Desenvolvimento de um aplicativo mobile em Flutter com estudo de caso Estacionamento da Fatec – Ribeirão Preto

Marcelo Henrriques Pugas Junior[1], Fabricio Gustavo Henrique[2]

[1,2] Faculdade de Tecnologia de FATEC Ribeirão Preto (FATEC)
Ribeirão Preto, SP – Brasil

marcelopugasjunior@gmail.com[1], fabricio.henrique@fatec.sp.gov.br[2]

Resumo. *Este artigo tem como objetivo demostrar como foi elaborado o desenvolvimento do aplicativo Fatec Estacionamento, aplicativo para o gerenciamento de entradas e saídas de veículos na instituição, apontando as principais etapas para chegar ao resultado, demostrando entre elas a construção de componentes, estrutura do projeto e gerenciadores de estados.*

Abstract. *This article aims to demonstrate how the development of the Fatec Estacionamento, an application for the management of vehicle entrances and exits at the institution, pointing out the main steps to arrive at the final result, showing among them the construction of components, project structure and state managers.*

1. Introdução

A necessidade do acesso rápido e fácil a informação torna o uso de dispositivos moveis cada vez popular. Com isso a demanda de aplicativos para as diferentes plataformas cresceu e a dificuldade de atendê-las também, pois cada plataforma possui seu ambiente e linguagem de desenvolvimentos específicos. Nesse contexto surgiram as aplicações hibridas, onde a partir de um código-fonte podem ser executadas em diferentes plataformas.

Os frameworks capazes de gerar aplicações hibridas já apresentam alguns benefícios significantes para o desenvolvimento sendo as principais custo e tempo. Porém tais aplicações não trazem a mesma experiência de usuário que as nativas, uma vez que são desenvolvidas em linguagens Web que são interpretadas pelo browser nativo do sistema. Elas apresentam alguns pontos que precisam ser levados em consideração ao optar por sua escolha, como o fato de que algumas lojas de aplicativos não aceitam a publicação do mesmo e o difícil acesso ao hardware dos dispositivos. (Prezotto; Boniati, 2014)

O presente trabalho propõe reescrever a aplicação a aplicação AutoSys, gerenciamento de estacionamento implantado na instituição Fatec Ribeirão Preto. O aplicativo está escrito usando o framework híbrido Ionic e será reescrito utilizando o framework multiplataforma nativo Flutter. A diferença entre o framework multiplataforma nativo em relação ao híbrido é que esta, não depende de outros artifícios como *webviews* ou interpretadores de código no dispositivo em tempo de execução. Os frameworks multiplataforma por meio de uma linguagem de programação, disponibilizam o acesso às API's nativas do dispositivo e ao fim de seu desenvolvimento o framework possibilita a compilação de seu código base para ambas as plataformas. (De Almeida, 2019)

2. Objetivo

Realizar um estudo de caso sobre a reconstrução de um aplicativo desenvolvido utilizando um framework híbrido para um desenvolvido em uma multiplataforma.

3. Materiais e métodos

Flutter é um SDK para o desenvolvimento de aplicativos multiplataforma, ou seja, ele pode ser compilado para aplicações web, desktop e dispositivos móveis IOS e Android a partir de um mesmo código fonte. Seu kit de desenvolvimento possuí as seguintes ferramentas:

a) Skia, mecanismo de renderização 2D

b) Pacote de widgets baseados no Material Design e Cupertino.

c) APIs para testes

d) APIs de interoperabilidade

e) Dart Dev Tools, um conjunto de ferramentas de para debug, performance e testes.

f) Ferramenta de linha de comando (CLI) para criação, build, teste e compilar aplicações.

Flutter utiliza Dart, é uma linguagem de desenvolvimento orientada a objetos, capaz de compilar em AOT (Ahead-of-Time) para gerar os códigos nativos ARM e JIT (Just in time), que compila em tempo de execução isso permite o uso do **Hot Reload** (permite a atualização da aplicação sem perder seu estado) no desenvolvimento de aplicativos. (Flutter, 2020)

3.1. Packages

Packages são agrupamento de classes com o objetivo de prover um determinado recurso afim de facilitar o desenvolvimento por meio de reutilização de código. Elas podem ser desenvolvidas tanto pela Google ou pela própria comunidade e serem publicadas no gerenciador de pacotes do Dart. Para utiliza-las basta adicionar a *package* no arquivo pubspec.yaml do projeto e importa-las nas classes (Pub dev, 2020). As seguintes *packages* foram utilizadas no projeto:

a) **shared_preferences:** Realiza persistência de dados no formato de chave-valor no armazenamento local do dispositivo.

b) **dio:** Ferramenta para fazer requisições http escrito em Dart, que suporta interceptores, download de arquivo, timeout, entre outras funções.

c) **flutter_modular:** Gerenciamento de estrutura de projeto visando a modularização. Ele fornece soluções para lidar com esse problema, como injeção de dependência, sistema de roteamento e auto *dispose* de módulos que não estão sendo requisitados.

d) **flutter_mobx:** Gerenciamento de estados que simplifica a conexão dos dados reativos com a interface do usuário.

4. Desenvolvimento

4.1. Widgets

Widgets são classes usadas para construção de uma interface, sendo eles visíveis como botões, imagens e textos ou invisíveis como linhas e colunas. Eles são organizados em forma de árvore hierárquica onde cada widget pode possuir 1(*child*) ou vários(*children*) filhos, más cada widget tem apenas um pai. As figuras abaixo demostram a construção da um widget, a partir de sua arvore de widgets.

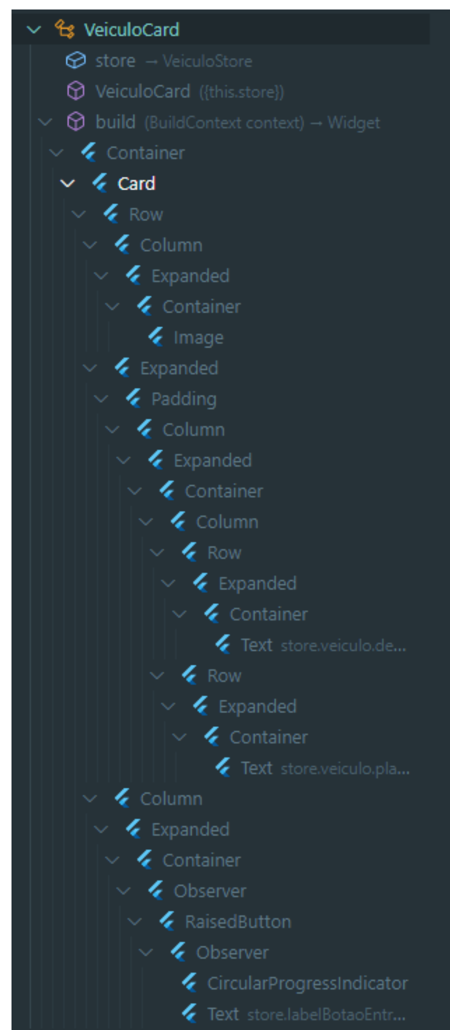

Figura 1. Arvore de widgets do componente VeiculoCard.

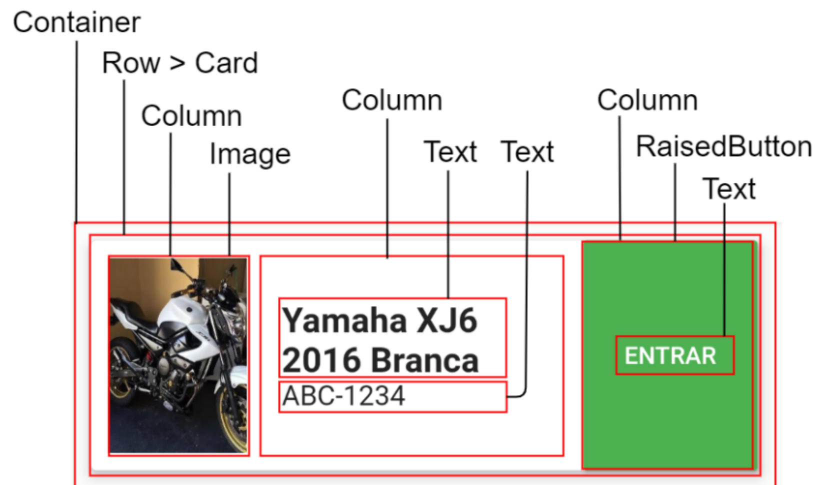

Figura 2. Demonstração gráfica da arvore de widgets do componente VeiculoCard.

4.2. Gerência de estados

Gerencia de estados é o ato de controlar os dados alterados em tempo de execução. No Flutter, quando um estado sofre alteração, precisamos atualizar a interface para que o novo dado seja exibido. Para isso foram criados alguns *designs patterns*, dentre eles são setState, bloc e mobx os mais comuns.

1.1.1 SetState

SetState é o padrão mais simples dentre os demais, é a forma nativa do Flutter de definir um novo estado para aplicação. Existem dois tipos de widget, o sem estado (StatelessWidget) e com estado (StateFulWidget), somente é possível utilizar o setState em widgets que possuem estado. Ao fim da execução do setState, ele chama o build do StateFulWidget, então toda a árvore de widgets é reconstruído e esse pode ser o principal motivo da utilização dos outros padrões pois dependendo do tamanho da árvore que será reconstruída pode gerar alguns gargalos.

```
52        floatingActionButton: FloatingActionButton(
53          onPressed: () {
54            setState(() {
55              _counter++;
56            });
57          },
58          tooltip: 'Increment',
59          child: Icon(Icons.add),
60        ), // FloatingActionButton
```

Figura 3. Exemplo do uso do setState ao pressionar o botão o contador é incrementado e o estado é atualizado.

1.1.2 BLoC

BLoC (*Business Logic of Component*) é o padrão que tem como objetivo segregar a regra

de negócio da interface de usuário, realizando a conexão entre elas por meio de programação reativa. Os estados são gerenciados através de *streams*, ou seja, sequência lógica de eventos assíncronos. StreamController é a classe responsável por controlar os eventos no BLoC, utilizando o método sink para adicionar um evento ao controle e para atualizar a interface é necessário o uso do widget StreamBuilder, onde informamos a stream(propriedade do StreamController) que vai ser "ouvido" então quando é adicionado um evento ao controlador o método build é chamado e feito a atualização de acordo com as condições definidas. (MELO, 2020)

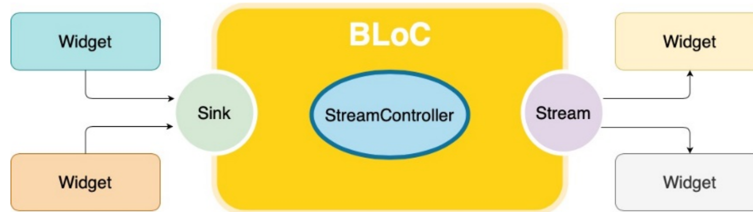

Figura 4. Fluxo do StreamController no BLoC.

```
31      body: Center(
32        child: Column(
33          mainAxisAlignment: MainAxisAlignment.center,
34          children: <Widget>[
35            StreamBuilder<int>(
36                stream: bloc.minhaStream,
37                initialData: 0,
38                builder: (context, snapshot) {
39                  if (snapshot.hasError) {
40                    return Text('Ocorreu um erro na Stream');
41                  } else {
42                    return Text(
43                      '${snapshot.data}',
44                    ); // Text
45                  }
46                }), // StreamBuilder
47          ], // <Widget>[]
48        ), // Column
49      ), // Center
```

Figura 5. Exemplo do uso do widget StreamBuilder.

Por conta do uso de *streams* é indispensável destruir as instancias que não vão ser utilizadas, pois enquanto não são destruídas, elas continuam ocupando espaço em memória. É uma boa prática sobrescrever o método *dispose*, que é chamado quando um estado não será construído novamente para fechar todas *streams* que foram instanciadas. (MELO, 2020)

```
58    @override
59    void dispose() {
60      bloc.blocController.close();
61      super.dispose();
62    }
63  }
```

Figura 6. Exemplo de sobrescrita do método dispose para fechar as streams.

1.1.3 **Mobx**

Mobx é o padrão de gerenciamento de estados simples e escalável que tem como conceito simplificar a conexão entre os dados reativos e as interfaces. Existem três conceitos importantes que formam a base do mobx, são eles *Action, Observables e Reaction*.

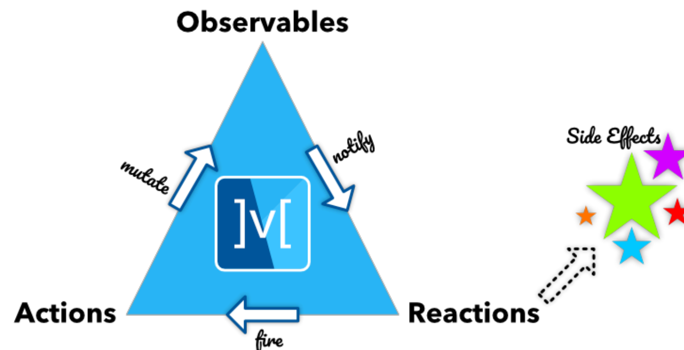

Figura 7. Três principais conceitos do mobx e suas interações.

Actions geralmente são representadas por métodos que são chamados pelos widgets para modificar os *Observables*. Quando o *Observable* sofre uma alteração, ela é refletida novamente para o widget assim mudando o estado da aplicação. Tudo isso é feito nas Stores, classes responsáveis por gerenciar tais estados. O mobx possui uma ferramenta chamada *build_runner*, onde basta adicionar anotação nos métodos e propriedades informando o que elas representam, então é gerado o código responsável por gerenciar todo o estado em uma classe particionada de forma automática. (Pub dev, 2020)

```dart
import 'package:mobx/mobx.dart';

part 'counter_controller.g.dart';

class CounterController = _CounterControllerBase with _$CounterController;

abstract class _CounterControllerBase with Store {
  @observable
  int value = 0;

  @action
  void increment() {
    value++;
  }
}
```

Figura 8. Exemplo de classe Store no Mobx.

4.3. Estrutura do projeto

O Flutter não restringe o uso de nenhum *design patern* sendo assim é reponsabilidade do desenvolvedor definir seus próprios padrões. O projeto Fatec Estacionamento foi estruturado seguindo o padrão proposto pelo Modular.

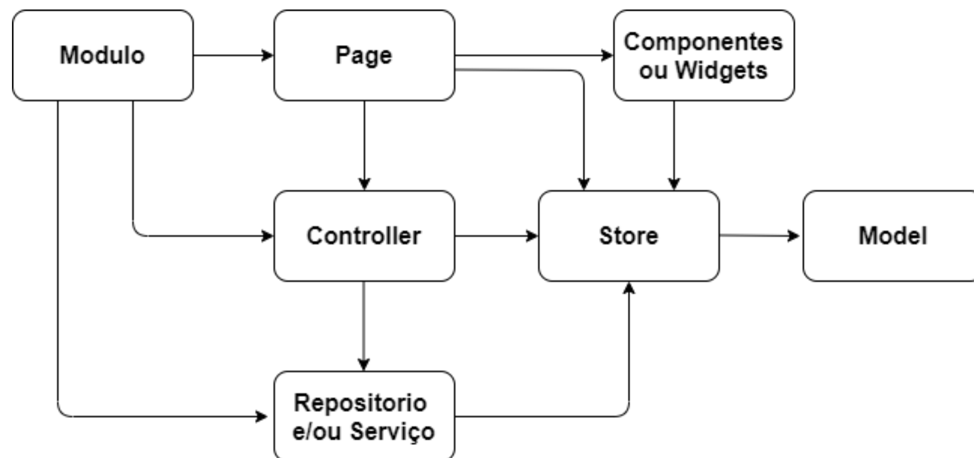

Figura 9. Estrutura do projeto e suas relações.

a) **Modulo:** Onde são feitos os controles das injeções de dependência e definido as rotas de um determinado modulo. Em um modulo, podemos ter referências de outros módulos para realizar as interações entre eles.

b) **Page:** Arquivo onde é feito a implementação da interface de usuário. Um Módulo pode ter várias Pages e cada Page pode ter ou não um controller.

c) **Componentes:** São partes da Page que foram componentizadas para serem utilizadas novamente sem ter a necessidade de reescrevê-las.

d) **Store:** São classes responsáveis por gerenciar os estados da aplicação e suas regras de negócio. No contexto do Modular chamamos de Stores quando temos a necessidade de compartilhar os estados com outros contextos na aplicação.

e) **Controller:** São Stores que podem ser instanciadas em apenas uma única Page, ou seja, podem controlar apenas um contexto da aplicação.

f) **Model:** Modelos de dados que representam uma entidade.

g) **Serviço:** Camada para intermediar as comunicações com o hardware dos dispositivos, tais como banco de dados local, GPS, Bluetooth e Câmera.

h) **Repositório**: Camada para intermediar as comunicações com serviços externos por meio de APIs. (MOURA, 2020)

5. Resultados

O aplicativo Fatec Estacionamento conta com a funcionalidade básica de gerenciar entradas e saídas dos veículos e foi adicionado a funcionalidade de comunicados, onde é possível compartilhar avisos cadastrados no sistema.

A tela de Login foi implementado o switch para lembrar dados de login onde quando está ativo, ao abrir o aplicativo os dados são carregados e preenchidos em tela com a possibilidade de serem alterados ou não.

A página do estacionamento conta com uma lista de VeiculoCard, onde é possível selecionar um veículo e dar entrada ou sair com apenas um toque, então é enviado uma

notificação para o sistema já implantado na guarita informando ao segurando a solicitação de entrada ou saída de um veículo.

Na página de comunicação é possível visualizar a lista de ComunicadoCard, e clicar sobre um comunicado para ler todo o conteúdo de um comunicado.

Em todas as telas foram preservadas algumas particularidades de cada sistema, tendo como objetivo a melhor experiencia de usuário (UX), como por exemplo, no sistema Android os títulos das páginas são alinhados à esquerda, enquanto no IOS são centralizados. Esses comportamentos são implementados pelo próprio framework Flutter que automaticamente identifica e constrói a interface de acordo com o sistema.

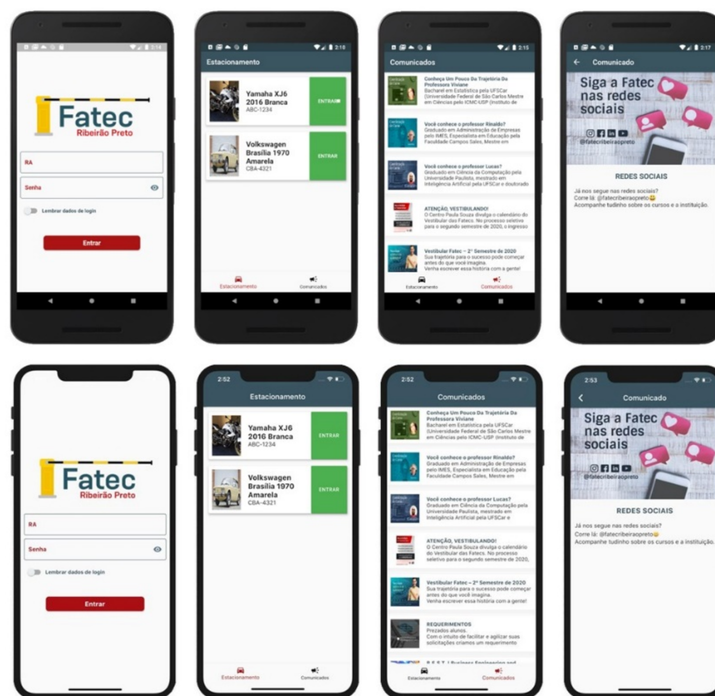

Figura 10. Aplicação Fatec Estacionamento nos dispositivos Android e IOS.

6. Conclusão

O aplicativo AutoSys foi completamente remodelado pensando em usabilidade e experiência de usuário utilizando o framework Flutter. O projeto foi arquitetado de forma que seja facilitado a manutenibilidade e escalabilidade, assim possibilitando a implementação de novas funcionalidades.

Podemos concluir que Flutter é uma poderosa ferramenta para desenvolvimento multiplataforma que possibilita a criação de aplicativos em um tempo atrativo, por conta de todas as facilidades que o framework trás com o *Hot Reload* e seu variado pacote de widgets, por exemplo.

Por conta das medidas e recomendações de relativas a pandemia do COVID-19, não foi possível concluir a implantação e testes em base real do aplicativo. O projeto de reconstrução do sistema, contará com modulo web para gerenciamento do

estacionamento da FATEC, por meio de um painel de administração onde será possível realizar cadastros e controles referentes as funcionalidades do aplicativo.

7. Referências

PREZOTTO, Ezequiel; BONIATI, Bruno. “Estudo de Frameworks Multiplataforma Para Desenvolvimento de Aplicações Mobile Híbridas”. 2014. Disponível em: <http://www.eati.info/eati/2014/assets/anais/artigo8.pdf>. Acesso em: 22 julho 2020.

DE ALMEIDA, Robson Rosa; MOREIRA, João Padilha. "Tecnologias para o desenvolvimento de aplicações multiplataforma". 2019. Disponível em: <http://raam.alcidesmaya.com.br/index.php/projetos/article/view/54/54>. Acesso em: 22 julho 2020.

Documentação Flutter. 2020. Disponível em: <https://flutter.dev/docs>. Acesso em: 23 julho 2020.

Gerenciador de pacotes da linguagem Dart. 2020. Disponível em: <https://pub.dev >. Acesso em: 24 jul. 2020.

MELO, Rubens. “Flutter para iniciantes”. Disponível em: <https://www.flutterparainiciantes.com.br>. Acesso em: 25 de julho de 2020.

MOURA, Jacob. “Quais os problemas que o Flutter Modular veio resolver?”. Disponível em: <https://medium.com/flutterando/quais-os-problemas-que-o-flutter-modular-veio-resolver-deaed96b71b3 >. Acesso em: 25 de julho de 2020.