**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 ii](#_Toc215799960)

[1 Introdução 1](#_Toc215799961)

[2 Arquitetura 2](#_Toc215799962)

[2.1 Tecnologia 3](#_Toc215799963)

[2.2 Sensores 4](#_Toc215799964)

[2.3 protocolos 4](#_Toc215799965)

[3 Implementação 5](#_Toc215799966)

[3.1 Sistema Embebido 5](#_Toc215799967)

[3.1.1 Gestão do dispositivo 5](#_Toc215799968)

[3.1.2 Deteção e controlo do evento SOS 7](#_Toc215799969)

[3.1.3 Sincronização periódica com o backend 8](#_Toc215799970)

[3.1.4 Sinalização local através de LEDs e buzzer 11](#_Toc215799971)

[*3.2* *Backend* 12](#_Toc215799972)

[3.2.1 Base de Dados 13](#_Toc215799973)

[3.2.2 Arquitetura e configuração geral 14](#_Toc215799974)

[3.2.3 Gestão de Utilizadores e Ligações 15](#_Toc215799975)

[3.2.4 Gestão de Dispositivos 16](#_Toc215799976)

[3.2.5 Gestão de alertas SOS 18](#_Toc215799977)

[3.2.6 Gestão de Respostas 19](#_Toc215799978)

[3.3 Frontend 19](#_Toc215799979)

[3.3.1 Estrutura geral 19](#_Toc215799980)

[3.3.2 Comunicação (Retrofit + Shared Preferences) 20](#_Toc215799981)

[3.3.3 *Activities* Principais 21](#_Toc215799982)

[3.3.4 *Adapters* e Representação de Dados 25](#_Toc215799983)

[4 Resultados 28](#_Toc215799984)

[4.1 Hardware 28](#_Toc215799985)

[*4.2* *Serial Monitor* 29](#_Toc215799986)

[*4.3* *Frontend* 29](#_Toc215799987)

[5 Conclusão 32](#_Toc215799988)

# Lista de Figuras

[Figura 1 - Diagrama de arquitetura 3](#_Toc215799989)

[Figura 2 - Inicialização Serial 5](#_Toc215799990)

[Figura 3 - Obtenção e armazenamento do *device ID* 5](#_Toc215799991)

[Figura 4 - Funções auxiliares de acesso à *EEPROM* (armazenamento e leitura, respetivamente 6](#_Toc215799992)

[Figura 5 - Função auxiliar de conexão ao WiFi 7](#_Toc215799993)

[Figura 6 - Função auxiliar de envio de estado (online) ao backend 7](#_Toc215799994)

[Figura 7 - Deteção do clique no botão 8](#_Toc215799995)

[Figura 8 - Função auxiliar alerta SOS (toggle) 8](#_Toc215799996)

[Figura 9 - Função de obtenção do estado de resposta ao sinal SOS 9](#_Toc215799997)

[Figura 10 - Verificação periódica do estado help\_event (início da função) 10](#_Toc215799998)

[Figura 11 - Processamento da resposta help\_event e gestão de timeout 10](#_Toc215799999)

[Figura 12 - Sinalização sonora e visual durante um alerta SOS. 11](#_Toc215800000)

[Figura 13 - Verificação periódica do estado de ajuda 12](#_Toc215800001)

[Figura 14 - Atualização do estado *help\_event* e gestão de *timeout* 12](#_Toc215800002)

[Figura 15 - Diagrama ERD 13](#_Toc215800003)

[Figura 16 - Configuração inicial da API 14](#_Toc215800004)

[Figura 17 - Estrutura Geral da API 15](#_Toc215800005)

[Figura 18 - Função *login* (*backend*) 16](#_Toc215800006)

[Figura 19 - Função de criação de relações 16](#_Toc215800007)

[Figura 20 - Registo e login de dispositivos na API, respetivamente 17](#_Toc215800008)

[Figura 21 - Trecho da função gestora de eventos SOS (*toggle*) 18](#_Toc215800009)

[Figura 22 - Estrutura geral aplicação android 20](#_Toc215800010)

[Figura 23 - Classe ApiClient 21](#_Toc215800011)

[Figura 24 - Exemplo de pedido HTTP da classe ApiInterface 21](#_Toc215800012)

[Figura 25 - Bloco de código da classe *SharedPrefHelper* 21](#_Toc215800013)

[Figura 26 - Lógica de *login* (*frontend*) 22](#_Toc215800014)

[Figura 27 - Apresentação dos contactos 23](#_Toc215800015)

[Figura 28 – *listener* de botão de resposta 23](#_Toc215800016)

[Figura 29 - Carregamento de alertas SOS 24](#_Toc215800017)

[Figura 30 - Notificação visual de alerta SOS 24](#_Toc215800018)

[Figura 31 - Carregamento do histórico de eventos 25](#_Toc215800019)

[Figura 32 - Apresentação de contactos 26](#_Toc215800020)

[Figura 33 - Apresentação do histórico de eventos 27](#_Toc215800021)

[Figura 34 - Circuito completo, estado normal 28](#_Toc215800022)

[Figura 35 – Sinalização dos estados SOS (vermelho para SOS ativo e azul para ajuda confirmada). 28](#_Toc215800023)

[Figura 36 - Exemplos de mensagens no Serial Monitor do Arduino (sequência completa) 29](#_Toc215800024)

[Figura 37 - Página principal e histórico de utilizador, respetivamente 30](#_Toc215800025)

[Figura 38 – UI de alerta SOS e respetiva resposta 31](#_Toc215800026)

# 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.

# 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.

A diagram of a computer network

AI-generated content may be incorrect.

Figura 1 - Diagrama de arquitetura

## 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*.

## 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.

## 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.

# Implementação

## 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*.

### 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*.

A screenshot of a computer program

AI-generated content may be incorrect.

Figura 2 - Inicialização Serial

A screen shot of a computer code

AI-generated content may be incorrect.

Figura 3 - Obtenção e armazenamento do *device ID*

A screenshot of a computer program

AI-generated content may be incorrect.

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:

A screen shot of a computer program

AI-generated content may be incorrect.

Figura 5 - Função auxiliar de conexão ao WiFi

A screen shot of a computer program

AI-generated content may be incorrect.

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

### 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*:

A screen shot of a computer program

AI-generated content may be incorrect.

Figura 7 - Deteção do clique no botão

A screenshot of a computer program

AI-generated content may be incorrect.

Figura 8 - Função auxiliar alerta SOS (toggle)

### 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*:

A screenshot of a computer program

AI-generated content may be incorrect.

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

A computer screen shot of a program

AI-generated content may be incorrect.

Figura 10 - Verificação periódica do estado help\_event (início da função)

A screen shot of a computer program

AI-generated content may be incorrect.

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.

### 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*.

A screen shot of a computer program

AI-generated content may be incorrect.

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

A screenshot of a computer program

AI-generated content may be incorrect.

Figura 13 - Verificação periódica do estado de ajuda

A screen shot of a computer program

AI-generated content may be incorrect.

Figura 14 - Atualização do estado *help\_event* e gestão de *timeout*

## *Backend*

### 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.

A screenshot of a computer

AI-generated content may be incorrect.

Figura 15 - Diagrama ERD

### 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.

A screen shot of a computer

AI-generated content may be incorrect.

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.

A screenshot of a computer program

AI-generated content may be incorrect.

Figura 17 - Estrutura Geral da API

### 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*.

A computer screen shot of text

AI-generated content may be incorrect.

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

A computer screen shot of text

AI-generated content may be incorrect.

Figura 19 - Função de criação de relações

### 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.

A screen shot of a computer program

AI-generated content may be incorrect.

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

### 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.

A screen shot of a computer program

AI-generated content may be incorrect.

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

### 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.

## Frontend

### 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.

A screenshot of a computer

AI-generated content may be incorrect.

Figura 22 - Estrutura geral aplicação android

### 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.

A screen shot of a computer program

AI-generated content may be incorrect.

Figura 23 - Classe ApiClient

A screen shot of a computer program

AI-generated content may be incorrect.

Figura 24 - Exemplo de pedido HTTP da classe ApiInterface

A computer screen with text

AI-generated content may be incorrect.

Figura 25 - Bloco de código da classe *SharedPrefHelper*

### *Activities* Principais

#### 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.

A computer screen shot of text

AI-generated content may be incorrect.

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

#### 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().*

##### 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.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figura 27 - Apresentação dos contactos

A screen shot of a computer screen

AI-generated content may be incorrect.

Figura 28 – *listener* de botão de resposta

##### *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.

A screen shot of a computer code

AI-generated content may be incorrect.

Figura 29 - Carregamento de alertas SOS

##### *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 qual dos seus contactos acabou de ativar um SOS, tirando partido do sistema de notificações sem introduzir complexidade adicional.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figura 30 - Notificação visual de alerta SOS

#### 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.

A computer screen shot of text

AI-generated content may be incorrect.

Figura 31 - Carregamento do histórico de eventos

### *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.

##### *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.

A screen shot of a computer program

AI-generated content may be incorrect.

Figura 32 - Apresentação de contactos

##### *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.

A computer screen shot of a program

AI-generated content may be incorrect.

Figura 33 - Apresentação do histórico de eventos

##### 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*.

# 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.

## 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.

A circuit board with wires connected to it

AI-generated content may be incorrect.

Figura 34 - Circuito completo, estado normal

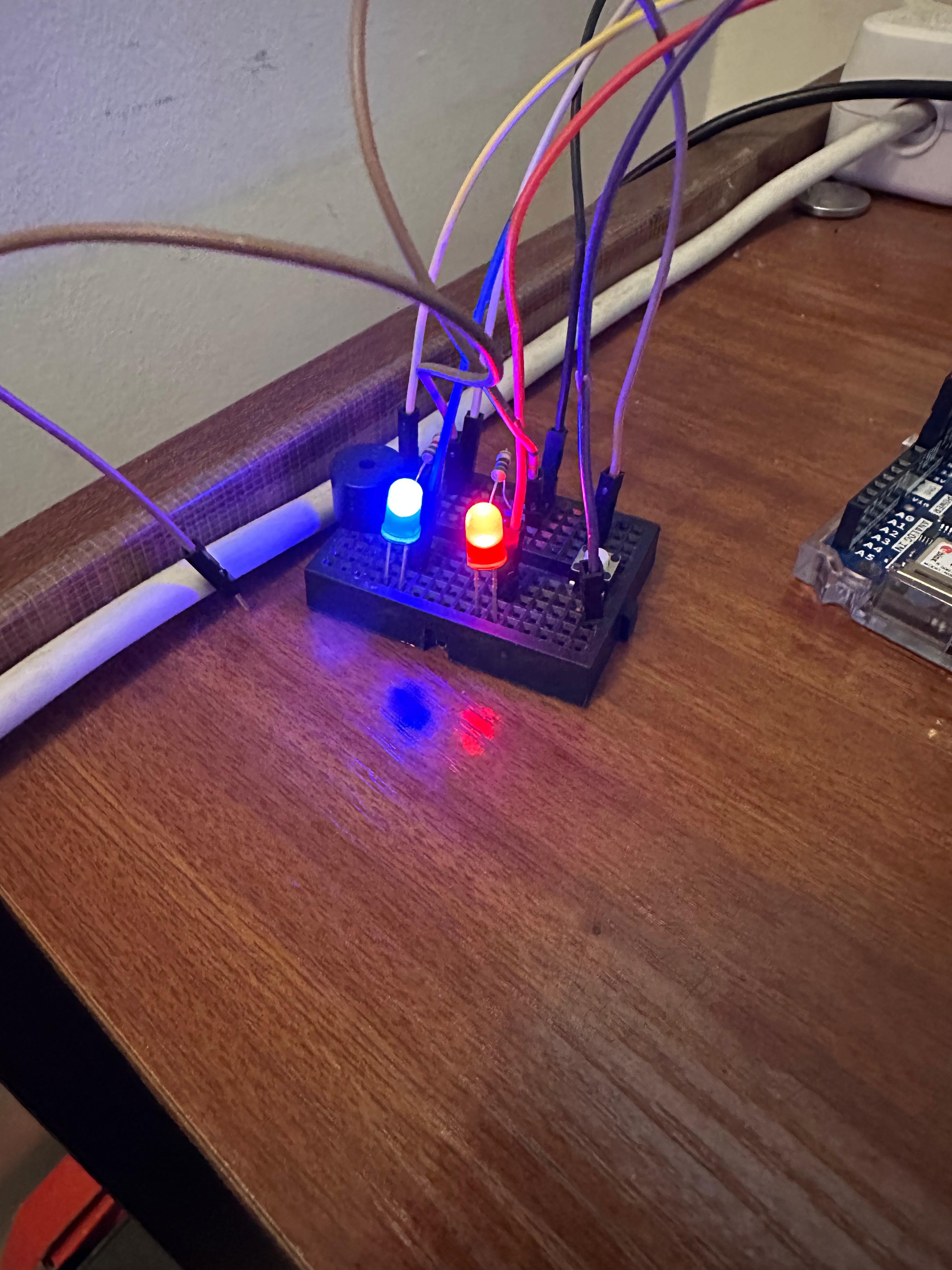
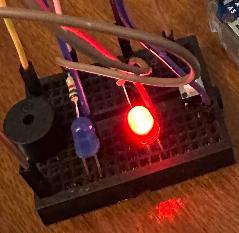


Figura 35 – Sinalização dos estados SOS (vermelho para SOS ativo e azul para ajuda confirmada).

## *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.

A screenshot of a computer program

AI-generated content may be incorrect.

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

## *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*.

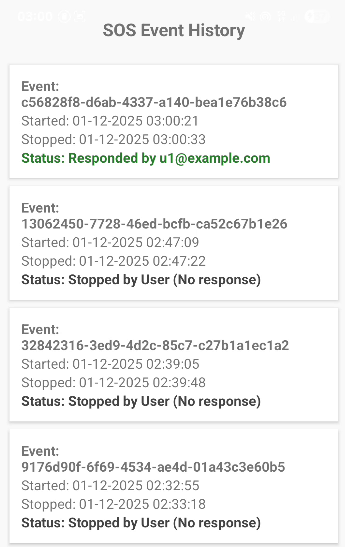
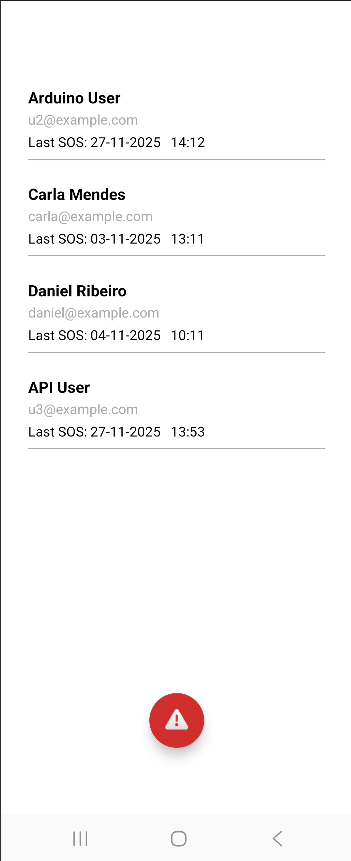


Figura 37 - Página principal e histórico de utilizador, respetivamente

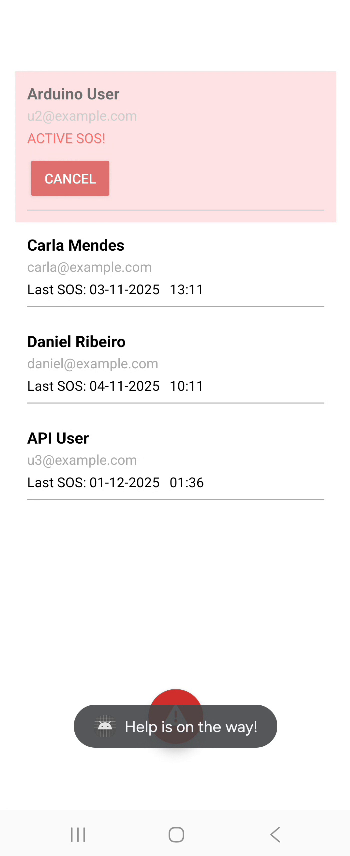
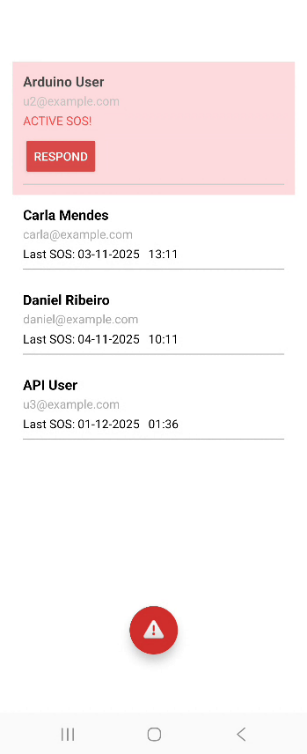


Figura 38 – UI de alerta SOS e respetiva resposta

# 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.