Desenvolver Interface Gráfica Para Dispositivos Móveis

Aula 08

Professor: Henrique Augusto Maltauro

Fecomércio PR
Sesc Senac IFPD
Senac

Flutter/Dart

- **main()**

O Dart, assim como qualquer linguagem de programação, possui um ponto de entrada principal.

Esse ponto de entrada normalmente é uma função, a qual é a primeira coisa que é executada no nosso programa.

No caso do Dart, essa função é o main().

Flutter/Dart

- **main()**

No Flutter, cada elemento de tela é tratado como um widget (componente).

Então, o objetivo da função main() é construir e apresentar esses widgets para o usuário.

Flutter/Dart

- main()

```
void main()
{
    // Bloco de Código
}
```

Flutter/Dart

- **runApp()**

No Flutter, o método runApp() é usado para inicializar e executar o aplicativo.

Ele recebe um widget como parâmetro, o qual normalmente é a raiz da árvore de widgets do aplicativo.

Este widget é então passado para o FlutterEngine para ser renderizado na tela.

Flutter/Dart

- **runApp()**

O método runApp() normalmente é chamado na função main() de um aplicativo Flutter.

Depois que o método runApp() é chamado, a estrutura configura a árvore de widgets e inicia o processo de renderização dos widgets na tela.

Flutter/Dart

- **runApp()**

```
void main()
{
    runApp(widget);
}
```

Flutter/Dart

- **MaterialApp()**

O MaterialApp() é uma classe/widget predefinida no Flutter, que irá gerar um aplicativo seguindo as diretrizes do Material Design.

Ele fornece uma espécie de embrulho, no qual todos os outros widgets serão envoltos.

Normalmente, ele é utilizado dentro do método runApp().

Flutter/Dart

- **MaterialApp()**

Ele possui mais de 30 parâmetros, que permite configurar a aplicação de acordo com as necessidades, mas num primeiro momento, vamos nos preocupar apenas com 3 deles.

➔ **title**

➔ **home**

➔ **debugShowCheckedModeBanner**

Flutter/Dart

- **MaterialApp(title)**

O parâmetro title recebe uma string, que vai definir o título do aplicativo, o qual aparece no gerenciador de tarefas do Android.

Flutter/Dart

- **MaterialApp(home)**

O parâmetro home recebe um widget, que vai definir o componente padrão que será apresentado quanto o aplicativo for executado.

Flutter/Dart

- **MaterialApp(debugShowCheckedModeBanner)**

O parâmetro debugShowCheckedModeBanner recebe um booleano, que vai definir se aquela faixinha no canto aparece ou não.

Flutter/Dart

- **MaterialApp()**

```
void main()
{
    runApp(MaterialApp(
        title: string,
        home: widget,
        debugShowCheckedModeBanner: boolean
    )); // MaterialApp
}
```

Flutter/Dart

- **Estado/State**

No geral: o estado são dados e informações utilizadas pela aplicação.

Estado da Aplicação: diz respeito a dados globais, como usuário autenticado ou produtos carregados.

Estado do Widget: diz respeito a dados internos do widget, como variáveis, valores de entrada, animações internadas do widget.

Flutter/Dart

- **E porque o state é importante?**

No Flutter, nós temos dois tipos de componentes:

➔ **StatelessWidget**

➔ **StatefulWidget**

Flutter/Dart

- **StatelessWidget**

O StatelessWidget é um componente com state constante, ou seja, uma vez renderizado, ele nunca muda.

Esses widget podem receber valores de parâmetros, e terem a sua aparência alterada de acordo com esses parâmetros.

Flutter/Dart

- **StatelessWidget**

Mas, qualquer variável interna desses widgets não irão fazer o mesmo ser re-renderizado.

Para fazer esses widgets serem re-renderizados, é necessário que do lado de fora dele, sejam alterados os valores dos seus parâmetros.

Flutter/Dart

- **StatelessWidget**

Para criar um StatelessWidget, é necessário criar uma classe que estende a classe abstrata StatelessWidget.

Flutter/Dart

- **StatelessWidget**

```
class SLWidget extends StatelessWidget
{
    // Bloco de Código
}
```

Flutter/Dart

- **StatelessWidget**

Depois é preciso definir o método construtor do componente, lembrando de passar o parâmetro Key, que é uma chave de identificação para os componentes.

Flutter/Dart

- **StatelessWidget**

```
class SLWidget extends StatelessWidget
{
    const SLWidget({ Key? key }); : super(key: key);
}
```

Flutter/Dart

- **StatelessWidget**

Por último é preciso reescrever o método build, que irá retornar a interface que representa aquele componente que está sendo criado.

Esse método recebe um parâmetro do tipo BuildContext, que serve para gerenciar a localização desse widget dentro da árvore de widgets.

Flutter/Dart

- **StatelessWidget**

```
class SLWidget extends StatelessWidget
{
    const SLWidget({ Key? key }) : super(key: key);

    @override
    Widget build(BuildContext context) {
        return widget;
    }
}
```

Flutter/Dart

- **StatelessWidget**

Muito importante lembrar que, qualquer parâmetro que seja passado no construtor de um StatelessWidget, é recomendado que o mesmo seja definido como final.

Como esse componente não vai ser afetado por alterações internas, é melhor que as suas variáveis sejam na verdade constantes.

Flutter/Dart

- **StatelessWidget**

```
class SLWidget extends StatelessWidget
{
     final String text;

     const SLWidget({
          this.text,
          Key? key
     }) : super(key: key);
}
```

Flutter/Dart

- **final vs const**

No Dart, nós possuímos duas palavras chaves para definir uma constante, o final e o const.

Com o final, eu consigo definir a constante e somente atribuir um valor para ela dentro do método do construtor.

O const, eu não consigo fazer isso, eu preciso definir um valor junto com a sua declaração.

Flutter/Dart

- **final vs const**

```
class ExampleClass
{
    final String finalText;
    const String constText = 'Text';

    ExampleClass({this.text});
}
```

Flutter/Dart

- **StatefulWidget**

O StatefulWidget é um componente com state flexível, ou seja, ele pode ser re-renderizado de acordo com os dados e informações contidos dentro dele.

Esses widget também podem receber valores de parâmetros, e terem a sua aparência alterada de acordo com esses parâmetros.

Flutter/Dart

- **StatefulWidget**

Enquanto o StatelessWidget precisa ser re-renderizado por inteiro, o StatefulWidget é re-renderizado apenas onde é necessário.

Se as suas variáveis ou parâmetros tiverem seus valores alterados, ele irá passar por um processo de re-renderização, apenas nas partes da tela que são pertinentes aquelas informações.

Flutter/Dart

- **StatefulWidget**

Para criar um StatefulWidget, é um pouquinho mais trabalhoso.

Primeiro é necessário criar uma classe que estende a classe abstrata StatefulWidget.

Flutter/Dart

- **StatefulWidget**

```
class SFWidget extends StatefulWidget
{
    // Bloco de Código
}
```

Flutter/Dart

- **StatefulWidget**

Depois é preciso definir o método construtor do componente, lembrando de passar o parâmetro Key, que é uma chave de identificação para os componentes.

Flutter/Dart

- **StatefulWidget**

```
class SFWidget extends StatefulWidget
{
    const SFWidget({ Key? key }); : super(key: key);
}
```

Flutter/Dart

- **StatefulWidget**

Até aqui, está igual ao processo do StatelessWidget, mas agora começa a mudar.

Agora é preciso criar uma outra classe, que vai ficar responsável por gerenciar o state do componente.

Essa classe estende a classe abstrata State, recebendo como parâmetro de tipo, o StatefulWidget que será gerenciado.

Flutter/Dart

- **StatefulWidget**

```
class SFWidget extends StatefulWidget
{
    const SFWidget({ Key? key }); : super(key: key);
}

class _SFWidgetState extends State<SFWidget>
{
    // Bloco de Código
}
```

Flutter/Dart

- **StatefulWidget**

Depois, dentro da classe principal, será reescrito o método createState(), para que assim seja definido quem será responsável por gerenciar o state do componente.

Flutter/Dart

- **StatefulWidget**

```
class SFWidget extends StatefulWidget
{
    const SFWidget({ Key? key }); : super(key: key);

    @override
    State<SFWidget> createState() => _SFWidgetState();
}
```

Flutter/Dart

- **StatefulWidget**

E por último, é dentro da classe state que será feito a reescrita do método build, que irá retornar a interface que representa aquele componente que está sendo criado.

Novamente, esse método recebe um parâmetro do tipo BuildContext, que serve para gerenciar a localização desse widget dentro da árvore de widgets.

Flutter/Dart

- **StatefulWidget**

```
class _SFWidgetState extends State<SFWidget>
{
    @override
    Widget build(BuildContext context) {
        return widget;
    }
}
```

Flutter/Dart

- **StatefulWidget**

```
class SFWidget extends StatefulWidget
{
	const SFWidget({ Key? key }); : super(key: key);

	@override
	State<SFWidget> createState() => _SFWidgetState();
}

class _SFWidgetState extends State<SFWidget>
{
	@override
	Widget build(BuildContext context) {
		return widget;
	}
}
```

Flutter/Dart

- **StatefulWidget**

Para se alterar o state do componente, é necessário executar o método setState().

Esse método recebe uma função como parâmetro, que irá atualizar os valores pertinentes a alteração do state.

Flutter/Dart

- **StatefulWidget**

```
class _SFWidgetState extends State<SFWidget>
{
    int number = 0;

    void _add() {
        setState(() {
            number++;
        });
    }
}
```

Flutter/Dart

- **StatefulWidget**

Também é possível, reescrever o método initState(), que será executado uma vez, quando o componente for inserido na árvore de componentes.

Qualquer processo que precisa ser executado antes da tela ser renderizada, deve ser executado aqui dentro.

Flutter/Dart

- **StatefulWidget**

```
class _SFWidgetState extends State<SFWidget>
{
    @override
    void initState() {
        // Bloco de Código
    }
}
```