Gerenciamento de Estado e Boas Práticas e Padrões em Flutter

Danilo Perez

Full Stack Developer

perez-danilo

Fala devs

Objetivo Geral

Objetivo deste módulo é apresentar de forma teórica e prática o conceito de gerenciamento de estado em Flutter. Conheceremos diversos pacotes que serão uteis nessa tarefa de desenvolvermos com gerência de estado e reatividade. Abordaremos o conceito de injeção de dependência e inversão de controle. Finalizaremos refartando nosso projeto, separando em camadas.

O que é estado no Flutter?

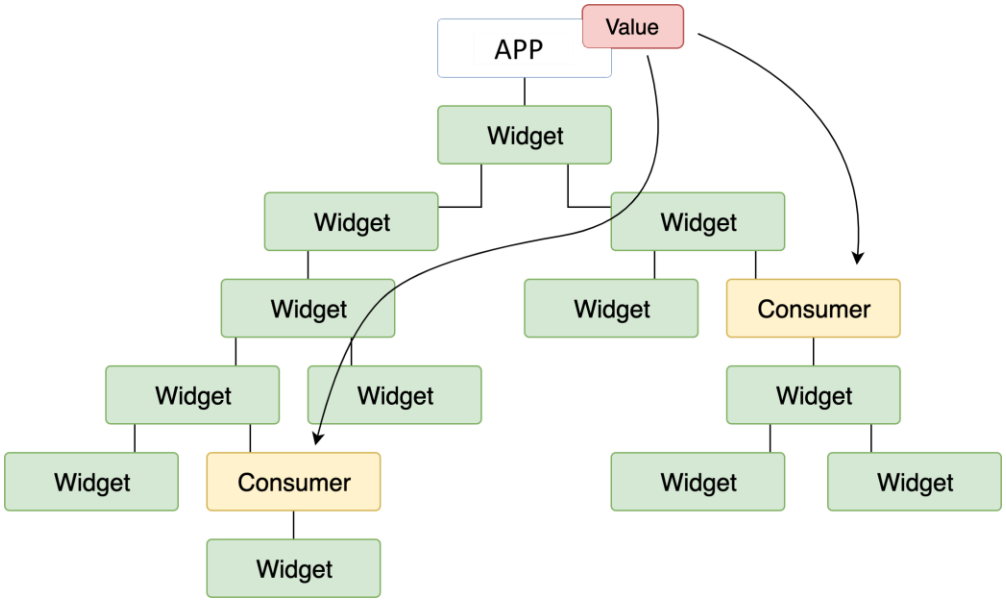
Estado em Flutter é considerado como a situação atual que o aplicativo se encontra num determinada posição no tempo. Como as variáveis que estão preenchidas, a forma que a tela está sendo exibida e etc.

O Flutter possui uma função que nativa que altera o estado da aplicação, que seria o setState({}), ele é gerenciador de estado mais rápido do Flutter

Só que muitas vezes precisamos evoluir nosso aplicativo para alterações não apenas em nossa tela, mas também em Widgets de outras telas, e para isso usamos outras formas de gerenciamento de estado e reatividade.

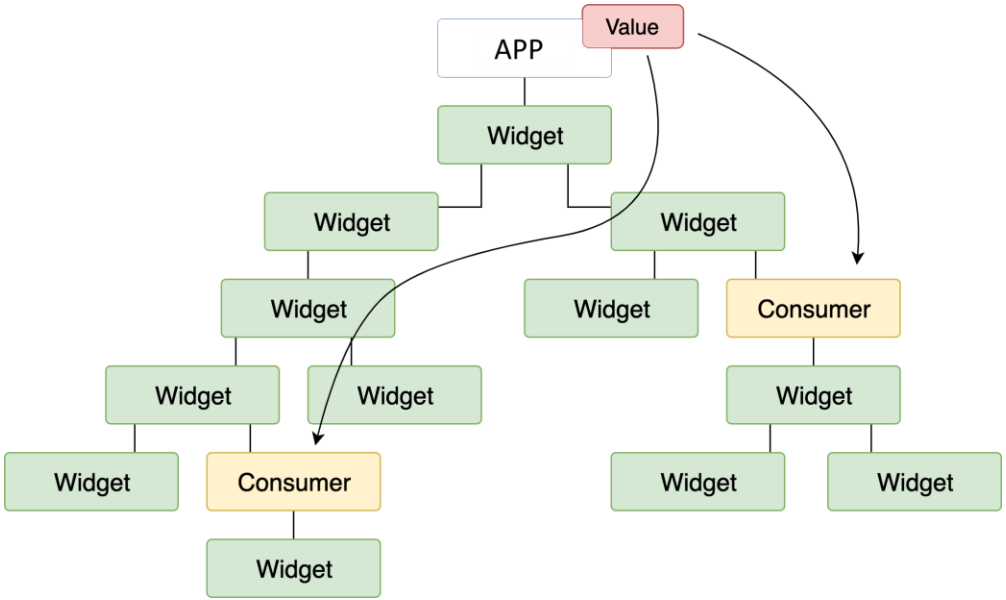
Alterando Widgets

Muitas vezes em nossas aplicações precisamos que uma mesma informação seja vista por mais de um widget, onde cada alteração possa ser identificada e ser alterada



Propagando as informações

Para isso usamos o gerenciamento de estado, juntamente com a reatividade aplicada a ele, onde temos vários pontos que ficam escutando as alterações e fazer a alteração do widget automaticamente



Principais Pacotes

- setState
- Provider
- MobX
- GetX
- Bloc/RX

Injeção de dependência
Inversão de controle

Injeção de dependência / inversão de controle é um dos principais padrões e de projeto, é é a letra D, do SOLID (Dependency Inversion Principle), que visa que uma classe não seja responsável por buscar as classes, configurações, arquivos, etc, mas sim, ela recebe essa informação de forma automática, um exemplo é o recebimento desses dados via Construtor.

Com isso podemos instanciar classes com configurações diferentes, dependendo da forma que instanciamos ela. Facilitando até nos casos de testes, onde precisamos mockar dados.

Exemplo

```dart
class CommentsDioRepsositoy implements CommentsRepository {
  final JsonPlaceHolderCustonDio jsonPlaceHolderCustonDio = JsonPlaceHolderCustonDio();

  @override
  Future<List<CommentModel>> retornaComentarios(int postId) async {
    var response =
```

```dart
class CommentsDioRepsositoy implements CommentsRepository {
  final JsonPlaceHolderCustonDio jsonPlaceHolderCustonDio;

  CommentsDioRepsositoy(this.jsonPlaceHolderCustonDio);
  @override
  Future<List<CommentModel>> retornaComentarios(int postId) async {
    var response =
```

Percurso

Etapa 1 Gerenciamento de estado com Provider

Etapa 2 Gerenciamento de estado com MobX

Etapa 3 Gerenciamento de estado com Get

Etapa 4 Boas Práticas e Padrões em Flutter

Etapa 2

Flutter

// Gerenciamento de estado com Provider

Conhecendo o Provider

O provider é um dos pacotes mais usados pela comunidade Flutter. Ele pode ser usados de varias maneiras, e basicamente ele armazena propriedades que podem ser reativas e quando elas são alteradas chamamos um evento “notifyListeners()” que faz com que todos os widgets que observam esta classe sejam notificados e com isso atualizam seus valores.

Exemplo

```
You, 2 days ago | 1 author (You)
class ContadorProviderService extends ChangeNotifier {
  int _contador = 0;

  int get contador => _contador;        You, 2 days ago • Mod

  void incrementar() {
    _contador = _contador + 1;
    notifyListeners();
  }
}
```

```
      ); // Text
      Consumer<ContadorProviderService>(
          builder: (_, contadorService, wiidget) {
        return Text(
          contadorService.contador.toString(),
          style: const TextStyle(fontSize: 26),
        ); // Text
      }), // Consumer
```

Percurso

Etapa 1 ~~Gerenciamento de estado com Provider~~

Etapa 2 Gerenciamento de estado com MobX

Etapa 3 Gerenciamento de estado com Get

Etapa 4 Boas Práticas e Padrões em Flutter

Etapa 2

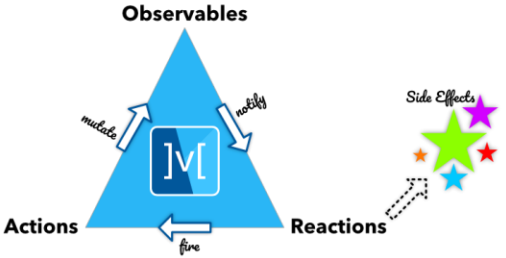
Flutter

// Gerenciamento de estado com MobX

Conhecendo o MobX

O MobX é um gerenciador de estado muito conhecido na comunidade e é vastamente usado por diversas empresas, principalmente em projetos médios a grandes.

Devido a sua forma de trabalhar onde ele se baseia no conceito de observadores e esse só pode ser alterada via uma ação, e esta ação reflete uma reação que altera alerta os widgets que estão escutando essa propriedade e são automaticamente atualizados.



Exemplo

```
You, 2 days ago | 1 author (You)
class CounterMobXService {
  final _contador = Observable(0);
  int get contador => _contador.value;

  late Action incrementar;
  set contador(int newValue) => _contador.value = newValue;
  void _incrementar() {
    _contador.value++;
  }       You, 2 days ago • Modulo7

  CounterMobXService() {
    incrementar = Action(_incrementar);
  }
}
```

```
// The store-class
You, 2 days ago | 1 author (You)
abstract class _CounterMobXSore with Store {
  @observable
  int contador = 0;

  @action
  void incrementar() {       You, 2 days ago • Modu
    contador++;
  }
}
```

```
21          Observer(builder: (context) {
22            return Text(
23              "${contadorMobXService.contador}",
24              style: const TextStyle(fontSize: 26),
25            ); // Text
26          }), // Observer
```

Percurso

Etapa 1 ~~Gerenciamento de estado com Provider~~

Etapa 2 ~~Gerenciamento de estado com MobX~~

Etapa 3 Gerenciamento de estado com Get

Etapa 4 Boas Práticas e Padrões em Flutter

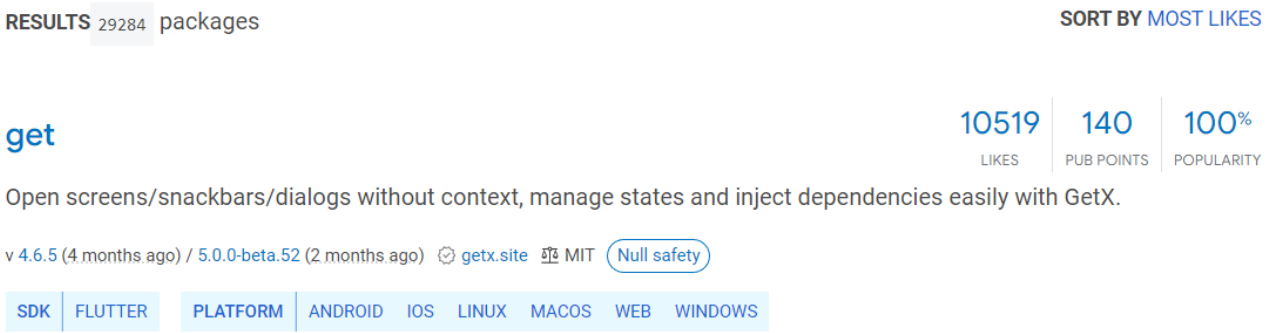
Etapa 3

Flutter

// Gerenciamento de estado com Get

Conhecendo o Get

O Get é o pacote com mais likes no pub.dev e com isso um dos mais usados, ele nos entrega não só gerenciamento de estado, mas também internacionalização, facilidade em rotas, entre muitas outras funcionalidades, devido a isso ele é bem controverso na comunidade, mas com certeza é um ótimo pacote e trás diversas facilidades.



Exemplo

```
class ContadorGetController extends GetxController {
  var contador = 0.obs;
  incrementar() {
    contador = contador + 1;
  }
}
```

```
Obx(() {
  return Text(
    "${contadorGetXService.contador}",
    style: const TextStyle(fontSize: 26),
  ); // Text
}), // Obx
```

Percurso

Etapa 1 ~~Gerenciamento de estado com Provider~~

Etapa 2 ~~Gerenciamento de estado com MobX~~

Etapa 3 ~~Gerenciamento de estado com Get~~

Etapa 4 Boas Práticas e Padrões em Flutter

Etapa 4

Flutter

// Boas Práticas e Padrões em Flutter

Boas Práticas e Padrões em Flutter

Neste módulo trataremos de utilizar práticas para torna nosso projeto mais organizado, separando nas pastas corretas, para que os arquivos fiquem mais fáceis de serem encontrados.

Refartaremos uma parte de nossa lista de tarefas, fazendo com o que o item, fique em um arquivo separado, e podendo ser reutilizado em outras situações.

Get

Conheceremos o pacote Get, que nos ajuda a criar instancias de nossas classes que faz com que as mesmas fiquem visíveis e de forma SingleTon par a aplicação interfira.

Com isso veremos como essa solução nos ajuda para evitar situações de que criemos diversas instancias do mesmo objeto e percamos os dados armazenados.

Links Úteis

- digitalinnovationone/dio-flutter (github.com)
- Dart packages (pub.dev)
- List of state management approaches | Flutter

Para saber mais

Artigos e cursos da DIO

“Fala Devs” youtube

Dúvidas?

> Fórum/Artigos

> Comunidade Online (Discord)

Desafio – Lista de Tarefas

- Criar uma aplicação Flutter
- Criar um banco de dados / Back4App
- Criar uma tela de lista de tarefas
- Listar as tarefas de forma reativa
- Obtendo os dados da base de dados e não da memória
- Utilizar o gerenciamento de estado de sua preferencia