# Arduino

## 1. Identidade e Persistência do Dispositivo

O firmware do SafeSenior começa por garantir que cada dispositivo tem uma identidade própria e permanente. Este identificador, o *device_id*, é atribuído pelo backend e guardado na EEPROM do Arduino.

A escolha da EEPROM foi deliberada, trata-se de uma memória não volátil que preserva os dados mesmo quando ocorre falha de energia, permitindo que o dispositivo volte a operar automaticamente sem necessitar de reconfiguração.

O armazenamento limita-se apenas ao *device_id*, seguindo uma filosofia de design minimalista. A EEPROM possui um número limitado de ciclos de escrita e, sendo este valor essencialmente estático após a primeira configuração, evita-se desgaste e complica-se menos a lógica interna. Assim que o firmware inicia, tenta ler o *device_id*. Caso esteja ausente (primeira utilização), solicita-o via Serial Monitor e grava-o permanentemente. A partir deste momento, cada comunicação com a API identifica inequivocamente o dispositivo.

---

## 2. Conectividade e Notificação do Estado Online/Offline

Depois de garantir que o dispositivo tem identidade, o firmware estabelece ligação Wi-Fi. Este processo é repetido até obter sucesso, uma vez que em ambientes domésticos é comum o router demorar alguns segundos ou minutos a disponibilizar conectividade. A ligação à rede é um pré-requisito para a correta operação do sistema, e por isso a tentativa persistente garante que o dispositivo só avança quando está efetivamente online.

Quando finalmente se liga, o dispositivo comunica ao backend que está online, enviando um pedido simples que atualiza o estado da base de dados. Esta notificação é relevante para quem monitoriza o dispositivo através da aplicação, pois permite saber se o hardware está funcional.

O processo inverso também é automaticamente tratado: se a ligação cair, o firmware deteta a perda de conectividade, informa o backend que está offline, tenta restabelecer a ligação e, quando o consegue, envia novamente a sinalização de online. Este comportamento garante coerência entre o estado real e o estado representado e evita que o cuidador seja induzido em erro relativamente à disponibilidade do dispositivo.

---

## 3. Gestão do Evento SOS através do Botão Físico

O botão de emergência é o elemento central do sistema. Foi configurado com INPUT_PULLUP, uma abordagem simples e fiável onde o botão não pressionado corresponde a HIGH e pressionado a LOW. O firmware monitoriza continuamente esta entrada, mas reage apenas à transição de HIGH para LOW, assegurando que o SOS é acionado apenas uma vez por toque, mesmo que o utilizador mantenha o botão pressionado durante mais tempo.

Quando deteta essa transição, o firmware envia um pedido POST ao endpoint responsável por gerir SOS no backend. Independentemente da resposta recebida, o firmware alterna o estado local *sosActive*. Esta é uma decisão técnica relevante: o firmware assume sempre que o backend recebeu e processou o pedido.

Embora funcione de forma simples e previsível, esta abordagem pode originar divergências temporárias caso exista falha na comunicação, porque o dispositivo irá comportar-se como se o estado tivesse realmente mudado. Uma melhoria futura poderia consistir em validar a resposta HTTP antes de atualizar o estado local ou ainda sincronizar o estado SOS com o backend no arranque, algo que atualmente não acontece.

---

## 4. Sinalização Local: LEDs e Buzzer

A resposta física ao utilizador é feita através de dois LEDs (vermelho e azul) e de um buzzer. Quando um SOS está ativo, o LED vermelho pisca e o buzzer emite pequenos sons intercalados por pausas. O objetivo deste padrão é transmitir a ideia de urgência. O padrão não segue uma norma específica, mas foi desenhado para ser intuitivo: beeps curtos e frequentes acompanham situações que requerem atenção imediata.

Quando alguém responde ao alerta no frontend, o backend cria um evento de ajuda (*help_event*) e o firmware passa a exibir um comportamento diferente. O LED azul acende-se e o buzzer mantém o mesmo tipo de som, mas com pausas mais longas, indicando ao utilizador que a ajuda já está a caminho. O contraste entre o padrão “rápido” e o padrão “lento” é a base conceptual da distinção entre SOS e ajuda confirmada. Os valores numéricos usados (150 ms de beep, 450 ms ou 850 ms de pausa) não têm um fundamento científico ou clínico; foram escolhidos empiricamente, garantindo apenas clareza e distinção percetível entre estados.

---

## 5. Sincronização com o Backend: Polling do Help Event

O firmware comunica com o backend de duas formas distintas: envia eventos SOS (modelo push) e pergunta periodicamente pelo estado de ajuda (modelo pull). Esta separação foi natural para o protocolo usado. O Arduino envia SOS porque esse evento é sempre iniciado

localmente; por outro lado, não recebe automaticamente sinal do backend quando um cuidador responde, já que esse fluxo exigiria outro tipo de comunicação (como WebSockets ou MQTT). Assim, optou-se por *polling*: o Arduino pergunta ao backend, de 2 em 2 segundos, se existe ajuda ativa associada ao seu *device_id*.

O pedido de ajuda é feito através de um GET e responde com uma estrutura simples onde apenas interessa verificar se `"help": true` ou `false`. O firmware interpreta esse valor e acende ou apaga o LED azul em conformidade.

O método foi pensado para ser não bloqueante: utiliza timers baseados em `millis()` em vez de `delay()`, permitindo que o dispositivo continue a piscar LEDs, tocar buzzer e verificar Wi-Fi enquanto aguarda a resposta.

O firmware também inclui um mecanismo de timeout de 1,5 segundos que impede que a ausência de resposta bloqueie o loop principal. Quando ocorre timeout, a tentativa é descartada e retomada no ciclo seguinte. Esta abordagem é suficientemente robusta para um ambiente realista, onde a latência doméstica pode variar bastante.

---

## 6. Arquitetura Não Bloqueante e Multitarefas Simples

Um dos pontos tecnicamente mais importantes do firmware é o facto de nunca utilizar `delay()`. Esta decisão evita o congelamento do programa durante períodos de espera, que afetaria negativamente a perceção de resposta. Em vez disso, recorre a `millis()` para controlar intervalos de tempo. Esta escolha permite que diferentes comportamentos — como piscar LEDs, emitir sons, verificar o botão, pedir estado ao backend e monitorizar a ligação Wi-Fi — ocorram de forma paralela e fluida. Apesar de não existir verdadeira concorrência num Arduino, este estilo de programação imita de forma eficaz um sistema multitarefa cooperativo.

---

## 7. Robustez a Falhas e Comportamento em Situação Real

O firmware está preparado para lidar com falhas de rede e oscilações de resposta do servidor.

- Se a ligação Wi-Fi falhar, o dispositivo tenta novamente estabelecer conexão e mantém o loop ativo sem bloquear.
- A sincronização da ajuda através de polling é resiliente: se um pedido falhar, o estado anterior é mantido e uma nova tentativa será feita brevemente.
- Quando o botão SOS é pressionado, o firmware alterna o estado local, mesmo que a API não responda; embora esta seja uma limitação, permite que o dispositivo mantenha feedback imediato ao utilizador, essencial num contexto de emergência. Em termos de usabilidade, este comportamento é preferível a “não fazer nada” caso exista falha de rede.

## 8. Limitações Reais e Possíveis Melhorias

| Limitação | Melhoria |
|---|---|
| Quando o botão é pressionado, o firmware alterna o estado sosActive mesmo que a API não responda ou devolva erro. Isto pode gerar discrepâncias entre o estado exibido no dispositivo e o estado real na API. | Implementar verificação da resposta HTTP antes de atualizar o estado local, garantindo que o SOS só muda quando o backend confirma a operação. |
| Se o dispositivo reiniciar enquanto existe um SOS ativo na API, o firmware assume sempre que não há SOS em curso, criando inconsistência entre backend e dispositivo. | Realizar um pedido inicial ao backend durante o setup() para verificar se existe algum SOS ativo e ajustar sosActive em conformidade. |
| O dispositivo só pergunta pelo estado da ajuda a cada 2 segundos. Isto significa que a informação pode demorar até 2 segundos a refletir-se no hardware. | Reduzir o intervalo de polling para um valor menor (por exemplo, 1 segundo), ou migrar futuramente para mecanismos que suportem comunicação em tempo real, como WebSockets ou MQTT, se o hardware permitir. |
| Se o pedido GET ao endpoint /help/state falhar, o firmware mantém o estado anterior, podendo prolongar a exibição de estados desatualizados. | Adicionar lógica de "estado incerto", onde sucessivas falhas no polling levam a temporariamente ocultar o indicador de ajuda ou solicitar uma verificação mais intensa até obter resposta válida. |
| O Arduino identifica-se unicamente pelo device_id, sem utilizar tokens de autenticação que garantam maior segurança. | Implementar autenticação por token JWT específico para dispositivos, podendo o firmware guardá-lo também na EEPROM para uso persistente. |
| Caso o firmware entre num estado anómalo (ex.: loop bloqueado por erro de biblioteca), o dispositivo pode deixar de responder. | Ativar o watchdog interno do microcontrolador, configurando-o para reiniciar automaticamente o Arduino se o firmware ficar preso mais tempo do que o esperado. |
| O SOS é enviado apenas como um pedido único de toggle, sem reenvio automático em caso de falha e sem "acknowledgement" robusto. | Adicionar um mecanismo opcional de retransmissão ou confirmação visual só após resposta válida do backend, garantindo consistência entre as duas camadas. |

# API

## 1. Introdução

A API do SafeSenior foi construída em Flask com uma arquitetura simples e altamente explícita, mas suficientemente robusta para suportar comunicação bidirecional entre um dispositivo IoT (Arduino), aplicações móveis e a base de dados Supabase. A filosofia central da API é a transparência. Cada endpoint tem uma função única e clara, não há camadas de abstração complexas e toda a comunicação com a base de dados é feita através da REST API nativa do Supabase, sem SQL direto.

Esta abordagem facilita depuração, aumenta previsibilidade e reduz os riscos de inconsistência entre a lógica do backend e a estrutura da base de dados, ao mesmo tempo que se adequa a um projeto académico onde legibilidade e rastreabilidade são essenciais.

---

## 2. Arquitetura Geral e Comunicação com o Supabase

A API carrega inicialmente as variáveis de ambiente, configura a chave JWT e define as URLs base para todas as tabelas do Supabase. Cada tabela (user, connection, sos_device, sos_event, help_event, notification) tem uma URL REST correspondente, permitindo aceder à base de dados com operações HTTP standard. O Supabase trata internamente da execução SQL — o backend limita-se a construir pedidos bem formados e interpretar respostas. Esta abordagem remove complexidade sem sacrificar funcionalidade.

O método `supabase_headers()` centraliza todos os cabeçalhos necessários para autenticação e comportamento do Supabase. O backend usa sempre a configuração `Prefer: return=representation`, que faz com que o Supabase devolva o registo acabado de criar ou atualizar. Isto permite que o código obtenha imediatamente IDs, estados atualizados ou resultados úteis, evitando pedidos adicionais.

A API não usa ORM nem camadas sofisticadas; cada pedido HTTP ao Supabase é explícito. Isto torna o fluxo das operações extremamente claro e fácil de explicar numa defesa: a API funciona como uma ponte literal entre os clientes e a base de dados, sem esconder comportamento em abstrações desnecessárias.

---

## 3. Mecanismo de Autenticação

A autenticação é feita através de dois decoradores distintos: um para utilizadores (`auth_user`) e outro para dispositivos (`auth_device`). Esta separação é fundamental para manter a coerência

do sistema, visto que o Arduino não tem capacidade para lidar com o mesmo fluxo de autenticação que a aplicação móvel.

O decorador auth_user valida tokens JWT emitidos pela API durante o login. Cada token contém o ID do utilizador e expira após oito horas. Se o token for válido, o decorador anexa ao objeto request o campo user_id, permitindo que os endpoints subsequentes saibam de forma inequívoca quem está a fazer a chamada. A particularidade interessante é que este decorador também aceita tokens de dispositivo. Quando o payload contém "device_id" em vez de "id", o backend procura o dispositivo correspondente no Supabase, identifica o proprietário e atribui esse proprietário ao request.user_id. Com isto, um mesmo decorador permite que tanto pessoas como dispositivos acionem funcionalidades protegidas — algo muito útil para permitir que o Arduino desencadeie notificações sem que haja ambiguidade sobre a identidade do utilizador associado.

Já o decorador auth_device é usado quando o Arduino não envia um token JWT (que seria possível, mas menos prático). Em vez disso, envia o device_id no corpo do pedido, e o backend valida se esse dispositivo é conhecido. Se for, atribui ao objeto request tanto o device_id como o user_id do proprietário. Este fluxo permite que endpoints acionados pelo Arduino saibam sempre quem é o utilizador associado ao evento.

Este modelo de autenticação *duplo* torna-se essencial numa arquitetura onde dispositivos e utilizadores coexistem e interagem com os mesmos dados, mas mediante credenciais diferentes.

---

## 4. Gestão de Utilizadores

A API fornece endpoints básicos para criação de contas, autenticação e obtenção de perfil. O registo inclui validação de campos, verificação de unicidade de email e encriptação da password usando SHA-256. A escolha de SHA-256, apesar de simples, é suficiente no contexto académico, garantindo que a password nunca é guardada em texto simples. O login compara o hash recebido com o armazenado e devolve um JWT com validade de oito horas. Este token representa a sessão do utilizador na aplicação Android.

O endpoint /user/me permite recuperar informação de perfil de forma segura, garantindo que apenas utilizadores autenticados conseguem aceder aos seus próprios dados.

---

## 5. Relação entre Cuidadores e Dependentes

A tabela connection representa relações entre utilizadores. Cada relação contém dois IDs e é interpretada como bidirecional. O endpoint de criação impede um utilizador de se ligar a si próprio e confia no Supabase para evitar duplicados.

O endpoint de listagem é mais complexo e importante: não devolve apenas as relações, mas também informação enriquecida sobre cada contacto. Para cada utilizador ligado, a API obtém o seu dispositivo (caso exista), recolhe o último SOS emitido e agrega tudo numa só resposta. Isto melhora o desempenho da aplicação móvel, reduzindo a necessidade de múltiplos pedidos.

A lógica de enriquecimento de dados é também um bom exemplo do modelo da API: cada operação é simples, mas combinada com outras para produzir um resultado altamente funcional.

---

## 6. Gestão de Dispositivos

O endpoint /devices cria um novo dispositivo para um utilizador autenticado. O servidor gera um UUID, grava-o no Supabase e devolve-o ao cliente — normalmente a aplicação móvel, que depois o transmite ao utilizador para que este configure o Arduino.

O endpoint /devices/login é usado caso se pretenda gerar um token JWT dedicado ao dispositivo, embora o firmware atual não utilize este mecanismo.

A API suporta ainda operações de remoção e consulta de dispositivos, mas o aspeto mais importante consiste na sinalização de *online/offline*. Os endpoints /device/online e /device/offline atualizam diretamente o campo is_online, registam a última vez que o dispositivo foi visto e asseguram que eventos de ajuda em curso são encerrados quando o dispositivo volta a ligar-se. Esta lógica evita estados inconsistentes, como ajudas ativas sobre dispositivos que reiniciaram ou deixaram de estar disponíveis.

---

## 7. Gestão de SOS: Ciclo de Vida Completo do Evento

O endpoint /sos representa o núcleo funcional do sistema. A sua responsabilidade é dupla: ativar um novo evento SOS ou encerrar um evento ativo. Para isso, começa por determinar quem desencadeia o pedido, distinguindo entre frontend e dispositivo. Se o pedido traz um token JWT, assume-se que o utilizador acionou o SOS através da aplicação; se não traz, então o pedido deve conter device_id, e o firmware do Arduino é o responsável.

Depois de identificar a fonte, o backend procura na tabela sos_event se existe algum evento do utilizador ainda ativo — isto é, o query procura eventos com handled = false. Caso encontre, interpreta o pedido como um cancelamento e encerra o evento, registando o timestamp.

Quando um evento é cancelado, o backend encerra automaticamente qualquer help_event relacionado. Esta lógica assegura coerência entre o estado do botão físico e o estado da ajuda remota: se o SOS terminou, ajuda não pode permanecer ativa.

Se não existirem eventos ativos, a API cria um novo evento. Caso o SOS venha do frontend e o utilizador ainda não tenha dispositivo registado, um dispositivo é criado automaticamente, garantindo que o evento permanece associado ao modelo de dados correto. Após criar o evento, a API atualiza o dispositivo, marcando-o como online e anotando o instante em que foi acionado. Este fluxo garante que cada SOS possui um ciclo de vida completo e rastreável.

---

## 8. Listagem de Eventos SOS e Monitorização Global

A API disponibiliza diferentes endpoints para consulta de SOS. Um deles devolve todos os eventos de um utilizador específico, acessível por email, respeitando a ordem cronológica inversa. Outro devolve todos os utilizadores que têm um evento SOS ativo. Este segundo endpoint é útil para o frontend, que, durante a operação normal, precisa de identificar rapidamente quem está em situação de emergência.

O backend não filtra nem interpreta os eventos; devolve-os tal como estão armazenados, permitindo que o frontend decida como apresentá-los.

---

## 9. Help Event: Indicador de Resposta do Cuidador

Os eventos de ajuda funcionam de forma semelhante aos eventos SOS, mas num canal separado. O endpoint /help/toggle alterna entre criar ou encerrar um help_event. Se já existir um evento ativo para um dispositivo, ele é encerrado e o cuidador é registado como o responsável pela manipulação. Se não existir, um novo evento é criado.

Sempre que um help_event é criado, o backend procura automaticamente um evento SOS ativo correspondente e preenche o campo handled_by, registando quem respondeu primeiro. Este mecanismo permite identificar o cuidador que assumiu responsabilidade pelo alerta — informação que pode ser valiosa para o frontend ou para registos futuros.

O endpoint /help/state/<device_id> devolve um estado booleano simples, sendo perfeito para consumo pelo Arduino, que apenas precisa de saber se ajuda está ou não ativa.

---

## 10. Notificações: Reação a Eventos do Sistema

A API contém endpoints adicionais para registar e consultar notificações. Estes dados não influenciam o funcionamento do Arduino, mas são relevantes para o frontend. Quando um SOS é iniciado, o backend envia notificações a todos os cuidadores ligados ao utilizador. Quando um SOS é encerrado, as notificações associadas são marcadas como "seen". As notificações funcionam como um registo histórico, útil para a experiência do cuidador.

## 11. Robustez, Decisões de Design e Possíveis Melhorias

Tal como no firmware, a API representa um equilíbrio entre simplicidade e funcionalidade. A escolha de usar apenas REST do Supabase elimina complexidade, mas também limita flexibilidade. Por exemplo, o sistema depende de várias chamadas HTTP encadeadas para enriquecer dados (como acontece nas ligações ou nos eventos SOS), o que poderia tornar-se mais pesado se a escala aumentasse.

A autenticação do dispositivo é funcional, mas poderia ser reforçada com um JWT distinto ou com mecanismos de expiração mais controlados. O toggle de SOS poderia também devolver estados mais ricos, permitindo ao dispositivo ou ao frontend otimizar o comportamento local. Os fluxos de cancelamento, apesar de robustos, assumem sempre consistência entre as tabelas; sistemas maiores podem exigir transações para evitar inconsistências temporárias. Contudo, para a escala e requisitos do SafeSenior, o design adotado cumpre perfeitamente o seu objetivo.

# Frontend

## 1. Papel do frontend no sistema SafeSenior

A aplicação Android é a interface através da qual os cuidadores interagem com o sistema SafeSenior. Enquanto o Arduino representa o "corpo" (botão, LEDs, buzzer) e a API representa o "cérebro" (gestão de SOS, help_event, utilizadores, notificações), o frontend é a "cara" do sistema: mostra quem está em SOS, quem tem dispositivo associado, o histórico desses eventos e permite a um cuidador responder a um alerta com um simples toque no ecrã.

Do ponto de vista arquitetural, a app foi desenhada para ser o mais simples e explícita possível: Activities tratam da interface e da lógica de apresentação, uma camada de rede baseada em Retrofit encapsula a comunicação com a API, SharedPrefHelper gere o token JWT e os modelos (Connection, Event, Device, Notification, etc.) espelham diretamente as estruturas da base de dados, sem reinterpretação complexa.

## 2. Estrutura geral da aplicação e organização de código

A aplicação está organizada de forma modular. No topo estão as Activities principais, cada uma com uma responsabilidade bem definida: autenticação, painel principal com contactos e SOS, e consulta de histórico. Numa camada intermédia surgem os adapters para ligar dados às listas (RecyclerViews) e, na base, a camada de rede com Retrofit e a gestão de sessão com SharedPreferences. Os modelos de dados ligam a API à UI, funcionando como contentores simples de informação.

Esta organização segue um padrão natural: Activities não falam diretamente com a API em termos de HTTP "cru". Em vez disso, trabalham com métodos da interface Retrofit (ApiInterface), que devolve objetos de modelo (Connection, Event, Notification, etc.). ApiClient centraliza a construção do Retrofit com o URL base da API, e SharedPrefHelper garante que, sempre que um endpoint protegido é chamado, o token JWT está disponível e aplicado nos cabeçalhos.

---

## 3. Camada de rede: ApiClient, ApiInterface e SharedPrefHelper

A comunicação com o backend é feita com Retrofit, que foi escolhido por ser simples, tipado e muito adequado a arquiteturas REST. O ApiClient é responsável por criar a instância de Retrofit com o baseUrl correto (o endereço da tua API Flask) e converter JSON para objetos Java. Esta instância é reutilizada em toda a app, evitando construções repetidas.

ApiInterface define, através de anotações, todos os endpoints disponíveis: login, registo, listagem de ligações, listar eventos SOS, ativar ajuda, carregar notificações, etc. Cada método corresponde a um endpoint da API, especificando o tipo de pedido (GET, POST, DELETE), o caminho e os parâmetros (por query, path ou body). Em vez de a Activity saber como construir um POST para /sos com device_id, limita-se a chamar um método como toggleSOS(...) definido nesta interface; isto remove lógica de HTTP das Activities e concentra o protocolo num único sítio.

SharedPrefHelper é o componente que trata da persistência do token JWT no dispositivo. Quando o utilizador faz login com sucesso, o token devolvido pela API é guardado em SharedPreferences. Mais tarde, sempre que um pedido autenticado é enviado, este helper recupera o token e adiciona-o ao cabeçalho Authorization. Isto permite que quase toda a app trate a autenticação de forma transparente: as Activities não precisam de se preocupar em passar o token manualmente em cada chamada, apenas em garantir que o utilizador está autenticado antes de aceder a ecrãs protegidos.

---

## 4. Fluxo de autenticação: LoginActivity e RegisterActivity

A LoginActivity é o ponto de entrada lógico da aplicação. O utilizador insere email e password, e a Activity chama o endpoint de login através do ApiInterface. O backend valida as credenciais, gera o JWT e devolve o token juntamente com o ID do utilizador. A Activity, ao receber esta resposta, guarda o token com o SharedPrefHelper e redireciona o utilizador para a MainActivity. Se as credenciais forem inválidas, a Activity apresenta uma mensagem de erro.

A RegisterActivity permite a criação de uma nova conta. Ela constrói um objeto com nome, email e password, invoca o endpoint /register e trata as mensagens de sucesso ou erro (como email duplicado). A lógica mantém-se simples: esta Activity não tenta fazer login automático

após registo (a não ser que tenhas implementado isso), limitando-se a orientar o utilizador para voltar ao ecrã de login depois do sucesso.

Uma vez autenticado, o utilizador só volta a interagir com estas Activities se fizer logout (caso tenhas previsto isso) ou se limpar os dados da aplicação. Em termos de defesa, é relevante sublinhar que toda a interação crítica (ver SOS, responder a SOS, ver histórico) depende de um token válido, reforçando a segurança do sistema.

---

## 5. MainActivity: painel principal, ligações, SOS ativos e notificações

A MainActivity é o centro operacional da aplicação para o cuidador. Quando inicia, começa por carregar a lista de ligações do utilizador, chamar endpoints que indicam quem tem SOS ativo e ler as notificações relevantes. Esta Activity conjuga três vertentes: apresentação da rede de contacto, detecção de emergências em curso e resposta direta a esses eventos.

O método responsável por carregar ligações faz uma chamada ao endpoint de listagem de connections. A resposta já vem "enriquecida" pelo backend, incluindo nome, email, device_id e a data do último SOS de cada contacto. A Activity recebe esta lista, instancia o ConnectionsAdapter e associa-a a uma RecyclerView para apresentação. Cada item da lista mostra, pelo menos, o nome do contacto e informação adicional útil (como o último SOS registado) e permite interação (por exemplo, ver histórico ou acionar ajuda).

A MainActivity também faz consultas periódicas para verificar quem se encontra com SOS ativo. Esta lógica é essencial para que o painel reflita, em tempo quase real, o estado de emergência dos contactos. Sempre que o backend indica que um utilizador está em SOS, a Activity comunica isso ao ConnectionsAdapter, que atualiza visualmente o item correspondente – por exemplo, mudando cor de fundo, texto, ícone ou outra marca visual que destaque essa situação.

No escopo atual, também existe um mecanismo básico de notificações no frontend. A Activity pode carregar notificações recentes do endpoint /notifications e, pelo menos, identificar rapidamente qual dos contactos acionou um alerta SOS, destacando-o na interface. Mesmo que não exista um sistema avançado de push, o simples facto de carregar notificações e estados ativos periodicamente já oferece uma experiência suficientemente dinâmica para o contexto do projeto.

---

## 6. UserEventsActivity: histórico de eventos SOS

A UserEventsActivity é aberta a partir da MainActivity quando o cuidador seleciona um contacto e pretende ver o seu histórico de SOS. Esta Activity recebe, por Intent, o email ou identificador do contacto e, usando o ApiInterface, faz uma chamada ao endpoint que lista eventos daquele utilizador (ordenados por data).

O papel desta Activity é relativamente simples: obter os eventos do backend, lidar com casos de erro (como ausência de registos ou falha de comunicação) e entregar a lista ao EventsAdapter. A lógica de apresentação visual — formatação de datas, ordenação definida, distinção de SOS concluídos e ativos, e títulos como "SOS em curso" ou "SOS concluído" — é tratada pelo adapter. Se a lista vier vazia, a Activity deve indicar claramente que não existem eventos para esse contacto.

---

## 7. Adapters: ConnectionsAdapter e EventsAdapter

Os adapters são o ponto intermédio entre os dados e a interface visual. O ConnectionsAdapter recebe a lista de Connection e é responsável por criar e ligar as views de cada item. Ele conhece a estrutura do item de layout (por exemplo, nome, email, estado de SOS, botão "responder", botão "ver histórico") e, para cada objeto Connection, preenche os campos adequados.

Um aspecto importante é que o adapter também contém lógica de atualização dinâmica. Por exemplo, quando a MainActivity descobre que um determinado contacto está em SOS, pode informar o adapter, que marca esse item como "em emergência" — talvez mudando o fundo ou mostrando um indicador vermelho. Isto permite atualizar só o fragmento de interface que mudou, em vez de reconstruir toda a lista, garantindo fluidez.

O EventsAdapter lida com a lista de objetos Event. O seu trabalho é mais direto: pega na lista de eventos retornados pela API e representa cada um numa linha, com datas, hora de início, hora de fim, estado, e eventualmente quem respondeu. A lógica aqui é propositadamente simples: o adapter não reinterpreta os dados, apenas os apresenta tal como vêm do backend, aplicando formatação mínima (por exemplo, converter um timestamp ISO para formato "dd-MM-yyyy HH:mm").

---

## 8. Modelos de dados: Connection, Device, Event, Notification, e outros

Os modelos são classes Java que espelham diretamente as entidades presentes no backend. O objetivo é manter a app o mais alinhada possível com a API e a base de dados, evitando conversões complexas ou lógicas duplicadas.

A classe Connection representa uma ligação entre o utilizador atual e outro utilizador. Tipicamente contém campos como other_user_name, other_user_email, device_id e talvez last_sos. Estes campos são exatamente os que o endpoint /connections devolve após enriquecer os dados: nome e email são usados para apresentação textual, device_id serve como chave para acionar SOS ou help_event, e last_sos permite mostrar ao cuidador quando foi o último episódio de emergência daquele contacto.

O modelo Device representa um dispositivo físico associado a um utilizador, geralmente com campos como device_id, owner_id, is_online, last_triggered_at e possivelmente last_seen_at. Este

modelo é relevante para ecrãs que mostram detalhes do dispositivo ou para lógicas internas que precisam de saber se um contacto tem ou não um dispositivo registado e se está online.

A classe Event representa um registo na tabela sos_event. Os campos mais típicos incluem event_id, device_id, triggered_by, on_at, off_at, handled e handled_by. A Activity de histórico consome diretamente estes campos: on_at e off_at para mostrar a duração e a cronologia, handled para indicar se o evento ainda está em aberto e handled_by para mostrar quem respondeu.

O modelo Notification é um reflexo direto da tabela notification na base de dados, com campos como event_id, notified_user, sent_at, seen_at, trigger_name e trigger_email. Na prática, estes dados permitem dizer ao utilizador "X acionou um SOS às Y horas", e identificar se essa notificação já foi "vista" ou tratada.

Outros modelos auxiliares podem existir para mapear respostas específicas da API (por exemplo, respostas do login que trazem token e userId), mas a filosofia mantém-se constante: são classes simples, com campos que correspondem aos nomes das chaves JSON e getters/setters mínimos, sem lógica adicional.

---

## 9. Fluxo completo de interação frontend–API–firmware

Do ponto de vista do cuidador, o fluxo normal passa por abrir a app, entrar na MainActivity e observar o estado da sua rede. Quando um contacto aciona o SOS, seja a partir do Arduino seja através da app, o backend cria um sos_event e atualiza o estado. A MainActivity, ao fazer polling dos endpoints de SOS ativo e notificações, acaba por receber esta informação e destacá-la na lista de ligações. O cuidador vê visualmente que um contacto está em emergência.

Se decidir responder, pressiona o botão apropriado na interface, o que aciona o endpoint /help/toggle e cria um help_event para o dispositivo. O Arduino, por sua vez, passa a ver help = true no endpoint /help/state/<device_id> e altera o padrão dos LEDs e do buzzer para transmitir ao utilizador final que a ajuda foi confirmada. Este ciclo fecha o loop entre o botão do idoso, a lógica da API e a experiência visual da app do cuidador.

---

## 10. Limitações, opções conscientes e possíveis melhorias no frontend

Tal como o firmware e a API, a aplicação Android equilibra simplicidade com funcionalidade. O uso de polling em vez de push notifications significa que o estado de SOS e de ajuda pode ter ligeiros atrasos na interface, dependentes dos intervalos configurados. No entanto, esta escolha foi coerente com a restante arquitetura (API REST simples, Arduino com polling) e foi suficiente para validar o sistema end-to-end.

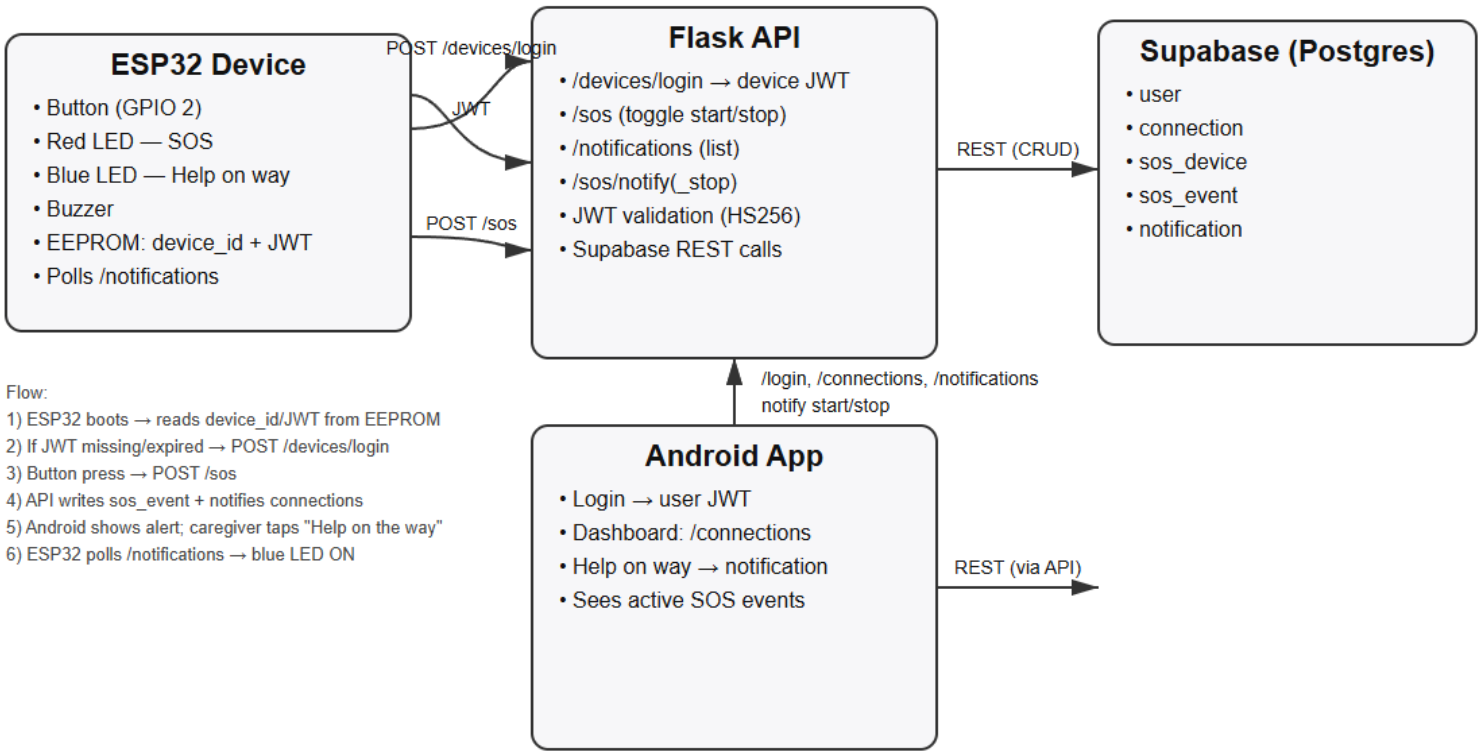
Outra limitação é que a aplicação depende fortemente do backend para pré-processar dados. Por exemplo, o enriquecimento das ligações é feito no servidor; a app recebe já os dados combinados. Este desenho é intencional, pois reduz a complexidade no frontend, mas torna a app dependente da estabilidade dessas respostas. Se o projeto fosse crescer, seria possível introduzir uma camada de view-models mais rica no Android, ou mesmo cache local de dados para funcionamento offline.

Em termos de estrutura, a app já segue uma separação sensata entre Activities, camada de rede e modelos, mas poderia, num cenário mais avançado, evoluir para um padrão arquitetural como MVVM, com ViewModels dedicados e LiveData/Flow para atualização automática da interface. Para o escopo atual, no entanto, a solução adotada é clara, legível e suficiente para demonstrar a integração completa com o firmware e a API.

---

# SafeSenior — End-to-End Architecture

ESP32 (Wi-Fi) ↔ Flask API ↔ Supabase (Postgres) ↔ Android App




Flow:

1) ESP32 boots → reads device_id/JWT from EEPROM
2) If JWT missing/expired → POST /devices/login
3) Button press → POST /sos
4) API writes sos_event + notifies connections
5) Android shows alert; caregiver taps "Help on the way"
6) ESP32 polls /notifications → blue LED ON