SI700 – Programação para Dispositivos Móveis

Aula 6 – Gerenciamento de Estados (Parte II)

Prof. Ulisses Martins Dias

2023

Faculdade de Tecnologia – Unicamp

Roteiro

Mudando de Telas com o BlocBuilder

ListView

Drawer

Navigator

Introdução

Em aulas anteriores, temos aprendido formas de fazer a movimentação entre telas. O que aprendemos até o momento foi:

1. Usando *swype* com **PageView**. Nesse caso, as telas que navegaremos serão pré-alocadas passando um lista de telas ao parâmetro **children** do **PageView**.
2. Usando abas com a **TabBar**. Nesse caso, o **DefaultTabController** faz a mudança de telas, que serão pré-alocadas passando um lista de telas ao parâmetro **children** da **TabBarView**. Este padrão pode até ser colocado em um **StatelessWidget**, dado que o **DefaultTabController** é quem faz a mudança de telas.
3. Usar um **IndexedStack** com o **BottomNavigationBar**. Nesse caso, alocamos as telas que precisamos trocar no **IndexedStack** e depois fazemos as mudanças de tela programaticamente colocando os gatilhos na **BottomNavigationBar**.

Introdução

Na aula de hoje, tentaremos encerrar este assunto de mudança de telas, adicionando dois novos padrões.

- Usando um **Builder**, que em nosso caso será um **BlocBuilder** para tormar uma decisão sobre qual tela deve ser mostrada. Este padrão permite criar sessões privadas em um aplicativo.
- Usando uma **ListView** para efetuar a mudança de telas. Nesse caso, a mudança de telas pode ser feita da seguinte forma:
  - Com um **IndexedStack** como fizemos com o **BottomNavigationBar**. Isso ocorre porque o uso do **IndexedStack** é um padrão que pode ser combinado com qualquer elemento da interface gráfica.
  - Com o **Navigator**, que recebe rotas por meio da invocação de um método **push**. Esse método empilha uma nova tela "em cima" da tela atual, sendo que esta pode ser desempilhada programaticamente ou pelo botão voltar do dispositivo.

Mudando de Telas com o BlocBuilder

Mudando de Telas com o BlocBuilder

Vimos na aula passada que um **BlocBuilder** reage a mudanças de estados em um **Bloc**. Nesse caso, para usar o **BlocBuilder**, precisamos de:

- Uma a estrutura de um **Bloc**. Neste exemplo, o nosso **Bloc** fará uma autenticação.
- Uma tela que envie *login* e *senha* para o **Bloc**.
- Um botão que envie um pedido para desautenticar o usuário.
- Eventos e estados para tornar tudo isso possível.

Vamos começar com os Eventos, temos apenas dois:

```
abstract class AuthEvent {}

class LoginUser extends AuthEvent {
  String username;
  String password;

  LoginUser({required this.username, required this.password});
}

class Logout extends AuthEvent {}
```

Vamos agora para os Estados, temos apenas três:

```
abstract class AuthState {}

class Unauthenticated extends AuthState {}

class Authenticated extends AuthState {
  String username;
  Authenticated({required this.username});
}

class AuthError extends AuthState {
  final String message;
  AuthError({required this.message});
}
```

Mudando de Telas com o BlocBuilder

Podemos agora propor um **Bloc** que aceita a senha "senha" de qualquer usuário.

```
on<LoginUser>((LoginUser event, Emitter emit) {
      if (event.password == "senha") {
        emit(Authenticated(username: event.username));
      } else {
        emit(AuthError(message: "Erro com ${event.username}"));
      }
    });
    on<Logout>((Logout event, Emitter emit) async {
      emit(Unauthenticated());
    });
```

Mudando de Telas com o BlocBuilder

Por fim, um **BlocBuilder** ou **BlocConsumer** deve devolver telas diferentes, dependendo do estado:

```
return BlocConsumer<AuthBloc, AuthState>(
      listener: (context, state) {
        if (state is AuthError) {
          showDialog(
              context: context,
              builder: (context) {
                return AlertDialog(
                  title: const Text("Impossível Logar"),
                  content: Text(state.message),
                );
              });
        } },
```

Mudando de Telas com o BlocBuilder

```
return BlocConsumer<AuthBloc, AuthState>(
   builder: (context, state) {
     if (state is Authenticated) {
       return Column(
         children: [
           Text("Você está autenticado ${state.username}"),
           ElevatedButton(
                onPressed: () {
                  BlocProvider.of<AuthBloc>(context).add(Logout());
                },
                child: const Text("Logout"))
         ],
       );
     } else {
       ...
```

Mudando de Telas com o BlocBuilder

```
if (state is Authenticated) {
  ...
} else {
  return Center(
        child: Form(
          key: formKey,
          child: Column(
            children: [
              usernameFormField(),
              passwordFormField(),
              const Divider(),
              submitButton(),
            ],
          ),
        ),
```

ListView

ListView

- Antes de Falar do **Drawer**, precisamos falar da **ListView**.
- **ListView** é talvez o widget mais onipresente em aplicativos para dispositivos móveis.
- Em sua forma mais simples, você declarará o widget **ListView** e depois colocará como filhos no parâmetro **children** uma série de **ListTiles**. Entretanto, note que os filhos de uma **ListView** podem ser qualquer coisa que você desejar.
- Uma **ListTile** pode ter **title**, **subtitle**, **leading** e **trailing** para colocar informações. Em **leading** e **trailing**, é comum adicionar ícones que ficam antes e depois do que está em **title**, respectivamente.
- Uma **ListView** pode fazer scroll na vertical ou na horizontal, dependendo do que você colocar na propriedade **scrollDirection**.

ListView

```
ListView(
  children: [
    child: ListTile(
      title: const Text('Ulisses Martins Dias'),
      subtitle: const Text("Professor de SI700"),
      leading: const Icon(Icons.access_time),
      trailing: const Icon(Icons.add_a_photo),
      onTap: () {},
    ),
  ),
);
```

Note que **leading** e **trailing** podem ser clicáveis. Nesse contexto, a escolha do ícone é importante para que o usuário note que existe essa funcionalidade.

```
ListTile(
  title: const Text('Luís Meira'),
  trailing: GestureDetector(
    child: const Icon(Icons.delete),
    onTap: () {},
)),
```

ListView

Um **Container** pode também ser filho de **ListView**.

```
ListView(
  children : [
    Container(
      decoration: const BoxDecoration(
        gradient: LinearGradient(
          colors: [Colors.blue, Colors.red, Colors.yellow,
↪  Colors.green]),
      ),
      child: const ListTile(
        title: Text('Outro Professor Qualquer'),
      ),
    ),
], );
```

ListView

- No caso de termos muitos elementos para colocar na **ListView**, é interessante usar um construtor que garanta um pouco mais de eficiência.
- No exemplo a seguir, queremos criar uma **ListView** com **1000** termos. Não queremos colocar todos esses termos na memória, então usaremos um construtor que vai garantir a criação apenas daqueles visíveis ao usuário.
- Caso o usuário faça uma rolagem para outro ponto da **ListView**, o construtor vai tratar de gerar novos widgets para popular a **ListView**.

ListView

```
var items = [];
for (var i = 0; i < 1000; i++) {
  items.add("Item $i");
}

ListView.builder(
  itemBuilder: (context, index) {
    return ListTile(
      title: Text(items[index]
      )
    );
  }
)
```

Drawer

Drawer

- Um uso muito comum de uma **ListView** é com o parâmetro **drawer** do widget **Scaffold**. Isso permite criar o menu lateral comum em aplicativos.
- Neste caso, é importante o uso de **DrawerHeader** como primeiro filho da **ListView**. Isso permite criar uma seção padronizada no canto superior esquerdo do **Drawer** para adicionar informações.
- Note também a invocação de Navigator.pop(context) para fechar a tela do NavigationDrawer quando algum item do menu for clicado. Você deverá adicionar essa invocação no parâmetro **onTap** de todos os **ListTiles**.

Drawer

```
ListView(children: [
    const DrawerHeader(
      child: Text('Drawer Header'),
      decoration: BoxDecoration(
        color: Colors.blue,
      ),
    ),
    ListTile(
      leading: Icon(Icons.cake),
      title: Text("1"),
      onTap: () {
        Navigator.pop(context);
      },
      trailing: Icon(Icons.pets),
),  ], );
```

Drawer

O Drawer pareado com uma **IndexedStack** pode mudar telas.

```
Scaffold(
  body: IndexedStack(index: _currentScreen,
    children: [Tela1(), Tela2()] )
  drawer: Drawer(
    child: ListView(children: [
        const DrawerHeader(child: Text(""), ),
        ListTile(
          title: const Text("Tela 1"),
          onTap: () {
            setState(() {
                _currentScreen = 0;
                Navigator.pop(context);
}); }, ), ], ), ), );
```

Navigator

Navigator

- Podemos fazer mudança de telas utilizando o **Navigator** proporcionado pelo nosso **MaterialApp**. Para isso, precisamos entender um pouco mais sobre **routes**.
- Todo aplicativo precisa de pelo menos uma **route**. Até o momento, temos utilizado o parâmetro **home** do **MaterialApp** para adicionar uma **route**, que naquele momento era adicionada simplesmente como um *widget*. Isso significa que aquela **Route** apareceria por padrão no carregamento do nosso aplicativo.
- Como não colocamos nenhuma outra **Route** em aulas anteriores, as nossas mudanças de tela eram feitas sem alterar a **Route**.

Navigator

- Como o **Navigator**, podemos adicionar novas **Routes** que farão o empilhamento de uma tela sobre outra.
- O termo "empilhamento" aqui é de suma importância, pois as telas realmente são empilhadas, podendo ser desempilhadas programaticamente ou com o botão voltar do dispositivo.
- Perceba então que todas as vezes que invocamos um **push** no **Navigator**, uma tela é colocada no topo, e toda vez que invocamos um **pop**, a tela do topo é removida revelando na tela do usuário a tela anterior.
- Existem outras ações além de **push** e **pop**, mas recomendo que se atenha ao básico por um tempo.
- Na **widget tree**, o efeito do empilhamento é que criamos uma nova ramificação filha de **MaterialApp**, o que pode, por exemplo, ter impacto na visibilidade dos elementos da árvore.

- No caso de um **Bloc** na *widget tree*, podemos fazer uma distinção entre os que têm acesso **local** e os que têm acesso **global**.

  Global: **Blocs** visíveis em todas as **Routes**. Eles precisam ser adicionados como pais de **MaterialApp**.

  Local: **Blocs** visíveis apenas na **Route** onde eles foram adicionados. Eles foram adicionados na widget tree em algum lugar abaixo de **MaterialApp**.

- Um problema pode ocorrer quando você tem um **Bloc** de acesso local precisa ser passado para uma outra **route**.
- Vamos supor que nem todas as **routes** do seu aplicativo precisam daquele **Bloc**, o que tornaria a decisão de deixar aquele **Bloc** como global questionável.
- Nesse caso, o que você precisa fazer é passaro **Bloc** para a nova **route** de alguma forma. O modo como você escreverá o seu código irá mudar dependendo de como você usa o Navigator.

Navigator

Por exemplo, você pode usar o Navigator e passar uma **route** anônima.

```
Navigator.push(
  context,
  MaterialPageRoute(builder: (_) {
      return MyTabBarLayout();
    },
));
```

Nesse caso, o que você deve fazer é passar adiante o **route** que você possui usando o construtor nomeado **value** do **BlocProvider**.

```
Navigator.push(
  context,
  MaterialPageRoute(builder: (_) {
      return
       BlocProvider.value(
         value: existingBloc,
         child: MyTabBarLayout();
       )
    },
));
```

O efeito disso é:

- Será passado para a nova tela um **Bloc** que já existe, sendo que o mesmo pode ser usado então na nova **route**, bem como na antiga. Mudanças em uma **route** geram mudanças na outra.
- A **route** que recebeu o Bloc com o construtor nomeado não irá destruir o Bloc quando for desempilhada. Nesse caso, se você quiser destruir o Bloc, precisará fazer isso manualmente.

Navigator

No caso de você ter um aplicativo com vários **Blocs** e várias **routes**, é recomendável que você crie todas as suas rotas em uma classe externa.

Essa classe deve possuir um método que recebe um parâmetro do tipo **RouteSettings** e devolve um objeto do tipo **Route**.

```
class AppRouter {
  final CounterBloc _counterBloc = CounterBloc();
  final AuthBloc _authBloc = AuthBloc();

  Route onGenerateRoute(RouteSettings routeSettings) {
  }
}
```

O objetivo é passar o método dessa classe ao parâmetro **onGenerateRoute** do **MaterialApp**.

- Os **Blocs** e qualquer variável necessária para se comunicar entre uma **route** e outra podem ser colocados como atributos.
- Nesse caso, você sempre usará o construtor nomeado **BlocProvider**.**value** para passar o **Bloc** entre uma classe e outra.

- Note, no entanto, que se você utilizar **onGenerateRoute** do **MaterialApp**, então não mais poderá utilizar o parâmetro **home**.
- Nesse caso, uma das rotas que você deve gerar será a "/", que substitui o **home** para efeitos práticos.
- A seguir, veja um código funcionando.

Navigator

```
class AppRouter {
  final CounterBloc _counterBloc = CounterBloc();
  final AuthBloc _authBloc = AuthBloc();

  Route onGenerateRoute(RouteSettings routeSettings) {
    switch (routeSettings.name) {
      case "/":
        return MaterialPageRoute(builder: (_) => const
↪   DrawerNavigator());
```

```
      case "/counter":
        return MaterialPageRoute(
            builder: (_) => BlocProvider.value(
                value: _counterBloc,
                child: Scaffold(
                  appBar: AppBar(
                    title: const Text("Counter Screen"),
                  ),
                  body: const CounterScreen(),
                )));
```

```
      case "/auth":
        return MaterialPageRoute(
            /*
              Abaixo um MultiBlocProvider sem necessidade,
              apenas para mostrar o uso
              dele mais uma vez.
          */
            builder: (_) => MultiBlocProvider(
                    providers: [
                      BlocProvider.value(value: _authBloc),
                    ],
                    child: Scaffold(
                      appBar: AppBar(
                        title: const Text("Auth Screen"),
                      ),
```