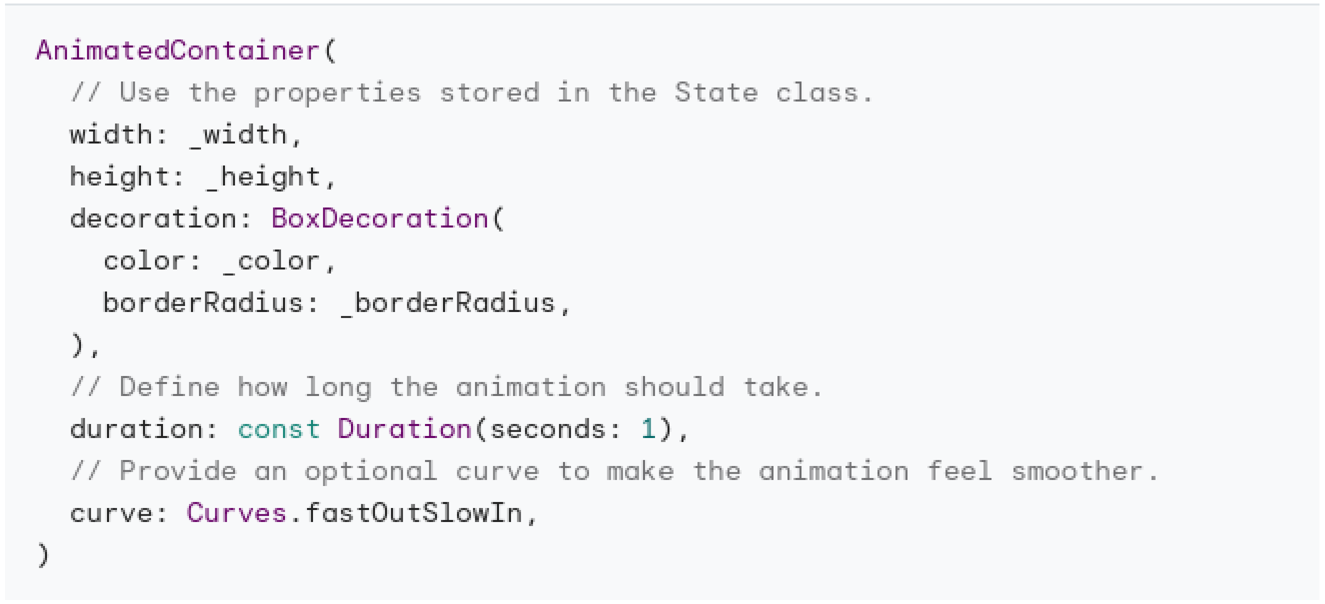
222301504 Cross Platform Mobile Application Development

Using Animated Container

- The Container class provides a convenient way to create a widget with specific properties: width, height, background color, padding, borders, and more.
- To animate these properties, Flutter provides the AnimatedContainer widget.
- Like the Container widget, AnimatedContainer allows you to define the width, height, background colors, and more. However, when the AnimatedContainer is rebuilt with new properties, it automatically animates between the old and new values. In Flutter, these types of animations are known as “implicit animations.”

- This recipe describes how to use an AnimatedContainer to animate the size, background color, and border radius when the user taps a button using the following steps:
- Create a StatefulWidget with default properties.
- Build an AnimatedContainer using the properties.
- Start the animation by rebuilding with new properties.

- The AnimatedContainer constructor has arguments called duration, curve, color, height, width, child, decoration, transform, and many others.

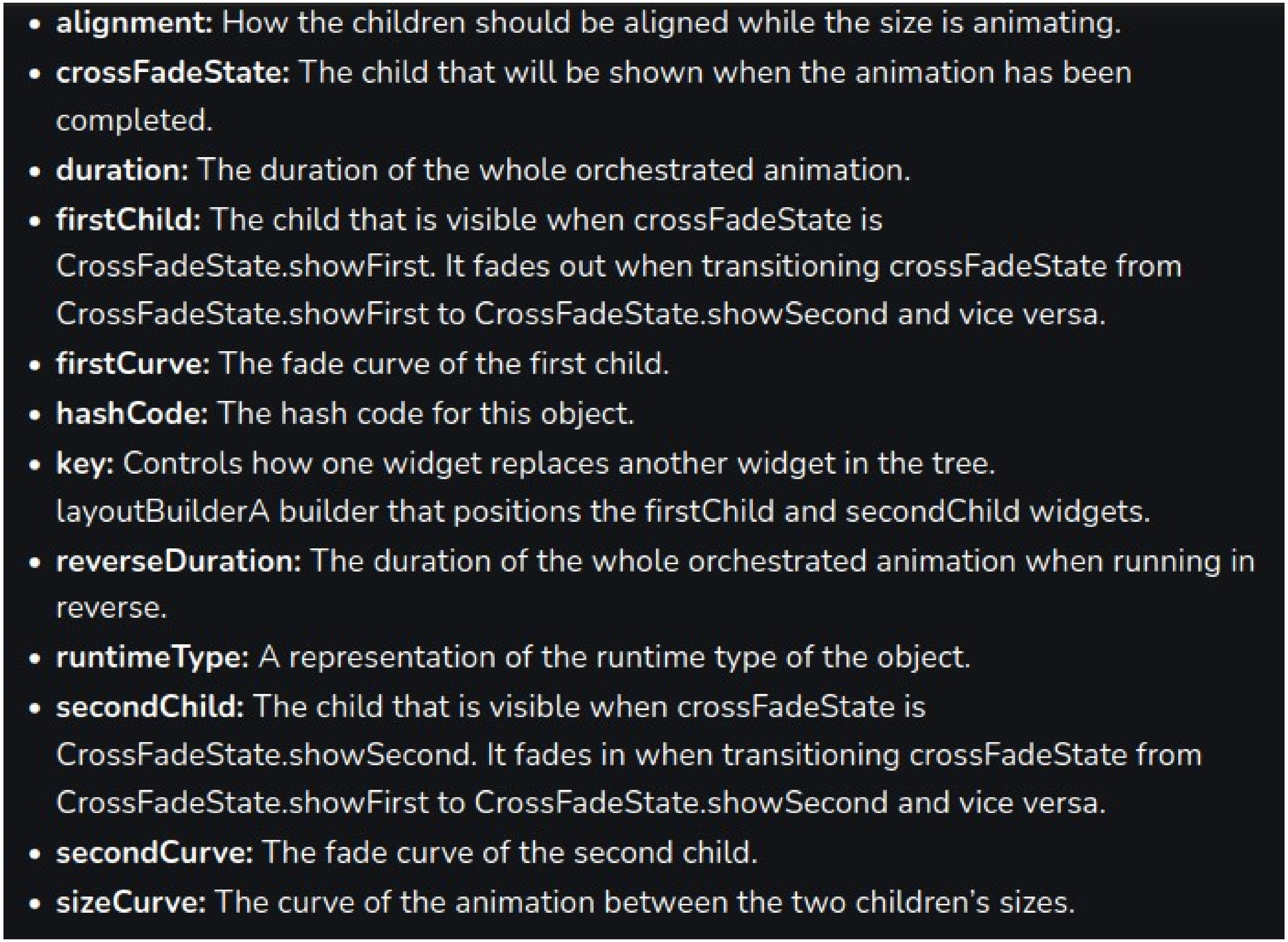


```
AnimatedContainer(
  // Use the properties stored in the State class.
  width: _width,
  height: _height,
  decoration: BoxDecoration(
    color: _color,
    borderRadius: _borderRadius,
  ),
  // Define how long the animation should take.
  duration: const Duration(seconds: 1),
  // Provide an optional curve to make the animation feel smoother.
  curve: Curves.fastOutSlowIn,
)
```

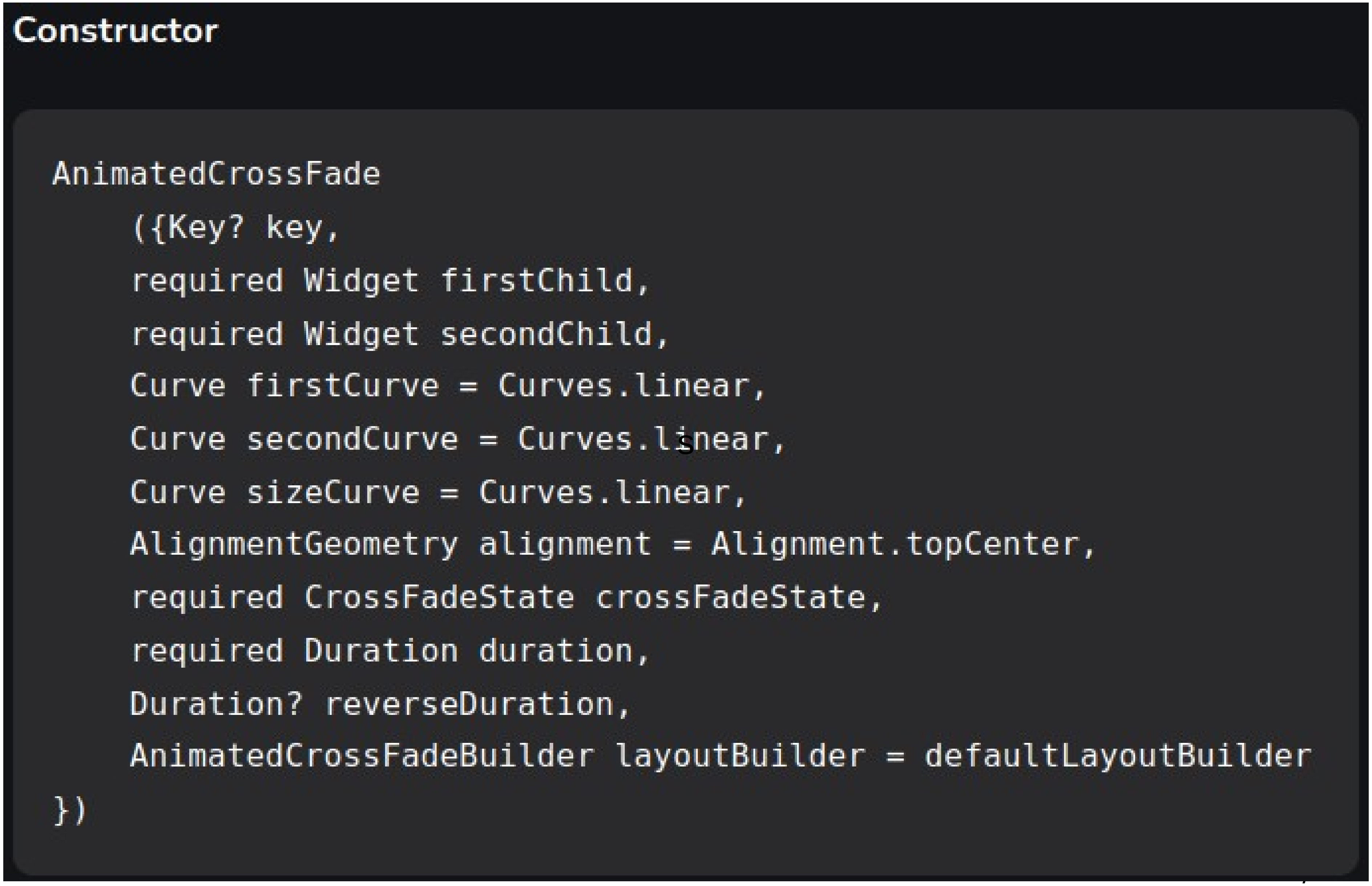
AnimatedCrossFade

- The AnimatedCrossFade widget provides a great cross-fade between two children widgets.
- TheAnimatedCrossFade constructor takes duration, firstChild, secondChild, crossFadeState, sizeCurve, and many other arguments.
- It is used when you need to give a fade kind of transition in between two widgets. It supports any kind of Flutter Widget like Text, Images, Icon as well as anything that is extended from the widget in Flutter.

Crossfade properties



- **alignment:** How the children should be aligned while the size is animating.
- **crossFadeState:** The child that will be shown when the animation has been completed.
- **duration:** The duration of the whole orchestrated animation.
- **firstChild:** The child that is visible when crossFadeState is CrossFadeState.showFirst. It fades out when transitioning crossFadeState from CrossFadeState.showFirst to CrossFadeState.showSecond and vice versa.
- **firstCurve:** The fade curve of the first child.
- **hashCode:** The hash code for this object.
- **key:** Controls how one widget replaces another widget in the tree. layoutBuilderA builder that positions the firstChild and secondChild widgets.
- **reverseDuration:** The duration of the whole orchestrated animation when running in reverse.
- **runtimeType:** A representation of the runtime type of the object.
- **secondChild:** The child that is visible when crossFadeState is CrossFadeState.showSecond. It fades in when transitioning crossFadeState from CrossFadeState.showFirst to CrossFadeState.showSecond and vice versa.
- **secondCurve:** The fade curve of the second child.
- **sizeCurve:** The curve of the animation between the two children's sizes.

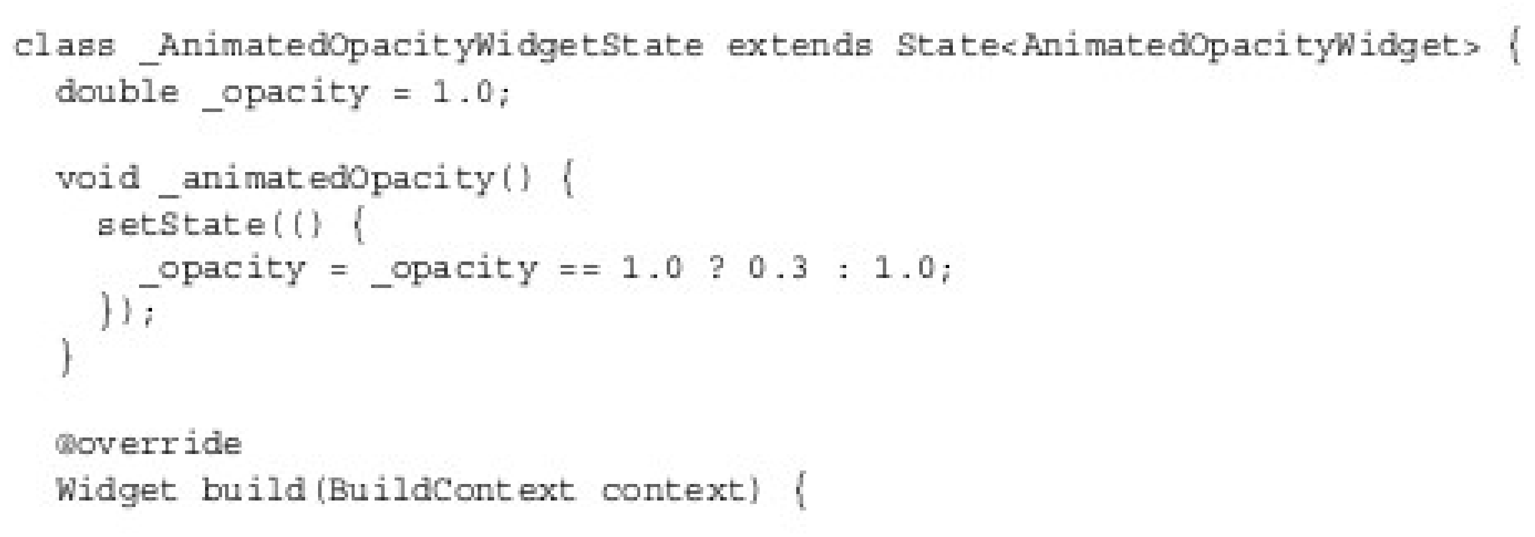


Constructor

```
AnimatedCrossFade
    ({Key? key,
    required Widget firstChild,
    required Widget secondChild,
    Curve firstCurve = Curves.linear,
    Curve secondCurve = Curves.linear,
    Curve sizeCurve = Curves.linear,
    AlignmentGeometry alignment = Alignment.topCenter,
    required CrossFadeState crossFadeState,
    required Duration duration,
    Duration? reverseDuration,
    AnimatedCrossFadeBuilder layoutBuilder = defaultLayoutBuilder
})
```

Animation Opacity

- Animated version of Opacity which automatically transitions the child's opacity over a given duration whenever the given opacity changes.



```
class _AnimatedOpacityWidgetState extends State<AnimatedOpacityWidget> {
  double _opacity = 1.0;

  void _animatedOpacity() {
    setState(() {
      _opacity = _opacity == 1.0 ? 0.3 : 1.0;
    });
  }

  @override
  Widget build(BuildContext context) {
```

AnimationController

- Special Animation object to control the animation itself. It generates new values whenever the application is ready for a new frame. It supports linear based animation and the value starts from 0.0 to 1.0
-

AnimationController

AnimationController is a special Animation object that generates a new value whenever the hardware is ready for a new frame. By default, an AnimationController linearly produces the numbers from 0.0 to 1.0 during a given duration. For example, this code creates an Animation object, but does not start it running:

```
controller =
    AnimationController(duration: const Duration(seconds: 2), vsync: this);
```

AnimationController derives from Animation<double>, so it can be used wherever an Animation object is needed. However, the AnimationController has additional methods to control the animation. For example, you start an animation with the .forward() method. The generation of numbers is tied to the screen refresh, so typically 60 numbers are generated per second. After each number is generated, each Animation object calls the attached Listener objects. To create a custom display list for each child, see RepaintBoundary.

When creating an AnimationController, you pass it a vsync argument. The presence of vsync prevents offscreen animations from consuming unnecessary resources. You can use your stateful object as the vsync by adding SingleTickerProviderStateMixin to the class definition. You can see an example of this in animate1 on GitHub.

Routes and Navigator in Flutter

- Route: Apps are the new trend. The number of apps available in the Play Store nowadays is quite a lot. The apps display their content in a full-screen container called pages or screens. In flutter, the pages or screens are called Routes. In android, these pages/screens are referred to as Activity and in iOS, it is referred to as ViewController. But, in a flutter, routes are referred to as Widgets. In Flutter, a Page / Screen is called a Route.

- Navigator: As the name suggests, Navigator is a widget that helps us to navigate between the routes. The navigator follows stack method when dealing with the routes. Based on the actions made by the user, the routes are stacked one over the other and when pressed back, it goes to the most recently visited route. Navigator is a widget that follows a stack discipline.

- Navigating to a Page: Since we have defined our Home, all the remaining is to navigate from home to another route of the app. For that the navigator widget has a method called Navigator.push(). This method pushes the route on top of the home, thereby displaying the second route. The code for pushing a route into the stack is as follows:
- // Within the `HomeRoute` widget

  onPressed: () {

  Navigator.push(

  context,

  MaterialPageRoute(builder: (context) => const SecondRoute()),

  );

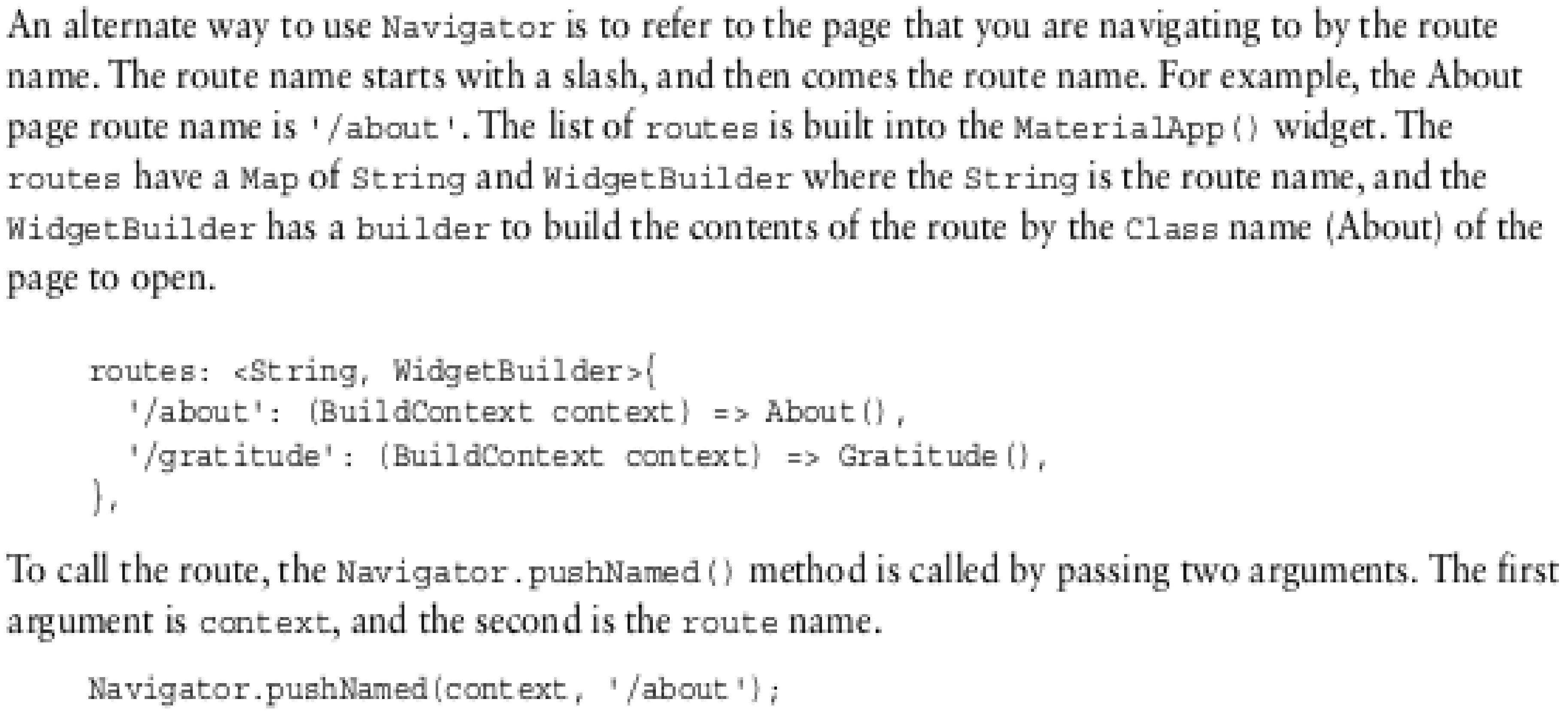
  }),

- Navigating Back to Home: Now we have arrived at our destination, but how do we go back home? For that, the navigator has a method called Navigator.pop(). This helps us to remove the present route from the stack so that we go back to our home route. This can be done as follows:
- // Within the SecondRoute widget
- onPressed: () {

  Navigator.pop(context);

  }

Flutter – Named Routes



An alternate way to use `Navigator` is to refer to the page that you are navigating to by the route name. The route name starts with a slash, and then comes the route name. For example, the About page route name is `'/about'`. The list of `routes` is built into the `MaterialApp()` widget. The `routes` have a `Map` of `String` and `WidgetBuilder` where the `String` is the route name, and the `WidgetBuilder` has a `builder` to build the contents of the route by the `Class` name (About) of the page to open.

```
routes: <String, WidgetBuilder>{
  '/about': (BuildContext context) => About(),
  '/gratitude': (BuildContext context) => Gratitude(),
},
```

To call the route, the `Navigator.pushNamed()` method is called by passing two arguments. The first argument is `context`, and the second is the `route` name.

```
Navigator.pushNamed(context, '/about');
```

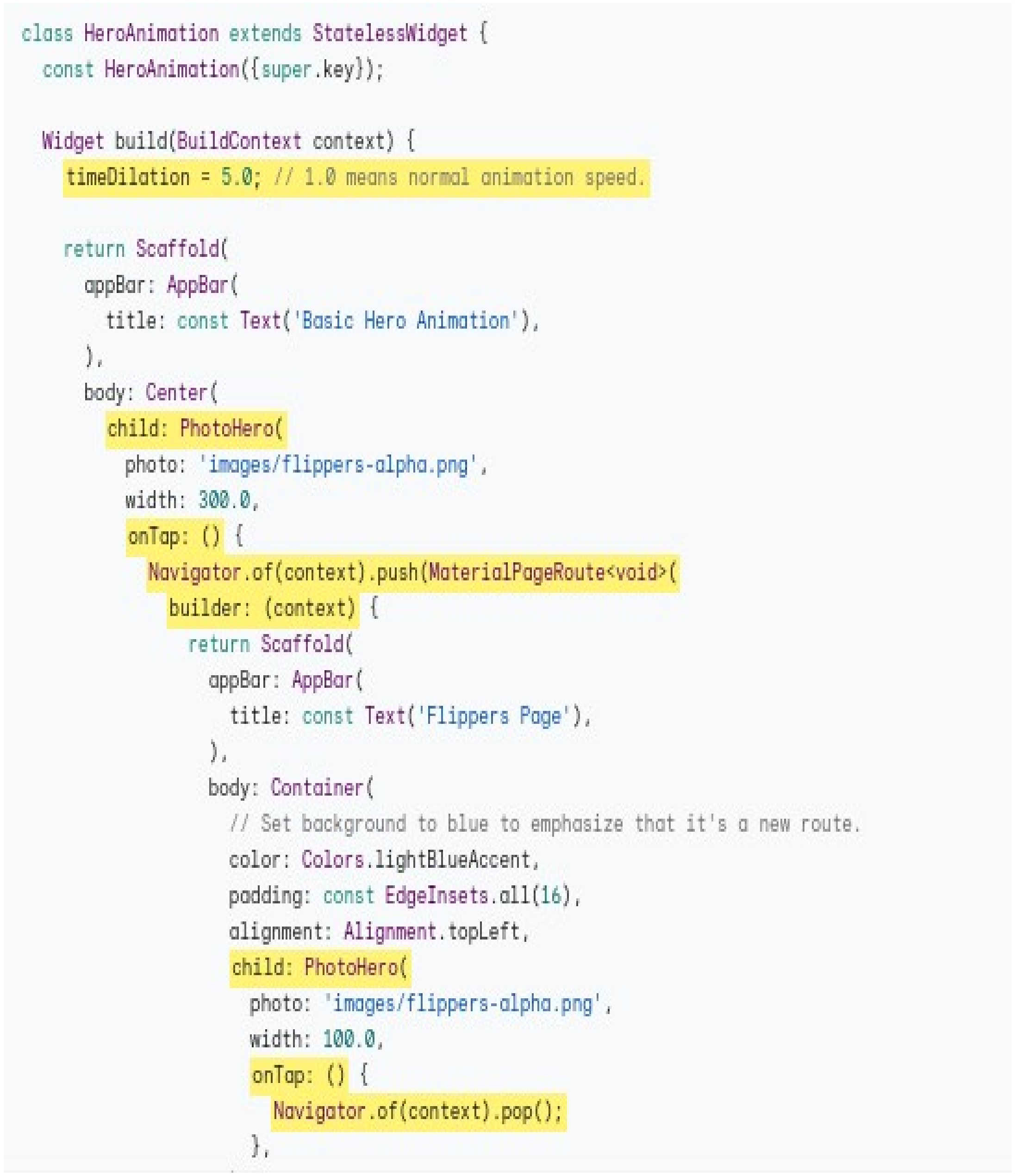
Hero Animation

- The hero refers to the widget that flies between screens.
- Fly the hero from one screen to another.
- A Route describes a page or screen in a Flutter app.
- A standard hero animation flies the hero from one route to a new route, usually landing at a different location and with a different size.
- In radial hero animation, as the hero flies between routes its shape appears to change from circular to rectangular.

Hero animation code has the following structure:

- Define a starting Hero widget, referred to as the source hero. The hero specifies its graphical representation (typically an image), and an identifying tag, and is in the currently displayed widget tree as defined by the source route.
- Define an ending Hero widget, referred to as the destination hero. This hero also specifies its graphical representation, and the same tag as the source hero. It’s essential that both hero widgets are created with the same tag, typically an object that represents the underlying data. For best results, the heroes should have virtually identical widget trees.
- Create a route that contains the destination hero. The destination route defines the widget tree that exists at the end of the animation.
- Trigger the animation by pushing the destination route on the Navigator’s stack. The Navigator push and pop operations trigger a hero animation for each pair of heroes with matching tags in the source and destination routes.

Hero Class



```
class HeroAnimation extends StatelessWidget {
  const HeroAnimation({super.key});

  Widget build(BuildContext context) {
    timeDilation = 5.0; // 1.0 means normal animation speed.

    return Scaffold(
      appBar: AppBar(
        title: const Text('Basic Hero Animation'),
      ),
      body: Center(
        child: PhotoHero(
          photo: 'images/flippers-alpha.png',
          width: 300.0,
          onTap: () {
            Navigator.of(context).push(MaterialPageRoute<void>(
              builder: (context) {
                return Scaffold(
                  appBar: AppBar(
                    title: const Text('Flippers Page'),
                  ),
                  body: Container(
                    // Set background to blue to emphasize that it's a new route.
                    color: Colors.lightBlueAccent,
                    padding: const EdgeInsets.all(16),
                    alignment: Alignment.topLeft,
                    child: PhotoHero(
                      photo: 'images/flippers-alpha.png',
                      width: 100.0,
                      onTap: () {
                        Navigator.of(context).pop();
                      },
```

BottomNavigationBar

- 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.
- We can display only a small number of widgets in the bottom navigation that can be 2 to 5.
- It must have at least two bottom navigation items. Otherwise, we will get an error.
- It is required to have the icon and title properties, and we need to set relevant widgets for them.

Properties of the BottomNavigationBar Widget

- items: It defines the list to display within the bottom navigation bar.
- currentIndex: It determines the current active bottom navigation bar item on the screen.
- onTap: It is called when we tapped one of the items on the screen.
- onTap: It is called when we tapped one of the items on the screen.
- fixedColor: It is used to set the color of the selected item. If we have not set a color to the icon or title, it will be shown.
- type: It determines the layout and behavior of a bottom navigation bar. It behaves in two different ways that are: fixed and shifting. If it is null, it will use fixed. Otherwise, it will use shifting where we can see an animation when we click a button.

Bottom appbar

USING THE BOTTOMAPPBAR

The BottomAppBar widget behaves similarly to the BottomNavigationBar, but it has an optional notch along the top. By adding a FloatingActionButton and enabling the notch, the notch provides a nice 3D effect so it looks like the button is recessed into the navigation bar (Figure 8.2). For example, to enable the notch, you set the BottomAppBar shape property to a NotchedShape class like the CircularNotchedRectangle() class and set the Scaffold floatingActionButtonLocation property to FloatingActionButtonLocation.endDocked or centerDocked. Add to the Scaffold floatingActionButton property a FloatingActionButton widget, and the result shows the FloatingActionButton embedded into the BottomAppBar widget, which is the notch.

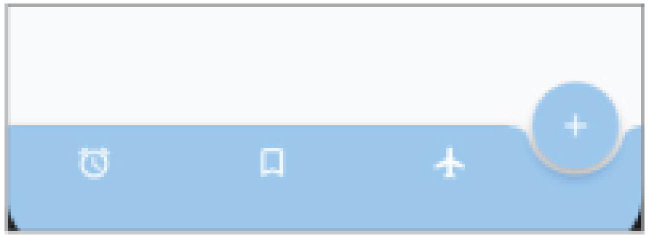
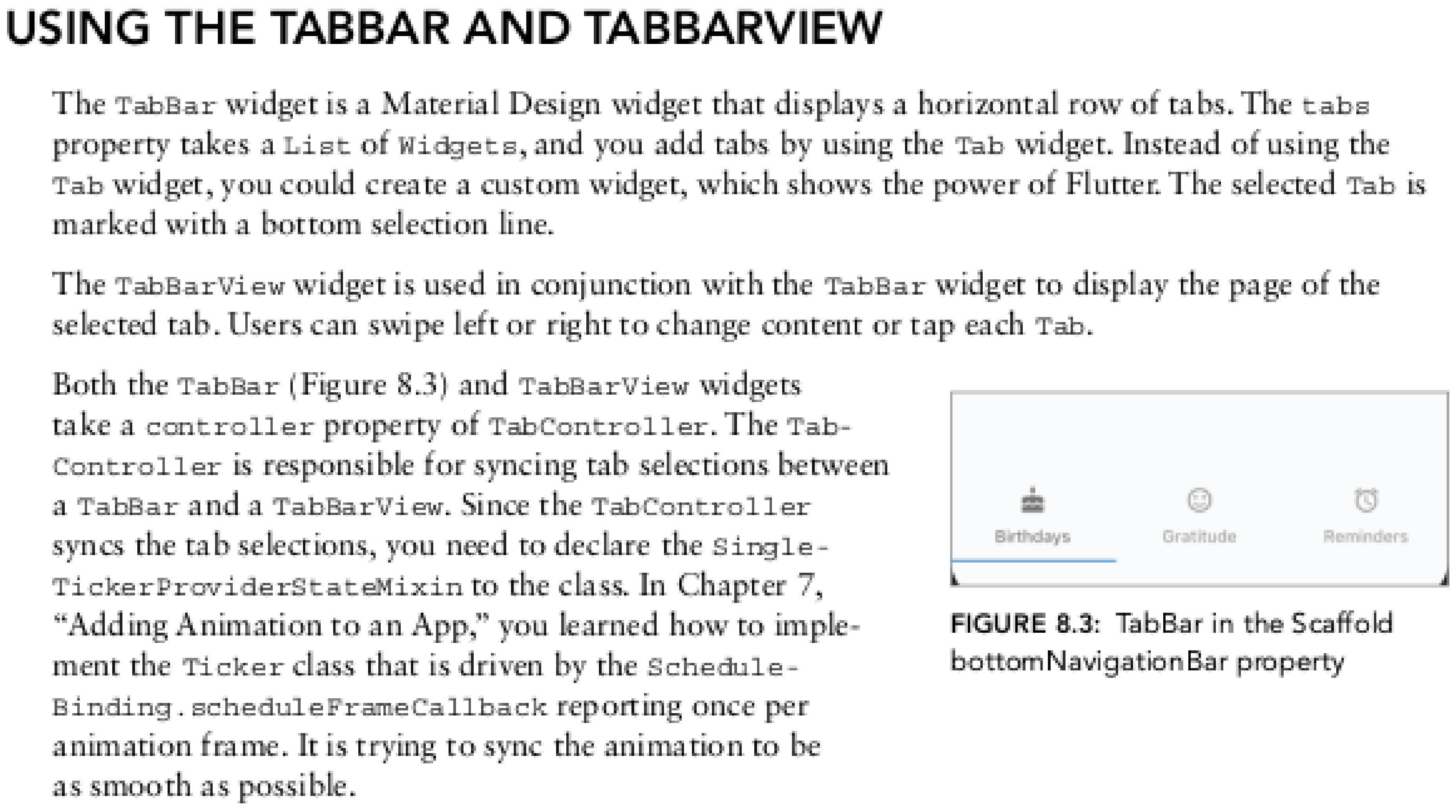


FIGURE 8.2: BottomAppBar with embedded FloatingActionButton creating a notch

```
BottomAppBar(
  shape: CircularNotchedRectangle(),
)

floatingActionButtonLocation: FloatingActionButtonLocation.endDocked,
floatingActionButton: FloatingActionButton(
  child: Icon(Icons.add),
),
```

Using the TabBar and TabBarView



USING THE TABBAR AND TABBARVIEW

The `TabBar` widget is a Material Design widget that displays a horizontal row of tabs. The `tabs` property takes a `List` of `Widgets`, and you add tabs by using the `Tab` widget. Instead of using the `Tab` widget, you could create a custom widget, which shows the power of Flutter. The selected `Tab` is marked with a bottom selection line.

The `TabBarView` widget is used in conjunction with the `TabBar` widget to display the page of the selected tab. Users can swipe left or right to change content or tap each `Tab`.

Both the `TabBar` (Figure 8.3) and `TabBarView` widgets take a `controller` property of `TabController`. The `Tab-Controller` is responsible for syncing tab selections between a `TabBar` and a `TabBarView`. Since the `TabController` syncs the tab selections, you need to declare the `Single-TickerProviderStateMixin` to the class. In Chapter 7, "Adding Animation to an App," you learned how to implement the `Ticker` class that is driven by the `Schedule-Binding.scheduleFrameCallback` reporting once per animation frame. It is trying to sync the animation to be as smooth as possible.

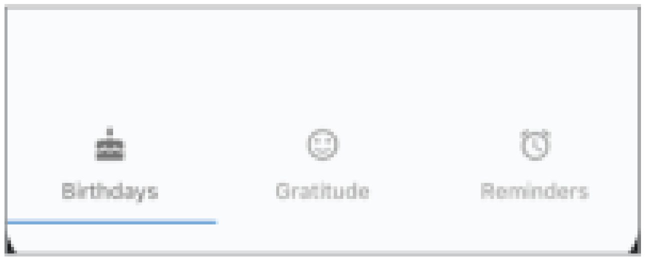


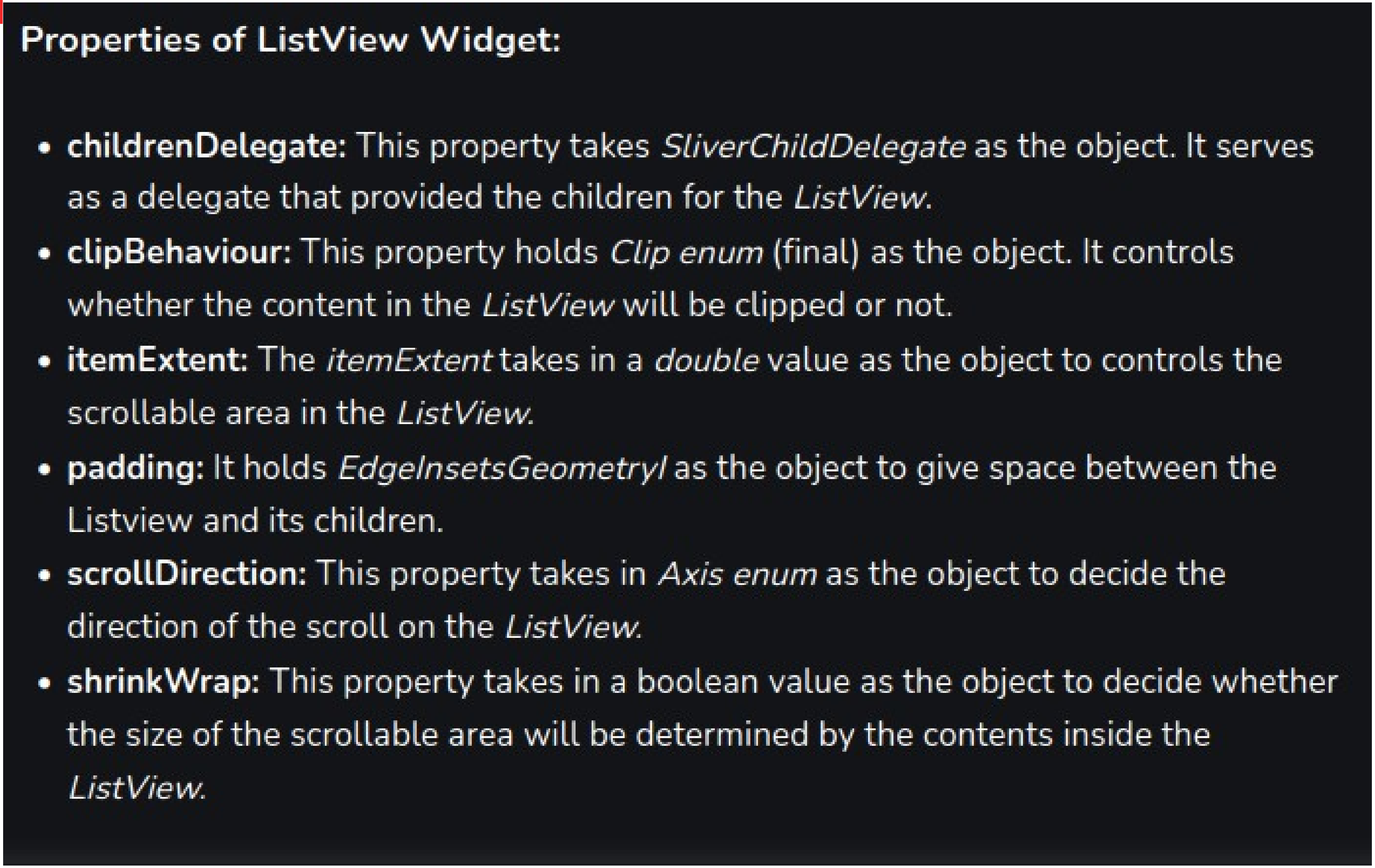
FIGURE 8.3: TabBar in the Scaffold bottomNavigationBar property

USING THE DRAWER AND LISTVIEW

`Drawer` is a Material Design panel that slides horizontally from the left or right edge of the `Scaffold`, the device screen. `Drawer` is used with the `Scaffold` `drawer` (left-side) property or `endDrawer` (right-side) property. `Drawer` can be customized for each individual need but usually has a header to show an image or fixed information and a `ListView` to show a list of navigable pages. Usually, a `Drawer` is used when the navigation list has many items.

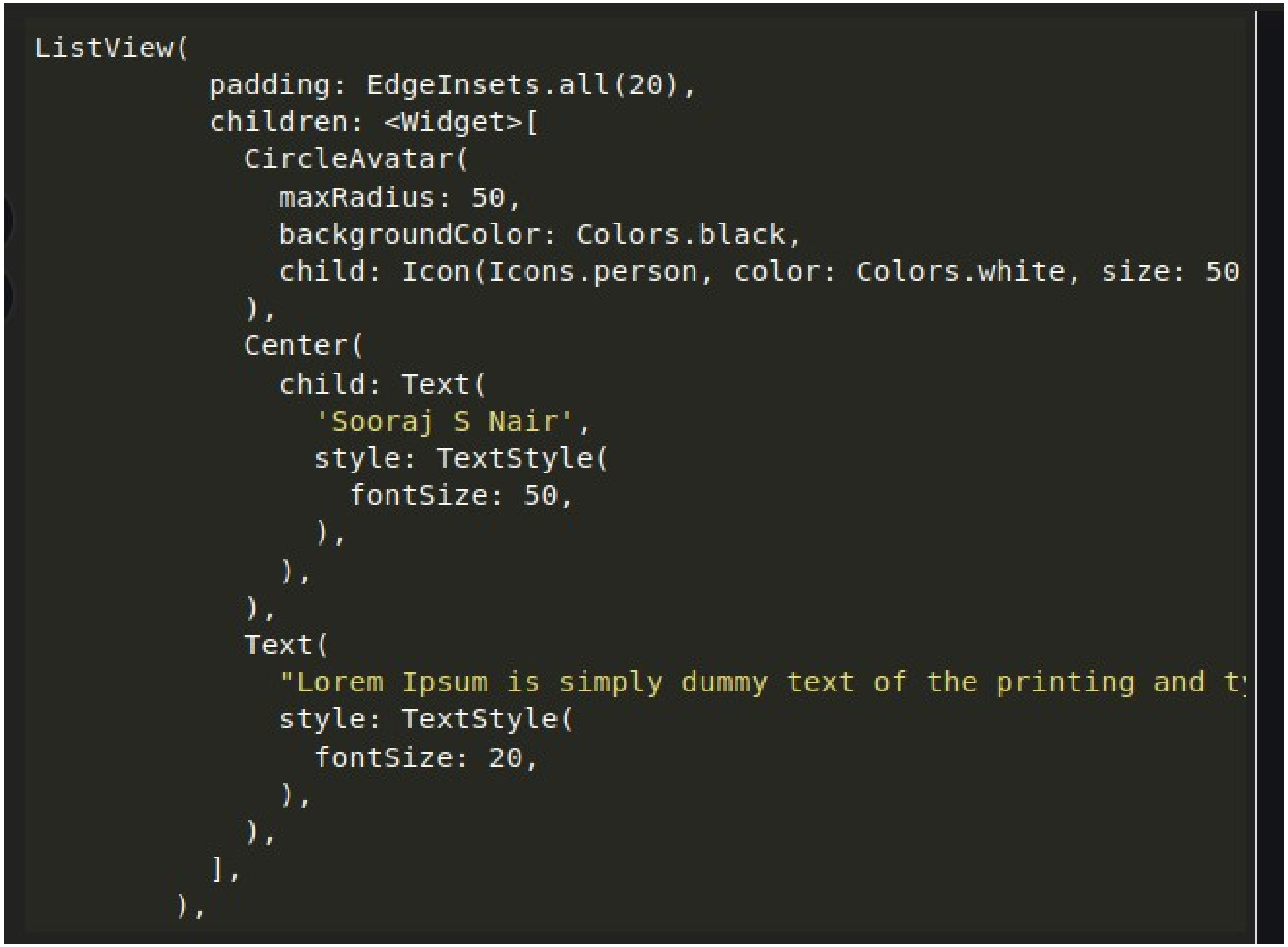
To set the `Drawer` header, you have two built-in options, the `UserAccountsDrawerHeader` or the `DrawerHeader`. The `UserAccountsDrawerHeader` is intended to display the app's user details by setting the `currentAccountPicture`, `accountName`, `accountEmail`, `otherAccountsPictures`, and `decoration` properties.

```
// User details
UserAccountsDrawerHeader(
  currentAccountPicture: Icon(Icons.face,),
  accountName: Text('Sandy Smith'),
  accountEmail: Text('sandy.smith@domainname.com'),
  otherAccountsPictures: <Widget>[
    Icon(Icons.bookmark_border),
  ],
  decoration: BoxDecoration(
    image: DecorationImage(
      image: AssetImage('assets/images/home_top_mountain.jpg'),
      fit: BoxFit.cover,
    ),
  ),
),
```



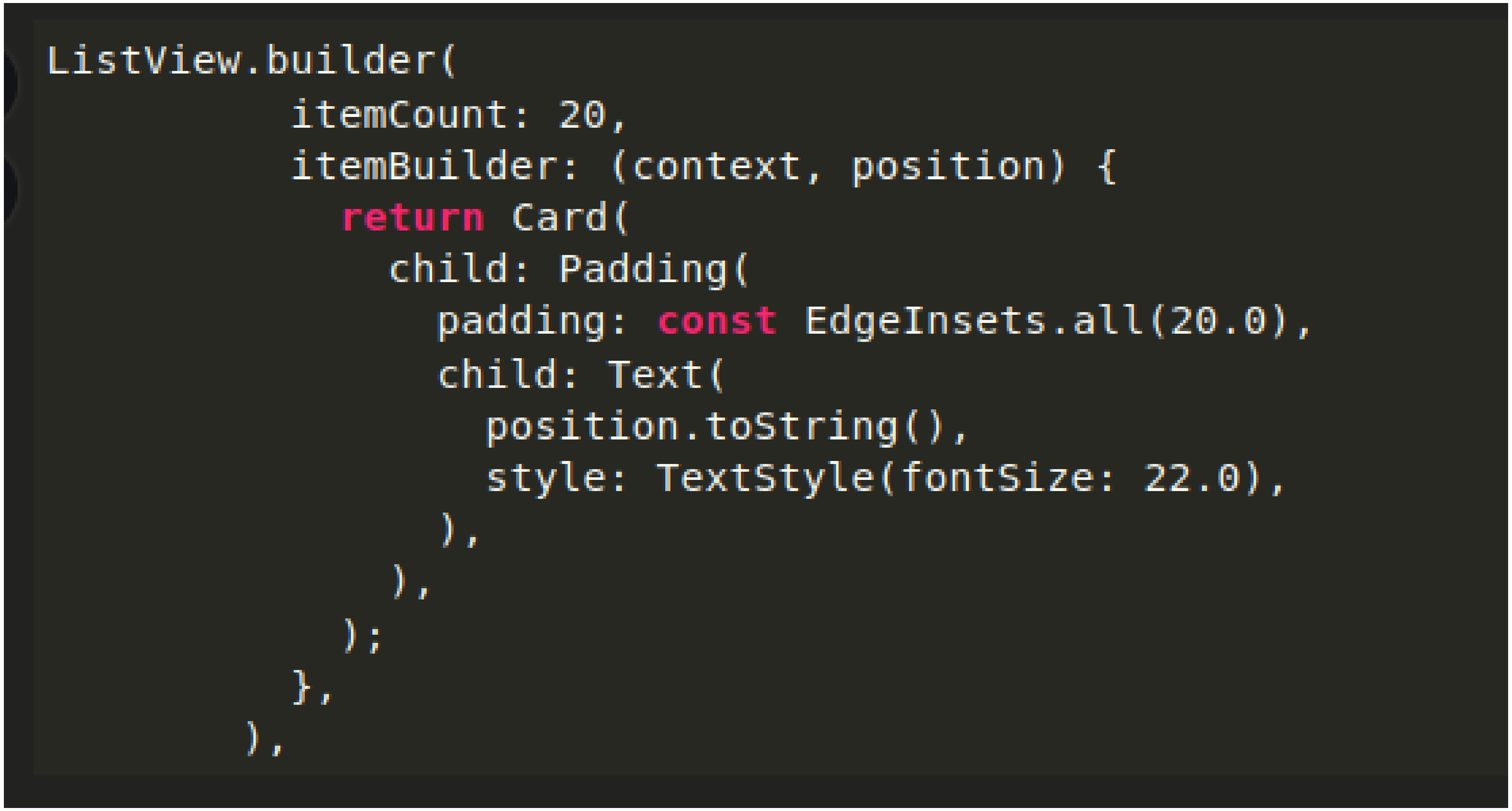
Properties of ListView Widget:

- **childrenDelegate:** This property takes *SliverChildDelegate* as the object. It serves as a delegate that provided the children for the *ListView.*
- **clipBehaviour:** This property holds *Clip enum* (final) as the object. It controls whether the content in the *ListView* will be clipped or not.
- **itemExtent:** The *itemExtent* takes in a *double* value as the object to controls the scrollable area in the *ListView.*
- **padding:** It holds *EdgeInsetsGeometryI* as the object to give space between the Listview and its children.
- **scrollDirection:** This property takes in *Axis enum* as the object to decide the direction of the scroll on the *ListView.*
- **shrinkWrap:** This property takes in a boolean value as the object to decide whether the size of the scrollable area will be determined by the contents inside the *ListView.*



```
ListView(
          padding: EdgeInsets.all(20),
          children: <Widget>[
            CircleAvatar(
              maxRadius: 50,
              backgroundColor: Colors.black,
              child: Icon(Icons.person, color: Colors.white, size: 50
            ),
            Center(
              child: Text(
                'Sooraj S Nair',
                style: TextStyle(
                  fontSize: 50,
                ),
              ),
            ),
            Text(
              "Lorem Ipsum is simply dummy text of the printing and t
              style: TextStyle(
                fontSize: 20,
              ),
            ),
          ],
        ),
```

- ListView.builder()
- The builder() constructor constructs a repeating list of widgets. The constructor takes two main parameters:
- An itemCount for the number of repetitions for the widget to be constructed (not compulsory).
- An itemBuilder for constructing the widget which will be generated ‘itemCount‘ times (compulsory).



```
ListView.builder(
          itemCount: 20,
          itemBuilder: (context, position) {
            return Card(
              child: Padding(
                padding: const EdgeInsets.all(20.0),
                child: Text(
                  position.toString(),
                  style: TextStyle(fontSize: 22.0),
                ),
              ),
            );
          },
        ),
```

More Details & Example

- https://docs.flutter.dev/