Flutter
Migrating to the New Material Buttons and their Themes

SUMMARY

A guide to migrating existing apps to the new Flutter button classes and their themes.

Author: Hans Muller (@hansmuller)
Go Link: flutter.dev/go/material-button-migration-guide
Created: August 2020 / **Last updated:** August 2020

A guide to upgrading Flutter apps that depend on the original button classes, FlatButton, RaisedButton, OutlineButton, ButtonTheme, to the new classes: TextButton, ElevatedButton, OutlinedButton, TextButtonTheme, ElevatedButtonTheme, and OutlinedButtonTheme.

A slightly newer version of this document is available on the Flutter website: https://flutter.dev/docs/release/breaking-changes/buttons.

The changes defined in this document were originally proposed in flutter.dev/go/material-button-system-updates and landed in Flutter in #59702 (July 2020) and #61262. They will be included in the stable Flutter release that follows Flutter 1.20.

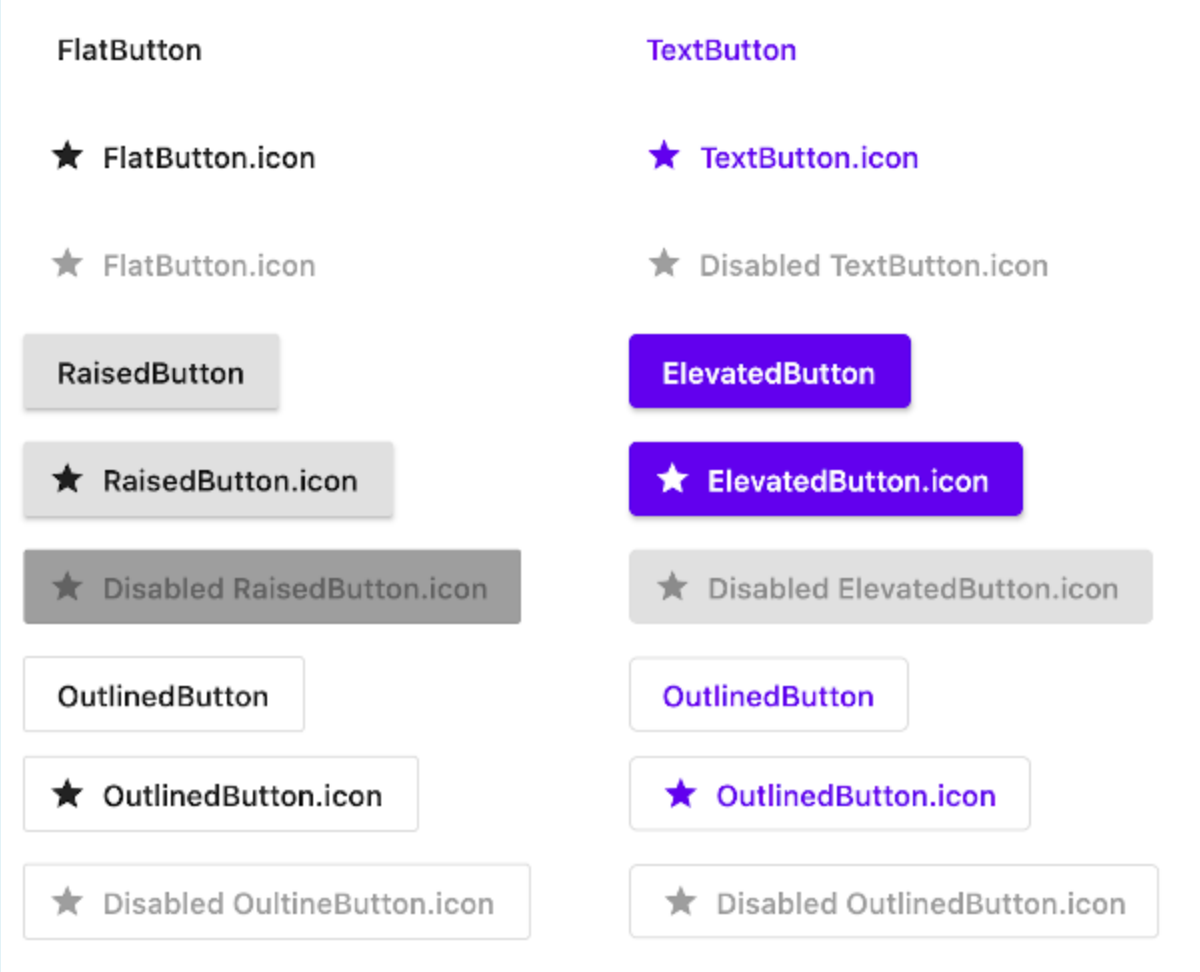
Changes Overview

The original button classes - FlatButton, RaisedButton, OutlineButton, ButtonTheme, as well as their supporting classes MaterialButton and RawMaterialButton, have not yet been deprecated or removed. Eventually they will be deprecated, and after a suitably long period of time, removed.

Old Widget	Old Theme	New Widget	New Theme
FlatButton	ButtonTheme	TextButton	TextButtonTheme
RaisedButton	ButtonTheme	ElevatedButton	ElevatedButtonTheme
OutlineButton	ButtonTheme	OutlinedButton	OutlinedButtonTheme

Visual Changes

Visually, the new buttons look a little different, because they match the current Material Design spec and because their colors are configured in terms of the overall Theme's ColorScheme. There are other small differences in padding, rounded corner radii, and the hover/focus/pressed feedback.



In the screenshot above, the overall Theme for the buttons matches the latest purple Material Design default, and was created with:

```
MaterialApp(
  theme: ThemeData.from(colorScheme: ColorScheme.light()),
  ...
```

```
)
```

The new button themes follow the "normalized" pattern that Flutter adopted for new Material widgets about a year ago. Theme properties and widget constructor parameters are null by default. Non-null theme properties and widget parameters specify an override of the component's default value. Implementing and documenting default values is the sole responsibility of the button widgets. The defaults themselves are based primarily on the overall Theme's colorScheme and textTheme.

API Change: ButtonStyle instead of individual style properties

Except for simple use cases, the APIs of the new button classes are not compatible with the old classes. The visual attributes of the new buttons and themes are configured with a single ButtonStyle object, similar to how a TextField or a Text widget can be configured with a TextStyle object. Most of the ButtonStyle properties are defined with MaterialStateProperty, so that a single property can represent different values depending on the button's pressed/focused/hovered/etc state.

A button's ButtonStyle doesn't define the button's visual properties, it only defines *overrides* of the buttons default visual properties, where the default properties are computed by the button widget itself. For example, to override a TextButton's default foreground (text/icon) color for all states, one could write:

```
TextButton(
  style: ButtonStyle(
    foregroundColor: MaterialStateProperty.all<Color>(Colors.blue),
  ),
  onPressed: () { },
  child: Text('TextButton'),
)
```

This kind of override is common; however, in many cases what's also needed are overrides for overlay colors that the text button uses to indicate its hovered/focus/pressed state. This can be done by adding the overlayColor property to the ButtonStyle.

```
TextButton(
  style: ButtonStyle(
    foregroundColor: MaterialStateProperty.all<Color>(Colors.blue),
    overlayColor: MaterialStateProperty.resolveWith<Color>(
      (Set<MaterialState> states) {
        if (states.contains(MaterialState.hovered))
```

```
          return Colors.blue.withOpacity(0.04);
        if (states.contains(MaterialState.focused) ||
            states.contains(MaterialState.pressed))
          return Colors.blue.withOpacity(0.12);
        return null; // Defer to the widget's default.
      },
    ),
  ),
  onPressed: () { },
  child: Text('TextButton')
)
```

A color MaterialStateProperty only needs to return a value for the colors whose default should be overridden. If it returns null, the widget's default will be used instead. For example, to just override the text button's focus overlay color:

```
TextButton(
  style: ButtonStyle(
    overlayColor: MaterialStateProperty.resolveWith<Color>(
      (Set<MaterialState> states) {
        if (states.contains(MaterialState.focused))
          return Colors.red;
        return null; // Defer to the widget's default.
      }
    ),
  ),
  onPressed: () { },
  child: Text('TextButton'),
)
```

The Material Design spec defines the foreground and overlay colors in terms of the color scheme's primary color. The primary color is rendered at different opacities, depending on the button's state. To simplify creating a button style that includes all of the properties that depend on color scheme colors, each button class includes a static styleFrom() method which constructs a ButtonStyle from a simple set of values, including the ColorScheme colors it depends on.

This example creates a button that overrides it's foreground color, as well as its overlay color, using the specified primary color and the opacities from the Material Design spec. It creates the same button style as the previous example.

```
TextButton(
  style: TextButton.styleFrom(
    primary: Colors.blue,
  ),
  onPressed: () { },
  child: Text('TextButton'),
)
```

The TextButton documentation indicates that the foreground color when the button is disabled is based on the color scheme's onSurface color. To override that as well, using styleFrom():

```
TextButton(
  style: TextButton.styleFrom(
    primary: Colors.blue,
    onSurface: Colors.red,
  ),
  onPressed: null,
  child: Text('TextButton'),
),
```

Using the styleFrom() method is the preferred way to create a ButtonStyle if you're trying to create a Material Design variation. The most flexible approach is defining a ButtonStyle directly, with MaterialStateProperty values for the states whose appearance you want to override.

Migrating Simple Use Cases

In many cases it's possible to just switch from the old button class to the new one. That's assuming that the small changes in size/shape and the likely bigger change in colors, aren't a concern.

To preserve the original buttons' appearance in these cases, one can define button styles that match the original as closely as you like. For example, the following style will make a TextButton look like a default FlatButton:

```
final ButtonStyle flatButtonStyle = TextButton.styleFrom(
  primary: Colors.black87,
  minimumSize: Size(88, 36),
  padding: EdgeInsets.symmetric(horizontal: 16.0),
  shape: const RoundedRectangleBorder(
    borderRadius: BorderRadius.all(Radius.circular(2.0)),
  ),
);

TextButton(
  style: flatButtonStyle,
  onPressed: () { },
  child: Text('Looks like a FlatButton'),
)
```

Similarly, to make an ElevatedButton look like a default RaisedButton:

```
final ButtonStyle raisedButtonStyle = ElevatedButton.styleFrom(
  onPrimary: Colors.black87,
  primary: Colors.grey[300],
  minimumSize: Size(88, 36),
  padding: EdgeInsets.symmetric(horizontal: 16),
  shape: const RoundedRectangleBorder(
    borderRadius: BorderRadius.all(Radius.circular(2)),
  ),
);
ElevatedButton(
  style: raisedButtonStyle,
  onPressed: () { },
  child: Text('Looks like a RaisedButton'),
)
```

The OutlineButton style for OutlinedButton is a little more complicated because the outline's color changes to the primary color when the button is pressed. The outline's appearance is defined by a BorderSide and we'll use a MaterialStateProperty to define the pressed outline color:

```
final ButtonStyle outlineButtonStyle = OutlinedButton.styleFrom(
  primary: Colors.black87,
  minimumSize: Size(88, 36),
  padding: EdgeInsets.symmetric(horizontal: 16),
  shape: const RoundedRectangleBorder(
    borderRadius: BorderRadius.all(Radius.circular(2)),
  ),
).copyWith(
  side: MaterialStateProperty.resolveWith<BorderSide>(
    (Set<MaterialState> states) {
      if (states.contains(MaterialState.pressed))
        return BorderSide(
          color: Theme.of(context).colorScheme.primary,
          width: 1,
        );
      return null; // Defer to the widget's default.
    },
  ),
);


OutlinedButton(
  style: outlineButtonStyle,
  onPressed: () { },
  child: Text('Looks like an OutlineButton'),
)
```

To restore the default appearance for buttons throughout an application, one can configure the new button themes in the application's theme:

```
MaterialApp(
  theme: ThemeData.from(colorScheme: ColorScheme.light()).copyWith(
    textButtonTheme: TextButtonThemeData(style: flatButtonStyle),
    elevatedButtonTheme: ElevatedButtonThemeData(style: raisedButtonStyle),
    outlinedButtonTheme: OutlinedButtonThemeData(style: outlineButtonStyle),
  ),
)
```

To restore the default appearance for buttons in part of an application you can wrap a widget subtree with TextButtonTheme, ElevatedButtonTheme, or OutlinedButtonTheme. For example:

```
TextButtonTheme(
  data: TextButtonThemeData(style: flatButtonStyle),
  child: myWidgetSubtree,
)
```

Migrating buttons with custom colors

The following sections cover use of the following FlatButton, RaisedButton, and OutlineButton color parameters:

- textColor
- disabledTextColor
- color
- disabledColor
- focusColor
- hoverColor
- highlightColor*
- splashColor

*The new button classes do not support a separate highlight color because it's no longer part of the Material Design.

Migrating buttons with custom foreground and background colors

Two common customizations for the original button classes are a custom foreground color for FlatButton, or custom foreground and background colors for RaisedButton. Producing the same result with the new button classes is simple:

```
FlatButton(
```

```
  textColor: Colors.red, // foreground
  onPressed: () { },
  child: Text('FlatButton with custom foreground/background'),
)

TextButton(
  style: TextButton.styleFrom(
    primary: Colors.red, // foreground
  ),
  onPressed: () { },
  child: Text('TextButton with custom foreground'),
)
```

In this case the TextButton's foreground (text/icon) color as well as its hovered/focused/pressed overlay colors will be based on Colors.red. By default, the TextButton's background fill color is transparent.

Migrating a RaisedButton with custom foreground and background colors:

```
RaisedButton(
  color: Colors.red, // background
  textColor: Colors.white, // foreground
  onPressed: () { },
  child: Text('RaisedButton with custom foreground/background'),
)

ElevatedButton(
  style: ElevatedButton.styleFrom(
    primary: Colors.red, // background
    onPrimary: Colors.white, // foreground
  ),
  onPressed: () { },
  child: Text('ElevatedButton with custom foreground/background'),
)
```

In this case the button's use of the color scheme's primary color is reversed relative to the TextButton: primary is button's background fill color and onPrimary is the foreground (text/icon) color.

Migrating buttons with custom overlay colors

Overriding a button's default focused, hovered, highlighted, or splash colors is less common. The FlatButton, RaisedButton, and OutlineButton classes have individual parameters for these state-dependent colors. The new TextButton, ElevatedButton, and OutlinedButton classes use a single MaterialStateProperty<Color> parameter instead. The new buttons allow one to specify state-dependent values for all of the

colors, the original buttons only supported specifying what's now called the "overlayColor".

```
FlatButton(
  focusColor: Colors.red,
  hoverColor: Colors.green,
  splashColor: Colors.blue,
  onPressed: () { },
  child: Text('FlatButton with custom overlay colors'),
)
```

```
TextButton(
  style: ButtonStyle(
    overlayColor: MaterialStateProperty.resolveWith<Color>(
      (Set<MaterialState> states) {
        if (states.contains(MaterialState.focused))
          return Colors.red;
        if (states.contains(MaterialState.hovered))
            return Colors.green;
        if (states.contains(MaterialState.pressed))
            return Colors.blue;
        return null; // Defer to the widget's default.
    }),
  ),
  onPressed: () { },
  child: Text('TextButton with custom overlay colors'),
)
```

The new version is more flexible although less compact. In the original version, the the precedence of the different states is implicit (and undocumented) and fixed, in the new version, it's explicit. For an app that specified these colors frequently, the easiest migration path would be to define one or more ButtonStyles that match the example above - and just use the style parameter - or to define a stateless wrapper widget that encapsulated the three color parameters.

Migrating buttons with custom disabled colors

This is a relatively rare customization. The FlatButton, RaisedButton, and OutlineButton classes have disabledTextColor and disabledColor parameters that define the background and foreground colors when the button's onPressed callback is null.

By default, all of the buttons use the color scheme's onSurface color, with opacity 0.38 for the disabled foreground color. Only ElevatedButton has a non-transparent background color and its default value is the onSurface color with opacity 0.12. So in many cases one can just use the styleFrom method to override the disabled

colors:

```
RaisedButton(
  disabledColor: Colors.red.withOpacity(0.12),
  disabledTextColor: Colors.red.withOpacity(0.38),
  onPressed: null,
  child: Text(‘RaisedButton with custom disabled colors'),
),

ElevatedButton(
  style: ElevatedButton.styleFrom(onSurface: Colors.red),
  onPressed: null,
  child: Text('ElevatedButton with custom disabled colors'),
)
```

For complete control over the disabled colors, one must define the ElevatedButton’s style explicitly, in terms of MaterialStateProperties:

```
RaisedButton(
  disabledColor: Colors.red,
  disabledTextColor: Colors.blue,
  onPressed: null,
  child: Text('RaisedButton with custom disabled colors'),
)

ElevatedButton(
  style: ButtonStyle(
    backgroundColor: MaterialStateProperty.resolveWith<Color>(
      (Set<MaterialState> states) {
        if (states.contains(MaterialState.disabled))
          return Colors.red;
        return null; // Defer to the widget's default.
    }),
    foregroundColor: MaterialStateProperty.resolveWith<Color>(
      (Set<MaterialState> states) {
        if (states.contains(MaterialState.disabled))
          return Colors.blue;
        return null; // Defer to the widget's default.
    }),
  ),
  onPressed: null,
  child: Text('ElevatedButton with custom disabled colors'),
)
```

As with the previous case, there are obvious ways to make the new version more compact, in an app where this migration comes up often.

Migrating buttons with custom elevations

This is also a relatively rare customization. Typically, only ElevatedButtons (née RaisedButtons) include elevation changes. For elevations that are proportional to a baseline elevation (per the Material Design specification), one can override all of them quite simply.

By default a disabled button's elevation is 0, and the remaining states are defined relative to a baseline of 2:

- disabled: 0
- hovered or focused: baseline + 2
- pressed: baseline + 6

So to migrate a RaisedButton for which all elevations have been defined:

```
RaisedButton(
  elevation: 2,
  focusElevation: 4,
  hoverElevation: 4,
  highlightElevation: 8,
  disabledElevation: 0,
  onPressed: () { },
  child: Text('RaisedButton with custom elevations'),
)


ElevatedButton(
  style: ElevatedButton.styleFrom(elevation: 2),
  onPressed: () { },
  child: Text('ElevatedButton with custom elevations'),
)
```

To arbitrarily override just one elevation, like the pressed elevation:

```
RaisedButton(
  highlightElevation: 16,
  onPressed: () { },
  child: Text('RaisedButton with a custom elevation'),
)


ElevatedButton(
  style: ButtonStyle(
    elevation: MaterialStateProperty.resolveWith<double>(
      (Set<MaterialState> states) {
        if (states.contains(MaterialState.pressed))
          return 16;
```

```
        return null;
      }),
  ),
  onPressed: () { },
  child: Text('ElevatedButton with a custom elevation'),
)
```

Migrating buttons with custom shapes and borders

The original FlatButton, RaisedButton, and OutlineButton classes all provide a shape parameter which defines both the button's shape and the appearance of its outline. The corresponding new classes and their themes support specifying the button's shape and its border separately, with *OutlinedBorder shape* and *BorderSide side* parameters.

In this example the original OutlineButton version specifies the same color for border in its highlighted (pressed) state as for other states.

```
OutlineButton(
  shape: StadiumBorder(),
  highlightedBorderColor: Colors.red,
  borderSide: BorderSide(
    width: 2,
    color: Colors.red
  ),
  onPressed: () { },
  child: Text('OutlineButton with custom shape and border'),
)
```

```
OutlinedButton(
  style: OutlinedButton.styleFrom(
    shape: StadiumBorder(),
    side: BorderSide(
      width: 2,
      color: Colors.red
    ),
  ),
  onPressed: () { },
  child: Text('OutlinedButton with custom shape and border'),
)
```

Most of the new OutlinedButton widget's style parameters, including its shape and border, can be specified with MaterialStateProperty values, which is to say that they can have different values depending on the button's state. To specify a different border color when the button is pressed:

```
OutlineButton(
```

```
  shape: StadiumBorder(),
  highlightedBorderColor: Colors.blue,
  borderSide: BorderSide(
    width: 2,
    color: Colors.red
  ),
  onPressed: () { },
  child: Text('OutlineButton with custom shape and border'),
)


OutlinedButton(
  style: ButtonStyle(
    shape: MaterialStateProperty.all<OutlinedBorder>(StadiumBorder()),
    side: MaterialStateProperty.resolveWith<BorderSide>(
      (Set<MaterialState> states) {
        final Color color = states.contains(MaterialState.pressed)
          ? Colors.blue
          : Colors.red;
        return BorderSide(color: color, width: 2);
      }
    ),
  ),
  onPressed: () { },
  child: Text('OutlinedButton with custom shape and border'),
)
```