Flutter Basic UI elements cheat Sheet

Animations

Animation is used to add visual effects to notify users that some part of the app's state has changed. It gives your app a polished look that witnesses the quality of your app to the end users.

- **Linear Animations**

The main three parts of our animations are the ticker to control our time, the controller to register our parameters like our duration, and then the values we want changed. Before our widget is rendered, in our initState, we can set our controller to its parameters, set its direction, and add a listener to reset our widgets state with every change.

```
class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> with SingleTickerProviderStateMixin {
  AnimationController controller;

  void initState() {
    super.initState();
    controller = AnimationController(
      duration: Duration(seconds: 1),
      vsync: this); // Links this controller to this widget so it won't run if the parent widge

    controller.forward();
    controller.addListener(() => setState(() {}));
  }
}
      body: Center(
        child: Opacity(
          opacity: controller.value,
          child: Container(width: 50, height: 50, color: Colors.red),
        ),
      ),
```

Animations

- ## AnimationController

AnimationController is what you normally use to play, pause, reverse, and stop animations. Instead of pure “tick” events, AnimationController tells us at which point of the animation we are, at any time. For example, are we halfway there? Are we 99% there? Have we completed the animation?

Normally, you take the AnimationController, maybe transform it with a Curve, put it through a Tween, and use it in one of the handy widgets like FadeTransition or TweenAnimationBuilder. But, for educational purposes, let’s not do that. Instead, we will directly call setState.

```
class _MyWidgetState extends State<MyWidget>
   with SingleTickerProviderStateMixin<MyWidget> {
  AnimationController _controller;  int i = 0;  @override
  void initState() {
    super.initState();
    _controller = AnimationController(
      vsync: this,
      duration: const Duration(seconds: 1),
    );
    _controller.addListener(_update);
    _controller.forward();
  }  void _update() {
    setState(() {
      i = (_controller.value * 299792458).round();
    });
  }  @override
  Widget build(BuildContext context) {
    return Text('$i m/s');
  }
}
```

Animations

- ## AnimatedBuilder

The AnimatedBuilder widget in Flutter is a powerful utility widget that is used for creating complex animations by rebuilding a part of the widget tree in response to changes in an animation's value. It is especially useful when you want to animate properties of child widgets that cannot be directly animated using widgets like AnimatedContainer or AnimatedOpacity

```
class AnimatedBuilderExample extends StatefulWidget {
@override
_AnimatedBuilderExampleState createState() => _AnimatedBuilderExampleState();
}
class _AnimatedBuilderExampleState extends State<AnimatedBuilderExample>
 with SingleTickerProviderStateMixin {
late AnimationController _controller;  late Animation<double> _animation;
@override
void initState() {   super.initState();  _controller = AnimationController(
 duration: Duration(seconds: 2), // Animation duration
 vsync: this,   );
 // Tween animation
 _animation = Tween<double>(begin: 50.0, end: 200.0).animate(_controller);
 _controller.forward();  } // Start the animation
@override
Widget build(BuildContext context) {
 return Scaffold(  appBar: AppBar(
  title: Text('AnimatedBuilder Demo'), ), // App bar title
 body: Center(   child: AnimatedBuilder(
  animation: _controller,   builder: (BuildContext context, Widget? child) {
   return Container(   width: _animation.value, // Animate the width
   height: _animation.value,   color: Colors.green,    child: Center(    child: Text(
    'Hello',    style: TextStyle(color: Colors.white), ),   ),   );  },  ),  ),  );
@override
void dispose() {  _controller.dispose(); // Dispose of the animation controller
 super.dispose(); } }
```

Flutter Basic UI elements cheat Sheet

Animations

- **Implicitly**

These widgets automatically animate changes to their properties. When you rebuild the widget with new property values, such as with a StatefulWidget's setState, the widget handles driving the animation from the previous value to the new value.

```
Widget build(BuildContext context) {
  return Scaffold(
    body: Column(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        children: <Widget>[
      SizedBox(height: 1),
      Row(
        mainAxisAlignment: MainAxisAlignment.spaceAround,
        children: <Widget>[
          Hero(tag: 'icon', child: Icon(Icons.add)),
        ]),
      Navbar()
  ]));
}
```

```
Widget build(BuildContext context) {
  return Scaffold(
    body: Column(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: <Widget>[
        SizedBox(height: 1),
        Row(
          mainAxisAlignment: MainAxisAlignment.spaceAround,
          children: <Widget>[
            SizedBox(width: 1),
            Hero(
              tag: 'icon',
              child: Icon(Icons.add, color: Colors.red, size: 75)),
          ]),
        Navbar()
      ]),
  );
}
```

Animations

- ## Curved Animations

Instead of boring linear animations we can use different curved variations to control exactly how our controller changes. To do this we can user CurvedAnimation to create a kind of wrapper over our original controller. This wrapper will take its parent, our controller, and the curve we want to apply. When using a curved animation we can't have lower and upper bounds besides 0 and 1, so when we apply it we can just multiply its value

```
class _MyHomePageState extends State<MyHomePage> with SingleTickerProviderStateMixin {
  AnimationController controller;
  Animation animation;

  void initState() {
    super.initState();
    controller = AnimationController(
        duration: Duration(seconds: 1),
        vsync: this);
    animation = CurvedAnimation(parent: controller, curve: Curves.slowMiddle);

    controller.forward();
    animation.addListener(() => setState(() {}));

    controller.addStatusListener((status) {
      if (status == AnimationStatus.completed) controller.reverse(from: 400);
      else if (status == AnimationStatus.dismissed) controller.forward();
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Container(
          margin: EdgeInsets.only(bottom: animation.value * 400),
          width: 50,
          height: 50,
          color: Colors.red),
      ),
    );
  }
```

Animations

- **Tweens**

Instead of just working with number values, we can also work with ranges of other things, like colors, using tweens (short for between). Like with our curved animation, it will act as a wrapper for our controller or animation and we can set our color to its value.

```
class _MyHomePageState extends State<MyHomePage> with SingleTickerProviderStateMixin {
  AnimationController controller;
  Animation animation;
  Animation changeColor;

  void initState() {
    super.initState();
    controller =
        AnimationController(duration: Duration(seconds: 1), vsync: this);
    animation = CurvedAnimation(parent: controller, curve: Curves.slowMiddle);
    changeColor = ColorTween(begin: Colors.red, end: Colors.blue).animate(animation);
    controller.forward();
    animation.addListener(() => setState(() {}));

    controller.addStatusListener((status) {
      if (status == AnimationStatus.completed) controller.reverse(from: 400);
      else if (status == AnimationStatus.dismissed) controller.forward();
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Container(
          margin: EdgeInsets.only(bottom: animation.value * 400),
          width: 50,
          height: 50,
          color: changeColor.value),
      ),
    );
  }
```

Flutter Basic UI elements cheat Sheet

Animations

- **Explicit**

when in implicit animations, merely changing a value inside an AnimatedFoo or TweenAnimationBuilder widget triggered an animation? Well, explicit animations don't animate until "explicitly" being told to animate. And you tell them to animate and how to animate and "control" their animation using an AnimationController.

```
class _ExplicitAnimationsState extends State<ExplicitAnimations>
with SingleTickerProviderStateMixin {
late AnimationController _controller;
late final Animation<AlignmentGeometry> _alignAnimation;
late final Animation<double> _rotationAnimation;

@override
void initState() { super.initState(); _controller = AnimationController(
duration: const Duration(milliseconds: 800),vsync: this, )..repeat(reverse: true );

_alignAnimation = Tween<AlignmentGeometry>(
begin: Alignment.centerLeft, end: Alignment.centerRight,).animate(
CurvedAnimation( parent: _controller, curve: Curves.easeInOutCubic, ), );

_rotationAnimation = Tween<double>(begin: 0, end: 2).animate(
CurvedAnimation( parent: _controller, curve: Curves.easeInOutCubic,
), );  }

@override
void dispose() {    _controller.dispose();   super.dispose();   }

@override
Widget build(BuildContext context) { return BlurContainer(
containerHeight: 200, child: AlignTransition( alignment: _alignAnimation,
child: RotationTransition( turns: _rotationAnimation,
child: const Rectangle( color1: pink, color2: pinkDark, width: 50,
height: 50, ), ), ), ); } }
```

Flutter Basic UI elements cheat Sheet

Animations

- **Hero**

Hero animations are probably one of the easiest animations to do in Flutter and don't require much setup. Taking a look at the example above, we can see that the same app icon widget exists on both pages. All we need is a way to tell Flutter that both of them are linked.

We do this by **wrapping an element like an icon in a Hero widget**.

```
Hero(
  tag: "DemoTag",
  child: Icon(
    Icons.add,
    size: 70.0,
  ),
),
```

We supply it a **tag** to give this specific hero a name.

This is necessary because if we have multiple heroes on the same page, each hero knows where to fly to.

Now the app knows that there is a hero widget that wants to fly to the next page. Now all we need **is a place to fly to**.

All we need is a **Hero widget on the second page with the same tag**.

```
Hero(
  tag: "DemoTag",
  child: Icon(
    Icons.add,
    size: 150.0,
  ),
),
```

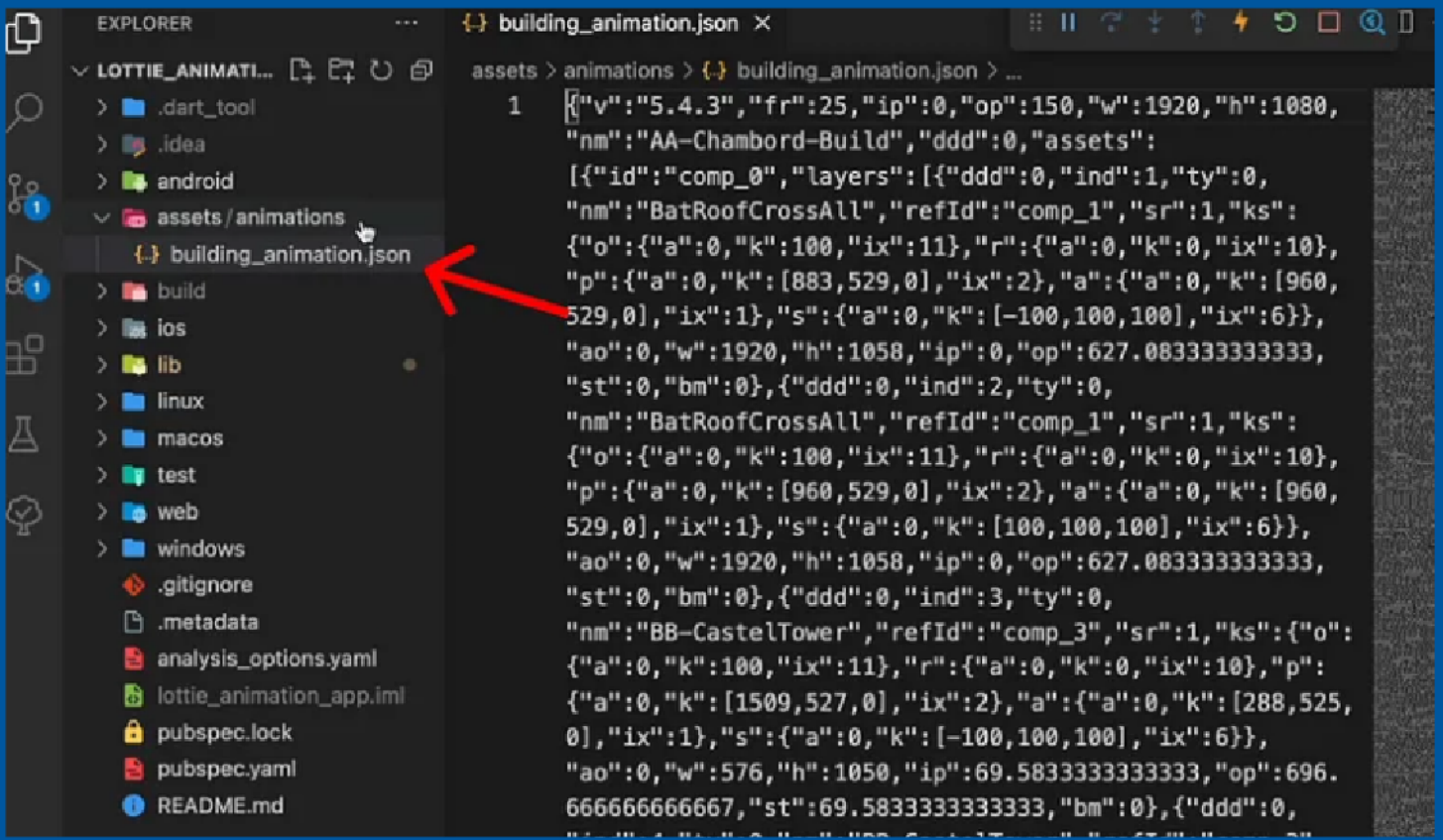
Flutter Basic UI elements cheat Sheet

Animations

- ### Lottie Animation

Lottie actually provides us with plenty of high quality animation assets which we can easily integrate or embed in our flutter apps. One of the main reasons for considering lottie animations is that we have entire control of the animation that is been rendered in the UI screen. Here let's understand this by adding a lottie animation in our flutter app and try to pause/play the animation based on button press event.

The lottie assets can be downloaded as a JSON file or as a gif or as a video. Let's now try to download any animation asset from lottie as a JSON file and add them inside the asset folder in the project root directory and similarly update the pubspec.yaml file by specifying the assets.



Flutter Basic UI elements cheat Sheet

Lottie Animation

Now let's move on to the coding side, First let's create a stateful widget class called MyHomePage and try to initialize the animation controller.

```
class MyHomePage extends StatefulWidget {
 const MyHomePage({super.key});

 @override
 State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> with
TickerProviderStateMixin
{
 late final AnimationController _controller;

 @override
 void initState() {
 super.initState();  _controller = AnimationController(vsync: this);  }

 @override
 void dispose() {  _controller.dispose();  super.dispose();  }

 @override
 Widget build(BuildContext context) {  return Scaffold(
 backgroundColor: Colors.blueGrey, body: Column(
 mainAxisAlignment: MainAxisAlignment.center,
 children: [ Lottie.asset(
 "assets/animations/building_animation.json",
 controller: _controller, onLoaded: (composition) {
 _controller.duration = composition.duration; }, ), ], ),

 );}  }
```

Flutter Basic UI elements cheat Sheet

Animations

- **Rive Animation**

An effective way of achieving an exceptional user experience for your app is through the integration of animations that captivate and engage users. Enters Rive(formerly Flare), a platform that brings vector design and animation together seamlessly. Rive allows the designers to create interactive graphics in either .riv format or a video format of their choice using their cloud renderer.

To add and integrate your preferred RIve animation into your apps, there are a few different methods that you could follow.

1. via URL: Use the RiveAnimation.network() method along with the link to the Rive animation.

```
RiveAnimation.network(
  'https://cdn.rive.app/animations/vehicles.riv',
)
```

2. via Asset Bundle : Make sure to add the downloaded Rive file to your assets folder and reference it in your pubspec.yaml file.

```
...

# To add assets to your application, add an assets section, like this:
assets:
    - assets/vehicles.riv
```

```
RiveAnimation.asset(
  'assets/vehicles.riv',
)
```

Flutter Basic UI elements cheat Sheet

Floating Action Buttons

floating action button (FAB) performs the primary, or most common, action on a screen. It appears in front of all screen content, typically as a circular shape with an icon in its center. FABs come in three types: small, regular, and large FAB. A small FAB is used for a secondary, supporting action, or in place of a default FAB on compact window sizes. A large FAB is useful when the layout calls for a clear and prominent primary action, and where a larger footprint would help the user engage. For example, when appearing in a medium window size.

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

void main() {
 runApp(const FloatingButtonDemo());  }

class FloatingButtonDemo extends StatefulWidget {
const FloatingButtonDemo({super.key});

@override
Widget build(BuildContext context) {
 return MaterialApp(  home: Scaffold(  appBar: AppBar(
 title: const Text('Welcome to Flutter'),  centerTitle: true,
 backgroundColor: const Color.fromRGBO(82, 170, 94, 1.0), ),

 floatingActionButton: FloatingActionButton(
 backgroundColor: const Color.fromRGBO(82, 170, 94, 1.0),
 tooltip: 'Increment',  onPressed: (){},
 child: const Icon(Icons.add, color: Colors.white, size: 28), ),  ) );  }
}
```

Flutter Basic UI elements cheat Sheet

AppBar

AppBar is usually the topmost component of the app (or sometimes the bottom-most), it contains the toolbar and some other common action buttons. As all the components in a flutter application are a widget or a combination of widgets. So AppBar is also a built-in class or widget in flutter which gives the functionality of the AppBar out of the box. The AppBar widget is based on Material Design and much of the information is already provided by other classes like MediaQuery, Scaffold as to where the content of the AppBar should be placed.

```
MaterialApp MyApp() {
return MaterialApp( home: Scaffold(
 appBar: AppBar(  title: const Text("WsCubeTech"),
  titleSpacing: 00.0,  centerTitle: true,  toolbarHeight: 60.2,
  toolbarOpacity: 0.8,
  shape: const RoundedRectangleBorder(
  borderRadius: BorderRadius.only(
   bottomRight: Radius.circular(25),
   bottomLeft: Radius.circular(25)),  ),  elevation: 0.00,
  backgroundColor: Colors.greenAccent[400], ), //AppBar
 body: const Center(  child: Text(  'WsCubeTech',
  style: TextStyle(fontSize: 24),  ), ),
 ), //Scaffold
  debugShowCheckedModeBanner: false, //Removing Debug Banner
);
}
```

Flutter Basic UI elements cheat Sheet

Tabbar

The tabs are mainly used for mobile navigation. The styling of tabs is different for different operating systems. For example, it is placed at the top of the screen in android devices while it is placed at the bottom in iOS devices.
Working with tabs is a common pattern in Android and iOS apps that follow the Material Design guidelines. Flutter provides a convenient way to create a tab layout. To add tabs to the app, we need to create a TabBar and TabBarView and attach them with the TabController. The controller will sync both so that we can have the behavior which we need.

```
class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
 return MaterialApp(  home: DefaultTabController(  length: 2,
 child: Scaffold(  appBar: AppBar(  title: Text('Flutter Tabs Demo'),
 bottom: TabBar(
tabs: [
 Tab(icon: Icon(Icons.contacts), text: "Tab 1"),
 Tab(icon: Icon(Icons.camera_alt), text: "Tab 2")
 ],  ),  ),
 body: TabBarView(  children: [
 FirstScreen(),  SecondScreen(),  ],  ),  ),
                        ),
                           ) ;
                                 }
                                   }
```

Flutter Basic UI elements cheat Sheet

Drawer

Drawers are very easy to implement as well as handy to balance different functionalities of your mobile application at the same time. Creating a drawer makes visiting different destinations in your app quite easy and manageable to a large extent, especially in the case of a complex application with many screens.
You can easily switch between different screens and perform tasks.

```
drawer: Drawer(
      child: ListView(
        padding: const EdgeInsets.all(0),
        children: [
          ListTile(    leading: const Icon(Icons.person),
           title: const Text(' My Profile '),
           onTap: () {
             Navigator.pop(context);     },      ),
         ListTile(
           leading: const Icon(Icons.book),
           title: const Text(' My Course '),   onTap: () {
             Navigator.pop(context);     },    ),
         ListTile(
           leading: const Icon(Icons.workspace_premium),
           title: const Text(' Go Premium '),
           onTap: () {
             Navigator.pop(context);      },    ),    ],   ),
    ), //Drawer
```

Flutter Basic UI elements cheat Sheet

Bottom Navigation Bar

The Bottom Navigation bar has become popular in the last few years for navigation between different UI. Many developers use bottom navigation because most of the app is available now using this widget for navigation between different screens.
The bottom navigation bar in Flutter can contain multiple items such as text labels, icons, or both. It allows the user to navigate between the top-level views of an app quickly. If we are using a larger screen, it is better to use a side navigation bar.

```
bottomNavigationBar: BottomNavigationBar(
    items: const <BottomNavigationBarItem>[
      BottomNavigationBarItem(
        icon: Icon(Icons.home),
        title: Text('Home'),
        backgroundColor: Colors.green          ),
      BottomNavigationBarItem(
        icon: Icon(Icons.search),          title: Text('Search'),
        backgroundColor: Colors.yellow          ),
      BottomNavigationBarItem(
        icon: Icon(Icons.person),          title: Text('Profile'),
        backgroundColor: Colors.blue,          ),        ],
    type: BottomNavigationBarType.shifting,
    currentIndex: _selectedIndex,
    selectedItemColor: Colors.black,
    iconSize: 40,        onTap: _onItemTapped,        elevation: 5
  ),
);
```