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:

```dart
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

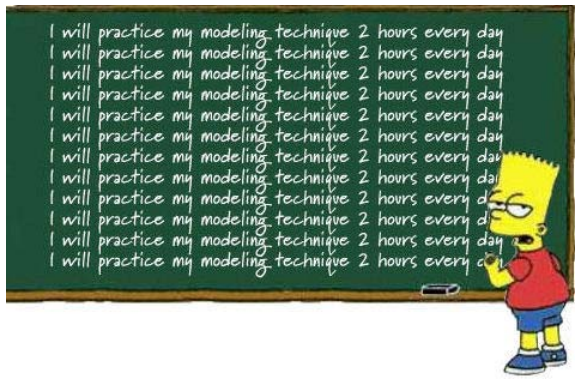
```
void main() {
  runApp(MaterialApp(
    home: Text('Hello Flutter!'),
  ));
}
```

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.

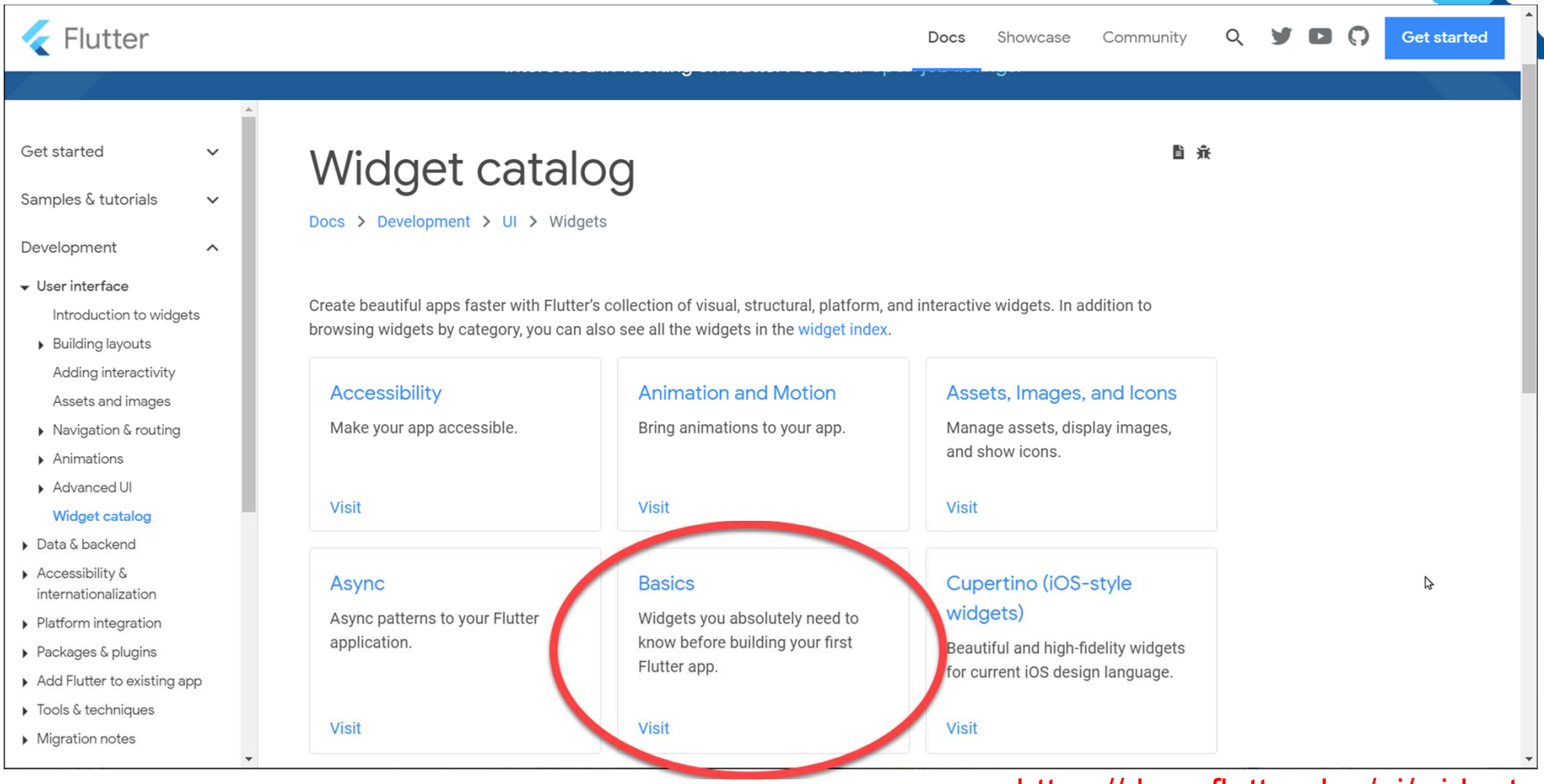
Layout widgets

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

Creating the structure of your lay-out

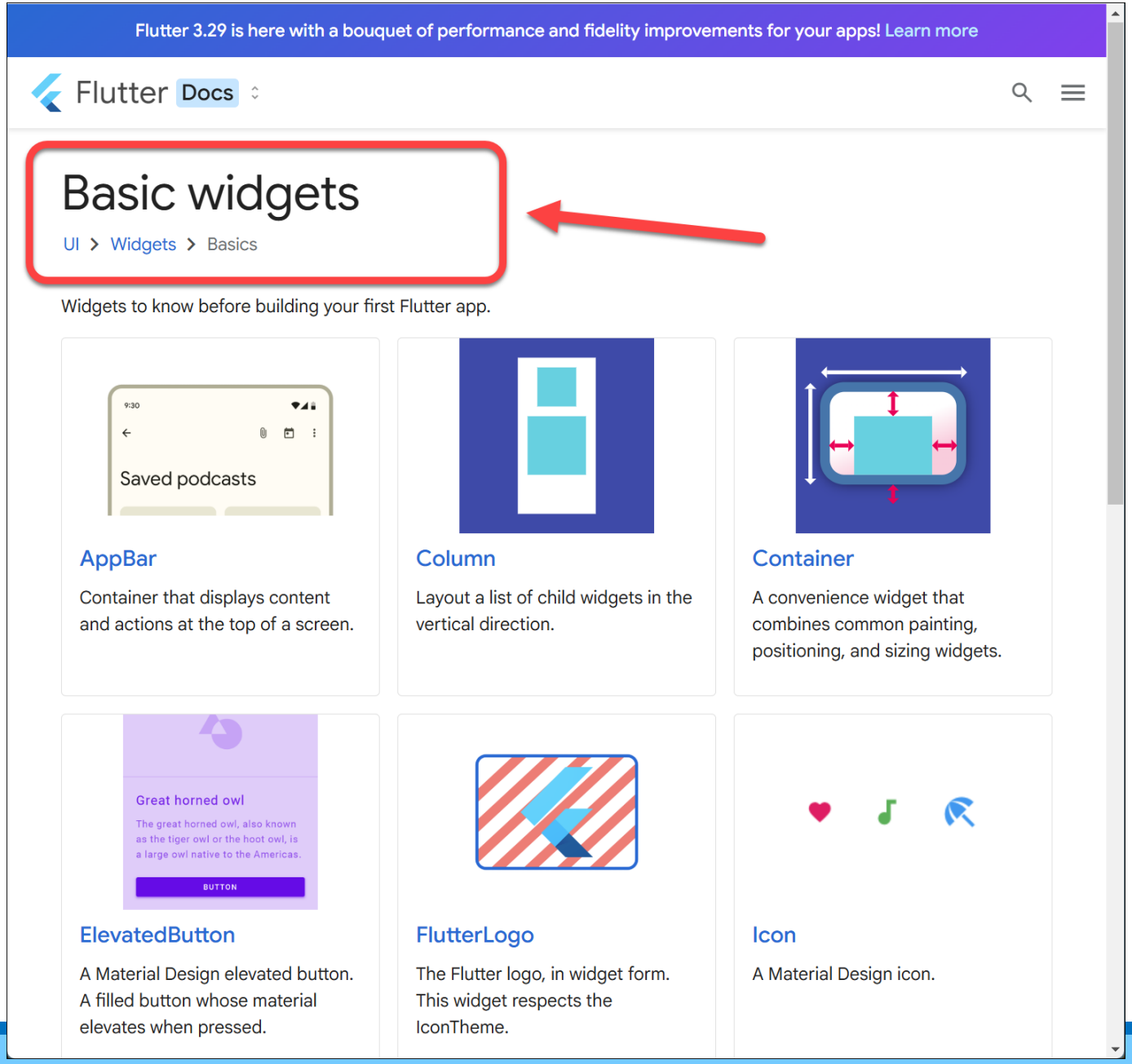
- Scaffold()
- AppBar()
- Themes, fonts & colors
- Images
- Icons
- Rows, Columns

The Widget Catalog



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

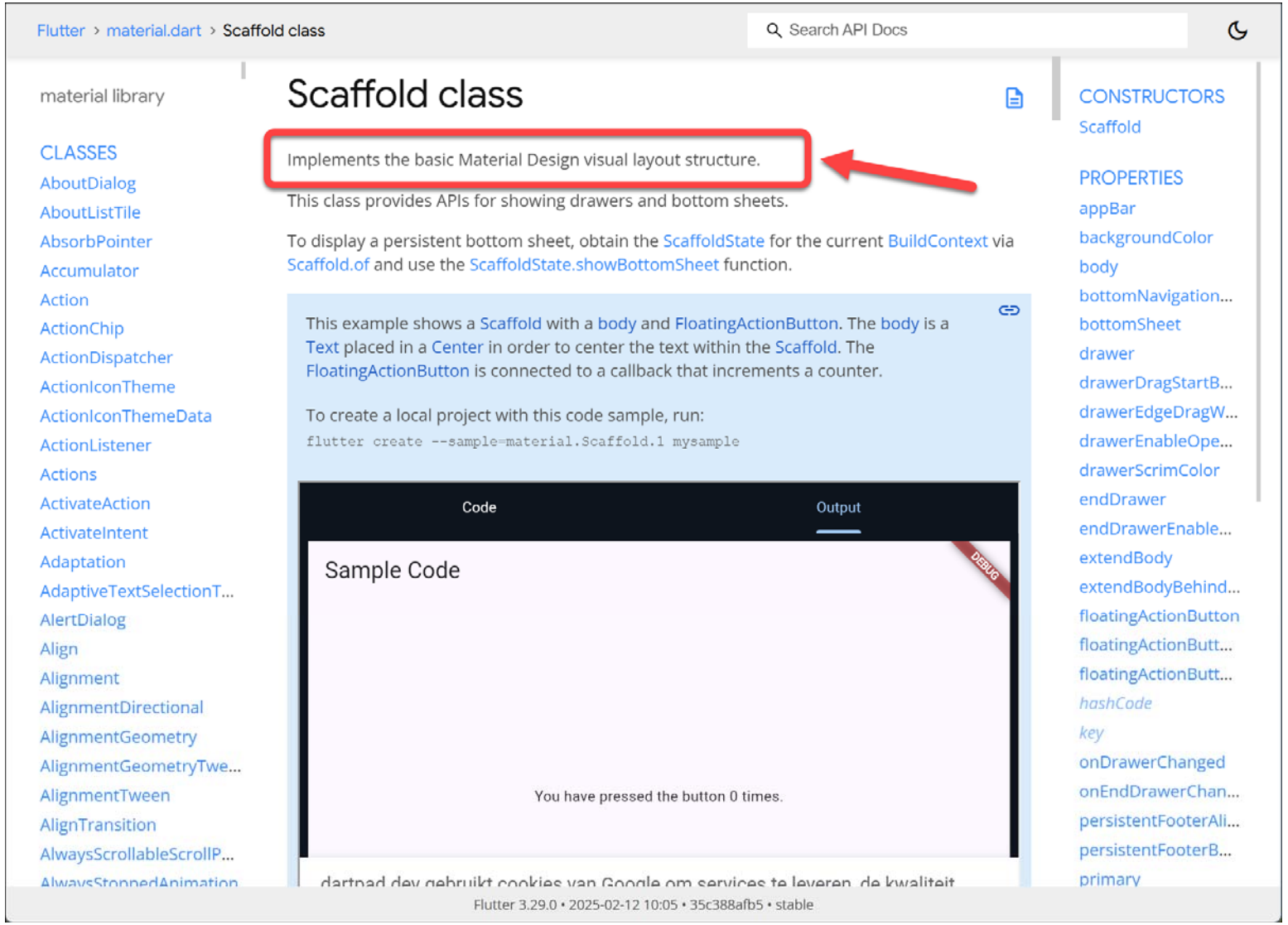
Learn the Basic widgets first



Using `Scaffold()`

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

Scaffold class



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

Using Scaffold()

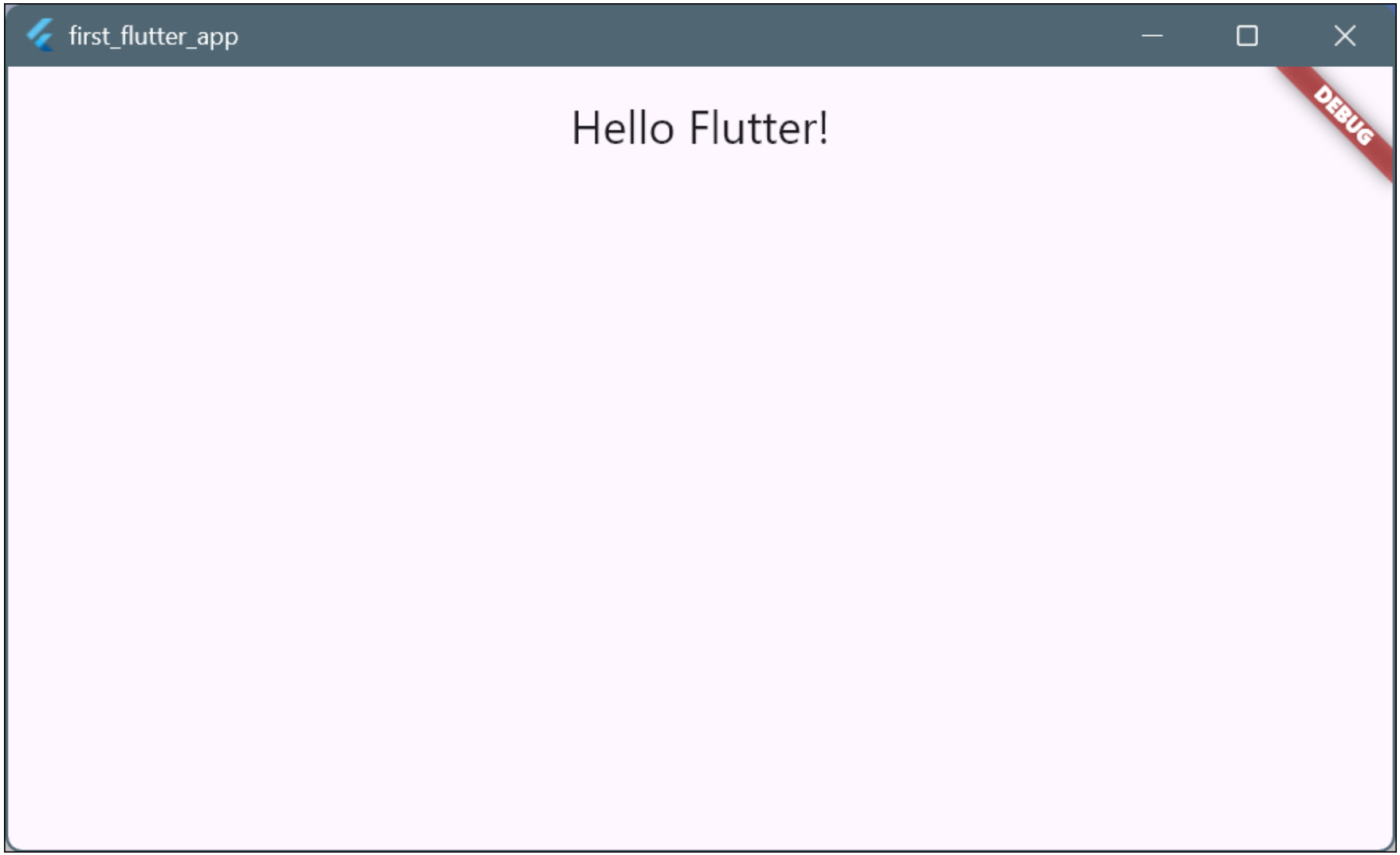
- The `Scaffold()` widget implements the basic Material Design visual lay-out structure
- It can hold other widgets like
  - `AppBar()`
  - `Drawer()`
  - `SnackBar()`
  - `BottomSheet()`, and many more
- It has properties like
  - `backgroundColor:…`
  - `body: …`
  - …

```dart
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` wich 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

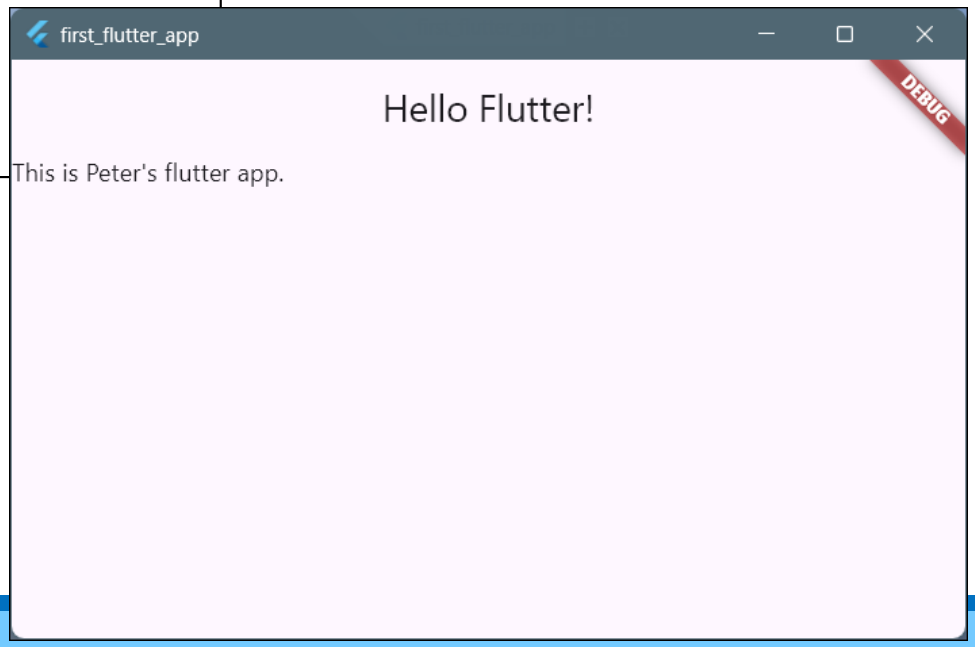
tree of widgets”

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

```dart
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'),
    )
  ));
}
```



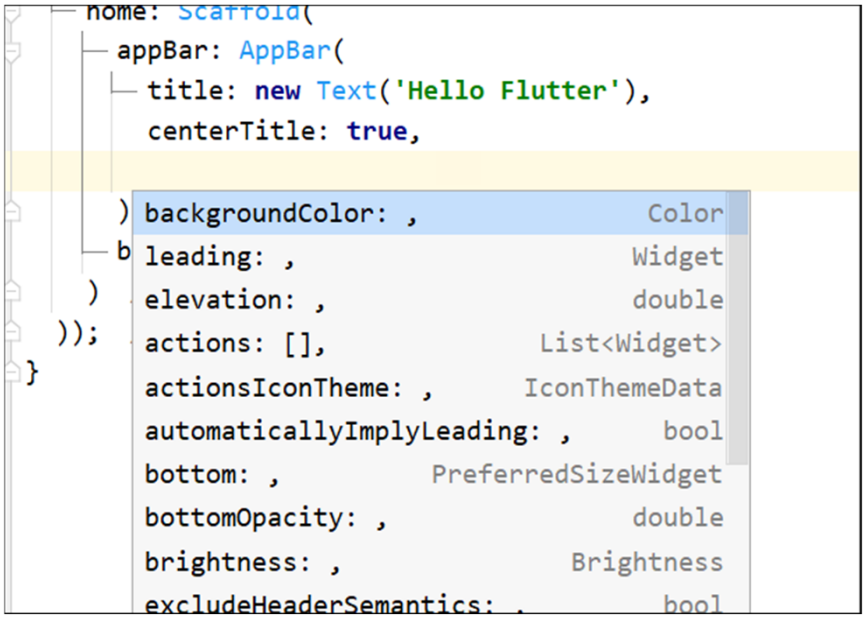
Some background information

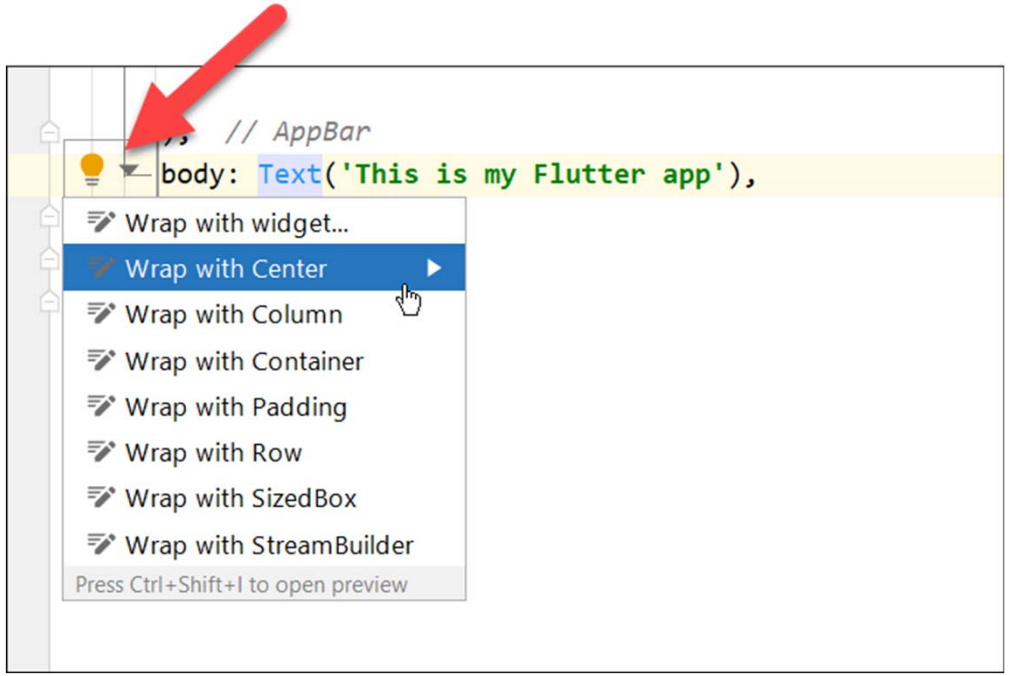
- Old: using the new keyword:
  - `title: new Text('Hello Flutter’),`
  - This still works, but is not required anymore
- New: using anonymous instances
  - `title: Text('Hello Flutter’),`
- IntelliJ/Android Studio automatically places `//comment` if closing a Widget.
  - These are not part of the code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

Comments added by Flutter in your IDE

Tips

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

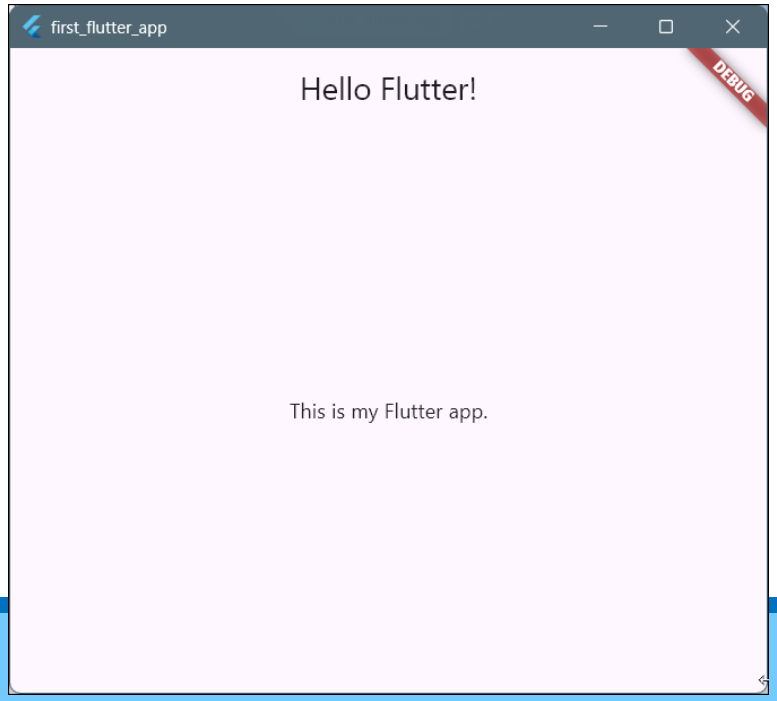




The Center() Widget

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

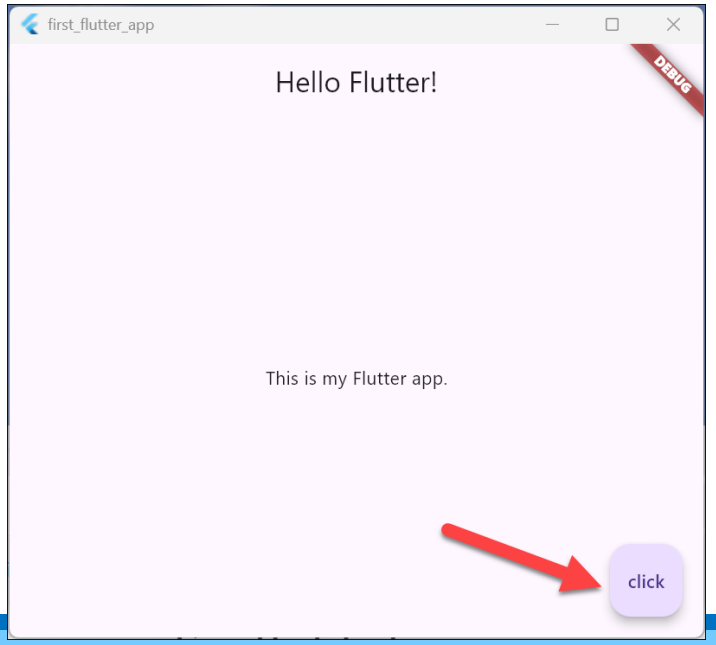
```
import 'package:flutter/material.dart';

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!) `FloatingActioButton(…)`
- 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

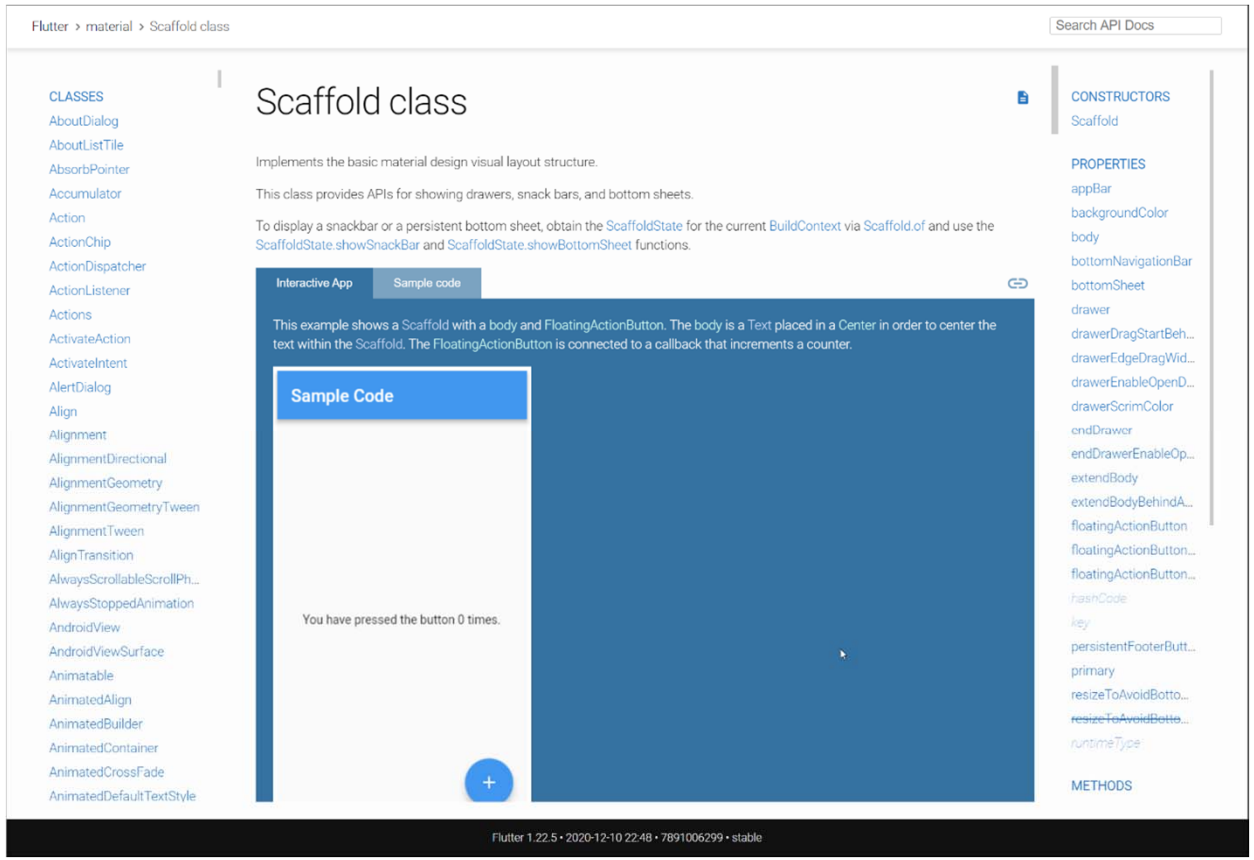
- Android Studio might be complaining that `FloatingActionButton` is missing a required property `onPressed`
- This function is called if the button is pressed
- Just add an empty function for now, to get rid of the error message. We're dealing with functionality later

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

More on Scaffold() and other widgets

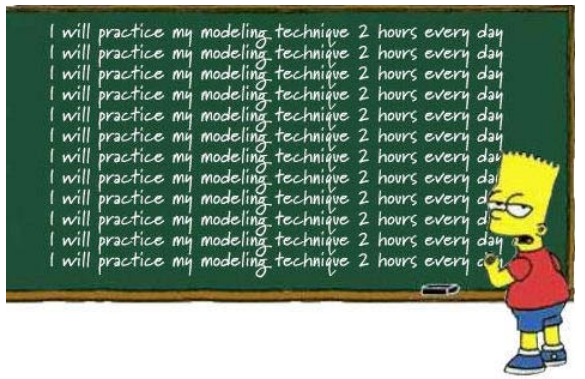
- See official documentation at

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



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



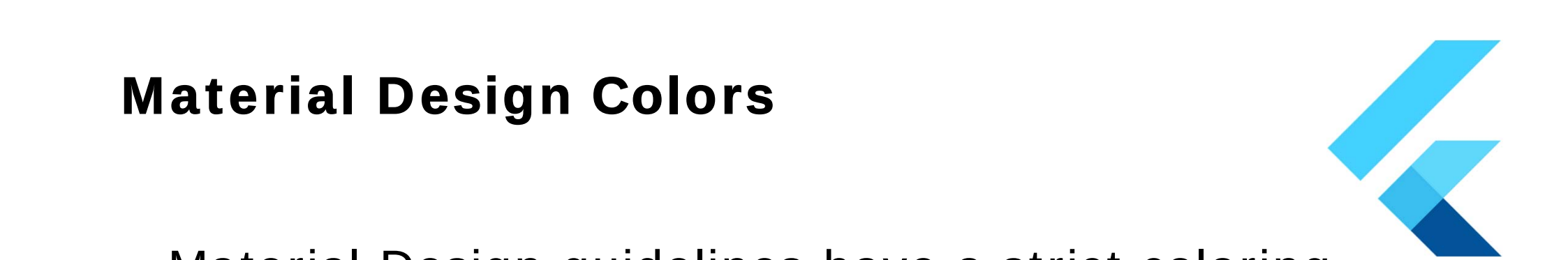
Checkpoint

- 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.

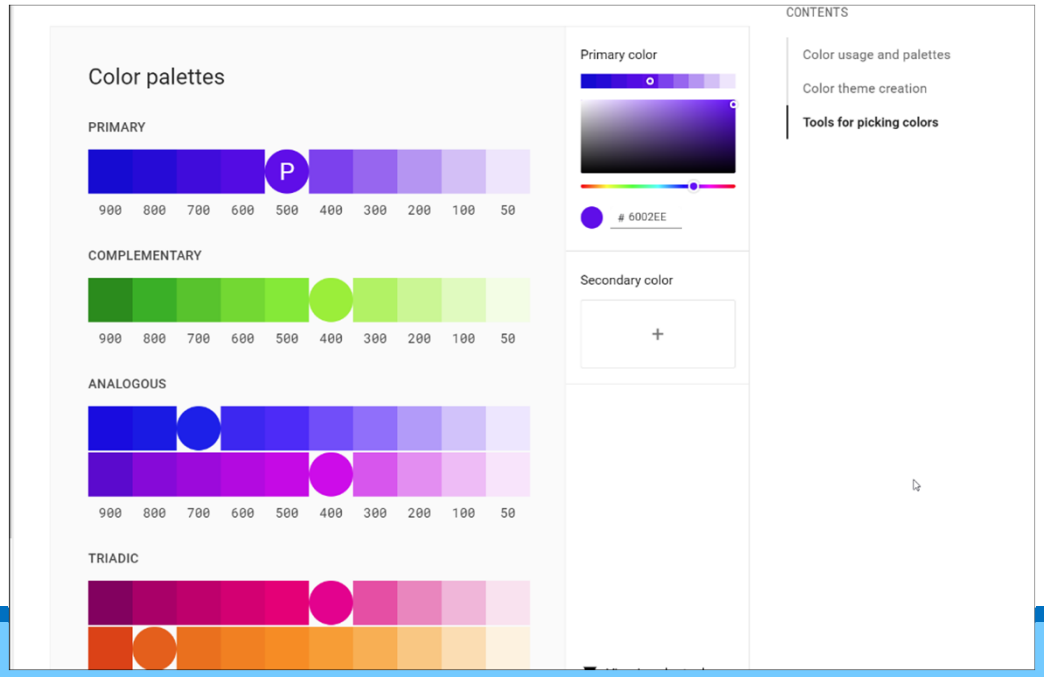
Setting colors and fonts

Updating the `Text()` Widget with additional properties

Material Design Colors



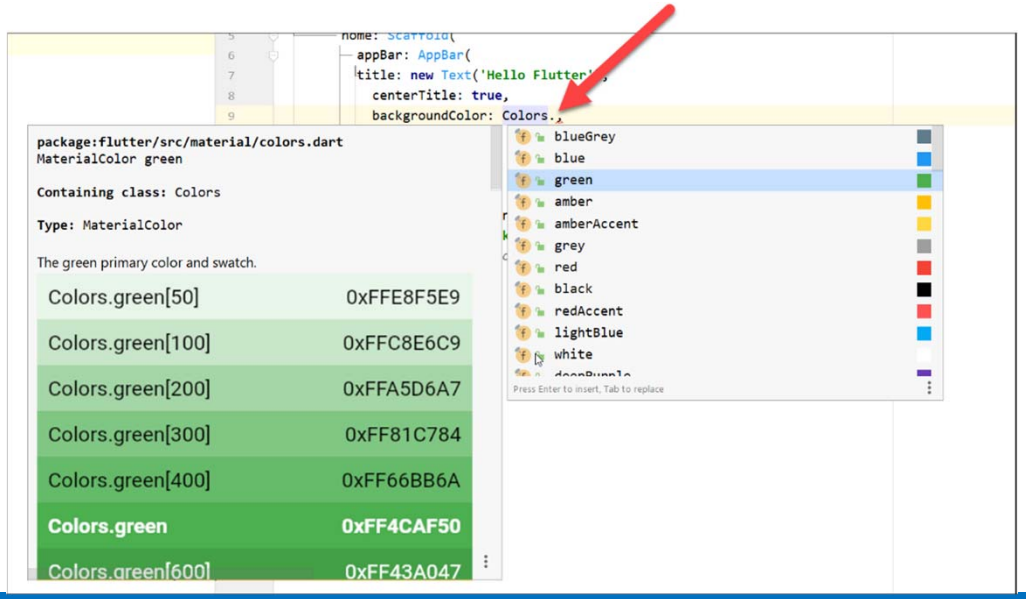
- Material Design guidelines have a strict coloring system
- material.io/design/color/the-color-system.html#tools-for-picking-colors



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

Setting the color

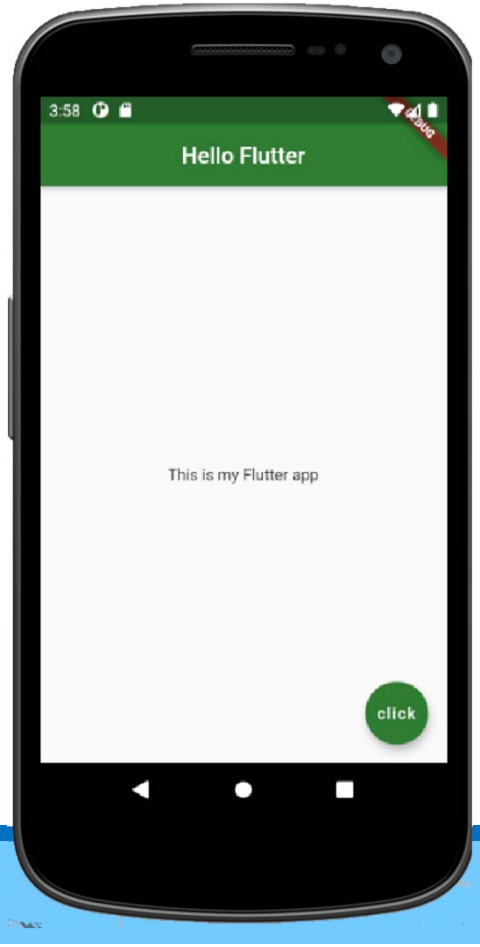
- All colors live inside the `Colors` constant.
  - Don't type the color directly!
- A lot of widgets have the `backgroundColor` property
- Type `Colors.` And press `Ctrl+Q` to see a palette



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

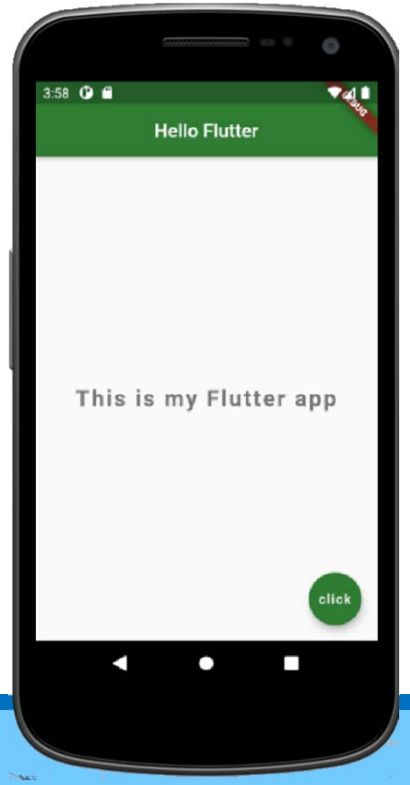
```
floatingActionButton: FloatingActionButton(
  onPressed: (){},
  backgroundColor: Colors.green[800],
  child: Text('click'),
),
```



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

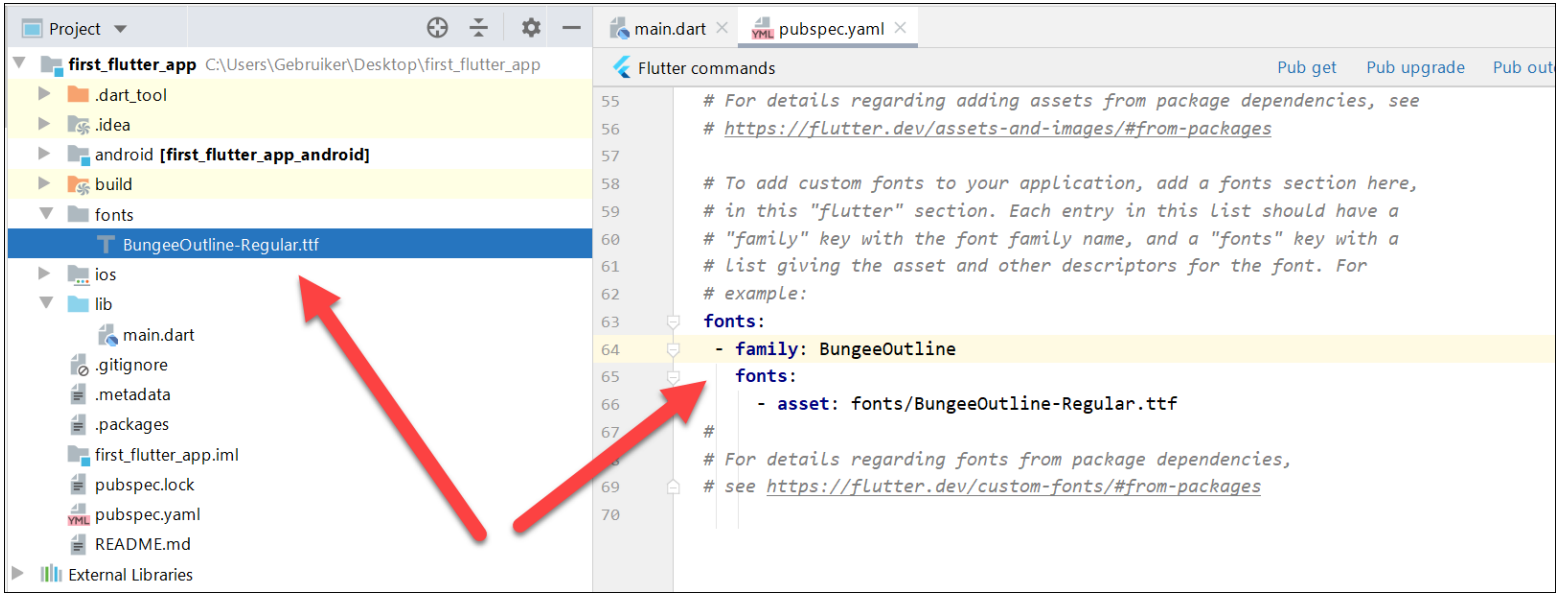
Using Custom fonts from the web

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

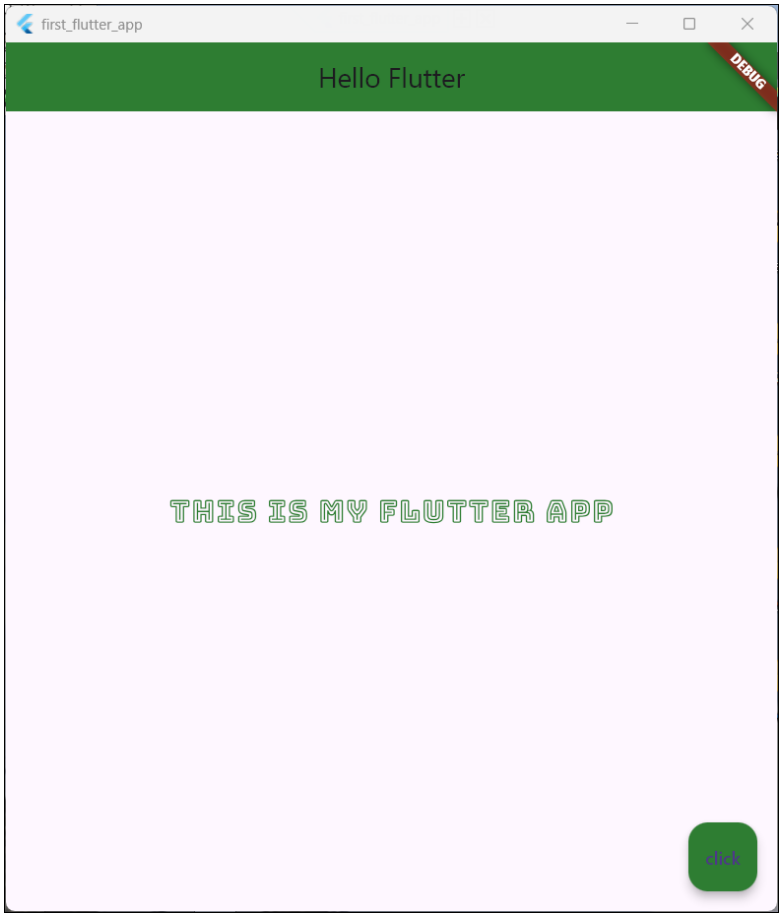
Updating the Editor

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



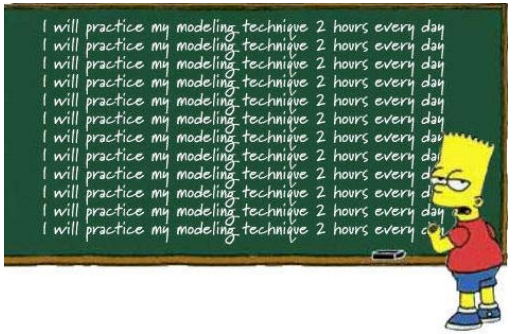
Result

```
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

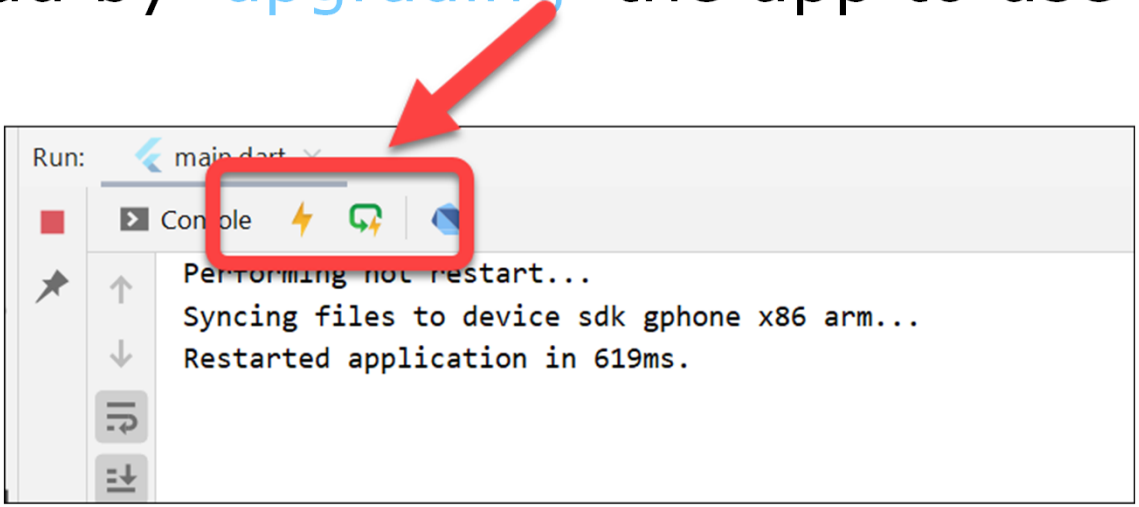
- You can use any color of your choice in your app
- You know the meaning of the `Colors` class
- You are able to download extra fonts from the internet, add them to the project and use them in the app

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 Stateless 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

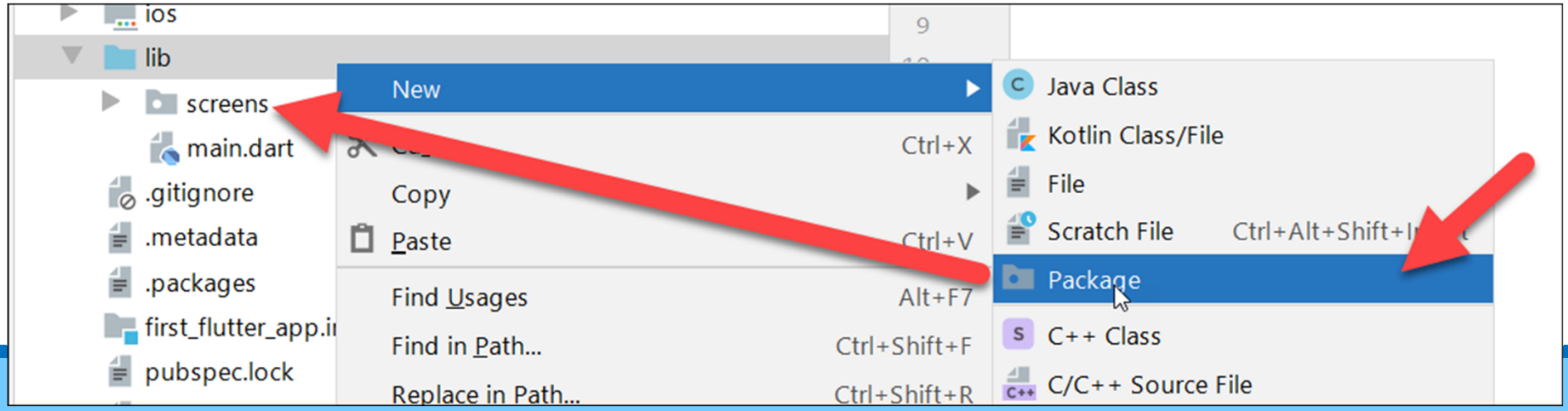
Result

```dart
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

```dart
// 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(…)
   );
  }
}
```

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

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

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

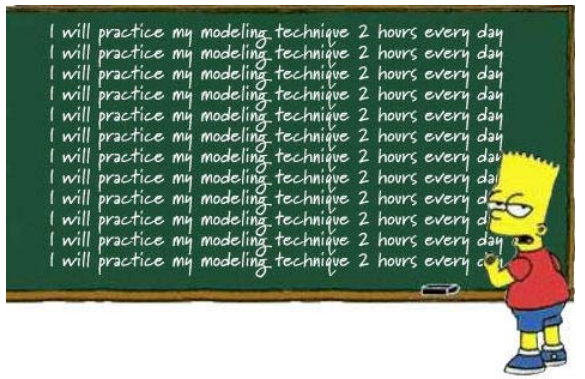
You don't have to extract widgets,

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

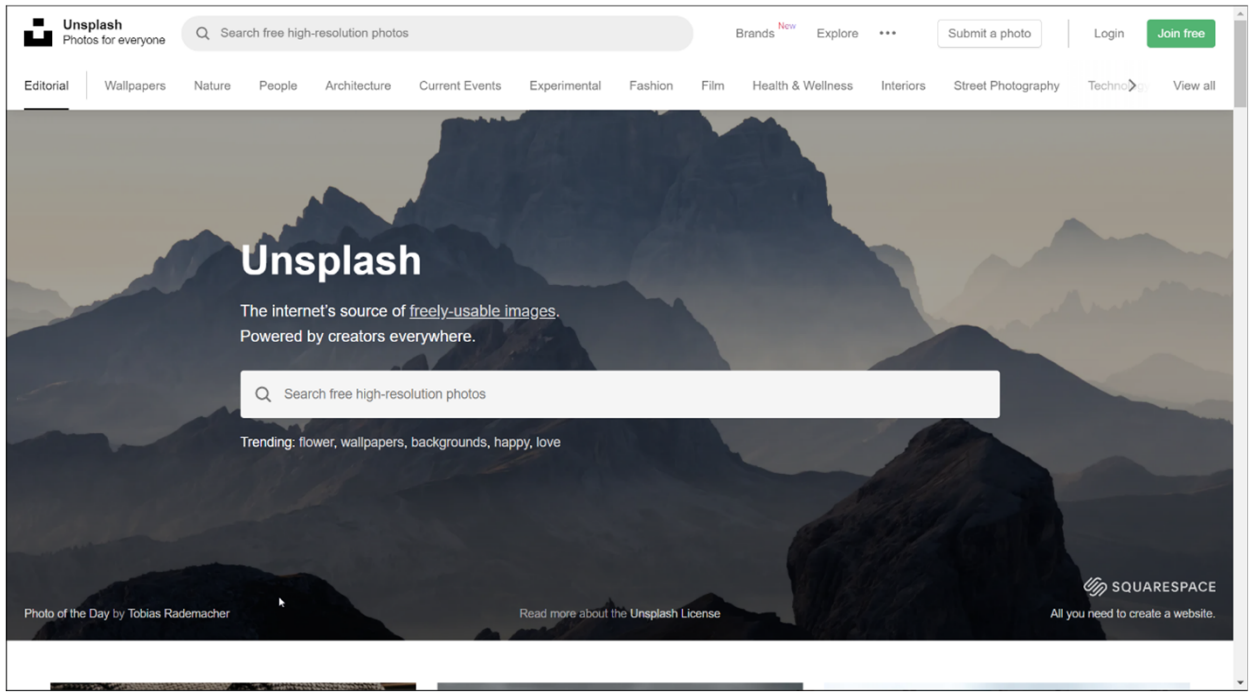
- You know the difference between `Hot Start` and `Hot Reload`.
- You know you need (at least one) `StatelessWidget` to enable Hot Reload
- You can create and/or extract code to its own `StatelessWidget()` class.
- You know about the shortcut for `StatelessWidget's` in your code
- You can extrac widgets 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



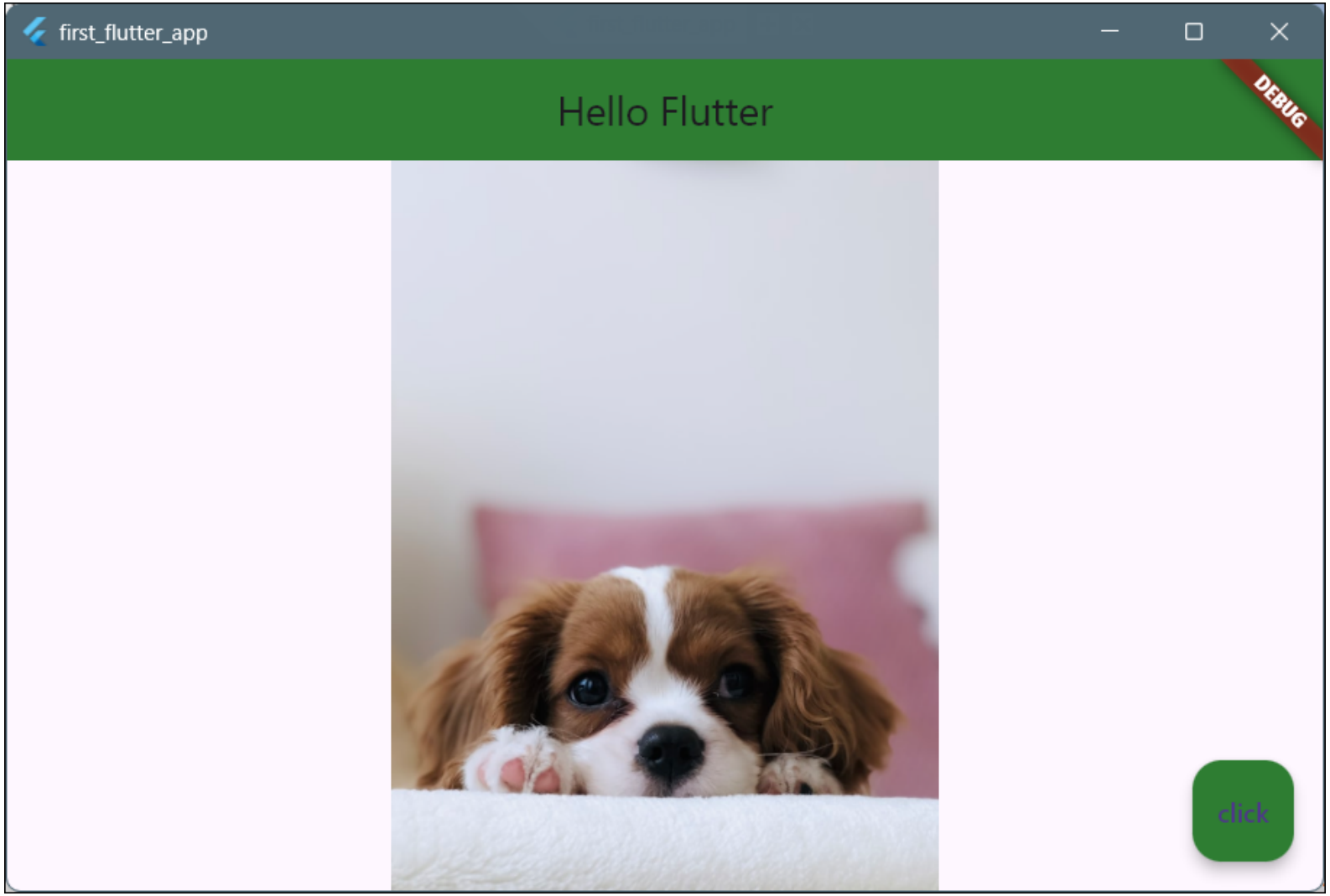
https://unsplash.com/

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

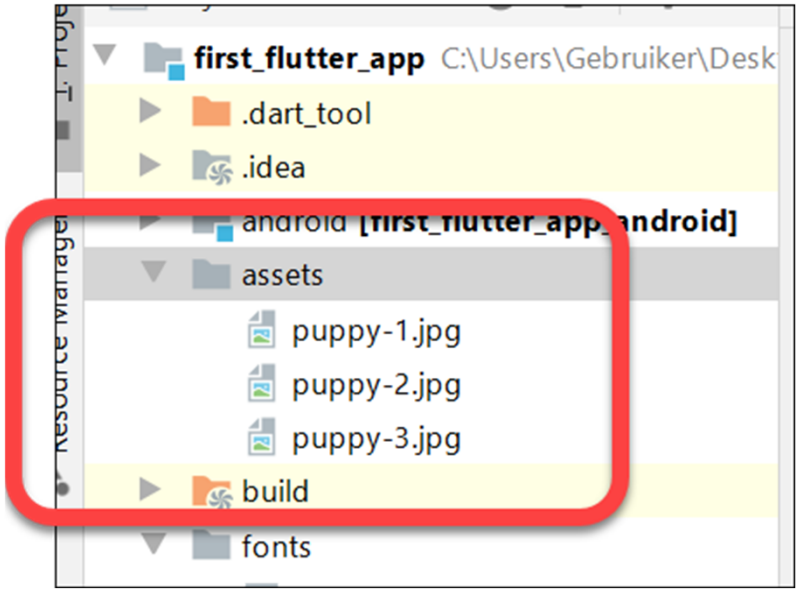


Using `AssetImage()`

- 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
```

`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

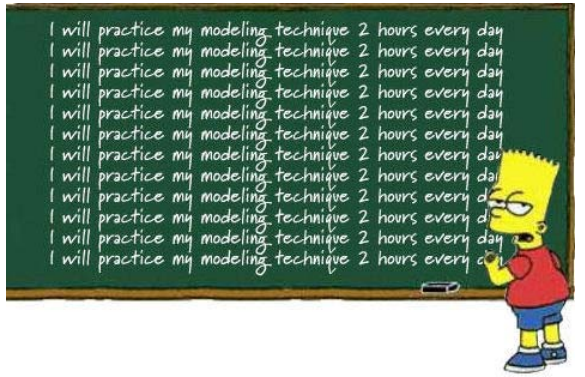
- So far, we're just loading one (1) image at a time
- Yes – I *know* you want to load more images in an array, or `List` or something :-).

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

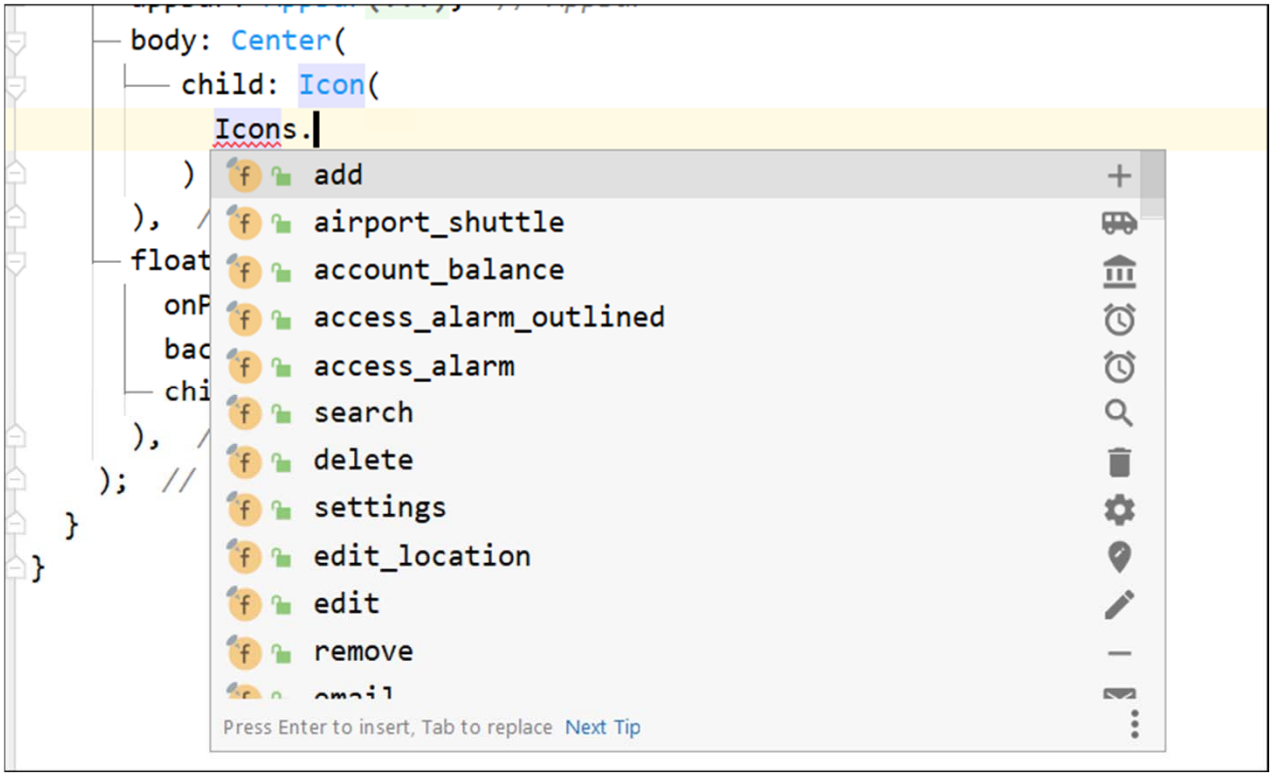


Icons & Buttons

Using Icons and Buttons in your application

Using Icons

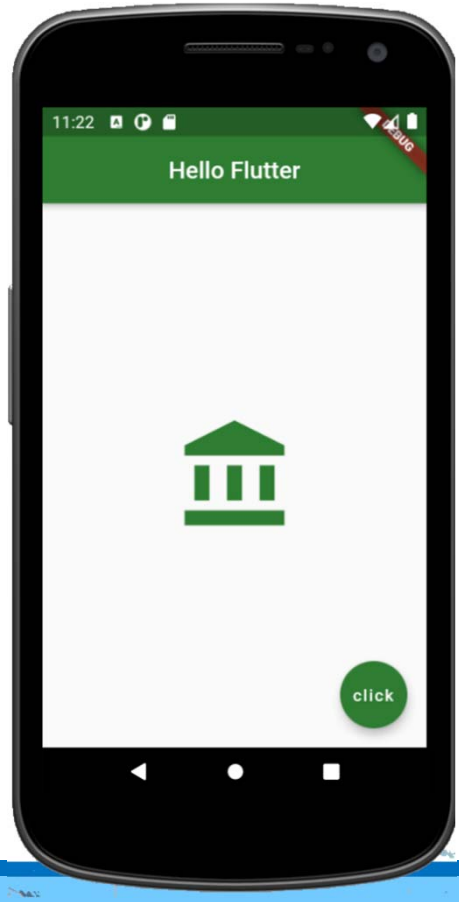
- Flutter has *a lot* of built-in Icons
- Part of the material design-spec, using the `Icon()` widget



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()`

Google Docs document on migrating buttons

When to Use Which Button?

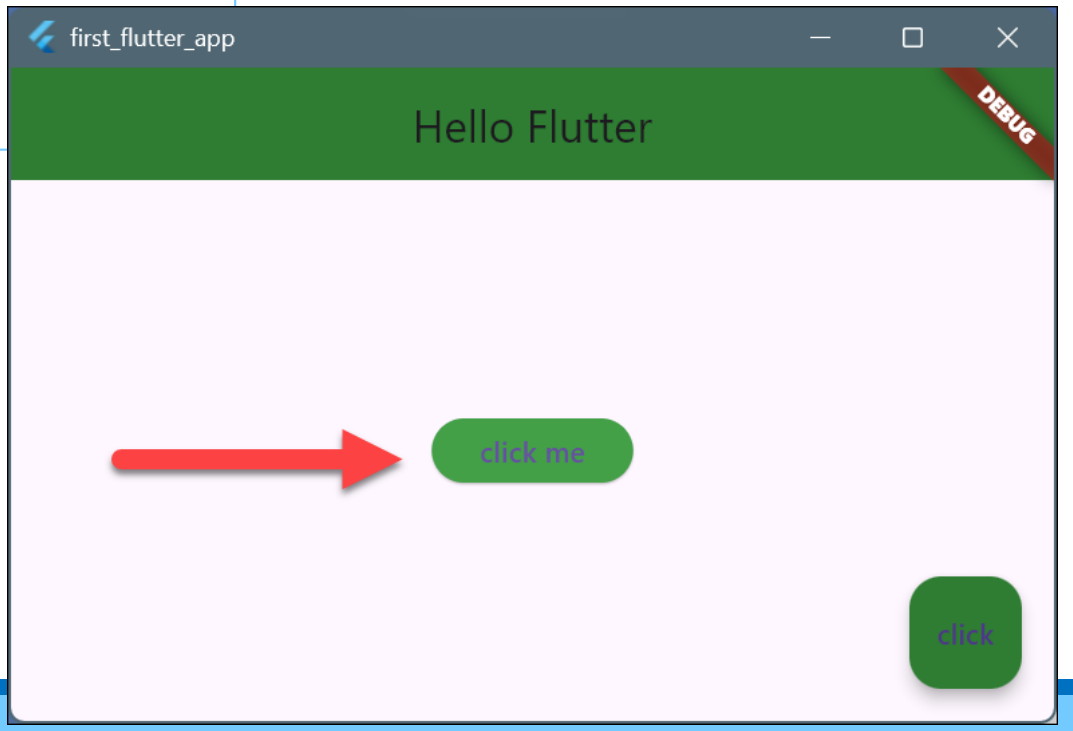
- Use `TextButton()` for actions like "Cancel" or "Skip"
- Use `ElevatedButton()` for calls to action like "Save" or "Submit"
- Use `OutlinedButton()` if you want to provide a secondary option, e.g., "Learn More"
- Use `IconButton()` to represent quick actions such as "Search," "Settings," or similar.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

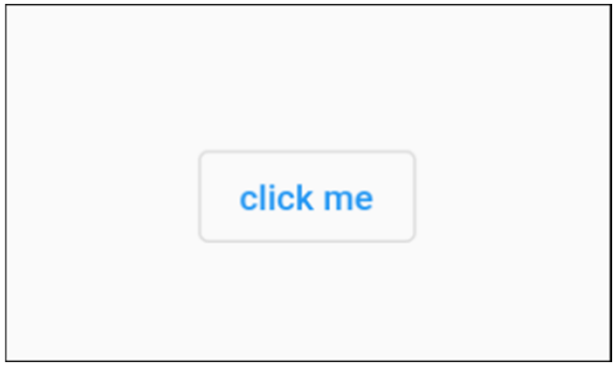
Button example

```
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'),
    )
),
```

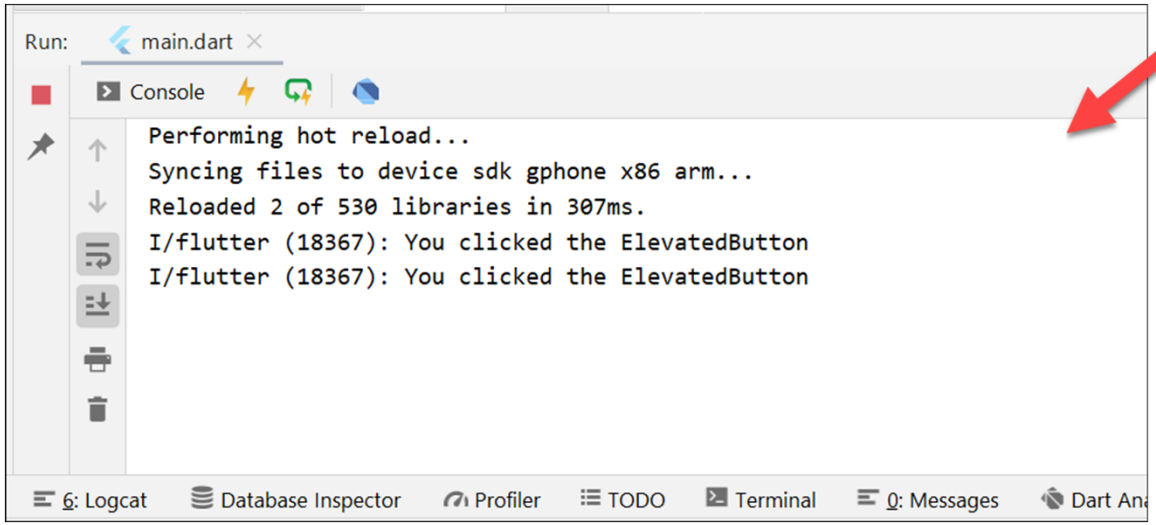
Using the onPressed() option

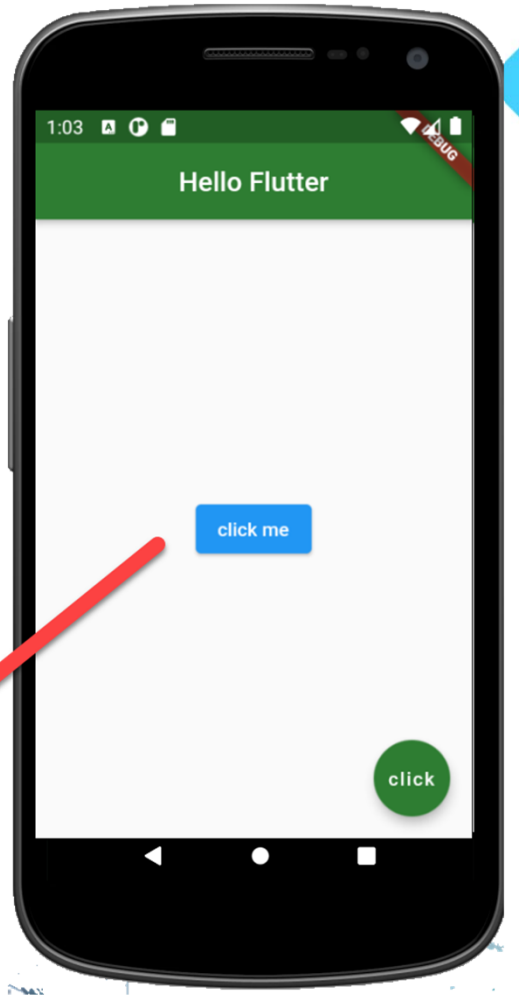
- We are now using a `StatelessWidget`
  - So `onPressed` cannot really change the widget
- But we can send text to the (Android Studio) Console

```
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

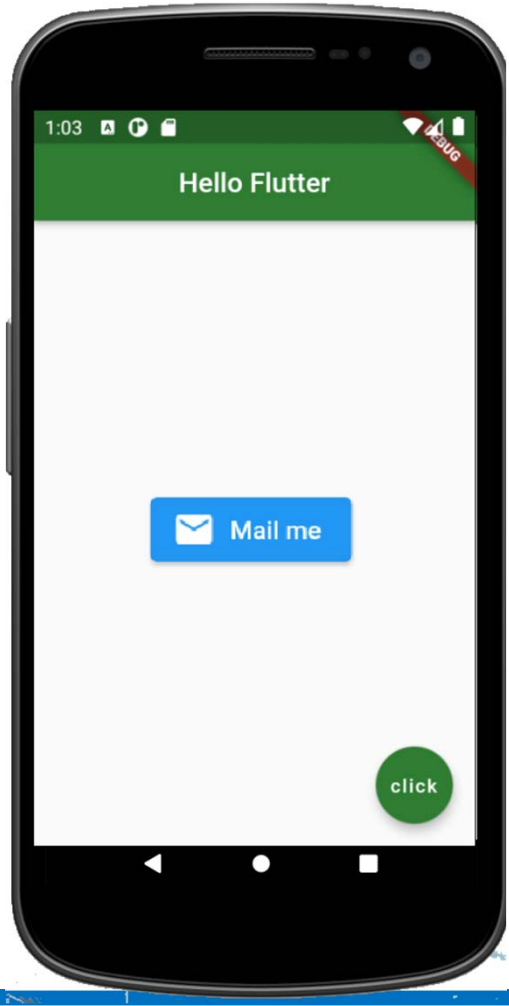




Icons on buttons

- 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

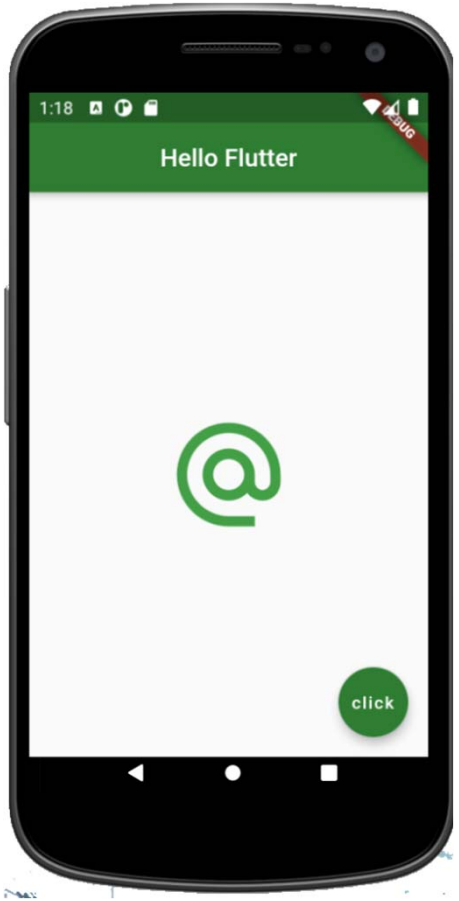
```
body: Center(
    child: ElevatedButton.icon(
      onPressed: () {
        print('You clicked the Mail Icon Button');
      },
      icon: Icon(Icons.mail),
      label: Text('Mail me'),
    )
),
```



An Icon as a button

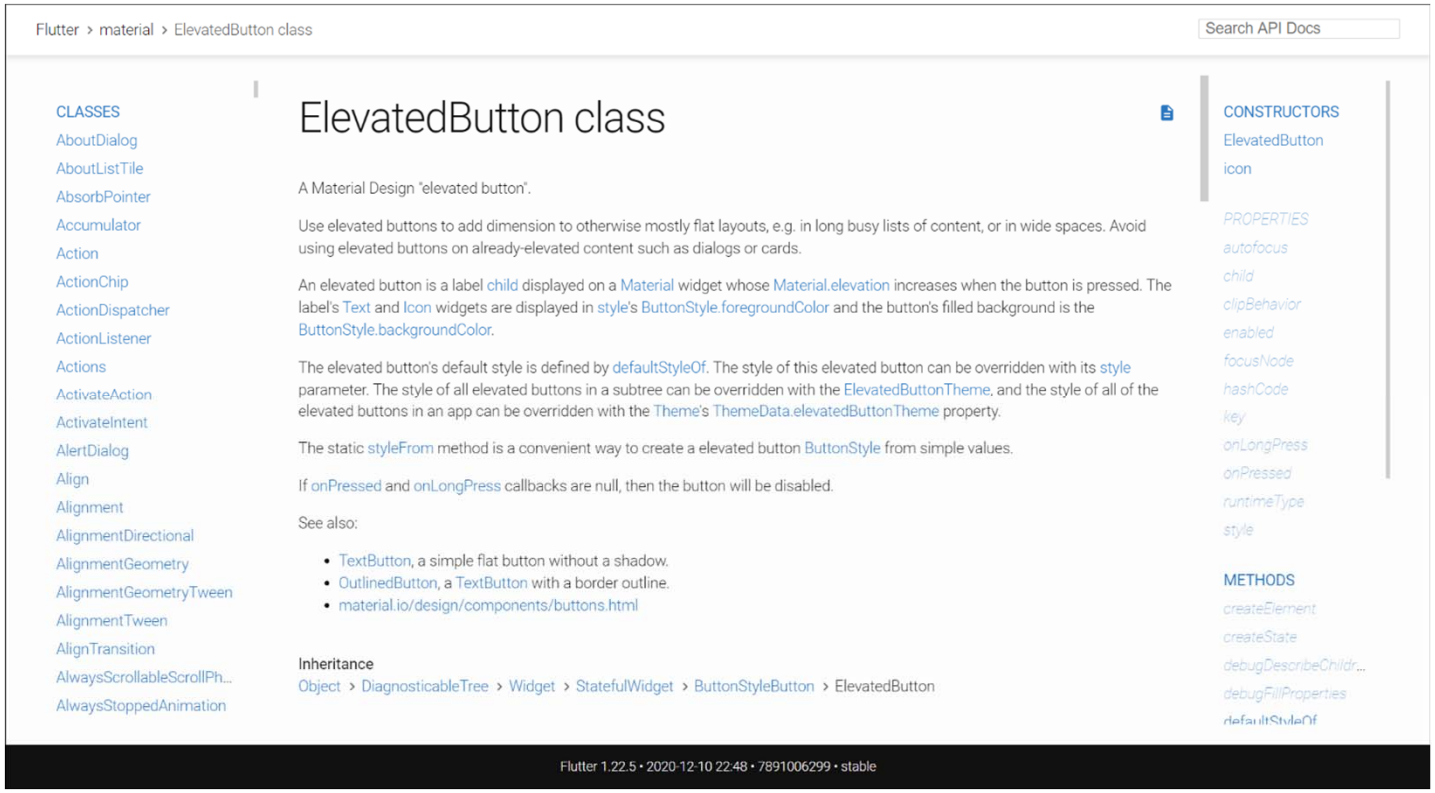
- Not an Icon *on* a button, but a pressable Icon itself
- Use the `IconButton()` widget

```
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!



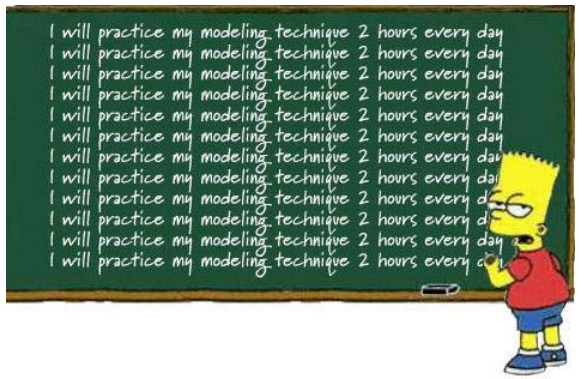
https://api.flutter.dev/flutter/material/ElevatedButton-class.html

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