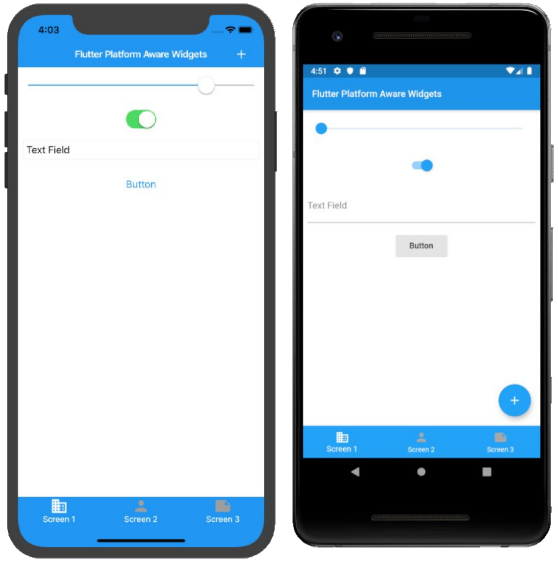
Flutter Avançado - Aula 6

Fernando Duwe

v 1.0

Dicas de hoje

- Youtube: EVERY Flutter Cupertino Widgets [Ver mais]



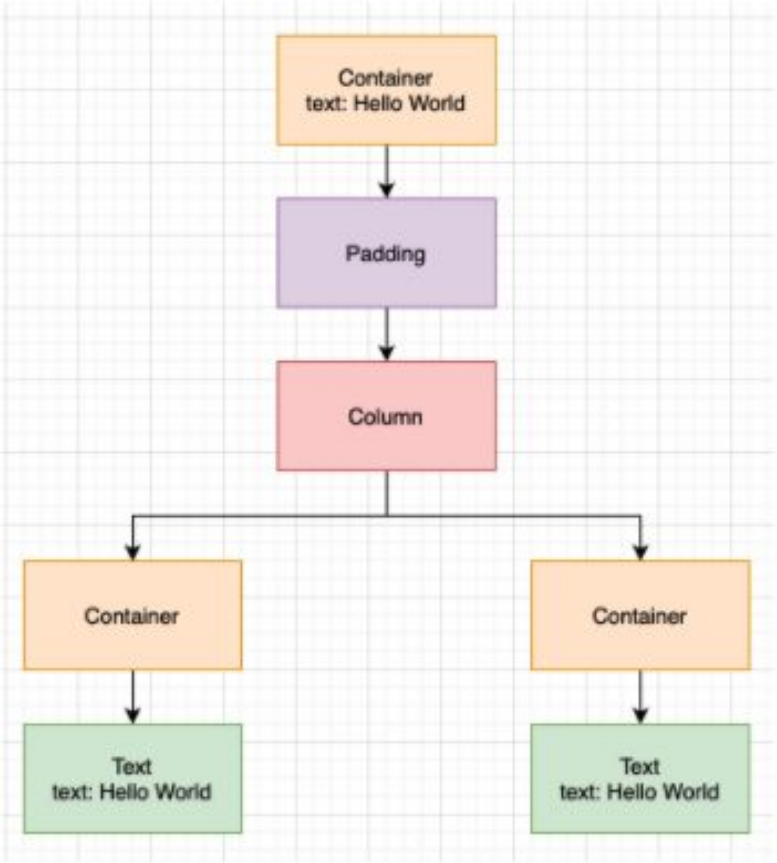
- A arquitetura de aplicações são as técnicas e padrões usados para projetar e desenvolver uma aplicação
- Fornece um caminho e práticas recomendadas a serem seguidos durante o desenvolvimento, para que tenhamos uma aplicação bem estruturada
- Design patterns
- Architectural patterns

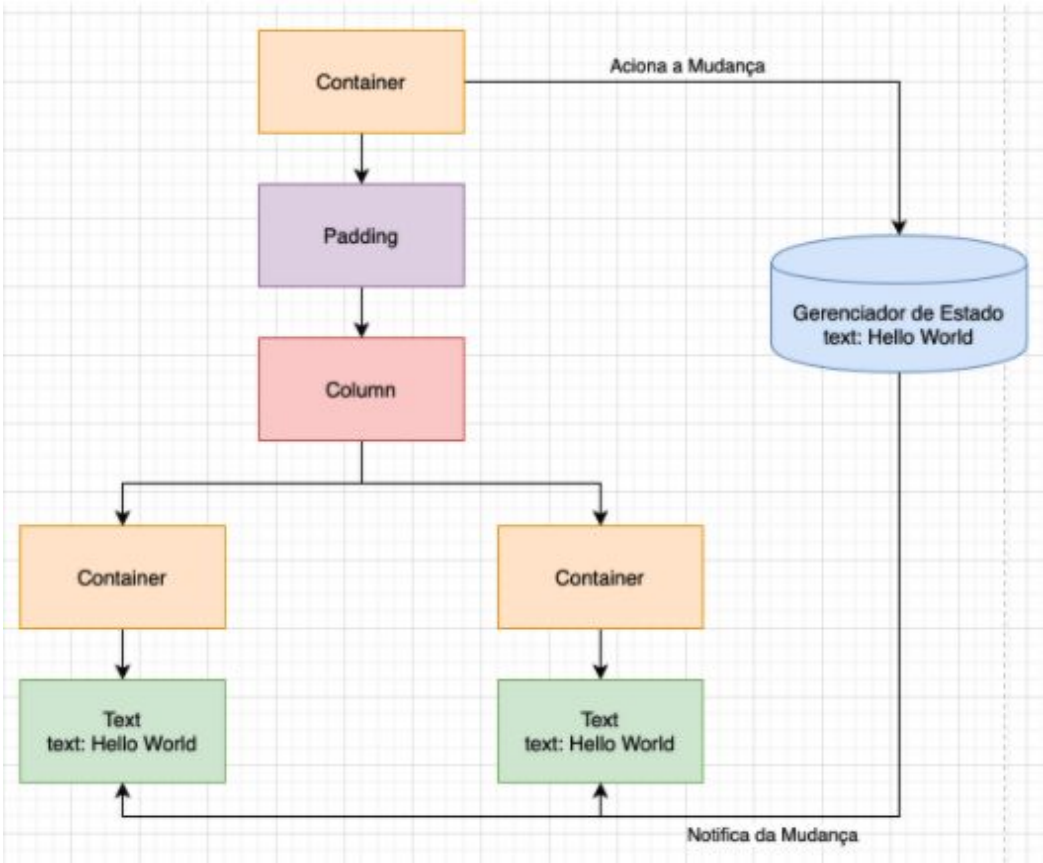
- O que são os estados?
- setState: a opção mais simples, mas não tão performático
- Como gerenciar meus estados de uma forma otimizada?

STATE

Gerenciamento de estados - cont

Ver mais



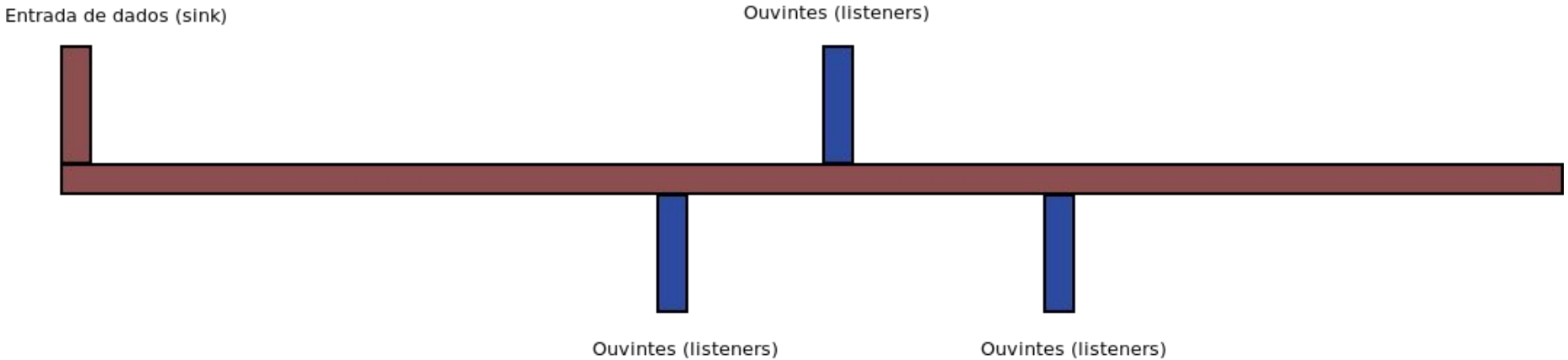


Ver mais

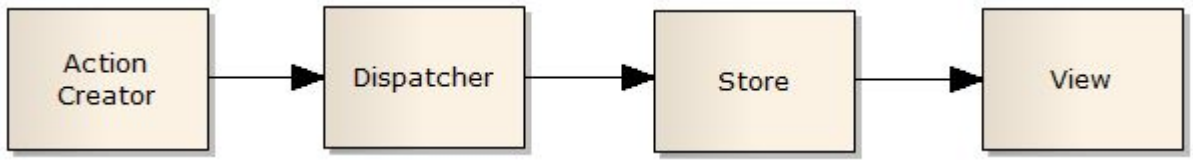
- Algumas opções para de gerenciamento de estados:
    1. Redux
    2. BLoC
    3. MobX

STATE

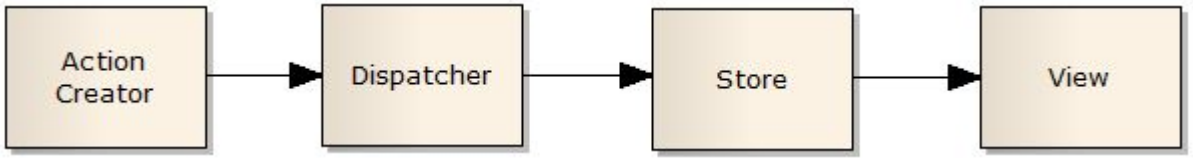
- Stream (fluxo)
- Fluxo assíncrono de dados
- Podemos adicionar listeners, que serão notificados sempre que uma nova informação for enviada através do fluxo



- Um padrão arquitetural (*architectural pattern*) proposto pelo *Facebook*
- Seu objetivo é separar os estados dos widgets
    - **Action:** Evento que disparará uma ação
    - **Dispatcher:** Recebe a ação e repassa as stores necessárias
    - **Store:** Armazena os dados e altera-os conforme a ação recebida
    - **View:** Visão do usuário, os widgets



- Biblioteca criada pelo Facebook, para o React
- Implementa a arquitetura Flux
- Bibliotecas necessárias: *flutter_redux* e *redux*



1º vamos criar um estado Essa será a classe imutável que guardará os nossos dados, ou estados

```
class ContactListState {
  late List<ContactModel> list;

  ContactListState(this.list);

  int get favoriteCounter {
    int vrCounter = 0;

    list.forEach((element) {
      if (element.favorito)
        vrCounter++;
    });

    return vrCounter;
  }
}
```

Ver mais

2º vamos criar um reducer

O método reducer é que efetuará as alterações no estado atual

```
ContactListState contactListReducer(ContactListState prContactList, dynamic prAction) {
  if (prAction.action == ContactListActionType.Add) {
    List<ContactModel> vrList = prContactList.list;

    vrList.add(prAction.model);

    return ContactListState(vrList);
  }

  if (prAction.action == ContactListActionType.Remove) {
    List<ContactModel> vrList = prContactList.list;

    for (var i = vrList.length - 1; i >= 0; i--) {
      if (vrList[i].nome == prAction.model.nome)
        vrList.removeAt(i);
    }

    return ContactListState(vrList);
  }
```

Porque criar um estado imutável e usar o reducer?

A ideia por trás de utilizar o reducer é de que, como o estado é imutável, somente o reducer será responsável por alterar alguma informação dos estados. Nesse caso, se uma informação do estado for alterada, basta ao desenvolvedor buscar em um único local, facilitando a manutenção do código escrito, ou a busca de algum bug que pode surgir.

3º vamos criar uma store

A Store guarda o estado atual, além do *reducer*.

É a partir dela que os nossos estados serão visualizados e manipulados.

```
final Store<ContactListState> contactListStore = Store<ContactListState>(
    contactListReducer, initialState: ContactListState([])
);
```

4º Tornando nossa store visível em todo o projeto

No nosso método main, antes de criarmos nosso *MaterialApp*, vamos utilizar o widget *StoreProvider*.

O *StoreProvider* será responsável por injetar nossa *store* dentro da árvore de *widgets*.

```
return StoreProvider(
        store: contactListStore,
        child: MaterialApp(
          debugShowCheckedModeBanner: false,
          theme: ThemeData.dark(),
          initialRoute: "/",
          routes: {
            "/": (context) => HomePage()
          },
        ));
```

5º Disparando ações da nossa Store

Para enviar ações a nossa store, basta chamar - método dispatch, enviando por parâmetro a ação desejada. Nos casos abaixo, encapsulamos a ação enviando junto os dados do contato, além da ação desejada.

```
contactListStore.dispatch(
          ContactListAction(ContactModel(
                  nome: this.nomeController.text,
                  email: this.emailController.text), ContactListActionType.Add));
```


```
contactListStore.dispatch(
          ContactListAction(ContactModel(
                  nome: this.nomeController.text,
                  email: this.emailController.text), ContactListActionType.Add));
```

6º Fazendo as alterações visíveis nos widgets - 1ª forma

Utilizando abaixo o StoreBuilder para atualizar os widgets conforme a store muda. O StoreBuilder ouvirá qualquer alteração na store e alterará toda a árvore de widgets.

```
StoreBuilder<ContactListState>(
        builder: (BuildContext context, Store<ContactListState> store) {
          return ListView.builder(
              itemCount: store.state.list.length,
              itemBuilder: (context, index) {
                return ContactListTile(model: store.state.list[index]);
              });
        }
      ),
```

6º Fazendo as alterações visíveis nos widgets - 2ª forma

Utilizando abaixo o StoreConnector para atualizar os widgets conforme a store muda. O *StoreConnector* é mais performático, pois além de atualizar os widgets conforme o state muda. Posso separar o estado da visão, criando um *view model*. Esse *view model* ao ser criado, pode utilizar as informações relevantes a somente uma parte daquele widget. O Redux será inteligente a ponto de saber quais pontos do sistema devem ou não ser atualizados.

```
StoreConnector<ContactListState, ContactListState>(
      converter: (store) {
        return store.state;
      },
      builder: (context, vm) {
        return Text(vm.favoriteCounter.toString());
      },
    );
```