



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO



Nurse

Dorgival da Rocha Filho

Orientador: Prof. Dr. Severino Lampeão

Co-orientador: Prof. Dr. Zé Baiano

Natal, RN, 2 de dezembro de 2022



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO



Nurse

Dorgival da Rocha Filho

Orientador: Prof. Dr. Itamir Moraes Filho

Trabalho de Conclusão de Curso de Graduação na modalidade Monografia, submetido como parte dos requisitos necessários para conclusão do curso de Engenharia de Computação pela Universidade Federal do Rio Grande do Norte (UFRN/CT).

Natal, RN, 2 de dezembro de 2022

Resumo

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec vehicula vitae lectus ut pretium. Vestibulum tristique leo eu purus vehicula ullamcorper. Nulla ut ultricies massa. Suspendisse eu neque pharetra, faucibus erat ac, pretium augue. Vivamus id euismod leo. Cras eget neque pellentesque, fringilla dolor eu, pretium libero. Mauris sed justo feugiat, varius ligula sed, posuere metus. Fusce lacus mi, molestie a rutrum id, scelerisque ut lacus. In hac habitasse platea dictumst. In vitae elit faucibus, molestie orci efficitur, consectetur neque. Ut placerat, augue eu pellentesque euismod, dui enim euismod elit, quis sollicitudin lectus lorem gravida mi. Donec ut leo pretium, finibus arcu in, tincidunt sem. Phasellus diam ante, pulvinar vel neque non, sagittis aliquam nibh. Praesent id condimentum nunc, quis interdum metus. Curabitur eget diam vitae enim consequat mollis quis dictum turpis.

Palavras-chave: Processamento de texto, L^AT_EX, Preparação de Teses, Relatórios Técnicos.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec vehicula vitae lectus ut pretium. Vestibulum tristique leo eu purus vehicula ullamcorper. Nulla ut ultricies massa. Suspendisse eu neque pharetra, faucibus erat ac, pretium augue. Vivamus id euismod leo. Cras eget neque pellentesque, fringilla dolor eu, pretium libero. Mauris sed justo feugiat, varius ligula sed, posuere metus. Fusce lacus mi, molestie a rutrum id, scelerisque ut lacus. In hac habitasse platea dictumst. In vitae elit faucibus, molestie orci efficitur, consectetur neque. Ut placerat, augue eu pellentesque euismod, dui enim euismod elit, quis sollicitudin lectus lorem gravida mi. Donec ut leo pretium, finibus arcu in, tincidunt sem. Phasellus diam ante, pulvinar vel neque non, sagittis aliquam nibh. Praesent id condimentum nunc, quis interdum metus. Curabitur eget diam vitae enim consequat mollis quis dictum turpis.

Keywords: Document Processing, L^AT_EX, Thesis Preparation, Technical Reports.

Sumário

Sumário	i
Lista de Figuras	iii
Lista de Tabelas	v
Lista de Códigos	vii
1 Introdução	1
1.1 Contextualização	1
1.2 Problema	1
1.2.1 Procedimento comum de vacinação	1
1.3 Objetivos	2
1.4 Estrutura do Trabalho	2
2 Fundamentação Teórica	3
2.1 Dart	3
2.1.1 Introdução	3
2.1.2 Compilação	3
2.2 <i>Flutter</i>	5
2.2.1 Arquitetura do <i>framework</i>	5
2.2.2 <i>Widgets</i>	5
2.2.3 Diferenças para outras tecnologias	9
2.2.4 Gerenciamento de estado	9
2.2.5 <i>Provider</i>	10
2.2.6 <i>MobX</i>	10
2.3 Persistência de Dados	11
2.3.1 <i>Banco de Dados SQLite</i>	12
2.3.2 Modelagem do banco de dados	13
3 Trabalhos relacionados	15
4 Nurse: uma aplicação para produtividade em vacinações	17
4.1 Requisitos do Sistema	17
4.1.1 Requisitos Funcionais	17

4.1.2	Requisitos Não Funcionais	17
4.1.3	Casos de Uso	18
4.2	Arquitetura do Sistema	18
4.2.1	Camadas de Responsabilidade	20
4.3	Telas	28
4.4	Pacotes e Bibliotecas	29
4.4.1	<i>Plugins sqflite_sqlcipher e sqflite</i>	30
4.4.2	Bibliotecas <i>mobx</i> e <i>flutter_mobx</i>	30
4.4.3	Pacote <i>provider</i>	32
4.4.4	Biblioteca Excel (XlsIO)	33
4.5	Persistência de Dados	35
4.5.1	Diagrama de Classes	35
4.5.2	Uso do Banco de Dados	37
5	Experimentos e Resultados	39
5.1	Testes Unitários	39
5.2	Fluxos de Telas	41
5.2.1	Fluxo de cadastro de entidade	41
5.2.2	Fluxo de cadastro de vacinação	41
5.2.3	Fluxo de exportação	42
6	Conclusão	49
	Referências bibliográficas	50
A	Requisitos funcionais detalhados	55
B	Pacotes e versões utilizados	59

Lista de Figuras

2.1	Plataformas de compilação do Dart	4
2.2	Estrutura simplificada de <i>widgets</i> da aplicação <i>Nurse</i>	6
2.3	Exemplo de uma árvore de widgets com <i>InheritedWidget</i> . (Faust 2020) (Boelens 2018)	9
2.4	Ciclo básico do gerenciamento de estado com o MobX (P. Podila 2018) (Podila & Weststrate 2018)	11
2.5	Ciclo detalhado do gerenciamento de estado com o MobX (P. Podila 2018) (Podila & Weststrate 2018)	12
2.6	Modelagem conceitual de parte do banco de dados	13
2.7	Modelagem lógico de parte do banco de dados	13
4.1	Diagrama de casos de uso da aplicação <i>Nurse</i>	20
4.2	Camadas da aplicação Nurse e suas dependências	21
4.3	Principais telas da aplicação <i>Nurse</i>	29
4.4	Telas de listagem e cadastro de entidade	30
4.5	Descritivo de fluxo dos métodos da classe ExcelService	34
4.6	Diagrama de classe para a aplicação Nurse	35
5.1	Resultado dos testes unitários da aplicação <i>Nurse</i>	40
5.2	Cobertura de testes na aplicação <i>Nurse</i>	40
5.3	Fluxo de cadastro de nova campanha (parte 1)	42
5.4	Fluxo de cadastro de nova campanha (parte 2)	43
5.5	Fluxo de cadastro de nova vacinação (parte 1)	44
5.6	Fluxo de cadastro de nova vacinação (parte 2)	45
5.7	Fluxo de exportação da planilha de vacinação (parte 1)	46
5.8	Fluxo de exportação da planilha de vacinação (parte 2)	47

Lista de Tabelas

4.1	Requisitos funcionais da aplicação Nurse	18
4.2	Requisitos não funcionais da aplicação Nurse	19
4.3	Descrição dos atores envolvidos na aplicação Nurse	19
A.1	Requisitos funcionais da aplicação Nurse: detalhes do requisito RF01 . . .	56
A.2	Requisitos funcionais da aplicação Nurse: detalhes do requisito RF02 . . .	57
A.3	Requisitos funcionais da aplicação Nurse: detalhes do requisito RF04 . . .	58
B.1	Pacotes utilizados no projeto e suas respectivas versões	60

Lista de Códigos

2.1	Exemplo de um <i>stateless widget</i> que representa um botão.	6
2.2	Exemplo de um <i>stateful widget</i> que representa um campo de texto.	7
4.1	Trechos da classe <i>PatientFormController</i>	22
4.2	Trechos da classe <i>PatientStore</i>	24
4.3	Trecho da implementação da classe Vaccine	25
4.4	Interface GenericModel	26
4.5	Interface CampaignRepository	26
4.6	Exemplo de uso da interface CampaignRepository	27
4.7	Implementação na classe DatabaseCampaignRepository da interface CampaignRepository	27
4.8	Uso do <i>MobX</i> na classe HomeController	31
4.9	Uso do <i>MobX</i> no <i>widget</i> Home	31
4.10	Uso do <i>Provider</i> no <i>widget</i> Appliers	32
5.1	Teste unitário para verificação da correta criação de uma instância de Campaign	39

1

Introdução

1.1 Contextualização

Em 2019 surgiu o vírus SARS-Cov-2, responsável por causar a maior pandemia já registrada na história, chamada Covid-19. Desde então, uma grande mobilização em todos os países começou visando a criação de uma vacina contra a doença causada pelo vírus e a consequente corrida para a sua produção em massa. De acordo com a Organização Mundial da Saúde (OMS), no início de 2021 existiam 236 vacinas candidatas em fases pré-clínicas ou fase clínica.

No Brasil, a vacinação contra a COVID-19 começou em 18 de janeiro de 2021, com a vacina Coronavac, produzida pelo Instituto Butantan em parceria com a farmacêutica chinesa Sinovac. A vacinação foi realizada em etapas, à medida em que as vacinas eram fabricadas e priorizando os grupos de maior risco de contaminação e de complicações da doença: profissionais da saúde, idosos, pessoas com comorbidade, pessoas com deficiência e população indígena. A vacinação foi feita em todo o país e com a utilização de um plano nacional, mas cada estado e município teve sua própria estratégia para a vacinação, em geral, pautada nesse plano.

Independente da estratégia de vacinação de cada estado e em decorrência da necessidade de mapear com exatidão as pessoas que receberam a vacina, o Ministério da Saúde criou um módulo para o registro de dados nominais sobre a vacinação, o que inclui dados pessoais d(a) vacinado(a), e dose e lote administrados.

Esse sistema foi denominado Sistema de Informação de Programas de Imunização (Novo SI-PNI), e é usado para o registro de dados sobre as campanhas de vacinação. O SIPNI é um sistema web, que pode ser acessado pelo site da SES-SP, e é usado por profissionais de saúde para o registro de dados sobre a vacinação. O SIPNI é dividido em duas partes: o SIPNI Web e o SIPNI Mobile. O SIPNI Web é usado para o registro de dados sobre as campanhas de vacinação, e o SIPNI Mobile é usado para o registro de dados sobre a vacinação em campo.

1.2 Problema

1.2.1 Procedimento comum de vacinação

O material de trabalho comumente utilizado é o papel com a planilha para preenchimento impressa. Feitas as anotações, esses dados são digitados em um sistema de cadastro de vacinação. Dessa forma, existe alguns pontos de falha no processo, como a possibilidade de erro de escrita da informação no momento da sua coleta ou de digitação, no momento em que esses dados são repassados para o sistema online; perda dos dados e dificuldade de compartilhamento dos mesmos.

1.3 Objetivos

O objetivo geral do trabalho é desenvolver a aplicação *mobile* Nurse, que tem como fim facilitar o registro de dados sobre as vacinações realizadas pelos profissionais da saúde, reduzir a possibilidade de erros de escrita e de perda de dados, além de agilizar a integração desses dados com o sistema nacional de imunizações. Como objetivos específicos, tem-se:

- Apresentar o contexto ao qual a aplicação se destina, destacando-se o problema que ela pretende resolver;
- Relatar o processo de desenvolvimento da aplicação, desde a sua concepção até a sua implementação;
- Apresentar os resultados obtidos com o desenvolvimento da aplicação;

1.4 Estrutura do Trabalho

O trabalho está estruturado em seções, como descrito a seguir:

Seção 1 Introdução: trata-se desta seção. Aqui são apresentados o problema, os objetivos e estrutura geral do trabalho.

Seção 2 Fundamentação Teórica: são apresentados os principais conceitos relacionados à plataforma e à modelagem do banco de dados utilizados no desenvolvimento da aplicação.

Seção 3 Implementação: os requisitos, casos de uso e arquitetura da aplicação Nurse, assim como o seu desenvolvimento e os pacotes utilizados para isso são apresentados nessa seção.

Seção 4 Experimentos e Resultados: é nessa seção que são apresentados os testes desenvolvidos para avaliar as funcionalidades da aplicação e os seus resultados. Além disso, são apresentados os principais fluxos de uso da aplicação.

Seção 5 Conclusão: seção dedica às considerações finais sobre o trabalho e as perspectivas futuras para novas funcionalidades que poderão vir a ser implementadas.

Seção 6 Referências: apresenta as referências utilizadas no trabalho.

Seção 7 Informações Adicionais: apresenta, em formato de apêndices, informações adicionais que complementam as informações apresentadas em outras seções do trabalho.

2

Fundamentação Teórica

Neste capítulo serão apresentados os conceitos básicos relacionados ao *framework Flutter*, assim como a linguagem de programação Dart, a biblioteca MobX e o *wrapper Provider*, utilizados para o desenvolvimento desta aplicação.

Também serão discutidas as principais ideias relacionadas à Programação Orientada a Objeto (POO), aos bancos de dados relacionais e, por fim, alguns conceitos relacionados às boas práticas da programação no quesito de arquitetura de sistemas e os princípios SOLID.

2.1 Dart

Nessa seção, busca-se apresentar os conceitos básicos da linguagem Dart, utilizada para o desenvolvimento do *framework Flutter*. O foco agora é mostrar as características que fizeram com que a linguagem Dart foi escolhida para ser a linguagem de programação do *framework Flutter*.

2.1.1 Introdução

Dart é uma linguagem de programação criada pela Google em 2011, com o objetivo de ser utilizada, inicialmente, para o desenvolvimento de aplicações web, mas que tem como objetivo atual ser executada em qualquer plataforma.

A linguagem é orientada a objetos, fortemente tipada e compilada. Ela possui uma sintaxe similar às linguagens C, mas com algumas diferenças, como a utilização de palavras-chave como *var* e *final* para declarar variáveis e *null* para representar a ausência de valor. A linguagem também se assemelha a outras comumente utilizadas no desenvolvimento de aplicações móveis, web e desktop, como Java, Kotlin, Swift e Typescript, por exemplo.

2.1.2 Compilação

Dart possui duas estratégias de compilação que são utilizadas em conjunto, mas em períodos diferentes do processo de desenvolvimento e de utilização da aplicação. Além disso, a linguagem utiliza-se de uma máquina virtual chamada *Dart VM* para executar o código durante o desenvolvimento da aplicação ou traduz o código *Dart* em *JavaScript* para plataforma *web*.

Linguagens dinâmicas como JavaScript, às quais permitem ao desenvolvedor criar variáveis dinâmicas, ou seja, que podem mudar seus tipos em tempo de execução, utilizam a compilação *Just-in-time (JIT)*. Esse tipo de compilação permite que o desenvolvimento se torne mais produtivo ao diminuir o tempo de espera entre uma mudança no código e a execução do mesmo para avaliar o resultado. Por outro lado, linguagens estáticas como C

utilizam-se da estratégia de compilação *Ahead-of-time (AOT)*. Dessa forma, todo o código deve ser compilado antes da execução do programa, o que torna o tempo de espera entre uma mudança no código e sua nova execução maior. Enquanto o *JIT* tem vantagens sobre o *AOT* em termos de produtividade, o *AOT* tem vantagens em termos de performance, pois não há a necessidade de pausa na execução no código para análise ou compilação *JIT*. Isso faz com que a aplicação seja inicializada e executada mais rapidamente.

No *Dart*, a estratégia de compilação *JIT* é utilizada no período de desenvolvimento da aplicação, enquanto a compilação *AOT* é utilizada para a compilação final da aplicação. O código compilado é otimizado para a plataforma em que será executado. Por exemplo, o código compilado para dispositivos móveis é otimizado para a arquitetura *ARM*, enquanto o código compilado para *desktop* é otimizado para a arquitetura *x64*.

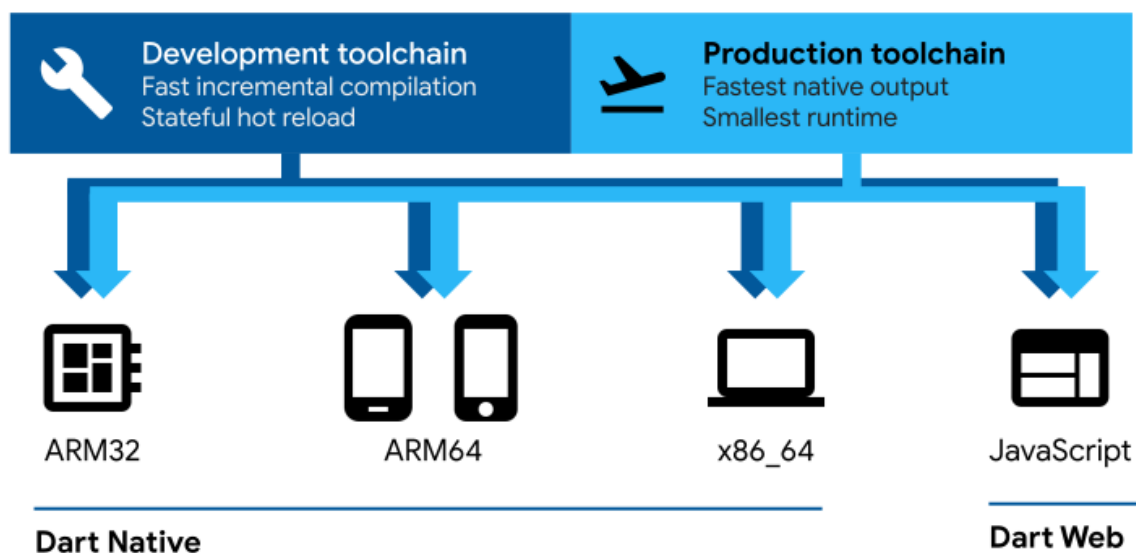


Figura 2.1: Plataformas de compilação do Dart (Time Dart 2022)

Outras características do *Dart* levaram ao seu uso no *Flutter*:

- **ausência de troca de contexto** com a plataforma: como o código compilado é nativo e não possui análises ou novas compilações *JIT*, não há a necessidade de troca de contexto entre a plataforma e a linguagem, o que torna a execução mais rápida;
- **single-threaded**: a preemptividade, isto é, o procedimento de interromper uma tarefa para a execução de outra *thread*, utilizadas em linguagens como *Java* ou *Kotlin* não existe, pois o *Dart* é *single-threaded*, ou seja, não há compartilhamento de memória, nem a necessidade de gerenciamento de estado entre as *threads*, o que evita a possibilidade de erros como condições de corrida ou *deadlocks*;
- Uso de **isolates**: todo o código em *Dart* roda dentro de *isolates*, o qual possui uma única *thread* e se comunica com outros *isolates* por meio de mensagens. Mesmo ações assíncronas são gerenciadas com o uso de classes como *Future* e

Stream ou com o uso de *async/await*. Dessa forma, o *Dart* é capaz de executar código assíncrono sem a necessidade de novas *threads*. Em última instância, é possível criar um *background isolate*, que será responsável por realizar algum tipo de tarefa em segundo plano, como por exemplo, a leitura ou escrita de um arquivo e devolver seu resultado ao *main isolate* na forma de mensagem quando finalizado.

- Esquema de **garbage collection** (GC): a linguagem *Dart* utiliza um avançado esquema de coleta de lixo geracional e que se divide em duas fases. Esse algoritmo favorece a rápida criação e destruição de objetos, o que é comum em aplicações feitas com Flutter, que possui muitos objetos sendo alocados e desalocados, como por exemplo, os *Stateless Widgets*.

2.2 Flutter

Busca-se apresentar, nesta seção, uma visão em alto nível do que é o *framework Flutter*, sua arquitetura, as principais características da tecnologia e suas diferenças para as demais tecnologias no mercado.

O *Flutter*, criado pela *Google*, é desenvolvido em código aberto e visa possibilitar o desenvolvimento de aplicações *Android*, *Web*, *Desktop* e de *software* embarcado a partir de uma única base de código, compiladas nativamente. A aplicação é mantida não só pela empresa que a criou, mas também recebe novas atualizações e incorporações advindas da comunidade.

2.2.1 Arquitetura do *framework*

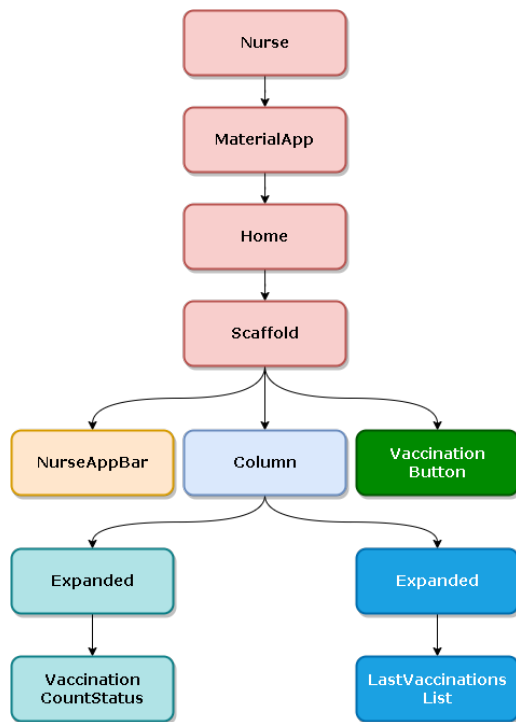
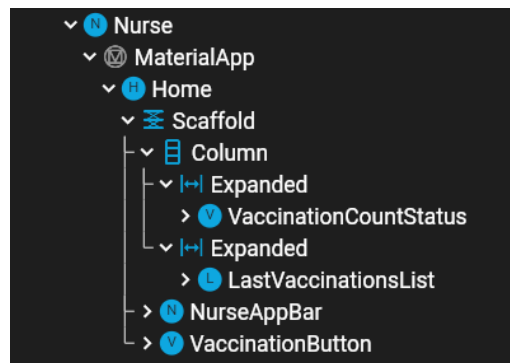
A fazer.

2.2.2 *Widgets*

O *framework* recebe inspiração do *React* e, dessa forma, utiliza-se de *widgets* (2.2) para compor os elementos visuais das aplicações, assim como os estados dos mesmos. Estes *widgets* devem ser responsáveis por gerir, com boa performance, a renderização dos elementos na tela, suas animações e também controlar as interações do usuário com a aplicação.

Além disso, os *Widgets* devem ser customizáveis e extensíveis, de forma que possam ser flexíveis em relação às suas propriedades e reuso. O Flutter possibilita isso ao trazer a responsabilidade de criação e renderização dos *Widgets* da plataforma à qual está rodando para própria aplicação. Posteriormente, na seção 2.2.3, será apresentada uma breve explicação da diferença entre o tratamento de *Widgets* feito pelo Flutter e por outras tecnologias, como o *React Native*, *WebViews* e aplicações escritas em *Android* ou *iOS*.

A seguir, serão apresentados os dois principais tipos de *widgets* utilizados no desenvolvimento da aplicação, suas principais características e diferenças. São eles: *Stateless Widgets* e *Stateful Widgets*, que diferem entre si, essencialmente, pelo gerenciamento ou não de seu estado interno.

(a) Representação da árvore de *widgets*

(b) Árvore apresentada pelo Flutter DevTools

Figura 2.2: Estrutura simplificada de *widgets* da aplicação *Nurse*

Stateless Widgets

Os *stateless widgets* são *widgets* que não possuem mudança estado, ou seja, não possuem propriedades que podem ser alteradas ao longo do tempo, como por exemplo, quando há uma animação ou um campo de texto ao qual um usuário possa interagir e, assim, só dependem das configurações definidas pelo próprio objeto e o contexto em que ele está inserido. Esses *widgets* podem ser utilizados para representar elementos estáticos da interface, como textos, imagens, botões etc. A seguir, é apresentado um exemplo de um *stateless widget* que representa um botão.

```

1  class BotaoPrincipal extends StatelessWidget {
2    final String texto;
3
4    const BotaoPrincipal(Key? key, this.texto) : super(key: key);
5
6    @override
7    Widget build(BuildContext context) {
8      return TextButton(
9        onPressed: () => {print("Clicou no botao principal")},
10       child: Text(texto),
11      );
12    }
13  }
  
```

Código 2.1: Exemplo de um *stateless widget* que representa um botão.

No código 2.1 temos a **classe BotaoPrincipal** que estende um *StatelessWidget* e possui um atributo **texto** que é passado como parâmetro para o construtor da classe e não pode ser modificado posteriormente (regra garantida pela palavra chave *final* que antecede o tipo da variável). A classe possui um método **build** que retorna um *widget* do tipo *TextButton* que recebe como parâmetro uma função anônima que imprime uma mensagem no console e um *widget* do tipo *Text* que recebe como parâmetro o atributo **texto** da classe. No lugar da função de impressão da mensagem em tela poderia ser passado uma função que redireciona o usuário para outra tela, por exemplo.

Stateful Widgets

Os *stateful widgets* são *widgets* que possuem mudança de estado, ou seja, possuem propriedades que podem ser alteradas ao longo do tempo, como por exemplo, quando há uma animação ou um campo de texto ao qual um usuário possa interagir e, assim, dependem não só das configurações definidas pelo próprio objeto e o contexto em que ele está inserido, mas também do estado interno do objeto. A seguir, é apresentado um exemplo de um *stateful widget* que representa um campo de texto.

```
1 class CampoTexto extends StatefulWidget {  
2   final String texto;  
3  
4   const CampoTexto({Key? key, required this.texto}) : super(key: key)  
5   ;  
6  
7   @override  
8   State<CampoTexto> createState() => _CampoTextoState();  
9 }  
10  
11 class _CampoTextoState extends State<CampoTexto> {  
12   String texto = "";  
13  
14   @override  
15   Widget build(BuildContext context) {  
16     return TextField(  
17       onChanged: (value) {  
18         setState(() {  
19           texto = value;  
20         });  
21         print(texto);  
22       },  
23       decoration: InputDecoration(  
24         labelText: widget.texto,  
25         hintText: "Digite aqui",  
26       ),  
27     );  
28   }
```

Código 2.2: Exemplo de um *stateful widget* que representa um campo de texto.

O *stateful widget* é dividido em duas classes, ou em dois objetos distintos que se contemplam, como pode ser observado no código 2.2. A primeira classe, a qual estende de

StatefulWidgets é imutável e também podem receber variáveis via construtor, mas que são finais, assim como as propriedades das classes do tipo *StatelessWidget*'s. Para que hajam mudanças, um objeto do tipo *State* é adicionado e este será responsável pelas mudanças que poderão ocorrer no ciclo de vida do *StatefulWidget*.

A classe que estende *State* (chamada **CampoTextoState** no exemplo), criada no momento em que a função *StatefulWidget.createState()* é chamada, representa a informação lida pelo *widget* e que pode ser alterada durante o tempo de vida do mesmo. Além disso, o estado em si pode ter um ciclo de vida maior que o do seu próprio *widget*, mantendo-se em memória mesmo quando este é reconstruído.

O ciclo de vida de um *State* engloba as seguintes etapas principais:

1. criação, a partir da função *createState* e sua associação a um *BuildContext*, responsável por determinar a posição do *widget* que contém o *State* na árvore de *widgets*;
2. inicialização, com o uso da função *initState*, que também depende do contexto ao qual o *widget* está associado ou às suas propriedades;
3. construção, utilizando-se da função *build*, que pode ser chamada inúmeras vezes ao longo do ciclo de vida do *State*, como por exemplo, quando algum dos seus estados internos é alterado;
4. destruição, com o uso da função *dispose*, que é chamada quando o *State* é removido da árvore de *widgets*.

Processos intermediários podem ocorrer e funções como *didChangeDependencies* e *didUpdateWidget* são chamadas. Elas podem ser respectivamente utilizadas quando a inicialização do *State* envolve *InheritedWidget*'s ou quando quer-se responder às mudanças provocadas pelo *State* aos seus *widgets* associados.

No código exemplo 2.2 temos a **classe CampoTexto** que estende um *StatefulWidget* e possui um atributo **texto** que é passado como parâmetro para o construtor da classe. A classe possui o método **createState** que retorna um objeto do tipo **_CampoTextoState**, uma classe interna da classe **CampoTexto**.

Essa classe interna também possui um atributo **texto** que é inicializado com uma string vazia e pode ser modificado. O método **build** retorna um *widget* do tipo *TextField* que, por sua vez, recebe uma função anônima responsável por alterar o estado interno do objeto que, nesse caso, se trata do atributo **texto**, que recebe o valor passado como parâmetro. Esse método chama a função *setState* que é responsável por notificar o *framework* que o estado interno do objeto foi alterado e que o *widget* deve ser reconstruído.

Além disso, o *widget* do tipo *TextField* recebe como segundo parâmetro um objeto do tipo *InputDecoration* que define o rótulo do campo com o atributo **texto** da classe **CampoTexto** e uma dica do que o usuário pode fazer.

InheritedWidget

InheritedWidget é um tipo especial de *widget* que permite o envio de informações eficientemente entre os nós da árvore de *widgets* que compõe a aplicação. Isto é, a partir

de qualquer *widget* descendente de um *InheritedWidget*, pode-se recuperar dados sobre esse componente pai sem que haja a necessidade de repassar a informação por cada um dos nós que compõem o caminho entre os dois, como é demonstrado da imagem 2.3.

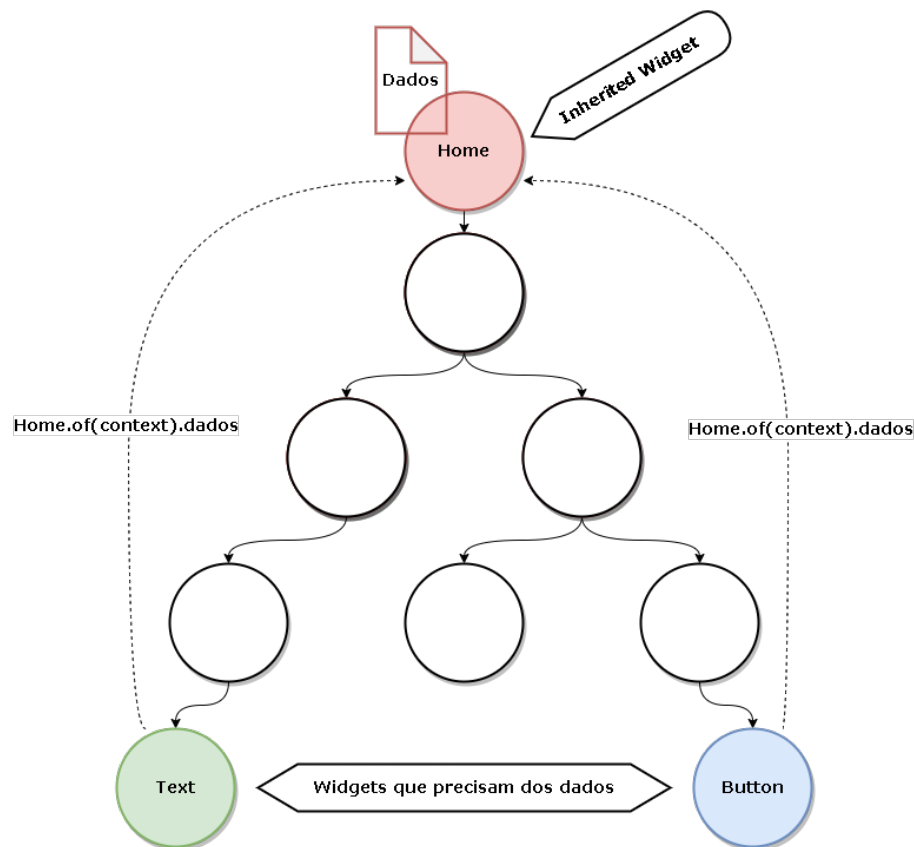


Figura 2.3: Exemplo de uma árvore de widgets com *InheritedWidget*. (Faust 2020) (Boelens 2018)

Neste exemplo, o estado é mantido pelo *widget* nomeado como **Home** e é acessado pelos *widgets* **Text** e **Button** por meio do método estático *Home.of(BuildContext context)* que, por sua vez, busca no contexto dessa sub-árvore, o *widget* mais próximo que seja do exato tipo *Home* e que seja uma extensão concreta da classe *InheritedWidget*.

2.2.3 Diferenças para outras tecnologias

A fazer.

2.2.4 Gerenciamento de estado

O gerenciamento de estado no *Flutter* não segue apenas uma arquitetura. Na verdade, o *framework* oferece uma série de opções para que o desenvolvedor possa escolher a que melhor se adapta ao seu projeto (Faust 2020). Algumas das principais opções são: *InheritedWidget*, *Provider*, *MobX BLoC* e *Redux*. Cada uma dessas opções possui suas

vantagens e desvantagens, sendo que algumas são mais adequadas para projetos pequenos e outras para projetos maiores. Anteriormente, descreveu-se o funcionamento do *InheritedWidget* e, mais a frente, serão apresentadas as principais características sobre o *Provider* e o *MobX*.

Antes, porém, é importante destacar que o *Flutter*, diferente de outros *frameworks* de desenvolvimento de aplicações *mobile*, como o *Android SDK* e o *iOS UIKit*, não segue uma perspectiva imperativa, onde é possível alterar o estado de um *widget* diretamente. Em vez disso, o *Flutter* segue uma perspectiva declarativa, onde a interface do usuário é alterado através de uma função que retorna um novo *widget*. Isso significa que, ao alterar o estado de um *widget*, o *framework* irá reconstruir esse *widget*, mesmo que isso aconteça a cada novo *frame* da aplicação. O processo pode ser custoso, mas o *Flutter* é rápido o suficiente para isso e existem vantagens associadas, principalmente o benefício de determinar como a interface deve ser construída dado um estado específico e mantê-la assim até que haja a necessidade de uma reconstrução dessa interface.

2.2.5 *Provider*

Provider é um *wrapper*, isto é, um encapsulamento que envolve o *InheritedWidget* com o objetivo de simplificar seu uso e, com isso, facilitar o gerenciamento de estados em aplicações *Flutter*.

Ao utilizar o *Provider*, pode-se encapsular o estado em uma nova classe que herda de *ChangeNotifier* ou se mistura a ela com o uso da palavra reservada *with*, do Dart. Dessa forma, essa classe ganha a capacidade de notificar os *widgets* que se inscrevem a ela quando o estado é alterado. Para isso, basta utilizar o método *notifyListeners()* em métodos que alteram esse estado.

Para injetar o estado aos *widgets* que tem interesse nele, utiliza-se o *ChangeNotifierProvider*. Esse *widget* é colocado acima dos *widgets* que devem receber a instância do *ChangeNotifier*. Por fim, para consumir o estado e responder às suas mudanças, utiliza-se um *widget* chamado *Consumer*, o qual é normalmente colocado ao redor do *widget* que deve ser atualizado quando o estado é alterado. Também pode-se utilizar o método *Provider.of(context)* para obter o estado ou alterá-lo.

2.2.6 *MobX*

MobX é uma biblioteca para gerenciamento de estado baseado em reatividade (P. Podila 2018). A reatividade, nesse contexto, é um conceito que permite que o estado de um *widget* mude automaticamente quando uma ação é realizada em outro local, como por exemplo, um outro *widget*. Para isso, utiliza-se a ideia de observabilidade a partir do *widget* que depende do estado a ser observado e, assim que este é alterado, o *widget* é notificado e reconstruído. Esse ciclo pode ser demonstrado na imagem 2.4.

A **ação** é responsável por gerar mudança no estado da aplicação. Ela pode ser disparada por uma interação do usuário com algum *widget*, como um botão, assim como pode ser resposta a um efeito colateral causado por uma outra ação anterior a esta e que gerou uma mudança de estado anterior, a qual foi consumida por um observador.

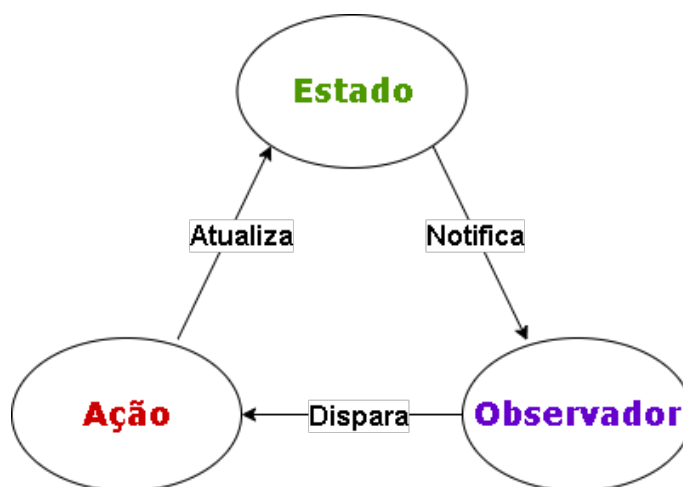


Figura 2.4: Ciclo básico do gerenciamento de estado com o MobX (P. Podila 2018) (Podila & Weststrate 2018)

O **observador**, por sua vez, pode ser um *widget*, que apresentará uma mudança visual ao usuário ou uma função responsável por lidar com a mutação desse estado, mas sem que haja, necessariamente, uma reconstrução da interface.

E, por fim, o **estado** é o valor que será observado e que, quando alterado, notificará os observadores. Esse estado pode ser uma variável, um objeto ou uma lista de objetos, por exemplo.

A notificação realizada pelo estado pode ocasionar efeitos colaterais em outros observadores, que por sua vez, podem disparar novas ações, gerando um ciclo de reatividade. A imagem 2.5 apresenta uma representação mais detalhada do ciclo com o *MobX*.

2.3 Persistência de Dados

A persistência de dados no contexto do Flutter é o processo de armazenamento em disco das informações relativas à aplicação ou ao seu usuário. A persistência de dados em um aplicativo móvel permite que o usuário possa acessar as informações mesmo quando não estiver conectado à internet ou mesmo quando o aplicativo estiver fechado. As principais formas de armazenamento de dados em uma aplicação Flutter são: salvamento em *Shared Preferences*, leitura e escrita de arquivos e salvamento em banco de dados *SQLite* (Time Flutter 2019).

A forma como as informações são salvas dependem da quantidade, complexidade e finalidade delas. Para pequenos conjunto de dados, o salvamento em chave-valor, utilizando o *Shared Preferences*, é uma boa opção (Time Flutter 2018c). Já para arquivos de texto, como por exemplo, um arquivo *JSON*, o salvamento em disco é uma boa opção, pois permite a leitura e escrita de dados de forma mais abrangente em relação ao tipo de arquivo salvo e sua utilidade posterior (Time Flutter 2018b). Por fim, para grandes conjuntos de dados, como por exemplo, uma lista de contatos, o *SQLite* é preferível, pois ele é uma biblioteca que implementa o mecanismo de um banco de dados relacional (*SQLite*

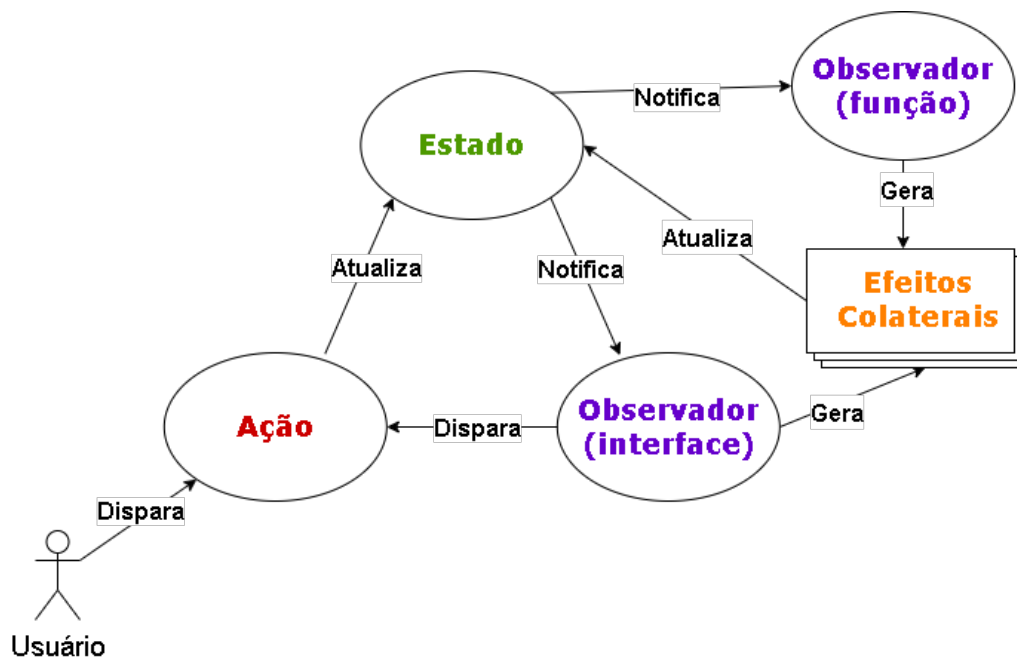


Figura 2.5: Ciclo detalhado do gerenciamento de estado com o MobX (P. Podila 2018) (Podila & Weststrate 2018)

Organization n.d.) que permite a criação de tabelas e a manipulação de dados de forma simples e rápida (Time Flutter 2018a) (SQLite Organization 2007b).

No contexto da aplicação, a quantidade de dados pode ser consideravelmente grande e estão bastante relacionados entre si, como descrito na 4.5.1. Por conta disso, a estratégia utilizada é a de salvar os dados em banco relacional, utilizando o *SQLite*.

2.3.1 Banco de Dados *SQLite*

Um banco de dados pode ser entendido como um conjunto organizado e integrado de dados ou informações que atendem a um conjunto de usuários (Heuser 2009) (Oracle Organization n.d.). Em geral, os bancos de dados tem o objetivo de armazenar dados de vários sistemas e/ou usuários diferentes e, por conta disso, utilizam-se de sistemas de gerenciamento de banco de dados (SGBD) modularizados (Heuser 2009) que, em alguns casos, exigem um servidor a parte para realizar os processos que recebem em uma arquitetura conhecida como cliente/servidor (Time SQLiteTutorial n.d.).

Para esse tipo de banco de dados, o foco está mais atrelado à escalabilidade e concorrência de dados, visto que inúmeros novos sistemas podem ser acoplados ao banco de dados pré-existente. No caso do *SQLite*, o foco está mais voltado para a economia, eficiência e simplicidade, ao passo em que ele atenderá a um único sistema ou aplicação que, nesse caso, se trata do dispositivo móvel e dos poucos usuários (em geral, apenas um), que o utilizarão (SQLite Organization 2007a).

2.3.2 Modelagem do banco de dados

A modelagem dos dados que serão salvos em um banco fornece uma representação abstrata da estrutura das informações que estarão contidas ali. Essa modelagem normalmente segue dois níveis de abstração diferentes que são criados de acordo com a finalidade e com o momento de desenvolvimento do projeto. O **modelo conceitual** normalmente possui informações menos detalhadas, mas que passam uma ideia geral da estrutura que será criada no banco de dados e os tipos de informação que estarão presente. Posteriormente, cria-se um **modelo lógico**, o qual já depende do SGBD utilizado e possui mais detalhes sobre os tipos de informações que possui e a forma como elas serão gravadas. (Heuser 2009).

A seguir, na figura 2.6, é apresentada um exemplo de modelagem conceitual do banco de dados, utilizando-se uma parte simplificada da aplicação e, na figura 2.7, a modelagem lógica para a mesma parte da aplicação.

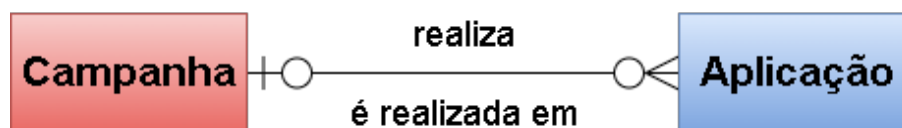


Figura 2.6: Modelagem conceitual de parte do banco de dados

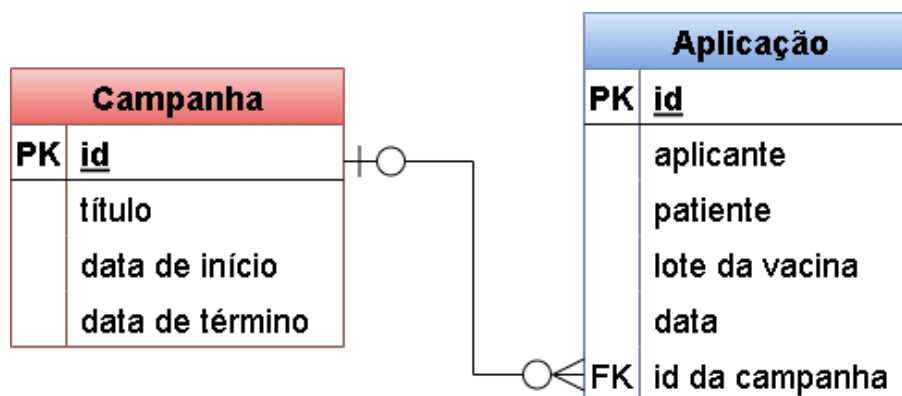


Figura 2.7: Modelagem lógico de parte do banco de dados

Observa-se que a notação utilizada no relacionamento entre as entidades 'Campanha' e 'Aplicação', para ambas as modelagens, segue o padrão conhecido como notação Engenharia de Informações. Nesse tipo de notação, a relação é demonstrada por uma linha e a cardinalidade é representada pelos símbolos nas extremidades dessa linha, ligados à entidade. Ademais, a denominação do relacionamento é dado pelas frases verbais que estão presentes na relação (Heuser 2009). No caso do relacionamento 'Campanha' e 'Aplicação', ele é representado por 'realiza' e 'é realizada em'.

3

Trabalhos relacionados

4

Nurse: uma aplicação para produtividade em vacinações

Nessa seção, os requisitos e casos de usos da aplicação serão descritos. Além disso, será apresentada a arquitetura escolhida para a aplicação, tanto no sentido das camadas hierárquicas de responsabilidade (Faust 2020), quanto na divisão de pastas e subpastas dos arquivos que compõem o código em módulos e, por fim, as telas do aplicativo e os pacotes utilizados para o desenvolvimento das funcionalidades serão mostrados.

O projeto foi desenvolvido utilizando a linguagem de programação *Dart*, com o *framework Flutter*. O seu repositório pode ser encontrado no *GitHub*, em <https://github.com/Dojak220/nurse>. A documentação, que inclui a versão dessa monografia em formato *pdf* e o seu respectivo projeto em *L^AT_EX* também pode ser encontrada no *Github*, em <https://github.com/Dojak220/nurse-docs>.

4.1 Requisitos do Sistema

Os requisitos da aplicação foram definidos de acordo com a análise feita a partir do problema identificado e com os objetivos descritos nas seções 1.2 e 1.3, respectivamente. Em suma, eles foram definidos com o intuito de suprir as necessidade de seus usuários com base naquilo que quer-se resolver. Esses requisitos, por sua vez, podem ser divididos em três tipos: funcionais, não funcionais e de domínio. Certos requisitos podem não estar claramente definidos como uma das três opções acima e, em alguns casos, um dado requisito pode se subdividir em mais requisitos menores e de tipos diferentes daquele que o originou (Sommerville 2007).

4.1.1 Requisitos Funcionais

Os requisitos funcionais (RF) são aqueles que descrevem as funcionalidades que o sistema deve possuir para atender às necessidades do usuário (Sommerville 2007). A seguir (tabela 4.1), são apresentados os requisitos funcionais do sistema e os detalhes para alguns deles podem ser encontrados no apêndice A.

4.1.2 Requisitos Não Funcionais

Os requisitos não funcionais (RNF) são aqueles que descrevem as características do sistema que não são diretamente implementadas como uma funcionalidade específica na aplicação, mas que são importantes para o seu funcionamento, como o nível de confiabilidade da aplicação, performance e segurança. Além disso, são esses requisitos que

Código	Requisito	Descrição
RF01	Cadastrar entidades	Permitir que o usuário cadastre novas entidades (sub-requisitos na tabela A.1)
RF02	Visualizar entidades cadastradas	Permitir que o usuário visualize as entidades cadastradas e seus detalhes (sub-requisitos na tabela A.2)
RF03	Editar entidades	Permitir que o usuário edite entidades já cadastradas seguindo fluxo análogo ao cadastro de nova entidade, porém, com campos pré-preenchidos
RF04	Gerar tabela de vacinações	Permitir que o usuário gere uma tabela de vacinações para exportação (sub-requisitos na tabela A.3)

Tabela 4.1: Requisitos funcionais da aplicação Nurse

apresentam as restrições inerentes à aplicação (Sommerville 2007). A seguir (tabela 4.2), são apresentados os requisitos não funcionais do sistema.

4.1.3 Casos de Uso

Uma das formas de representar um sistema, seu escopo e limites é através da identificação dos seus casos de uso e dos atores que interagem com o sistema (Schneider & Winters 2001). Estes atores podem ser desde pessoas a outras aplicações e os casos de uso descrevem o que os atores querem que o sistema realize. A seguir, na Figura 4.1, é apresentado o diagrama dos casos de uso da aplicação Nurse e, logo abaixo, a tabela 4.3 com a descrição dos atores envolvidos.

4.2 Arquitetura do Sistema

A definição de uma arquitetura para o desenvolvimento de um projeto é essencial para que o mesmo seja desenvolvido de forma organizada e eficiente, mas também tornar a sua manutenção mais simples. Em outras palavras, o objetivo em definir uma arquitetura é diminuir os recursos humanos e, consequentemente, financeiros necessários durante todo

Código	Requisito	Descrição
RNF01	Correta estrutura dos dados	Garantir que os dados fornecidos pelo usuário sejam aceitos e salvos apenas se seguirem as especificações e regras relativas a eles (por exemplo, CPF que deve seguir um conjunto de regras para ser considerado válido)
RNF02	Dados em planilha	Apenas os dados referentes às vacinações devem postos em uma planilha e esta deve seguir o formato apresentado na seção 1.2.1)
RNF03	Cadastros não podem ser apagados	As informações que forem salvas sobre qualquer uma das entidades pelo usuário não devem ser apagadas. Elas podem apenas ser alteradas (funcionalidade RF03)

Tabela 4.2: Requisitos não funcionais da aplicação Nurse

Ator	Descrição
Usuário	Usuário que utiliza a aplicação Nurse para gerenciar os dados das vacinações realizadas
Nurse	É a própria aplicação utilizada, mas que também realiza chamada para abertura e/ou compartilhamento de planilhas em outras aplicações

Tabela 4.3: Descrição dos atores envolvidos na aplicação Nurse

o ciclo de vida de um projeto, desde a sua concepção até a sua manutenção, passando pelo desenvolvimento de suas funcionalidades (Martin 2019).

Para se alcançar esse objetivo, pode-se utilizar de técnicas, como as supracitadas divisão em camadas de responsabilidade e a modularização, entre outras. Além disso, é importante construir um modelo do domínio-problema da aplicação para que as complexidades inerentes a ele sejam melhor controladas (Evans 2017). O domínio, ou seja, a

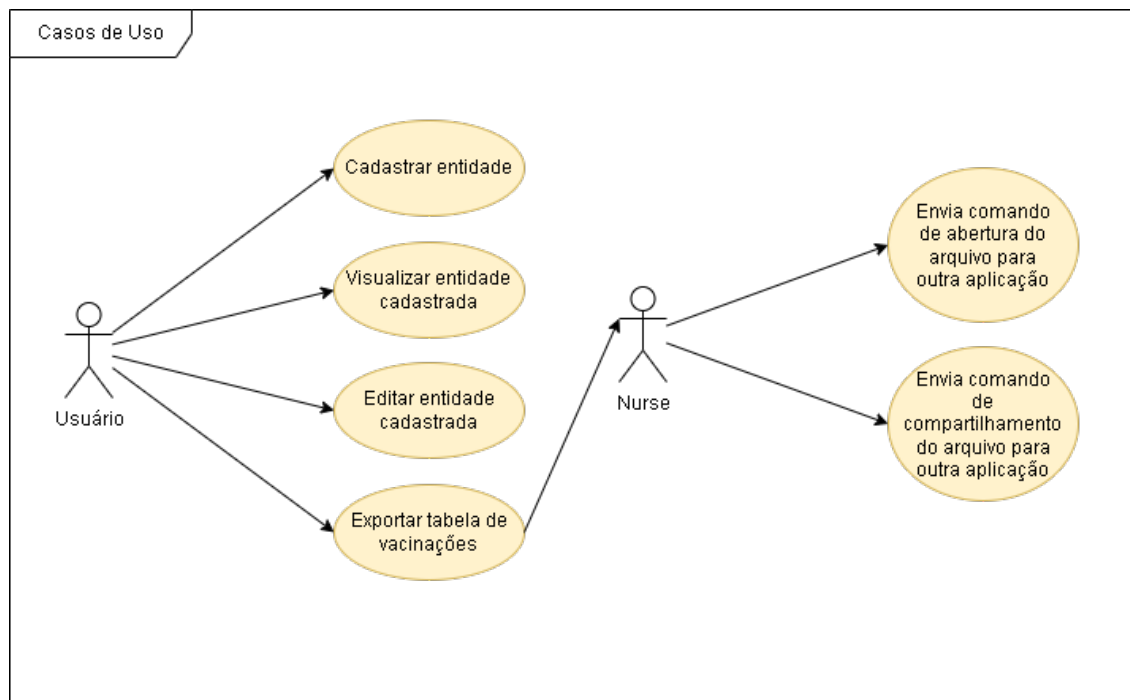


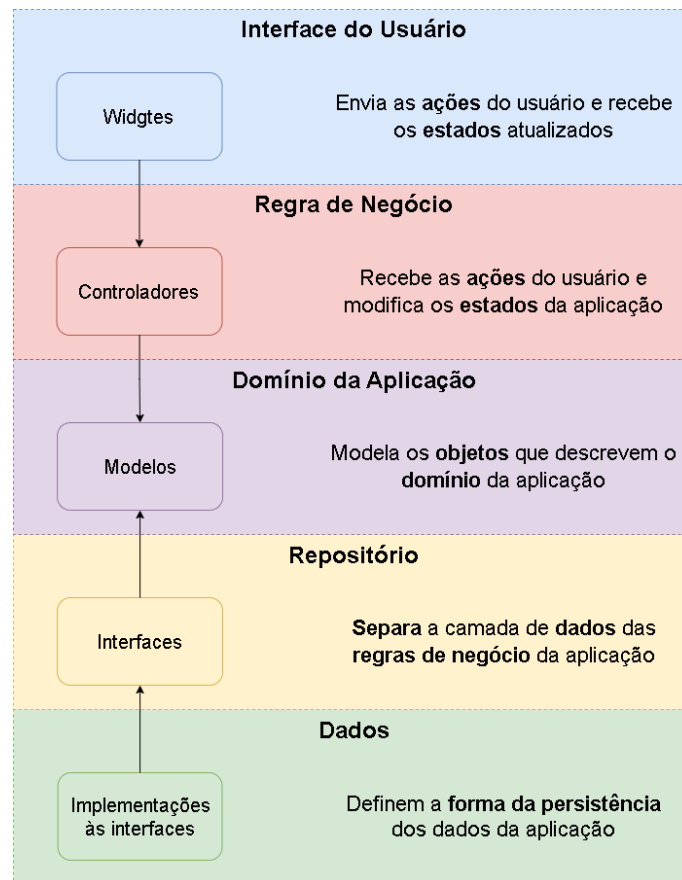
Figura 4.1: Diagrama de casos de uso da aplicação Nurse

área na qual a aplicação Nurse está inserida e qual problema quer-se resolver, engloba do todo o conjunto de ações necessárias para realizar uma vacinação, desde o cadastro das entidades envolvidas (paciente, aplicante, vacina etc...) até a posterior exportação dos dados coletados. O modelo criado para o banco de dados é usado para modelar as entidades envolvidas e a relação entre elas, já as classes desenvolvidas para criar os arquivos a serem exportados é um exemplo da implementação no código das ações necessárias dentro desse domínio-problema.

4.2.1 Camadas de Responsabilidade

A divisão em camadas hierárquicas de responsabilidade tem como objetivo dividir o projeto em partes menores, cada uma com uma responsabilidade específica e de forma a torná-las o mais independente possível entre elas (Faust 2020) e, dessa forma, diminuir o acoplamento e aumentar a coesão das classes. Um exemplo desse tipo de arquitetura é o sistema MVC (Model-View-Controller), que divide o projeto em três camadas: a **camada de modelo**, que contém as classes que representam as entidades do domínio-problema e gerencia os dados associados a elas; a **camada de visualização**, que contém as classes que representam as telas da aplicação e gerenciam as suas mudanças textuais e gráficas; e a **camada de controle**, que contém as classes que intermedeiam as outras duas camadas ao interpretar as ações do usuário e enviar comandos às classes do modelo e/ou da visualização para realizar quaisquer mudanças necessárias (Burbeck 1992).

Essa divisão pode ser realizada em n camadas. No caso da aplicação Nurse, dividiu-se o projeto em cinco, como mostrado na figura 4.2.

Figura 4.2: Camadas da aplicação **Nurse** e suas dependências

Interface do Usuário

Engloba todos os elementos (*widgets*) visuais e de interação com o usuários, como campos de texto ou botões, cores, imagens, ícones, etc... Essa camada é responsável por receber os comandos do usuário e enviar comandos para as outras camadas, como a camada de controle, para que as ações necessárias sejam realizadas. Além disso, é responsável por gerenciar as mudanças de estado da interface, como a mudança de cor de um botão ao ser clicado, por exemplo. As telas apresentadas na seção seguinte (4.3) mostram esses elementos com mais detalhes.

Camada de Regra de Negócio

A camada de controle recebe as ações tomadas pelo usuário na camada de interface e toma decisões a partir delas. Essas decisões são tomadas com base nas regras de negócio da aplicação (daí o nome da camada). Essas regras são implementadas em classes que fazem parte dessa camada, aqui nomeadas como **Controladores**.

Por exemplo, a classe *AddPatientFormController* é responsável por gerenciar as ações relacionadas às páginas de inserção de um novo paciente ou de edição de outro já cadastrado. Durante o preenchimento dos dados nos campos de texto e de seleção em lista

suspensa, essa classe salva os dados em um objeto que representa um paciente e, ao tentar salvar, verifica-se a validade dos dados inseridos pelo usuário e se esse cadastro já não havia sido realizado. Caso algum desses requisitos não seja atendido, um estado de erro é enviado à camada anterior para que esta apresente uma mensagem ao usuário, caso contrário, o objeto (um novo paciente, nesse exemplo) é enviado à camada responsável por salvá-lo no banco de dados e o usuário é redirecionado para a página de listagem de pacientes, onde poderá ver o novo item adicionado.

Vale ressaltar que essa validação, assim como o salvamento dos dados, não é responsabilidade dos controladores, então essas ações são delegadas a outras classes e/ou camadas e apenas os resultados que interessam aos controladores são retornados. A seguir, no trecho de código 4.1 são apresentados alguns detalhes da classe *AddPatientFormController*, ressaltando o uso do *MobX* e os métodos que conectam as camadas adjacentes à da regra de negócio.

```

1  class AddPatientFormController = _AddPatientFormControllerBase
2      with _$AddPatientFormController;
3
4  abstract class _AddPatientFormControllerBase extends
5      AddFormController
6      with Store {
7      // ...
8
9      @observable
10     ObservableList<Locality> localities =
11         ObservableList.of(List<Locality>.empty(growable: true));
12
13     @observable
14     ObservableList<PriorityCategory> categories =
15         ObservableList.of(List<PriorityCategory>.empty(growable: true));
16
17     @observable
18     PatientStore patientStore = PatientStore();
19
20     final Patient? initialPatientInfo;
21
22     _AddPatientFormControllerBase(this.initialPatientInfo) {
23         if (initialPatientInfo != null) {
24             patientStore.setInfo(initialPatientInfo!);
25         }
26
27         getLocalities();
28         getPriorityCategories();
29     }
30
31     @action
32     Future<List<Locality>> getLocalities() async {
33         final localities = await _localityRepository.getLocalities();
34
35         this.localities
36             ..clear()
37             ..addAll(await _localityRepository.getLocalities());

```

```

37
38     return localities;
39 }
40
41 @action
42 Future<List<PriorityCategory>> getPriorityCategories() async {
43     // ...
44 }
45
46 @override
47 Future<bool> saveInfo() async {
48     if (submitForm(formKey)) {
49         final PatientStore p = patientStore;
50
51         // ...
52
53     } else {
54         return false;
55     }
56 }
57
58 @override
59 Future<bool> updateInfo() async {
60     if (initialPatientInfo == null) return false;
61
62     if (submitForm(formKey)) {
63         final PatientStore p = patientStore;
64
65         // ...
66
67     } else {
68         return false;
69     }
70 }
71
72 // ...
73 }

```

Código 4.1: Trechos da classe *PatientFormController*

Inicialmente, na declaração da classe, utilizou-se o padrão sugerido na própria documentação do *mobx* no Flutter para geração automática de código por meio do pacote de desenvolvimento *mobx_codegen* (Time MobX.dart n.d.). Essa geração torna mais simples a utilização dos recursos do *MobX*, como os **Observables** e **Actions** e, por isso, foi utilizada. Adiciona-se à declaração, uma extensão à classe abstrata *AddFormController*, responsável por definir métodos comuns a todas as classes de controle de formulários de inserção de dados como, por exemplo, os métodos *saveInfo* e *updateInfo*.

Em seguida, podemos observar três propriedades observáveis, sendo duas listas e um objeto do tipo *PatientStore* que, por sua vez, também possui propriedades observáveis internamente. É nessa última que estão os valores sobre o paciente, enviados pelo usuário na interface, como mostra o trecho de código 4.2. Nesse exemplo são mostradas algumas propriedades e métodos responsável por atualizá-las.

24CAPÍTULO 4. NURSE: UMA APLICAÇÃO PARA PRODUTIVIDADE EM VACINAÇÕES

```
1 class PatientStore = _PatientStoreBase with _$PatientStore;
2
3 abstract class _PatientStoreBase with Store {
4     @observable
5     Locality? selectedLocality;
6
7     @observable
8     String? cns;
9
10    @observable
11    String? name;
12
13    ...outras propriedades
14
15    @action
16    void setLocality(Locality? value) => selectedLocality = value;
17
18    @action
19    void setCns(String value) => cns = value;
20
21    @action
22    void setName(String value) => name = value;
23
24    // ...outros metodos
25
26    @action
27    void setInfo(Patient patient) {
28        selectedLocality = patient.person.locality;
29        selectedBirthDate = patient.person.birthDate;
30        cns = patient.cns;
31        name = patient.person.name;
32
33        // ...
34    }
35
36    @action
37    void clearAllInfo() {
38        selectedLocality = null;
39        selectedBirthDate = null;
40        cns = null;
41        name = null;
42
43        // ...
44    }
45 }
```

Código 4.2: Trechos da classe *PatientStore*

De volta à listagem 4.1, logo após as propriedades observáveis, temos os métodos ***getLocalities*** e ***getPriorityCategories***, que agem como ações (marcadas com o ***@action***) e são responsáveis por buscar as informações necessárias para preencher as listas usadas para seleção de localidade e categoria de prioridade, respectivamente. Esses métodos são chamados no construtor da classe (simplificado nesse trecho de código), que recebe

como parâmetro um objeto do tipo *Patient*, que é utilizado para preencher os campos do formulário quando o usuário deseja editar um paciente já existente. Essa classe também possui os métodos *saveInfo* e *updateInfo* que são responsáveis por salvar e atualizar os dados do paciente no banco de dados, respectivamente.

Domínio da Aplicação

Nessa camada estão as classes que representam as entidades do domínio-problema da aplicação. Observa-se que essa camada não depende de nenhuma outra e pode ser considerada aquela que vai dirigir as implementações das demais (Evans 2017). Essas classes são definidas com base no mesmo modelo descrito, com mais detalhes, na seção 4.5.1. No trecho de código 4.3 a seguir, é possível ver como a classe *Vaccine*, que modela uma vacina presente no domínio do problema, é implementada.

```
1  class Vaccine implements GenericModel {
2      @Override
3      final int? id;
4      final String sipniCode;
5      final String name;
6      final String laboratory;
7
8      Vaccine({
9          this.id,
10         required String sipniCode,
11         required String name,
12         required String laboratory,
13     }) : sipniCode = sipniCode.trim(),
14         name = name.trim(),
15         laboratory = laboratory.trim() {
16         _validateVaccine();
17     }
18
19     void _validateVaccine() {
20         if (id != null) Validator.validate(ValidatorType.id, id!);
21         Validator.validateAll([
22             ValidationPair(ValidatorType.numericalString, sipniCode),
23             ValidationPair(ValidatorType.name, name),
24             ValidationPair(ValidatorType.name, laboratory),
25         ]);
26     }
27     ...
28 }
```

Código 4.3: Trecho da implementação da classe **Vaccine**

A partir dessa classe, podemos verificar aquilo que compõe essa camada. A classe em si representa o que seria a vacina aplicada pelo profissional da saúde dentro de um processo de vacinação a um paciente e, portanto, é uma entidade do domínio-problema. Interno a essa classe estão os atributos que representam as informações que a vacina possui, como o seu nome, seu código SIPNI e o laboratório que a produziu. Além disso, é possível ver que a classe implementa a interface *GenericModel*, que é uma interface

que define os atributos e métodos que devem ser implementados por todas as classes que representam entidades do domínio-problema. Essa interface é definida no arquivo *generic_model.dart* e pode ser vista no trecho de código 4.4 a seguir.

```

1  abstract class GenericModel {
2      final int? id;
3
4      GenericModel(this.id);
5
6      Map<String, dynamic> toMap();
7  }

```

Código 4.4: Interface **GenericModel**

A classe *Vaccine* também implementa os métodos *fromMap* e *toMap*, que são responsáveis por converter um objeto dessa classe em um *Map* e vice-versa. Esses métodos são utilizados pelos repositórios para salvar e recuperar os dados.

Há, também, um processo de validação realizado pelo **Validator**. Essa classe de suporte transita entre as camadas de domínio e de regra de negócio, pois ela é utilizada pelos *models* para garantir que as informações que estão tentando ser cadastradas são válidas de acordo com as regras que definem o domínio-problema. Caso haja algum erro, essa informação é devolvida enviada para a camada de regra de negócio.

Camada de Repositório

A camada de repositório separa a camada de dados da camada da regra de negócio (Faust 2020) (Bizzotto n.d.). Ela é responsável por abstrair a forma como os dados são armazenados, permitindo que a aplicação seja portada para diferentes bancos de dados ou para API's sem que seja necessário alterar a regra de negócio. Essa camada é composta por classes abstratas (4.5) que definem uma interface com os métodos a serem implementados pelas classes que utilizam-se dessa interface (4.7). A seguir, no trecho de código 4.5, é apresentada a interface **CampaignRepository** e sua implementação para uso de banco de dados, **DatabaseCampaignRepository**, pode ser vista no trecho de código 4.7, na seção 4.2.1.

```

1  abstract class CampaignRepository {
2      Future<int> createCampaign(Campaign campaign);
3      Future<int> deleteCampaign(int id);
4      Future<Campaign> getCampaignById(int id);
5      Future<Campaign> getCampaignByTitle(String title);
6      Future<List<Campaign>> getCampaigns();
7      Future<int> updateCampaign(Campaign campaign);
8  }

```

Código 4.5: Interface **CampaignRepository**

Pode-se observar que a classe **CampaignRepository** define seis funções relacionadas à criação, atualização, busca e remoção de campanhas de vacinação. Contudo, ela não determina como essas funções devem ser implementadas, apenas que as sejam. Dentro da camada de regra de negócio, a dependência, as classes recebem como dependência um objeto do tipo **CampaignRepository** e, dessa forma, não precisam se preocupar com a

forma como os dados serão armazenados, mas apenas garantir que a chamada às funções de armazenamento sigam a assinatura da interface definida. No trecho de código a seguir (4.6) é apresentado um exemplo de uso.

```
1 class AddCampaignFormController extends AddFormController {
2     final CampaignRepository _repository;
3
4     ...
5
6     AddCampaignFormController (
7         [CampaignRepository? campaignRepository])
8         : _repository = campaignRepository ??
9           DatabaseCampaignRepository () { ... }
10
11     ...
12 }
```

Código 4.6: Exemplo de uso da interface **CampaignRepository**

A classe de controle da adição de novas campanhas de vacinação recebe o repositório **CampaignRepository** como parâmetro no construtor. Dessa forma, ao criar-se uma instância desse controlador, é possível passar um repositório diferente, como um repositório que armazena os dados em um banco de dados relacional ou outro que utiliza um banco de dados não relacional. Como, nesse projeto, utiliza-se apenas uma forma de armazenamento, definiu-se como repositório padrão o **DatabaseCampaignRepository**, contudo, ainda é possível passar outros tipos de repositório como parâmetro sem a necessidade de mudar nenhuma linha de código dentro do controlador.

Camada de Dados

É nessa camada que serão realizadas as implementações dos repositórios definidos na camada de repositório. Nesse projeto, foi utilizado o banco de dados SQLite para armazenar os dados, como será melhor explicado nas seções 4.4.1 e 4.5.

No trecho de código abaixo (4.7) é apresentada a implementação do repositório **DatabaseCampaignRepository**. Pode-se observar que a classe implementa a interface **CampaignRepository** e, portanto, deve implementar todos os métodos definidos nessa interface (apenas dois dos métodos foram apresentados aqui, pois os outros possuem uma estrutura análoga a estes). Além disso, a classe herda de **DatabaseInterface** que, por sua vez, define métodos de acesso ao banco de dados que serão utilizados por todas as classes concretas de repositórios que realizam esse tipo de armazenamento.

Tanto a classe **DatabaseInterface** como a classe **DatabaseManager**, que é utilizada pela primeira, serão melhor descritas na seção 4.5.2.

```
1 class DatabaseCampaignRepository extends DatabaseInterface
2     implements CampaignRepository {
3     // ignore: constant_identifier_names
4     static const String TABLE = "Campaign";
5
6     DatabaseCampaignRepository ([DatabaseManager? dbManager])
```

```

7      : super(TABLE, dbManager);
8
9      @Override
10     Future<int> createCampaign(Campaign campaign) async {
11         final int result = await create(campaign.toMap());
12
13         return result;
14     }
15
16     @Override
17     Future<int> deleteCampaign(int id) async {
18         final int count = await delete(id);
19
20         return count;
21     }
22
23     ... // Demais implementacoes
24 }

```

Código 4.7: Implementação na classe **DatabaseCampaignRepository** da interface **CampaignRepository**

4.3 Telas

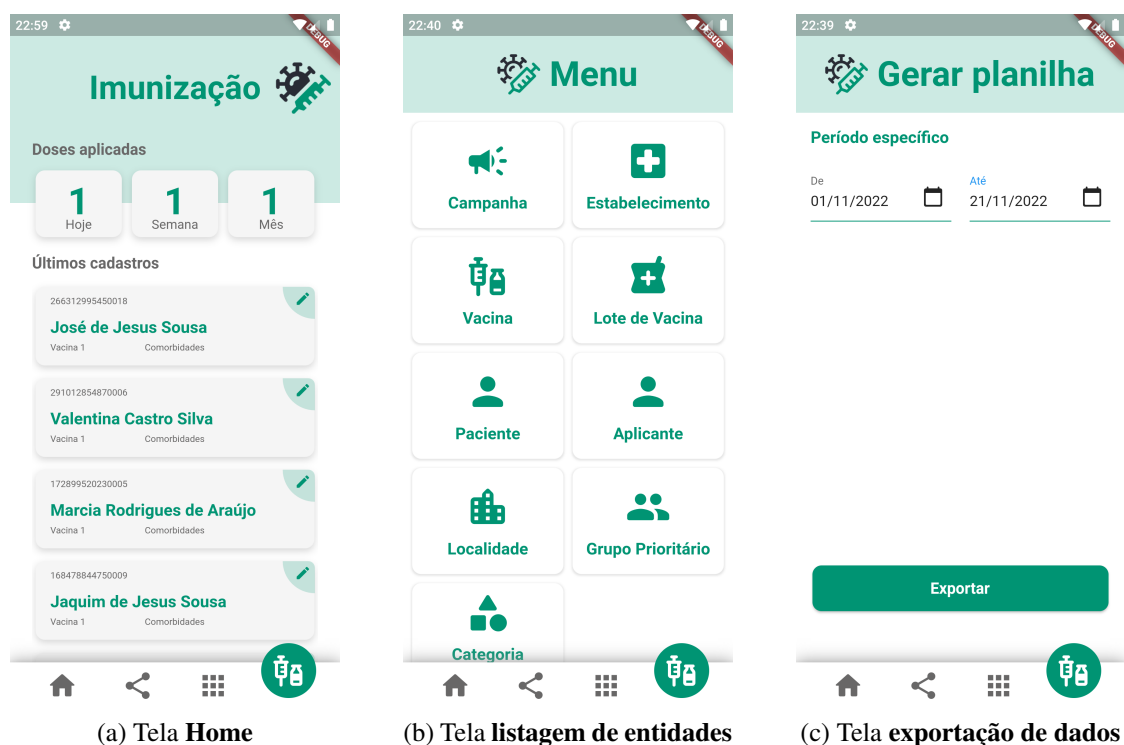
Nessa seção, serão apresentadas as principais telas desenvolvidas para o aplicativo **Nurse**. O design dessas páginas foram criados por Yasmim, utilizando o *software XD Adobe*. A seguir, serão apresentadas as telas **Home**, **listagem de entidades**, **listagem de campanhas de vacinação**, **cadastro de nova campanhas de vacinação** e a tela de **exportação de dados**.

Todas as páginas apresentadas possuem um cabeçalho em verde que apresenta o título da página, um ícone da aplicação e, em alguns casos, um botão para voltar à página anterior. Além disso, as principais páginas possuem um rodapé que serve como sistema de navegação entre elas. Essa barra de navegação possui quatro ícones e cada um leva a uma página diferente. O primeiro ícone leva à página **Home**, o segundo à página de **listagem de entidades**, o terceiro à página de **exportação de dados** e o último e principal, ao conjunto de páginas de **cadastro de vacinação**. Essa última será apresentada em detalhes na seção 5.2.2.

A página **Home** é a primeira página que o usuário visualiza quando abre o aplicativo. Nela, é possível ver o número de vacinas aplicadas no dia, na semana e no mês. Também é mostrado ao usuário a lista das últimas vacinas aplicadas e o paciente que as recebeu, além de outras informações sobre este.

A página **listagem de entidades** é a página que mostra ao usuário a lista de entidades cadastradas no aplicativo. Nela, é possível ver o nome da entidade e o ícone que a representa. Ao clicar em uma entidade, o usuário é levado à página de listagem daquela entidade.

Por fim, a página **exportação de dados** é a página que permite ao usuário exportar os dados do aplicativo referentes à vacinação para um arquivo **.csv**. Nela, o usuário pode

Figura 4.3: Principais telas da aplicação *Nurse*

filtrar o período que quer exportar os dados.

As telas da figura 4.4 apresentam um exemplo das telas que listam as campanhas cadastradas e o formulário de uma nova campanha. Na tela de listagem, pode-se ver duas campanhas cadastradas, um botão para novos cadastros na parte inferior na tela e um botão de edição em formato de lápis no canto superior direito de cada cartão. Esse cartão, por sua vez, apresenta as principais informações sobre aquela entidade. Por fim, quando quer-se editar um item, a página do formulário é iniciada com as informações previamente cadastradas daquele item.

4.4 Pacotes e Bibliotecas

Nessa seção, serão apresentados os pacotes e bibliotecas utilizados no desenvolvimento do sistema. A tabela B.1, presente no apêndice B, apresenta os pacotes utilizados no desenvolvimento do sistema e suas respectivas versões. Além dos pacotes principais, tem-se também os pacotes de desenvolvimento, que são utilizados apenas durante o desenvolvimento do sistema, como o *flutter_test* e o *mobx_codegen*, que são responsáveis pela criação de testes (ver seção 5.1) e geração de arquivos relacionados ao gerenciamento de estado (ver seções 2.2.6, 4.2.1 e 4.4.2), respectivamente.

Entre os pacotes e plugins utilizados, se destacam aqueles que estão relacionados com a persistência de dados (*sqlite_sqldcipher*), com o gerenciamento de estado (*mobx* e *flutter_mobx*), com a injeção de dependências (*provider*) e com a criação da planilha que



Figura 4.4: Telas de listagem e cadastro de entidade

será compartilhada (syncfusion_flutter_xlsio).

4.4.1 Plugins *sqflite_sqcipher* e *sqflite*

O plugin *sqflite* permite o uso do **SQLite** em aplicações *Flutter* (Roux 2017). O *sqflite_sqcipher*, por sua vez, adiciona ao primeiro a funcionalidade de criptografia do banco de dados através da biblioteca *SQLCipher* (Martos 2020).

4.4.2 Bibliotecas *mobx* e *flutter_mobx*

O pacote *mobx* é utilizado para a implementação do *MobX* 2.2.6 nas aplicações em *Dart/Flutter*. É com ele que o gerenciamento de estado segue o conceito de reatividade visto anteriormente, utilizando-se das suas principais classes: **Observable**, responsável por criar o estado reativo da aplicação; e **Action**, que definirá a função que muda esse estado. Além disso, para completar a tríade do *MobX*, este utiliza-se de um conjunto de reações, em forma de função, que são chamadas no momento em que o estado observado muda (P. Podila 2018).

Em adição, têm-se o pacote *flutter_mobx*, o qual é responsável por implementar um *widget* chamado **Observer**. Este, por sua vez, garante que o seu *widget* filho seja atualizado sempre que o estado relacionado a ele mude. O *Observer* também é um representante das reações do *MobX* que, nesse caso, reage atualizando a interface do usuário (P.

Podila 2019).

No desenvolvimento da classe **HomeController**, a qual gerencia o estado observado pelos componentes da página **Home**, utilizou-se as classes **Action**, para definir a função responsável por alterar o estado observável, e **ObservableList**, uma variante da classe **Observable** para listas.

```

1  class HomeController {
2      final ApplicationRepository applicationRepository;
3
4      final applications = ObservableList<Application>.of(
5          List<Application>.empty(growable: true),
6      );
7
8      late final fetchApplications = Action(getApplications);
9
10     HomeController() : applicationRepository =
11         DatabaseApplicationRepository() {
12         fetchApplications();
13     }
14
15     Future<List<Application>> getApplications() async {
16         final result = await applicationRepository.getApplications();
17         applications.clear();
18         applications.addAll(result.reversed);
19
19         return applications;
20     }
21
22     /*...*/
23 }

```

Código 4.8: Uso do *MobX* na classe **HomeController**

A propriedade *applications* recebe, inicialmente, um **ObservableList** vazio do tipo *Application* e a função *fetchApplications* foi definida como a ação que modifica o estado observável, isto é, a lista de aplicações. A ação é chamada no construtor da classe **HomeController** para que a lista de aplicações seja preenchida assim que a classe for instanciada. Essa lista, por sua vez, é preenchida quando a busca realizada pela classe **ApplicationRepository** no banco de dados finaliza com sucesso. A função *fetchApplications* é chamada, também, toda vez que o usuário finaliza um novo cadastro de aplicação de vacina e é redirecionado novamente à tela inicial **Home**.

```

1  class Home extends StatelessWidget {
2      /*...*/
3
4      @override
5      Widget build(BuildContext context) {
6          return Scaffold(
7
8              /*...*/
9
10             floatingActionButton: VaccinationButton(
11                 newPage: "/vaccinations/new",

```

```

12         onCallback: () => context.read<HomeController>().
13         fetchApplications(),
14     ),
15
16     /*...*/
17
18 );
19 }
20 }

```

Código 4.9: Uso do *MobX* no *widget Home*

No código que define a classe **Home**, mostrada parcialmente no trecho de código 4.9, têm-se o *widget Scaffold*, o qual possui a propriedade *floatingActionButton*, que recebe o *widget VaccinationButton*. A sua propriedade *onCallback* recebe uma função que chama *fetchApplications* da classe **HomeController**, para que a lista de aplicações seja atualizada. A forma como o estado é injetado na classe **Home** é explicado adiante, na 4.4.3.

4.4.3 Pacote *provider*

Como descrito anteriormente, na 2.2.5, o estado pode ser injetado em qualquer *widget* por meio do *Provider*. Sendo assim, utilizou-se o *Provider* nesse projeto não para gerenciamento do estado diretamente, mas para injetar o estado gerenciado pelo *mobx* em qualquer *widget* da aplicação. Para isso, foi utilizado o pacote *provider* (R. Rousselet e Time Flutter 2019).

Desse pacote, utilizou-se duas estratégias principais:

- **Provider.of<T>(context)**: injeta a classe T a partir do contexto passado via parâmetro. O Provider, então, busca na árvore de *widgets* acima do *widget* atual a instância mais próxima da classe T (R. Rousselet e Time Flutter 2019). A seguir, no trecho de código 4.10, tem-se um exemplo de uso dessa estratégia.
- **context.read<T>().fn()**: assim como o anterior, utiliza-se do contexto para buscar a classe T desejada e, em seguida, faz uma chamada da função denominada *fn* no exemplo, mas não passa a observar as mudanças de estado que ocorrem nessa classe, diferentemente da função *watch* (R. Rousselet e Time Flutter 2019). Um exemplo do seu uso foi apresentado no trecho de código 4.9, no qual *fetchApplications* representa a função *fn* aqui descrita.

```

1 class Appliers extends StatelessWidget {
2     const Appliers({Key? key}) : super(key: key);
3
4     @override
5     Widget build(BuildContext context) {
6         final controller = Provider.of<AppliersPageController>(context);
7         /*...*/
8
9         return EntityList<Applier>(

```



```
10     title: "Aplicantes",
11     controller: controller,
12     /*...*/
13 );
14 }
15 }
```

Código 4.10: Uso do *Provider* no widget **Appliers**

4.4.4 Biblioteca Excel (XlsIO)

Utilizando-se do pacote *syncfusion_flutter_xlsio*, foi possível gerar um arquivo *.xlsx* a partir de uma lista de aplicações de vacina. Para isso, criou-se uma classe chamada **ExcelService** que possui dois métodos públicos: *shareExcelFile* e *openExcelFile*. Cada um desses métodos é chamado quando o usuário segue o fluxo apresentado na figura 5.8 e descrito em detalhes na seção 5.2.3.

O método *shareExcelFile* cria um arquivo *.xlsx* e permite que o usuário o compartilhe, enquanto o método *openExcelFile* cria um arquivo *.xlsx* e o abre no aplicativo de planilhas do usuário. A seguir, na figura 4.5, tem-se o fluxo de execução de ambos os métodos dentro da classe **ExcelService**.

A ordem de execução dos métodos segue a ordem das setas numeradas. Cada um dos métodos é descrito a seguir:

- **Etapas 1:** os métodos *shareExcelFile* e *openExcelFile* recebem como parâmetros a lista das aplicações cadastradas e o período de tempo que o usuário deseja exportar ou visualizar. Estas informações são, então, organizadas em um mapa de datas e aplicações e enviadas ao método *_generateSheetData*
- **Etapas 2 e 3:** o método *_generateSheetData* recebe como parâmetro o mapa de datas e aplicações. A partir dos métodos *_setTemplateForWorksheet* e *_setDataInWorksheet*, um objeto do tipo *Workbook* é, respectivamente estruturado e preenchido com os dados recebidos. Esse *Workbook* é, então transformado em uma lista de bytes por meio de seu método interno *Workbook.saveAsStream()*. Por fim, cada agrupamento de bytes é retornado em um novo mapa, onde a chave é a data das aplicações e o valor é um outro mapa de doses aplicadas e agrupamento de bytes.
- **Etapas 4:** o método *_generateSheetData* retorna o agrupamento de bytes para as funções *shareExcelFile* ou *openExcelFile*, a depender de qual fluxo está sendo executado, e cada um desses agrupamentos será processado individualmente, como se segue.
- **Etapas 5:** para o método *openExcelFile*, será verificado a quantidades de agrupamentos retornados. Caso seja apenas um, o arquivo será criado a partir desse agrupamento e passará ao próximo passo. Caso contrário, o fluxo desse método será finalizado e os dados serão redirecionados ao método *shareExcelFile*. Já para este último, o fluxo é direto e cada conjunto de bytes separados por datas de aplicação e doses aplicadas será salvo em um arquivo.

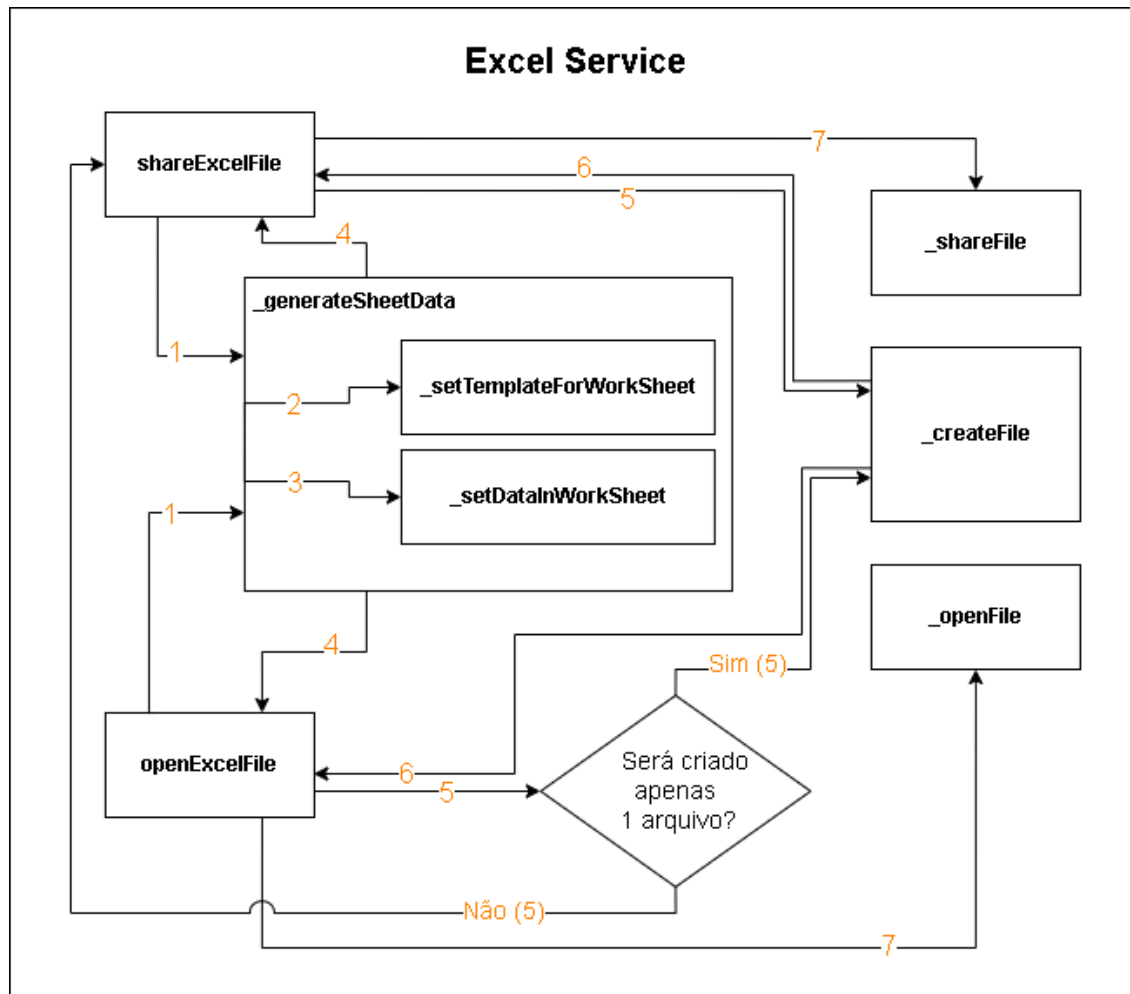


Figura 4.5: Descritivo de fluxo dos métodos da classe **ExcelService**

- **Etapa 6:** considerando que o fluxo dos métodos *shareExcelFile* e *openExcelFile* foi o mesmo, ou seja, a criação dos arquivos na função *_createFile*, o(s) arquivo(s) recebido(s) será(ão) enviados de volta para as funções principais.
- **Etapa 7:** Por fim, o método *shareExcelFile* chama a função *_shareFile* para compartilhar o arquivo gerado, enquanto o método *openExcelFile* chama a função *_openFile* para abrir o arquivo gerado. Ambas as funções são descritas a seguir.

A função *_shareFile* recebe como parâmetro o arquivo gerado e, a partir da biblioteca *share_plus* e do seu método *Share.shareFiles*, permite que o usuário compartilhe o arquivo gerado (Flutter Community 2020). A função *_openFile*, por sua vez, também recebe o arquivo gerado de forma análoga à primeira função e, a partir da biblioteca *open_file* e do seu método *OpenFile.open*, realiza a chamada a uma aplicação que possa abrir um arquivo no formato *.xlsx*.

4.5 Persistência de Dados

A persistência de dados, como descrita na seção 2.3, é realizada por meio de um banco de dados relacional, o *SQLite*. A biblioteca *sqflite* é utilizada para a comunicação com o banco de dados (Roux 2017). A estrutura do banco de dados da aplicação como ela foi implementada é descrita nesta seção.

4.5.1 Diagrama de Classes

Foram definidas 9 entidades (ou classes) para a aplicação **Nurse**, as quais foram agrupadas em 3 macro-grupos: **Paciente**, **Vacinação** e **Infraestrutura**. Cada uma dessas entidades possui um identificador único, chamada chave primária (do inglês, *Primary Key* (PK)) e algumas delas possuem uma ou mais chaves estrangeiras (do inglês, *Foreign Key* (FK)) (Heuser 2009). Além disso, cada entidade possui um ou mais atributos, que são apresentados na figura 4.6 e descritos a seguir.

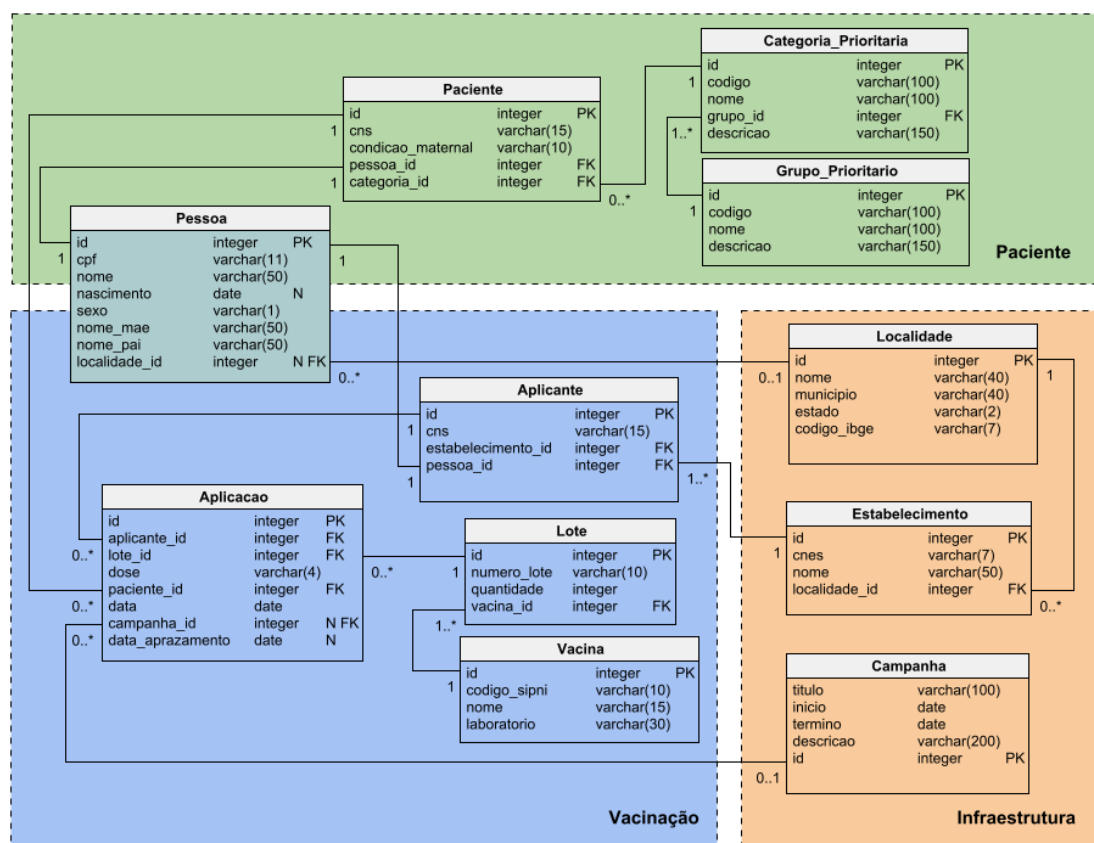


Figura 4.6: Diagrama de classe para a aplicação Nurse

- **Pessoa:** é a entidade que representa uma pessoa, seja ela um paciente ou um profissional de saúde. Essa entidade possui os atributos **nome**, **cpf**, **data de nascimento**, **sexo**, **nome da mãe**, **nome do pai** e **localidade** onde reside. Essa entidade faz parte de dois grupo distintos: grupo 'Paciente' e grupo 'Aplicação', pois ela pode estar associada a um paciente ou a um aplicante da vacina.
- **Grupo Paciente**
 - **Paciente:** é a entidade que representa um paciente. Essa entidade possui os atributos **cns** (número do Cartão Nacional de Saúde) e **condição maternal**. Além disso, a entidade **Paciente** possui duas chaves estrangeiras para as entidades **Pessoa** e **Categoria Prioritária**.
 - **Categoria Prioritária:** é a entidade que representa uma categoria prioritária do paciente. Essa entidade possui os atributos **nome**, **código** e **descrição**. Além disso, a entidade **Categoria Prioritária** possui uma chave estrangeira para o seu **Grupo Prioritário**.
 - **Grupo Prioritário:** é a entidade que representa um conjunto de categorias prioritárias. Um exemplo de grupo é 'Faixa Etária' e as categorias desse grupo são subconjuntos de idades (pessoas com mais de 60 anos, pessoas com menos de 18 anos etc...). Essa entidade possui os atributos **nome**, **código** e **descrição**.
- **Grupo Infraestrutura**
 - **Localidade:** é a entidade que representa uma localidade, seja ela uma comunidade ou uma cidade. Essa entidade possui os atributos **nome da localidade**, **município**, **estado** e **código do IBGE**.
 - **Estabelecimento:** é a entidade que representa um estabelecimento de saúde. Essa entidade possui os atributos **nome** e **CNES** (Cadastro Nacional de Estabelecimentos de Saúde). Além disso, a entidade **Estabelecimento** possui uma chave estrangeira para a entidade **Localidade**.
 - **Campanha:** é a entidade que representa uma campanha de vacinação. Essa entidade possui os atributos **título**, datas de **início** e **término** da campanha e sua **descrição**.
- **Grupo Vacinação**
 - **Aplicante:** é a entidade que representa o profissional de saúde que realizou a aplicação da vacina no paciente. Essa entidade possui o atributo **cns**, assim como o paciente. Além disso, a entidade **Aplicante** possui duas chaves estrangeiras para as entidades **Pessoa** e **Estabelecimento**.
 - **Vacina:** é a entidade que representa o agente imunizante que será aplicado no paciente. Essa entidade possui os atributos **nome** da vacina, seu **código SI-PNI** e o **laboratório** do fabricante.

- **Lote:** é a entidade que representa um lote de vacinas. Essa entidade possui os atributos **número do lote** e **quantidade de vacinas** no lote. Além disso, a entidade **Lote** possui uma chave estrangeira para a entidade **Vacina**.
- **Vacinação:** é a entidade central da aplicação. Ela representa todo o conjunto de informações que estão associadas ao ato de vacinar. Seus atributos representam essa centralidade. São eles: **Aplicante**, **Lote da vacina**, **Paciente** e **Campanha de vacinação**, as quais são todas chaves estrangeiras para outras tabelas. Além disso, a entidade **Vacinação** possui os atributos **dose da vacina**, **data de aplicação** e **data prazo** para próxima aplicação.

Alguns desses atributos são obrigatórios, como o **cpf** e o **nome**, já outros podem ser deixados nulos, como é o caso do campo **sexo**, todos da tabela **Pessoa**. Além dessa, a tabela **Aplicação** também possui dois atributos chamados anuláveis, que são o identificador da tabela **Campanha** e o atributo **data de aprazamento**.

4.5.2 Uso do Banco de Dados

O banco de dados é gerenciado pela classe **DatabaseManager**. Essa classe possui o método público **tryToInit**, que é chamado logo na inicialização do app **Nurse** e é responsável por abrir o banco de dados ou criá-lo quando este ainda não existe. Uma vez aberto, a classe **DatabaseManager** é responsável por gerenciar as requisições de inserção, atualização e/ou consulta realizadas pelo usuário. Isso é possível por meio da interface **DatabaseInterface** e do objeto **Database**, que define os métodos de acesso ao banco de dados e envia os comandos sql para o banco, respectivamente.

5

Experimentos e Resultados

Nesse capítulo são apresentados os resultados obtidos com o desenvolvimento da aplicação Nurse, os quais serão divididos em duas partes:

- i os testes automatizados desenvolvidos para garantir que as funções individuais criadas realizem o que espera-se delas e;
- ii os fluxos de utilização das funcionalidades da aplicação, segundo os cenários de uso definidos na seção 4.1.3.

Nas seções que se seguem, são apresentados os detalhes referentes a cada uma dessas partes.

5.1 Testes Unitários

Durante todo o ciclo de desenvolvimento de uma aplicação e mesmo durante sua manutenção, quando esta já encontra-se em produção, os testes às suas funcionalidades são essenciais. Eles garantem que o objetivo pelo qual o sistema foi desenvolvido seja alcançado e que a experiência do usuário não seja negativamente impactada por possíveis erros que possam acontecer durante o seu uso (Kong et al. 2018).

Os testes manuais das funcionalidades, no entanto, são limitados. Eles podem consumir muito tempo para serem executados, são propensos a erros humanos e quanto mais complexo o sistema, mais difícil é a sua manutenção e execução. Testes automatizados, por outro lado, resolvem todas essas questões, o que garante a performance do sistema e dos desenvolvedores que o mantêm (Time Flutter 2016) (Kong et al. 2018). Não apenas isso, mas os testes em geral e, principalmente os automatizados, são responsáveis por manter a aplicação flexível a mudanças, acrescentam ao sistema uma documentação sobre o comportamento esperado ao utilizá-lo e tornam os códigos mais reutilizáveis (Martin 2007) (Martin 2011).

Os código unitários (Time Flutter 2016) (Martin 2011) verificam uma pequena parte da aplicação, que pode ser uma função ou parte de uma e como como propósito definir um comportamento padrão para aquela funcionalidade específica. A seguir, é apresentado o trecho de código 5.1, referente ao teste da criação de um objeto que representa uma campanha de vacinação e, logo após, nas Figuras 5.1 e 5.2, são apresentados o resultado obtido com a execução de todos os testes unitários e a cobertura de código obtida, respectivamente.

```
1 void main() {  
2   setUp(() {  
3     final Campaign validCampaign = Campaign(  
4       id: 1,  
5       title: "Campaign Title",
```

```

6      description: "Campaign Description",
7      startDate: DateTime(2022),
8    );
9  });
10
11  group('criacao de nova instancia de campanha', () {
12    test('deve criar uma instancia valida', () {
13      expect(validCampaign, isA<Campaign>());
14      expect(validCampaign.id, 1);
15      expect(validCampaign.title, "Campaign Title");
16      expect(validCampaign.description, "Campaign Description");
17      expect(validCampaign.startDate, DateTime(2022));
18      expect(validCampaign.endDate, DateTime(2023));
19    });
20  });
21 }

```

Código 5.1: Teste unitário para verificação da correta criação de uma instância de *Campaign*

```

$ flutter test
00:11 +247: All tests passed!

```

Figura 5.1: Resultado dos testes unitários da aplicação Nurse.

LCOV - code coverage report

Current view: top level		Hit	Total	Coverage
Test: lcov.info	Lines:	1107	1218	90.9 %
Date: 2022-12-01 23:09:23	Functions:	0	0	-

Directory	Line Coverage	Functions
models/infra	100.0 % 129 / 129	- 0 / 0
models/patient	100.0 % 205 / 205	- 0 / 0
models/vaccination	95.0 % 171 / 180	- 0 / 0
repositories/database	95.8 % 23 / 24	- 0 / 0
repositories/database/infra	81.5 % 66 / 81	- 0 / 0
repositories/database/patient	84.7 % 144 / 170	- 0 / 0
repositories/database/vaccination	89.4 % 220 / 246	- 0 / 0
utils	81.4 % 149 / 183	- 0 / 0

Generated by: [LCOV version 1.15.alpha0w](#)

Figura 5.2: Cobertura de testes na aplicação Nurse.

Para a realização dos testes unitários, utilizou-se um dos pacotes nativos do *framework Flutter*, chamado *flutter_test*, que foi adicionado como uma das dependências de desenvolvimento da aplicação. Então, criou-se um arquivos de testes para cada classe relacionada aos modelos da aplicação e aos seus repositórios. Para cada um dessas classes, foram criados testes unitários para verificar se as funções públicas estão realizando o que se espera delas. E, para rodar os testes, bastou executar o comando *flutter test* no terminal.

Já para verificação de cobertura de testes do código, utilizou-se a flag *-coverage* no comando de testes, o que forneceu um relatório de cobertura de código. Em posse desse relatório e utilizando-se da ferramenta *lcov*, foi possível gerar uma página HTML (do inglês, HyperText Markup Language - Linguagem de Marcação de Hipertexto) com a visão geral da cobertura dos testes, como mostrado na Figura 5.2 e com os detalhes de cada arquivo com código associado (Source Forge 2019).

5.2 Fluxos de Telas

As seções mostradas a seguir representam os principais fluxos de telas da aplicação Nurse. A primeira seção apresenta o fluxo de telas para o cadastro de uma nova entidade, como um paciente ou uma vacina, por exemplo. A segunda seção também apresenta o cadastro de uma entidade, mas a vacinação é a entidade central da aplicação e possui um fluxo ligeiramente diferente das demais entidades. A seção seguinte mostra o fluxo para uma edição de entidade e, por fim, a última seção compreende o fluxo de telas para a geração e exportação de uma planilha de vacinações em um dado período.

5.2.1 Fluxo de cadastro de entidade

O fluxo de cadastro de entidade é composto por três telas, sendo a primeira tela responsável por apresentar a lista de tipos de entidade presentes na aplicação (Aplicante, Vacina, Lote de Vacina...). Ao selecionar uma dessas opções, uma lista com todos os cadastros individuais dessa entidade é apresentada na segunda tela. Por fim, na última tela, o usuário terá um formulário com os campos obrigatórios e opcionais para o cadastro da entidade selecionada e um botão de salvamento.

Caso haja algum erro de validação dos campos, o usuário será informado por meio de uma mensagem de erro que aparecerá logo abaixo do campo ao qual o erro se refere. Com o formulário preenchido, o usuário poderá tentar salvar a nova entrada e, caso o salvamento seja bem sucedido, será redirecionado para a tela de listagem da entidade cadastrada. Caso contrário, será informado por meio de uma mensagem de erro que aparecerá no centro da tela, em um *pop-up*.

Vale ressaltar que parte do fluxo de cadastro de uma nova campanha foi apresentado na seção 4.3, Figura 4.4. Sendo assim, o exemplo apresentado nessa seção (Figuras 5.3 e 5.4) é semelhante àquele, assim como às demais telas de cadastro de entidades. A exceção é a tela de cadastro de uma nova vacinação, que será apresentada a seguir, na seção 5.2.2.

5.2.2 Fluxo de cadastro de vacinação

Como essa é uma entidade que engloba todas as outras, seu formulário é mais extenso e, por conta disso, decidiu-se subdividir esse formulário em mais partes. Além disso, o formulário é acessado não pela listagem de entidades, mas por meio de um botão de destaque presente na barra de navegação da aplicação.

O fluxo inicia-se nesse ponto, quando o usuário aperta no botão com o símbolo da aplicação. Em seguida, é exibido um formulário com múltiplas páginas e dois botões

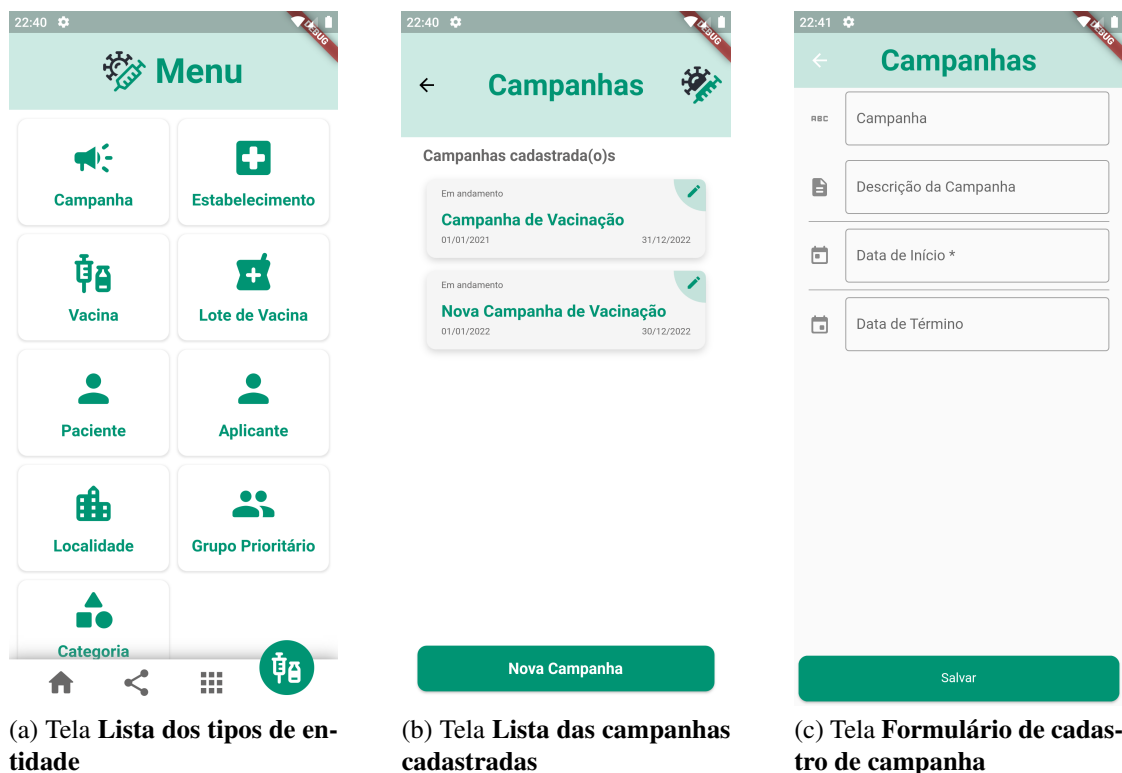


Figura 5.3: Fluxo de cadastro de nova campanha (parte 1)

na parte inferior da tela para navegação entre as páginas. Na última página o botão de salvar aparece e o usuário pode salvar o novo registro de vacinação, seguindo as regras de preenchimento apresentadas na seção anterior (seção 5.2.1).

Ao fim do cadastro, o usuário é redirecionado à tela **Home**, onde será possível ver a nova vacinação cadastrada na lista de vacinações e os cartões com a contagem de doses aplicadas aumentar.

As Figuras 5.5 e 5.6 apresentam o fluxo descrito acima.

5.2.3 Fluxo de exportação

O fluxo de exportação inicia-se na tela que pode ser acessada ao apertar no ícone de exportação, presente na barra de navegação da aplicação. Nessa tela, o usuário pode escolher o período que deseja buscar vacinações para exportação.

Assim que o período é escolhido no calendário, o botão de exportar é habilitado e, ao apertá-lo, duas opções são apresentadas: abrir o arquivo ou compartilhá-lo.

Esse fluxo é apresentado nas Figuras 5.7 e 5.8.

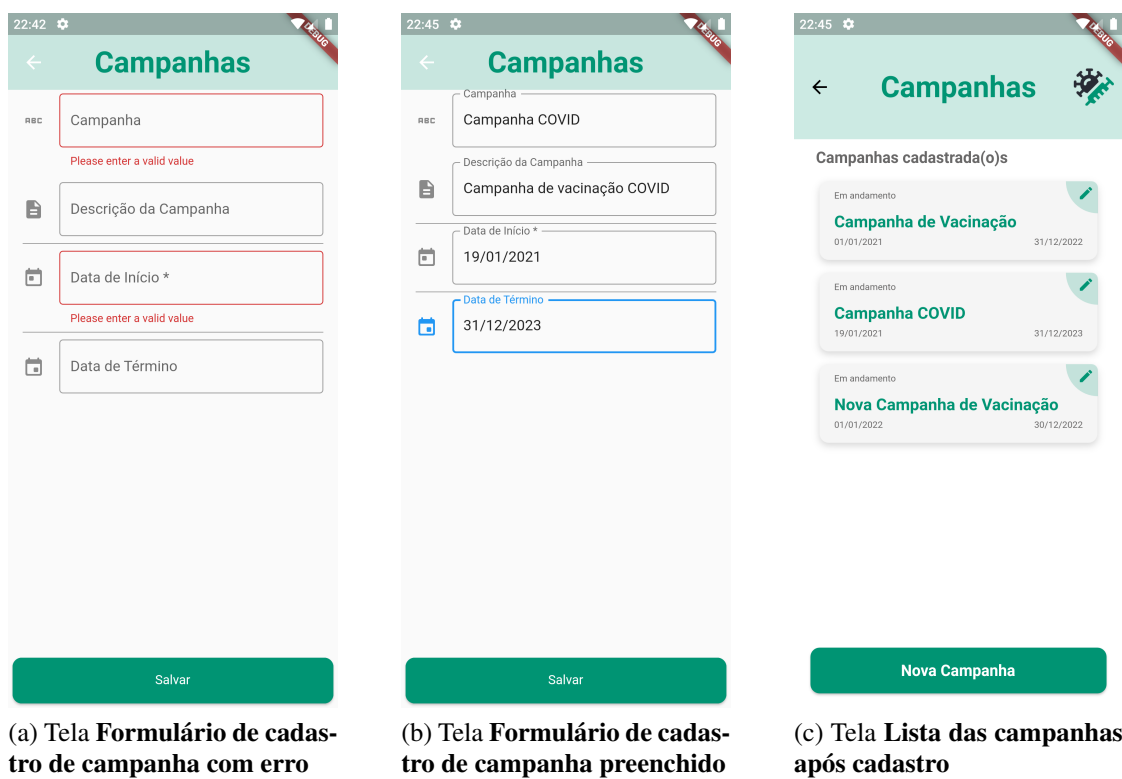
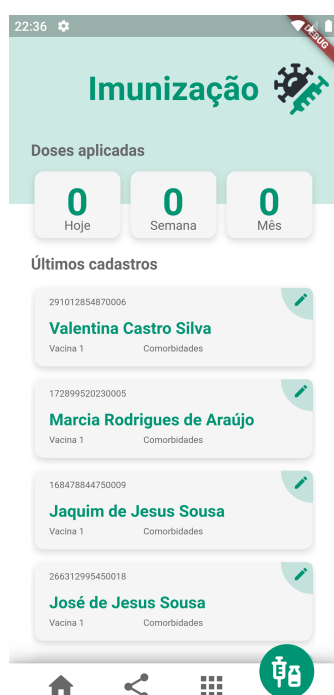
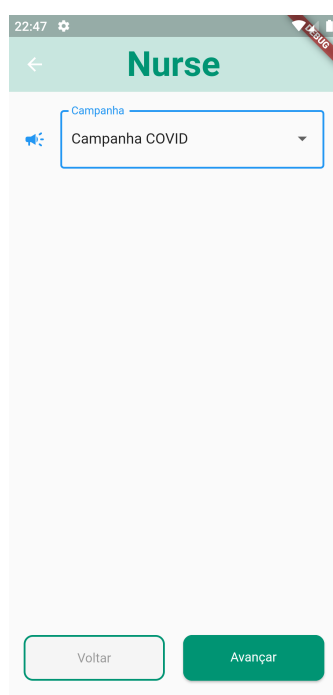


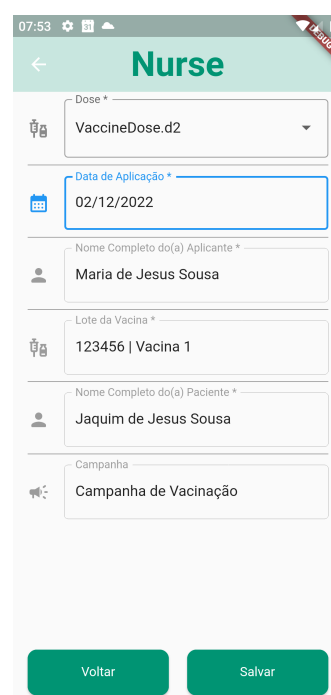
Figura 5.4: Fluxo de cadastro de nova campanha (parte 2)



(a) Botão para formulário de vacinação em destaque na barra de navegação



(b) Primeira tela do formulário



(c) Última tela do formulário

Figura 5.5: Fluxo de cadastro de nova vacinação (parte 1)

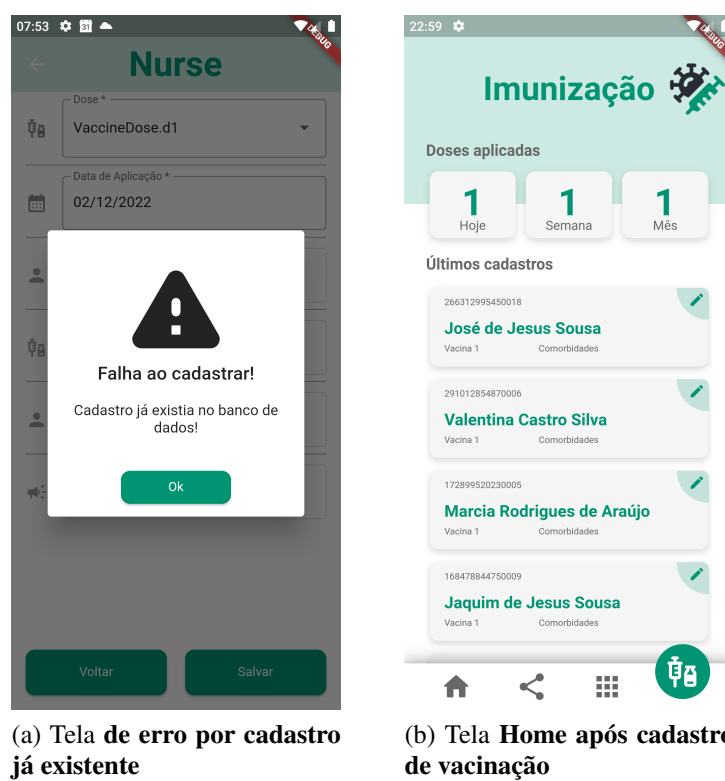
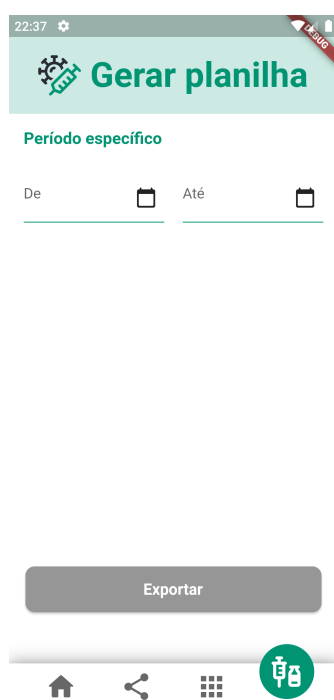
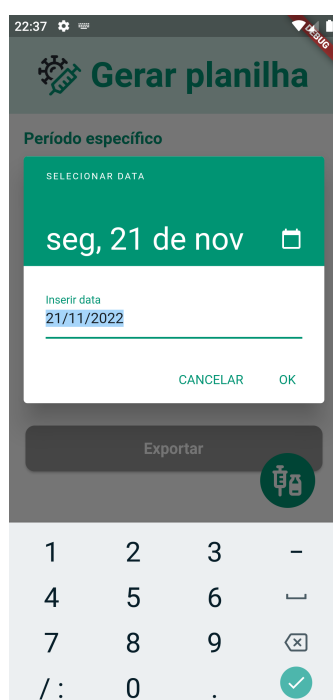


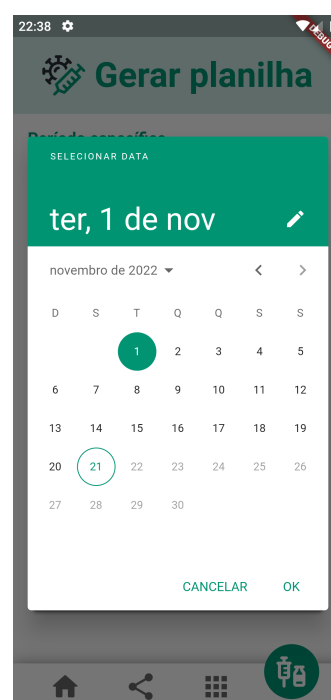
Figura 5.6: Fluxo de cadastro de nova vacinação (parte 2)



(a) Tela exportação inicialmente



(b) Calendário 1 para escolha do período de exportação



(c) Calendário 2 para escolha do período de exportação

Figura 5.7: Fluxo de exportação da planilha de vacinação (parte 1)

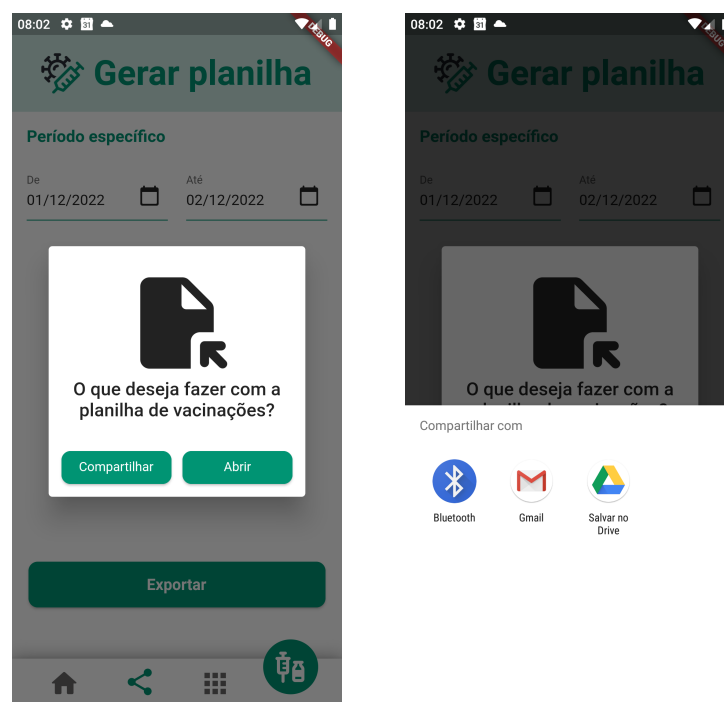


Figura 5.8: Fluxo de exportação da planilha de vacinação (parte 2)

6

Conclusão

Referências Bibliográficas

- Bizzotto, Andrea (n.d.), 'Flutter app architecture: The repository pattern', Disponível em: <https://codewithandrea.com/articles/flutter-repository-pattern/>. Acesso em: 01 de Junho de 2022.
- Boelens, Didier (2018), 'Widget - state - context - inheritedwidget', Disponível em: <https://www.didierboelens.com/2018/06/widget-state-context-inheritedwidget/>. Acesso em: 09 de Novembro de 2022.
- Burbeck, Steve (1992), 'Applications programming in smalltalk-80: how to use model-view-controller (mvc)', p. 2.
- Collins, James (2018), 'flutter_dotenv | dart package', Disponível em: https://pub.dev/packages/flutter_dotenv. Acesso em: 15 de Novembro de 2022.
- crazecoder (2018), 'open_file | dart package', Disponível em: https://pub.dev/packages/open_file. Acesso em: 15 de Novembro de 2022.
- Evans, Eric (2017), *Domain-Driven Design: Atacando as complexidades no coração do software*, 3ª edição, Alta Books.
- Faust, Sebastian (2020), Using Google s Flutter Framework for the Development of a Large-Scale Reference Application, Tese de doutorado, Hochschulbibliothek der Technischen Hochschule Köln.
- Flutter Community (2020), 'share_plus | dart package', Disponível em: https://pub.dev/packages/share_plus. Acesso em: 15 de Novembro de 2022.
- Heuser, Carlos Alberto (2009), *Projeto de banco de dados: Volume 4 da Série Livros didáticos informática UFRGS*, Bookman Editora.
- Kong, Pingfan, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé & Jacques Klein (2018), 'Automated testing of android apps: A systematic literature review', *IEEE Transactions on Reliability* **68**(1), 45–66.
- Martin, Robert C (2007), 'Professionalism and test-driven development', *Ieee Software* **24**(3), 32–36.

- Martin, Robert C (2011), *Código limpo: habilidades práticas do Agile software*, Alta Books.
- Martin, Robert C (2019), *Arquitetura Limpa: O guia do artesão para estrutura e design de software*, Alta Books Editora.
- Martos, David (2020), 'sqflite_sqldcipher | pacote dart', Disponível em: https://pub.dev/packages/sqflite_sqldcipher. Acesso em: 14 de Novembro de 2022.
- Oracle Organization (n.d.), 'O que é um banco de dados?', Disponível em: <https://www.oracle.com/br/database/what-is-database/>. Acesso em: 12 de Novembro de 2022.
- P. Podila (2018), 'mobx | pacote dart', Disponível em: <https://pub.dev/packages/mobx>. Acesso em: 11 de Novembro de 2022.
- P. Podila (2019), 'flutter_mobx | pacote dart', Disponível em: https://pub.dev/packages/flutter_mobx. Acesso em: 11 de Novembro de 2022.
- Podila, Pavan & Michel Weststrate (2018), *MobX Quick Start Guide: Supercharge the client state in your React apps with MobX*, Packt Publishing Ltd.
- R. Rousselet e Time Flutter (2019), 'provider | pacote dart', Disponível em: <https://pub.dev/packages/provider>. Acesso em: 11 de Novembro de 2022.
- Roux, Alexandre (2017), 'sqflite | pacote dart', Disponível em: <https://pub.dev/packages/sqflite>. Acesso em: 14 de Novembro de 2022.
- Schneider, Geri & Jason P Winters (2001), *Applying use cases: a practical guide*, Pearson Education.
- Sommerville, Ian (2007), *Software Engineering*, Pearson Education Limited.
- Source Forge (2019), 'Lcov - the ltp gcov extension', Disponível em: <https://codewithandrea.com/articles/flutter-repository-pattern/>. Acesso em: 02 de Dezembro de 2022.
- SQLite Organization (2007a), 'Appropriate uses for sqlite', Disponível em: <https://www.sqlite.org/whentouse.html>. Acesso em: 12 de Novembro de 2022.
- SQLite Organization (2007b), 'Features of sqlite', Disponível em: <https://www.sqlite.org/features.html>. Acesso em: 12 de Novembro de 2022.
- SQLite Organization (n.d.), 'Sqlite | home', Disponível em: <https://www.sqlite.org/index.html>. Acesso em: 13 de Novembro de 2022.
- Time Dart (2012), 'intl | dart package', Disponível em: <https://pub.dev/packages/intl>. Acesso em: 25 de Novembro de 2022.

- Time Dart (2013), 'path | dart package', Disponível em: <https://pub.dev/packages/path>. Acesso em: 15 de Novembro de 2022.
- Time Dart (2022), 'Dart overview', Disponível em: <https://dart.dev/overview>. Acesso em: 04 de Novembro de 2022.
- Time Flutter (2016), 'Testing flutter apps', Disponível em: <https://docs.flutter.dev/testing>. Acesso em: 02 de Dezembro de 2022.
- Time Flutter (2017), 'path_provider | dart package', Disponível em: https://pub.dev/packages/path_provider. Acesso em: 15 de Novembro de 2022.
- Time Flutter (2018a), 'Persist data with sq_lite', Disponível em: https://docs.flutter.dev/cookbook/persistence/sq_lite. Acesso em: 12 de Novembro de 2022.
- Time Flutter (2018b), 'Read and write files', Disponível em: <https://docs.flutter.dev/cookbook/persistence/reading-writing-files>. Acesso em: 12 de Novembro de 2022.
- Time Flutter (2018c), 'Store key-value data on disk', Disponível em: <https://docs.flutter.dev/cookbook/persistence/key-value>. Acesso em: 12 de Novembro de 2022.
- Time Flutter (2019), 'Persistence', Disponível em: <https://docs.flutter.dev/cookbook/persistence>. Acesso em: 12 de Novembro de 2022.
- Time Flutter (2020), 'cupertino_icons | dart package', Disponível em: https://pub.dev/packages/cupertino_icons. Acesso em: 15 de Novembro de 2022.
- Time Kineapps (2020), 'flutter_archive | dart package', Disponível em: https://pub.dev/packages/flutter_archive. Acesso em: 15 de Novembro de 2022.
- Time MobX.dart (n.d.), 'Core concepts', Disponível em: <https://mobx.netlify.app/concepts>. Acesso em: 27 de Novembro de 2022.
- Time SQLiteTutorial (n.d.), 'What is sq_lite', Disponível em: https://www.sqlitetutorial.net/what-is-sq_lite/. Acesso em: 12 de Novembro de 2022.
- Time Syncfusion (2020), 'syncfusion_flutter_xlsio | dart package', Disponível em: https://pub.dev/packages/syncfusion_flutter_xlsio. Acesso em: 15 de Novembro de 2022.

A

Requisitos funcionais detalhados

Código	Requisito	Descrição
RF01	Cadastrar entidades	Permitir que o usuário cadastre novas entidades
RF01-01	Cadastrar Vacinações	Fornecer para preenchimento os campos obrigatórios aplicante, estabelecimento de aplicação, vacina, lote, informações do paciente (pré-cadastrado ou não), dose e data da aplicação da vacina e os campos opcionais campanha da vacinação que está sendo efetuada e data da próxima vacina
RF01-02	Cadastrar Campanhas	Fornecer para preenchimento os campos obrigatórios título e datas de início de fim e o campo opcional descrição da campanha
RF01-03	Cadastrar Estabelecimentos	Fornecer para preenchimento os campos obrigatórios nome, CNES e endereço do estabelecimento de saúde
RF01-04	Cadastrar Vacinas	Fornecer para preenchimento os campos obrigatórios nome, fabricante e código da vacina
RF01-05	Cadastrar Lotes de Vacina	Fornecer para preenchimento os campos obrigatórios código, nome e quantidade de doses da vacina no lote
RF01-06	Cadastrar Pacientes	Fornecer para preenchimento os campos obrigatórios CNS, CPF, nome, data de nascimento, localidade, categoria prioritária e condição maternal e os campos opcionais sexo e nomes do pai e da mãe do paciente
RF01-07	Cadastrar Aplicantes	Fornecer para preenchimento os campos obrigatórios CNS, CPF, nome, localidade e estabelecimento de saúde de atuação e os campos opcionais data de nascimento, sexo e nomes do pai e da mãe do paciente

RF01-08	Cadastrar Localidades	Fornecer para preenchimento os campos obrigatórios código do IBGE, nome, cidade e estado da localidade
RF01-09	Cadastrar Grupos Prioritários	Fornecer para preenchimento o campo obrigatório código e os campos opcionais nome e descrição do grupo prioritário
RF01-10	Cadastrar Categorias Prioritárias	Fornecer para preenchimento os campos obrigatórios* código e grupo prioritário pertencente e os campos opcionais nome e descrição da categoria prioritária
RF01-11	Botão de salvamento	Fornecer para preenchimento os campos obrigatórios* botão para salvamento dos dados cadastrados em todos os formulários

Tabela A.1: Requisitos funcionais da aplicação Nurse: detalhes do requisito RF01

Código	Requisito	Descrição
RF02	Visualizar entidades cadastradas	Permitir que o usuário visualize as entidades cadastradas e seus detalhes
RF02-01	Visualizar Vacinações	Apresentar CNS, nome e grupo do paciente e vacina aplicada
RF02-02	Visualizar Campanhas	Apresentar título, datas de início de fim e status da campanha
RF02-03	Visualizar Estabelecimentos	Apresentar nome, CNES e endereço do estabelecimento de saúde
RF02-04	Visualizar Vacinas	Apresentar nome, fabricante e código da vacina
RF02-05	Visualizar Lotes de Vacina	Apresentar código, nome e quantidade de doses da vacina no lote
RF02-06	Visualizar Pacientes	Apresentar CNS, nome, categoria prioritária e condição maternal do paciente
RF02-07	Visualizar Aplicantes	Apresentar CNS, nome e estabelecimento de saúde do profissional
RF02-08	Visualizar Localidades	Apresentar código do IBGE, nome e endereço da localidade
RF02-09	Visualizar Grupos Prioritários	Apresentar código, nome e descrição do grupo prioritário
RF02-10	Visualizar Categorias Prioritárias	Apresentar código, nome e descrição da categoria prioritária

Tabela A.2: Requisitos funcionais da aplicação Nurse: detalhes do requisito RF02

Código	Requisito	Descrição
RF04	Gerar tabela de vacinações	Permitir que o usuário gere uma tabela de vacinações para exportação
RF04-01	Escolher período de tempo	Permitir que o usuário escolha o período desejado para coleta dos dados sobre vacinação a serem exportados
RF04-02	Escolher o que fazer com o arquivo gerado	Permitir que o usuário escolha entre abrir o arquivo com a planilha gerada ou exportá-la utilizando para isso alguma aplicação externa compatível com a escolha, a depender do caso

Tabela A.3: Requisitos funcionais da aplicação Nurse: detalhes do requisito RF04

B

Pacotes e versões utilizados

Pacote	Versão	Descrição
cupertino_icons	^ 1.0.2	Repositório de ícones utilizados pelos <i>wid-gets</i> do Cupertino (Time Flutter 2020)
intl	^ 0.17.0	Repositório utilizado para formatação de datas nessa aplicação (Time Dart 2012)
sqflite_sqcipher	^ 2.1.0	Extensão ao <i>sqflite</i> (Roux 2017) que adiciona senha ao acesso o banco de dados (Martos 2020)
path	^ 1.8.0	Biblioteca para manipulação de caminhos em multi-plataformas (Time Dart 2013)
provider	^ 6.0.3	Encapsula o <i>InheritedWidget</i> , tornando-o reutilizável e mais fácil de usar (R. Rousselet e Time Flutter 2019)
flutter_archive	^ 5.0.0	Biblioteca para criação e extração de arquivos ZIP (Time Kineapps 2020)
flutter_dotenv	^ 5.0.2	Carrega configurações para aplicação em tempo de execução (Collins 2018)
flutter_mobx	^ 2.0.5	Integração do MobX aos <i>wid-gets</i> do <i>Flutter</i>
mobx	^ 2.0.7	Biblioteca para gerenciamento de estado na aplicação
syncfusion_flutter_xlsio	^ 20.2.50-beta	Pacote para criação de arquivos Excel (.xlsx) (Time Syncfusion 2020)
path_provider	^ 2.0.11	<i>Plugin</i> para encontrar locais no sistema em múltiplas plataformas (Time Flutter 2017)

open_file	^ 3.2.1	<i>Plugin</i> para abertura de arquivos do sistema em múltiplas plataformas (crazecoder 2018)
share_plus	^ 4.4.0	<i>Plugin</i> para compartilhamento de arquivos a partir da aplicação (Flutter Community 2020)

Tabela B.1: Pacotes utilizados no projeto e suas respectivas versões