INSTITUTO FEDERAL GOIANO - CAMPUS MORRINHOS
CURSO SUPERIOR DE TECNÓLOGO EM SISTEMAS PARA INTERNET

FELIPE NÁRIO DE SOUZA FONTINELLI

DOEBRASIL.ORG: PLATAFORMA PARA INTERMEDIAÇÃO DE DOAÇÕES

MORRINHOS - GO 2023

FELIPE NÁRIO DE SOUZA FONTINELLI

DOEBRASIL.ORG: PLATAFORMA PARA INTERMEDIAÇÃO DE DOAÇÕES

Monografia apresentada ao Curso Superior de Tecnólogo em sistemas para internet do Instituto Federal Goiano – Campus Morrinhos, como requisito parcial para obtenção de título de Tecnólogo em sistemas para internet.

Área de concentração: Tecnólogo em sistemas para internet

Orientador: Prof. Me. Marcel da Silva Melo

MORRINHOS - GO 2023

Sistema desenvolvido pelo ICMC/USP
Dados Internacionais de Catalogação na Publicação (CIP)
Sistema Integrado de Bibliotecas - Instituto Federal Goiano

FF684d

Fontinelli, Felipe Nário de Souza
DOEBRASIL.ORG: PLATAFORMA PARA INTERMEDIAÇÃO DE DOAÇÕES / Felipe Nário de Souza Fontinelli; orientador Marcel da Silva Melo. -- Morrinhos, 2023.
49 p.

TCC (Graduação em Tecnólogo em Sistemas para Internet) -- Instituto Federal Goiano, Campus Morrinhos, 2023.

1. Flutter. 2. Dart. 3. Doação. 4. Covid-19. 5. Aplicativo. I. Melo, Marcel da Silva, orient. II. Título.

Responsável: Johnathan Pereira Alves Diniz - Bibliotecário-Documentalista CRB-1 n°2376

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO

PARA DISPONIBILIZAR PRODUÇÕES TÉCNICO-CIENTÍFICAS NO REPOSITÓRIO INSTITUCIONAL DO IF GOIANO

Com base no disposto na Lei Federal nº 9.610, de 19 de fevereiro de 1998, AUTORIZO o Instituto Federal de Educação, Ciência e Tecnologia Goiano a disponibilizar gratuitamente o documento em formato digital no Repositório Institucional do IF Goiano (RIIF Goiano), sem ressarcimento de direitos autorais, conforme permissão assinada abaixo, para fins de leitura, download e impressão, a título de divulgação da produção técnico-científica no IF Goiano.

IDENTIFICAÇÃO DA PRODUÇÃO TÉCNICO-CIENTÍFICA

- ☐ Tese (doutorado)
- ☐ Dissertação (mestrado)
- ☐ Monografia (especialização)
- ☑ TCC (graduação)
- ☐ Artigo científico
- ☐ Capítulo de livro
- ☐ Livro
- ☐ Trabalho apresentado em evento
- ☐ Produto técnico e educacional - Tipo:

Nome completo do autor:
Felipe Nário de Souza Fontinelli

Matrícula:
2018104211710140

Título do trabalho:
DoeBrasil.org: Plataforma para Intermediação de Doações

RESTRIÇÕES DE ACESSO AO DOCUMENTO

Documento confidencial: ☑ Não ☐ Sim, justifique:

Informe a data que poderá ser disponibilizado no RIIF Goiano: ☐/☐/☐

O documento está sujeito a registro de patente? ☐ Sim ☑ Não

O documento pode vir a ser publicado como livro? ☐ Sim ☑ Não

DECLARAÇÃO DE DISTRIBUIÇÃO NÃO-EXCLUSIVA

O(a) referido(a) autor(a) declara:

• Que o documento é seu trabalho original, detém os direitos autorais da produção técnico-científica e não infringe os direitos de qualquer outra pessoa ou entidade;

• Que obteve autorização de quaisquer materiais inclusos no documento do qual não detém os direitos de autoria, para conceder ao Instituto Federal de Educação, Ciência e Tecnologia Goiano os direitos requeridos e que este material cujos direitos autorais são de terceiros, estão claramente identificados e reconhecidos no texto ou conteúdo do documento entregue;

• Que cumpriu quaisquer obrigações exigidas por contrato ou acordo, caso o documento entregue seja baseado em trabalho financiado ou apoiado por outra instituição que não o Instituto Federal de Educação, Ciência e Tecnologia Goiano.

RIO QUENTE — Local
02 /01 /2024 — Data

gov.br Documento assinado digitalmente
FELIPE NARIO DE SOUZA FONTINELLI
Data: 04/01/2024 13:56:18-0300
Verifique em https://validar.iti.gov.br

Assinatura do autor e/ou detentor dos direitos autorais

Ciente e de acordo:

Assinatura do(a) orientador(a)

gov.br Documento assinado digitalmente
MARCEL DA SILVA MELO
Data: 10/01/2024 16:19:47-0300
Verifique em https://validar.iti.gov.br

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA GOIANO

Ata nº 27/2023 - CCEPTNM-MO/CEPTNM-MO/DE-MO/CMPMHOS/IFGOIANO

ATA DE DEFESA DE TRABALHO DE CURSO

Ao(s) **28** dia(s) do mês de Setembro de 2023, às 19 horas e 30 minutos, reuniu-se a banca examinadora composta pelos docentes: Marcel da Silva Melo (orientador), Odilon Fernandes Neto (membro), José Pereira Alves (membro), para examinar o Trabalho de Curso intitulado "**DOEBRASIL.ORG: PLATAFORMA PARA INTERMEDIAÇÃO DE DOAÇÕES** " do(a) estudante Felipe Nário de Souza Fontinelli, Matrícula nº 2018104211710140 do Curso de Tecnologia em Sistemas para Internet do IF Goiano – Campus Morrinhos. A palavra foi concedida ao(a) estudante para a apresentação oral do TC, houve arguição do(a) candidato pelos membros da banca examinadora. Após tal etapa, a banca examinadora decidiu pela APROVAÇÃO do(a) estudante. Ao final da sessão pública de defesa foi lavrada a presente ata que segue assinada pelos membros da Banca Examinadora.

(Assinado Eletronicamente)

Marcel da Silva Melo

Orientador(a)

(Assinado Eletronicamente)

Odilon Fernandes Neto

Membro

(Assinado Eletronicamente)

José Pereira Alves

Membro

Documento assinado eletronicamente por:

- **Felipe Nario de Souza Fontinelli, 2018104211710140 - Discente**, em 05/10/2023 23:44:10.
- **Jose Pereira Alves, PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 04/10/2023 22:09:29.
- **Odilon Fernandes Neto, PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 04/10/2023 08:33:38.
- **Marcel da Silva Melo, PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 03/10/2023 20:36:45.

Este documento foi emitido pelo SUAP em 28/09/2023. Para comprovar sua autenticidade, faça a leitura do QRCode ao lado ou acesse https://suap.ifgoiano.edu.br/autenticar-documento/ e forneça os dados abaixo:

Código Verificador: 535391
Código de Autenticação: 97030d318d

INSTITUTO FEDERAL GOIANO
Campus Morrinhos
Rodovia BR-153, Km 633, Zona Rural, SN, Zona Rural, MORRINHOS / GO, CEP 75650-000
(64) 3413-7900

FELIPE NÁRIO DE SOUZA FONTINELLI

DOEBRASIL.ORG: PLATAFORMA PARA INTERMEDIAÇÃO DE DOAÇÕES

Data da defesa:

Resultado: _____

BANCA EXAMINADORA

_____

Profº Me. Marcel da Silva Melo (Presidente da banca)
Instituto Federal Goiano campus Morrinhos

_____

Profº Odilon Fernandes Neto
Instituto Federal Goiano campus Morrinhos

_____

Profº José Pereira Alves
Instituto Federal Goiano campus Morrinhos

**MORRINHOS -
GO 2023**

AGRADECIMENTOS

Gostaria de começar agradecendo a minha mãe, que esteve ao meu lado em todos os momentos durante essa jornada de conclusão de curso. Sua dedicação e apoio incondicional foram fundamentais para que eu pudesse chegar até aqui. Sua força e coragem sempre me inspiraram a perseguir meus objetivos e acreditar no meu potencial.

Além disso, gostaria de agradecer a todos os professores que me ajudaram ao longo do curso. O conhecimento e a orientação que eles me proporcionaram foram essenciais para o meu desenvolvimento pessoal e profissional. Em especial, gostaria de agradecer ao professor Marcel Melo, que me deu a oportunidade de trabalhar ao seu lado e com outros colegas na criação do aplicativo DoeBrasil.

O DoeBrasil é um projeto que tem um grande valor social e que, graças ao professor Marcel Melo e ao nosso grupo, pude colaborar na sua concepção e desenvolvimento. Esse aplicativo é uma ferramenta muito importante para conectar pessoas que precisam de ajuda com aquelas que desejam ajudar, seja por meio de doações ou de voluntariado. É uma grande honra poder apresentá-lo como parte do meu trabalho de conclusão de curso.

Mais uma vez, gostaria de expressar minha gratidão à minha mãe e aos professores que me apoiaram e me guiaram nessa jornada. Sem vocês, eu não teria chegado até aqui. Sou imensamente grato por tudo o que fizeram por mim.

A pandemia de COVID-19, causada pelo coronavírus SARS-CoV-2, impactou profundamente a sociedade em todo o mundo. Além das consequências sanitárias, a crise econômica resultante levou milhares de pessoas a perderem sua única fonte de renda familiar, agravando ainda mais a situação de vulnerabilidade social. Nesse contexto, o projeto DoeBrasil.org surge como uma solução digital inovadora para facilitar e agilizar o processo de doação de alimentos e produtos de primeira necessidade às famílias carentes. Esta monografia tem como objetivo apresentar o projeto DoeBrasil.org, uma plataforma de intermediação de doações desenvolvida durante a pandemia de COVID-19. Através de um ambiente digital, a plataforma conecta doadores, donatários e entidades parceiras, permitindo a realização de solicitações de doação e a oferta de ajuda de forma eficiente e segura.

Palavras-chave: *Flutter , Dart , Mobx , Modular , API Rest , Covid , Doação , Aplicativo.*

1. INTRODUÇÃO

A pandemia causada pelo COVID-19 trouxe muitos desafios para a sociedade, incluindo uma crise financeira para muitas pessoas e instituições. Neste contexto, as doações têm se tornado uma importante fonte de auxílio para aqueles que mais precisam. Com o objetivo de facilitar o processo de doação e torná-lo mais acessível, foi desenvolvido um aplicativo em Flutter que possibilita aos usuários doar de forma rápida e segura. Este trabalho de conclusão de curso apresenta a criação do aplicativo, destacando suas funcionalidades, processo de desenvolvimento e sua importância na atual conjuntura.

1.1 TECNOLOGIAS UTILIZADAS

1.1.1 Dart

Dart é a linguagem de programação utilizada para desenvolver aplicativos em Flutter. Ela foi criada pela Google em 2011 e tem sido amplamente adotada por desenvolvedores em todo o mundo. (DART, 2023)

De acordo com o criador da linguagem, Lars Bak, o objetivo do Dart era criar "uma linguagem de programação para a próxima década". Ele acreditava que as linguagens de programação existentes, como Java e JavaScript, não eram adequadas para lidar com a complexidade das aplicações modernas. (BACON, John et al. The Dart Programming Language).

Dart é uma linguagem de tipagem estática, o que significa que o tipo de cada variável é definido em tempo de compilação. Isso permite que os erros sejam identificados mais cedo no processo de desenvolvimento e melhora a segurança e a confiabilidade do código. (DART, 2023)

Além disso, Dart possui recursos que tornam a programação mais fácil e eficiente. Por exemplo, ela oferece suporte a programação assíncrona, que permite que várias tarefas sejam executadas em paralelo sem bloquear o thread principal. Isso é especialmente útil em aplicativos móveis, onde a resposta rápida do usuário é fundamental. (DART, 2023)

Dart também possui uma sintaxe limpa e fácil de entender, o que a torna acessível para desenvolvedores iniciantes. Ela possui uma curva de aprendizado suave e muitos recursos disponíveis para ajudar os desenvolvedores a aprender e aprimorar suas habilidades em Dart. (DART, 2023)

De acordo com a documentação oficial da linguagem Dart, "Dart é uma linguagem orientada a objetos com classes e interfaces, herança, polimorfismo e outras características familiares a outras linguagens orientadas a objetos." (DART, 2023)

1.1.2 Flutter

De acordo com a documentação do Flutter, ele é um framework de desenvolvimento de aplicativos móveis multiplataforma, criado pelo Google em 2017, que tem ganhado bastante popularidade na comunidade de desenvolvedores (Google, 2022).

De acordo com Liu et al. (2019), o Flutter utiliza a linguagem de programação Dart, que é mais fácil de aprender do que outras linguagens de programação utilizadas para o desenvolvimento de aplicativos móveis, como Java e Swift. Além disso, Flutter tem a vantagem de ser multiplataforma, o que significa que é possível desenvolver um aplicativo que funcione tanto em dispositivos Android quanto iOS. Segundo Liu et al. (2019), o Flutter tem se mostrado uma opção promissora para o desenvolvimento de aplicativos móveis, especialmente para pequenas e médias empresas.

Outra vantagem do Flutter é a sua arquitetura de *widgets*, que permite o desenvolvimento de interfaces de usuário ricas e responsivas. Segundo Felchlin et al. (2020), a arquitetura de *widgets* do Flutter é baseada em uma abordagem de "tudo é widget", o que significa que todos os elementos da interface de usuário, como botões, caixas de texto e imagens, são *widgets*. Isso permite que o desenvolvedor construa uma interface de usuário complexa combinando *widgets* simples.

O Flutter também tem uma grande comunidade de desenvolvedores e suporte da Google, o que torna o aprendizado e o desenvolvimento mais fácil. Segundo Felchlin et al. (2020), a comunidade de desenvolvedores do Flutter é ativa e há muitos recursos online para aprendizado e resolução de problemas.

O Flutter é uma opção promissora para o desenvolvimento de aplicativos móveis multiplataforma. Sua linguagem de programação fácil de aprender, arquitetura de widgets e comunidade ativa de desenvolvedores são algumas de suas principais vantagens. (Google, 2022)

O Flutter é conhecido por sua capacidade de criar aplicativos com alta performance e rapidez. Segundo a pesquisa realizada por Chan e Chong (2020), o Flutter apresentou tempos de execução mais rápidos em comparação com outros frameworks de desenvolvimento móvel. Os autores também destacaram a facilidade

de desenvolvimento de aplicativos móveis com Flutter, especialmente para desenvolvedores que estão familiarizados com a programação orientada a objetos.

Uma das características mais interessantes do Flutter é o *hot reload*, que permite aos desenvolvedores visualizar rapidamente as mudanças feitas em seu código na interface do aplicativo. De acordo com Reyes (2021), o *hot reload* é uma das principais razões pelas quais desenvolvedores preferem o Flutter para o desenvolvimento de aplicativos móveis. O *hot reload* torna o processo de desenvolvimento mais ágil e eficiente.

O Flutter também é capaz de criar interfaces de usuário altamente personalizáveis. Segundo a pesquisa realizada por Han e Wang (2021), o Flutter tem uma grande variedade de *widgets* pré-construídos que podem ser usados para criar interfaces de usuário complexas e personalizadas. Além disso, permite a criação de widgets personalizados, o que oferece uma maior flexibilidade no *design* da interface de usuário.

Em resumo, o Flutter oferece uma série de vantagens para o desenvolvimento de aplicativos móveis, incluindo sua facilidade de uso, alta performance, hot reload e capacidade de criar interfaces de usuário personalizadas.

1.1.3 Arquitetura MVC (*Model - View - Controller*)

Segundo Burbeck (1992) a arquitetura MVC (*Model-View-Controller*) é um padrão de projeto de software que separa a lógica de negócios da aplicação da interface do usuário, permitindo uma maior flexibilidade e manutenção do código.

De acordo com Burbeck (1992), "o padrão MVC separa os objetos de domínio do usuário da interface do usuário, permitindo que os desenvolvedores trabalhem com cada um independentemente do outro." Isso significa que o modelo (*Model*) contém a lógica de negócios da aplicação, enquanto a visão (*View*) representa a interface do usuário e o controlador (*Controller*) faz a mediação entre os dois.

Para Fowler et al. (2002), "a ideia principal do MVC é ter uma separação clara entre a lógica da aplicação e a interface do usuário.". Essa separação permite que diferentes visões sejam criadas a partir do mesmo modelo, tornando a aplicação mais flexível e fácil de manter. Além disso, o controlador atua como um intermediário entre o modelo e a visão, permitindo que a comunicação entre eles

seja mais clara e organizada.

Segundo Gamma et al. (1995), "o padrão MVC separa o comportamento do usuário em três tipos de objetos: o modelo, a visão e o controlador." Essa separação permite que cada objeto possa ser modificado independentemente dos outros, tornando a aplicação mais modular e fácil de manter. Além disso, o controlador pode ser reutilizado em diferentes visões, aumentando a eficiência do código.

Por fim, para Leff e Rayfield (2001), "o padrão MVC fornece uma estrutura flexível para a construção de aplicativos orientados a objetos que suportam interfaces gráficas de usuário." Essa estrutura permite que os desenvolvedores possam criar aplicativos mais robustos e flexíveis, com uma arquitetura clara e organizada.

1.1.4 API REST

Uma API REST (*Application Programming Interface Representational State Transfer*) é uma interface de programação de aplicações que permite a comunicação entre diferentes sistemas de software através de protocolos HTTP (*Hypertext Transfer Protocol*) e JSON (*JavaScript Object Notation*). É uma das arquiteturas mais utilizadas para o desenvolvimento de APIs, devido à sua simplicidade e eficiência.

De acordo com a definição do próprio criador do termo, Roy Fielding,"Uma API Rest é um sistema distribuído que permite que recursos sejam identificados e manipulados por meio de um conjunto padronizado de operações baseadas no protocolo HTTP".

Uma API REST é composta por recursos, que podem ser qualquer coisa que possa ser identificada e manipulada pela aplicação, como dados, objetos ou funcionalidades. Esses recursos são acessados através de URLs (*Uniform Resource Locators*) que representam cada recurso (Martin, 2020).

Além disso, as operações que podem ser realizadas em cada recurso são padronizadas e representadas pelos verbos HTTP, como GET, POST, PUT e DELETE. Isso torna a API REST fácil de entender e utilizar, além de permitir que diferentes sistemas se comuniquem de forma transparente (Martin, 2020).

As APIs Rest são amplamente utilizadas em aplicações web e *mobile*, como forma de permitir que diferentes sistemas se comuniquem entre si e acessem informações de forma segura e eficiente. Elas também são amplamente utilizadas em aplicações de Internet das Coisas (IoT) e em sistemas de integração de dados (Martin, 2020).

1.1.5 MobX

O MobX é uma biblioteca de gerenciamento de estado reativo que pode ser usada em várias plataformas, incluindo Flutter. Segundo a documentação oficial do MobX (2023), ele é uma solução poderosa e fácil de usar para gerenciar o estado do aplicativo em plataformas móveis, web e *desktop*.

No contexto do Flutter, o MobX é uma das opções de gerenciamento de estado mais populares entre os desenvolvedores (Schutz, 2021). Segundo a documentação do Flutter (2023), o MobX é uma das ferramentas recomendadas para gerenciamento de estado em aplicativos Flutter, juntamente com o Provider e outras bibliotecas.

O MobX para Flutter funciona com três principais elementos: observáveis, ações e reações. De acordo com a documentação oficial do MobX (2023), as observáveis representam o estado do aplicativo, as ações modificam o estado e as reações atualizam a interface do usuário em resposta às mudanças nas observáveis.

Um dos principais benefícios do MobX para Flutter é a sua capacidade de simplificar o gerenciamento de estado e torná-lo mais fácil de entender e manter. Segundo Nunes (2021), o MobX ajuda a reduzir a complexidade do código, permitindo que os desenvolvedores se concentrem na lógica de negócios do aplicativo.

Além disso, o MobX para Flutter é altamente escalável e pode ser usado em projetos de todos os tamanhos. De acordo com Schutz (2021), o MobX é uma das bibliotecas mais populares para gerenciamento de estado em aplicativos Flutter, e é frequentemente escolhido por desenvolvedores que desejam um código mais

organizado e fácil de manter.

Por fim, o MobX para Flutter também oferece uma série de recursos avançados, como a capacidade de reutilizar a lógica de negócios em diferentes partes do aplicativo e a integração com outras bibliotecas importantes do Flutter, como o Provider (MobX, 2023). De acordo com a documentação oficial do MobX (2023), ele é altamente customizável e pode ser ajustado para atender às necessidades de diferentes projetos e equipes de desenvolvimento.

Em resumo, o MobX é uma biblioteca poderosa e fácil de usar para o gerenciamento de estado em aplicativos Flutter. Ele oferece uma abordagem reativa para gerenciar o estado do aplicativo e ajuda a reduzir a complexidade do código, tornando mais fácil a manutenção do aplicativo a longo prazo. Além disso, o MobX para Flutter é altamente escalável e oferece uma série de recursos avançados que podem ser personalizados para atender às necessidades específicas de diferentes projetos e equipes de desenvolvimento.

1.1.6 Hive

O pacote Hive é uma biblioteca de gerenciamento de banco de dados NoSQL para o Flutter, que oferece um sistema de armazenamento local rápido, eficiente e fácil de usar. O Hive foi projetado para ser extremamente rápido, utilizando gravação em memória para melhorar a performance e implementando um modelo de dados baseado em caixas (boxes), que oferece alta flexibilidade e escalabilidade (Hive, 2022).

De acordo com a documentação oficial do Hive (Hive, 2022), "Hive é uma escolha perfeita para aplicativos Flutter que exigem persistência de dados, como aplicativos de notas, aplicativos de lista de tarefas e muito mais" . Além disso, a documentação também destaca que o Hive é uma opção mais rápida e eficiente do que outras bibliotecas de gerenciamento de banco de dados disponíveis para o Flutter.

Em um artigo publicado no Medium (Alnemr, 2021), o autor Youssef Alnemr descreve a experiência de usar o Hive em um projeto Flutter e destaca que

a biblioteca oferece uma sintaxe simples e intuitiva para trabalhar com dados armazenados localmente. Ele também menciona a velocidade do Hive, afirmando que "o Hive é realmente rápido e tem um desempenho excelente em todas as operações de leitura e gravação" (Alnemr, 2021).

Outro autor, Brandon Donnelson, em um artigo publicado no seu blog pessoal (Donnelson, 2021), destaca a capacidade do Hive de trabalhar com grandes quantidades de dados sem comprometer a performance, além da facilidade de uso do Hive em comparação com outras bibliotecas. Ele afirma que "o Hive é muito fácil de usar e é capaz de lidar com grandes quantidades de dados com facilidade, sem perder a eficiência" (Donnelson, 2021).

1.1.7 Provider

De acordo com a documentação, o Provider é uma opção popular para gerenciamento de estado em aplicativos Flutter, que utiliza o padrão de Injeção de Dependência (*Dependency Injection*) para fornecer um estado global para o aplicativo. O objetivo principal do Provider é fornecer uma maneira simples de compartilhar objetos entre diferentes partes do aplicativo sem a necessidade de passá-los manualmente de um *widget* para outro. (PROVIDER, 2023)

A Injeção de Dependência é um padrão de *design* que permite criar um objeto em um local centralizado e injetá-lo em outros objetos conforme necessário. Isso torna o código mais modular e fácil de entender, permitindo que diferentes partes do código sejam modificadas independentemente uma da outra (PROVIDER, 2023).

O Provider utiliza a Injeção de Dependência para fornecer um estado global para o aplicativo. Ele cria um objeto de estado centralizado e o injeta em diferentes partes do código, conforme necessário, tornando o código mais modular e fácil de entender.

O Provider fornece uma maneira fácil e intuitiva de injetar dependências em qualquer lugar da árvore de widgets do aplicativo. Ele usa uma abordagem baseada em widgets para criar um objeto compartilhado e, em seguida, fornece

acesso a esse objeto por meio de widgets específicos.

1.1.8 Flutter Modular

O flutter_modular é um pacote para o desenvolvimento de aplicativos Flutter que fornece recursos de injeção de dependência e roteamento. Ele foi criado com o objetivo de facilitar a organização e a estruturação do código dos aplicativos Flutter, permitindo uma maior reutilização de código (FLUTTERANDO, 2023).

De acordo com a documentação do flutter_modular, "A ideia principal do Flutter Modular é oferecer uma solução para o desenvolvimento de aplicativos escaláveis e organizados, onde o código seja reutilizável, testável e facilmente mantido." (FLUTTERANDO, 2023).

O flutter_modular permite que os desenvolvedores dividam seus aplicativos em módulos independentes, cada um com sua própria estrutura e conjunto de dependências. Isso torna o código mais organizado e legível, além de permitir que diferentes partes do aplicativo sejam desenvolvidas de forma independente.

O flutter_modular usa um sistema de injeção de dependência para gerenciar as dependências entre os diferentes módulos. Isso permite que os desenvolvedores usem diferentes formas de injeção de dependência, como a injeção manual, a injeção automática e a injeção baseada em anotações.

Além disso, o flutter_modular fornece recursos de roteamento, permitindo que os desenvolvedores definam rotas nomeadas para cada módulo e gerenciem a navegação de maneira simples e intuitiva.

O flutter_modular também inclui recursos para a criação de *widgets* reutilizáveis, que podem ser usados em vários módulos do aplicativo. Isso permite que os desenvolvedores criem *widgets* personalizados de maneira mais fácil e rápida, economizando tempo e esforço.

2 DESENVOLVIMENTO

2.1 CONSIDERAÇÕES INICIAIS

O Doe Brasil é um projeto de plataforma digital com o objetivo de conectar doadores, donatários e entidades parceiras através de um ambiente online. A primeira versão foi lançada em maio de 2020 durante o período em que o Brasil registrava um grande número de casos de COVID-19. A ideia principal do DoeBrasil é a de tornar o processo de doação mais fácil e eficiente, simplificando a interação entre doadores e entidades que recebem as doações. A plataforma foi projetada para funcionar em qualquer cidade brasileira e inicialmente oferecia duas formas de doação: direta e indireta. Na doação direta, os doadores entregam os itens solicitados diretamente ao solicitante, enquanto na doação indireta, o doador pode entregar os itens em um ponto de entrega cadastrado pela entidade parceira.

Após o lançamento da plataforma, foi realizada uma análise de uso e constatou-se que muitos usuários e parceiros cadastraram-se, mas não realizaram ações na plataforma. Foi então agendada uma reunião com os parceiros cadastrados para discutir possíveis melhorias na plataforma. Entre as sugestões propostas pelos parceiros, destacam-se a possibilidade de cadastro de usuários, solicitações e doações pelos parceiros, a retirada da funcionalidade de doação direta e a geração de relatórios com estatísticas detalhadas.

Após ampla discussão entre a equipe de desenvolvimento, decidiu-se adaptar toda a plataforma para atender às necessidades dos parceiros.

2.2 ARQUITETURA INICIAL DA APLICAÇÃO

A estrutura do aplicativo foi desenvolvida utilizando padrão MVC com uma estrutura modularizada por escopo, onde cada módulo da aplicação tem sua própria estrutura MVC (Figura 01). Os dois principais pacotes considerados *core* do projeto

são o Flutter Modular, utilizado para injeção de dependência e gerenciamento de rotas e o MobX para a gerência de estado da aplicação.

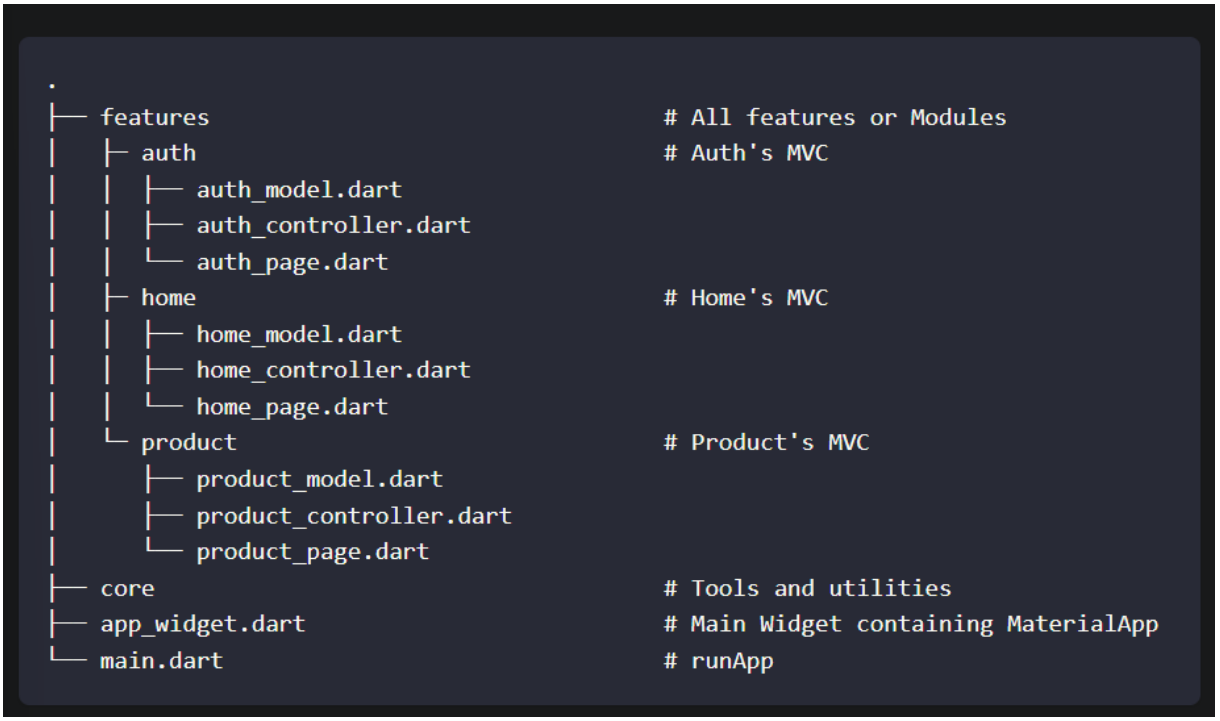

Figura 01: Estrutura Modular MVC.
Fonte: Retirado da documentação do Flutter Modular
(https://modular.flutterando.com.br/docs/intro/)

O Flutter Modular traz consigo um padrão que visa modularizar a nossa aplicação, dividindo melhor o escopo de features, trazendo uma modularidade para aplicação.

Um bom exemplo de como é utilizado o Flutter Modular é na Figura 02, onde é recuperada a instância *singleton* do MinhasDoacoesController, uma Store do MobX que é responsável pela gerência do estado do módulo de Minhas Doações, tendo sua instância criada assim que o usuário acessa o módulo de Minhas Doações. Essa instância é mantida enquanto o usuário estiver utilizando este módulo e é destruída (junto com qualquer outra instância desse módulo) assim que o usuário sai dele.

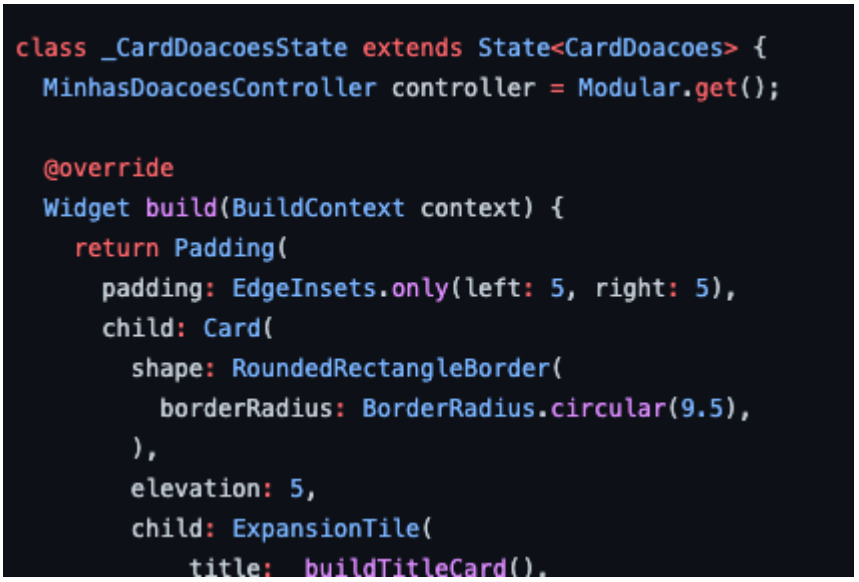

Figura 02: Código do *Widget* de *Card*
Fonte: criado pelo próprio autor

2.3 PROCESSO DE DESENVOLVIMENTO

O desenvolvimento teve início pela tela de cadastro onde o usuário faz seu cadastro no aplicativo, Figura 03. Nesta tela o usuário irá se deparar com os campos necessários para seu cadastro. Todos os campos possuem ícones para melhor identificação, como por exemplo o número de telefone com WhatsApp ou como o campo dinâmico de cidade carregado a partir do estado selecionado pelo usuário. Além disso, todos os campos possuem algum tipo de máscara (como telefone), facilitando o preenchimento por parte do usuário. O cadastro em si não possui muitos campos e é bem fácil e intuitivo para o usuário. Após o preenchimento de todos os campos obrigatórios basta o usuário clicar no botão de cadastrar ao final da tela para finalizar o seu cadastro na aplicação.

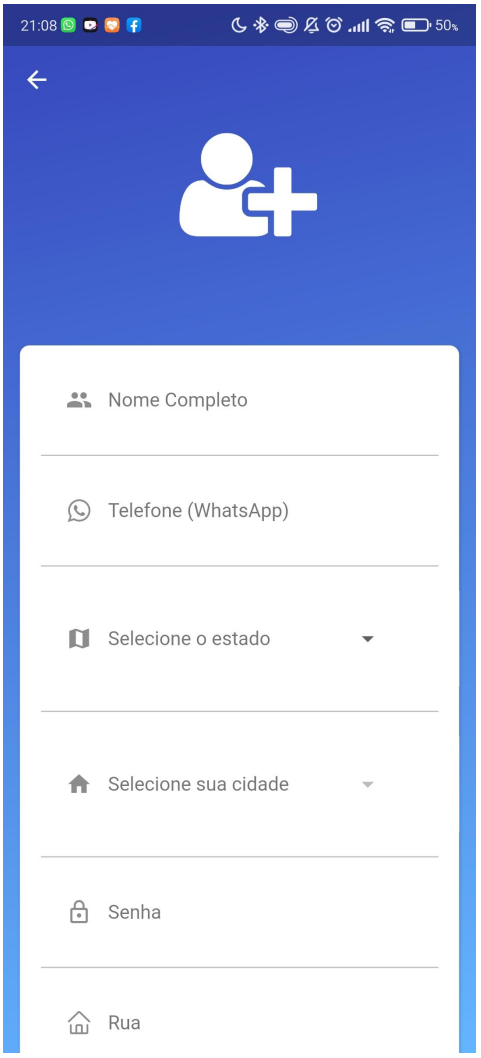

Figura 03: Tela de cadastro
Fonte: criado pelo próprio autor

A construção da paleta de cores do DoeBrasil foi iniciada pela escolha da cor primária, que neste caso foi escolhido o Azul. Esta cor, que é a cor predileta de grande porção das pessoas, além de trazer uma sensação de simpatia, harmonia e fidelidade, e culturalmente ser a cor da “Humanidade” (Eiseman, L. 2017). Além da cor principal também foi necessário definir pelo menos uma cor secundária para construir uma paleta de cores tanto para a construção das interfaces do aplicativo quanto para a logo. Esta cor deveria ser uma cor com um bom contraste com a cor primária.

Depois de um tempo de pesquisa foi decidido por utilizar a cor Branca, uma cor que além de contrastar bem com o Azul, também tem o significado de cor do bem e da perfeição. Com as cores definidas, foi criado um degradê utilizado para um toque mais moderno nas telas.

As funções do aplicativo somente estão disponíveis após o usuário fazer o cadastro e login. As telas foram desenvolvidas pensando em simplicidade e

facilidade de acesso para cada usuário. As telas, onde são apresentadas as informações das doações, foram divididas em sessões para melhor controle do usuário durante o uso, sendo elas: Home, Meus Pedidos e Minhas Doações, conforme a Figura 04.

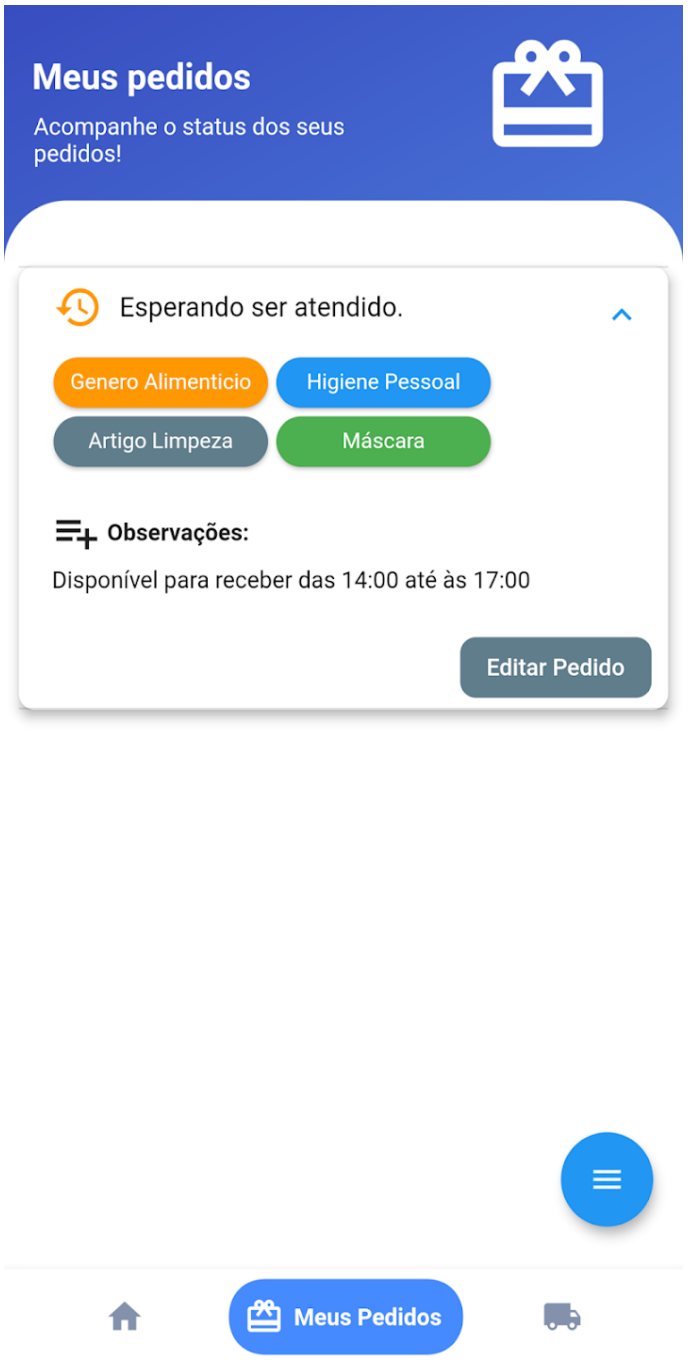

Figura 04: Tela de Meus Pedidos.
Fonte: criado pelo próprio autor

Durante o desenvolvimento, foram enfrentadas dificuldades com a injeção de dependência utilizando o Modular. Estas dificuldades se deram, não por problemas com o *Package* em si, mas sim por falta de experiência com o mesmo.

Além disso, na época em que foi iniciado o projeto do DoeBrasil, o Modular tinha apenas um mês de lançamento de sua versão estável. Portanto, ainda havia muito pouco material para pesquisa, sendo a maior parte dele disponibilizada pela Flutterando, a comunidade brasileira que o desenvolveu (https://flutterando.com.br).

Um ponto onde foi enfrentado diversos problemas foi a manutenção do login do usuário, mesmo após o encerramento do aplicativo. Onde para que o usuário não tenha que logar toda vez que abrir o aplicativo, uma função de “autorun” do Modular, Figura 05, seria executada durante a SplashScreen do aplicativo para fazer a checagem se o usuário está logado ou não.

```dart
@override
void initState() {
  super.initState();
  disposer = autorun((_){
    final login = Modular.get<LoginController>();
    if(login.status == AuthStatus.login) {
      Future.delayed(Duration(seconds: 2), (){
        Modular.to.pushReplacementNamed('/home');
      });  // Future.delayed
    }else if(login.status == AuthStatus.logoff){
       Future.delayed(Duration(seconds: 2), (){
        Modular.to.pushReplacementNamed('/login');
      });  // Future.delayed
    }});
}
```

Figura 05: Código da função *autorun*
Fonte: criado pelo próprio autor

A função, que é responsável pela checagem do Token, Figura 06, e que ativa a função de autorun, é chamada assim que o LoginController é criado. Nela o valor da variável de AuthStatus presente no controller é alterado caso o usuário já tenha feito login ou não. Após essa mudança de *status* a função de *autorun* é executada fazendo o redirecionamento para a rota correta de acordo com o AuthStatus.

```
@action
getToken() async{
  tokenPersistence = await _localStorage.getToken("token").then((tokenPersistence){
    if (tokenPersistence != null) {
      status = AuthStatus.login;
    }else{
      status = AuthStatus.logoff;
    }
    return tokenPersistence;
  });
}
```

Figura 06: Código da função getToken
Fonte: criado pelo próprio autor

Outro problema foi a alta complexidade e a verbosidade para um projeto tão simples. Conforme apresentado na Figura 07, o Modular traz uma modularidade para a aplicação, onde cada módulo é formado por *Page*(s), um Module e Controller(s). Esta modularidade é extremamente valiosa para projetos maiores e que são mantidos por mais de um desenvolvedor, o que não era o caso do DoeBrasil. Essa complexidade e verbosidade somada a pouca experiência com o *package* levava a atrasos na implementação de novas funcionalidades.

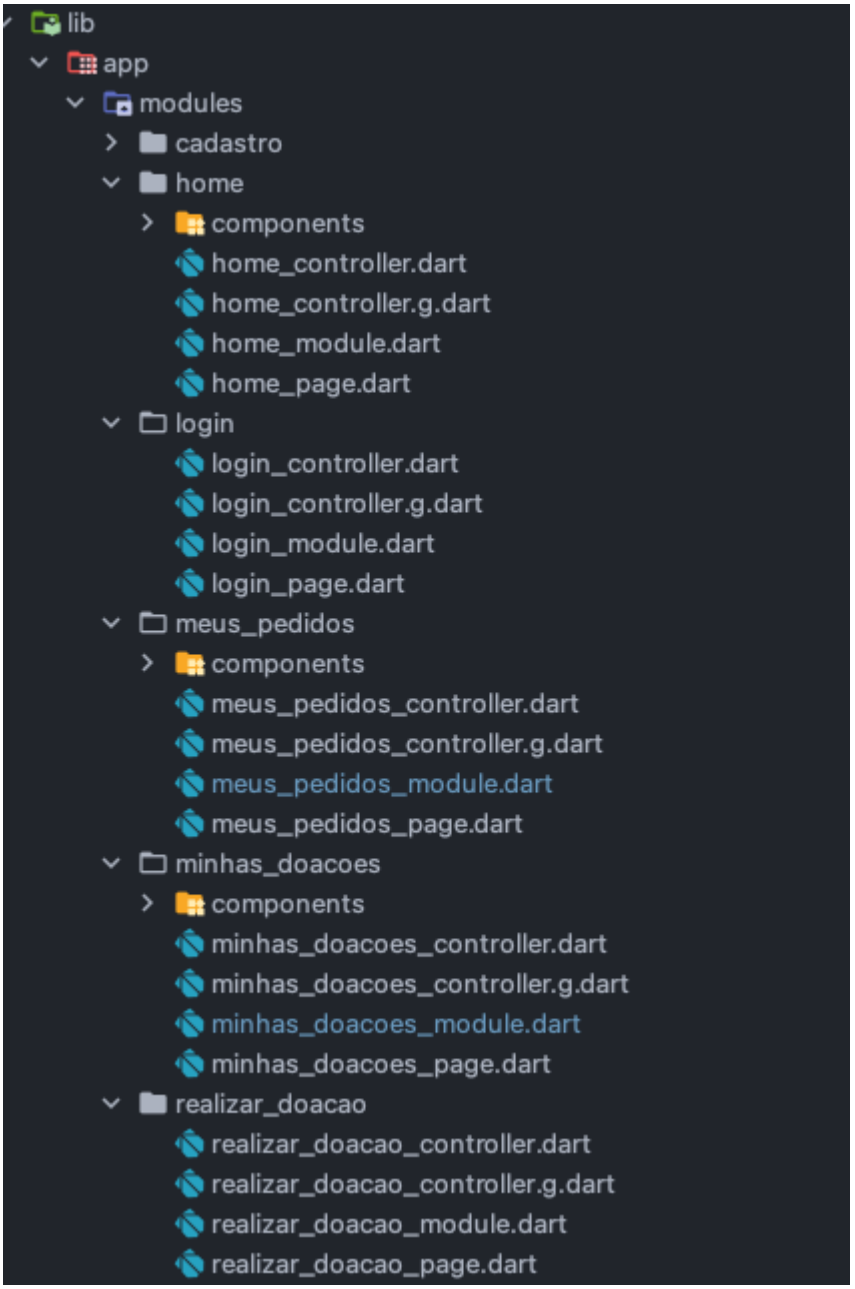

Figura 07: Estrutura de pastas(modularizada) do projeto
Fonte: criado pelo próprio autor

Além dos módulos específicos (como Home, Login e etc) também há os que são compartilhados, ou *shared*, apresentados na Figura 08. Nestes módulos são localizadas as classes de modelos (*models*) e os *utils* que contém variáveis e funções que serão reutilizadas em toda aplicação. Tanto os modelos quanto às funções e variáveis do *utils* são compartilhados entre vários módulos e por isso estão no *shared*.

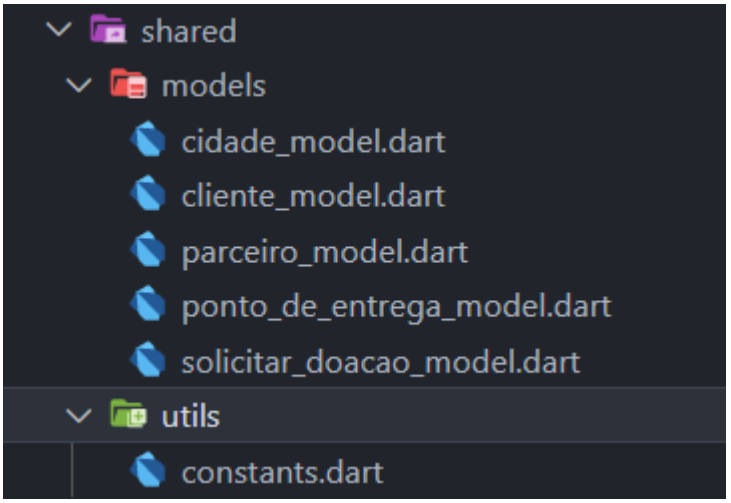

Figura 08: Módulo *Shared*
Fonte: criado pelo próprio autor

Há também a pasta de *repositories*, apresentada na Figura 09. Nesta pasta estão centralizadas as chamadas do LocalStorage e chamadas para API que vão ser utilizadas quando necessárias nos controllers, apresentadas na Figura 10.

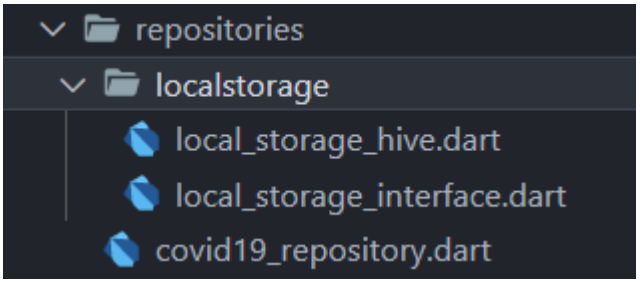

Figura 09: *Repositories* do projeto
Fonte: criado pelo próprio autor

```
@action
fetchParceiros() async {
  var token = await localStorage.getToken("token");
  parceiros = await repository.getParceiros(token);
  adicionarParceirosDrop();
}

@action
fetchPontosDeEntrega(String parceiroSelecionado) async {
  var token = await localStorage.getToken("token");
  pontosDeEntrega =
      await repository.getPontosDeEntrega(token, parceiroSelecionado);
}
```

Figura 10: Funções que utilizam as funções presentes nos *repositories*
Fonte: criado pelo próprio autor

Acima de todos os módulos fica o AppModule, apresentado na Figura 11, que encapsula todos os outros módulos e, portanto, todos os módulos passam pelo AppModule. Nele também é feito o *bind* de *Singletons*, *Lazy Singletons*, *Factories* de Controllers, Repositories e etc, e tem o objetivo de serem disponibilizados para uso por todos os módulos da aplicação, conforme bem estruturado na Figura 12.

```dart
class AppModule extends MainModule {
  @override
  List<Bind> get binds => [
        Bind((i) => AppController()),
        Bind((i) => Dio(BaseOptions(baseUrl: URL_BASE))),
        Bind<ILocalStorage>((i) => LocalStorageHive()),
        Bind((i) => MinhasDoacoesController(i.get<CovidAPI>(), i.get<ILocalStorage>())),
        Bind((i) => MeusPedidosController(i.get<CovidAPI>(), i.get<ILocalStorage>())),
        Bind((i) => LoginController(i.get<CovidAPI>())),
        Bind((i) => CovidAPI(i.get<Dio>())),
      ];

  @override
  List<Router> get routers => [
        Router('/', child: (_, args) => SplashPage()),
        Router('/login', module: LoginModule(), transition: TransitionType.rightToLeft),
        Router('/home', module: HomeModule(), transition: TransitionType.rightToLeft),
        Router('/cadastro', module: CadastroModule(), transition: TransitionType.rightToLeft),
        Router('/realizarDoacao', module: RealizarDoacaoModule(), transition: TransitionType.rightToLeft),
        Router('/solicitarDoacao', module: SolicitarDoacaoModule(), transition: TransitionType.rightToLeft),
        Router('/meusPedidos', module: MeusPedidosModule()),
        Router('/minhasDoacoes', module: MinhasDoacoesModule())
      ];

  @override
  Widget get bootstrap => AppWidget();

  static Inject get to => Inject<AppModule>.of();
}
```

Figura 11: Binds e rotas no *AppModule*
Fonte: criado pelo próprio autor

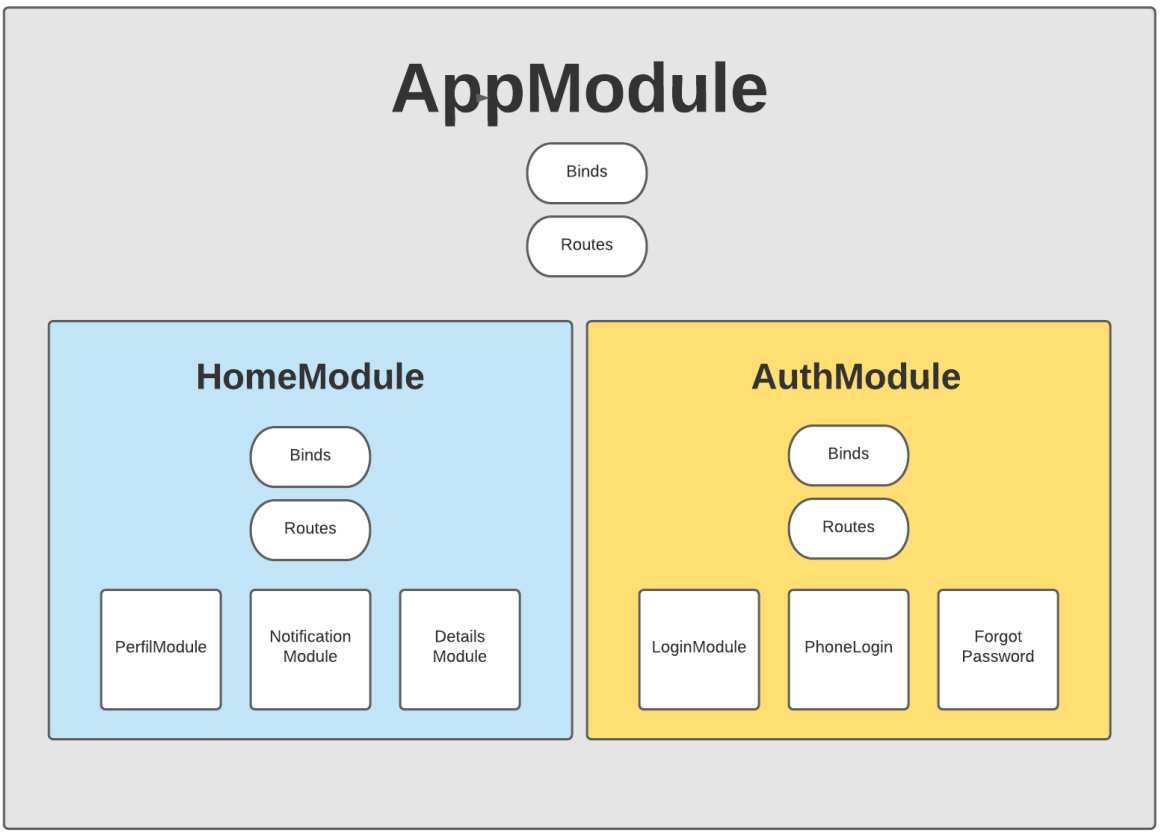

Figura 12: Estrutura de módulos do Modular
Fonte: Retirada do blog da Flutterando no post de anúncio do Flutter Modular 3.0, https://blog.flutterando.com.br/announcing-flutter-modular-3-0-beta-with-null-safety-b0a0e13f67b6.

O Modular também trouxe uma facilidade nas animações de transições de tela personalizadas fornecidas pelo próprio package, conforme apresentado na Figura 13. Outra função extremamente útil disponibilizada pelo Modular é o dispose (ou remover) automático do módulo e seus recursos automaticamente. O Modular faz isso para instâncias que não são mais necessárias, por exemplo ao sair da tela de login todo o módulo de login com seus Controllers são fechados. Essa funcionalidade é de extrema importância para gerenciar memória e processamento consumido por parte do app.

```
Router('/login', module: LoginModule(), transition: TransitionType.rightToLeft),
Router('/home', module: HomeModule(), transition: TransitionType.rightToLeft),
Router('/cadastro', module: CadastroModule(), transition: TransitionType.rightToLeft),
Router('/realizarDoacao', module: RealizarDoacaoModule(), transition: TransitionType.rightToLeft),
Router('/solicitarDoacao', module: SolicitarDoacaoModule(), transition: TransitionType.rightToLeft),
```

Figura 13: Gerenciamento de Rotas do Modular.
Fonte: criado pelo próprio autor

Foi muito interessante em um primeiro projeto com uma complexidade maior utilizar um Package totalmente brasileiro como esse, que não só forneceu um meio de injeção de dependência, que era necessário, como também trouxe uma estrutura para todo o projeto, além de facilitar na implementação de muitas outras funcionalidades, como por exemplo na animação de transição entre telas. No entanto, este é voltado para aplicações de maior estrutura, que não era o caso do DoeBrasil.

2.4 MUDANÇA DE ARQUITETURA E NOVA VERSÃO

Mais tarde, após um estudo com outros Packages, e re-avaliação do tamanho e complexidade da aplicação, foi decidido por ser feita a substituição do Modular na próxima versão do App, pois a complexidade em implementação do Modular, em certas partes devido a pouca experiência com o package, não era tão recompensadora pelo tamanho e complexidade do DoeBrasil.

Já o MobX ainda se manteve como o gerenciador de estado. Por ter sido extremamente simples de se utilizar, não houve a necessidade de fazer uma mudança. A gerência de estados é, basicamente, responsável por notificar o Flutter quando ele deve re-construir a tela de acordo com algum novo estado que mudou. Colocando em um exemplo prático, temos que a tela de login inicia-se em um estado “0”, onde os *textfields* estão vazios. No momento em que o usuário começar a digitar algum valor em qualquer um dos *textfields*, o estado da aplicação é alterado, armazenando o valor digitado pelo usuário e atualizando o estado na tela para ser apresentado ao usuário.

Pelo tamanho e complexidade do projeto não existia a necessidade de um *package* como esse para gerência de estado, pois tudo poderia ser desenvolvido utilizando somente o *setState* (que é a forma para gerenciar estado disponibilizada nativamente pelo Flutter) em conjunto com funções de *Callback*. No entanto, como uma forma de colocar em prática conhecimentos adquiridos em estudo foi decidido por utilizar o MobX, um *package* de gerência de estado muito utilizado na comunidade devido a sua geração de código, o que torna alguns processos bem mais simples.

Apesar das vantagens oferecidas pelo MobX, a escolha de utilizá-lo para gerenciar o estado da aplicação não foi apenas baseada na facilidade de geração de código. Outro motivo importante foi a capacidade do MobX de simplificar a lógica de atualização e reatividade dos componentes, tornando o código mais limpo e legível. Além disso, o MobX oferece recursos como observadores reativos e a capacidade de trabalhar com objetos observáveis, o que facilita a detecção de mudanças e atualizações no estado da aplicação. Essas características foram consideradas essenciais para lidar com a complexidade crescente do projeto e garantir a escalabilidade e manutenção do código.

A gerência de estado através do MobX é muito simples e intuitiva, sendo a sua base uma tríade (Figura 14). As *Stores* (Controllers) vão existir variáveis que serão observadas com anotação de @observable. Para fazer mudanças nessas variáveis é utilizada as funções com anotação de @action. Também há os componentes de *Reaction*, ou reação, que vão ser notificados de qualquer mudança que aconteça nas variáveis observáveis e assim refletir a mudança necessária para a tela.

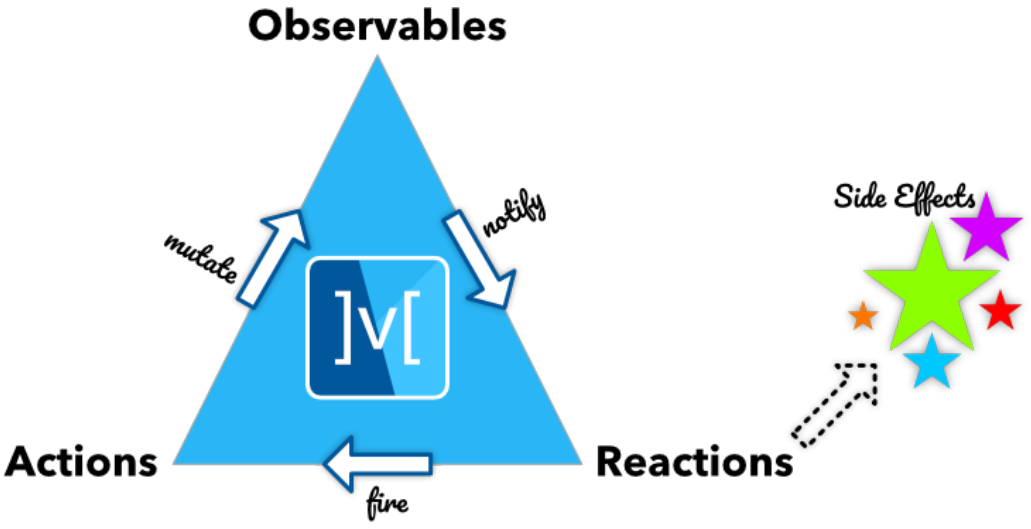

Figura 14: Tríade de Obsevables, Actions e Reactions do MobX.
Fonte: Documentação do MobX no pub.dev (https://pub.dev/packages/mobx)

Quanto à utilização do MobX, não houve muita dificuldade de implementação, pois além de ser tudo muito simples e intuitivo, existem muitos exemplos disponíveis, facilitando a resolução de problemas e sanando dúvidas. No início a maior dificuldade foi conciliar recursos de dois *packages* distintos, a injeção de dependências do Modular com as Stores do MobX. No entanto, essas dificuldades foram diminuindo conforme o desenvolvimento avançava.

Outro ponto extremamente positivo é a centralização de toda a regra de negócio da aplicação em um controller do MobX, conforme apresentado na Figura 15. Essa centralização deixou o componente da *View* somente com a função apresentar elementos referentes a tela, e a regra de negócio sendo delegada aos *controllers*, implementando assim corretamente o MVC.

```dart
part 'home_controller.g.dart';

class HomeController = _HomeControllerBase with _$HomeController;

abstract class _HomeControllerBase with Store {
  final LoginController _loginController;
  final CovidAPI repository;
  final ILocalStorage localStorage;

  @observable
  bool isExpanded = false;

  @observable
  List<DoacaoPedidoModel> meusPedidos =
      <DoacaoPedidoModel>[].asObservable();

  _HomeControllerBase(
      this._loginController, this.repository, this.localStorage) {
    recuperarPedidosMinhaCidade();
  }

  @action
  Future<int> fazerLogoutFAB() {
    return _loginController.fazerLogout();
  }

  @action
  Future<List<DoacaoPedidoModel>> recuperarPedidosMinhaCidade() async {
    meusPedidos = null;
    var token = await localStorage.getToken("token");
```

Figura 15: *HomeController* feita como *Store* do *MobX*.
Fonte: criado pelo próprio autor

Outro *package* também utilizado na aplicação foi o Hive, um *package* de banco de dados baseado em chave e valor utilizado para salvar informações localmente quando necessárias. No caso do DoeBrasil, o Hive foi utilizado para salvar o *token* de acesso de usuário ao fazer *login*, podendo assim persistir o *login* do usuário na aplicação caso este tenha feito o *login* anteriormente.

Após o lançamento da versão 1.0 do app, houve algumas sugestões de

mudanças nas funcionalidades principais, no qual agora seria mais focado nas ONGs parceiras do DoeBrasil.

Por ser uma grande mudança, que iria levar um grande tempo, muita coisa do projeto foi adaptada para uma melhor manutenção futura, entre essas mudanças estariam a substituição do Modular pelo Provider, que gerou uma implementação mais simples, padronização de todo o código em língua inglesa e uma maior componentização dos *widgets*.

A troca do Modular pelo Provider foi feita pensando na dimensão do projeto, que é de pequeno a médio, e o Modular apesar de trazer muitos benefícios também estava trazendo muita complexidade sem necessidade para um projeto desse porte, sendo que a funcionalidade mais utilizada do Modular no projeto que era a de fazer injeção de dependências também é providenciada pelo Provider de uma forma mais clara e concisa, porém ainda mantendo a robustez já que é um pacote amplamente utilizado e que está em constante melhoria.

Na nova arquitetura, fortemente baseada no MVC (Model-View-Controller), mas sem a modularização, o aplicativo ficou mais "simplista" na estrutura de pastas, que ficou subdividido em "*Controllers", "Exceptions", "Models", "Repository", "Utils", "Views"* e *"Widgets*" conforme a Figura 16.

Figura 16: Nova estrutura de pastas na nova versão da aplicação.
Fonte: criado pelo próprio autor

Na camada de *Controllers*, apresentada Figura 17, estão os *controllers*, que nesse caso são as "*Stores*" do MobX. Estes *controllers* são responsáveis por conter a regra de negócio da aplicação e o estado de certas telas da aplicação. Um exemplo é o login_controller.dart, controller responsável por armazenar tanto a

regra de negócio quanto o estado da login_screen.dart.

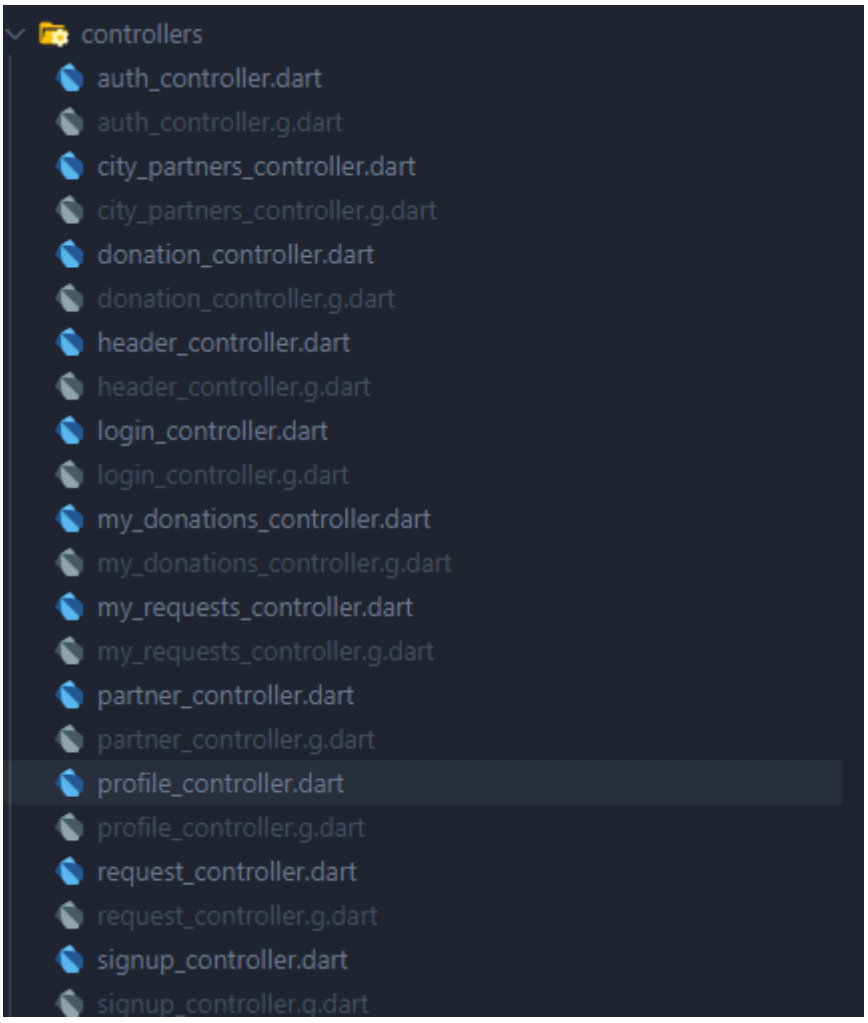

Figura 17: *Controllers* na nova versão da aplicação.
Fonte: criado pelo próprio autor

Porém nem sempre é um caso de possuir um *controller* para cada tela. No caso o controller my_donations_controller.dart , Figura 18, contém o estado geral de “Doações” e é utilizado tanto pela tela donations_screen.dart (Figura 19) quanto pela tela my_donations_screen.dart (Figura 20).

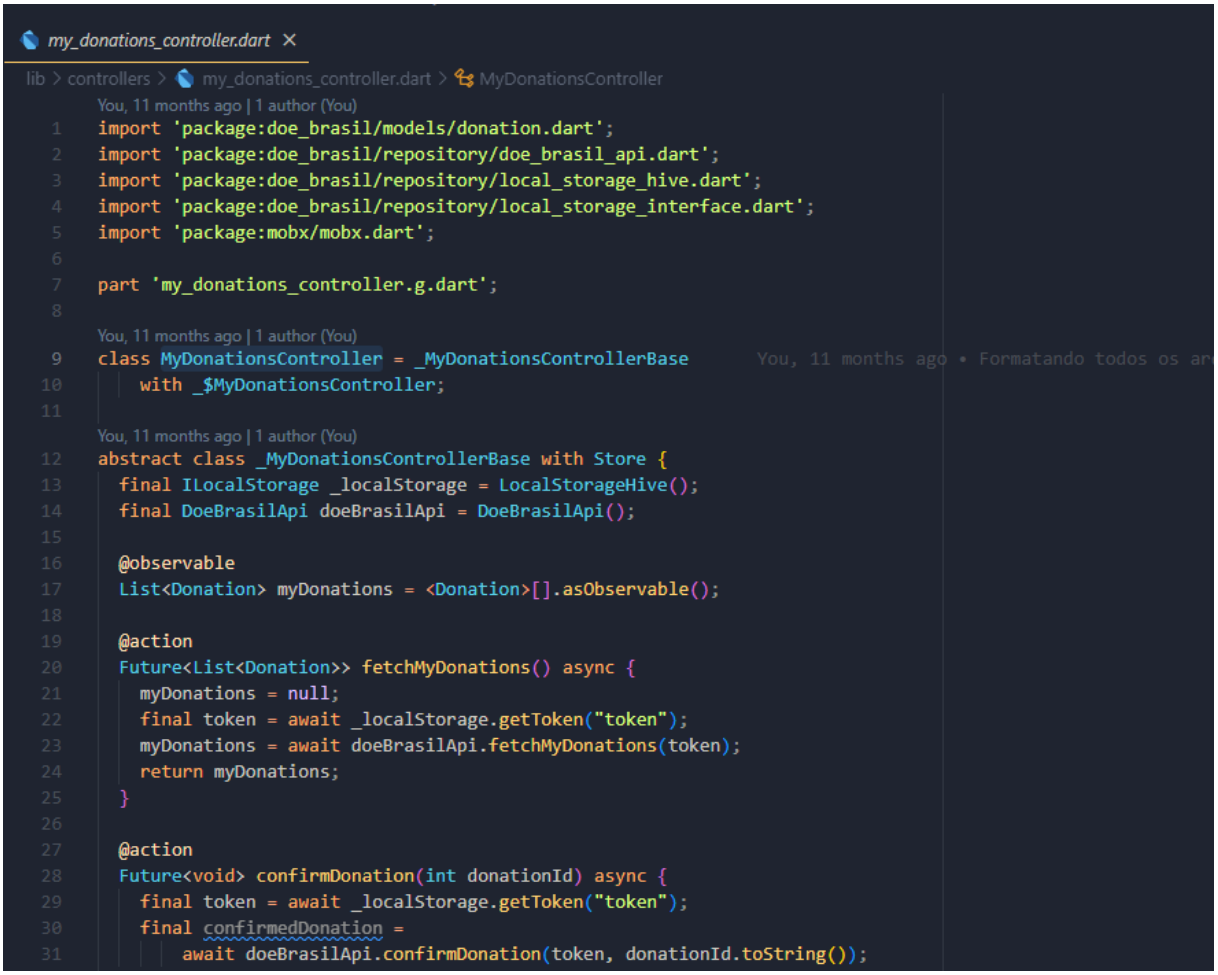

Figura 18: Código do *MyDonationsController*.
Fonte: criado pelo próprio autor

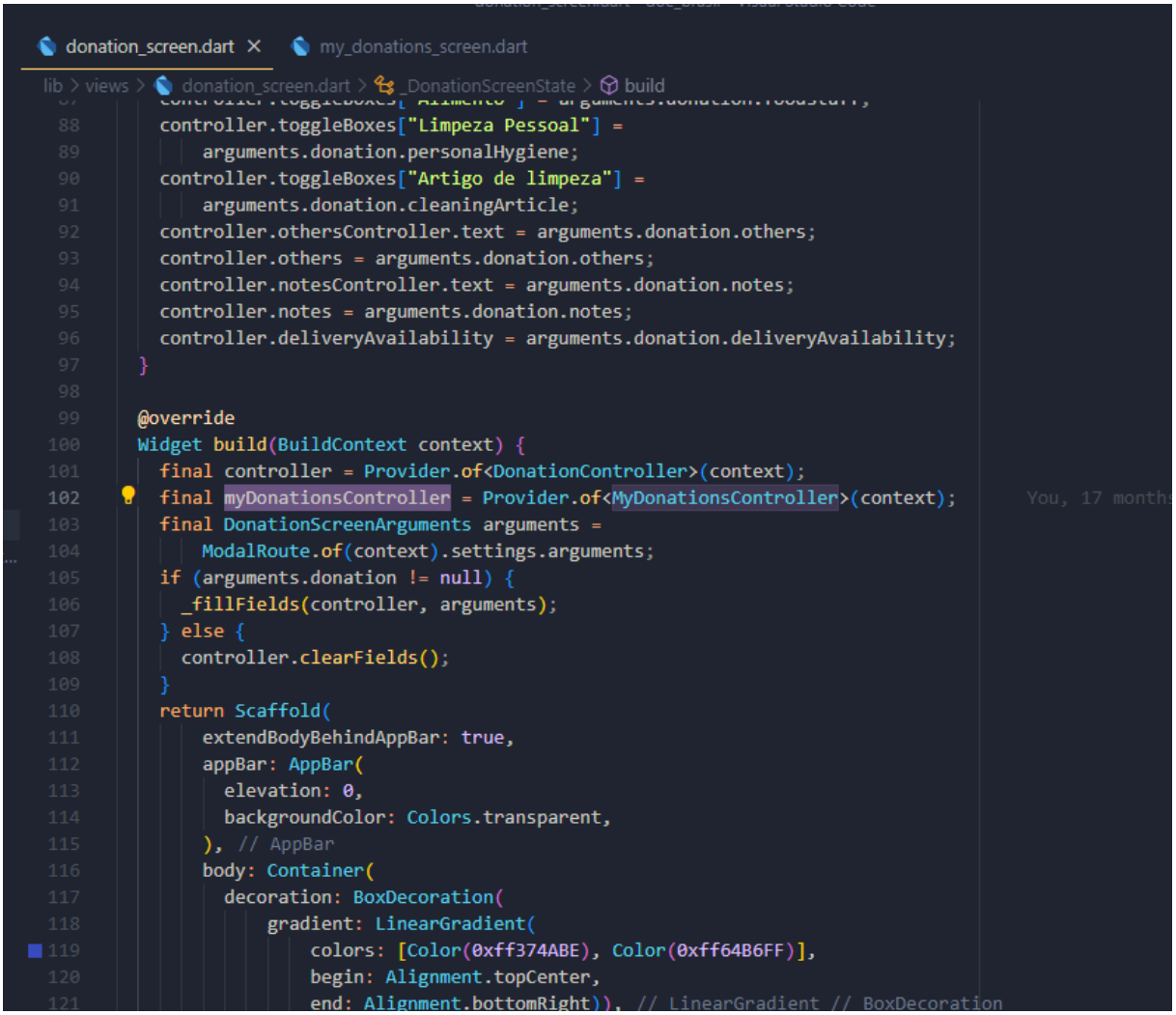

Figura 19: Código da *DonationScreen*.
Fonte: criado pelo próprio autor

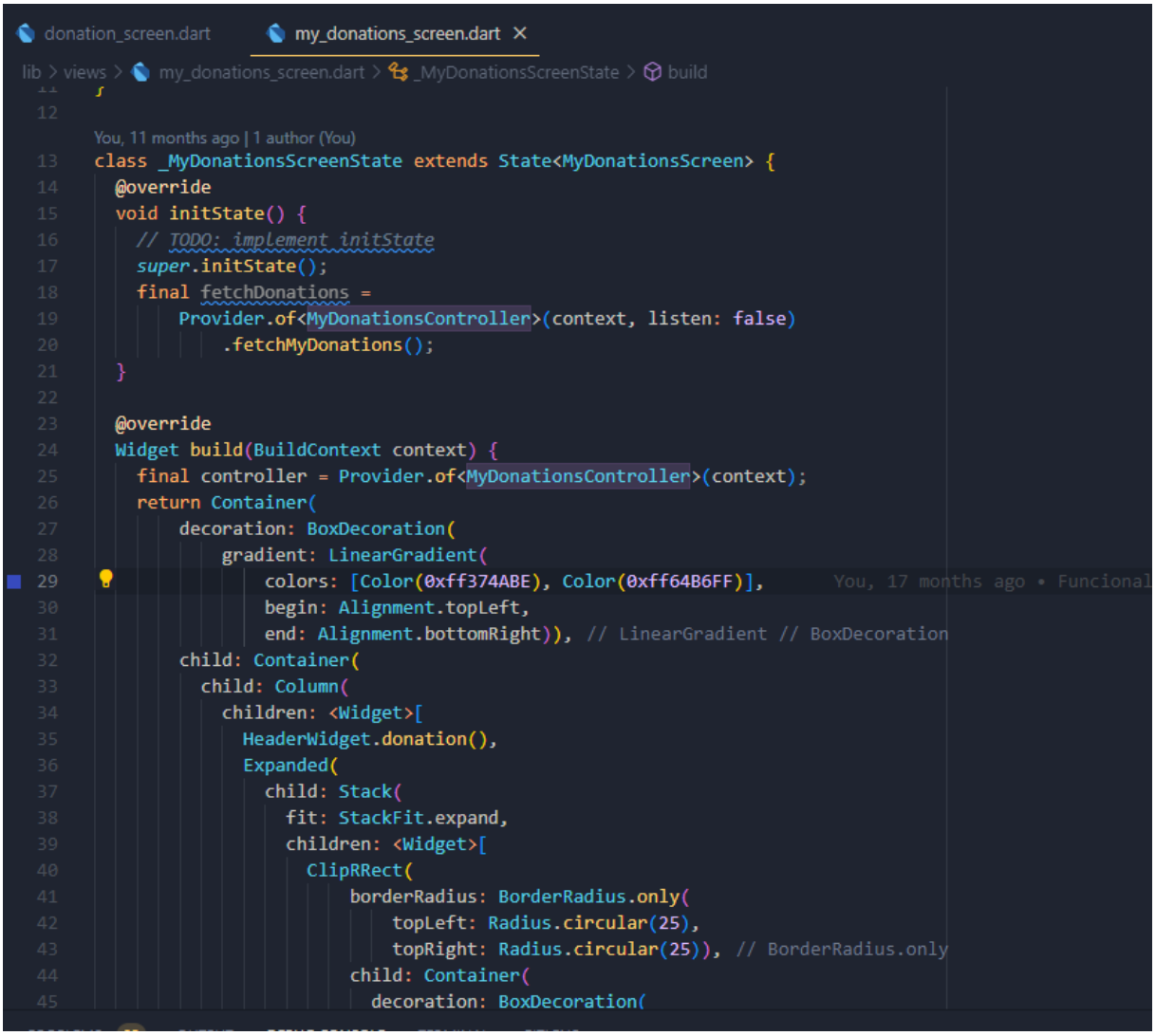

Figura 20: Código da tela *MyDonationsScreen*.
Fonte: criado pelo próprio autor

A função dos controllers não mudou, apenas mudaram sua organização nas pastas do projeto e sua injeção na aplicação, sendo feita agora pelo Provider e não mais pelo Modular.

Na camada de "*Exceptions*" está uma *exception* customizada feita com um método para converter o erros que ocorrerem nas chamadas para a API em String, apresentado na Figura 21.

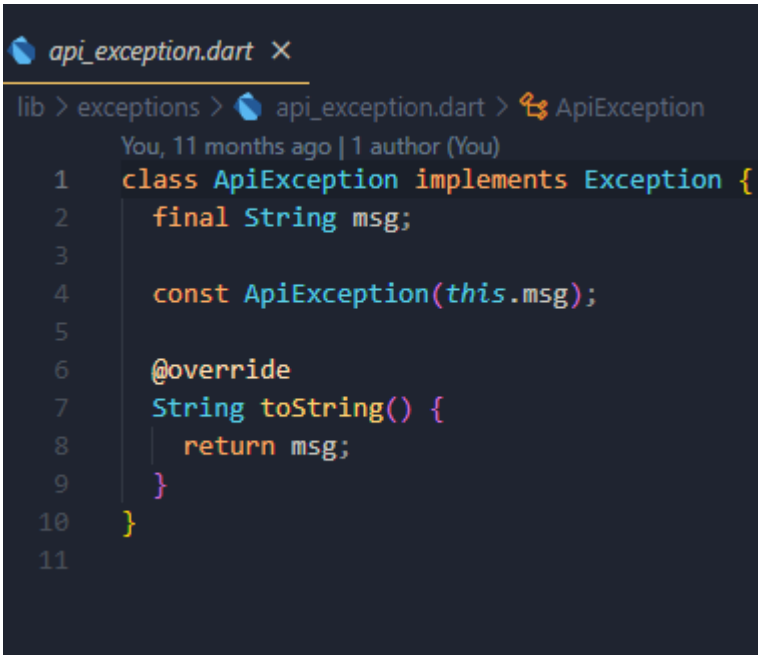

Figura 21: *Exception* customizada.
Fonte: criado pelo próprio autor

Já na camada de "*Models*" temos os modelos de objetos que são utilizados pela aplicação, algumas classes de modelos contém métodos para facilitar na conversão de Json para objeto e vice-versa , apresentados na Figura 22 e Figura 23 .

request.dart

lib > models > request.dart > ...

You, 11 months ago | 1 author (You)

```dart
class Request {
  final int id;
  final bool foodstuff;
  final bool personalHygiene;
  final bool cleaningArticle;
  final String others;
  final String notes;
  final int status;
  final Partner partner;
  final int partnerId;

  Request(
      {this.id,
      this.foodstuff,
      this.personalHygiene,
      this.cleaningArticle,
      this.others,
      this.notes,
      this.status,
      this.partner,
      this.partnerId});

  bool get haveNotes {
    return notes != null && notes != "";
  }

  bool get haveOthers {
    return others != null && others != "";
  }
```

Figura 22: Classe de *Request* (Pedido).
Fonte: criado pelo próprio autor

```dart
  Map<String, dynamic> editedRequestToJson() {
    Map<String, dynamic> map = {
      "id": this.id,
      "generoAlimenticio": this.foodstuff,
      "higienePessoal": this.personalHygiene,
      "artigoLimpeza": this.cleaningArticle,
      "outros": this.others,
      "observacoes": this.notes,
    };

    return map;
  }

  factory Request.fromJson(Map<String, dynamic> json) {
    return Request(
      id: json["id"] as int,
      foodstuff: json["generoAlimenticio"] as bool,
      personalHygiene: json["higienePessoal"] as bool,
      cleaningArticle: json["artigoLimpeza"] as bool,
      others: json["outros"] as String,
      notes: json["observacoes"] as String,
      status: json["status"] as int,
      partner: json["parceiro"] != null
          ? Partner(
              id: json["parceiro"]["id"] as int,
              name: json["parceiro"]["nome"] as String,
              cpfCnpj: json["parceiro"]["cpfCnpj"] as String,
              email: json["parceiro"]["email"] as String,
              phone: json["parceiro"]["telefone"] as String) // Partner
          : null,
    ); // Request
  }
}
```

Figura 23: Métodos utilitários de conversão da classe de *Request (Pedido)*.
Fonte: criado pelo próprio autor

Indo para a camada de "*Repository*" temos os repositórios que vão centralizar os métodos seja para o banco local, apresentando na Figura 24 quanto para a API remota.

Figura 24: Código do repositório de *LocalStorage* implementado utilizando o Hive.
Fonte: criado pelo próprio autor

O recomendado para estas chamadas a API remota seria separar um repositório para cada *model*, como por exemplo, um repositório para "*Partner*" um para "*Donation*". No entanto, aqui foi feita a centralização de todas as chamadas em um só repositório chamado de "DoeBrasilApi", Figura 25, pois inicialmente, como eram poucas chamadas, era mais simples de se fazer manutenção.

```dart
import 'package:doe_brasil/exceptions/api_exception.dart';      You, 17 months ago • Tela de Login, Te
import 'package:doe_brasil/models/city.dart';
import 'package:dio/dio.dart';
import 'package:doe_brasil/models/client.dart';
import 'package:doe_brasil/models/donation.dart';
import 'package:doe_brasil/models/partner.dart';
import 'package:doe_brasil/models/partner_action.dart';
import 'package:doe_brasil/models/request.dart';
import 'package:doe_brasil/utils/constants.dart';

You, 11 months ago | 1 author (You)
class DoeBrasilApi {
  Dio dio = Dio();

  Future<List<City>> getCities(String state) async {
    var response = await dio.get('${URL_BASE}/cidades/${state}');
    List<City> list = [];
    for (var cityJson in (response.data['cidades'] as List)) {
      City city = City(cityJson);
      list.add(city);
    }
    return list;
  }

  Future<void> signUp(Client client) async {
    try {
      if (client.cpfCnpj.length == 14) {
        print("Entrou!");
        final response =
            await dio.post('${URL_BASE}/usuario/signup', data: client.toJson());
      } else {
        final response = await dio.post('${URL_BASE}/parceiro/signup',
            data: client.toJson());
      }
```

Figura 25: Código do repositório remoto, contendo as chamadas para a API do DoeBrasil.
Fonte: criado pelo próprio autor

A camada de “Utils” continua com a mesma utilidade de antes, contendo variáveis, funções e animações que vão ser utilizadas por toda a aplicação. Na Figura 26 é apresentado o arquivo app_routes.dart, onde é adicionada as *Strings* constantes de todas as rotas nomeadas da aplicação.

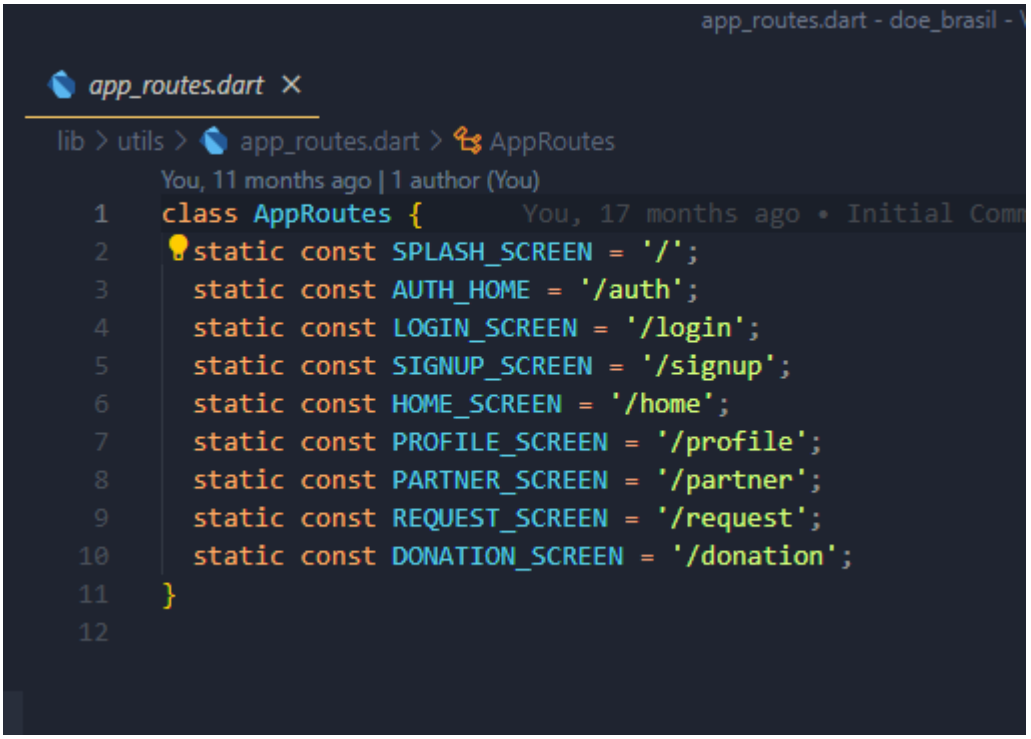

Figura 26: Classe de AppRoutes, contendo os caminhos de todas as rotas da aplicação.
Fonte: criado pelo próprio autor

Já no arquivo custom_route.dart, apresentado na Figura 27, temos a animação de transição de tela feita do zero, pois antes essas animações eram provindas do *package* do Modular.

```dart
class CustomPageTransitionBuilder extends PageTransitionsBuilder {
  @override
  Widget buildTransitions<T>(
    PageRoute<T> route,
    BuildContext context,
    Animation<double> animation,
    Animation<double> secondaryAnimation,
    Widget child,
  ) {
    const Curve curve = Curves.easeInOut;
    return SlideTransition(
      transformHitTests: false,
      position: Tween<Offset>(
        begin: Offset(1.0, 0.0),
        end: Offset.zero,
      ).animate(CurvedAnimation(parent: animation, curve: curve)), // Tween
      child: SlideTransition(
        position: Tween<Offset>(
          begin: Offset.zero,
          end: Offset(-1.0, 0.0),
        ).animate(CurvedAnimation(parent: secondaryAnimation, curve: curve)), // Tween
        child: child,
      ), // SlideTransition
    ); // SlideTransition
  }
}
```

Figura 27: Código de rota customizada com animação.
Fonte: criado pelo próprio autor

Na camada “*View*” é organizada todas as telas do aplicativo, Figura 28, desde a *SplashScreen*, Figura 29, até as telas de doações, antes separadas por módulos aqui elas centralizadas em uma só pasta.

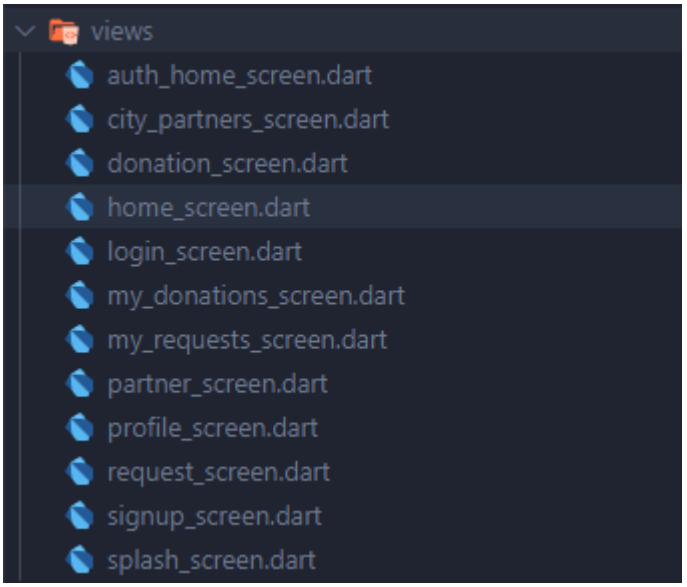

Figura 28: Camada de *view*.
Fonte: criado pelo próprio autor

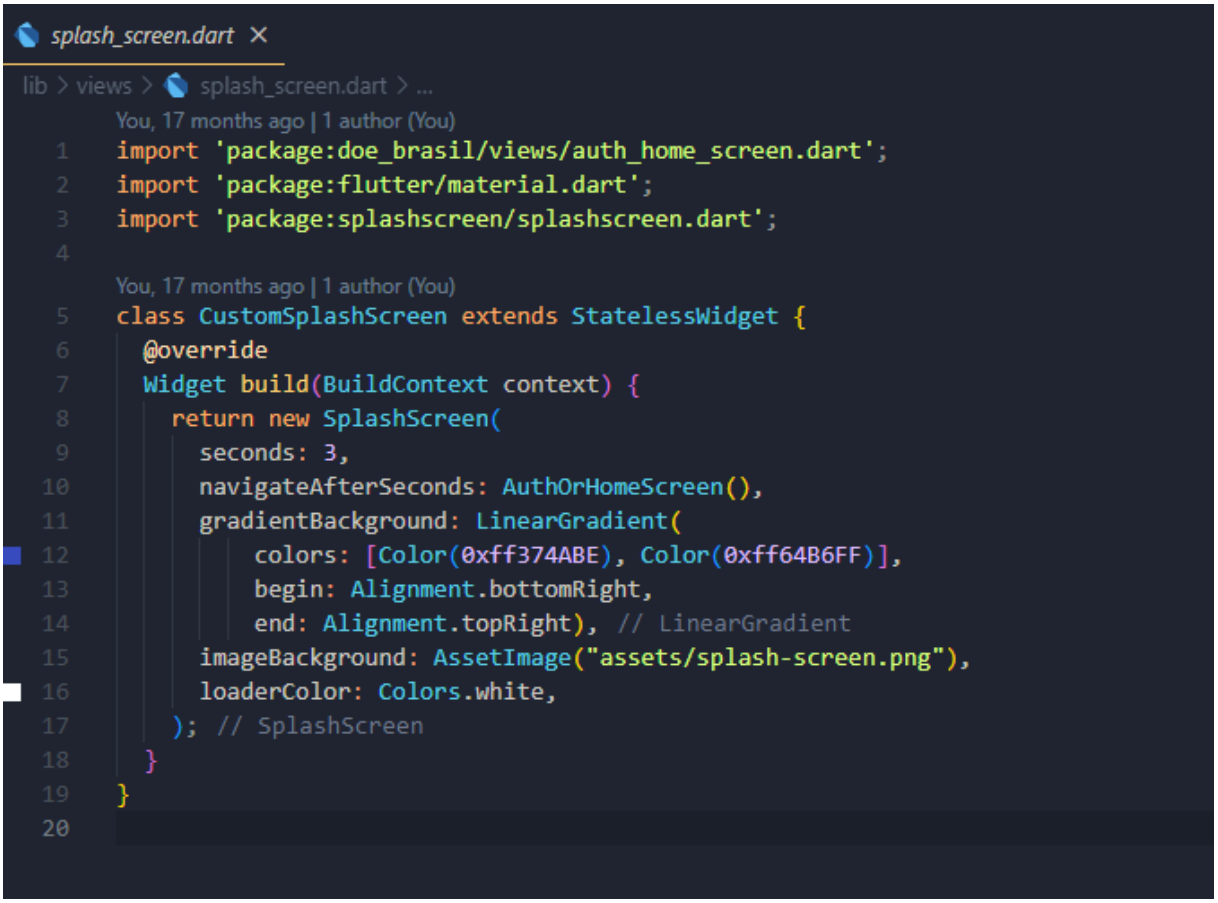

Figura 29: Código da *SplashScreen*.
Fonte: criado pelo próprio autor

Com essa mudança e estruturação nova sempre que era necessário fazer alguma mudança era bem mais simples de encontrar o que era necessário alterar já que houve uma boa separação de pastas todo o projeto. Graças a abordagem de deixar os componentes mais genéricos e consequentemente mais reutilizáveis aumentou muito a legibilidade do código, além de facilitar muito também o desenvolvimento de novas funcionalidades. Além disso, tais mudanças trouxeram também uma melhora na performance do aplicativo, já que a reconstrução da tela fica mais inteligente pelo *framework* quando tudo está bem "componentizado". Neste caso, o *framework* sabe o que deve renderizar novamente, ou não, quando ocorrer uma mudança de estado.

Graças ao Flutter ser um *framework* pensado para ser multiplataforma, a geração da versão Android e IOS eram simples comandos executados no terminal. Porém, por limitações da própria Apple, o *build* para a plataforma IOS somente pode ser feito em um computador com o sistema operacional MacOS. Além disso, também é necessário fazer o pagamento de uma anuidade para disponibilização da aplicação na App Store. Por esses motivos foi decidido por fazer o lançamento do

aplicativo inicialmente somente para o sistema operacional Android, através da Play Store.

3. CONCLUSÃO

Em conclusão, o aplicativo de doações desenvolvido neste trabalho de conclusão de curso tem o objetivo de ajudar aqueles que precisam em tempos difíceis, como a pandemia do COVID-19. A utilização do Flutter como plataforma de desenvolvimento permitiu a criação de um aplicativo intuitivo e fácil de usar, que facilita o processo de doação e torna-o mais acessível para todos. Além disso, a implementação de medidas de segurança garante a proteção dos dados dos usuários e a transparência nas transações. É esperado que o aplicativo contribua para o bem-estar da sociedade e para a melhoria das condições de vida de muitas pessoas. Em tempos de crise, é importante que a tecnologia seja utilizada para o bem da humanidade, e esse aplicativo é um exemplo disso.

REFERÊNCIAS

DART. Dart Language Tour. Disponível em: https://dart.dev/guides/language/language-tour. Acesso em: 9 fev. 2023.

BACON, John et al. The Dart Programming Language. ACM SIGPLAN Notices, v. 46, n. 10, p. 1-10, out. 2011.

Google. (2022). Flutter Documentation. Disponível em: https://flutter.dev/docs. Acesso em: 9 fev. 2023.

Liu, J., Ma, X., Zhang, L., & Dai, X. (2019). Exploring the Applicability of Google Flutter for Mobile App Development. In 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC) (pp. 658-663). IEEE.

Felchlin, T., Schmid, J., Pfister, J., & Steiner, R. (2020). Flutter: A Multi-Platform Framework for Mobile and Web Applications. In 2020 IEEE 15th International Conference on Global Software Engineering (ICGSE) (pp. 21-28). IEEE.

Chan, J., & Chong, S. C. (2020). Cross-Platform Mobile Application Development: A Comparative Study of React Native and Flutter. In Proceedings of the 2020 3rd International Conference on E-Society, E-Education and E-Technology (pp. 36-41).

Han, S., & Wang, C. (2021). Study on the Application of Flutter in Mobile Development. In 2021 IEEE 5th International Conference on Control Science and Systems Engineering (ICCSSE) (pp. 307-310). IEEE.

Reyes, J. A. (2021). Flutter: A Rapid Application Development Framework for Multi-Platform Mobile Applications. In 2021 IEEE 13th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment, and Management (HNICEM) (pp. 21-27). IEEE.

Flutter (2023). State management. Acesso em 20 de fevereiro de 2023, disponível em https://flutter.dev/docs/development/data-and-backend/state-mgmt

MobX (2023). Official documentation. Acesso em 20 de fevereiro de 2023, disponível

em https://mobx.netlify.app/

Nunes, V. (2021). MobX - A melhor solução para gerenciamento de estado em Flutter. Acesso em 20 de fevereiro de 2023, disponível em https://medium.com/@vitorhugond/mobx-a-melhor-solu%C3%A7%C3%A3o

BURBECK, Steve. Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC). SIGSMALL/PC Symposium, 1992.

FOWLER, Martin et al. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002.

GAMMA, Erich et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1995.

LEFF, Avi; RAYFIELD, William. Web Application Architecture: Principles, Protocols and Practices. John Wiley & Sons, 2001.

FIELDING, Roy T. Architectural Styles and the Design of Network-based Software Architectures. University of California, 2000.

MARTIN, Ricardo. APIs REST e sua importância para a Internet das Coisas (IoT). Embarcados, 2020. Disponível em: https://www.embarcados.com.br/apis-rest-e-sua-importancia-para-a-internet-das-coisas-iot/. Acesso em: 24 fev. 2023.

MENDES, João. O que é API RESTful? Medium, 2019. Disponível em: https://medium.com/@joaomendes_39424/o-que-%C3%A9-api-restful-9e4e4d4b1c56. Acesso em: 24 fev. 2023.

Hive Documentation. Disponível em: https://docs.hivedb.dev/. Acesso em: 15 de fevereiro de 2023.

ALNEMR, Youssef. Flutter & Hive—The Perfect Duo for Local Database. Medium, 8 de março de 2021. Disponível em: https://youssef-alnemr.medium.com/flutter-hive-the-perfect-duo-for-local-database-f0bae71b1875. Acesso em: 15 de fevereiro de 2023.

DONNELSON, Brandon. Flutter: Using the Hive Database. Brandon Donnelson, 10 de setembro de 2021. Disponível em: https://www.brandondonnelson.com/flutter-using-the-hive-database/. Acesso em: 15 de fevereiro de 2023.

PROVIDER. Flutter Provider package. Disponível em: https://pub.dev/packages/provider. Acesso em: 20 fev. 2023.

FLUTTERANDO. flutter_modular. Disponível em: https://modular.flutterando.com.br/docs/intro/. Acesso em: 25 fev. 2023.

Eiseman, L. (2017). A Psicologia das Cores: Como as cores afetam a emoção e a razão. Gustavo Gili.