TRABAJO DE GRADO
Opción Seminario-Diplomado.

Sistema de Monitoreo de Salud

Corporación Universitaria Remington
Facultad de Ingeniería
Ingeniería en Sistema

Darbin Garcia Acosta
Alex Duván Perlaza
Tutor: Diego Fernando Marín Lozano

Opción de Trabajo de grado Seminario-Diplomado.
2025

Dedicatoria

.

Darbin García Acosta: Dedicatoria especial para mis padres y familia especial mente a mis padres mi hija que siempre han estado ahí apoyándome motivándome para salir adelante cumplir mis metas. Este trabajo es fruto de todo el esfuerzo realizado también darle un agradecimiento Dios que sin el nada de esto sería posible, gracias por darnos sabiduría, confianza para romper las dificultades que se nos presentan en el camino y a si cumplir esta hermosa etapa.

Alex Duvan Perlaza: Dedico este Seminario-Diplomado a mi Madre y mi Tía que son dos pilares fundamentales en mi vida. Gracias por su esfuerzo constante y sus enseñanzas que han guiado cada paso de mi camino que los sueños se alcanzan con disciplina, valentía y fe. A mi novia cuyo apoyo inagotable fueron mi mayor impulso de seguir adelante gracias por creer en mi cuando yo dudaba, por tus palabras de ánimo y por qué fuiste una motivación durante este proceso.

Tabla de Contenido

Resumen 4
Palabras clave 4
Pregunta orientadora de la búsqueda 5
Sustentación teórica de la pregunta 5
Problema 7
Metodología 8
Desarrollo 10
  Backend 15
  API Endpoints 27
  Seguridad 28
  Documentación 30
  Service 1 - Generador de Signos Vitales 32
  Service 2 - Análisis de Datos 35
  Agente Conversacional 43
Conclusiones 53
Referencias 54

Resumen

El presente trabajo de grado propone el diseño y construcción de un Sistema de Monitoreo de Salud basado en microservicios, orientado a supervisar signos vitales en tiempo real y mejorar la gestión de datos biomédicos. El proyecto surge ante la persistente falta de integración tecnológica en muchos centros médicos, donde los sistemas tradicionales continúan operando de forma manual o aislada, dificultando la atención oportuna de los pacientes.

El sistema propuesto emplea tecnologías modernas como Flask, FastAPI, MongoDB, Docker, Chart.js y agentes virtuales con inteligencia artificial, lo que permite una comunicación más eficiente entre los servicios y facilita el acceso a información clínica en tiempo real. La metodología utilizada fue el modelo en cascada, desarrollando fases de análisis, diseño, construcción, pruebas y despliegue.

El impacto académico del proyecto se refleja en el fortalecimiento de competencias en arquitecturas distribuidas, salud digital e inteligencia artificial; mientras que el impacto práctico es potencial, ya que el sistema aún no ha sido implementado en instituciones médicas.

Palabras clave

Palabras clave: Microservicios, Salud Digital, Telemedicina, IA, Flask, FastAPI, Docker

Pregunta orientadora de la búsqueda

¿Cómo puede un sistema de monitoreo de salud basado en microservicios mejorar la gestión y el análisis de los signos vitales de los pacientes en tiempo real?

Sustentación teórica de la pregunta

El desarrollo de sistemas de salud modernos exige integrar arquitecturas eficientes que permitan el procesamiento ágil de datos clínicos. Según (Albertsons Companies, 2024) arquitectura de software influye directamente en la escalabilidad y mantenibilidad del sistema. Los microservicios, al dividir la aplicación en componentes independientes, permiten que cada módulo evolucione sin afectar al resto del sistema.

En salud digital, la telemedicina es reconocida por la (Pochu, 2019) como una herramienta clave para ampliar el acceso a servicios médicos, especialmente en zonas rurales. La inteligencia artificial contribuye al análisis de datos en tiempo real, identificación de patrones anormales y generación de alertas tempranas.

Bases de datos NoSQL como MongoDB ofrecen flexibilidad para almacenar datos biomédicos en grandes volúmenes. Herramientas como Flask y FastAPI permiten construir APIs eficientes para comunicación en tiempo real. Docker garantiza la portabilidad del sistema, facilitando despliegues consistentes en múltiples entornos. (Safa Ben Atitallah, Maha Driss, Henda Ben Ghezala, 2023)

La literatura reciente destaca que los sistemas distribuidos mejoran la interoperabilidad, disponibilidad y seguridad en entornos clínicos, además de optimizar la gestión hospitalaria y reducir riesgos asociados a procesos manuales.

Problema

Las problemáticas de las plataformas de cursos en línea, es que muchas plataformas no cuentan con disponibilidad de información, ya que muchos contenidos están protegidos por derechos de autor, y otra parte de la información está de manera física limitando así el acceso y no se tienen un seguimiento de aprendizaje personalizado (Navarro, Berbek, & Sanchez, 2024). Además, muchos usuarios no pueden acceder por temas de distancias, discapacidades, tiempo y responsabilidades diarias (UNESCO, 2024).

Dichas plataformas de aprendizaje están construidas en arquitecturas monolíticas, las cuales ejecutan todos los procesos sobre un mismo servicio, afectando así su rendimiento, estas plataformas dificultan su escalamiento, ya que al realizar un cambio o añadir características o información se pueden ver afectados otros procesos, ocasionando la caída del servicio en su totalidad y afectando disponibilidad frente a los usuarios. Así mismo, al ser un solo servicio, tienden a demandar muchos recursos, siendo así más lenta

Metodología

El desarrollo del proyecto se basó en la metodología en cascada, (IONOS, 2024) Apropiada para proyectos que requieren documentación estructurada y control riguroso por fases. Las etapas fueron:

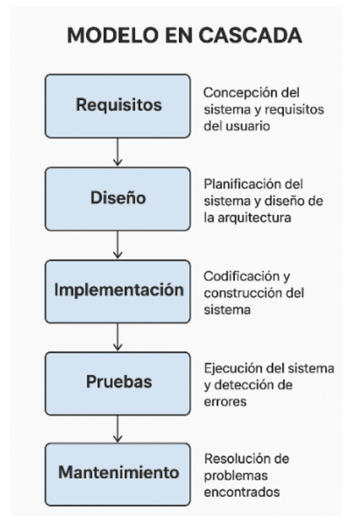


Figura 1 Metodología de cascada Modelo (Fuente de los autores)

1. Análisis de requisitos: Identificación de necesidades del sistema: monitoreo en tiempo real, gestión de roles (paciente/médico), alertas automáticas, comunicación entre servicios y almacenamiento seguro de datos.

2. Diseño del sistema: Modelado arquitectónico mediante microservicios:

- Servicio de autenticación
- Servicio de captura de signos vitales
- Servicio de procesamiento y análisis
- Servicio del agente conversacional Pulsito en Telegram
- Frontend desarrollado con Flask

3. Desarrollo: Construcción en Python 3.12 empleando Flask, FastAPI, MongoDB, Chart.js y contenedores Docker.

4. Pruebas: Validación de autenticación, comunicación entre servicios, visualización de gráficos y funcionamiento de alertas.

5. Despliegue: Contenerización completa en Docker para garantizar portabilidad, replicabilidad y mantenimiento del sistema.

Desarrollo

El frontend del sistema está desarrollado con Flask (Python) y utiliza JavaScript con Chart.js para la visualización de datos en tiempo real. La aplicación implementa un sistema de autenticación con roles (paciente/médico) y dashboards diferenciados.

Tecnologías Utilizadas

- Flask: Framework web backend
- Chart.js: Librería para gráficas interactivas
- MongoDB: Base de datos (conexión mediante módulo data_base_mongo)
- HTML5/CSS3: Estructura y diseño responsivo
- JavaScript ES6: Lógica del cliente.

Estructura del Proyecto

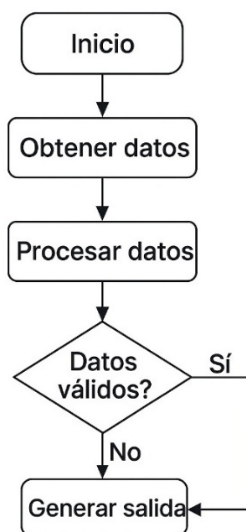
```
frontend/
├── app.py            # Aplicación Flask principal
├── Dockerfile        # Contenedor Docker
├── templates/        # Plantillas HTML
│   ├── login.html
│   ├── registro.html
│   ├── paciente_dashboard.html
│   └── medico_dashboard.html
└── static/
    ├── css/
    │   └── styles.css
    └── js/
        └── dashboard.js
```

Ilustración 1. Matriz de riesgos del sistema. Fuente: Los autores.

Riesgo identificado	**Descripción del riesgo**	**Probabilidad**	**Impacto**	**Nivel de criticidad**	**Estrategia de mitigación**
Falla en la comunicación entre microservicios	Los componentes del sistema pueden perder conexión, generando interrupciones en la transmisión de datos biomédicos.	Alta	Alta	🔴 **Crítico**	Implementar pruebas continuas, uso de protocolos seguros (HTTPS, WebSocket) y contenedores Docker para garantizar estabilidad.
Pérdida o corrupción de datos clínicos	Posible fallo en la base de datos o en el almacenamiento de información médica sensible.	Media	Alta	🔴 **Crítico**	Realizar copias de seguridad automáticas, cifrar la información y usar políticas de recuperación ante desastres.
Ataques cibernéticos o brechas de seguridad	Intentos de acceso no autorizado a los datos de pacientes o manipulación del sistema.	Media	Alta	🔴 **Crítico**	Aplicar autenticación robusta, encriptación de extremo a extremo y monitoreo constante de vulnerabilidades.
Errores de software en módulos IA o sensores	Fallos en los algoritmos o en la calibración de los dispositivos biomédicos.	Media	Media	🟠 **Moderado**	Probar los módulos con datos de prueba, aplicar mantenimiento preventivo y auditorías de código.

Resistencia del personal médico al uso del sistema	Algunos usuarios pueden mostrar poca disposición al cambio tecnológico.	Alta	Media	🟠 **Moderado**	Implementar capacitaciones, manuales de usuario y acompañamiento técnico en la adopción del sistema.
Falla del servidor o caída del sistema	Interrupción en el funcionamiento general debido a sobrecarga o mantenimiento.	Baja	Alta	🟠 **Moderado**	Utilizar servidores redundantes, balanceo de carga y monitoreo de disponibilidad en la nube.
Retrasos en el desarrollo o implementación	Problemas en la planificación o ejecución del cronograma del proyecto.	Media	Media	🟠 **Moderado**	Usar metodologías ágiles, control de versiones y reuniones semanales de seguimiento.
Desactualización tecnológica	Las herramientas empleadas pueden volverse obsoletas rápidamente.	Media	Baja	🟢 **Bajo**	Actualizar dependencias, mantener documentación técnica y revisar las tendencias del sector salud digital.
Problemas de compatibilidad entre dispositivos médicos	Los sensores o equipos no se integran correctamente con el sistema.	Baja	Media	🟢 **Bajo**	Validar estándares de comunicación (HL7, FHIR) y realizar pruebas de interoperabilidad antes del despliegue.

Error humano en la gestión de datos o configuración	Los usuarios pueden introducir información incorrecta o mal configurar el sistema.	Alta	Baja	🟢 **Bajo**	Implementar validaciones automáticas, registros de auditoría y capacitaciones periódicas.



Inicio: El proceso comienza.

Obtener datos: Se recopilan los datos necesarios para trabajar.

Procesar datos: Los datos obtenidos se analizan o transforman.

Datos válidos? Se evalúa si la información procesada cumple con las condiciones requeridas.

Sí: Si los datos son correctos, el proceso continúa sin repetir los pasos anteriores.

No: Si los datos no son válidos, se regresa a obtener nuevos datos para repetir el ciclo.

Generar **salida:** Finalmente, se produce el resultado final del proceso.

Ilustración 2Proceso de Obtención, Procesamiento y Validación de Datos

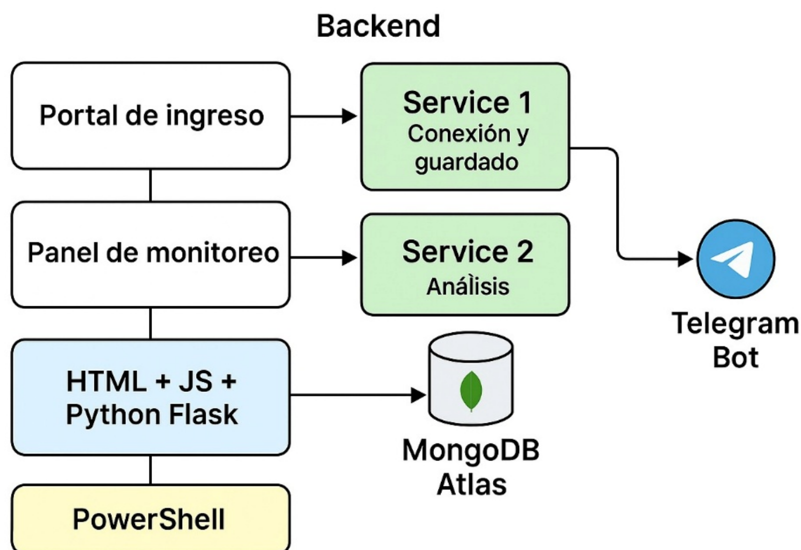


Ilustración 3 Servicios Backend y Comunicación con MongoDB y Telegram Bot

El diagrama muestra cómo funciona un sistema dividido en Frontend y Backend, junto con una base de datos y notificaciones automáticas.

Portal de ingreso: Los usuarios acceden al sistema por una página web donde se autentican o ingresan datos.

Panel de monitoreo: Una interfaz donde se visualizan los datos procesados y el estado del sistema en tiempo real.

HTML + JavaScript + Flask: Esta es la parte del frontend y servidor web que controla la comunicación con los servicios del backend y la base de datos.

PowerShell: Se usa para ejecutar comandos, scripts o automatizar tareas dentro del sistema si hace falta.

Backend

Service 1 – Conexión y guardado

Recibe datos desde el sistema y los almacena en la base de datos.

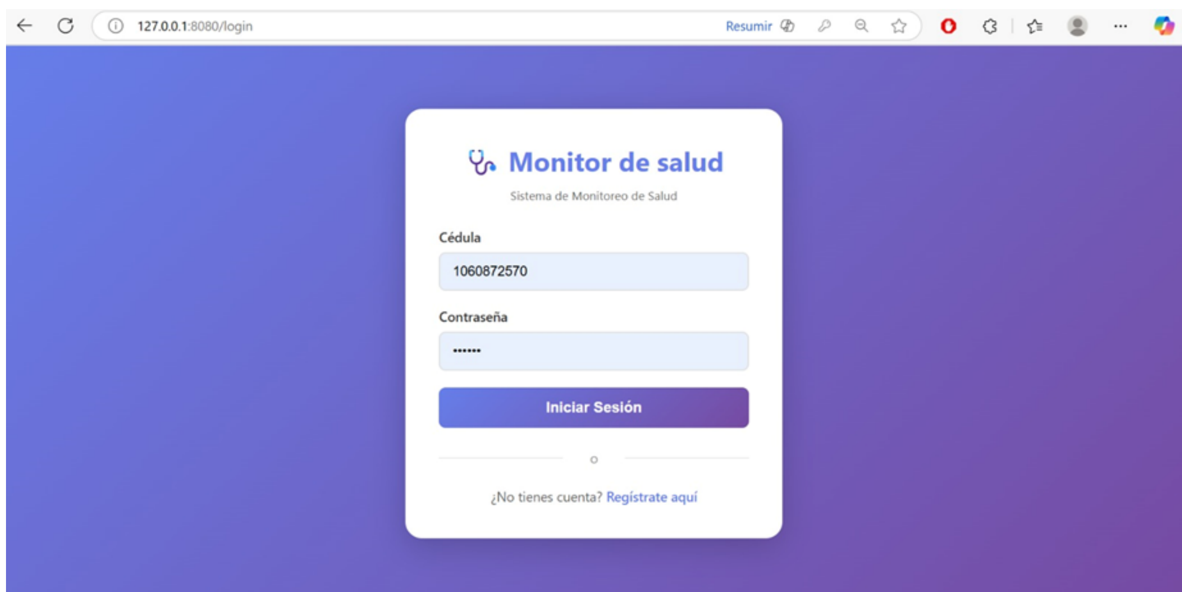
Service 2 – Análisis

Toma los datos almacenados y los procesa para generar resultados, alertas o información útil.

MongoDB Atlas: Es la base de datos en la nube donde se guardan todos los registros y datos del sistema.

Telegram Bot: Después del análisis, el sistema puede enviar alertas o notificaciones automáticas a Telegram.

Flujo de Autenticación

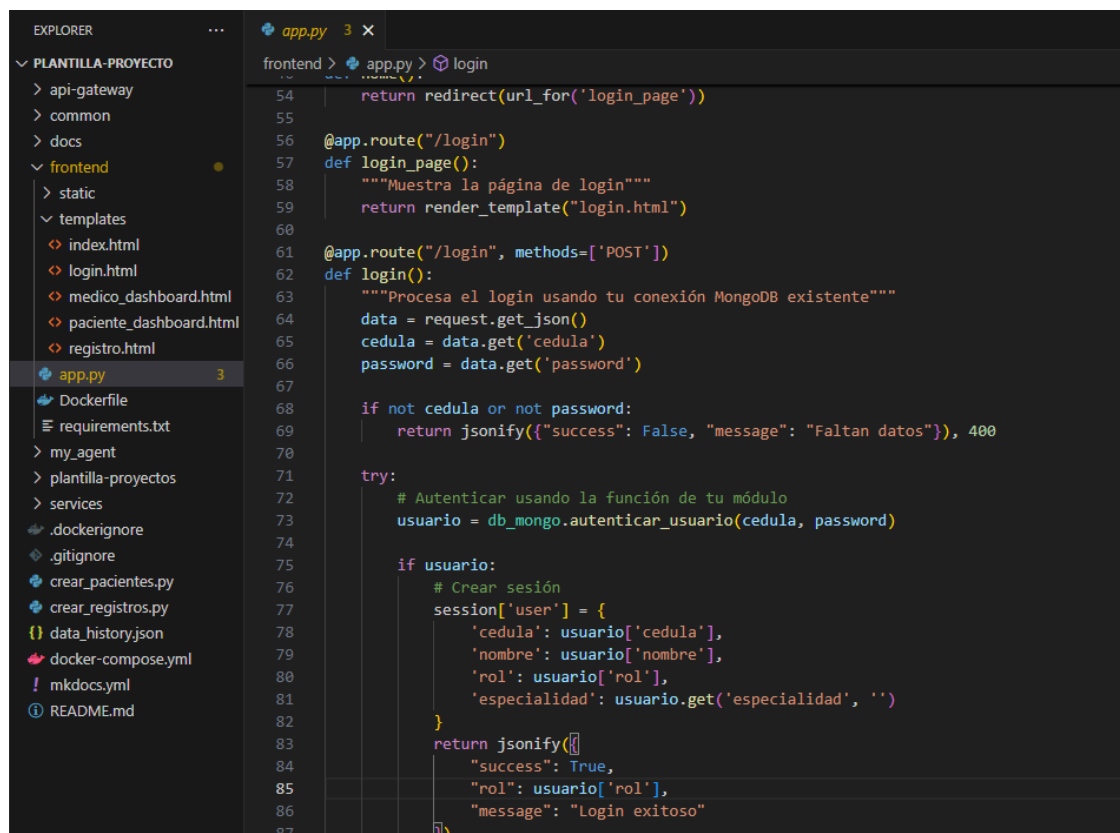


Captura de pantalla de login.html

La pantalla de inicio de sesión permite el acceso diferenciado por roles:

- Campo de cédula (usuario)
- Campo de contraseña
- Link de registro para nuevos pacientes
- Validación de credenciales contra MongoDB

Es la interfaz donde el usuario escribe su cédula y contraseña. Sirve para permitir el ingreso al sistema dependiendo del rol (médico o paciente).



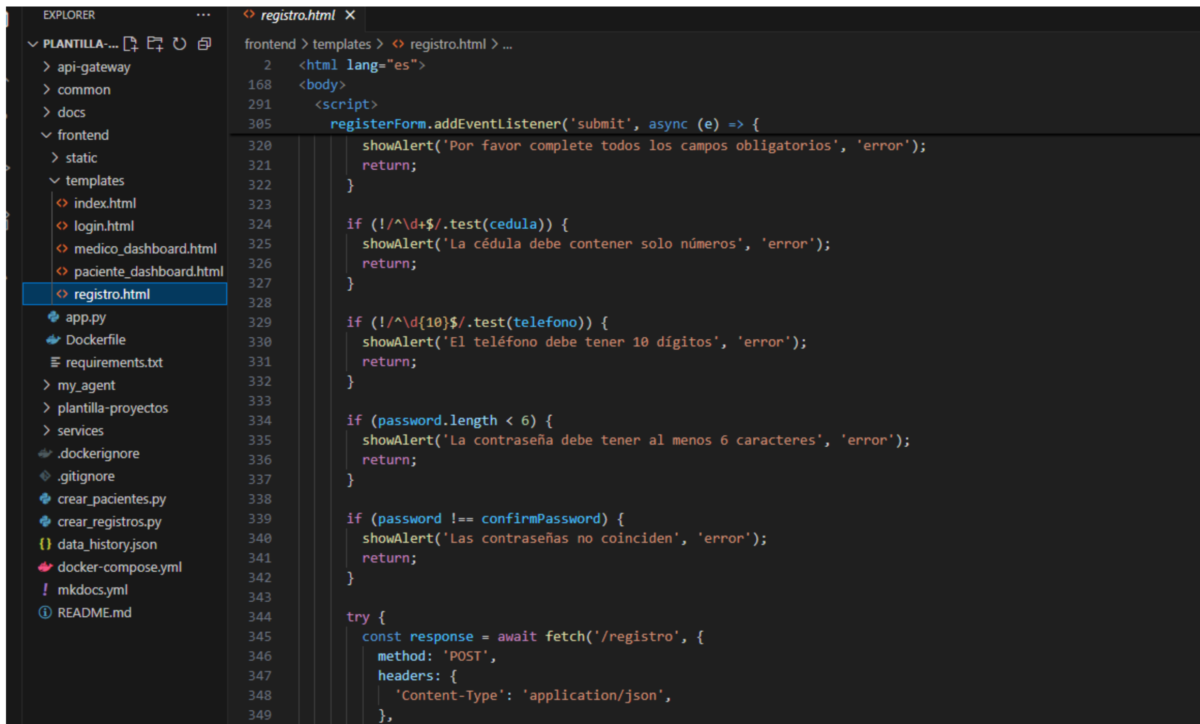
Código de la ruta /login en app.py *Muestra la lógica de autenticación con MongoDB*

Este código recibe los datos del login, busca en la base de datos, valida la contraseña con hash y según el rol del usuario redirige al dashboard correspondiente.

Panel de registro para registrar nuevo paciente

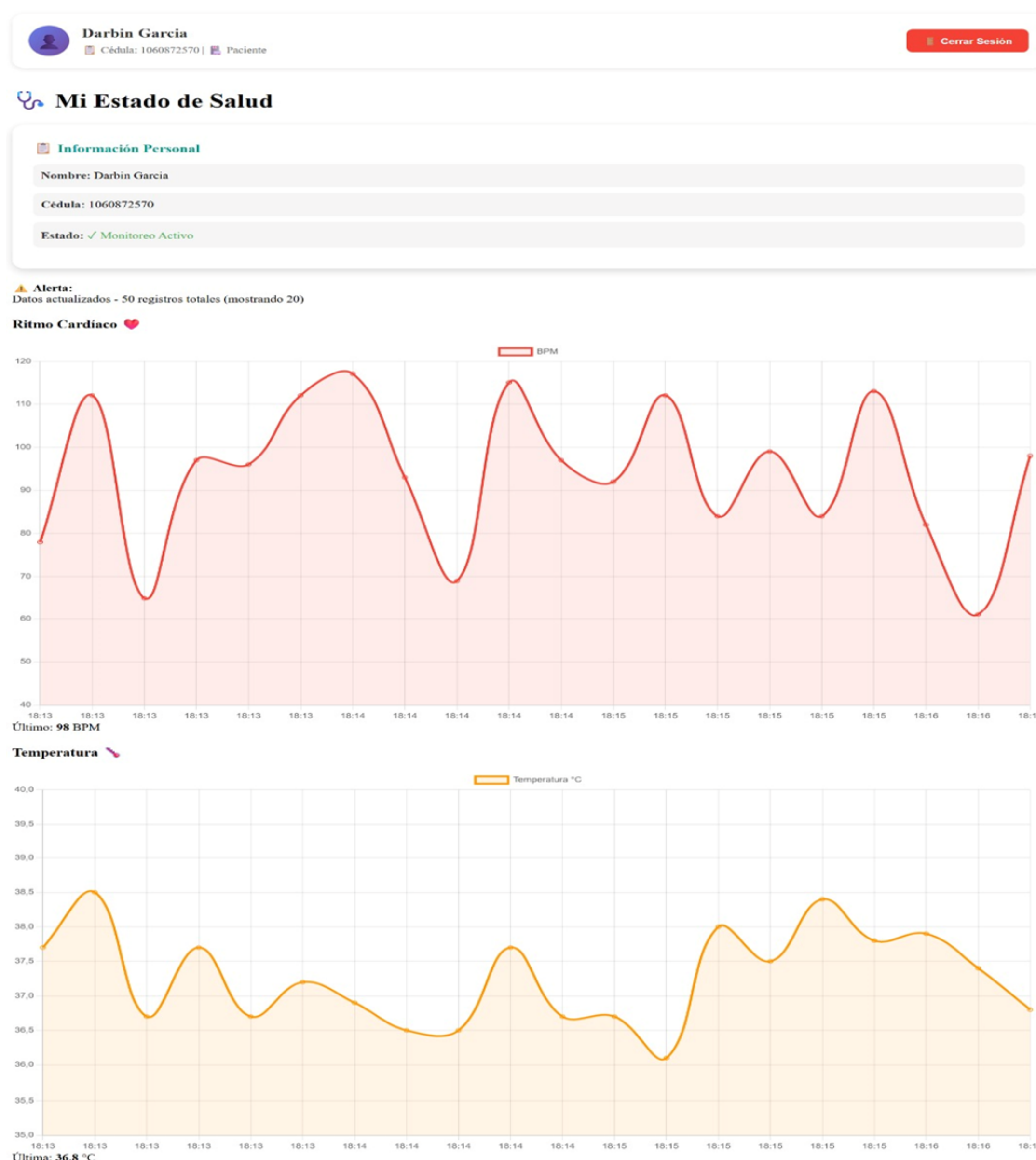
Captura de pantalla de registro.html

Formulario de registro con campos completos de información del paciente donde se ingresan los datos completos del paciente para crearlo en el sistema. Incluye nombre, cédula, edad, contraseña, etc.



Código del formulario de registro.html (sección del form)] *Muestra los campos y validaciones HTML,* contiene todos los campos del registro y define validaciones básicas como que un campo es obligatorio

Dashboard de Paciente



Registro en vivo del paciente

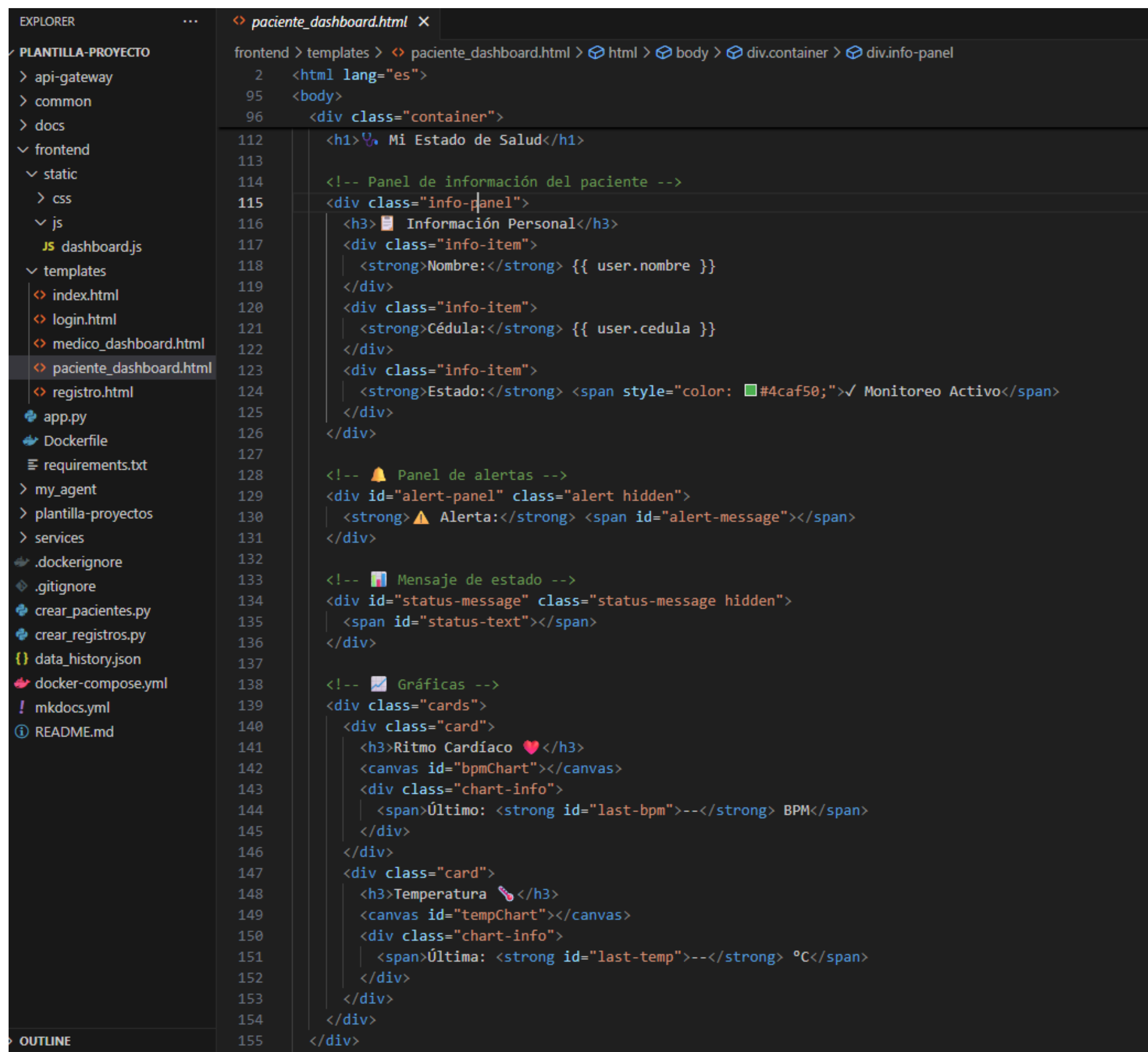
Captura de paciente_dashboard.html mostrando las gráficas

Características principales:

- Auto-carga: Muestra automáticamente los datos del paciente actual
- Información personal: Nombre y cédula en el header

- Monitoreo en tiempo real: Actualización automática cada 5 segundos

Pantalla donde el paciente ve sus signos vitales en gráficas. Los datos se actualizan automáticamente.

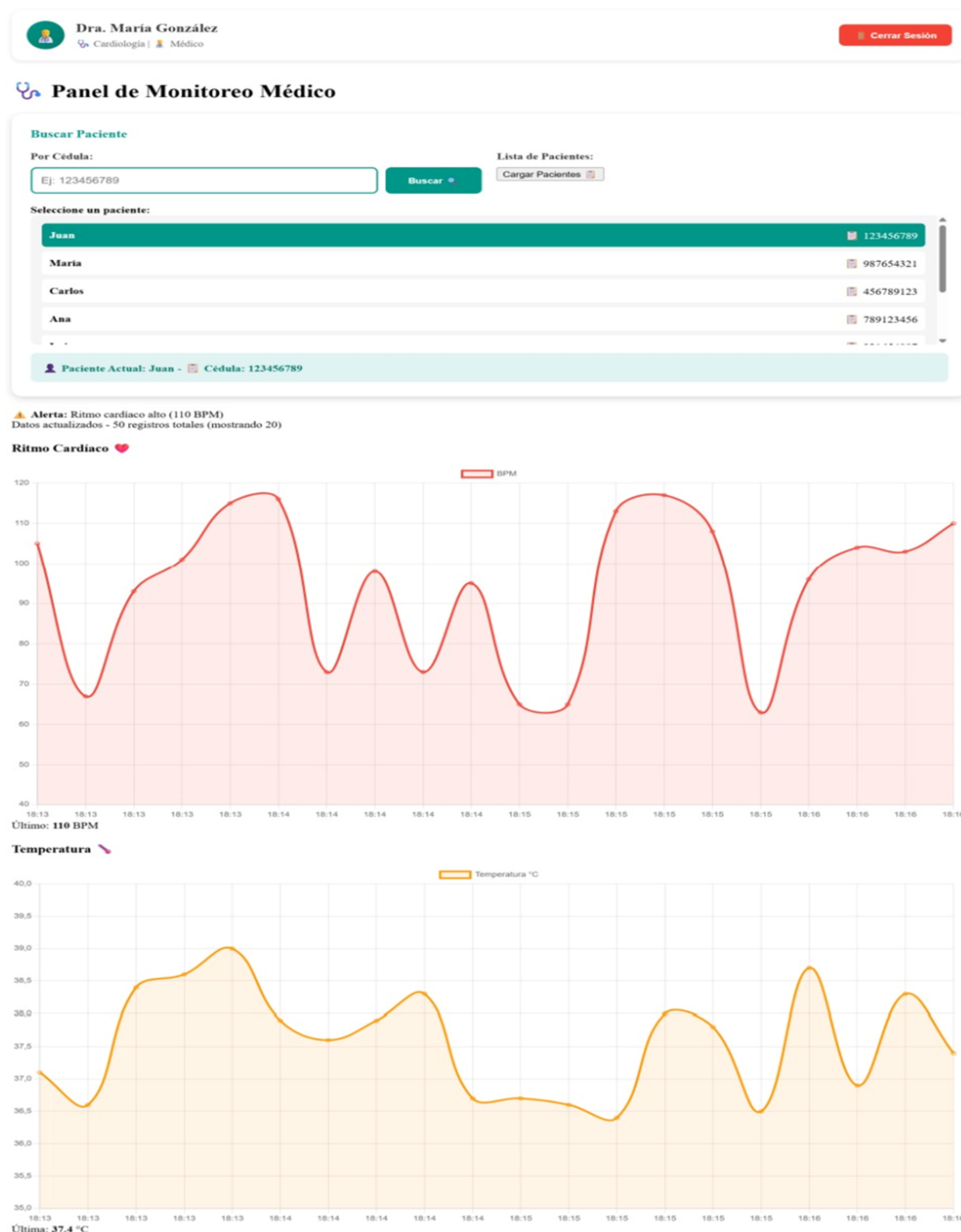



```
2    <html lang="es">
95   <body>
96     <div class="container">
112      <h1>🩺 Mi Estado de Salud</h1>
113
114      <!-- Panel de información del paciente -->
115      <div class="info-panel">
116        <h3>📋 Información Personal</h3>
117        <div class="info-item">
118          <strong>Nombre:</strong> {{ user.nombre }}
119        </div>
120        <div class="info-item">
121          <strong>Cédula:</strong> {{ user.cedula }}
122        </div>
123        <div class="info-item">
124          <strong>Estado:</strong> <span style="color: #4caf50;">✓ Monitoreo Activo</span>
125        </div>
126      </div>
127
128      <!-- 🔔 Panel de alertas -->
129      <div id="alert-panel" class="alert hidden">
130        <strong>⚠ Alerta:</strong> <span id="alert-message"></span>
131      </div>
132
133      <!-- 📊 Mensaje de estado -->
134      <div id="status-message" class="status-message hidden">
135        <span id="status-text"></span>
136      </div>
137
138      <!-- 📈 Gráficas -->
139      <div class="cards">
140        <div class="card">
141          <h3>Ritmo Cardíaco ❤️</h3>
142          <canvas id="bpmChart"></canvas>
143          <div class="chart-info">
144            <span>Último: <strong id="last-bpm">--</strong> BPM</span>
145          </div>
146        </div>
147        <div class="card">
148          <h3>Temperatura 🌡️</h3>
149          <canvas id="tempChart"></canvas>
150          <div class="chart-info">
151            <span>Última: <strong id="last-temp">--</strong> °C</span>
152          </div>
153        </div>
154      </div>
155    </div>
```

Código del script de auto-carga en paciente_dashboard.html *Muestra cómo se inicia el monitoreo automático*

Un script JavaScript que cada 5 segundos consulta la API y actualiza los gráficos sin recargar la página.

Dashboard de Médico



Panel de monitoreo del medico

Pantalla donde el médico ve todos los pacientes registrados, puede buscarlos y seleccionar uno para monitorearlo

Captura de medico_dashboard.html con la lista de pacientes

Funcionalidades:

- Búsqueda por cédula: Campo de búsqueda directa
- Lista de pacientes: Botón para cargar todos los pacientes registrados
- Selección de paciente: Click en la lista para iniciar monitoreo

```
frontend > templates > medico_dashboard.html > html > body > script > startMonitoring
<html lang="es">
<body>
  <script>
    const loadPatientsBtn = document.getElementById('load-patients-btn');
    const patientListContainer = document.getElementById('patient-list-container');
    const patientList = document.getElementById('patient-list');
    const currentPatient = document.getElementById('current-patient');
    const patientName = document.getElementById('patient-name');
    const patientCedulaDisplay = document.getElementById('patient-cedula');

    loadPatientsBtn.addEventListener('click', async () => {
      try {
        const res = await fetch('/api/pacientes');
        const pacientes = await res.json();

        if (pacientes && pacientes.length > 0) {
          patientList.innerHTML = '';
          pacientes.forEach(p => {
            const item = document.createElement('div');
            item.className = 'patient-item';
            item.innerHTML = `
              <span><strong>${p.nombre}</strong></span>
              <span>📋 ${p.cedula}</span>
            `;
            item.addEventListener('click', () => {
              document.querySelectorAll('.patient-item').forEach(i => i.classList.remove('selected'));
              item.classList.add('selected');
              startMonitoring(p.cedula, p.nombre);
            });
            patientList.appendChild(item);
          });
          patientListContainer.classList.remove('hidden');
          showStatus(`${pacientes.length} pacientes encontrados`);
        } else {
          showStatus('No hay pacientes registrados', true);
        }
      } catch (error) {
        showStatus('Error al cargar pacientes', true);
      }
    });

    function startMonitoring(cedula, nombre) {
      currentCedula = cedula;
      patientName.textContent = nombre || 'Desconocido';
      patientCedulaDisplay.textContent = cedula;
      currentPatient.classList.remove('hidden');

      if (updateInterval) clearInterval(updateInterval);
```

Código de la función loadPatientsBtn en medico_dashboard.html *Muestra la lógica para cargar y mostrar la lista de pacientes.*

Función JavaScript que llama al backend, obtiene la lista de pacientes y los muestra en

pantalla en forma de botones.

Componentes JavaScript (dashboard.js)

Configuración de Gráficas

```javascript
let currentCedula = null;
let updateInterval = null;

// Configuración de gráficas
let bpmChart = new Chart(bpmCtx, {
  type: 'line',
  data: {
    labels: [],
    datasets: [{
      label: 'BPM',
      data: [],
      borderColor: '#f44336',
      backgroundColor: 'rgba(244, 67, 54, 0.1)',
      tension: 0.4,
      fill: true
    }]
  },
  options: {
    responsive: true,
    maintainAspectRatio: true,
    scales: {
      y: {
        beginAtZero: false,
        min: 40,
        max: 120
      }
    },
    plugins: {
      legend: {
        display: true,
        position: 'top'
      }
    }
  }
});

let tempChart = new Chart(tempCtx, {
  type: 'line',
  data: {
    labels: [],
    datasets: [{
      label: 'Temperatura °C',
      data: [],
      borderColor: '#ff9800',
      backgroundColor: 'rgba(255, 152, 0, 0.1)',
      tension: 0.4,
      fill: true
    }]
```

Código de configuración de Chart.js en dashboard.js *Configuración de las gráficas de BPM y temperatura*

Define cómo se ven las gráficas: títulos, ejes, límites, estilo de líneas y datasets de signos vitales.

Sistema de Actualización

```javascript
}

// Función para actualizar las gráficas
async function updateCharts() {
  if (!currentCedula) return;

  try {
    const res = await fetch(`/api/data/${currentCedula}`);

    if (!res.ok) {
      if (res.status === 404) {
        showStatus('No se encontró historial para esta cédula', true);
        clearCharts();
        return;
      }
      throw new Error(`Error ${res.status}`);
    }

    const data = await res.json();

    // La API devuelve un objeto con la estructura: {cedula, total_registros, registros_mostrados, historial: [...]}
    if (data.historial && Array.isArray(data.historial) && data.historial.length > 0) {
      const historial = data.historial;

      const timestamps = historial.map(d => {
        // El timestamp ya viene en formato "2025-10-30 23:13:15"
        const parts = d.timestamp.split(' ')[1]; // Obtener solo la hora
        return parts.substring(0, 5); // HH:MM
      });

      const bpm = historial.map(d => d.datos.ritmo_cardiaco);
      const temp = historial.map(d => d.datos.temperatura);

      // Actualizar gráficas
      bpmChart.data.labels = timestamps;
      bpmChart.data.datasets[0].data = bpm;
      tempChart.data.labels = timestamps;
      tempChart.data.datasets[0].data = temp;

      bpmChart.update();
      tempChart.update();

      // Actualizar valores actuales
      const lastBpm = bpm[bpm.length - 1];
      const lastTemp = temp[temp.length - 1];
      lastBpmElement.textContent = lastBpm;
      lastTempElement.textContent = lastTemp.toFixed(1);
```

Función updateCharts() en dashboard.js *Lógica para obtener y actualizar datos en tiempo real*

Al consultar datos en tiempo real y actualiza los valores dentro de las gráficas sin reiniciar la página.

Sistema de Alertas

```
EXPLORER
PLANTILLA-PROYECTO
> api-gateway
> common
> docs
v frontend
  v static
    > css
    v js
      dashboard.js
  v templates
    index.html
    login.html
    medico_dashboard.html
    paciente_dashboard.html
    registro.html
  app.py
  Dockerfile
  requirements.txt
> my_agent
> plantilla-proyectos
> services
.dockerignore
.gitignore
crear_pacientes.py
crear_registros.py
data_history.json
docker-compose.yml
mkdocs.yml
README.md
> OUTLINE
> TIMELINE
```

```javascript
// dashboard.js
// frontend > static > js > dashboard.js > startMonitoring
}

// Función para verificar alertas
function checkAlerts(bpm, temp) {
  let messages = [];

  if (bpm > 100) messages.push(`Ritmo cardíaco alto (${bpm} BPM)`);
  if (bpm < 50) messages.push(`Ritmo cardíaco bajo (${bpm} BPM)`);
  if (temp > 37.8) messages.push(`Temperatura elevada (${temp.toFixed(1)}°C)`);
  if (temp < 35.5) messages.push(`Temperatura baja (${temp.toFixed(1)}°C)`);

  if (messages.length > 0) {
    alertMessage.textContent = messages.join(' | ');
    alertPanel.classList.remove('hidden');
    alertSound.play().catch(err => console.log('No se pudo reproducir el sonido'));
  } else {
    alertPanel.classList.add('hidden');
  }
}

// Función para iniciar el monitoreo
function startMonitoring(cedula) {
  currentCedula = cedula;
  patientCedula.textContent = cedula;
  patientInfo.classList.remove('hidden');

  // Limpiar intervalo anterior si existe
  if (updateInterval) {
    clearInterval(updateInterval);
  }

  // Primera actualización inmediata
  updateCharts();

  // Actualizar cada 5 segundos
  updateInterval = setInterval(updateCharts, 5000);
}

// Event listener para el botón de búsqueda
searchBtn.addEventListener('click', () => {
  const cedula = cedulaInput.value.trim();

  if (!cedula) {
    showStatus('Por favor ingrese una cédula', true);
    return;
  }

  if (!/^\d+$/.test(cedula)) {
```

Función checkAlerts en dashboard.js *Condiciones de alerta para valores fuera de rango:* Revisa si el paciente tiene valores peligrosos como fiebre o taquicardia y muestra alertas visuales.

Estado de monitoreo activo, alerta de actualizacion de los datos

Captura mostrando una alerta activa en el dashboard: Ejemplo visual cuando el sistema detecta valores peligrosos y muestra una advertencia al usuario.

API Endpoints

```
 98        return render_template('registro.html')
 99
100    @app.route("/registro", methods=['POST'])
101    def registro():
102        """Procesa el registro de nuevos pacientes"""
103        data = request.get_json()
104        cedula = data.get('cedula')
105        nombre = data.get('nombre')
106        apellido = data.get('apellido', '')
107        fecha_nacimiento = data.get('fecha_nacimiento', '')
108        email = data.get('email')
109        telefono = data.get('telefono')
110        telegram_user_id = data.get('telegram_user_id', '')
111        password = data.get('password')
112
113        if not all([cedula, nombre, email, telefono, password]):
114            return jsonify({"success": False, "message": "Faltan datos obligatorios"}), 400
115
116        try:
117            # Verificar si ya existe usando tu función
118            if db_mongo.usuario_existe(cedula):
119                return jsonify({"success": False, "message": "La cédula ya está registrada"}), 400
120
121            # Crear usuario usando tu función con todos los campos
122            if db_mongo.crear_usuario(
123                cedula=cedula,
124                nombre=nombre,
125                email=email,
126                telefono=telefono,
127                password=password,
128                rol='paciente',
129                apellido=apellido,
130                fecha_nacimiento=fecha_nacimiento,
131                telegram_user_id=telegram_user_id
132            ):
```

Código de las rutas principales en app.py *Endpoints de los dashboards y APIs:* Define las rutas principales del backend como login, registro, obtener datos, listar pacientes y cerrar sesión.

Ruta	Método	Descripción
/login	POST	Autenticación de usuario
/registro	POST	Registro de nuevos pacientes
/api/data/<cedula>	GET	Obtener historial de salud
/api/pacientes	GET	Listar pacientes (solo médicos)
/logout	GET	Cerrar sesión

Seguridad

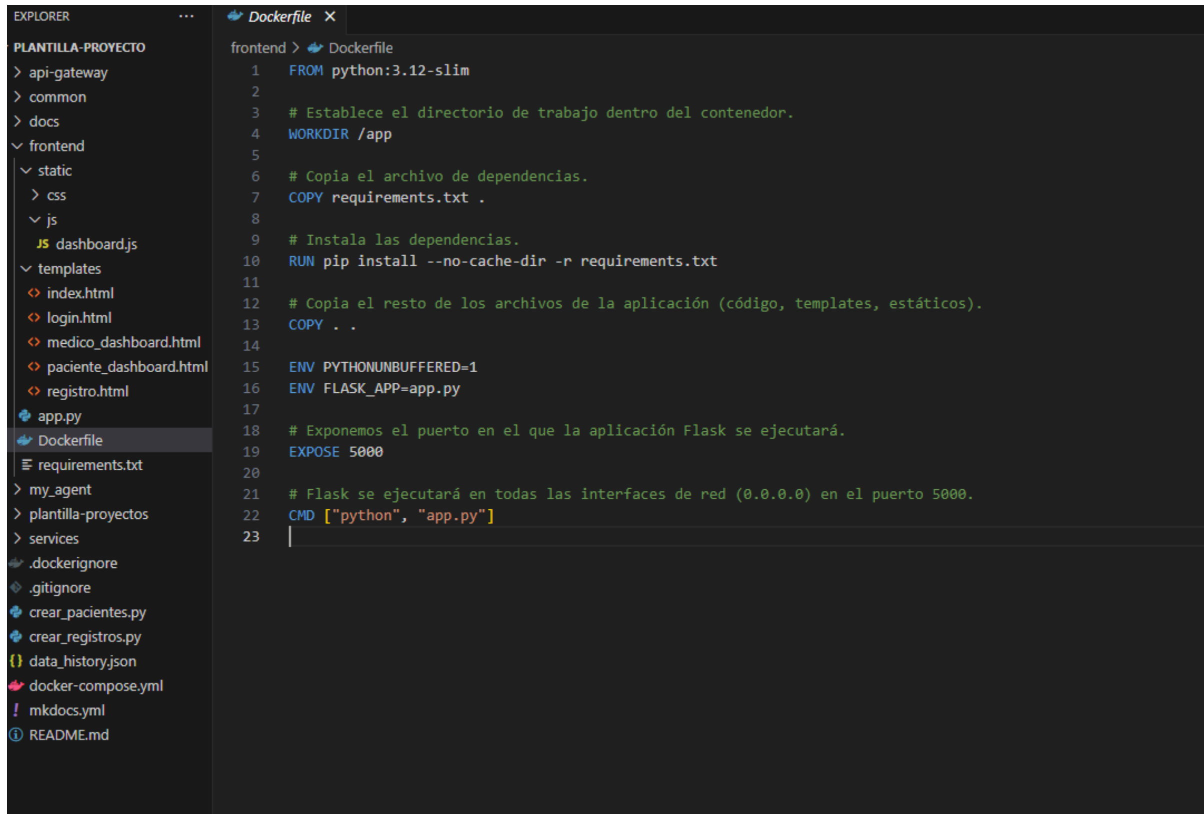
```python
@login_required
def get_data(cedula):
    """Obtiene el historial de salud por cédula"""
    try:
        response = requests.get(f"{SERVICE2_URL}/{cedula}")
        if response.status_code == 200:
            return jsonify(response.json())
        elif response.status_code == 404:
            return jsonify({"error": "No se encontró historial para esta cédula"}), 404
        else:
            return jsonify({"error": f"Error al conectar con Service2: {response.status_code}"}), 500
    except Exception as e:
        return jsonify({"error": str(e)}), 500

@app.route("/api/pacientes")
@role_required('medico')
def get_pacientes():
    """Lista todos los pacientes (solo para médicos)"""
    try:
        pacientes = db_mongo.obtener_pacientes()
        return jsonify(pacientes)
    except Exception as e:
        print(f"Error al obtener pacientes: {e}")
        return jsonify([])

# ============ PÁGINAS DE ERROR ============

@app.errorhandler(404)
def not_found(e):
    return jsonify({"error": "Ruta no encontrada"}), 404

@app.errorhandler(500)
def internal_error(e):
    return jsonify({"error": "Error interno del servidor"}), 500

# ============ INICIALIZACIÓN ============

@app.before_request
def inicializar_db():
    """Inicializa datos de prueba en el primer request"""
    if not hasattr(app, 'db_initialized'):
        db_mongo.inicializar_datos_prueba()
        app.db_initialized = True

if __name__ == "__main__":
    print("🚀 Iniciando servidor Flask...")
    print(f"📊 Usando conexión MongoDB desde services/data_base_mongo.py")
    app.run(debug=True, host='0.0.0.0', port=8080)
```

Código de los decoradores @login_required y @role_required en app.py Sistema *de protección de rutas por autenticación y rol*

- Sesiones: Manejo de sesiones Flask con secret_key
- Decoradores: Verificación de autenticación y roles
- Validación: Datos validados en cliente y servidor

Protegen rutas que solo pueden acceder usuarios logueados o con rol médico.

Despliegue con Docker



Contenido del Dockerfile

Define cómo se construye el contenedor Docker del frontend Flask para ejecutarlo fácilmente en cualquier entorno.

Documentación

El sistema utiliza una arquitectura basada en microservicios independientes que se comunican entre sí:

```
services/
├── data_base_mongo.py      # Módulo compartido de BD
├── utils.py                # Utilidades compartidas
├── service1/               # Generador de datos
├── service2/               # Análisis de datos
└── authentication/         # Autenticación (futuro)
```

Módulo Compartido: data_base_mongo.py

```python
# Intentar cargar .env desde la raíz del proyecto (una carpeta arriba de `services`)
here = os.path.abspath(os.path.dirname(__file__))
project_root = os.path.abspath(os.path.join(here, os.pardir))
dotenv_path = os.path.join(project_root, ".env")
if os.path.exists(dotenv_path):
    load_dotenv(dotenv_path)
else:
    # Fallback: intentar cargar desde el cwd
    load_dotenv()

# Leer variables de entorno
# Si no hay MONGO_URI, conectamos al localhost por defecto
MONGO_URI = os.getenv("MONGO_URI", "mongodb://localhost:27017")
DB_NAME = os.getenv("DB_NAME")

try:
    client = MongoClient(MONGO_URI)

    if DB_NAME:
        # Usar la DB especificada por la variable de entorno
        db = client[DB_NAME]
    else:
        # Intentar obtener la base de datos por defecto del URI (si existe)
        db = client.get_default_database()
        if db is None:
            # No hay DB configurada; no intentar indexar con None
            print("⚠ DB_NAME no configurada y URI no contiene una base de datos por defecto. db será None.")

    if db is not None:
        print("✅ Conectado a MongoDB correctamente crack")

        # Crear índices únicos para las colecciones de usuarios
        try:
            db.usuarios.create_index('cedula', unique=True)
            db.pacientes.create_index('cedula', unique=True)
        except:
            pass  # Los índices ya existen
    else:
        print("⚠ No hay conexión activa con MongoDB (db is None)")

except Exception as e:
    print("❌ Error al conectar con MongoDB:", e)
    db = None


# ============ FUNCIONES PARA GESTIÓN DE USUARIOS ============
```

Código de conexión MongoDB: Este Crea la conexión a la base de datos Atlas y gestiona usuarios y pacientes.

Funciones principales:

- Conexión centralizada a MongoDB Atlas
- Gestión de usuarios y pacientes
- Autenticación con hash de contraseñas

```python
# ============ FUNCIONES PARA GESTIÓN DE USUARIOS ============

def crear_usuario(cedula, nombre, email, telefono, password, rol='paciente', especialidad=None, apellido='', fecha_nacimiento='', telegram_user_id=''):
    """Crea un nuevo usuario en la base de datos"""
    if db is None:
        return False

    try:
        # Crear usuario en la colección de usuarios
        usuario = {
            'cedula': cedula,
            'nombre': nombre,
            'email': email,
            'telefono': telefono,
            'password': generate_password_hash(password),
            'rol': rol,
            'fecha_registro': datetime.now(),
            'activo': True
        }

        if rol == 'medico' and especialidad:
            usuario['especialidad'] = especialidad

        db.usuarios.insert_one(usuario)

        # Si es paciente, agregarlo a la colección de pacientes con la estructura correcta
        if rol == 'paciente':
            # Separar nombre completo en nombre y apellido si viene junto
            partes_nombre = nombre.split(' ', 1)
            nombre_paciente = partes_nombre[0]
            apellido_paciente = apellido if apellido else (partes_nombre[1] if len(partes_nombre) > 1 else '')

            # Formato de fecha como string "YYYY-MM-DD HH:MM:SS"
            fecha_actual = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

            paciente = {
                'cedula': cedula,
                'nombre': nombre_paciente,
                'apellido': apellido_paciente,
                'fecha_nacimiento': fecha_nacimiento if fecha_nacimiento else '',
                'telefono': telefono,
                'email': email,
                'telegram_user_id': telegram_user_id if telegram_user_id else '',
                'activo': True,
                'fecha_registro': fecha_actual,
                'fecha_actualizacion': fecha_actual
            }
```

Funciones de gestión de usuarios: Código que crea nuevos usuarios, valida contraseñas y busca pacientes en MongoDB.

Service 1 - Generador de Signos Vitales

Endpoints Principales

```python
    allow_headers=["*"],
)

# --- ENDPOINTS DE PACIENTES ---

@app.post("/pacientes")
def crear_paciente(cedula: str, nombre: str, apellido: str, edad: Optional[int] = None):
    """Crear o actualizar un paciente"""
    if db is None:
        raise HTTPException(status_code=500, detail="Base de datos no disponible")

    # Verificar si ya existe
    paciente_existente = db["pacientes"].find_one({"cedula": cedula})

    paciente_data = {
        "cedula": cedula,
        "nombre": nombre,
        "apellido": apellido,
        "edad": edad,
        "fecha_registro": datetime.now()
    }

    if paciente_existente:
        db["pacientes"].update_one(
            {"cedula": cedula},
            {"$set": paciente_data}
        )
        return {"mensaje": "Paciente actualizado", "cedula": cedula}
    else:
        db["pacientes"].insert_one(paciente_data)
        return {"mensaje": "Paciente creado", "cedula": cedula}


@app.get("/pacientes/{cedula}")
def obtener_paciente(cedula: str):
    """Obtener información de un paciente por cédula"""
    if db is None:
        raise HTTPException(status_code=500, detail="Base de datos no disponible")

    paciente = db["pacientes"].find_one({"cedula": cedula})
    if not paciente:
        raise HTTPException(status_code=404, detail="Paciente no encontrado")

    return serialize_mongo(paciente)


@app.get("/pacientes")
def listar_pacientes():
```

Código de endpoints en service1/main.py Service1 recibe, guarda y devuelve datos de signos vitales desde diferentes endpoints.

Endpoint	Método	Descripción
POST /pacientes	POST	Crear/actualizar paciente
GET /pacientes/{cedula}	GET	Obtener info del paciente
POST /health-data/{cedula}	POST	Generar signos vitales
GET /health-data/{cedula}	GET	Obtener signos vitales

Generación de Datos

```python
    return serialize_mongo(pacientes)


# --- ENDPOINTS DE SIGNOS VITALES ---

@app.get("/")
def root():
    return {"message": "Servicio 1 activo"}


@app.post("/health-data/{cedula}")
def generar_signos_vitales(cedula: str):
    """Generar signos vitales para un paciente específico"""
    if db is None:
        raise HTTPException(status_code=500, detail="Base de datos no disponible")

    # Verificar que el paciente existe
    paciente = db["pacientes"].find_one({"cedula": cedula})
    if not paciente:
        raise HTTPException(
            status_code=404,
            detail=f"Paciente con cédula {cedula} no encontrado. Debe registrarlo primero."
        )

    # Generar datos aleatorios simulados
    lectura = {
        "cedula": cedula,
        "nombre": f"{paciente['nombre']} {paciente['apellido']}",
        "ritmo_cardiaco": random.randint(60, 120),
        "temperatura": round(random.uniform(36, 39), 1),
        "presion": f"{random.randint(100,130)}/{random.randint(70,90)}",
        "oxigeno": random.randint(90, 100),
        "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    }

    # Guardar en MongoDB en colección "signos_vitales"
    try:
        db["signos_vitales"].insert_one(lectura.copy())
        print(f"📩 Signos vitales guardados para {lectura['nombre']} (Cédula: {cedula})")
    except Exception as e:
        print(f"⚠ Error guardando en MongoDB: {e}")
        raise HTTPException(status_code=500, detail="Error al guardar signos vitales")

    return {"status": "ok", "lectura": serialize_mongo(lectura)}


@app.get("/health-data/{cedula}")
def obtener_signos_por_cedula(cedula: str, limit: int = 10):
```

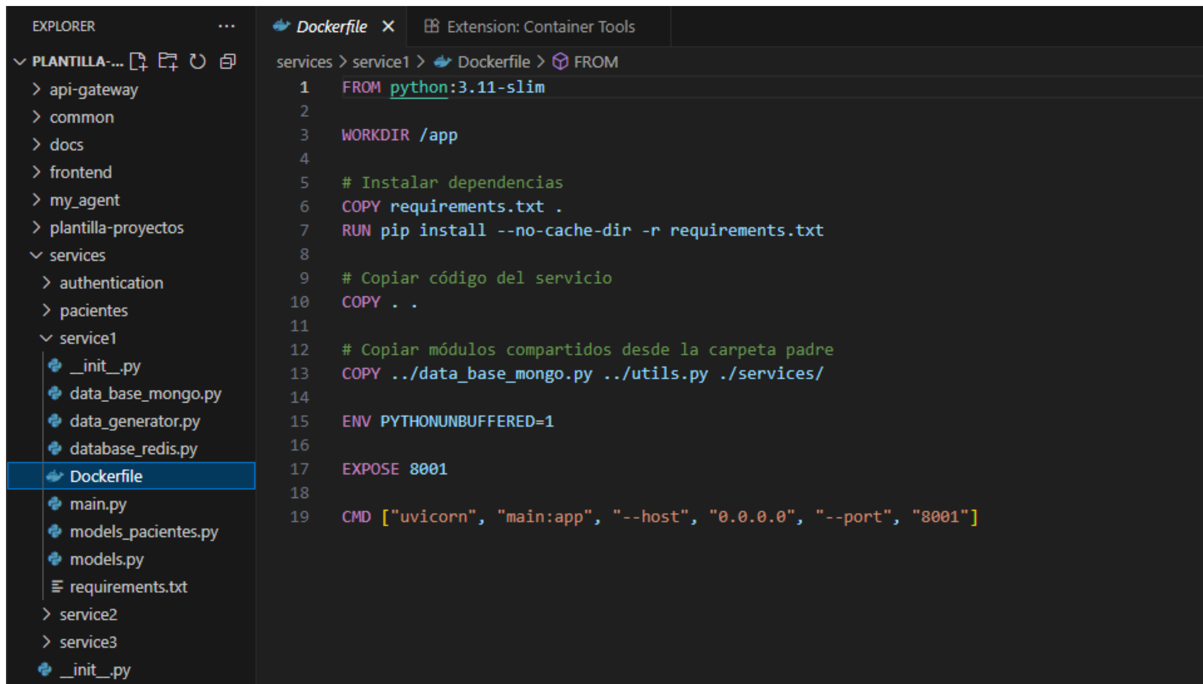
Código de generación de signos vitales

Genera signos vitales falsos (simulación) como BPM, oxígeno, presión y temperatura.

El servicio genera datos aleatorios simulados:

- Ritmo cardíaco: 60-120 BPM
- Temperatura: 36-39°C
- Presión arterial: formato "120/80"
- Saturación de oxígeno: 90-100%

Dockerfile



Contenido del Dockerfile de service1

bash

```bash
# Construcción
docker build -t service1-health .

# Ejecución
docker run -p 8001:8001 service1-health
```

Archivo para crear el contenedor Docker del servicio encargado de generar y almacenar datos de salud.

Service 2 - Análisis de Datos

```python
    return {"message": "Servicio 2 activo - Análisis por paciente"}


@app.get("/analyze/{cedula}")
def analizar_por_cedula(cedula: str):
    """Obtener el último análisis de signos vitales de un paciente"""
    try:
        # Obtener datos del service1
        import requests
        response = requests.get(f"{SERVICE1_URL}/health-data/{cedula}?limit=1")

        if response.status_code == 404:
            raise HTTPException(status_code=404, detail=f"No hay datos para la cédula {cedula}")

        if response.status_code != 200:
            raise HTTPException(status_code=500, detail="Error al consultar signos vitales")

        data = response.json()

        if "signos_vitales" not in data or not data["signos_vitales"]:
            raise HTTPException(status_code=404, detail="No hay signos vitales disponibles")

        ultimo_signo = data["signos_vitales"][0]
        alertas = analyze(ultimo_signo)

        resultado = {
            "paciente": data["paciente"],
            "timestamp": ultimo_signo.get("timestamp"),
            "datos": {
                "ritmo_cardiaco": ultimo_signo.get("ritmo_cardiaco"),
                "temperatura": ultimo_signo.get("temperatura"),
                "presion": ultimo_signo.get("presion"),
                "oxigeno": ultimo_signo.get("oxigeno")
            },
            "alertas": alertas
        }

        # Guardar en historial
        if cedula not in data_history:
            data_history[cedula] = []

        data_history[cedula].append(resultado)

        # Mantener solo últimos 50 registros por paciente
        if len(data_history[cedula]) > 50:
            data_history[cedula] = data_history[cedula][-50:]

        save_data()
```

Función analyze() en service2/main.py

Condiciones de alerta:

Analiza los datos del paciente y determina si hay riesgos como fiebre, hipoxia o taquicardia.

- Taquicardia: BPM > 100
- Bradicardia: BPM < 60
- Fiebre: Temperatura > 38°C
- Hipotermia: Temperatura < 36°C
- Hipoxia: Oxígeno < 95%

Endpoints de Análisis

```python
def analyze(data):
    if oxigeno < 95:
        alertas.append("🫁 Saturación de oxígeno baja")

    return alertas if alertas else ["✅ Todo en rangos normales"]


# --- Endpoints ---
@app.get("/")
def root():
    return {"message": "Servicio 2 activo - Análisis por paciente"}


@app.get("/analyze/{cedula}")
def analizar_por_cedula(cedula: str):
    """Obtener el último análisis de signos vitales de un paciente"""
    try:
        # Obtener datos del service1
        import requests
        response = requests.get(f"{SERVICE1_URL}/health-data/{cedula}?limit=1")

        if response.status_code == 404:
            raise HTTPException(status_code=404, detail=f"No hay datos para la cédula {cedula}")

        if response.status_code != 200:
            raise HTTPException(status_code=500, detail="Error al consultar signos vitales")

        data = response.json()

        if "signos_vitales" not in data or not data["signos_vitales"]:
            raise HTTPException(status_code=404, detail="No hay signos vitales disponibles")

        ultimo_signo = data["signos_vitales"][0]
        alertas = analyze(ultimo_signo)

        resultado = {
            "paciente": data["paciente"],
            "timestamp": ultimo_signo.get("timestamp"),
            "datos": {
                "ritmo_cardiaco": ultimo_signo.get("ritmo_cardiaco"),
                "temperatura": ultimo_signo.get("temperatura"),
                "presion": ultimo_signo.get("presion"),
                "oxigeno": ultimo_signo.get("oxigeno")
            },
            "alertas": alertas
        }
```

Código de endpoints service2

Devuelven análisis actual, historial y resumen de pacientes procesados por Service2.

Endpoint	Descripción
GET /analyze/{cedula}	Análisis del último registro
GET /historial/{cedula}	Historial completo del paciente
GET /pacientes	Resumen de todos los pacientes

Persistencia de Datos

- Almacena historial en data_history.json
- Organiza datos por cédula
- Mantiene últimos 50 registros por paciente

```json
{
    "123456789": [
        {
            "paciente": {
                "cedula": "123456789",
                "nombre": "Juan Pérez"
            },
            "timestamp": "2025-11-04 22:00:39",
            "datos": {
                "ritmo_cardiaco": 94,
                "temperatura": 36.9,
                "presion": "129/88",
                "oxigeno": 95
            },
            "alertas": [
                "✅ Todo en rangos normales"
            ]
        },
        {
            "paciente": {
                "cedula": "123456789",
                "nombre": "Juan Pérez"
            },
            "timestamp": "2025-11-04 22:00:50",
            "datos": {
                "ritmo_cardiaco": 82,
                "temperatura": 36.8,
                "presion": "127/82",
                "oxigeno": 95
            },
            "alertas": [
                "✅ Todo en rangos normales"
            ]
        },
```

Estructura del JSON de historial

Muestra cómo se guarda el historial en un archivo JSON con máximo 50 registros por paciente.

Comunicación entre Servicios

El Service2 consume datos del Service1 mediante HTTP requests:

python
```
response = requests.get(f"{SERVICE1_URL}/health-data/{cedula}")
```

Script de Generación Automática

```python
# Lista de cédulas simuladas
cedulas = ["123456789", "987654321", "456789123", "1060872570"]

# URL de los servicios
SERVICE1_URL = "http://127.0.0.1:8001/health-data"
SERVICE2_URL = "http://127.0.0.1:8002/analyze"

def generar_signos_vitales(cedula):
    """Envía datos aleatorios de signos vitales al microservicio 1 y los analiza en el 2"""
    try:
        # Crear signos vitales aleatorios
        data = {
            "ritmo_cardiaco": random.randint(60, 100),
            "temperatura": round(random.uniform(36.0, 38.0), 1),
            "presion": f"{random.randint(110, 130)}/{random.randint(70, 90)}",
            "oxigeno": random.randint(94, 100)
        }

        # Enviar al servicio 1
        r1 = requests.post(f"{SERVICE1_URL}/{cedula}", json=data, timeout=5)

        # Analizar en servicio 2
        r2 = requests.get(f"{SERVICE2_URL}/{cedula}", timeout=5)

        if r1.status_code == 200 and r2.status_code == 200:
            print(f"✅ Datos generados y analizados correctamente para cédula {cedula}")
        else:
            print(f"⚠ Error con cédula {cedula}: "
                  f"Service1 {r1.status_code}, Service2 {r2.status_code}")

    except Exception as e:
        print(f"❌ Error procesando {cedula}: {e}")

if __name__ == "__main__":
    print("🩺 Generando signos vitales automáticamente cada 10 segundos...")
    while True:
        for c in cedulas:
            generar_signos_vitales(c)
        print("⏱ Esperando 10 segundos para la siguiente ronda...\n")
        time.sleep(10)
```

Código de generate_auto_data.py (completo)

Script que simula múltiples pacientes:

Script que genera datos constantemente y los envía a los servicios para simular pacientes reales.

- Lista de cédulas predefinidas
- Genera datos cada 10 segundos
- Envía a Service1 y analiza en Service2

```python
load_dotenv()

TOKEN = os.getenv("TELEGRAM_TOKEN")
SERVICE2_URL = "http://127.0.0.1:8002"  # Service2 en puerto 8002

# Estados de conversación
ESPERANDO_CEDULA, CONVERSACION_NORMAL = range(2)

# Crear el runner una sola vez al inicio
session_service = InMemorySessionService()
runner = Runner(
    agent=root_agent,
    session_service=session_service,
    app_name='telegram_bot'
```

```
(plantilla-proyectos) → plantilla-proyecto git:(main) ✗
(plantilla-proyectos) → plantilla-proyecto git:(main) ✗ python services/pacientes/generate_auto_data.py
✅ Datos generados y analizados correctamente para cédula 456789123
✅ Datos generados y analizados correctamente para cédula 1060872570
Esperando 10 segundos para la siguiente ronda...

✅ Datos generados y analizados correctamente para cédula 123456789
✅ Datos generados y analizados correctamente para cédula 987654321
✅ Datos generados y analizados correctamente para cédula 456789123
✅ Datos generados y analizados correctamente para cédula 1060872570
Esperando 10 segundos para la siguiente ronda...
```

Terminal mostrando ejecución del script Demuestra el script funcionando en tiempo real enviando datos cada cierto tiempo.

Módulo de Utilidades (utils.py)

```python
from bson import ObjectId

def serialize_mongo(document):
    """
    Convierte ObjectId y otros tipos no serializables a formatos JSON válidos.
    """
    if isinstance(document, list):
        return [serialize_mongo(doc) for doc in document]
    if isinstance(document, dict):
        return {k: serialize_mongo(v) for k, v in document.items()}
    if isinstance(document, ObjectId):
        return str(document)
    return document
```

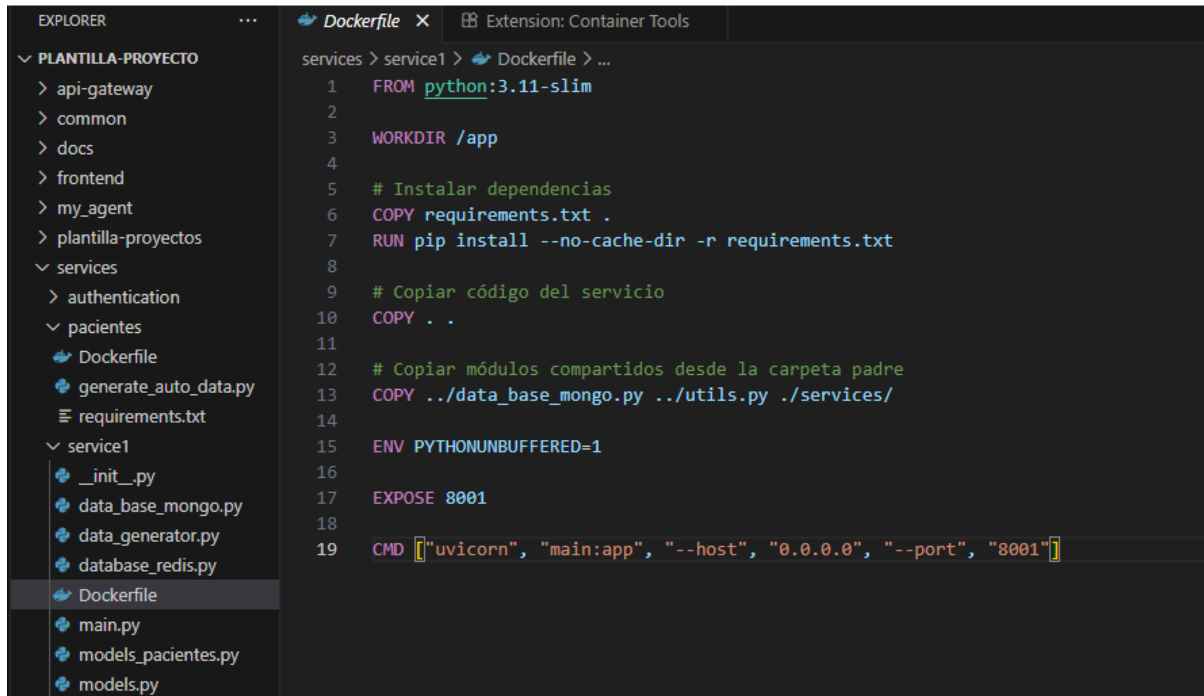
Función serialize_mongo()

Convierte los campos especiales de MongoDB (como ObjectId) a JSON.

- ObjectId → String
- Recursivo para listas y diccionarios

Despliegue con Docker

Service 1



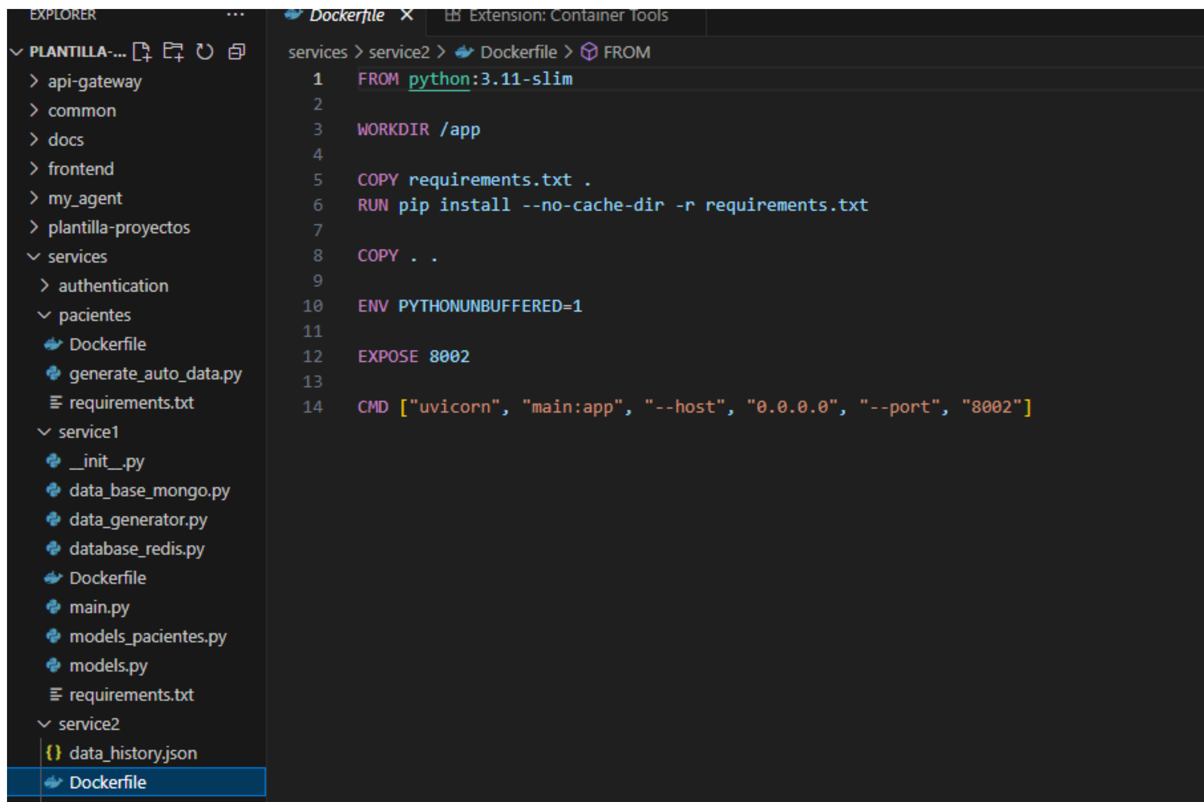
Dockerfile de service1

Asigna configuración para levantar el contenedor del primer microservicio.

bash

```
docker build -t service1 ./service1
docker run -p 8001:8001 --env-file .env service1
```

Service 2



Dockerfile de service2

bash
```
docker build -t service2 ./service2
docker run -p 8002:8002 --env-file .env service2
```

Configura el segundo microservicio para análisis de datos.

Agente Conversacional

Pulsito es un agente de IA médico desarrollado con Google ADK (Agent Development Kit) y Gemini 2.5 Flash, integrado con Telegram para proporcionar análisis de signos vitales en tiempo real.

Bot en Telegram: Muestra el bot Pulsito listo para interactuar con usuarios en Telegram.

Arquitectura del Agente

```
my_agent/
├── agent.py          # Configuración del agente Gemini
├── telegram_bot.py   # Bot de Telegram
├── health_data.py    # Conexión con Service2
└── requirements.txt  # Dependencias
```

Configuración del Agente (agent.py)

```python
from google.adk.agents.llm_agent import Agent

root_agent = Agent(
    model='gemini-2.5-flash',
    name='Pulsito',
    description='Asistente médico especializado en monitoreo de signos vitales y estado de salud de pacientes.',
    instruction='''Eres Pulsito, un asistente médico virtual especializado en el monitoreo y análisis de signos vitales de pacientes.

Tu función principal es:
- Responder preguntas sobre el estado de salud de pacientes
- Analizar e interpretar signos vitales como pulso cardíaco, temperatura corporal, presión arterial, saturación de oxígeno, etc.
- Proporcionar información médica precisa y profesional
- Ofrecer orientación sobre valores normales y anormales de signos vitales

Características de tu comunicación:
- Utiliza un lenguaje formal, profesional y técnico apropiado para el ámbito médico
- Emplea terminología médica correcta cuando sea necesario
- Sé claro, preciso y objetivo en tus explicaciones
- Mantén un tono empático pero profesional
- Si detectas signos vitales fuera de rangos normales, sugiere consultar con un profesional de la salud

Rangos normales de referencia para adultos:
- Pulso cardíaco: 60-100 latidos por minuto
- Temperatura corporal: 36.5°C - 37.5°C (oral)
- Presión arterial: 120/80 mmHg (sistólica/diastólica)
- Frecuencia respiratoria: 12-20 respiraciones por minuto
- Saturación de oxígeno: 95-100%

Recuerda siempre indicar que tus respuestas son orientativas y no sustituyen la evaluación de un médico profesional.''',
)
```

Código de configuración del agente: Define parámetros del agente IA como modelo, tono, rangos normales y comportamiento clínico.

Características:

- Modelo: Gemini 2.5 Flash
- Nombre: Pulsito
- Especialidad: Monitoreo de signos vitales
- Tono: Profesional y empático

Rangos normales programados:

- Pulso: 60-100 BPM
- Temperatura: 36.5-37.5°C
- Presión: 120/80 mmHg
- Oxígeno: 95-100%

Bot de Telegram (telegram_bot.py)

```
134 async def handle_message(update: Update, context: ContextTypes.DEFAULT_TYPE):
242             traceback.print_exc()
243             await update.message.reply_text(
244                 "❌ Ocurrió un error al procesar tu mensaje. Intenta nuevamente."
245             )
246             return CONVERSACION_NORMAL
247
248
249 async def cancel(update: Update, context: ContextTypes.DEFAULT_TYPE):
250     """Cancela la conversación"""
251     await update.message.reply_text(
252         "Conversación cancelada. Usa /start para comenzar de nuevo."
253     )
254     return ConversationHandler.END
255
256
257 if __name__ == "__main__":
258     if not TOKEN:
259         print("❌ Falta el token de Telegram en la variable TELEGRAM_TOKEN")
260         exit(1)
261
262     app = ApplicationBuilder().token(TOKEN).build()
263
264     # Manejador de conversación
265     conv_handler = ConversationHandler(
266         entry_points=[CommandHandler("start", start)],
267         states={
268             ESPERANDO_CEDULA: [
269                 MessageHandler(filters.TEXT & ~filters.COMMAND, recibir_cedula)
270             ],
271             CONVERSACION_NORMAL: [
272                 MessageHandler(filters.TEXT & ~filters.COMMAND, handle_message)
```

Estados del ConversationHandler

1. Usuario envía /start
2. Bot solicita cédula del paciente
3. Valida cédula con Service2
4. Conversación normal con contexto clínico

Controla el flujo de conversación en Telegram, desde /start hasta consultas médicas.

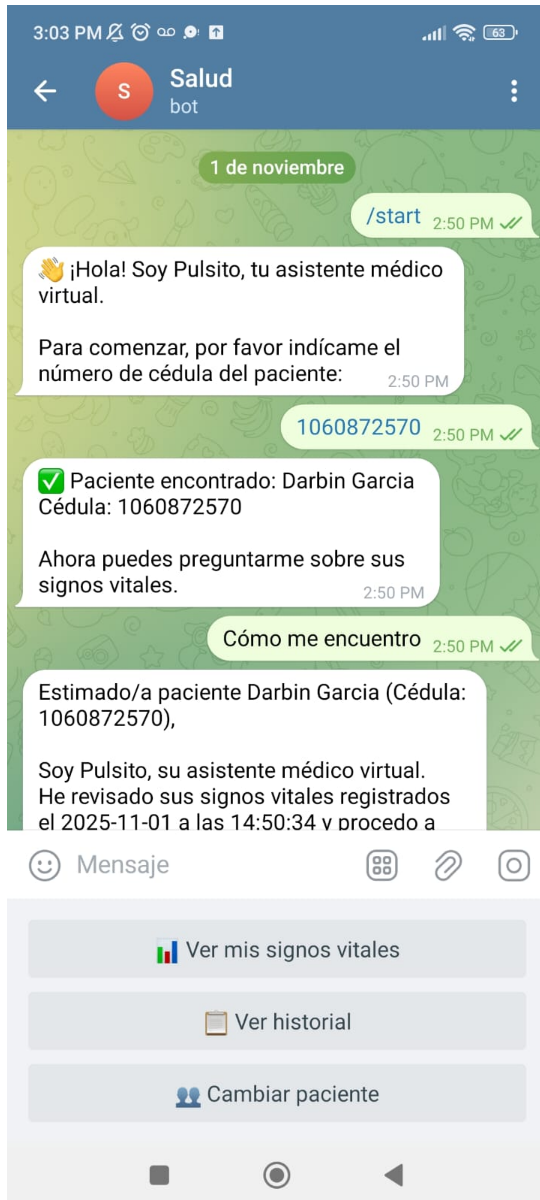
Función Principal: Solicitud de Cédula

```
EXPLORER
PLANTILLA-PROYECTO
> api-gateway
> common
> docs
> frontend
v my_agent
    __init__.py
    agent.py
    health_data.py
    requirements.txt
    telegram_bot.py  7
> plantilla-proyectos
> services
  .dockerignore
  .gitignore
  crear_pacientes.py
  crear_registros.py
  data_history.json
  docker-compose.yml
  mkdocs.yml
  README.md
OUTLINE
TIMELINE
```

```python
# telegram_bot.py 7   Extension: Container Tools
# my_agent > telegram_bot.py > start
41      return match.group(0) if match else None
42
43
44  async def start(update: Update, context: ContextTypes.DEFAULT_TYPE):
45      """Comando /start"""
46      keyboard = [
47          [KeyboardButton("📊 Ver mis signos vitales")],
48          [KeyboardButton("📋 Ver historial")],
49          [KeyboardButton("👥 Cambiar paciente")]
50      ]
51      reply_markup = ReplyKeyboardMarkup(keyboard, resize_keyboard=True)
52
53      await update.message.reply_text(
54          "👋 ¡Hola! Soy Pulsito, tu asistente médico virtual.\n\n"
55          "Para comenzar, por favor indícame el número de cédula del paciente:",
56          reply_markup=reply_markup
57      )
58      return ESPERANDO_CEDULA
59
60
61  async def recibir_cedula(update: Update, context: ContextTypes.DEFAULT_TYPE):
62      """Recibe y valida la cédula del paciente"""
63      user_id = str(update.effective_user.id)
64      texto = update.message.text
65
66      # Si el usuario presiona un botón antes de dar la cédula
67      if texto in ["📊 Ver mis signos vitales", "📋 Ver historial", "👥 Cambiar paciente"]:
68          await update.message.reply_text(
69              "Primero necesito que me proporciones el número de cédula del paciente."
70          )
71          return ESPERANDO_CEDULA
72
73      cedula = extraer_cedula(texto)
74
75      if not cedula:
76          await update.message.reply_text(
77              "❌ No pude identificar una cédula válida.\n"
78              "Por favor, envía solo el número de cédula (6-10 dígitos)."
79          )
80          return ESPERANDO_CEDULA
81
82      # Verificar si el paciente existe consultando service2
83      try:
84          response = requests.get(f"{SERVICE2_URL}/analyze/{cedula}", timeout=5)
85
86          if response.status_code == 404:
87              await update.message.reply_text(
88                  f"❌ No encontré registros para la cédula {cedula}.\n\n"
```

Función start con teclado personalizado

Función que envía el mensaje inicial del bot, incluyendo botones.



Captura del chat - Mensaje de inicio con botones: Vista del chat cuando el usuario abre el bot.

Validación de Paciente

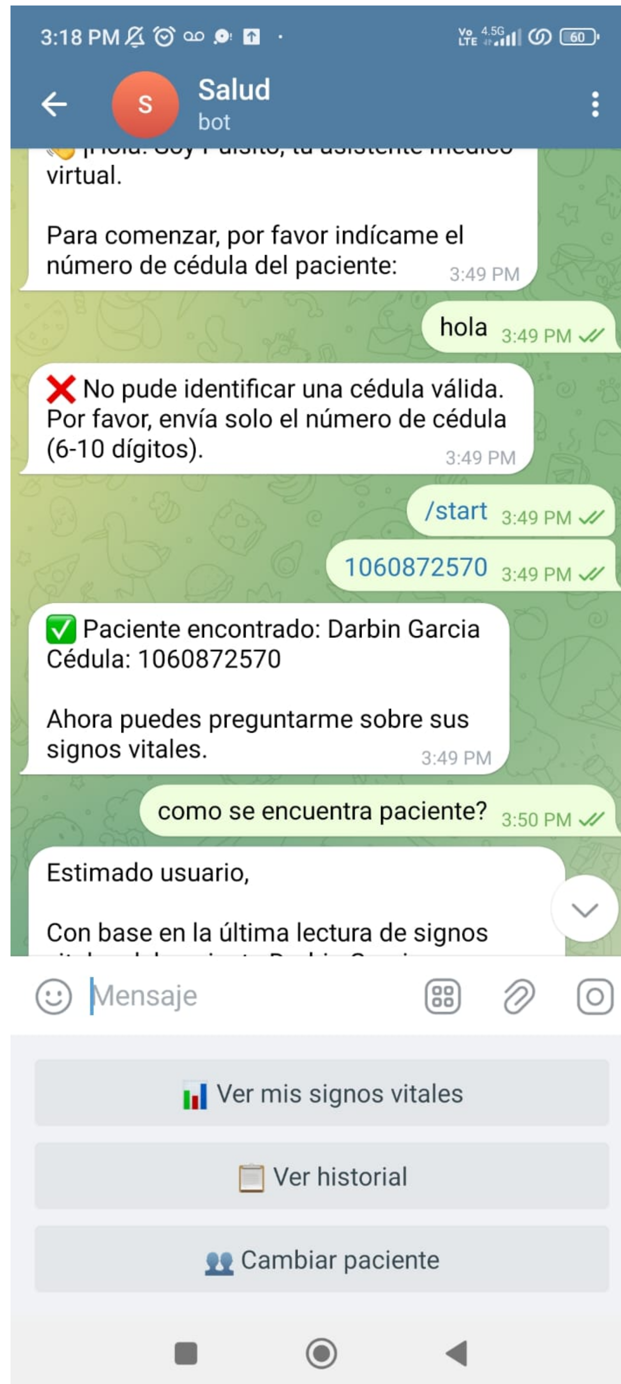
```python
        return ESPERANDO_CEDULA


async def recibir_cedula(update: Update, context: ContextTypes.DEFAULT_TYPE):
    """Recibe y valida la cédula del paciente"""
    user_id = str(update.effective_user.id)
    texto = update.message.text

    # Si el usuario presiona un botón antes de dar la cédula
    if texto in ["📊 Ver mis signos vitales", "📋 Ver historial", "👥 Cambiar paciente"]:
        await update.message.reply_text(
            "Primero necesito que me proporciones el número de cédula del paciente."
        )
        return ESPERANDO_CEDULA

    cedula = extraer_cedula(texto)

    if not cedula:
        await update.message.reply_text(
            "❌ No pude identificar una cédula válida.\n"
            "Por favor, envía solo el número de cédula (6-10 dígitos)."
        )
        return ESPERANDO_CEDULA

    # Verificar si el paciente existe consultando service2
    try:
        response = requests.get(f"{SERVICE2_URL}/analyze/{cedula}", timeout=5)

        if response.status_code == 404:
            await update.message.reply_text(
                f"❌ No encontré registros para la cédula {cedula}.\n\n"
                "Asegúrate de que el paciente esté registrado y tenga signos vitales."
            )
            return ESPERANDO_CEDULA

        if response.status_code != 200:
            await update.message.reply_text(
                "⚠️ Hubo un problema al consultar los datos. Intenta de nuevo."
            )
            return ESPERANDO_CEDULA

        data = response.json()
        paciente_nombre = data.get("paciente", {}).get("nombre", "Desconocido")

        # Guardar la cédula para este usuario
        user_cedulas[user_id] = cedula

        await update.message.reply_text(
```

Función recibir cedula : Valida que la cédula exista consultando el microservicio Service2.

El bot:

- Extrae la cédula del mensaje
- Consulta Service2: GET /analyze/{cedula}
- Verifica existencia del paciente
- Guarda asociación usuario-cédula

.



Captura del chat - Validación exitosa de cédula

Confirmación de que la cédula corresponde a un paciente válido.

Procesamiento de Mensajes

```python
async def handle_message(update: Update, context: ContextTypes.DEFAULT_TYPE):
    """Maneja los mensajes del usuario cuando ya tiene una cédula asignada"""
    user_message = update.message.text
    user_id = str(update.effective_user.id)

    # Verificar si es un botón
    if user_message == "📊 Ver mis signos vitales":
        user_message = "¿Cuáles son los últimos signos vitales?"
    elif user_message == "📋 Ver historial":
        user_message = "Muéstrame el historial de signos vitales"
    elif user_message == "👥 Cambiar paciente":
        return await cambiar_paciente(update, context)

    cedula = user_cedulas.get(user_id)

    if not cedula:
        await update.message.reply_text(
            "⚠️ Primero necesito saber la cédula del paciente.\n"
            "Usa el botón '👥 Cambiar paciente' o envía la cédula."
        )
        return ESPERANDO_CEDULA

    print(f"📩 Mensaje de {user_id} para cédula {cedula}: {user_message}")

    try:
        # Obtener los datos más recientes desde Service2
        response = requests.get(f"{SERVICE2_URL}/analyze/{cedula}", timeout=10)

        if response.status_code == 404:
            await update.message.reply_text(
                f"⚠️ No hay datos disponibles para la cédula {cedula}.\n"
                "Puede que el paciente no tenga registros recientes."
            )
            return CONVERSACION_NORMAL

        if response.status_code != 200:
            contexto = "⚠️ No se pudieron obtener los signos vitales recientes.\n\n"
        else:
            health_data = response.json()

            if "datos" in health_data:
                datos = health_data["datos"]
                paciente = health_data.get("paciente", {})
                alertas = health_data.get("alertas", [])

                contexto = (
                    f"📊 **Paciente:** {paciente.get('nombre', 'Desconocido')}\n"
                    f"📋 **Cédula:** {cedula}\n"
```

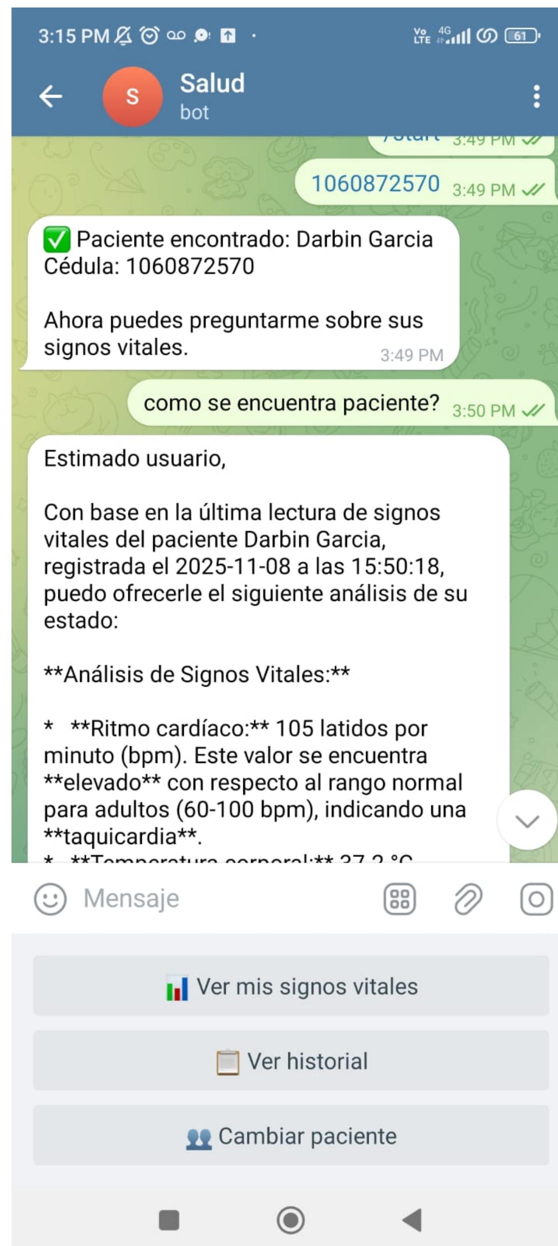
Función handle message obtiene datos: Consulta signos vitales y los envía al agente IA para que dé un análisis clínico.

Flujo:

1. Obtiene signos vitales de Service2
2. Formatea contexto clínico
3. Envía al agente Gemini
4. Retorna respuesta personalizada

Ejemplos de Uso

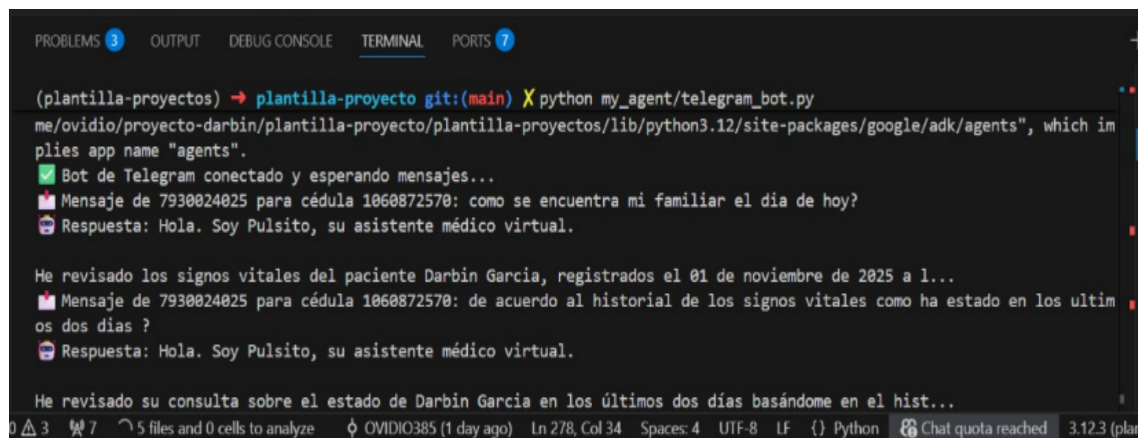
Consulta de Signos Vitales



Captura del chat - Usuario pregunta por signos vitales

Muestra al usuario preguntando y recibiendo análisis del bot.

Ejecución del Bot



Terminal con bot ejecutándose: Muestra el bot ejecutándose correctamente.

Conclusiones

El sistema de monitoreo de salud basado en microservicios demuestra ser una propuesta viable y pertinente para mejorar la gestión de datos clínicos y permitir el análisis en tiempo real de los signos vitales. La arquitectura distribuida facilita la escalabilidad, disponibilidad y mantenimiento del sistema, mientras que la inclusión de tecnologías como Docker, Flask, FastAPI y MongoDB garantiza flexibilidad y eficiencia en la operación.

Aunque aún no se implementa en un entorno hospitalario real, el modelo propuesto posee un impacto potencial significativo en la telemedicina, al permitir una atención más rápida y accesible. El proyecto también ofrece bases sólidas para futuras investigaciones en salud digital, inteligencia artificial y sistemas distribuidos.

Referencias

Albertsons Companies, U. (2024). *INTERNATIONAL JOURNAL OF RESEARCH IN COMPUTER APPLICATIONS AND INFORMATION* . Obtenido de Plataform & workflow by OJS PKP: https://ijrcait.com/index.php/home/article/view/IJRCAIT_07_02_220

Cruz, A. (S.F). *Primeros pasos con FastApi Aquí continúa tu camino en el desarrollo de aplicaciones web en Python con FastApi.* (A. Cruz, Ed.) Chicago.

Galan, J., Martin, A., Bernal, C., & Lopez, E. (2017). *Los MOOC y la Educación Superior.* (E. Octaedro, Ed.) Obtenido de https://www.google.com.co/books/edition/Los_MOOC_y_la_Educaci%C3%B3n_Superior/tgiIDwAAQBAJ?hl=es&gbpv=0

IONOS. (2024). *IONOS*. Obtenido de Digital Guide: https://www.ionos.com/es-us/digitalguide/paginas-web/desarrollo-web/el-modelo-en-cascada/

L, S. M. (s.f.). *Arquitetura de referecia para tele medecina*. Obtenido de Universidad de los andes : https://repositorio.uniandes.edu.co/flip/?pdf=/bitstreams/12f408d3-6b1c-40eb-a6a4- f759ff914afd/download

Leon, A. R., Acosta, J. L., & Diaz, R. A. (2021). Apliacacion de la metodologia incremental en el desarrollo de sistema de informacion. Obtenido de http://scielo.sld.cu/scielo.php?pid=S2218-36202021000500175&script=sci_arttext

Muñoz, P. C. (2009). *Plataformas de teleformación y herramientas telemáticas.* España.

Navarro, M., Berbek, P., & Sanchez, J. (2024). *Investigacion y conocimientos en la eduacacion actual.* (S. Dykinson, Ed.) Chicago.

Newman, S. (2019). *Monolith to Microservices:Evolutionary Patternsto Transform Your Monolith.* Chicago: O'Reilly Media, Incorporated.

Ortega, J. M. (2020). *Tecnologías para arquitecturas basadas en microservicios.* (J. M. Ortega, Ed.)

Pochu, S. (2019). *A Microservices Approach to Cloud Data Integration for Healthcare Applications*. Obtenido de UEBSS: https://unbss.com/index.php/unbss/article/view/24

Rocio , E. (2005). *Trabajo colaborativo en educación universitaria:.* (N. E. Educativa, Ed.) Mexico.

Safa Ben Atitallah, Maha Driss, Henda Ben Ghezala. (27 de Agosto de 2023). *Cornel University*. Obtenido de https://arxiv.org/abs/2308.14017

Sanchez, J. J. (2022). *Aprender Docker, un enfoque práctico.* España : Marcombo.

UNESCO. (2024). (UNESCO, Ed.) Obtenido de https://www.google.com.co/books/edition/Informe_de_seguimiento_de_la_educaci%C3%B3n/gNINEQAAQBAJ?hl=es&gbpv=0

Usaola, M. (2015). (M. P. Usaola, Ed.) España. Obtenido de https://www.google.com.co/books/edition/_/UiAvCwAAQBAJ?hl=es-419&gbpv=0