Flutter Avançado - Aula 3

Exercícios 8 a 10

Fernando Duwe

v 1.0

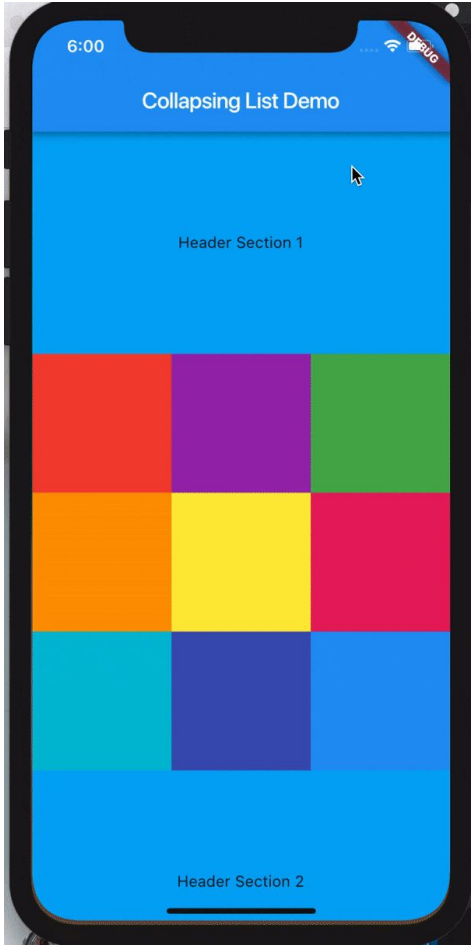
Dicas de hoje

- FlutLab.io Ver mais

Slivers

Ver mais

- Componente de *scroll* (*ScrollArea*)
- *Widgets* que nós já vimos, como *ListView* e *GridView* utilizam *Slivers*
- Com *slivers*, temos maior controle sobre o comportamento de *scroll*
- Além de maior controle,

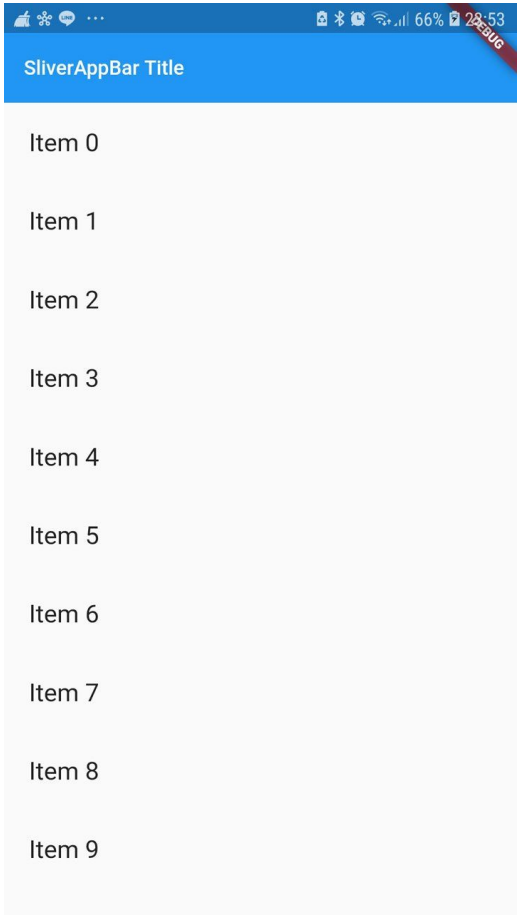


SliverList

Ver mais

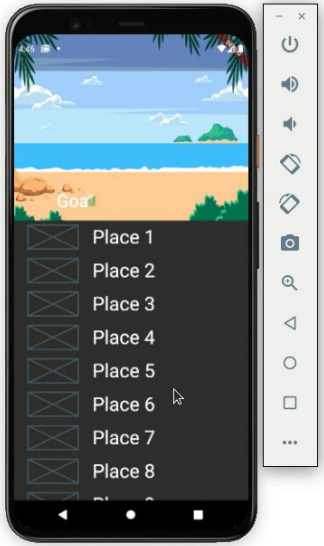
- Uma lista, utilizando *Slivers*
- Para cada lista, deve ser utilizado um *delegate*
- Deve ser sempre utilizado em conjunto a uma *CustomScrollView*

```
SliverList(
  delegate: SliverChildBuilderDelegate((context, index) {
    return Container(
      height: 50,
      child: Text("${index}"),
    );
  }, childCount: 20),
)
```



- Semelhante a *AppBar*, se expande e colapsa conforme é efetuado o *scroll* em tela
- Deve ser sempre utilizado em conjunto a uma *CustomScrollView*

```
SliverAppBar(
  pinned: true,
  snap: true,
  floating: true,
  expandedHeight: 160,
  flexibleSpace: FlexibleSpaceBar(
    title: Text("Meu teste"),
    background: FlutterLogo(),
  ),
),
```

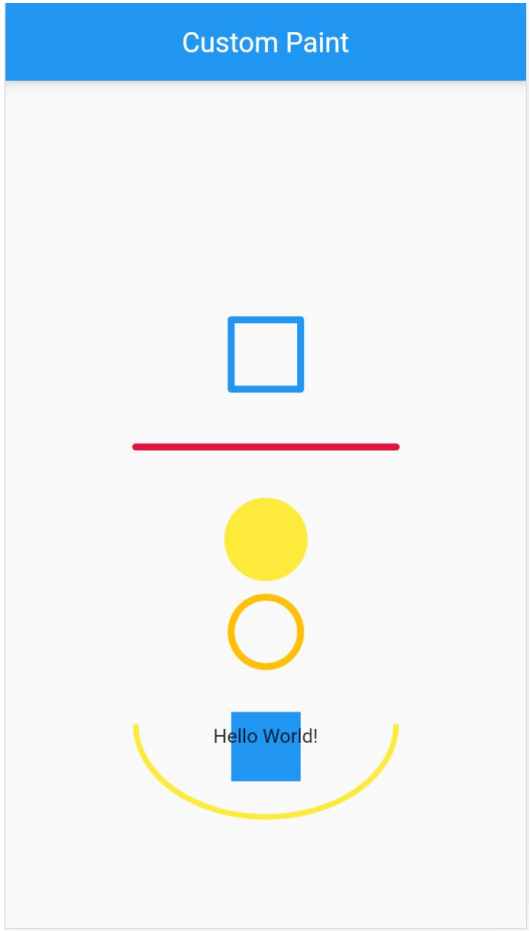


Ver mais

- **pinned:** Determina se a barra ficará sempre visível em tela
- **snap:** Adiciona suavidade ao esconder/exibir a AppBar
- **floating:** Determina que a barra ficará visível assim que o usuário começar a efetuar o *scroll* para cima

Ver mais

- Componente que permite a criação de desenhos
- O *widget* disponibilizará um *canvas* (uma tela) para desenho
- Deveremos implementar um *delegate*, chamado de *painter*



paint

Esse método é chamado sempre que o componente deverá ser repintado. Receberemos por parâmetro o tamanho disponível e a tela em que iremos pintar.

```
void paint(Canvas canvas, Size size) {
    …
}
```

shouldRepaint

Esse método determinará se nossa imagem deve ser pintada novamente ou não. Recebemos por parâmetro o nosso antigo *delegate*, e retornamos um valor *boolean*, determinando se a imagem deve ser refeita ou não.

```
bool shouldRepaint(SunflowerPainter oldDelegate) {

}
```

Desenhando um círculo

```
void _drawEye(Canvas prCanvas, double prX, double prY, double prSize) {
    Paint vrPaint = Paint();
    vrPaint.style = PaintingStyle.fill;
    vrPaint.color = Colors.black;

    prCanvas.drawCircle(Offset(prX, prY), prSize, vrPaint);
}
```

Desenhando uma linha

```
prCanvas.drawLine(Offset(10, 10), Offset(100, 100), vrPaint);
```

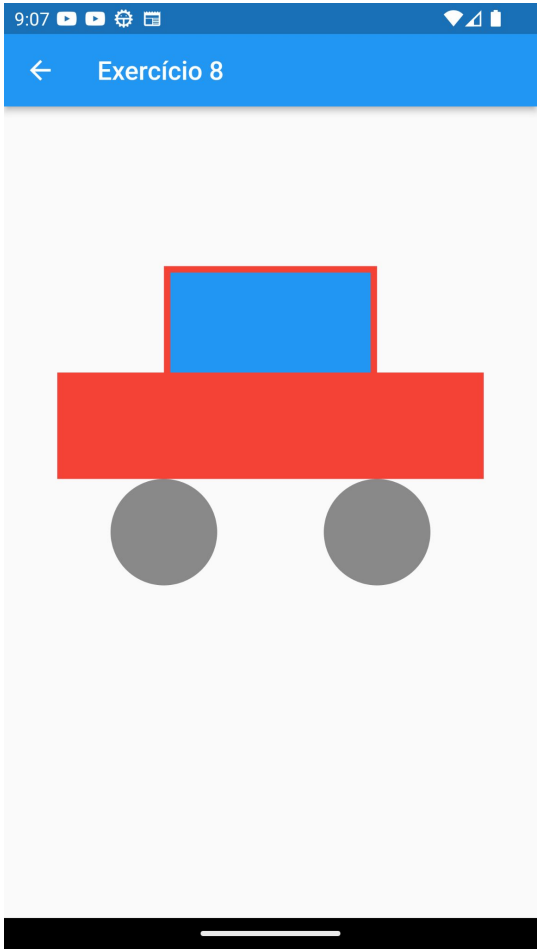
Desenhando um retângulo

```
prCanvas.drawRect(Rect.fromLTRB(left, top, right, bottom), vrPaint);
```

Exercício 8

Ver mais

- Vamos desenhar um carrinho, usando um *CustomPaint*
- Crie um delegate para desenhar o carrinho



Ver mais

- **Animações Controladas:** A partir de controllers, conseguimos controlar a animação pausar, iniciar, inverter, etc

- **Animações Implícitas:** Alguns widgets do flutter já possuem animações implícitas, ou seja, já são automaticamente animados, sem que seja necessário ao desenvolvedor alterar grandes quantidades de código

Animações controladas

Ver mais

```
class _AnimationScreenState extends State<AnimationScreen>
      with SingleTickerProviderStateMixin {
      late AnimationController animationController;

      @override
      void initState() {
            super.initState();

            animationController = AnimationController(vsync: this, duration: Duration(seconds: 5));
      }

      @override
      void dispose() {
            animationController.dispose();

            super.dispose();
      }

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

initState

O *initState* é chamado assim que o *widget* é adicionado na árvore de *widgets* do *app*. Nesse método, estamos criando o controller de animação, adicionando um *vsync* e uma duração para a animação.

dispose

O *dispose* é chamado assim que o *widget* é retirado da árvore de *widgets* do app permanentemente. Nesse método, estamos destruindo o *controller* de animação.

vsync

Todo *AnimationController* precisa possuir um *vsync*. o *vsync* é utilizado para sincronizar a animação a taxa de atualização do nosso app. Se nosso app é renderizado a 60 *fps* (*frames* por segundo), o vsync será responsável por sincronizar essa renderização com o controller de animação.

SingleTickerProviderStateMixin

O *SingleTickerProveriderStateMixin* será disparado somente quando a árvore atual onde ele está localizado for renderizada.

Animações controladas - cont

Ver mais

```
late AnimationController animationController;
late Animation<double> animation;

@override
void initState() {
      super.initState();

      animationController = AnimationController(vsync: this, duration: Duration(seconds: 2));

      animationController.addListener(() {
            setState(() {

            });
      });

      animation = Tween<double>(begin: 0, end: 300).animate(animationController);

      animationController.forward();
}
```

Adicionando um *listener* no *controller* da animação, para que sempre que a animação avança, o *widget* seja renderizado.

```
animationController.addListener(() {
     setState(() {

     });
});
```

Ver mais

Criando um range de 0 a 300, que será incrementado conforme a animação avança. No exemplo dos slides, como a animação tem uma duração de 2 segundos, o range será dividido de 0 a 300, por 2 segundos.

```
animation = Tween<double>(begin: 0, end: 300).animate(animationController);
```

Executa a animação

```
animationController.forward();
```

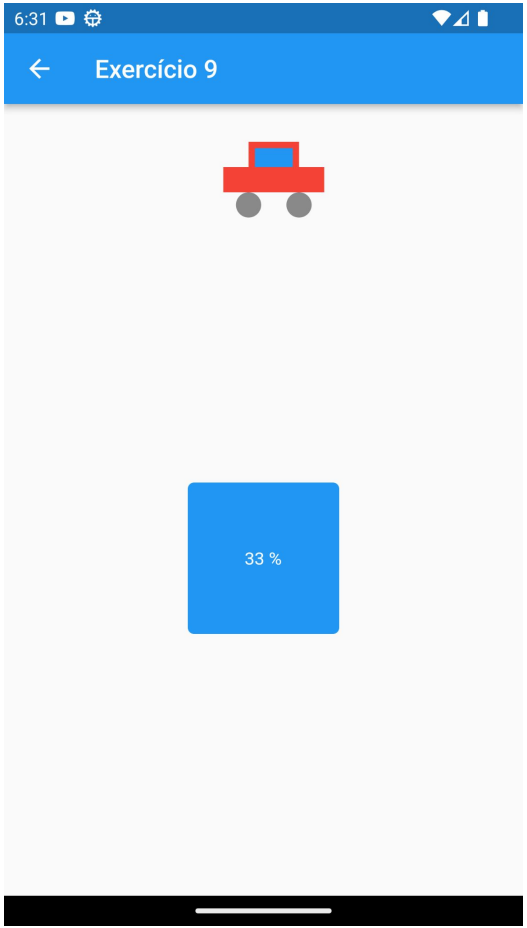
Controlando o status da animação

```
animationController.addStatusListener((status) {
  if (status == AnimationStatus.completed) {
    animationController.reverse();

    return;
  }

  if (status == AnimationStatus.dismissed) {
    animationController.forward();

    return;
  }
});
```

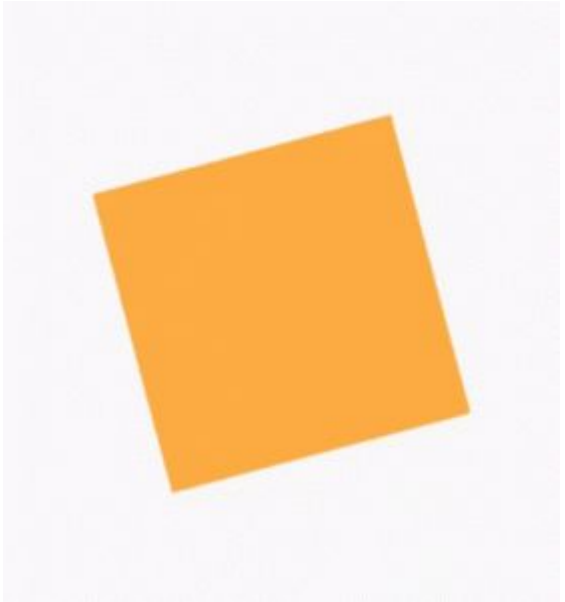
Exercício 9

Ver mais

- Vamos usar o nosso carrinho do exercício anterior
- Com uma animação, vamos fazer o carrinho andar em tela
- No exercício, utilizamos uma duração de 3 segundos
- Um intervalo de 0 a 500, mas a seu critério quais parâmetros utilizar



- *AnimatedWidget* é um *widget* que efetua um *rebuild* sempre que seu *listenable* muda
- É utilizado majoritariamente para animações
- Utilizando-o, não precisaremos mais do *setState*

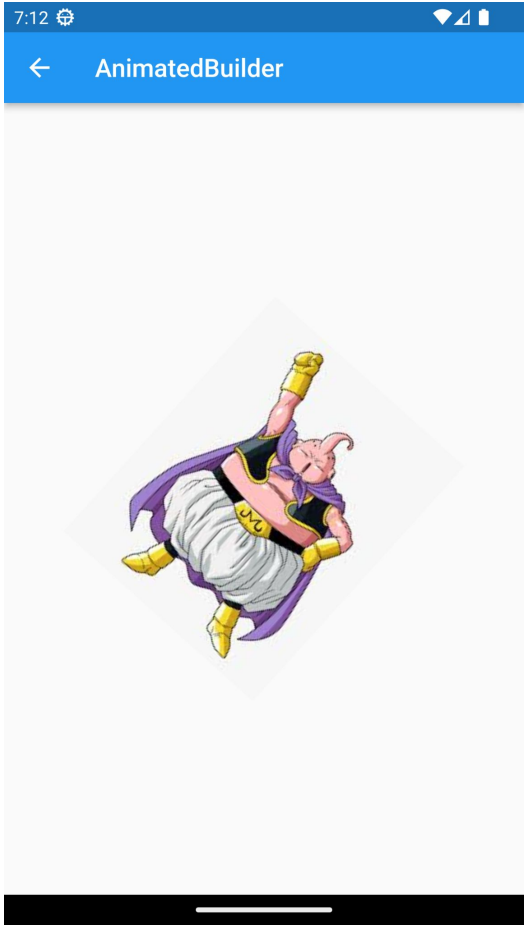


Abaixo um exemplo com o fonte do exercício 9

```
class AnimatedCar extends AnimatedWidget {
  AnimatedCar(Animation<double> prAnimation) : super(listenable:
prAnimation);

  @override
  Widget build(BuildContext context) {
    Animation<double> vrAnimation = listenable as Animation<double>;

    return CustomPaint(
      painter: CarRunPainterDelegate(vrAnimation.value),
    );
  }
}
```

Ver mais

- O *AnimatedBuilder* disponibiliza um construtor para construirmos nossas animações
- Podemos utilizá-lo para a criação de animações mais complexas
- Também pode ser utilizado para aumentar a performance de atualização de tela, sem a utilização de animações

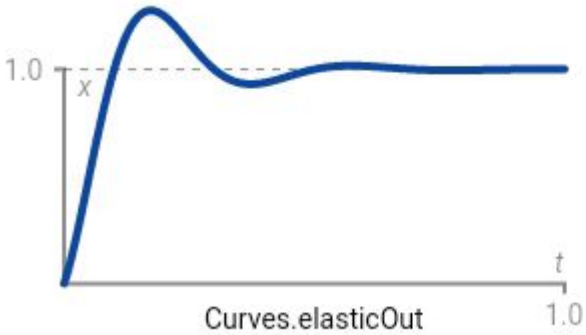


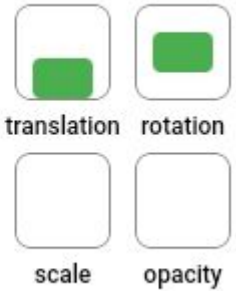
AnimatedBuilder - cont

Ver mais

```
void initState() {
	super.initState();

	this.controller = AnimationController(vsync: this, duration: Duration(seconds: 5));

	controller.repeat();
}

@override
Widget build(BuildContext context) {
	return Scaffold(
		appBar: AppBar(
			title: Text("AnimatedBuilder"),
		),
		body: Center(
			child: AnimatedBuilder(
				animation: this.controller,
				child: Image(
					image: NetworkImage(
"https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcRQjWTOkhwMsSK5GOZ3OjgAudb2BXbcPXQNXg&usqp=CAU")),
				builder: (context, child) {
					return Transform.rotate(
								angle: this.controller.value * 2.0 * math.pi,
								child: child,
					);
				},
			),
		),
	);
}
```

- As animações que criamos até agora foram lineares
- Com *CurvedAnimation*, podemos criar animações não lineares ou que não sejam constantes
- Ao criar o *widget*, precisamos definir um *parent*, que será o nosso *controller*
- E também uma curva





Ver mais

- Vamos criar uma tela de login
- Ao clicar no botão de login, o botão deve diminuir, até aparecer um *CircularProgressIndicator*

Exercício 10 - cont

Ver mais