

Flutter Fundamentals

Apps, widgets, properties



Peter Kassenaar –
info@kassenaar.com



Starting from scratch

Building the UI from scratch – based on the Default Starter App

Working with Widget Trees



- Starting from the Default App and **deleting** (most of) the code
- Open `main.dart`
- Remove all classes and empty `runApp()`
- Your app will **not work** at the moment:

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

void main() {
  runApp();
}
```

Using `MaterialApp()`



- `MaterialApp()` is often the [start widget](#) of your Widget Tree
- It adheres to the Google Material Design Specs
 - <https://material.io/design>
- You configure it with *properties*, for instance `home:`
- But: you [can not](#) simply type some text to be shown on the home page

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

void main() {
  runApp(MaterialApp(
    home: 'Hello Flutter!'
  ));
}
```

Invalid!

Using Widgets



- *Rule* – ALL content **MUST** be wrapped inside a **Widget**
- There are **thousands** of predefined widgets
- You can build/compose **custom widgets** yourself
- For instance: wrap text inside a `Text()` Widget
- But – **EVERY piece of content** is some form of widget or property

```
void main() {  
  runApp(MaterialApp(  
    home: Text('Hello Flutter!'),  
  ));  
}
```

Valid!

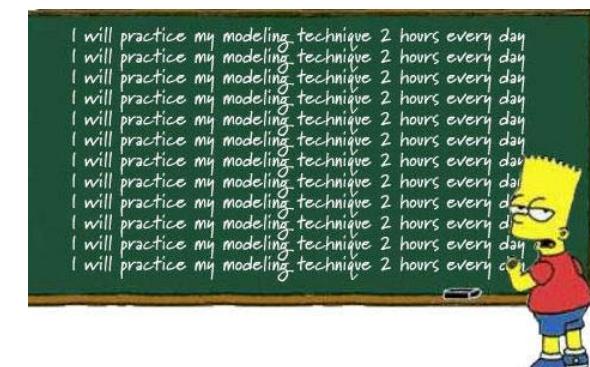
Result – ugly but working



Workshop



- Remove all content from the default application
- Replace the contents of `runApp()` with your own text message inside `MaterialApp()`
- Don't copy paste (for now!). Make sure to at least type the code once yourself, so you understand where the (...) and {...} go.
- Place a trailing comma after the `Text()` widget
 - Technically not necessary, just best practice.
- Make sure the application still works



Checkpoint



- You understand – and can create – applications from scratch
- You know how to make an application run and what widget-requirements an app has.



Layout widgets

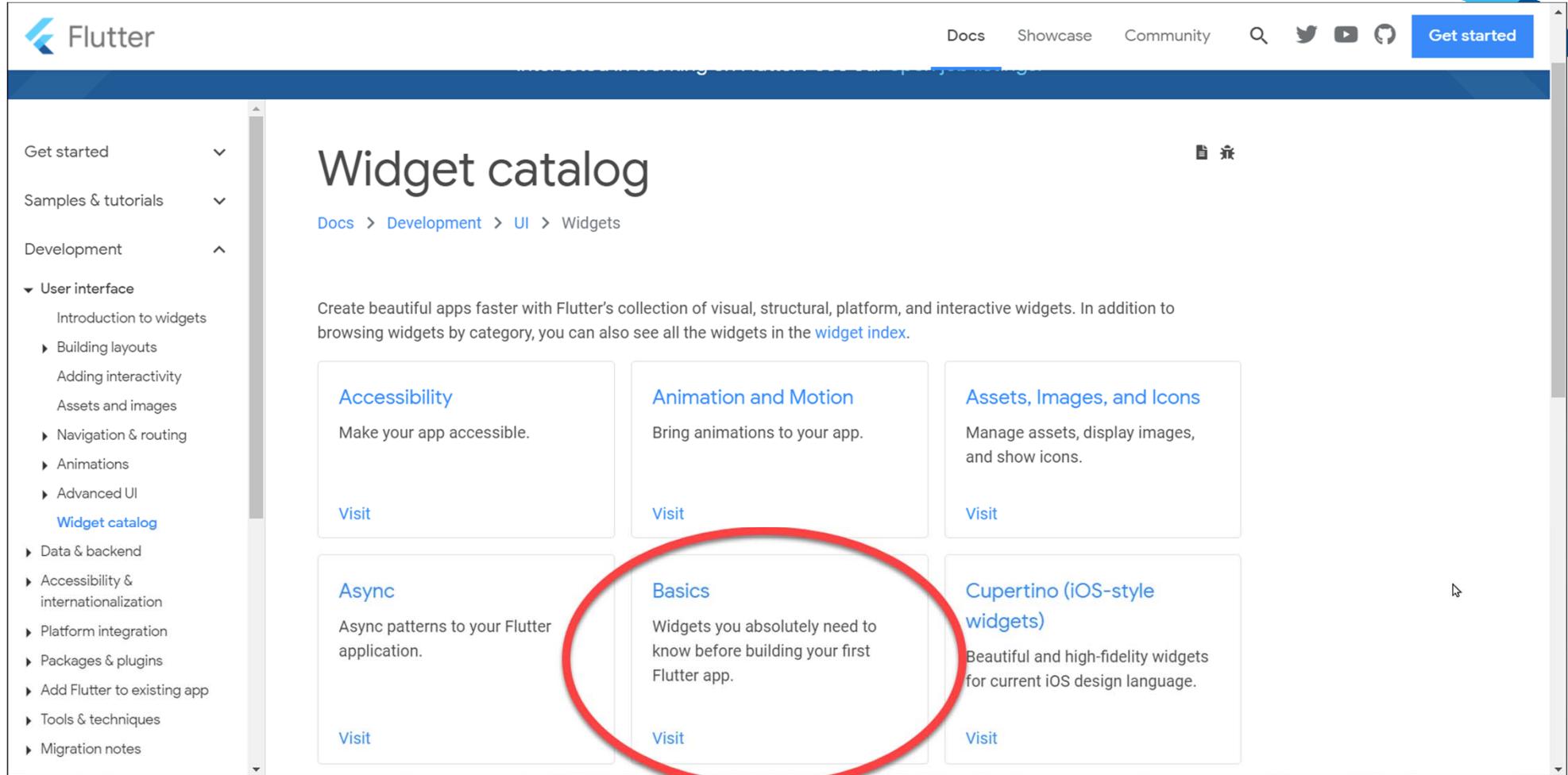
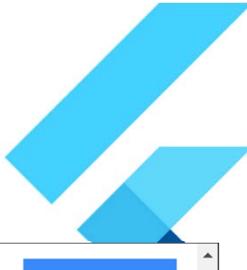
Default widgets to create the lay-out of your app.

Creating the structure of your lay-out



- Scaffold()
- AppBar()
- Themes, fonts & colors
- Images
- Icons
- Rows, Columns

The Widget Catalog



A screenshot of the Flutter documentation website showing the Widget catalog page. The page has a dark blue header with the Flutter logo, navigation links for Docs, Showcase, Community, and a search bar. A prominent blue button labeled "Get started" is visible. On the left, a sidebar menu is open under the "User interface" section, with "Widget catalog" highlighted. The main content area features a large title "Widget catalog" and a breadcrumb trail: Docs > Development > UI > Widgets. Below this, a text block encourages creating apps faster using Flutter's widgets. The page is divided into several cards: Accessibility, Animation and Motion, Assets, Images, and Icons, Async, Basics (circled in red), Cupertino (iOS-style widgets), and Data & backend. Each card has a brief description and a "Visit" button.

Widget catalog

Docs > Development > UI > Widgets

Create beautiful apps faster with Flutter's collection of visual, structural, platform, and interactive widgets. In addition to browsing widgets by category, you can also see all the widgets in the [widget index](#).

Accessibility
Make your app accessible.
[Visit](#)

Animation and Motion
Bring animations to your app.
[Visit](#)

Assets, Images, and Icons
Manage assets, display images, and show icons.
[Visit](#)

Async
Async patterns to your Flutter application.
[Visit](#)

Basics
Widgets you absolutely need to know before building your first Flutter app.
[Visit](#)

Cupertino (iOS-style widgets)
Beautiful and high-fidelity widgets for current iOS design language.
[Visit](#)

Data & backend

Accessibility & internationalization

Platform integration

Packages & plugins

Add Flutter to existing app

Tools & techniques

Migration notes

<https://docs.flutter.dev/ui/widgets>

Learn the Basic widgets first



Flutter 3.29 is here with a bouquet of performance and fidelity improvements for your apps! Learn more

Flutter Docs

Basic widgets

UI > Widgets > Basics

Widgets to know before building your first Flutter app.

AppBar Container that displays content and actions at the top of a screen.	Column Layout a list of child widgets in the vertical direction.	Container A convenience widget that combines common painting, positioning, and sizing widgets.
ElevatedButton A Material Design elevated button. A filled button whose material elevates when pressed.	FlutterLogo The Flutter logo, in widget form. This widget respects the IconTheme.	Icon A Material Design icon.



Using Scaffold()

Wrapping your content inside the Scaffold() widget for better lay-out options

Scaffold class



Flutter > material.dart > Scaffold class

Search API Docs

material library

CLASSES

- AboutDialog
- AboutListTile
- AbsorbPointer
- Accumulator
- Action
- ActionChip
- ActionDispatcher
- ActionIconTheme
- ActionIconThemeData
- ActionListener
- Actions
- ActivateAction
- ActivateIntent
- Adaptation
- AdaptiveTextSelectionT...
- AlertDialog
- Align
- Alignment
- AlignmentDirectional
- AlignmentGeometry
- AlignmentGeometryTwe...
- AlignmentTween
- AlignTransition
- AlwaysScrollableScrollP...
- AlwaysStoppedAnimation

Scaffold class

Implements the basic Material Design visual layout structure.

This class provides APIs for showing drawers and bottom sheets.

To display a persistent bottom sheet, obtain the `ScaffoldState` for the current `BuildContext` via `Scaffold.of` and use the `ScaffoldState.showBottomSheet` function.

This example shows a `Scaffold` with a `body` and `FloatingActionButton`. The `body` is a `Text` placed in a `Center` in order to center the text within the `Scaffold`. The `FloatingActionButton` is connected to a callback that increments a counter.

To create a local project with this code sample, run:

```
flutter create --sample=material.Scaffold.1 mysample
```

The screenshot shows the Flutter DevTools interface with two tabs: "Code" and "Output". The "Code" tab contains the scaffold code. The "Output" tab shows a preview of the app running on an iPhone. The app has a black header bar with the title "Sample Code". The main body is white with a centered text "You have pressed the button 0 times.". A red floating action button is at the bottom right. A red arrow points from the explanatory text above to this button.

CONSTRUCTORS

- `Scaffold`

PROPERTIES

- `appBar`
- `backgroundColor`
- `body`
- `bottomNavigation...`
- `bottomSheet`
- `drawer`
- `drawerDragStartB...`
- `drawerEdgeDragW...`
- `drawerEnableOpe...`
- `drawerScrimColor`
- `endDrawer`
- `endDrawerEnable...`
- `extendBody`
- `extendBodyBehind...`
- `floatingActionButton`
- `floatingActionButt...`
- `floatingActionButt...`
- `hashCode`
- `key`
- `onDrawerChanged`
- `onEndDrawerChan...`
- `persistentFooterAli...`
- `persistentFooterB...`
- `primary`

<https://api.flutter.dev/flutter/material/Scaffold-class.html>

Using Scaffold()



- The Scaffold() widget implements the [basic Material Design](#) visual lay-out structure
- It can [hold other widgets](#) like
 - AppBar()
 - Drawer()
 - SnackBar()
 - BottomSheet(), and many more
- It has properties like
 - backgroundColor: ...
 - body: ...
 - ...

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

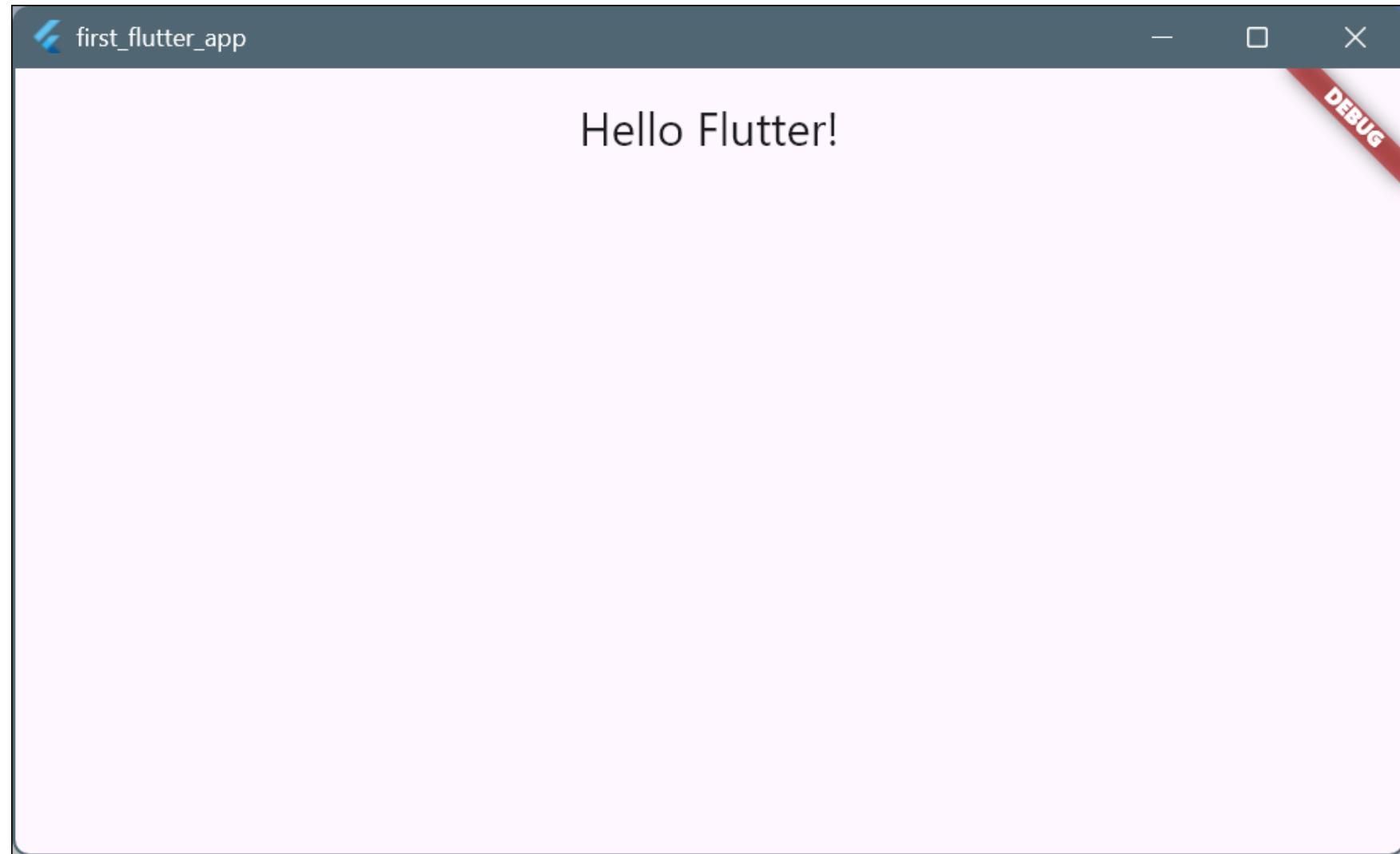
void main() {
  runApp(MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: Text('Hello Flutter'),
        centerTitle: true,
      ),
    )
  )));
}
```

Inside Scaffold()



- Again: `don't` just type text or add images inside a Scaffold Widget
 - Everything must be `wrapped` inside its own widget
- In the previous example `appBar: AppBar(...):`
 - `AppBar`: is the property
 - `AppBar(...)` is the widget
- The `AppBar` Widget has a `property` `title` which holds a content `Widget` `Text(...)` as its value
- It has a property `centerTitle`: whose value is `true`

Result: still not fantastic, but much better



“Every Flutter App

is composed as a

tree of widgets”

Adding a body



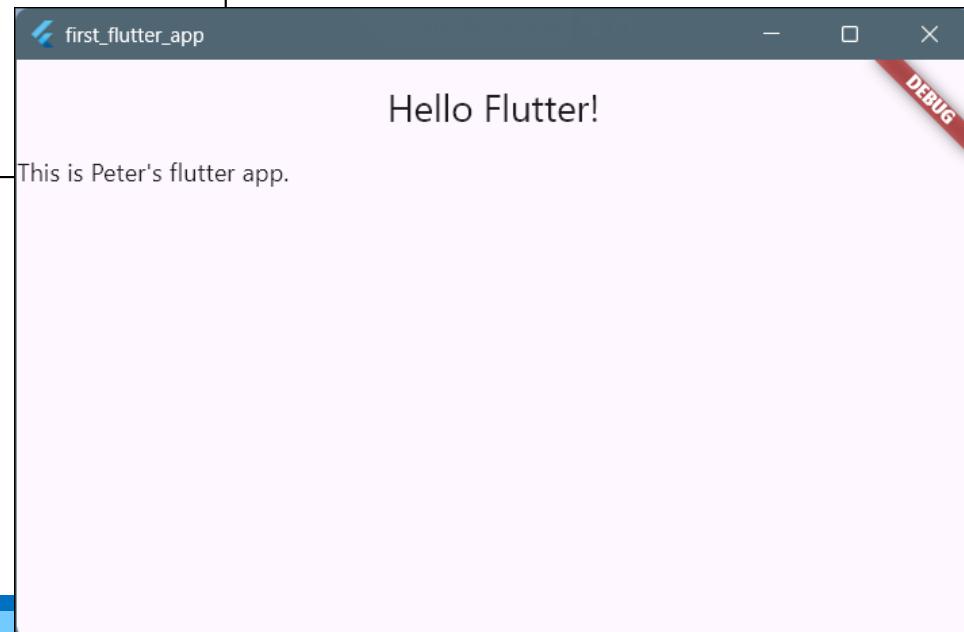
- All content below the `AppBar()` goes into the `body:` property
- This – again – is also a tree of Widgets.
- Convention:
 - Properties start with a `lowercase letter` and then `camelCase`
 - Widgets start with an `Uppercase Letter` and then `PascalCase`
 - Like in `appBar: AppBar()`
- A widget *always* is followed by `(...)`,

Expanding the app



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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: Text('Hello Flutter'),
        centerTitle: true,
      ),
      body: Text('This is Peter\'s Flutter app'),
    )
  ));
}
```



Some background information



- Old: using the `new` keyword:
 - `title: new Text('Hello Flutter'),`
 - This still works, but is `not required` anymore
- New: using `anonymous` instances
 - `title: Text('Hello Flutter'),`
- IntelliJ/Android Studio automatically places`// comment if closing a Widget.`
 - These are `not part` of the code
- Flutter refreshes its widget tree at `60fps`

Comments added by Flutter in your IDE



```
        arguments: {
            'name': countries[index]['name']['common'],
        },
    },
),
),
// ListTile
],
// <Widget>[]
),
// Column
);
// Padding
},
),
// ListView.builder
),
// Expanded
],
// <Widget>[]
),
// Column
);
// Scaffold
}
```

Placed by IntelliJ - you
can not remove them!
(and, they are not part of
your code)

Tips



- Use [Ctrl+” ” \(space\)](#) to activate IntelliSense in Android Studio
- Use the lightbulb ([Alt+Enter](#)) to quickly wrap Widgets

A screenshot of the Android Studio code editor. The cursor is positioned over the 'backgroundColor' field of an 'AppBar' widget. A dropdown menu from the 'IntelliSense' feature shows various color swatches and names like 'Color', 'MaterialColor', 'Red', etc., for selection.

```
name: Scaffold(  
    appBar: AppBar(  
        title: new Text('Hello Flutter'),  
        centerTitle: true,  
        backgroundColor: ,  
        leading: ,  
        elevation: ,  
        actions: [],  
        actionsIconTheme: ,  
        automaticallyImpliesLeading: ,  
        bottom: ,  
        bottomOpacity: ,  
        brightness: ,  
        excludeHeaderSemantics: .
```

A screenshot of the Android Studio code editor showing a context menu triggered by a lightbulb icon. The menu is titled 'Wrap with Center' and includes options like 'Wrap with widget...', 'Wrap with Center', 'Wrap with Column', 'Wrap with Container', 'Wrap with Padding', 'Wrap with Row', 'Wrap with SizedBox', and 'Wrap with StreamBuilder'. A red arrow points to the 'Wrap with Center' option. The background code shows an 'AppBar' with a 'body' child containing a 'Text' widget.

```
// AppBar  
body: Text('This is my Flutter app'),  
    Wrap with Center ▾
```

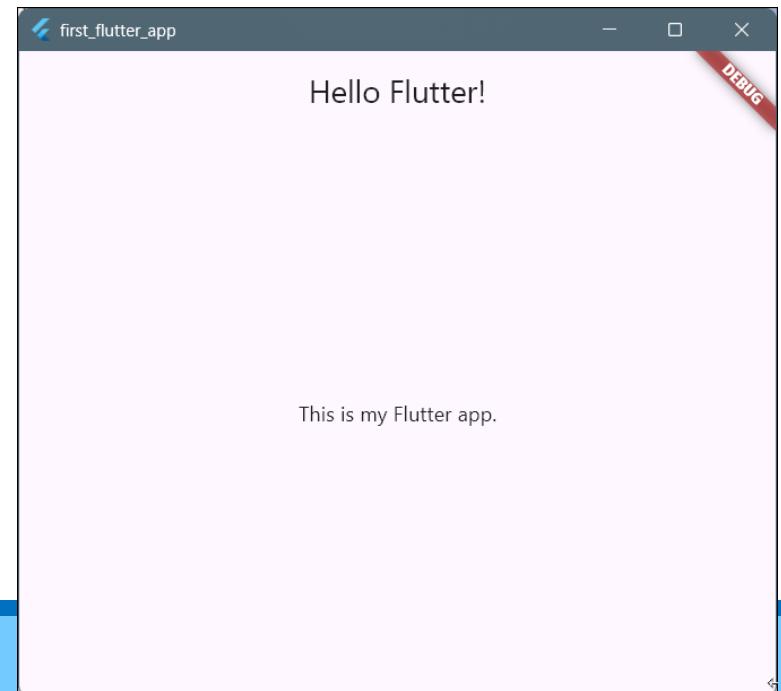
The Center() Widget



- Use Center() to (duhhh) center items in their parent widget
- Center() has a property child that contains the child widget(s)

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: new Text('Hello Flutter'),
        centerTitle: true,
      ),
      body: Center(
        child: Text('This is my Flutter app')
      ),
    ),
  );
}
```

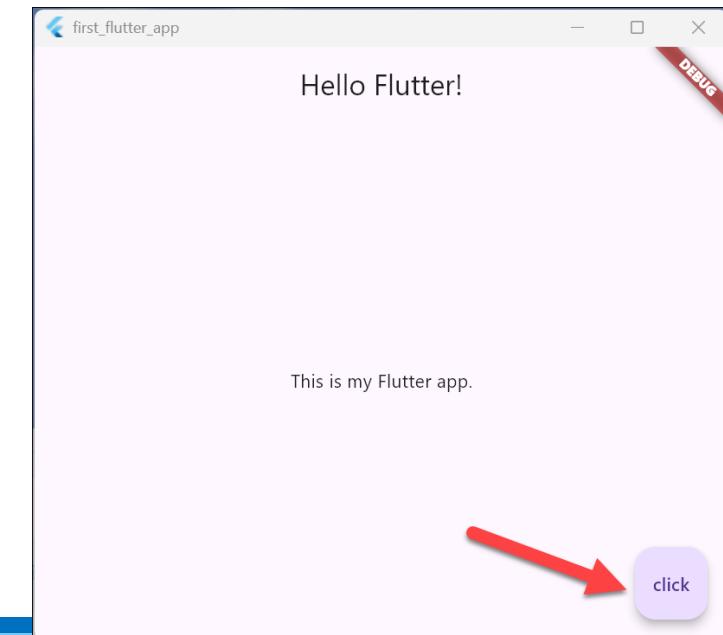


FloatingActionButton()



- Every Scaffold() can have 1 (one!) FloatingActionButton(...)
- The contents of the FAB are other Widgets
- They go inside the child: property

```
void main() {  
  runApp(MaterialApp(  
    home: Scaffold(  
      ...  
      floatingActionButton: FloatingActionButton(  
        child: Text('click'),  
      ),  
    ),  
  );  
}
```



Mandatory properties



- Android Studio might be complaining that `FloatingActionButton` is missing a `required property` `onPressed`
- This function is called if the button is pressed
- Just add an `empty function` for now, to get rid of the error message. We're dealing with functionality later

```
floatingActionButton: FloatingActionButton(  
    onPressed: (){}, // or: null  
    child: Text('click'),  
,
```

More on Scaffold() and other widgets



- See official documentation at

api.flutter.dev/flutter/material/Scaffold-class.html

The screenshot shows the official Flutter API documentation for the `Scaffold` class. The page is titled "Scaffold class" and includes the following sections:

- CLASSES**: A sidebar listing various Flutter classes, with `AboutDialog` currently selected.
- Properties**: A list of properties for the `Scaffold` class, including `appBar`, `backgroundColor`, `body`, `bottomNavigationBar`, `bottomSheet`, `drawer`, `drawerDragStartBehavior`, `drawerEdgeDragWidth`, `drawerEnableOpenDuration`, `drawerScrimColor`, `endDrawer`, `endDrawerEnableOpenDuration`, `extendBody`, `extendBodyBehindAppBar`, `floatingActionButton`, `floatingActionButtonCenter`, `floatingActionButtonDuration`, `hashCode`, `key`, `persistentFooterButtons`, `primary`, `resizeToAvoidBottomInset`, `resizeToAvoidBottomPadding`, and `runtimeType`.
- METHODS**: A list of methods for the `Scaffold` class.

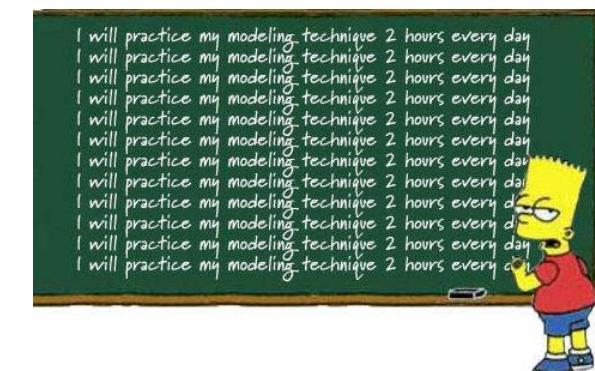
The main content area displays the `Scaffold` class definition and a sample code snippet. The sample code shows a `Scaffold` widget with a `body` containing a `Text` widget and a `FloatingActionButton` at the bottom right. The `Text` widget displays the message "You have pressed the button 0 times.", and the `FloatingActionButton` has a plus sign icon.

At the bottom of the page, a footer notes: "Flutter 1.22.5 • 2020-12-10 22:48 • 7891006299 • stable".

Workshop



- Replace the body of your app with `Scaffold()`
- Create other widgets inside the `Scaffold()`. We practiced with:
 - `AppBar()`
 - `body:`
 - `FloatingActionButton()`
- Look up and practice with other properties, found in the documentation
- Example: `.../110-scaffold`
- Official docs: api.flutter.dev/flutter/material/Scaffold-class.html



Checkpoint



- You know about the Scaffold() Widget and its importance for Flutter apps
- You can add and use widgets inside the Scaffold() widget.
- You know about properties for widgets and where to look them up online.



Setting colors and fonts

Updating the `Text()` Widget with additional properties

Material Design Colors



- Material Design guidelines have a strict coloring system
- material.io/design/color/the-color-system.html#tools-for-picking-colors

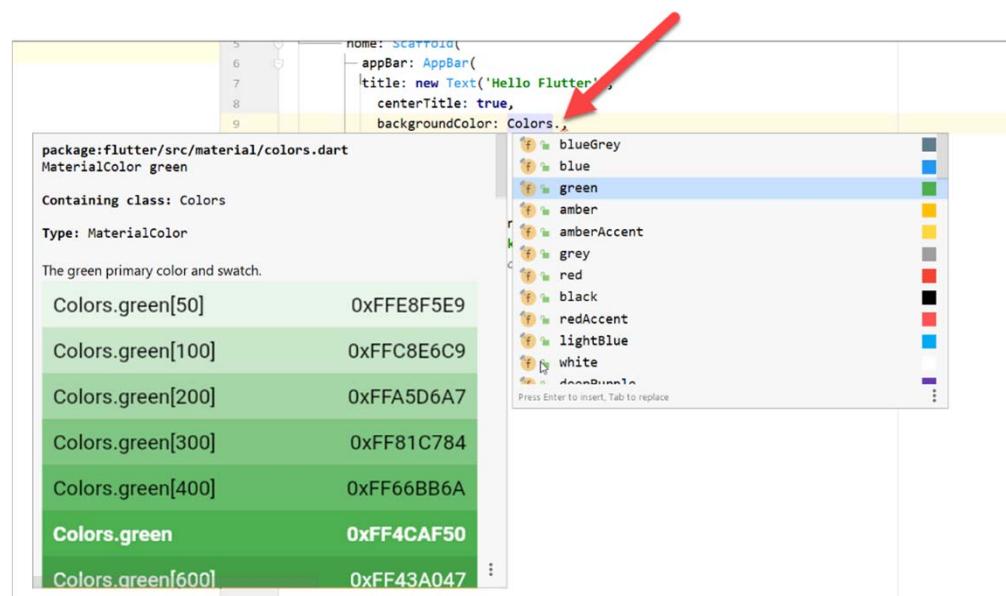
A screenshot of a web-based color palette tool. On the left, there's a section titled "Color palettes" with four categories: PRIMARY, COMPLEMENTARY, ANALOGOUS, and TRIADIC. Each category shows a horizontal color bar with a central circle indicating a specific shade. Below each bar is a numerical scale from 900 to 50. To the right of the palettes is a sidebar titled "CONTENTS" with sections for "Primary color" (showing a purple gradient), "Secondary color" (with a plus sign), "Color usage and palettes", "Color theme creation", and "Tools for picking colors". A color swatch with the hex code "#6002EE" is also present.

Blues, Greens, Purples, and so on. Each different color is available in a variety of shades

Setting the color



- All colors live inside the Colors constant.
 - Don't type the color directly!
- A lot of widgets have the backgroundColor property
- Type Colors. And press `Ctrl+Q` to see a palette



Colors in your UI



- Try to be [consistent](#)
- For instance, using Colors.green[800] on multiple elements
- We can also use a [Theme](#) to be consistent

```
floatingActionButton: FloatingActionButton\(  
    onPressed: (){},  
    backgroundColor: Colors.green[800],  
    child: Text\('click'\),  
),
```

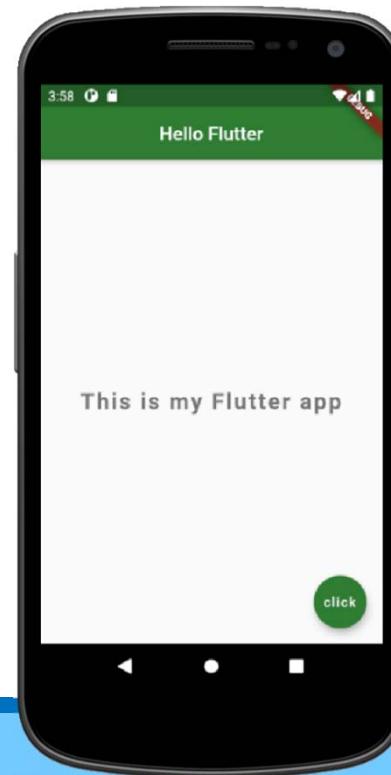


Formatting Text



- Text is formatted via a style property
- Its content is a `TextStyle()` Widget
- Use `Ctrl+Q` or `Ctrl+space` for available options!

```
child: Text(  
  'This is my Flutter app',  
  style: TextStyle(  
    fontSize: 24,  
    fontWeight: FontWeight.bold,  
    letterSpacing: 2,  
    color: Colors.grey[600]  
  ),  
)
```





Using Custom Fonts

Using Custom fonts from the web

Downloading and using custom fonts



- Find, download and extract a font from the internet
 - For instance: <https://fonts.google.com/>
- Create a /fonts folder in your project
 - Move or copy the font there
- Add the font in pubspec.yaml
 - This is like the configuration file for your project.
 - Very important – don't mess up its format!
- Uncomment the fonts section
 - Update the name and location of the font

Updating the Editor



- Pubspec has been edited → Get Dependencies
- Inside the TextStyle() Widget, refer to the just imported fontFamily.

The screenshot shows the Android Studio interface with the following details:

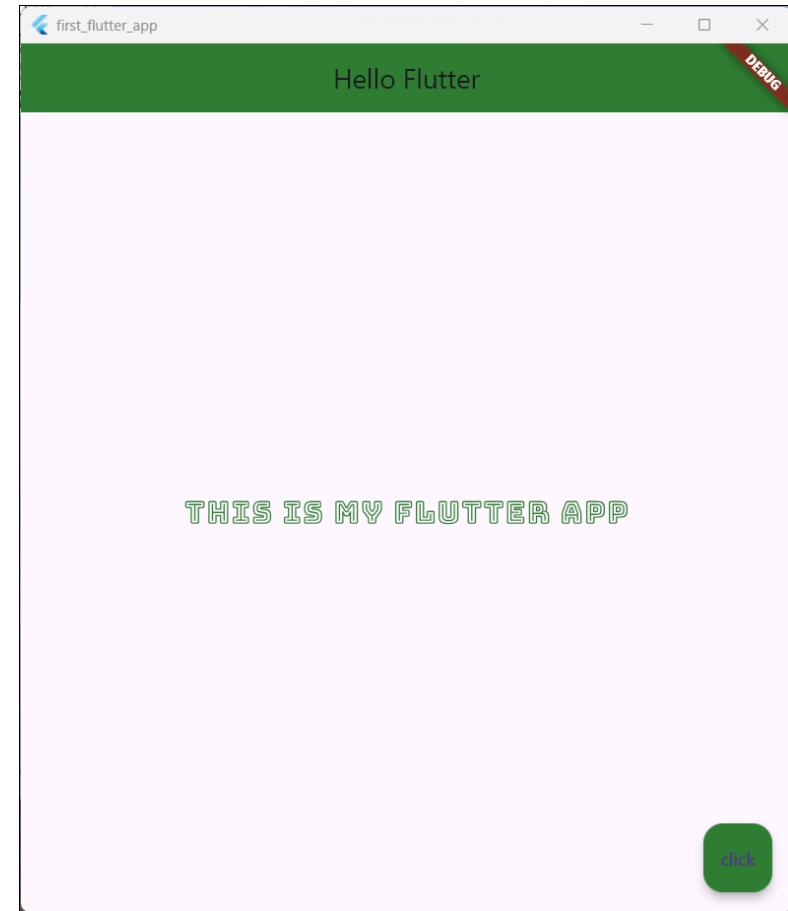
- Project View:** Shows the project structure under "first_flutter_app". It includes .dart_tool, .idea, android, build, fonts (containing BungeeOutline-Regular.ttf), ios, lib (containing main.dart), .gitignore, .metadata, .packages, first_flutter_app.iml, pubspec.lock, pubspec.yaml, README.md, and External Libraries.
- pubspec.yaml Tab:** The file is open in the editor. A red arrow points from the file tree to the line where the font asset is defined in the code.
- Code Editor:** The content of pubspec.yaml is as follows:

```
55 # For details regarding adding assets from package dependencies, see
56 # https://flutter.dev/assets-and-images/#from-packages
57
58 # To add custom fonts to your application, add a fonts section here,
59 # in this "flutter" section. Each entry in this list should have a
60 # "family" key with the font family name, and a "fonts" key with a
61 # list giving the asset and other descriptors for the font. For
62 # example:
63 fonts:
64   - family: BungeeOutline
65     fonts:
66       - asset: fonts/BungeeOutline-Regular.ttf
67
68 # For details regarding fonts from package dependencies,
69 # see https://flutter.dev/custom-fonts/#from-packages
```

Result



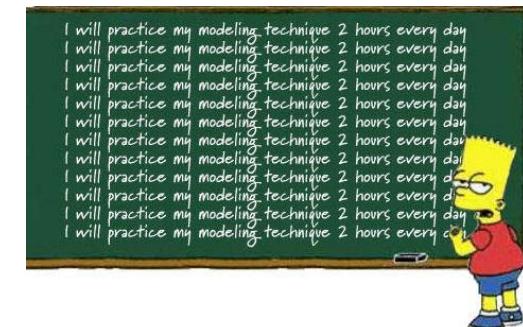
```
child: Text(  
  'This is my Flutter app',  
  style: TextStyle(  
    fontSize: 24,  
    fontWeight: FontWeight.bold,  
    letterSpacing: 2,  
    color: Colors.green[800],  
    fontFamily: 'BungeeOutline'  
  ),  
)
```



Workshop



- **Update** the `AppBar()` Widget to use a different background color.
- **Create** a `TextStyle()` Widget inside your centered text. Use some properties like:
 - `fontSize`, `fontWeight`, `letterSpacing`, `color`
 - api.flutter.dev/flutter/painting/TextStyle-class.html
- Download a **font of your choice** from Google Fonts and use it in your app
- Example: `.../_120-color-font`
- Official docs: api.flutter.dev/flutter/material/Colors-class.html



Checkpoint



- You can use any color of your choice in your app
- You know the meaning of the Colors class
- You are able to download extra fonts from the internet, add them to the project and use them in the app



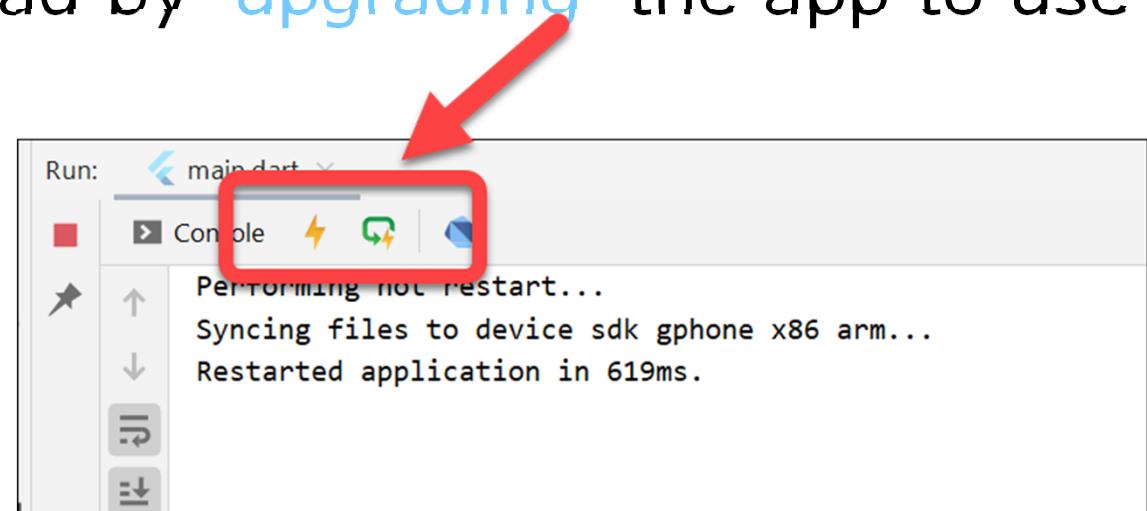
Hot Restart vs. Hot Reload

Refreshing your application without pressing Hot Restart every time

Hot Restart



- If we now want to see changes, we have to press **Hot Restart** every time
 - Because all code is directly inside `void main()`, `runApp()`
- This is not convenient
- We can use Hot Reload by 'upgrading' the app to use a **Stateless Widget**



Hot Reload



- Creating a stateless Widget
 - Type `stless` snippet and type Tab or Enter to expand
- You can call it anything you want

```
// Our own class, extending the base StatelessWidget class
class Home extends StatelessWidget {
  const Home({super.key});

  @override
  Widget build(BuildContext context) {
    return const Placeholder();
  }
}
```

Stateless vs. Stateful widgets



Stateless Widgets

- The state of the class **can not** change over time
- Fast – no change detection needed

Stateful Widgets

- The state of the widget **can change** over time
- Widget is rerendered every time the state has changed
 - e.g. the `build()` function is called and executed automatically.

(**state** == not just user input or variables, also colors, fonts, etc)

Return type of Widgets



- A widget *has* to return a Widget Tree
 - By default the snippet is returning an empty Placeholder()
- So we can move the Home screen layout to our newly created StatelessWidget widget
 - i.e. everything inside the Scaffold() widget
- If we now change something (development time), Flutter can rebuild – and Hot Reload – the widget.
 - Not the complete app, just the changed widget

Result



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

void main() => runApp(MaterialApp(
    home:Home() ←
)
);

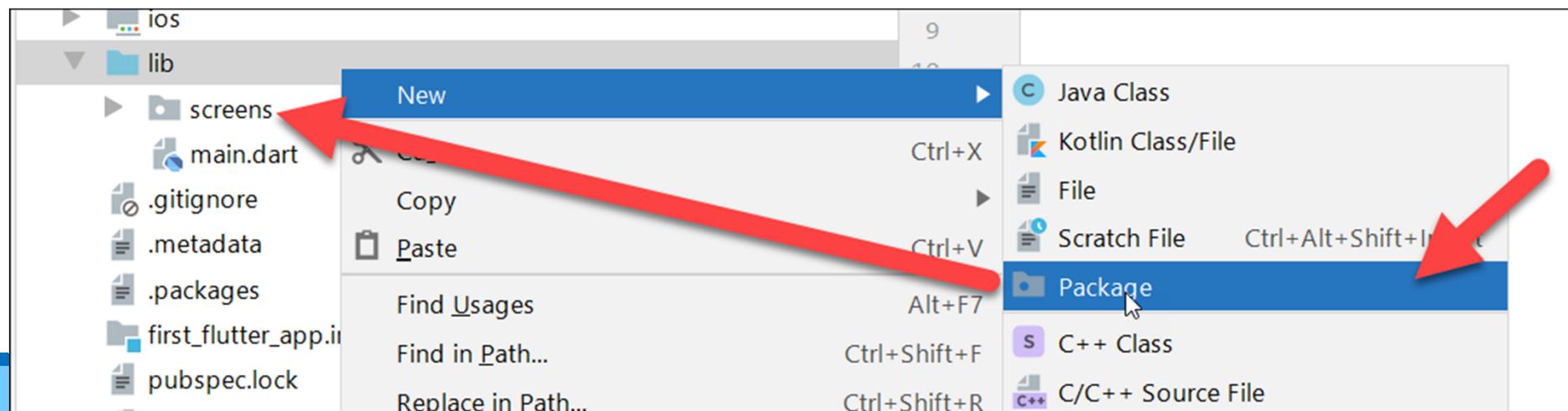
// Our own class, extending the base StatelessWidget class
class Home extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Scaffold(←
            appBar: AppBar(...)
            body: Center(...)
            floatingActionButton: FloatingActionButton(...)
        );
    }
}
```

Try to change something
in the `Home()` class – the
widget will Hot Reload

D.R.Y – extracting widgets



- We can now also **extract** our widget to a separate file and reuse it – D.R.Y.
 - Not very useful for the Home Screen, just showing the procedure
- Convention:
 - Put screens in a \screens folder
 - Folders are called **Packages** in Android Studio



Extracting Widgets



```
// home.dart - the Home Screen
import 'package:flutter/material.dart';

// Our own class, extending the base StatelessWidget class
class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(...),
      body: Center(...),
      floatingActionButton: FloatingActionButton(...),
    );
  }
}
```



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

// Import the screen
import 'screens/home.dart';

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

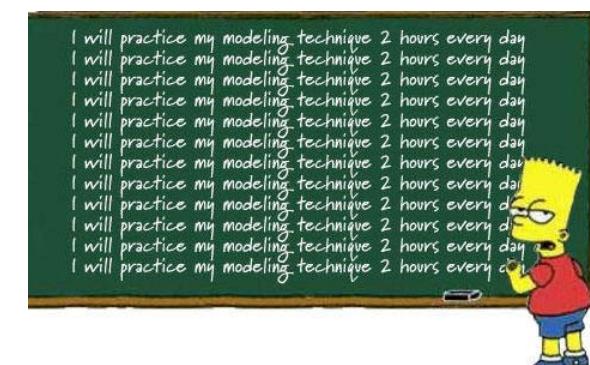


*You don't **have** to extract widgets,
but it keeps the `main.dart` file
simple and tidy*

Workshop



- **Create** a StatelessWidget and move the logic to that widget
- Make sure **Hot Reload** works as intended
- Move the widget to its own file and directory (Package) and import it in main.dart
- Example: `.../_130-stateless-widget`
- Official docs: api.flutter.dev/flutter/widgets/StatelessWidget-class.html



Checkpoint



- You know the difference between Hot Start and Hot Reload.
- You know you need (at least one) StatelessWidget to enable Hot Reload
- You can create and/or extract code to its own StatelessWidget() class.
- You know about the shortcut for StatelessWidget's in your code
- You can extract widgets to their own file and import them where they are needed



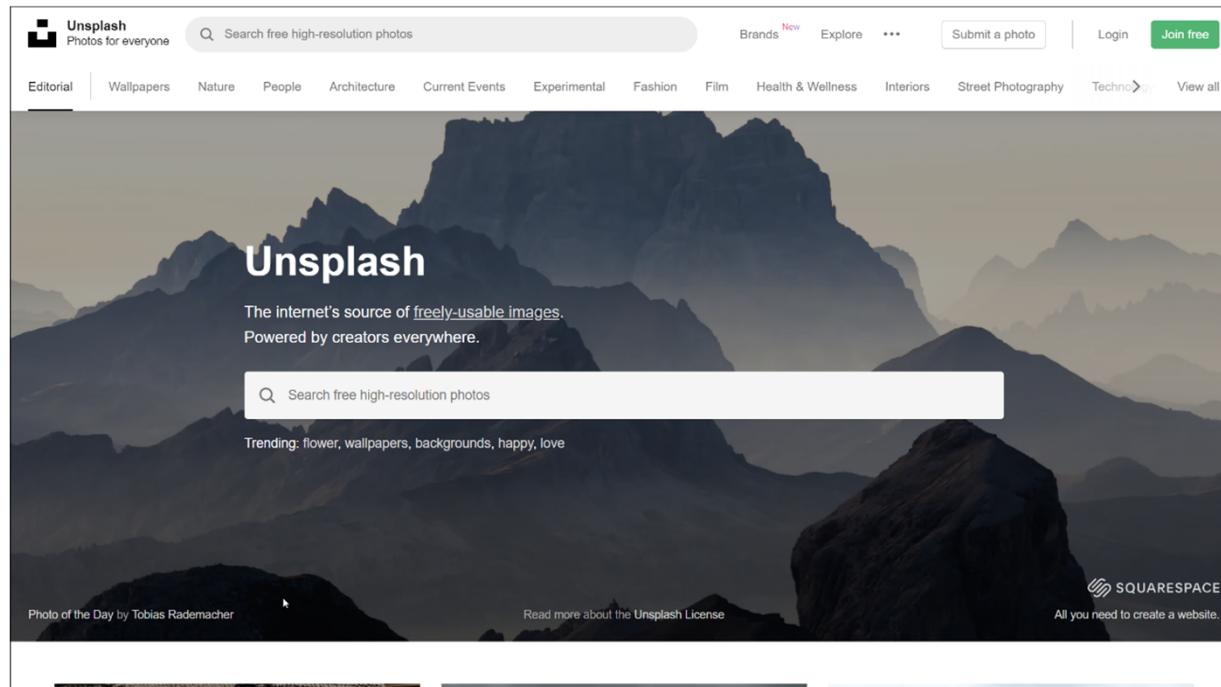
Images and Assets

Using images from the web and from within your application

Using Images



- Flutter knows two (2) types of images
 - `NetworkImage()` – images hosted on the web somewhere
 - `AssetImage()` – images bundles with your application



<https://unsplash.com/>

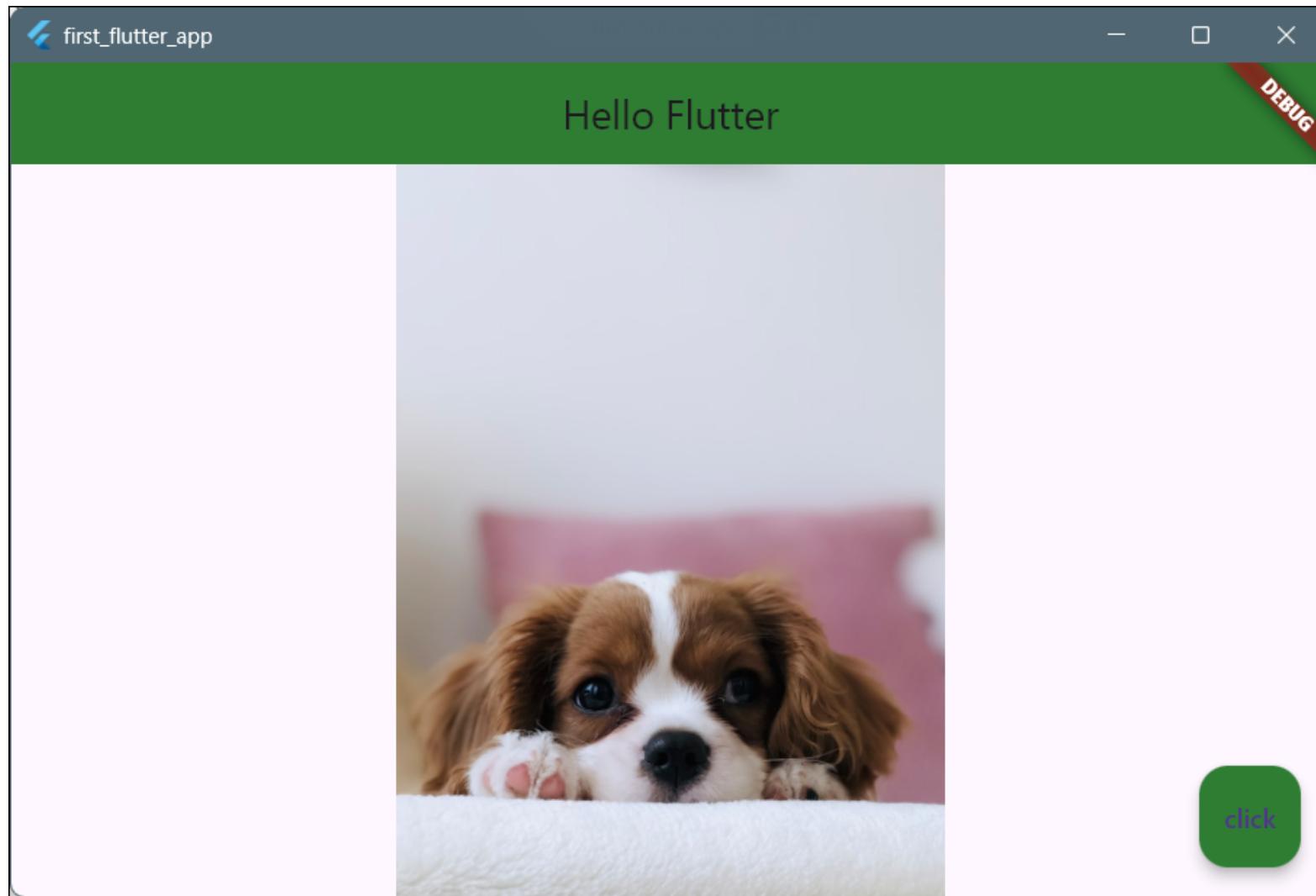
Using NetworkImage()



- Use the `Image()` widget and the `image:` property
- The value of the `image:` property is a `NetworkImage()` widget
- Inside: just copy the URL.

```
body: Center(  
    child: Image(  
        image: NetworkImage('https://images.unsplash.com/...'),  
    )  
,
```

Result

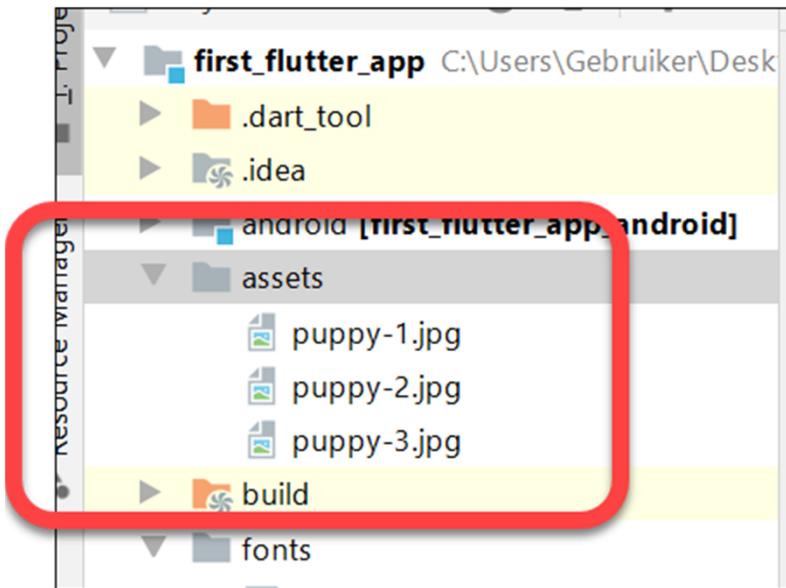


Using AssetImage()



- An AssetImage() is an image [inside your application](#)
 1. [Create](#) a directory /images or /assets in project
 2. [Copy](#) images there
 3. [Update](#) pubspec.yaml to bundle the assets
 4. [Use](#) the local image URL in your widget

Update project and pubspec.yaml



```
46
47      # To add assets to your application, add an assets section, like thi
48  assets:
49      - assets/puppy-1.jpg
50       - assets/puppy-2.jpg
51      - assets/puppy-3.jpg
52
53      # An image asset can refer to one or more resolution-specific "variants"
```

`Pubspec.yaml` – uncomment assets section and point to correct files.

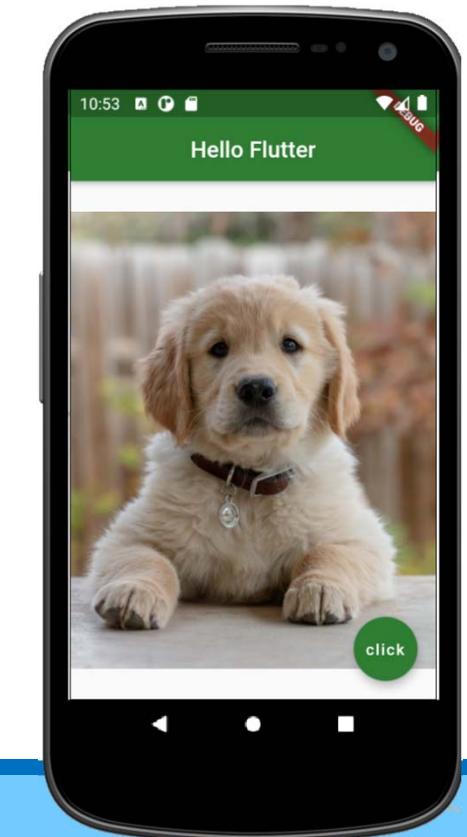
Click Pub get (!) and/or Get dependencies to update the project.

Using AssetImage()



- Make sure pubspec.yaml has correct layout/number of spaces!
- Use AssetImage() in the body of your widget

```
body: Center(  
    child: Image(  
        image: AssetImage('assets/puppy-1.jpg')  
    )  
,
```



If you have *a lot* of images



- Don't point to every single image in pubspec.yaml
- Point to the [complete directory](#) instead
 - Pro : faster
 - Con: you might bundle unnecessary images

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

Shortcuts for images



- Instead of using the `Image()` widget, you can use a shortcut
 - `Image.asset('local-url')` for `AssetImage()`
 - `Image.network('http-url')` for `NetworkImage()`

```
body: Center(  
    // Simply use the image Shortcuts  
    child: Image.asset('assets/puppy-2.jpg'),  
,
```



Loading more images



- So far, we're just loading one (1) image at a time
- Yes – I *know* you want to load more images in an array, or List or something :-).

We're covering that later

- Make sure you now understand [how to work with images](#)

Workshop



- Remove the `Text()` from your `StatelessWidget` and replace it with an `Image()` widget
- Use `NetworkImage()` and get it to work
- Add 3 images to your app in an `/assets` folder and use them as `AssetImage()`
- Example: `.../_140-images`
- Official docs: api.flutter.dev/flutter/widgets/Image-class.html





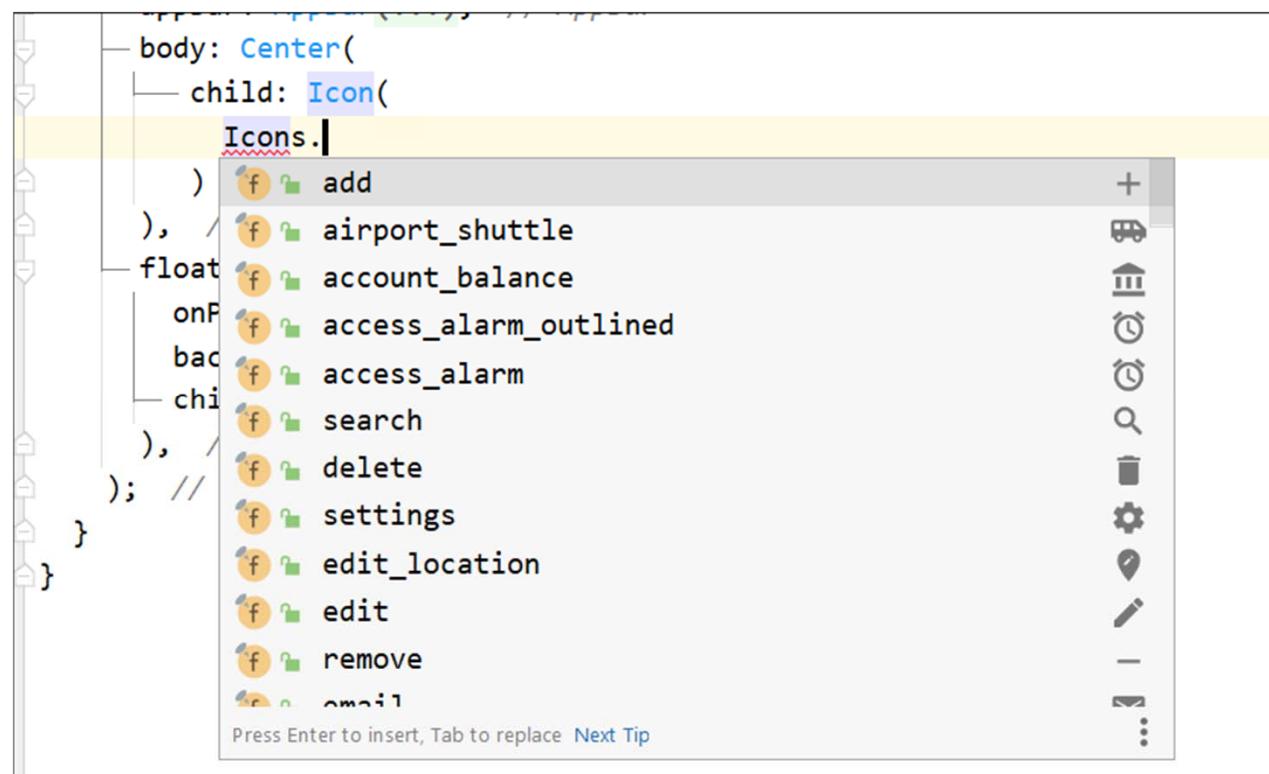
Icons & Buttons

Using Icons and Buttons in your application

Using Icons



- Flutter has *a lot* of built-in Icons
- Part of the material design-spec, using the `Icon()` widget

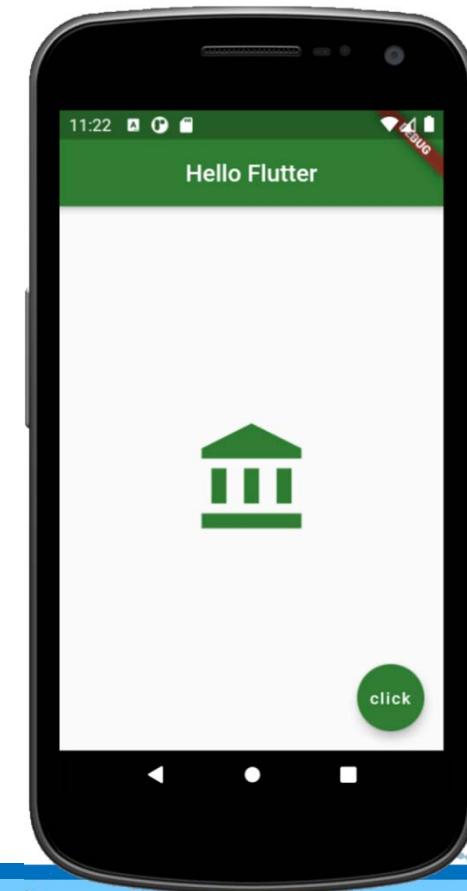


Properties for Icons



- Except the Icon itself, you can use properties:
 - color, size, key, textDirection, semanticLabel

```
body: Center(  
    child: Icon(  
        Icons.account_balance,  
        color: Colors.green[800],  
        size: 100,  
    ),  
,
```



Buttons



- Different types of buttons
 - `TextButton()`
 - `ElevatedButton()`
 - `OutlinedButton()`
 - `IconButton()`
 - `FloatingActionButton()`
 - `FilledButton()`
- Deprecated (but still working)
 - `FlatButton()`
 - `RaisedButton()`
 - `OutlineButton()`

PUBLICLY SHARED

Flutter
Migrating to the New Material Buttons and their Themes

You're suggesting

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`.

[Google Docs document on migrating buttons](#)

When to Use Which Button?



- Use `TextButton()` for actions like "Cancel" or "Skip"
- Use `ElevatedButton()` for calls to action like "Save" or "Submit"
- Use `OutlinedButton()` if you want to provide a secondary option, e.g., "Learn More"
- Use `IconButton()` to represent quick actions such as "Search," "Settings," or similar.
- Use `FloatingActionButton()` for the most prominent action on the screen, like "Add" or "Compose"

Button properties

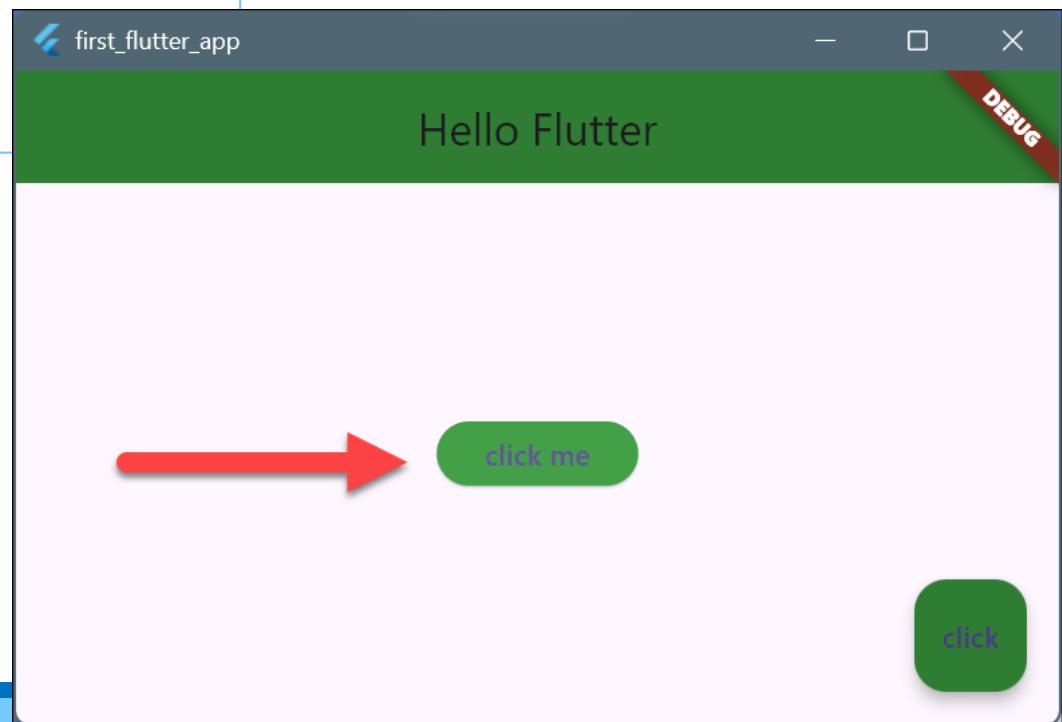


- A Button *MUST* have the following properties
 - onPressed: () {...} – or null, even if the function is empty
 - child: ... – to show something on the button
- A Button *CAN* have
 - style: ButtonStyle (...) Widget with **lots** of different formatting options
 - api.flutter.dev/flutter/material/ButtonStyle-class.html
 - More finegrained control over lay-out than the previous button types but also (much) more complex.

Button example



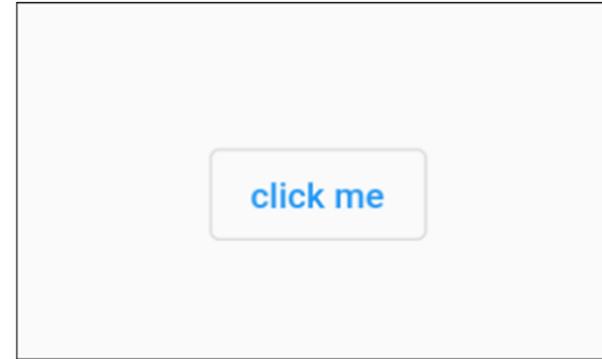
```
body: Center(  
    child: ElevatedButton(  
        onPressed: (){},  
        child: Text('click me'),  
        style: ElevatedButton.styleFrom(  
            background: Colors.green[600]  
        )  
    ),  
)
```



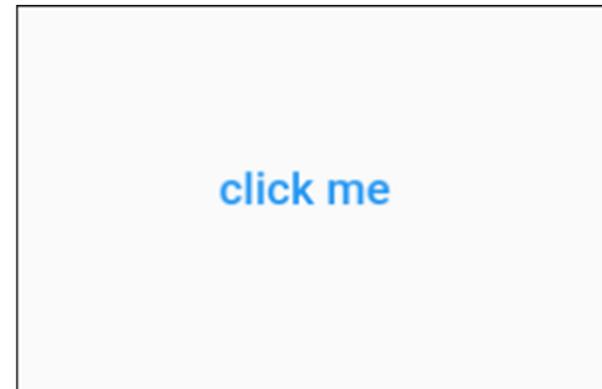
More button types



```
body: Center(  
    child: OutlinedButton(  
        onPressed: (){},  
        child: Text('click me'),  
    )  
,
```



```
body: Center(  
    child: TextButton(  
        onPressed: (){},  
        child: Text('click me'),  
    )  
,
```



Using the onPressed() option



- We are now using a StatelessWidget
 - So onPressed **cannot** really change the widget
- But we can send text to the (Android Studio) Console

```
body: Center(  
    child: ElevatedButton(  
        onPressed: (){  
            print('You clicked the ElevatedButton');  
        },  
        child: Text('click me'),  
    ),  
,
```



Logging to the console

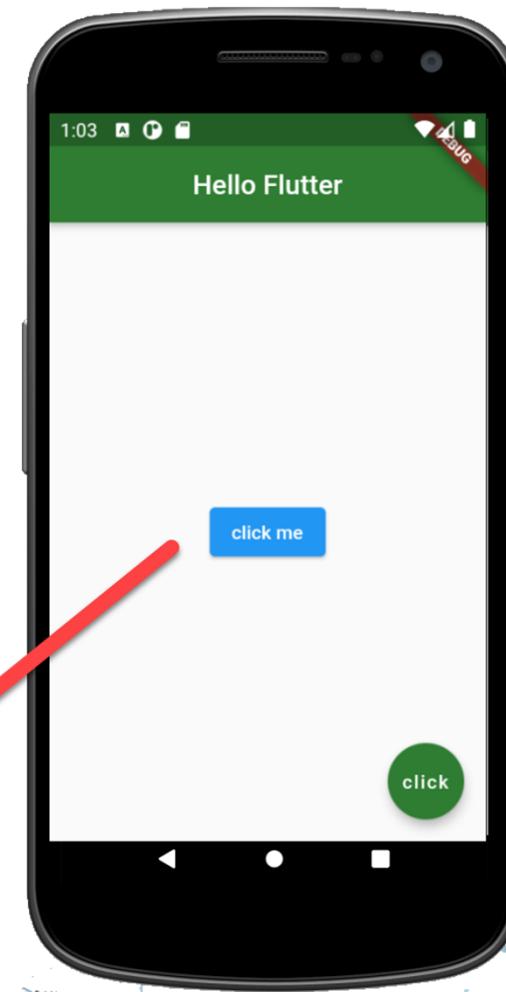


In this case, the console is in your IDE, or
in your terminal. NOT in the browser, or
in the (Windows) app



```
Run: main.dart ×
Console ⚡ 🚀 | 🔍 | ⚙️
Performing hot reload...
Syncing files to device sdk gphone x86 arm...
Reloaded 2 of 530 libraries in 307ms.
I/flutter (18367): You clicked the ElevatedButton
I/flutter (18367): You clicked the ElevatedButton
```

6: Logcat Database Inspector Profiler TODO Terminal 0: Messages Dart An

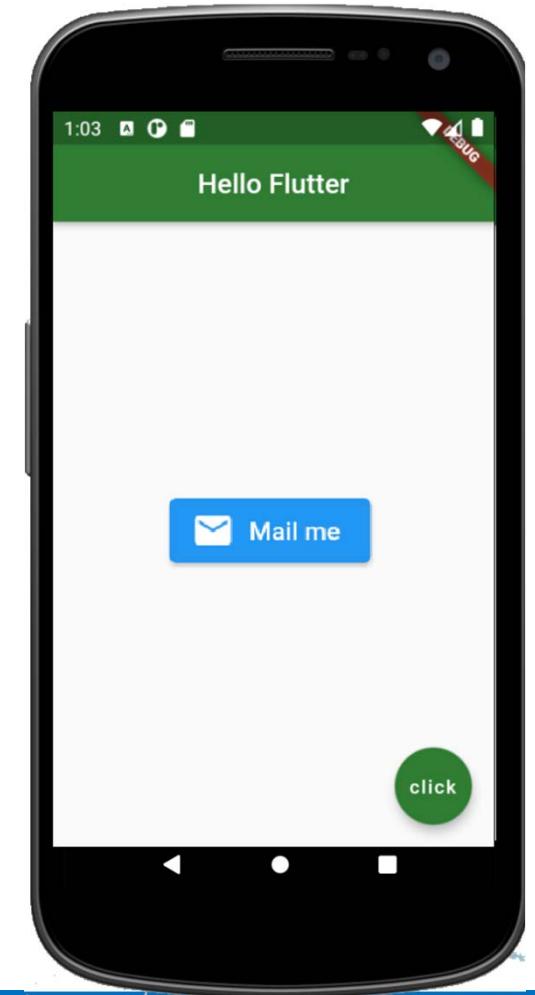


Icons on buttons



- Simply add the icon property to the button
- But notice the different properties
- Inside the button, you can format the Text() any way you want of course

```
body: Center(  
    child: ElevatedButton.icon(  
        onPressed: () {  
            print('You clicked the Mail Icon Button');  
        },  
        icon: Icon(Icons.mail),  
        label: Text('Mail me'),  
    ),  
,
```



An Icon as a button



- Not an Icon *on* a button, but a pressable Icon itself
- Use the IconButton() widget

```
body: Center(  
    child: IconButton(  
        onPressed: (){  
            print ('you clicked the IconButton');  
        },  
        icon: Icon(Icons.alternate_email),  
        color: Colors.green[600],  
        iconSize: 100.0,  
    ),  
,
```



Lots of properties available



- We used just a **very small subset** of properties
- Always check the documentation!

Flutter > material > ElevatedButton class

Search API Docs

ElevatedButton class

A Material Design "elevated button".

Use elevated buttons to add dimension to otherwise mostly flat layouts, e.g. in long busy lists of content, or in wide spaces. Avoid using elevated buttons on already-elevated content such as dialogs or cards.

An elevated button is a label `child` displayed on a `Material` widget whose `Material.elevation` increases when the button is pressed. The label's `Text` and `Icon` widgets are displayed in `style`'s `ButtonStyle.foregroundColor` and the button's filled background is the `ButtonStyle.backgroundColor`.

The elevated button's default style is defined by `defaultStyleOf`. The style of this elevated button can be overridden with its `style` parameter. The style of all elevated buttons in a subtree can be overridden with the `ElevatedButtonTheme`, and the style of all of the elevated buttons in an app can be overridden with the `Theme`'s `ThemeData.elevatedButtonTheme` property.

The static `styleFrom` method is a convenient way to create a elevated button `ButtonStyle` from simple values.

If `onPressed` and `onLongPress` callbacks are null, then the button will be disabled.

See also:

- `TextButton`, a simple flat button without a shadow.
- `OutlinedButton`, a `TextButton` with a border outline.
- [material.io/design/components/buttons.html](#)

CLASSES

- AboutDialog
- AboutListTile
- AbsorbPointer
- Accumulator
- Action
- ActionChip
- ActionDispatcher
- ActionListener
- Actions
- ActivateAction
- ActivateIntent
- AlertDialog
- Align
- Alignment
- AlignmentDirectional
- AlignmentGeometry
- AlignmentGeometryTween
- AlignmentTween
- AlignTransition
- AlwaysScrollableScrollView
- AlwaysStoppedAnimation

CONSTRUCTORS

- ElevatedButton
- icon

PROPERTIES

- autofocus
- child
- clipBehavior
- enabled
- focusNode
- hashCode
- key
- onLongPress
- onPressed
- runtimeType
- style

METHODS

- createElement
- createState
- debugDescribeChildren
- debugFillProperties
- defaultStyleOf

Inheritance

`Object` > `DiagnosticableTree` > `Widget` > `StatefulWidget` > `ButtonStyleButton` > `ElevatedButton`

Flutter 1.22.5 • 2020-12-10 22:48 • 7891006299 • stable

<https://api.flutter.dev/flutter/material/ElevatedButton-class.html>

Workshop

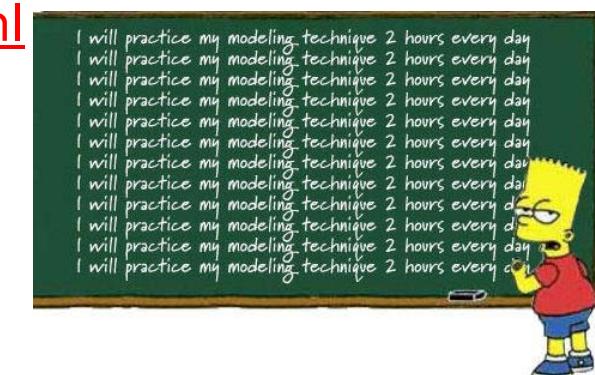


- Create some Buttons in your StatelessWidget
- Check the following types
 - ElevatedButton, TextButton
 - OutlinedButton, IconButton
- Print a text to the console if the button is pressed
- Study the ButtonStyle() class on

api.flutter.dev/flutter/material/ButtonStyle-class.html

and use it for formatting your buttons

- Example: `.../_150-buttons`



Checkpoint



- You know how to create Flutter layouts from scratch
- You know the importance of the Widgets concept
- You can load and use (external) fonts in your app
- You know Hot Reload vs Hot Restart
- You are aware of various types of Buttons
- You can use Icons and Images