**2.4.2 Structure and Functionality (index.html, script.js):**

* **UI Elements:** The HTML defines a simple chat interface containing:
  + A div (chat-box) to display messages.
  + An input field (user-input) for typing messages.
  + A div (bool-options) containing ‘Yes’ and ‘No’ buttons, initially hidden.
  + A ‘Send’ button (send-button).
* **JavaScript Logic (script.js):**
  + **Initialization:** On DOMContentLoaded, it initiates the chat by calling startSession().
  + **startSession():** Sends a POST request to /api/start, stores the returned sessionId, and calls handleQuestion() with the first question.
  + **sendMessage():** Triggered by the Send button or Enter key. It reads the user input, displays it in the chatbox, clears the input field, and sends a POST request to /api/chat with the sessionId and message.
  + **Response Handling:** Processes the JSON response from /api/chat. If it contains an error, it displays the error. If it indicates the ‘assessment’ stage, it updates the input placeholder and displays the assessment prompt. If it contains a next\_question, it calls handleQuestion(). If it contains an LLM response, it displays the response.
  + **handleBoolean(answer):** Called when Yes/No buttons are clicked. Sets the input field value to the answer (‘Yes’ or ‘No’) and calls sendMessage().
  + **handleQuestion(question):** Displays the question text received from the backend. It dynamically shows or hides the boolean buttons and the text input field based on the question.type.
  + **addMessage(sender, text):** A utility function to create and append new message elements (user or bot) to the chat-box and scroll to the bottom.

#### 2.5 External Services

* **Groq API:** The system relies heavily on the Groq cloud platform to access the Llama 3 70B LLM. This external service performs the natural language understanding and generation based on the prompts provided by the backend. Communication is secured via an API key managed through environment variables.

#### 2.6 Data Flow and Sequence

The interaction sequence, as detailed in the sequence diagram (medipulse\_sequence.puml), typically follows these steps:

1. User accesses the web frontend.
2. Frontend calls /api/start on the backend.
3. Backend creates a session, returns session ID and the first question (Language).
4. Frontend displays the language question.
5. User provides input (e.g., selects language).
6. Frontend sends input to /api/chat.
7. Backend processes input (sets language), determines the next question (Age), and returns it.
8. This question-answer cycle repeats for Age, Smoker, Conditions.
9. After the last question, the backend transitions the stage to ‘assessment’ and prompts for symptoms.
10. Frontend displays the symptom prompt.
11. User describes symptoms.
12. Frontend sends symptoms to /api/chat.
13. Backend prepares the full prompt (System Prompt + History + Symptoms) and sends it to the Groq LLM.
14. Groq LLM processes the prompt and returns a generated response.
15. Backend receives the LLM response, saves the interaction to session history, and sends the response to the frontend.
16. Frontend displays the AI’s response to the user.

#### 2.7 Design Principles and Considerations

* **Modularity:** Clear separation between frontend presentation logic, backend application logic, and the external AI service.
* **Safety First:** The design places paramount importance on user safety through the detailed, rule-based MEDICAL\_PROMPT\_BASE that strictly governs the AI’s behavior, preventing diagnosis, prescriptions, and ensuring appropriate disclaimers and emergency referrals.
* **Stateless API (with Server-Side State):** While REST APIs aim for statelessness, this application maintains conversation state using server-side sessions (Flask-Session). Each request from a client is implicitly associated with its session via a cookie, allowing the backend to retrieve context.
* **User Experience:** The structured questioning provides context, multilingual support broadens accessibility, and the chat interface offers a familiar interaction paradigm.
* **Scalability:** Using Flask and Gunicorn allows for horizontal scaling by running multiple instances of the backend application behind a load balancer, although the filesystem session storage might become a bottleneck in a highly scaled, multi-server environment (alternatives like Redis could be considered if needed).
* **Maintainability:** Using a framework like Flask, clear separation of concerns, and configuration via environment variables contribute to maintainability.
* **Security:** Use of environment variables for secrets (API key, Flask secret key) and Flask-Session for managing sessions are basic security measures. CORS is configured, likely for development flexibility.

#### 2.8 Anticipated Android Integration

While the code for the Android application is not present in this repository, the architecture is designed to support it. The Android app would function similarly to the web frontend:

1. It would make HTTP POST requests to the same /api/start and /api/chat endpoints.
2. It would need to handle session management, likely by storing and sending the session cookie received from the backend with subsequent requests.
3. It would parse the JSON responses from the API to display questions, prompts, and AI responses within its native UI components.
4. It would implement native UI elements for text input and potentially dedicated buttons for boolean questions, mirroring the web frontend’s functionality.

## MediPulse Project Documentation

### Section 3: Implementation Details

#### 3.1 Introduction

This section delves into the specific implementation details of the MediPulse system, building upon the architectural overview provided in the previous section. It examines the key code segments, logic flows, data handling, and configurations within the backend API and the web frontend, based on the analysis of the medipulseGROQ.py, templates/index.html, and static/script.js files from the provided GitHub repository.

#### 3.2 Backend Implementation (medipulseGROQ.py)

The backend is implemented as a single Python file using the Flask framework. This monolithic structure is suitable for a project of this scale but could be refactored into modules (e.g., routes, services, models) for larger applications.

**3.2.1 Initialization and Configuration:**

* **Imports:** The script begins by importing necessary libraries: os, re, secrets, logging, datetime, Flask components (Flask, request, jsonify, render\_template, session), Flask-CORS, Flask-Session, dotenv, Langchain components (SystemMessage, HumanMessage, AIMessage, ConversationBufferMemory), and ChatGroq.
* **Environment Variables:** load\_dotenv() is called to load configuration from a .env file. This is crucial for managing sensitive data like FLASK\_SECRET\_KEY and GROQ\_API\_KEY without hardcoding them.
* **Flask App Instantiation:** app = Flask(\_\_name\_\_, static\_folder="static", template\_folder="templates") creates the Flask application instance, correctly pointing to the directories containing frontend assets.
* **CORS:** CORS(app) enables cross-origin requests for all routes, simplifying frontend development and deployment scenarios.
* **Session Configuration:** Flask-Session is configured:
  + SECRET\_KEY: Essential for signing session cookies. It attempts to load from FLASK\_SECRET\_KEY environment variable, falling back to a securely generated random hex token using secrets.token\_hex(16) if the variable is not set.
  + SESSION\_TYPE = "filesystem": Specifies that session data should be stored on the server’s filesystem.
  + SESSION\_FILE\_DIR = "./session\_cache": Sets the directory for storing session files.
  + SESSION\_PERMANENT = True and PERMANENT\_SESSION\_LIFETIME = timedelta(hours=1): Configures sessions to be permanent (relative to browser session) but expire after 1 hour of inactivity.
  + Session(app) initializes the Flask-Session extension with the app.
* **Logging:** logging.basicConfig(...) sets up basic logging to capture informational messages, warnings, and errors, including timestamps.
* **Groq LLM Initialization:** llm = ChatGroq(...) creates the client instance to interact with the Groq API, specifying the llama3-70b-8192 model, a low temperature (0.1) for focused responses, and the API key from GROQ\_API\_KEY.

**3.2.2 Prompt Engineering Implementation:**

* **MEDICAL\_PROMPT\_BASE:** This multi-line string is the core of the AI’s persona and safety constraints. It uses f-string-like placeholders (e.g., {language\_instruction}) to allow for dynamic language injection.
* **PROMPT\_COMPONENTS:** A dictionary mapping language codes (‘en’, ‘ar’) to sub-dictionaries. Each sub-dictionary contains the actual text snippets corresponding to the placeholders in MEDICAL\_PROMPT\_BASE for that language.
* **get\_dynamic\_prompt(language) Function:**
  + Takes a language code (‘en’ or ‘ar’) as input.
  + Defaults to ‘en’ if the provided language is unsupported.
  + Retrieves the corresponding component dictionary from PROMPT\_COMPONENTS.
  + Uses the .format() string method to insert the language-specific text components into the MEDICAL\_PROMPT\_BASE string.
  + Includes error handling (try-except KeyError) to fall back to the English prompt if a specific language key is missing, logging an error in the process.
  + Returns the fully constructed, language-specific system prompt string. This dynamic approach is efficient for supporting multiple languages without duplicating the entire base prompt structure.

**3.2.3 Structured Questioning Implementation:**

* **questions List:** A list defines the sequence and content of initial questions. Each element is a dictionary with id, text (English), text\_ar (Arabic), and type (‘language’, ‘number’, ‘boolean’). The order in this list dictates the conversation flow during the ‘questions’ stage.

**3.2.4 API Endpoint Implementation:**

* **@app.route("/") def home():**
  + Maps to the root URL (/).
  + Uses render\_template("index.html") to serve the main web page.
* **@app.route("/api/start", methods=["POST"]) def start\_chat():**
  + Mapped to /api/start, accepting only POST requests.
  + session.clear(): Ensures a fresh start by removing any previous session data for the client.
  + Initializes session variables: memory\_history (empty list), current\_index (0), stage (‘questions’), language (‘en’ default), created\_at (current timestamp).
  + Retrieves the first question (questions[0]).
  + Constructs the question text including both English and Arabic for the initial language selection: f"{first\_question['text']} / {first\_question['text\_ar']}".
  + Logs the start of a new session.
  + Returns a JSON response containing the session.sid (unique session identifier managed by Flask-Session) and the first question’s details (id, combined text, type).
* **@app.route("/api/chat", methods=["POST"]) def handle\_chat():**
  + Mapped to /api/chat, accepting only POST requests.
  + data = request.get\_json(): Retrieves the JSON payload sent by the client (expected to contain message).
  + **Session Validation:** Checks if required keys (memory\_history, current\_index, stage) exist in the session. If not, it logs a warning and returns a 401 Unauthorized error with a bilingual message prompting the user to restart.
  + Retrieves user\_input, current\_index, stage, and language from the request and session.
  + **Conditional Logic based on stage:**
    - **Language Selection (stage == "questions" and current\_index == 0):**
      * Checks user\_input for keywords indicating Arabic (‘arab’, ‘عربي’).
      * Sets session["language"] to ‘ar’ or ‘en’.
      * Logs the language selection.
      * Increments session["current\_index"].
      * Retrieves the next question’s data (questions[current\_index]).
      * Gets the question text in the selected language using next\_q\_data.get(f"text\_{language}", next\_q\_data["text"]) (provides fallback to English text).
      * Returns a JSON response with the next\_question details and calculated progress.
    - **Question Stage (stage == "questions" and current\_index > 0):**
      * Retrieves current question data and text.
      * **Boolean Input Validation (current\_q\_data["type"] == "boolean"):**
        + Defines lists of valid ‘yes’ and ‘no’ inputs in both languages.
        + Normalizes the user\_input.lower() to ‘Yes’/‘No’ (English) or ‘نعم’/‘لا’ (Arabic) if it matches any valid option.
        + If no match, returns a 400 Bad Request error with a language-specific prompt to answer Yes/No.
        + Stores the normalized answer in user\_input\_processed.
      * For non-boolean questions, user\_input\_processed is simply the original user\_input.
      * **Memory Update:** Temporarily creates a ConversationBufferMemory (though not strictly necessary as history is directly managed in the session), saves the current question (current\_q\_text) and the processed answer (user\_input\_processed) using memory.save\_context. **Crucially, it also appends the (current\_q\_text, user\_input\_processed) tuple to session["memory\_history"], which is the persistent storage.**
      * Increments session["current\_index"].
      * **Stage Transition Check:** If current\_index >= len(questions), it means all initial questions are answered. Sets session["stage"] = "assessment", defines the symptom prompt in the correct language, logs the stage change, and returns a JSON response with the message (symptom prompt) and stage (‘assessment’).
      * **Next Question:** If questions remain, retrieves the next question’s data and text, and returns it in a JSON response along with the progress.
      * Includes a try...except block for general error handling during this stage, returning a 500 Internal Server Error if issues occur.
    - **Assessment Stage (stage == "assessment"):**
      * **Memory Reconstruction:** Creates a ConversationBufferMemory and populates it by iterating through the session["memory\_history"] list and using memory.save\_context for each past input/output pair. This reconstructs the conversation context for the Langchain LLM call.
      * system\_prompt = get\_dynamic\_prompt(language): Gets the appropriate system prompt.
      * **Message Formatting:** Creates the messages list required by the Langchain LLM:
        + Starts with SystemMessage(content=system\_prompt).
        + Unpacks the history from the reconstructed memory object: \*memory.load\_memory\_variables({})["history"].
        + Adds the current user input: HumanMessage(content=user\_input).
      * **LLM Invocation:** Calls response = llm.invoke(messages).content. This sends the structured messages to the Groq API and retrieves the content of the AI’s response.
      * Logs the invocation and reception of the LLM response.
      * **Memory Update:** Saves the current user\_input and the received LLM response to the temporary memory object and, importantly, appends (user\_input, response) to session["memory\_history"].
      * Returns a JSON response containing the LLM’s response.
      * Includes a try...except block for error handling during LLM interaction, returning a 500 error if needed.

**3.2.5 Main Execution Block:**

* The standard if \_\_name\_\_ == "\_\_main\_\_": block, which typically contains app.run(), is commented out. This is correct practice for production deployment, as the application will be run by a WSGI server like Gunicorn (as specified in the Procfile) rather than Flask’s built-in development server.

#### 3.3 Frontend Implementation

The frontend consists of a single HTML file, a CSS file (styling assumed), and a JavaScript file for dynamic behavior.

**3.3.1 HTML Structure (templates/index.html):**

* Standard HTML5 boilerplate.
* Links the style.css stylesheet.
* Contains a main div.chat-container holding:
  + div.chat-box: Where messages will be displayed.
  + div.input-area: Contains the input elements.
    - input#user-input: Text field for user messages.
    - div#bool-options: Container for Yes/No buttons (initially hidden via CSS likely).
      * button#yes-btn, button#no-btn: Buttons for boolean answers.
    - button#send-button: Button to submit messages.
* Includes the script.js file at the end of the body.

**3.3.2 JavaScript Logic (static/script.js):**

* **DOM Ready:** Wraps all code in a DOMContentLoaded event listener to ensure HTML elements are available before manipulation.
* **Element References:** Gets references to key DOM elements (chat box, input field, buttons).
* **sessionId Variable:** Declared to store the session ID received from the backend.
* **Event Listeners:**
  + sendButton: Calls sendMessage on click.
  + userInput: Calls sendMessage on ‘Enter’ key press.
  + yesBtn, noBtn: Call handleBoolean with ‘Yes’ or ‘No’ respectively on click.
* **startSession() Function (async):**
  + Uses fetch to make a POST request to /api/start.
  + Parses the JSON response.
  + Stores data.session\_id in the sessionId variable.
  + Calls handleQuestion with the initial question data.
  + Includes basic error handling to display errors in the chat.
* **sendMessage() Function (async):**
  + Gets the trimmed message from userInput.
  + If the message is empty, it returns.
  + Calls addMessage('user', message) to display the user’s message.
  + Clears the userInput field.
  + Uses fetch to make a POST request to /api/chat:
    - Sets Content-Type header to application/json.
    - Sends the sessionId and message in the JSON body.
  + Parses the JSON response.
  + **Response Processing:** Checks the structure of the data object:
    - If data.error, throws an error (caught by the catch block).
    - If data.stage === "assessment", updates the input placeholder, hides boolean buttons, shows the text input, and displays the assessment prompt message.
    - If data.next\_question, calls handleQuestion with the question data.
    - If data.response, calls addMessage('bot', data.response) to display the AI’s answer.
  + Includes error handling to display errors in the chat.
* **handleBoolean(answer) Function:**
  + Sets the userInput value to the provided answer (‘Yes’ or ‘No’).
  + Calls sendMessage() to submit the answer.
* **handleQuestion(question) Function:**
  + Calls addMessage('bot', question.text) to display the question.
  + Uses boolOptions.classList.toggle('visible', question.type === 'boolean') to show/hide the Yes/No buttons based on the question type (assumes a ‘visible’ CSS class controls display).
  + Sets the userInput display style to ‘none’ for boolean questions and ‘block’ otherwise.
  + Sets focus to the userInput field.
* **addMessage(sender, text) Function:**
  + Creates a new div element.
  + Assigns CSS classes (${sender}-message, message) for styling.
  + Sets the textContent of the div.
  + Appends the div to the chatBox.
  + Scrolls the chatBox to the bottom to show the latest message.
* **Initial Call:** startSession() is called at the end to begin the interaction when the page loads.

#### 3.4 Data Structures

* **Session Data (session object):** A dictionary-like object managed by Flask-Session, stored server-side. Key fields: memory\_history (list of tuples), current\_index (int), stage (str), language (str), created\_at (str).
* **Question Definition (questions list):** A list of dictionaries, each defining a question’s id, text, text\_ar, and type.
* **API Payloads (JSON):** Communication between frontend and backend uses JSON. Requests to /api/chat contain {"session\_id": ..., "message": ...}. Responses vary but include keys like session\_id, question, next\_question, progress, stage, message, response, error.
* **Langchain Messages:** SystemMessage, HumanMessage, AIMessage objects are used internally in the backend to structure the conversation for the ChatGroq LLM.

#### 3.5 Error Handling

* **Backend:** Uses try...except blocks in the /api/chat endpoint to catch errors during input processing, stage transitions, and LLM interaction. Returns JSON responses with an error key and appropriate HTTP status codes (400, 401, 500). Logging is used to record error details.
* **Frontend:** Uses try...catch blocks around fetch calls in startSession and sendMessage. If an error occurs during the fetch or if the backend response contains an error key, it displays the error message in the chatbox using addMessage('bot', ...). Basic, but functional for informing the user.

## MediPulse Project Documentation

### Section 4: Features and Functionality

#### 4.1 Introduction

This section outlines the core features and functionalities provided by the MediPulse system. It describes what the system does from both the end-user perspective interacting with the frontend and the perspective of the system’s internal operations and API capabilities.

#### 4.2 Core Features

**4.2.1 Conversational AI Medical Assistant:**

The primary feature of MediPulse is its ability to act as a conversational AI assistant focused on preliminary health information. Users can interact with the system through a chat interface, describing their symptoms in natural language. \* **Functionality:** The system uses the powerful Groq Llama 3 70B Large Language Model (LLM) to understand user input and generate relevant, informative responses based on a predefined medical protocol. \* **Implementation:** Handled by the /api/chat endpoint in the backend, which formats prompts, interacts with the ChatGroq instance, and processes responses.

**4.2.2 Multilingual Support (English & Arabic):**

MediPulse is designed to interact with users in either English or Arabic, enhancing accessibility. \* **Functionality:** The system explicitly asks the user for their preferred language at the beginning of the conversation. All subsequent interactions, including questions, prompts, AI responses, and error messages, are provided in the selected language. \* **Implementation:** The /api/start endpoint presents the initial language choice. The /api/chat endpoint detects the selection, stores it in session["language"], and uses the get\_dynamic\_prompt function and PROMPT\_COMPONENTS dictionary to retrieve and format all text in the chosen language.

**4.2.3 Structured Context Gathering:**

Before providing any assessment, MediPulse gathers essential contextual information from the user through a series of structured questions. \* **Functionality:** The system sequentially asks the user for their age, smoking status, and any existing medical conditions after the language selection. This information provides context for the LLM during the assessment phase. \* **Implementation:** Defined by the questions list in medipulseGROQ.py. The /api/chat endpoint manages the flow through the session["current\_index"] and session["stage"] == 'questions'. The frontend (script.js) handles displaying questions and managing input types (text, boolean buttons).

**4.2.4 Symptom Assessment and Preliminary Information:**

After gathering context, the system prompts the user to describe their symptoms and provides general, preliminary information based on the input. \* **Functionality:** Based on the user’s symptoms and context, the AI provides potential general categories of conditions, suggests *limited* and *safe* Over-The-Counter (OTC) options for mild symptoms (with specific dosage examples and disclaimers), and offers guidance on when to seek professional medical attention. It explicitly avoids definitive diagnoses. \* **Implementation:** Handled during the session["stage"] == 'assessment'. The backend constructs a detailed prompt including the MEDICAL\_PROMPT\_BASE, conversation history, and user symptoms, sends it to the LLM via the Groq API, and returns the generated response. The prompt engineering ensures the response adheres to the defined guidelines.

**4.2.5 Robust Safety Protocols:**

Safety is a paramount design principle, implemented through strict rules embedded in the AI’s operational prompt. \* **Functionality:** \* **Emergency Identification:** The AI is instructed to immediately output a specific emergency warning and advise seeking urgent care if symptoms suggest a potential emergency. \* **No Diagnosis/Prescriptions:** The AI is explicitly forbidden from providing diagnoses or suggesting prescription medications. \* **Limited OTC Guidance:** Strict rules govern when and how OTC suggestions can be made (mild symptoms, specific dosage, mandatory disclaimer). \* **Referral for Decisions:** The AI deflects questions about the necessity of diagnostic tests or specific treatments, referring the user to a qualified doctor. \* **Disclaimers:** Mandatory starting and ending disclaimers reinforce the informational, non-diagnostic nature of the interaction in every assessment response. \* **Scope Limitation:** The AI is instructed to deflect off-topic conversations. \* **Implementation:** Primarily enforced through the detailed instructions and constraints within the MEDICAL\_PROMPT\_BASE string, which is used as the SystemMessage for the LLM in the assessment stage.

**4.2.6 Session Management:**

The system maintains the context of the conversation for each user within a defined timeframe. \* **Functionality:** Each user interaction is part of a session, allowing the system to remember the selected language, answers to initial questions, and the ongoing conversation history. Sessions expire after a period of inactivity (1 hour). \* **Implementation:** Utilizes the Flask-Session extension configured for filesystem storage. The backend manages session variables (language, current\_index, stage, memory\_history). The frontend receives a session ID cookie and implicitly uses it for subsequent requests.

**4.2.7 Web Interface Functionality:**

A simple web-based chat interface allows users to interact with MediPulse. \* **Functionality:** Provides a real-time chat experience where users can type messages, receive questions and responses, and answer boolean questions using dedicated buttons. Displays user and bot messages distinctly. \* **Implementation:** templates/index.html provides the structure, static/style.css (assumed) provides styling, and static/script.js handles dynamic behavior: initiating the session, sending messages via fetch to the backend API, processing responses, displaying messages, and dynamically adjusting the input controls based on question type.

**4.2.8 RESTful API Endpoints:**

The backend exposes API endpoints that enable communication with frontend clients (web and potentially mobile). \* **Functionality:** Provides programmatic access to start chats (/api/start) and handle messages (/api/chat). This allows different frontend applications to integrate with the core MediPulse logic. \* **Implementation:** Defined using @app.route decorators in medipulseGROQ.py. Endpoints handle POST requests, process JSON data, interact with the session and LLM, and return JSON responses.

#### 4.3 User Workflow

From an end-user perspective interacting with the web application, the typical workflow is:

1. **Access:** User navigates to the MediPulse web application URL.
2. **Language Selection:** The system presents the first question, asking for language preference (English/Arabic).
3. **Context Questions:** The user answers a sequence of questions: Age, Smoker status (Yes/No buttons), Existing Conditions (Yes/No buttons).
4. **Symptom Input:** The system prompts the user to describe their symptoms.
5. **Assessment:** The user types their symptoms and sends the message.
6. **Response:** The system displays the AI-generated response, containing preliminary information, potential categories, possible OTC suggestions (if applicable and safe), advice on seeking professional help, and mandatory disclaimers.
7. **Further Interaction (Optional):** The user can continue the conversation by asking follow-up questions or providing more details, and the AI will respond based on the ongoing context and its programmed protocol.
8. **Session Timeout:** If the user is inactive for an hour, the session expires, and they would need to start a new chat.

#### 4.4 Limitations (Recap)

It is essential to reiterate the functional limitations:

* Does not provide diagnoses.
* Does not offer treatment plans or prescriptions.
* Cannot replace a consultation with a healthcare professional.
* Information is general and preliminary.
* Not suitable for managing emergencies or complex/chronic conditions.

## MediPulse Project Documentation

### Section 5: Frontend Integration (Website and Android)

#### 5.1 Introduction

This section details how the frontend components of the MediPulse system integrate with the backend API. It covers the existing web application interface and outlines the anticipated integration strategy for the Android mobile application. Both frontends rely on the same set of RESTful API endpoints provided by the Flask backend, ensuring consistent functionality across platforms.

#### 5.2 Web Frontend Integration

The primary user interface currently implemented is a web application, served directly by the Flask backend.

**5.2.1 Technologies:**

* **HTML:** templates/index.html defines the structure of the chat interface.
* **CSS:** static/style.css provides the visual styling (specific styles not analyzed).
* **JavaScript:** static/script.js handles all client-side logic and communication with the backend.

**5.2.2 API Communication (script.js):**

The JavaScript code orchestrates the interaction between the user’s browser and the backend API:

1. **Session Initialization:** Upon page load (DOMContentLoaded), the startSession() function sends an asynchronous POST request to /api/start using the fetch API. It retrieves the session\_id (though the script doesn’t explicitly use the ID itself, relying on the browser’s cookie management) and the first question.
2. **Sending Messages:** When the user types a message and clicks ‘Send’ or presses Enter, or clicks a ‘Yes’/‘No’ button, the sendMessage() function (or handleBoolean which calls sendMessage) is triggered. This function:
   * Retrieves the user’s input.
   * Displays the user’s message in the chatbox.
   * Sends an asynchronous POST request to /api/chat using fetch.
   * The request includes the Content-Type: application/json header and a JSON body containing the user’s message. The session\_id is not explicitly sent in the body, as Flask-Session relies on the session cookie automatically sent by the browser with each request.
3. **Receiving and Processing Responses:** The fetch call in sendMessage() waits for the response from /api/chat. The JavaScript code then parses the JSON response and acts accordingly:
   * **Errors:** If the response contains an error key, the error message is displayed in the chatbox.
   * **Next Question:** If the response contains next\_question, the handleQuestion() function is called to display the question text and adjust the input controls (showing/hiding boolean buttons and the text input field).
   * **Assessment Stage:** If the response indicates stage: "assessment", the input placeholder is updated, and the prompt to describe symptoms is displayed.
   * **AI Response:** If the response contains the AI’s response, it is displayed in the chatbox using addMessage('bot', ...).

**5.2.3 Session Handling:**

Session management is handled implicitly from the frontend’s perspective. The Flask backend, using Flask-Session, sets a session cookie upon the initial /api/start response. The browser automatically includes this cookie in all subsequent requests to the same domain, allowing the backend to identify the user’s session and maintain conversation state (language, stage, history, etc.). The frontend JavaScript does not need to manage tokens or session IDs manually.

**5.2.4 User Interface Updates:**

The JavaScript code dynamically updates the DOM to create the interactive chat experience: \* addMessage() appends new message divs to the chatbox. \* handleQuestion() modifies the visibility and content of input elements based on the question type. \* Input placeholders are updated based on the conversation stage.

#### 5.3 Android Application Integration (Anticipated)

While the Android application code is not part of the analyzed repository, its integration with the existing backend API is straightforward due to the RESTful design.

**5.3.1 Technologies:**

* **Language:** Kotlin or Java.
* **HTTP Client:** A library like Retrofit, Volley, or OkHttp would be used to handle network communication with the backend API.
* **JSON Parsing:** Libraries like Gson or Moshi would be used to serialize request bodies and deserialize JSON responses.
* **UI Toolkit:** Android’s native UI toolkit (XML layouts, Jetpack Compose) would be used to build the chat interface.

**5.3.2 API Communication:**

The Android app would replicate the communication pattern of the web frontend:

1. **Session Initialization:** When the user starts a chat in the app, the app would use its HTTP client library to send a POST request to the backend’s /api/start endpoint.
2. **Session Cookie Management:** The HTTP client library must be configured to handle cookies. It needs to receive the session cookie set by the backend in the /api/start response and automatically include it in all subsequent requests to /api/chat. Most standard Android HTTP clients provide mechanisms for cookie persistence (e.g., using a CookieJar with OkHttp).
3. **Sending Messages:** When the user inputs a message or selects an option:
   * The app would construct a JSON object containing the message.
   * It would use the HTTP client to send a POST request to /api/chat with the JSON payload and the automatically included session cookie.
4. **Receiving and Processing Responses:** The app would receive the JSON response from /api/chat.
   * It would parse the JSON data using a library like Gson.
   * Based on the content of the response (error, next\_question, assessment stage prompt, AI response), it would update the native Android UI accordingly (displaying text in TextViews, showing/hiding buttons, updating input fields).

**5.3.3 User Interface Implementation:**

The Android app would need to implement its own native chat interface, likely using components like RecyclerView to display the list of messages, EditText for text input, and Button elements for sending messages and potentially for boolean choices (similar to the web frontend).

#### 5.4 Unified API Interface

The key to seamless integration for both web and mobile frontends is the well-defined RESTful API provided by the Flask backend. By exposing functionality through /api/start and /api/chat endpoints that accept JSON requests and return JSON responses, the backend remains agnostic to the type of client interacting with it. As long as the client can make HTTP requests, handle cookies for session management, and process JSON data, it can integrate with the MediPulse core logic.

### Section 6: Appendices

#### 6.1 Sequence Diagram (PlantUML)

@startuml MediPulse Sequence Diagram  
  
actor User  
participant "Web Frontend" as WebApp  
participant "Android App" as AndroidApp  
participant "Backend API (Flask)" as Backend  
participant "Groq LLM" as LLM  
  
autonumber  
  
User -> WebApp: Access MediPulse Website  
WebApp -> Backend: POST /api/start  
note right of Backend: Create session, set index=0, stage=\"questions\", lang=\"en\"  
Backend --> WebApp: 200 OK (session\_id, question: Language?)  
WebApp --> User: Display Language Selection (EN/AR)  
  
User -> WebApp: Select Language (e.g., Arabic)  
WebApp -> Backend: POST /api/chat (message: \"Arabic\")  
note right of Backend: Update session language=\"ar\", index=1  
Backend --> WebApp: 200 OK (next\_question: Age?)  
WebApp --> User: Display Age Question (in Arabic)  
  
User -> WebApp: Enter Age (e.g., 30)  
WebApp -> Backend: POST /api/chat (message: \"30\")  
note right of Backend: Store age, index=2  
Backend --> WebApp: 200 OK (next\_question: Smoker?)  
WebApp --> User: Display Smoker Question (in Arabic)  
  
User -> WebApp: Answer Smoker (e.g., No)  
WebApp -> Backend: POST /api/chat (message: \"No\")  
note right of Backend: Store smoker status, index=3  
Backend --> WebApp: 200 OK (next\_question: Conditions?)  
WebApp --> User: Display Conditions Question (in Arabic)  
  
User -> WebApp: Answer Conditions (e.g., No)  
WebApp -> Backend: POST /api/chat (message: \"No\")  
note right of Backend: Store conditions, index=4, stage=\"assessment\"  
Backend --> WebApp: 200 OK (message: Describe symptoms?, stage: assessment)  
WebApp --> User: Prompt for Symptoms (in Arabic)  
  
User -> WebApp: Describe Symptoms (e.g., \"I have a headache and fever\")  
WebApp -> Backend: POST /api/chat (message: \"Headache and fever\")  
  
Backend -> Backend: Retrieve conversation history & context from session  
Backend -> Backend: Format prompt using MEDICAL\_PROMPT\_BASE (Arabic)  
Backend -> LLM: Send formatted prompt + history + symptoms  
LLM --> Backend: Generate response based on protocol  
  
note right of Backend: Process LLM response, check for safety flags  
Backend --> WebApp: 200 OK (message: AI Response)  
WebApp --> User: Display AI Response (General info, disclaimers, etc.)  
  
== Alternative Flow: Android App ==  
  
User -> AndroidApp: Open MediPulse App  
AndroidApp -> Backend: POST /api/start  
note right of Backend: Create session, set index=0, stage=\"questions\", lang=\"en\"  
Backend --> AndroidApp: 200 OK (session\_id, question: Language?)  
AndroidApp --> User: Display Language Selection (EN/AR)  
  
User -> AndroidApp: Select Language (e.g., English)  
AndroidApp -> Backend: POST /api/chat (message: \"English\")  
note right of Backend: Update session language=\"en\", index=1  
Backend --> AndroidApp: 200 OK (next\_question: Age?)  
AndroidApp --> User: Display Age Question (in English)  
  
... (Interaction continues similarly to WebApp) ...  
  
@enduml