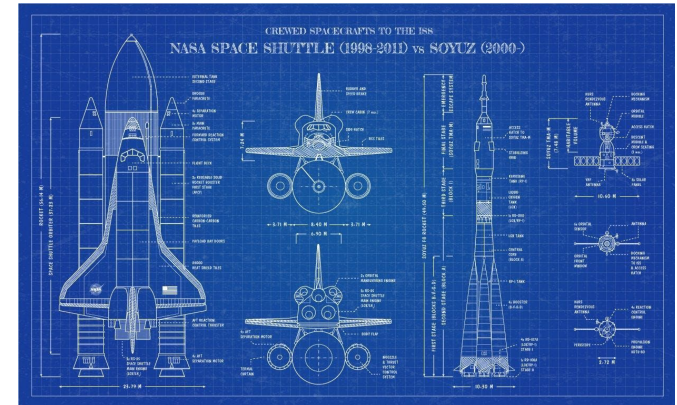


Gerência de estado

Flutter

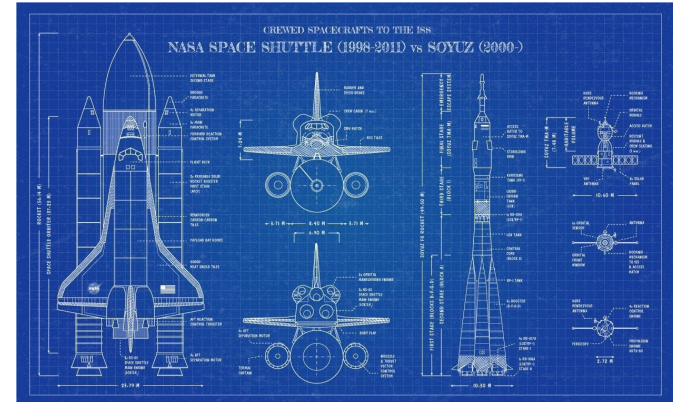
Widgets

- Widgets são os blocos de construção da UI
- A UI é construída aninhando widgets entre si
 - formação de uma árvore
- Reconstrói apenas quando o estado muda
- Dados e configurações que ditam a aparência dos elementos na tela
- A renderização acontece, de fato, apenas posteriormente



Widgets

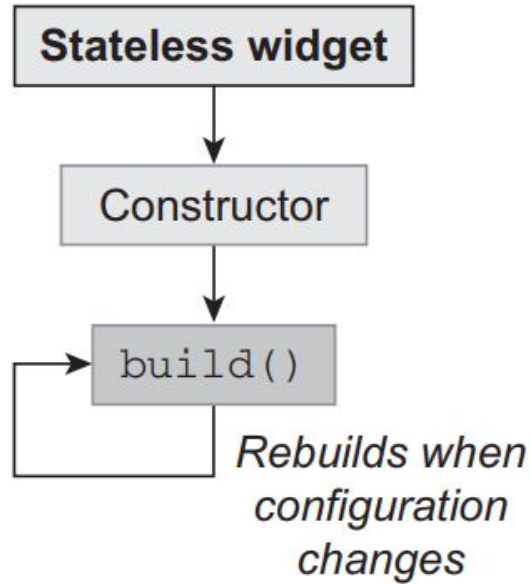
- Widgets aninhados formam um widget maior
- Widgets mais comuns
 - Layout—Row, Column, Scaffold, Stack
 - Estilos—TextStyle, Color, Padding
 - Animações—FadeInPhoto, transformations
 - Posicionamento and alinhamento—Center, Padding



Stateless Widget

- Adequado quando o estado não muda
 - Text, Icon, Button
- Estático e ausente de lógica de negócios
- Não há responsabilidade com variáveis ou estados
- Ausente de controle de estado (setState())

Ciclo de vida do Stateless Widget





```
class ViewHome extends StatelessWidget {  
  
  @override  
  Widget build(BuildContext context) {  
    return Text('Sou um texto');  
  }  
  
}
```

Stateful Widget

- Adequado quando o estado muda
- O valor de suas variáveis muda ao longo do tempo
- A mudança reflete na UI por meio do método `setState()`
- Rastreia seu estado interno
- Tem um objeto de estado correspondente
- Imutável, igual ao stateless widget, porém seu state object é mutável e mantém seus valores após redrawn

```
class MaximumBid extends StatefulWidget {  
  @override  
  _MaximumBidState createState() => _MaximumBidState();  
}  
class _MaximumBidState extends State<MaximumBid> {  
  double _maxBid = 0.0;  
  void _increaseMyMaxBid() {  
    setState(() {  
      // Add $50 to my current bid  
      _maxBid += 50.0;  
    });  
  }  
  @override  
  Widget build(BuildContext context) {  
    return Column(  
      children: <Widget>[  
        Text('My Maximum Bid: $_maxBid'),  
        FlatButton.icon(  
          onPressed: () => _increaseMyMaxBid(),  
          icon: Icon(Icons.add_circle),  
          label: Text('Increase Bid'),  
        ),  
      ],  
    );  
  }  
}
```


Stateful Widget - responsabilidades

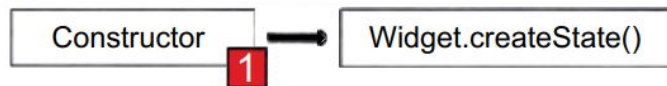
- Stateful Widget
 - Cria uma instância de state
- State
 - Armazena os dados de quando o widget é criado. Esses dados podem mudar com o tempo
 - Estado de um state object é a informação que
 - (1) pode ser lida sincronamente a partir de quando um widget é construído
 - (2) pode mudar durante o ciclo de vida do widget.
 - É responsabilidade do implementador (aquele implements State<T>) garantir que o Flutter é notificado quando o estado muda, usando `State.setState()`
- `setState()`
 - Dentro da classe State, o `setState()` serve para notificar a atualização dos dados alterados
 - Isso sinaliza ao framework que os dados mudaram e é preciso uma nova renderização
- Mais detalhes no ciclo de vida

Stateful Widget - ciclo de vida

- O framework provê métodos no object state para permitir resposta às alterações na árvore de elements

Stateful Widget - ciclo de vida

1 Stateful widget



2 State object

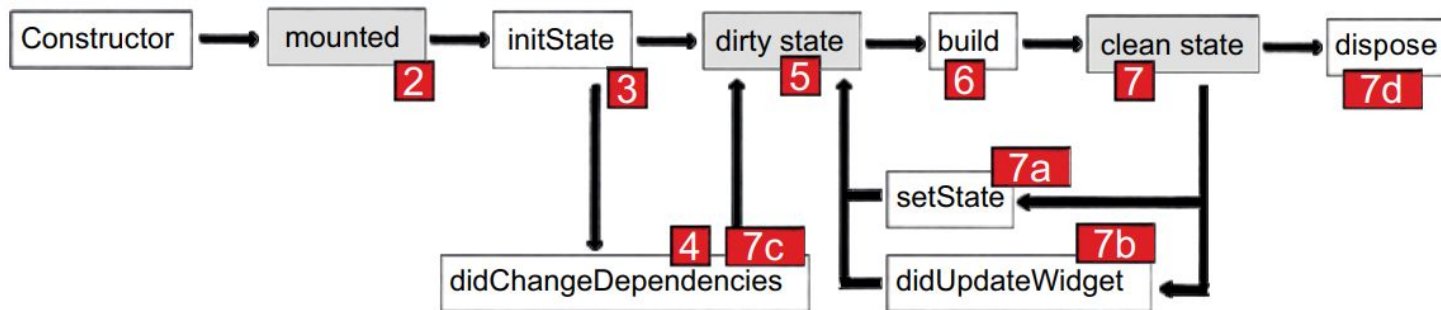


Figure 8.2 State object lifecycle

Stateful Widget - ciclo de vida

1. O construtor da classe é chamado. O widget ainda não está na árvore nesse ponto.
2. O objeto de estado é associado com um BuildContext, um local na árvore. Agora o widget está montado. Cheque pelo valor da variável widget.mounted.
3. O State.initState() é chamado. Esse método é chamado apenas uma vez para a inicialização do widget. Ele deve ser usado para inicializar propriedades no state object

Stateful Widget - ciclo de vida

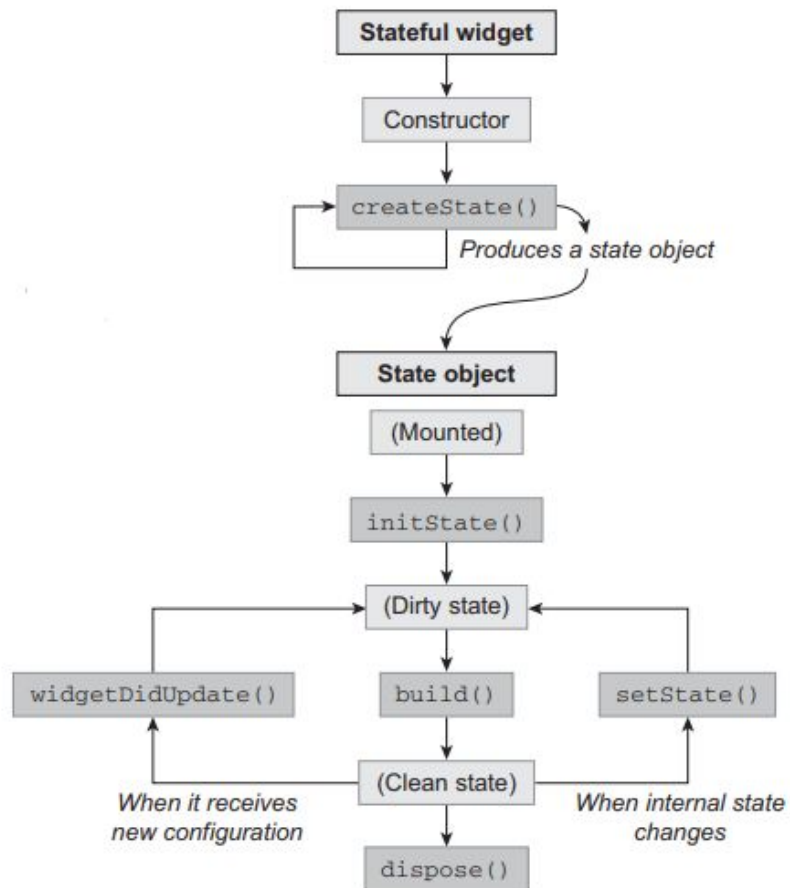
4. O `State.didChangeDependencies()` é chamado apenas uma vez após o `State.initState`. Serve para inicialização que envolve o `InheritedWidget`.
5. O estado está como “sujo”. Logo, os widgets precisam ser reconstruídos. “Sujo” significa que a tela precisa ser renderizada para refletir o novo estado. Variáveis e tela estão em desarmonia
6. O object state é completamente inicializado e o `State.build()` é chamado

Stateful Widget - ciclo de vida

7. Após build, o estado está “limpo”. As variáveis e telas estão em harmonia
 - a. `state.setState()` é chamado a nível de código (sua responsabilidade, dev). Essa chamada torna o estado como “sujo”
 - b. Um widget ancestral pode requisitar que esse local na árvore seja reconstruído. Se o local é reconstruído com o mesmo tipo de widget e mesma chave, o Flutter chama o `State.didUpdateWidget()` com o widget anterior como argumento. O estado também torna-se “sujo”.
 - c. Se seu widget depende de um `InheritedWidget`, este mudando, então o Flutter chama o `State.didChangeDependencies()`. O widget vai “rebuildar”
 - d. O state object sendo removido da árvore chama o `State.dispose()`. Aqui você limpa tudo usado pelo widget: para animações, fecha streams e controladores. Não deve chamar `setState()`

Stateful Widget

- ciclo de vida



Stateful Widget - initState()

- Chamado assim que o widget é montado na árvore e o state object é construído
 - Executa apenas uma vez
 - Evita chamadas repetidas de funções que são executadas no build (quando setState() é usado)
- State.initState() é um método a qual permite a inicialização de dados antes de o Flutter renderizar a tela pela primeira vez
 - Buscar uma lista de dados que vieram de uma API

Stateful Widget - SetState()

- Chamado quando quer que a tela seja renderizada novamente
 - Necessário porque os dados mudaram

Stateful Widget - SetState()

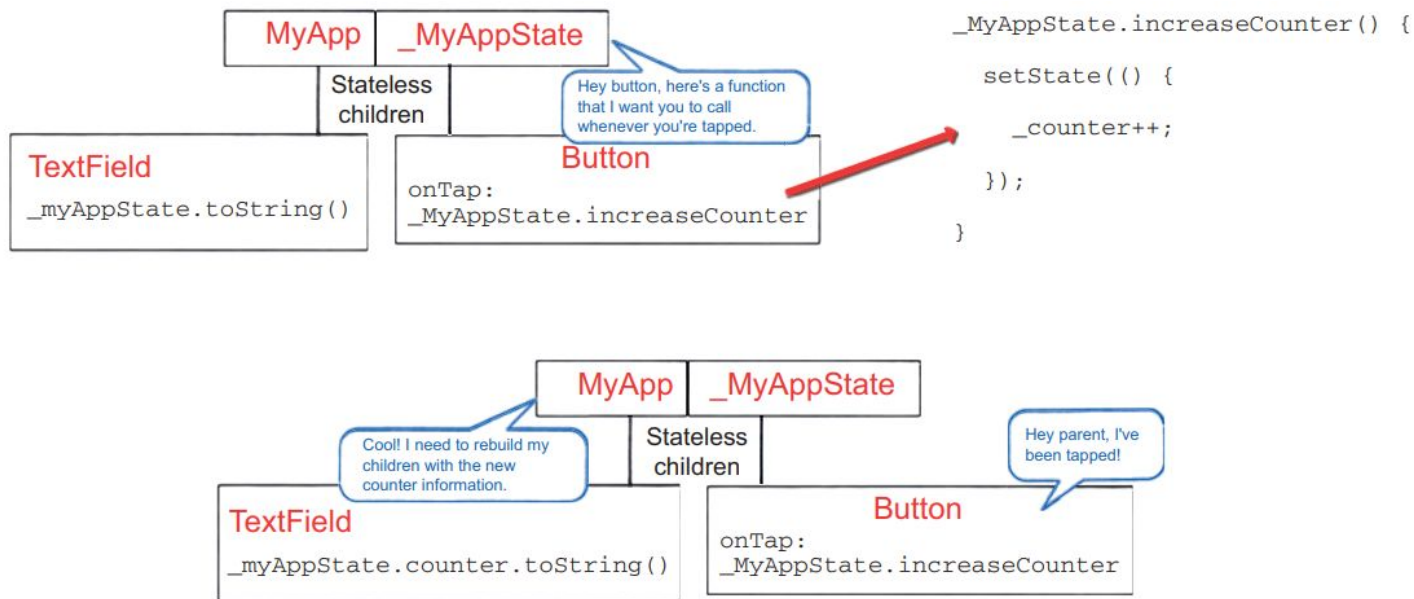


Figure 3.5 `setState` visual

Problemas do setState

- Chama o método build inteiro
- Variável única do widget
 - Não permite outro widget modificar a variável
- Lógica de negócios e view fortemente acopladas

Widget - BuildContext

- Rastreia a árvore de widgets
 - Localiza a posição dos widgets na árvore
- Passado em todo método build(BuildContext context)
- MediaQuery.of(context).size.height e MediaQuery.of(context).size.width
- Tipo.of(context) Busca o pai mais próximo do tipo que está procurando
- Toast, Modal e Rotas usam o context
 - O flutter precisa saber em qual local da árvore o modal precisa ser inserido, por exemplo
- O foco é o local dos widgets na árvore

Árvore de widget

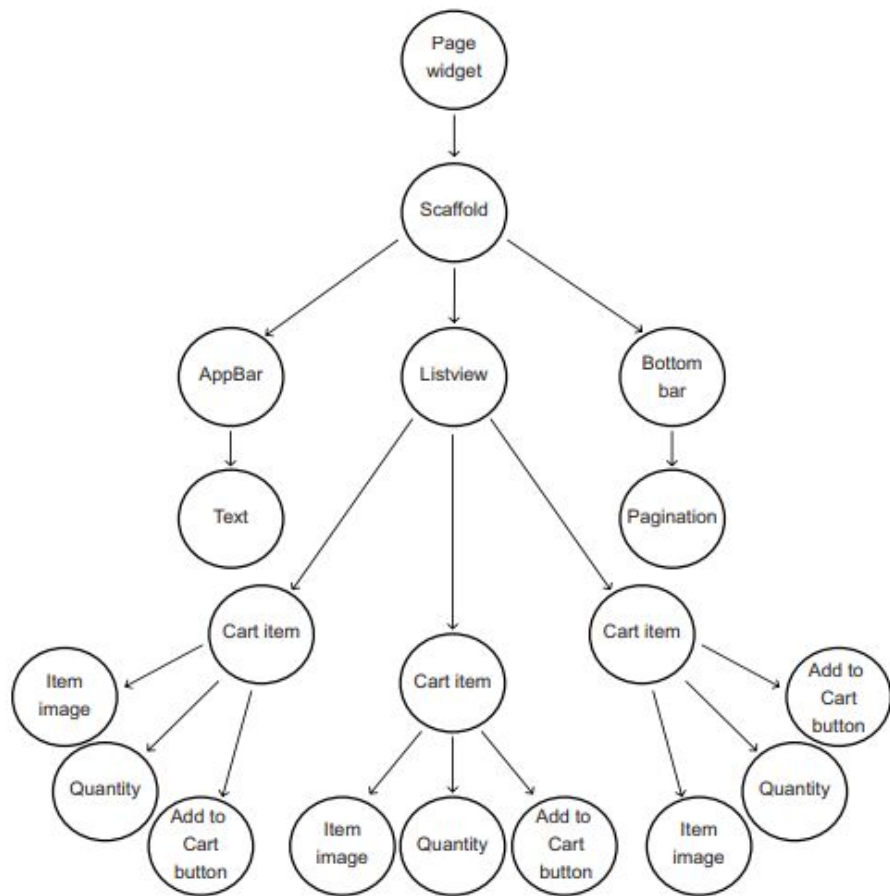


Figure 1.10 This represents a widget tree, but in reality there are far more widgets in the tree than I can show.

Árvore de widget - BuildContext

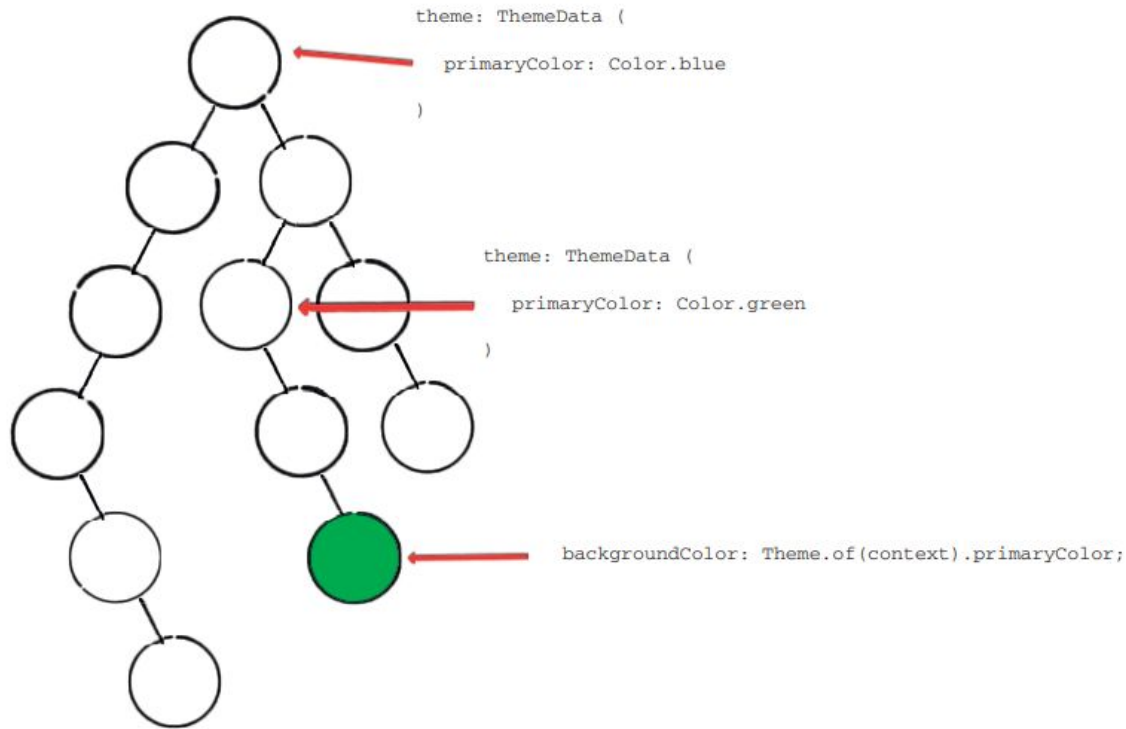


Figure 3.7 Using Theme in Flutter

Widgets e Elements

- Elements são criados por widgets
 - `Widget.createElement(this)`
- O element tem uma referência ao widget criador
- Widgets são substituídos. Elements são atualizados
- Widgets são descartáveis
 - Barato de substituir sem alterar a árvore que vai ser renderizada (elements tree)
- Elements gerenciam state objects
- render objects são baseados em elements

Widgets e a árvore de stateless/stateful elements

- Relação 1:1
 - 1 stateful widget tem 1 stateful element correspondente
- O framework chama o `createElement()` e o monta na árvore de elements
- O stateless/stateful element requer que o widget crie um objeto de estado (state object)
 - Chamada do `createState()`
 - Não permite outro widget modificar a variável
- Stateful Widget \leftarrow Stateful element \rightarrow State
- Stateless Widget \leftarrow Stateless element
- Processo ocorre iterativamente para os filhos
- Duas árvores são criadas

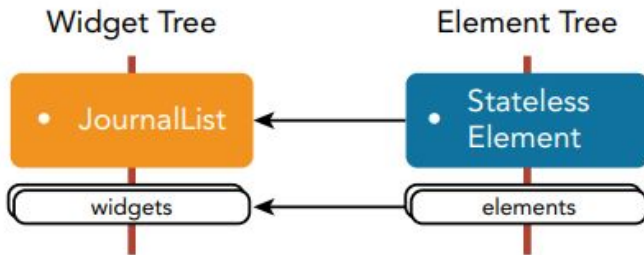


FIGURE 1.3: Widget tree and element tree

Widgets e a árvore de stateless/stateful elements

```
class JournalList extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Row(  
      children: [  
        Icon(),  
        Text(),  
      ],  
    );  
  }  
}
```

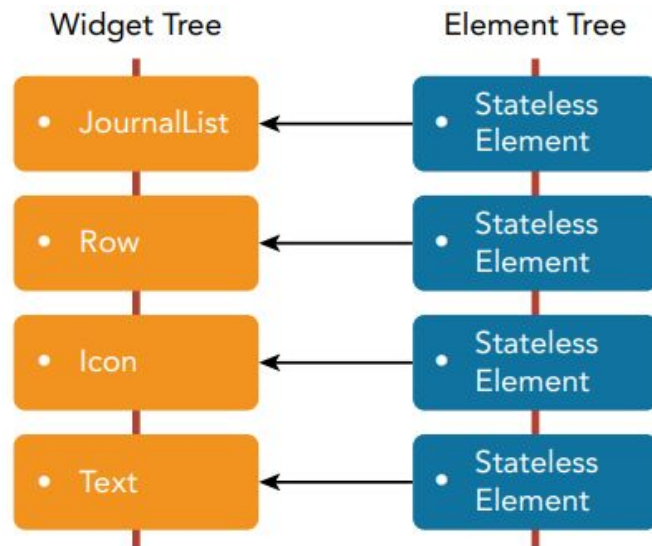


FIGURE 1.4: StatelessWidget widget tree and element tree

Árvore de elements e state objects

- State objects são administrados pelos elements
- State objects e widgets tornam-se independentes
 - Esses objetos têm vida longa. Widgets são destruídos e construídos a todo momento
- State objects são reusados
- Elements são simples porque contêm apenas metainformações e uma referência para o widget, mas sabem atualizar a referência para um novo widget

Árvore de widgets, elements e state object



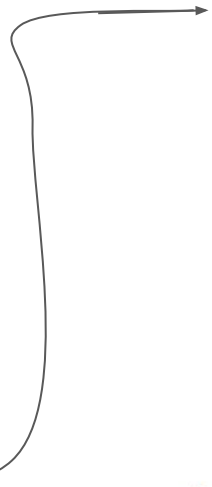
Figure 3.17 The relationship between an element and a widget

```
// Simplified sample code
class JournalEdit extends StatefulWidget {
  @override
  _JournalEditState createState() => _JournalEditState();
}
```

```
class _JournalEditState extends State<JournalEdit> {
  String note = 'Trip A';
```

```
void _onPressed() {
  setState(() {
    note = 'Trip B';
  });
}
```

```
@override
Widget build(BuildContext context) {
  return Column(
    children: <Widget>[
      Icon(),
      Text('$note'),
      FlatButton(
        onPressed: _onPressed,
      ),
    ],
  );
}
```



Árvore de widgets, elements e state object

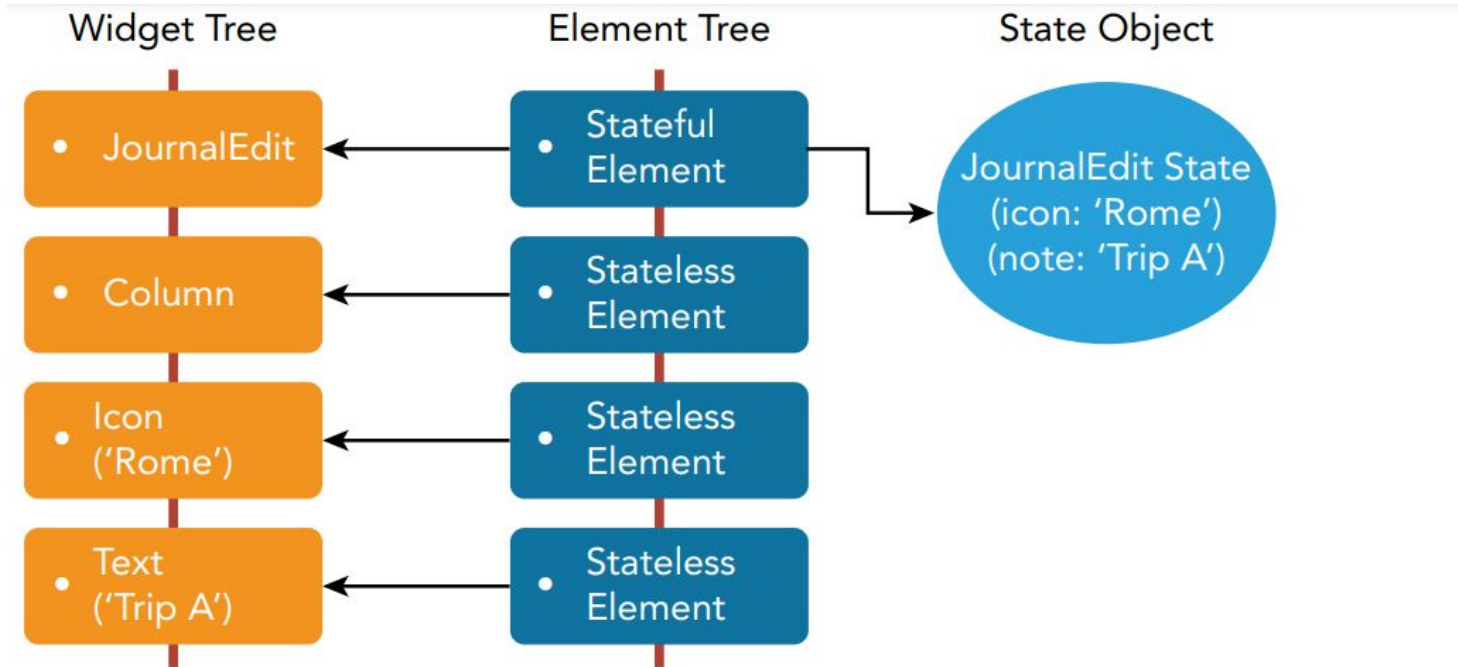


FIGURE 1.5: StatefulWidget widget tree, the element tree, and the state object

Árvore de widgets, elements e state object

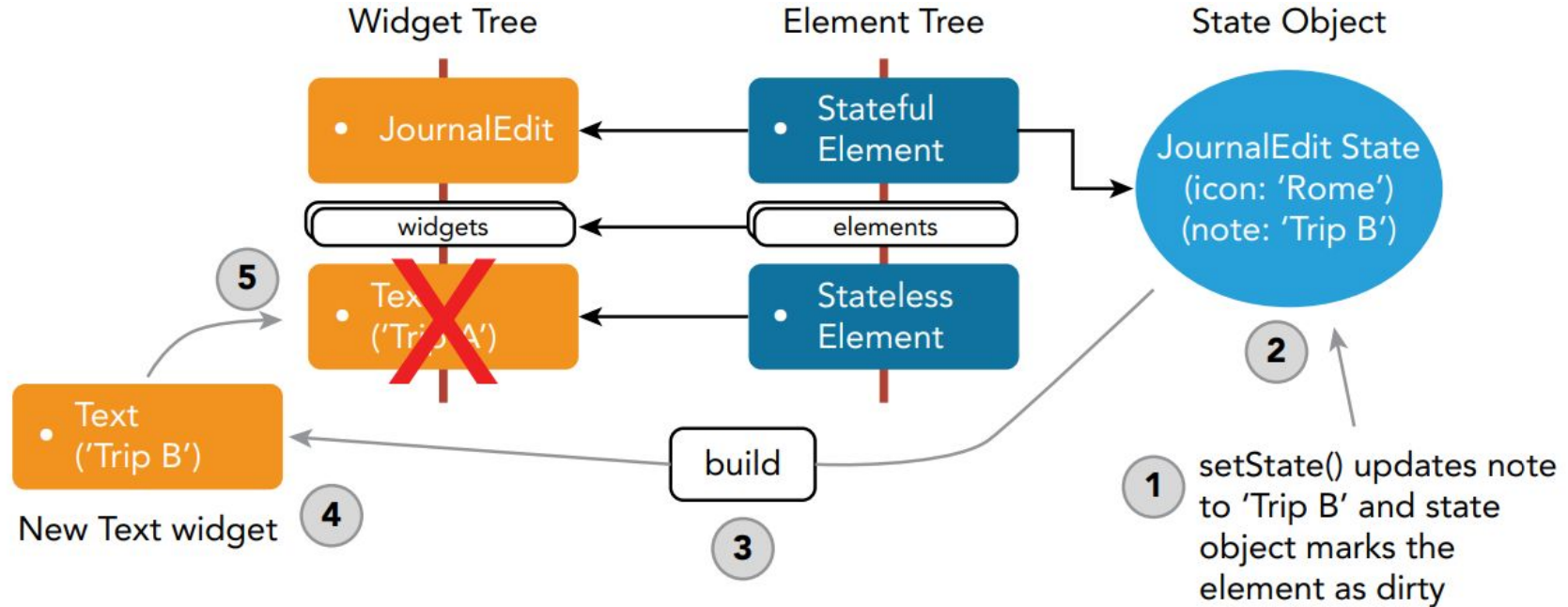


FIGURE 1.6: Updating the state process

Árvore de widgets, elements e state object

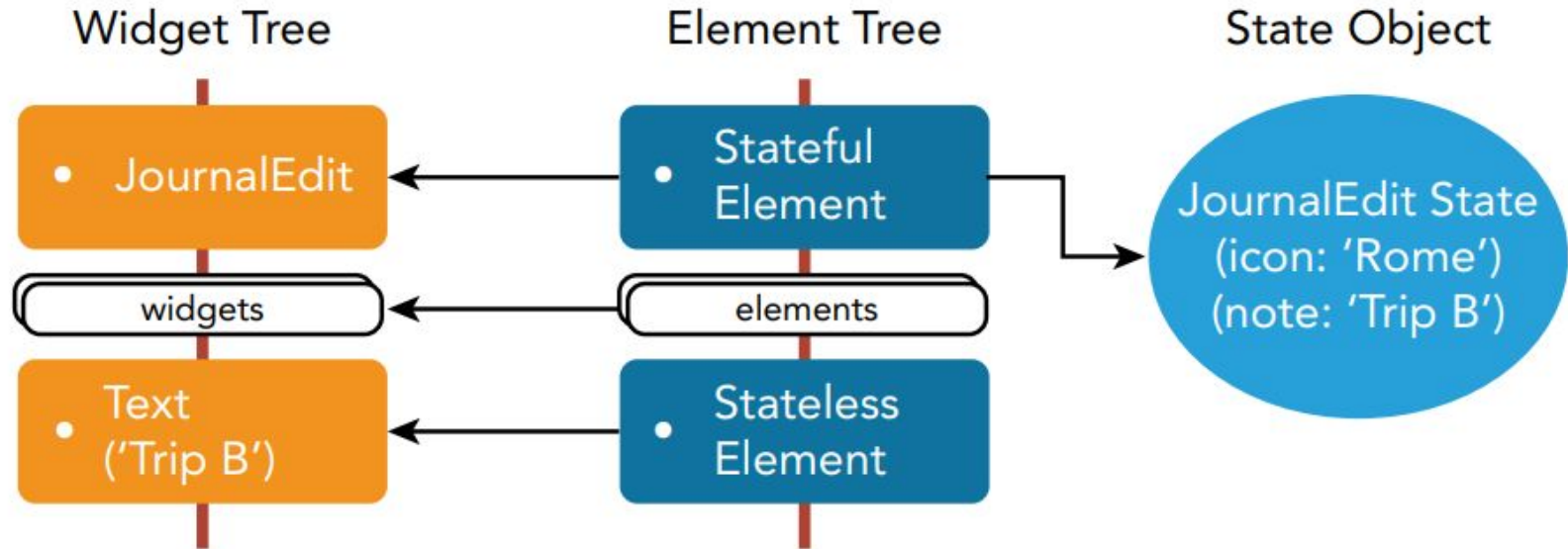


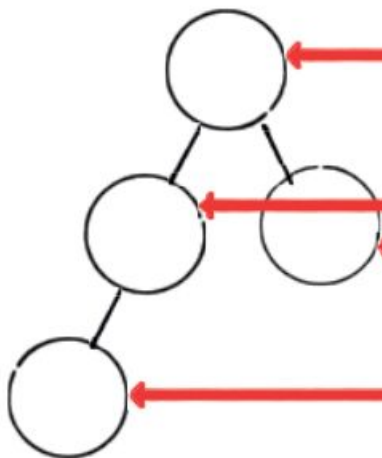
FIGURE 1.7: Updated state for the widget tree and element tree

RenderObject

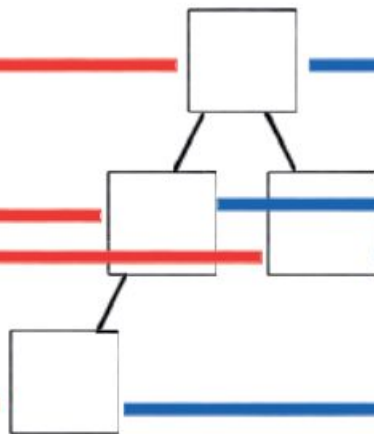
- Uso interno
- Responsável por “pintar”, literalmente, pixels na tela
- Árvore de renderização
 - Terceira árvore
 - Widgets, Elements e Renders
- Tem seus widgets correspondentes
- Toda estilização e layout dos widgets são abstrações para os objetos de renderização
 - Lembre, widgets são *blueprints*
- Não possuem estado ou lógica de negócios

As três árvores

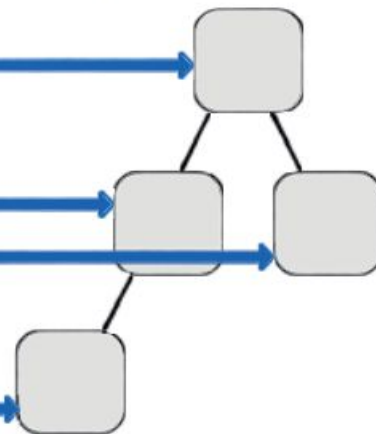
Simple widget tree



Simple element tree



Simple render tree



Elements have references to widgets, which are configurations for elements.

Elements have render objects, which use `dart:ui` to paint the elements to the screen.

Processo de renderização - SKIA 2D

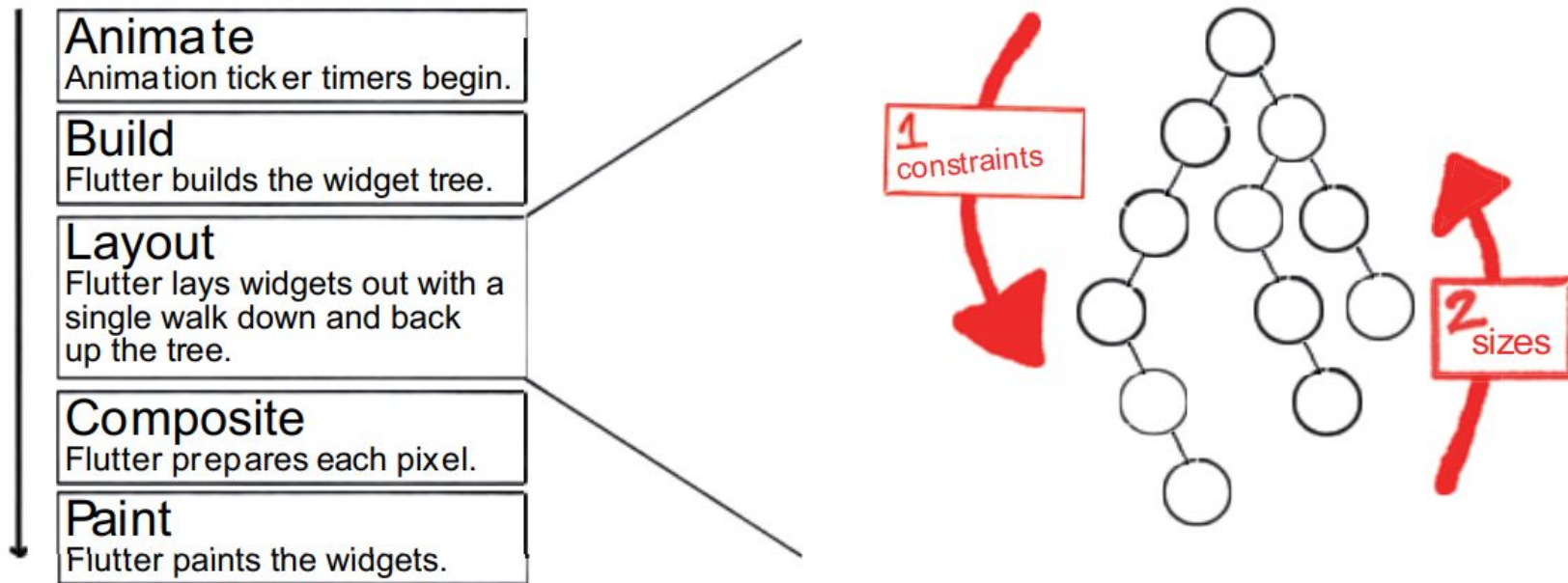


Figure 1.11 The high-level steps of the rendering process

Compondo a árvore de widgets e layout

- Flutter caminha “para baixo” na árvore
 - Apenas uma vez e em tempo linear $O(n)$
- Coleta de informações sobre a posição dos widgets
 - Layout e restrições de tamanho são definidas pelos widgets pais

Compondo a árvore de widgets e layout

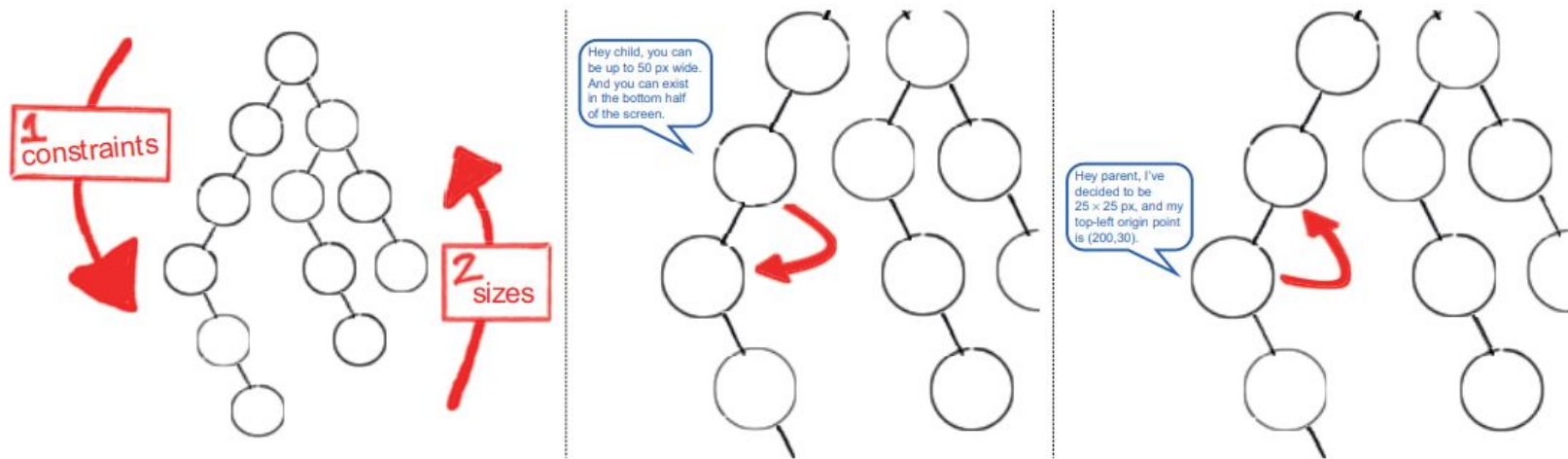


Figure 1.12 Flutter lays out all the widgets in one walk down and back up the tree, because widgets dictate their children's size constraints in Flutter.

Lembre...

- Tudo é widget
- State objects têm vida longa e são reusados
- Constraints dos widgets são definidas pelos widgets pais
- <https://www.youtube.com/watch?v=UUfXWzp0-DU>

Resumo

- Em Flutter tudo é widget
- Widgets são abstrações de alto nível que fornecem uma abordagem declarativa para construção de UI
- Todo widget precisa de um método build
- Widgets devem ser imutáveis (stateful e stateless), porém state objects não devem ser
- O Stateful Widget rastreia seu próprio estado interno, via um objeto de estado associado. Stateless Widget não possui controle de estado
- Elements e render objects podem ser manipulados via código, mas é desaconselhado
- Reúso de elements e render objects. Descartabilidade de widgets

Resumo

- Ciclo de vida do stateful widget dá um controle fino sobre o reconstrução de widgets usando esses métodos:
 - initState
 - didChangeDependencies
 - build
 - widgetDidUpdate
 - setState
 - dispose
- SetState() é usado para dizer ao Flutter que algum estado foi atualizado e a tela deve ser redesenhada
- InitState() serve para inicializar as variáveis do state object
 - Antes mesmo da primeira renderização
- BuildContext é uma referência para o local do widget na árvore

Resumo

- Flutter cria sua tela para refletir o estado do aplicativo
- Mudança de estado dispara um redrawn

The diagram illustrates the Flutter UI model as a function: **UI = f(state)**.

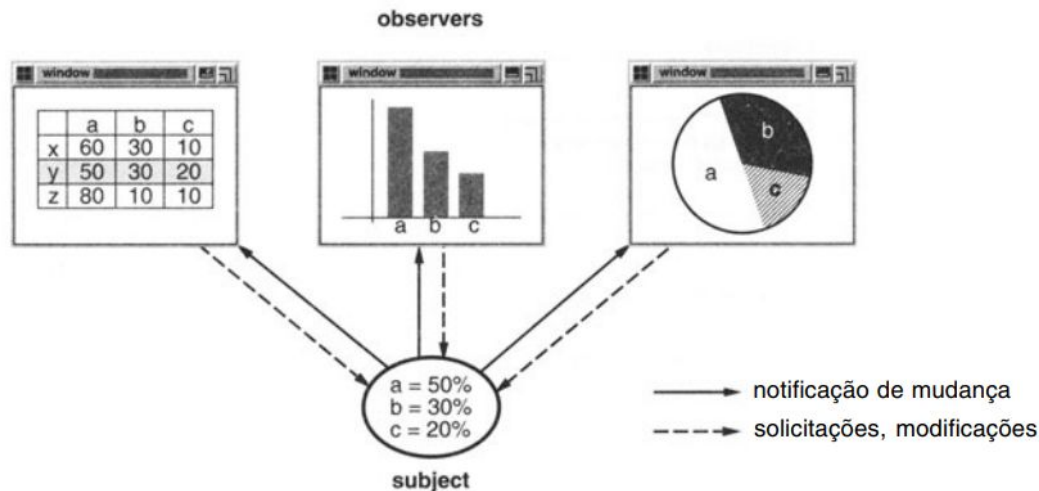
- UI** (red text): The layout on the screen
- =** (gray text):
- f** (blue text): Your build methods
- (** (blue text):
- state** (green text): The application state
-)** (blue text):

Padrão de projeto Observer

- Padrão comportamental
- Define uma dependência 1:n entre objetos
 - Um objeto muda de estado
 - Todos os seus dependentes são notificados e atualizados automaticamente
- Motivação
 - Necessidade de objetos cooperarem e continuarem consistentes
 - Evitar forte acoplamento entre classes
 - Separar a UI e as regras de negócios permite reúso

Padrão de projeto Observer

- Separação da UI e regras de negócios
 - Reutilização independente
- Planilha, gráfico de barras e de setor circular (pizza) apresentam visualizações diferentes para os mesmos dados
- Classes de visualização desacopladas, porém comportamentalmente sincronizadas



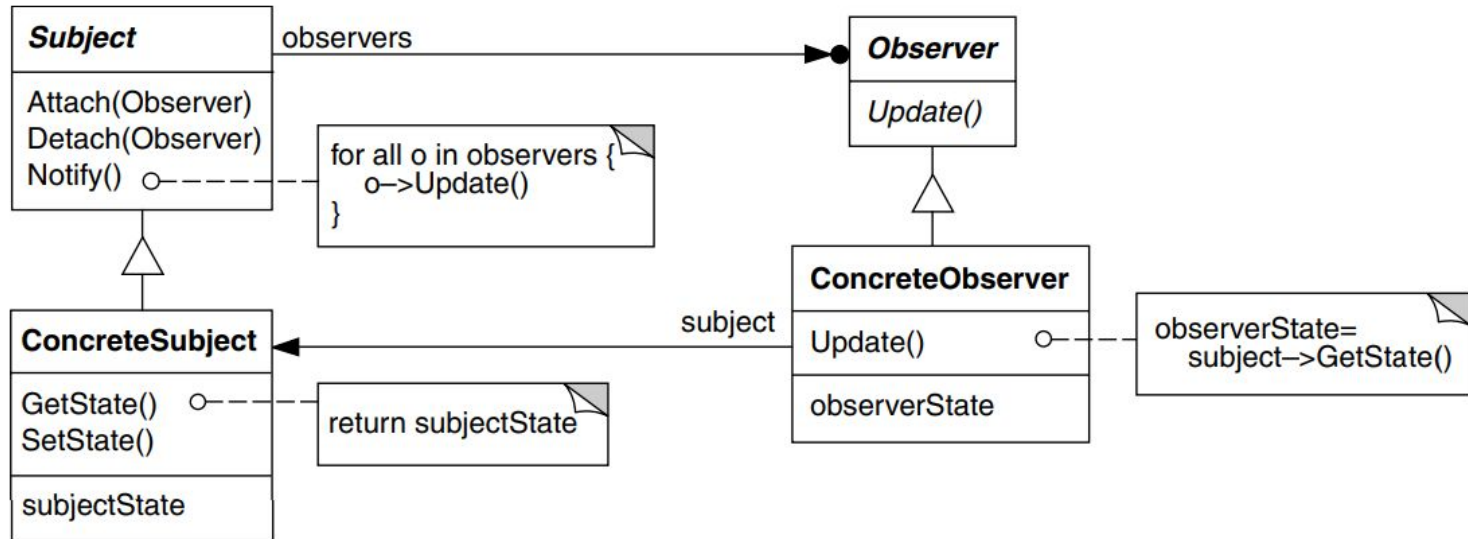
Padrão de projeto Observer

- O padrão observer descreve como estabelecer os relacionamentos entre objetos observados e observadores
- Objetos subject (assunto) e observer (observador)
 - Não há limites para o número de observadores dependentes
- Todos os observadores (observers) são notificados quando há uma alteração no estado do objeto observado (subject)
- Publish-subscribe
- Subject publica notificações
 - “Broadcast” sem conhecer os recebedores

Observer - Aplicabilidade

- Quando a mudança em um objeto causa modificações em outros
 - Pode não saber a quantidade desses outros
- Notificar outros objetos sem conhecê-los
 - Baixo acoplamento

Observer - Estrutura



Observer - Participantes

- **Subject**

- Conhece os seus observadores
 - Não há limites de quantidade
- Fornece interface para adicionar e remover observadores

- **Observer**

- Define interface de atualização para os objetos que vão ser notificados sobre as mudanças

- **ConcreteSubject**

- Armazena os estados/variáveis de interesse dos concreteObserver
- Envia notificação para os observadores quando seu estado muda

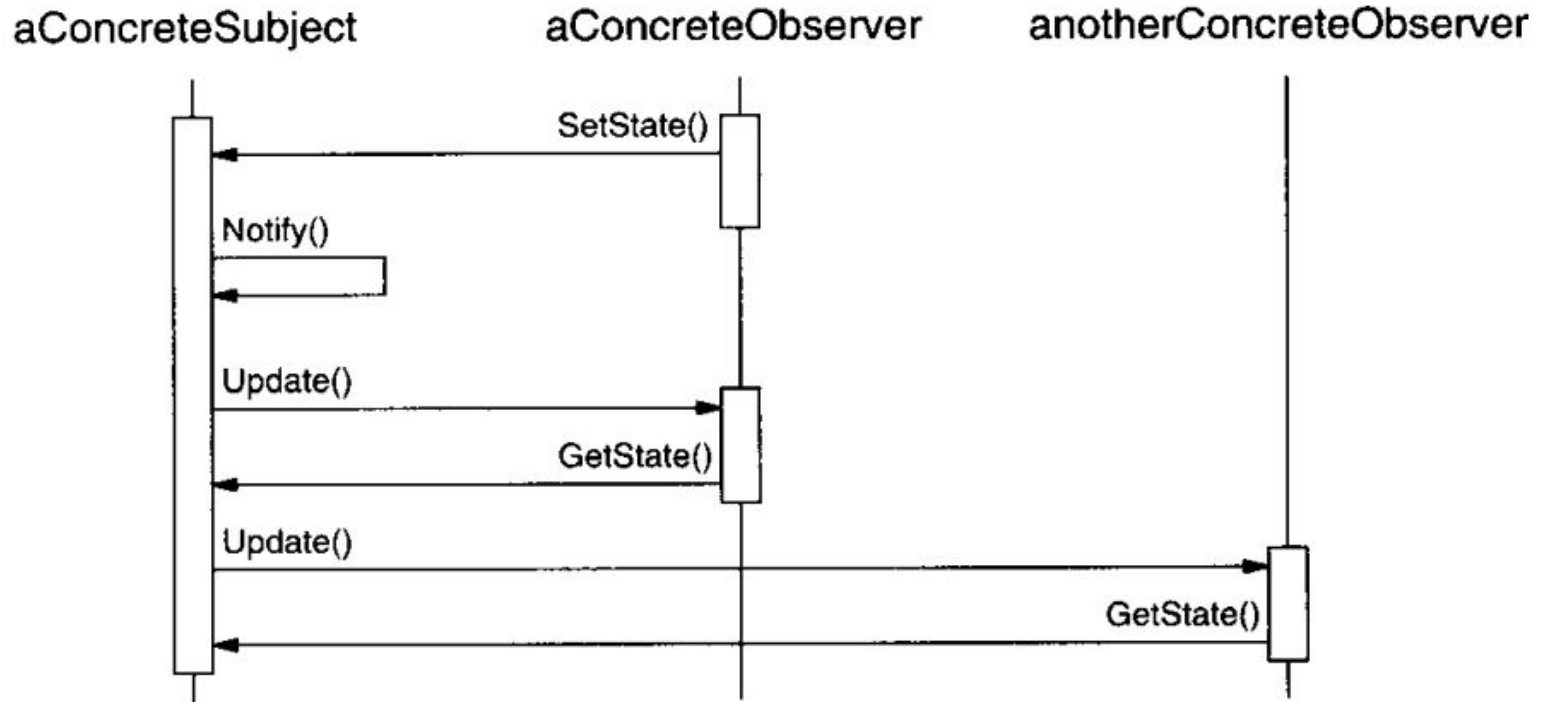
- **ConcreteObserver**

- Possui referência para o objeto concreteSubject
- Implementa interface de atualização do observer para manter sincronia com os estados do concreteSubject

Observer - Colaboração

- ConcreteSubject notifica seus observadores sempre que ocorre uma mudança de estado
 - Inconsistência entre o estado dele e dos observadores
- Após notificação, o concreteObserver consulta o concreteSubject (subject) para sincronização dos estados

Observer - Colaboração



Service Locator

- Encapsula o processo de busca de dependências por meio de uma abstração
- Há um centralizador (service locator) que retorna o que foi solicitado
- Pattern x anti-pattern
- Gerência de dependências do GetX usa algo similar



```
//simulação apenas
```

```
main(){
```

```
    ServiceLocator sl = ServiceLocator();
```

```
    sl.register(Pessoa());
```

```
    var pessoa = sl.find<Pessoa>();
```

```
    Pessoa pessoa = sl.find();
```

```
}
```

O que são estados?

- Em POO, os dados armazenados dentro de um objeto representam o seu estado
- Estado do sistema ou aplicação é o conjunto de valores de todos os objetos em um dado momento
- Métodos agem alterando os estados dos objetos



Nome = Maria
Saldo = R\$ 987.437,86
Idade = 29



Nome = José
Saldo = R\$ 674.137,96
Idade = 48



Nome = Francisco
Saldo = R\$ 897.882,09
Idade = 31



Nome = Maria
Saldo = R\$ 817.221,07
Idade = 33



Nome = José
Saldo = R\$ 919.233,45
Idade = 52



Nome = Francisco
Saldo = R\$ 627.982,02
Idade = 35

Ephemeral state x app state

- Estado - Todos os dados que você usa para reconstruir a UI a todo momento
- Dois tipos de estados gerenciáveis pelo dev
 - Ephemeral state
 - App state

Ephemeral state

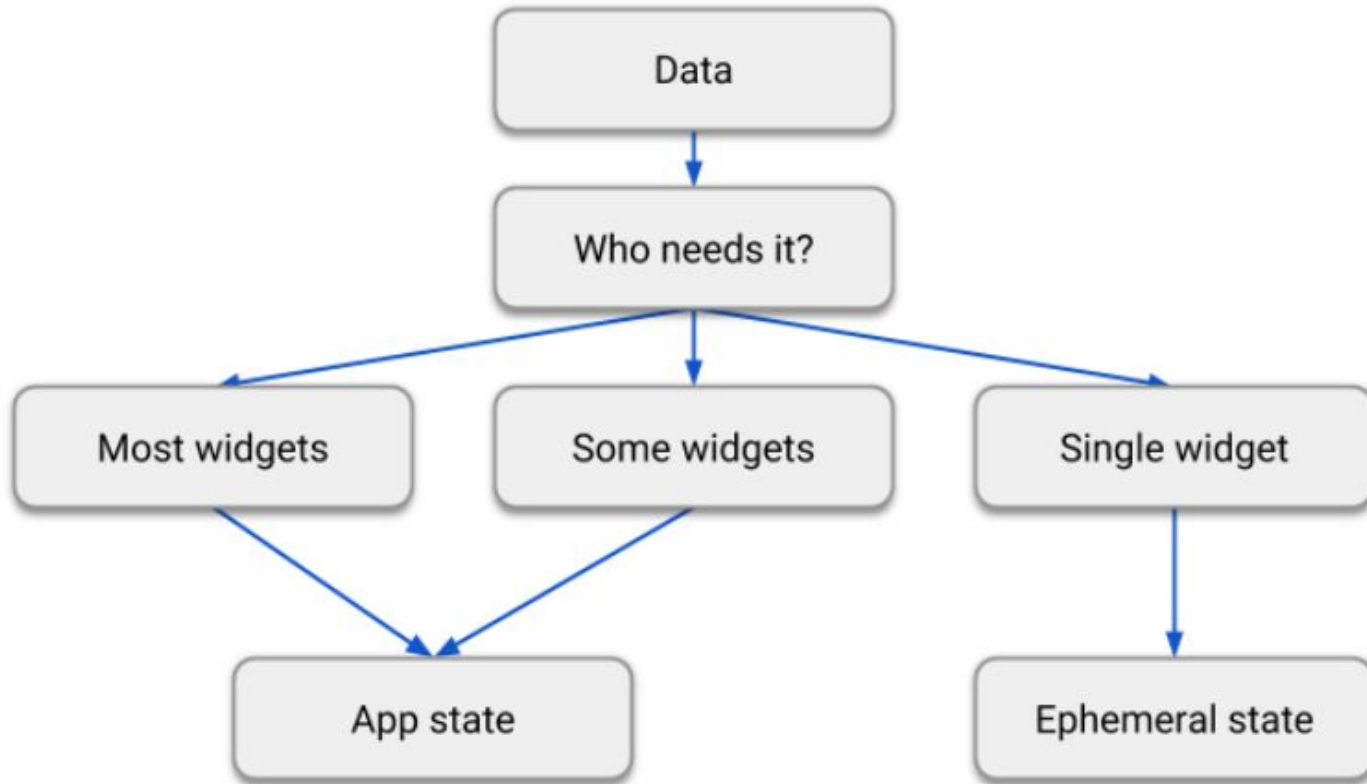
- Ephemeral state ou local é aquele que você pode conter em um único widget
 - `SetState()`
 - Index atual de uma `PageView`
 - Porcentagem de um upload
 - Index de uma aba selecionada em um `BottomNavigationBar`
- Outros widgets, normalmente, não precisam de acessá-los
- Não há necessidade de serializá-los para o HD/SSD
- Não muda de maneira complexa
- `Stateful Widget` é suficiente



```
class MyHomepage extends StatefulWidget {  
  const MyHomepage({super.key});  
  
  @override  
  State<MyHomepage> createState() => _MyHomepageState();  
}  
  
class _MyHomepageState extends State<MyHomepage> {  
  int _index = 0;  
  
  @override  
  Widget build(BuildContext context) {  
    return BottomNavigationBar(  
      currentIndex: _index,  
      onTap: (newIndex) {  
        setState(() {  
          _index = newIndex;  
        });  
      },  
      // ... items ...  
    );  
  }  
}
```

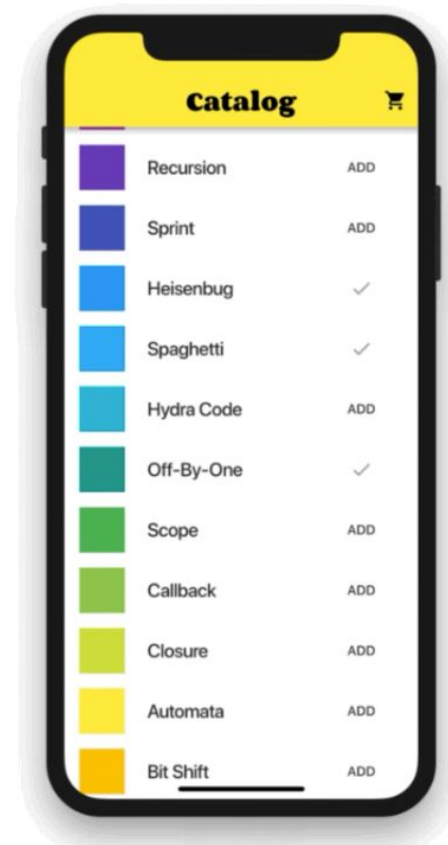
App state

- App state ou estado da aplicação é aquele que você pode compartilhar entre vários widgets
 - Informações de login
 - Carrinho de compras
 - Lista de favoritos
 - Lista telefônica
- Usa opções de terceiros/libs
- Pode usar apenas estado ephemeral/local, porém é improdutivo e de difícil manutenção



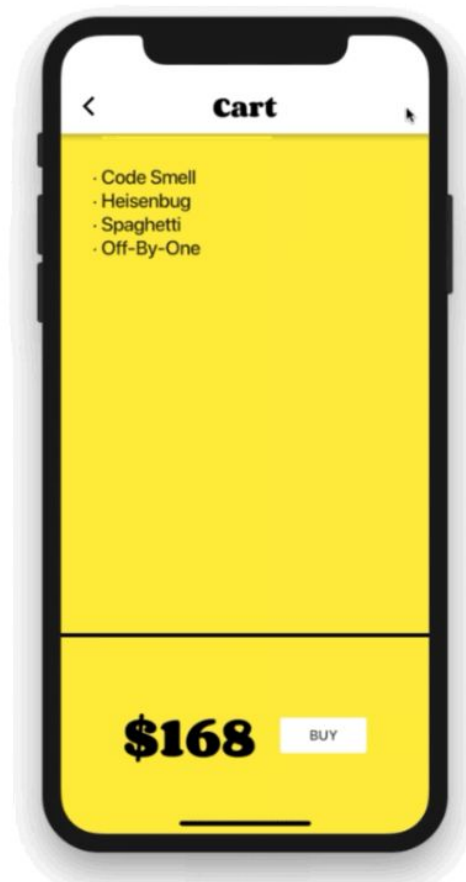
Gerência simples de estado

- Duas telas separadas
 - Catálogo de cores
 - Carrinho
 - MyCatalog e MyCart, respectivamente



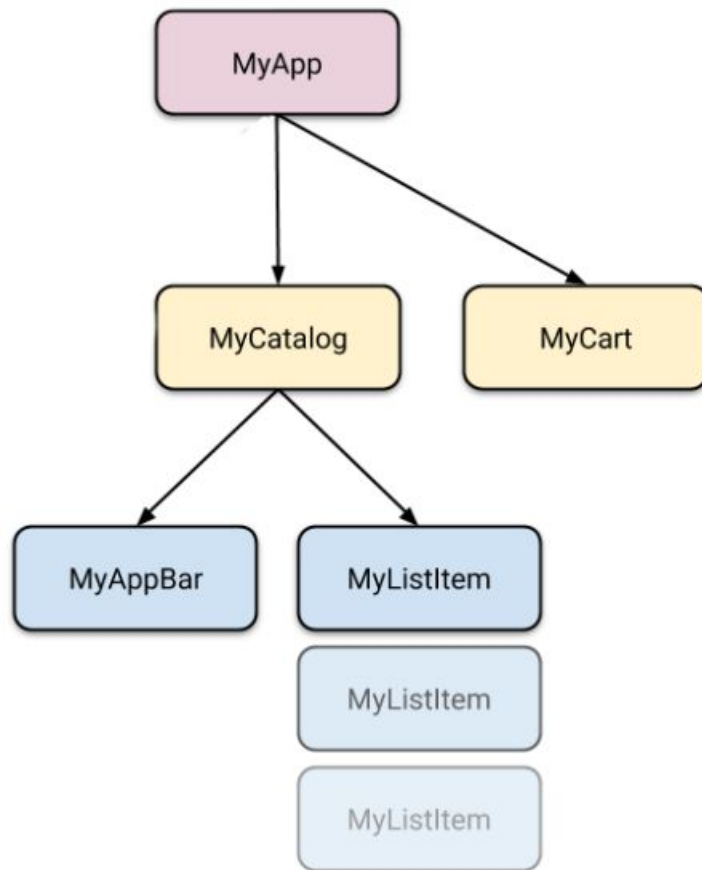
Gerência simples de estado

- Duas telas separadas
 - Catálogo de cores
 - Carrinho
 - MyCatalog e MyCart, respectivamente



Gerência simples de estado

- Onde colocar o estado do carrinho?
 - Outros widgets podem querer acessá-lo



Lifting state up

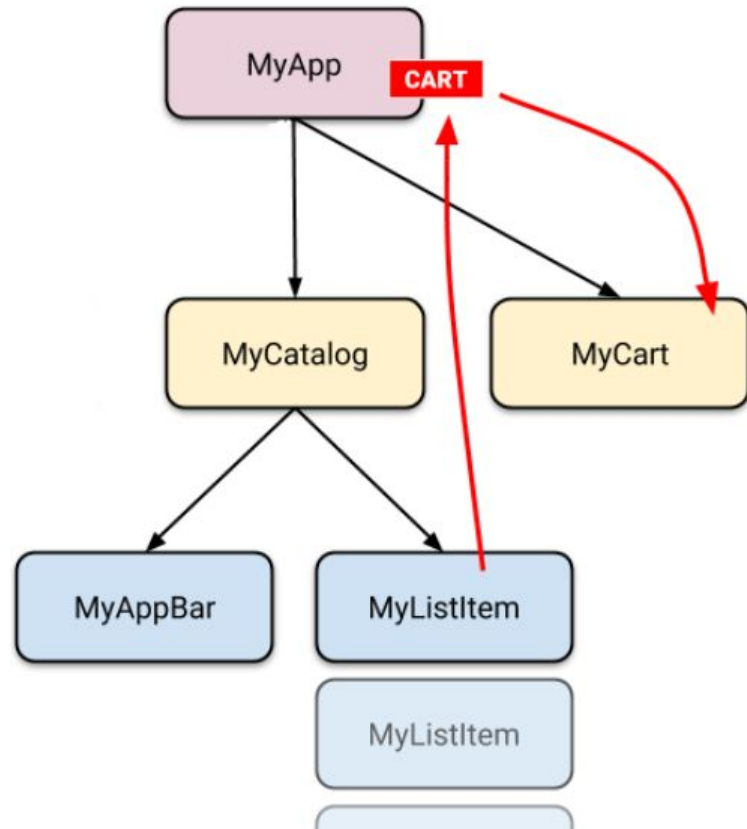
- Estado do carrinho o mais alto possível
 - Estado acima dos widgets que o utiliza

Lifting state up

- Flutter constrói um novo widget a cada mudança no estado
- `MyCart(contents)`
- Só é possível construir novos widgets no build dos widgets pais
 - Para alterar o contents, ele precisa estar no pai do `MyCart`

Lifting state up

- Estado de cart sobe para MyApp
- Se cart muda, MyListItem e MyCart são reconstruídos recebendo o cart lá de cima
- Widgets são imutáveis porque são substituídos por novos



Acessando o estado

- Descida de uma função de callback
- InheritedWidget, InheritedNotifier, InheritedModel (Provider)


```

class _DropDownButtonExampleState extends State<DropDownButtonExample> {
  String dropdownValue = list.first;

  void callbackDrop (String? value) {
    setState(() {
      dropdownValue = value!;
    });
  },

  @override
  Widget build(BuildContext context) {
    return DropdownButton<String>(
      value: dropdownValue,
      icon: const Icon(Icons.arrow_downward),
      elevation: 16,
      style: const TextStyle(color: Colors.deepPurple),
      underline: Container(
        height: 2,
        color: Colors.deepPurpleAccent,
      ),
      onChanged: callbackDrop,
      items: list.map<DropDownMenuItem<String>>((String value) {
        return DropdownMenuItem<String>(
          value: value,
          child: Text(value),
        );
      }).toList(),
    );
  }
}

```



Gerenciadores disponíveis

- Provider
- Mobx
- Bloc
- Getx
- Entre outros

Getx

- Gerenciador de estado
- Gerenciador de rotas/navegação
- Injetor de dependência
- Entre outros



GetX

State Manager | Navigation Manager
Dependencies Manager

Fast, Stable, Extra-light and Powerful Flutter Framework



Flutter made easy

```
// Easy navigation
Get.to(Home());

// Snackbars/dialogs/bottomsheets with no context
Get.snackbar('Hi', 'Message');
Get.defaultDialog(title: "I am a dialog");

// Easiest state manager
var count = 0.obs;
Obx(() => Text(count.string)); // Text() will be updated when count changes

// Access data from one instance in another screen easily.
Get.put(Instance());
Instance inst = Get.find();
Text(inst.name);

// Easy key/value storage
GetStorage box = GetStorage();
box.write('key', 'value');
box.read('key'); // out: value

// Easy internationalization
Text('Hello World'.tr);
Get.changeLocale(Locale('pt')); // out: Text('Olá mundo');

// Easy change theme:
Get.changeTheme(ThemeData.dark());

// Easy validators:
GetUtils.isEmail('abc@gmail.com') ? validate() : errorMessage();
```



getx flutter



Videos

Tutorial

Example

Documentation

Imagens

GitHub

Clean Architecture

Arguments in



Todos os fi

Aproximadamente 274.000 resultados (0,25 segundos)



pub.dev

<https://pub.dev/packages/get> · Traduzir esta página

get | Flutter Package - Pub.dev

GetX is an extra-light and powerful solution for **Flutter**. It combines high-performance state management, intelligent dependency injection, and route ...

Example · [Getx.site](#) · [5.0.0-release-candidate-4](#) · [Versions](#)

Você já visitou esta página várias vezes. Última visita: 01/08/23



medium.com

<https://kheronn-machado.medium.com/flutter-e-getx...>

Flutter e GetX — Criando uma aplicação para ...

29 de jul. de 2020 — **Flutter** e **GetX** — Criando uma aplicação para compartilhamento de ... O

GetX que entre outras coisas facilita o gerenciamento de rotas e ...



github.io

<https://chornthorn.github.io/getx...> · Traduzir esta página

Flutter Getx Documentation

Getx is a library that helps you to manage your app state, reactive programming and User

get 4.6.5

Published 14 months ago • [getx.site](#) Dart 3 compatible • Latest: 4.6.5 / Prerelease: 5.0.0-release-candidate-4

[SDK](#) [FLUTTER](#) [PLATFORM](#) [ANDROID](#) [IOS](#) [LINUX](#) [MACOS](#) [WEB](#) [WINDOWS](#)

 12.5K

[Readme](#) [Changelog](#) [Example](#) [Installing](#) [Versions](#) [Scores](#)

Use this package as a library

Depend on it

Run this command:

With Flutter:

```
$ flutter pub add get
```

This will add a line like this to your package's pubspec.yaml (and run an implicit `flutter pub get`):

```
dependencies:  
  get: ^4.6.5
```

Alternatively, your editor might support `flutter pub get`. Check the docs for your editor to learn more.

12572 120
LIKES PUB POINTS

Publisher

[getx.site](#)

Metadata

Open
screens/snackbars/
s without context, m
states and inject
dependencies easily
GetX.

[Repository \(GitHub\)](#)

[View/report issues](#)

Documentation

[API reference](#)

Uso inicial

- Step 1: Add "Get" before your MaterialApp, turning it into GetMaterialApp

```
void main() => runApp(GetMaterialApp(home: Home()));
```

- Step 2: Create your business logic class and place all variables, methods and controllers inside it. You can make any variable observable using a simple ".obs".

```
class Controller extends GetxController{  
  var count = 0.obs;  
  increment() => count++;  
}
```



- Step 3: Create your View, use StatelessWidget and save some RAM, with Get you may no longer need to use StatefulWidget.

Uso inicial

```
class Home extends StatelessWidget {  
  
  @override  
  Widget build(context) {  
  
    // Instantiate your class using Get.put() to make it available for all "child" routes to  
    final Controller c = Get.put(Controller());  
  
    return Scaffold(  
      // Use Obx(()=> to update Text() whenever count is changed.  
      appBar: AppBar(title: Obx(() => Text("Clicks: ${c.count}"))),  
  
      // Replace the 8 lines Navigator.push by a simple Get.to(). You don't need context  
      body: Center(child: ElevatedButton(  
        child: Text("Go to Other"), onPressed: () => Get.to(Other()))),  
      floatingActionButton:  
        FloatingActionButton(child: Icon(Icons.add), onPressed: c.increment));  
    )  
  }  
}  
  
class Other extends StatelessWidget {  
  // You can ask Get to find a Controller that is being used by another page and redirect you  
  final Controller c = Get.find();  
  
  @override  
  Widget build(context){  
    // Access the updated count variable  
    return Scaffold(body: Center(child: Text("${c.count}")));  
  }  
}
```


Resolvendo os problemas do setState

- ★ Chama o método build inteiro
- ★ Variável única do widget
 - Não permite outro widget modificar a variável
- ★ Lógica de negócios e view fortemente acopladas

Os três pilares

- Gerência de dependências
 - Dependência disponível em todo o app
 - Binding - Liga controller à view
- Gerência de estado
 - GetX tem duas (02) categorias de gerenciadores de estado: simples (GetBuilder) e o reativo (GetX/Obx)
- Gerência de rotas/navegação
 - Sem contexto

Os três pilares - Gerência de dependências

Get has a simple and powerful dependency manager that allows you to retrieve the same class as your Bloc or Controller with just 1 lines of code, no Provider context, no inheritedWidget:

```
Controller controller = Get.put(Controller()); // Rather Controller controller = Controller
```

Imagine that you have navigated through numerous routes, and you need data that was left behind in your controller, you would need a state manager combined with the Provider or Get_it, correct? Not with Get. You just need to ask Get to "find" for your controller, you don't need any additional dependencies:

```
Controller controller = Get.find();  
//Yes, it looks like Magic, Get will find your controller, and will deliver it to you. You c
```

And then you will be able to recover your controller data that was obtained back there:

```
Text(controller.textFromApi);
```

Os três pilares - Gerência de dependências

- Criação de controladores
 - Get.put();
 - Insere automaticamente na memória
 - Permanent: true
 - Permite a instância do controlador ficar ativa durante todo a execução do app
 - Previne o dispose
 - tag: 'unique'
 - Permite identificar unicamente objetos do mesmo tipo

```
Get.put<SomeClass>(SomeClass());  
Get.put<LoginController>(LoginController(), permanent: true);  
Get.put<ListItemController>(ListItemController, tag: "some unique string");
```

Os três pilares - Gerência de dependências

- Criação de controladores
 - `Get.lazyPut()`;
 - Insere em um local especial da memória
 - Após o primeiro `find<T>()`, vai para o local padrão da memória
 - Ideal quando vários controllers vão ser inicializados
 - Evita sobrecarga

```
Get.lazyPut<ApiMock>(() => ApiMock());
```

```
Get.lazyPut<FirebaseAuth>(  
  () {  
    // ... some logic if needed  
    return FirebaseAuth();  
  },  
  tag: Math.random().toString(),  
  fenix: true  
)
```

Os três pilares - Gerência de dependências

- Busca de controladores

- Acessa os dados do controller em qualquer local/tela

```
final controller = Get.find<Controller>();  
// OR  
Controller controller = Get.find();  
  
Text(controller.textFromApi);
```

- Remoção de controladores

- Get faz isso automaticamente
- smartManagement

```
Get.delete<Controller>();
```

Os três pilares - Gerência de dependências

- Binding

```
class HomeBinding implements Bindings {  
    @override  
    void dependencies() {  
        Get.lazyPut<HomeController>(() => HomeController());  
        Get.put<Service>(() => Api());  
    }  
}  
  
class DetailsBinding implements Bindings {  
    @override  
    void dependencies() {  
        Get.lazyPut<DetailsController>(() => DetailsController());  
    }  
}
```

- dependencies() é responsável por inicializar todos os controllers que uma tela precisará

Os três pilares - Gerência de dependências

- Binding
 - Associa os controladores às telas
 - Deixa mais organizada a injeção de dependências

```
getPages: [  
  GetPage(  
    name: '/',  
    page: () => HomeView(),  
    binding: HomeBinding(),  
  ),  
  GetPage(  
    name: '/details',  
    page: () => DetailsView(),  
    binding: DetailsBinding(),  
  ),  
];
```

→ `Get.lazyPut<HomeController>(() => HomeController());`
→ `Get.put<Service>(() => Api());`

→ `Get.lazyPut<DetailsController>(() => DetailsController());`

Os três pilares - Gerência de dependências

- SmartManagement

- por padrão, o GetX “dispose” os controladores não utilizados
- A classe SmartManagement configura o modo de dispose dos controladores
 - SmartManagement.full
 - Modo padrão
 - “Dispose” as classes que não estão sendo usadas e não foram marcadas como permanent
 - SmartManagement.onlyBuilder
 - Apenas controladores declarados no ‘init:’ ou em um binding com lazyPut() vão ser disposed
 - Get.put(), Get.putAsync() ou outro modo estão protegidos do dispose
 - SmartManagement.keepFactory
 - Assim como o SmartManagement.full, ele dá dispose dos controladores que não estão em uso, porém mantém as factories
 - Isso dá a possibilidade de recriar a dependência/controller se necessitar da instância novamente

Os três pilares - Gerência de dependências

- SmartManagement

```
void main () {  
    runApp(  
        GetMaterialApp(  
            smartManagement: SmartManagement.onlyBuilder //here  
            home: Home(),  
        )  
    )  
}
```

Os três pilares - Gerência de estado - simples - GetBuilder

- O init é responsável por sinalizar qual o controlador daquele trecho específico
 - Controller() - Inicia um novo controller específico para aquele trecho
 - Passa apenas a variável com o controlador
 - Um find<T>() anterior busca o controlador do tipo T
 - Exige um Get.put() ou Get.lazyPut()
 - Instancia o objeto e coloca na variável
 - Não necessita de Get.put() ou Get.lazyPut()
 - A ausência do init faz a procura ser na memória
 - Busca um controlador do tipo especificado
 - Exige um Get.put() ou Get.lazyPut()
 - Não necessita de find<T>()
 - Binding é opcional

Os três pilares - Gerência de estado - simples

```
GetBuilder<Controller>(  
  init: Controller(),  
  builder: (value) => Text(  
    '${value.counter}',  
  ),  
),
```


```
var controller = Get.find<Controller>();  
// ou var controller = Controller()  
GetBuilder<Controller>(  
  init: controller,  
  builder: (value) => Text(  
    '${value.counter}',  
  ),  
),
```

```
Get.put(Controller());  
GetBuilder<Controller>(  
  builder: (value) => Text(  
    '${value.counter}',  
  ),  
),
```


Os três pilares - Gerência de estado - reativo GetX

- O init é responsável por sinalizar qual o controlador daquele trecho específico
 - Controller() - Inicia um novo controller específico para aquele trecho
 - Passa apenas a variável com o controlador
 - Um find<T>() anterior busca o controlador do tipo T
 - Exige um Get.put() ou Get.lazyPut()
 - Instancia o objeto e coloca na variável
 - Não necessita de Get.put() ou Get.lazyPut()
 - A ausência do init faz a procura ser na memória
 - Busca um controlador do tipo especificado
 - Exige um Get.put() ou Get.lazyPut()
 - Não necessita de find<T>()
 - Binding é opcional


Os três pilares - Gerência de estado - reativo GetX



```
GetX<Controller>(  
  init: Controller(),  
  builder: (value) => Text(  
    '${value.counter}',  
  ),  
)
```



```
var controller = Get.find<Controller>();  
// ou var controller = Controller()  
GetX<Controller>(  
  init: controller,  
  builder: (value) => Text(  
    '${value.counter}',  
  ),  
)
```

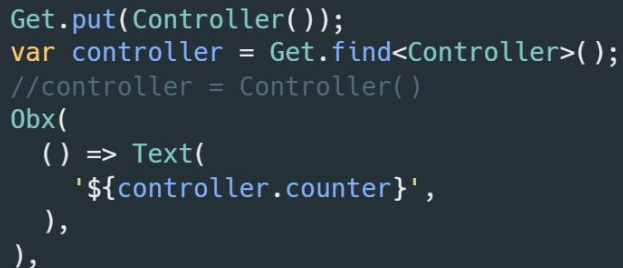


```
Get.put(Controller());  
GetX<Controller>(  
  builder: (value) => Text(  
    '${value.counter}',  
  ),  
)
```

Os três pilares - Gerência de estado - reativo Obx

- Não possui init
 - Apenas usa internamente a variável com o controlador
 - Um `find<T>()` anterior busca o controlador do tipo T
 - Exige um `Get.put()` ou `Get.lazyPut()`
 - Instancia o objeto e coloca na variável
 - Não necessita de `Get.put()` ou `Get.lazyPut()`
- GetView
 - Permite achar o controlador sem necessidade do `find<T>()`

Os três pilares - Gerência de estado - reativo Obx



```
Get.put(Controller());  
var controller = Get.find<Controller>();  
//controller = Controller()  
Obx(  
  () => Text(  
    '${controller.counter}',  
  ),  
),
```


Os três pilares - Gerência de estado - reativo

- Gerenciador reativo (Obx e GetX)
 - Definição de variáveis reativas

1 - The first is using `Rx{Type}` .

```
// initial value is recommended, but not mandatory
final name = RxString('');
final isLoggedIn = RxBool(false);
final count = RxInt(0);
final balance = RxDouble(0.0);
final items = RxList<String>([]);
final myMap = RxMap<String, int>({});
```

2 - The second is to use `Rx` and use Darts Generics, `Rx<Type>`

```
final name = Rx<String>('');
final isLoggedIn = Rx<Bool>(false);
final count = Rx<Int>(0);
final balance = Rx<Double>(0.0);
final number = Rx<Num>(0);
final items = Rx<List<String>>([]);
final myMap = Rx<Map<String, int>>({});

// Custom classes - it can be any class, literally
final user = Rx<User>();
```

3 - The third, more practical, easier and preferred approach, just add `.obs`

```
final name = ''.obs;
final isLoggedIn = false.obs;
final count = 0.obs;
final balance = 0.0.obs;
final number = 0.obs;
final items = <String>[].obs;
final myMap = <String, int>{}.obs;
```

```
// Custom classes - it can be any class, literally
final user = User().obs;
```

Os três pilares - Gerência de estado - reativo

- O valor de uma variável reativa é obtido com o `.value`
- A variável reativa engloba/encapsula uma variável de tipo primitivo ou classe criada
 - `var name = 'Flutter'.obs;`
 - `print(name);` ❌
 - `print(name.value);`
- `RxList<T>` é um caso especial
 - `var colors = ['Azul','Amarelo','Verde'].obs`
 - `print(colors.value);`
 - Funciona, mas não recomendado
 - `print(colors.toList());`

Os três pilares - Gerência de estado - reativo

- Classes próprias observáveis precisam de atenção quando seus atributos não são RX
- A classe inteira é observável no controlador, mas os atributos não são (name e age)

```
// on the model file
// we are going to make the entire class observable instead of each attribute
class User() {
    User({this.name = '', this.age = 0});
    String name;
    int age;
}

// on the controller file
final user = User().obs;
```

Os três pilares - Gerência de estado - reativo

- A reatividade é percebida se substituir o objeto inteiro

```
user(User(name: 'Jean', age: '63'))  
//ou  
user.value = User(name: 'Jean', age: '63')
```

- A reatividade também é percebida se utilizar o método update

```
user.update( (user) { // o parâmetro user é o próprio objeto user.value  
    user.name = 'Jonny';  
    user.age = 18;  
});
```

Controlador GetX - ciclo de vida

- OnInit
 - similar ao initState() do stateful widget
- onClose
 - similar ao dispose() do stateful widget
- GetX permite usar stateless widget
 - stateful Widget torna-se opcional

Workers

- Funcionam observando eventos e disparando callbacks quando adequado

```
/// Called every time `count1` changes.  
ever(count1, (_) => print("$_ has been changed"));  
  
/// Called only first time the variable $_ is changed  
once(count1, (_) => print("$_ was changed once"));  
  
/// Anti DDos - Called every time the user stops typing for 1 second, for example.  
debounce(count1, (_) => print("debouce$_"), time: Duration(seconds: 1));  
  
/// Ignore all changes within 1 second.  
interval(count1, (_) => print("interval $_"), time: Duration(seconds: 1));
```

Workers

- **ever**
 - Chamado toda vez que a variável Rx emite um novo valor
- **everAll**
 - Igual ao ever, porém verifica uma lista de variáveis Rx
- **once**
 - Chamado apenas na primeira vez que a variável Rx muda seu valor
- **debounce**
 - útil em funções de busca, pois garante que a requisição vai ser enviada após o fim da digitação. Evita que cada letra digitada faça uma nova busca
- **interval**
 - Ignora todas as ações do usuário por um intervalo de tempo determinado

Os três pilares - Navegação - Projeto roteamento

Add "Get" before your MaterialApp, turning it into GetMaterialApp

```
GetMaterialApp( // Before: MaterialApp(  
  home: MyHome(),  
)
```

Navigate to a new screen:

To close snackbars, dialogs, bottomsheets, or anything you would normally close with Navigator.pop(context);

```
Get.to(NextScreen());
```

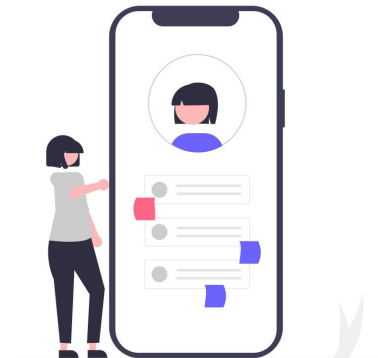
```
Get.back();
```

To go to the next screen and no option to go back to the previous screen (for use in SplashScreens, login screens, etc.)

```
Get.off(NextScreen());
```


União dos três pilares: Projeto **Phonebook**

1. Gerência de dependências
 2. Gerência de estado
 3. Gerência de rotas
- Lista telefônica
 - CRUD de contatos
 - Contagem de contatos



Por que GetX?

1. Compatibilidade

- a. Pacote único

2. Menos verbosidade (boilerplate)

- a. `Navigator.of(context).push (context, builder [...]) → Get.to(Home())`
- b. Desenvolvimento simplificado

3. Facilidade sem preocupação com desempenho

- a. SmartManagement (similar ao Garbage collector do Java)

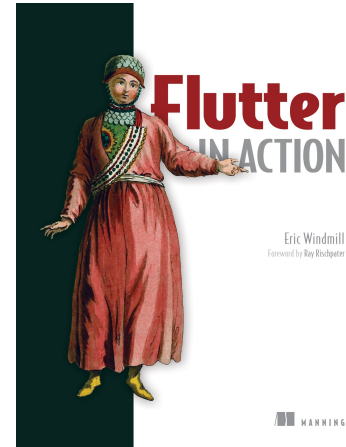
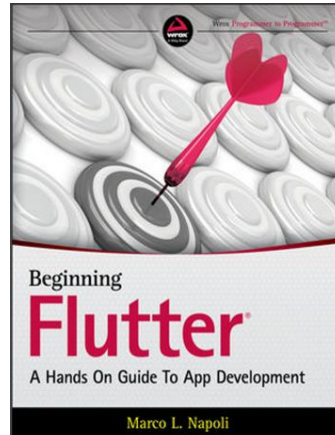
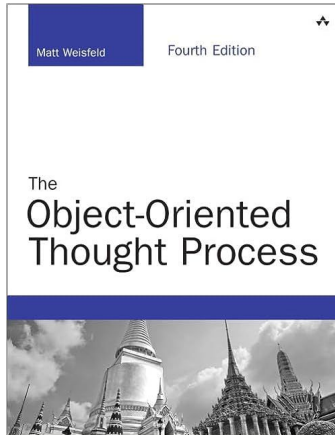
4. Desacoplamento

- a. Não necessita de contexto

Finalizando

- Percebeu algum erro? Sugestões? Melhorias? Discord!

Referências



Referências

1. <https://api.flutter.dev/index.html>
2. <https://docs.flutter.dev/data-and-backend/state-mgmt/intro>
3. <https://pub.dev/packages/get>
4. <https://www.youtube.com/watch?v=XeUiJJN0vsE&list=PLIBnlCol-g-d-J57QIz6Tx5xtUDGQdBFB&index=1>