SI700 – Programação para Dispositivos Móveis

Aula 5 – Gerenciamento de Estados

Prof. Ulisses Martins Dias

2022

Faculdade de Tecnologia – Unicamp

Roteiro

IndexedStack

BottomNavigationBar

BLoC

BlocProvider

BlocBuilder

Builder e BuildContext.watch

Lançando Eventos para o BLoC

Outros Widgets Importantes

IndexedStack

IndexedStack

- **IndexedStack** é um widget que permite que você troque o que o usuário está vendo em um determinado momento.
- Imagine o **IndexedStack** como uma televisão em que você troca aquilo que o usuário está vendo, sem perder o estado do "canal" que acabou de ser retirado.
- A única vantagem do **IndexedStack** é justamente preservar os estados de todos os seus filhos.

IndexedStack

```
IndexedStack(
  index : _widgetIndex,
  children : [
    WidgetOne(),
    WidgetTwo()
  ]
);

// Em algum outro lugar do código
setState(
  () => _widgetIndex = 2
);
```

BottomNavigationBar

BottomNavigationBar

- **BottomNavigationBar** permite colocar uma barra de botões na parte inferior da tela, o que pode ser usado como uma forma de navegação entre telas em conjunto com o **IndexedStack**.
- Diferente de **TabBar** visto na aula anterior, o **BottomNavigationBar** precisará ser construído dentro de um **StatefulWidget**, ou com algum outro sistema de gerenciamento de estados. Isso ocorre porque o **BottomNavigationBar** precisa que você realize a troca de telas no seu próprio código.
- No código a seguir, vamos assumir que temos na classe **State** associada a um **StatefulWidget** uma declaração do atributo **_currentScreen**. Além disso, temos no **body** do **Scaffold** uma **IndexedStack** com três telas.

BottomNavigationBar

```
class _MyHomePageState extends State<MyHomePage> {
  var _currentScreen = 0;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(widget.title)),
      body: IndexedStack(
        index: _currentScreen,
        children: [
          FirstScreen(),
          SecondScreen(),
          ThirdScreen(),
        ],
      ), ... )
} }
```

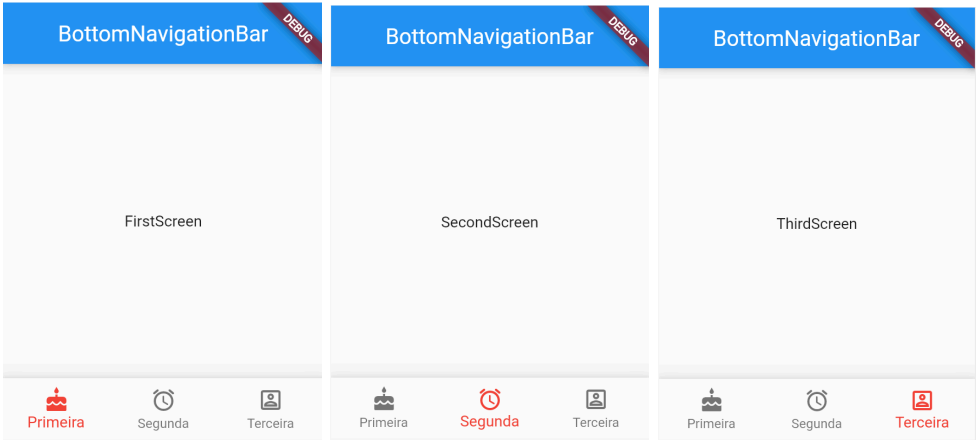
BottomNavigationBar

- Em **Scaffold**, devemos inserir um objeto da classe **BottomNavigationBar** no parâmetro **bottomNavigationBar**.
- O construtor da classe **BottomNavigationBar** possui vários parâmetros que precisamos utilizar em conjunto para gerar a tela:
  - **items**: é onde adicionamos o layout dos botões, o que é geralmente feito com a classe **BottomNavigationBarItem**, que permite incluir um texto e adicionar um ícone.
  - **currentIndex**: informamos qual aba deve estar ativa no momento. Isso permite fazer com que o botão da aba ativa tenha uma cor diferente.
  - **onTab**: informamos o comportamento quando um dos botões for clicado.
  - **fixedColor**: cor que irá realçar o botão da aba selecionada.

BottomNavigationBar

```
Scaffold(
  bottomNavigationBar: BottomNavigationBar(
    items: const [
      BottomNavigationBarItem(
        icon: Icon(Icons.cake), label: "Primeira"),
      BottomNavigationBarItem(
        icon: Icon(Icons.access_alarms_rounded), label: "Segunda"),
      BottomNavigationBarItem(
        icon: Icon(Icons.account_box_outlined), label: "Terceira"),
    ],
    currentIndex: _currentScreen,
    onTap: (int novoItem) {
      setState(() => _currentScreen = novoItem);   },
    fixedColor: Colors.red,
));
```

BottomNavigationBar

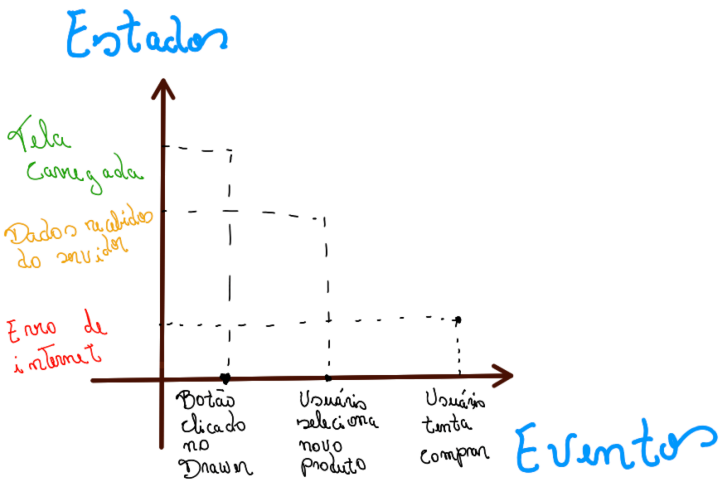


BLoC

BLoC

- BLoC é a sigla para Business Logic Object Components, nada mais sendo do que a separação das regras de negócio da sua aplicação em relação à interface gráfica.
- Vamos imaginar que estejamos construindo um aplicativo que tenha duas interfaces, uma específica para Android e outra específica para iOS. Embora as telas sejam diferentes, as regras de negócio são as mesmas.
- A independência das regras de negócio faz com que os aplicativos sejam mais fáceis de manter.
- BLoC permite guardar estados que podem mudar após a ocorrência de certos eventos. A mudança de estados pode causar uma nova renderização da interface gráfica.

Em geral, podemos imaginar BLoC como um mapeamento de estados para eventos.
Você fará o projeto da camada de controle dessa forma.



Para definir o BLoC, é preciso responder as seguintes perguntas:

- Quem será o estado inicial?
- Quais interações ocorrerão com o BLoC?
- Quais serão os estados gerados pelo BLoC?
- Como os eventos gerarão novos estados?

```
class RedBloc extends Bloc<RedEvent, RedState> {
  RedBloc(RedState initialState) : super(initialState) {
    on<SemRed>((event, emit) => emit(RedState(amount: 0)));
    on<PoucoRed>((event, emit) => emit(RedState(amount: 50)));
    on<NormalRed>((event, emit) => emit(RedState(amount: 150)));
    on<MuitoRed>((event, emit) => emit(RedState(amount: 255)));
  }
}
```

Os estados de um BLoC serão definidos como objetos de qualquer tipo. Você definirá os tipos como classes, enums, ou usará tipos pré-definidos.

```
class RedState {
  int amount;
  RedState({this.amount = 0});
}
```

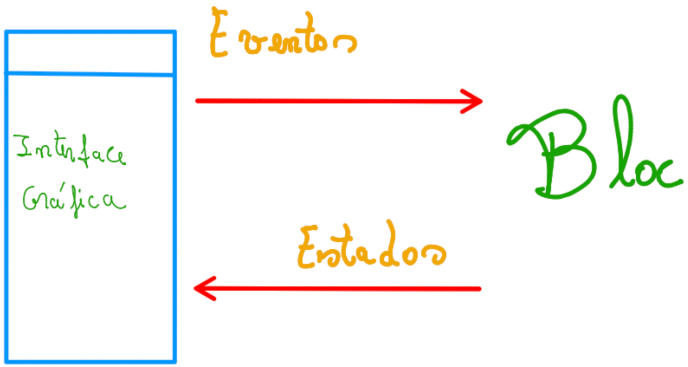
O mesmo acontece com os eventos, são definidos como tipos.

```
abstract class RedEvent {}

class MuitoRed extends RedEvent {}

class NormalRed extends RedEvent {}

class PoucoRed extends RedEvent {}

class SemRed extends RedEvent {}
```

Ressaltando que tanto estados quanto eventos podem ser definidos com o comando enum.

```
enum  RedEvent {
  SemRed,
  PoucoRed,
  NormalRed,
  MuitoRed
}
```

A comunicação do BLoC com a GUI se dará recebendo desta uma stream de eventos que gerarão possivelmente mudanças de estados.



BlocProvider

BlocProvider

- Para que a Interface Gráfica se comunique com um BLoC, ele deverá estar na Widget Tree.
- Um BLoC pode ser adicionado em qualquer ponto da Widget Tree, sendo que essa posição definirá a visibilidade dele.
- Se um BLoC for adicionado acima (como pai) de **MaterialApp**, então será visível em toda a aplicação.
- A adição de um BLoC na Widget Tree deve ser feita por um **BlocProvider**.

BlocProvider

- O construtor default de **BlocProvider** permite que você crie um novo BLoC e o ciclo de vida dele será automaticamente gerenciado.
- Em outras palavras, se o ramo da árvore que contém o BLoC for removido, então o BLoC será finalizado.

```
BlocProvider(create: (_) =>
  RedBloc(RedState(amount: 50))
),
```

- O construtor nomeado **BlocProvider.value** permite que você adicione uma instância já existente do BLoC.
- É função de quem criou o BLoC finalizar o ciclo de vida dele, sendo que o mesmo continuará a existir mesmo que o ramo da árvore onde ele foi inserido seja finalizado.

```
RedBloc sharedRed = RedBloc(
  RedState(amount: 0)
);

// Posteriormente
BlocProvider.value(value: sharedRed),
```

BlocProvider

A finalização de um BLoC é feita invocando o método **close**. Para tanto, use a chamada ao método **dispose** do ciclo de vida do **StatefulWidget**.

```
class _MyHomePageState extends State<MyHomePage> {
  final RedBloc sharedRed = RedBloc();
  final GreenBloc sharedGreen = GreenBloc();
  final BlueBloc sharedBlue = BlueBloc();
  // ...
  void dispose() {
    sharedBlue.close();
    sharedGreen.close();
    sharedRed.close();
    super.dispose();
  }
}
```

BlocProvider

Se quisermos inserir vários BLoCs em um mesmo ponto, usamos um **MultiBlocProvider** para conter vários **BlocProviders**.

```
MultiBlocProvider(providers: [
    BlocProvider(create: (_) => Bloc1()),
    BlocProvider(create: (_) => Bloc2()),
    BlocProvider(create: (_) => Bloc3()),
  ], child: const SomeWidget()),
```

BlocBuilder

BlocBuilder

- O **BlocBuilder** é o componente que criará um novo widget em resposta a mudanças de estados.
- O **BlocBuilder** irá requerer um BLoC e uma função adicionada no parâmetro **builder**. Essa função irá receber o **BuildContext** daquele ponto na árvore e também o novo estado gerado pelo BLoC para que a geração de um novo widget possa ser feita.
- A função colocada em **builder** não deve depender de atributos da classe ou de variáveis externas para evitar problemas.
- Também é importante lembrar que essa função poderá ser invocada várias vezes, não devendo executar nada além da criação do novo widget.

BlocBuilder

```
BlocBuilder<RedBloc, RedState>(
  builder: (context, state) {
    int r = state.amount;
    return AnimatedContainer(
      duration: const Duration(seconds: 1),
      height: 100,
      width: double.infinity,
      color: Color.fromRGBO(r, 0, 0, 1));
});
```

Builder e BuildContext.watch

Builder e BuildContext.watch

- A class **Builder** é usada quando você pretende criar um widget usando informações que já estão na widget tree.
- Note que a classe **Builder** possui um uso genérico, você deve usá-la sempre que precisar de um **BuildContext** para criar um novo widget, uma situação que ocorre com frequência.
- Dado um objeto context da classe **BuildContext**, a maneira como você buscará informações na árvore é usando o método estático **of**. Esse método é implementado pelas classes que permitem ser buscadas na widget tree.

Builder e BuildContext.watch

- Por exemplo, a chamada **ScaffoldMessenger.of**(context) indica que queremos buscar na widget tree um objeto da classe **ScaffoldMessenger** a partir do ponto indicado por context até a raiz da árvore.
- Caso você queira fazer essa chamada no momento da criação de um widget, possui duas opções.
  - Criar uma nova classe que herda de **StatefulWidget** ou **StatelessWidget** para poder usar o método **build** que possui um **BuildContext**. Esta opção deve ser usada em caso de widgets mais complexos.
  - Usar a classe **Builder** que proporciona um **BuildContext** para você. Esta opção deve ser usada quando você não precisa realmente de uma classe, apenas de um **BuildContext** para ser eventualmente usado para buscas na widget tree.

Builder e BuildContext.watch

Um **Builder** para criar um botão que abre uma **SnackBar**.

```
Scaffold (
  body : Builder(
    builder : (BuildContext context) {
      return ElevatedButton(
        onPressed : () {
          ScaffoldMessenger.of(context).showSnackBar(
            SnackBar(context: Text('Great')),
          );
      });
  })
)
```

Builder e BuildContext.watch

- Caso você tenha um widget que pode reagir a mudanças de estados ocasionadas por mais de um BLoC na widget tree, então o uso do **BlocBuilder** não é recomendável.
- Nesses casos, você pode usar o método **watch** de um objeto context da classe **BuildContext**. Esse objeto da classe **BuildContext** deve ter sido obtido dentro de um **Builder**.
- O método **watch** precisa da classe do BLoC como parâmetro genérico e permite obter o estado emitido pelo BLoC que disparou o gatilho.
- Dessa forma, devido à semelhança entre **watch** e **BlocBuilder**, é possível dizer que podem ser intercambiáveis. Entretanto, recomenda-se usar o **watch** apenas quando for reagir a mais de um BLoC.

Builder e BuildContext.watch

```
/*
  Os códigos a seguir são equivalmentes
*/
Widget build(BuildContext context) {
  final state = context.watch<MyBloc>().state;
}

Widget build(BuildContext context) {
  return BlocBuilder<MyBloc, MyState>(
    builder: (context, state) {...}
  );
}
```

Builder e BuildContext.watch

```
Builder(
  builder: (context) {
    final redState = context.watch<RedBloc>().state;
    final blueState = context.watch<BlueBloc>().state;
    final greenState = context.watch<GreenBloc>().state;
    return Container(
      height: 100,
      width: double.infinity,
      color: Color.fromRGBO(
        redState.amount, greenState.amount, blueState.amount, 1));
  },
),
```

Lançando Eventos para o BLoC

Lançando Eventos

- Até agora, sabemos como adicionar um BLoC na widget tree com **BlocProvider** e reagir a mudanças de estados com **BlocBuilder**.
- O lançamento de eventos é o nosso próximo passo, sendo isso feito com o método **add** do BLoC.
- De posso de uma instância do BLoC, é possível fazer o lançamento do evento de qualquer lugar do código.
- Como adicionamos o BLoC na widget tree, precisamos buscar o BLoC nesse mesmo lugar usando novamente a classe **BlocProvider**.

Lançando Eventos

Note que usamos a classe **BlocProvider** apenas para fazer uma busca na árvore com o método estático **of**. Feito isso, podemos disparar eventos com **add**.

```
RedBloc redBloc = BlocProvider.of<RedBloc>(context);
return ElevatedButton(
  child: Text("${redEvent.runtimeType}"),
  onPressed: () {
    redBloc.add(redEvent);
  },
);
```

Outros Widgets Importantes

Outros Widgets Importantes

- **BlocListener** é um widget que reage a mudanças de estados da mesma forma que um **BlocBuilder**, mas possui diferenças importantes:
  - É invocado apenas uma vez por mudança de estados.
  - Não retorna um widget para ser colocado na widget tree, mas permite mostrar um **SnackBar**, um **Dialog**, e assim por diante.
  - Não é invocado no estado inicial.

```
BlocListener<BlocA, BlocAState>(
  listener: (context, state) {
    // Código baseado no estado do BLoC
  },
  child: Container(),
)
```

Outros Widgets Importantes

- **BlocConsumer** permite que você adicione um **builder** e um **listener** ao mesmo tempo.
- Nesse caso, considere que o **BlocConsumer** é uma mistura de **BlocListener** e **BlocBuilder**.
- **BlocConsumer** só deve ser utilizado quando for necessário reconstruir a GUI e executar outras reações a alterações de estado.

Outros Widgets Importantes

```
BlocConsumer<BlocA, BlocAState>(
  listenWhen: (previous, current) {
    // Retorna true/false para determinar se
    // o listener precisa ser invocado
  },
  listener: (context, state) {
    // Execute ações como BlocListener
  },
  buildWhen: (previous, current) {
    // Retorna true/false para determinar se
    // será preciso recriar o widget.
  },
  builder: (context, state) {
    // Recria o widget baseado no estado
  } )
```

Outros Widgets Importantes

- **BlocSelector** é análogo ao **BlocBuilder**, mas permite filtrar as atualizações selecionando um novo valor com base no estado actual do bloco.
- Renderizações desnecessárias são evitadas se o valor selecionado não for alterado.
- O valor selecionado deve ser imutável para que o **BlocSelector** determine com precisão se a renderização deve ser feita novamente.