UNIVERSIDADE FEDERAL DO MARANHÃO
Engenharia da Computação

Miquéias Santos Rodrigues

Sistema de Monitoramento Residencial com Alertas em Aplicativo Móvel

São Luís - MA
2024

Miquéias Santos Rodrigues

Sistema de Monitoramento Residencial com Alertas em Aplicativo Móvel

Trabalho de Conclusão de Curso submetido à Coordenação do Curso de Engenharia da Computação da Universidade Federal do Maranhão como requisito à obtenção do grau de Bacharelado em Engenharia da Computação.

Engenharia da Computação

Universidade Federal do Maranhão

São Luís - MA

2024

Miquéias Santos Rodrigues

Sistema de Monitoramento Residencial com Alertas em Aplicativo Móvel

Trabalho de Conclusão de Curso submetido à Coordenação do Curso de Engenharia da Computação da Universidade Federal do Maranhão como requisito à obtenção do grau de Bacharelado em Engenharia da Computação.

Prof. Dr. Paulo Rogério de Almeida Ribeiro
Orientador
Universidade Federal do Maranhão

Prof. Dr. Haroldo Gomes Barroso Filho
Universidade Federal do Maranhão

Profa. Dra. Vandecia Rejane Monteiro Fernandes
Universidade Federal do Maranhão

São Luís - MA
2024

Agradecimentos

Agradeço primeiramente a Deus pela Tua misericórdia e pela Tua graça. Em Ti, ó Senhor, encontramos força, pois de onde virá nossa força senão de Ti? Muito obrigado por tudo.

Agradeço a minha amada e digníssima esposa, Carla Regina Brandão Rodrigues, por todo apoio que me deste para a realização desse trabalho. Agradeço a todos os meus familiares, em especial meu pai Washington Luis Pereira Rodrigues e minha mãe Meire Lourdes Costa Santos. Obrigado por tudo.

Agradeço ao Professor Dr. Paulo Rogério de Almeida Ribeiro, responsável pela orientação desse trabalho e por toda a ajuda. Muito obrigado pela paciência para a realização desse projeto.

Agradeço aos meus amigos e colegas do curso de Bacharelado, pelas experiências e amizades.

Resumo

Este trabalho propõe o desenvolvimento de um sistema de monitoramento residencial, motivado pela grande quantidade de assaltos a residências no Brasil e pela falta de personalização dos sistemas de monitoramento existentes. Este sistema utilizará uma aplicação móvel que se comunicará com diferentes modelos de câmeras por meio do protocolo Real-Time Publish-Subscribe Protocol (RTSP).

A arquitetura do sistema utiliza a linguagem de programação Java com o framework Spring Boot para a construção dos serviços de usuário, autenticação, câmera e notificação, e Python, que é utilizado para a construção do servidor de processamento de imagem, o qual possui um algoritmo para detecção de pessoas utilizando conceitos de visão computacional. A aplicação móvel foi construída usando o React Native, com uma arquitetura Model-View-ViewModel (MVVM), tornando-a mais otimizada e facilitando a manutenção do código.

Foram realizados testes com uma e duas pessoas em diferentes cenários, com objetos ao redor. O servidor de processamento de imagem conseguiu detectar pelo menos uma pessoa no frame e nos testes, não ocorreu falso positivo, apresentando resultados muito bons para uma pessoa. No entanto, ao detectar mais de duas pessoas, houve a contabilização de apenas uma pessoa em um frame que continha duas. Isso ocorreu, em média, a cada cinco frames, sendo dois registrados com apenas uma pessoa. Nos testes, foram constatadas perdas consideráveis na aquisição de imagens causadas pela câmera, mas que não impediram a detecção das pessoas.

Palavras-chave: Aplicação móvel, Visão computacional, Monitoramento Residencial, Java, Python.

Abstract

This work proposes the development of a residential monitoring system, motivated by the large number of home burglaries in Brazil and the lack of customization in existing monitoring systems. This system will use a mobile application that will communicate with different camera models through the Real-Time Publish-Subscribe Protocol (RTSP).

The system architecture uses the Java programming language with the Spring Boot framework for building user, authentication, camera, and notification services, and Python, which is used for building the image processing server. This server has an algorithm for detecting people using computer vision concepts. The mobile application was built using React Native, with a Model-View-ViewModel (MVVM) architecture, making it more optimized and facilitating code maintenance.

Tests were conducted with one and two people in different scenarios with objects around. The image processing server was able to detect at least one person in the frame, and in the tests, no false positives occurred, presenting very good results for detecting one person. However, when detecting more than two people, it only detected one person in a frame that contained two. This occurred on average every five frames, with two frames registering only one person. In the tests, considerable image acquisition losses caused by the camera were observed, but these did not prevent the detection of people.

Keywords: Mobile Application, Computer Vision, Residential Monitoring, Java, Python.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

JVM	*Java Virtual Machine*
API	*Application Programming Interface*
ID	*Identifier*
RTSP	*Real-Time Streaming Protocol*
HTTP	*HyperText Transfer Protocol*
JPA	*Java Persistence API*
MVVM	*Model-View-ViewModel*
IP	*Internet Protocol*

Sumário

1 Introdução

Nos últimos anos, houve um aumento em tecnologias que têm o objetivo de facilitar e melhorar a vida da população. O avanço e melhoramento da tecnologia de comunicação possibilitou que as pessoas possam trocar mensagens de maneira assíncrona, sem precisar que elas estejam disponíveis naquele momento. Com isso, houve um aumento de dispositivos móveis, pois esses são os mais fáceis de levar consigo e acessar os aplicativos de comunicação.

Um dos conceito que está atrelado ao avanço da tecnologia e à melhoria da vida da população é a segurança residencial. Com o avanço das tecnologias, observou-se também um crescimento populacional em grandes polos. No Brasil, uma pesquisa do IBGE revelou que, em 2021, roubos em domicílios representaram 11,3% dos casos de roubo, mostrando a importância de melhorar a segurança residencial (Agência Brasil, 2022). Embora as tecnologias de segurança tenham evoluído, os sistemas convencionais ainda apresentam limitações, especialmente no que diz respeito à personalização.

A integração de sistemas de monitoramento com as plataformas móveis é uma inovação crucial, pois proporciona ao usuário uma forma de acessar informações em tempo real, em qualquer lugar que ele estiver, facilitando que o usuário possa comunicar à polícia o mais rápido possível e facilitando a ação policial para prender o envolvido. Essa abordagem não só melhora a segurança residencial, como também a segurança pessoal, pois evitaria que o usuário encontrasse o assaltante em sua residência.

Este projeto de trabalho de conclusão de curso tem como objetivo implementar um sistema de monitoramento residencial utilizando uma aplicação móvel que irá se comunicar com diferentes modelos de câmeras, permitindo o controle remoto do dispositivo de segurança. O sistema fará uso de um algoritmo que irá fazer a detecção de indivíduos, utilizando conceitos de visão computacional. O usuário terá liberdade de definir horários de monitoramento e receberá em tempo real notificações sobre como está a sua residência.

A escolha de utilizar uma aplicação móvel se deve ao crescimento no número

de pessoas que utilizam dispositivos móveis no seu dia a dia. De acordo com o governo, o Brasil registrou mais de 234 milhões de acessos móveis em 2020 (Governo Federal, 2021). O que possibilita o acesso às notificações e informações de maneira rápida. Este trabalho se propõe a explorar as diversas etapas envolvidas na criação desse sistema, incluindo o design da aplicação, a utilização de algoritmos para a detecção de pessoas e a integração com o hardware, que consiste nas câmeras de segurança.

1.1 Objetivo Geral

O objetivo deste trabalho é desenvolver uma solução computacional personalizada para o monitoramento residencial, utilizando câmeras que possibilitem o uso do Real-Time Publish-Subscribe Protocol (RTSP).

1.1.1 Objetivos Específicos

Especificamente, este trabalho busca os seguintes objetivos:

- Desenvolver e integrar uma aplicação móvel que se comunique com câmeras de segurança;
- Implementar um algoritmo para a detecção de pessoas;
- Configurar horários de monitoramento;
- Enviar notificações em tempo real;

1.1.2 Contribuições

- Utilização de recursos tecnológicos para melhorar a segurança residencial;
- Mitigação de danos que podem ocorrer na residência, com o acionamento das autoridades de maneira rápida;
- Solução acessível à maioria da população;

- Prevenção de danos que assaltantes podem causar ao indivíduo, caso seja pego desprevenido.

1.2 Trabalhos Relacionados

Este capítulo apresenta uma análise de alguns trabalhos/aplicativos relacionados a aplicação que se pretende construir.

1.2.1 Sistema de monitoramento baseado em redes neurais artificiais e detecção de imagem

O trabalho desenvolve um protótipo de monitoramento ativado remotamente via aplicativo de mensagens. O sistema detecta faces humanas ou olhos pela câmera, enviando alertas e imagens ao celular do usuário e acionando o alarme local apenas quando necessário (ARMENDANI, 2018).

1.2.2 Detecção de Intrusão com Reconhecimento Facial em Imagens Geradas por Câmeras de Segurança

O trabalho propõe um framework para sistemas de segurança baseados em câmeras de vigilância, integrando técnicas de visão computacional. O sistema oferece funcionalidades como delimitação de regiões críticas, detecção e reconhecimento de faces, e geração de logs de intrusão, utilizando recursos avançados das câmeras e auxiliando na melhoria da segurança (KUROIWA; CARRO, 2023).

1.2.3 Considerações Finais

Os dois trabalhos apresentados possuem uma grande relação com a base deste trabalho. Entretanto, eles se diferenciam no tocante ao desenvolvimento de um aplicativo móvel que visa facilitar a usabilidade para o usuário. Esses dois trabalhos não desenvolveram uma aplicação móvel para integrar com as câmeras. O segundo trabalho possui uma aplicação de framework que podemos configurar os dias de monitoramento, mas não é tão simples quanto uma aplicação. Já neste trabalho, está presente esse

diferencial, que possibilita o armazenamento das informações coletadas de maneira acessível para o usuário comum, que possui o aplicativo.

2 Fundamentação Teórica

Este capítulo apresenta os conceitos explorados para o desenvolvimento do estudo referente ao monitoramento residencial por meio de uma aplicação móvel.

2.1 Linguagem de Programação

As linguagens de programação podem ser aplicadas em diferentes cenários, desde controle de usinas até em jogos eletrônicos em telefones celulares (SEBESTA, 2018). Por esse motivo, diferentes linguagens foram desenvolvidas para suprir as necessidades de determinadas áreas de forma eficiente.

2.1.1 Java

Java é uma linguagem de programação orientada a objetos, destacando-se principalmente por sua portabilidade, segurança e robustez. Como Horstmann afirma, apesar de existirem outras linguagens que oferecem também essas vantagens, a linguagem Java possui um diferencial com sua biblioteca consistente (HORSTMANN, 2019). Isso está relacionado à portabilidade: o código, quando compilado, pode ser executado em qualquer dispositivo que tenha uma Java Virtual Machine (JVM) compatível, sem necessidade de recompilação, pois o formato compilado, que é o bytecode, pode ser utilizado em qualquer outra máquina que possua a JVM. Um das ferramentas mais utilizados no ecossistema Java é o Spring Boot que facilita e agiliza o desenvolvimento de aplicativos da web e de microsserviços com o Java.

2.1.2 Python

De acordo com Eric Matthes no livro "Python Crash Course", a linguagem Python foi desenvolvida com o objetivo de ser fácil de ler e escrever. O código escrito tem uma sintaxe simples e limpa, permitindo que os programadores se concentrem em resolver problemas (MATTHES, 2019). Esse princípio torna as aplicações feitas

em Python mais fáceis de manter por conta da legibilidade. O ecossistema Python também é muito grande, pois existem várias bibliotecas que podem ser usadas tanto para análise de dados quanto para processamento de imagem.

2.1.3 React Native

O React foi desenvolvido pelo Facebook, é de código aberto e permite desenvolver aplicações móveis em multiplataforma (iOS e Android) usando JavaScript. Diferente de outras abordagens híbridas tradicionais, que utilizam WebView para renderizar os componentes, o React Native utiliza componentes que são nativos, o que resulta em aplicativos mais fáceis de implementar e com desempenho próximo ao nativo. A principal ideia do React Native é o conceito de "aprender uma vez, escrever em qualquer lugar". Isso quer dizer que, embora você use JavaScript para escrever a lógica do aplicativo, o React Native traduz os componentes de interface de usuário diretamente para seus equivalentes nativos (DABIT, 2019).

2.2 API

Uma Application Programming Interface (API) é uma interface que um programa apresenta para, humanos, o mundo web ou outro programas. Apesar das API serem projtadas para funcionarem para outros programas, elas são principalmente destinadas a serem compreendidas e usadas por humanos que escrevem esses outros programas (JIN; SAHNI; SHEVAT, 2019). Em tese as APIs são os blocos de construção que permitem a interoperabilidade para plataformas, possibilitando a comunicação de difentes négocios.

2.3 Arquitetura de Microsserviços

De acordo com Richardson, a arquitetura de microsserviços possui várias definições. Algumas interpretam o conceito literalmente e defendem que um serviço deve ser extremamente pequeno, por exemplo, contendo apenas 100 linhas de código (RICHARDSON, 2018). Mas, em geral, é uma abordagem para desenvolver aplicações

como uma grande coleção de serviços pequenos, autônomos e independentes. Cada microsserviço é responsável por uma parte específica do sistema e se comunica com outros serviços através de APIs. Isso possibilita que nosso programa seja bastante modular e fácil de manter, além de permitir o escalonamento do software.

2.4 Arquitetura Model-View-ViewModel (MVVM)

O padrão Model-View-ViewModel (MVVM) organiza o código em 3 camadas principias: A View que representa a interface gráfica do usuário; A ViewModel que faz a transferença de dados do Model em um formato que a View pode apresentar; e a Model que é responsável pelos dados que o ViewModel utiliza para fornecer informações à View (ENGINEERING, 2021).

2.5 Processamento de Imagem

O processamento de imagem digital envolve a manipulação de imagens digitais, com um dos objetivos de melhorar a sua qualidade ou extrair informações que podem ser relevantes. Uma imagem digital é composta por uma matriz de valores que representam a intensidade dos tons da imagem em uma grade de pixels. O objetivo do processamento de imagem é transformar essa representação digital para alcançar resultados específicos, como realce da imagem, segmentação, compressão, remoção de ruídos ou até mesmo análise de conteúdo, como reconhecimento de padrões (GONZALEZ; WOODS, 2018).

2.6 Visão Computacional

De acordo com o livro "Practical Python and OpenCV + Case Studies"de Adrian Rosebrock, podemos definir a visão computacional como um campo da inteligência artificial e da ciência da computação que busca ensinar os computadores a interpretar e entender o mundo visual, como seres humanos. Ela envolve o desenvolvimento de algoritmos e modelos que permitem que as máquinas "vejam"e abstraiam informações úteis a partir de imagens ou vídeos, identificando objetos que, para nós, são fáceis de distinguir, mas para as máquinas são difíceis (ROSEBROCK, 2016).

2.7 Docker

De acordo com Nigel Poulton, o Docker é definido como uma plataforma que automatiza o processo de implantação de aplicações dentro de um contêiner, fornecendo uma maneira padrão de empacotar e executar aplicações, garantindo que elas funcionem de maneira consistente, independentemente do ambiente onde serão executadas (POULTON, 2021).

O contêiner Docker é uma unidade leve e portátil que inclui tudo o que uma aplicação precisa para ser executada. Todas as dependências são supridas, o que elimina erros que podem ocorrer durante a execução em uma determinada plataforma e não em outra. Essa maneira de implementação de aplicações torna mais viável a utilização e manutenção de aplicações.

3 Metodologia

Este capítulo aborda a metodologia utilizada para a realização do projeto.

3.1 Descrição do problema

Os sistemas de segurança tradicionais apresentam várias limitações, como, por exemplo, a restrição aos modelos de câmeras compatíveis, monitoramentos pouco personalizáveis e a ocorrência de falhas ao identificar movimentos que não correspondem a pessoas.

3.2 Solução adotada

A solução será um sistema de monitoramento residencial, um aplicativo móvel capaz de se comunicar com diferentes câmeras de segurança que dispõem do protocolo RTSP. O sistema será configurado pelo usuário para realizar a detecção de pessoas em horários específicos. A integração envolve um servidor responsável por processar as imagens capturadas, identificar a presença de suspeitos e enviar notificações em tempo real ao aplicativo, proporcionando maior controle e segurança ao usuário. Os testes foram realizados utilizando uma câmera Intelbras Mibo IM4 C. As versões das tecnologias utilizadas estão na Tabela 1.

Tecnologia	**Versão**
React	18.3.1
React Native	0.76.7
Java	17
Spring Boot	3.3.4
Python	3.12.3

Tabela 1 – Versões das Tecnologias Utilizadas

3.2.0.1 Desenvolvimento do aplicativo

Para o desenvolvimento da aplicação mobile, utilizaremos o framework React Native. Será aplicada uma padrão MVVM, que tornará nosso código mais conciso, facilitando a integração com os serviços. Serão desenvolvidas as seguintes telas: tela de cadastro de usuário, login, lista de câmeras, cadastro de câmera, personalização de monitoramento, configuração de câmera e notificações.

De acordo com a Figura 1, a View será responsável por mostrar os componentes da nossa aplicação, enquanto o viewModel contém as regras de negócio e o Model possibilita a comunicação com a base de dados.

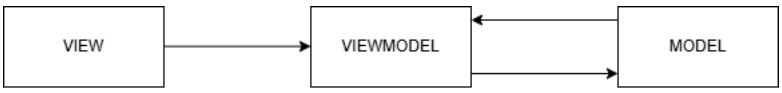

Figura 1 – Arquitetura MVVM
Fonte: Autor

3.2.0.2 Desenvolvimento das APIs em Java

Serão desenvolvidos microsserviços para cada funcionalidade da aplicação, utilizando a linguagem de programação Java. Os serviços que serão desenvolvidos incluem: autenticação, usuário, câmera e notificação. O serviço de autenticação tem como objetivo validar o usuário na aplicação, permitindo que ele acesse as informações de maneira segura.

O serviço de cadastro de usuário permitirá que o usuário acesse os recursos da aplicação, além de poder alterar informações já cadastradas. O serviço de câmera tem o objetivo de cadastrar as câmeras que serão utilizadas para o monitoramento, bem como gerenciar as câmeras ativas e configurar horários de monitoramento. O serviço de notificação será responsável por enviar as notificações recebidas do servidor de monitoramento, alertando o usuário diretamente no aplicativo.

Além de todos esses serviços, teremos um API Gateway, que será uma das principais aplicações, permitindo o acesso aos serviços por meio de suas chamadas. A aplicação móvel se comunicará exclusivamente com o Gateway, que funcionará como o ponto de acesso único para arquitetura. A partir do Gateway, as solicitações serão

redirecionadas para os serviços específicos. Também teremos o *Naming Server*, um servidor de registro de serviços, que permitirá englobar todos os serviços em nossa arquitetura.

3.2.0.3 Servidor para processamento de imagens

O servidor para processamento de imagem construído em Python é a parte principal, pois ele receberá os Internet Protocols (IPs) das câmeras para realizar o monitoramento e acionar o serviço de notificação quando algo for detectado por meio de requisições ao serviço. Nele será executado um algoritmo para detectar pessoas, sendo responsável por alertar a aplicação quando alguma pessoa for detectada dentro do perídio de monitoramento.

3.3 Arquitetura da Solução

A arquitetura da solução criada leva em consideração tanto as APIs construídas e a aplicação mobile desenvolvida.

De acordo com a Figura 2, a API Gateway será o ponto de acesso principal da nossa aplicação. Os serviços de autenticação, usuário, câmera e notificação serão registrados no *Naming Server*, que permitirá à API Gateway localizar nossos serviços. O banco de dados será utilizado para armazenar as informações de usuários, câmeras e notificações, além de senhas que é do serviço de autenticação. O processamento de imagem é um servidor dedicado a processar as informações que são obtidas pela câmera.

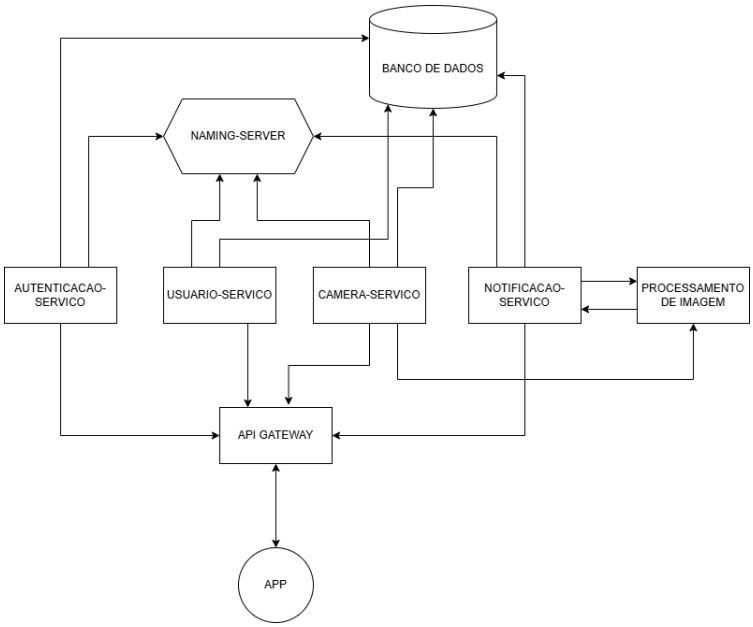

Figura 2 – Arquitetura da Solução
Fonte: Autor

3.4 Implementações das APIs em Java

Para a construção das APIs em Java foi utilizado o Spring Boot, que facilita o desenvolvimento de nossas aplicações, oferecendo uma ampla gama de recursos que garantem eficiência e resiliência nas aplicações em Java.

3.4.1 Serviço de Autenticação

O serviço de autenticação tem como finalidade proteger as informações do usuário e garantir que apenas usuários autenticados e com a conta correta possam acessar, modificar e criar informações referentes a câmeras de maneira segura.

Vamos fazer uso do Spring Security, uma biblioteca do Spring que fornece proteção e autenticação. Para isso, todos os projetos foram iniciados utilizando o *Spring Initializr*, que facilita a geração de projetos configurados com as dependências necessárias. A Figura 3 mostra a configuração do projeto.

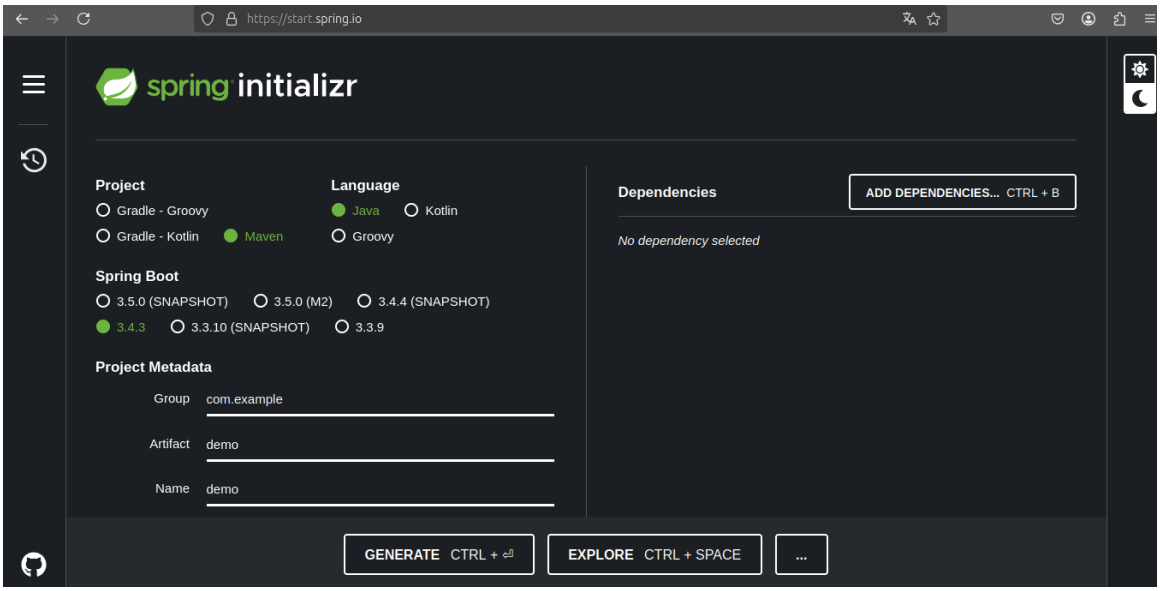

Figura 3 – Interface do Spring Initializr para configuração inicial do projeto
Fonte: Autor

Vamos fazer uso de um conceito de segurança baseado em chaves pública e privada. O serviço de autenticação irá gerar um token com base na chave privada. Na API Gateway, que funciona como a "porta de entrada", estará configurada a chave pública. Quando o usuário informar o seu token, o Spring Security verificará se ele está no formato esperado e se foi gerado a partir da chave privada no serviço de autenticação. Dessa forma, podemos garantir que apenas usuários com conta na aplicação possam acessar suas informações de maneira segura.

Para gerar esse token, o usuário precisa ter uma conta na aplicação. O serviço de autenticação também é responsável por criar essa conta. Para isso, é necessário informar o e-mail e a senha. Esse e-mail será utilizado para buscar o usuário no serviço de usuários. Caso um registro com esse e-mail seja encontrado, uma conta será criada; caso contrário, será retornado um erro.

A Figura 4 ilustra o processo de criação de conta. O usuário informa suas credenciais, como e-mail e senha, e a aplicação realiza a chamada ao serviço de usuários por meio do Feign, que permite a comunicação HTTP entre APIs. Caso o e-mail seja encontrado, o serviço de usuários retorna um status HTTP 200 (requisição

bem-sucedida) para o serviço de autenticação, que finaliza o processo de criação da conta retornando o status HTTP 202 (criado) para o cliente. Em caso de erro, como e-mail não encontrado, o serviço devolve diretamente um status HTTP 404 (usuário não encontrado).

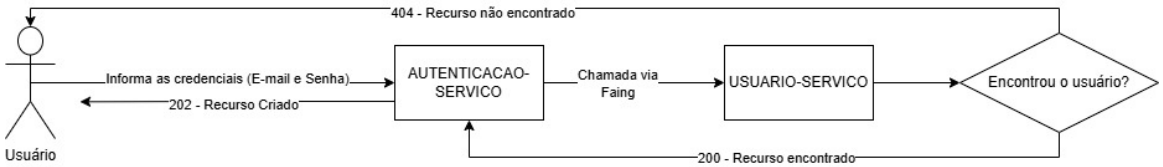

Figura 4 – Processo de criação de conta
Fonte: Autor

A geração de token funciona de maneira semelhante. O usuário deverá informar suas credenciais, como e-mail e senha. Em seguida, será realizada uma consulta na tabela de usuários. Caso o e-mail e a senha estejam corretos, um token será gerado, permitindo que o usuário utilize os demais serviços da aplicação.

3.4.2 Serviço de Usuário

O serviço de usuários foi inicializado da mesma forma que o serviço de autenticação por meio do *Spring Initializr*. Para persistir as informações do usuário, é necessário criar um modelo que será mapeado pelo Java Persistence API (JPA) em nosso banco de dados. O código abaixo mostra a classe que foi mapeada no banco de dados. O Trecho de Código 3.1 mostra a classe que foi mapeada no banco de dados.

```
1  @Entity
2  @Table(name = "tb_usuarios", schema = "public")
3  public class Usuario implements Serializable {
4      @Serial
5      private static final long serialVersionUID = 1L;
6      @Id
7      @GeneratedValue(strategy = GenerationType.IDENTITY)
8      private Long id;
9
10     @Column(nullable = false, length = 150)
```

```
11      private String nome;
12
13      @Column(nullable = false, unique = true, length = 255)
14      private String email;
15
16       @Column(nullable = false, length = 150)
17      private String senha;
18
19      public Usuario() {
20      }
21
22    // Getters e Setters
```

Trecho de Código 3.1 – Classe Usuario (Autor)

Essa classe servirá para criar o repositório, onde será realizada a comunicação real com o banco de dados, e também o Mapper, que é um mapeamento de uma classe que servirá como um objeto de transferência entre as camadas de serviço e controladores. No Trecho de Código 3.2, temos o repositório com um método que foi implementado para buscar por email do usuário, que é usado quando é feita a chamada via feign da API de autenticação.

```
1 @Repository
2 public interface UsuarioRepository extends JpaRepository<Usuario,
     Long> {
3      @Query("SELECT u FROM Usuario u WHERE u.email = :email")
4      Optional<Usuario> findByEmail(String email);
```

Trecho de Código 3.2 – Repositório de Usuario (Autor)

O controlador é responsável por expor os endpoints do serviço, permitindo que os clientes realizem chamadas para as funções disponibilizadas. O Trecho de Código 3.3 apresenta a implementação do controlador do serviço de usuários.

```
1 @RestController
2 @RequestMapping("api/usuario")
3 public class UsuarioController {
```

```
4       @Autowired
5       private UsuarioServiceImpl service;
6
7       @GetMapping(value = "/{id}",
8               produces = MediaType.APPLICATION_JSON_VALUE
9       )
10      public ResponseEntity<UsuarioDTO> findById(@PathVariable(value =
            "id") Long id){
11          UsuarioDTO usuarioDTO = service.findById(id);
12          return ResponseEntity.ok(usuarioDTO);
13      }
14
15      ...
16  }
```

Trecho de Código 3.3 – Controlador do serviço de Usuario (Autor)

Cada endpoint ou método desta classe pode ser chamado via HTTP para acionar uma funcionalidade. Por exemplo, a autenticação utiliza o endpoint findByEmail para buscar o usuário por meio do e-mail.

3.4.3 Serviço de Câmera

O serviço de câmera tem a finalidade de cadastrar as câmeras e configurar os parâmetros de monitoramento que serão utilizados no servidor de processamento de imagens. Para armazenar o registro das câmeras foi definida uma classe que representa a entidade correspondente no banco de dados. O Trecho de Código 3.4 apresenta a implementação dessa classe.

```
1   @Entity
2   @Table(name = "tb_cameras", schema = "public")
3   public class Camera implements Serializable {
4       @Serial
5       private static final long serialVersionUID = 1L;
6       @Id
7       @GeneratedValue(strategy = GenerationType.IDENTITY)
8       private Long id;
```

```
9       @Column
10      private String modelo;
11      @Column
12      private String marca;
13      @Column
14      private String ip;
15      @Column
16      private String usuarioCamera;
17      @Column
18      private String senhaCamera;
19      @Column
20      private Long usuarioId;
21      @Column(nullable = false)
22      @Enumerated(EnumType.STRING)
23      private Status status;
24
25      public Camera() {
26      }
27
28    // Getters e Setters
29  }
```

Trecho de Código 3.4 – Classe Camera (Autor)

O IP, o usuário e a senha são obtidos da câmera, pois utilizaremos o protocolo RTSP, um protocolo de rede usado para a transmissão de dados de áudio e vídeo em tempo real. Dessa forma, será possível conectar nossa câmera à arquitetura. Essas informações são cruciais para realizar o monitoramento.

A classe de monitoramento é responsável por definir os dias e horários em que uma câmera cadastrada será monitorada pelo servidor de processamento de imagens. Para isso, foi criada uma entidade que representa essa configuração no banco de dados. O Trecho de Código 3.5 apresenta a implementação dessa classe. Esta classe será responsável pela criação dos serviços e repositórios

```
1  @Entity
2  @Table(name = "tb_monitoramentos", schema = "public")
```

```
3  public class Monitoramento implements Serializable {
4      @Serial
5      private static final long serialVersionUID = 1L;
6      @Id
7      @GeneratedValue(strategy = GenerationType.IDENTITY)
8      private Long id;
9      @Column(nullable = false)
10     @Enumerated(EnumType.STRING)
11     private Dia dia;
12     @ManyToOne
13     @JoinColumn(name = "camera_id", nullable = false)
14     private Camera camera;
15     @Column(nullable = false)
16     private LocalTime horarioInicial;
17     @Column(nullable = false)
18     private LocalTime horarioFinal;
19
20     public Monitoramento() {
21     }
22
23   // Getters e Setters
24 }
```

Trecho de Código 3.5 – Classe Monitoramento (Autor)

Conforme descrito respectivamente nos Trechos de Códigos 3.6 e 3.7, serão adicionadas consultas personalizadas nos repositórios de câmera e monitoramento.

```
1 @Repository
2 public interface CameraRepository extends JpaRepository<Camera, Long
      > {
3     @Query("SELECT c FROM Camera c WHERE c.usuarioId = :usuarioId
          ORDER BY c.ip ASC")
4     List<Camera> findAllByUsuarioIdOrderByIp(Long usuarioId);
5 }
```

Trecho de Código 3.6 – Repositório da Câmera (Autor)

A consulta apresentada no Trecho de Código 3.6 permite que, ao listar as câmeras, elas sejam ordenadas pelo endereço IP, proporcionando uma apresentação mais organizada das informações no aplicativo. Além disso, a consulta definida no Trecho de Código 3.7 possibilita a busca de todos os monitoramentos relacionados a um ID de câmera específico. Isso facilita a localização dos monitoramentos associados a cada câmera, otimizando a operação do servidor de processamento de imagens.

```
1 @Repository
2 public interface MonitoramentoRepository extends JpaRepository<
    Monitoramento, Long> {
3     @Query("SELECT m FROM Monitoramento m WHERE m.camera.id = :
        cameraId")
4     List<Monitoramento> findAllByCameraId(Long cameraId);
5 }
```

Trecho de Código 3.7 – Repositório de Monitoramento (Autor)

3.4.4 Serviço de Notificação

O serviço de notificação possui duas funções. A primeira tem como objetivo receber imagens do servidor de processamento de imagem e armazená-las em um volume. Através do mesmo serviço, podemos obter a imagem em base64, que é mais leve e mais fácil de exibi-la. O funcionamento dessa parte do serviço está ilustrado na Figura 5.

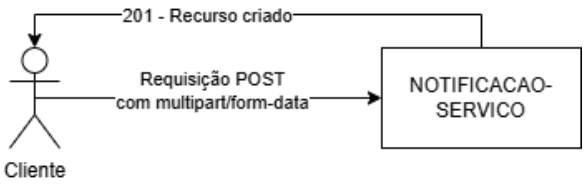

Figura 5 – Envio de imagem
Fonte: Autor

Na Figura 5, o cliente é o servidor de processamento de imagem. Ele enviará, por meio de multipart/form-data, a imagem e o ID da notificação para que o nosso serviço

possa atrelá-la à notificação gerada. Dessa forma, quando o servidor de processamento de imagem identificar uma pessoa, ele poderá armazenar essa imagem, e a nossa aplicação poderá recuperá-la por meio de requisições do tipo GET, na Figura 6 é possível observar esse processo.

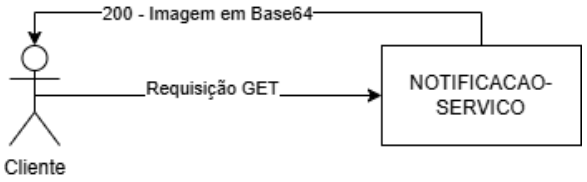

Figura 6 – Recuperação da imagem
Fonte: Autor

A segunda funcionalidade é o envio de e-mails para alertar o usuário quando uma pessoa for detectada. Quando o servidor de processamento de imagem detecta uma ou mais pessoas, ele aciona o endpoint do serviço de notificação, que registra as informações detectadas e envia a notificação para o e-mail do usuário, além de acionar o serviço de push notification para alertar no dispositivo móvel do usuário cadastrado.

3.5 Implementação do Servidor de Processamento de Imagem

O servidor de processamento de imagem é responsável por se conectar com a câmera por meio das informações obtidas pelo serviço de câmera e enviar as informações para o serviço de notificação, que serão transmitidas para a aplicação móvel.

O Trecho de Código 3.8 apresenta a função responsável por iniciar o monitoramento. Ela faz a chamada para consultar a API que irá iniciar o procedimento de monitoramento de câmeras.

```
1     def iniciar_monitoramento():
2         try:
3             consultar_api("http://192.168.0.12:8888")
4         except KeyboardInterrupt:
5             print("Monitoramento encerrado")
```

```
6
7     if __name__ == "__main__":
8         iniciar_monitoramento()
```

Trecho de Código 3.8 – Função para iniciar o monitoramento (Autor)

O Trecho de Código 3.9 mostra a função responsável pelo consumo da API. Ela realiza a consulta para obter todas as câmeras e seus respectivos monitoramentos, e, em seguida, chama a função para monitorar cada câmera.

```
1  def consultar_api(ws):
2      while True:
3          try:
4              response = requests.get(f'{ws}/camera-service/camera/
                   todas')
5              if response.status_code == 200:
6                  cameras = response.json()
7
8                  for camera in cameras:
9                      monitoramentos_response = requests.get(f'{ws}/
                           camera-service/monitoramento/camera/{camera["
                           id"]}')
10                     if monitoramentos_response.status_code == 200:
11                         monitoramentos = monitoramentos_response.
                               json()
12                         monitorar_cameras(camera, monitoramentos, ws
                               )
13             else:
14                 print("Erro ao consultar API de cameras")
15
16         except Exception as e:
17             print(f"Erro na consulta a API: {e}")
18         time.sleep(5)
```

Trecho de Código 3.9 – Função de consumo de api (Autor)

Essa função tem como objetivo buscar todas as câmeras para monitorar aquelas que estão ativas e, depois, chamar a função de monitoramento de câmeras, caso a

busca de câmeras seja bem-sucedida. Também é feito o uso de threads com o objetivo de paralelizar o consumo das câmeras, pois existe uma lista de câmeras que devemos monitorar.

O Trecho de Código 3.10 mostra a função responsável por monitorar a câmera. Ela conecta-se via RTSP à câmera e realiza a detecção de movimento no vídeo, acionando notificações caso detecte movimento.

Essa função vai estabelecer a conexão com a nossa câmera por meio da URL formada, que é obtida dos dados da câmera que serão informados. Também vamos criar um subtrator de fundo usando o método MOG2 do OpenCV. Esse subtrator de fundo é usado para detectar movimento em cenas, e o parâmetro *detectShadows=True* permite que ele detecte sombras, ajudando a melhorar a precisão da detecção.

```
1  def monitorar_cameras(camera, monitoramentos, ws):
2      detected_persons = {}
3      url = f"rtsp://{camera['usuarioCamera']}:{camera['senhaCamera']}
           @{camera['ip']}:554/cam/realmonitor?channel=1&subtype=0"
4
5      background_subtractor = cv2.createBackgroundSubtractorMOG2(
           detectShadows=True)
6      cap = cv2.VideoCapture(url)
7
8      dias_semana = {
9          "SEGUNDA": 0,
10         "TERCA": 1,
11         "QUARTA": 2,
12         "QUINTA": 3,
13         "SEXTA": 4,
14         "SABADO": 5,
15         "DOMINGO": 6
16     }
17
18     while cap.isOpened():
19         try:
20             ret, frame = cap.read()
```

```
21              if not ret:
22                  print("Erro ao ler o quadro, tentando novamente...")
23                  continue
24
25              frame = cv2.resize(frame, (320, 240), interpolation=cv2.
                    INTER_AREA)
26
27              now = datetime.now()
28              for monitoramento in monitoramentos:
29                  horario_inicial = datetime.strptime(monitoramento['
                        horarioInicial'], '%H:%M:%S').time()
30                  horario_final = datetime.strptime(monitoramento['
                        horarioFinal'], '%H:%M:%S').time()
31                  inicio = dt_time(horario_inicial.hour,
                        horario_inicial.minute)
32                  fim = dt_time(horario_final.hour, horario_final.
                        minute)
33                  dia_semana_monitoramento = dias_semana.get(
                        monitoramento['diaSemana'].upper(), -1)
34
35                  if dia_semana_monitoramento == now.weekday() and
                        inicio <= now.time() <= fim:
36                      frame, pessoas = identificar_pessoa(frame,
                            background_subtractor)
37
38                      if pessoas > 0:
39                          ultima_notificacao = detected_persons.get(
                                camera['id'], None)
40                          if not ultima_notificacao or (now -
                                ultima_notificacao).seconds > 60:
41                              titulo = f"Movimento Detectado: {pessoas
                                    } pessoas"
42                              conteudo = f"Movimento detectado na
                                    camera ID {camera['id']} ({camera['
                                    modelo']})"
43                              enviar_notificacao(camera['id'], camera[
                                    'usuarioID'], titulo, conteudo, frame
                                    , ws)
44                              detected_persons[camera['id']] = now
```

```
45
46              cv2.imshow(f"Monitoramento - {camera['id']}", frame)
47
48              if cv2.waitKey(30) & 0xFF == ord('q'):
49                  break
50
51          except Exception as e:
52              print(f"Erro ao processar camera {camera['id']}: {e}")
53              break
54
55      cap.release()
56      cv2.destroyAllWindows()
```

Trecho de Código 3.10 – Função de monitoramento de câmera (Autor)

Depois faremos um loop enquanto a conexão estiver aberta, com o objetivo de capturar um frame e passá-lo para a função de identificar pessoas. Nessa função, vamos passar o frame e o nosso subtrator de fundo para podermos identificar se há pessoas nesse frame. O monitoramento será realizado com base na lista de monitoramentos personalizados que estão relacionados com a câmera.

Se forem detectadas pessoas, será criada uma notificação com o título e conteúdo apropriados, e chamaremos a função *enviar_notificacao*. Depois disso, liberamos a câmera para a entrada de um novo frame.

A função *identificar_pessoa* é uma das mais importantes no processo de monitoramento. Nela, estará os valores pré-configurados para o tamanho máximo e mínimo do objeto a ser detectado. Esses valores foram definidos com base em testes, nos quais pessoas passaram pela câmera para determinar os limites que permitem a identificação de uma pessoa. A função recebe como parâmetros o frame da câmera e o subtrator de fundo. Primeiro, ela converte o frame para escala de cinza e aplica o subtrator de fundo para obter uma máscara, que será usada para encontrar os contornos do objeto. Esses contornos correspondem às bordas em movimento. Para cada contorno encontrado, a função calcula a área e verifica se ela está dentro dos limites pré-definidos. Se estiver, ela adiciona ao contador de pessoas detectadas. O Trecho de Código 3.11 apresenta a implementação da função *identificar_pessoa*.

```
1 def identificar_pessoa(frame, background_subtractor, min_size=600,
     max_size=1000):
2     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
3     fg_mask = background_subtractor.apply(gray)
4
5     contours, _ = cv2.findContours(fg_mask, cv2.RETR_EXTERNAL, cv2.
         CHAIN_APPROX_SIMPLE)
6     pessoas = sum(1 for cnt in contours if min_size < cv2.
         contourArea(cnt) < max_size)
7
8     return frame, pessoas
```

Trecho de Código 3.11 – Função identificar_pessoa para detecção de movimento

Se uma ou mais pessoas forem identificadas, isso acionará uma notificação, que será enviada para o serviço de notificação junto com a imagem. A aplicação móvel, por sua vez, consultará essas informações para alertar o usuário.

3.6 Implementação da Aplicação Móvel

A aplicação móvel vai seguir um padrão MVVM, onde terá uma camada de view construída com componentes do React Native, uma camada de viewModel responsável por gerenciar a lógica de apresentação e os dados intermediários, e uma camada de model que será responsável por interagir com a nossa arquitetura criada.

3.6.1 Camada de View

A camada de view será composta por componentes do React Native que irão renderizar as telas que desenvolveremos. Os componentes serão organizados de forma modular, facilitando a reutilização de código e a manutenção da aplicação. Desenvolveremos as telas de login, cadastro de usuário, listagem de câmeras, cadastro de câmeras e configuração do monitoramento. A camada de view segue um padrão, pois nela sempre será chamado o viewModel referente a essa view, com as funções adequadas àquela tela.

O Trecho de Código 3.12 segue o padrão de view, onde chamamos o viewModel, responsável pela parte lógica. Todas as funções de ação são chamadas pelo viewModel, que no caso é o *useLoginViewModel*. No exemplo acima, temos o view de login, onde chamamos nossa classe de credencial, que é responsável por padronizar as informações. Dentro dos inputs de texto, temos a adição dos textos aos parâmetros da classe. Esse padrão é repetido em toda a nossa aplicação.

```
export const LoginView: React.FC = () => {
    const {
        credencial,
        setCredencial,
        login,
        cadastroUsuarioNavigation,
        isLoading
    } = useLoginviewModel();

    return (
        <PaperProvider>
            <View style={globals.container}>
                {isLoading() ?
                    <IndicatorComponent /> :
                    <View>
                        // Implementacao de componentes
                    </View>
                }
            </View>
        </PaperProvider>
    );
};
```

Trecho de Código 3.12 – Exemplo de implementação de View (Autor)

3.6.2 Camada de viewModel

A camada de *ViewModel* terá a responsabilidade de atuar como intermediária entre a camada de *View* e a camada de *Model*. Ela irá manipular os dados por meio de

funções, que serão utilizadas para realizar a persistência ou recuperação dos dados na camada de modelo. Nessa camada, haverá uma validação básica dos dados fornecidos pelo usuário.

O Trecho de Código 3.13 é o viewModel de login. Aqui, temos a validação e chamada dos métodos da camada de modelo para persistir ou recuperar informações da aplicação. Por meio da classe, temos recursos para validar informações rápidas para garantir que iremos chamar os métodos da camada de modelo apenas se forem cumpridas as validações. Esse padrão será seguido para todos os viewModels da aplicação.

```
export const useLoginviewModel = () => {
  const { dispatch, navigation, isLoading } = viewModelHook();

  const [credencial, setCredencial] = useState<Credencial>(new
     Credencial());

  const autenticacaoHandler = new AutenticacaoHandler();

  const cadastroUsuarioNavigation = () => {
    navigation.navigate("CadastroUsuario");
  };

  const login = async (): Promise<void> => {
       // Logica da implementacao
  };

  return {
    credencial,
    setCredencial,
    login,
    cadastroUsuarioNavigation,
    isLoading
  };
};
```

Trecho de Código 3.13 – Exemplo de implementação de ViewModel

3.6.3 Camada de Model

A camada de *Model* será responsável por interagir com a arquitetura criada. Dessa forma, poderemos persistir ou recuperar os dados dos serviços que foram desenvolvidos na arquitetura. Essa camada terá as funções necessárias para realizar as devidas chamadas dos verbos HTTP, utilizando bibliotecas como Axios para efetuar as requisições. A camada de *Model* é essencial, pois ela é nossa integração entre a arquitetura criada e a aplicação desenvolvida. Na aplicação, a camada *Model* é composta por Handlers (manipuladores) e Services (Serviços), sendo o handler uma classe que engloba serviços familiares de uma mesma controller de nossa API.

O Trecho de Código 3.14 é o padrão para estender todas as demais handlers da aplicação. Assim, temos os métodos básicos de cada controller, que são busca, criação, atualização e deleção. Com essa classe, o código se tornará menos repetitivo. Nela, temos uma função que irá transformar todas as strings que são vazias em nulas, para não termos o risco de salvar valores que não condizem com os dados que queremos exibir. Por exemplo, não seria desejável para nós possuir uma marca de câmera com caractere vazio, entretanto, optamos por não armazenar essa informação. O Trecho de Código 3.15 representa essa implementação.

```
1  export class BaseHandler {
2      protected service: any;
3      protected path: string;
4
5      constructor(path: string, service: any) {
6          this.service = service;
7          this.path = path;
8      }
9
10     async find(pathArray: any[] = [], params: Record<string, any> |
           null = null): Promise<Response> {
11         return await this.service.get(params, this.path, pathArray);
12     }
13
14     // Implementacao dos metodos: post, put e delete
15 }
```

Trecho de Código 3.14 – Classe base para a implementação dos Handlers

O Trecho de Código 3.15 apresenta uma classe que será responsável por estender o serviço principal, a API Service, e estabelecer a comunicação com os microsserviços desenvolvidos. Nele, temos os verbos HTTP, que são o GET, POST, PUT e DELETE, que vão chamar os nossos endpoints dos nossos microsserviços. Esses verbos são chamados por meio da implementação dos métodos da classe. Por exemplo, no método GET, temos algumas validações que serão feitas na formatação para chamar o endpoint de maneira correta. Neste método, por exemplo, realiza-se a validação para evitar o envio de valores nulos, além da separação dos parâmetros de path e query para a construção da URL a ser requisitada.

```
1  export class BaseService {
2      private baseURL: string;
3      private headers: Record<string, string>;
4
5      constructor(baseURL: string, headers: Record<string, string> =
           generateHeader()) {
6          this.baseURL = baseURL;
7          this.headers = headers;
8      }
9
10     async get(
11         params: Record<string, any>,
12         path: string,
13         pathArray: any[],
14     ): Promise<Response> {
15         try {
16                 // Logica da implementacao
17             };
18         } catch (error: any) {
19                 // Caso ocorra um erro
20         }
21     }
22
```

```
23      // Implementacao dos metodos: post, put e delete
24  }
```

Trecho de Código 3.15 – Classe base para a implementação dos Services

3.6.4 Conteinização

Após a conclusão de todas essas etapas, foi feita a conteinerização dos serviços, uma vez que essa abordagem permite a execução dos mesmos de forma eficiente, sem sobrecarregar o ambiente de desenvolvimento. Para realizar a conteinerização das aplicações em Java, foi utilizado o Docker. Criaremos em cada serviço Java um arquivo *Dockerfile* que servirá para adquirir todos os recursos necessários para rodar a aplicação e criar o contêiner. O arquivo Dockerfile 3.16 tem a seguinte configuração:

```
1  FROM openjdk:17-jdk-slim
2  WORKDIR /app
3  COPY target/*.jar /app/app.jar
4  EXPOSE $PORTA
5  ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

Trecho de Código 3.16 – Dockerfile (Autor)

onde *$PORTA* é a porta que iremos abrir para poder rodar a aplicação. Dessa forma podemos fazer o builder da aplicação e containizar para poder rodar sem consumir muito recurso da máquina.

4 Resultados

Neste capítulo serão abordados os resultados obtidos do projeto.

4.1 Funcionamento dos microsserviços com a aplicação móvel

4.1.1 Servidor de registro de serviços

É necessário que o sistema inicie primeiramente com o servidor de registro de serviços, pois com ele será possível que a API Gateway encontre nossos serviços e faça o redirecionamento adequado a cada requisição que será enviada pela aplicação móvel.

Como podemos ver pela Figura 7, todos os serviços foram iniciados com sucesso, principalmente a API Gateway.

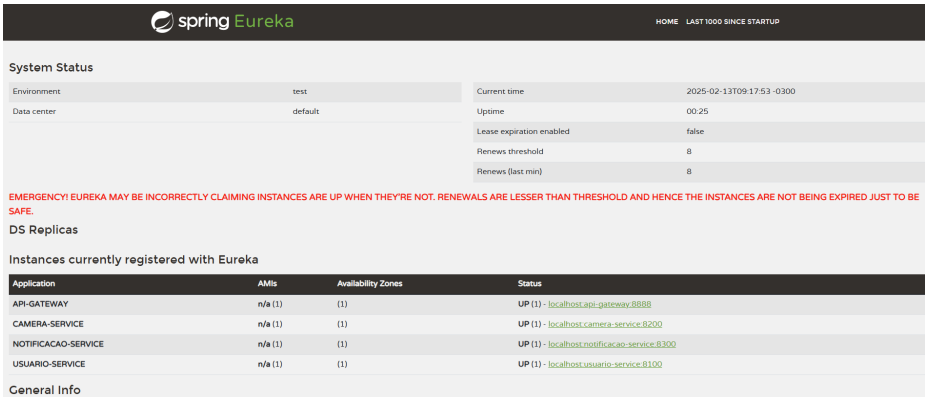

Figura 7 – Eureka iniciado com sucesso
Fonte: Autor

4.1.2 Cadastro de Usuário na Aplicação

Ao iniciar a aplicação móvel, será exibida a tela de login, que permitirá acessar a tela de cadastro de usuário para realizar a autenticação em nosso sistema. As Figuras

8 e 9 mostram a tela de login e a tela de cadastro de usuário, respectivamente.

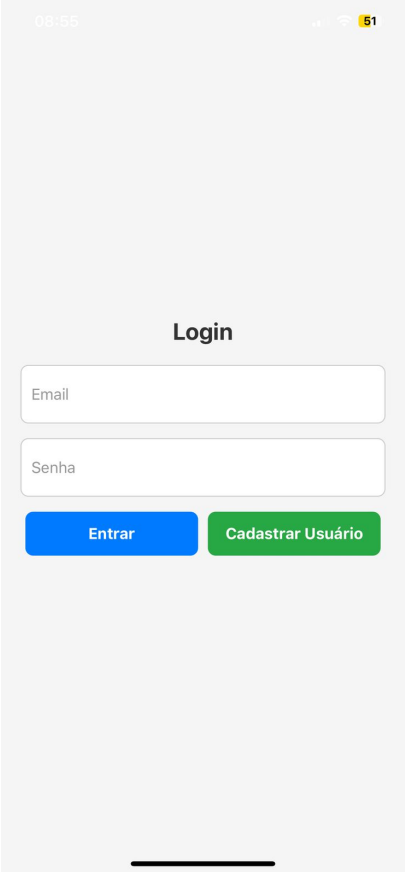

Figura 8 – Tela de Login
Fonte: Autor

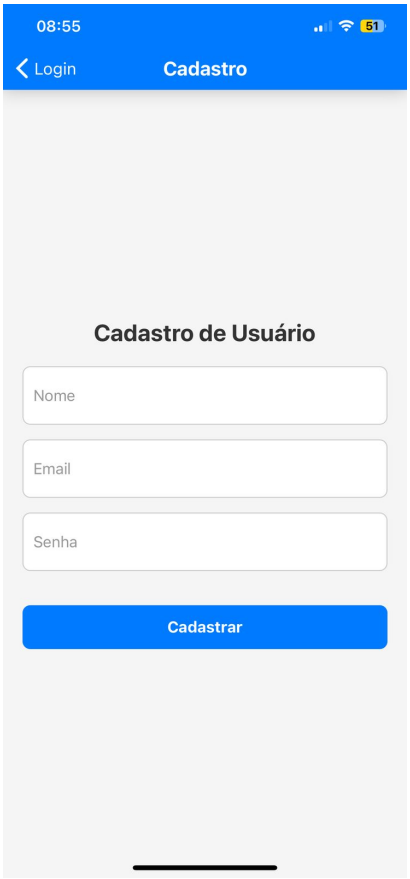

Figura 9 – Tela de Cadastro de Usuário
Fonte: Autor

Para realizar o cadastro de usuário, deveremos informar o nome, email e senha, sendo o email responsável por informar ao usuário sobre qualquer notificação que ele receber vindo do serviço de processamento de imagem. Ao realizar o cadastro das informações, o usuário será redirecionado para a tela de login para poder se autenticar na aplicação. As Figuras 10 e 11 ilustram esse processo.

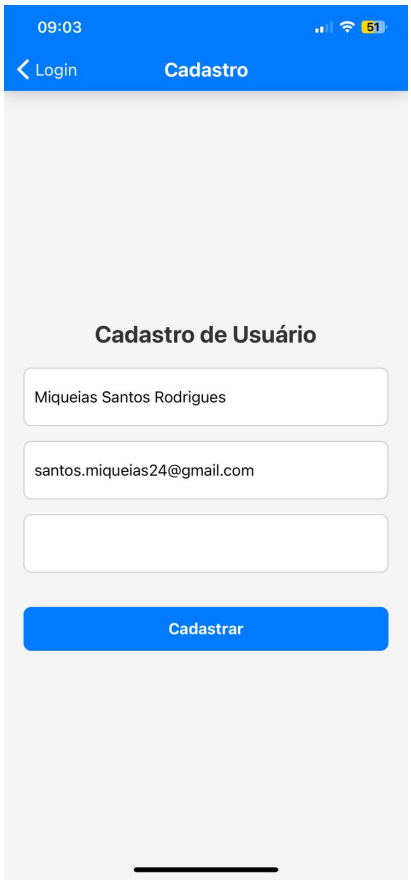

Figura 10 – Cadastro de Usuário
Fonte: Autor

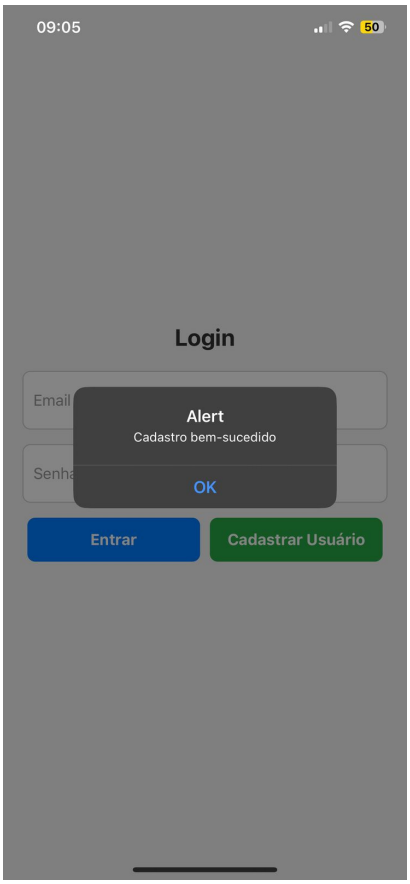

Figura 11 – Redirecionado para tela de login
Fonte: Autor

4.1.3 Cadastro de Câmera

Após autenticar na aplicação, podemos cadastrar a nossa câmera, informando o IP da câmera, que pode ser facilmente obtido ao acessar a configuração do roteador e verificar qual é o IP utilizado pela câmera. Também devemos informar o usuário e a senha da câmera. As Figuras 12 e 13 ilustram a navegação até o cadastrado de câmera. A senha da câmera geralmente pode ser obtida por uma etiqueta que existe embaixo da câmera ou em seu manual, assim como o usuário. Com essas informações, podemos cadastrar nossa câmera e fazer a personalização do monitoramento.

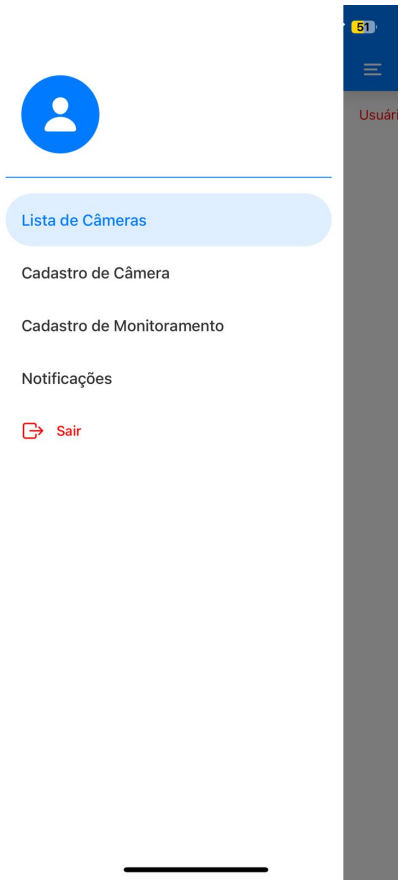

Figura 12 – Navegando até a câmera
Fonte: Autor

Figura 13 – Tela de cadastro de câmera
Fonte: Autor

Após a configuração dos parâmetros da câmera, um objeto de câmera será criado. As Figuras 14 e 15 ilustram a criação do objeto câmera.

Figura 14 – Informando os parâmetros da Câmera
Fonte: Autor

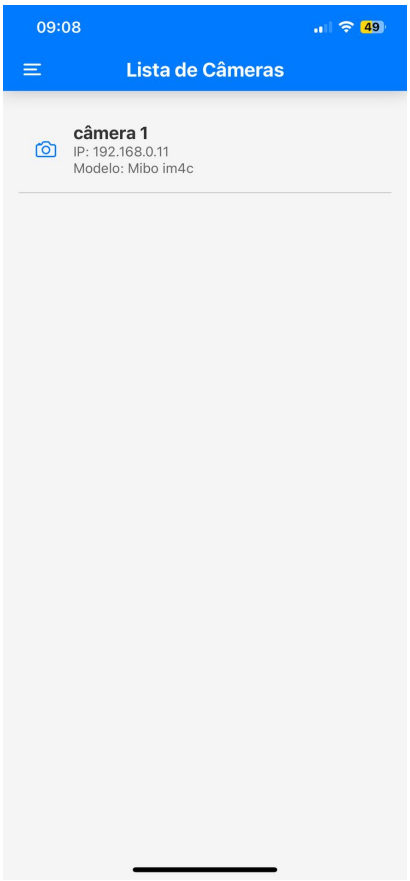

Figura 15 – Listagem de câmera cadastrada
Fonte: Autor

4.1.4 Cadastro de Monitoramento

Após cadastrar a câmera no sistema, é possível adicionar monitoramentos para especificar quais aspectos ou áreas serão monitorados. As Figuras 16 e 17 ilustram como realizar o cadastro de monitoramento.

Na Figura 16, é possível ver a tela de cadastro de monitoramento. Nesta tela, o usuário pode definir os parâmetros específicos do monitoramento, como dia da semana, horário de inicio do monitoramento e final.

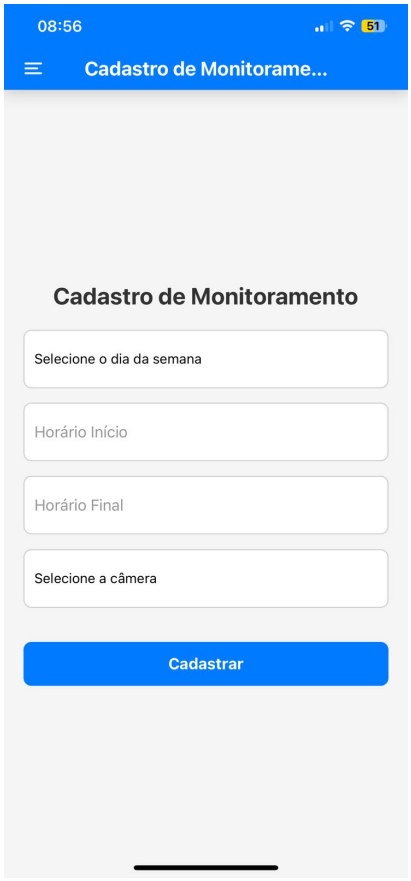

Figura 16 – Tela de Monitoramento
Fonte: Autor

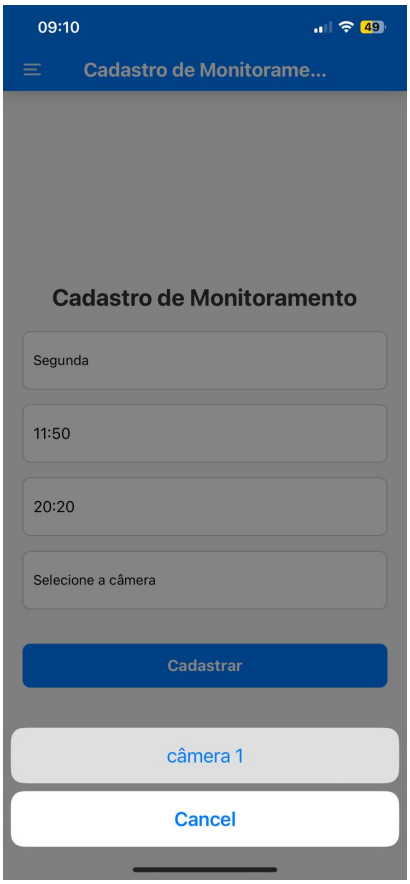

Figura 17 – Seleção da câmera que iremos usar
Fonte: Autor

A Figura 17 mostra a seleção da câmera que será utilizada para o monitoramento. O sistema permite escolher entre várias câmeras cadastradas anteriormente. Após concluir o cadastro do monitoramento, Figura 18, todos os monitoramentos associados a uma câmera específica serão exibidos ao selecionar a respectiva câmera. Essas informações serão então enviadas ao servidor de processamento de imagem, que executará o monitoramento conforme as configurações definidas.

Na Figura 19, vemos um exemplo de monitoramento cadastrado, onde os detalhes definidos pelo usuário são exibidos. A figura apresenta uma lista de todos os monitoramentos cadastrados. Com essas configurações, o servidor de processamento

de imagem poderá realizar o monitoramento conforme os parâmetros estabelecidos.

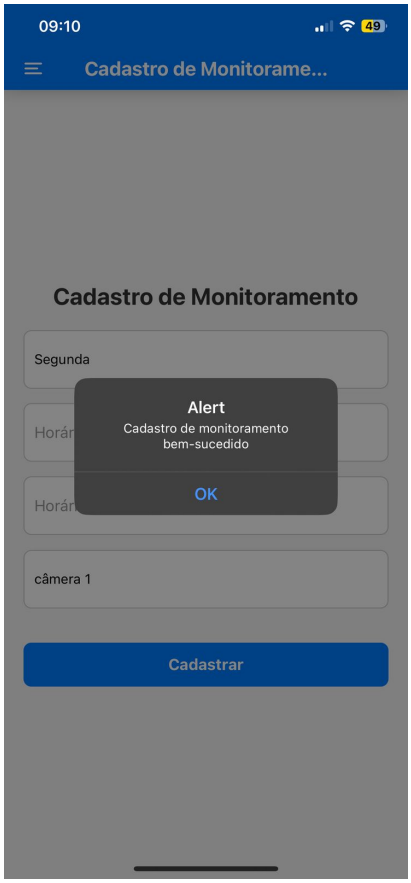

Figura 18 – Monitoramento cadastrado
Fonte: Autor

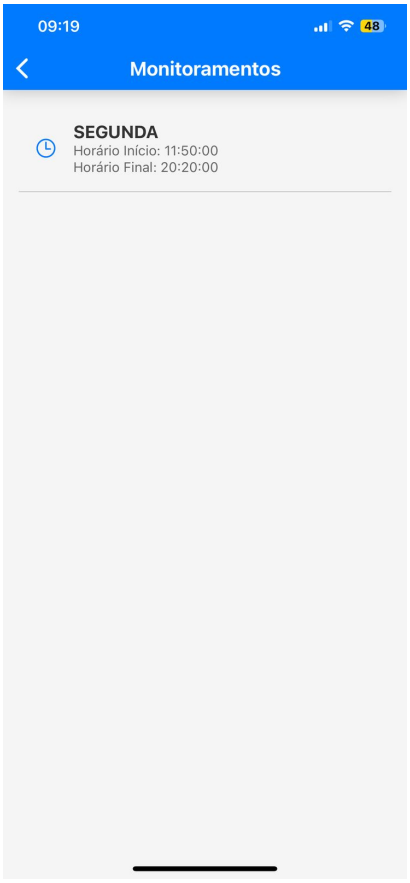

Figura 19 – Lista de monitoramento
Fonte: Autor

4.2 Detecção de Pessoas

Ao iniciar o servidor de processamento de imagem, este estabelecerá comunicação com os demais serviços para obter os dados das câmeras cadastradas. A Figura 20 ilustra a execução do servidor, onde, no console é possível observar o salvamento e o envio da notificação para o aplicativo. Além disso, ocorrem algumas perdas de frames devido a erros na decodificação da imagem proveniente da câmera. Com relação aos frames obtidos nesse intervalo de tempo, havia a presença de uma pessoa e vários

objetos, porém, apenas uma pessoa foi detectada. Isso ocorreu devido à configuração da função *identificar_pessoa*, na qual foi definido um parâmetro de 600 a 1000, limitando a detecção a objetos em movimento que estejam dentro desse intervalo de tamanho.

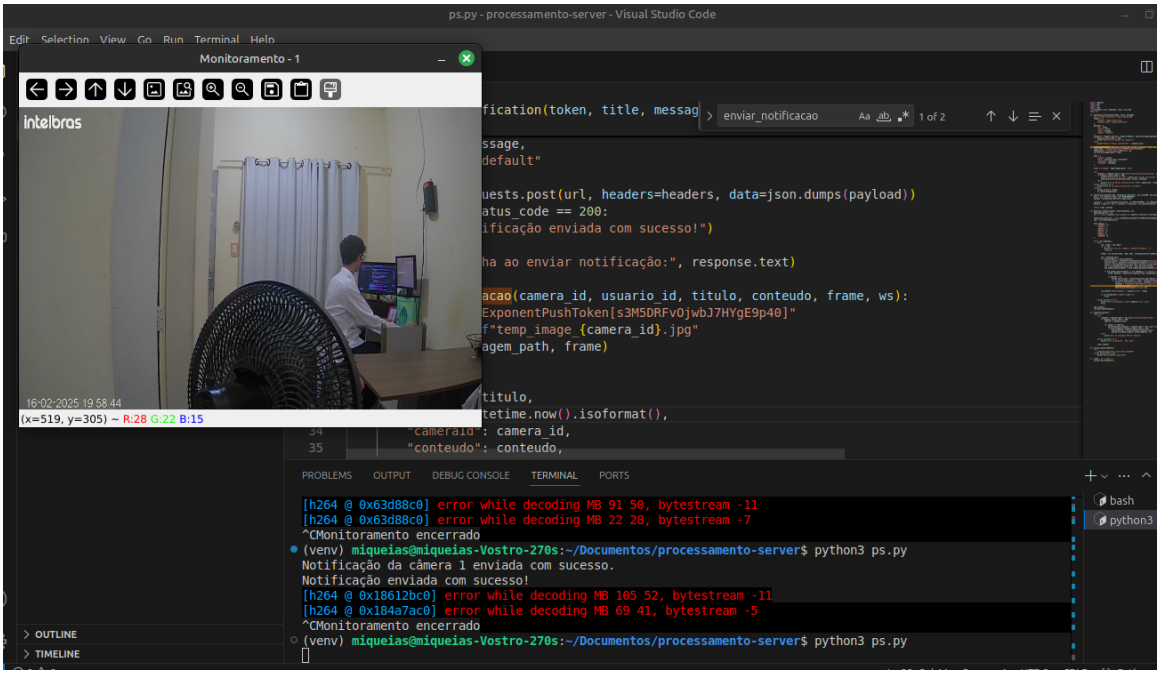

Figura 20 – Servidor de Processamento de Imagem
Fonte: Autor

Ao realizar a detecção de uma ou mais pessoas, o serviço de notificação e o push notification são acionados. O primeiro irá persistir essa notificação no banco de dados para consulta na aplicação, além de enviar um e-mail informando o ocorrido. Já o segundo irá gerar um alerta no dispositivo móvel. A Figura 21 apresenta a notificação recebida no dispositivo, exibindo a quantidade de pessoas detectadas e o ID da câmera. Já a Figura 22 mostra a notificação que chegou no email cadastrado do usuário, demonstrando que o serviço de notificação foi acionado com sucesso e o disparo de email ocorreu de maneira adequada.

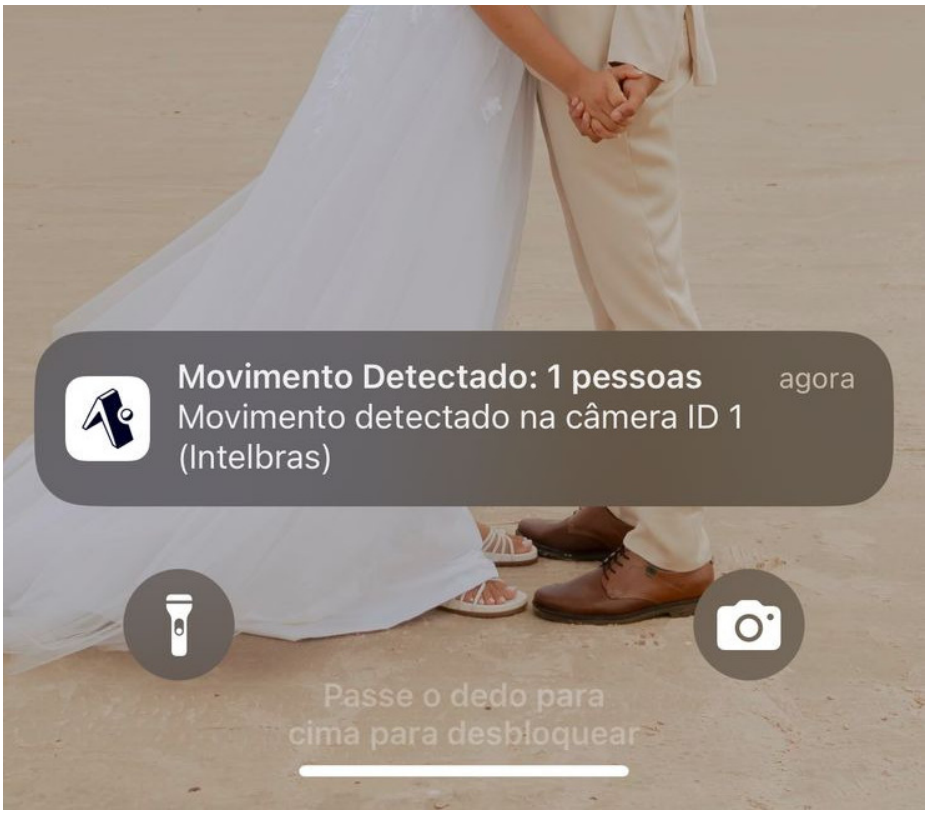

Figura 21 – Notificação no dispositivo móvel
Fonte: Autor

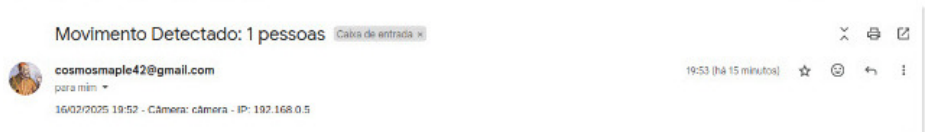

Figura 22 – Notificação no email do usuário
Fonte: Autor

A Figura 23 mostra todas as notificações obtidas no teste da aplicação, sendo a última notificação a primeira imagem obtida pelo nosso servidor. Isso mostra que as notificações estão sendo persistidas na base de dados. A Figura 24 é referente à última notificação obtida, que é a mesma imagem que o nosso servidor mostra na Figura 20, demonstrando que a função presente no servidor de notificação de armazenamento de arquivos está funcionando corretamente.

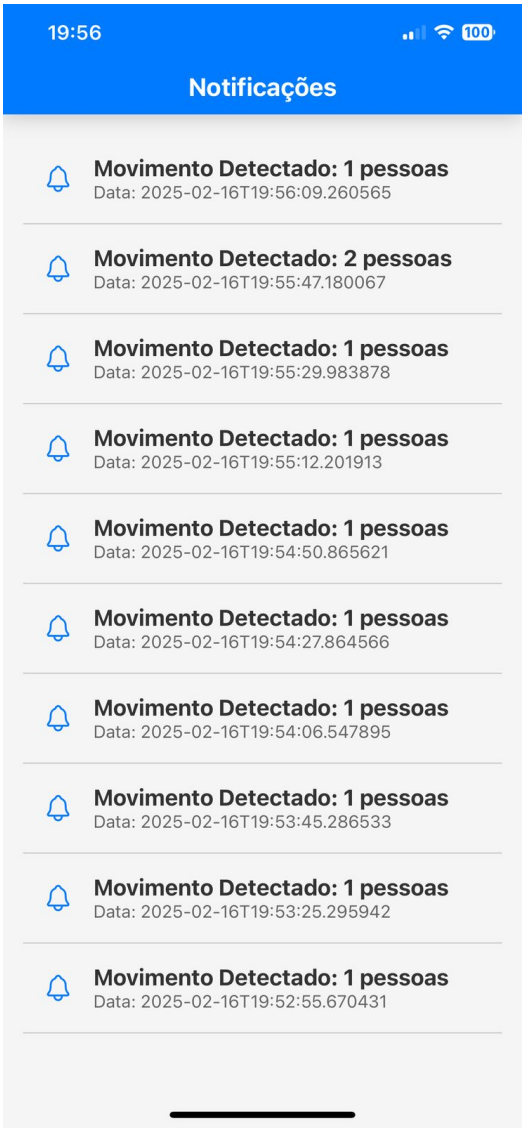

Figura 23 – Notificações registradas pelo serviço de notificação
Fonte: Autor

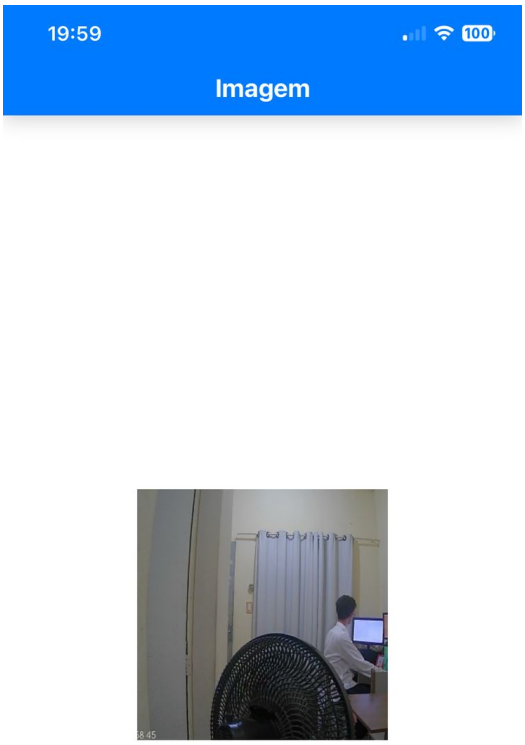

Figura 24 – Imagem que foi detectado uma pessoa em movimento
Fonte: Autor

Foram realizados testes com duas pessoas para verificar se foi possível identificá-las. Na Figura 25, as duas pessoas foram identificadas; entretanto, ocorreu uma falha na obtenção da imagem, o que pode indicar uma dificuldade do nosso servidor em trazer todas as imagens sem falha. A Figura 20 evidencia que muitos dos frames da

câmera sofrem problemas na hora de serem decodificados. Já na Figura 26, foram identificadas também duas pessoas, entretanto, a imagem foi obtida perfeitamente, os registros estão presentes. A Figura 27 mostra a notificação que chegou no aparelho móvel enquanto estava bloqueado a tela.

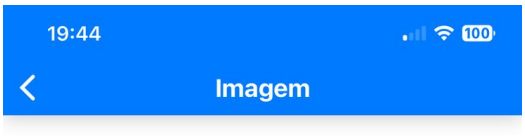

Figura 25 – Imagem com duas pessoas, imagem parcialmente falha na aquisição
Fonte: Autor

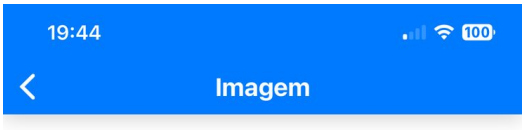

Figura 26 – Imagem com duas pessoas, sem falhas
Fonte: Autor

Figura 27 – Notificações das duas pessoas identificadas
Fonte: Autor

Nos testes realizados, não houve detecção de pessoas quando estas não estavam presentes. Contudo, observou-se razoavelmente a identificação de apenas uma pessoa em um frame onde havia duas pessoas, de cada 5 imagens com duas pessoas presentes duas delas foram registradas como tendo apenas uma pessoa.

5 Conclusão

Este trabalho propôs um sistema de monitoramento residencial personalizável que se comunica com diferentes câmeras por meio do protocolo RTSP. A partir das análises dos resultados obtidos, concluiu-se que a arquitetura da solução atingiu os objetivos propostos, visto que foi possível a detecção de pessoas nos frames registrados pelo servidor de processamento de imagem, além de um aplicativo móvel que permite personalizar o monitoramento da câmera, com dias da semana e horários específicos em que o monitoramento será realizado. A partir do aplicativo móvel, é possível receber notificações oriundas da arquitetura, informando se alguma pessoa foi detectada no intervalo de monitoramento cadastrado.

Ademais, foi implementada a funcionalidade de cadastro de câmeras, na qual o usuário informa um apelido, marca, modelo, IP, usuário e senha da câmera, que serão utilizados para realizar o cadastro do monitoramento da câmera, permitindo que cada câmera tenha seu próprio monitoramento. As seções 3.4.2 e 3.4.3 mostram os métodos utilizados para alcançar a construção dessa funcionalidade com êxito.

Além disso, o desenvolvimento do servidor de processamento de imagens obteve resultados ótimos na detecção de pessoas nos frames, sem apresentar falsos positivos em nenhum dos cenários testados. Entretanto, o servidor apresentou um número considerável de perda de frames, mas nada que impedisse a detecção de pessoas. Outra ressalva é na detecção de mais de uma pessoa, onde o servidor conseguiu detectar uma pessoa, mas não a outra, quando existia duas pessoas na cena. Em média, a cada cinco fotos com duas pessoas, duas foram registradas com apenas uma pessoa na cena.

Por fim, acredita-se que o desenvolvimento deste trabalho ocorreu com êxito, visto que todas as funcionalidades do sistema de monitoramento e do aplicativo móvel que foram planejadas foram implantadas, além de obter os resultados esperados. A arquitetura da solução demonstrou resiliência na comunicação entre todos os microsserviços, desde o usuário, autenticação, câmera e notificações, pois nenhum serviço sofreu qualquer queda.

Referências

Agência Brasil. *Pesquisa do IBGE mostra subnotificação de roubos e furtos no Brasil*. 2022. Acesso em 12 set. 2024. Disponível em: <https://agenciabrasil.ebc.com.br/geral/noticia/2022-12/pesquisa-do-ibge-mostra-subnotificacao-de-roubos-e-furtos-no-brasil>. Citado na página 11.

ARMENDANI, W. A. *Sistema de Monitoramento Baseado em Redes Neurais Artificiais e Detecção de Imagem*. Dissertação (Mestrado) — Faculdades Doctum de Caratinga, Caratinga, 2018. Citado na página 13.

DABIT, N. *React Native in Action*. [S.l.]: Manning Publications, 2019. ISBN 9781617294051. Citado na página 16.

ENGINEERING, D. *Android MVVM and Repositories in the Real World*. 2021. Accessed: 2024-09-12. Disponível em: <https://medium.com/draftkings-engineering/android-mvvm-and-repositories-in-the-real-world-e54c3582f832>. Citado na página 17.

GONZALEZ, R. C.; WOODS, R. E. *Digital Image Processing*. 4th. ed. [S.l.]: Pearson, 2018. ISBN 9780133356724. Citado na página 17.

Governo Federal. *Brasil registrou mais de 234 milhões de acessos móveis em 2020*. 2021. Acesso em 12 set. 2024. Disponível em: <https://www.gov.br/pt-br/noticias/transito-e-transportes/2021/05/brasil-registrou-mais-de-234-milhoes-de-acessos-moveis-em-2020>. Citado na página 12.

HORSTMANN, C. S. *Core Java Volume I: Fundamentals*. 11. ed. Boston: Pearson Education, 2019. 1-2 p. ISBN 9780135166307. Citado na página 15.

JIN, B.; SAHNI, S.; SHEVAT, A. *Designing Web APIs: Building APIs That Developers Love*. [S.l.]: O'Reilly Media, 2019. 1-1 p. ISBN 978-1492026919. Citado na página 16.

KUROIWA, B. T.; CARRO, S. Detecção de intrusão com reconhecimento facial em imagens geradas por câmeras de segurança. *Faculdade de Informática - FIPP, Universidade do Oeste Paulista - UNOESTE*, 2023. Citado na página 13.

MATTHES, E. *Python Crash Course*. 2nd. ed. San Francisco: No Starch Press, 2019. Citado na página 15.

POULTON, N. *Docker Deep Dive*. 3rd. ed. [S.l.]: Nigel Poulton Publications, 2021. ISBN 978-1916585195. Citado na página 18.

RICHARDSON, C. *Microservices Patterns: With examples in Java*. Manning Publications, 2018. 8-8 p. ISBN 978-1617294549. Disponível em: <https://www.manning.com/books/microservices-patterns>. Citado na página 16.

ROSEBROCK, A. *Practical Python and OpenCV + Case Studies*. [S.l.]: PyImageSearch, 2016. ISBN 9780981392115. Citado na página 17.

SEBESTA, R. W. *Conceitos de Linguagens de Programação*. 11. ed. São Paulo: Pearson Education do Brasil, 2018. 23-24 p. ISBN 9788543004792. Citado na página 15.