DESENVOLVIMENTO *MOBILE*

SOLUÇÕES EDUCACIONAIS INTEGRADAS

Programação com contêiner de estado em Flutter

Maikon Lucian Lenz

OBJETIVOS DE APRENDIZAGEM

- Conceituar gerenciamento de estado via contêineres.
- Exemplificar o uso do pacote Redux.
- Implementar gerenciamento de estado via pacote GetX.

Introdução

No Flutter, a definição da forma como se dará o controle de estados é um dos pontos mais importantes no desenvolvimento. Embora o Flutter implemente o `setState()`, este pode exigir uma gestão complexa demais à medida que a aplicação cresce. Seja como for, esse formato de desenvolvimento dificulta a separação de conceitos, e é muito utilizado por aumentar a autonomia de desenvolvedores com diferentes focos dentro da aplicação.

Neste capítulo, você vai conhecer os pacotes GetX e Redux para controle de estados. Serão apresentados exemplos, com aplicações similares entre si para que seja fácil compreender a diferente abordagem de cada método disponível. Ao final, você perceberá como é mais fácil controlar o estado de suas variáveis numa aplicação Flutter fazendo uso dessas técnicas.

Gerenciamento de estado via contêineres

No Flutter, um aplicativo é dividido em *widgets* mantidos pela Árvore de Widgets (Widget Tree) à medida que novos elementos são adicionados ou destruídos (FLUTTER..., 2021). A interface é desenhada a partir da mudança de estados da aplicação, e o Flutter conduz a reconstrução dos *widgets* sempre que uma alteração o estado ocorre. A Figura 1 representa o conceito de interface de usuário declarativa, como é essa abordagem é conhecida.

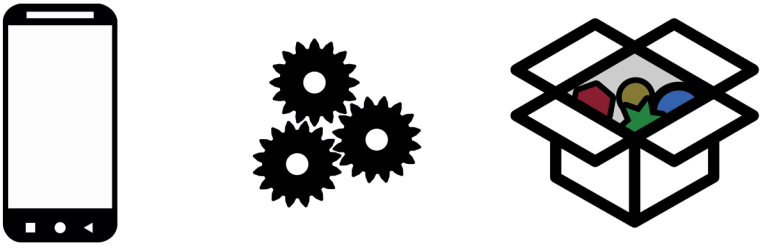

Figura 1. Representação clássica do relacionamento entre estado, métodos e interface de usuário em sistemas que utilizam a metodologia de interface declarativa.
Fonte: Adaptada de Maikon Lucian Lenz/Shutterstock.com.

Nesse processo, é a composição de vários *widgets* dá origem ao aplicativo. Esse conceito também é de fundamental importância, e deve-se tomar o cuidado para não confundir **composição** com **herança**: um objeto X pode compor um objeto Y sem ser seu herdeiro, operando apenas como um bloco construtivo.

Todos esses *widgets* interagem com a aplicação, podendo alterar o estado em que ela se encontra. A relação entre os *widgets* pode ser representada por meio da Árvore de Widgets. De forma genérica, podemos dividir esses *widgets* em dois grandes grupos: aqueles que permitem alteração de estado e aqueles que não o permitem, conhecidos respectivamente pelos termos em inglês ***stateful*** e ***stateless*** (BURD, 2020).

Quando algo sofre uma alteração de estado, é necessário redesenhá-lo de forma a atualizar a interface segundo o novo estado. Logo, parte importante do desenvolvimento de aplicações usando o Flutter é controlar adequadamente a reconstrução de *widgets*, a fim de evitar o desperdício de recursos e

processamento com a atualização de *widgets* em momentos desnecessários. Em outras palavras, precisamos recorrer ao **gerenciamento de estado**.

Os estados da aplicação podem ser de caráter global, quando todos os demais componentes têm acesso à informação, ou local, quando apenas o *widget* que teve seu próprio estado alterado tem conhecimento da mudança. A Figura 2 demonstra de que forma são classificados os estados.

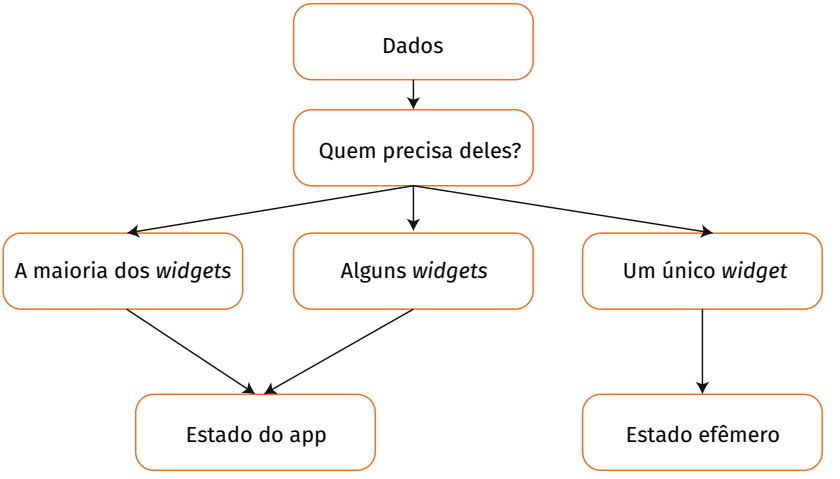

Figura 2. Fluxo que representa a tomada de decisão quanto à forma de controle de estados necessária para uma determinada aplicação.
Fonte: Adaptada de Flutter... (2021).

A abordagem básica, que utiliza o método `setState()`, pode ser aprimorada com o uso de `InheritedWidget`, que permite a comunicação entre objetos pais e filhos dentro da Árvore de Widgets ou pelo uso de Provider, que, apesar de ser o recomendado pela documentação oficial, nem sempre será a melhor solução para cada caso (FLUTTER..., 2021).

Um dos maiores desafios na hora de construir um aplicativo utilizando o Flutter é determinar a maneira mais adequada de controlar o estado dos *widgets*. Pois é justamente aí que entram os contêineres de estado, que buscam facilitar esse controle e acelerar o desenvolvimento. Os contêineres de estado centralizam o armazenamento de estados globais da aplicação, facilitando o acesso por qualquer parte do sistema e permitindo que os *widgets* alterem o estado da aplicação e percebam outras mudanças ocorridas.

A seguir, examinaremos o pacote Redux, uma abordagem de controle de estados utilizando contêineres.

Pacote Redux

O pacote Redux foi originalmente desenhado para ser utilizado com o React em situações em que era necessário gerir o estado da aplicação. Entretanto, ao contrário do React, o Flutter implementa um conceito de interface de usuário declarativa, tirando muito mais proveito das ferramentas de gerenciamento de estado, como é o caso do Redux.

O Redux é baseado em três premissas básicas (REDUX..., 2021b):

- fonte única da verdade;
- estados imutáveis;
- funções puras.

Sempre que alguma funcionalidade do Redux for utilizada, deve-se importar os pacotes `redux` ou `flutter _ redux`, a depender da classe que se está tentando utilizar. Ambas as importações podem ser vistas a seguir:

```
import 'package:redux/redux.dart';
import 'package:fluter _ redux/flutter _ redux.dart';
```

Incialmente, no entanto, deve-se declarar as dependências no arquivo "pubspec.yaml", cada qual referente a um dos pacotes importados no exemplo anterior.

```
dependencies:
  redux:
  flutter _ redux:
```

Ao utilizar o Redux, um único objeto contendo todos os estados globais da aplicação é criado, referendando o primeiro princípio, segundo o qual existe uma única fonte da verdade (REDUX..., 2021b). Sendo imutáveis, os estados armazenados no contêiner de estados, chamados no Redux de Store, somente podem ser lidos e, caso seja necessária alguma alteração, é preciso remover o estado atual e substituí-lo por outro. Logo, qualquer mudança de estado deve ser efetuada por uma ação, conforme será detalhado mais adiante.

A restrição da mudança de estados tem por consequência o aumento da transparência da aplicação, já que qualquer mudança deixará clara a origem, o momento e outras informações. Para manipular a Store, são utilizadas as funções autorizadas `Reducer`, chamadas de funções puras (REDUX..., 2021b). Entretanto, para estabelecer a ponte entre a Store e a função `Reducer`, é necessário implementar as `Actions`, como são conhecidas no Redux. São elas que sinalizam qual `Reducer` deverá ser executada.

```
enum Actions { Inc }

int varReducer(int state, action) {
  return action == Actions.Inc ? state + 1 : state;
}
```

Sempre que a função `varReducer` for invocada, será primeiramente testada qual `Action` foi utilizada para definir qual atitude quanto ao novo estado a ser retornado. No exemplo anterior, se a ação `Inc` é invocada, o estado atual da variável é aumentado em uma unidade, do contrário o mesmo valor é retornado.

Já o acesso à Store é feito pelo `StoreProvider`, que deve envolver o `wiget` sobre o qual se pretende manter o controle de estados. O `StoreProvider` tem um campo para informar a Store, e qualquer `StoreConnector` dentro de sua hierarquia poderá acessar os estados da Store. A estrutura de uma classe que utiliza esse controle de estados fica como no exemplo a seguir:

```
void main() {
  final store = new Store(varReducer, initialState: 0);
  runApp(new SagahReduxApp(store: store));
}

class SagahReduxApp extends StatelessWidget {
  final Store<int> store;

  SagahReduxApp({required this.store});

  @override
```

```
  Widget build(BuildContext context) {
    return new StoreProvider(
      store: store,
      child: new MaterialApp(
        home: new Scaffold(
          appBar: new AppBar(
              title: new StoreConnector<int, String>(
            converter: (store) => store.state.toString(),
            builder: (context, numero) => new Text(numero),
          )),
          floatingActionButton: new StoreConnector<int,
VoidCallback>(
            converter: (store) {
              return () => store.dispatch(Actions.Inc);
            },
             builder: (context, callback) => new
FloatingActionButton(
              onPressed: callback,
            ),
          ),
        ),
      ),
    );
  }
}
```

A Store pertence a todo o escopo da classe `SagahReduxApp`, e sempre que o botão for pressionado, é invocada a função `varReducer` por meio da `Action.Inc`, que fará a soma de uma unidade no valor da variável, representando o estado vinculado ao título da aplicação. Uma vez que o *widget* de texto do título está dentro de um `StoreConnector`, este perceberá a mudança no estado e reconstruirá o componente.

Para utilizar essa página dentro da aplicação, pode-se simplesmente criar uma instância para o método `runApp` dentro de *main*, como é possível observar nas quatro primeiras linhas do exemplo anterior.

Fique atento

Para se certificar de que todas as rotas do seu aplicativo podem ter acesso à Store e, consequentemente, ao controle de estados da aplicação, é necessário que o objeto `StoreProvider` preceda o `MaterialApp` (ou `WidgetApp`, se for o caso), envolvendo-o (REDUX..., 2021b).

O pacote Redux é muito utilizado para controlar estados da aplicação, mas existem outros pacotes que podem simplificar ainda mais esse processo. Entre essas alternativas, o GetX, que será abordado a seguir, talvez seja o mais simplificado.

Pacote GetX

O pacote GetX pode ser aplicado tanto para gerir o estado da aplicação quanto para outras funções, que incluem a injeção de dependência e controle de rotas (GETX..., 2021a). Em geral, o GetX tende a diminuir significativamente a quantidade de código necessário para criar a aplicação, se comparado a outras formas de gerenciar o estado. Esse, aliás, é um dos motivos pelo qual ele tem se difundido rapidamente entre desenvolvedores Flutter.

Nesse caso, você encontrará duas formas de controlar o estado da aplicação: uma simplificada, utilizando o `GetBuilder`, e outra de metodologia reativa, utilizando GetX e `Obx`. Todos estes elementos serão detalhados ao longo desta seção.

A biblioteca foi desenvolvida focando em pilares básicos. Para começar, fornece soluções de alto desempenho, motivo pelo qual não são utilizados recursos como `Streams` e `ChangeNotifiers`, presentes em outras formas de gerenciar estado. Além disso, promove alta produtividade, sendo talvez uma das bibliotecas com sintaxe mais objetiva e menor quantidade de código a ser redigido para atingir soluções similares em controle de estados. Outro pilar básico é a organização, resultante em parte da sintaxe simplificada, mas principalmente do desacoplamento do código, garantindo boa separação dos conceitos em telas, lógica das telas, regras de negócio, injeção de dependência e navegação (GETX..., 2021a).

Ainda que a biblioteca ofereça tantas soluções, os recursos estão desacoplados de forma a possibilitar, por exemplo, que apenas o controle de estados seja utilizado. Para começar a utilizar a biblioteca, basta inserir a dependência dentro do arquivo "pubspec.yaml", como no exemplo a seguir.

```
dependencies:
  get:
```

Além disso, também é preciso importar o pacote dentro de cada arquivo que venha a utilizar algum componente do GetX, como no exemplo a seguir:

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

Para criar um controle de estado reativo, em que a mudança de valor em uma variável forçará a construção de novas visualizações dos objetos envolvidos, sinaliza-se a variável como um objeto observável, por meio do sufixo ".obs" junto dela, como ilustram os exemplos a seguir:

```
var temp1 = 'O GetX não perceberá as alterações'
var temp2 = 'O GetX perceberá as alterações'.obs
```

A mudança de estado será percebida pelos objetos criados com auxílio do método `Obx`, como detalhado no próximo exemplo:

```
Obx(() => Text("${controller.temp2}"));
```

Há inclusive uma terminologia própria adotada para esse tipo de variável: variável reativa observável, ou simplesmente `Rx`. Já o *widget* `Obx` é chamado de observador, já que é quem monitora as mudanças de estado nos `Rx` (GETX..., 2021a).

Ao contrário de outros métodos, nesse caso a tela será atualizada somente quando alguma mudança real for observada no `Rx`. Desse modo, caso seja atribuído um novo valor a uma variável que seja idêntico ao valor antigo que ela possuía, o `Obx` não fará com que uma atualização ocorra, poupando recursos.

Ainda mais importante que isso é que o `Obx` atualizará apenas os valores que foram alterados, mantendo intocados os demais relacionados ao restante da mesma classe, o que, mais uma vez, é capaz de poupar muitos recursos computacionais que normalmente seriam desperdiçados reconstruindo *widgets* que não sofreram alteração alguma de estado efetiva.

Há ainda outras duas formas de declarar variáveis observáveis (`Rx`), como demonstrado nos exemplos a seguir. O primeiro apresenta métodos criados especificamente para cada tipo de variável:

```
final texto = RxString('');
final logico = RxBool(false);
final inteiro = RxInt(1);
final decimal = RxDouble(1.0);
final items = RxList<String>([]);
final mapa = RxMap<String, int>({});
```

O outro utiliza um método de tipo genérico:

```
final texto = Rx<String>('');
final logico = Rx<Bool>(false);
final inteiro = Rx<Int>(1);
final decimal = Rx<Double>(1.0);
final items = Rx<List<String>>([]);
final mapa = Rx<Map<String, int>>({});
final personalizado = Rx<SuaClasse>();
```

No caso de utilizar suas próprias classes, pode-se declarar as variáveis como observáveis utilizando quaisquer uma das formas apresentadas anteriormente, como também é possível tornar observável o objeto retornado pelo construtor, o que facilita a implementação do código. O código a seguir, por exemplo, cria uma classe de nome `Pessoa` contendo três variáveis e um construtor com todos os parâmetros opcionais. Assim é possível criar uma instância dessa classe com qualquer combinação de parâmetros.

```
class Pessoa {
    Pessoa({String nome, int idade, String endereco});
    var nome;
    var idade;
    var cidade;
}
```

Ao instanciar um novo objeto dessa classe, pode-se torná-lo integralmente observável inserindo o sufixo ".obs" após o construtor.

```
final pessoa = Pessoa(nome: "João", idade: 50, cidade:
"Curitiba").obs;
```

É preciso ter em mente, no entanto, que desta forma o pacote GetX sempre atualizará o objeto como um todo, ainda que algumas das variáveis não tenham sido alteradas.

Há outra maneira de fazer uso do controle de estados simplificado, utilizando o método `GetBuilder`. A diferença entre este último e o `Obx` é que o `Obx` atualiza apenas os estados que sofreram alguma mudança de fato, enquanto o `GetBuilder` atualiza toda a classe envolvida, mas conta com uma implementação simplificada.

Ao contrário do modo reativo de controle de estados, ao utilizar o `GetBuilder` você precisará invocar o método `update()` sempre que uma mudança for necessária. Apesar disso, o `GetBuilder` economiza uma quantidade significativa de memória e pode ser a melhor solução, especialmente em grandes aplicativos.

No exemplo a seguir, o código está dividido em duas classes, em que `SagahGetBuilderState` é responsável pelas regras que constroem a interface do usuário, enquanto a classe `SagahController` dispõe apenas das regras de negócio como declaração dos estados e mudança de estados. As regras são denominadas `View` e `Controller`, respectivamente, e fazem parte de um princípio conhecido como **separação de conceitos**, em que é possível dividir o projeto em pedaços tão independentes entre si quanto possível, permitindo alterar a interface do usuário sem alterar as regras de negócio e vice-versa.

```
class SagahGetBuilderState extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        body: Column(
      children: [
        GetBuilder(
            init: SagahController(),
```

```
          builder: (controller) {
            return Column(
              children: [
                Text('${controller?.numero}'),
                RaisedButton(onPressed: () {
                  controller?.inc();
                })
              ],
            );
          })
      ],
    ));
  }
}

class SagahController extends GetxController {
  int numero = 0;

  inc() {
    numero = numero + 1;
    update();
  }
}
```

Como no exemplo acima, os estados são acessados e alterados com o auxílio de um controlador (`GetxController`), e o controle de estados é alertado para a mudança pelo método `update()` dentro de `inc()`, logo após a variável `numero` ter um novo valor atribuído.

Um meio-termo entre o `GetBuilder` e o `Obx` é o uso do método GetX, que não demanda a invocação manual de um método para informar a necessidade de atualização, e já é capaz de filtrar quais variáveis requerem atualização e quais não. Porém, também não atinge a praticidade e objetividade do método `Obx`, em que basta sinalizar a variável que será observada para manter os estados atualizados em qualquer tela que a utilize.

Qualquer que seja a forma de controle de estados reativa (GetX ou `Obx`), deve-se alterar o controlador, podendo-se remover a chamada para a função `update`, mas adicionando o sufixo ".obs" na variável `numero`. Repare ainda que a variável `numero` não é mais um inteiro primitivo, mas será declarada como um `RxInt`.

```
class SagahGetState extends StatelessWidget {
  final cont = Get.put(SagahController());

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        body: Column(
      children: [
        GetX(builder: (controller) {
          return Column(
            children: [
              Text('${controller?.numero}'),
              RaisedButton(onPressed: () {
                controller?.inc();
              })
            ],
          );
        })
      ],
    ));
  }
}

class SagahController extends GetxController {
  var numero = 0.obs;
```

```
  inc() {
    numero = numero + 1;
  }
}
```

Por fim, quando utilizado o método GetX, pode-se recorrer à extensão da classe `Bindings` para conectar o `View` ao `Controller`, mantendo a separação de conceitos. A função do `Binding` trata de conectar as rotas criadas para cada uma das páginas ao respectivo gerenciador de estados e de dependências, quando utilizado.

```
class ApplicationBindings implements Bindings {
  @override
  void dependencies() {
    Get.put(SuaClasse());
  }
}
```

O parâmetro `initialBinding` do método `GetMaterialApp` deverá receber a instância de `ApplicationBindings` e então qualquer que seja a página a tentar acessá-lo poderá fazer através do método `Get.find<T>()`.

```
final instancia = Get.find<SuaClasse>();
```

Tendo acesso à instância procurada, pode-se acessar qualquer variável normalmente.

```
Text(instancia.nomeDaVariavel);
```

Como se vê, o pacote GetX simplifica bastante o controle de estados, sendo certamente um dos mais simples de se implementar, além de ser bastante eficiente, sobretudo quando utilizado o método `GetBuilder`.

Saiba mais

Não são apenas estados efêmeros que podem ser controlados com facilidade pelo pacote GetX. Na verdade, a grande vantagem se dá principalmente no controle de estados globais, em que o acesso a um mesmo dado se faz necessário em múltiplas páginas. O exemplo a seguir apresenta a criação de uma outra página, que acessa o valor de `numero` buscando pelo controlador no `Get` por meio do método `Get.find()`.

```
class SagahPagina2 extends StatelessWidget {
  final Controller cont = Get.find();

  @override
  Widget build(context){
     return Scaffold(body: Center(child: Text("${cont.
numero}")));
  }
}
```

Certamente o desenvolvimento de aplicativos utilizando os métodos de controle de estados, seja mediante o pacote Redux ou o GetX, pode ser facilitado e melhorar significativamente a escalabilidade e legibilidade do código.

Referências

BURD, B. *Flutter for dummies*. Hoboken: John Wiley & Sons: 2020.

FLUTTER documentation. *In:* FLUTTER SDK. *[S. l.]*, 2021. Disponível em: https://flutter.dev/docs. Acesso em: 13 jul. 2021.

GETX documentation. *In:* PUB.DEV. *[S. l.]*, 2021a. Disponível em: https://pub.dev/documentation/get/latest/. Acesso em: 13 jul. 2021.

REDUX documentation. *In:* PUB.DEV. *[S. l.]*, 2021b. Disponível em: https://pub.dev/documentation/redux/latest/. Acesso em: 13 jul. 2021.

Leituras recomendadas

DART. *[S. l.]*, 2021. Disponível em: https://dart.dev/. Acesso em: 13 jul. 2021.

ZAMMETTI, F. *Practical flutter*: improve your mobile development with Google's latest open-source SDK. Apress: Pottstown, 2019.

Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS