Cubits e Gerenciamento de Estados

Em Flutter, temos mais de uma forma de gerenciar estados como;

- Cubit.
- Provider (utilizando Change Notifier).
- BLoCs.
- GetX.
- MobX.

Neste texto falaremos sobre Cubit. Em conjunto com os Blocs, o Cubit é responsável por disparar **eventos** e emitir **estados**, que serão lidos pela interface. Esses **estados** nada mais são do que dados que são trafegados entre uma base de dados, que pode ser uma variavel ou um banco de dados propriamente dito e a UI do usuario. Por exemplo, imagine que temos uma tela que mostra alguns produtos de uma determinada loja, os dados desta loja serão armazenados em algum banco de dados. Caso não houvesse uma forma de fazer consultas nesse banco de dados de forma reativa, teríamos que implementar muito código diretamente na interface, fazendo com que nosso código de UI e regra de negócios ficassem misturados, gerando assim um código completamente complexo e difícil de entender.
Fazendo uso de Cubit, conseguimos separar esses códigos, tendo arquivos separados para manipular dados e a própria UI. Desta forma nosso código fica organizado e fácil de fazer alterações/manutenções.

Como o Cubit funciona?

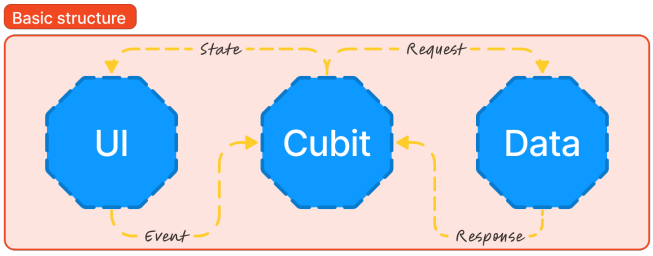
Como dito acima, o Cubit funciona com **eventos** e **estados**. Vamos continuar com nosso exemplo da loja acima. Imagine que precisamos incluir uma nova televisão no nosso banco para ser disponibilizado para venda, neste caso, teremos nome, fabricante, modelo, imagem e preço. Então inserimos nossos dados nos devidos campos, e quando pressionamos o botão para registrar, nosso evento começa.
O evento basicamente é uma chamada de um método dentro de uma classe préviamente configurada para gerênciar todo o estado, e nesse exemplo, esse método receberá nossos parâmetros (nome, fabricante, modelo, imagem e preço) para fazer a inserção no banco de dados. Quando estes dados chegam ao método para serem processados o mesmo atualiza o estado para carregando (aqui sinta-se livre para escolher o nome que dará ao estado) antes de começar a fazer a manipulação dos dados em sequência. Veja na próxima imagem como seria o método.

```dart
Future<void> addTodo({required String todo}) async {
    emit(LoadingTodoState());

    if (_todos.contains(todo)) {
      emit(ErrorTodoState(message: 'This Todo is already input'));
    } else {
      _todos.add(todo);
    }

    await Future.delayed(const Duration(seconds: 2));
    emit(LoadedTodoState(todo: _todos));
  }
```

Este método é um Future que não retorna nenhum dado, quando uma função ou método é assíncrono, ela sempre será um Future e precisamos dela para utilizarmos a keyword **await** que nos permitirá escrever códigos assincronos.
Recebemos como parâmetro obrigatório uma string que por sua vez será inserida na lista de todo.
Veja que na segunda linha, usamos o emit para atualizar o estado da UI para carregando o estado (dados). De forma reativa, como o caso desta aplicação exemplo, na tela para o usuário, aparecerá um indicador circular indicando que algo está carregando.
Logo após fazemos uma checagem para ter certeza de que a string não está vazia, caso estiver, novamente um emit é lançado, mas com um estado de erro desta vez e, caso contrario (string não vazia) a string vai ser inserida na lista. Na linha 10 temos um await para simular um tempo de resposta que pode ser de uma **API** ou **banco de dados**. E terminado todo o processamento, um novo emit é lançado com o estado de carregado, que informa a UI que o processamento terminou e que temos novos dados para serem mostrados. Esses dados podem ser uma notificação ou uma atualização de tela.

Um pouco mais a fundo

Veja na imagem abaixo o fluxo que é percorrido pelos dados ao utilizar um Cubit.



Para esta explicação utilizarei um todo list bem simples que conterá uma app bar, um ListView, um TextFormField e um Botão para fazer a incerção dos dados. A imagem ao lado ilustra nossa tela.

Vamos iniciar então de fato a entender como tudo funciona. Para começar a utilizar nossos estados, precisamos criar algumas classes que serão responsáveis por

gerência-los. Essas classes devem ser criadas em um unico arquivo separado para manter a organização do código e facilitar sua leitura posteriormente.

Vamos precisar construir uma classe abstrata que servirá de modelo para as outras, assim, garantimos que todas terão o mesmo tipo. Escolherei o nome **TodoState** para a minha classe modelo e, todas as outras classes de estados, irão herdar dessa classe modelo. Os nomes das nossas classes de estados serão;

- InitialTodoState.
- LoadingTodoState.
- LoadedTodoState.
- ErrorTodoState.

As classe InitialTodoState e LoadingTodoState serão vazias, pois o único intuito das mesmas será informar a UI em qual estado estamos. A classe LoadedTodoState neste caso, deverá ter um atributo do tipo lista/string. Fácilmente por meio de generics conseguimos tipa-la da seguinte forma List<String>. Assim, temos um dado que pode ser retornado para a UI e posteriormente processado da forma que for necessária.

A classe ErrorTodoState é responsável por emitir um estado de erro como o nome supõe, mas ela também tem um atributo do tipo String que retorna um erro caso este acontecer na aplicação. Após esta configuração, nosso arquivo deve se parecer com a imagem abaixo.

```dart
abstract class TodoState {}

class InitialTodoState extends TodoState {}

class LoadingTodoState extends TodoState {}

class LoadedTodoState extends TodoState {
  final List<String> todos;
  LoadedTodoState({
    required this.todos,
  });
}

class ErrorTodoState extends TodoState {
  final String message;
  ErrorTodoState({
    required this.message,
  });
}

```

Com os nossos estados devidamente configurados, devemos agora começar a escrever a nossa classe que irá gerenciar isso tudo, nela teremos métodos que farão a inserção e retirada dos dados da nossa lista.

Criaremos um novo arquivo que conterá somente a nossa classe de gerenciamento de estado. Neste arquivo, vamos criar uma classe chamada **TodoManagement** que herdará de Cubit passando o generics que iremos utilizar, o qual nesse caso será a nossa classe modelo e abstrata **TodoState.**

Instale o pacote BloC no arquivo yaml ou rodando o comando flutter pub bloc e importe noseu arquivo;

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

Então a nossa classe deverá ficar assim;

```
class TodoManagement extends Cubit<TodoState> {


};
```

Dentro dela criaremos um atributo anônino chamado Todo que também é do tipo List<String> e que será usado para manipular os nossos dados. Precisamos agora iniciar o nosso método construtor e aqui passamos o estado inicial para ele da seguinte forma;

```
class TodoManagement extends Cubit<TodoState> {

    List<String> _todo = [];

    TodoManagement() : super(InitialTodoState());

};
```

Com nosso atributo e método construtor configurado, poderemos agora dar seguimento na construção dos nossos métodos. Vamos então implementar dois métodos que serão **addTodo** e **removeTodo.** São nomes autoexplicativos, mas vamos agora ver de forma geral o corpo desses métodos.

```
Future<void> addTodo({required String todo}) async {
    emit(LoadingTodoState());

    if (_todos.contains(todo)) {
      emit(ErrorTodoState(message: 'This Todo is already inputed'));
    } else {
      _todos.add(todo);
    }

    await Future.delayed(const Duration(seconds: 2));
    emit(LoadedTodoState(todos: _todos));
  }
```

Este método ja foi explicado anteriormente neste **artigo** então, se você não lembra exatamente o que ele faz, sinta-se livre para voltar um pouco e reler a explicação.

Vamos para a ultima parte da configuração do nosso Cubit antes de usa-lo. No método **removeTodo**, temos então um método que é um Future e não retorna nada (o único propósito dele aqui é atualizar o estado), ele recebe como parâmetro um int que será utilizado para apagar a String ou Todo da nossa lista. Na segunda linha, chamamos o nosso atributo e em seguida chamamos um **removeAt**, que irá remover o item da lista baseado no index passado para o método. Logo após temos um **if** que faz uma

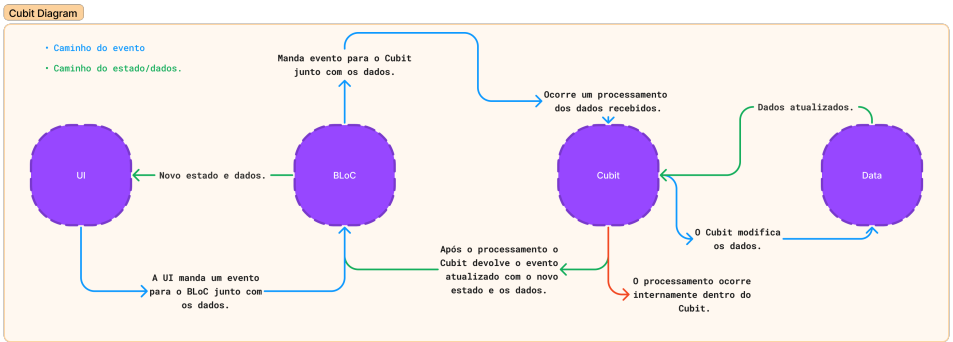
checagem para saber se a lista está vazia ou não e, caso esteja, chamamos somente a classe LoadedTodoState para atualizar a nossa lista para vazia. Uma vez que a lista está vazia, podemos emitir um estado, por meio do **emit(InitialTodoState())** para voltarmos para nosso estado inicial. Se nossa lista não estiver vazia, chamaremos um **emit**() novamente passando o **LoadedTodoState** como parâmetro e assim atualizando a nossa UI.

Sua classe deve se parecer com a imagem abaixo nesse ponto.

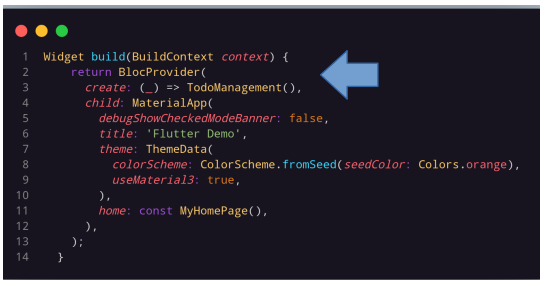
```dart
import 'package:bloc/bloc.dart';
import 'package:cubit/screen/cubit_manager.dart/todo_state.dart';

class TodoManagement extends Cubit<TodoState> {
  final List<String> _todos = [];
  List<String> get todos => _todos;
  TodoManagement() : super(InitialTodoState());

  Future<void> addTodo({required String todo}) async {
    emit(LoadingTodoState());

    if (_todos.contains(todo)) {
      emit(ErrorTodoState(message: 'This Todo is already inputed'));
    } else {
      _todos.add(todo);
    }

    await Future.delayed(const Duration(seconds: 2));
    emit(LoadedTodoState(todos: _todos));
  }

  Future<void> removeTodo({required int index}) async {
    _todos.removeAt(index);
    if (todos.isEmpty) {
      LoadedTodoState(todos: _todos);
      emit(InitialTodoState());
    } else {
      emit(LoadedTodoState(todos: _todos));
    }
  }
}
```

Integrando a UI

Para integrar a UI, precisamos antes entender como os eventos e dados são trafegados entre a UI, BloC e Cubit, no diagrama a seguir, temos discriminado todo o caminho feito. Você precisará ainda se aprofundar no assundo, levando em conta que este é apenas uma ilustração do caminho percorrido.



Agora vamos adicionar nosso BlocProvider na UI e finalmente fazer tudo funcionar. Para isso, adicionarei um BlocProvider antes do retorno do MaterialApp na raiz do nosso app como na imagem abaixo.

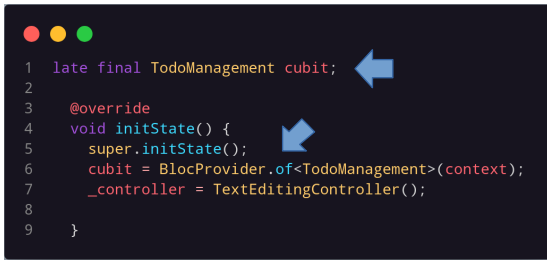


```
Widget build(BuildContext context) {
    return BlocProvider(
      create: (_) => TodoManagement(),
      child: MaterialApp(
        debugShowCheckedModeBanner: false,
        title: 'Flutter Demo',
        theme: ThemeData(
          colorScheme: ColorScheme.fromSeed(seedColor: Colors.orange),
          useMaterial3: true,
        ),
        home: const MyHomePage(),
      ),
    );
  }
```

Após isso vou adicionar um BlocBuilder antes do meu ListView para que seja possivel utilizar os estados. Aqui precisamos passar o parametro bloc que sera uma instancia da nossa classe TodoManagement que vamos recuperar pelo context da seguinte forma:

Definimos a variavel: Late TodoManagement cubit.

Após a variável ser definida, temos que inicia-la dentro do nosso initState da seguinte maneira: cubit = BlocPrrovider.of<TodoManagement>(context);.



```
late final TodoManagement cubit;

  @override
  void initState() {
    super.initState();
    cubit = BlocProvider.of<TodoManagement>(context);
    _controller = TextEditingController();

  }
```

Agora podemos fazer as checagens de estados dentro do builder. Para cada estado, teremos um if de checagem (com exceção do ErrorTodoState que utilizaremos de outra forma). Aqui, deixei meu ListView dento de um método para facilitar a leitura do código.

```
BlocBuilder(
            bloc: cubit,
            builder: (_, state) {
              if (state is InitialTodoState) {
                return SizedBox(
                  height: usefulHeight * 0.91,
                  child: const Center(
                    child: Text(
                      'There`s no data to show',
                      style: TextStyle(fontSize: 20),
                    ),
                  ),
                );
              } else if (state is LoadingTodoState) {
                return SizedBox(
                  height: usefulHeight * 0.91,
                  child: const Center(
                    child: CircularProgressIndicator(),
                  ),
                );
              } else if (state is LoadedTodoState) {
                return todoListView(state.todos);
              } else {
                return todoListView(cubit.todos);
              }
            },
          ),
```

Cada if checa se o state é igual a nossa classe estado que definimos inicialmente, esses states são retornados pelo emit na classe TodoManagement, caso o retorno for verdadeiro, o widget dentro do if será retornado. Veja que alem dos estados possíveis, temos mais um else. Necessitamos dele pos, quando os arquivos forem carregados, uma nova checagem será feita e como nenhuma mudança foi feita no estado, precisaremos retornar somente nossa ListView juntamente com os dados. Aqui não teremos como pegas os dados pelo nosso state, então usaremos nossa variável que foi instânciada mais acima. Com isso nossa tela já deve estar reativa.

Mas e se der erro?

Bem, aqui podemos usar um listen da stream para monitorar quando recebermos erros e reagir da forma que quisermos. Observe na imagem que chamamos nosso cubit.stream.listen((state) {}).

```dart
@override
 void initState() {
   super.initState();
   cubit = BlocProvider.of<TodoManagement>(context);
   _controller = TextEditingController();
   cubit.stream.listen((state) {
     if (state is ErrorTodoState) {
       ScaffoldMessenger.of(context).showSnackBar(
         SnackBar(
           content: Text(
             state.message,
           ),
         ),
       );
     }
   });
 }
```

Aqui faremos uma nova checagem para saber se nosso estado é igual a ErrorTodoState e se for, mostramos uma SnackBar com o erro explícito.

Desta forma poderemos utilizar a programação reativa no nosso aplicativo utilizando o Cubit. O Cubit é simples e rápido de ser implementado, necessitando de pouco código para que tudo funcione corretamente.

Obrigado por ler até aqui. Foi uma explicação um pouco extensa mas acredito que tenha valido a pena, te vejo na próxima.