Politécnico de Coimbra

Escola Superior de Tecnologia e Gestão

Licenciatura de Engenharia Informática

Tecnologias e Aplicações Móveis

Ano letivo: 2025/2026

Sistemas Embebidos – *SafeSenior*

Elaborado em: 05/12/2025

Gabriel Lopes, a2019153399

Índice

Lista de Figuras

1 Introdução

SafeSenior, é um sistema embebido baseado em Arduino com conectividade *Wi-Fi*, concebido para aumentar a segurança e a qualidade de vida de idosos ou outros dependentes, em contexto doméstico.

O sistema visa combinar sensores de movimento e sinais vitais com um botão físico de SOS e um módulo de localização, permitindo deteção automática de quedas, acompanhamento de parâmetros de saúde e comunicação imediata de emergências para familiares ou cuidadores.

O presente documento, estrutura o trabalho realizado no alcance do primeiro módulo deste objetivo. Este módulo visou tornar possível que um botão, ao ser acionado, comunicasse com uma *API* que, por sua vez, comunicaria com um utilizador cuidador, através de uma interface. Ao observar um evento SOS, o utilizador poderia responder a este alerta ao acionar uma luz no dispositivo, por vias de comunicação remota, de forma a informar o afetado.

2 Arquitetura

A arquitetura final do *SafeSenior* resulta da evolução natural do planeamento inicial, mantendo os princípios definidos mas consolidando-os num sistema funcional composto pelas camadas sistema embebido, *backend* e *frontend*.

A arquitetura implementada suporta este fluxo, definindo responsabilidades claras em cada camada:

- **Sistema embebido** – Gere toda a interação física, nomeadamente, a leitura do botão, controlo dos LEDs e buzzer, armazenamento do *device_id* e comunicação com a *API*.
- ***Backend*** – Desenvolvido em *Flask* e ligado ao *Supabase PostgreSQL* via *REST*, centraliza toda a lógica de negócio desde a gestão de utilizadores, dispositivos, SOS, eventos de ajuda, notificações à sincronização bidirecional com o Arduino.
- ***Frontend Android*** – Fornece a interface para utilizadores, permitindo consultar estados, histórico de eventos, responder a emergências e interagir com o dispositivo de forma remota.

Esta organização garante que cada camada cumpre o seu papel de forma integrada, permitindo:

- Comunicação estável Arduino >> *API* >> Base de Dados
- Sincronização remota cuidador >> *API* >> Arduino
- Persistência coerente de eventos críticos
- Extensibilidade futura (*e.g.* ao adicionar sensores biométricos, localização, *etc.*)

Tal como previsto no planeamento original, a arquitetura manteve-se fiel ao desenho conceptual, mas a implementação final introduziu decisões práticas que reforçaram a segurança, estabilidade e simplicidade operacional, nomeadamente o uso de *tokens* distintos para utilizadores e dispositivos, a separação entre SOS e *help_event* e a sincronização periódica do estado de ajuda no *Firmware*.

Além disso, a implementação final foi preparada para garantir que o sistema opera de forma assíncrona e concorrente, permitindo que vários utilizadores possam emitir alertas SOS em simultâneo e que múltiplos cuidadores respondam independentemente a cada evento. Para isso, foi necessário introduzir mecanismos estruturais específicos:

1. Cada evento SOS é armazenado como um registo isolado na tabela *sos_event*, identificado por um *event_id* único, evitando bloqueios ou colisões entre diferentes alertas;
2. A tabela *help_event* funciona como um canal paralelo de resposta, permitindo que vários cuidadores acionem "*Help on the way*" para o mesmo dispositivo ou para dispositivos distintos, sem interferência entre si;
3. O *backend Flask* opera de forma *stateless* sobre a *REST API* do *Supabase*, o que garante que cada pedido *HTTP* é tratado de forma independente, possibilitando múltiplas requisições concorrentes sem dependência temporal entre elas;
4. O *Firmware* do Arduino realiza *polling* periódico ao estado do *help_event*, interpretando respostas remotas de forma contínua, mesmo quando vários cuidadores interagem com o sistema;
5. O uso de *tokens* distintos para utilizadores e dispositivos garante isolamento completo entre sessões, permitindo que múltiplos dispositivos e múltiplos cuidadores operem em paralelo sem sobreposição de estado.

Na figura 1, é possível analisar a arquitetura de todo este sistema e como as diversas partes interagem entre si.

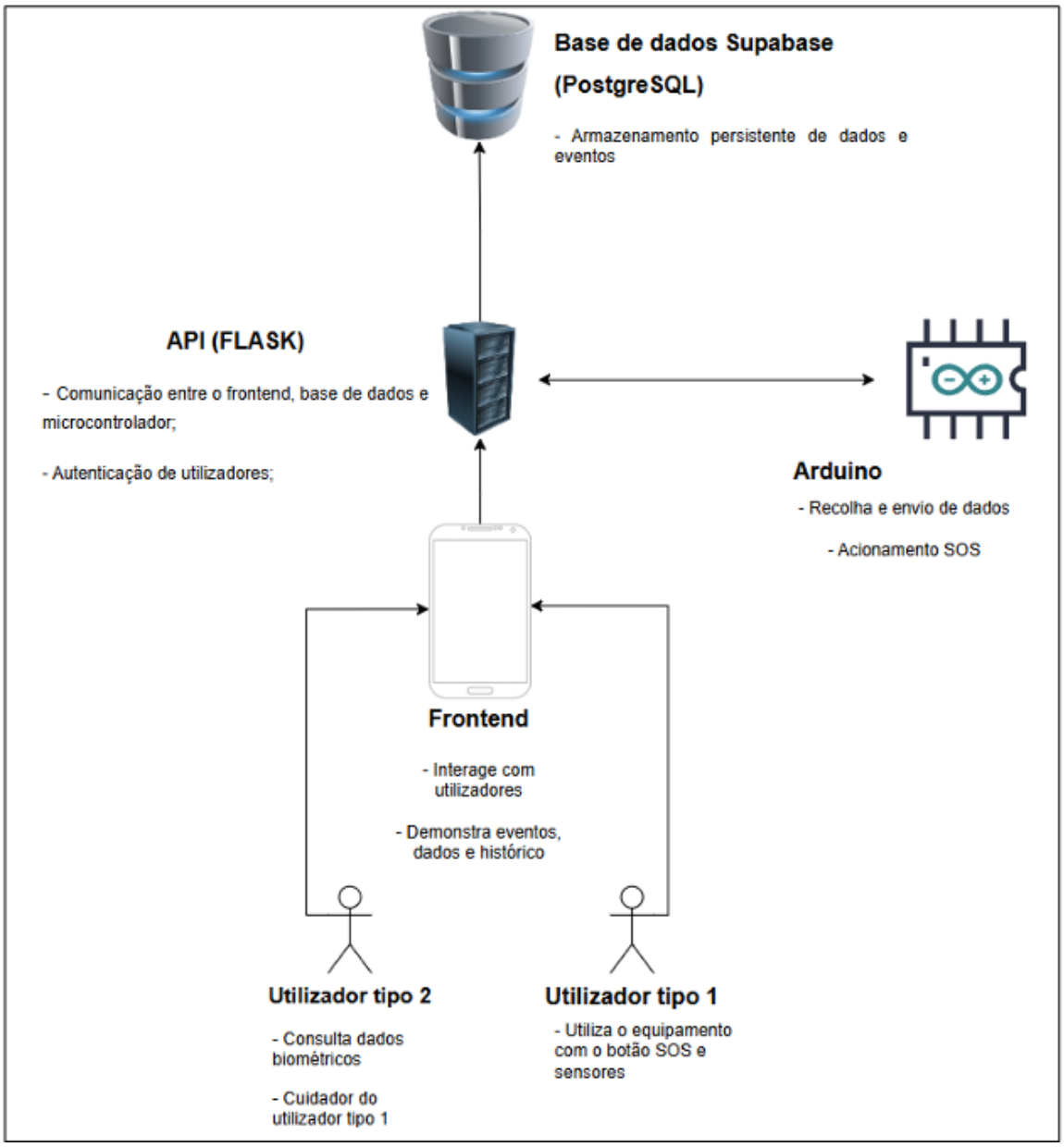


Figura 1 - Diagrama de arquitetura

2.1 Tecnologia

As decisões tecnológicas refletem tanto os requisitos funcionais definidos no planeamento inicial como as necessidades identificadas durante o desenvolvimento real do protótipo.

O sistema embebido utiliza uma placa *Arduino UNO WiFi Rev2*, escolhida pela combinação entre conectividade Wi-Fi integrada, baixo consumo energético e facilidade de desenvolvimento. Esta plataforma permite gerir o ciclo completo de recolha de eventos físicos, acionamento visual e sonoro, armazenamento persistente do *device_id* e comunicação direta com o *backend*, mantendo o *Firmware* leve e determinístico.

O *backend* foi desenvolvido em *Flask* (*Python*), funcionando como camada central de lógica e controlo. A utilização deste *microframework* garante elevada legibilidade, rapidez de desenvolvimento e integração simples com *APIs* externas.

O frontend *Android*, desenvolvido em *Java*, foi escolhido pela necessidade de implementar rapidamente uma interface dedicada a cuidadores. Esta camada permite visualizar eventos, responder a emergências e enviar comandos remotos para o dispositivo, funcionando como consumidor direto das *APIs* expostas pelo *backend*.

2.2 Sensores

Embora o planeamento inicial previsse a integração de múltiplos sensores biométricos e ambientais, o foco desta fase do projeto centrou-se na implementação fiável do mecanismo essencial, o botão físico de SOS. Este atuador é o componente crítico que desencadeia todo o fluxo de comunicação do sistema.

O botão, ligado ao Arduino através de uma entrada digital configurada em *INPUT_PULLUP*, garante deteção estável de pressões isoladas, evitando falsos positivos e possibilitando a criação de um ciclo SOS simples e eficaz. Em resposta ao mesmo, os *LEDs* e o buzzer funcionam como mecanismos de sinalização local, refletindo visual e sonoramente o estado atual do evento (SOS ativo, ajuda a caminho, cancelamento, etc.).

Os restantes sensores previstos como acelerómetro, giroscópio ou módulos de sinais vitais não foram integrados nesta etapa por não serem necessários para validar a arquitetura principal do sistema. No entanto, a arquitetura permanece preparada para a sua adição futura, uma vez que o *backend*, a base de dados e o *Firmware* já suportam rotinas e estruturas compatíveis com a expansão do conjunto de dados.

2.3 protocolos

A comunicação entre os diferentes elementos do *SafeSenior* assenta na utilização de protocolos simples, previsíveis e amplamente suportados, garantindo robustez e compatibilidade entre dispositivos de baixo consumo, servidores *web* e aplicações móveis.

A interação entre o Arduino e o *backend* ocorre exclusivamente através do protocolo *HTTP*, utilizando pedidos *REST* do tipo *POST* e *GET*. Os dados são enviados e recebidos em formato *JSON*, permitindo uma troca de informação leve e facilmente analisável em todas as camadas. Esta abordagem reduz complexidade e elimina a necessidade de bibliotecas de comunicação proprietárias.

A comunicação *backend*–base de dados é estabelecida também via *REST* através da *API* automática do *Supabase*. Cada operação (inserção, consulta, atualização ou marcação de eventos) é tratada como um pedido *HTTP* independente, garantindo isolamento total entre requisições e suportando naturalmente operações concorrentes.

No *frontend*, a comunicação com a *API* segue igualmente o modelo *RESTful*, garantindo consistência com o dispositivo. O uso de autenticação por *JWT* para utilizadores e dispositivos permite assegurar que cada chamada é validada antes de ser processada, reforçando a segurança do sistema sem comprometer a simplicidade operacional.

3 Implementação

3.1 Sistema Embebido

A implementação do *Firmware* do *SafeSenior* tem como objetivo garantir que o dispositivo consegue, de forma autónoma e fiável, identificar-se perante o *backend*, comunicar eventos de emergência, reagir à resposta do cuidador e manter um comportamento consistente mesmo perante falhas de energia ou de rede. Para isso, a lógica do *Firmware* foi organizada em quatro pilares principais:

1. Gestão do dispositivo, incluindo armazenamento persistente do *device_id*;
2. Deteção e controlo do evento SOS;
3. Sincronização periódica com o *backend*;
4. Sinalização local através de *LEDs* e *buzzer*.

3.1.1 Gestão do dispositivo

Para que o *backend* consiga distinguir vários dispositivos e associar cada SOS ao utilizador correto, cada placa Arduino necessita de um identificador único (*device_id*). Esse identificador é gerado no *backend* e passado ao dispositivo apenas uma vez. A partir daí, o *Firmware* tem de o preservar entre reinicializações e falhas de energia.

Para este efeito foi utilizada a memória *EEPROM* do *Arduino*. A *EEPROM* (*Electrically Erasable Programmable Read-Only Memory*) é uma memória não volátil integrada no microcontrolador que permite guardar pequenos blocos de dados mesmo quando o dispositivo é desligado. Esta característica torna-a ideal para guardar configurações persistentes como o *device_id*, sem obrigar o utilizador a reconfigurar o equipamento sempre que o liga.

No arranque, o *Firmware* tenta ler da *EEPROM* o *device_id* previamente armazenado. Se encontrar um valor válido, usa-o como identidade do dispositivo. Caso contrário, por exemplo, no primeiro arranque, pede ao utilizador que introduza o *device_id* através do *Serial Monitor* e grava-o de forma permanente.

A figura 2 apresenta a inicialização do sistema, a 3 a lógica na eventualidade de nenhum ID ser obtido e a 4, apresenta as funções auxiliares que permitem ler e guardar o *device_id* na *EEPROM*.

```
298   // Initialize serial monitor
299   Serial.begin(9600);
300   // Waits until the USB connection is established
301   while (!Serial);
```

Figura 2 - Inicialização Serial

```
317   deviceId = readStringFromEEPROM(0);
318
319   // If no device_id exists, wait for user input
320   if (deviceId.length() == 0) {
321
322     Serial.println("Enter device ID:");
323     while (deviceId.length() == 0) {
324       if (Serial.available()) {
325         deviceId = Serial.readStringUntil('\n');
326         deviceId.trim();
327       }
328     }
329
330     saveStringToEEPROM(0, deviceId);
331   }
```

Figura 3 - Obtenção e armazenamento do *device ID*

```
// ================================================================
//Saves device_id permanently so it survives power loss
// ================================================================
void saveStringToEEPROM(int startAddr, const String& str) {

  // Write characters one by one until null terminator
  int len = str.length();
  for (int i = 0; i < len && (startAddr + i) < MY_EEPROM_SIZE - 1; i++) {
    EEPROM.update(startAddr + i, str[i]);
  }

  // Null terminator to mark end
  EEPROM.update(startAddr + len, '\0');
}

// ================================================================
// Loads a string from EEPROM (device_id)
// ================================================================
String readStringFromEEPROM(int startAddr) {

  // Read sequential bytes until null terminator or empty cell
  String result = "";
  for (int i = startAddr; i < MY_EEPROM_SIZE; i++) {
    byte b = EEPROM.read(i);
    if (b == '\0' || b == 0xFF) break;
    result += char(b);
  }

  return result;
}
```

Figura 4 - Funções auxiliares de acesso à *EEPROM* (armazenamento e leitura, respetivamente

Depois de obter um *device_id* válido, o passo seguinte é garantir conectividade com a rede. O *Firmware* contém uma função dedicada a estabelecer ligação *Wi-Fi*, que tenta sucessivamente ligar-se ao *SSID* configurado até obter sucesso. Este comportamento é importante num contexto doméstico, em que o router pode não estar pronto quando o Arduino arranca.

Assim que a ligação é estabelecida, o *Firmware* notifica o *backend* de que o dispositivo está online, enviando um pedido *HTTP* para o *endpoint* de "*device online*". Do lado do servidor, este pedido atualiza a tabela de dispositivos, marcando o equipamento como disponível e registando o instante em que foi visto pela última vez. Esta sincronização inicial garante que o cuidador consegue saber, através do frontend, se o dispositivo está operacional.

As figuras 5 e 6 apresentam o código utilizado para estabelecer a ligação *Wi-Fi* e comunicar ao *backend* que o dispositivo está *online*, respetivamente:

```
89  // ============================================
90  // Attempts Wi-Fi connection until success.
91  // ============================================
92  void connectWiFi() {
93
94    Serial.print("Connecting");
95
96    // Try until connected
97    while (WiFi.status() != WL_CONNECTED) {
98      WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
99      delay(1000);
100     Serial.print(".");
101   }
102
103   Serial.println("\nConnected!");
104   Serial.print("IP Address: ");
105   Serial.println(WiFi.localIP());
106 }
```

Figura 5 - Função auxiliar de conexão ao WiFi

```
222 // ============================================================================
223 // Notifies API that device is online
224 // ============================================================================
225 void sendDeviceOnline() {
226
227   // Do nothing if no device_id is stored (device cannot identify itself yet)
228   if (deviceId == "") return;
229
230   // Prepare JSON payload
231   String body = "{\"device_id\":\"" + deviceId + "\"}";
232
233   int status;
234   String resp;
235
236   // Send POST
237   Serial.println("Sending ONLINE");
238   httpPostJson(ENDPOINT_ONLINE, body, status, resp);
239
240   Serial.print("Status: ");
241   if (status == 200)
242     Serial.println("Ready!");
243   else if (status == -3)
244     Serial.println("Check IP addresses");
245   else
246     Serial.println(status);
247 }
```

Figura 6 - Função auxiliar de envio de estado (online) ao backend

3.1.2 Deteção e controlo do evento SOS

A gestão do botão de SOS é o ponto de partida para todo o fluxo de emergência. O botão está ligado a um pino digital configurado como *INPUT_PULLUP*. Nesta configuração, o pino está naturalmente em nível lógico alto, passando abaixo apenas quando o botão é pressionado, o que reduz problemas de ruído e dispensa resistências externas.

No ciclo principal (*loop()*), o estado do botão é lido continuamente e comparado com o estado anterior. Em vez de reagir enquanto o botão está carregado, o código procura a transição de *HIGH* para *LOW*, evitando o disparo múltiplo de SOS numa única pressão.

Quando essa transição é detetada, é chamada a função que comunica o SOS ao *backend*. Essa função constrói uma mensagem *JSON* que inclui o device_id e envia um pedido *HTTP* para o *endpoint /sos*. Do lado do *backend*, este pedido é tratado como um *toggle*:

1. Se não existir um SOS ativo para aquele utilizador/dispositivo, é criado um novo registo na tabela de eventos (*sos_event*), marcando o instante de início
2. Se já existir um SOS ativo, esse evento é encerrado (registando o *off_at* e alterando o estado para *handled*)

No *Firmware*, o estado local *sosActive* é alternado em conformidade. Esta variável é depois utilizada para controlar os *LEDs* e o *buzzer*, indicando visual e sonoramente se o dispositivo se encontra em situação de SOS ou em estado normal.

As figuras 7 e 8 apresentam, respetivamente os blocos de código que detetam a pressão do botão SOS e enviam o pedido correspondente para o *backend*:

```
350   // Detect the moment the button is pressed (not held)
351   // Avoids multiple triggers while the button is held down
352   if (btn == LOW && lastButtonState == HIGH) {
353     toggleSOS();
354     delay(250);   // debounce
355   }
```

Figura 7 - Deteção do clique no botão

```
270 // ============================================================================
271 // Sends an SOS request to the backend and updates the device's local alarm state.
272 // ============================================================================
273 void toggleSOS() {
274
275   // Build JSON payload
276   String body = "{\"device_id\":\"" + deviceId + "\"}";
277
278   int status;
279   String resp;
280
281   // Send SOS toggle request
282   Serial.println("Sending SOS toggle...");
283   httpPostJson(ENDPOINT_SOS, body, status, resp);
284
285   Serial.print("SOS Status: ");
286   Serial.println(status);
287   Serial.println("Response: " + resp);
288
289   // Flip local alarm state
290   sosActive = !sosActive;
291 }
```

Figura 8 - Função auxiliar alerta SOS (toggle)

3.1.3 Sincronização periódica com o backend

Em vez de esperar passivamente que o servidor envie informação para o Arduino (o que exigiria outro modelo de comunicação), a placa pergunta periodicamente ao *backend* qual é o estado atual do *help_event* associado ao seu *device_id* (se houver). Esta abordagem é mais simples de implementar sobre *HTTP* e adequada ao contexto do projeto.

A sincronização é feita através de um pedido *GET* ao *endpoint /help/state/<device_id>*. Para evitar que o Arduino fique bloqueado à espera de resposta, o código não utiliza *delays* prolongados, em vez disso, recorre à função *millis* para:

1. determinar quando é altura de iniciar um novo pedido;
2. medir há quanto tempo está à espera de resposta para poder declarar *timeout;*
3. permitir que, enquanto aguarda resposta, o dispositivo continue a executar as restantes tarefas.

Quando o *backend* responde, é analisado o corpo da resposta e atualiza a variável *helpActive* consoante o valor do campo "*help*". Se o pedido falhar ou atingir *timeout*, o *Firmware* descarta aquela tentativa e continuará a tentar no ciclo seguinte, garantindo robustez perante falhas ocasionais da rede doméstica ou do servidor.

As figuras seguintes apresentam o código responsável por consultar periodicamente o *backend* para obter o estado do *help_event*:

```
136  // ============================================================
137  // Gets help-state (blue LED state) from API
138  // ============================================================
139  bool httpGetHelpState(bool& helpValue) {
140
141    HttpClient client(wifiClient, API_HOST, API_PORT);
142
143    // Build URL: /help/state/<device_id>
144    String path = String(ENDPOINT_HELP_STATE) + deviceId;
145
146    client.get(path);
147
148    int status = client.responseStatusCode();
149    String resp = client.responseBody();
150
151    client.stop();
152
153    if (status != 200) {
154      return false;
155    }
156
157    // Detect if help (on the way) is true OR false
158    helpValue = resp.indexOf("\"help\":true") != -1;
159    return true;
160  }
```

Figura 9 - Função de obtenção do estado de resposta ao sinal SOS

```
// ============================================================================
// Checks the help-on-the-way status from the backend at regular interva
// ============================================================================
void checkHelpOnTheWayStatus() {
  unsigned long now = millis();

  // Start a new request if enough time has passed
  if (!waitingHelpResponse && (now - lastHelpCheck >= HELP_INTERVAL)) {
    lastHelpCheck = now;

    helpClient = new HttpClient(wifiClient, API_HOST, API_PORT);
    String path = String(ENDPOINT_HELP_STATE) + deviceId;

    helpClient->get(path);
    waitingHelpResponse = true;
    helpRequestStart = now;
    return;
  }
```

Figura 10 - Verificação periódica do estado help_event (início da função)

```
  // Handle an ongoing request
  if (waitingHelpResponse) {
    // Response available
    if (helpClient->available()) {
      String resp = helpClient->responseBody();
      bool newHelp = resp.indexOf("\"help\":true") != -1 || resp.indexOf("\"help\": true") != -1;

    if (newHelp != helpActive) {
        helpActive = newHelp;

        if (newHelp) {
            digitalWrite(BLUE_LED, HIGH);
            Serial.println("Help on the way!");
        } else {
            digitalWrite(BLUE_LED, LOW);
            Serial.println("Help cancelled!");
        }
    }


      helpClient->stop();
      delete helpClient;
      helpClient = nullptr;
      waitingHelpResponse = false;
      return;
    }

    // Request timeout
    if (now - helpRequestStart > HELP_TIMEOUT) {
      Serial.println("[HELP] Request TIMEOUT — no response");

      helpClient->stop();
      delete helpClient;
      helpClient = nullptr;
      waitingHelpResponse = false;
      return;
```

Figura 11 - Processamento da resposta help_event e gestão de timeout

Para além do *help_event*, o *Firmware* também monitoriza o estado da ligação *Wi-Fi* dentro do *loop*. Sempre que deteta que a ligação foi perdida, envia um pedido a indicar que o dispositivo está *offline*, tenta restabelecer a ligação e, após recuperar conectividade, volta a notificar o *backend* de que está *online*. Isto mantém a coerência entre o estado registado na base de dados e o estado real do dispositivo.

3.1.4 Sinalização local através de LEDs e buzzer

A forma como o dispositivo comunica estados ao utilizador localmente, sem depender do *frontend* é feita através de dois *LEDs* (vermelho e azul) e de um *buzzer*.

O comportamento foi definido de forma a distinguir claramente três situações:

1. SOS inativo, ajuda inativa (Estado normal);
2. SOS ativo, ainda sem resposta do cuidador;
3. SOS ativo, com ajuda a caminho.

Quando o sinal SOS é acionado, o *LED* vermelho pisca a um intervalo regular e o *buzzer* emite sons curtos, com pausas curtas entre eles. Este padrão transmite a ideia de alarme imediato. Quando, este alerta é respondido por outro utilizador, ou seja, ajuda está a caminho, o *LED* azul é ligado e o padrão sonoro passa a ter pausas mais longas, sinalizando ao utilizador que alguém já respondeu ao alerta.

Toda esta lógica é implementada recorrendo a medições e intervalos de tempo (*millis*) para alternar o estado do *LED* vermelho em intervalos fixos, iniciar e parar o *buzzer* e atualizar o estado do *LED* azul consoante a resposta ou cancelamento do respetivo utilizador.

A figura 12 apresenta o código responsável pelo comportamento dos *LEDs* e do *buzzer* nos diferentes estados do dispositivo, nomeadamente em estado SOS. Já as figuras 13 e 14, representam a verificação e/ou atualização do estado de resposta ao alerta, mesmo sendo por *timeout*.

```
358     static unsigned long lastBlink = 0;
359     // Faster beep when SOS only, slower beep when help is on the way
360     unsigned long blinkInterval = helpActive ? 1000 : 500;
361
362     static unsigned long lastBeep = 0;
363     static bool isBeeping = false;
364
365     unsigned long pauseTime  = helpActive ? 850 : 450; // pause duration
366     unsigned long beepTime   = 150;                    // short beep, always the same
367
368     if (sosActive) {
369         unsigned long now = millis();
370
371         if (!isBeeping) {
372             // Check if it's time to beep
373             if (now - lastBeep >= pauseTime) {
374                 tone(BUZZER_PIN, 500);  // short beep
375                 isBeeping = true;
376                 lastBeep = now;
377             }
378         } else {
379             // If currently beeping, turn off
380             if (now - lastBeep >= beepTime) {
381                 noTone(BUZZER_PIN);
382                 isBeeping = false;
383                 lastBeep = now;
384             }
385         }
386
387         // LED blinking logic stays unchanged
388         static unsigned long lastBlinkLED = 0;
389         unsigned long ledInterval = 500;
390         if (now - lastBlinkLED >= ledInterval) {
391             lastBlinkLED = now;
392             static bool ledState = false;
393             ledState = !ledState;
394             digitalWrite(RED_LED, ledState ? HIGH : LOW);
395         }
396
397         checkHelpOnTheWayStatus();
```

Figura 12 - Sinalização sonora e visual durante um alerta SOS.

```
// ================================================================================
// Checks the help-on-the-way status from the backend at regular intervals
// ================================================================================
void checkHelpOnTheWayStatus() {
  unsigned long now = millis();

  // Start a new request if enough time has passed
  if (!waitingHelpResponse && (now - lastHelpCheck >= HELP_INTERVAL)) {
    lastHelpCheck = now;

    helpClient = new HttpClient(wifiClient, API_HOST, API_PORT);
    String path = String(ENDPOINT_HELP_STATE) + deviceId;

    helpClient->get(path);
    waitingHelpResponse = true;
    helpRequestStart = now;
    return;
  }

```

Figura 13 - Verificação periódica do estado de ajuda

```
  // Handle an ongoing request
  if (waitingHelpResponse) {
    // Response available
    if (helpClient->available()) {
      String resp = helpClient->responseBody();
      bool newHelp = resp.indexOf("\"help\":true") != -1 || resp.indexOf("\"help\": true") != -1;

    if (newHelp != helpActive) {
        helpActive = newHelp;

        if (newHelp) {
            digitalWrite(BLUE_LED, HIGH);
            Serial.println("Help on the way!");
        } else {
            digitalWrite(BLUE_LED, LOW);
            Serial.println("Help cancelled!");
        }
    }


      helpClient->stop();
      delete helpClient;
      helpClient = nullptr;
      waitingHelpResponse = false;
      return;
    }

    // Request timeout
    if (now - helpRequestStart > HELP_TIMEOUT) {
      Serial.println("[HELP] Request TIMEOUT – no response");

      helpClient->stop();
      delete helpClient;
      helpClient = nullptr;
      waitingHelpResponse = false;
      return;
    }
  }
}
```

Figura 14 - Atualização do estado *help_event* e gestão de *timeout*

3.2 Backend

3.2.1 Base de Dados

A base de dados foi implementada no *Supabase* utilizando o seu serviço *PostgreSQL* integrado, recorrendo exclusivamente à *API REST* automática disponibilizada pela plataforma. Toda a aplicação comunica apenas através dos endpoints *REST* gerados pelo próprio *Supabase*, sem *queries SQL* diretas no *backend*.

O modelo, foi estruturado da seguinte forma:

- ***user*** – Guarda a identidade do utilizador (*ID*, *email*, nome, *password* encriptada).
- ***sos_device*** – Representa cada dispositivo físico associado a um utilizador, incluindo estado *online*/*offline*.
- ***sos_event*** – Regista cada evento SOS individual, com *timestamps* de início e fim e indicação de quem respondeu.
- ***help_event*** – Guarda as respostas dos cuidadores e é independente do SOS, permitindo vários cuidadores em paralelo.
- ***connection*** – Representa as ligações entre utilizadores.
- ***notification*** – Armazena todas as notificações enviadas pela *API*. Incluída para suportar um sistema mais completo de alertas, mas na versão atual não tem um papel central.

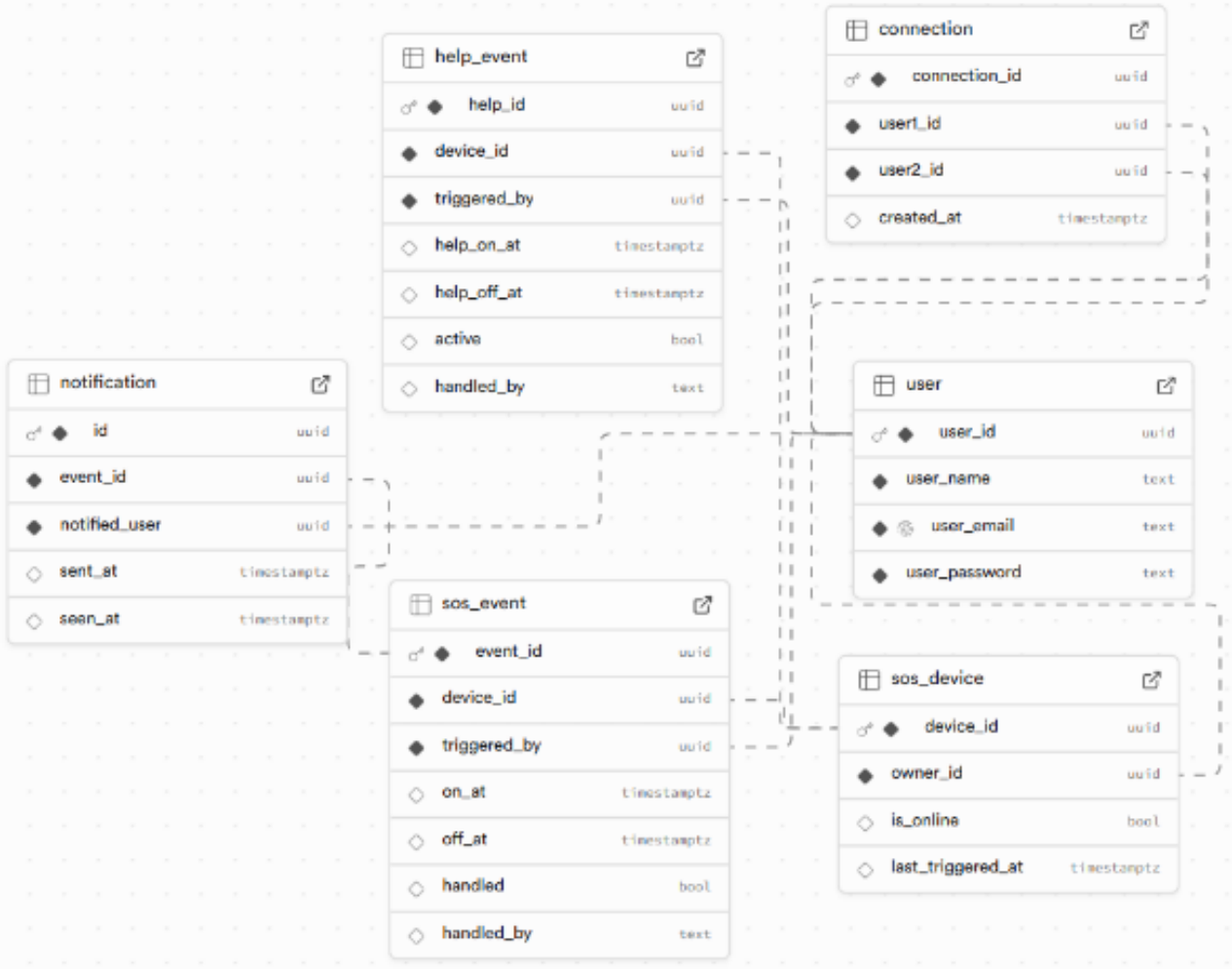


Figura 15 - Diagrama ERD

3.2.2 Arquitetura e configuração geral

O *backend* do *SafeSenior* foi organizada num único ficheiro, estruturada por blocos lógicos bem definidos. *API* valida quem está a chamar, aplica a regra de negócio necessária e comunica com a base de dados, devolvendo sempre respostas em JSON.

A estrutura do ficheiro segue uma ordem consistente:

- Configuração inicial da aplicação, carregamento de variáveis de ambiente e definição dos *URLs* base para acesso às tabelas do *Supabase*.
- Definição de constantes de apoio, como códigos de estado *HTTP* e a função que centraliza os cabeçalhos necessários em todas as chamadas à BD (*supabase_headers*).
- Decoradores de autenticação (*auth_user* e *auth_device*), responsáveis por validar *tokens JWT*, distinguir entre pedidos de utilizadores e de dispositivos e anexar o *user_id* correto ao contexto do pedido.
- Agrupamento de *endpoints* por responsabilidade como gestão de utilizadores e ligações, gestão de dispositivos, gestão de eventos SOS, gestão de eventos de ajuda e gestão de notificações.

```python
from flask import Flask, jsonify, request
from functools import wraps
from datetime import datetime, timezone, timedelta
import os, jwt, requests
import uuid
from dotenv import load_dotenv
import os
import hashlib

# load variables from .env into environment
load_dotenv()

app = Flask(__name__)
app.config['SECRET_KEY'] = os.environ.get("SECRET_KEY", "default_secret_key")

# Supabase handles SQL execution automatically from REST queries
SUPABASE_KEY = os.getenv("SUPABASE_KEY")
SUPABASE_URL = os.environ.get("SUPABASE_URL")

# Base REST paths
SUPABASE_REST_URL = f"{SUPABASE_URL}/rest/v1"
USER_URL = f"{SUPABASE_REST_URL}/user"
CONNECTION_URL = f"{SUPABASE_REST_URL}/connection"
DEVICE_URL = f"{SUPABASE_REST_URL}/sos_device"
EVENT_URL = f"{SUPABASE_REST_URL}/sos_event"
HELP_URL = f"{SUPABASE_REST_URL}/help_event"
NOTIF_URL = f"{SUPABASE_REST_URL}/notification"

# HTTP Status Codes
OK = 200
CREATED = 201
BAD_REQUEST = 400
UNAUTHORIZED= 401
FORBIDDEN = 403
NOT_FOUND = 404
CONFLICT= 409
SERVER_ERROR = 500
```

Figura 16 - Configuração inicial da API

Em termos de funcionamento, o fluxo típico de um pedido é sempre semelhante. O cliente (aplicação *Android* ou dispositivo) envia um pedido *HTTP* para um *endpoint* específico, acompanhado de um *token* ou de um *device_id*. O decorador correspondente valida esse contexto, o *endpoint* constrói a operação

pretendida e comunica com a tabela adequada no *Supabase* através de pedidos *GET*, *POST*, *PATCH* ou *DELETE*. No final, a *API* devolve uma resposta *JSON* simples, que o *frontend* ou o *firmware* conseguem interpretar sem lógica adicional.

Esta organização mantém o *backend* compacto, legível e fácil de estender. A adição de novas funcionalidades resume-se, na prática, à criação de novos *endpoints* que seguem o mesmo padrão de validação, chamada ao *Supabase* e resposta *JSON*, sem necessidade de alterar a arquitetura base.

```
39  ################################### Supabase Request Headers ###################################
40  def supabase_headers():
41      return {
42          "apikey": SUPABASE_KEY,
43          "Authorization": f"Bearer {SUPABASE_KEY}",
44          "Content-Type": "application/json",
45          "Prefer": "return=representation"
46      }
47
48  @app.route('/', methods=['GET'])
49  def home():···
51
52  #|-------------------------------------------------------------------------------------------|
53  #|                                    AUTH DECORATORS                                        |
54  #|-------------------------------------------------------------------------------------------|
55  # ================================== USER ==================================
56  def auth_user(f):···
89
90
91  # ================================== SOS DEVICE ==================================
92  def auth_device(f):···
114
115 #|-------------------------------------------------------------------------------------------|
116 #|                                    USER ENDPOINTS                                         |
117 #|-------------------------------------------------------------------------------------------|
```

Figura 17 - Estrutura Geral da API

3.2.3 Gestão de Utilizadores e Ligações

A *API* disponibiliza um conjunto de *endpoints* dedicados à criação e autenticação de utilizadores, bem como à definição das relações entre eles. Cada utilizador pode estabelecer relações bidirecionais com outros utilizadores, permitindo que recebam e respondam aos seus alertas SOS.

O processo de criação de conta é simples. A *API* recebe o nome, *email* e *password*, valida se o *email* já existe e guarda a *password* encriptada com *SHA-256* antes de inserir o registo na tabela *user*. O *login* segue o mesmo princípio de simplicidade, compara o *hash* da *password* fornecida com o valor guardado na base de dados e, em caso de sucesso, gera um token *JWT* associado ao *user_id*.

As ligações entre utilizadores são registadas na tabela *connection*. Cada relação é representada por um par (*user1_id*, *user2_id*) e permite que ambos os lados sejam notificados quando ocorre um evento SOS.

A API verifica automaticamente que um utilizador não crie relações consigo próprio e que relações duplicadas não sejam inseridas. A listagem de relações devolve também informação adicional, incluindo o *device_id* do utilizador ligado e o último SOS registado, para que o *frontend* apresente dados relevantes sem múltiplos pedidos ao *backend*.

```python
# ================================ LOGIN ======================================
@app.route("/login", methods=["POST"])
def login_user():
    """Login a user"""

    content = request.get_json()
    if "user_email" not in content or "user_password" not in content:
            return jsonify({"message": "Missing credentials"}), BAD_REQUEST

    res = requests.get(USER_URL, headers=supabase_headers(),
                       params={"user_email": f"eq.{content['user_email'].lower()}"}).json()
    if not res:
        return jsonify({"message": "Invalid credentials"}), UNAUTHORIZED

    user = res[0]
    encrypted_pass = hashlib.sha256(content["user_password"].encode()).hexdigest()
    if encrypted_pass != user["user_password"]:
        return jsonify({"message": "Invalid credentials"}), UNAUTHORIZED

    token = jwt.encode(
        {"id": user["user_id"], "email": user["user_email"],
        "exp":  datetime.now(timezone.utc) + timedelta(hours=8)},
        app.config["SECRET_KEY"], algorithm="HS256"
    )
    return jsonify({"token": token, "userId": user["user_id"]}), OK
```

Figura 18 - Função *login* (*backend*)

```python
# ============================== CREATE CONNECTION ================================
@app.route("/connections", methods=["POST"])
@auth_user
def add_connection():
    """Create a two ended connection between two users"""

    content = request.get_json()
    if "other_user_id" not in content:
        return jsonify({"message": "Missing other_user_id"}), BAD_REQUEST

    if content["other_user_id"] == request.user_id:
        return jsonify({"message": "User cannot connect to themselves"}), BAD_REQUEST

    payload = {
        "user1_id": request.user_id,
        "user2_id": content["other_user_id"]
    }

    res = requests.post(CONNECTION_URL, headers=supabase_headers(), json=payload)
    if res.status_code == CONFLICT:
        return jsonify({"message": "Connection already exists"}), CONFLICT

    return jsonify({"message": "Connection created"}), CREATED
```

Figura 19 - Função de criação de relações

3.2.4 Gestão de Dispositivos

Cada utilizador pode associar um ou mais dispositivos SOS à sua conta e é através destes registos que a *API* identifica e valida as ações vindas destes. A tabela *sos_device* representa cada dispositivo físico, contendo o *device_id*, o *owner_id* e o seu estado atual (*online*/*offline*).

O *endpoint* de registo gera um identificador único (UUID) no servidor e associa-o ao utilizador autenticado. Este valor é depois inserido na tabela *sos_device* e transmitido ao *Firmware*, que o guarda na *EEPROM* e passa a utilizá-lo em todas as comunicações seguintes.

Para permitir que o dispositivo envie pedidos autenticados, existe um *endpoint* de *device login* que gera um *token JWT* contendo o *device_id*. Esse *token* não identifica diretamente o proprietário, mas permite ao decorador *auth_device* recuperar automaticamente o *owner_id* a partir da base de dados sempre que o dispositivo comunica com o backend.

A *API* mantém também o estado *online/offline* do dispositivo. Sempre que o *Firmware* envia um pedido para */device/online* ou */device/offline*, a *API* atualiza o registo correspondente no *Supabase*. Esta informação é relevante para o *frontend*, que pode indicar ao cuidador se o dispositivo está operacional e a comunicar corretamente.

```python
# ================================ REGISTER ================================
@app.route("/devices", methods=["POST"])
@auth_user
def register_device():
    """Register a new SOS device and set up its ID"""

    device_id = str(uuid.uuid4())
    data = {
        "device_id": device_id,
        "owner_id": request.user_id,
        "is_online": False,
        "last_triggered_at": None
    }

    res = requests.post(DEVICE_URL, headers=supabase_headers(), json=data)
    if res.status_code in (OK, CREATED):
        # Return the new ID
        return jsonify({
            "message": "Device registered",
            "device_id": device_id,
            "owner_id": request.user_id
        }), CREATED
    else:
        return jsonify({"message": "Failed to register device"}), SERVER_ERROR

# ================================ DEVICE LOGIN ================================
@app.route("/devices/login", methods=["POST"])
def device_login():
    """Return a temporary JWT for a registered device"""

    content = request.get_json()
    if "device_id" not in content:
        return jsonify({"message": "Missing device_id"}), BAD_REQUEST

    device_id = content["device_id"]

    # Check if device exists
    res = requests.get(f"{DEVICE_URL}?device_id=eq.{device_id}", headers=supabase_headers())
    data = res.json()
    if not data:
        return jsonify({"message": "Device not found"}), NOT_FOUND

    # Generate token
    token = jwt.encode(
        {"device_id": device_id, "exp": datetime.now(timezone.utc) + timedelta(hours=8)},
        app.config["SECRET_KEY"], algorithm="HS256"
    )
    return jsonify({"token": token}), OK
```

Figura 20 - Registo e login de dispositivos na API, respetivamente

3.2.5 Gestão de alertas SOS

A gestão dos alertas SOS é um dos núcleos do *backend*. O mesmo *endpoint* cria um novo evento SOS se não existir nenhum ativo ou encerra o evento existente quando já está em curso. Esta abordagem simplifica a comunicação com o dispositivo, que apenas envia um sinal genérico de SOS, sem precisar de distinguir entre início e fim.

O *backend* identifica automaticamente a origem do pedido. Se o alerta for acionado pelo dispositivo, o decorador *auth_device* associa o *device_id* ao *owner_id* correspondente. Se o alerta for enviado a partir da aplicação móvel, o decorador *auth_user* identifica o utilizador autenticado. Em ambos os casos, o *backend* garante que apenas um SOS pode estar ativo por utilizador, evitando duplicações ou estados inconsistentes.

Para iniciar um SOS, a *API* cria um novo registo na tabela *sos_event* com o *timestamp* de início e marca o dispositivo como *online*. Caso exista já um registo ativo (*handled = false*), o backend interpreta o pedido como um encerramento, atualizando o *timestamp* de fim e assinalando o evento como tratado. A lógica garante também que, ao terminar um SOS, qualquer *help_event* ativo associado ao dispositivo é imediatamente desativado para manter coerência entre os estados.

Os *endpoints* complementares permitem consultar histórico e estados atuais. A listagem de eventos devolve todos os SOS emitidos por um utilizador, ordenados cronologicamente, enquanto o *endpoint* de ativos devolve apenas os utilizadores que têm um SOS em curso, útil para o frontend identificar situações urgentes.

```
403  # ================================== TOGGLE SOS EVENT ==================================
404  @app.route("/sos", methods=["POST"])
405  def toggle_sos():
406      """Toggle SOS state from app or device."""
407
408      # Determine source (frontend (JWT) or DEVICE (device_id))
409      auth_header = request.headers.get("Authorization")
410
411      # If header (token) exists, then frontend
412      if auth_header:
413          # SOS triggered from frontend
414          result = auth_user(lambda: None)()
415          if result:
416              return result
417          user_id = request.user_id
418          device_id = None
419      # SOS alert comes from a device
420      else:
421          # SOS triggered from device
422          result = auth_device(lambda: None)()
423          if result:
424              return result
425          user_id = request.user_id
426          device_id = request.device_id
427
428      now = datetime.now(timezone.utc).isoformat()
429
430      # Check for active SOS events
431      active_url = f"{EVENT_URL}?triggered_by=eq.{user_id}&handled=eq.false"
432      active_res = requests.get(active_url, headers=supabase_headers())
433      active_events = active_res.json()
```

Figura 21 - Trecho da função gestora de eventos SOS (*toggle*)

3.2.6 Gestão de Respostas

As respostas aos alertas SOS são tratadas através da tabela *help_event*, permitindo que cuidadores independentes indiquem que vão responder a um alerta. Cada resposta funciona também como um *toggle*, isto é, um cuidador ativa a ajuda e, ao clicar novamente, cancela a indicação de que está a caminho. Esta abordagem simplifica o uso na aplicação móvel e evita duplicação de eventos.

Cada *help_event* está associado ao *device_id* que ativou o alerta, não a um utilizador específico. Isto permite que vários cuidadores respondam em paralelo ao mesmo alerta SOS, sem sobrepor registos ou impedir novas respostas. Quando um cuidador responde ao alerta, a *API* cria um novo registo com o *timestamp* de início e marca o evento como ativo. Quando cancela, o evento é encerrado com *timestamp* de fim, mantendo o histórico completo.

O *backend* verifica sempre se o dispositivo tem um SOS ativo. Quando a resposta é criada, o *backend* atualiza o respetivo registo em *sos_event*, indicando quem foi o primeiro cuidador a responder. Esta ligação é importante para a aplicação móvel, que utiliza esta informação para informar o utilizador sobre quem assumiu responsabilidade pela situação.

O *endpoint* de leitura *(/help/state/<device_id>)* devolve apenas o estado atual (*help=true/false*), permitindo ao *Firmware* adaptar o padrão dos *LEDs* e do buzzer sem necessitar de interpretar detalhes adicionais. Esta simplicidade garante um comportamento consistente entre *backend* e dispositivo.

3.3 Frontend

3.3.1 Estrutura geral

A aplicação *Android* foi organizada de forma modular, separando a interface, a lógica de apresentação e a comunicação com o *backend*. Esta organização permite manter o código claro, escalável e alinhado com os fluxos definidos pela *API*.

A camada de interface é composta pelas *Activities* principais, responsáveis por gerir cada ecrã da aplicação, autenticação, visualização de contactos, visualização de eventos e interação com o botão de SOS. Cada *Activity* tem responsabilidades bem definidas, assegurando que apenas coordena a interação do utilizador e os pedidos enviados ao *backend*.

A camada de comunicação está concentrada no diretório *network*, que inclui o cliente *Retrofit*, a interface dos *endpoints* e uma classe dedicada à gestão do *token* de sessão. Esta separação permite que a lógica de rede permaneça isolada, facilitando a manutenção e garantindo que todas as chamadas seguem o mesmo formato e tratamento de erros.

Os *adapters* asseguram a ligação entre dados e componentes visuais, nomeadamente as listas de utilizadores ligados e o histórico de eventos SOS. A existência de *models* específicos para cada tipo de resposta da *API* garante que os dados são tratados de forma tipada e consistente ao longo da aplicação.

Na figura 22, pode ser observada esta estruturação.

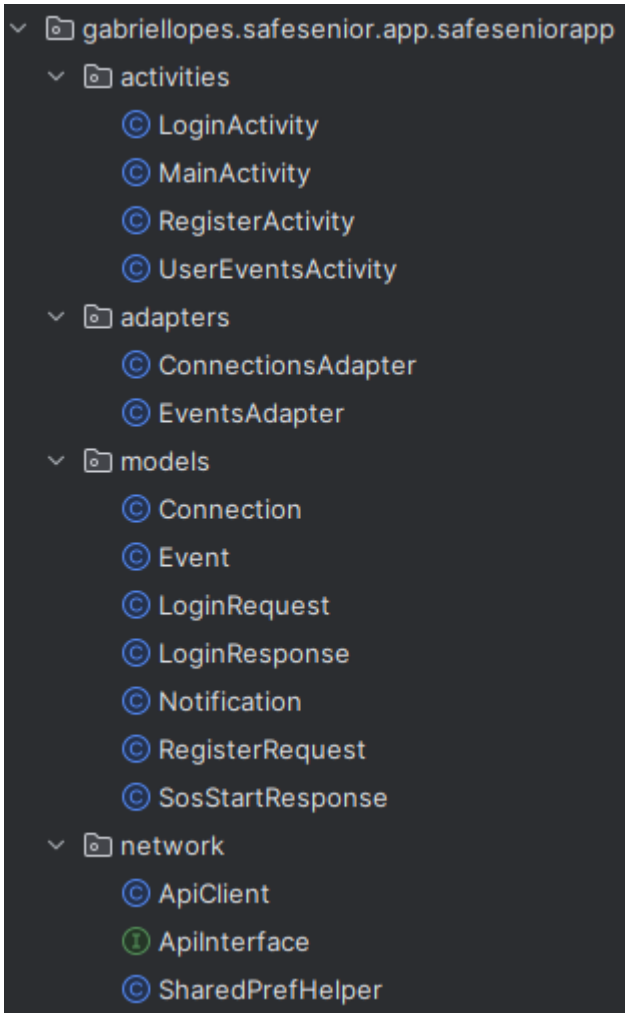


Figura 22 - Estrutura geral aplicação android

3.3.2 Comunicação (Retrofit + Shared Preferences)

A comunicação com o *backend* é efetuada através de *Retrofit*, que permite estruturar todos os pedidos de rede de forma tipada e consistente. Através deste mecanismo, a aplicação envia e recebe informação relativa à autenticação, contactos, eventos SOS e notificações, mantendo uma interface simples e organizada.

O ficheiro *ApiClient* (figura 23) concentra a configuração do *Retrofit*, incluindo a definição do *URL* base e a criação da instância utilizada no resto da aplicação. A classe *ApiInterface (*figura 24*)* descreve cada *endpoint* disponibilizado pelo *backend*, permitindo que as *Activities* invoquem funções diretas em vez de construírem manualmente pedidos *HTTP* ou realizarem processos de *serialization*.

A classe *SharedPrefHelper* (figura 25) assegura a gestão do *token JWT*, responsável por manter a sessão ativa. O *token* é recuperado quando necessário e aplicado automaticamente no cabeçalho *Authorization*, garantindo autenticação transparente durante o acesso às operações protegidas do *backend*.

Esta abordagem centralizada garante uniformidade, reduz código repetido e facilita a extensão futura da aplicação, permitindo acrescentar novos *endpoints* ou funcionalidades sem alterar a estrutura principal.

```
public class ApiClient {  8 usages
    private static Retrofit retrofit;  3 usages
    private static final String BASE_URL = "https://safe-senior-njhw.vercel.app/";

    public static Retrofit getClient() {  4 usages
        if (retrofit == null) {
            retrofit = new Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(GsonConverterFactory.create())
                    .build();
        }
        return retrofit;
    }
```

Figura 23 - Classe ApiClient

```
public interface ApiInterface {  12 usages

    // User
    @POST("login")  1 usage
    Call<LoginResponse> login(@Body LoginRequest body);
```

Figura 24 - Exemplo de pedido HTTP da classe ApiInterface

```
public SharedPrefHelper(Context context) {  3 usages
    prefs = context.getSharedPreferences(PREFS_NAME, Context.MODE_PRIVATE);
}

public void saveAuth(String token, String userId) {  1 usage
    prefs.edit().putString(TOKEN_KEY, token)
            .putString(USER_ID_KEY, userId)
            .apply();
}

public String getToken() { return prefs.getString(TOKEN_KEY,  s1: null); }
```

Figura 25 - Bloco de código da classe *SharedPrefHelper*

3.3.3 *Activities* Principais

3.3.3.1 LoginActivity

A *LoginActivity* constitui o ponto de entrada da aplicação. Permite ao utilizador introduzir as suas credenciais, que serão validados ao comunicar com a *API*, através de *Retrofit*, para obter um *token JWT*. Caso a autenticação seja bem-sucedida, o *token* é guardado pela classe *SharedPrefHelper* e o utilizador é redirecionado para a *MainActivity*. Este processo assegura, assim, uma utilização simples e direta, apresentando apenas os campos essenciais e mantendo o foco na entrada segura no sistema.

```
private void login() {  1 usage
    String email = emailInput.getText().toString().trim();
    String password = passwordInput.getText().toString().trim();

    if (email.isEmpty() || password.isEmpty()) {
        Toast.makeText( context: this,  text: "Fill in all fields", Toast.LENGTH_SHORT).show();
        return;
    }

    // Perform login request
    api.login(new LoginRequest(email, password)).enqueue(new Callback<LoginResponse>() {
        @Override
        public void onResponse(Call<LoginResponse> call, Response<LoginResponse> response) {
            if (response.isSuccessful() && response.body() != null) {
                // Save token and userId locally
                prefHelper.saveAuth(response.body().token, response.body().userId);

                startActivity(new Intent( packageContext: LoginActivity.this, MainActivity.class));
                finish();
            } else {
                Toast.makeText( context: LoginActivity.this,  text: "Invalid credentials", Toast.LEN
            }
        }

        @Override
        public void onFailure(Call<LoginResponse> call, Throwable t) {
            Toast.makeText( context: LoginActivity.this,  text: "Network error: " + t.getMessage(),
        }
```

Figura 26 - Lógica de *login* (*frontend*)

3.3.3.2 MainActivity

A *MainActivity* funciona como um painel em tempo quase real do estado da rede de cuidadores. Para isso, combina três blocos principais de lógica, carregamento inicial das ligações, deteção de SOS ativos e tratamento de notificações recebidas. Estes comportamentos estão concentrados sobretudo nos métodos *loadConnections(), loadActiveSOS()* e *loadNotifications().*

3.3.3.2.1 Método *loadConnections()*

O carregamento das ligações é o primeiro passo da *MainActivity* e define a informação que será apresentada no painel. A atividade solicita ao *backend* a lista de contactos associados ao utilizador e, quando a resposta é válida, prepara a estrutura necessária para apresentar esses dados e permitir a interação com cada contacto. Neste processo, são também configuradas as ações associadas a cada elemento da lista, nomeadamente a consulta do histórico de SOS e a possibilidade de sinalizar ajuda para um dispositivo.

Depois de concluir esta preparação, a atividade desencadeia automaticamente os mecanismos responsáveis por atualizar o estado dos SOS ativos, verificar notificações recentes e manter o painel sincronizado com o *backend* sem intervenção do utilizador. Deste modo, a informação apresentada permanece atualizada ao longo do tempo.

```
api.getConnections(token).enqueue(new Callback<List<Connection>>() {
    @Override
    public void onResponse(@NonNull Call<List<Connection>> call, @NonNull Response<List<Connection>> response) {
        // If connections exist, build adapter
        if (response.isSuccessful() && response.body() != null && !response.body().isEmpty()) {
            // Attach adapter and load initial dashboard data
            adapter = new ConnectionsAdapter(response.body(), new ConnectionsAdapter.OnConnectionClickListener() {
                @Override 1 usage
                public void onConnectionClick(Connection connection) {
                    // Open user SOS events history
                    Intent i = new Intent( packageContext: MainActivity.this, UserEventsActivity.class);
                    i.putExtra( name: "email", connection.user_email);
                    i.putExtra( name: "name", connection.user_name);
                    startActivity(i);
                }
```

Figura 27 - Apresentação dos contactos

```
    @Override 1 usage
    public void onSendMessageClick(Connection c) {
        // Help event requires the user device's ID
        if (c.device_id == null || c.device_id.isEmpty()) {
            Toast.makeText( context: MainActivity.this, text: "No device ID for this user", Toast.LENGTH_SHORT).show(
            return;
        }

        JsonObject body = new JsonObject();
        body.addProperty( property: "device_id", c.device_id);
        // Toggle help event
        api.toggleHelp(token, body).enqueue(new Callback<Void>() {
            @Override
            public void onResponse(Call<Void> call, Response<Void> response) {

            }

            @Override
            public void onFailure(Call<Void> call, Throwable t) {
                Toast.makeText( context: MainActivity.this, text: "Failed to send help", Toast.LENGTH_SHORT).show();
            }
        });
    }
});
recyclerView.setAdapter(adapter);
loadActiveSOS();
loadNotifications();
startActiveSOSAutoRefresh();
```

Figura 28 – *listener* de botão de resposta

3.3.3.2.2 loadActiveSOS()

A atualização do estado dos SOS ativos é feita recorrendo a uma consulta periódica ao *backend*. A atividade solicita a lista de utilizadores que se encontram em situação de emergência e, sempre que a resposta é válida, transmite essa informação ao componente responsável pela apresentação dos contactos. O objetivo é garantir que a lista reflete, em cada momento, quem se encontra com um alerta ativo.

Esta verificação é executada inicialmente após o carregamento das ligações e continua a ser repetida em intervalos regulares, assegurando que alterações do estado dos contactos são detetadas e refletidas automaticamente no painel.

```
// Refresh dashboard SOS state
private void loadActiveSOS() {  3 usages
    String token = prefHelper.getToken();
    if (token == null || adapter == null)
        return;
    // Get the list of users who currently have an active SOS from API
    api.getActiveSosUsers(token).enqueue(new Callback<List<Connection>>() {
        @Override
        public void onResponse(Call<List<Connection>> call, Response<List<Connection>> response) {
            if (response.isSuccessful() && response.body() != null) {
                adapter.setActiveSOSUsers(response.body());
            }
        }

        @Override
        public void onFailure(Call<List<Connection>> call, Throwable t) {}
    });
}
```

Figura 29 - Carregamento de alertas SOS

3.3.3.2.3 loadNotifications()

No escopo atual do projeto, *loadNotifications()* identifica qual dos contactos acionou um alerta SOS e destaca essa informação na interface. O objetivo imediato é apenas garantir que o utilizador perceba a qual dos seus contactos acabou de ativar um SOS, tirando partido do sistema de notificações sem introduzir complexidade adicional.

```
// Display unseen notifications
private void loadNotifications() {  1 usage
    String token = prefHelper.getToken();
    if (token == null) return;

    api.getNotifications(token).enqueue(new Callback<List<Notification>>() {
        @Override
        public void onResponse(@NonNull Call<List<Notification>> call,
                               @NonNull Response<List<Notification>> response) {
            if (response.isSuccessful() && response.body() != null) {
                // highlight unseen SOS alerts
                for (Notification n : response.body()) {
                    if (n.seen_at == null && n.trigger_name != null) {
                        Toast.makeText( context: MainActivity.this, text: n.trigger_name + " triggered an SOS!",
                                Toast.LENGTH_LONG).show();
                        if (adapter != null && n.trigger_email != null) {
                            adapter.highlightUserByEmail(n.trigger_email);
                        }
                    }
                }
            }
        }

        @Override
        public void onFailure(@NonNull Call<List<Notification>> call, @NonNull Throwable t) {}
    });
}
```

Figura 30 - Notificação visual de alerta SOS

3.3.3.3 UserEventsActivity

A *UserEventsActivity* é responsável por consultar o histórico de SOS de um utilizador específico. Quando é aberta, a atividade recebe o *e-mail* do contacto selecionado e utiliza-o para solicitar ao *backend* todos os eventos associados a esse utilizador. O pedido é autenticado com o *token* guardado no dispositivo e devolve uma lista com todos os registos existentes.

Depois de receber os dados, a atividade limita-se a entregá-los diretamente ao adaptador correspondente, que trata da organização e apresentação dos eventos. No fundo, o seu papel é apenas garantir que o pedido é executado corretamente, que as respostas são válidas e que o histórico obtido corresponde integralmente ao que se encontra registado no *backend*. Caso o utilizador não tenha eventos, ou ocorra uma falha durante a comunicação, a atividade interrompe o fluxo e assinala a situação ao utilizador.

```
// Load User SOS events
private void loadEvents() {  1 usage
    String token = prefHelper.getToken();
    if (token == null || selectedUserEmail == null) {
        Toast.makeText( context: this,  text: "Missing email", Toast.LENGTH_SHORT).show();
        return;
    }

    api.getEvents(token, selectedUserEmail).enqueue(new Callback<List<Event>>() {
        @Override
        public void onResponse(Call<List<Event>> call, Response<List<Event>> response) {
            if (response.isSuccessful() && response.body() != null) {
                // Bind retrieved events to the RecyclerView
                adapter = new EventsAdapter(response.body());
                recyclerEvents.setAdapter(adapter);
            } else {
                Toast.makeText( context: UserEventsActivity.this,  text: "No events found", Toast.LEN
            }
        }

        @Override
        public void onFailure(Call<List<Event>> call, Throwable t) {
            Toast.makeText( context: UserEventsActivity.this,  text: "Network error: " + t.getMessage
        }
    });
```

Figura 31 - Carregamento do histórico de eventos

3.3.4 *Adapters* e Representação de Dados

A aplicação recorre a *Adapters* para transformar a informação obtida do *backend* em elementos estruturados que podem ser apresentados na interface. Estes componentes funcionam como intermediários entre os dados recebidos das *API calls* e as listas exibidas ao utilizador, mantendo a separação entre lógica de rede e lógica de apresentação.

3.3.4.1.1 ConnectionsAdapter

Este adaptador recebe a lista de contactos devolvida pelo *backend* e organiza-a de forma consistente para ser utilizada pela *RecyclerView* da atividade principal. Para além de representar cada ligação, o adaptador incorpora a lógica necessária para reagir a ações do utilizador, como a abertura do histórico de SOS ou o envio de um pedido de ajuda. Sempre que o *backend* indica que um contacto tem um SOS ativo, o

adaptador atualiza diretamente o estado correspondente, permitindo ao painel refletir a situação em tempo quase real.

Considerando a natureza dinâmica da informação, o adaptador disponibiliza ainda um método responsável por atualizar apenas o conjunto de contactos que se encontram em situação de emergência. Esta abordagem evita reconstruções desnecessárias e garante que a interface se mantém fluida mesmo quando os pedidos ao *backend* são frequentes.

```
// List of all connections displayed in the RecyclerView
private final List<Connection> connections;  4 usages
// Handle actions when the user clicks anything
private final OnConnectionClickListener listener;  3 usages
// Stores the emails of users who currently have an active SOS
private final List<String> activeSosEmails = new ArrayList<>();  5 usages

// Receives connection list
public ConnectionsAdapter(List<Connection> connections, OnConnectionClickListener listener) {
    this.connections = connections;
    this.listener = listener;
}

// Holds references to each row's UI components
public static class ViewHolder extends RecyclerView.ViewHolder {  4 usages
    TextView name, email, lastSos;  2 usages
    android.widget.Button btnMessage;  8 usages

    public ViewHolder(@NonNull View itemView) {  1 usage
        super(itemView);
        name = itemView.findViewById(R.id.txtName);
        email = itemView.findViewById(R.id.txtEmail);
        lastSos = itemView.findViewById(R.id.txtLastSos);
        btnMessage = itemView.findViewById(R.id.btnMessage);
    }
}

// Create a new inflated row layout
@NonNull
@Override
public ConnectionsAdapter.ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType)
    View view = LayoutInflater.from(parent.getContext())
            .inflate(R.layout.item_connection, parent,  attachToRoot: false);
    return new ViewHolder(view);
}
```

Figura 32 - Apresentação de contactos

3.3.4.1.2 EventsAdapter

O *EventsAdapter* tem como função representar o histórico de SOS de um utilizador. Recebe a lista de registos devolvida pelo *backend* e converte-a num conjunto ordenado de elementos que podem ser apresentados de forma clara. Não existe qualquer transformação complexa dos dados, o adaptador limita-

se a expor cada evento tal como é recebido, respeitando os valores devolvidos pela *API*, incluindo datas, estados e informações complementares.

A simplicidade deste adaptador reflete o propósito da *UserEventsActivity*, cujo objetivo é permitir a consulta íntegra do histórico sem adicionar lógica extra.

```java
// Inflate the layout for a single event row
@NonNull
@Override
public ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
    View v = LayoutInflater.from(parent.getContext())
            .inflate(R.layout.item_event, parent, attachToRoot: false);
    return new ViewHolder(v);
}

// Bind each event's data into the row views
@Override
public void onBindViewHolder(@NonNull ViewHolder holder, int position) {
    Event e = events.get(position);

    // Show event ID
    holder.txtEventId.setText("Event: " + e.event_id);

    // Format item display
    if (e.on_at != null) holder.txtOnAt.setText("Started: " + formatDate(e.on_at));
    else holder.txtOnAt.setText("Started: -");
    if (e.off_at != null) holder.txtOffAt.setText("Stopped: " + formatDate(e.off_at));
    else holder.txtOffAt.setText("Stopped: Active");

    // Status text
    if (!e.handled) {
        holder.txtHandled.setText("Status: Active");
        holder.txtHandled.setTextColor(android.graphics.Color.RED);
    } else if (e.handled && e.handled_by != null && !e.handled_by.isEmpty()) {
        holder.txtHandled.setText("Status: Responded by " + e.handled_by);
        holder.txtHandled.setTextColor(android.graphics.Color.parseColor( colorString: "#2E7D32"));
    } else {
        holder.txtHandled.setText("Status: Stopped by User (No response)");
        holder.txtHandled.setTextColor(android.graphics.Color.DKGRAY);
    }
```

Figura 33 - Apresentação do histórico de eventos

3.3.4.1.3 Representação dos Dados

As classes que representam os modelos (*Connection*, *Event*, *Notification* e restantes estruturas de pedido e resposta) funcionam como espelhos diretos das entidades existentes no *backend*. A aplicação não reinterpreta nem reestrutura estes dados; apenas os armazena tal como são obtidos. Esta decisão reduz complexidade, evita inconsistências e garante total alinhamento entre o que é devolvido pela *API* e o que é apresentado na aplicação.

Todas as estruturas são minimalistas, contendo apenas os campos estritamente necessários para identificar utilizadores, eventos ou notificações. Este modelo facilita a manutenção e reduz o risco de divergência entre versões do *backend* e do *frontend*.

4 Resultados

Depois de desenvolvidos o *Firmware*, a API e a aplicação móvel, foi necessário validar o funcionamento integrado do sistema em três pontos complementares. O comportamento físico do dispositivo, a comunicação em tempo real vista no *Serial Monitor* e a reação do *frontend* a cada evento.

Estas três perspetivas permitem confirmar que toda a arquitetura definida nas secções anteriores se traduz num sistema funcional, coerente e estável. A seguir apresentam-se os registos visuais recolhidos durante os testes finais.

4.1 Hardware

As imagens, 34 e 35, mostram o dispositivo em funcionamento, incluindo a montagem em *breadboard*, a ligação entre *o Arduino* e os componentes externos e os três estados principais do sistema (normal, SOS ativo e SOS com ajuda confirmada). Estes registos permitem verificar que o sistema é controlado corretamente garantindo que o utilizador tem sempre *feedback* visual e sonoro sobre o estado atual.

No contexto de testes e validação, o hardware foi sujeito a vários ciclos de SOS consecutivos, diferentes períodos de inatividade e simulações de perda de Wi-Fi. O objetivo foi confirmar que o dispositivo se mantém estável mesmo com múltiplas transições de estado e que retoma o funcionamento normal após falhas momentâneas. Todos os testes mostraram que o sistema responde de forma consistente, sem bloqueios, sem reinícios inesperados e com níveis de latência impercetíveis ao utilizador final.

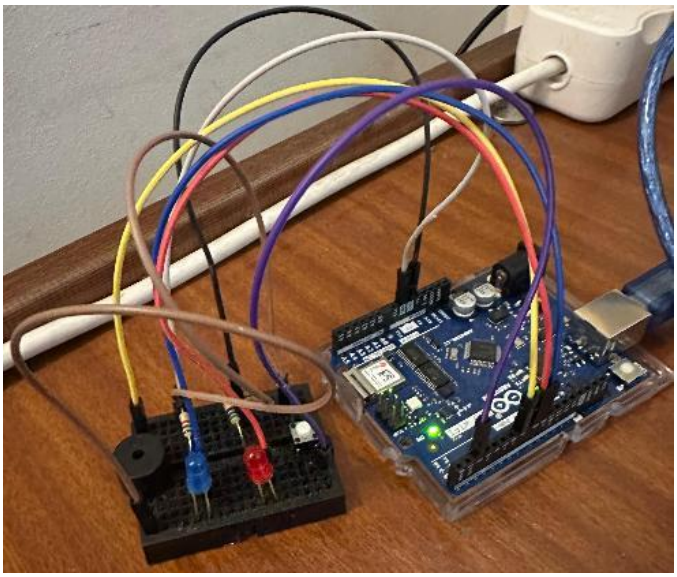


Figura 34 - Circuito completo, estado normal

Figura 35 – Sinalização dos estados SOS (vermelho para SOS ativo e azul para ajuda confirmada).

4.2 Serial Monitor

O *Serial Monitor* regista a comunicação direta entre o *Firmware* e a *API*, permitindo observar cada passo do ciclo de operação, ligação Wi-Fi, envio de estado, emissão de SOS, receção da resposta do *backend* e deteção do sinal de ajuda. Estes registos confirmam a correta interpretação das respostas da *API* e que a comunicação segue sempre os protocolos definidos.

Nos testes de validação, analisaram-se tempos de resposta, estabilidade da comunicação e coerência dos estados recebidos. Foram repetidos vários cenários, incluindo SOS sucessivos, alternância rápida entre responder/cancelar e períodos de espera prolongada. Os resultados mostraram que a *API* mantém a consistência dos estados e que o dispositivo processa todas as respostas sem perda de mensagens, permitindo concluir que o canal de comunicação atende aos requisitos do sistema.

Na figura 36, pode ser observado um exemplo de uma sequência completa de teste. Conexão wifi, envio do estado do dispositvo (*online*), pronto para comunicação, (após um clique do botão) envio de alerta SOS, (Após resposra de um utilizador) receção do sinal de resposra, (Após clique do botão) cancelamento do alerta SOS.

```
03:00:14.341 -> ..
03:00:15.722 -> Connected!
03:00:15.754 -> IP Address: 10.159.248.206
03:00:15.785 -> Device ID: 12eff89c-ff83-4f26-993e-822931579423
03:00:15.817 -> Sending ONLINE
03:00:17.281 -> Status: Ready!
03:00:20.528 -> Sending SOS toggle...
03:00:22.101 -> SOS Status: 201
03:00:22.101 -> Response: {
03:00:22.133 ->   "active": true,
03:00:22.133 ->   "device_id": "12eff89c-ff83-4f26-993e-822931579423",
03:00:22.197 ->   "event_id": "c56828f8-d6ab-4337-a140-bea1e76b38c6",
03:00:22.261 ->   "message": "SOS triggered"
03:00:22.298 -> }
03:00:22.298 ->
03:00:26.970 -> Help on the way!
03:00:33.397 -> Sending SOS toggle...
03:00:35.244 -> SOS Status: 200
03:00:35.244 -> Response: {
03:00:35.276 ->   "active": false,
03:00:35.276 ->   "message": "SOS stopped"
03:00:35.320 -> }
03:00:35.320 ->
```

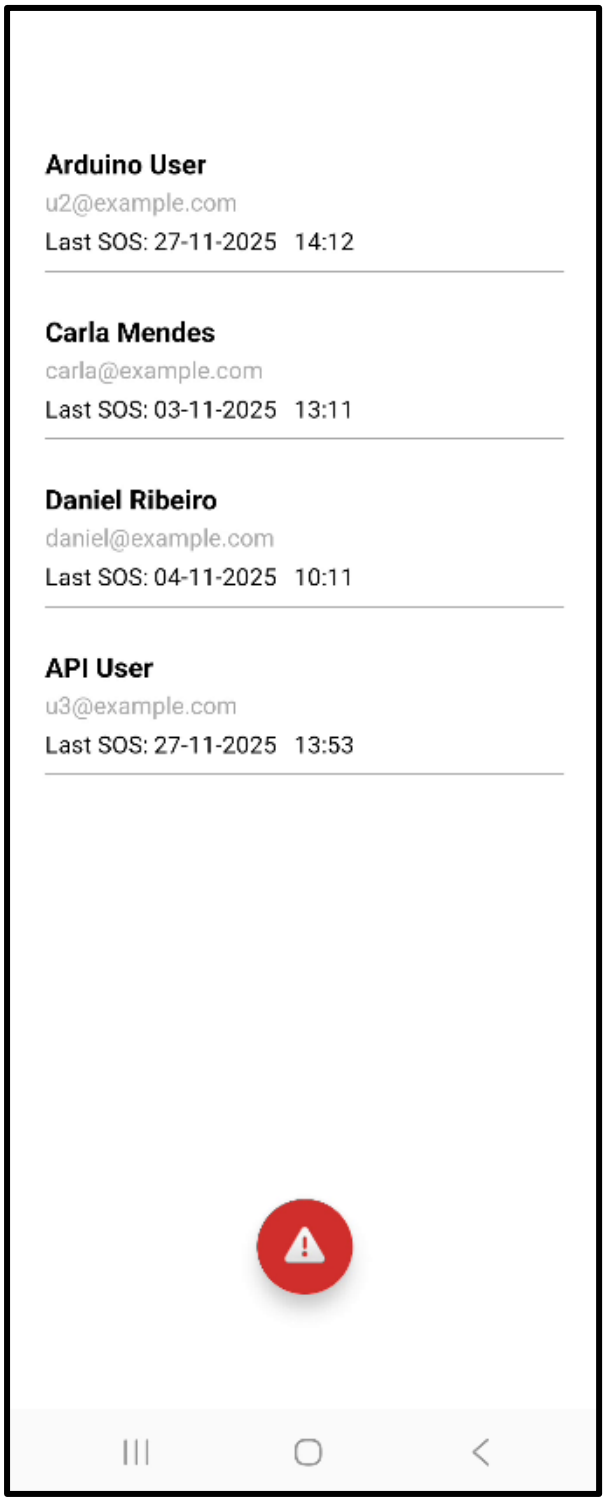
Figura 36 - Exemplos de mensagens no Serial Monitor do Arduino (sequência completa)

4.3 Frontend

A aplicação móvel reflete em tempo real o estado do dispositivo e dos eventos registados no *backend*. Durante os testes, foram observados todos os fluxos principais, visualização de contactos (figura 37, esquerda) e consulta do histórico de eventos (figura 37, direita), deteção de um SOS ativo (figura 38, esquerda), resposta ao alerta (figura 38, direita) e cancelamento da resposta (estado retorna a figura 37, esquerda).

No âmbito da validação funcional, verificou-se que todas as ações do utilizador são refletidas corretamente na API e no estado do dispositivo. Foram testados cenários com múltiplos cuidadores, SOS

consecutivos, variações de conectividade e simultaneidade de ações. A aplicação manteve sempre estabilidade, atualizou os estados sem erros e apresentou tempos de resposta compatíveis com um cenário real de emergência, validando assim a fiabilidade do sistema *end-to-end*.



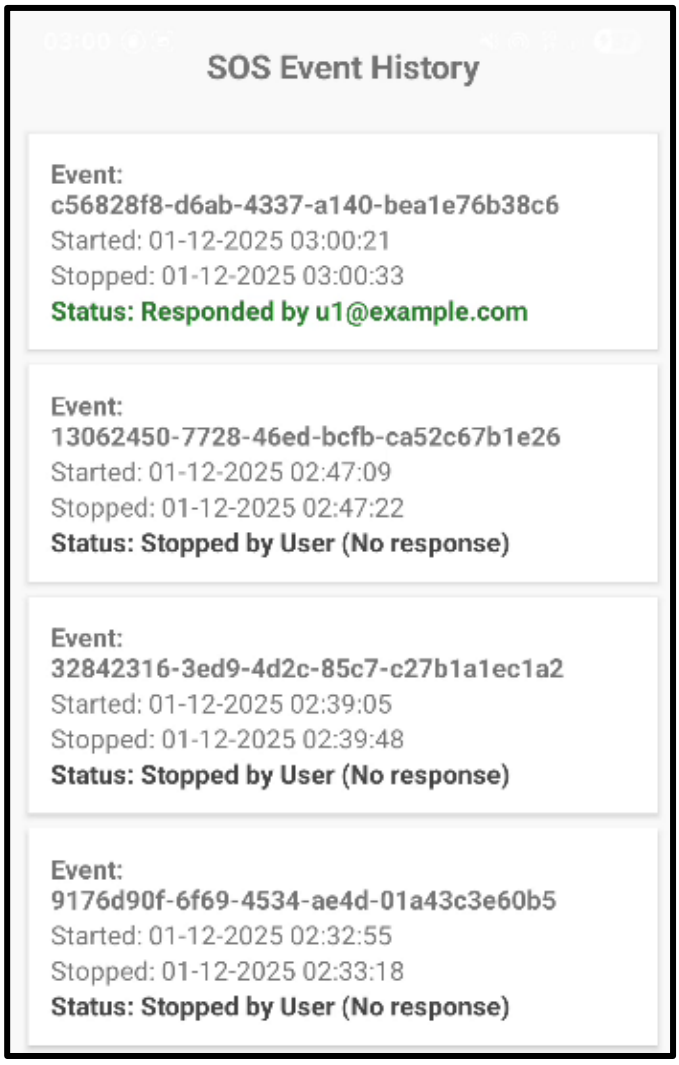


Figura 37 - Página principal e histórico de utilizador, respetivamente

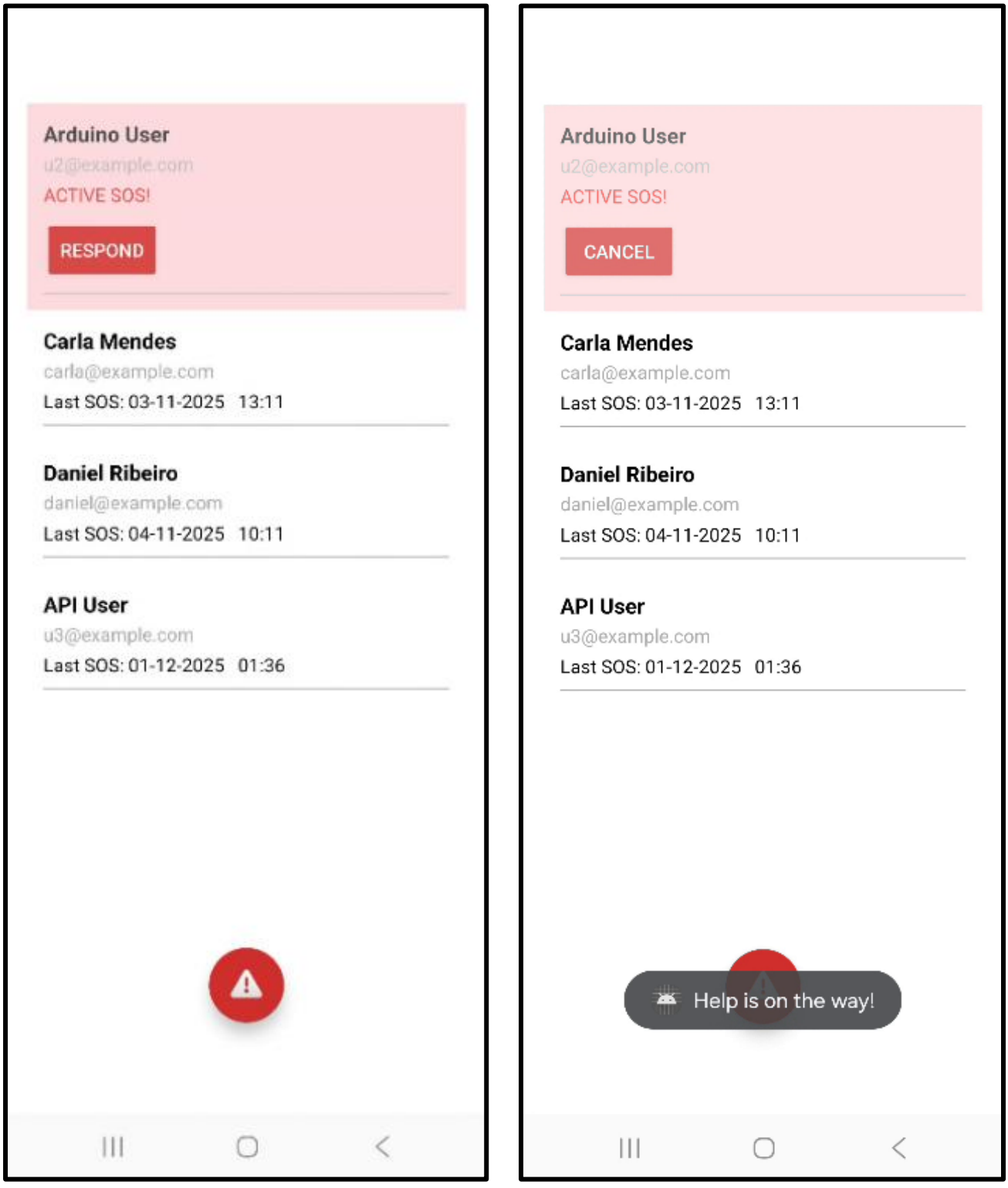


Figura 38 – UI de alerta SOS e respetiva resposta

5 Conclusão

O desenvolvimento do *SafeSenior* permitiu construir um sistema funcional, integrando um dispositivo embebido, *backend* e aplicação *Android* num fluxo coerente e operacional em contexto real. O objetivo principal definido no planeamento foi cumprido, garantindo um mecanismo simples e fiável para sinalizar situações de emergência, notificar cuidadores e assegurar que a informação circula com segurança e consistência entre os vários componentes.

O processo de desenvolvimento revelou vários desafios, nomeadamente a necessidade de gerir eventos concorrentes, sincronizar estados entre diferentes dispositivos, distinguir corretamente identidades de utilizadores e dispositivos e adaptar o planeamento inicial à complexidade real das operações do sistema. As soluções adotadas contribuíram para uma arquitetura mais robusta e coerente, reforçando a estabilidade geral do projeto.

Embora o sistema seja totalmente funcional, algumas limitações foram identificadas. A gestão de notificações encontra-se numa fase inicial e poderá ser expandida com mecanismos mais avançados. A aplicação Android pode beneficiar de melhorias visuais e maior modularização, e a ausência de notificações nativas obriga à utilização de ciclos de atualização periódicos. No backend, a ausência de filas de processamento e de mecanismos avançados de recuperação limita o desempenho sob carga mais elevada.

Considerando a profundidade da implementação, a coerência arquitetural e a validação prática do sistema, o desempenho alcançado corresponde a um nível satisfatório, esperado para um projeto desta natureza.