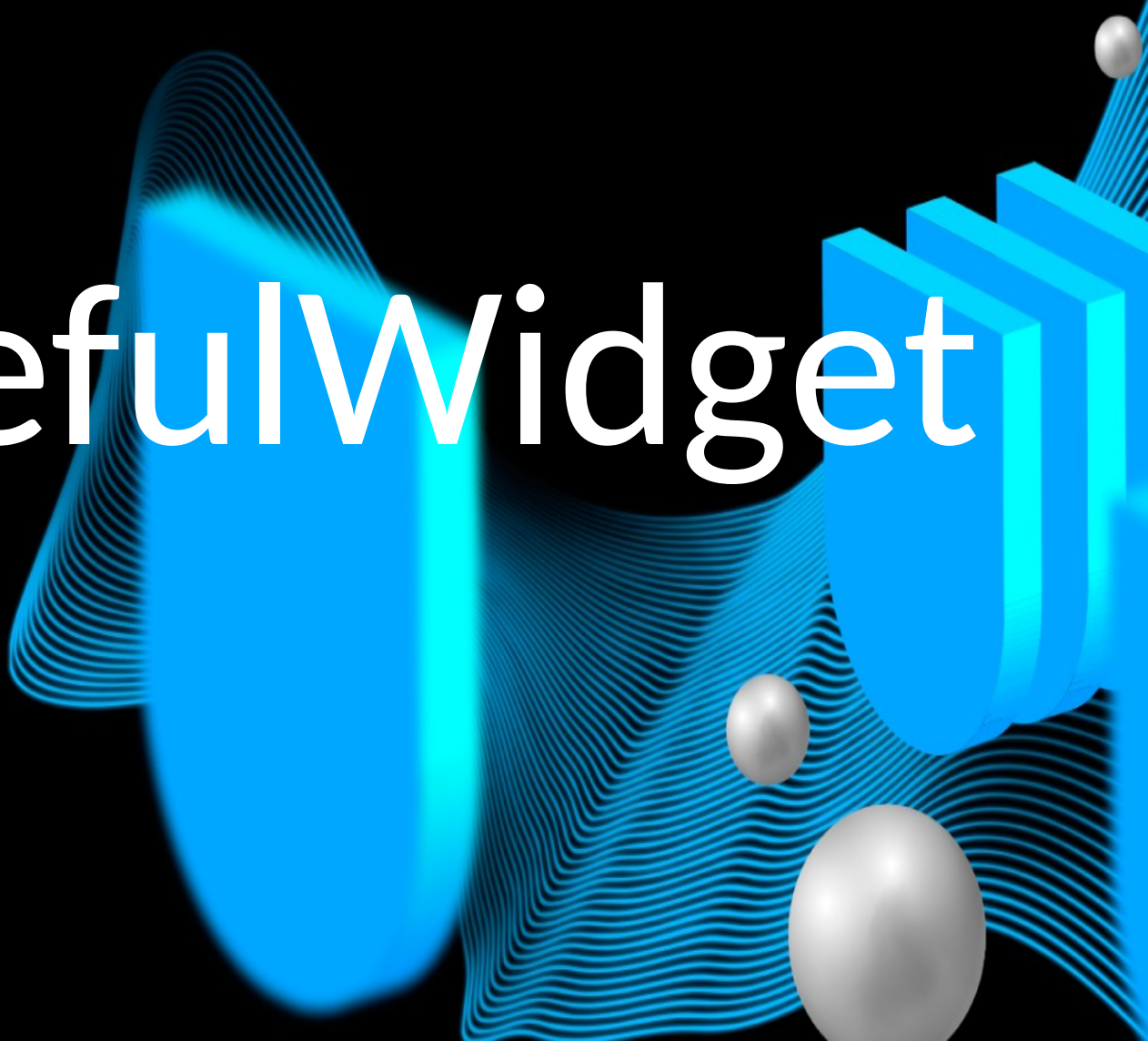

UNITAU
Universidade de Taubaté

StatefulWidget



StatefulWidget

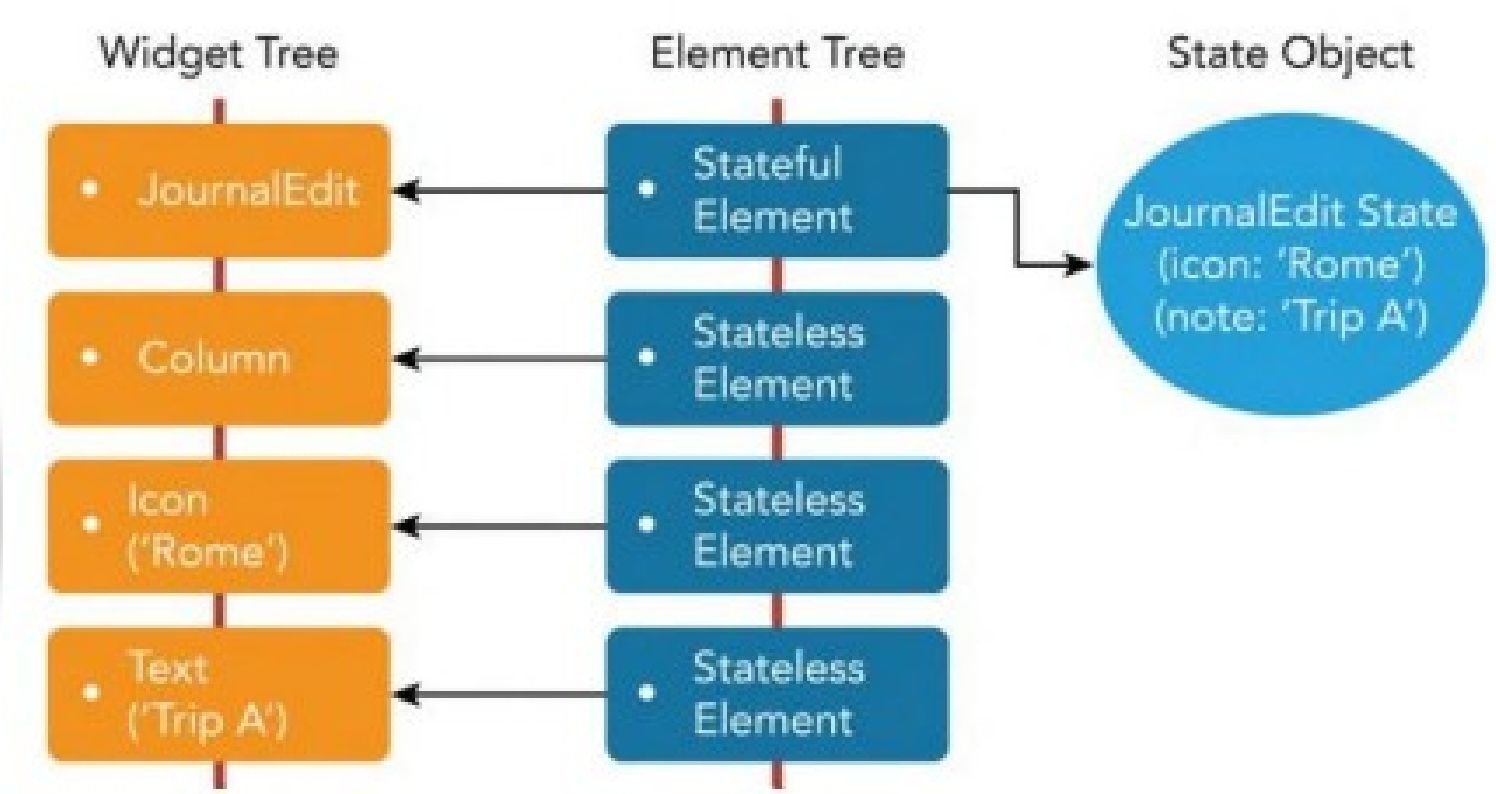
StatefulWidget são widgets que são alterados dinamicamente durante a execução da aplicação.

Ao criar um **StatefulWidget**, teremos dois objetos, o widget propriamente, que é imutável, e um objeto da classe **State** que armazena os dados alteráveis.

https://medium.com/@rkishan516/flutter-book-stateless-and-stateful-widgets-8db946f45df5

Element Tree

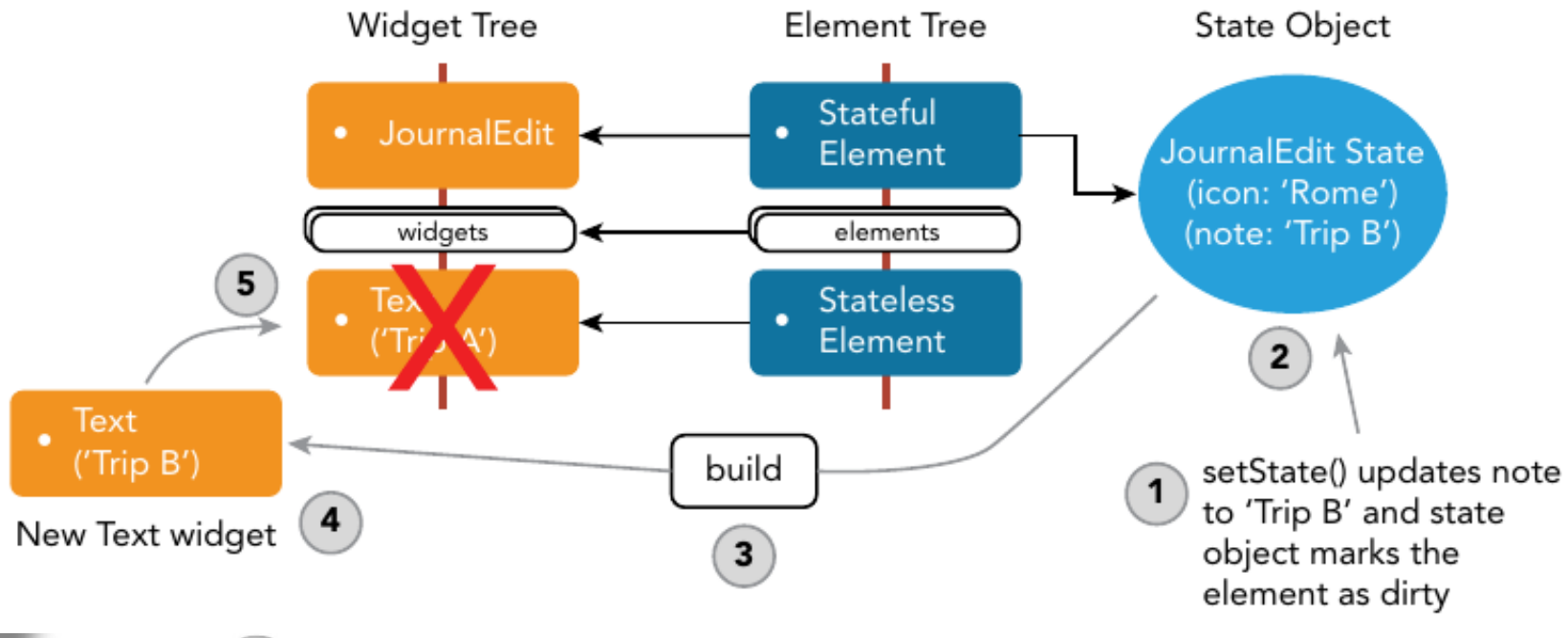
Os widgets que serão renderizados na tela são colocados em uma Element Tree. A árvore de elementos tem uma ligação para o widget na Widget Tree e, para os StatefulWidget, uma ligação para o objeto State do widget.



Fonte: NAPOLI, M. L., 2019

build

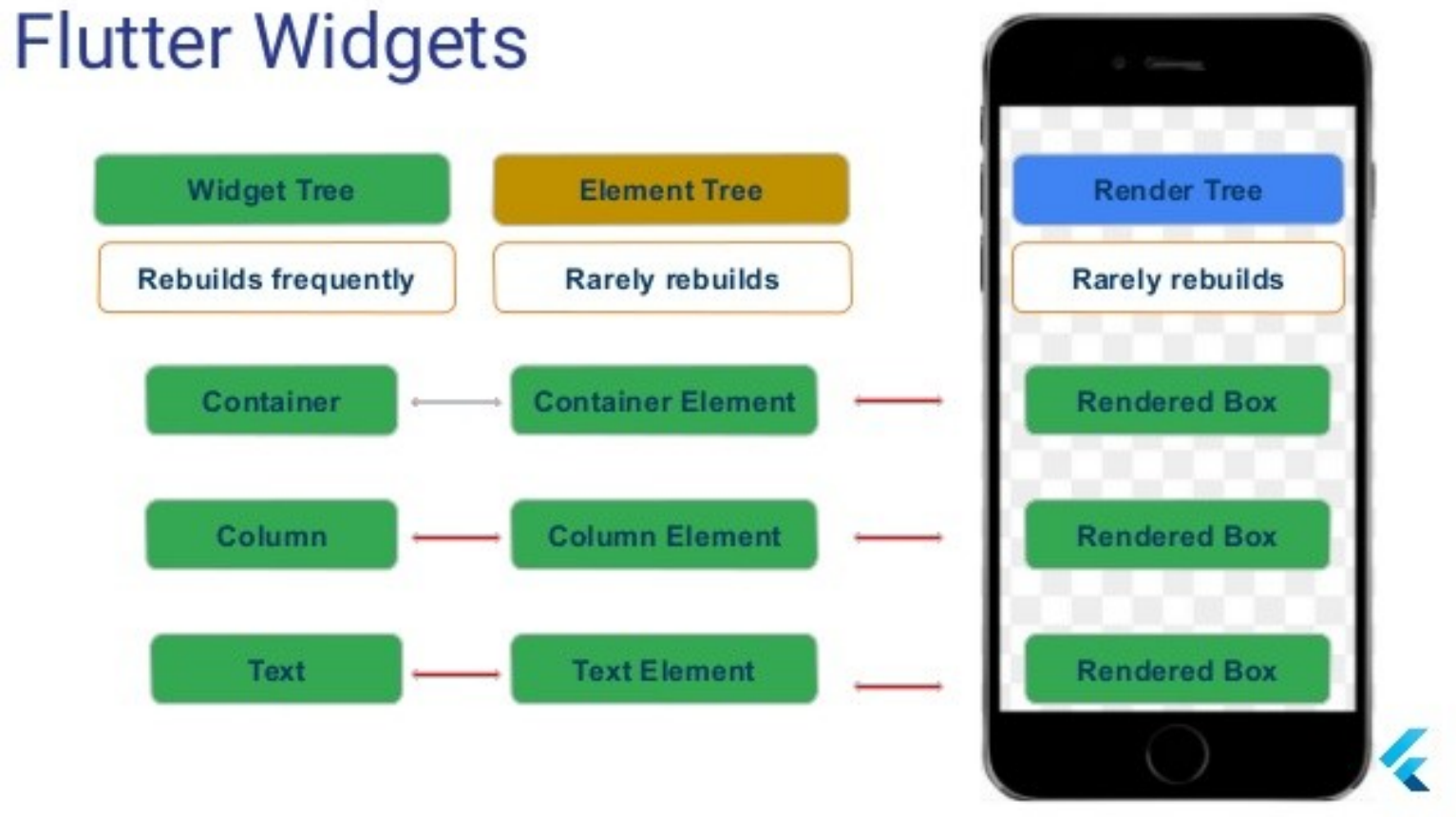
Quando o objeto State é alterado, o widget é removido e outro widget é criado refletindo o novo estado.



Fonte: NAPOLI, M. L., 2019

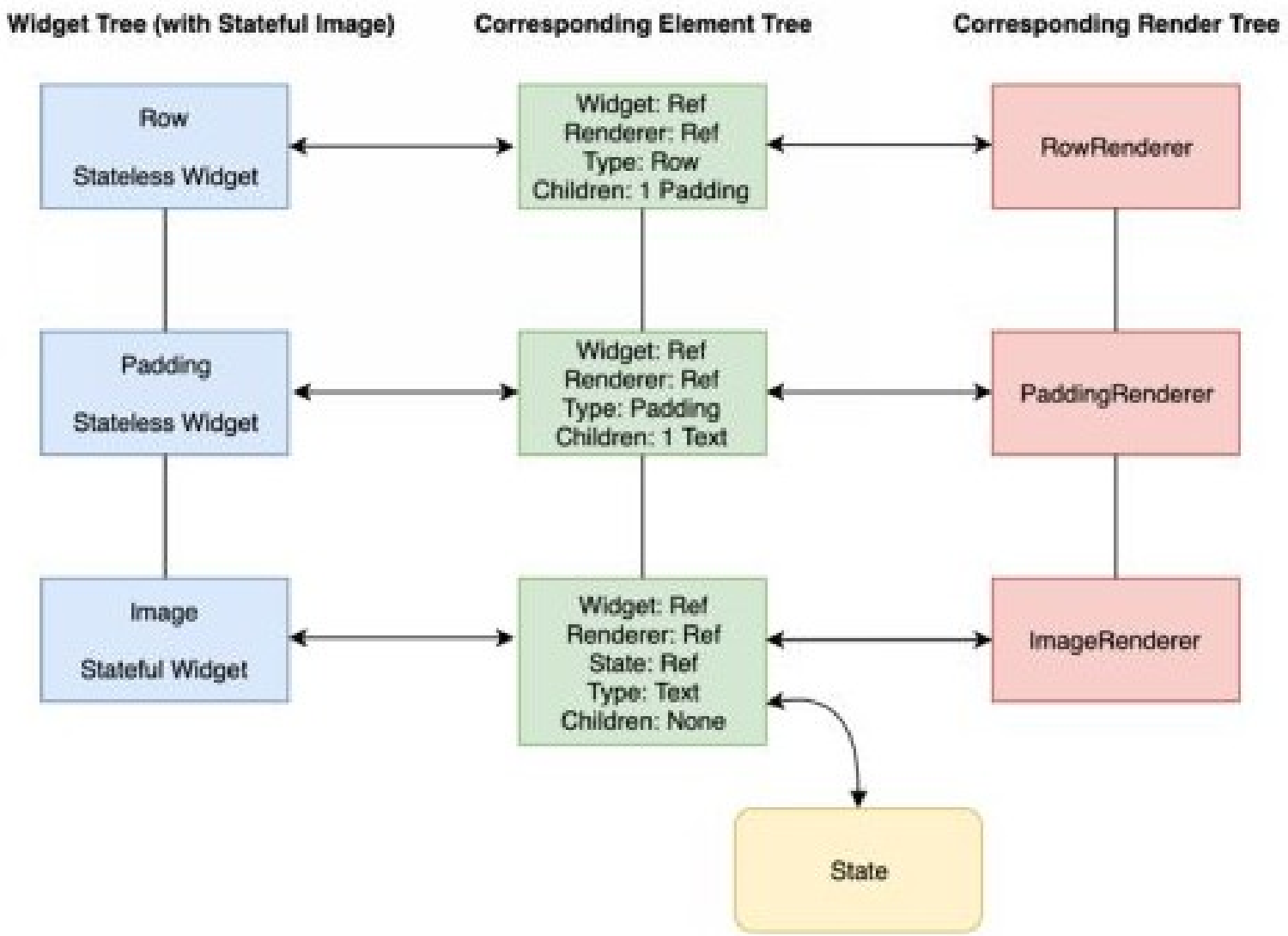
Render Tree

Os objetos da Element Tree também tem uma ligação para objetos na Render Tree.



https://www.slideshare.net/abueldahab/flutter-beyond-hello-world-talk

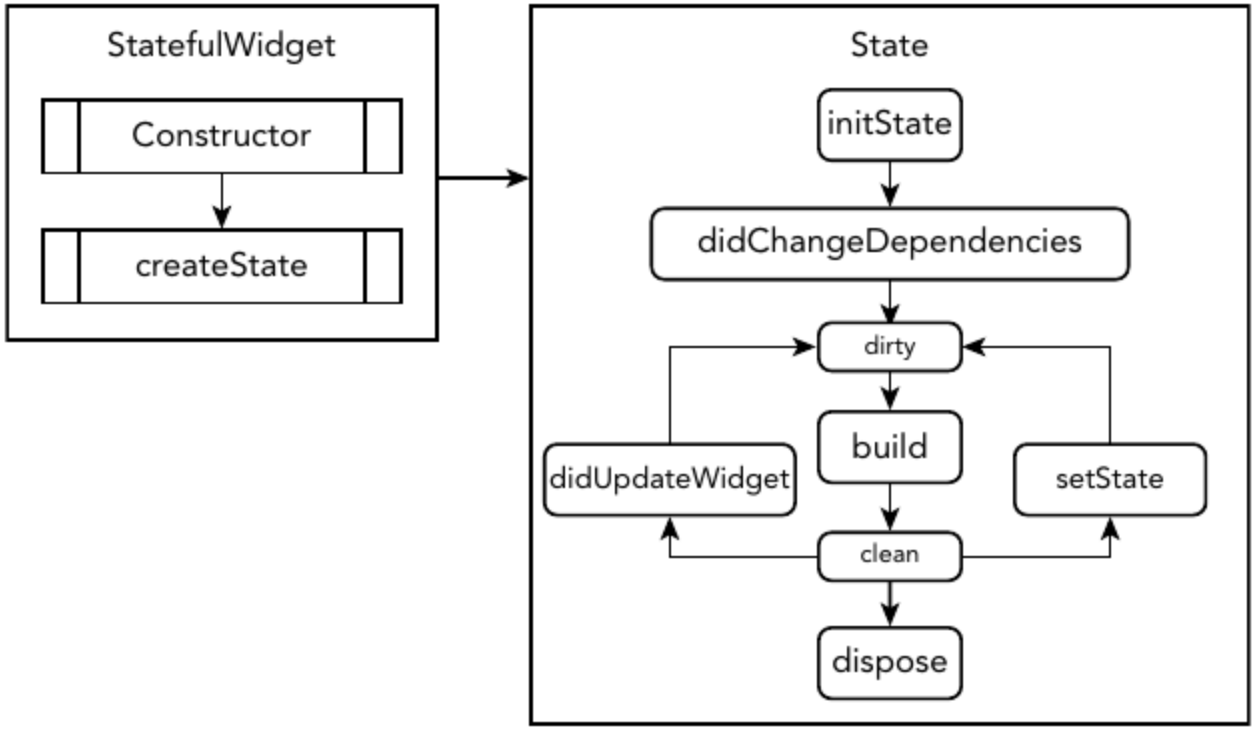
Flutter Trees



Fonte: CLOW, M., 2019

Ciclo de Vida

O ciclo de vida de um StatefulWidget inicia com a criação do widget e do objeto State correspondentes e a cada alteração de estado, o widget é recriado.



Fonte: NAPOLI, M. L., 2019

Ciclo de Vida

Crie uma nova aplicação Flutter com o nome stateful_widget e altere main.dart para:

```
import 'package:flutter/material.dart';
import 'package:widgets/pages/home.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Flutter Demo',
      theme: ThemeData(primarySwatch: Colors.blue,
        visualDensity:
VisualDensity.adaptivePlatformDensity,
      ),
      home: Home(),
    );
  }
}
```

Ciclo de Vida

Crie o subdiretório **lib/pages** e o arquivo **home.dart** nesse subdiretório, com o seguinte conteúdo:

```
import 'package:flutter/material.dart';

int _counter = 0;

class Home extends StatefulWidget {

  @override
  _HomeState createState() => _HomeState();
}

class _HomeState extends State<Home> {

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }
```

Ciclo de Vida

```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Ciclo de Vida"),),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ),
            Counter(),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),),
    );
  }
}
```

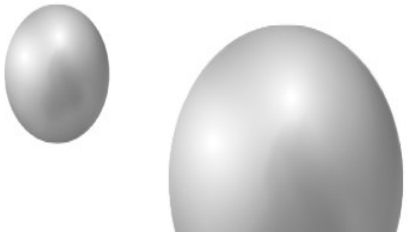
Ciclo de Vida

```
class Counter extends StatefulWidget {

  @override
  _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {

  @override
  Widget build(BuildContext context) {
    return Text(
      '$_counter',
      style: Theme.of(context).textTheme.headline4,
    );
  }
}
```

createState

O método createState do StatefulWidget é chamado imediatamente quando o framework cria o StatefulWidget. Ao criar uma classe derivada de StatefulWidget obrigatoriamente tem que implementar createState, normalmente precisa apenas desse método.

Quando createState cria o estado do widget, o buildContext é atribuído para o widget e a propriedade booleana this.mounted é setada para true. Não se deve chamar setState para o objeto que não está montado.



initState

Após a criação do widget, o primeiro método executado é **initState**. Esse método é executado uma única vez. Pode ser usado para inicializar as propriedades do estado do widget. Quando sobrescrever esse método, primeiro deve ser executado o método correspondente da superclasse.

Acrescente o método a seguir na classe **_HomeState**:

```
@override
void initState() {
  super.initState();
  print("Home initState");
}
```

E o método a seguir na classe **_CounterState**:

```
@override
void initState() {
  super.initState();
  print("Counter initState");
}
```

didChangeDependencies

O método **didChangeDependencies** é executado logo após **initState**. Também é executado quando uma dependência do objeto **State** é alterada. Quando sobrescrever esse método, primeiro deve ser executado o método correspondente da superclasse.

Acrescente o método a seguir na classe **_HomeState**:

```
@override
void didChangeDependencies() {
  super.didChangeDependencies();
  print("Home didChangeDependencies");
}
```

E o método a seguir na classe **_CounterState**:

```
@override
void didChangeDependencies() {
  super.didChangeDependencies();
  print("Counter didChangeDependencies");
}
```

build

O método **build** é executado sempre que é necessário recriar o widget. É obrigatório sobrescrever esse método que deve retornar um widget.

Acrescente a linha abaixo no início do método **build** da classe **_HomeState**:

```
print("Home build");
```

E a linha abaixo no início do método **build** da classe **_CounterState**:

```
print("Counter build");
```

didUpdateWidget

O método didUpdateWidget é executado quando a configuração do objeto é alterada. Quando sobrescrever esse método, primeiro deve ser executado o método correspondente da superclasse.

Acrescente o método a seguir na classe _HomeState:

```
@override
void didUpdateWidget(Home oldWidget) {
  super.didUpdateWidget(oldWidget);
  print("Home didUpdateWidget");
}
```

E o método a seguir na classe _CounterState:

```
@override
void didUpdateWidget(Counter oldWidget) {
  super.didUpdateWidget(oldWidget);
  print("Counter didUpdateWidget");
}
```

setState

O método setState notifica o framework que o estado do objeto foi alterado.

Acrescente o método a seguir na classe _HomeState:

```
@override
void setState(fn) {
  print("Home setState");
  super.setState(fn);
}
```

E o método a seguir na classe _CounterState:

```
@override
void setState(fn) {
  print("Counter setState");
  super.setState(fn);
}
```

deactivate

O método **deactivate** é executado quando o objeto **State** é removido da árvore mas pode ser reinserido novamente. Quando sobrescrever esse método, o método correspondente da superclasse deve ser executado no final do método.

Acrescente o método a seguir na classe **_HomeState**:

```
@override
void deactivate() {
  print("Home deactivate");
  super.deactivate();
}
```

E o método a seguir na classe **_CounterState**:

```
@override
void deactivate() {
  print("Counter deactivate");
  super.deactivate();
}
```

dispose

O método **dispose** é executado quando o objeto **State** é removido permanentemente da árvore e o widget não será mais reconstruído. Quando sobrescrever esse método, o método correspondente da superclasse deve ser executado no final do método.

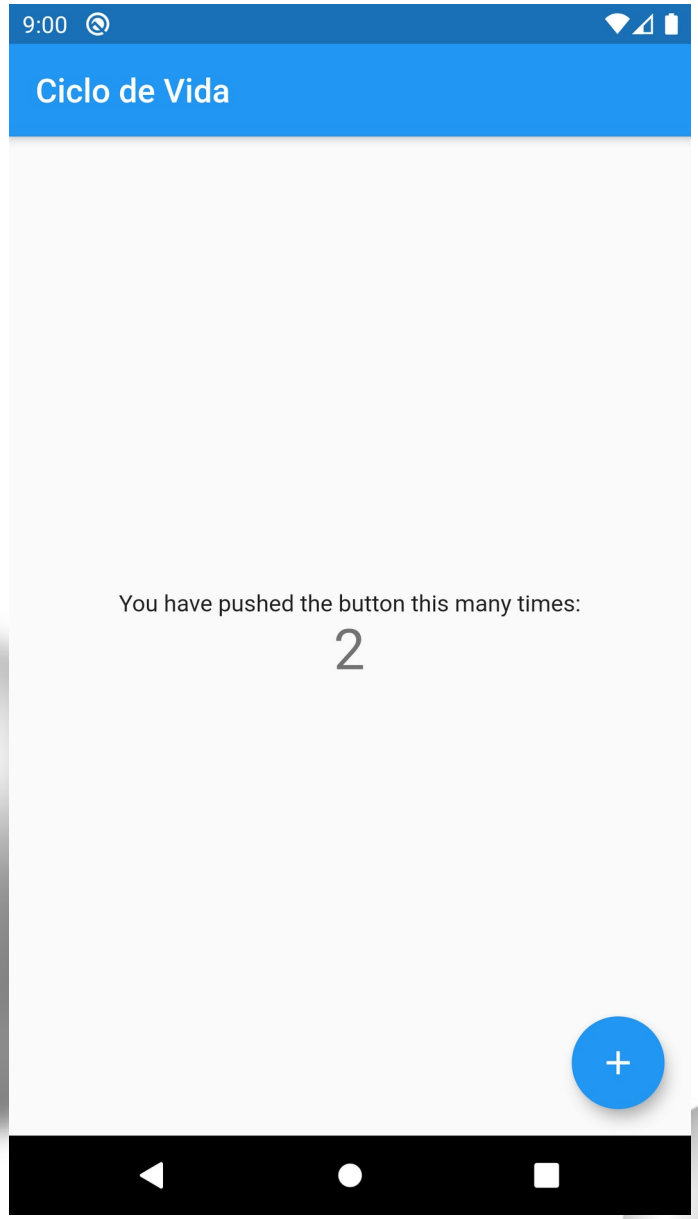
Acrescente o método a seguir na classe **_HomeState**:

```
@override
void dispose() {
  print("Home dispose");
  super.dispose();
}
```

E o método a seguir na classe **_CounterState**:

```
@override
void dispose() {
  print("Counter dispose");
  super.dispose();
}
```

Ciclo de Vida




```
An Observatory debugger and profiler on Android SDK bu
http://127.0.0.1:35155/OYqAyp5tjPc=/
I/flutter ( 8549): Home initState
I/flutter ( 8549): Home didChangeDependencies
I/flutter ( 8549): Home build
I/flutter ( 8549): Counter initState
I/flutter ( 8549): Counter didChangeDependencies
I/flutter ( 8549): Counter build
I/flutter ( 8549): Home setState
I/flutter ( 8549): Home build
I/flutter ( 8549): Counter didUpdateWidget
I/flutter ( 8549): Counter build
I/flutter ( 8549): Home setState
I/flutter ( 8549): Home build
I/flutter ( 8549): Counter didUpdateWidget
I/flutter ( 8549): Counter build
```

Keys

Os widgets podem ter uma chave (**Key**) usada para identificar o widget em diversas situações.

Um dos usos das chaves é preservar o estado do widget quando este for movido na Widget Tree.

Crie uma nova aplicação Flutter com o nome **keys_stateless** e altere **main.dart** para:

```
import 'package:flutter/material.dart';
import 'dart:math';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: PositionedTiles(),);
  }
}
```

Keys

Inclua a lista de quadros a exibir:

```
//
// Lista de quadros a exibir
//
List<Widget> tiles = [
  ColorTile(),
  ColorTile(),
];
```

Acrescente a classe para a tela principal da aplicação:

```
//
// Classe para a tela principal
//
class PositionedTiles extends StatefulWidget {
  @override
  State<StatefulWidget> createState() =>
PositionedTilesState();
}
```

Keys

Acrescente a classe para gerenciar o estado da tela principal da aplicação:

```
//
// Clase para gerenciar o estado da tela principal
//
class PositionedTilesState extends State<PositionedTiles> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Row(children: tiles),
      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.sentiment_very_satisfied),
        onPressed: swapTiles,
      ),
    );
  }
```

Keys

```
  // Funcao para trocar a ordem dos quadros
  swapTiles() {
    print("Trocando os quadros");
    setState(() {
      tiles.insert(1, tiles.removeAt(0));
    });
  }
}
```

Keys

Acrescente a classe para gerar uma cor aleatória:

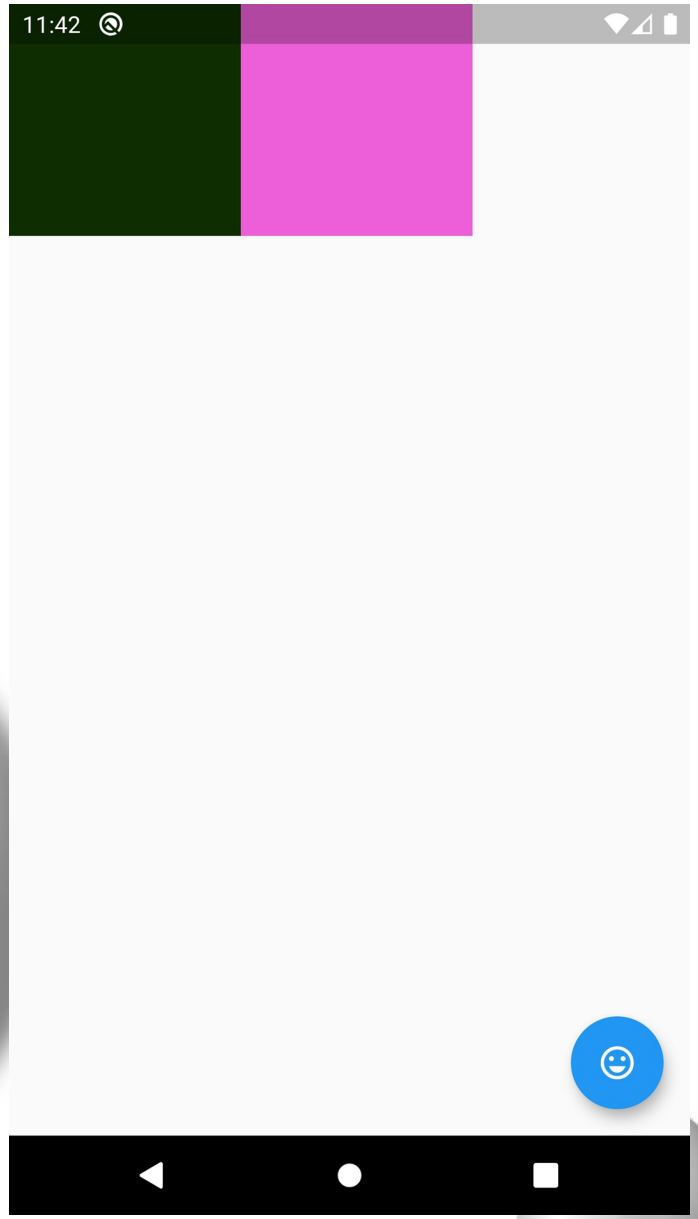
```
//
// Classe para gerar uma cor aleatoria
//
class UniqueColorGenerator {
  static Random random = Random();
  static Color getColor() {
    return Color.fromARGB(255, random.nextInt(255),
random.nextInt(255), random.nextInt(255));
  }
}
```

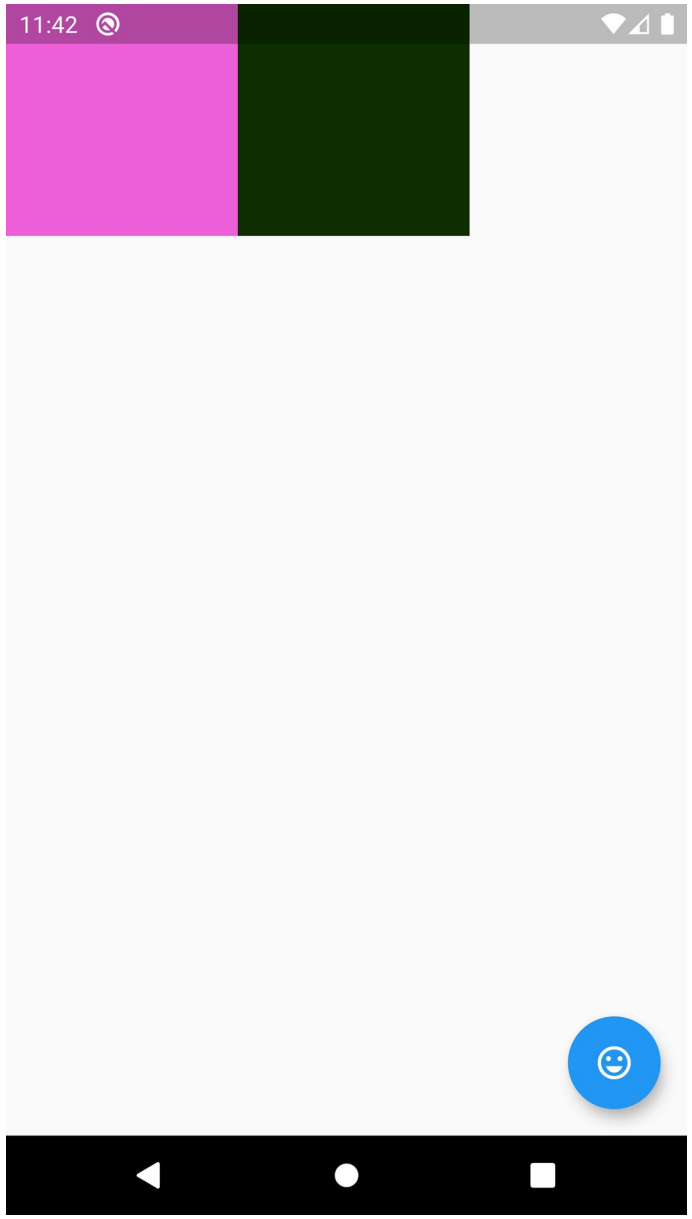
Keys

Acrescente a classe para os quadros a exibir, usando **StatelessWidget**:

```
//
// Classe para quadros a exibir
//
class ColorTile extends StatelessWidget {
  Color myColor = UniqueColorGenerator.getColor();
  @override
  Widget build(BuildContext context) {
    return Container(
      color: myColor,
      child: Padding(
        padding: EdgeInsets.all(70.0)
      )
    );
  }
}
```

Keys





Keys

Crie uma nova aplicação Flutter com o nome **keys_statefull** e copie o arquivo **main.dart** da aplicação anterior.

Substitua a classe para os quadros a exibir, utilizando **StatefulWidget**:

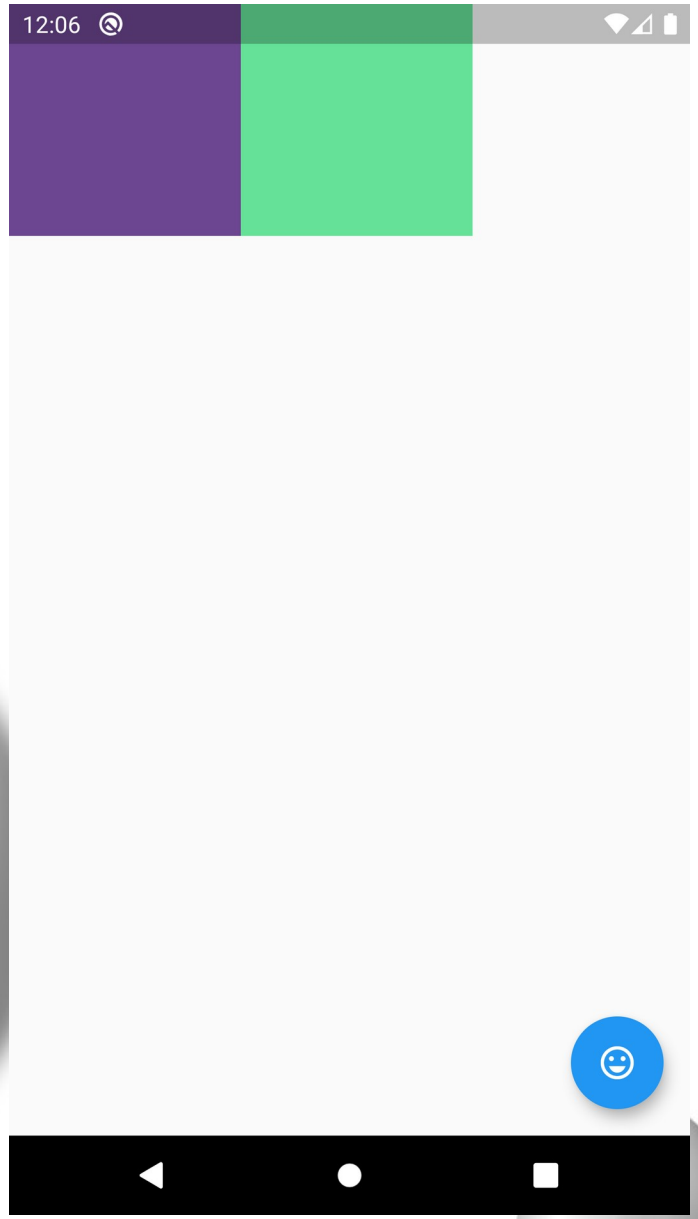
```
//
// Classe para quadros a exibir
//
class ColorTile extends StatefulWidget {
  @override
  ColorTileState createState() => ColorTileState();
}
```

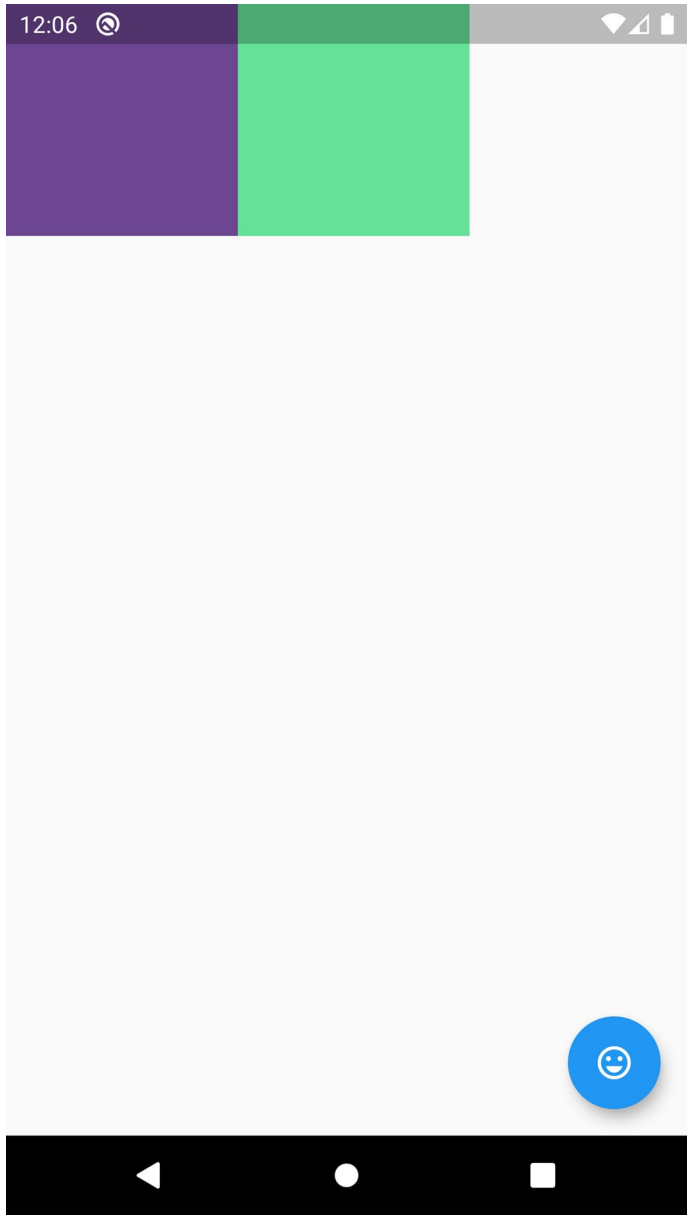
Keys

Inclua a classe para gerenciar o estado dos quadros a exibir:

```
//
// Classe para gerencia o estado do quadro a exibir
//
class ColorTileState extends State<ColorTile> {
  Color myColor;

  @override
  void initState() {
    super.initState();
    myColor = UniqueColorGenerator.getColor();
  }

  @override
  Widget build(BuildContext context) {
    return Container(
      color: myColor,
      child: Padding(padding: EdgeInsets.all(70.0),)
    );
  }
}
```

Keys

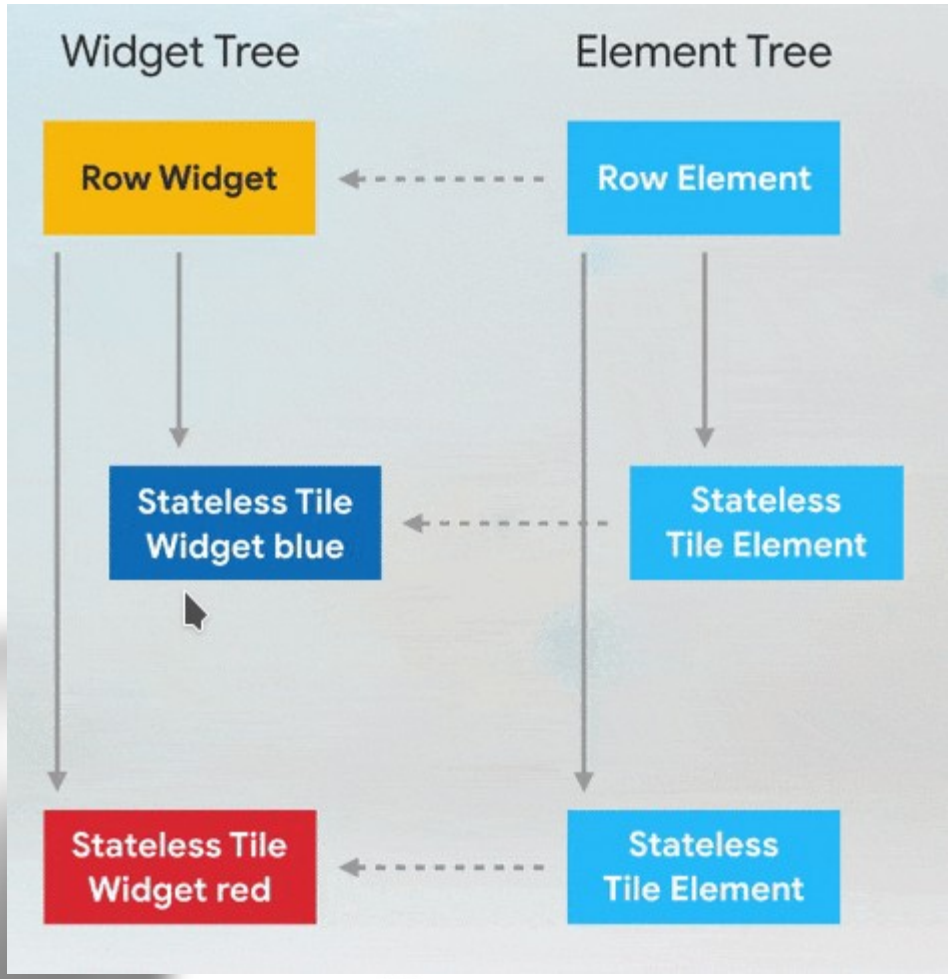


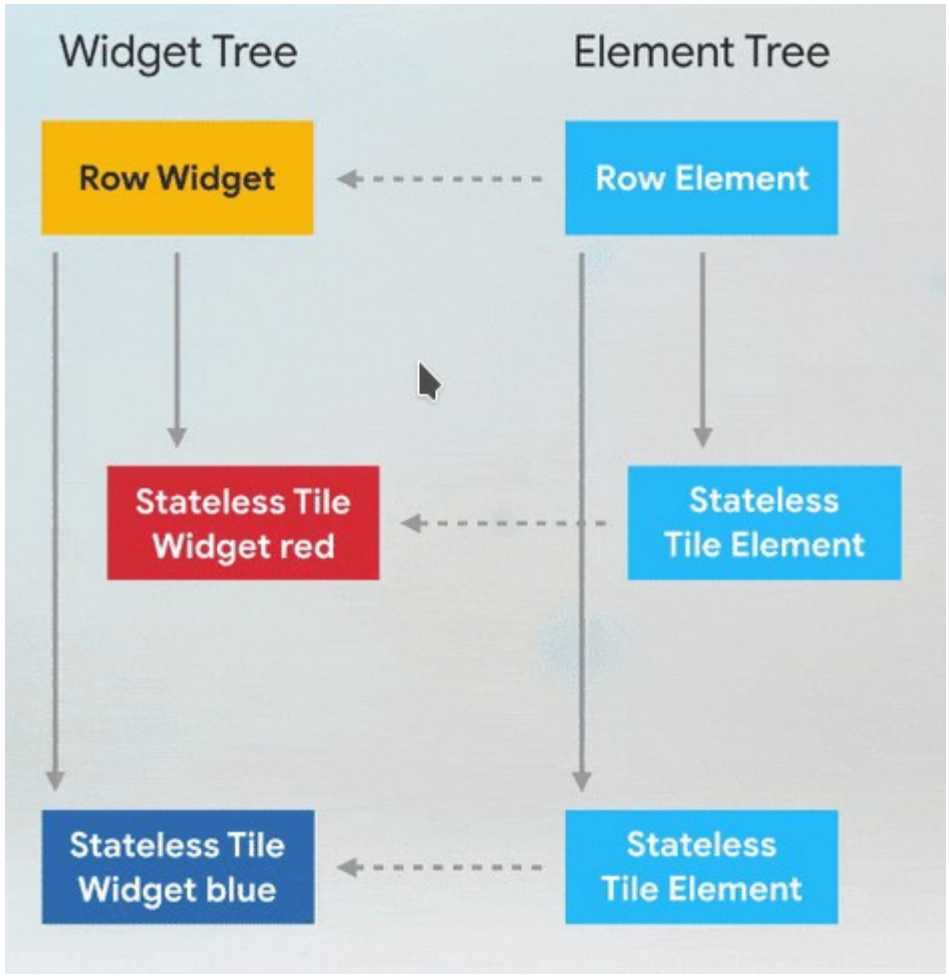


Keys

A aplicação com quadros stateless funciona porque, ao clicar no botão, é trocada a ordem dos wiggets na Widget Tree mas não dos elementos na Element Tree. Porém, como são StatelessWidgets, o Flutter não identifica nenhum problema nos widgets a serem reconstruídos porque os dois widgets são do mesmo tipo e não tem estados para indicar alguma inconsistência.

Keys



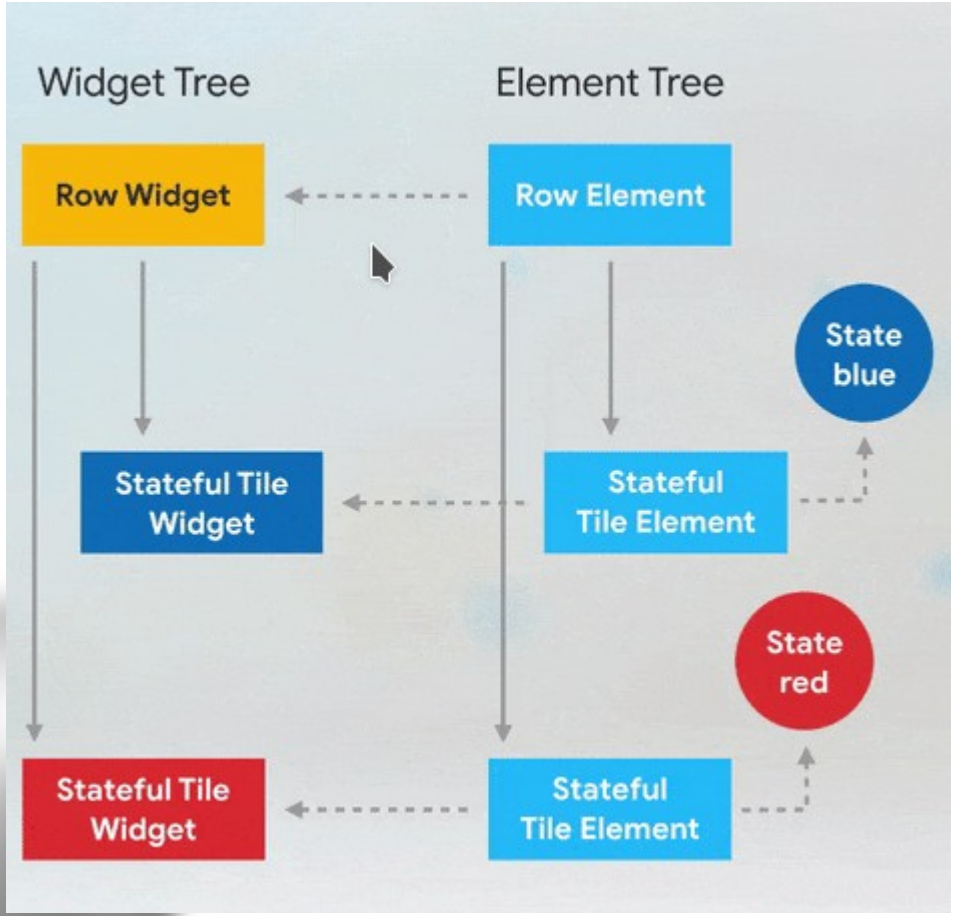


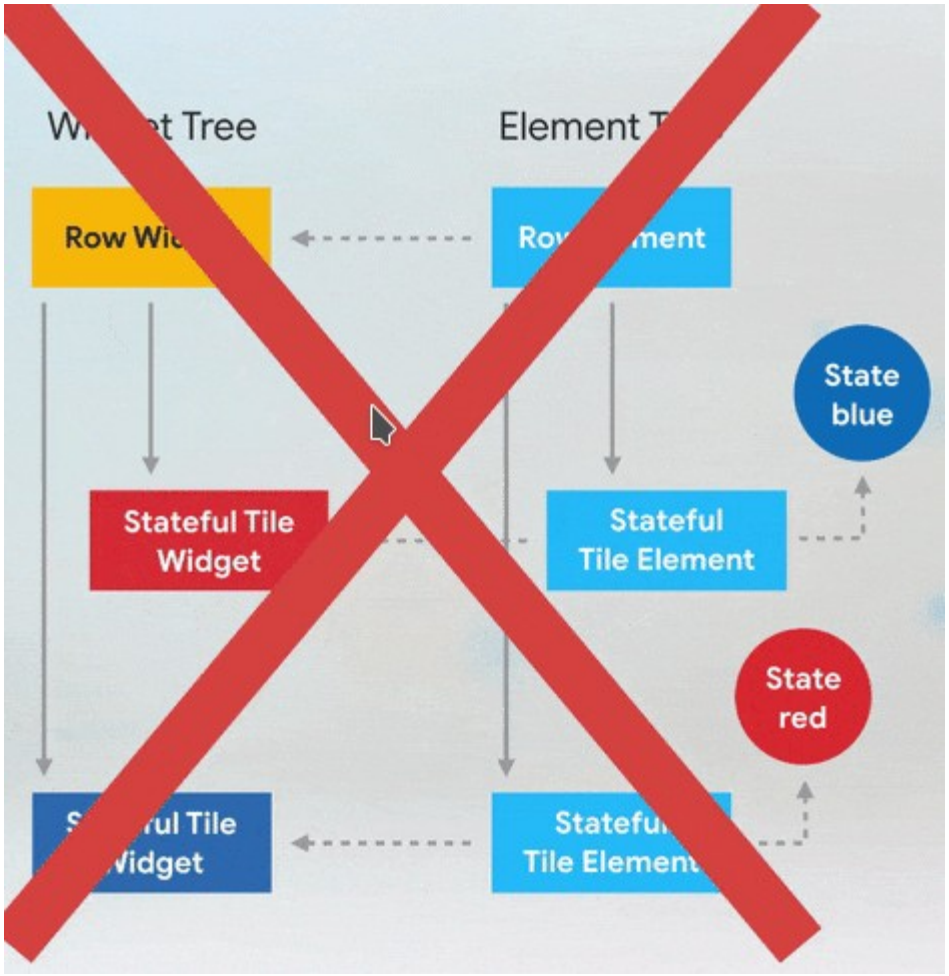
https://medium.com/flutter/keys-what-are-they-good-for-13cb51742e7d

Keys

A aplicação com quadros stateful não funciona porque, ao clicar no botão, é trocada a ordem dos wiggets na Widget Tree mas não dos elementos na Element Tree. E ao tentar restabelecer a ligação dos elementos aos widgets, é identificada que não são correspondentes, mas o Flutter não consegue fazer a ligação correta e portanto não reconstrói os widgets adequadamente.

Keys





https://medium.com/flutter/keys-what-are-they-good-for-13cb51742e7d

Keys

Para corrigir o problema é necessário adicionar uma chave aos quadros a exibir para permitir que o Flutter identifique a necessidade de substituir os elementos na Element Tree e refazer corretamente a ligação entre os elementos e os widgets.

Substitua a lista de quadros a exibir por:

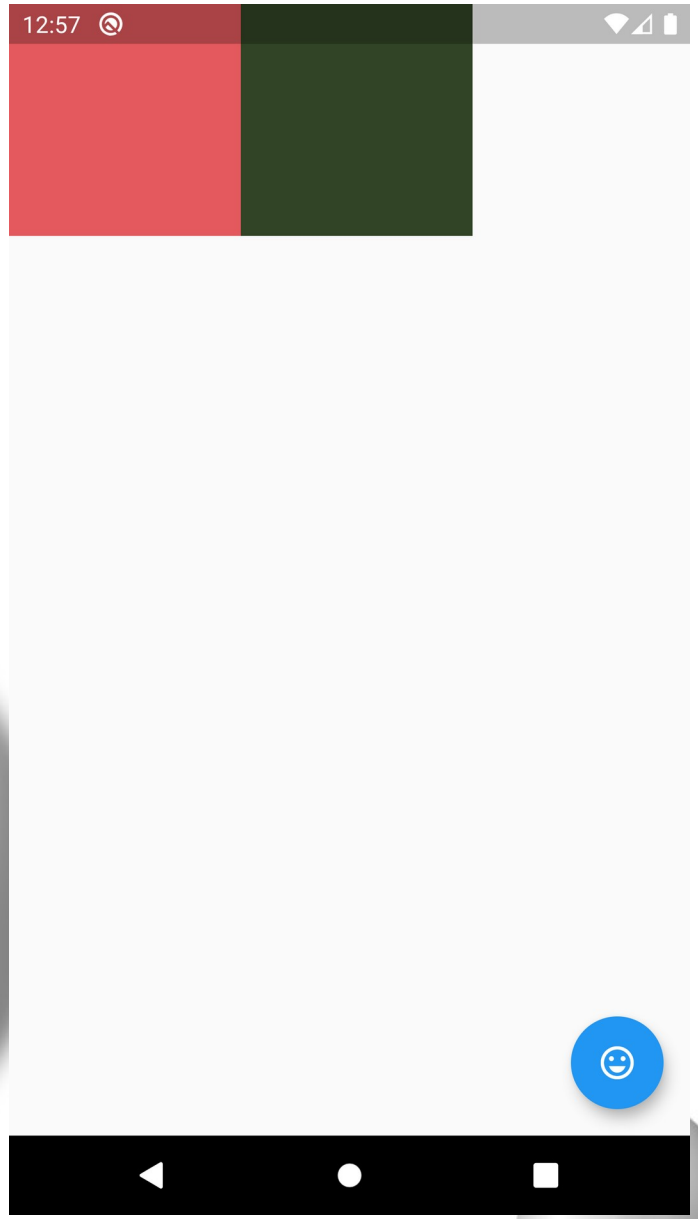
```
//
// Lista de quadros a exibir
//
List<Widget> tiles = [
  ColorTile(key: UniqueKey()), // Keys added here
  ColorTile(key: UniqueKey()),
];
```

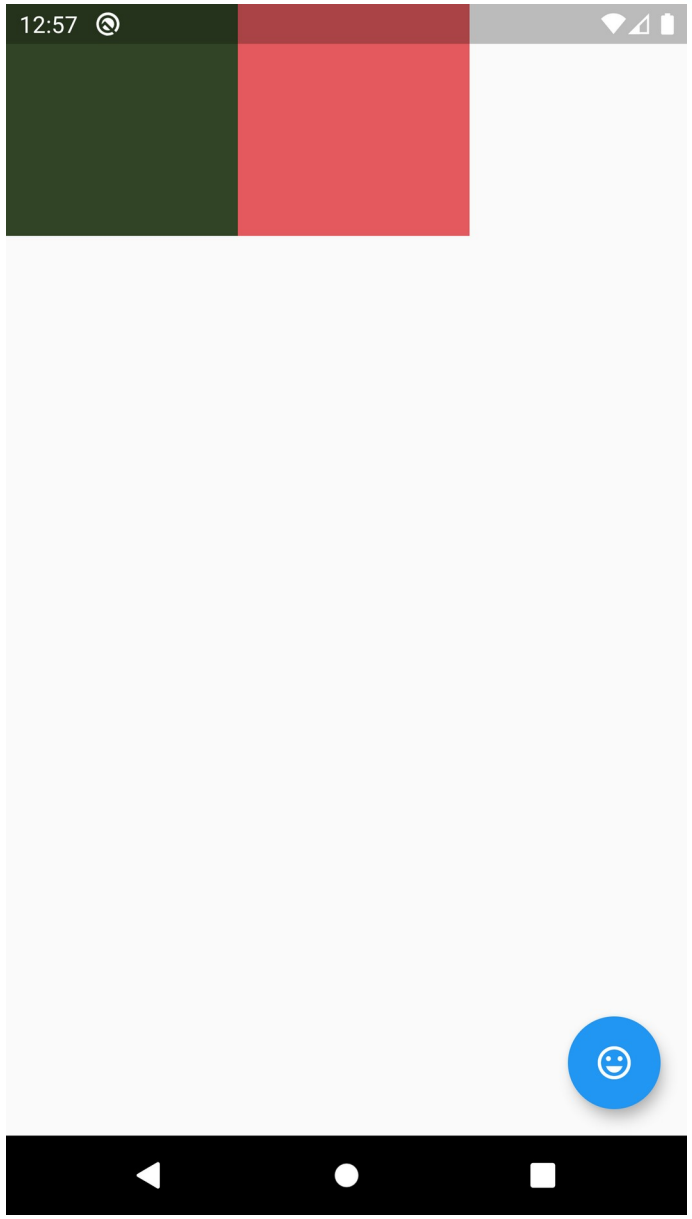
Keys

Acrescente o construtor na classe dos quadros a exibir por:

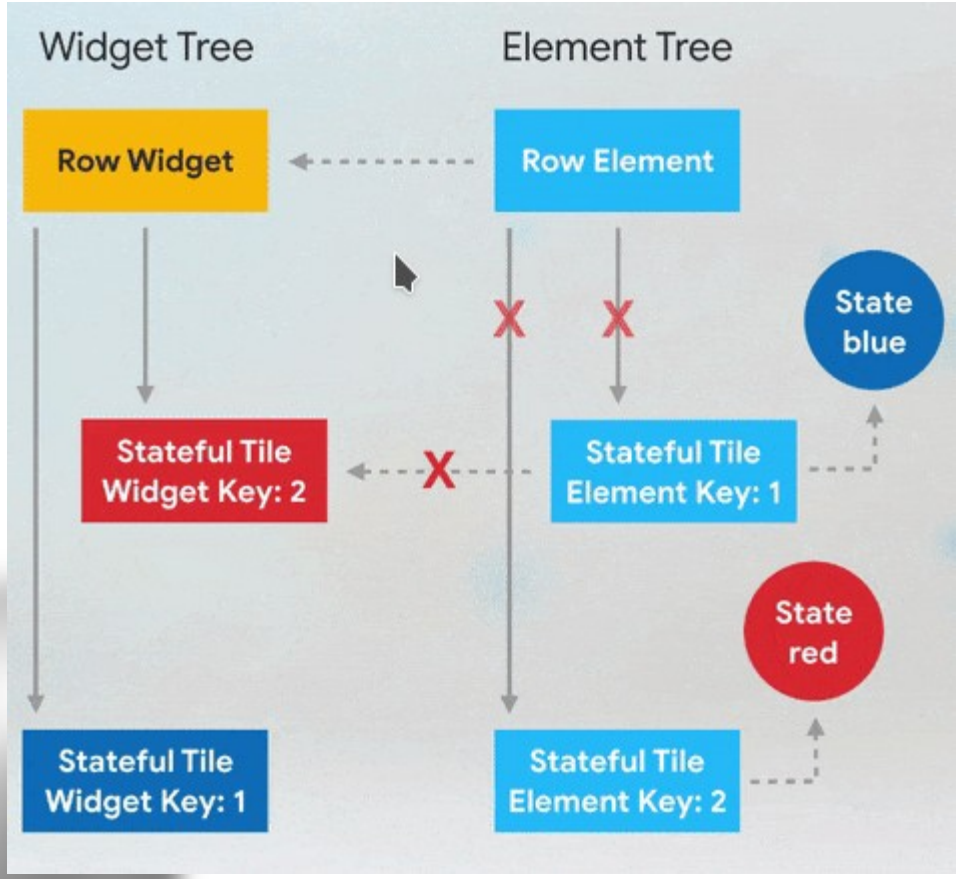
```
//
// Classe para quadros a exibir
//
class ColorTile extends StatefulWidget {
  ColorTile({Key key}) : super(key: key);  // NEW
CONSTRUCTOR

  @override
  ColorTileState createState() => ColorTileState();
}
```

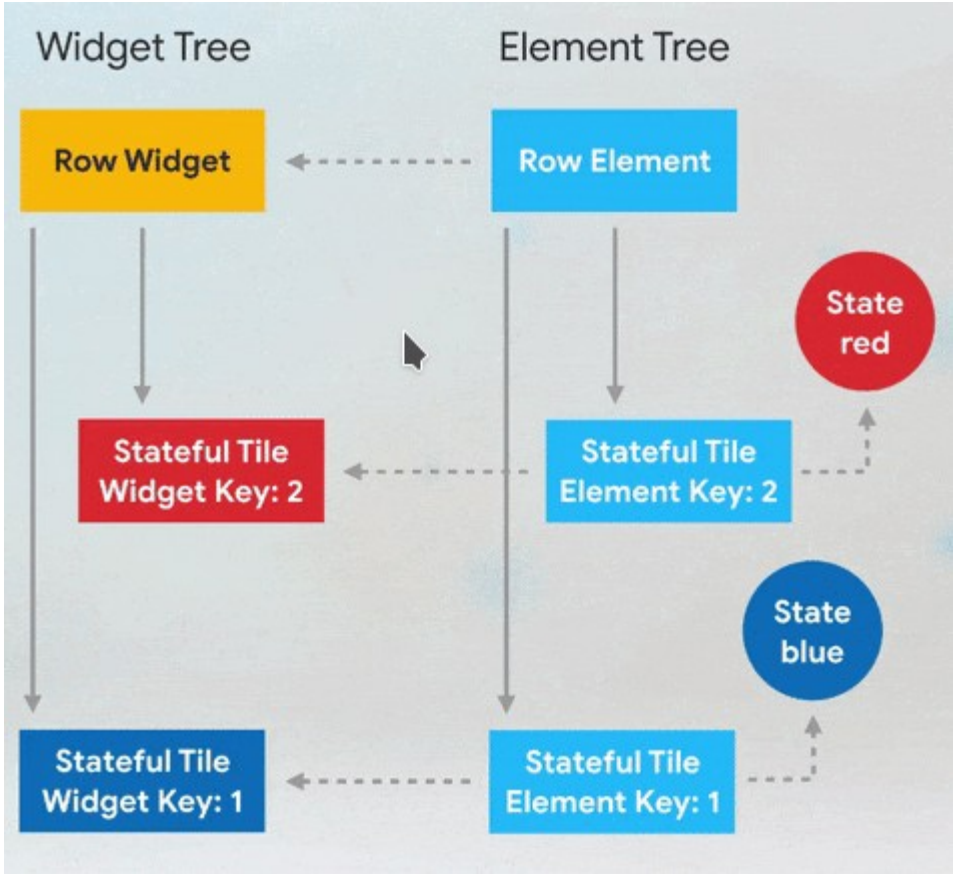
Keys





Keys





https://medium.com/flutter/keys-what-are-they-good-for-13cb51742e7d

Keys

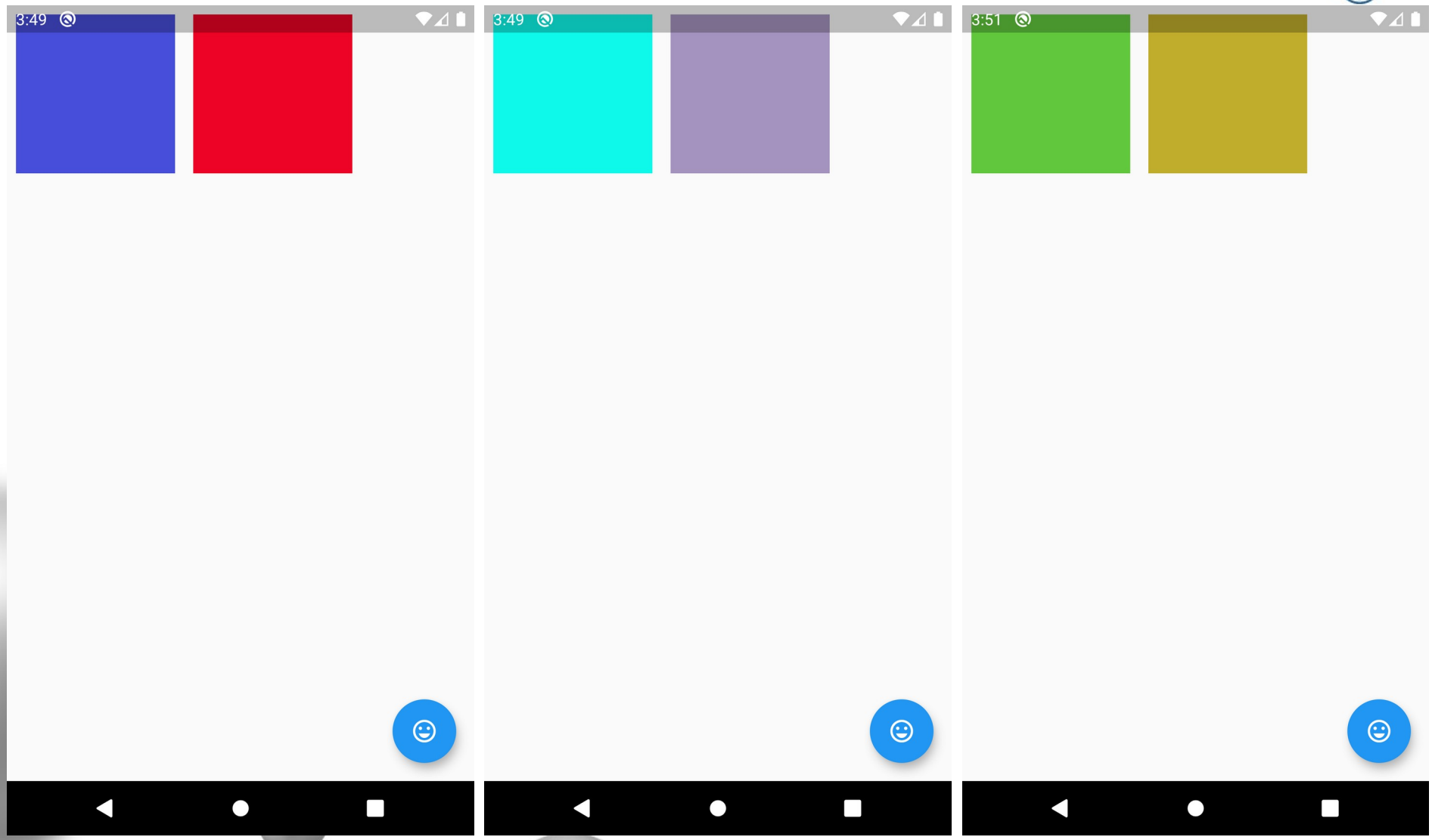
A chave deve ser posicionada no inicio da subárvore que precisa ter o estado preservado.

Crie uma nova aplicação Flutter com o nome **keys_where** e copie o arquivo **main.dart** da aplicação anterior.

Substitua a lista de quadros a exibir por:

```
//
// Lista de quadros a exibir
//
List<Widget> tiles = [
  Padding(
    padding: const EdgeInsets.all(8.0),
    child: ColorTile(key: UniqueKey()),
  ),
  Padding(
    padding: const EdgeInsets.all(8.0),
    child: ColorTile(key: UniqueKey()),
  ),
];
```

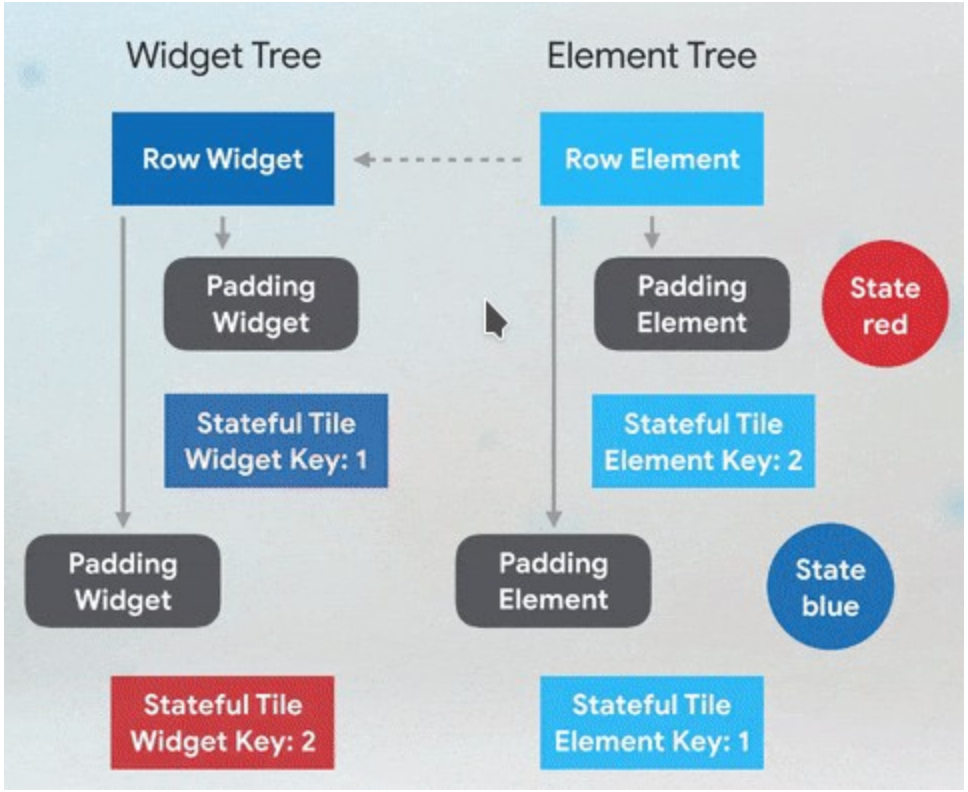
Keys



Keys

O Flutter verifica um nível das árvores por vez. Ao analisar o primeiro nível não há nenhum problema pois o widget e o elementos dos **Paddings** correspondem. Ao analisar o segundo nível, a chave do **StatefulWidget** e do elemento não correspondem. O Flutter pesquisa apenas na subárvore e ao não conseguir corrigir a correspondência entre widget e elemento, recria o elemento, mudando a cor do estado.

Keys



https://medium.com/flutter/keys-what-are-they-good-for-13cb51742e7d

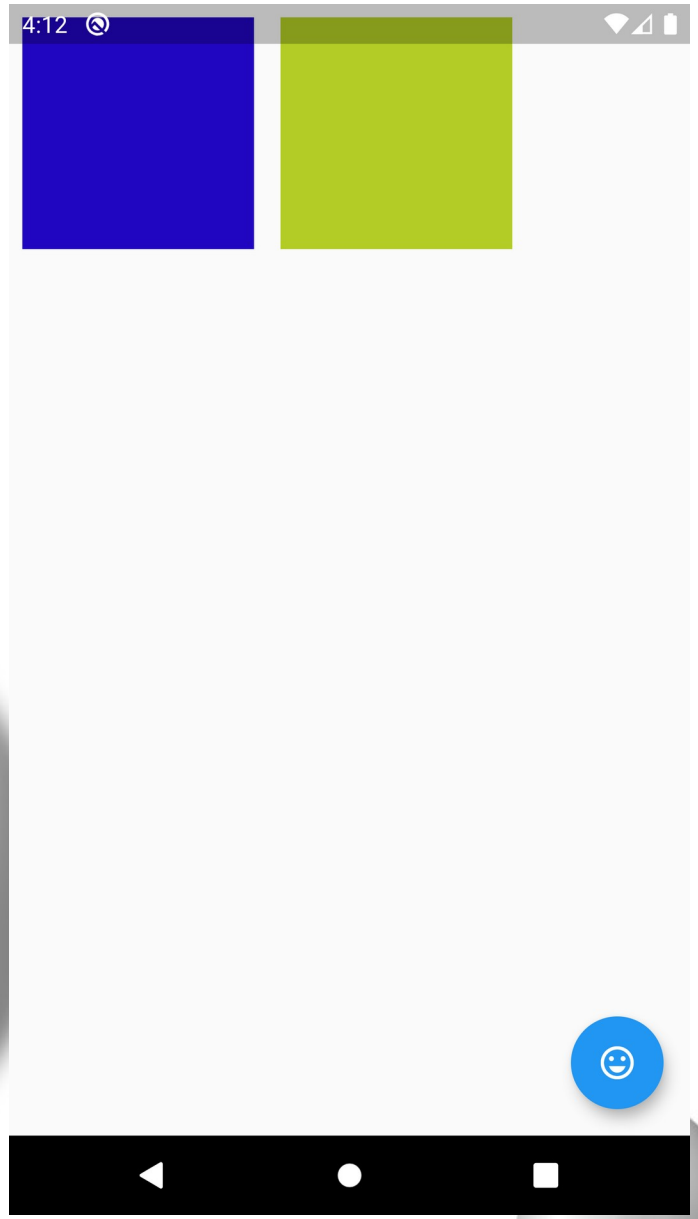
Keys

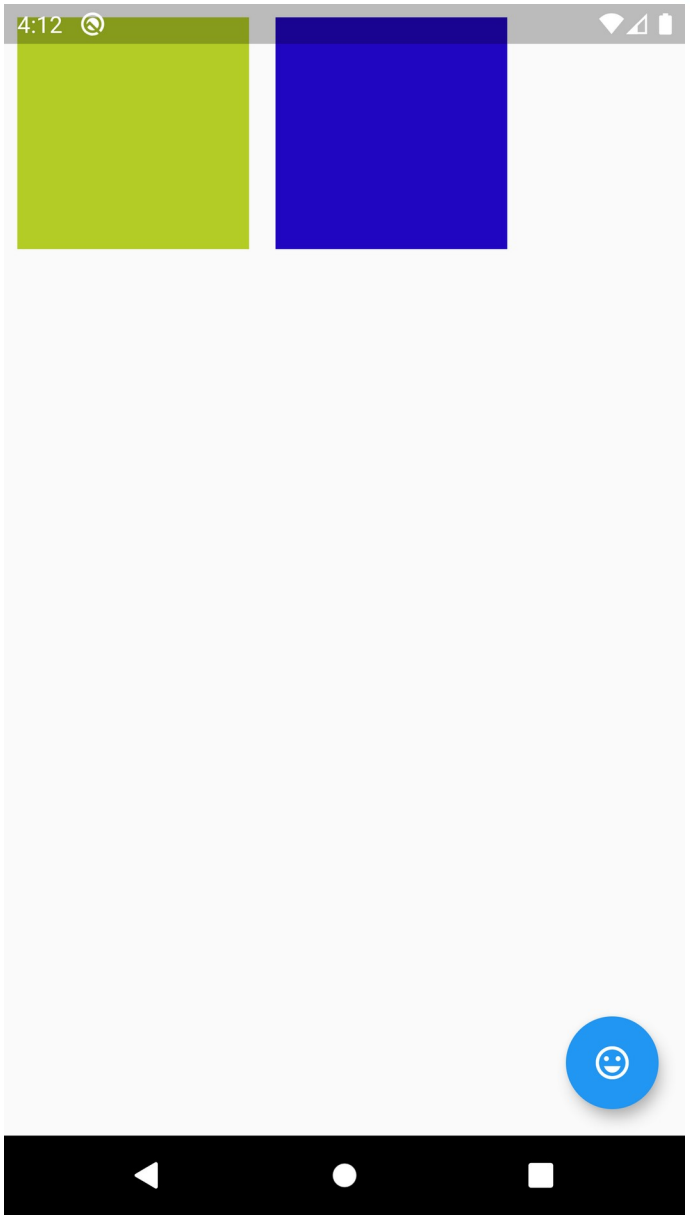
Para corrigir o problema é necessário colocar a chave no **Padding** para que o Flutter identifique o problema ao analisar o primeiro nível.

Substitua a lista de quadros a exibir por:

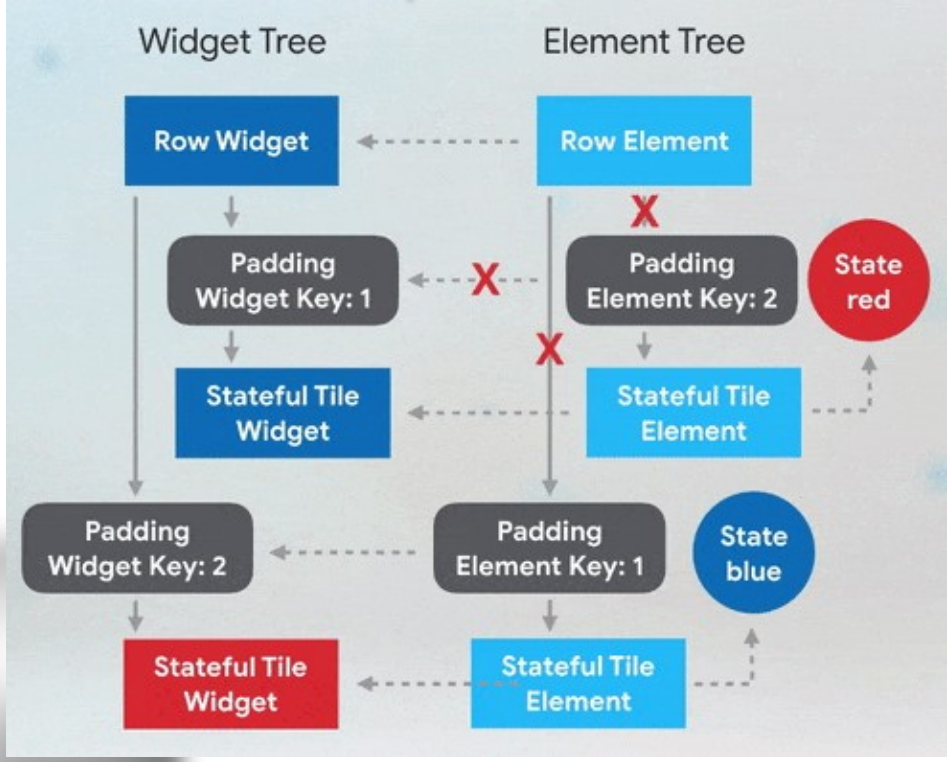
```
//
// Lista de quadros a exibir
//
List<Widget> tiles = [
  Padding(
    // Place the keys at the *top* of the tree
    key: UniqueKey(),
    padding: const EdgeInsets.all(8.0),
    child: ColorTile(),
  ),
  Padding(
    key: UniqueKey(),
    padding: const EdgeInsets.all(8.0),
    child: ColorTile(),
  ),
];
```

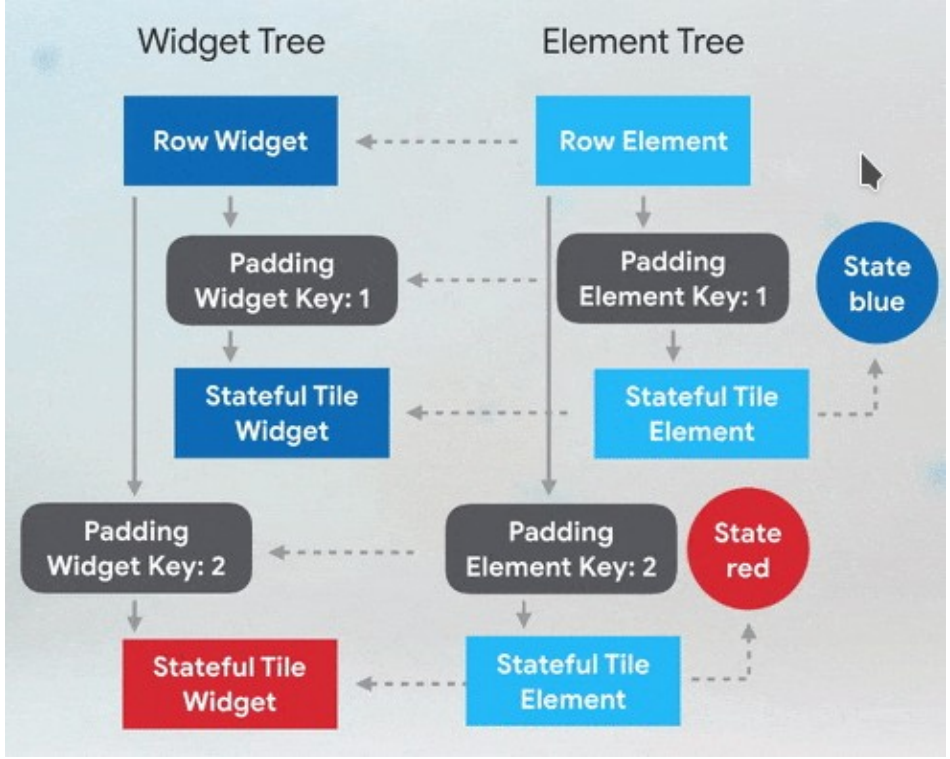
Keys





Keys





https://medium.com/flutter/keys-what-are-they-good-for-13cb51742e7d

Keys

Chaves não são obrigatórias. Apenas em algumas situações é necessário o uso das chaves. Situações como o reposicionamento dos widgets é um dos exemplos. Existem vários tipos de chaves, que devem ser usadas de acordo com o motivo pelo qual a chave será usada.

LocalKey é uma chave que não se repetem dentro da subárvore.

GlobalKey é uma chave que não se repete em toda a aplicação. Permite que um widget seja movido de uma subárvore para outra. Permite que o widget seja acessado fora da sua subárvore. Prove acesso a outros objetos do widget como **BuildContext** e **State**.

Devido ao consumo de recursos, deve se preferir o uso de **LocalKey** quando os recursos da **GlobalKey** não forem necessários.