DESENVOLVIMENTO *MOBILE*

SOLUÇÕES EDUCACIONAIS INTEGRADAS

Gerenciamento básico de estado em Flutter

Marcelo da Silva dos Santos

OBJETIVOS DE APRENDIZAGEM

- Diferenciar estado efêmero e estado da aplicação.
- Apresentar técnicas de implementação de estado.
- Exemplificar o gerenciamento simples de estado.

Introdução

Com o aumento da escala de projetos e a constante busca por incrementar a produtividade das equipes, aprender a utilizar os recursos de uma linguagem, ou *framework*, é essencial. Como o Flutter é um *framework* fortemente relacionado ao conceito de *widgets*, conhecer seu comportamento, sua forma de comunicação e seu ciclo de vida é um pré-requisito para desenvolver aplicações que atendam melhor aos desejos e às necessidades de seus clientes.

Os *widgets*, que podem representar diversos tipos de componentes da aplicação, podem ser influenciados por informações do sistema, conhecidas como estados. Esses estados podem ser do tipo efêmero (também chamado de estado da IU [interface de usuário] ou estado local) ou estado da aplicação (também chamado de estado global ou estado compartilhado), cada um com seus conceitos e suas metodologias de implementação.

Neste capítulo, vamos detalhar as diferenças entre os tipos de estado efêmero e estado da aplicação, destacando algumas técnicas para seu gerenciamento. Também será apresentado um exemplo de aplicação do pacote Provider em gerenciamento simples de estado.

Introdução ao gerenciamento básico de estados

O *framework* Flutter, nos últimos tempos, tem caído nas graças da comunidade de desenvolvedores de aplicativos. Com sua estrutura moderna, o Flutter se baseia na utilização de *widgets*, principalmente para a construção das IUs (ou UIs, de *user interfaces*, em inglês). Esses *widgets* podem representar diversos tipos de elementos, como lista de itens, texto, conjuntos de botões, imagem, formulários, etc., e podem ser afetados por alterações de estados (FLUTTER, [2019?]).

Em resumo, **estado** é tudo o que está na memória durante o tempo de execução do aplicativo, como *assets*, conteúdo de variáveis, fontes, texturas, etc. Alguns *widgets* utilizam esses estados para desempenhar algumas ações, como efetuar a reconstrução da IU de forma a refletir o estado atual do aplicativo, como mencionado anteriormente. Logo, considerando a utilização dos estados, podemos classificar os *widgets* da seguinte maneira.

- ***Widget* sem estado (StatelessWidget):** um dos mais simples tipos de *widget*, uma vez que, depois de criado, não pode mais ter suas informações alteradas. Por isso, costuma ser chamado de imutável, e precisa ser recriado com parâmetros diferentes para que possa exibir dados diferentes.
- ***Widget* com estados (StatefulWidget):** bastante utilizado na construção de IU, sendo baseado em estados que se alteram ao longo do tempo. Ou seja, sempre que o estado do *widget* muda, o *widget* é renderizado novamente com o novo estado, refletindo a mudança de estado em sua árvore de *widgets*.

Portanto, logo ao iniciarmos o estudo do Flutter, é essencial aprender sobre **gerenciamento de estados** (ou *state management*, em inglês), em que são decididos quais estados de um *widget* precisam ser imutáveis (StatelessWidget) e quais precisam ser mutáveis (StatefulWidget). Um dos focos do gerenciamento de estados é identificar formas mais adequadas, dependendo

da situação, de notificar, ao Flutter, quando e como redesenhar a IU de uma forma mais performática, ou seja, com o menor custo de processamento possível.

Outro ponto importante é que, quando criamos um StatefulWidget, seu estado só estará disponível para o próprio *widget*. No entanto, em diversos momentos, será essencial compartilhar um estado comum entre diferentes *widgets* para que possam usar os mesmos dados de estado concomitantemente. Nesse caso, as alterações de estados são informadas por meio de notificações. Uma **notificação** é um aviso de que algo foi alterado, sendo que o Flutter tem notificadores nativos, como o `setState()`,entre outras técnicas de gerenciamento.

Por exemplo, imagine a necessidade de construir uma listagem de itens qualquer. Para isso, podem ser utilizadas informações provenientes de APIs (do inglês *application programming interfaces*, ou interfaces de programação de aplicação), Firebase, entre outras fontes; porém, como essas informações podem não ser estáticas, no caso de mudanças, o Flutter necessita receber uma notificação de que ocorreram alterações nas informações (estados) para que a interface possa refletir essas alterações.

Saiba mais

Alguns conceitos, apesar de simples e até intuitivos, devem ser explicitados aqui para facilitar a compreensão do leitor. Por exemplo, um *widget* pode ou não ter um estado, assim como um estado pode ser compartilhado ou não. Eles são caracterizados por:

- **estado local**, que é o estado isolado de um *widget*, quando não há a necessidade de compartilhá-lo com nenhum outro *widget*, ou seja, alterações que são realizadas nele não afetam nenhum outro *widget* da árvore;
- **estado global**, caracterizado por um estado compartilhado para todo o aplicativo, como quando o usuário realiza uma autenticação, por exemplo, e toda a árvore de *widgets* necessita ter essa informação para poder guiar suas regras de negócio.

Com esses conceitos em mente, o gerenciamento de estados no Flutter pode ser separado em dois tipos: estado efêmero e estado da aplicação. O estudo desses dois modelos é importante para a compreensão e a identificação da forma mais adequada de realizar esse gerenciamento.

Estado efêmero

O estado efêmero, também conhecido como estado da IU ou estado local, é definido como o estado que é utilizado por único *widget*, sendo um dos métodos mais simples de gerenciar. Seu principal método é o `setState`, pertencente à classe `State`. Ele é responsável por emitir as notificações sobre as alterações nos estados para o SDK (do inglês *software development kit*, ou *kit* de desenvolvimento de *software*) do Flutter. A partir dessas notificações, o *framework* é capaz de reconstruir toda uma árvore de *widgets*, ou seja, todos os *widgets* que estejam hierarquicamente abaixo de um *widget*-pai, realizando a atualização para refletir o novo estado.

Normalmente, o método `setState` é escrito dentro de algum método disparado por ações do usuário. Para deixar mais claro, vamos a um exemplo disponibilizado pela documentação oficial do Flutter (FLUTTER, [2019?]), que pode ser observado no trecho de código a seguir. Nele, podemos observar a implementação de um *widget* que constrói um botão e um campo de texto na tela. A cada clique no botão, um contador é atualizado e o campo de texto receberá o novo valor, apresentando o número de vezes que o botão foi pressionado.

Exemplo de uso de `setState` (FLUTTER, [2019?]):

```
class CounterDisplay extends StatelessWidget {
  CounterDisplay({required this.count});
  final int count;
  @override
  Widget build(BuildContext context) {
    return Text('Count: $count');
  }
}

class CounterIncrementor extends StatelessWidget {
  CounterIncrementor({required this.onPressed});
  final VoidCallback onPressed;
  @override
```

```
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed,
      child: Text('Increment'),
    );
  }
}

class Counter extends StatefulWidget {
  @override
  _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _counter = 0;
  void _increment() {
    setState(() {
      ++_counter;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        CounterIncrementor(onPressed: _increment),
        SizedBox(width: 16),
        CounterDisplay(count: _counter),
      ],
    );
  }
}
```

```
void main() {
  runApp(
    MaterialApp(home: Scaffold(body: Center(child:
Counter(),),),),);
}
```

Note que a classe `Counter` é o *widget* que guarda o estado da variável `_counter` por meio de `CounterState`. Ela também tem a responsabilidade de implementar o método `_increment()`, que, por sua vez, vai atualizar o `_counter`. A referência para o método responsável pelo incremento é enviada para os *widgets* que estão abaixo dele (filhos) na árvore, até chegar em `ElevatedButton`, onde está sendo usado como implementação no `onPressed`. Para auxiliar na compreensão, observe a estrutura dessa árvore na Figura 1.

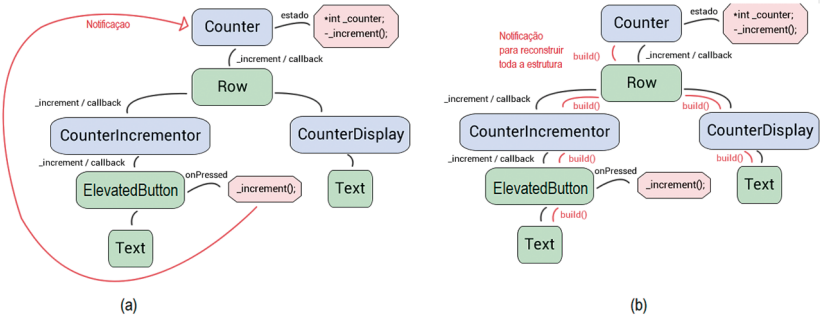

Figura 1. Diagrama do exemplo: (a) alteração reconhecida pelo acionamento do botão e notificação para *widget*-pai e (b) notificação em cascata para todos os *widgets* abaixo, informando que o estado foi alterado e acionando seus métodos de reconstrução (`build`).
Fonte: Adaptada de Stackoverflow (2020).

No momento em que pressionarmos o botão, o `onPressed` será disparado. Um detalhe: a implementação do incremento está no *widget*-pai; logo, uma notificação vai disparar um `callback` (um *widget* que está abaixo da árvore invoca um *widget* acima). Ao executar o método de incremento, o valor de `_counter` será atualizado e o `setState` será executado, notificando o SDK do Flutter de que será necessário atualizar o estado dos *widgets* (Figura 1a).

Como vemos na Figura 1b, quando atualizamos o `setState`, ocorre a atualização em cascata, em que o método `build()` aciona os *widgets* que estão abaixo na árvore para que reflitam o novo estado. Logo, o `Text()`, responsável por mostrar o valor de `_ counter`, mostrará o novo valor.

Nesse exemplo, cada vez que ocorrer uma chamada ao método `setState()`, o *widget* será reconstruído utilizando o valor atualizado na variável contador, após o incremento. Como já mencionado, esse é um dos métodos mais simples de gerenciar estados, pois, seja pelo incremento de variável ou por uma requisição efetuada a uma API, entre outros eventos capazes de alterar o estado do *widget*, somente é necessário utilizar o método `setState()` para que ocorra o redesenho que representa o novo estado.

Embora simples, o método também é considerado um dos mais custosos computacionalmente, visto que, à medida que o aplicativo for aumentando em escala e em complexidade, possivelmente será necessário realizar alterações em mais de um *widget* de forma concomitante, tornando inviável gerenciar todas as chamadas do `setState()` em cada um. Ou seja, embora seja possível aplicar `setState()`, a complexidade pode ser um fator crítico ao projeto.

Estado do aplicativo

O estado da aplicação (também conhecido como estado global ou estado compartilhado), diferentemente do efêmero, é aquele em que existe a necessidade de compartilhar o estado entre diferentes partes do aplicativo. Nesse contexto, são exemplos:

- preferências de um usuário;
- informação de *login*;
- notificações em uma aplicação de rede social;
- um carrinho de compras em um aplicativo de vendas *on-line*.

Como exemplo de método para a gerência de estados de aplicação, o Provider (PUB.DEV, 2021a) e o Redux (PUB.DEV, 2021b) são alguns dos mais conhecidos. O Redux, por exemplo, é utilizado na organização dos estados e das ações do aplicativo a serem despachados de forma modular. Resumindo, ele separa as preocupações de diferentes partes de nosso aplicativo. Caso o usuário dispare um evento (acione um botão de incremento, como no exemplo anterior), uma ação é despachada. A ação é processada e a aplicação vai apresentar o novo estado. Todo esse fluxo desse gerenciamento de estado é síncrono (a ação a ser despachada é, simplesmente, incrementar a contagem).

Afinal, qual modelo escolher?

Assim como podemos utilizar um estado efêmero para gerenciar todo o estado do aplicativo, de outra forma, podemos entender que (no contexto específico do aplicativo) o mesmo contexto deve ser mantido entre as sessões, sendo representadas pelo estado da aplicação. Dessa forma, não existe uma regra universal para escolha entre um ou outro modelo. Talvez você até tenha que implementá-los em conjunto. Por exemplo, o projeto pode iniciar com algum estado claramente efêmero, mas, conforme seu aplicativo escala em recursos, pode ser necessário transformá-lo para o estado de aplicativo.

Então, como escolher? Segundo o autor do Redux, Dan Abramov, a regra prática é: faça o que for menos difícil. Ou seja, existem dois modelos conceituais de gerenciamento de estado no Flutter, o efêmero, que geralmente é local e direcionado para um único *widget*, e o estado da aplicação, indicado para os demais casos. Ambos os tipos têm seu lugar em qualquer aplicativo Flutter, e a opção entre um e outro vai depender de sua preferência e da complexidade do aplicativo.

Métodos para gerenciamento de estados

Como visto anteriormente, o gerenciamento de estados pode ser implementado tanto para o estado efêmero quanto para o estado da aplicação. Foi apresentado o gerenciamento com `setState`, para estados efêmeros, e comentamos brevemente sobre Redux e Provider. O Flutter possui uma longa lista de soluções para gerenciamento de estado e, embora não seja o objetivo deste capítulo implementar os modelos mais avançados, podemos listá-los e apresentar alguns dados sobre eles.

BLoC

Foi apresentado em 2018 na Google DartConf como uma solução para auxiliar no compartilhamento de código entre aplicações *web* e *mobile* (Flutter e AngularDart, respectivamente). Seu nome é um acrônimo para *business logic component* (BLoC) e representa uma forma de abstração do estado da aplicação (MELO, 2021).

Um dos principais destaques desse método são os `Streams`, representados por sequências de eventos assíncronos. Imagine utilizarmos uma aplicação

em Flutter e uma versão *web* em AngularDart: eles não podem compartilhar estados, pois estes são mecanismos internos do *framework* e não podemos utilizá-los fora do Flutter. Como os `Streams` são recursos nativos do Dart (linguagem relacionada a ambas as plataformas), são utilizados independentemente da plataforma, ou seja, como o gerenciamento de estadoa é realizado em código Dart, é possível ser compartilhado tanto em Flutter quanto em AngularDart.

O BLoC, por ser um componente lógico, responde apenas a eventos, não se preocupando com a interface. Um **evento** é qualquer ação capaz de alterar estados em nossa aplicação, por exemplo uma chamada de API, uma autenticação ou um contator para incremento em variável. Os *widgets* respondem a eventos de outros *widgets*, gerando a possibilidade de separação entre a regra de negócio e a interface. Observe a Figura 2 (PUB.DEV, 2021c).

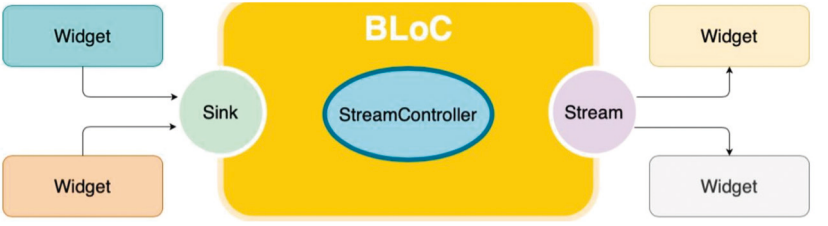

Figura 2. Estrutura do BLoC.
Fonte: Pub.dev (2021c, documento *on-line*).

Na Figura 2, podemos ver o núcleo de um BLoC, o `StreamController`. Esse elemento gerencia como serão realizadas as operações de entrada (`sink`) e saída (`stream`). A partir desses componentes, temos os principais elementos do padrão BLoC. Seu processo é bem simples:

- um evento é enviado para o BLoC por meio de um `sink`;
- após receber o evento, o BLoC aplica a lógica de negócio respectiva;
- em seguida, o `stream` recebe a saída do evento;
- o novo valor associado a `stream` do BLoC é notificado a qualquer elemento que esteja aguardando.

Mobx

Um pacote bastante popular para gerenciamento de estados e que utiliza geração de código para reduzir a complexidade e o volume de código escrito (GITHUB, [2021?]). Sua intenção é que o desenvolvedor se concentre somente em quais dados necessitam ser consumidos na IU (e em outros lugares), deixando a sincronia entre os estados e a UI por conta do Mobx. Para tanto, ele mantém o controle sobre o que está sendo consumido (chamado de observáveis) e sobre como serão as respostas (chamadas de reações).

A estrutura padrão do MobX é apresentada na Figura 3, que destaca seus principais pilares, listados a seguir.

- **Observáveis (*observables*):** retratam os estados reativos da aplicação. Ao representar o estado da aplicação como uma árvore de observáveis, será possível expor uma árvore de estados reativos que a IU pode consumir.
- **Ações (*actions*):** é o modo utilizado para alterar os observáveis, no qual são incluídos conceitos semânticos às operações, em vez de alterá-los diretamente. Para deixar mais claro, uma função `increment()` tem mais significado do que apenas efetuar um `value++`. Outra característica das ações é que elas agrupam as notificações, garantindo que as alterações disparem notificações somente após sua conclusão.
- **Reações (*reactions*):** atuam como "ouvintes" do sistema, recebendo notificações sempre que um de seus observáveis sofre alteração.

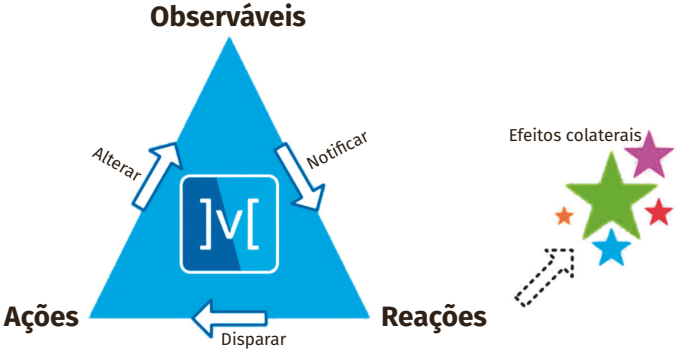

Figura 3. Estrutura de relacionamento dos módulos Mobx.
Fonte: Adaptada de Pub.dev (2021d).

Quando qualquer dos observáveis sofrer alteração, todas as reações serão redefinidas. Nesse modelo, uma reação pode ser qualquer tipo de evento, como um simples *log* de console, uma chamada de rede para renderizar novamente a IU. Uma das características do modelo é que não são necessárias definições explícitas, ou seja, todos os observáveis são automaticamente rastreados pelas reações.

InheritedWidget

Inherited (herdado, traduzido do inglês) deve sua denominação ao fato de que os elementos descendentes de um InheritedWidget são capazes de acessar as propriedades com simplicidade, por herdar as características do elemento-pai (FLUTTER, [2019?]). Geralmente, encontraremos o InheritedWidget no topo da árvore de *widgets*, como visto na Figura 4.

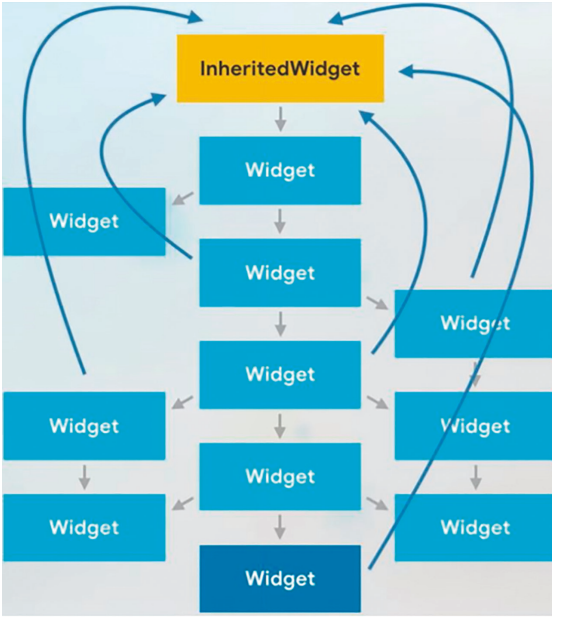

Figura 4. Utilização de um InheritdWidget no topo de uma árvore de *widgets*.
Fonte: Adaptada de Developer Libs (2019, documento *on-line*).

Portanto, para situações em que a passagem de informações seja no sentido de cima para baixo (*top-down*), não importando o tamanho da árvore de *widgets*, o modelo InheritedWidget pode ser uma opção interessante.

Para a criação de um *widget* que estende um InheritedWidget (FLUTTER, [2019?]), devemos realizar a sobrescrita dos métodos `of` (utilizado para facilitar o acesso ao *widget*) e `updateShouldNotify` (que indica o momento no qual o *widget* necessita ser redesenhado). Por sua simplicidade, essa é uma das soluções mais conhecidas pela comunidade Flutter.

GetIt

Conforme ocorre o aumento de escala do aplicativo, pode ser necessário, em algum momento, realizar a separação da lógica da aplicação em classes separadas dos respectivos *widgets*. Eliminar essas dependências diretas com os *widgets* torna o código mais organizado e simplifica sua manutenção (PUB.DEV, 2021e). Entretanto, uma vez separados, será necessário implementar uma forma de realizar o acesso aos objetos a partir da IU. Nesse ponto, entra em ação o GetIt, que, atuando como um localizador de objetos associado ao `get _ it _ mixin`, mostra-se uma solução um pouco mais avançada para o gerenciamento de estados (PUB.DEV, 2021f). Sua comunidade desenvolvedora destaca algumas características interessantes:

- bom desempenho em relação à velocidade;
- baixa curva de aprendizado ao desenvolvedor;
- evita sobrecarga na árvore de IU com a criação de *widgets* especiais para acesso os dados (como é feito pelo Provider e o Redux, por exemplo).

Ainda podemos citar uma lista extensa de *packages*, arquiteturas e outros métodos, como o `setState`, Provider, Redux (visto anteriormente), Fish-Redux, GetX, Binder, Riverpod, entre outros (PUB.DEV, c2021). A área de gerenciamento de estados ainda será muito explorada e tem muitas opções. É muito válido, para quem está começando, experimentar o maior número de opções e avaliar qual se adapta melhor ao estilo de programação.

Exemplo de gerenciamento simples de estado utilizando Provider

Nesta seção, você vai acompanhar um exemplo de implementação de um aplicativo que utiliza Provider para realizar o gerenciamento simples de estado. O material está disponível na documentação do Flutter sobre gerenciamento de estados simples (FLUTTER, [2019?]). Como o projeto é extenso, vamos nos deter nos pontos mais relevantes relacionados ao gerenciamento de estados, mas você pode ter acesso ao projeto na íntegra acessando o repositório da comunidade Flutter (GITHUB, [2020?]).

Na Figura 5, vemos as telas iniciais do aplicativo de exemplo. Ele é composto por três interfaces principais: uma tela de *login*, um catálogo de produtos e um carrinho de compras, representados pelos widgets `MyloginScreen`, `MyCatalog` e `MyCart`, respectivamente. Note que é um modelo de aplicação bastante comum, podendo ser aplicada a mesma estrutura a diversos outros contextos.

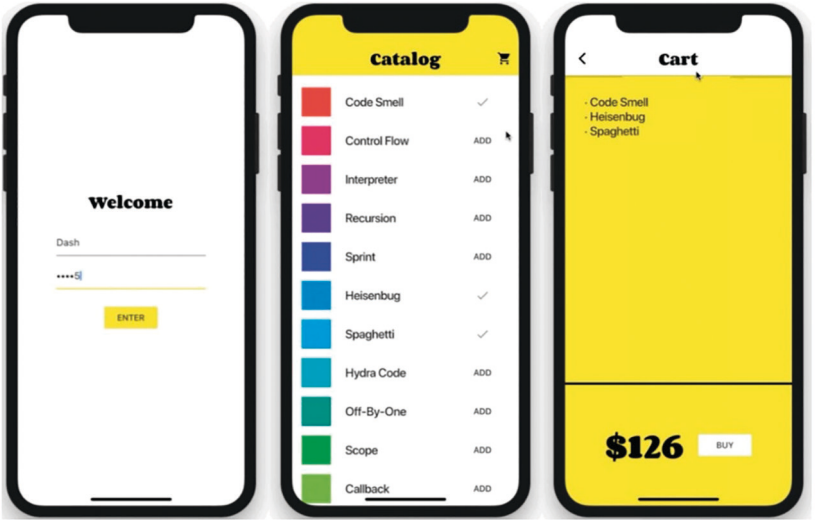

Figura 5. Telas do aplicativo exemplo.
Fonte: Adaptada de Flutter ([2019?]).

Primeiramente, para que possamos utilizar o Provider, devemos incluir as dependências necessárias no projeto. Para a implantação desse e de quaisquer outros pacotes necessários ao projeto, basta buscá-los no Pub.dev (c2021) e incorporá-los ao desenvolvimento.

Após a criação de um novo projeto, busque, na raiz do projeto, o arquivo "pubspec.yaml" e encontre a seção `dependencies`. Acrescente, então, a dependência para o pacote Provider (PUB.DEV, 2021a):

```
dependencies:
  provider: ^5.0.0
```

Em seguida, no mesmo local, acione, em seu IDE (do inglês *integrated development environment*, ou ambiente de desenvolvimento integrado), a função `pub get` (ou `package get`, dependendo do IDE) para realizar o *download* e a instalação dos respectivos arquivos em seu projeto. Alternativamente, você pode realizar a inclusão da dependência via linha de comando, utilizando o comando `$ flutter pub add provider`. Isso adiciona uma linha como essa ao "pubspec.yaml" de seu pacote (e executará um implícito `dart pub get`).

Uma vez realizada a inclusão das dependências necessárias a seu funcionamento, podemos voltar ao projeto em si. Ele é composto uma pequena árvore de *widgets*, que são listados a seguir.

- `MyLoginScreen`: interface para a autenticação do usuário.
- `MyApp`: representa a raiz da aplicação, o *widget*-pai (ou raiz).
- `MyCatalog`: logo após o usuário realizar a autenticação, é apresentada a tela inicial contendo a barra superior (que dá acesso ao carrinho de compras) e a lista de itens selecionáveis.
- `MyAppBar`: a barra que dá acesso ao carrinho de compras.
- `MyCart`: representa o carrinho de compras. Ele é que recebe os itens da lista logo após serem selecionados.
- `MyListItem`: cada um dos itens disponíveis para seleção.

Na Figura 6, estão apresentados todos os componentes dessa árvore de *widgets*.

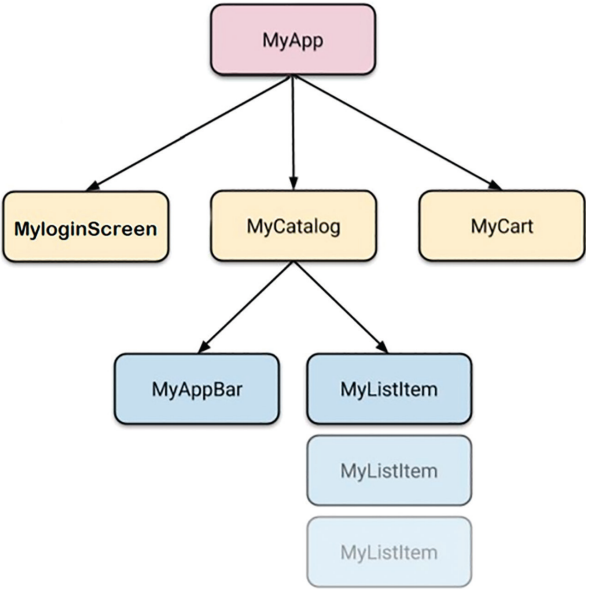

Figura 6. Uma árvore de *widgets* com `MyApp` na parte superior e seus filhos.
Fonte: Flutter ([2019?], documento *on-line*).

Veja, na Figura 6, que alguns desses componentes, apesar de estarem inclusive em ramificações diferentes, necessitam compartilhar seus estados. A necessidade de gerenciamento dos estados acontece, por exemplo, ao adicionar itens ao carrinho, pois será necessário notificar ao widget `MyCart` a mudança de seu estado, já que agora ele contém mais um item (`MyListItem`).

Desse modo, podemos ter diversas situações em que os *widgets* necessitarão buscar ou atualizar o estado da aplicação. Ou seja, além de armazenar esses estados, a informação deve estar em um local acessível a toda a estrutura.

Selecionando o posicionamento do estado

Como comentamos nas seções anteriores, é comum manter o estado em um ponto mais elevado do que os *widgets* que o estão utilizando. Considerando que um *widget* pode ser reconstruído por alterações ocorridas em seu estado, ele também vai disparar a atualização de seus filhos. Logo, esse processo inicia no topo e segue em direção às partes mais baixas da árvore.

Caso nosso estado estivesse posicionado no *widget* `MyCart` e necessitássemos atualizar `MyCatalog`, por exemplo, até poderíamos tentar criar mecanismos para gerar o comportamento de reconstrução, mas estaríamos indo contra a estrutura, no lugar de fazê-la nos ajudar. Em nosso exemplo, o estado necessita ser implementado junto ao `MyApp`; assim, sempre que ele for alterado, toda a estrutura abaixo será atualizada com os novos valores. Podemos visualizar esse posicionamento na Figura 7.

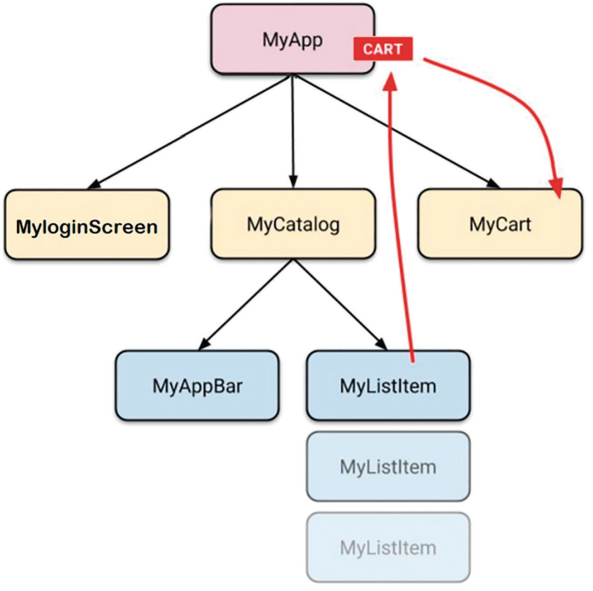

Figura 7. Atualização do estado representado pela caixa `Cart`. Na imagem, a seleção de um item da lista dispara uma alteração no estado, o que provoca a atualização do carrinho de compras.
Fonte: Flutter ([2019?], documento *on-line*).

Definido, então, onde será posicionado o estado do carrinho, passamos às formas de acessá-lo.

Acesso aos estados

Quando um usuário seleciona qualquer dos itens do catálogo, ele será adicionado ao carrinho. Porém, como adicionar um `MyListItem` a um elemento no

topo da árvore? Nesse ponto, entram em ação os *providers* (ou provedores, em tradução livre), que serão responsáveis por prover pontos para acesso aos objetos da árvore que necessitamos utilizar. Como apresentado anteriormente, eles normalmente estarão posicionados acima dos *widgets* que os utilizarão.

O *provider* tem algumas classes e estruturas características, entre as quais podemos destacar o `ChangeNotifier`, o `ChangeNotifierProvider` e o `Consumer`, que analisaremos a seguir.

ChangeNotifier

A classe `ChangeNotifier` é responsável por fornecer as notificações de alteração dos estados para seus *listeners*. Um ***listener*** é uma implementação destinada a "escutar" o que está acontecendo com um objeto e enviar notificações caso algo ocorra. Quando um *widget* implementa um `ChangeNotifier`, qualquer outro pode se inscrever para receber suas alterações.

Voltando ao nosso exemplo, para realizarmos o gerenciamento do estado do carrinho de compras utilizando a classe `ChangeNotifier`, devemos estendê-la na classe que vai utilizá-la. Observe o exemplo de implementação da classe `CartModel`.

```
class CartModel extends ChangeNotifier {
  // Estado interno privado do carrinho.
  final List<Item> _items = [];

  // Uma visão inalterável dos itens no carrinho.
  UnmodifiableListView<Item> get items =>
UnmodifiableListView(_items);

  // Cálculo do total, considerando que todos os itens
  // tenham o mesmo preço.
  int get totalPrice => _items.length * 42;

  //Adicione um item ao carrinho.
  void add(Item item) {
    _items.add(item);
    notifyListeners(); //notifica os ouvintes que houveram
alterações
  }
```

```
  // Remove todos os itens do carrinho.
  void removeAll() {
    _items.clear();
    notifyListeners(); //notifica os ouvintes que houveram
alterações
  }
}
```

Nesse exemplo, estão destacadas as partes relevantes para nosso estudo. Veja que a classe `CartModel` realiza a extensão de `ChangeNotifier`, e assim podemos utilizar os `notifyListeners()`. Esae método será invocado sempre que houver inclusão ou remoção de itens no carrinho, disparando um aviso aos *listeners* que possam necessitar redesenho da IU. O restante da codificação da classe CartModel são elementos do próprio modelo e outras regras de negócio.

ChangeNotifierProvider

O `ChangeNotifierProvider` tem o mesmo intuito do `ChangeNotifier`, embora sua implementação seja um pouco diferente. Ele vai prover instâncias de `ChangeNotifier` para seus descendentes e reconstruir a estrutura sempre que um *listener* for chamado. Nesse exemplo, um `ChangeNotifierProvider` é utilizado "envolvendo" o construtor de `CartModel()`.

```
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => CartModel(),
      child: MyApp(),
    ),
  );
}
```

Nesse exemplo, estão destacadas as partes relevantes para nosso estudo. Veja que a classe `CartModel` realiza a extensão de `ChangeNotifier`, e assim podemos utilizar os `notifyListeners()`. Esse método será invocado sempre que houver inclusão ou remoção de itens no carrinho, disparando

um aviso aos *listeners* que possam necessitar redesenho da IU. O restante da codificação da classe `CartModel` são elementos do próprio modelo e outras regras de negócio.

Observe que, dessa forma, acabamos de definir o construtor que vai gerar as instâncias de `CartModel`. A implementação da classe `ChangeNotifierProvider` fará a gerência e somente vai disparar a construção de `CartModel` quando realmente indispensável. Ela também fará o `dispose()` (descarte) automático de `CartModel` no momento em que a instância perder sua utilidade.

Consumer

Como o próprio nome indica, um `Consumer` permite acessar (consumir) os valores de um modelo. Agora que nosso modelo `CartModel` está acessível para a árvore de *widgets* por meio do `ChangeNotifierProvider`, podemos configurar um *widget* `Consumer`. Observe o seguinte exemplo de implementação.

```
return Consumer<CartModel>(
  builder: (context, cart, child) {
    return Text("Total price: ${cart.totalPrice}");
  },
);
```

Para utilizá-lo, devemos definir quais serão os modelos que precisamos acessar; nesse caso, o `CartModel`. Assim, note que foi implementado `Consumer<CartModel>`, realizando uma especificação genérica `<CartModel>`, pois, como o *provider* é baseado em tipos, definir um tipo (mesmo que genérico) é necessário para que possa ser utilizado.

Outro ponto a ser observado é que seu único parâmetro é o construtor (chamado sempre que `ChangeNotifier` é alterado no modelo), de modo que os métodos do construtor serão invocados para todos os `Consumer` correspondentes. Seu construtor é composto por três argumentos:

- o `context`, comum em qualquer método de construção;
- a instância do `ChangeNotifier` (nesse caso, o `cart`);
- o `child`, que existe para otimização. Caso o modelo apresente uma grande subárvore de *widgets* sob sua `Consumer`, que não sofre alterações quando o modelo é alterado, é possível construí-la uma vez e acessá-la via construtor.

Ao contrário dos demais, é recomendado que o `Consumer` seja posicionado no local mais baixo possível na árvore, porque não é interessante reconstruir grandes partes da IU por pequenas alterações.

`Provider.of`

É utilizado quando precisamos acessar os dados, mas não é necessário que a IU seja alterada. Diferentemente do `Consumer`, que reconstrói a IU, podemos implementar um método que simplesmente remova itens do carrinho, mas que não reflita essa ação na tela. Por exemplo, um botão de `ClearCart` responsável por remover itens; para isso, não é preciso exibir o conteúdo do carrinho, basta invocar um método `clear()` e remover tudo.

Poderíamos utilizar um `Consumer<CartModel>` para executar essa ação, mas estaríamos desperdiçando recursos do aplicativo, forçando desnecessariamente a reconstrução da IU. Sua sintaxe é apresentada no exemplo a seguir, incluindo o parâmetro `listen` atribuindo-lhe `false`.

```
Provider.of<CartModel>(context, listen: false).removeAll();
```

Ao invocar o trecho anterior em um método de construção, evitaremos que o *widget* seja reconstruído quando um método `notifyListeners` for invocado.

Neste capítulo, você conheceu um pouco mais sobre o desenvolvimento declarativo da IU e as diferenças entre estados efêmeros e estados da aplicação. Ainda, foram apresentados alguns outros métodos para a gerência de estados, alguns mais avançados. Como comentado, não era a intenção detalhar esses modelos avançados, mas mostrar que existem e explicar sua importância. Com esses conhecimentos e de posse do projeto de exemplo, agora é a sua vez de testar o *provider* e outras formas de gerência de estados em aplicações Flutter para dominar essas habilidades.

Referências

DEVELOPER LIBS. *Flutter*: propagate information with InheritedWidget. 2019. Disponível em: https://www.developerlibs.com/2019/12/flutter-propagate-information-inheritedwidget-example.html. Acesso em: 30 jun. 2021.

FLUTTER. *Flutter documentation*. [2019?]. Disponível em: https://flutter.dev/docs. Acesso em: 30 jun. 2021.

GITHUB. *Mobx.dart*. [2021?]. Disponível em: https://github.com/mobxjs/mobx.dart. Acesso em: 30 jun. 2021.

GITHUB. *Provider shopper*. [2020?]. Disponível em: https://github.com/flutter/samples/tree/master/provider_shopper. Acesso em: 30 jun. 2021.

MARINHO, L. H. *Iniciando com Flutter framework*: desenvolva aplicações móveis no Dart Side! São Paulo: Casa do Código, 2020. (*E-book*).

MELO, R. de. *Flutter para iniciantes*. 2021. Disponível em: https://www.flutterparainiciantes.com.br/. Acesso em: 30 jun. 2021.

PUB.DEV. *Bloc 7.0.0*. 2021c. Disponível em: https://pub.dev/packages/bloc. Acesso em: 30 jun. 2021.

PUB.DEV. *Find and use packages to build Dart and Flutter apps*. c2021. Disponível em: https://pub.dev. Acesso em: 30 jun. 2021.

PUB.DEV. *Get_it 7.1.4*. 2021e. Disponível em: https://pub.dev/packages/get_it. Acesso em: 30 jun. 2021.

PUB.DEV. *Get_it_mixin 3.1.3*. 2021f. Disponível em: https://pub.dev/packages/get_it_mixin. Acesso em: 30 jun. 2021.

PUB.DEV. *Mobx 2.0.1*. 2021d. Disponível em: https://pub.dev/packages/mobx. Acesso em: 30 jun. 2021.

PUB.DEV. *Provider 5.0.0*. 2021a. Disponível em: https://pub.dev/packages/provider. Acesso em: 30 jun. 2021.

PUB.DEV. *Redux 5.0.0*. 2021b. Disponível em: https://pub.dev/packages/redux. Acesso em: 30 jun. 2021.

STACKOVERFLOW. *Fluxo de notificação de alteração*. 2020. Disponível em: https://pt.stackoverflow.com/questions/406285/fluxo-de-notifica%C3%A7%C3%A3o-de-altera%C3%A7%C3%A3o. Acesso em: 30 jun. 2021.

Leitura recomendada

DART. *Dart documentation*. c2021. Disponível em: https://dart.dev/docs. Acesso em: 03 jun. 2021.

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