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, 13 de novembro 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, 13 de novembro 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, LaTeX, 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, LaTeX, Thesis Preparation, Technical Reports.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

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

1.4 Estrutura 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 mudam 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`.

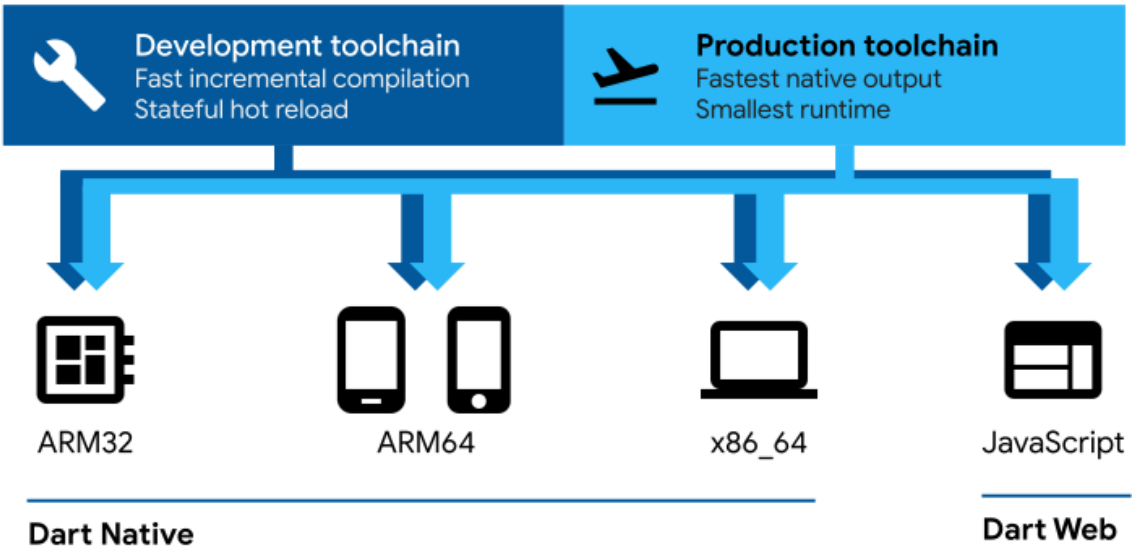


Figura 2.1: Plataformas de compilação do Dart (Time Dart 2022)

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 *deadlock*s;
- Uso de ***isolates***: todo o código em *Dart* roda dentro de *isolates*, o qual possui um 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* me 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*

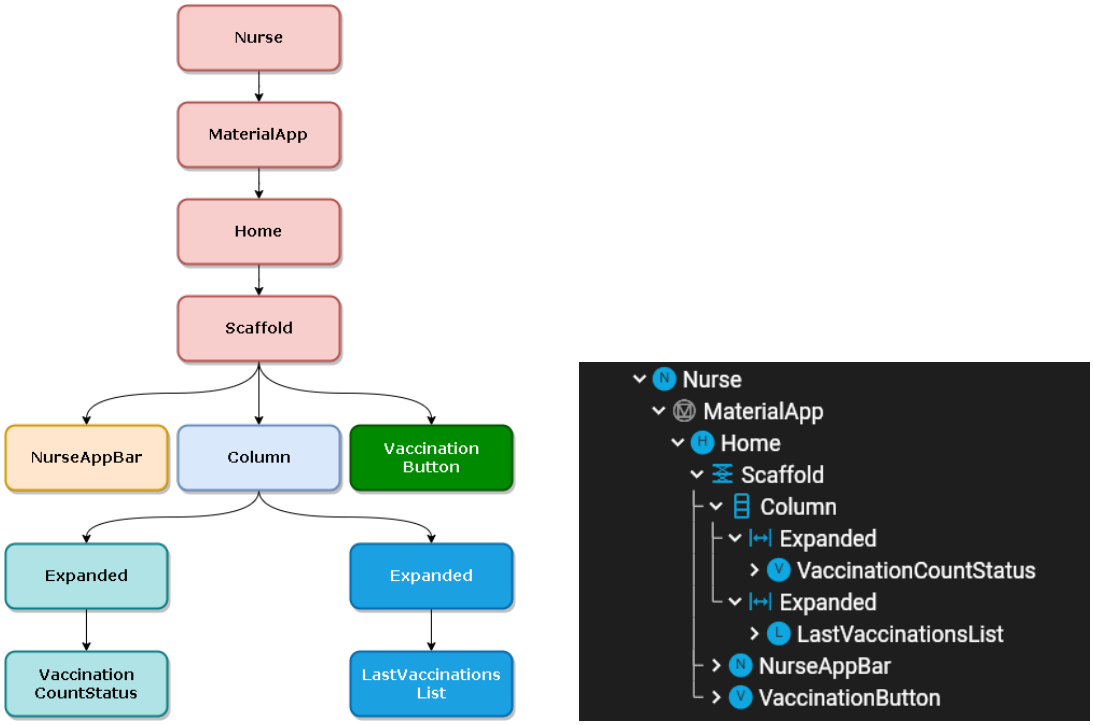
A fazer.

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.

```
class BotaoPrincipal extends StatelessWidget {
  final String texto;

  const BotaoPrincipal(Key? key, this.texto) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return TextButton(
      onPressed: () => {print("Clicou no botao principal")},
      child: Text(texto),
    );
  }
}
```

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.

```
class CampoTexto extends StatefulWidget {
  final String texto;

  const CampoTexto({Key? key, required this.texto}) : super(key: key)
  ;

  @override
  State<CampoTexto> createState() => _CampoTextoState();
}

class _CampoTextoState extends State<CampoTexto> {
  String texto = "";

  @override
  Widget build(BuildContext context) {
    return TextField(
      onChanged: (value) {
        setState(() {
          texto = value;
        });
        print(texto);
      },
      decoration: InputDecoration(
        labelText: widget.texto,
        hintText: "Digite aqui",
      ),
    );
  }
}
```

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

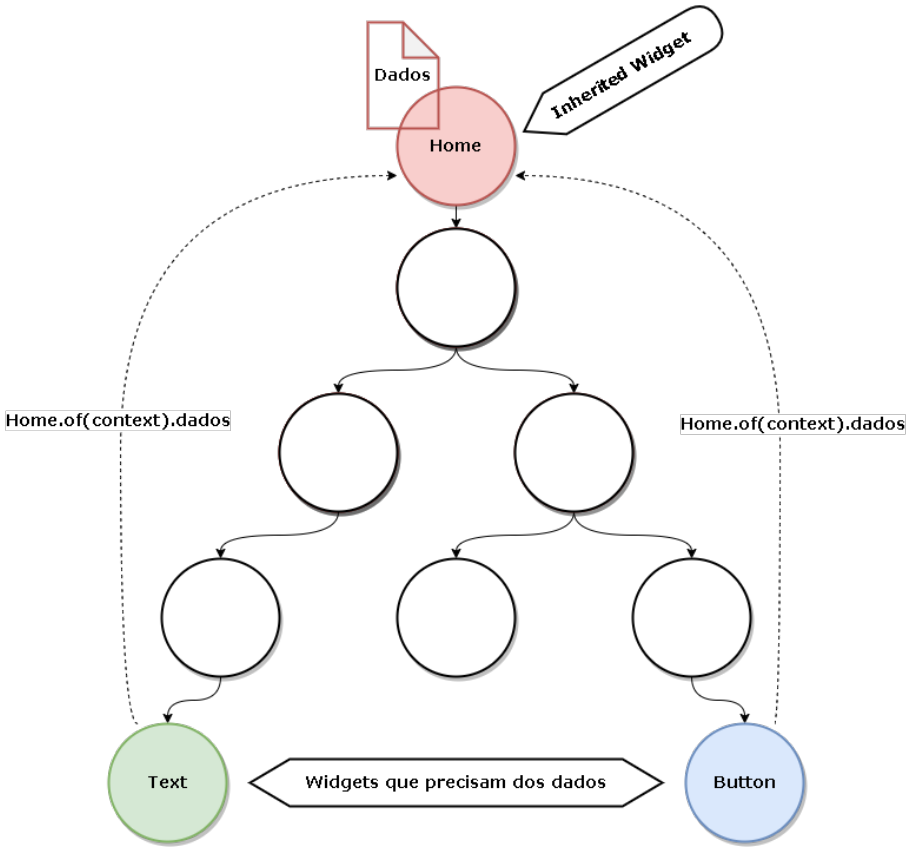


Figura 2.3: Exemplo de uma árvore de widgets com InheritedWidget. (Faust 2020) (Didier Boelens 2018)

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.

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 peque-

nos 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 *framework*s 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 subscrevem a ela quando o estado é alterado. Para isso, basta utilizar o método *notifyListeners()* em métodos que alteram esse estado.

Para fornecer 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 e Time Flutter 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.

O **observador**, por sua vez, pode ser um *widget*, que apresentará uma mudança visual

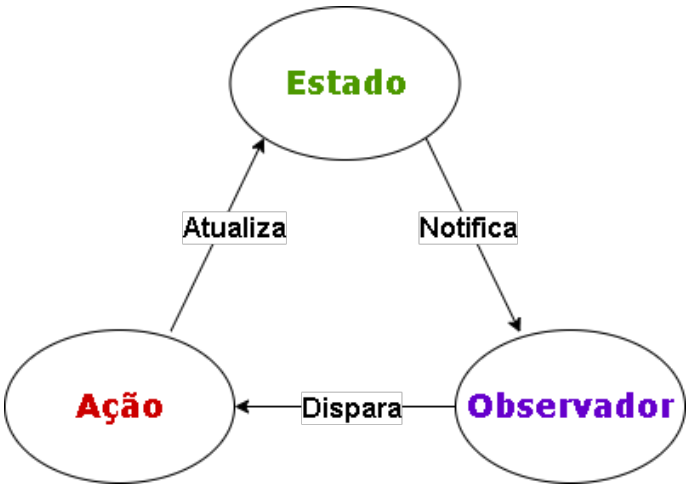


Figura 2.4: Ciclo básico do gerenciamento de estado com o MobX (P. Podila e Time Flutter 2018) (Podila & Weststrate 2018)

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 2018*c*). 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 2018*b*). 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 Organization n.d.) que permite a criação de tabelas e a manipulação de dados de forma

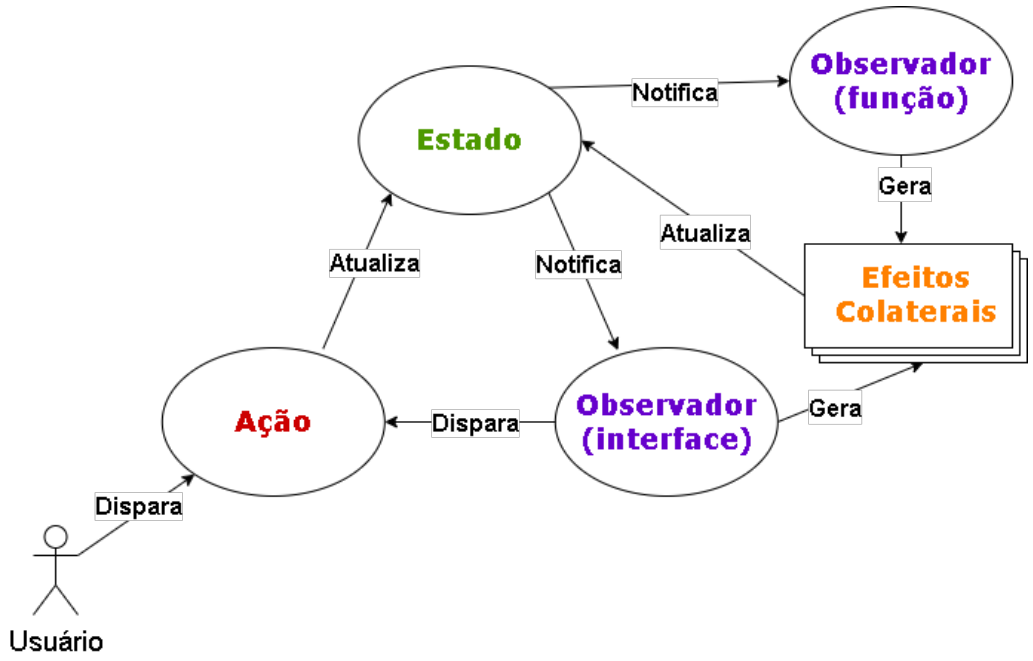


Figura 2.5: Ciclo detalhado do gerenciamento de estado com o MobX (P. Podila e Time Flutter 2018) (Podila & Weststrate 2018)

simples e rápida (Time Flutter 2018*a*) (SQLite Organization 2007*b*).

No contexto da aplicação, a quantidade de dados pode ser consideravelmente grande e estão bastante relacionados entre si, como descrito na 4.3.1. 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 2007*a*).

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.

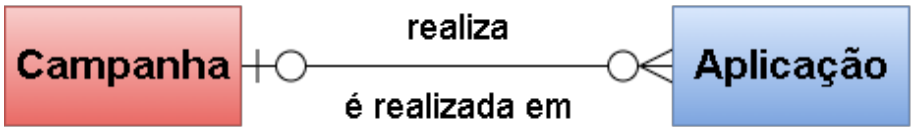


Figura 2.6: Modelagem conceitual de parte do banco de dados

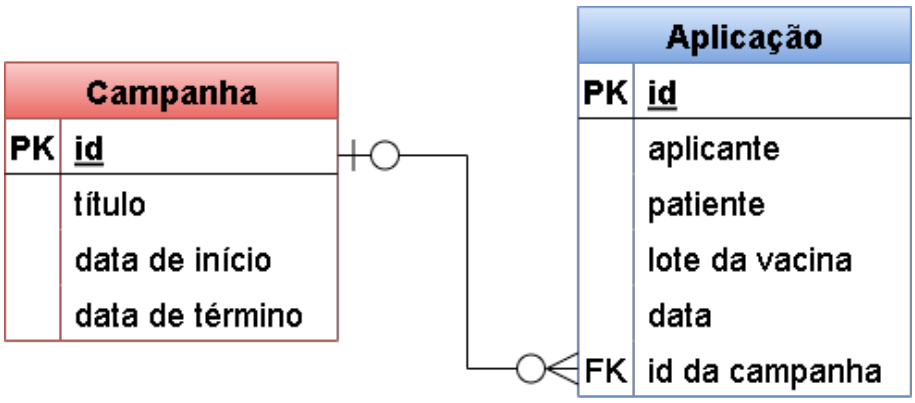


Figura 2.7: Modelagem lógico de parte do banco de dados

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

4.1 Telas

4.2 Arquitetura do Sistema

4.3 Persistência de Dados

4.3.1 Diagrama de Classes

4.3.2 Uso do Banco de Dados

4.4 Pacotes e Bibliotecas

4.4.1 MobX

4.4.2 Provider

4.4.3 excel

5
Experimentos e Resultados

Posso escrever sobre os testes feitos Escrever sobre os fluxos possíveis no sistema. Entrada de dados, mudanças de telas e saída de dados.

5.1 Testes Unitários

Mostrar a cobertura de testes unitários do sistema.

5.2 Testes de Performance

Buscar os métodos de testes de performance do Flutter e do Dart.

5.3 Fluxos de Telas

Mostrar os principais fluxos de telas do sistema.

5.4 Testes de segurança

Aqui posso falar sobre a tentativa de adicionar dados inválidos.

5.5 Fluxos de dados

Mostrar os fluxos de cadastro e compartilhamento de dados em planilha.

6
Conclusão

Referências Bibliográficas

Didier Boelens (2018), 'Widget - state - context - inheritedwidget', Disponível em: https://www.didierboelens.com/2018/06/widget-state-context-inheritedwidget/. Acesso em: 09 de Novembro de 2022.

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.

Heuser, Carlos Alberto (2009), *Projeto de banco de dados: Volume 4 da Série Livros didáticos informática UFRGS*, Bookman Editora.

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 e Time Flutter (2018), 'mobx | pacote dart', Disponível em: https://pub.dev/packages/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.

SQLite Organization (2007*a*), 'Appropriate uses for sqlite', Disponível em: https://www.sqlite.org/whentouse.html. Acesso em: 12 de Novembro de 2022.

SQLite Organization (2007*b*), '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 (2022), 'Dart overview', Disponível em: https://dart.dev/overview. Acesso em: 04 de Novembro de 2022.

Time Flutter (2018*a*), 'Persist data with sqlite', Disponível em: https://docs.flutter.dev/cookbook/persistence/sqlite. Acesso em: 12 de Novembro de 2022.

Time Flutter (2018*b*), 'Read and write files', Disponível em: https://docs.flutter.dev/cookbook/persistence/reading-writing-files. Acesso em: 12 de Novembro de 2022.

Time Flutter (2018*c*), '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 SQLiteTutorial (n.d.), 'What is sqlite', Disponível em: https://www.sqlitetutorial.net/what-is-sqlite/. Acesso em: 12 de Novembro de 2022.

A
Informações adicionais

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi tristique, orci mollis tincidunt dignissim, purus lectus molestie odio, vitae pharetra nisi sapien et justo. Fusce consequat et elit condimentum tincidunt. In eu venenatis tortor, quis lobortis tellus. Mauris tincidunt gravida ante. Pellentesque ante elit, lacinia interdum bibendum sed, cursus non orci. Ut et diam nec justo efficitur tincidunt. Fusce convallis facilisis varius. Integer tempor hendrerit maximus.

Donec pulvinar, libero semper pretium fringilla, arcu ligula iaculis urna, id faucibus ante massa a augue. Ut iaculis fermentum convallis. Vestibulum ac porttitor urna, sit amet dignissim dui. Curabitur sollicitudin tincidunt risus quis vulputate. In in leo ut mi mattis tempor sed non nunc. Sed et velit nibh. Duis mollis lectus risus, ut vestibulum purus semper eu. Donec rhoncus ex sed scelerisque dictum. Praesent sit amet neque accumsan, euismod sapien eget, tincidunt magna. Etiam nec diam ac elit feugiat aliquet. Pellentesque et neque metus. Nulla molestie libero sed ullamcorper mollis. In a risus quis felis eleifend dapibus at mollis odio. Curabitur nisl nibh, placerat ac quam nec, vulputate blandit mauris.

Vestibulum neque lacus, fringilla a urna quis, egestas tincidunt orci. Phasellus rutrum elit at mauris feugiat, non egestas tortor dictum. Aliquam faucibus, velit eu aliquam fermentum, lacus sem pellentesque nibh, et tristique urna nibh eget purus. Morbi vitae felis posuere, rutrum turpis quis, rutrum lorem. Nam lacinia cursus neque sit amet fermentum. Mauris eu mauris diam. Quisque lacinia consequat quam, at convallis urna dignissim eget. Proin sit amet varius sem, vel facilisis purus. Sed non blandit sapien. Sed auctor venenatis nibh eu ornare. Quisque euismod urna ligula, quis vestibulum sapien lacinia vehicula. Aenean sit amet vehicula felis, a imperdiet tellus. Fusce eu sem urna. Integer placerat nibh in purus consectetur mattis. Aliquam erat neque, tincidunt ac porta sit amet, egestas non lacus.