UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Tecnologia

Gabriel Domingues Ferreira

RASTREAMENTO DE EXPOSIÇÃO À DOENÇAS INFECCIOSAS POR APLICATIVO CELULAR

Limeira
2021

Gabriel Domingues Ferreira

RASTREAMENTO DE EXPOSIÇÃO À DOENÇAS INFECCIOSAS POR APLICATIVO CELULAR

Monografia apresentada à Faculdade de Tecnologia da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Plínio Roberto Souza Vilela

Este exemplar corresponde à versão final da Monografia defendida por Gabriel Domingues Ferreira e orientada pelo Prof. Dr. Plínio Roberto Souza Vilela.

Limeira
2021

FOLHA DE APROVAÇÃO

Abaixo se apresentam os membros da comissão julgadora da sessão pública de defesa de dissertação para o Título de Bacharel em Sistemas de Informação na área de concentração , a que se submeteu o aluno Gabriel Domingues Ferreira, em 30 de novembro de 2021 na Faculdade de Tecnologia – FT/UNICAMP, em Limeira/SP.

Prof. Dr. Plínio Roberto Souza Vilela
Presidente da Comissão Julgadora

Profa. Dra. Segunda Avaliadora
Instituição da segunda avaliadora

Dr. Terceiro Avaliador
Instituição do terceiro avaliador

Ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria de Pós Graduação da FT.

Resumo

O início do ano de 2020 foi marcado pela pandemia causada pelo coronavírus, e por conta disso, diversos países perceberam as ameaças que doenças infecciosas podem trazer à humanidade e voltaram suas atenções para combater sua transmissão. Uma das soluções para diminuir a alta taxa de transmissão da doença foi o rastreamento de contatos. Esse rastreamento acontece da seguinte forma: no momento que uma pessoa é diagnosticada como infectada, ela deve responder algumas perguntas sobre quem ela se encontrou nos últimos dias. As pessoas que forem rastreadas a partir dessas perguntas devem se isolar mesmo que não estejam apresentando sintomas, para impedir que o vírus continue se espalhando.

Essa é uma maneira pouco eficiente de rastreamento por dois principais motivos: o primeiro é que é totalmente manual, isso reflete na necessidade de muitos recursos humanos serem alocados para entrevistar os infectados, fazer ligações, envio de e-mails e mensagens para as pessoas que tiveram contato com ele. E quanto mais recursos envolvidos nesse processo, mais caro ele se torna. O segundo motivo é que o rastreamento depende inteiramente das lembranças do indivíduo, ou seja, se ele não se recordar de alguém que teve contato, o rastreamento acaba falhando em encontrar todas as pessoas que podem ter sido expostas.

Com base nesse contexto, neste trabalho foi proposto e desenvolvido um aplicativo para dispositivos móveis que armazena os lugares visitados pelos usuários, quando algum usuário for infectado por uma doença infecciosa, o aplicativo verificará automaticamente as pessoas que estiveram no mesmo local e intervalo de tempo que ele e as notificará, informando que podem ter sido expostas a alguma doença e que devem se isolar para impedir novos casos de infecção.

Abstract

The beginning of the year 2020 was marked by the pandemic caused by the coronavirus, and because of that, several countries realized the threats that infectious diseases can bring to humanity and turned their attention to combat its transmission. One of the solutions to reduce the high rate of transmission of the disease was contact tracing. This tracing works as follows: the moment a person is diagnosed as infected, he must answer some questions about who he has met in the past few days. People who are traced for these questions should isolate themselves even if they are not showing symptoms, just to prevent the virus from spreading.

This is an inefficient way of tracking for two main reasons: the first is that it is completely manual, this reflects the need for many human resources to be allocated to interview the infecteds, make phone calls, send e-mails and messages to people who had contact with him. And the more resources involved in this process, the more expensive it becomes. The second reason is that the contact tracing depends entirely on the individual's memories, that is, if he does not remember someone who had contact, the tracing ends up failing to find all the people who may have been exposed.

Based on this context, in this project it was proposed and developed an application for mobile devices that stores the locations visited by users and when one of them is infected by an infectious disease, the application will automatically check people who have been in the same place, in the same time and will notify them, stating that they may have been exposed to the disease and that they should isolate themselves to prevent new cases of infection.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

1.1 Contexto

A pandemia de COVID-19 mudou radicalmente as percepções do mundo em relação a propagação de doenças infecciosas como uma importante ameaça à humanidade. De acordo com o Instituto Ipsos (GEBREKAL, 2021), o coronavírus se estabeleceu como a maior preocupação da população mundial por mais de um ano consecutivo.

Durante uma conferência realizada em Vancouver, Gates (2015) afirmou que especialistas alertavam que o mundo não estava preparado para o surgimento de um novo vírus transmissível pelo ar com alta taxa de contágio. A facilidade que a humanidade possui em se locomover rapidamente para qualquer lugar do mundo através de aeroportos, ferrovias e estradas, a falta de um sistema de combate ao proliferamento de doenças infecciosas, métodos de prevenção e equipes de epidemiologistas altamente treinados contribuíram para o crescimento descontrolado dos casos de COVID-19 no ano de 2020.

Com o objetivo de diminuir a propagação do vírus, as mídias sociais foram bombardeadas com informações sobre métodos de prevenção, tais como lavar as mãos, evitar o compartilhamento de objetos pessoais, uso de máscaras, isolamento social e o rastreamento de contatos.

O rastreamento de contatos tem o objetivo de encontrar as pessoas que tiveram contato com as que foram infectadas pelo vírus, e instruí-las para não se encontrarem com família e amigos, ir aos mercados, bancos, ou seja, permanecerem em isolamento social em suas casas, tudo isso com o objetivo de evitar que a corrente de transmissão do vírus continue e diminuir a sua taxa de contágio.

Apesar do rastreamento reduzir a propagação do vírus avisando as pessoas expostas que elas podem estar infectadas, existe um grande problema: tudo é feito manualmente. Isso envolve um custo humano muito alto. São necessários muitos profissionais para que o rastreamento tenha algum resultado expressivo.

Além do alto custo para o governo manter a operação funcionando, ela depende integralmente da memória do paciente. Se o paciente entrevistado não lembrar de todas as pessoas que teve contato, ou se ele não souber os dados referentes às pessoas que ele esteve nas últimas semanas, isso afeta diretamente no resultado desse método de prevenção. Se muitas pessoas que foram expostas não forem identificadas, a corrente de transmissão continuará crescendo.

Quando o número de pessoas envolvidas em uma solução é alto demais para arcar com os custos ou quando existe um alto risco humano envolvido, que nesse caso é o risco relacionado à memória do paciente, uma tecnologia de automação surge como solução, e nesse caso não é diferente.

1.2 Motivação

A motivação desse trabalho reside na pandemia de coronavírus que se iniciou no começo de 2020. Soluções tecnológicas têm potencial de colaborar na diminuição de casos da doença, e consequentemente na preservação da vida humana.

A prioridade deste trabalho é automatizar o processo de rastreamento de contatos, que em sua grande parte é feito de maneira manual. Isso trará benefícios para o governo do país, que não necessitará de muitos recursos para manter o aplicativo funcionando.

Dessa forma, menos recursos deverão ser alocados para o rastreamento de contatos, ele será feito de forma mais assertiva, já que não dependerá da memória dos pacientes e sim de um Banco de Dados (BD) e a solução proposta neste trabalho demanda somente o uso dos dispositivos móveis dos usuários, recurso já existente na maioria da população brasileira.

1.3 Objetivos

O objetivo geral deste trabalho é o rastreamento automático de contatos. Esse rastreamento é uma aplicação para dispositivos móveis que armazena locais visitados pelos usuários e analisa se houve algum contato entre um usuário infectado com outros que possam ter sido expostos à

doença. Essa análise é feita utilizando rotinas no servidor da aplicação, que compara os locais e envia uma notificação ao dispositivo caso houver a possibilidade de exposição.

Com isso, o aplicativo deve ser capaz de automatizar todo o processo de rastreamento de contatos, para que usuários que forem possivelmente expostos à uma doença infecciosa transmissível pelo ar sejam notificados, e a partir disso adotem medidas de segurança e evitem ter contato com outras pessoas, para que o alastramento da doença seja contido.

A solução proposta é um aplicativo celular que será responsável tanto pela coleta das localizações visitadas quanto pela notificação de possíveis exposições aos usuários.

1.4 Estrutura do texto

O projeto foi dividido em 7 capítulos e cada um deles possui uma responsabilidade específica.

O Capítulo 2 explica qual foi a metodologia utilizada no processo de pesquisa e desenvolvimento deste trabalho.

No Capítulo 3 é feita a apresentação das tecnologias e dos conceitos teóricos utilizados e das pesquisas feitas sobre o tema.

O Capítulo 4 contém a busca por patentes registradas e de aplicações correlatas ao tema deste projeto.

O Capítulo 6 contém o levantamento e análise de requisitos, a modelagem dos diagramas e da arquitetura utilizada no desenvolvimento e a explicação da implementação das funcionalidades.

O Capítulo 5 explica quais recursos de infraestrutura são necessários para que a aplicação funcione e qual a relação entre cada um deles.

Por fim, o Capítulo 7 contém as conclusões do trabalho.

Capítulo 2

Metodologia

Para elaboração desse trabalho uma série de etapas foram concluídas. Essas etapas formam a metodologia seguida durante todo o processo de pesquisa e desenvolvimento do sistema.

Foi feita uma reunião com o orientador em que foi sugerido o problema a ser explorado pelo trabalho e quais possíveis caminhos poderiam ser seguidos para encontrar uma solução.

Após essa reunião, foi realizada uma pesquisa de exploração de soluções existentes em relação ao rastreamento de contatos, utilizando ferramentas de busca na *Internet*. A partir dessa busca, foi identificado o problema explicitado no Capítulo 1 e algumas hipóteses foram levantadas para serem possíveis soluções.

Antes de começar o levantamento e análise de requisitos, foi realizada uma busca no Instituto Nacional da Propriedade Industrial, buscando por patentes relacionadas ao tema deste projeto. Além disso, soluções atuais para o problema também foram analisadas, elas foram encontradas na loja oficial de aplicativos para celulares *Android*.

O levantamento e análise de requisitos foi feito a partir da solução proposta e de funcionalidades analisadas em outras aplicações atuais disponíveis no mercado. Como consequência dos requisitos, foi escolhida a solução a ser explorada pelo trabalho.

Foram feitas pesquisas referentes à diferentes arquiteturas de *software*, procurando por uma que atenda aos casos de uso do projeto e ofereça boa qualidade de código.

Pesquisas em relação à *Unified Modeling Language* (UML) foram realizadas, com o objetivo de entender e levantar quais diagramas deveriam ser modelados para melhor análise do sistema.

Após isso, a modelagem dos diagramas foi realizada, utilizando os diagramas de caso de uso, de sequência, classes, pacotes, atividade e componentes. A maioria desses diagramas se

encontram na Seção 6.2. Além de diagramas UML, também foi feita a modelagem de dados do BD.

Pesquisas em relação à infraestrutura foram realizadas, como o sistema possui um *backend* e um BD remoto, essa pesquisa teve como objetivo entender quais componentes eram necessários, as suas responsabilidades e como interagem entre si.

Depois de toda análise de requisitos e diagramação do sistema, uma pesquisa sobre fluxos de trabalho utilizando a ferramenta *Git* foi feita, com o objetivo de escolher o melhor método de trabalho com integração contínua, separando cada *branch* de desenvolvimento com sua própria responsabilidade e aumentando ainda mais a qualidade do código desenvolvido.

Todas as telas do aplicativo foram construídas utilizando o programa *Figma*, com a principal preocupação sendo a experiência do usuário, para que seja um aplicativo simples, intuitivo e agradável de ser utilizado.

Antes de começar o desenvolvimento, pesquisas sobre quais linguagens de programação devem ser utilizadas no sistema foram realizadas, tendo em vista todos os requisitos levantados, a arquitetura e infraestrutura do sistema. Essas linguagens devem possuir suporte as decisões tomadas anteriormente.

Com a linguagem escolhida, a última pesquisa a ser feita antes da implementação foi sobre princípios de desenvolvimento de *software*. Essa pesquisa foi importante para que o sistema possua alta qualidade, legibilidade e outras características que são detalhadas no Capítulo 3.

Por fim, o desenvolvimento do sistema foi feito, juntamente com pesquisas de bibliotecas de auxílio para facilitar algumas operações específicas da aplicação. O desenvolvimento do sistema é composto pela codificação do aplicativo móvel, do *backend* e de *pipelines* responsáveis pela integração contínua.

Capítulo 3

Referencial teórico

Para sustentar o desenvolvimento deste trabalho, padrões de desenvolvimento, de modelagem, linguagens de programação e outros assuntos foram pesquisados. Assim, a Seção 3.1 trata das pesquisas realizadas em relação à doenças infecciosas, procurando entender quais são os tipos de doenças que o aplicativo pode ajudar no rastreamento. Na Seção 3.2 são elencados os tipos de diagramas UML que foram utilizados e os motivos dessa decisão. Na Seção 3.3 é explicado qual a arquitetura de *software* utilizada na codificação do aplicativo, como ela funciona e quais são os benefícios de sua utilização. A Seção 3.4 possui a explicação do Padrão de Repositório. A Seção 3.5 possui o propósito de explicar qual foi o *framework* de desenvolvimento escolhido e o motivo de sua escolha. Na Seção 3.6 é explicado cada um dos princípios utilizados no processo de desenvolvimento da aplicação. A Seção 3.7 trata da arquitetura utilizada no processo de *design* do sistema, explicando seu funcionamento e benefícios. Na Seção 3.8 são elencadas as principais características do tipo de BD escolhido para o sistema. A Seção 3.9 explica sobre o fluxo de trabalho baseado na ferramenta *Git* que foi utilizado no desenvolvimento do projeto. Na Seção 3.10 os benefícios de utilização de *pipelines* de integração contínua são elencados, juntamente com os objetivos de utilização dessa ferramenta. A Seção 3.11 explica o motivo de utilização de serviços na nuvem, quais foram necessários e os principais motivos dessa escolha.

3.1 Doenças infecciosas

O rastreamento de contatos que o aplicativo automatizará não serve como método de prevenção e controle de qualquer doença transmissível. Por isso, pesquisas relacionadas à

doenças infecciosas foram realizadas para entender quais são os tipos que o aplicativo ajudará na prevenção.

Segundo Duncan et al. (2013), a transmissão de doenças contagiosas podem ocorrer de forma direta ou indireta. A transmissão direta é a transferência direta e imediata de agentes infecciosos a uma porta de entrada receptiva no hospedeiro, que ocorre no contato direto com pele e mucosas.

A transmissão indireta ocorre por meio dos três mecanismos primários: veículos, vetores e ar. Veículo é qualquer objeto que sirva como meio pelo qual o agente infeccioso se transporta a um hospedeiro, alguns exemplos são os brinquedos, utensílios de cozinha e instrumentos cirúrgicos. Vetores são artrópodes, que são divididos entre vetores mecânicos e biológicos. A transmissão aérea ocorre quando partículas que contem os agentes infecciosos ficam suspensas no ar por longos períodos de tempo.

As doenças contagiosas que o rastreamento de contatos efetuado pelo aplicativo celular suporta são relacionadas as de transmissibilidade indireta, principalmente as que ocorrem através do ar. Com isso, alguns exemplos de doenças contagiosas que o aplicativo pode suportar são o coronavírus, a gripe, sarampo e a catapora (ODA, 2021).

3.2 *Unified Modeling Language*

Como parte do processo de análise do sistema, as diagramações foram feitas seguindo os padrões UML, que é uma linguagem de modelagem utilizada para especificar, visualizar e documentar modelos de sistemas de *software*, incluindo sua estrutura e *design* (UML, 2005).

Existem diversos tipos de diagramas UML e eles são divididos em duas categorias: os estruturais e os comportamentais. A Figura 3.1 representa os tipos de diagrama de cada uma das categorias.

A partir dos tipos de diagramas disponíveis, foram utilizados apenas seis deles. Esses seis diagramas foram escolhidos a partir do objetivo que cada modelagem deveria atender.

O primeiro deles tem a necessidade de modelar as funcionalidades necessárias para os atores que estão envolvidos com o sistema, representando os seus casos de uso. Por isso, esta diagramação foi feita utilizando o diagrama de casos de uso.

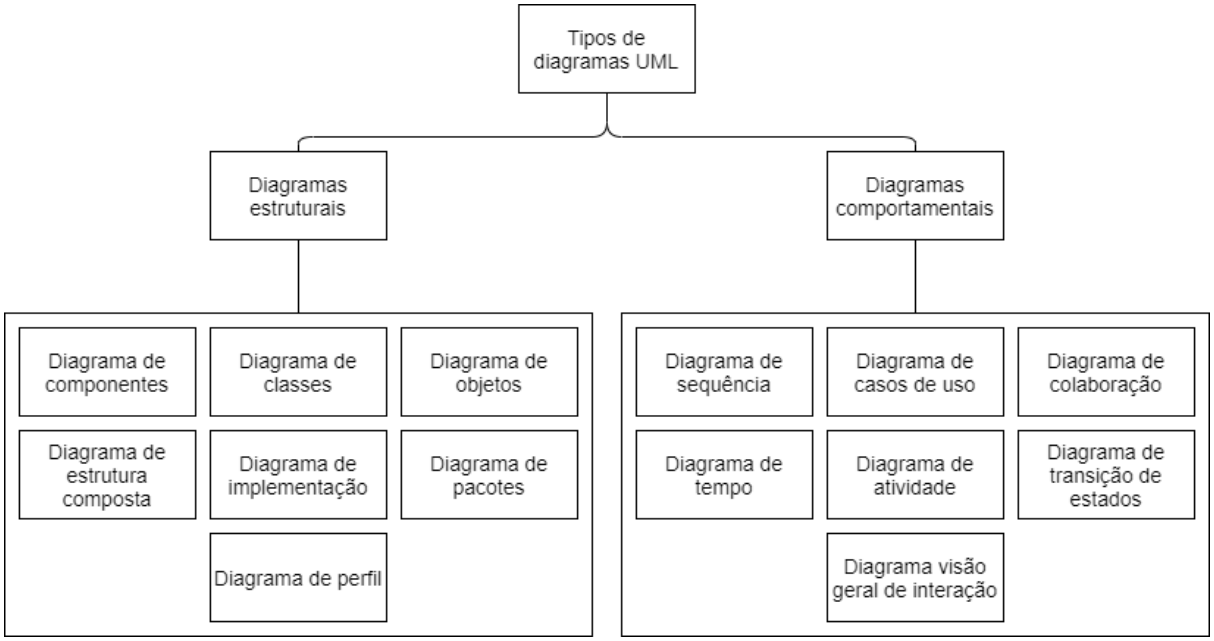

Figura 3.1: Tipos de diagramas UML

Para cada funcionalidade, foi necessário um diagrama que representasse a sequência de trocas de mensagens entre os objetos do sistema, e o diagrama UML utilizado que possui esse propósito é o diagrama de sequência.

Com o propósito de clarificar o que uma funcionalidade deve fazer, trazendo melhor documentação e explicação sobre a mesma, foi utilizado o diagrama de atividades. Esse diagrama é, essencialmente, um gráfico de fluxo que mostra o fluxo de controle entre uma atividade para outra. Neste trabalho, isso envolve a modelagem das etapas sequenciais de uma funcionalidade.

Como o sistema foi desenvolvido utilizando programação orientada a objetos, um diagrama para representar as classes, atributos, métodos e o relacionamento entre elas foi necessário. O diagrama que atende essa necessidade e foi utilizado no trabalho é o diagrama de classes.

Para representar a arquitetura de software do aplicativo móvel desenvolvido foi utilizado o diagrama de pacotes. Ele descreve os pacotes do sistema mostrando as dependências entre eles e o agrupamento de suas classes.

Por último, um diagrama que abrangesse o sistema como um todo, que represente o aplicativo cliente, o BD e outros componentes do sistema foi necessário. O diagrama de componentes tem exatamente esse propósito. Este diagrama mostra o relacionamento entre diferentes componentes de *software* do sistema.

O principal objetivo e benefício da aplicação dessas modelagens é trazer eficiência no processo de desenvolvimento do sistema. Como toda análise foi feita anteriormente, essa eficiência é consequência da facilidade de visualização do sistema, documentação das decisões tomadas e ajuda a entender um sistema complexo dividindo-o em pequenas partes bem estruturadas.

3.3 Arquitetura limpa

A arquitetura limpa, ou comumente conhecida como *Clean Architecture*, foi criada por Martin (2017) em seu livro *Clean Architecture: A Craftsman's Guide to Software Structure* e ela fornece uma metodologia de desenvolvimento que facilita o desenvolvimento de código, permite melhor atualização, manutenção e menos dependências entre os componentes do *software*.

Um dos objetivos da arquitetura limpa é o princípio de *design* de código conhecido como *Separation of Concerns* (SoC). Esse objetivo é atingido a partir da separação do *software* em camadas com diferentes responsabilidades.

Algumas das vantagens dessa separação por camadas são:

- Testabilidade: As regras de negócio desenvolvidas no sistema poder ser testadas isoladamente, ou seja, sem *User Interfaces* (UI), bancos de dados ou agentes externos;
- Independente de UI: As interfaces podem ser modificadas frequentemente sem haver a necessidade de modificar o resto do *software*;
- Independente de BD: O sistema não está preso a nenhum tipo de BD específico e pode ser trocado sem afetar as regras de negócio;
- Independente de agentes externos: As regras de negócio não conhecem nada externo ao sistema;

Cada círculo da Figura 3.2 representa diferentes áreas do sistema. Essas áreas só podem depender de outras mais internas, ou seja, a dependência só pode ocorrer "para dentro"do diagrama. A partir disso, nenhuma área conhece outras que sejam externas à ela.

Essa regra de dependência, representada pelas setas da Figura 3.2, garante que o sistema seja totalmente testável e poupa o desenvolvedor de problemas futuros com manutenções. Como o sistema é independente de camadas externas, como por exemplo *frameworks* e bancos

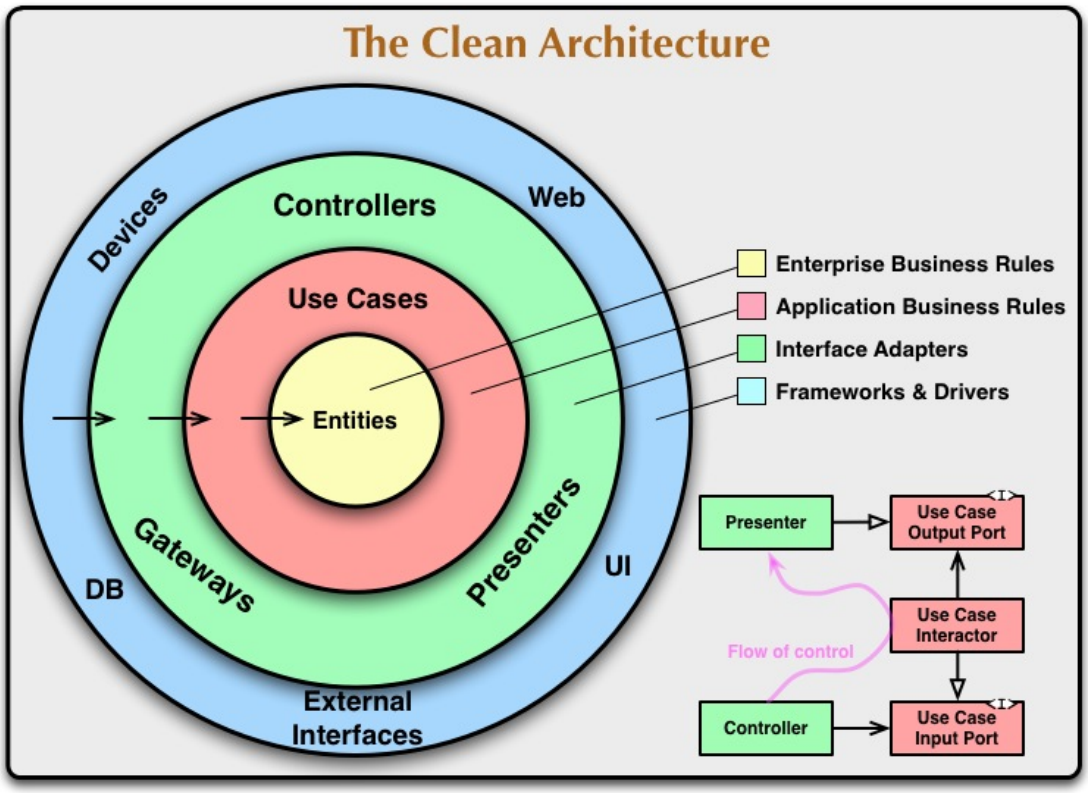

Figura 3.2: Arquitetura limpa. Fonte: (MARTIN, 2012)

de dados, no momento que for necessário efetuar a mudança destes, o processo será mais simples e garantirá a integridade das regras de negócio.

Segundo Martin (2012) e levando em consideração a legenda da Figura 3.2, cada camada possui as seguintes definições:

- *Enterprise Business Rules*: são os objetos de domínio da aplicação que encapsulam as regras gerais e de alto nível. Eles aplicam lógicas gerais para toda a entidade;
- *Application Business Rules*: são as ações do negócio, ou seja, contém as regras de negócio dos casos de uso do sistema. Esses casos de uso orquestram o fluxo de dados que são enviados e recebidos das entidades, direcionando-as para atingir um objetivo específico do caso de uso;
- *Interface Adapters*: Essa camada é responsável por transformar as estruturas de dados dos casos de uso para o formato adequado aos agentes externos, como bancos de dados ou *frameworks*, ou o contrário, convertendo os dados vindos dos agentes externos para a estrutura esperada pelos casos de uso;

- *Frameworks & Drivers*: Essa camada é composta por *frameworks* ou ferramentas, como o BD. Nela é feita a ligação entre os agentes externos com a próxima camada mais interna ao círculo;

A arquitetura limpa não implica regras de apenas utilizar essas quatro camadas citadas anteriormente, isso significa que existem casos em que mais camadas podem ser utilizadas. Entretanto, a regra de dependência sempre deve ser aplicada mesmo nas novas camadas, e quanto mais interna ela for, mais abstrata ela deve se tornar.

A regra de dependência é clara, mas existem situações em que uma camada interna precisa chamar uma externa à ela. Por exemplo, o canto inferior direito da Figura 3.2 mostra os *controllers* e *presenters* se comunicando através da camada de caso de uso. Analisando o fluxo, ele começa no *controller*, passa pelos casos de uso e termina no *presenter*.

Essa violação da regra de dependência, onde o caso de uso invoca o *presenter*, é resolvida através do princípio de inversão de dependências. Para isso, abstrações são utilizadas para que as dependências do código-fonte se oponham ao fluxo apenas nas fronteiras das camadas.

Considere o exemplo da necessidade do caso de uso chamar o *presenter*. O caso de uso não deve chamar o *presenter* diretamente, porque nesse caso ele violaria a regra de dependência. Então o caso de uso chama uma interface em sua camada, e as implementações dos métodos expostos pela interface são feitas pela camada *presenter*.

Com isso, através do polimorfismo são criadas dependências no código que se opõem ao fluxo de controle para que o código esteja em conformidade com a regra de dependência, independente da direção que o fluxo de controle está indo.

Construindo a aplicação a partir dessas regras e princípios fazem com que o sistema seja testável, de fácil manutenção, já que a troca de *frameworks* e bancos de dados podem ser realizadas sem impactar as camadas internas, criações de novas funcionalidades e refatorações podem ser feitas com mais facilidade e segurança, porque o código estará coberto de testes e a separação por camadas diminui o acoplamento, que minimiza o impacto de cada modificação.

3.4 Padrão de Repositório

O padrão de repositório é um *framework* conceitual responsável por encapsular um conjunto de objetos persistidos em um armazenamento de dados e as operações realizadas sobre eles,

provendo uma visão orientada à objetos da camada de persistência da aplicação (FOWLER, 2002).

Segundo Evans (2003), alguns dos benefícios da utilização desse padrão são:

- Interface simples para obter e gerenciar o ciclo de vida de objetos persistidos;
- Desacopla a camada de domínio da aplicação da camada de persistência de dados;
- Permite fácil substituição da implementação, facilitanto o desenvolvimento de testes;

O repositório será utilizado para atender a regra de dependência da arquitetura limpa, fazendo com que os casos de uso não dependam da implementação das consultas feitas aos agentes externos, que estão presentes em camadas mais exteriores.

3.5 *Frameworks* híbridos

Um dos principais requisitos do projeto, abordados com mais detalhes na Seção 6.1, é que o aplicativo deve funcionar em diversos tipos de sistemas operacionais de dispositivos móveis. Os dois principais do mercado são o *Android* e o *iOS*, que representam a grande maioria dos dispositivos.

Com isso, as linguagens de programação nativas de cada sistema foram excluídas, e os dois principais *frameworks* híbridos utilizados no mercado são o *Flutter* e o *React*.

O *framework* utilizado neste projeto é o *Flutter* e essa decisão foi tomada por alguns fatores. O principal deles foi o fato dele ter sido lecionado durante uma disciplina cursada na graduação, outro fator foi em relação a existência de uma grande comunidade ativa e receptiva, que como consequência disso já produziu muitas documentações e tutoriais envolvendo a solução, e por último, a curva de aprendizado, que por conta da experiência prévia e da grande quantidade de materiais disponíveis na internet, se torna menor quando comparada ao *React*.

3.6 Princípios SOLID

Os princípios SOLID serão utilizados no desenvolvimento do aplicativo por conta dos grandes benefícios que sua aplicação traz ao desenvolvimento de software e em relação a qualidade do código resultante dessa aplicação. SOLID é um acrônimo mnemônico dos 5

princípios de programação orientada a objetos introduzidos por Martin (2000) em seu artigo *Design Principles and Design Patterns*, onde cada letra representa um desses princípios.

Em seu artigo, Robert afirma que o software está em constante mudança e evolução, e conforme essa mudança acontece, a complexidade aumenta cada vez mais. Por conta disso, sem bons princípios de design de código, o software acaba se tornando de difícil manutenibilidade, testabilidade, legibilidade e extensibilidade. Os princípios SOLID foram desenvolvidos para combater esses problemas.

O objetivo geral desses princípios é reduzir as dependências para que diferentes áreas possam evoluir independentemente sem impactar umas às outras. Além disso, também possuem o propósito de construir designs fáceis de serem entendidos, mantidos, estendidos, escalados, reutilizados e testados. Por fim, a adoção dessas práticas evitam que problemas comuns no processo de desenvolvimento sejam enfrentados e possibilita a construção de software ágil, adaptável e eficiente.

Cada uma das letras do acrônimo representa um princípio, e são utilizados por muitos sistemas desenvolvidos atualmente.

3.6.1 Princípio da responsabilidade única

A letra “S” do acrônimo representa o *Single Responsibility Principle*, esse princípio tem relação direta com a coesão. Isso quer dizer que uma classe só terá uma única responsabilidade bem definida.

Em seu artigo, Robert descreve esse princípio dizendo que uma classe deve ter apenas um motivo para ser modificada; o que resulta na alta coesão citada no parágrafo anterior.

3.6.2 Princípio aberto/fechado

A letra “O” representa o *Open/Closed Principle*, esse princípio foi descrito por Robert da seguinte forma:

> "Devemos escrever nossos módulos de forma que possam ser estendidos, sem exigir que sejam modificados. Em outras palavras, queremos ser capazes de mudar o que os módulos fazem, sem alterar o código-fonte dos módulos."

Ou seja, Robert está se referindo ao conceito abstrato da extensão. Usar heranças ou interfaces que permitem polimorfismo é uma das maneiras mais comuns de cumprir esse princípio.

3.6.3 Princípio substituição de Liskov

A letra "L" do acrônimo significa *Liskov Substitution Principle*, esse princípio prega que uma classe derivada deve ser substituível por sua classe base; essa frase é a simplificação da definição científica que foi apresentada por Liskov e Wing (1999) no artigo *Behavioral Subtyping Using Invariants and Constraints*.

A definição traduzida apresentada no artigo científico é:

> "Se $\theta(x)$ é uma propriedade provável dos objetos x do tipo T. Então $\theta(y)$ deve ser verdadeiro para objetos y do tipo S onde S é um subtipo de T."

3.6.4 Princípio segregação de interfaces

A letra "I"representa o princípio *Interface Segregation Principle*, em que Robert diz que muitas interfaces específicas para o cliente são melhores do que uma grande interface de propósito genérico.

Para os desenvolvedores, isso significa que novos métodos ou funcionalidades não devem ser criados a partir de uma interface existente, ao invés disso, é recomendado que se crie uma nova interface e as classes poderão implementar múltiplas interfaces conforme for necessário.

3.6.5 Princípio da inversão de dependências

A última letra do acrônimo, a letra "D", representa o princípio *Dependency Inversion Principle*, ele oferece uma maneira de desacoplar módulos do *software*.

Robert explica esse princípio dizendo que módulos de alto nível não devem depender dos de baixo nível, ambos devem depender de abstrações. Além disso, as abstrações não devem depender dos detalhes, os detalhes que devem depender das abstrações.

3.7 Arquitetura orientada a eventos

Segundo a RedHat (2021), a arquitetura orientada a eventos é um modelo de *design* de sistemas que não depende de linguagens de programação ou *frameworks*, porque nele é abordado como modelar seu sistema de maneira agnóstica à linguagens.

Aplicando essa arquitetura, o acoplamento entre serviços no sistema é mínimo, ou seja, os diferentes componentes do sistema não terão altas dependências diretas entre si. Isso acontece porque os produtores dos eventos não conhecem os consumidores e o evento em si não conhece as consequências de sua ocorrência. Portanto, o principal benefício trazido pela arquitetura é que o sistema se torna flexível, se adapta a mudanças e toma decisões em tempo real.

A definição de um evento é uma mudança de estado no *software* ou *hardware*. Quando essa mudança acontece, uma mensagem é enviada por uma parte do sistema para avisar outra parte que alguma mudança ocorreu.

Com base nas definições apresentadas, a arquitetura orientada a eventos é composta por produtores e consumidores de eventos. O papel do produtor é detectar eventos e os representar como uma mensagem, que é enviada por meio de canais, que serão posteriormente processadas de maneira assíncrona pelo consumidor. O consumidor detectará novas mensagens publicadas no canal de comunicação e processará esse evento a partir das informações contidas nele.

Existem diferentes tipos de arquitetura orientada a eventos, o modelo utilizado neste trabalho será o *pub/sub*. *Pub/sub* é o nome utilizado para representar o modelo *Publish/Subscribe*, trata-se de uma infraestrutura de mensageria baseada em subscrições em um fluxo de eventos, ou seja, após ocorrer um evento, ele será publicado em um sistema de mensagens assíncronas, em que haverá um consumidor que receberá a publicação.

A maneira que o sistema desenvolvido neste trabalho utiliza o modelo *pub/sub* é explicado no Capítulo 6.

3.8 Banco de dados orientado a documentos

O banco de dados orientado a documentos é um tipo de BD não-relacional que armazena e consulta dados em formato JavaScript Object Notation (JSON), cuja principal característica é a organização de dados livres de esquemas. Por conta disso, esse tipo de banco facilita as consultas feitas pelos desenvolvedores, que usam o mesmo formato de modelo que usam no código do aplicativo.

Outras características desse tipo de BD *NoSQL* é a sua natureza semiestruturada, flexível e hierárquica dos documentos, que permite que o banco evolua conforme as necessidades do aplicativo.

3.9 Fluxo de trabalho com *Git*

Este projeto foi desenvolvido utilizando um sistema de controle de versão chamado *Git*. Ele foi escolhido por ser a ferramenta mais utilizada pela comunidade de desenvolvimento de *software*, gratuita e de código aberto. A utilização de um sistema de versionamento foi por conta dos benefícios citados abaixo.

- Registra todo o histórico de alterações dos arquivos. Isso significa que serão salvas todas as alterações de conteúdo, deleção ou adição dos arquivos, o autor, data e mensagens escritas em cada alteração.
- Facilidade na reversão de alterações, ou seja, caso uma nova funcionalidade não se comporte da maneira esperada e cause falhas no aplicativo, a reversão para alguma versão anterior é feita a partir do histórico registrado pelo versionamento.
- Suporte a ramificações e mesclas, que possibilitam a criação de diferentes linhas de desenvolvimento independente uma das outras, e posteriormente da mesclagem dessas linhas em uma ramificação principal, que conterá todo código testado e validado do projeto.

Apesar do *Git* ser uma ferramenta robusta para o desenvolvimento, existem alguns tipos de fluxos de trabalhos que são adotados para desenvolver um projeto. Os fluxos de trabalho procuram padronizar a maneira que os desenvolvedores colaboram dentro do projeto, a nomeação de ramificações, a responsabilidade que cada tipo possui e a maneira que novas funcionalidades são integradas ao código principal.

Por conta de sua robustez, padronização e organização elevada, o fluxo de trabalho utilizado neste projeto foi o *GitFlow*, criado por Driessen (2020). Ele é ideal para gerenciar projetos grandes, com ciclo de lançamento agendado e atribui responsabilidades específicas para cada tipo de ramificação.

Existem 5 tipos de ramificações nesse fluxo de trabalho, sendo elas a *main*, *develop*, *hotfix*, *release* e *feature*.

A *main* e a *develop* armazenam todo o histórico de alterações do desenvolvimento de um produto. Cada modificação na ramificação *main* representa uma nova versão lançada para os usuários, e a ramificação *develop* contém as iterações das funcionalidades que são desenvolvidas. Portanto, uma modificação na ramificação *main* representa um conjunto de funcionalidades que foram iteradas na ramificação *develop*.

A Figura 3.3 representa a evolução independente das duas ramificações, para melhor visualização desse diagrama e dos próximos, cada ramificação é representada por uma cor e a primeira letra de seu nome.

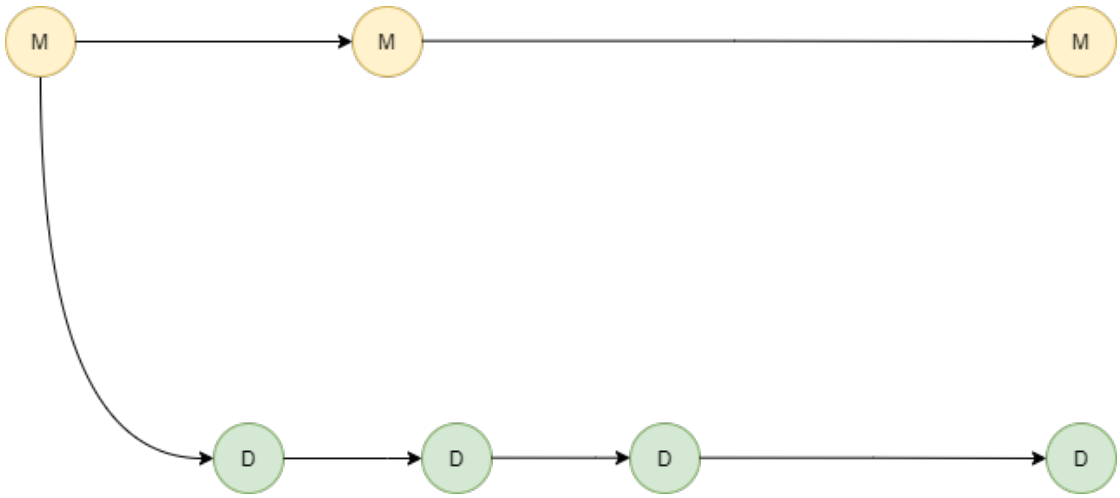

Figura 3.3: Representação das ramificações *main* e *develop*

Para o desenvolvimento de novas funcionalidades é utilizada a ramificação *feature*. Ela deve ser derivada da *develop* e mesclada logo após a finalização do desenvolvimento da funcionalidade, como mostra a Figura 3.4.

Depois de algumas iterações de funcionalidades na *develop*, é criado uma preparação para um lançamento de uma nova versão do produto. Essa preparação é feita pela ramificação *release*. Nela, nenhuma funcionalidade é desenvolvida, apenas pequenos ajustes em possíveis falhas ou documentações, como mostra a Figura 3.5.

É importante ressaltar que a *release* é derivada da *develop* e deve ser mesclada com a *main* e com a *develop* novamente, já que algumas atualizações no código podem ter sido feitas na ramificação *release*. Quando a mesclagem para a *main* for feita, uma nova versão do produto é lançada.

A última ramificação que esse fluxo de trabalho possui é a *hotfix*. Ela é responsável por fazer pequenas alterações diretamente na main, ou seja, eventuais falhas que foram entregues aos usuários são corrigidas nesta ramificação.

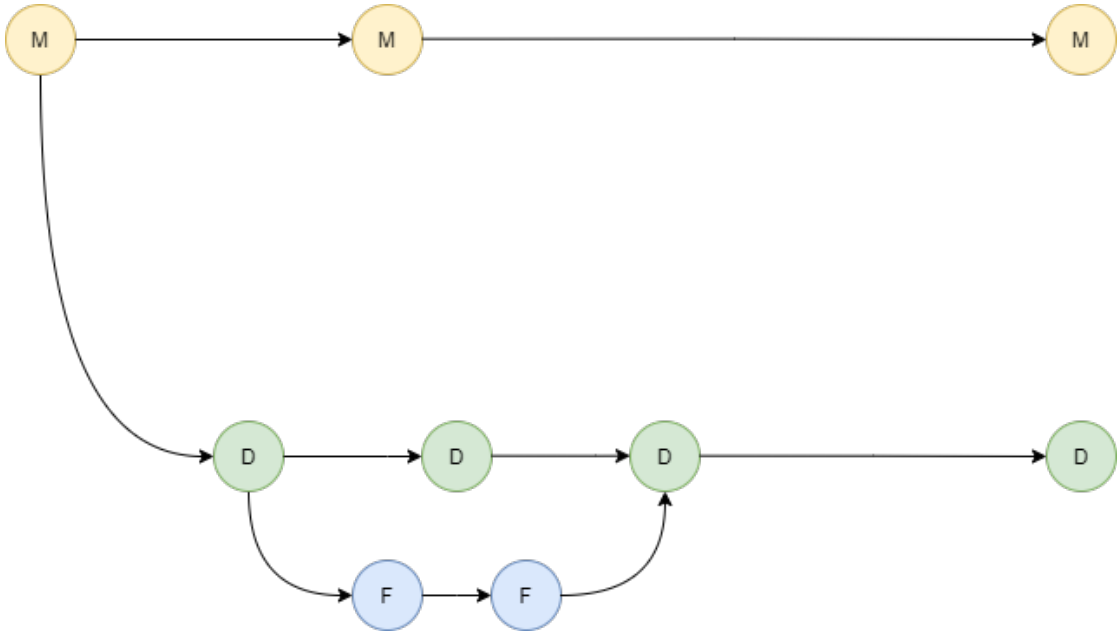

Figura 3.4: Representação das ramificações *main*, *develop* e *feature*

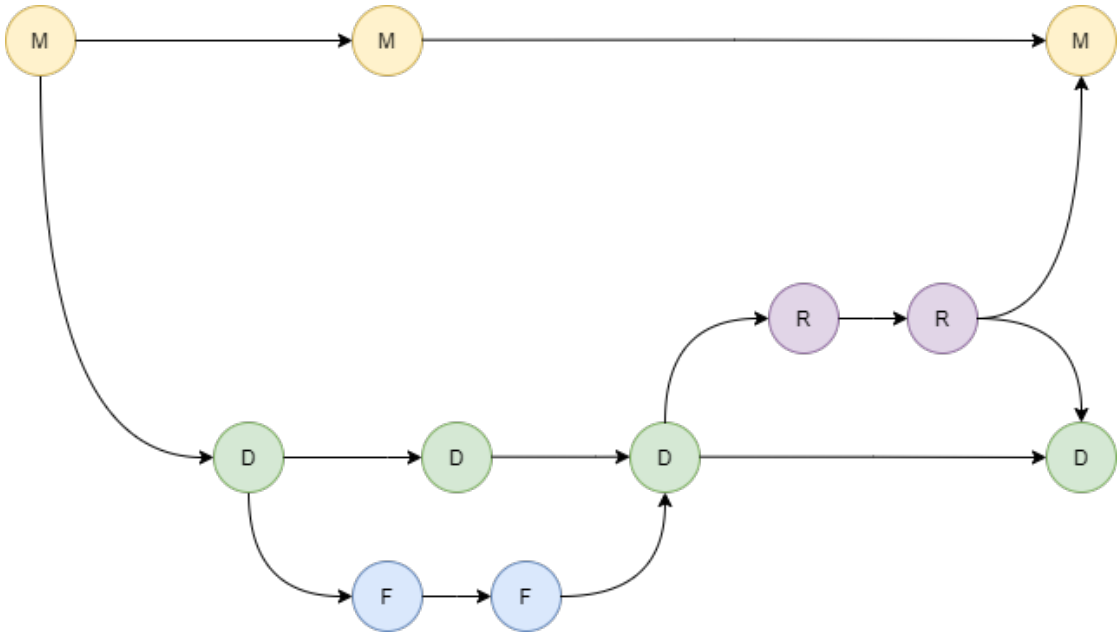

Figura 3.5: Representação das ramificações *main*, *develop*, *feature* e *release*

A Figura 3.6 mostra a *hotfix* atualizando alguma falha que estava na *main* e como todas as 5 ramificações estão presentes, consequentemente representa um ciclo completo utilizando o *GitFlow*.

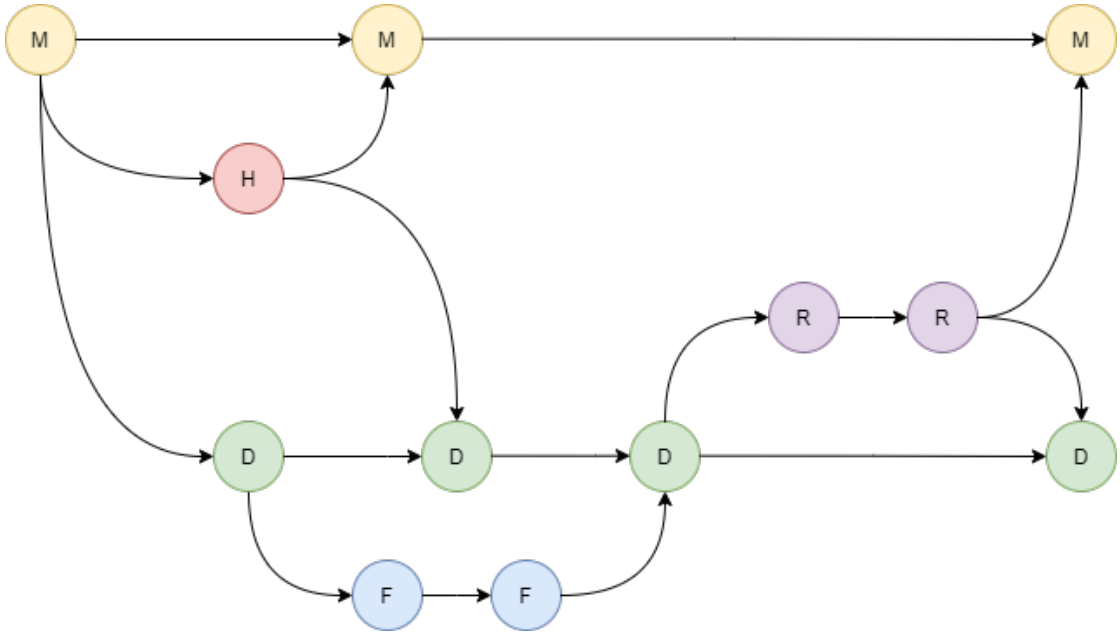

Figura 3.6: Representação de todas ramificações do *GitFlow*

3.10 *Pipelines* de integração e entrega contínua

Como já citado anteriormente, a qualidade do software é uma preocupação importante no processo de desenvolvimento deste projeto. Aliado a isso, a criação de pipelines trás benefícios ao sistema, alguns deles são:

- Maior taxa de entrega: em conjunto com o fluxo de trabalho adotado no desenvolvimento, a frequência com que novas funcionalidades ou correções de falhas são lançadas é maior. Como o processo de entrega é automatizado, o desenvolvedor foca no desenvolvimento de novas entregas, e não em passos repetitivos para efetuá-las;
- Confiabilidade de testes: os testes são efetuados de forma automática, e caso algum teste falhe, a entrega é impossibilitada de ser feita, garantindo que apenas pedaços de código validados possam ser entregues aos clientes;
- Redução de custos: menos tempo gasto em processos manuais e repetitivos, e mais tempo desenvolvendo novas funcionalidades resulta em redução de custos, já que o desenvolvedor se preocupa somente com atividades que agregam valor;

Os *pipelines* de *Continuous Integration and Continuous Delivery* (CI/CD) são um conjunto de etapas que integram com o código principal, efetuando principalmente testes nesse novo código, e o entregam, que seriam novas versões liberadas aos clientes.

Existem diversas ferramentas de CI/CD, e a ferramenta escolhida foi o *GitHub Actions*, porque possui integração perfeita com a central de repositórios *git* que será utilizada, o *GitHub*. Além disso, a ferramenta é gratuita, suporta as principais funcionalidades que um *pipeline* precisa e possui boa documentação.

3.11 Serviços computacionais na nuvem

Durante o processo de desenvolvimento, surgiu a necessidade do sistema possuir serviços remotos na nuvem, tais como BD e um servidor *backend*, onde rotinas pudessem rodar independentemente do dispositivo móvel do usuário.

Essa necessidade trouxe o requisito de se utilizar serviços computacionais na nuvem. Levando em consideração as tecnologias e os requisitos do aplicativo, o provedor utilizado foi o *Firebase*.

Esse provedor de nuvem é especializado em acelerar o processo de desenvolvimento de sistemas *mobile*, fornecendo ferramentas de fácil configuração, auto gerenciadas e de alta qualidade.

Os serviços que foram utilizados do provedor foi o BD orientado a documentos, o servidor *backend* para executar as rotinas de rastreamento de contatos, que são ativadas a partir dos eventos de escrita ocorridos em coleções específicas no BD.

Além da alta disponibilidade e confiabilidade dos serviços oferecidos pelo *Firebase*, esse conjunto de tecnologias é uma escolha comum para aplicações móveis. A comunidade de desenvolvimento possui diversos tutoriais e documentações que abrangem os mais diversos casos de uso que podem ser utilizados em conjunto, o que facilitou o processo de desenvolvimento.

Cada um dos recursos utilizados da plataforma são explicados no Capítulo 5, e a forma de como foram utilizados está descrita no Capítulo 6.

Capítulo 4

Trabalhos correlatos

Este capítulo se trata das pesquisas que foram realizadas procurando soluções no mercado que se propõem a resolver o problema apresentado pelo projeto e também consultas por algumas palavras-chave no BD do Instituto Nacional da Propriedade Industrial (INPI).

4.1 Busca por patentes de software registradas

A consulta foi feita utilizando a categoria de Programas de Computador, buscando os que tinham como parte de seu título as palavras-chave rastreamento de doenças, exposição a doenças, aplicativo de rastreamento ou rastreamento de contatos.

Levando em consideração a ordem de busca apresentada, apenas a terceira encontrou algum resultado, totalizando 3 respostas. Uma delas é um aplicativo de celular e as outras duas são aplicativos de análise de mercado da plataforma *NinjaTrader8*.

Analisando esse aplicativo móvel, ele foi registrado por Rodrigues (2021) no dia 27 de outubro de 2020 e se chama "MOOPE SISTEMA DE RASTREAMENTO E GESTÃO DE FROTAS - Módulo Aplicativo ios/Android, Módulo Frotas, Modulo Central Comandos, Modulo Central Notificações, Modulo MeusMoopes", que tem como campo de aplicação a gestão de frotas. Portanto, o escopo deste aplicativo difere do proposto por este trabalho.

4.2 Busca nas lojas de aplicativos

No começo da pandemia do coronavírus, as empresas Google e Apple (2020), detentoras dos dois principais sistemas operacionais para celulares - o *iOS* e *Android* - divulgaram o

desenvolvimento de uma *Application Programming Interface* (API) para monitorar a exposição das pessoas ao coronavírus. Esse monitoramento é feito a partir da captação do sinal entre pessoas próximas utilizando a tecnologia *Bluetooth*. Essa API identifica se pessoas permaneceram próximas durante alguns minutos e se o sinal entre elas for suficientemente forte, cada dispositivo salva o identificador do outro que tiveram contato, assim seria possível fazer o rastreamento de contatos.

A API só pode ser consumida pelo governo dos países que se inscrevem no programa, ou seja, apenas os aplicativos desenvolvidos pelas autoridades públicas podem utilizar a tecnologia criada pelas duas empresas. Além disso, as empresas divulgaram que o recurso será desativado após a pandemia de coronavírus.

O único aplicativo que funciona no Brasil que possui a finalidade de rastreamento de contatos e que utiliza a API citada anteriomente foi desenvolvido pelo DATASUS (2020) e se chama "Coronavirus - SUS".

As principais funcionalidades deste aplicativo são que os usuários podem consultar as notícias oficiais compartilhadas pelo Ministério da Saúde, conferir as principais notícias falsas que estão em circulação, informações sobre a doença, o compartilhamento de teste positivo e o rastreamento de contatos.

Além deste aplicativo, foram encontrados alguns desenvolvidos por outros países, que são o *CDC*, *Care19*, *Crush Covid RI* e *FEMA*. Nenhum desses aplicativos funcionam corretamente em território brasileiro, a única funcionalidade que pode ser aproveitada são as informações sobre a doença, que estão escritas em inglês.

Com isso, o aplicativo que possui o objetivo de rastreamento de contatos e que funciona corretamente no Brasil é o Coronavírus - SUS. É importante reafirmar que a API que ele utiliza será desativada após a pandemia de coronavírus.

Capítulo 5

Infraestrutura

Para a realização da solução proposta neste trabalho, recursos de infraestrutura são necessários para que seja possível a execução de rotinas de rastreamento de contatos no servidor, utilização do BD e da comunicação entre o cliente e o servidor. Assim, na Seção 5.1 será explicado o que é e para que serve um serviço de *backend*, quais recursos de infraestrutura serão utilizados e o que são os sistemas de autenticação e o envio de mensagens. Na Seção 5.2 será explicado sobre a solução de BD da plataforma *Firebase* e o que cada entidade dele significa. Por fim, na Seção 5.3 será explicado o que é a *Cloud Function* e qual será a responsabilidade dela neste projeto.

5.1 Serviço de *Backend*

A infraestrutura da aplicação será totalmente em nuvem e os principais motivos dessa decisão foram: as faturas de cobrança estão diretamente relacionadas à carga de utilização, e como o escopo deste trabalho é um teste de conclusão de curso, essa carga será baixa; totalmente gerenciado por grandes empresas; boa documentação e facilidade na criação de ambientes.

A nuvem que será utilizada neste trabalho é a *Google Cloud Platform* (GCP) porque ela entrega serviços que atendem perfeitamente os requisitos do trabalho, por exemplo, APIs de busca de locais que utilizam o BD do *Google*.

Neste projeto será utilizado um *Backend-as-a-Service* (BaaS), que é um serviço de *backend* gerenciado por uma empresa. Como a provedora de nuvem escolhida nesse projeto é a GCP, o serviço de BaaS dela se chama *Firebase*.

Firebase é uma plataforma digital que tem como objetivo acelerar o processo de desenvolvimento de aplicativos e oferecer serviços que atendam diversos requisitos, como a

criação de BDs escaláveis, sistemas de autenticação, sistemas de envio de notificações, entre outros serviços.

O *Firebase* está diretamente relacionado com a GCP. Todo projeto criado nele também está representado na GCP, que é a base dos serviços de computação em nuvem da *Google*. A diferença é que o *Firebase* abstrai grande parte da configuração da infraestrutura, gerenciando as configurações necessárias para que os serviços funcionem automaticamente, sem que o usuário precise entender sobre cada detalhe que acontece em segundo plano.

Alguns serviços existem nas duas plataformas, e no fundo são os mesmos, apenas com interfaces de consumo diferentes. Por exemplo, o serviço de *Cloud Functions* pode ser utilizado em ambas plataformas, e quando criado no *Firebase*, também é possível visualizá-lo na interface da GCP.

Porém algumas configurações não são possíveis de serem feitas diretamente pelo *Firebase*. Um exemplo disso é a gestão de identidades, que é responsável por definir quais permissões cada usuário tem dentro da plataforma e só pode ser configurada pela GCP.

A principal vantagem competitiva do *Firebase* quando comparado a outras plataformas é a alta produtividade que ele traz ao desenvolvimento de aplicativos.

Os serviços do *Firebase* que serão utilizados neste trabalho são o sistema de autenticação, o envio de notificações, o BD e as *Cloud Functions*.

Começando pela autenticação, ela é a peça chave para que os locais visitados pelos usuários sejam salvos em seus respectivos documentos no BD, dando uma experiência personalizada no aplicativo. Essa autenticação pode ser feita de diversas formas diferentes, algumas delas são o *login* utilizando *e-mail* e senha, número de telefone celular, redes sociais e autenticação anônima.

A autenticação anônima, que é o único método de autenticação utilizado pelo aplicativo e explicado com mais detalhes no Capítulo 6, é criado pelo próprio *Firebase*. Ela é necessária somente para que o aplicativo diferencie os usuários para efetuar o rastreamento, sem precisar de informações pessoais do mesmo.

Outro serviço que será utilizado é o *Firebase Cloud Messaging* (FCM), que é responsável pelo envio de notificações aos usuários que forem possivelmente expostos à alguma doença infecciosa.

O FCM é bastante flexível e possui algumas funcionalidades que atendem diferentes casos de uso. A principal funcionalidade são as formas que a notificação pode ser enviada para o

cliente, que são para dispositivos únicos, grupos de dispositivos ou para os dispositivos inscritos em tópicos.

O envio de notificações deve ser feito de forma automática, para isso, o *Firestore* e as *Cloud Functions* trabalharão em conjunto para que a implementação da lógica do envio de mensagens seja feita com sucesso.

5.2 *Firestore*

O Firestore (2021a) é um BD não relacional orientada a documentos oferecido pela *Google*. As principais características dele são:

- *Serverless*: sem servidor, totalmente gerenciado por tecnologias da Google e com escalonamento automático para atender qualquer carga de requisições;
- Mecanismo de consulta avançado: permite a execução de transações com Atomicidade, Consistência, Isolamento e Durabilidade (ACID) nos dados dos documentos armazenados;
- Segurança: integração com a autenticação do *Firebase* para possibilitar controles de acesso de segurança com base na identidade;
- Replicação multirregional: redundância de BD em diversos *data centers* espalhados pelo mundo com alta consistência, aumentando a disponibilidade do serviço, mesmo em caso de desastres;
- Flexibilidade: o modelo de dados disponibiliza a criação de estruturas hierárquicas flexíveis, onde os dados são armazenados em documentos, e estes organizados em coleções;
- Integração: o *Firestore* possui integração perfeita com outros produtos do *Firebase* ou da GCP.

O modelo de dados do Firestore (2021b) segue o formato JSON e é estruturado em entidades denominadas documentos e coleções. A Figura 5.1 demonstra um exemplo dessa estrutura.

Documento é a unidade de armazenamento do *Firestore*, com campos que são mapeados para valores. Esses valores podem ser de diversos tipos: números inteiros, booleanos, *timestamps*, *strings* e até estruturas de dados como listas e mapas.

Figura 5.1: Exemplo de estrutura JSON com anotações referentes ao *Firestore*.

Cada documento é identificado por um nome e não podem possuir outros documentos criados diretamente dentro dele.

Como o *Firestore* não possui nenhum esquema, o desenvolvedor tem total liberdade sobre quais campos colocar em cada documento e quais tipos de dados esses campos armazenam.

Utilizando os mesmos valores e estrutura da Figura 5.1, a Figura 5.2 representa um exemplo de documento do *Firestore*.

As coleções são recipientes que armazenam um conjunto de documentos e são referenciadas pelo seu nome, da mesma forma que os documentos.

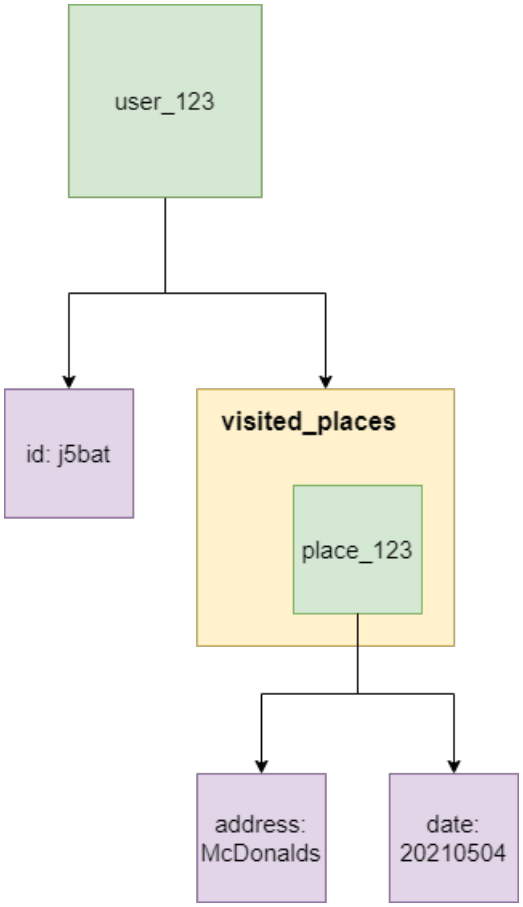

Figura 5.2: Exemplo de documento do *Firestore*.

Os documentos dentro das coleções devem ter identificadores únicos, ou seja, dois documentos diferentes não podem possuir o mesmo nome dentro da mesma coleção. Por conta disso, é comum que os documentos sejam nomeados com o Identificador (ID) do objeto que ele representa.

É importante ressaltar que coleções não podem armazenar dados diretamente, apenas através de documentos. Seguindo o mesmo raciocínio, os documentos não podem armazenar outros documentos. Cada entidade possui responsabilidades específicas. Por conta disso, para criar uma estrutura hierárquica no *Firestore*, as coleções armazenam exclusivamente documentos, e os documentos, além dos campos chaves-valor, podem armazenar subcoleções - que possuem as mesmas características de uma coleção.

Para melhor visualização da estrutura hierárquica do *Firestore*, a estrutura da Figura 5.1 está representada na Figura 5.3.

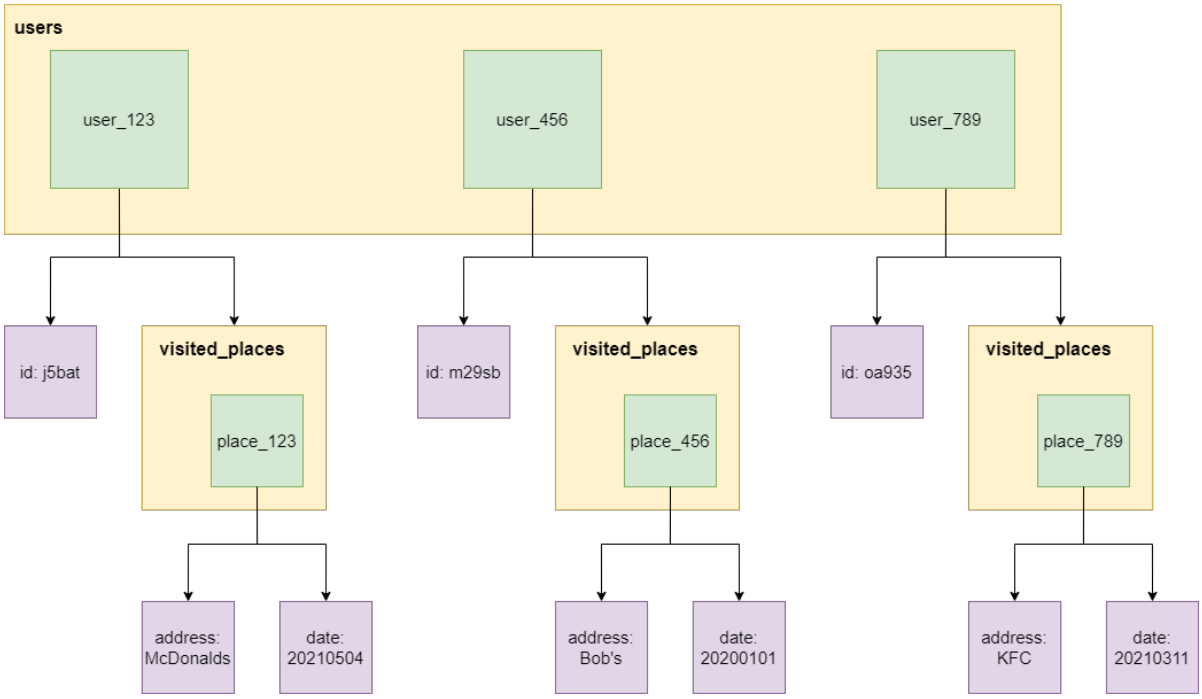

Figura 5.3: Exemplo de estrutura hierárquica do *Firestore*

5.3 *Cloud Functions*

A *Cloud Function* é o produto de função como serviço para criar aplicações com base em eventos, disponibilizado pelas plataformas *Firebase* e GCP. Ela é uma maneira de ampliar o comportamento de um aplicativo e integrar com outros recursos disponíveis na plataforma por meio da adição de código no servidor. Com base nisso, ela serve como camada conectiva, disponibilizando a construção de lógicas entre os serviços da plataforma por meio da detecção e da resposta a eventos.

No caso de uso deste trabalho, a *Cloud Function* consome os eventos de escrita em coleções específicas no *Firestore*, tratando-os através da execução do trecho de código implementado na função.

Capítulo 6

Desenvolvimento

Para realização da solução proposta nesse trabalho, é necessário o desenvolvimento de um aplicativo móvel para que o rastreamento de doenças infecciosas ocorra de forma automática. Assim, na Seção 6.1 são apresentados os requisitos funcionais e não funcionais do sistema, junto com a análise de cada um deles. A Seção 6.2 tem o propósito de apresentar todo o processo de modelagem do sistema, discutindo sobre o racional por trás de cada modelagem e apresentando os diagramas construídos. Na Seção 6.3 são apresentadas as interfaces construídas para o aplicativo. A Seção 6.4 contém a implementação das interfaces do usuário, modelagens e infraestrutura do sistema, utilizando o fluxo de trabalho *GitFlow*.

6.1 Levantamento e Análise de Requisitos

O aplicativo deve automatizar todo o ciclo do rastreamento, desde a captura da exposição à uma doença, até a notificação feita à pessoa possivelmente exposta. Com isso, a partir do problema de rastreamento de contatos apresentado no Capítulo 1, o processo de análise de requisitos foi feito pensando nos possíveis casos de uso do usuário.

Começando pelos requisitos não funcionais, a eficácia da solução depende diretamente do número de usuários que utilizarem o aplicativo. Isso significa que quanto mais tipos de dispotivos móveis forem suportados, mais potenciais usuários o aplicativo pode alcançar. Por conta disso, o aplicativo pode ser desenvolvido especialmente para cada tipo de Sistema Operacional (SO) ou utilizando um *framework* híbrido, em que a partir da mesma base de código é possível construir os arquivos para cada SO.

Levando em consideração o tempo de desenvolvimento do aplicativo e o número de pessoas envolvidas, a utilização de um *framework* híbrido é a melhor forma para atingir o maior número de usuários.

Além disso, como o aplicativo armazena os dados de localizações dos usuários, ele deve se preocupar em garantir a segurança dos mesmos. Essa segurança será garantida a partir da utilização de criptografia. Ela será aplicada aos dados em trânsito, ou seja, durante a comunicação entre cliente e servidor.

Ainda relacionado à segurança, a fim de preservar a privacidade das pessoas que utilizarem o aplicativo, a aplicação ou os usuários não devem ser capazes de descobrir quem possivelmente expôs outras pessoas à alguma doença. Isso significa que quando um usuário receber uma notificação de exposição, ele não será capaz de identificar quem foi, nem o local que isso ocorreu. Para isso, o aplicativo não terá a inserção de nenhum dado pessoal, ou seja, a autenticação será feita de forma anônima, onde cada usuário será representado por uma cadeia de caracteres aleatórios.

Por último, toda localização terá prazo de validade de 14 dias. Isso significa que todo local visitado pelo usuário será armazenado apenas durante esse prazo. Como a solução do problema é a automação do rastreamento de contatos, localizações mais antigas do que esses dias não são necessárias, já que o período de contágio e transmissão do vírus haveria terminado.

A partir do que foi explicado nos parágrafos anteriores, a Tabela 6.1 elenca os três requisitos não funcionais e suas respectivas prioridades, que podem ser obrigatórias, importantes ou opcionais.

Tabela 6.1: Requisitos não funcionais

Código	**Nome**	**Prioridade**
RNF01	Compatibilidade multiplataforma	Opcional
RNF02	Garantia de privacidade e segurança dos dados	Obrigatório
RNF03	Exclusão de localizações antigas	Importante

Como o aplicativo usará um *framework* multiplataforma para atender os requisitos não funcionais citados anteriormente, essa escolha trás um malefício: recursos nativos de cada dispositivo, como por exemplo o *Bluetooth*, não possuem o mesmo suporte caso fosse uma codificação utilizando uma linguagem nativa.

Levando isso em consideração, os contatos que eventualmente ocorrerem entre diferentes usuários do aplicativo podem ser rastreados a partir de duas principais formas:

- Armazenamento dos locais visitados pelos usuários utilizando um sistema de *check-in*, onde o usuário fará a inserção do local visitado utilizando um sistema de busca;
- Armazenamento dos contatos que foram efetuados utilizando o *Bluetooth*. O dispositivo detectará os sinais transmitidos ao seu redor, e fará a persistência desses contatos;

Ambas soluções solucionam o problema de formas distintas, porém, levando em consideração o requisito de compatibilidade multiplataforma e do baixo acesso à recursos nativos pelos *frameworks*, a solução utilizada neste trabalho será a partir do sistema de *check-in*.

Com isso, os usuários necessitam de um buscador para encontrar os locais que foram visitados. Esse buscador deve estar relacionado a um extenso BD para que a maioria dos locais inseridos pelos usuários sejam encontrados. Por conta disso, um BD externo deve ser utilizado, já que a criação de um novo não traria a experiência esperada para o usuário.

Depois que a busca foi feita, o usuário deve ser capaz de salvar o local encontrado. O salvamento deve ser feito em um BD remoto, para que a rotina de checagem do rastreamento de contatos seja feita no servidor. Com isso, o dispositivo móvel do usuário não precisará de conectividade com a *Internet*, nem que o aplicativo rode em segundo plano durante a execução da rotina, evitando o consumo desnecessário de dados e de bateria do dispositivo.

Se um usuário for infectado por uma doença, ele deve ser capaz de declarar no aplicativo que está infectado. A partir disso, a rotina de rastreamento de contatos deve ser capaz de relacionar os locais do usuário infectado com os usuários comuns, procurando por algum possível contato que possa ter ocorrido.

O usuário deve receber notificações caso o mesmo tenha tido algum possível contato com outras pessoas que utilizam o aplicativo e estavam infectadas. Essa notificação deve ser enviada de forma automática e sem a necessidade de intervenção humana, para que o problema seja resolvido de forma automática de ponta a ponta.

Para melhor usabilidade, é importante que o usuário consiga visualizar o local que está sendo buscado para confirmar que este é o local desejado para efetuar o *check-in*. Por isso, quando o usuário efetuar a busca de uma localização, a visualização dos detalhes, tais como nome, endereço e fotos, devem estar disponíveis.

Por último, o usuário deve ser capaz de conseguir visualizar todos os locais e as respectivas datas de visita que foram salvas no aplicativo. Essa funcionalidade não afeta no rastreamento de contatos, mas ajuda na experiência de utilização do aplicativo.

A partir de toda análise acima, foram levantados seis requisitos funcionais que estão representados na Tabela 6.2.

Tabela 6.2: Requisitos funcionais

Código	**Nome**	**Prioridade**
RF01	Buscar localizações	Obrigatório
RF02	Salvar localizações	Obrigatório
RF03	Declarar caso de infecção confirmado	Obrigatório
RF04	Enviar notificação de exposição	Obrigatório
RF05	Visualizar detalhes da localização	Importante
RF06	Listar locais salvos	Opcional

6.2 Modelagem

O processo de modelagem foi feito desde os aspectos mais abstratos do sistema, utilizando os diagramas de componentes e de pacotes, até os mais específicos, utilizando os diagrama de sequência e de atividades.

Para melhor apresentação e discussão de cada uma das modelagens, elas estão dividas nas subseções abaixo.

6.2.1 Modelagem dos componentes do sistema

A visão geral do sistema desenvolvido é representado pelo diagrama de componentes. Nele são contidos os componentes macros do sistema, como pode ser visto na Figura 6.1.

Cada um dos componentes do diagrama possui uma responsabilidade bem definida. O aplicativo móvel representa a interface do usuário e algumas regras de negócio que foram implementadas nas camadas interiores da *Clean Architecture*.

O *Firebase Auth* é o serviço responsável pela autenticação dos usuários, que nesse caso é utilizada apenas a autenticação anônima, como foi explicado nos capítulos anteriores.

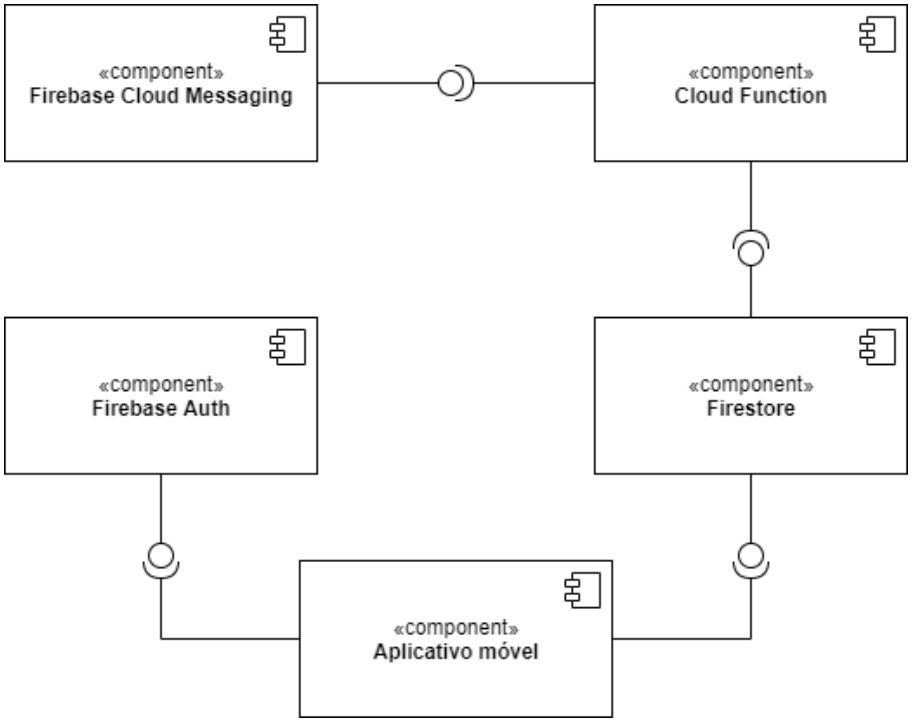

Figura 6.1: Diagrama de componentes.

O *Firestore* representa o BD do sistema, que armazenará as localizações visitadas por cada usuário e as que forem classificadas como infectadas.

A *Cloud Function* representa os *scripts* que fazem a análise do rastreamento de contatos e possuem a responsabilidade de ativar o sistema de notificações se algum encontro entre usuários ocorrer.

O *Firebase Cloud Messaging* é o serviço de envio de notificações.

Note que as conexões do diagrama mostram quais são os componentes que acessam as funcionalidades dos outros. Por exemplo, o aplicativo móvel não faz o envio de nenhuma notificação, a responsabilidade dessa tarefa são dos *scripts* das *Cloud Functions*.

6.2.2 Modelagem das funcionalidades do sistema

Cada funcionalidade do sistema foi herdada do levantamento de requisitos. Por conta disso, a partir do levantamento da Seção 6.1, foi criada a modelagem do diagrama de casos de uso, representado pela Figura 6.2. É importante ressaltar que este diagrama reflete a listagem de requisitos funcionais da Tabela 6.2.

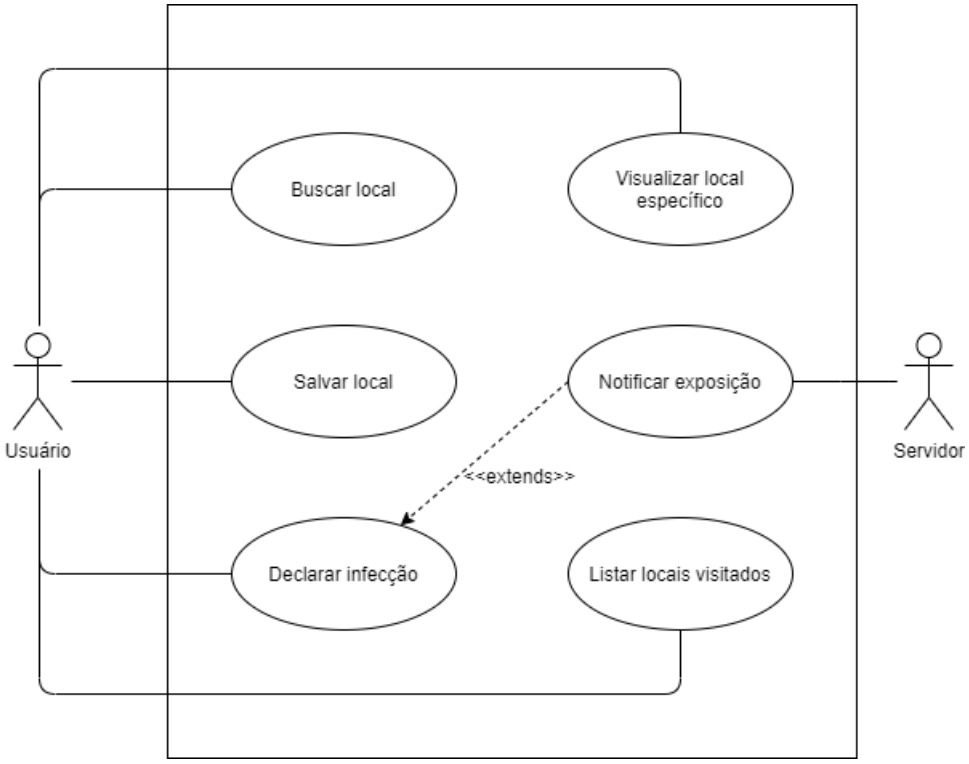

Figura 6.2: Diagrama de casos de uso.

Como cada caso de uso da Figura 6.2 é uma funcionalidade, os diagramas de atividades e sequência foram modelados para cada um deles. Começando pelo "Buscar local", temos os respectivos diagrama de atividades na Figura 6.3 e o de sequência na Figura 6.4.

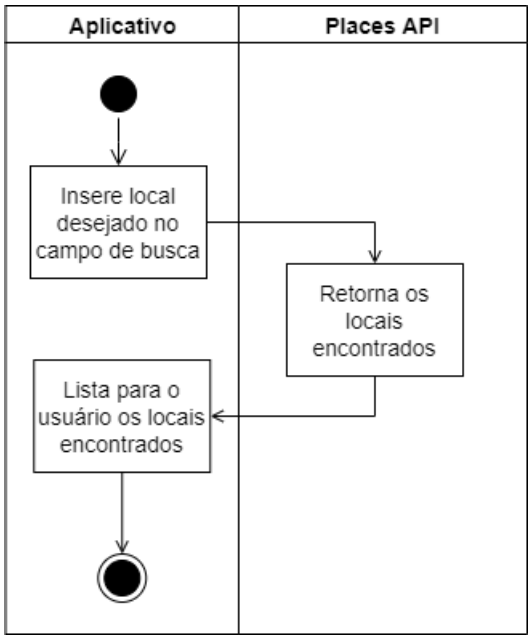

Figura 6.3: Diagrama de atividades da funcionalidade buscar local.

No diagrama de atividades da Figura 6.3 o usuário fará consultas em uma das APIs do *Google*, chamada *Places API*. Ela retornará qualquer localização no mundo que corresponda ao texto inserido pelo usuário.

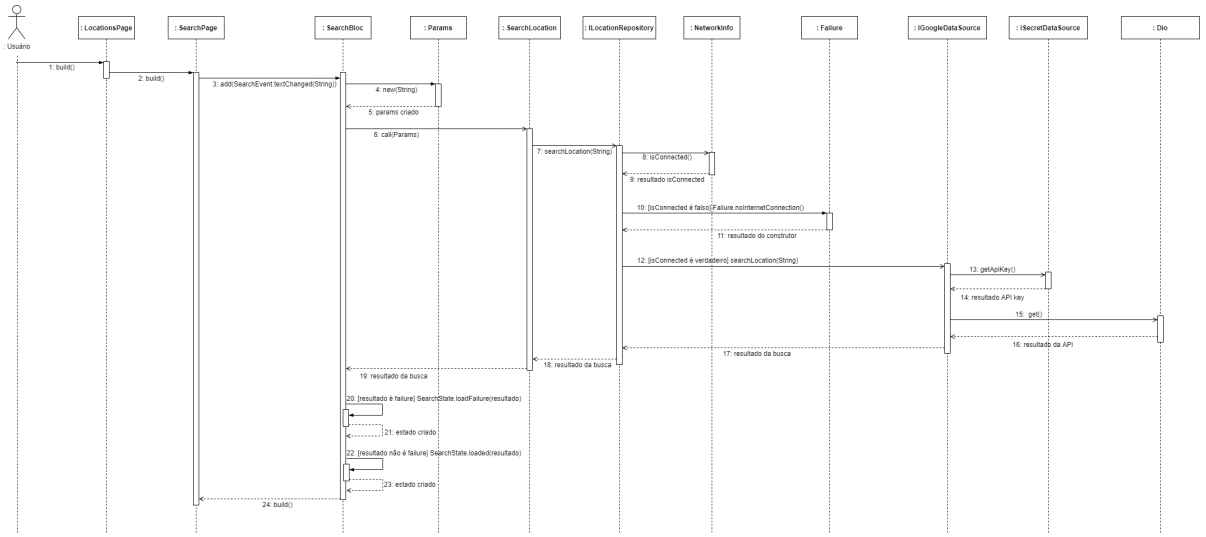

Figura 6.4: Diagrama de sequência da funcionalidade buscar local.

Para a funcionalidade "Salvar local", a Figura 6.5 representa o diagrama de atividades, e o de sequência na Figura 6.6.

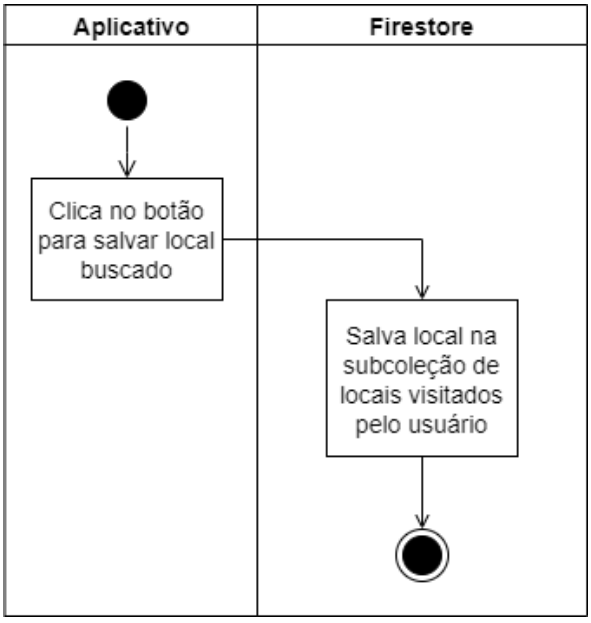

Figura 6.5: Diagrama de atividades da funcionalidade salvar local.

A funcionalidade "Declarar infecção"está representada pelo diagrama de atividades da Figura 6.7 e pelo de sequência da Figura 6.8

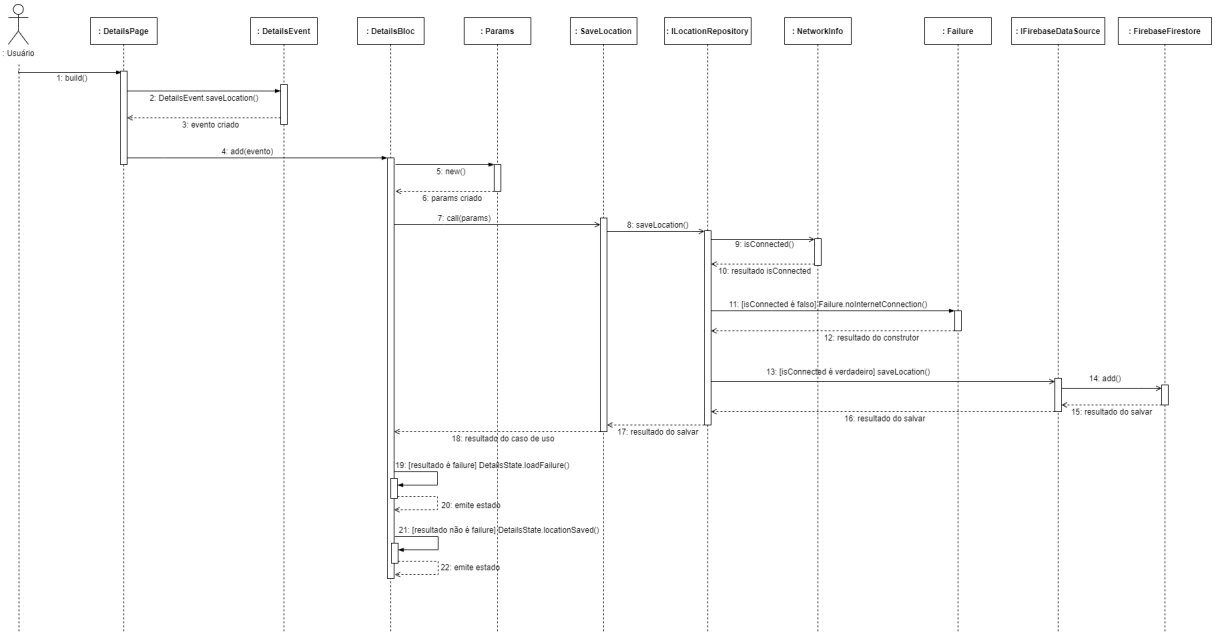

Figura 6.6: Diagrama de sequência da funcionalidade salvar local.

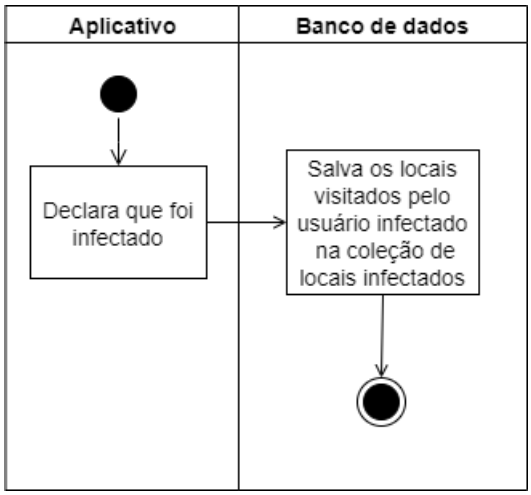

Figura 6.7: Diagrama de atividades da funcionalidade declarar infecção.

Para a modelagem da funcionalidade "Visualizar local específico"temos a Figura 6.9 para o diagrama de atividades e a Figura 6.10 para o de sequência.

A funcionalidade "Notificar exposição"não possui um diagrama de sequência. Como essa funcionalidade vai ser atendida através do desenvolvimento de *Cloud Functions*, a implementação não utiliza orientação a objetos, será um *script* de análise de encontro entre usuários e não seria bem representado por objetos no diagrama.

Por conta disso, apenas o diagrama de atividades foi modelado e está representado pela Figura 6.11.

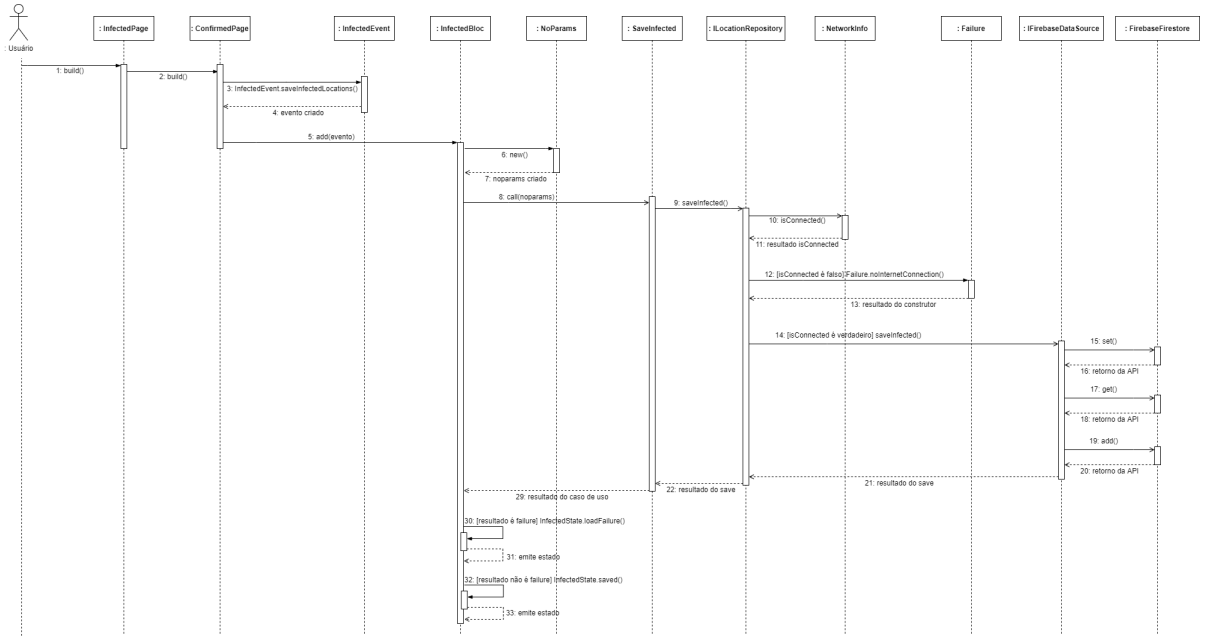

Figura 6.8: Diagrama de sequência da funcionalidade declarar infecção.

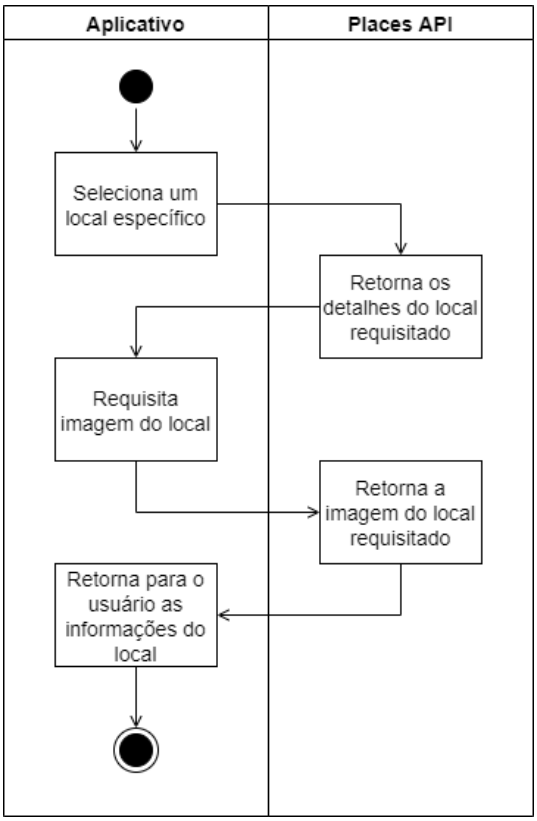

Figura 6.9: Diagrama de atividades da funcionalidade visualizar local.

Por último, a funcionalidade "Listar locais visitados"está representada pelo diagrama de atividades e de sequência da Figura 6.12 e da Figura 6.13.

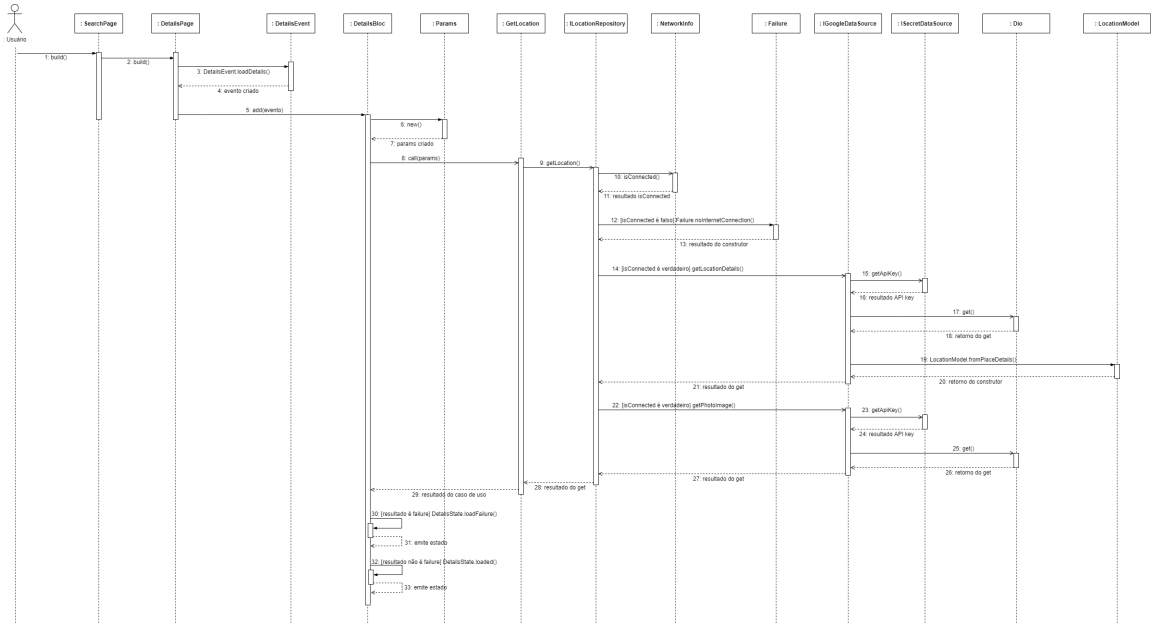

Figura 6.10: Diagrama de sequência da funcionalidade visualizar local.

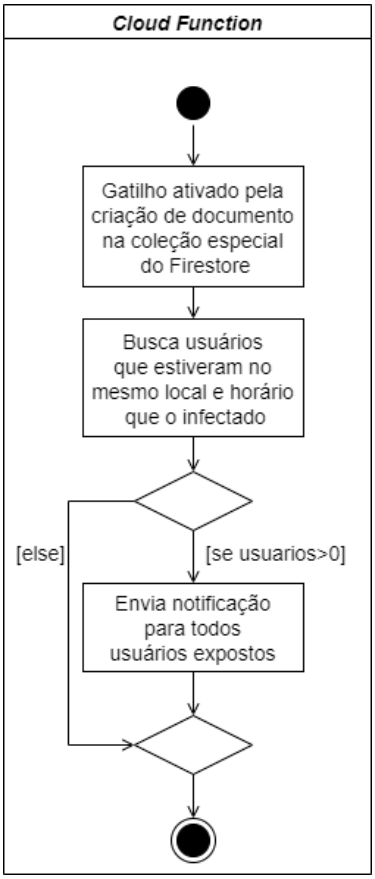

Figura 6.11: Diagrama de atividades da funcionalidade notificar exposição.

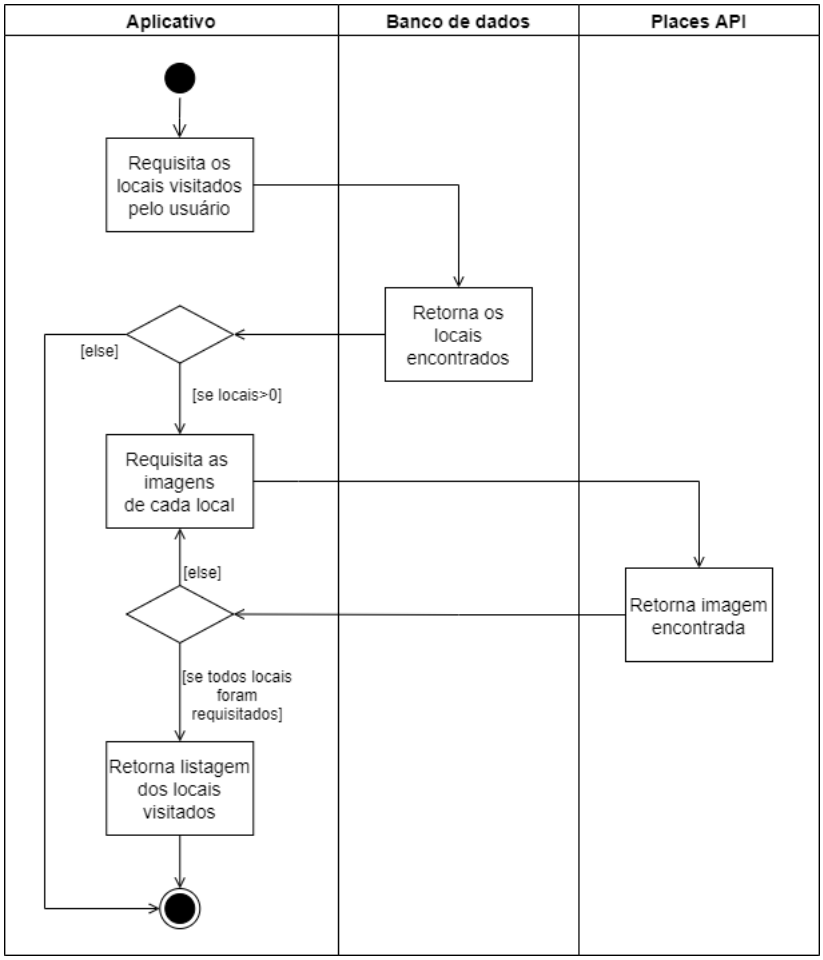

Figura 6.12: Diagrama de atividades da funcionalidade listar locais.

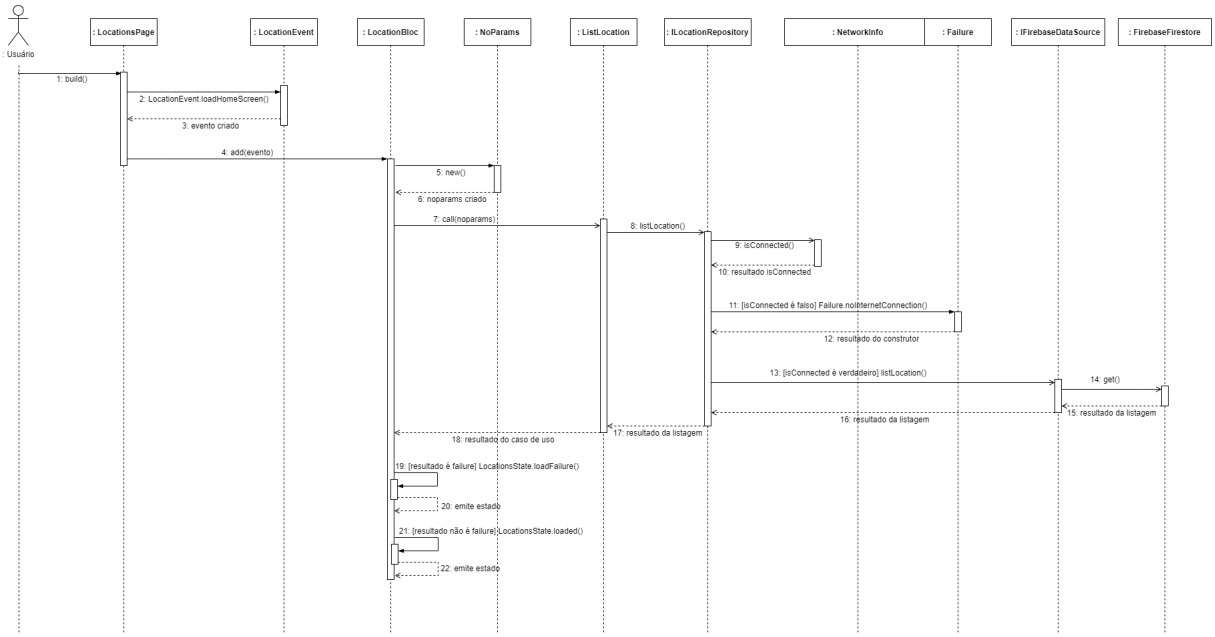

Figura 6.13: Diagrama de sequência da funcionalidade listar locais.

6.2.3 Modelagem da arquitetura do *software*

Como explicado na Seção 3.3, o aplicativo desenvolvido neste trabalho utiliza os princípios da arquitetura limpa. Por conta disso, a base da arquitetura do *software* é a Figura 3.2.

Antes de começar a diagramação, é importante entender a diferença entre dependência e fluxo de dados do aplicativo, porque essa diferença é crucial para que a modelagem não quebre nenhuma das regras da arquitetura limpa.

A dependência do código acontece no tempo de compilação, ou seja, só há dependência entre os componentes se houver uma referência direta para outro componente.

Por exemplo, imagine que em uma das classes da camada *Application Business Rules* do aplicativo possua um caso de uso que em sua implementação exista uma chamada para um método de uma classe presente na camada *Interface Adapters*. Nesse exemplo, o caso de uso possuiria dependência com a camada *Interface Adapters*, e estaria ferindo a regra de dependência, já que essa é uma camada externa em relação à *Application Business Rules*.

Já o fluxo de dados seria equivalente ao fluxo de chamadas dentro do código, que não acontece no tempo de compilação, e sim no de execução. Pode parecer contraditório, mas a dependência nem sempre reflete o fluxo de dados do código.

Utilizando o mesmo exemplo anterior, é possível fazer com que o caso de uso não dependa da camada *Interface Adapters*, mas ainda consiga atingir o mesmo comportamento esperado. Para que isso aconteça, uma interface deve ser criada na camada *Application Business Rules*, e ela conterá as assinaturas dos métodos que o caso de uso necessita. Na camada *Interface Adapters*, uma classe implementará os métodos definidos na interface através do relacionamento de herança.

Dessa maneira, através do *Dependency Inversion Principle*, explicado na subseção 3.6.5, a dependência existirá da camada *Interface Adapters* para a camada *Application Business Rules* por conta da herança, mas no fluxo de chamadas em tempo de execução, será da *Application Business Rules* para a camada *Interface Adapters*.

Com a diferença entre dependência e fluxo de dados compreendida, toda modelagem da arquitetura do sistema deve fazer com que nenhuma camada interna dependa de uma mais externa à ela. Para isso, todas as fronteiras das camadas haverão interfaces para possibilitar a comunicação sem que a regra de dependência seja ferida.

Como o aplicativo fará chamadas para componentes de persistência de dados, utilizando APIs para busca de locais e um BD remoto para o armazenamento, essas chamadas serão

realizadas utilizado o padrão de repositório. A partir disso, o fluxo dos dados entre os componentes do sistema serão nessa ordem:

1. *View* faz chamadas dos métodos da *View Model*;
2. *View Model* executa o caso de uso;
3. Caso de uso combina os dados dos repositórios das entidades;
4. Cada repositório retorna os dados dos *Data Sources*;
5. Os dados voltam para a *View* e são mostrados ao usuário;

A partir dessa ordem, a adaptação da Figura 3.2 com o padrão de repositório e da diferença entre dependência e fluxo de dados é representada na Figura 6.14.

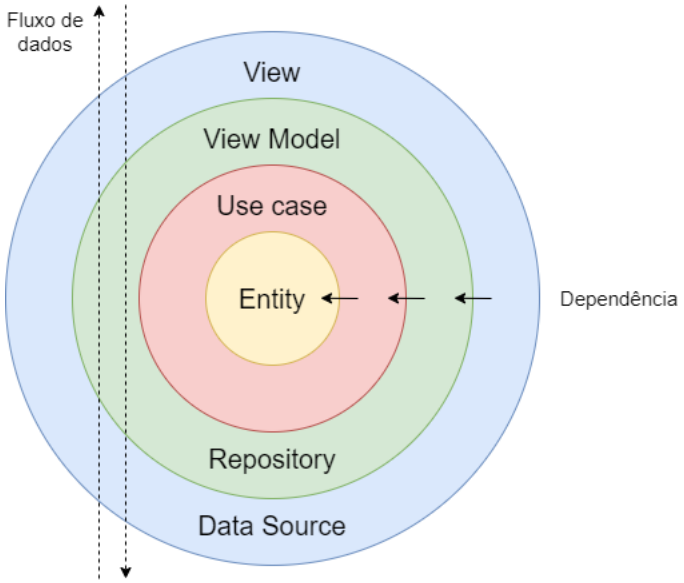

Figura 6.14: Diferença entre dependência e fluxo de dados.

Note que as dependências estão sempre apontando para o interior. Levando em consideração a explicação de como as fronteiras das camadas são passadas através da inversão de dependências, o diagrama de pacotes da Figura 6.15 possui a modelagem mais abstrata e de maior alto nível da arquitetura do aplicativo.

Para que a associação do diagrama da Figura 6.15 com as camadas e regras de dependência da Figura 6.14 sejam melhor visualizadas, a Figura 6.16 representa as camadas da arquitetura limpa com as mesmas colorações por cima do diagrama de pacotes.

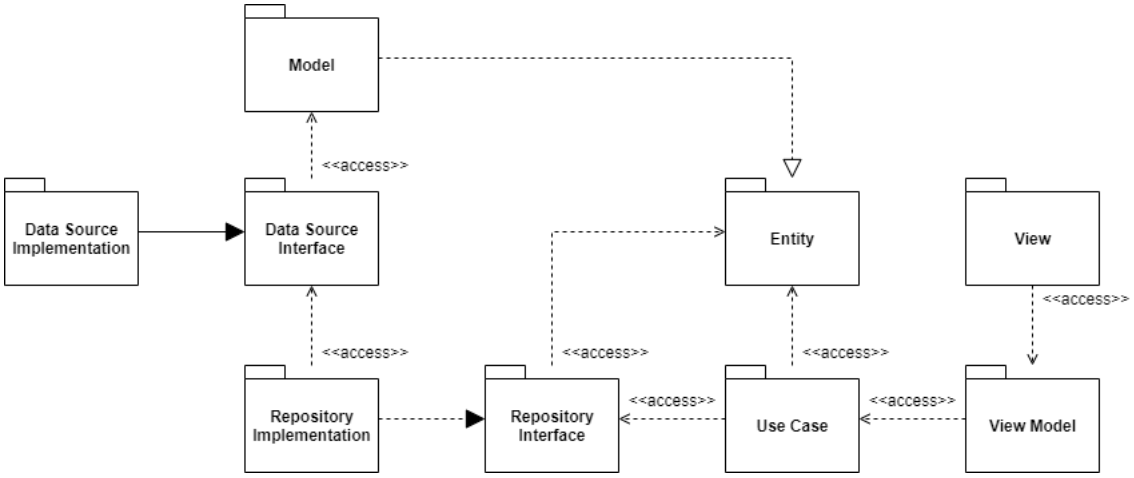

Figura 6.15: Diagrama de pacotes do aplicativo.

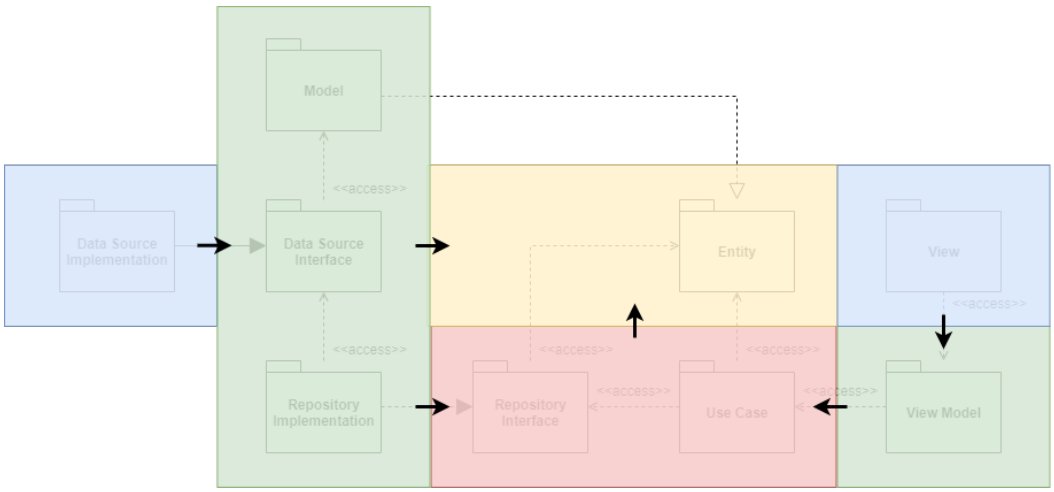

Figura 6.16: Diagrama de pacotes com as colorações das camadas da arquitetura limpa.

Com todos diagramas mostrados anteriormente modelados, o diagrama de classes foi feito com base nos outros e pode ser visualizado na Figura 6.17. Como alguns diagramas ficaram grandes, a visualização deles fica impossibilitada nas páginas deste trabalho, por isso, visualize a pasta figuras[1] no repositório do *GitHub* para uma visão mais detalhada.

6.2.4 Modelagem de dados

O sistema de rastreamento de dados necessita apenas de algumas informações persistidas sobre os usuários para que a análise de encontro entre eles seja efetuada com sucesso.

Para cada usuário deverá ser armazenado os locais visitados e um identificador aleatório que possibilita o envio de notificações ao seu dispositivo móvel. Para isso, será utilizado uma coleção no *Firestore* chamada *users*. Nela serão armazenados múltiplos documentos, em que

[1]Disponível em: <https://github.com/Feggah/tcc/tree/main/docs/figuras>. Acesso em: 21 out. 2021.

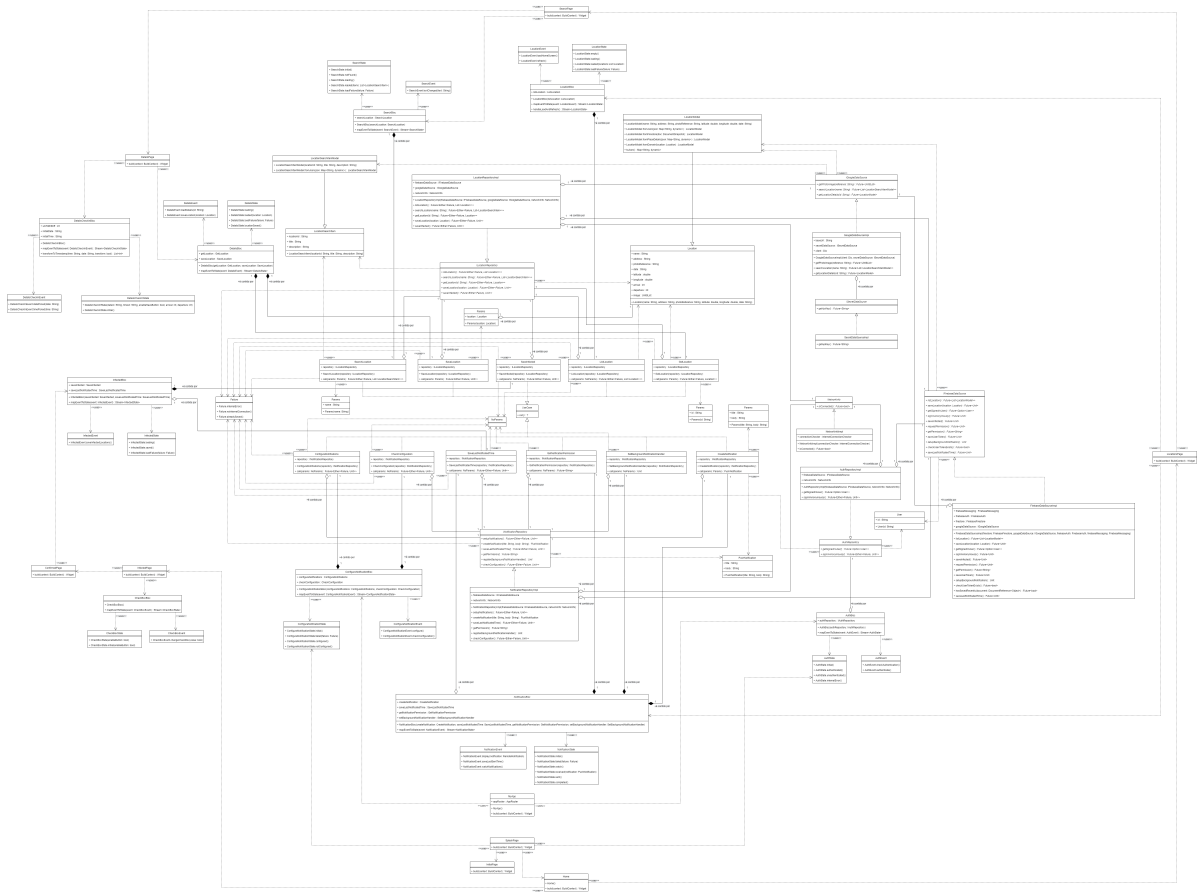

Figura 6.17: Diagrama de classes do aplicativo.

cada um representa um usuário. Em cada documento haverá o *token* e a subcoleção dos locais visitados.

Além disso, dois campos opcionais também serão necessários. Um deles garantirá que o usuário não seja notificado múltiplas vezes em um curto período de tempo, o outro garantirá que ele não consiga se declarar como infectado múltiplas vezes dentro do período de 14 dias. O nome desses dois campos são respectivamente chamados de *lastNotified* e *lastSaved*.

Os documentos que representam os locais visitados serão compostos por alguns valores de retorno da API do *Google*, que são os campos endereço, latitude, longitude, nome e o ID da foto. Para armazenar o horário que o usuário esteve no local, serão utilizados campos no formato *timestamp* do horário de chegada e de saída que forem preenchidos pelo usuário na interface do aplicativo.

Os locais considerados como infectados serão armazenados em outra coleção com o objetivo de diminuição de profundidade da estrutura do BD, resultando em menos requisições de leitura. A consequência disso é que o tempo de processamento do algoritmo de análise de rastreamento e os custos da nuvem diminuirão, porque será necessário apenas

uma requisição de leitura para obter todos os locais infectados e o valor cobrado é diretamente relacionado com o número de requisições efetuadas.

Essa coleção tem o nome *infected* e cada um de seus documentos representam um local, contendo o horário, latitude e longitude do local classificado como infectado.

Como resultado desse racional apresentado acima, a modelagem da Figura 6.18 representa a estrutura e todos atributos obrigatórios e opcionais de cada documento.

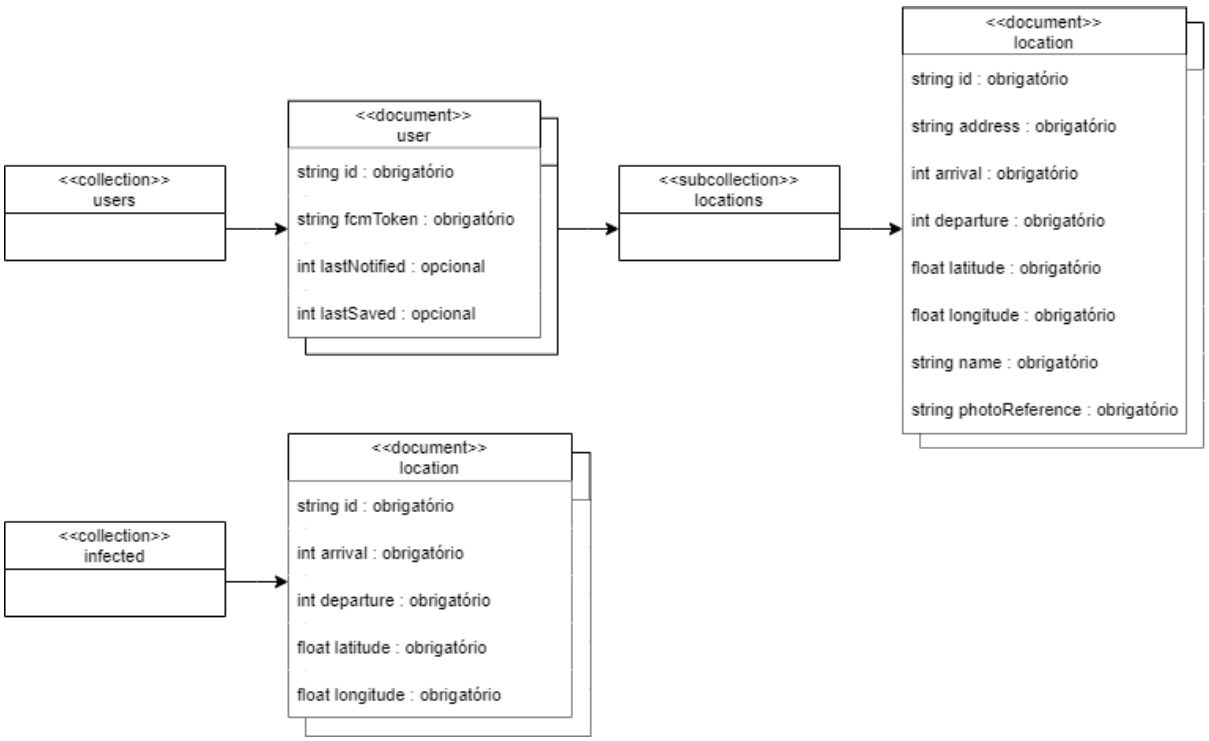

Figura 6.18: Modelagem do banco de dados não relacional.

6.3 Interfaces de usuário do aplicativo

O *design* das interfaces do aplicativo foi criado utilizando o *Figma*, que é um editor online gratuito de gráficos vetoriais com ênfase na prototipagem de interfaces gráficas. Todos protótipos de tela construídos podem ser visualizados na página do projeto[2] dentro da ferramenta.

As principais telas do aplicativo são as que estão diretamente associadas à uma das funcionalidades da Tabela 6.2 e representadas pelo diagrama de casos de uso da Figura 6.2, que são as telas de listagem, busca e visualização de locais e a de declaração de caso de infecção. Os protótipos das quatro telas podem ser visualizados na Figura 6.19.

[2]Disponível em: <https://www.figma.com/file/x11PMoHwniQ0e1Jb6T6LRk/TCC>. Acesso em: 23 out. 2021.

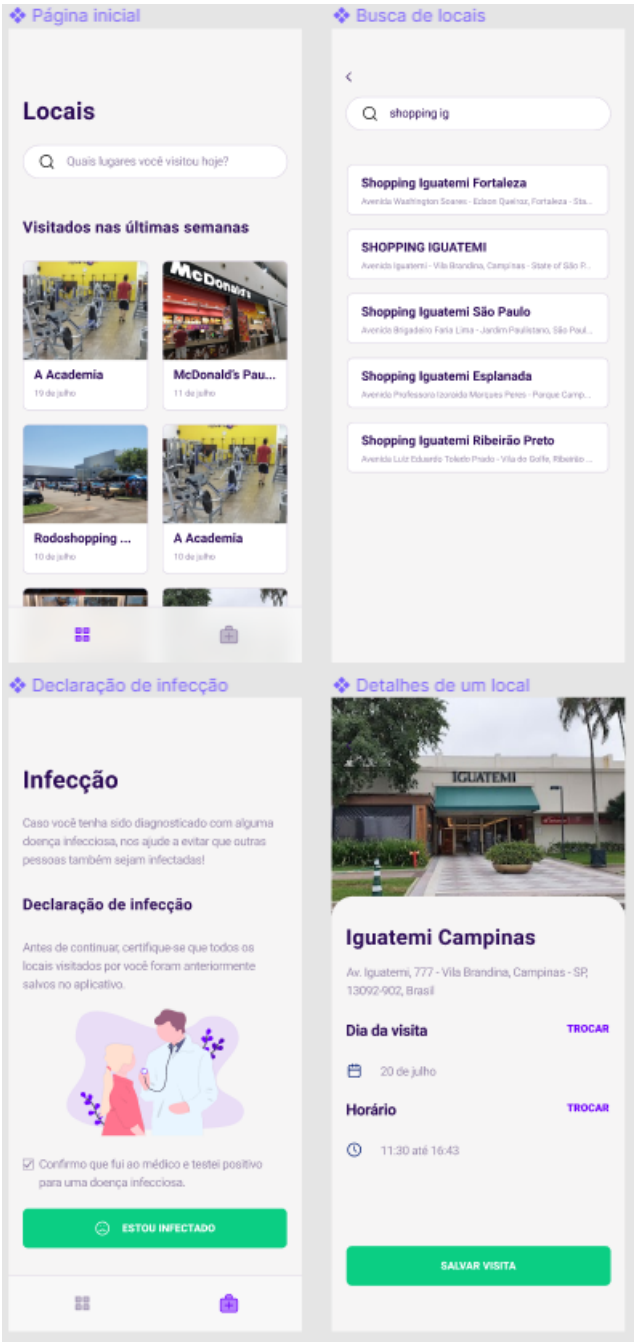

Figura 6.19: Protótipo das principais telas do aplicativo móvel.

Além das quatro telas principais, existem algumas variações das mesmas em casos de erro, falta de conexão com a *Internet* e páginas de *feedback* para alguma ação do usuário. Por exemplo, quando o usuário se auto declara como infectado, ele visualizará uma página de confirmação informando que o processamento foi concluído com sucesso.

6.4 Implementação

As tarefas necessárias para conclusão do projeto foram criadas no formato de *issues* no repositório do *GitHub*. Cada issue foi classificada entre *feature*, *release*, *diagram* e *documentation* através de rótulos adicionados à elas.

O rótulo de *release* significa que aquela *issue* será concluída no momento que uma nova versão do sistema seja lançada. Isso também significa que uma *release* contém várias *issues* do tipo *feature* que devem ser concluídas primeiramente.

Um exemplo de *release* foi o lançamento do *Minimum viable product* (MVP), que continha como pré-requisito a conclusão de todas as *issues* que representavam as funcionalidades levantadas na Seção 6.1. A Figura 6.20 é um *printscreen* que mostra como foi estruturada as *issues* do repositório.

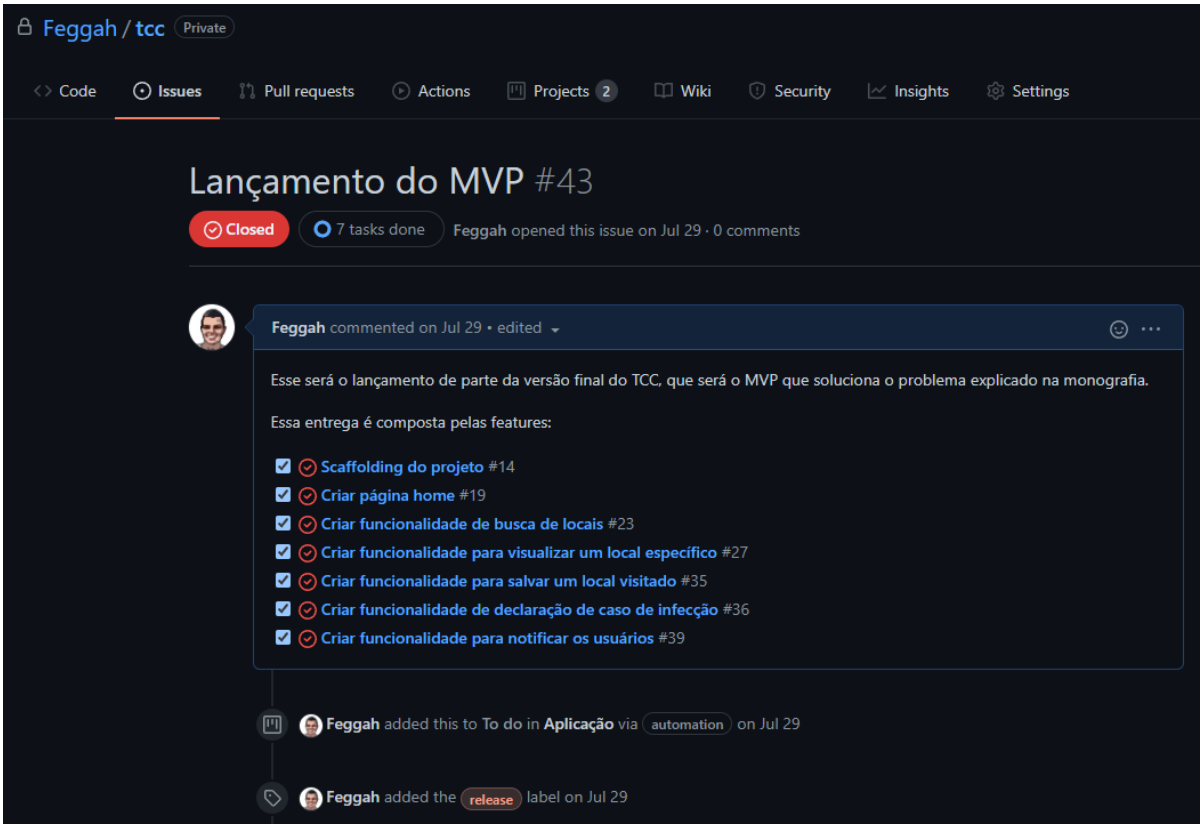

Figura 6.20: *Printscreen* da *issue* de lançamento do MVP.

Todas as *issues* seguem esse mesmo modelo: possuem um título, descrição do que deve ser feito e um rótulo classificando qual o seu tipo.

O rótulo *feature* representa uma funcionalidade a ser implementada no sistema. Essa funcionalidade não deve necessariamente representar uma relação um para um com as

funcionalidades levantadas na Seção 6.1, ou seja, elas podem ser quebradas em entregas menores.

O rótulo *documentation* está relacionado a tarefas de desenvolvimento do texto da monografia. Por exemplo, uma *issue* que representa o desenvolvimento de um dos capítulos deste trabalho.

A última classificação, o rótulo *diagram*, representa tarefas que estavam relacionadas a modelagem dos diagramas criados para este trabalho. No caso desse rótulo, foi utilizado uma *issue* para cada tipo de diagrama, por exemplo, a *issue* sobre o diagrama de atividades representava a modelagem dos seis diagramas criados.

Para que a visualização e progresso das *issues* fosse melhor visualizado, foi utilizado um quadro *Kanban* oferecido nativamente pelo *GitHub*, que são chamados de projetos. Foram criados dois projetos, um representando a implementação de todo o sistema e o outro o desenvolvimento da monografia e assuntos relacionados.

Os dois quadros *Kanbans* foram divididos em três colunas: *To do*, *in progress* e *done*. As *issues* que estavam dentro de cada quadro eram movidas conforme o seu estado mudava em relação a coluna que ela estava inserida. Por exemplo, a Figura 6.21 representa o estado do quadro da monografia durante o desenvolvimento desta seção.

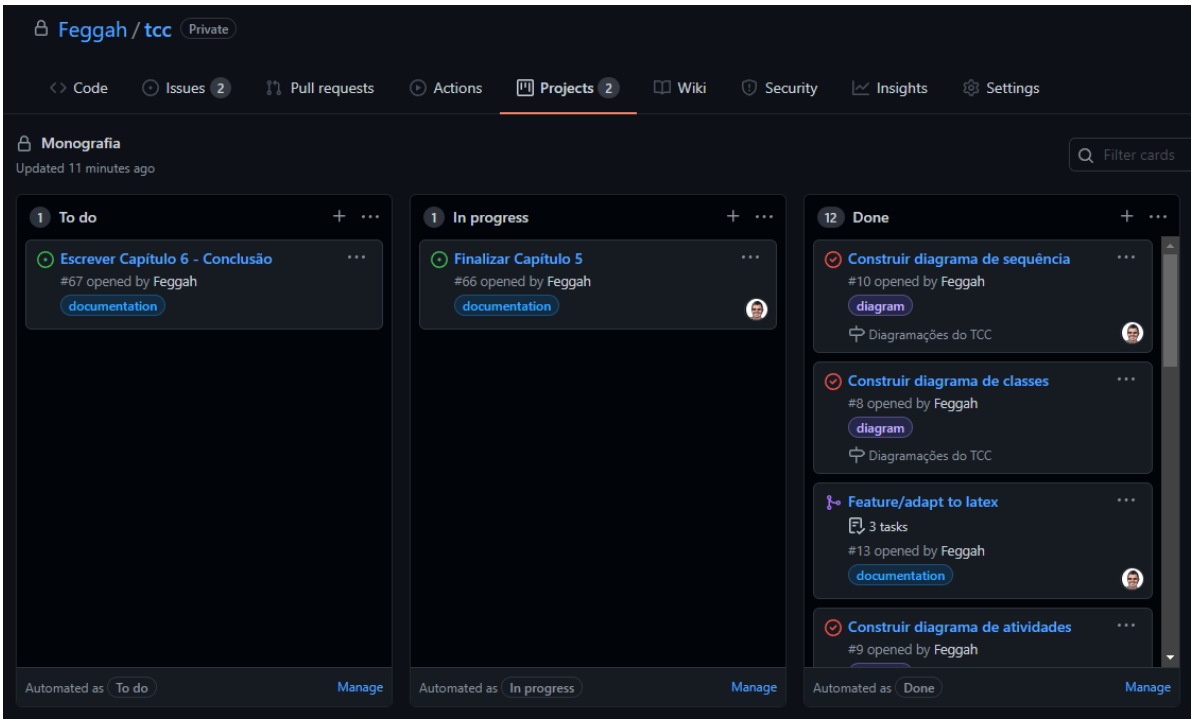

Figura 6.21: *Printscreen* do quadro *Kanban* da monografia.

Como explicado na Seção 3.9, o fluxo de trabalho escolhido para o processo de desenvolvimento do sistema foi o *GitFlow*. Por conta disso, o repositório possui duas ramificações de longa vida, a *main* e a *develop*.

Para exemplificar o fluxo completo do *GitFlow* na prática, a Figura 6.22 possui um exemplo de *pull request* para a ramificação *develop*.

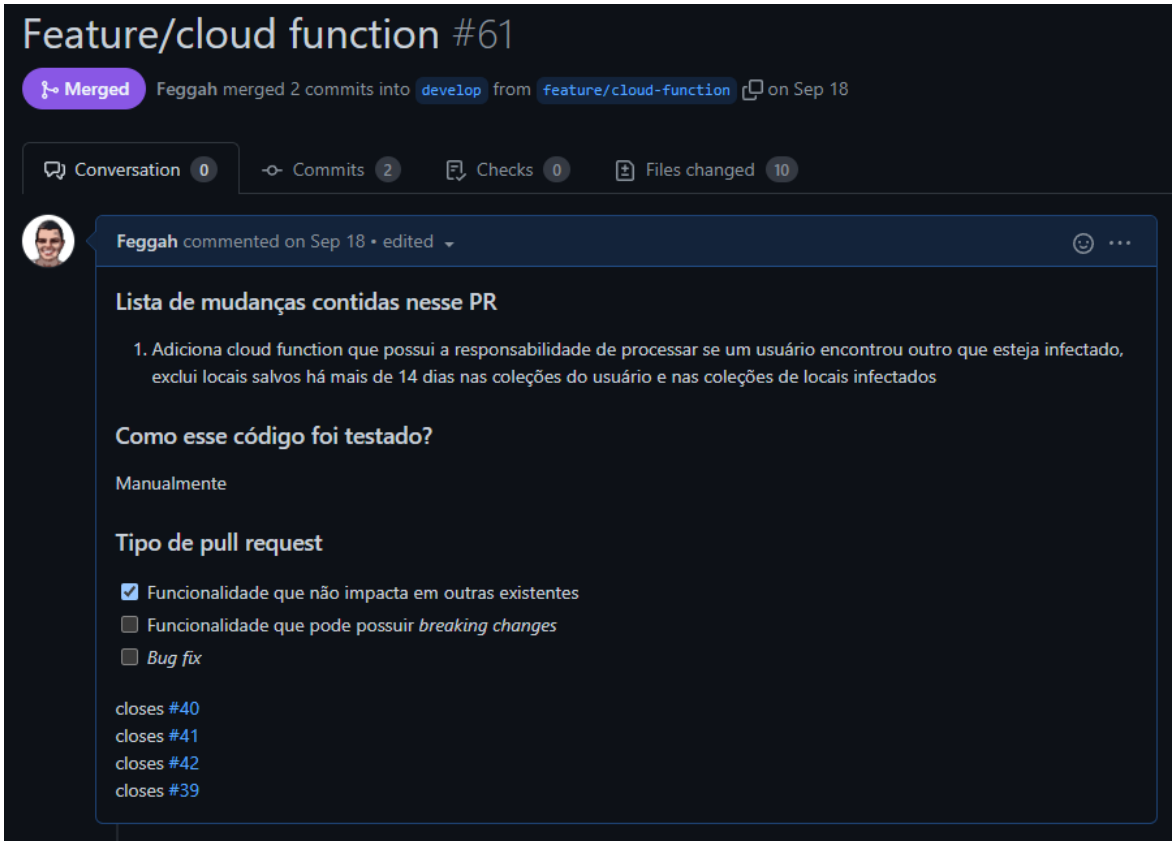

Figura 6.22: *Printscreen* de exemplo de um *pull request* do repositório.

Depois de algumas iterações de funcionalidades mescladas na ramificação *develop*, a ramificação *release* é criada a partir da *develop* e são criados dois *pull requests*, um para a *main* e outro para a *develop*. A Figura 6.23 representa o *pull request* para a main e a Figura 6.24 para a *develop*.

Com isso, a ramificação principal do repositório possui as novas funcionalidades desenvolvidas e uma nova versão está pronta para ser lançada. O lançamento de versão é feito adicionando uma *tag*, que fixa um ponto histórico no repositório e representa uma versão da aplicação.

Com a nova *tag* criada, um novo lançamento é feito através da funcionalidade de *releases* do *GitHub*. Uma *release* contém título, descrição, a *tag* que está associada à ela e arquivos que possam fazer parte desse lançamento. No caso do aplicativo, uma release contém o arquivo

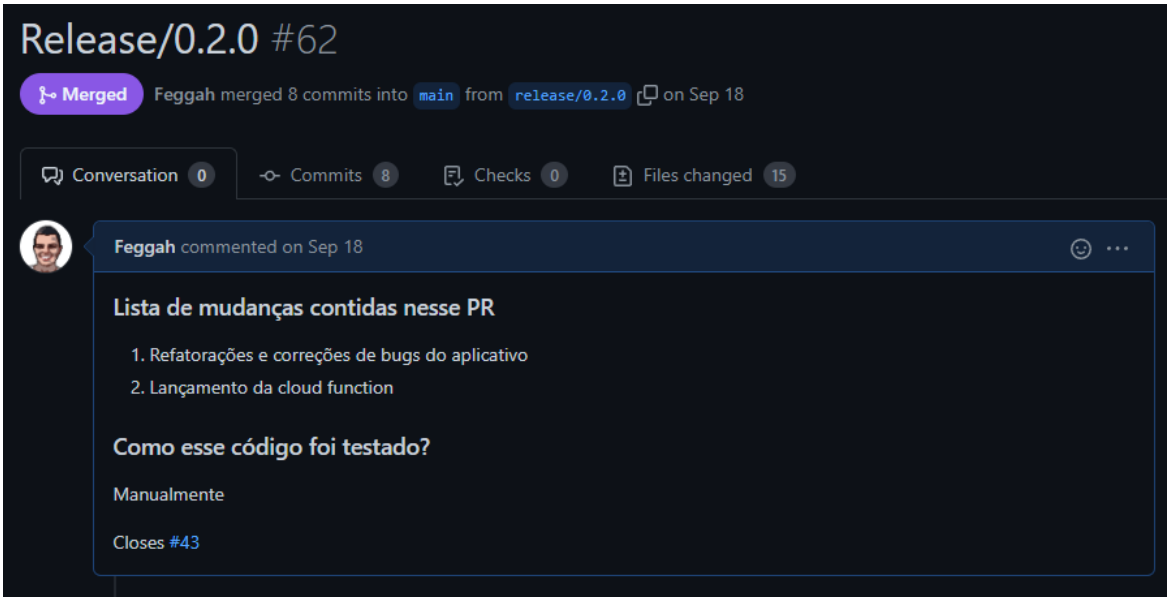

Figura 6.23: *Printscreen* de exemplo de um *pull request* da ramificação *release* para *main* do repositório.

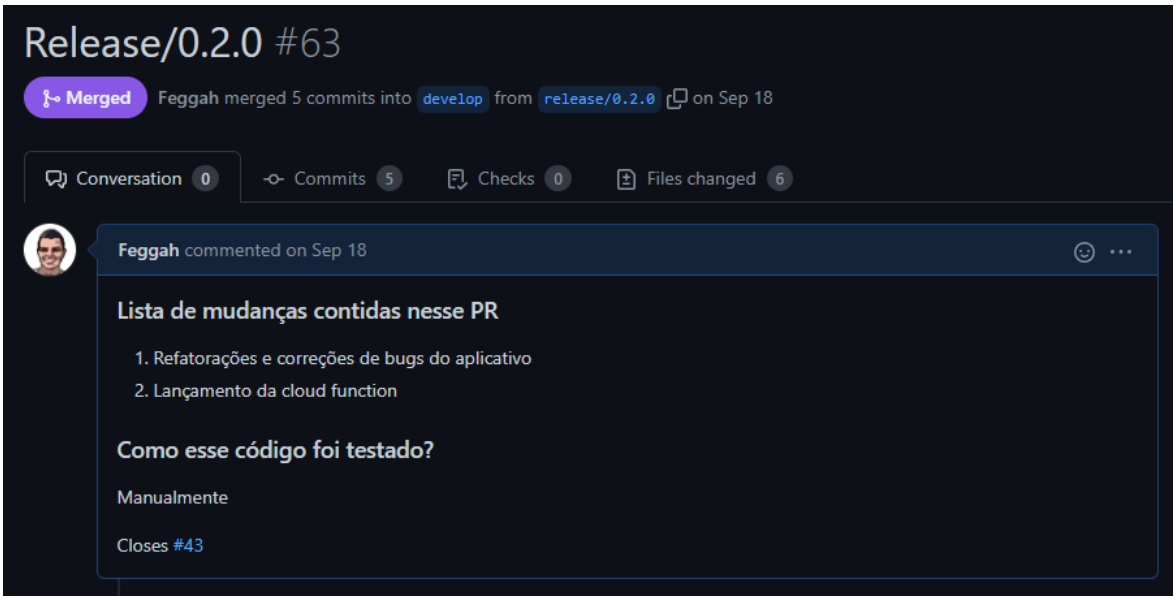

Figura 6.24: *Printscreen* de exemplo de um *pull request* da ramificação *release* para *develop* do repositório.

Android Package (APK) que pode ser baixado e instalado nos dispositivos. A Figura 6.25 representa os lançamentos feitos no repositório.

Além do fluxo usual de incremento de funcionalidades, algumas correções pontuais podem ser feitas na versão que já está na ramificação principal através de ramificações denominadas de *hotfix*. A Figura 6.26 representa um exemplo desse tipo de modificação.

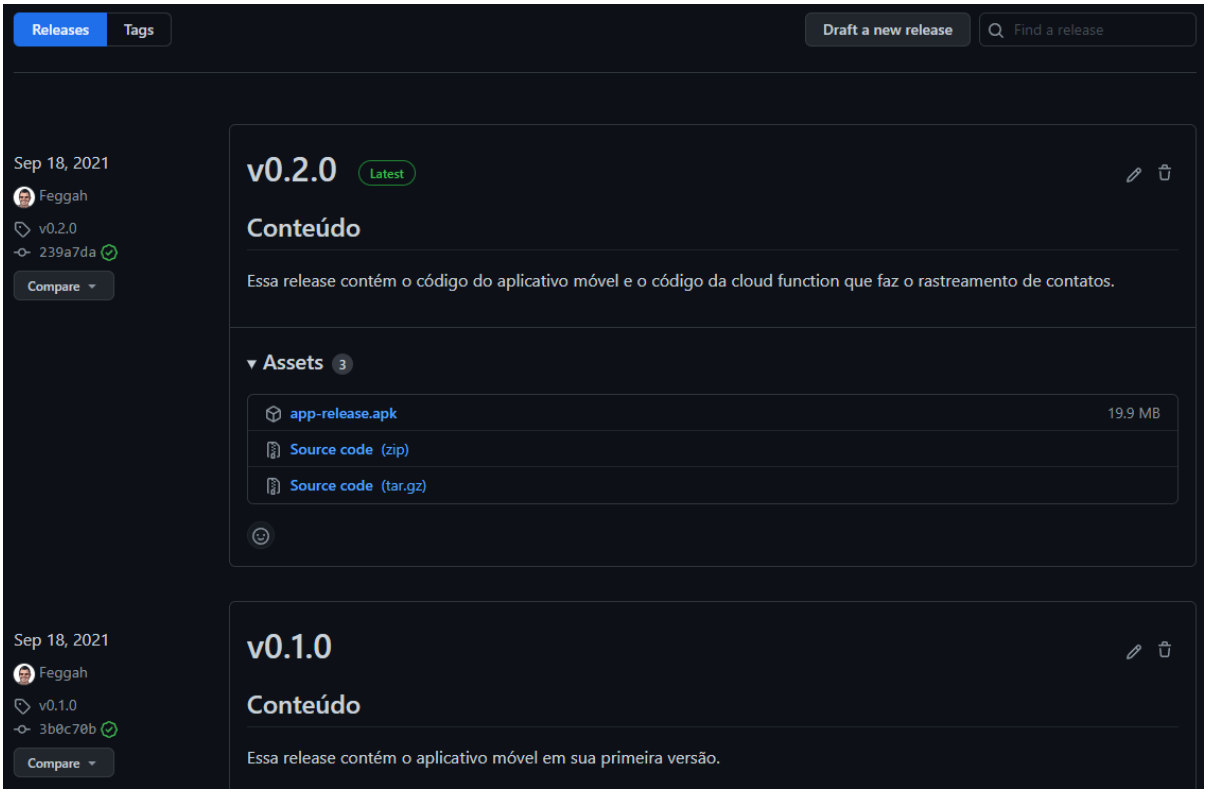

Figura 6.25: *Printscreen* das *releases* do repositório.

Figura 6.26: *Printscreen* de exemplo de um *pull request* de *hotfix*.

Como o fluxo de trabalho escolhido utiliza o modelo de *pull requests*, é importante que em cada um rode os testes automatizados, a fim de garantir que as novas modificações não estão quebrando o comportamento de funcionalidades antigas.

Para isso, foi desenvolvido um *pipeline* utilizando as *GitHub Actions*. Esse *pipeline* roda todos os testes quando um novo *pull request* é aberto e quando são feitos novos *commits* na ramificação principal do repositório. A Figura 6.27 exemplifica os testes automatizados que foram ativados no evento de *commit* da ramificação principal.

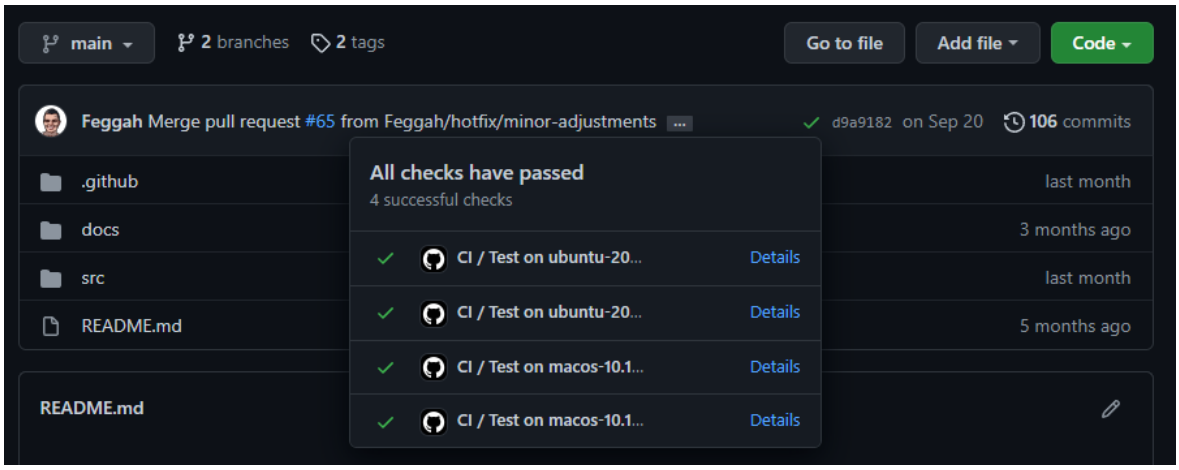

Figura 6.27: *Printscreen* dos testes automatizados de *commit* na ramificação principal.

Durante a implementação do aplicativo, a estrutura de pastas e arquivos reflete diretamente o diagrama de pacotes que foi apresentado na Seção 6.2. Existem quatro principais pastas: *data*, *domain*, *presentation* e *shared*.

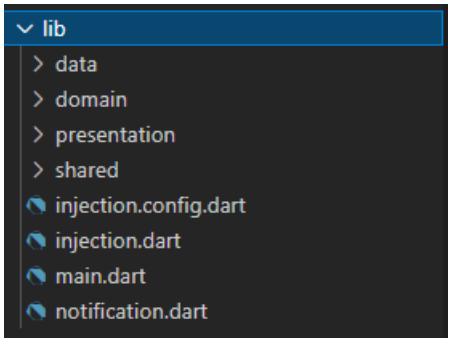

Figura 6.28: *Printscreen* da estrutura de pastas do aplicativo.

A pasta *data* contém as implementações dos repositórios, os modelos que são responsáveis pelas conversões entre a entidade interna do aplicativo e a sua representação externa em *frameworks* ou bibliotecas terceiras, e os *datasources*, que são responsáveis por se comunicarem com a *Internet*.

A pasta *domain* contém as entidades da aplicação, as interfaces de repositórios e os casos de uso. Essa pasta é a que deve sofrer menos alterações com o passar do tempo, porque não depende de *frameworks* ou outros fatores que mudam com frequência.

A pasta *presentation* contém as *viewmodels*, a *view* e as rotas do aplicativo. As rotas são utilizadas para fazer a navegação entre as diferentes telas. A *view* contém a definição da interface com os usuários, ou seja, todos os protótipos criados no *Figma* e citados na Seção 6.3. As *viewmodels* contém a lógica de apresentação das telas, sendo responsável por receber eventos e emitir estados que causam o recarregamento das interfaces para o estado desejado.

A pasta *shared* contém arquivos que são compartilhados entre diversas camadas. Esses arquivos representam as exceções, as possíveis falhas, extensões de bibliotecas terceiras, a implementação de checagem de conectividade com a *Internet* e a interface dos casos de uso.

A visão das pastas expandidas é representada pela Figura 6.29.

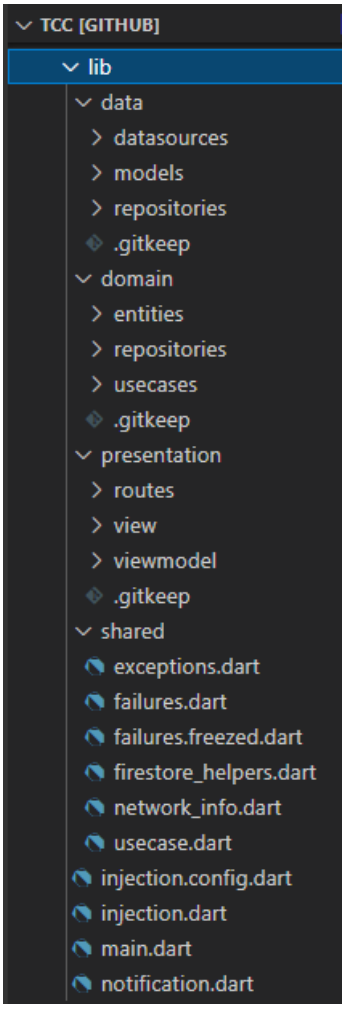

Figura 6.29: *Printscreen* da estrutura de pastas expandidas do aplicativo.

O aplicativo possui uma extensa lista de dependências em bibliotecas de terceiros que foram importadas para cumprir algum objetivo específico. As principais dependências do aplicativo são:

- Bibliotecas do *Firebase*, que são alguns *Software Development Kit* (SDK) que fazem a comunicação com as APIs do *Firebase*;
- *Flutter_bloc*, que é um pacote para gerenciamento de estados;
- *Dartz*, que é uma biblioteca que introduz programação funcional ao *Dart*;
- *Dio*, que é uma biblioteca para fazer requisições HTTP;
- *Get_it*, que é uma biblioteca para utilização do padrão de projeto *Service Locator* para *Dart* e *Flutter*. Esse padrão de projeto é usado para encapsular os processos envolvidos na obtenção de um serviço com uma forte camada de abstração;
- *Equatable*, que facilita a comparação de equalidade entre objetos;
- *Freezed*, que é um gerador de código para classes imutáveis, introduzindo métodos auxiliares para seus objetos;
- *Injectable*, que é um gerador de código para a biblioteca *get_it*;

Uma das partes mais importantes da implementação do aplicativo é a inversão de dependências, já explicado na subseção 3.6.5. Essa inversão foi implementada utilizando a biblioteca *injectable*.

Para fins de exemplificação, abaixo está exemplificado uma inversão de dependência entre a camada de domínio e a de dados, mais especificamente entre o caso de uso e o repositório.

A dependência do código está do repositório para o caso de uso e o fluxo de dados do caso de uso ao repositório. Para atigir isso, na camada de domínio foi definida a interface do repositório, que neste exemplo é a *ILocationRepository*, representada pela Figura 6.30.

src > exposure > lib > domain > repositories > i_location_repository.dart

```dart
import 'package:dartz/dartz.dart';
import 'package:exposure/domain/entities/location.dart';
import 'package:exposure/domain/entities/location_search_item.dart';
import 'package:exposure/shared/failures.dart';

abstract class ILocationRepository {
  Future<Either<Failure, List<Location>>> listLocation();
  Future<Either<Failure, List<LocationSearchItem>>> searchLocation(String name);
  Future<Either<Failure, Location>> getLocation(String id);
  Future<Either<Failure, Unit>> saveLocation(Location location);
  Future<Either<Failure, Unit>> saveInfected();
}
```

Figura 6.30: *Printscreen* da interface *ILocationRepository*.

Na mesma camada temos o caso de uso *ListLocation*, representado na Figura 6.31, que depende da interface *ILocationRepository*. Essa dependência não fere os princípios da arquitetura limpa porque ambos estão na mesma camada.

src > exposure > lib > domain > usecases > list_location.dart

```dart
import 'package:dartz/dartz.dart';
import 'package:exposure/domain/entities/location.dart';
import 'package:exposure/domain/repositories/i_location_repository.dart';
import 'package:exposure/shared/failures.dart';
import 'package:exposure/shared/usecase.dart';
import 'package:injectable/injectable.dart';

@LazySingleton()
class ListLocation implements UseCase<Future, NoParams> {
  final ILocationRepository repository;

  ListLocation(this.repository);

  @override
  Future<Either<Failure, List<Location>>> call(NoParams params) {
    return repository.listLocation();
  }
}
```

Figura 6.31: *Printscreen* do caso de uso *ListLocation*.

Para que a inversão de dependência ocorra, em tempo de execução o atributo *repository* da classe *ListLocation* receberá uma instância da classe *LocationRepositoryImpl* no lugar da *ILocationRepository*. Isso será possível por conta do princípio de Liskov, explicado na subseção 3.6.3.

A classe que implementa a interface *ILocationRepository* é chamada de *LocationRepositoryImpl* e está representada na Figura 6.32. Note que o decorador da classe, que foi importado da biblioteca *injectable*, faz com que as instâncias da classe sejam utilizadas nos atributos do tipo *ILocationRepository*.

```dart
@LazySingleton(as: ILocationRepository)
class LocationRepositoryImpl implements ILocationRepository {
  final IFirebaseDataSource firebaseDataSource;
  final IGoogleDataSource googleDataSource;
  final NetworkInfo networkInfo;

  LocationRepositoryImpl({
    required this.firebaseDataSource,
    required this.googleDataSource,
    required this.networkInfo,
  });

  @override
  Future<Either<Failure, List<Location>>> listLocation() async {
```

Figura 6.32: *Printscreen* da classe *LocationRepositoryImpl*.

Esse mesmo modelo foi implementado em todas as fronteiras entre camadas, definindo a interface em uma camada e sua implementação em outra. Com isso, esse princípio da arquitetura limpa foi cumprido em todo aplicativo.

A implementação da *cloud function* que faz a análise do rastreamento entre contatos é responsável por notificar os usuários que forem expostos à alguma doença. A notificação recebida por um dispositivo está exemplificada na Figura 6.33.

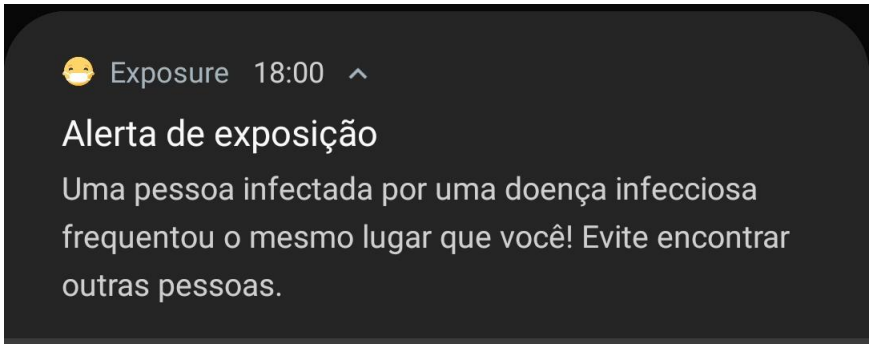

Figura 6.33: *Printscreen* da notificação recebida no dispositivo celular.

Um usuário deverá ser notificado se:

- A latitude e longitude dos locais salvos são as mesmas. Isso evita falsos positivos em locais com o mesmo nome mas em localizações diferentes. Por exemplo, franquias de uma empresa que estão localizadas em cidades diferentes;
- O horário de chegada do usuário é menor que o horário de saída do infectado;
- O horário de chegada do infectado é menor que o horário de saída do usuário sendo analisado;

Se todas as condições forem verdadeiras, utiliza-se o *fcmToken*, explicado anteriormente na subseção 6.2.4, do documento que representa o usuário para enviar uma notificação ao seu dispositivo. O trecho de código da Figura 6.34 referente ao *script* da *cloud function* representa essa análise.

As *cloud functions* são definidas a partir de uma sintaxe em que é especificado o evento e o caminho da coleção que serão utilizados para ativá-la. Os possíveis eventos são *OnCreate*, *onUpdate*, *onDelete* e *onWrite*, que respectivamente correspodem aos eventos de criação, alteração, deleção ou todos anteriores.

```
    const hasToBeNotified = visitedPlaces
        .some((place) => infectedVisitedPlaces
            .some((infectedPlace) => (
                place.latitude == infectedPlace.latitude
                && place.longitude == infectedPlace.longitude
                && place.arrival < infectedPlace.departure
                && infectedPlace.arrival < place.departure
            )));

    const deviceToken = (await db.collection('users')
        .doc(user.id)
        .get())
        .data()
        .fcmToken;

    if (hasToBeNotified) {
        return deviceToken;
    }
});

webpush = {
    notification: {
        notification: {
            title: "Alerta de exposição",
            body: "Uma pessoa infectada por uma doença infecciosa frequentou o mesmo lugar que você! Evite encontrar outras pessoas.",
        },
        data: {
            click_action: 'FLUTTER_NOTIFICATION_CLICK'
        }
    },
}
```

Figura 6.34: *Printscreen* do trecho de código da *cloud function*.

As duas *cloud functions* desenvolvidas neste trabalho são ativadas a partir de eventos de criação, uma na coleção de locais infectados e outra na subcoleção de locais salvos pelo usuário. A Figura 6.35 representa a declaração dessas funções.

```
exports.removeUserOldPlaces = functions.firestore.document('users/{userId}/locations/{location}')
    .onCreate(async (change, context) => {···
    });


exports.removeInfectedOldPlaces = functions.firestore.document('/infected/{wildcard}')
    .onCreate(async (change, context) => {···
    });
```

Figura 6.35: *Printscreen* da declaração das *cloud functions*.

Em relação a autenticação dos usuários, os IDs são criados aleatoriamente pelo serviço de autenticação do *Firebase*. Como explicado em capítulos anteriores, a autenticação é feita de forma anônima e por conta disso a real identidade do usuário é preservada. Os únicos dados que podem ser extraídos dessa autenticação são a data de criação da conta, a data do último *login* e o seu ID aleatório.

Todo mecanismo de autenticação é feito de forma transparente ao usuário no momento que ele inicia o aplicativo pela primeira vez. Caso o aplicativo seja deletado e instalado novamente, uma nova conta será criada, já que não há mecanismos de associação para definir a real identidade de cada usuário fora do sistema.

Capítulo 7

Conclusões

Este Capítulo conterá as discussões finais referentes ao problema e solução propostas neste trabalho.

Referências bibliográficas

DATASUS. **Coronavírus - SUS**. BR: Governo do Brasil, mar. 2020. Disponível em: <https://play.google.com/store/apps/details?id=br.gov.datasus.guardioes&hl=pt_BR&gl=US>. Acesso em: 3 jun. 2021.

DRIESSEN, V. (Ed.). **A successful Git branching model**. 2020. Disponível em: <https://nvie.com/posts/a-successful-git-branching-model/>. Acesso em: 13 jun. 2021.

DUNCAN, B. B. et al. Doenças Transmissíveis: Condutas Preventivas na Comunidade. In: TOSCANO, C. M. (Ed.). **Medicina ambulatorial: Condutas de Atenção Primária Baseadas em Evidências**. 4. ed. [S.l.]: Artmed, 2013. cap. 129, p. 1332–1347. ISBN 9788536326184.

EVANS, E. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. Illustrated. Boston: Addison-Wesley Professional, 2003. p. 529. ISBN 978-0321125217.

FIRESTORE (Ed.). **Cloud Firestore**. 2021. Disponível em: <https://firebase.google.com/docs/firestore>. Acesso em: 3 jun. 2021.

FIRESTORE (Ed.). **Modelo de dados do Cloud Firestore**. 2021. Disponível em: <https://firebase.google.com/docs/firestore/data-model>. Acesso em: 3 jun. 2021.

FOWLER, M. **Patterns of Enterprise Application Architecture**. 1. ed. Boston: Addison-Wesley Professional, 2002. p. 560. ISBN 978-0321127426.

GATES, B. (Ed.). **The next outbreak? We're not ready**. 2015. Disponível em: <https://www.ted.com/talks/bill_gates_the_next_outbreak_we_re_not_ready>. Acesso em: 26 mai. 2021.

GEBREKAL, T. (Ed.). **What Worries the World – March 2021**. 2021. Disponível em: <https://www.ipsos.com/en/what-worries-world-march-2021>. Acesso em: 26 mai. 2021.

GOOGLE; APPLE (Ed.). **Notificações de Exposição: tecnologia a serviço das autoridades de saúde pública no combate à COVID-19**. 2020. Disponível em: <https://www.google.com/covid19/exposurenotifications/>. Acesso em: 1 jun. 2021.

LISKOV, B. H.; WING, J. M. **Behavioral Subtyping Using Invariants and Constraints**. Pittsburgh, jul. 1999. DOI: 10.21236/ada367674.

MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. 1. ed. [S.l.]: Pearson, 2017. p. 432. ISBN 9780134494272.

MARTIN, R. C. (Ed.). **Design Principles and Design Patterns**. 2000. Disponível em: <https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf>. Acesso em: 13 jun. 2021.

MARTIN, R. C. (Ed.). **The Clean Architecture**. 2012. Disponível em: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>. Acesso em: 15 jun. 2021.

ODA, R. A. M. (Ed.). **Doenças transmissíveis pelo ar em Biologia**. 2021. Disponível em: <https://descomplica.com.br/d/vs/aula/doencas-transmissiveis-pelo-ar/>. Acesso em: 12 jun. 2021.

REDHAT (Ed.). **O que é arquitetura orientada a eventos?** 2021. Disponível em: <https://www.redhat.com/pt-br/topics/integration/what-is-event-driven-architecture>. Acesso em: 13 jun. 2021.

RODRIGUES, J. **MOOPE SISTEMA DE RASTREAMENTO E GESTÃO DE FROTAS - Modulo Aplicativo ios/Android,Módulo Frotas,Modulo Central Comandos,Modulo Central Notificações, Modulo MeusMoopes**. 21 jun. 2021. Registro no INPI: BR512020002327-4.

UML (Ed.). **INTRODUCTION TO OMG'S UNIFIED MODELING LANGUAGE**. 2005. Disponível em: <https://www.uml.org/what-is-uml.htm>. Acesso em: 14 jun. 2021.