Flutter Avançado - Aula 8

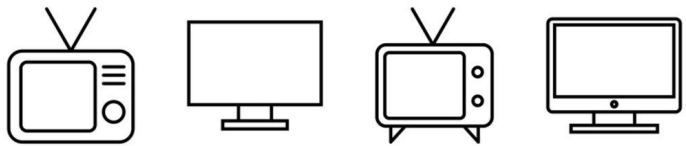
Exercício 21

Fernando Duwe

v 1.0

Dicas de hoje

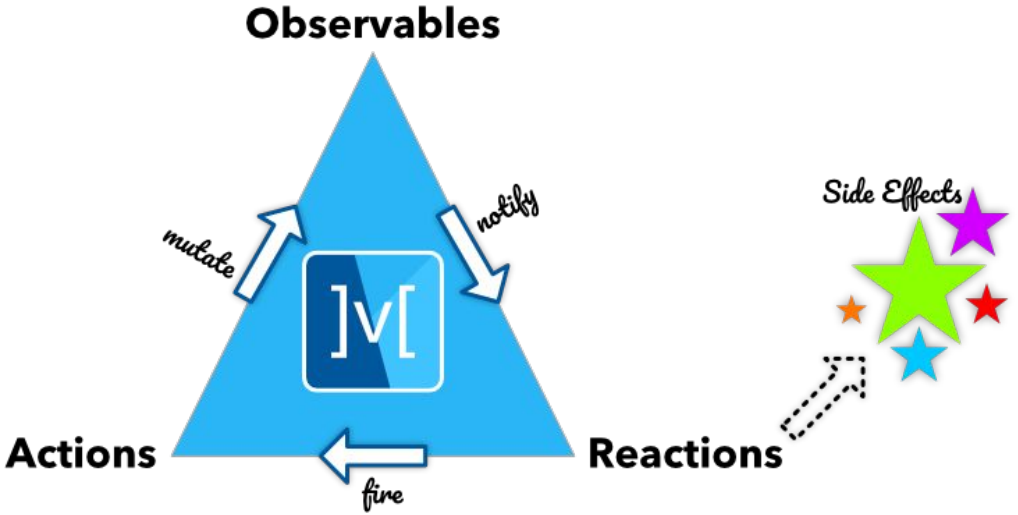
- Flutter TV Ver mais



- Biblioteca de gestão de estados, tanto locais quanto globais
- Reativo
- Muito semelhante ao Redux e BloC, mas com menos código envolvido
- Código auto gerado

Ver mais

- Actions
- Observables
- Computed



1º vamos importar os pacotes necessários para desenvolver com o MobX

Esses pacotes serão responsáveis por implementar o MobX e gerar o código necessário.

```
dependencies:
  mobx: ^2.1.4
  flutter_mobx: ^2.0.6+5
  provider: ^6.0.5


dev_dependencies:
  mobx_codegen: ^2.1.1
  build_runner: ^2.3.3
```

Ver mais

```
import 'package:mobx/mobx.dart';
part 'counterStore.g.dart';

class CounterStore = _CounterStore with _$CounterStore;

abstract class _CounterStore with Store {
  @observable
  int counter = 0;

  @computed
  int get negativeCounter => this.counter * -1;

  @action
  void incCounter() {
    this.counter++;
  }

  @action
  void decCounter() {
    this.counter--;
  }
}
```

2º escrevendo e entendendo a store

Dentro da nossa store, temos um contador, que será o nosso estado. Marcando-o com a diretiva @observable, sempre que ele for alterado, os widgets que o utilizam serão notificados

```
@observable
int counter = 0;
```

2º escrevendo e entendendo a store (cont)

Somente a partir de uma ação poderemos modificar os nosso estados. Para identificar nossas ações, adicionaremos a diretiva @action ao código.

```
@action
void incCounter() {
  this.counter++;
}

@action
void decCounter() {
  this.counter--;
}
```

2º escrevendo e entendendo a store (cont)

Semelhante ao observable, podemos calcular informações a partir dos nossos estados. Para isso, usamos a diretiva @computed.

```
@computed
int get negativeCounter => this.counter * -1;
```

2º escrevendo e entendendo a store (cont)

O part identifica que esse código faz parte do código `counterStore.g.dart`, que será gerado posteriormente.

```
part 'counterStore.g.dart';

class CounterStore = _CounterStore with _$CounterStore;

abstract class _CounterStore with Store {
```

2º escrevendo e entendendo a store (cont)

Criamos a classe abstracta para que não seja possível ao desenvolvedor instancia-la diretamente. Utilizamos um *mixin* com a classe Store.

```
part 'counterStore.g.dart';

class CounterStore = _CounterStore with _$CounterStore;

abstract class _CounterStore with Store {
```

2º escrevendo e entendendo a store (cont)

Por último, declaramos a classe CounterStore, sendo composta pela nossa classe abstracta e pela classe _$, que será gerada pelo MobX.

```
part 'counterStore.g.dart';

class CounterStore = _CounterStore with _$CounterStore;

abstract class _CounterStore with Store {
```

Android Studio - Live Templates

Se eu quiser criar um atalho no código (semelhante ao stless) para criar meu mobx ou qualquer outra estrutura, no Android Studio, podemos utilizar os *live templates™*.

Para acessar os templates, vá em File / Settings / Editor / Live Templates. Selecione Flutter.

```
import 'package:mobx/mobx.dart';
part '$FILENAME$.g.dart';

class $CLASSNAME$ = _$CLASSNAME$ with _$$$CLASSNAME$;

abstract class _$CLASSNAME$ with Store {

}
```

3º gerando o código fonte da nossa Store

Para executar o código fonte, execute o comando abaixo no terminal.

```
flutter packages pub run build_runner build
```

Para que o código seja gerado sempre que o desenvolvedor salvar o arquivo, execute o comando abaixo no terminal.

```
flutter packages pub run build_runner watch
```

4º utilizando a nossa Store nos nossos widgets

Para alterar os dados da nossa store, basta executar as ações.

```
    return Scaffold(
      appBar: AppBar(
        title: Text("Contador, só que top demais"),
        actions: [
          IconButton(onPressed: counter.incCounter, icon: Icon(Icons.add)),
          IconButton(onPressed: counter.decCounter, icon: Icon(Icons.remove))
        ],
      ),
```

5º modificando o layout do nosso widget sempre que a store mudar

Para alterar os dados da nossa store, basta executar as ações.

```
body: Observer(
        builder: (context) {
          return Column(
              children: [
                Text("Contador: ${counter.counter}"),
                Text("!Contador: ${counter.negativeCounter}",
                    style: TextStyle(color: Colors.blueGrey)),
              ],
          );
        },
      ),
```

6º tornando nossa store global, com Provider

Com o provider, ao criar nosso App, criaremos nossa store.

```
return Provider<ContactListStore>(
      create: (context) => new ContactListStore(),
      child: MaterialApp(
          debugShowCheckedModeBanner: false,
          initialRoute: "/counter",
          routes: {
            "/": (context) => HomePage(),
            "/counter": (context) => HomeCounter(),
          }),
    );
```

7º recuperando nossa store, nos widgets

A partir do provider, poderemos buscar a store dentro da nossa árvore de widgets, baseado no tipo da store.

```
ContactListStore vrStore = Provider.of<ContactListStore>(context);
```

Exercício 21

Ver mais

- Para fixarmos a utilização do MobX, vamos recriar a nossa calculadora

- Utilize o MobX para ver a diferença de código e as facilidades que o MobX oferece