INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE RONDÔNIA CAMPUS ARIQUEMES
ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GABRIEL NASCIMENTO DE OLIVEIRA

Monitor mobile: aplicativo mobile para servidor Zabbix

Ariquemes
2025

GABRIEL NASCIMENTO DE OLIVEIRA

Monitor mobile: aplicativo mobile para servidor Zabbix

Relatório técnico entregue como Trabalho de Conclusão de Curso ao Instituto Federal de Educação, Ciência e Tecnologia de Rondônia – (IFRO), Campus Ariquemes, como requisito parcial para obtenção do grau de tecnólogo junto ao curso de Análise e Desenvolvimento de Sistemas.
Orientador: Andrey Alencar Quadros

Ariquemes
2025

GABRIEL NASCIMENTO DE OLIVEIRA

Monitor mobile: aplicativo mobile para servidor Zabbix

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Analista e Desenvolvedor de Sistemas" e aprovado em sua forma final pelo Curso de Tecnologia em Análise e Desenvolvimento de Sistemas.

Ariquemes, 05 de Janeiro de 2025.

Banca Examinadora:

ICP Brasil
Documento assinado digitalmente
ANDREY ALENCAR QUADROS
Data: 23/04/2025 15:27:36-0300
Verifique em https://validar.iti.gov.br

Prof. Me. Andrey Alencar Quadros
Instituto Federal de Rondônia

ICP Brasil
Documento assinado digitalmente
LUCIANO TOPOLNIAK
Data: 23/04/2025 15:22:14-0300
Verifique em https://validar.iti.gov.br

Prof. Me. Luciano Topolniak
Instituto Federal de Rondônia

ASSINADO DIGITALMENTE
VAGNER SCHOABA
A conformidade com a assinatura pode ser verificada em:
http://serpro.gov.br/assinador-digital
SERPRO

Prof. Me. Vagner Schoaba
Instituto Federal de Rondônia

Ficha catalográfica elaborada pelo Sistema Gerador de Ficha Catalográfica do IFRO, com dados informados pelo(a) próprio(a) autor(a).

Oliveira, Gabriel Nascimento de.
Monitor mobile: aplicativo mobile para servidor Zabbix / Gabriel Nascimento de Oliveira, Ariquemes-RO, 2025.
52 f.

Orientador(a): Prof. Me. Andrey Alencar Quadros.

Trabalho de Conclusão de Curso (Superior de Tecnologia em Análise e Desenvolvimento de Sistemas) – Instituto Federal de Educação, Ciência e Tecnologia de Rondônia - IFRO, Ariquemes-RO, 2025.

1. Flutter. 2. Zabbix. 3. API. I. Quadros, Andrey Alencar (orient.). II. Instituto Federal de Educação, Ciência e Tecnologia de Rondônia - IFRO. III. Título.

Bibliotecário(a) Responsável: Renilce Silva Morais, CRB-11/906 (Campus Ariquemes)

AGRADECIMENTOS

Em primeiro lugar, gostaria de agradecer a todos que caminharam comigo ao longo desta jornada acadêmica. Em especial, agradeço à minha família, cuja confiança em mim e nos meus sonhos permaneceram inabaláveis, mesmo quando eu ainda não tinha clareza sobre qual rumo seguir.

Sou igualmente grato aos professores e orientadores, que, com sua experiência e dedicação, me incentivaram a ir além dos limites do meu próprio conhecimento, despertando em mim um olhar crítico e a vontade de buscar evoluir sempre. Obrigado também aos colegas de curso, que dividiram comigo dúvidas, desafios e conquistas, transformando cada dia em uma oportunidade de crescimento mútuo.

Por fim, dedico este trabalho a todas as pessoas que, direta ou indiretamente, contribuíram para que este projeto se tornasse realidade. Seja por meio de palavras de apoio, suporte emocional ou colaboração prática, cada gesto de incentivo foi essencial para que eu chegasse até aqui. Carrego comigo um profundo sentimento de gratidão por cada um que fez parte dessa caminhada.

RESUMO

O presente projeto propõe o desenvolvimento de uma aplicação mobile capaz de se integrar ao sistema de monitoramento Zabbix, um dos mais utilizados do mundo. A proposta visa permitir que administradores e equipes de TI acompanhem o status dos serviços e hosts monitorados, independentemente de onde estejam, trazendo praticidade e conforto ao dia a dia profissional. Conceitos de comunicação via API, otimização de processos, gerenciamento de sessões online e programação mobile são explorados. Com foco em usabilidade, segurança e desempenho, a aplicação busca otimizar o processo de tomada de decisões em TI, oferecendo uma solução robusta e eficiente para profissionais que necessitam de monitoramento constante e respostas rápidas às demandas do ambiente computacional.

Palavras-chave: Flutter; ZABBIX; API.

ABSTRACT

This project proposes the development of a mobile application capable of seamlessly integrating with the Zabbix monitoring system, one of the most widely used platforms worldwide. The main goal is to enable administrators and IT teams to monitor the status of services and hosts from anywhere, providing convenience and efficiency in their daily tasks. The solution explores concepts of API-based communication, process optimization, online session management, and mobile programming. Emphasizing usability, security, and performance, the application aims to optimize the IT decision-making process by offering a robust and efficient tool for professionals who require constant monitoring and rapid responses to the demands of the computing environment.

Keywords: Flutter; ZABBIX; API.

LISTA DE FIGURAS

Figura 4.1 – Fluxo incial do usuário 20
Figura 4.2 – Páginas de incidentes e *hosts* 21
Figura 4.3 – Página de detalhes do *host* 22
Figura 4.4 – Página de atualização do incidente 23
Figura 5.1 – Fluxo inicial do usuário e menu de navegação 44
Figura 5.2 – Listagem de *hosts* e incidentes 45
Figura 5.3 – Detalhes do *host* 45
Figura 5.4 – Detalhes do item 46
Figura 5.5 – Detalhes e atualização de incidentes 47
Figura 5.6 – Listagem de informações diversas 47

LISTA DE ABREVIATURAS E SIGLAS

API	Interface de Programação de Aplicação
FCM	Firebase Cloud Messaging
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
JSON-RPC	JavaScript Object Notation - Remote Procedure Call
MVC	Model, View and Controller
NoSQL	No Structured Query Language
SMS	Short Message Service
TI	Tecnologia da Informação
URL	Uniform Resource Locator
UX	User Experience
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO 10
2 FUNDAMENTAÇÃO 12
2.1 Monitoramento de Sistemas 12
2.2 Zabbix 13
2.3 Desenvolvimento Mobile 15
2.4 APIs REST 16
3 METODOLOGIA 17
3.1 Metodologia de Desenvolvimento 17
3.2 Ferramentas e Tecnologias 17
3.3 Etapas do Desenvolvimento 18
4 DESENVOLVIMENTO 20
4.1 Design das interfaces e fluxos do usuário 20
4.2 Estrutura e Organização do Código 24
4.3 Sincronização de Dados 25
4.4 Autenticação via API 26
4.5 Implementação dos controladores 31
4.6 Exibindo gráficos no aplicativo 34
4.7 Notificações de Incidentes 36
4.8 Otimização das requisições da API 41
5 RESULTADOS 43
5.1 Funcionalidades Implementadas 43
5.2 Interfaces Desenvolvidas 43
5.3 Limitações e Funcionalidades Faltantes 48
6 CONCLUSÃO 49
REFERÊNCIAS 50

1 INTRODUÇÃO

No cenário atual, as organizações são cada vez mais dependentes de sistemas computacionais para realizar suas atividades de negócio, desde processos internos de gestão até a entrega de serviços aos clientes. O avanço da transformação digital e a crescente complexidade das infraestruturas de TI (Tecnologia da Informação) impõem a necessidade de monitoramento constante e eficiente. Administradores de TI são responsáveis por manter a saúde de suas redes, assegurando que os recursos da rede alcancem altos níveis de confidencialidade, integridade e disponibilidade. O monitoramento de componentes da infraestrutura de TI facilita a identificação de problemas em tempo real, permitindo respostas rápidas para minimizar impactos na produtividade das organizações (AL-MASAH; AL-SHARAFI, 2013).

Nesse contexto, o monitoramento de servidores e aplicações desponta como uma atividade estratégica, pois falhas ou indisponibilidades podem acarretar prejuízos financeiros, danos à reputação e insatisfação dos usuários finais.

Para atender a essas demandas, uma das ferramentas amplamente utilizadas no mercado é o Zabbix, que se destaca por sua abrangência e capacidade de monitoramento detalhado de recursos, serviços e aplicações.

De acordo com a documentação oficial (ZABBIX, 2025b), o Zabbix é uma solução de monitoramento empresarial de código aberto, projetada para rastrear o desempenho e a disponibilidade de infraestrutura de TI, incluindo servidores, redes, aplicações e serviços em nuvem. Altamente escalável e flexível, oferece ferramentas para coleta de dados em tempo real, análise, visualização e notificação de problemas, permitindo às organizações identificar e resolver incidentes rapidamente. Compatível com diversos protocolos e tecnologias, o Zabbix é amplamente utilizado para melhorar a eficiência operacional e garantir a continuidade dos negócios.

Por meio de painéis de controle e alertas configuráveis, o Zabbix oferece visibilidade em tempo real do estado da infraestrutura, possibilitando a identificação e a resolução de problemas de maneira proativa. Entretanto, a forma como as informações são tradicionalmente acessadas – em geral, via interfaces web ou aplicativos desktop – apresenta limitações quando se trata de mobilidade e agilidade na resposta a incidentes. Em ambientes onde o fator tempo é crítico, a ausência de soluções práticas para dispositivos móveis pode resultar em atrasos na tomada de decisão e na implementação de ações corretivas.

A justificativa para a realização deste projeto repousa na necessidade crescente de garantir disponibilidade e confiabilidade dos serviços em um cenário de competitividade acirrada. Ao disponibilizar uma versão mobile do Zabbix, busca-se oferecer uma ferramenta capaz de integrar as informações de monitoramento às rotinas de profissionais de TI que, muitas vezes, atuam em regime de plantão ou fora do ambiente corporativo. A mobilidade torna-se, portanto, um fator essencial para reduzir o tempo de resposta e mitigar riscos

associados à indisponibilidade de serviços, garantindo maior eficiência operacional.

Além disso, o desenvolvimento de um aplicativo mobile voltado ao monitoramento de TI acompanha a tendência de crescimento do uso de dispositivos móveis no contexto profissional. Segundo pesquisa (OLIVEIRA, 2023), no contexto profissional, o uso de dispositivos móveis tem se tornado uma tendência crescente, pois sua utilização proporciona maior eficiência, permitindo acesso rápido e remoto a ferramentas e informações críticas, otimizando fluxos de trabalho e promovendo a flexibilidade no desempenho das tarefas. Nesse sentido, o presente projeto não apenas supre uma lacuna existente no ecossistema de monitoramento de TI, mas também agrega valor às operações das organizações, contribuindo para um gerenciamento mais ágil, seguro e confiável de seus recursos.

Por fim, este trabalho busca oferecer uma contribuição relevante à comunidade de profissionais de TI, sendo totalmente gratuito e fortalecendo a cultura de monitoramento proativo e contínuo. A partir do desenvolvimento e da validação do aplicativo proposto, espera-se que as empresas sejam capazes de otimizar seus fluxos de resposta a incidentes, reduzir custos de inatividade e aprimorar a experiência de uso de seus sistemas.

A estrutura deste documento contempla, além desta Introdução, capítulos que abordam a fundamentação teórica acerca de monitoramento de sistemas e tecnologias mobile, a descrição detalhada da arquitetura proposta, o desenvolvimento prático do aplicativo, os resultados obtidos e, por fim, as considerações finais, onde serão discutidas as limitações e possíveis evoluções do projeto.

2 FUNDAMENTAÇÃO

2.1 MONITORAMENTO DE SISTEMAS

O monitoramento de sistemas é um conjunto de práticas e ferramentas que buscam observar, coletar e analisar indicadores de desempenho de serviços de TI, visando manter a estabilidade e o desempenho satisfatório de toda a infraestrutura tecnológica de uma organização. O Manual de Gestão de Riscos, Controles Internos e Integridade, disponibilizado pelo governo brasileiro (SOUSA; FREITAS, 2020), diz que a qualidade das informações em monitoramento deve ser avaliada por critérios como relevância, integralidade, adequação, concisão, consistência, clareza e padronização.

Segundo Beyer (BEYER *et al.*, 2016), monitoramento não se restringe a fornecer dados para análise de negócios ou segurança; seu papel central está em identificar anomalias e antecipar falhas, habilitando ações corretivas proativas e reduzindo a probabilidade de interrupções em sistemas críticos. Dessa forma, quando a autocorreção não é possível, um alerta bem-configurado aciona a intervenção humana apenas para incidentes genuinamente relevantes, minimizando ruídos e garantindo que o tempo de especialistas seja utilizado de forma eficiente.

Nesse sentido, o blog da Forrester (FORRESTER, 2025) destaca que a indisponibilidade de serviços essenciais pode ter impactos financeiros imediatos e significativos, além de prejudicar a reputação da empresa junto a clientes e parceiros de negócio. Exemplos como o episódio de downtime enfrentado pelo *New York Times* em 2013 evidenciam como a falta de um monitoramento eficaz pode acarretar perdas milionárias e comprometer o fluxo produtivo. Contudo, o maior impacto do tempo de inatividade em algumas situações não se limita somente à queda no preço das ações, mas também à perda de confiança dos clientes e à redução da vantagem competitiva da organização no mercado. Portanto, um ambiente de TI sem monitoramento sistemático está mais vulnerável a falhas inesperadas, resultando em custos elevados para reparos emergenciais e interrupções críticas nas operações.

Sendo assim, o mercado de soluções de monitoramento de sistemas se expandiu para suprir as diferentes demandas das organizações, desde pequenas empresas até grandes corporações com ambientes complexos. Nesse contexto, diversas plataformas foram criadas, cada uma apresentando enfoques distintos para coleta e análise de dados e, em meio a tantas possibilidades, a escolha da ferramenta de monitoramento adequada deve considerar critérios como facilidade de uso, escalabilidade, custo, suporte à comunidade, recursos de alerta e automação, além da compatibilidade com diferentes protocolos e formatos de dados. Em muitas empresas, fatores como o perfil da equipe de TI, a complexidade da arquitetura de sistemas e a cultura interna de gestão de incidentes acabam sendo determinantes.

O monitoramento de sistemas desempenha um papel ainda mais crítico em um

cenário em que o trabalho híbrido se consolida como uma tendência mundial, combinando atuação presencial e remota de equipes. De acordo com pesquisa divulgada pela Unisys em parceria com a HFS Research, intitulada *Surviving to Thriving in Hybrid Work* (GRIES, 2023), organizações em todo o globo vêm buscando soluções que ofereçam maior estabilidade e flexibilidade em suas infraestruturas de TI, garantindo o bom desempenho das operações mesmo quando parte dos colaboradores atua fora do ambiente corporativo tradicional. Nesse contexto, torna-se essencial dispor de ferramentas de monitoramento capazes de acompanhar em tempo real o uso de recursos computacionais e detectar possíveis falhas em serviços críticos, de modo a manter a produtividade dos profissionais e assegurar a continuidade dos negócios.

2.2 ZABBIX

Com a crescente complexidade dos ambientes de Tecnologia da Informação, onde aplicações e serviços precisam permanecer disponíveis e com alto desempenho, o monitoramento de sistemas consolida-se como um componente estratégico. A escolha de ferramentas adequadas impacta diretamente no tempo de resposta a incidentes, na prevenção de problemas e, por consequência, na satisfação dos usuários finais. Entre as soluções disponíveis no mercado, o Zabbix destaca-se por sua robustez, flexibilidade, capacidade de integração e ampla comunidade de suporte.

O Zabbix reúne funcionalidades que permitem monitorar desde servidores e dispositivos de rede até aplicações específicas, oferecendo suporte a diversos protocolos de coleta de dados. Aguiar (AGUIAR, 2017) ressalta a utilização do Zabbix pela Força Aérea Brasileira por tamanha robustez e qualidade, além de ser um *software opensource*. Outra característica fundamental é sua API (Interface de Programação de Aplicação), que possibilita a criação de integrações personalizadas e a adoção de ferramentas móveis, permitindo um acompanhamento contínuo de métricas e eventos em tempo real. Essa abordagem proporciona maior agilidade na resolução de incidentes, pois os profissionais de TI podem ser notificados rapidamente e tomar ações corretivas a partir de qualquer lugar, reduzindo o tempo de inatividade e, consequentemente, os prejuízos causados pela indisponibilidade de serviços.

No presente projeto, a escolha pelo Zabbix 6.4 foi motivada pela maturidade da plataforma, pela gama abrangente de funcionalidades e, sobretudo, pela possibilidade de integração via API, que viabiliza o desenvolvimento de aplicativos específicos para necessidades de monitoramento. Esse cenário contribui para melhorar a eficiência operacional das equipes de suporte, pois as informações críticas ficam acessíveis em dispositivos móveis, agilizando a detecção e a correção de problemas. Além disso, a interface relativamente amigável do Zabbix resulta em uma curva de aprendizado menos íngreme, facilitando a adesão de novas equipes e promovendo boas práticas de gestão de infraestrutura.

A unificação de todas as métricas em uma única interface de gerenciamento permite

o acesso centralizado a dados de desempenho, alertas e relatórios de disponibilidade, eliminando a necessidade de alternar entre múltiplas ferramentas ou painéis para compreender o estado geral da infraestrutura. Segundo a documentação oficial do Zabbix (ZABBIX, 2025b), algumas das funcionalidades mais relevantes incluem:

- **Coleta de métricas diversificada**: a plataforma suporta tanto o modelo *agent-based* (por meio de agentes instalados em cada *host*) quanto o *agentless*, o que permite monitorar componentes de rede, sistemas operacionais e aplicações de maneira flexível.
- **Alertas e notificações personalizáveis**: com base em gatilhos (ou *triggers*) configurados, o Zabbix envia alertas via e-mail, SMS (Short Message Service), *messaging apps* ou integrações com ferramentas de *chatops*, adaptando-se ao fluxo de trabalho de cada equipe.
- **Escalabilidade e alta disponibilidade**: o Zabbix se adapta bem tanto a pequenas implantações quanto a grandes ambientes distribuídos, podendo monitorar milhares de nós sem perdas significativas de desempenho.
- **Gráficos e relatórios avançados**: a ferramenta oferece diversos recursos de visualização, tais como gráficos em tempo real, mapas de topologia de rede e relatórios históricos, que auxiliam na análise de tendências e na identificação de possíveis gargalos.

Outro ponto central é a capacidade de correlacionar eventos, agrupando alertas e simplificando a identificação de incidentes críticos. Essa funcionalidade é particularmente vantajosa em cenários de microserviços ou ambientes altamente distribuídos, onde múltiplas falhas podem ocorrer em cadeia, dificultando o entendimento das causas-raiz.

Para gerenciar o monitoramento de forma organizada, o Zabbix adota uma estrutura de dados bem definida, composta por elementos que facilitam o agrupamento e a padronização de configurações:

- **Incidentes (ou *Problems*)**: no Zabbix, um incidente ocorre quando um *trigger* é acionado devido a uma condição fora do estado normal, como o alto uso de CPU ou a indisponibilidade de um serviço. Esses incidentes são registrados e podem ser consultados em painéis de eventos, onde é possível filtrar e analisar a criticidade, o status de resolução e a data de ocorrência.
- **Hosts**: são os dispositivos ou serviços que se deseja monitorar, tais como servidores físicos, máquinas virtuais, *containers*, roteadores ou *switches*. Cada *host* possui um conjunto de métricas específicas, como o uso de memória, disponibilidade de portas, latência de rede e entre outras, todas definidas conforme suas características de *hardware* e *software*.
- **Grupos de Hosts**: servem para categorizar e organizar os *hosts* de forma lógica, facilitando o gerenciamento em ambientes complexos. Por exemplo, é

possível agrupar *hosts* por departamentos, por localizações geográficas ou por funcionalidades, cabendo ao administrador organizar da forma desejada.

- **Templates**: são conjuntos pré-configurados de itens, *triggers* e gráficos que podem ser aplicados a um ou mais *hosts*. Eles padronizam o processo de monitoramento e reduzem o esforço de configuração, pois as mesmas métricas podem ser replicadas para diversos *hosts* que compartilhem características semelhantes.

Assim, a relação entre *hosts*, grupos e *templates* proporciona uma visão hierárquica e padronizada do ambiente de TI, tornando mais simples a implementação inicial do monitoramento e a manutenção contínua das configurações. Já os incidentes (*problems*) criados a partir de *triggers* são a forma como o Zabbix gera alertas e registra o histórico de falhas, servindo de base para a tomada de decisão e a implementação de ações corretivas.

2.3 DESENVOLVIMENTO MOBILE

Com a crescente popularidade dos dispositivos móveis no século XXI, eles se tornaram ferramentas indispensáveis para bilhões de pessoas em tarefas que vão desde o lazer até atividades profissionais. Segundo pesquisa realizada pela Fundação Getulio Vargas (MEIRELLES, 2024), em maio de 2024, o Brasil possuia cerca de 480 milhões de dispositivos digitais. Dentro desse contexto, dispositivos como smartphones, tablets e laptops desempenham um papel central no desenvolvimento de uma infinidades de atividades.

Historicamente, o desenvolvimento de aplicativos móveis enfrentava limitações significativas, especialmente porque cada plataforma exigia ferramentas e linguagens de programação próprias. ZammetTI (ZAMMETTI, 2020) afima em seu livro a necessidade de criar códigos completamente distintos para cada sistema operacional para que fosse possível lançar o mesmo aplicativo em diferentes plataformas. Com o passar do tempo, a evolução tecnológica levou à redução do número de plataformas dominantes e à criação de ferramentas que simplificaram o desenvolvimento multiplataforma, como *React Native* e Flutter, que permitem criar aplicações para Android e iOS a partir de uma única base de código.

Lançado oficialmente em 2018 pelo Google, o Flutter oferece uma abordagem única para o desenvolvimento de aplicativos móveis multiplataforma. Através de uma estrutura baseada em *widgets* e do uso da linguagem Dart, o Flutter facilita o desenvolvimento e garante alto uma experiência de usuário fluida. O framework combina um motor de renderização, a biblioteca Skia, e abstrações que tornam simples acessar funcionalidades de dispositivos, como câmeras e sensores, independentemente do sistema operacional.

A escolha do Flutter para este projeto foi estratégica, pois ele atende às exigências de portabilidade e desempenho, enquanto reduz a complexidade do desenvolvimento multiplataforma. Durante a implementação, foram utilizados ambientes de desenvolvimento como Visual Studio Code e Android Studio, que, em conjunto, otimizaram o fluxo de

trabalho, desde a codificação até os testes e o *deploy*.

No fim, esta escolha permitiu unir design, performance e funcionalidade, resultando em uma aplicação que oferece uma experiência ágil e confiável para o monitoramento e gestão de sistemas. A integração de tecnologias modernas com boas práticas de desenvolvimento não só resolveu os desafios propostos, mas também estabeleceu uma base sólida para a evolução contínua do aplicativo.

2.4 APIS REST

As *APIs REST* (Representational State Transfer) representam uma abordagem arquitetural que utiliza os métodos do protocolo HTTP (Hypertext Transfer Protocol) para criar interfaces escaláveis e de fácil manutenção (HAT, 2023). De acordo com essa definição, as *APIs REST* facilitam a comunicação entre sistemas ao promover o intercâmbio de dados em formatos amplamente utilizados, como JSON (JavaScript Object Notation) ou XML (Extensible Markup Language). Essas interfaces são independentes de plataforma ou linguagem de programação, o que as torna ideais para conectar diferentes sistemas. No contexto deste projeto, as *APIs RESTful* desempenham um papel essencial ao conectar o servidor Zabbix ao aplicativo móvel, permitindo a consulta de métricas, a manipulação de incidentes e a automação de tarefas de monitoramento.

O Zabbix oferece uma API que segue princípios próximos ao modelo REST, embora internamente utilize JSON-RPC (JavaScript Object Notation - Remote Procedure Call) em suas chamadas (ZABBIX, 2025a). Quando um usuário acessa o aplicativo móvel, ocorre uma etapa de autenticação em que as credenciais são verificadas e o servidor retorna um *token* de sessão, o qual deve ser enviado em cada requisição subsequente, mantendo a segurança do sistema. Em seguida, é possível efetuar consultas de métricas, filtrar dados com base em parâmetros específicos e receber apenas as informações pertinentes ao contexto de uso, evitando sobrecarregar o tráfego de rede. Do lado do aplicativo, essa abordagem permite que as equipes de TI monitorem o ambiente em tempo real, independentemente de estarem presentes fisicamente no local onde se encontra a infraestrutura.

3 METODOLOGIA

3.1 METODOLOGIA DE DESENVOLVIMENTO

A organização das atividades de criação e evolução do aplicativo seguiu o modelo de desenvolvimento *Waterfall* (ou Cascata), cuja abordagem linear foi adotada em razão da clareza quanto às funcionalidades que o sistema deveria contemplar, definidas antes mesmo do início efetivo da implementação. Nesse tipo de processo, cada fase se baseia na conclusão e na aprovação dos resultados da etapa anterior, tornando a gestão de requisitos mais simples em projetos que possuem um escopo bem delimitado.

A primeira fase consistiu na elaboração e análise dos requisitos funcionais e não funcionais do aplicativo, incluindo a definição das métricas a serem exibidas, das telas necessárias e das formas de interação do usuário com as funcionalidades. Em seguida, concentrou-se a atenção no design das interfaces, contemplando a organização de elementos visuais, a navegação entre telas e a aplicação de boas práticas de UX (User Experience). Após a validação das propostas de *layout*, deu-se início à fase de implementação, na qual cada funcionalidade planejada foi codificada e integrada ao projeto.

Conforme o desenvolvimento de cada funcionalidade, eram realizados testes com um servidor local do Zabbix, permitindo verificar o correto consumo dos dados e a exibição de métricas em tempo real. Assim, o desenvolvimento ocorreu de forma simples e objetiva, garantindo que cada entrega fosse validada antes do avanço para a próxima fase, minimizando retrabalhos e assegurando a aderência às especificações previamente estabelecidas.

3.2 FERRAMENTAS E TECNOLOGIAS

O projeto foi construído utilizando Flutter, que disponibiliza uma alta gama de componentes de interface ("*widgets*") e oferece recursos como *Hot reload*, facilitando ajustes contínuos durante o processo de codificação. Além dos componentes nativos, foram adotadas bibliotecas adicionais que aprimoram a qualidade e funcionalidade do aplicativo, destacando-se pelos seus papéis específicos no desenvolvimento.

Para lidar com requisições HTTP e tratar possíveis erros, foi utilizada a biblioteca Dio (FLUTTER, 2025b). Utilizado diversas vezes nas requisições para a API, o Dio também permite gerenciar "timeouts" e tratar respostas de forma segura e padronizada. Essa flexibilidade é essencial para o manuseio de cabeçalhos e parâmetros de autenticação, simplificando a integração com a API do Zabbix.

No âmbito do gerenciamento de estado e navegação, a biblioteca *GetX* (FLUTTER, 2025d) desempenhou um papel central, proporcionando uma organização mais clara das regras de negócio e da lógica de controle. Foram adotados os padrões de rotas e controladores oferecidos pelo *GetX*, e também foi utilizado para o gerenciamento de estado de

algumas informações.

Em relação à persistência de dados, o *Hive* (FLUTTER, 2025e) foi escolhido como solução principal para armazenamento local e "off-line". Por ser leve e baseado em NoSQL (No Structured Query Language), o *Hive* elimina a necessidade de configurações complexas de servidor, permitindo leituras e escritas rápidas diretamente no dispositivo. Auxiliando o desenvolvimento do sistema de autenticação da aplicação, mantendo as credenciais no próprio dispositivo.

As notificações de novos incidentes foram feitas utilizando dois pacotes em conjunto. O primeiro, *background_fetch* (FLUTTER, 2025a), responsável por criar os serviços em segundo plano para buscar novos incidentes no servidor Zabbix. Já o pacote *flutter_local_notifications* (FLUTTER, 2025c) foi utilizado para criar as notificações locais apresentadas ao usuário.

Para assegurar a confiabilidade do aplicativo, foi configurado um servidor local do Zabbix na versão 6.4, que viabilizou a simulação de cenários reais de monitoramento. Esse servidor, executado em ambiente de desenvolvimento, permitiu a criação de *hosts*, grupos, *triggers* e incidentes controlados, fornecendo dados para o refinamento das funcionalidades do aplicativo.

Para complementar a experiência do usuário, funcionalidades de compartilhamento e identidade visual também foram consideradas. O componente *share_plus* foi incorporado para viabilizar o compartilhamento dos incidentes com outras aplicações instaladas no dispositivo, facilitando a colaboração entre equipes de suporte ao permitir o envio rápido de informações sobre alertas ou falhas detectadas. Já no que diz respeito à identidade visual, foram utilizados os pacotes *flutter_launcher_icons* e *flutter_native_splash* para configurar ícones personalizados e telas de "*splash*".

Com relação a testes em dispositivos móveis durante o desenvolvimento, foi feito o uso de um emulador Android, criado via *Android Studio*, garantindo que possíveis erros de layout, problemas de desempenho ou incompatibilidades de versão fossem identificados precocemente. Após essa etapa, o VS Code serviu como a principal ferramenta de edição e depuração do código, integrando-se ao emulador para execução e observação de qualquer alteração feita.

3.3 ETAPAS DO DESENVOLVIMENTO

A etapa inicial do desenvolvimento buscou compreender as necessidades e expectativas em relação ao monitoramento dos ambientes via aplicativo móvel. Foram descritas as funcionalidades a serem suportadas, incluindo a possibilidade de listar *hosts*, itens, incidentes, templates e grupos de *hosts*, bem como a exibição de uma *dashboard* de incidentes na qual o usuário pudesse filtrar por severidade, visualizar gráficos de ocorrência e tomar ações imediatas, como reconhecer eventos, alterar prioridades ou inserir comentários. Essas exigências estruturaram as bases funcionais do projeto, definindo também o escopo de

dados a serem obtidos via API.

Com as funcionalidades bem definidas, a etapa de modelagem priorizou a criação de telas e fluxos de navegação. A interface foi planejada para que se assemelhace o máximo possível com a versão *web* do Zabbix, porém adaptada para o uso em aparelhos móveis. Outro objetivo era apresentar de forma clara as informações essenciais, como lista de incidentes.

Em paralelo, foram delineadas telas adicionais para exibir os *hosts*, itens, templates e grupos, visando fornecer aos usuários acesso abrangente aos componentes monitorados. Durante essa fase, optou-se por definir classes específicas para representar cada tipo de dado retornado pela API, facilitando o tratamento das informações no código e promovendo uma estrutura mais organizada.

Durante a implementação foram desenvolvidos, inicialmente, os componentes relacionados à *dashboard* de incidentes, incluindo filtros de severidade, gráficos de ocorrência e botões de ação. Na sequência, codificaram-se as telas para visualização dos *hosts*, itens, templates e grupos, com as devidas integrações à API do Zabbix.

A adoção de classes específicas para cada entidade retornada pela API agilizou o processo de consumo de dados, pois as conversões de dados e unidades de medidas necessárias eram centralizadas em um único local, reduzindo a chance de erros. Além disso, a fase de implementação considerou recursos de experiência do usuário, como feedbacks visuais e mensagens de confirmação, voltados a tornar a navegação mais intuitiva.

Por fim, os testes foram executados em um ambiente que simulava cenários reais de monitoramento. Validaram-se as principais funcionalidades, incluindo a listagem de dados e a interação com incidentes . Os testes contemplaram tanto verificações funcionais, para assegurar que cada ação do usuário produzia o resultado esperado, quanto análises de desempenho e usabilidade, visando identificar pontos de melhoria no fluxo de navegação.

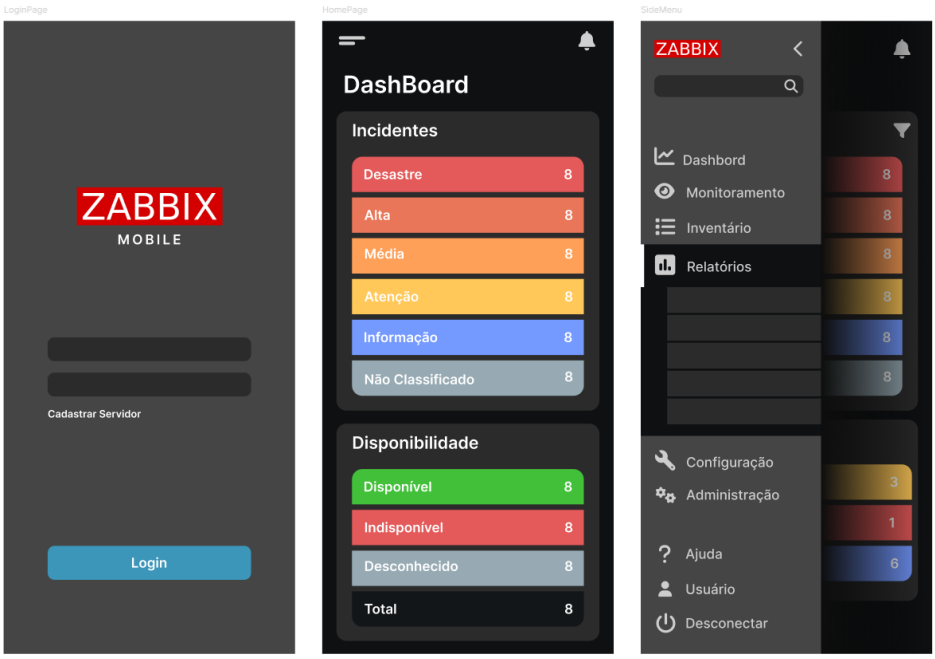
4 DESENVOLVIMENTO

4.1 DESIGN DAS INTERFACES E FLUXOS DO USUÁRIO

A primeira etapa do projeto consistiu em definir as funcionalidades que fariam parte da aplicação. Para tornar essa etapa mais compreensível, o desenvolvimento de protótipos das interfaces foi iniciado em conjunto, não necessariamente sendo o design final das telas do aplicativo, facilitando o entendimento do fluxo de ações que o usuário realizaria dentro do aplicativo. Todas as telas foram criadas na ferramenta Figma.

Na Figura 4.1, apresenta-se o primeiro contato do usuário com o sistema, começando por uma tela de login simples e objetiva. Após a autenticação, o usuário é direcionado à *dashboard*, onde pode visualizar uma lista com todos os incidentes ativos do servidor, organizados por severidade. Além disso, a *dashboard* exibe uma relação de *hosts* reconhecidos, categorizados pelo tipo de interface. Na última interface destacada está o planejamento do *drawer* (menu lateral), que oferece acesso rápido e intuitivo às principais telas da aplicação, proporcionando uma navegação ágil e eficiente ao usuário.

Figura 4.1 – Fluxo incial do usuário

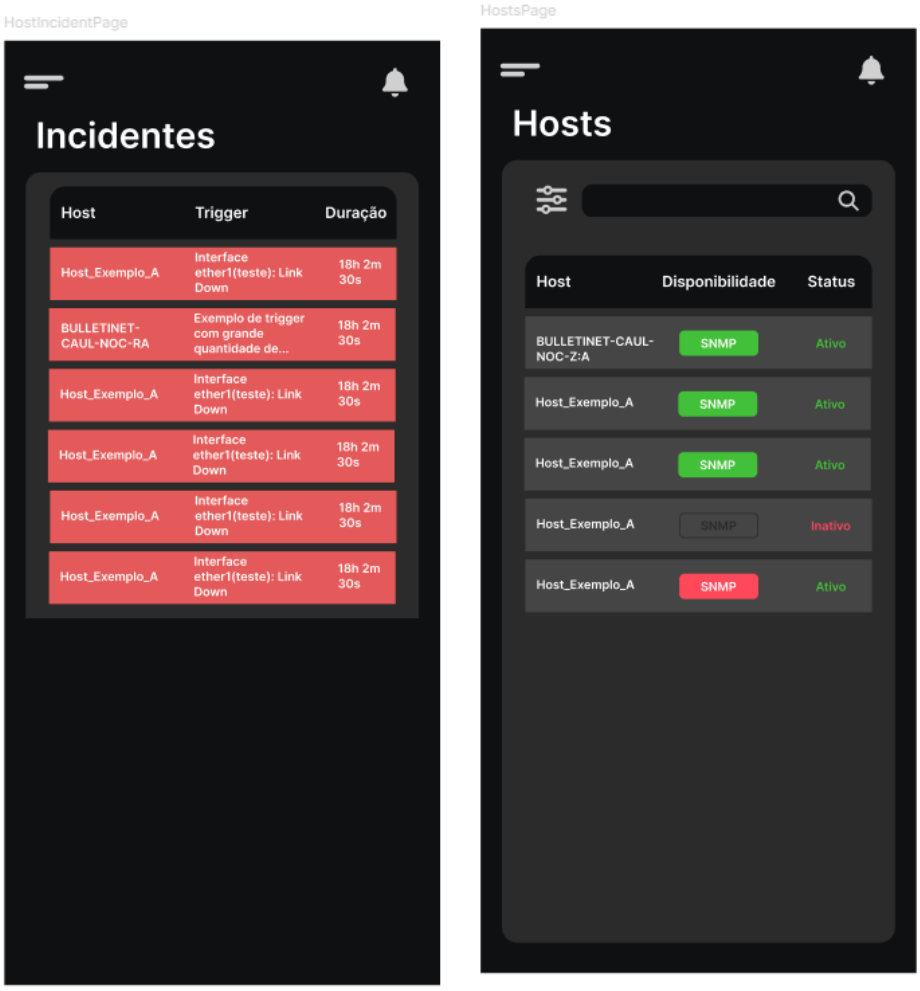


Fonte: Elaborado pelo autor, 2024

Na Figura 4.2, é apresentada a tela de incidentes, onde os problemas ativos do servidor são exibidos em formato de lista. A interface mantém o mesmo esquema de cores utilizado no sistema Zabbix, permitindo que usuários já familiarizados com a plataforma reconheçam facilmente a severidade de cada evento. Cada incidente é listado com informações detalhadas, como a severidade, o nome do host associado, o tempo de atividade do incidente e ícones que indicam atualizações relacionadas ao evento.

Na mesma imagem, também está a tela de listagem de hosts. Um campo de pesquisa foi incluído, permitindo que o usuário localize hosts pelo nome ou endereço IP (Internet Protocol). Cada item da lista exibe detalhes relevantes, como o nome do host, o status atual e, quando disponível, o endereço IP e status da interface principal de monitoramento.

Figura 4.2 – Páginas de incidentes e *hosts*



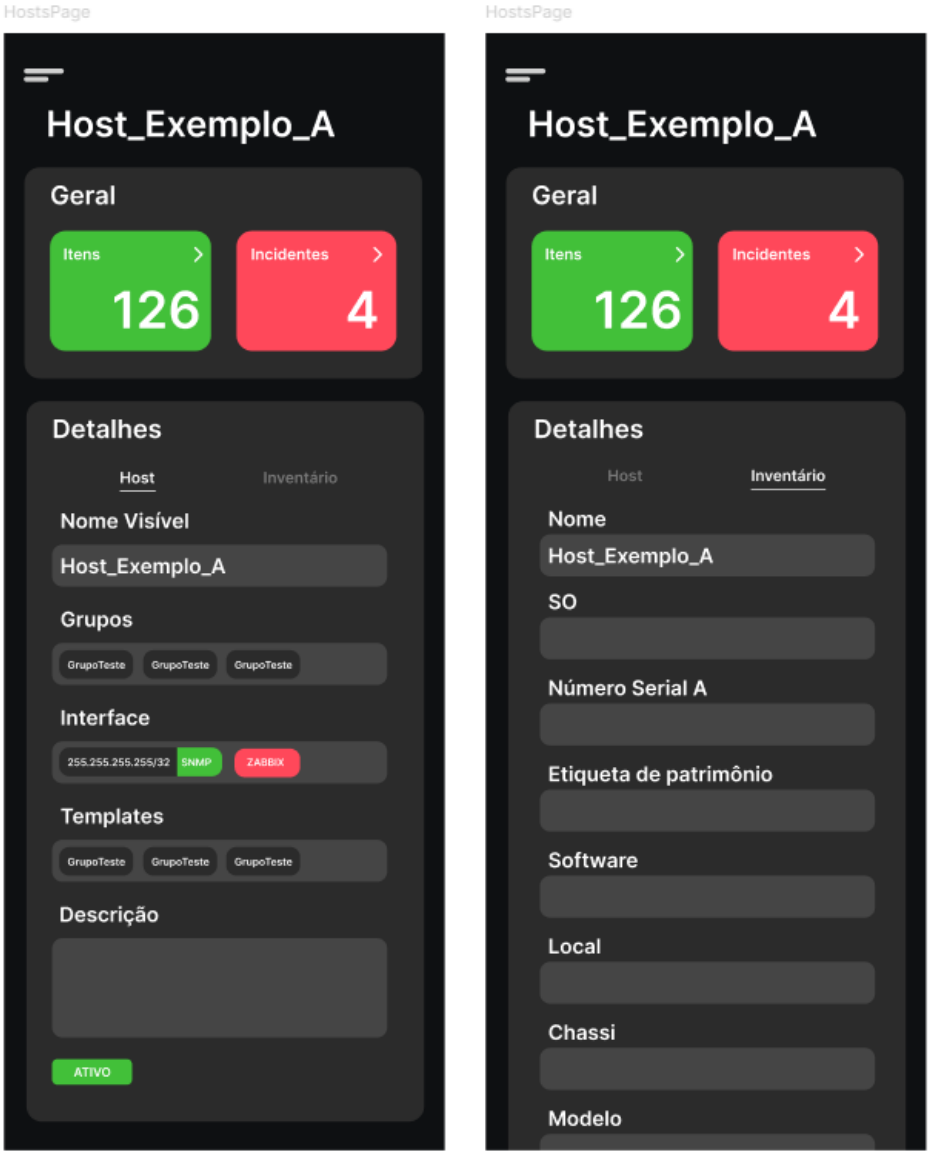
Fonte: Elaborado pelo autor, 2024

Na Figura 4.3, é apresentada a página de detalhes do *host*, cuja interface foi dividida em duas partes principais. A primeira parte é composta por dois *cards* que fornecem informações resumidas: a quantidade de itens monitorados pelo *host* e o número de incidentes ativos. Cada *card* é interativo, permitindo que o usuário acesse uma tela com listas detalhadas de itens monitorados ou incidentes ativos, dependendo do *card* selecionado.

A segunda parte da interface oferece um nível maior de detalhamento, com duas guias: *Host* e *Inventário*. Na guia *Host*, são exibidas diversas informações relacionadas ao *host* selecionado, incluindo o nome, os grupos aos quais pertence, os *templates* vinculados, as interfaces monitoradas, a descrição e o status do *host*.

Já a guia *Inventário* é dedicada à apresentação de dados específicos do inventário do *host*, como número de patrimônio, modelo, chassi e outras informações relevantes. É importante destacar que os dados do inventário só são exibidos se estiverem preenchidos no servidor. Caso contrário, uma mensagem informativa é exibida, indicando que não há informações disponíveis no momento.

Figura 4.3 – Página de detalhes do *host*



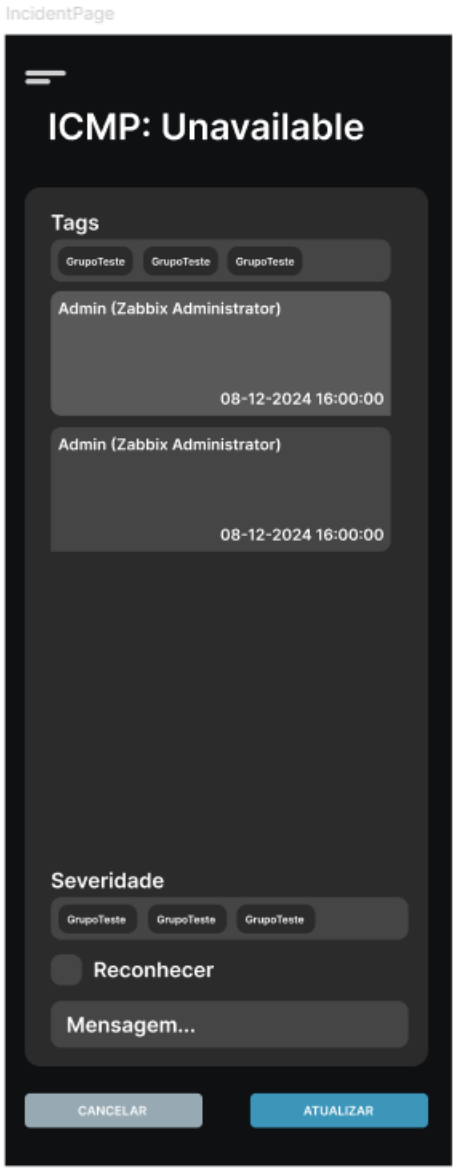
Fonte: Elaborado pelo autor, 2024

Na página de incidentes, foi planejada uma interface que permite ao usuário interagir diretamente com as atualizações de um incidente, oferecendo funcionalidades como enviar mensagens, reconhecer o incidente e até mesmo alterar sua severidade. Na Figura 4.4, é apresentada uma tela que simula um *chat* de texto, onde todas as mensagens relacionadas a um incidente são exibidas, acompanhadas do nome do usuário que as enviou.

Além das mensagens, o *chat* também registra as ações tomadas pelos usuários.

Por exemplo, caso a severidade de um incidente seja alterada, uma entrada específica é adicionada ao *chat*, sem conteúdo textual, apenas com um ícone representando a ação realizada. Esse comportamento foi inspirado no sistema Zabbix, garantindo familiaridade para os usuários habituados com a plataforma.

Figura 4.4 – Página de atualização do incidente



Fonte: Elaborado pelo autor, 2024

As telas destinadas à listagem de dados seguem um padrão visual consistente, com a utilização de *cards* para exibir as informações de forma organizada e acessível. Essas telas incluem listas de grupos de *hosts*, *templates*, gráficos e itens, cada uma adaptada ao tipo de dado apresentado, mas mantendo a uniformidade na estrutura e no layout. O uso dos *cards* permite que as informações sejam facilmente visualizadas, destacando os elementos mais importantes e proporcionando uma navegação intuitiva para o usuário.

4.2 ESTRUTURA E ORGANIZAÇÃO DO CÓDIGO

O projeto foi idealizado com o objetivo de ser de fácil manutenção e extensão, permitindo adaptações rápidas conforme novas funcionalidades são introduzidas no Zabbix. Para atingir este objetivo, foi preciso definir uma arquitetura para o sistema, e o modelo MVC (Model, View and Controller) se mostrou ideal para neste caso, auxiliando na separação das responsabilidades entre a comunicação com a API, os modelos do projeto e as interfaces de usuário.

Além da divisão proporcionada pelo MVC, a divisão estrutural do código foi aprimorada com a criação de duas pastas auxiliares: Jsons e Core. A primeira, como o nome sugere, serve apenas para armazenar os arquivos JSON utilizados no projeto, otimizando o processo de montagem das requisições para a API. Já a pasta Core centralizou elementos essenciais e reutilizáveis do sistema, incluindo arquivos de configuração das rotas, definições de temas e cores, variáveis constantes e funções utilitárias para o tratamento e formatação dos dados retornados pela API.

Essa abordagem proporciona uma série de benefícios à manutenção e evolução do projeto. Sempre que o Zabbix modificar sua estrutura de dados ou houver a necessidade de novos *endpoints*, basta atualizar os modelos e as classes *controllers* responsáveis, mantendo o *frontend* relativamente imune a alterações no *backend*. O quadro 4.1 mostra a árvore de diretórios do projeto, ilustrando como o código foi estruturado e separado.

Listagem de Código 4.1 – Árvore de diretórios do projeto

```
|src
|---controllers
|   |---api
|   |---chart
|   |---event
|   |   |---form
|   |---group
|   |---host
|   |   |---forms
|   |---host_interface
|   |   |---forms
|   |---item
|   |---problem
|   |---template
|   |---trigger
|   |---user
|---core
|   |---colors
|   |---routes
|   |---themes
|   |---utils
|---models
```

```
23 |   |---adapters
24 |---views
25 |---charts
26 |---dashboard
27 |   |---components
28 |---groups
29 |   |---components
30 |---hosts
31 |   |---hosts_page
32 |   |   |---components
33 |   |---host_charts
34 |   |   |---components
35 |   |---host_page
36 |   |---components
37 |   |---forms
38 |   |---host_infos
39 |   |   |---form
40 |   |---host_inventory
41 |   |---form
42 |---incidents
43 |   |---components
44 |---items
45 |   |---components
46 |---login
47 |   |---components
48 |   |---form
49 |---problems
50 |   |---components
51 |---shortly
52 |---templates
53 |   |---components
54 |---users
```

Fonte: Elaborado pelo autor, 2024

4.3 SINCRONIZAÇÃO DE DADOS

Diferentemente de um processo em tempo real (via *websockets* ou outra tecnologia reativa), a aplicação atualiza os dados através de requisições pontuais, solicitadas sempre que o usuário realiza uma ação de *refresh* ou navega para uma tela que exija conteúdo atualizado.

Isso significa que, ao acessar a lista de incidentes ou ao abrir detalhes de um *host*, o aplicativo encaminha uma requisição ao servidor Zabbix, aguarda a resposta JSON e, então, sincroniza os modelos locais para refletir as últimas informações. Esse processo de “atualização por demanda” evita consumo desnecessário de recursos do aparelho e

também diminui a sobrecarga do servidor, ao mesmo tempo em que mantém a experiência do usuário sob controle, já que ele decide quando quer receber os dados mais recentes.

4.4 AUTENTICAÇÃO VIA API

O processo de acesso ao servidor via API inicia assim que o usuário insere suas credenciais na tela de login do aplicativo. Utilizando o pacote *GetX*, as variáveis responsáveis por armazenar os dados do usuário, como login, senha e a URL do servidor, foram definidas como reativas. Isso significa que elas podem ser monitoradas automaticamente por toda a aplicação e qualquer alteração em seus valores será propagada para todas as partes do projeto.

Esse comportamento é obtido ao utilizar o método .obs do *GetX*, que transforma as variáveis em observáveis, conforme mostrado no quadro 4.2. Isso facilita o uso e compartilhamento de informações de acesso do usuário em outros arquivos ou módulos do aplicativo. Sempre que essas variáveis forem modificadas, as mudanças serão refletidas em tempo real em qualquer parte da aplicação que dependa delas, sem a necessidade de código adicional para sincronização.

Listagem de Código 4.2 – Declaração de variáveis observáveis

```
import 'package:get/get.dart';
import 'dart:convert';
import 'package:dio/dio.dart';

class UserApi extends GetxController {
  final dio = Dio();
  final userDataController = UserDataController();
  RxString apicode = ''.obs;
  RxString url = ''.obs;
  RxString server = ''.obs;
  RxString apiVersion = ''.obs;
  RxString usuario = ''.obs;
  RxString senha = ''.obs;

  [...]
```

Fonte: Elaborado pelo autor, 2024

Nesta mesma classe há o método *login*, responsável por estabelecer a conexão do aplicativo com o servidor Zabbix através de uma chamada à API. Ele também é responsável por salvar as informações de acesso do usuário no dispositivo onde a aplicação está instalada. Além disso, ele conta com o tratamento detalhado de erros para oferecer uma experiência mais robusta e informativa ao usuário.

Inicialmente, o método carrega um arquivo JSON localizado nos *assets* do projeto, que contém a estrutura necessária para a solicitação à API, em seguida, é decodificado para um objeto manipulável. Os dados de autenticação, como usuário e senha, são inseridos dinamicamente nesse objeto, garantindo que a requisição enviada ao servidor contenha os dados de acesso fornecidos pelo usuário.

Listagem de Código 4.3 – Declaração do método Login()

```
Future<void> login(String user, String pass, String urlSite) async {
  String json = await rootBundle.loadString('assets/json/user/login.json');
  String result = '';
  final jsonRequest = await jsonDecode(json);
  jsonRequest["params"]["username"] = user;
  jsonRequest["params"]["password"] = pass;

  [...]
```

Fonte: Elaborado pelo autor, 2024

A chamada à API utiliza o pacote Dio, com URL montada dinamicamente, envio de JSON no corpo da requisição e cabeçalhos apropriados. A requisição possui *timeout* de 5 segundos para evitar bloqueios. Se a resposta for bem-sucedida (*status* 200) e conter o campo "*result*", os dados são extraídos e armazenados. Uma mensagem de sucesso é exibida ao usuário com *Get.snackbar*, e as informações essenciais (*token*, URL, usuário e senha) são salvas no controlador *userDataController*. O método finaliza com um *update*() para notificar atualizações nos dados.

Listagem de Código 4.4 – Requisição de login via API

```
[...]

  try {
    final response = await dio
        .post('$urlSite/api_jsonrpc.php',
            data: jsonRequest,
            options: Options(headers: {
              'Content-Type': 'application/json',
            }))
        .timeout(const Duration(seconds: 5));

    if (response.statusCode == 200) {
      if (response.data["result"] != null) {
        result = response.data["result"].toString();
      } else {
        throw Exception('Erro ao buscar dados da API');
      }
      Get.snackbar('Sucesso', 'Usuario logado com sucesso.',
```

```
19          backgroundColor: const Color.fromARGB(255, 34, 95, 29),
20          duration: const Duration(seconds: 5),
21          snackPosition: SnackPosition.BOTTOM);
22      url.value = urlSite;
23      apicode.value = result;
24      usuario.value = user;
25      senha.value = pass;
26      await userDataController.saveUserLoginInfo(
27          result, urlSite, user, pass, '');
28      update();
29    }
30
31 [...]
```

Fonte: Elaborado pelo autor, 2024

O tratamento de erros foi projetado para capturar uma ampla gama de possíveis problemas. Se a solicitação ultrapassar o tempo limite, uma mensagem é exibida informando que o servidor não respondeu no tempo esperado. Para erros específicos, como URLs malformadas ou protocolo de criptografia ausente, mensagens detalhadas são apresentadas ao usuário, explicando o problema e oferecendo exemplos de como corrigir.

Listagem de Código 4.5 – Tratamento de erros durante a autenticação

```
1  [...]
2
3  } on TimeoutException {
4    Get.snackbar('Erro',
5        'O servidor cadastrado nao respondeu, verifique se um endereco
     valido foi preenchido!',
6        duration: const Duration(seconds: 5),
7        snackPosition: SnackPosition.BOTTOM);
8    return;
9  } catch (e) {
10   print('Deu erro samerda: $e');
11   if (e.toString().contains('No host specified in URI')) {
12     Get.snackbar(
13       'Url invalida',
14       'Verifique a Url cadastrada, ela precisa estar de acordo com o
     seguinte exemplo: \nhttp(s)://Endereco_ou_Dominio_do_Servidor(/zabbix)
     .',
15       duration: const Duration(seconds: 5),
16       backgroundColor: const Color.fromARGB(255, 128, 36, 46),
17       snackPosition: SnackPosition.BOTTOM,
18     );
19   }
20   if (e
```

```
21          .toString()
22          .contains('Scheme not starting with alphabetic character')) {
23        Get.snackbar('Url invalida',
24            'Preencha o protcolo do endereco corretamente: (http://) ou (
    https://).',
25            duration: const Duration(seconds: 5),
26            backgroundColor: const Color.fromARGB(255, 128, 36, 46),
27            snackPosition: SnackPosition.BOTTOM);
28      }
29    }
30  }
```

Fonte: Elaborado pelo autor, 2024

Para garantir a persistência de dados no aplicativo foi criada a classe *UserDataController*, utilizando o pacote *Hive*. Essa classe oferece métodos para salvar, recuperar e limpar informações do dispositivo.

No início da classe, o objeto *userBox* é inicializado, representando uma caixa do *Hive*. As caixas funcionam como bancos de dados locais simplificados, armazenando informações no formato de chave-valor, o que permite o acesso e a manipulação rápida dos dados.

O método *saveUserLoginInfo* é responsável por salvar os dados do usuário que realiza login. Ele recebe as informações necessárias, cria um objeto User e armazena esses dados na caixa com a chave userLoginInfo, utilizando o método put do Hive.

Listagem de Código 4.6 – Criação da caixa no Hive

```
1  class UserDataController {
2    final userBox = Hive.box<User>('userBox');
3    Future<void> saveUserLoginInfo(String apiCode, String server, String user
     ,
4        String pass, String apiVersion) async {
5      await userBox.put(
6        'userLoginInfo',
7        User(
8          apicode: apiCode,
9          url: server,
10         apiVersion: '',
11         usuario: user,
12         senha: pass,
13       ),
14     );
15   }
16
17   [...]
```

Fonte: Elaborado pelo autor, 2024

A classe também conta com os métodos *clearUserLoginInfo* e *getUserLoginInfo*, que complementam as funcionalidades de gerenciamento de dados.

O método *clearUserLoginInfo* é simples e direto, utilizando o método *delete* do *Hive* para apagar todos os dados armazenados na caixa, garantindo a limpeza completa das informações do usuário no dispositivo. Ele foi utilizado na classe *UserDataController*, dentro do método *logout*.

Por outro lado, o método *getUserLoginInfo* é responsável por recuperar os dados salvos na caixa. Ele utiliza o método *get* do *Hive*, passando o nome da caixa como parâmetro, para retornar o objeto *User* previamente armazenado.

Listagem de Código 4.7 – Métodos para recuperar e apagar dados do Hive

```
  [...]

  Future<void> clearUserLoginInfo() async {
    await userBox.delete('userLoginInfo');
  }

  Future<User?> getUserLoginInfo() async {
    if (userBox.containsKey('userLoginInfo')) {
      return userBox.get('userLoginInfo');
    }
    return null;
  }
}
```

Fonte: Elaborado pelo autor, 2024

O método *getUserLoginInfo* é utilizado sempre que a aplicação é iniciada, pois é necessário verificar se há credenciais salvas para realizar a requisição de dados ao servidor Zabbix. Para assegurar o funcionamento adequado, o *Hive* é inicializado no arquivo principal da aplicação, garantindo que as caixas de dados armazenadas no dispositivo estejam acessíveis desde o início.

Também foram implementados métodos auxiliares para facilitar a configuração inicial do *Hive*. Esses métodos incluem a inicialização do *Hive*, o registro do adaptador responsável pelo armazenamento de objetos personalizados, e a abertura da caixa *userBox*.

Listagem de Código 4.8 – Função main da aplicação

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await HiveConfig.start();
  await HiveConfig.startUserAdapter();
  await HiveConfig.openUserBox();
```

```
  Get.put(UserApi());

  runApp(MonitorMobile());
}
```

Fonte: Elaborado pelo autor, 2024

Por fim, ao iniciar a aplicação, as informações armazenadas na caixa são carregadas e atribuídas a variáveis observáveis, criadas com o *GetX*. Se as credenciais forem encontradas, o usuário é encaminhado diretamente para a tela principal da aplicação. Caso contrário, ele é redirecionado para a tela de login.

Listagem de Código 4.9 – Método para carregar as informações salvas no aparelho

```
Future<void> loadUserData() async {
  final user = await userDataController.getUserLoginInfo();
  if (user != null) {
    apicode.value = user.apicode;
    url.value = user.url;
    apiVersion.value = user.apiVersion;
    usuario.value = user.usuario;
    senha.value = user.senha;
  }
  update();
}
```

Fonte: Elaborado pelo autor, 2024

Listagem de Código 4.10 – Redirecionamento com base nas credenciais salvas

```
[...]

final bool isLoggedIn =
    userApi.apicode.isNotEmpty && userApi.url.isNotEmpty;
return GetMaterialApp(
  title: 'Monitor Mobile',
  theme: customDarkTheme(),
  getPages: customGetPages,
  initialRoute: isLoggedIn ? '/dashboard' : '/login',
);
```

Fonte: Elaborado pelo autor, 2024

4.5 IMPLEMENTAÇÃO DOS CONTROLADORES

Os arquivos de *controllers* são responsáveis por gerenciar todas as chamadas de API no projeto. Cada arquivo contém uma classe específica, que implementa os métodos

necessários para interagir com o servidor e retornar os dados. Todas as classes compartilham algumas características, como a presença de métodos *fetch* e uma variável que instancia a classe *GetData*, utilizada para realizar as chamadas à API.

A classe *GetData* centraliza a lógica para se conectar ao servidor e processar as requisições. Ela utiliza o pacote Dio para realizar as chamadas HTTP e gerenciar os cabeçalhos e a autenticação, possuindo também tratamento de erro para eventuais problemas que possam acontecer.

Listagem de Código 4.11 – Classe responsável pela conexão com servidor via API

```
class GetData {
  final dio = Dio();

  UserApi userapi;
  GetCookie cookiesManager = GetCookie();
  late String token;
  late String url;
  late String user;
  late String pass;

  GetData() : userapi = Get.find<UserApi>() {
    token = userapi.apicode.value;
    url = '${userapi.url.value}/api_jsonrpc.php';
    user = userapi.usuario.value;
    pass = userapi.senha.value;
  }

  Future<dynamic> getData(var json) async {
    json["auth"] = token;
    try {
      final response = await dio
          .post(url,
              data: json,
              options: Options(headers: {
                'Content-Type': 'application/json',
              }))
          .timeout(const Duration(seconds: 5));
      if (response.statusCode == 200) {
        final result = response.data["result"];
        return result;
      } else {
        return {};
      }
    } on TimeoutException {
      Get.snackbar('Erro',
          'O servidor cadastrado nao respondeu, verifique se um endereco
   valido foi preenchido!',
```

```
37          duration: const Duration(seconds: 5),
38          snackPosition: SnackPosition.BOTTOM);
39      return;
40    } catch (e) {
41      Get.snackbar('Erro', 'Erro ao buscar os dados no servidor.',
42          duration: const Duration(seconds: 5),
43          snackPosition: SnackPosition.BOTTOM);
44      return;
45    }
46  }
47  Future<String> getCookie() async {
48    String cookie =
49        await cookiesManager.getCookie(user, pass, userapi.url.value);
50    return cookie;
51  }
52 }
```

Fonte: Elaborado pelo autor, 2024

Já a classe *ItemDataController* exemplifica a estrutura e funcionamento das classes de controle no projeto, sendo responsável por gerenciar as chamadas de API relacionadas aos itens presentes no banco de dados do servidor Zabbix. A classe utiliza a instância da classe *GetData* para realizar as chamadas de API e implementar métodos específicos, como *fetchItemsByHost*.

Esse método é responsável por buscar todos os itens associados a um determinado *host* no servidor. O método carrega um arquivo JSON dos *assets* da aplicação e o altera para que o *hostId* seja repassado para a chamada de API. Caso ocorra tudo bem com a requisição, os dados são retornados.

Listagem de Código 4.12 – Classe responsável pela conexão com servidor via API

```
1 class ItemDataController {
2   final GetData apiGet = GetData();
3
4   Future<List<Item>> fetchItemsByHost(String hostId) async {
5     String getHostItensCall =
6         await rootBundle.loadString('assets/json/items/get_items_by_host.
    json');
7     final jsonRequest = await jsonDecode(getHostItensCall);
8     jsonRequest["params"]["hostids"] = hostId;
9     List<dynamic> itensData = await apiGet.getData(jsonRequest);
10    List<Item> itens = itensData
11        .map((item) => Item.fromJson(item))
12        .where((element) => element.error == "")
13        .toList();
14    return itens;
```

```
15  }
16
17  [...]
```

Fonte: Elaborado pelo autor, 2024

Outros métodos estão presentes na classe *ItemDataController*, como *fetchItensBy-Tempalte*, que retorna todos os itens pertecentes à um template no servidor Zabbix, e o método *fetchHistory*, que retorna o histórico de valores de um único item. Ambos seguem o padrão de carregar um JSON base, ajustar parâmetros conforme necessário e realizar a chamada à API para obter os dados do servidor.

Nos *controllers* do projeto, esse padrão apresentado é mantido, onde a lógica de manipulação de dados e chamadas de API é centralizada. Embora cada *controller* seja responsável por gerenciar informações específicas, como itens, *hosts* ou templates, o esquema geral é o mesmo, variando apenas os parâmetros utilizados e os dados retornados.

Uma característica adicional de algumas classes, como esta, é o método de pesquisa, que filtra itens em tempo real com base em critérios pesquisados pelo usuário na interface. O método filtra uma lista de itens com base em uma consulta (*query*) fornecida pelo usuário, verificando se o nome do item ou sua unidade de medida correspondem ao termo pesquisado.

Listagem de Código 4.13 – Método responsável pelo filtro na lista de itens.

```
1 List<Item> searchItensFilter(String query, List<Item> items) {
2     return items
3         .where((element) =>
4             element.name.toLowerCase().contains(query.toLowerCase()) ||
5             element.newUnits.toLowerCase().contains(query.toLowerCase()))
6         .toList();
7   }
```

Fonte: Elaborado pelo autor, 2024

4.6 EXIBINDO GRÁFICOS NO APLICATIVO

No sistema de monitoramento Zabbix, os gráficos são gerados dinamicamente com base no histórico dos valores de cada item, utilizando a linguagem PHP e acessados atráves de uma URL única. Por não se tratarem de imagens estáticas, o processo de integração com a aplicação se tornou um pouco mais complexo. O principal desafio foi obter acesso a essas páginas, pois era necessário estabelecer uma sessão autenticada no servidor para visualizá-las, não bastando somente a autenticação via API.

Para superar essa dificuldade, o pacote Dio foi utilizado novamente, principalmente por contas de sua capacidade de gerenciar *cookies* de forma eficiente. Toda vez que um usuário se autentica no sistema, um cookie de autenticação é gerado (*ZBX_SESSION_NAME*),

servindo como um identificador que informa ao servidor a validade daquela sessão. E tendo o valor deste *Cookie* seria possível utilizar o *webview_flutter* para acessar a página de qualquer gráfico desejado.

Para conseguir recuperar o valor desse *Cookie*, foi criada o método *getCookie*. Ele envia uma requisição *post* para o *endpoint* no servidor e, caso a autenticação ocorra normalmente, o valor do *Cookie* é retornado.

Listagem de Código 4.14 – Método responsável pela captura do Cookie de autenticação.

```
import 'package:dio/dio.dart';

class GetCookie {
  Dio dio = Dio();
  Future<String> getCookie(String user, String pass, String url) async {
    FormData formData = FormData.fromMap({
      "name": user,
      "password": pass,
      "enter": "Sign in",
    });

    try {
      Response<dynamic> response = await dio.post(
        "$url/index.php",
        data: formData,
        options: Options(
          followRedirects: false,
          validateStatus: (status) => status! < 500,
        ),
      );

      if (response.statusCode! >= 300 && response.statusCode! < 400) {
        String? setCookieHeader = response.headers.map['set-cookie']?.first
    ;
        if (setCookieHeader != null) {
          String cookie = setCookieHeader.split(';')[0];
          return cookie;
        }
      }
      print(
          'Falha na autorizacao do usuario: O nome de login ou a senha
    estao incorretos.');
      return 'null';
    } catch (e) {
      print('Erro durante solicitacao HTTPt: $e');
      return 'null';
    }
  }
```

```
}
```

Fonte: Elaborado pelo autor, 2024

Com o valor do *Cookie* da sessão salvo, é possível utiliza-lo no cabeçalho de uma requisição do *WebView*. Dessa forma, o servidor identifica a requisição como uma sessão valida e permite o acesso à página onde o gráfico é exibido.

Listagem de Código 4.15 – Utilizando o cookie de sessão para requisição do WebView

```
[...]

return WebViewWidget(
    controller: WebViewController()
      ..setJavaScriptMode(JavaScriptMode.unrestricted)
      ..setBackgroundColor(const Color(0x00000000))
      ..setNavigationDelegate(
        NavigationDelegate(
          onPageStarted: (String url) {},
          onPageFinished: (String url) {},
          onWebResourceError: (WebResourceError error) {},
          onNavigationRequest: (NavigationRequest request) {
            return NavigationDecision.navigate;
          },
        ),
      )
      ..loadRequest(Uri.parse(url),
          headers: {"cookie": cookie}));

[...]
```

Fonte: Elaborado pelo autor, 2024

4.7 NOTIFICAÇÕES DE INCIDENTES

A implementação das notificações no aplicativo foi um desafio maior do que o inicialmente previsto. A ideia original consistia em sempre que um incidente ocorresse no servidor Zabbix, o usuário receberia uma notificação no aplicativo em tempo real. Para evitar custos adicionais com serviços de terceiros, a solução planejada consistia em um serviço em segundo plano, que realizaria consultas ao servidor a cada minuto para verificar novos incidentes. Como se tratava de uma única chamada à API, o impacto no consumo de recursos do dispositivo seria mínimo.

No entanto, ao tentar implementar essa funcionalidade, descobriu-se que, tanto em sistemas Android quanto iOS, aplicações não podem executar tarefas em segundo plano em intervalos pequenos da forma como planejado. Além de outras restrições que poderiam

acarretar no mal funcionamento das notificações, como exemplo, caso o usuário não utilize a aplicação por um longo período de tempo, o sistema operacional deixa de executar a função em segundo plano.

Diante desse obstáculo, uma alternativa viável seria a integração com serviços de notificações *push*, como FCM (Firebase Cloud Messaging) ou OneSignal. No entanto, essa abordagem traria dois desafios principais. Primeiro, a possibilidade de custos adicionais para o usuário ou para a aplicação, dependendo do volume de notificações. Segundo, a necessidade de criar um *endpoint* intermediário para conectar o servidor Zabbix ao serviço de notificações. Essa etapa adicional seria necessária pois, para funcionarem, seria necessário configurar cada servidor Zabbix de usuários ao sistema de notificações *push*, tornando inviável repassar credenciais de acesso para cada usuário da aplicação.

Após análise dos pontos levantados anteriormente, foi decidido manter as notificações no aplicativo, mesmo considerando as limitações impostas pelo sistema operacional. O primeiro passo foi criar um método para recuperar os dados enquanto a aplicação estiver aberta e também permitir a execução periódica em segundo plano. Utilizando a função *BackgroundFetch.configure()*, o método foi configurado para ser executado a cada 15 minutos, sendo o menor intervalo permitido pelo sistema operacional e, ainda sim, este período pode ser aumentado automaticamente pelo sistema dependendo de restrições ou políticas de otimização.

As credenciais do usuários salvas são recuperadas do *Hive*, e o *token* da API é utilizado para realizar uma chamada que retorna os incidentes recentes ativos no servidor Zabbix. Para evitar notificações repetidas, o método *fetchProblemsWithToken* foi configurado para retornar apenas os incidentes que tenham ocorrido nos últimos 15 minutos. Caso a lista de incidentes tenha algum item, a notificação é enviada para o usuário. Vale ressaltar que, caso a aplicação esteja visível na tela do usuário, ao invés de receber uma notificação, um alerta via *Get.Snackbar* é enviado.

Listagem de Código 4.16 – Método responsável por retornar os dados em segundo plano.

```
1  Future<void> initBackgroundFetch() async {
2     int status = await BackgroundFetch.configure(
3         BackgroundFetchConfig(
4           minimumFetchInterval: 15,
5           stopOnTerminate: false,
6           enableHeadless: true,
7           requiresBatteryNotLow: false,
8           requiresCharging: false,
9           requiresStorageNotLow: false,
10          requiresDeviceIdle: false,
11          requiredNetworkType: NetworkType.NONE,
12        ), (String taskId) async {
13      User? user;
14      try {
```

```
15        user = await userDataController.getUserLoginInfo();
16
17        if (user == null || user.apicode.isEmpty) {
18          BackgroundFetch.finish(taskId);
19          return;
20        }
21      } catch (e) {
22        print("[BackgroundFetch] Erro ao acessar o Hive: $e");
23      }
24
25      final incidents =
26          await problemDataController.fetchProblemsWithToken(user!.apicode)
    ;
27
28      if (incidents.isNotEmpty) {
29        handleIncidents(incidents);
30      }
31
32      BackgroundFetch.finish(taskId);
33    }, (String taskId) async {
34      BackgroundFetch.finish(taskId);
35    });
36
37    BackgroundFetch.registerHeadlessTask(backgroundFetchHeadlessTask);
38  }
```

Fonte: Elaborado pelo autor, 2024

Para criar a notificação, foi desenvolvida uma classe chamada *NotificationService*. Essa classe é responsável por gerenciar todas as operações relacionadas às notificações dentro do aplicativo, garantindo que os alertas sejam configurados corretamente e exibidos quando necessário. Iniciando pelo método *initNotification*, responsável por setar os parâmetros iniciais da notificação, como ícones e algumas permissões.

Listagem de Código 4.17 – Método responsável setar os parâmetros da notificação.

```
1 class NotificationService {
2   final notificationsPlugin = FlutterLocalNotificationsPlugin();
3   bool _isInitialized = false;
4   bool get isInitialized => _isInitialized;
5
6   Future<void> initNotifications() async {
7     if (_isInitialized) return;
8
9     const initSettingsAndroid =
10        AndroidInitializationSettings('@mipmap/monitor_mobile_monochrome');
11
```

```
12    const initSettingsIOS = DarwinInitializationSettings(
13      requestAlertPermission: true,
14      requestBadgePermission: true,
15      requestSoundPermission: true,
16    );
17
18    const initSettings = InitializationSettings(
19      android: initSettingsAndroid,
20      iOS: initSettingsIOS,
21    );
22
23    await notificationsPlugin.initialize(initSettings);
24  }
25
26  [...]
```

Fonte: Elaborado pelo autor, 2024

O método *notificationDetails* retorna uma instância da classe NotificationDetails, que contém as configurações específicas para Android e iOS. No Android, a classe *AndroidNotificationDetails* é utilizada para definir as propriedades da notificação, como o canal de notificações, o nome exibido ao usuário, a importância da notificação e o nível de prioridade (Priority.high), o que faz com que a notificação seja exibida imediatamente na interface do usuário. No iOS, a classe *DarwinNotificationDetails* é utilizada para garantir a compatibilidade com o sistema operacional da Apple.

Já o método *showNotification* é responsável por exibir a notificação. Ele recebe como parâmetros um identificador único, um título e um corpo para a notificação, além de utilizar os detalhes configurados previamente no método *notificationDetails*.

Listagem de Código 4.18 – Métodos de detalhes e de visualização da notificação.

```
1  [...]
2
3 NotificationDetails notificationDetails() {
4   return const NotificationDetails(
5     android: AndroidNotificationDetails(
6       'zabbix_alerts_channel',
7       'Zabbix Alerts',
8       importance: Importance.high,
9       priority: Priority.high,
10    ),
11    iOS: DarwinNotificationDetails());
12 }
13
14 Future<void> showNotification(int id, String? title, String? body)
15   async {
```

```
  return notificationsPlugin.show(id, title, body,
    notificationDetails());
}

[...]
```

Fonte: Elaborado pelo autor, 2024

Para que o aplicativo pudesse exibir alertas no dispositivo foi necessário solicitar permissões específicas. No caso do Android, as permissões foram declaradas no arquivo *AndroidManifest.xml*, garantindo que o sistema operacional permitisse o envio e a exibição de notificações. No quadro 4.19 são mostradas duas permissões, a primeira sendo *android.permission.USE_EXACT_ALARM*, permitindo que o aplicativo utilize alarmes exatos, essencial para que as notificações funcionem conforme planejado. A outra permissão listada é a *android.permission.POST_NOTIFICATIONS*, responsável por permitir que o aplicativo possa exibir notificações ao usuário.

Listagem de Código 4.19 – Métodos de detalhes e de visualização da notificação.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" xmlns:
    tools="http://schemas.android.com/tools">
    <uses-permission android:minSdkVersion="34" android:name="android.
    permission.USE_EXACT_ALARM" />
    ...
    <uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>

[...]
```

Fonte: Elaborado pelo autor, 2024

Além de declarar a permissão para exibir notificações no arquivo *AndroidManifest.xml*, foi necessário solicitar explicitamente essa permissão ao usuário quando a aplicação fosse iniciada pela primeira vez. Para isso, o método *requestNotificationPermission* foi criado, reponsável por verificar se a permissão já foi concedida e, caso contrário, solicitar a autorização do usuário.

Listagem de Código 4.20 – Método para solicitar permissão ao usuário.

```
  Future<void> requestNotificationPermission() async {
    final bool? isGranted = await notificationsPlugin
        .resolvePlatformSpecificImplementation<
            AndroidFlutterLocalNotificationsPlugin>()
        ?.areNotificationsEnabled();

    if (isGranted == true) {
      return;
```

```
9     }
10
11    final bool? granted = await notificationsPlugin
12        .resolvePlatformSpecificImplementation<
13            AndroidFlutterLocalNotificationsPlugin>()
14        ?.requestNotificationsPermission();
15  }
```

Fonte: Elaborado pelo autor, 2024

4.8 OTIMIZAÇÃO DAS REQUISIÇÕES DA API

Durante o desenvolvimento do projeto, foram realizados testes em um servidor Zabbix que possuía uma quantidade significativa de itens, chegando a mais de trezentos *hosts*, sendo que alguns deles contavam com mais de seiscentos itens ativos. Embora essa quantidade de dados possa parecer expressiva em um primeiro momento, é relativamente pequena quando comparada a grandes infraestruturas. Esses testes revelaram um comportamento de desempenho na tela de listagem dos *hosts*, onde o intervalo entre a requisição via API e a apresentação dos dados na interface estava relativamente elevado.

Para investigar o problema, foram realizados testes detalhados para identificar a etapa responsável pela lentidão no processamento. Foi identificado que o gargalo estava relacionado à forma como os dados eram obtidos. No Zabbix, é necessário realizar uma requisição para retornar os *hosts* e outra para obter as interfaces associadas a eles. A demora era causada porque, após buscar os dados de todos os *hosts* no servidor, um laço de repetição (*FOR*) era executado para cada objeto de *host*, realizando requisições individuais para recuperar as interfaces de cada um. Esse processo gerava uma grande quantidade de chamadas ao servidor, sobrecarregando-o e aumentando o tempo de resposta.

Para otimizar esse procedimento e reduzir a carga sobre o servidor, foi implementada uma abordagem mais eficiente. Um novo método chamado *fetchHostInterfaces* (Quadro 4.22) foi desenvolvido para centralizar a busca de todas as interfaces cadastradas no servidor em uma única requisição, armazenando os dados em uma variável. Esse método trabalha em conjunto com o método *fetchHosts* (Quadro 4.21), que continua sendo responsável por retornar a lista de *hosts*. Após obter essa lista, o método *fetchHostInterfaces* itera sobre ela para associar as interfaces correspondentes a cada *host*, baseando-se nos dados previamente retornados. Assim, as interfaces são atribuídas diretamente aos objetos da lista de *hosts*, eliminando a necessidade de múltiplas requisições individuais ao servidor.

Essa abordagem reduziu significativamente a quantidade de chamadas ao servidor, aliviando sua carga e permitindo que o processamento dos dados fosse feito localmente no código.

Listagem de Código 4.21 – Método que retorna todos os hosts do servidor.

```
Future<List<Host>> fetchHosts({String name = ''}) async {
  try {
    String getHostsJson =
        await rootBundle.loadString('assets/json/hosts/get_hosts.json');
    final jsonRequest = await jsonDecode(getHostsJson);
    List<dynamic> getHostsResponse = await apiGet.getData(jsonRequest);
    List<Host> hosts =
        getHostsResponse.map((host) => Host.fromJson(host)).toList();

    await fetchHostInterfaces(hosts);
    sortByNameHosts(hosts);
    return hosts;
  } catch (e) {
    print('Erro ao buscar hosts com a API: $e');
    return [];
  }
}
```

Fonte: Elaborado pelo autor, 2024

Listagem de Código 4.22 – Método que retorna todas as interfaces de host.

```
Future<void> fetchHostInterfaces(List<Host> hosts) async {
  try {
    List<Interface> interfaces = await interfaceController.fetchInterfaces
   ();
    for (int i = 0; i < hosts.length; i++) {
      Host host = hosts[i];
       host.hostInterfaces =
          interfaces.where((element) => element.hostId == (host.id)).toList
   ();
      host.mainInterface = interfaces
          .where(
              (element) => element.hostId == (host.id) && element.main ==
   '1')
          .toList();
   }
  } catch (e) {
    print('Erro atribuir as interfaces do host: $e');
  }
}
```

Fonte: Elaborado pelo autor, 2024

5 RESULTADOS

5.1 FUNCIONALIDADES IMPLEMENTADAS

O desenvolvimento desta aplicação alcançou diversas funcionalidades que atenderam perfeitamente o escopo inicial do projeto. Em sua versão inicial, o aplicativo mostrou-se funcional e alinhado às expectativas, proporcionando uma ótima experiência. Entre as principais capacidades, destaca-se a exibição detalhada de hosts, incluindo informações como grupos associados, templates aplicados, detalhes das interfaces, gráficos personalizados gerados a partir dos templates e dados do inventário. Além disso, a aplicação oferece uma visão abrangente dos itens monitorados e dos incidentes ativos relacionados a cada host.

Os incidentes receberam atenção especial, com uma abordagem que permite ao usuário acessar uma lista completa dos incidentes ativos no servidor. É possível filtrar esses dados com base na severidade, através da dashboard, e visualizar informações detalhadas sobre cada ocorrência. Para facilitar a gestão, o aplicativo também permite atualizar os incidentes, seja adicionando mensagens, reconhecendo-os ou alterando suas severidades.

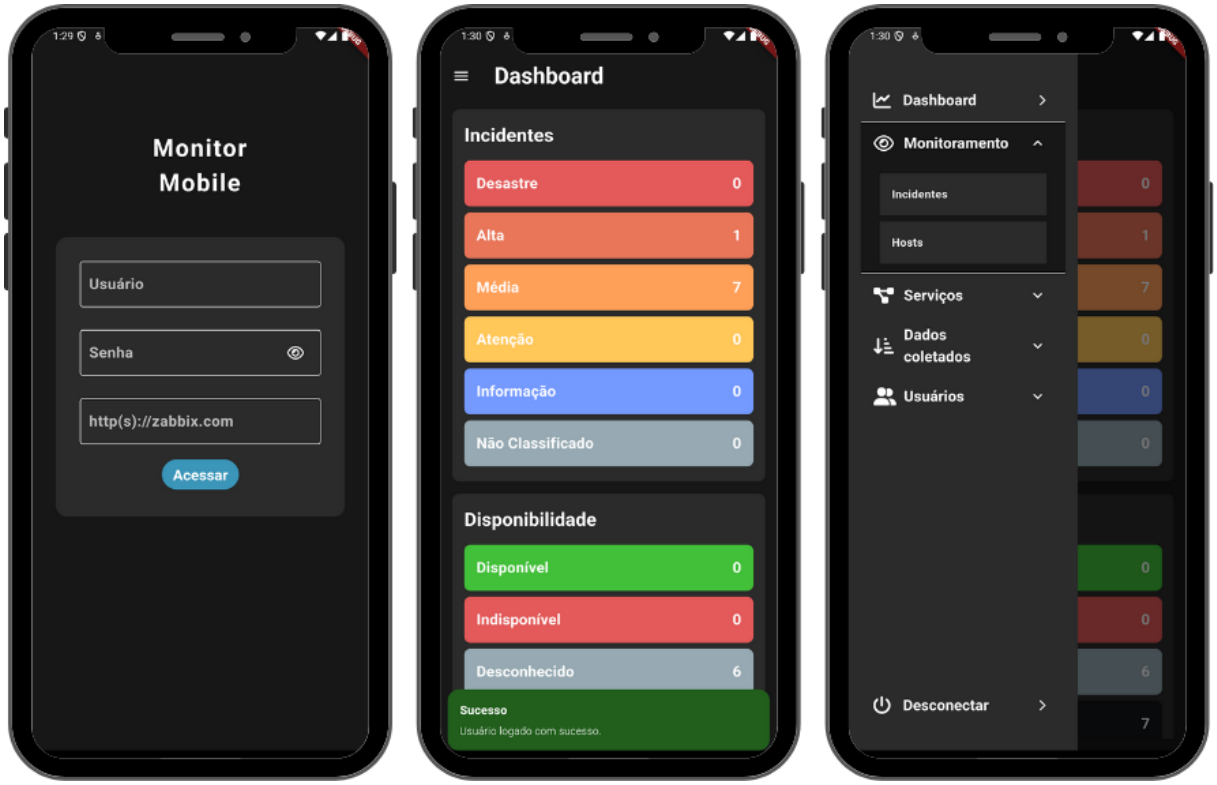
A interface foi projetada para ser intuitiva e acolhedora, contribuindo para que o usuário se sinta rapidamente familiarizado com o aplicativo. O menu lateral, por exemplo, foi inspirado no *layout* do servidor Zabbix, que transfere para o ambiente móvel a mobilidade e organização já presentes na versão *web*. Essa abordagem oferece uma estrutura familiar para os usuários habituados à plataforma, facilitando a transição entre os dois formatos e fortalecendo tanto a identidade visual quanto a funcionalidade do sistema.

Além das funcionalidades mencionadas, ele oferece acesso a informações sobre templates e grupos de *hosts*, permitindo que o usuário explore os hosts vinculados a cada um deles de forma prática e eficiente.

5.2 INTERFACES DESENVOLVIDAS

As interfaces do aplicativo foram desenvolvidas para proporcionar uma experiência intuitiva e fluida, garantindo que os usuários possam ter uma boa interação com as informações apresentadas. A Figura 5.1 mostra o fluxo inicial da aplicação, com a autenticação do usuário. A partir disso, o usuário pode navegar entre diferentes telas através do menu lateral, visualizar incidentes categorizados e acessar detalhes mais aprofundados conforme necessário.

Figura 5.1 – Fluxo inicial do usuário e menu de navegação

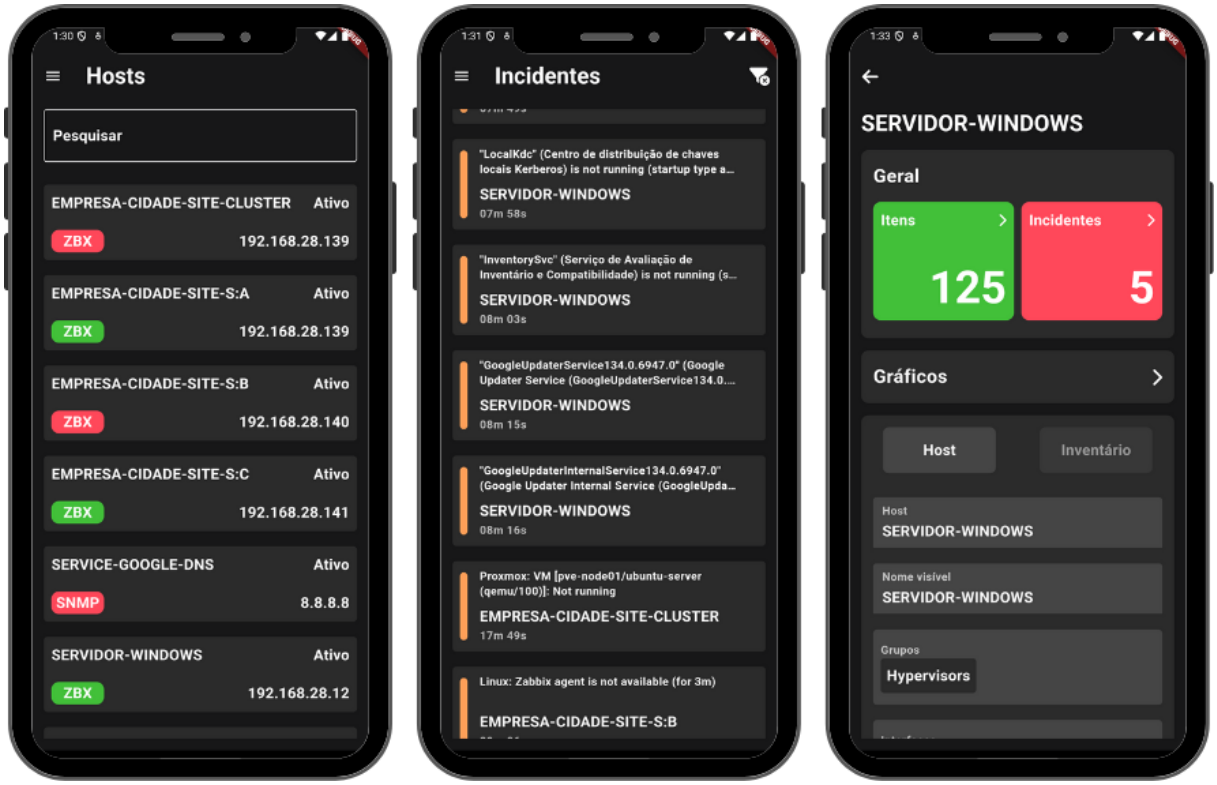


Fonte: Elaborado pelo autor, 2024

A Figura 5.2 apresenta as principais telas da aplicação. Na primeira tela, há a listagem de *hosts* cadastrados no servidor, Em seguida, há uma tela dedicada à listagem de incidentes ativos. Por fim, a terceira tela traz o detalhamento de um *host* específico, apresentando a quantidade de itens monitorados e de incidentes ativos associados.

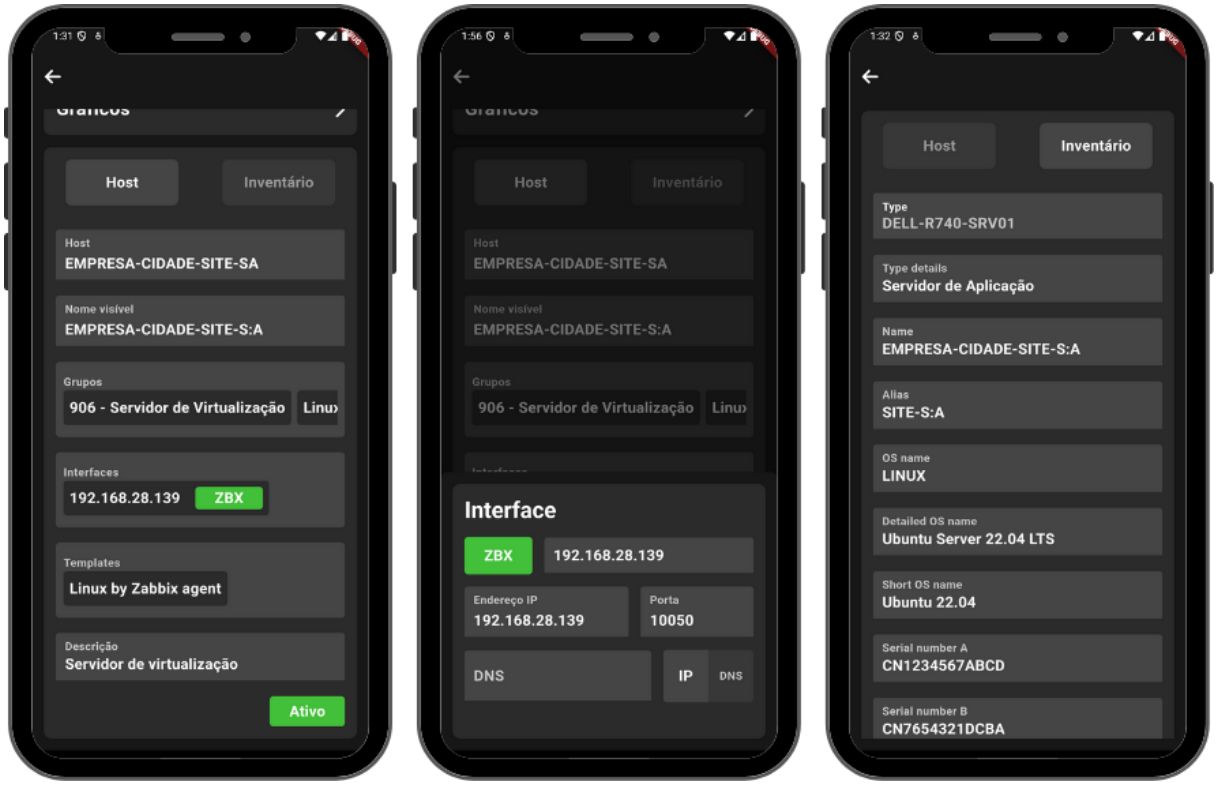
Já a Figura 5.3 apresenta a tela de detalhes de um *host*, que se encontra organizada em duas guias principais. A primeira guia apresenta informações básicas do *host*, como nome, interfaces configuradas, templates atribuídos e grupos aos quais ele pertence. Já a segunda guia concentra o inventário do *host*, fornecendo dados mais detalhados que auxiliam no entendimento do contexto e das características específicas daquele dispositivo em monitoramento.

Figura 5.2 – Listagem de *hosts* e incidentes



Fonte: Elaborado pelo autor, 2024

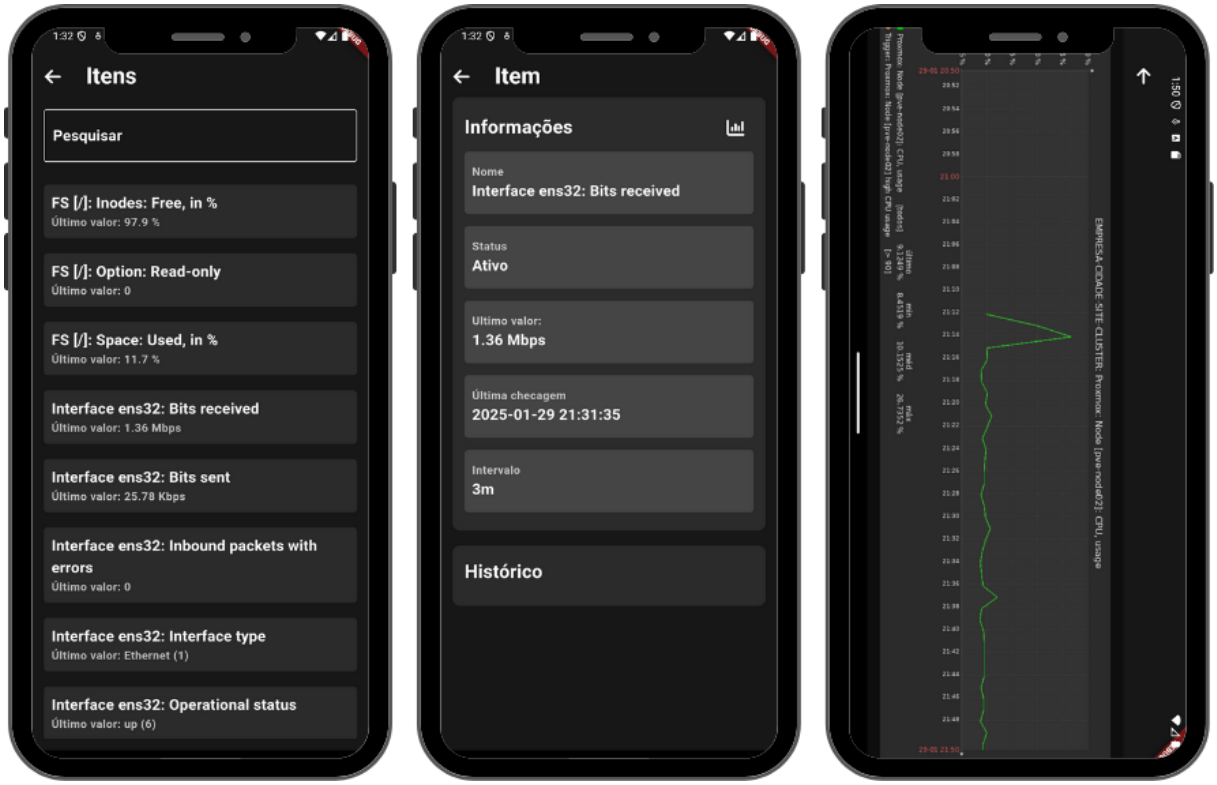
Figura 5.3 – Detalhes do *host*



Fonte: Elaborado pelo autor, 2024

Dentro dessa tela, ao clicar no *Card* que exibe a quantidade de itens, o usuário é direcionado para uma tela de listagem, conforme ilustrado na Figura 5.4. Essa listagem exibe todos os itens monitorados e, ao selecionar qualquer um deles, uma página dedicada ao detalhamento do item é aberta. Nessa página, o usuário tem acesso às mais informações relevantes sobre o item e tem a possibilidade de visualizar seu histórico de valores por meio de um gráfico.

Figura 5.4 – Detalhes do item

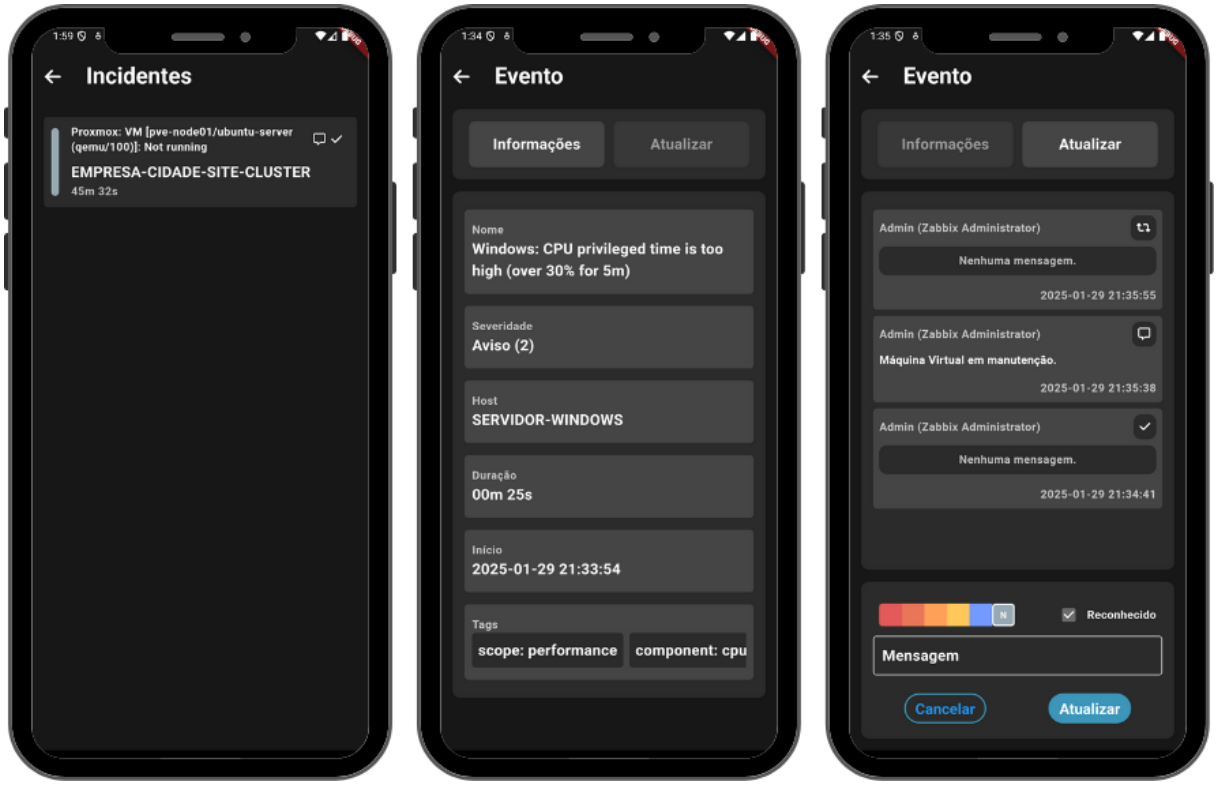


Fonte: Elaborado pelo autor, 2024

Seguindo o mesmo principio, ao clicar no *Card* que exibe a quantidade de incidentes ativos, o usuário é redirecionado para a tela de listagem de incidentes associados ao *host*, conforme Figura 5.5. Selecionando qualquer incidente da lista, o usuário tem acesso aos seus detalhes, e também consegue interagir com o incidentes, como o reconhecendo, deixando mensagens ou alterando a severidade.

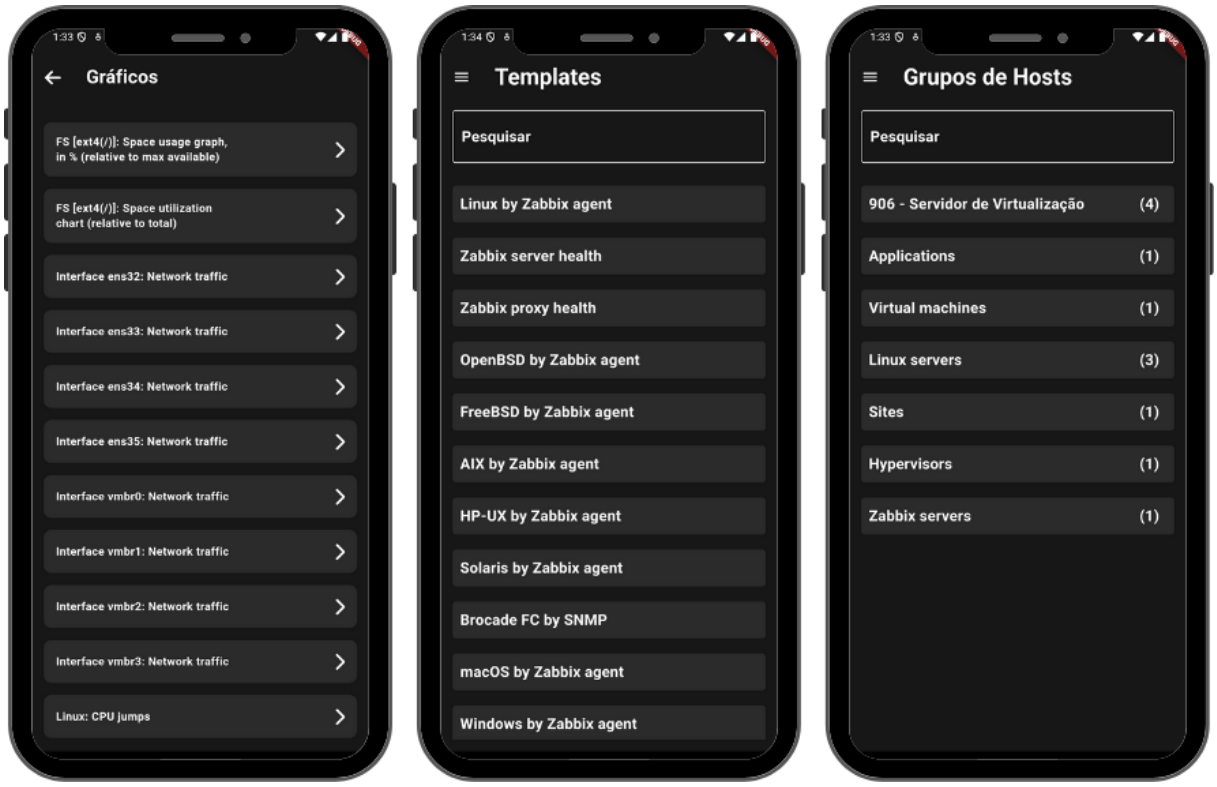
Por fim, o sistema também conta com algumas interfaces semelhantes, destinadas à listagem de dados. A Figura 5.6 ilustra algumas dessas telas, que em sua maioria seguem o mesmo padrão, exibindo os itens em formato de lista e permitindo que o usuário selecione qualquer um deles para visualizar mais detalhes.

Figura 5.5 – Detalhes e atualização de incidentes



Fonte: Elaborado pelo autor, 2024

Figura 5.6 – Listagem de informações diversas



Fonte: Elaborado pelo autor, 2024

5.3 LIMITAÇÕES E FUNCIONALIDADES FALTANTES

Apesar dos resultados positivos obtidos até o momento, nem todas as funcionalidades planejadas foram implementadas na versão atual. Um exemplo é o gerenciamento de usuários, que permitiria criar, editar e excluir perfis diretamente pelo aplicativo. Essa funcionalidade exigiria uma integração mais ampla com a API do Zabbix, além de um fluxo de ações bem definido para garantir sua operação adequada.

Outro recurso que ficou pendente foi a possibilidade de editar dados relacionados a hosts, templates e grupos de hosts. Embora a funcionalidade não tenha sido implementada, a estrutura necessária para sua futura inclusão já foi construída. Atualmente, as informações são apresentadas em campos de formulário desabilitados, o que facilita, em atualizações futuras, permitir que os usuários modifiquem esses campos sem grandes mudanças na arquitetura do sistema.

A principal limitação da aplicação, entretanto, foi a ausência de notificações em tempo real. A ideia original era adicionar alertas integrados ao aplicativo, mas, no decorrer do desenvolvimento, surgiram obstáculos que dificultaram essa implementação. Em especial, as restrições impostas pelos sistemas operacionais e os custos adicionais de serviços de terceiros acabaram inviabilizando, em um primeiro momento, o uso de notificações *push* de forma gratuita, algo importante para a proposta de manter o aplicativo acessível sem custos extras.

Apesar dessa limitação, uma alternativa para que os usuários ainda recebam atualizações sobre incidentes foi desenvolvida. O aplicativo conta com notificações periódicas, limitadas pelos recursos do sistema operacional do dispositivo. Embora não sejam em tempo real, elas oferecem uma forma de manter o usuário informado sobre eventos importantes. É uma solução intermediária que, embora não substitua completamente as notificações *push* ideais, cumpre uma função de alerta e ajuda a manter a experiência de uso satisfatória.

A funcionalidade de notificações em tempo real não foi descartada, e alternativas para viabilizá-la em versões futuras continuam sendo estudadas, de forma que os alertas de incidentes possam ser entregues prontamente.

6 CONCLUSÃO

Este trabalho apresentou o desenvolvimento do Monitor Mobile, um aplicativo que amplia as funcionalidades do sistema de monitoramento Zabbix para dispositivos móveis, oferecendo aos profissionais de TI uma ferramenta prática e eficiente para a gestão de incidentes e monitoramento de infraestrutura.

Um dos objetivos em relação ao desenvolvimento deste aplicativo era alcançar a simplicidade e usabilidade, garantindo que mesmo usuários sem experiência avançada com o Zabbix pudessem se adaptar rapidamente à interface proposta. A organização clara dos dados e a navegação intuitiva refletem o compromisso com uma experiência de uso fluida, focada na eficiência operacional.

Apesar de alcançar os objetivos iniciais, foram identificadas algumas limitações que abrem espaço para melhorias futuras. Entre elas, destaca-se a ausência de notificações de incidentes em tempo real e de algumas funcionalidades, como o gerenciamento de usuários e edição dos dados retornados da API. Além disso, a expansão para novos recursos alinhados às demandas emergentes e às evoluções do Zabbix será essencial nas próximas atualizações do projeto.

O Monitor Mobile demonstrou ser uma solução promissora, ao unir mobilidade e eficiência na gestão de TI, reduzindo o tempo de resposta a incidentes e promovendo a continuidade operacional. Com isso, espera-se que este projeto não apenas contribua para a comunidade de TI, mas também inspire o desenvolvimento de novas iniciativas no campo do monitoramento de sistemas.

REFERÊNCIAS

AGUIAR, Ítalo Fernandes. **Proposta de utilização da ferramenta Zabbix no gerenciamento de redes: um estudo de caso no ambiente da FAB segundo boas práticas de governança de TI**. [*S.l.*: *s.n.*], 2017. Orientador: Moacyr Henrique Cruz de Azevedo. Disponível em: https://pantheon.ufrj.br/handle/11422/3300. Acesso em: 19 jan. 2025.

BEYER, Betsy; JONES, Chris; PETOFF, Jennifer; MURPHY, Niall Richard. **Site Reliability Engineering: How Google Runs Production Systems**. [*S.l.*]: O'Reilly Media, 2016. Acesso em: dia mês ano.

FLUTTER. **Background Fetch: Flutter plugin for periodic background tasks**. [*S.l.*: *s.n.*], 2025. Acesso em: 20 jan. 2025. Disponível em: https://pub.dev/packages/background_fetch.

FLUTTER. **Dio: A powerful HTTP client for Dart**. [*S.l.*: *s.n.*], 2025. Acesso em: 19 jan. 2025. Disponível em: https://pub.dev/packages/dio.

FLUTTER. **Flutter Local Notifications: Plugin for displaying local notifications in Flutter**. [*S.l.*: *s.n.*], 2025. Acesso em: 20 jan. 2025. Disponível em: https://pub.dev/packages/background_fetch.

FLUTTER. **Get: State management, intelligent dependency injection, and route management for Flutter apps**. [*S.l.*: *s.n.*], 2025. Acesso em: 19 jan. 2025. Disponível em: https://pub.dev/packages/get.

FLUTTER. **Hive: Lightweight and blazing fast key-value database for Flutter and Dart**. [*S.l.*: *s.n.*], 2025. Acesso em: 19 jan. 2025. Disponível em: https://pub.dev/packages/get.

FORRESTER. **Quantificando o impacto do tempo de inatividade: o que podemos aprender com as recentes interrupções do New York Times, Google e Amazon**. [*S.l.*: *s.n.*], 2025. https://www.forrester.com/blogs/13-08-19-quantifying_the_impact_of_downtime_what_we_can_learn_from_recent_new_york_times_google_and_amazon_out/. Acesso em: 14 jan. 2025.

GRIES, Heather. **Parceria da Unisys com a HFS Research revela como empresas estão prosperando no trabalho híbrido**. [*S.l.*: *s.n.*], 2023. Acesso em: 14 jan. 2025. Disponível em: https://www.unisys.com/pt/news-release/hfs-partnership-research-surviving-to-thriving-in-hybrid-work/.

HAT, Red. **O que é uma API REST?** [*S.l.*: *s.n.*], 2023. Acessado em: 19 jan. 2025. Disponível em: https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api.

AL-MASAH, Ahmed Shamsan; AL-SHARAFI, Ali M. Benefits of Cloud Computing for Network Infrastructure Monitoring Service. **International Journal of Advances in Engineering & Technology**, v. 5, n. 2, p. 46–51, jan. 2013. ISSN 1476-4687.

MEIRELLES, Fernando S. **Uso da TI nas Empresas: 35ª Pesquisa Anual**. São Paulo, Brasil: FGVcia - Fundação Getulio Vargas, Centro de Tecnologia de Informação Aplicada, 2024. Acesso em: 18 jan. 2025. Disponível em: https://eaesp.fgv.br/producao-intelectual/pesquisa-anual-uso-ti?utm_source=portal-fgv&utm_medium=fgvnoticias&utm_id=fgvnoticias-2024-06-27.

OLIVEIRA, Paulo. **Como os dispositivos móveis podem aumentar a eficiência dos funcionários**. Acesso em: 11 jan. 2025. 2023. Disponível em: https://abes.com.br/como-os-dispositivos-moveis-podem-aumentar-a-eficiencia-dos-funcionarios/.

SOUSA, Flávia Amaral Silva de; FREITAS, Eduardo Augusto Lourenço. **Manual de Gestão de Riscos, Controles Internos e Integridade**. 1ª Edição. Esplanada dos Ministérios, Bloco E, S/N - Zona Cívico-Administrativa, 8º andar Brasília/DF - CEP 70 067-901, 2020. P. 65. Disponível em: https://www.gov.br/mdr/pt-br/acesso-a-informacao/governanca/manual_integridade_final.pdf.

ZABBIX. **API - Zabbix**. [*S.l.*: *s.n.*], 2025. Acesso em: 18 jan. 2025. Disponível em: https://www.zabbix.com/documentation/current/en/manual/api.

ZABBIX. **O que é o Zabbix**. [*S.l.*: *s.n.*], 2025. Acesso em: 11 jan. 2025. Disponível em: https://www.zabbix.com/features.

ZAMMETTI, Frank. **Flutter na prática: Melhore seu Desenvolvimento Mobile com o SDK Open Source Mais Recente do Google**. Novatec Editora: Novatec Editora, jan. 2020. ISBN 9788575228234, 8575228234.