

OBJECT-ORIENTED PROGRAMMING I IT 411

IT DEPT.

TIU

3RD GRADE

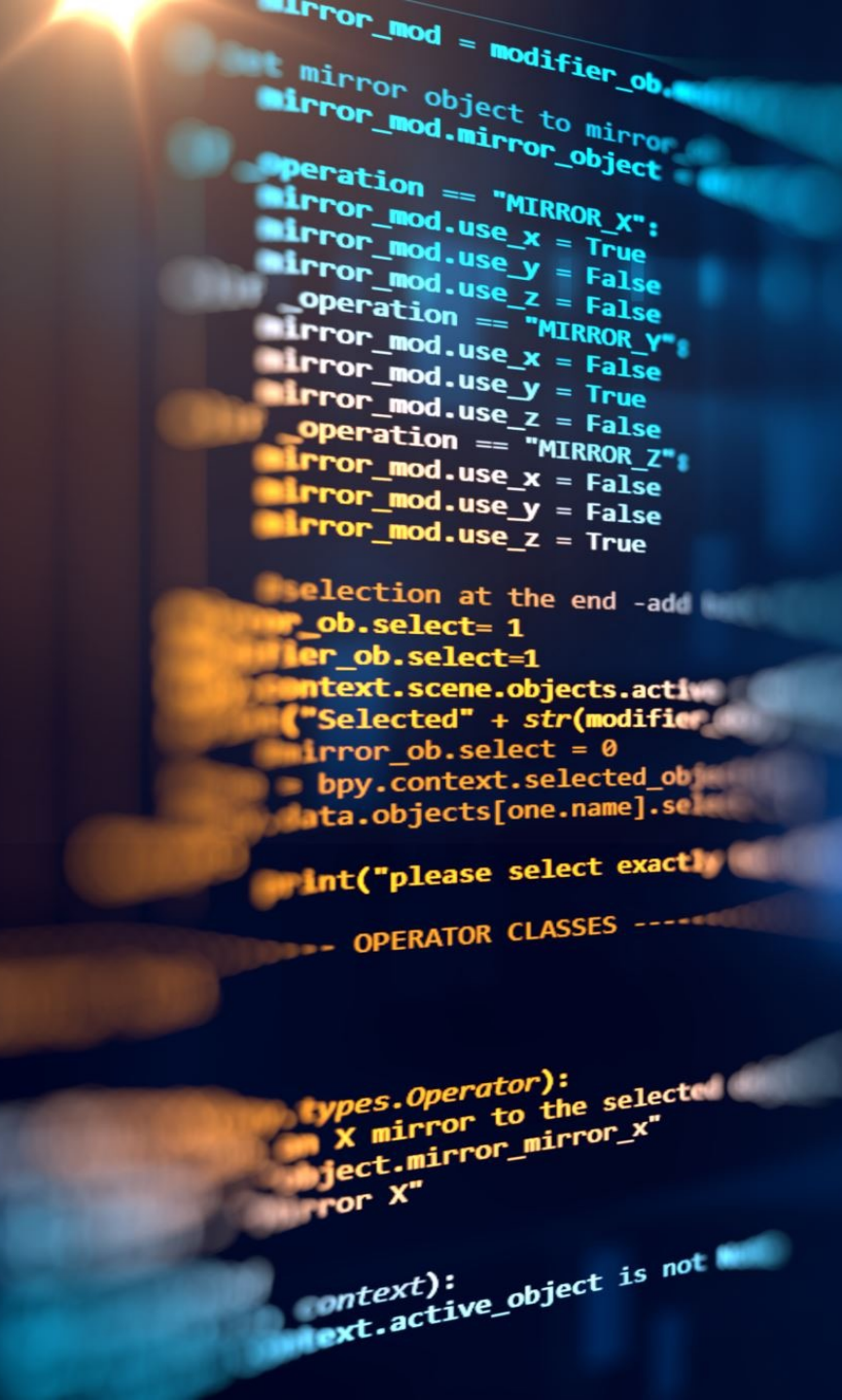
Application Bar, List View and
Build A Custom Widget

Lect. Mohammad Salim

COURSE CONTENT

- Flutter and OOP

Week	Hour	Date	Topic
1	2	4-7/10/2021	Introduction to OOP , Class diagram
2	2	10-14/10/2021	Introduction to OOP , Class diagram and Dart Packages
3	2	17-21/10/2021	Section 1: Build Your First Flutter App, structure of Flutter projects, create the UI o a Flutter app by Widgets
4	2	24-28/10/2021	Section 2: Everything's a Widget, start to build a full-featured recipe app named Fooderlich
5	2	31/10-4/11/2021	Section 2: Everything's a Widget, start to build a full-featured recipe app named Fooderlich
6	2	7-11/11/2021	Understanding widgets
7	2	14-18/11/2021	Midterm Exam
8	2	21-25/11/2021	Stateless widgets and build our personal profile application (HW2)
9	2	28/11-2/12/2021	Application bar, list view and build a custom widget
10	2	5-9/12/2021	Navigation in Flutte, Stateful Widgets and building an interactive applications
11	2	12-16/12/2021	Material Design, Build for Android and iOS platforms, Colors and Themes
12	2	19-23/12/2021	Handle user input and Handle gestures and responsive design
13	2	26-30/12/2021	Flutter Packages and Plugins, Images, Icons, Fonts
14	2	2-5/1/2022	APIs and how to get data from internet
15	2	9-13/1/2022	Final Exam
16	2	16-20/1/2022	Final Exam



CONTENTS

Material Design

Colors and Themes

Build for Android and iOS

Handle user input, gestures and responsive design

MATERIAL DESIGN



MATERIAL DESIGN

material.io/develop/flutter

- Material is an adaptable system of guidelines, components, and tools that support the best practices of user interface design.
- Backed by open-source code, Material streamlines collaboration between designers and developers, and helps teams quickly build beautiful products.

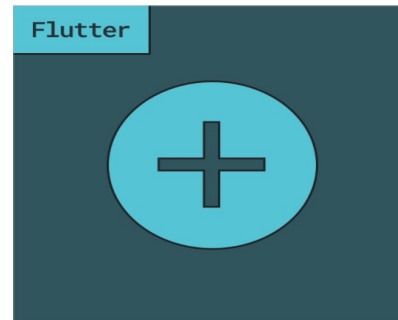
The screenshot shows the Material Design website for Flutter. The top navigation bar includes 'Design', 'Components', 'Develop', 'Resources', and 'Blog'. A left sidebar menu lists 'Flutter', 'Components', and 'Documentation'. The main content area features a large 'Flutter' heading with a sub-heading 'Build beautiful, usable products using Material Components for Flutter, a mobile UI framework'. Below this is a hero image with Flutter icons. To the right, a 'POPULAR' section lists links for 'Material Flutter GitHub repo', 'Transition patterns', and 'Checkbox'. At the bottom, there are two featured sections: 'DOCUMENTATION' titled 'Learn about the MaterialApp class for Flutter' and 'COMPONENTS' titled 'Outlined text field'.

MATERIAL FLUTTER TUTORIALS

- In the following slides we'll explore some examples from these tutorials.

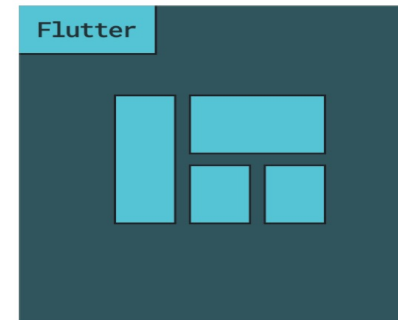
Material Flutter tutorials

Start implementing Material Components with these hands-on lessons for Flutter



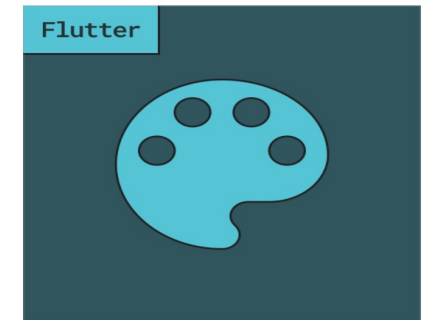
Flutter 101: Material basics

Learn the basics of using Material Components for Flutter by building a simple app with core components



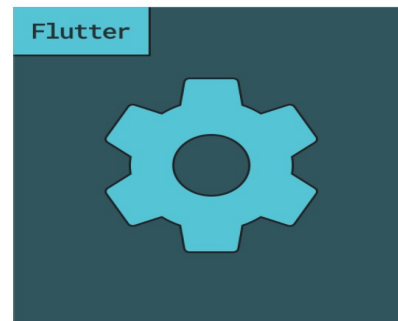
Flutter 102: Structure and layout

Learn how to use Material for structure and layout on Flutter



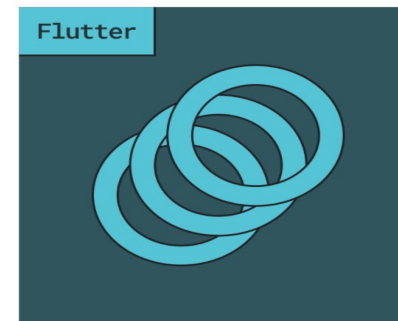
Flutter 103: Theming with color, shape, motion, and type

Discover how Material Components for Flutter make it easy to differentiate your product and express your brand through design



Flutter 104: Advanced components

Learn how to use an advanced component – backdrop menu – for Flutter

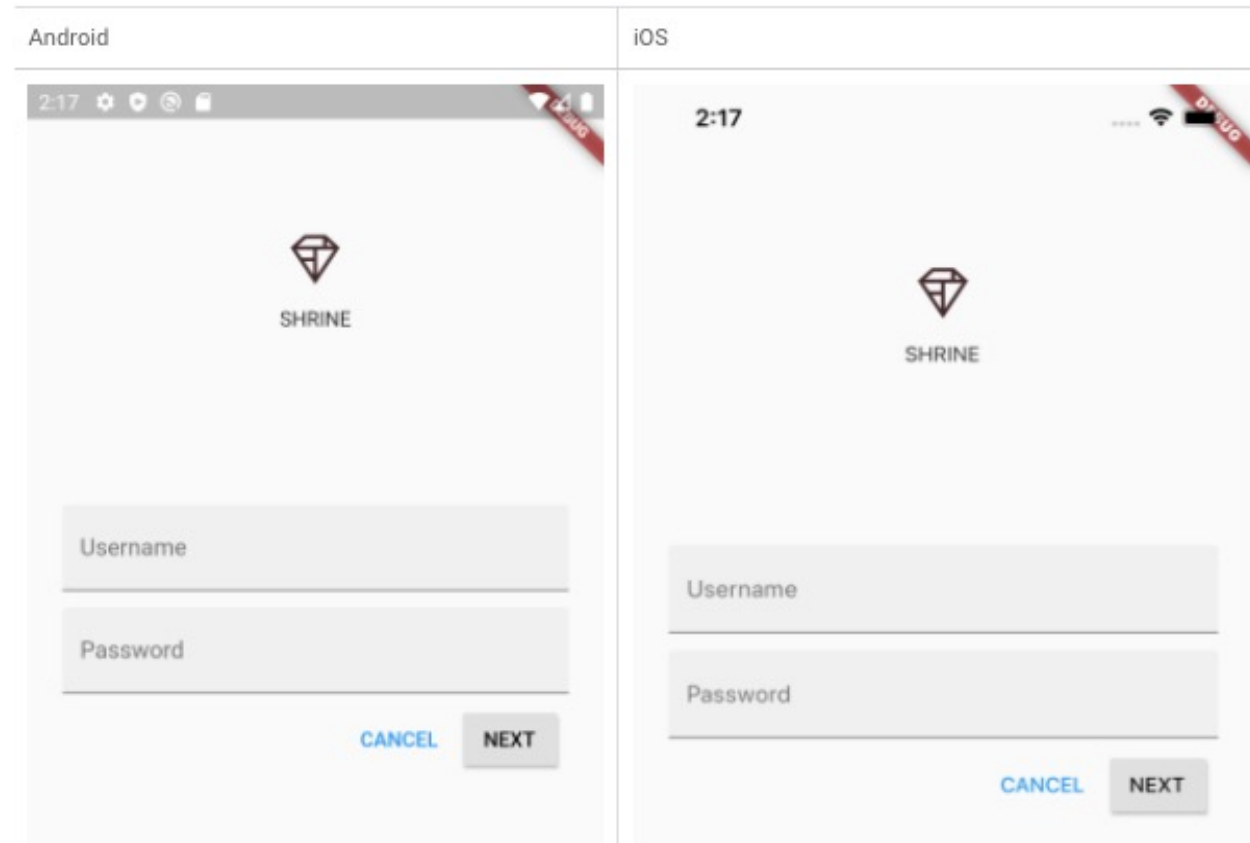


Building Beautiful Transitions with Material Motion for Flutter

Build Material's motion system into an example

MATERIAL COMPONENTS (MDC) BASICS (FLUTTER)

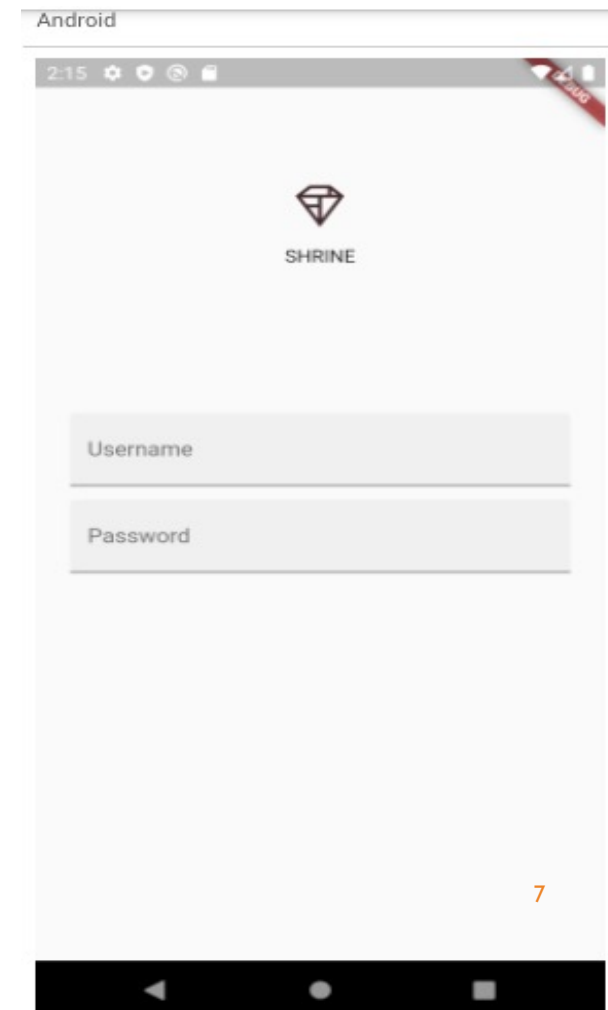
- In this **example**, you'll build a **login page** for Shrine that contains:
- An image of Shrine's logo
- The name of the app (Shrine)
- Two text fields, one for entering a username and the other for a password
- Two buttons



MATERIAL COMPONENTS (MDC) BASICS (FLUTTER)

- The text fields each have a **decoration**: field that takes an **InputDecoration** widget. The **filled**: field means the background of the text field is lightly filled in to help people recognize the tap or touch target area of the text field.
- The second text field's **obscureText: true** value automatically replaces the input that the user types with bullets, which is appropriate for passwords.

```
// TODO: Add TextField widgets (101)
// [Name]
TextField(
  decoration: const InputDecoration(
    filled: true,
    labelText: 'Username',
  ),
),
// spacer
const SizedBox(height: 12.0),
// [Password]
TextField(
  decoration: const InputDecoration(
    filled: true,
    labelText: 'Password',
  ),
  obscureText: true,
),
```

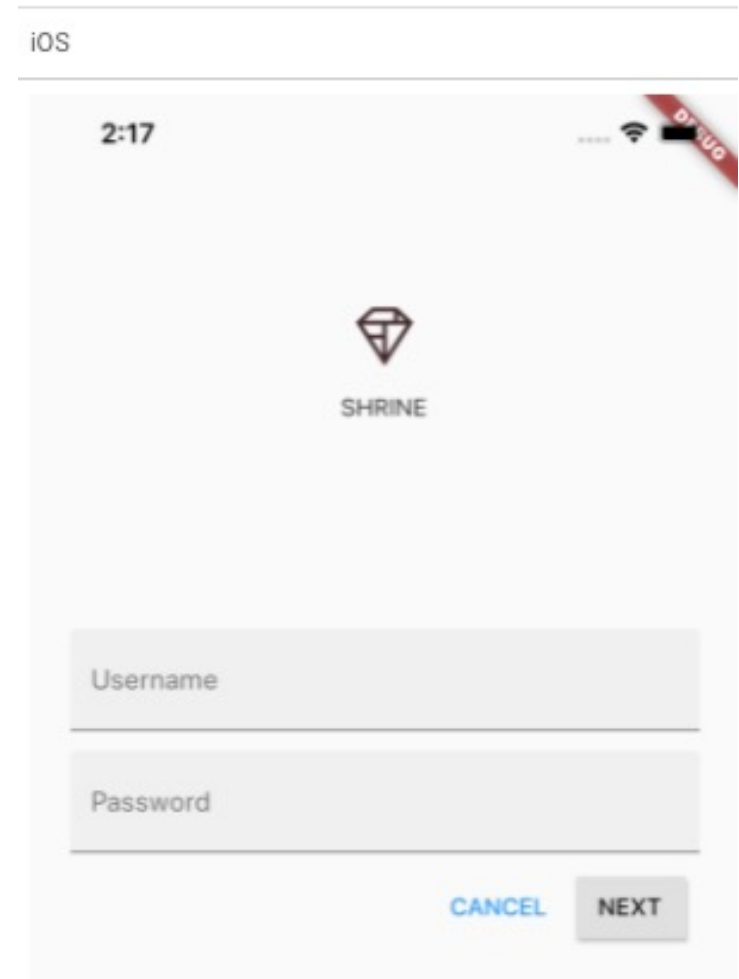


MATERIAL COMPONENTS (MDC) BASICS (FLUTTER)

Add Buttons

Choosing between text and elevated buttons

- We'll use two kinds of MDC button widgets: the **TextButton** and the **ElevatedButton**.
- Why not simply display two elevated buttons? Each button type indicates which actions are more important than others.
- The action we'd least like them to take is cancelling the login. Because an elevated button draws the eye with its raised appearance, it should be used for the more important action. By comparison, the plain text button to the left of it looks less emphasized.



MATERIAL COMPONENTS (MDC) BASICS (FLUTTER)

Add Buttons

Add the ButtonBar

After the text fields, add the `ButtonBar` to the `ListView`'s children:

```
// TODO: Add button bar (101)
ButtonBar(
  // TODO: Add a beveled rectangular border to CANCEL
  children: <Widget>[
    // TODO: Add buttons (101)
  ],
),
```

The `ButtonBar` arranges its children in a row.

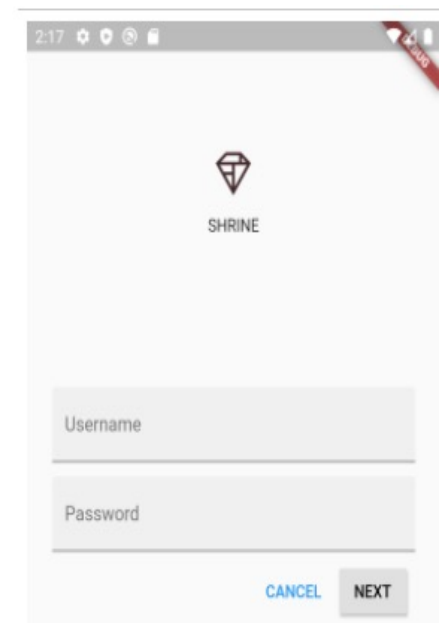
Why do we have empty blocks for the `onPressed:` fields?

If we passed `null`, or didn't include the field (which then defaults to `null`), the buttons would become **disabled**. There would be no feedback on touch and we couldn't get a good idea of their enabled behavior. Using **empty blocks** prevents them from being disabled.

Add the buttons

Then add two buttons to the `ButtonBar`'s list of `children`:

```
// TODO: Add buttons (101)
TextButton(
  child: const Text('CANCEL'),
  onPressed: () {
    // TODO: Clear the text fields (101)
  },
),
// TODO: Add an elevation to NEXT (103)
// TODO: Add a beveled rectangular border to NEXT
ElevatedButton(
  child: const Text('NEXT'),
  onPressed: () {
    // TODO: Show the next page (101)
  },
),
```



MATERIAL COMPONENTS (MDC) BASICS (FLUTTER)

Add Buttons

- The **AppBar** handles the layout work for you. It positions the buttons horizontally, so they appear next to one another.
- Touching a button initiates an ink ripple animation, without causing anything else to happen.
- Let's add functionality into the anonymous **onPressed**: functions, so that the cancel button clears the text fields, and the next button dismisses the screen:

Add TextEditingController

Now when you type something into the text fields, hitting cancel clears each field again!



To make it possible to clear the text fields' values, we'll add **TextEditingController** to control their text.

Right under the `_LoginPageState` class's declaration, add the controllers as `final` variables.

```
// TODO: Add text editing controllers (101)
final _usernameController = TextEditingController();
final _passwordController = TextEditingController();
```

On the first text field's `controller`: field, set the `_usernameController`:

```
// TODO: Add TextField widgets (101)
// [Name]
TextField(
  controller: _usernameController,
```

On the second text field's `controller`: field, now set the `_passwordController`:

```
// TODO: Add TextField widgets (101)
// [Password]
TextField(
  controller: _passwordController,
```

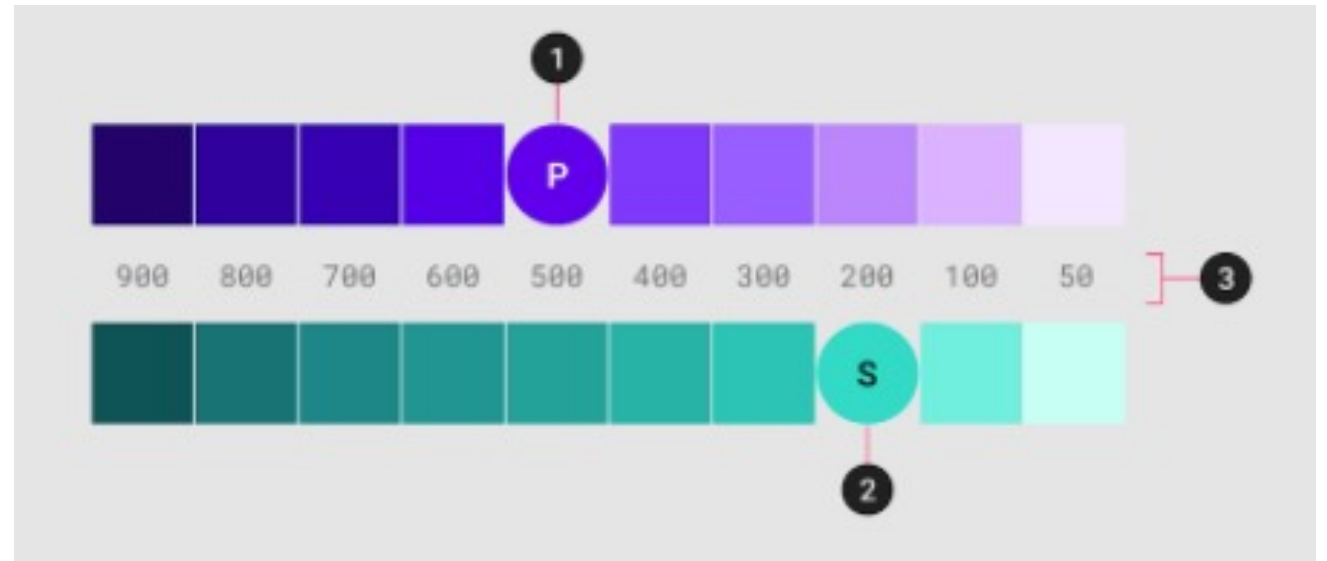
Edit onPressed

Add a command to clear each controller in the `TextButton`'s `onPressed`: function:

```
// TODO: Clear the text fields (101)
_usernameController.clear();
_passwordController.clear();
```

COLORS AND THEMES

- The Material Design color system can help you create a color theme that reflects your brand or style.
- The Material Design color system helps you apply color to your UI in a meaningful way. In this system, you select a primary and a secondary color to represent your brand. Dark and light variants of each color can then be applied to your UI in different ways.
- Color themes are designed to be harmonious, ensure accessible text, and distinguish UI elements and surfaces from one another.
- The [Material Design palette tool](#) or **2014 Material Design palettes** are available to help you select colors.



A sample primary and secondary palette

1. Primary color
2. Secondary color
3. Light and dark variants

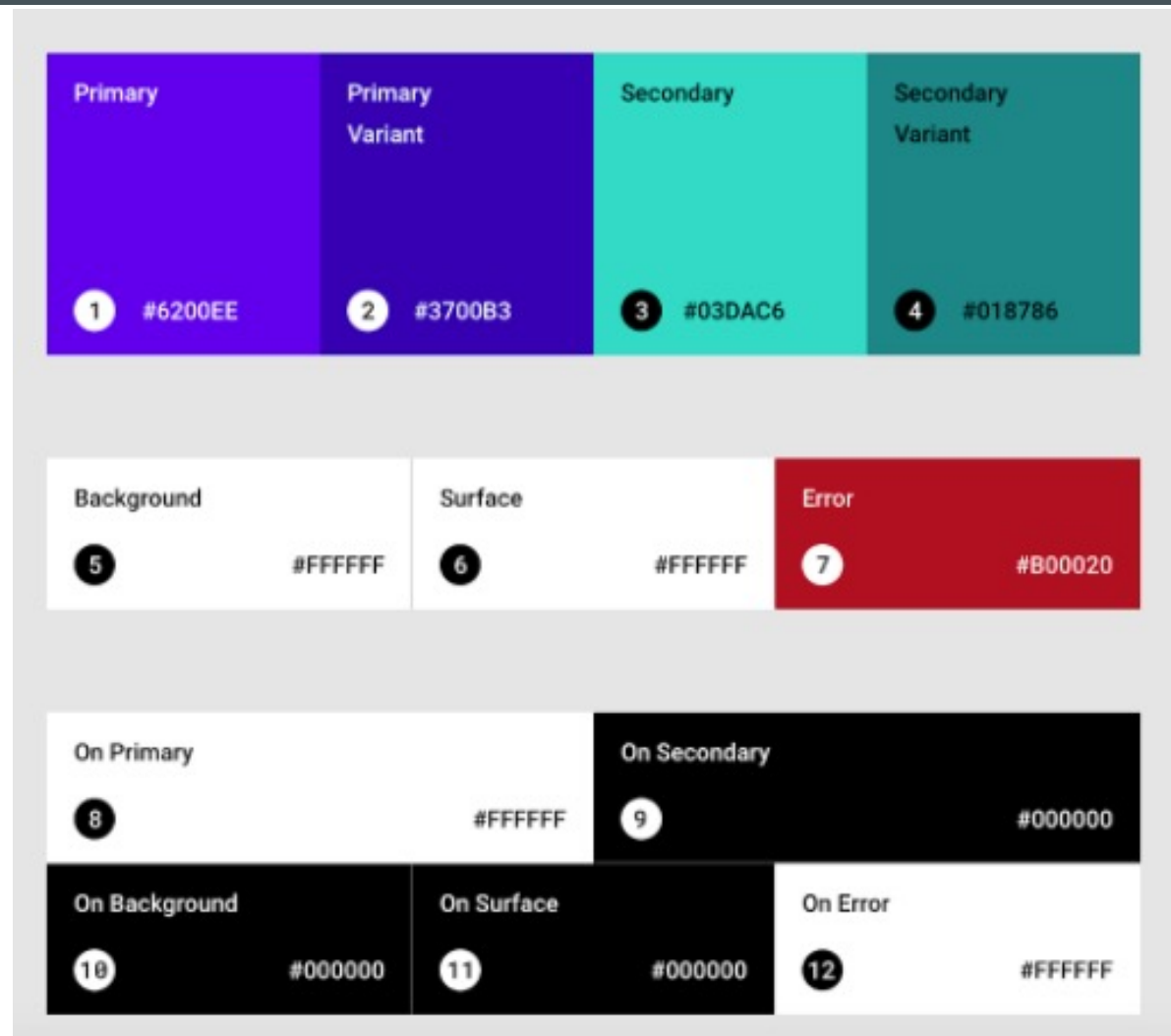
COLOR THEME CREATION

The baseline Material color theme

- Material Design comes designed with a built-in, baseline theme that can be used as-is, straight out of the proverbial box.

This includes default colors for:

- Primary and secondary colors
 - Variants of primary and secondary colors
 - Additional UI colors, such as colors for backgrounds, surfaces, errors, typography, and iconography.
- All of these colors can be customized for your app.

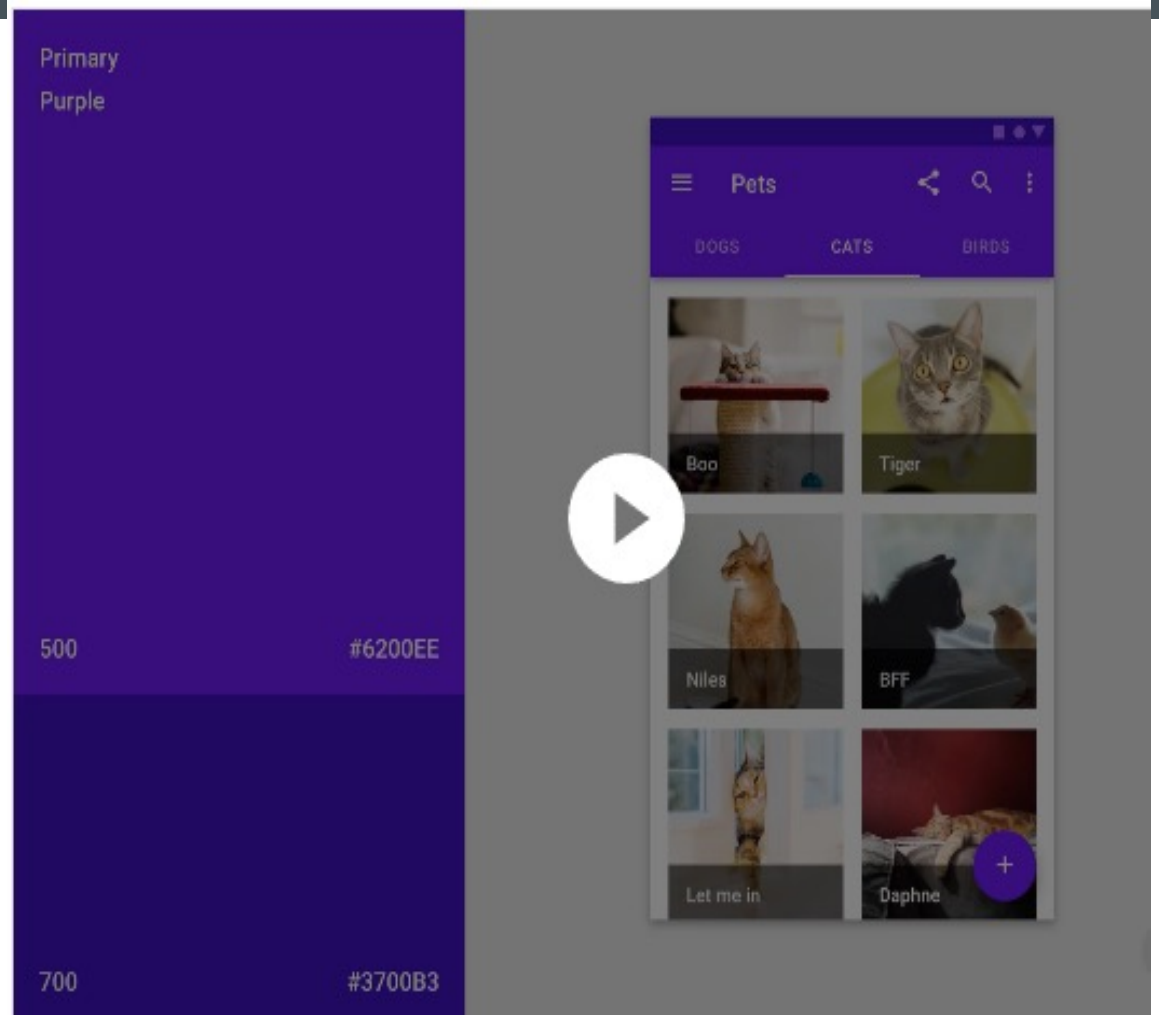


PRIMARY COLOR

- A **primary color** is the color displayed most frequently across your app's screens and components.

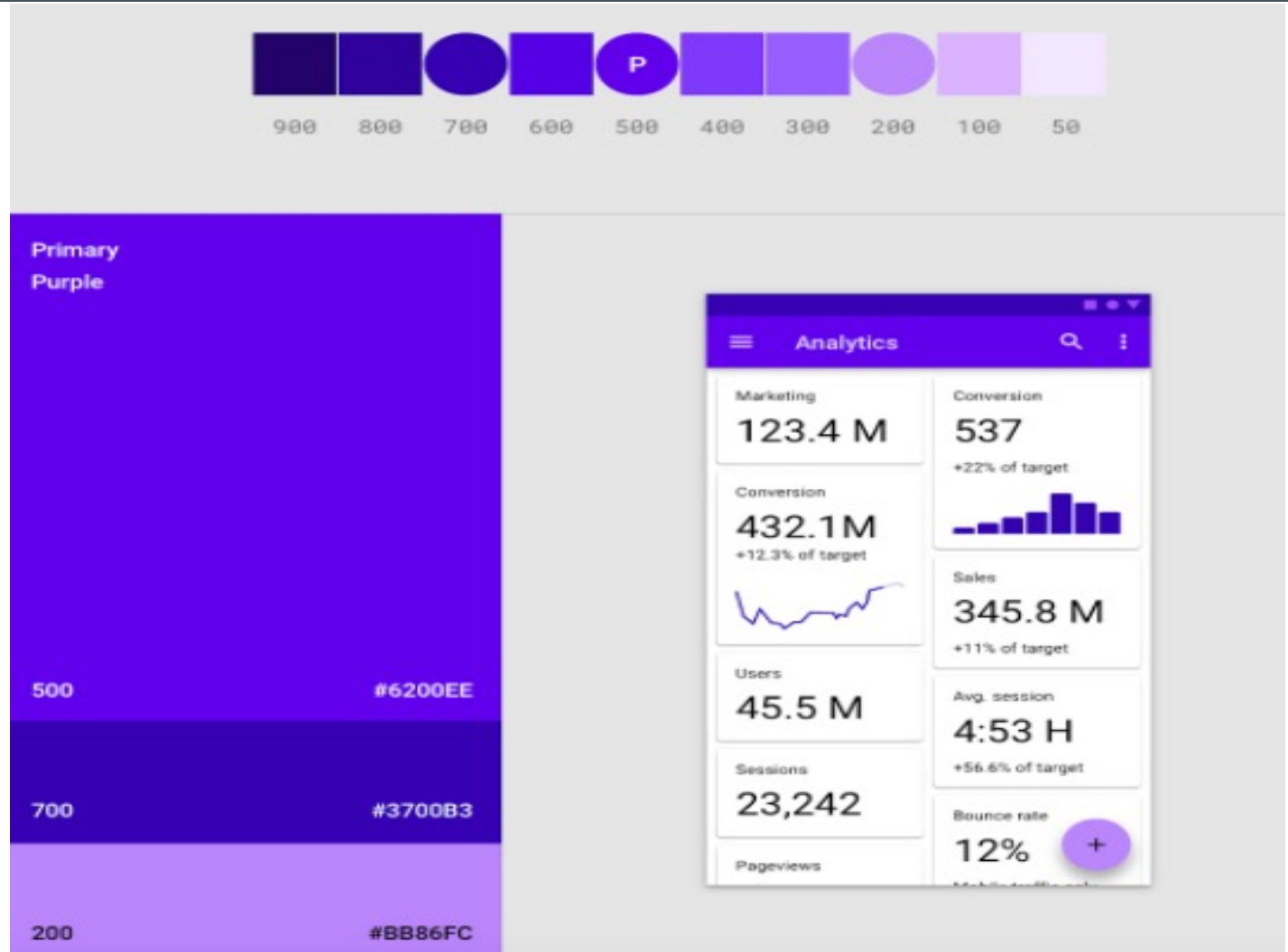
Dark and light primary variants

- Your primary color can be used to make a color theme for your app, including dark and light primary color variants.
- **Distinguish UI elements**
- To create contrast between UI elements, such as a top app bar from a system bar, you can use light or dark variants of your primary colors.
- You can also use these to distinguish elements within a component, such as the icon of a floating action button from its circular container.



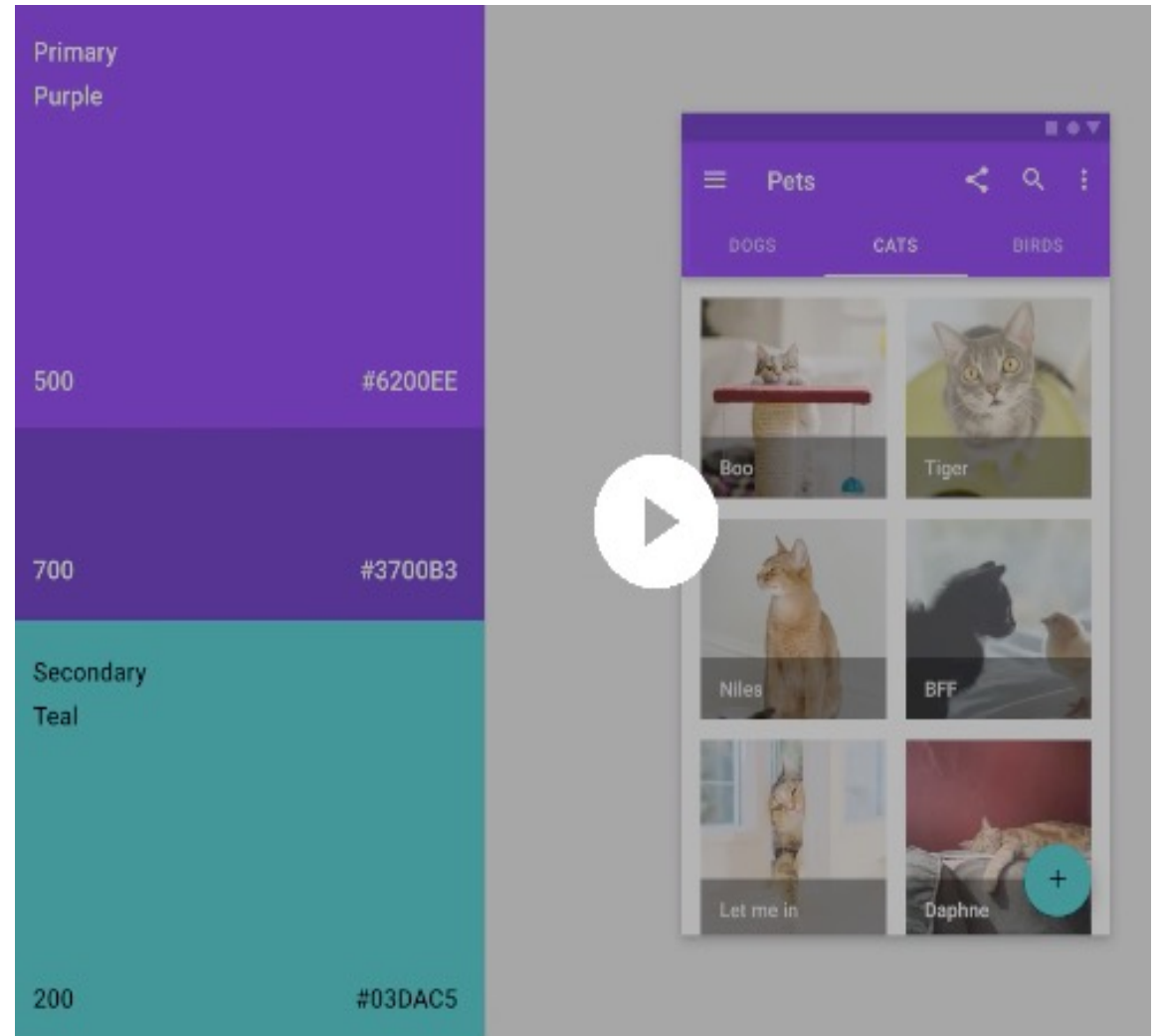
PRIMARY COLOR

- A top app bar uses light and dark primary color variants to distinguish it from a system bar.
- This UI uses a primary color and two primary variants.



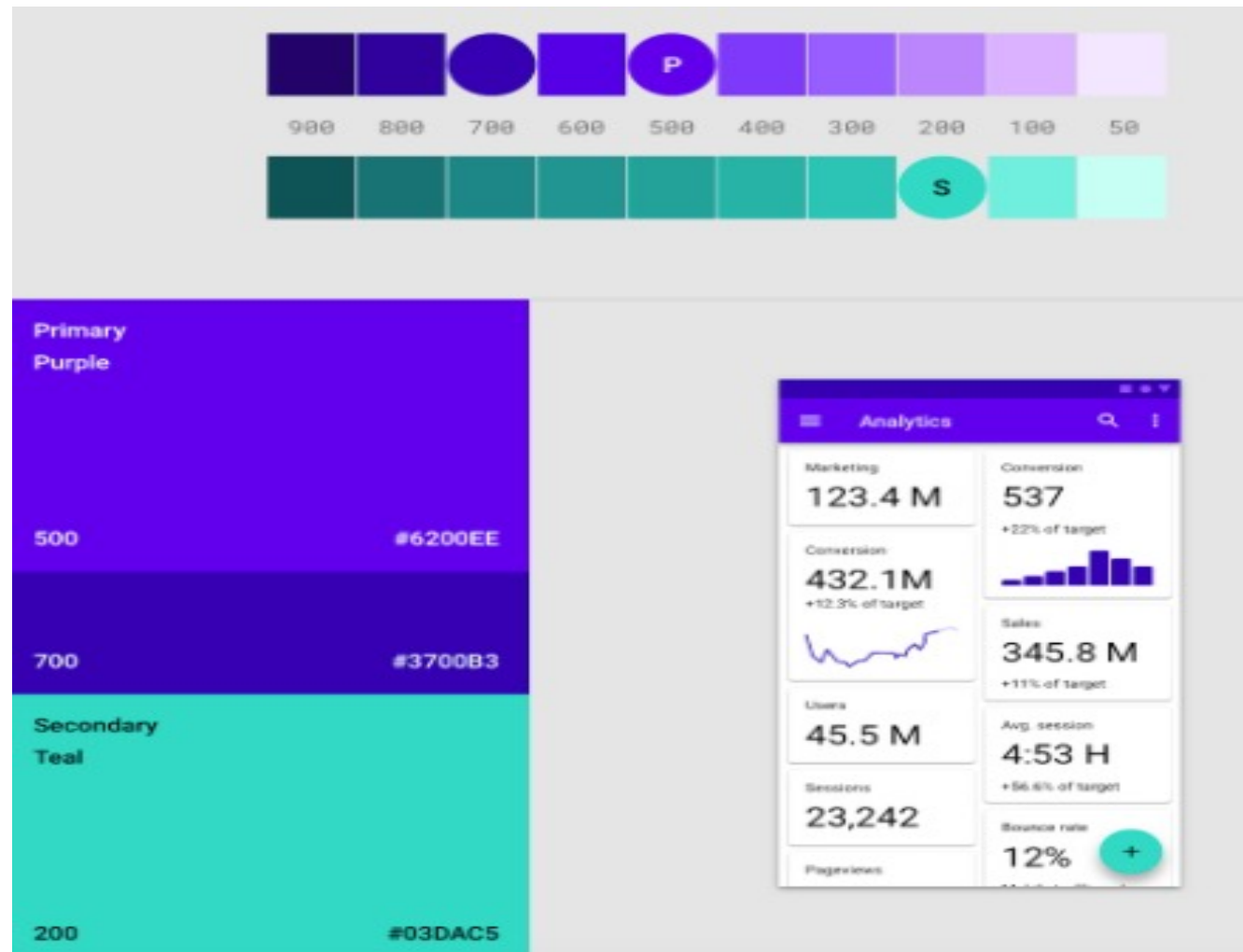
SECONDARY COLOR

- A **secondary color** provides more ways to accent and distinguish your product. Having a secondary color is optional, and should be applied sparingly to accent select parts of your UI.
- If you don't have a secondary color, your primary color can also be used to accent elements.
- Secondary colors are best for:
 1. Floating action buttons
 2. Selection controls, like sliders and switches
 3. Highlighting selected text
 4. Progress bars
 5. Links and headlines
- Just like the primary color, your secondary color can have dark and light variants. A color theme can use your primary color, secondary color, and dark and light variants of each color.



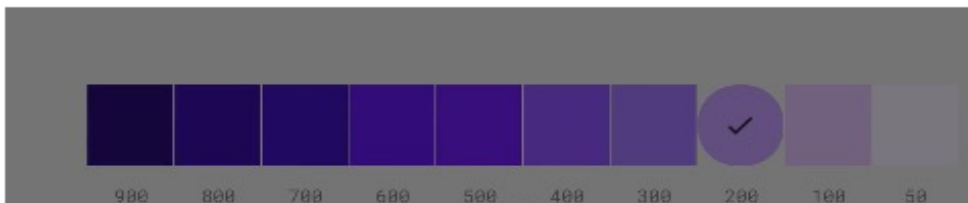
SECONDARY COLOR

- Dark and light variants of primary and secondary colors
- This UI uses a color theme with a primary color, a primary variant, and a secondary color.



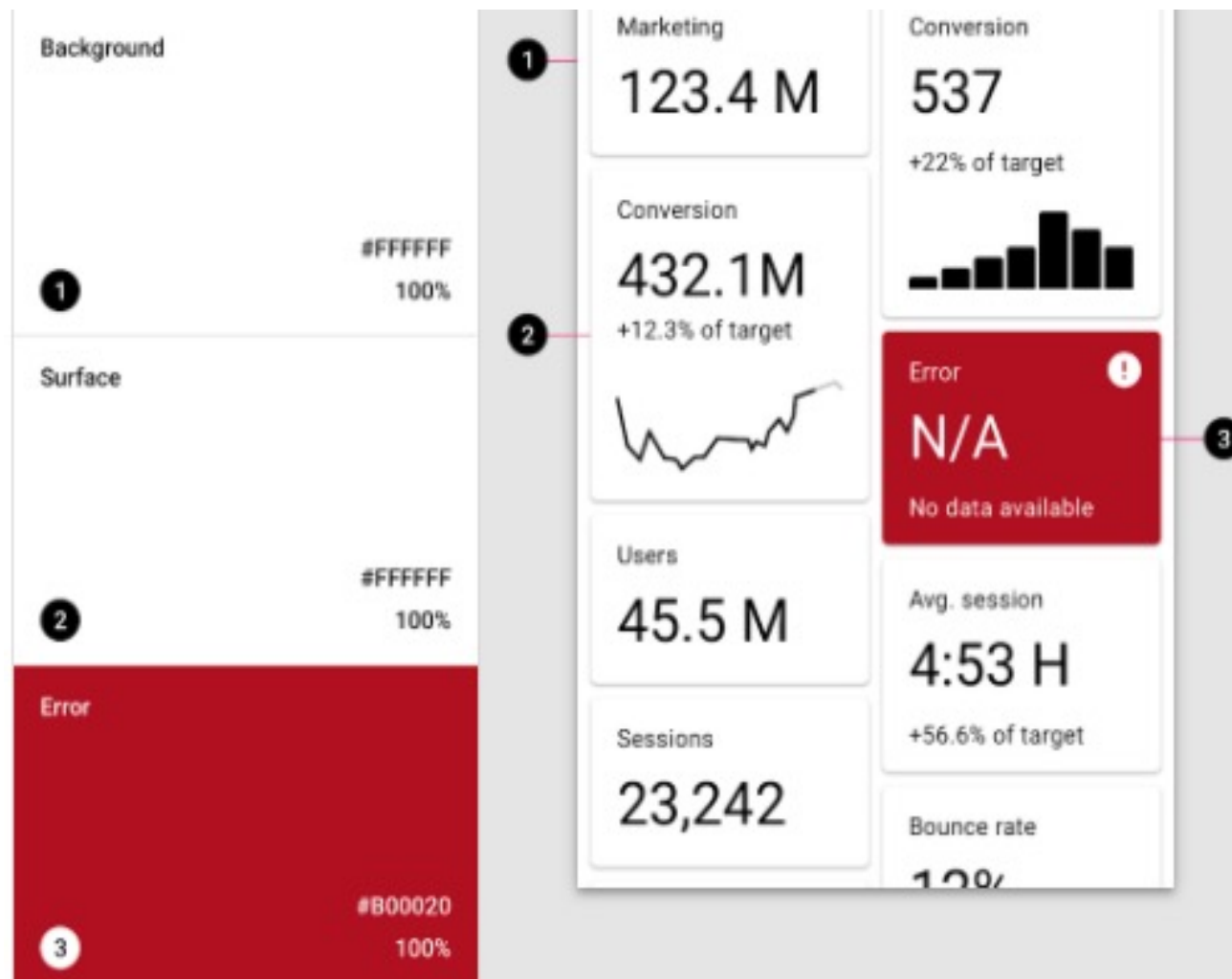
Color swatches

A **swatch** is a sample of a color chosen from a range of similar colors.



SURFACE, BACKGROUND, AND ERROR COLORS

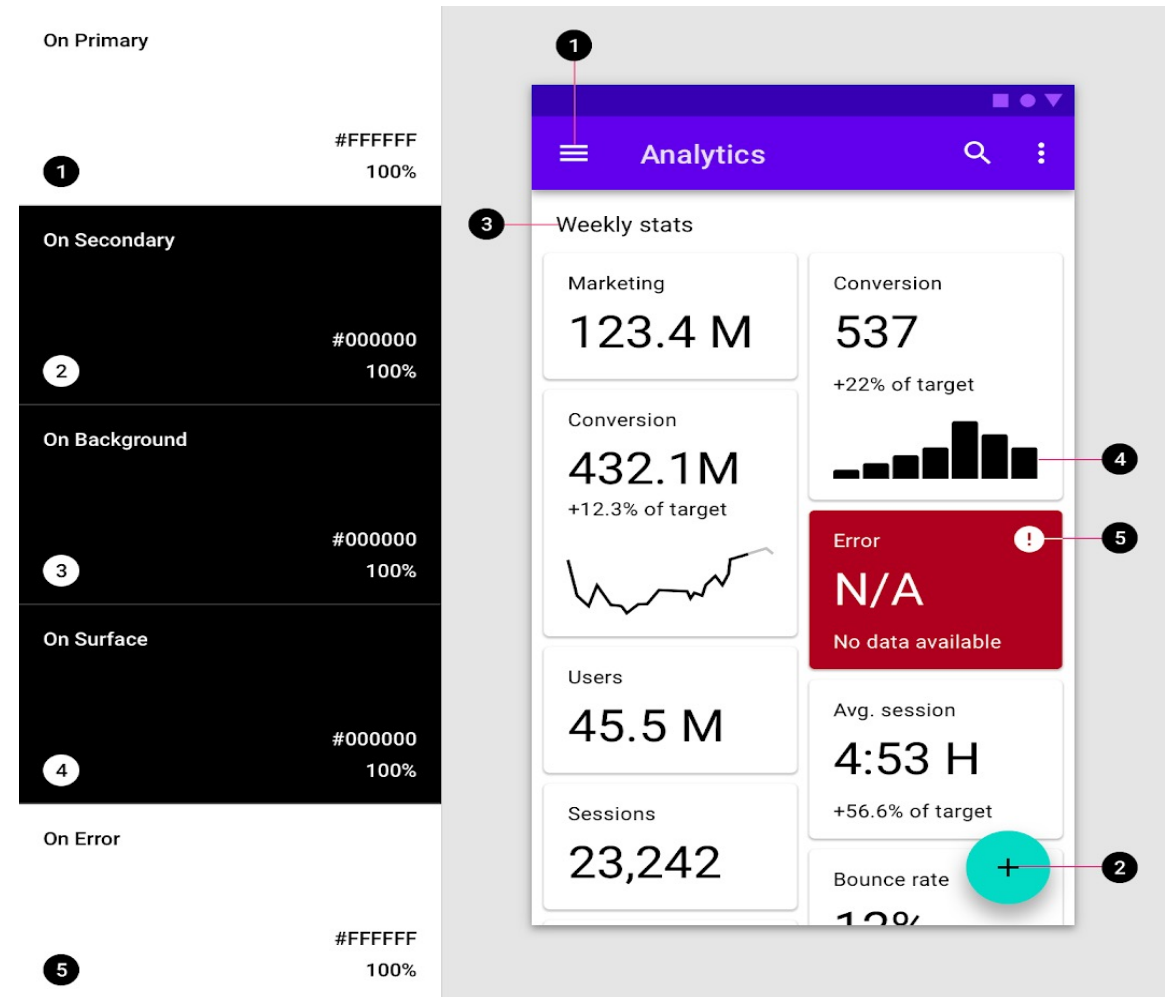
- Surface, background, and error colors typically don't represent brand:
- **Surface colors** affect surfaces of components, such as cards, sheets, and menus.
- The **background color** appears behind scrollable content. The baseline background and surface color is #FFFFFF.
- **Error color** indicates errors in components, such as invalid text in a text field. The baseline error color is #B00020.



TYPOGRAPHY AND ICONOGRAPHY COLORS

"On" colors

- App surfaces use colors from specific categories in your color palette, such as a primary color. Whenever elements, such as text or icons, appear in front of those surfaces, those elements should use colors designed to be clear and legible against the colors behind them.
- This category of colors is called “on” colors, referring to the fact that they color elements that appear “on” top of surfaces that use the following colors: a primary color, secondary color, surface color, background color, or error color. When a color appears “on” top of a primary color, it’s called an “on primary color.” They are labelled using the original color category (such as primary color) with the prefix “on.”
- “On” colors are primarily applied to text, iconography, and strokes. Sometimes, they are applied to surfaces.
- The default values for “on” colors are #FFFFFF and #000000.



ALTERNATIVE COLORS

- The Material Design color system supports **alternative colors**, which are colors used as alternatives to your brand's primary and secondary colors (they constitute additional colors to your theme). Alternative colors can be used to distinguish different sections of a UI.
 1. Alternative colors are best for:
 2. Apps with light and dark themes
 3. Apps with different themes in different sections
 4. Apps that are part of a suite of products
- Alternative colors should be used cautiously, because they can be challenging to implement cohesively with existing color themes.

ALTERNATIVE COLORS

Light and dark themes

- Some apps have both light and dark themes.
- To maintain visibility of elements and legibility of text, you can adapt the different color schemes for dark and light themes.



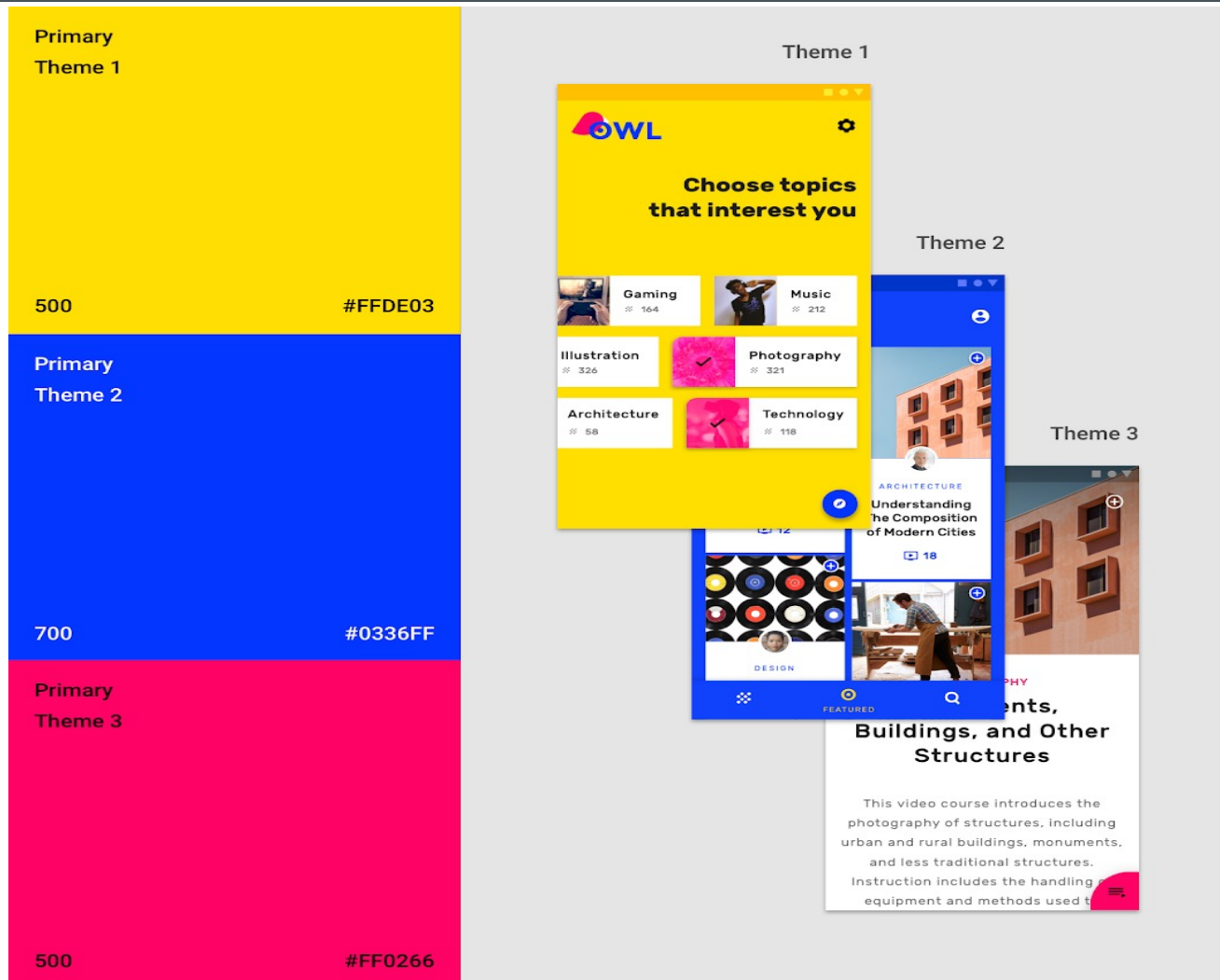
A news app in a light theme uses a primary and secondary scheme.



The same news app in dark theme uses a different color scheme to maintain legibility.

ALTERNATIVE COLORS FOR SECTION THEMES

- Alternative colors can be used to theme different parts of an app.
- This app has three primary colors.
- Distinct themes are used in different parts of the app, allowing users to better locate themselves within it.



ALTERNATIVE COLORS FOR SECTION THEMES

Theme I

- Yellow is used as the primary color for areas such as onboarding and choosing content of interest.

Primary
Option 1

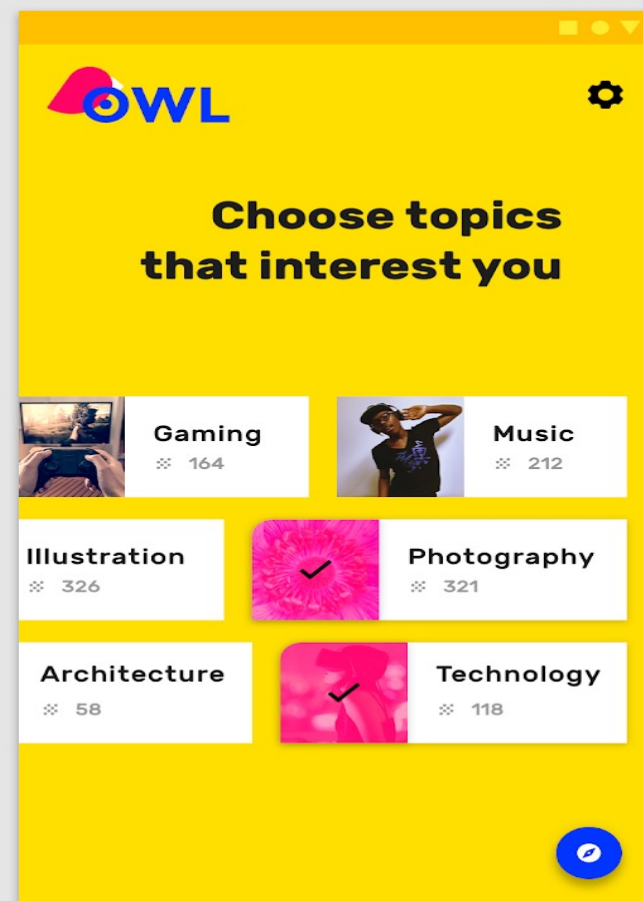
500

#FFDE03

Secondary
Option 1

700

#0336FF



ALTERNATIVE COLORS FOR SECTION THEMES

Theme 2

- Blue is used as the primary color for areas of the app that relate to the user's personal account, such as selected courses.

Primary
Owl Blue

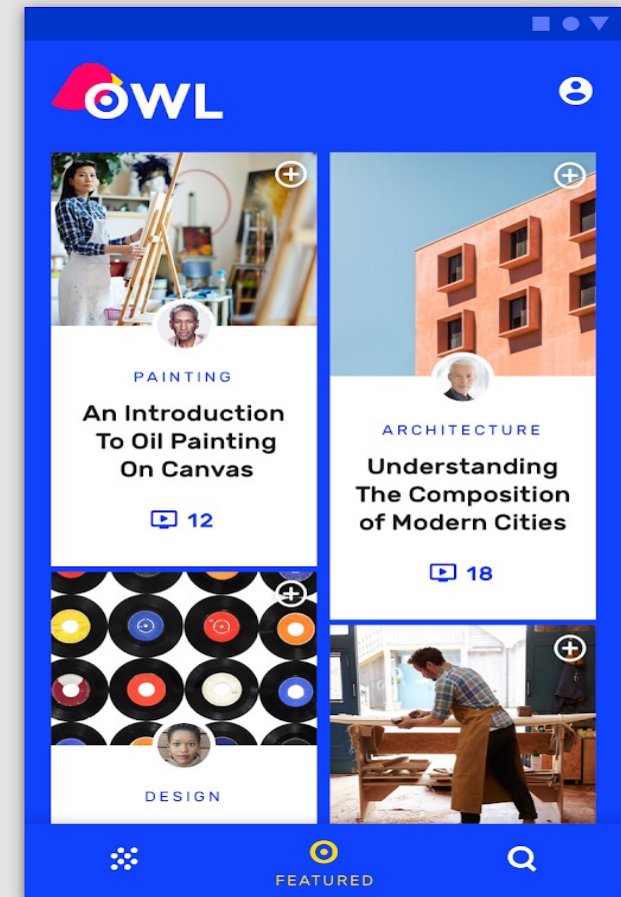
700

#0336FF

Primary
Owl Yellow

500

#FFDE03



ALTERNATIVE COLORS FOR SECTION THEMES

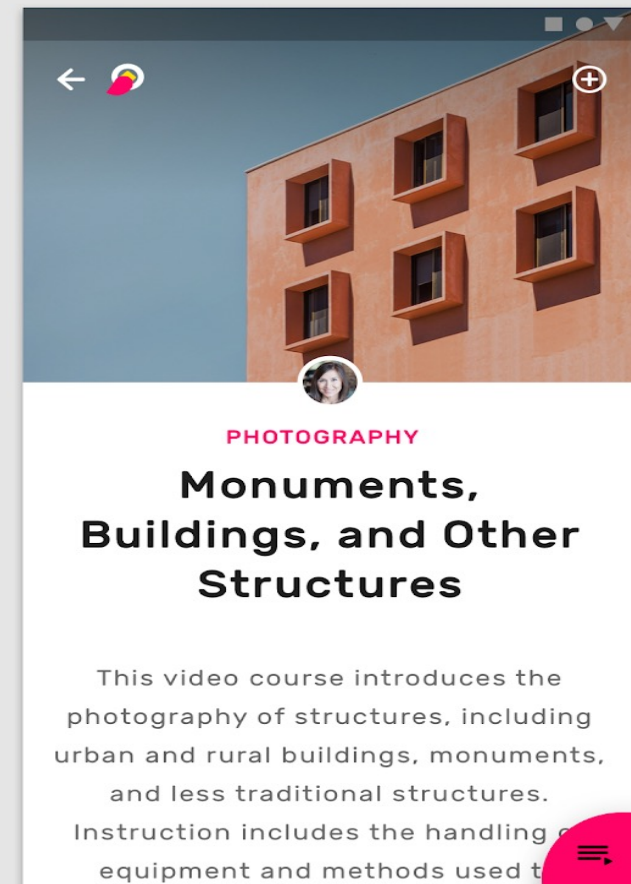
Theme 3

- Pink is used as the primary color for courses.

Primary
Option 3

500

#FF0266



TOOLS FOR PICKING COLORS

Tools for picking colors

Material palette generator

The Material palette generator can be used to generate a palette for any color you input. Hue, chroma, and lightness are adjusted by an algorithm that creates palettes that are usable and aesthetically pleasing.

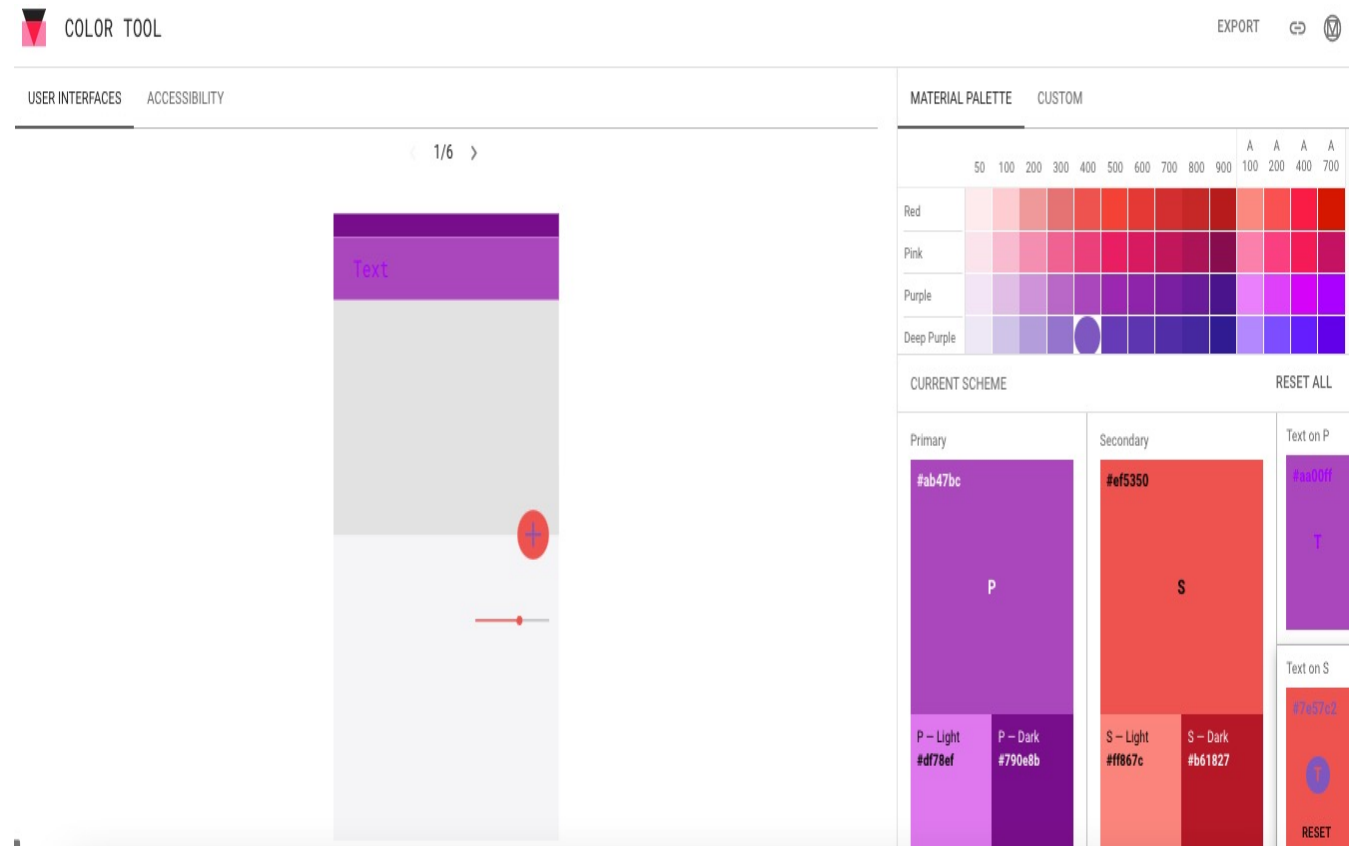
Input colors

Color palettes can be generated based on the primary input color, and whether the desired palette should be analogous, complementary, or triadic in relation to the primary color.

Alternatively, the tool can generate expanded palettes, based on any primary and secondary color.

Color variations for accessibility

These palettes provide additional ways to use your primary and secondary colors. They include lighter and darker options to separate surfaces and provide colors that meet accessibility standards.



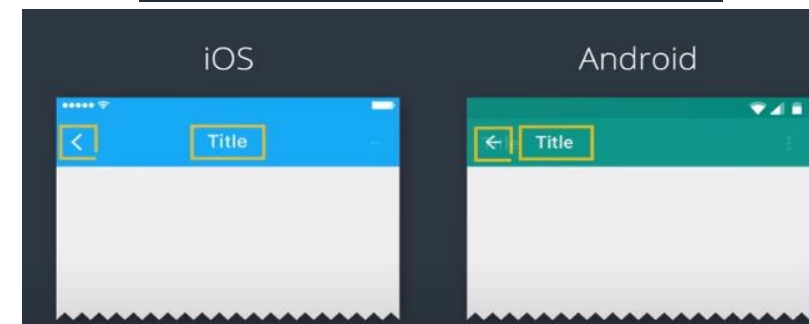
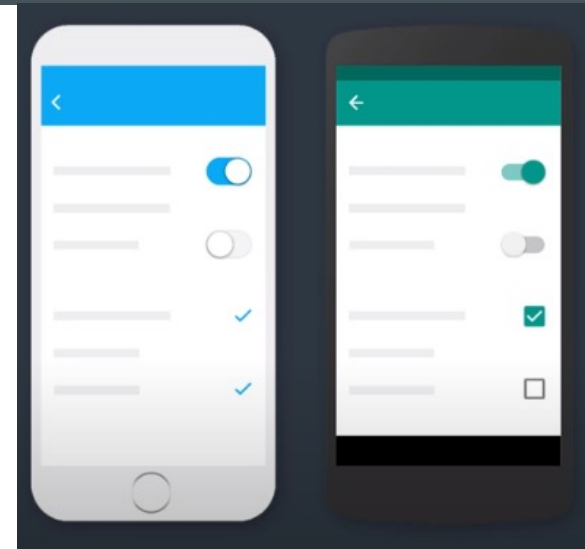
The screenshot displays the Material Color Tool interface. At the top, it says "COLOR TOOL" with a logo. There are tabs for "USER INTERFACES" and "ACCESSIBILITY". A navigation bar shows "1/6". The main area features a color selection tool with a hue slider and a color wheel. A preview of a mobile app interface is shown with a purple header and a red button. On the right, there is a "MATERIAL PALETTE" section with a grid of color swatches and a "CURRENT SCHEME" section showing a palette with primary and secondary colors and their variations.

<https://material.io/resources/color/>

BUILDING PLATFORM SPECIFIC UI (IOS & ANDROID)

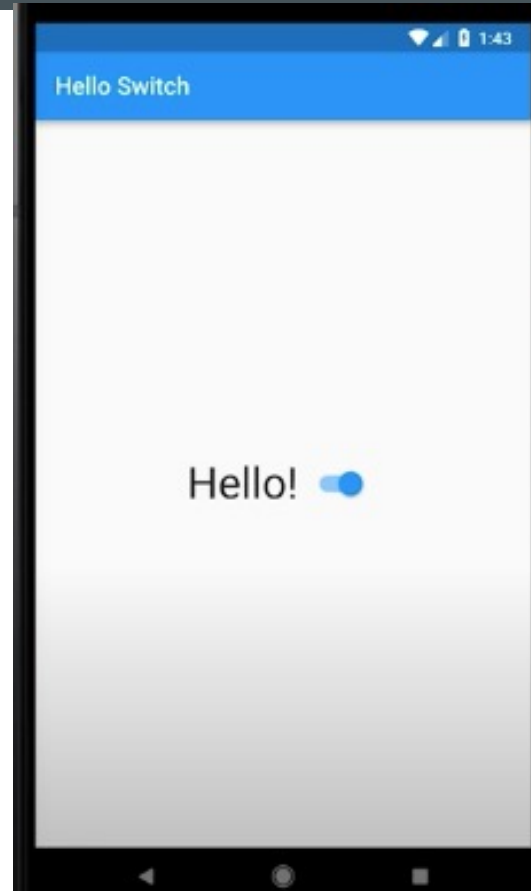
- With Flutter, we are able to design apps that look native to both iOS and Android using a single code base.
- For example, an **AppBar** will render differently on iOS and Android. The title text, position and font are appropriate to the platform as is the back navigation button.
- In Flutter, you can import **dart:io** and use Platform property to look up which platform you are currently running on. The API is quite nice:

```
import 'dart:io';  
  
Platform.isIOS // Returns true on iOS devices  
Platform.isAndroid // Returns true on Android devices
```



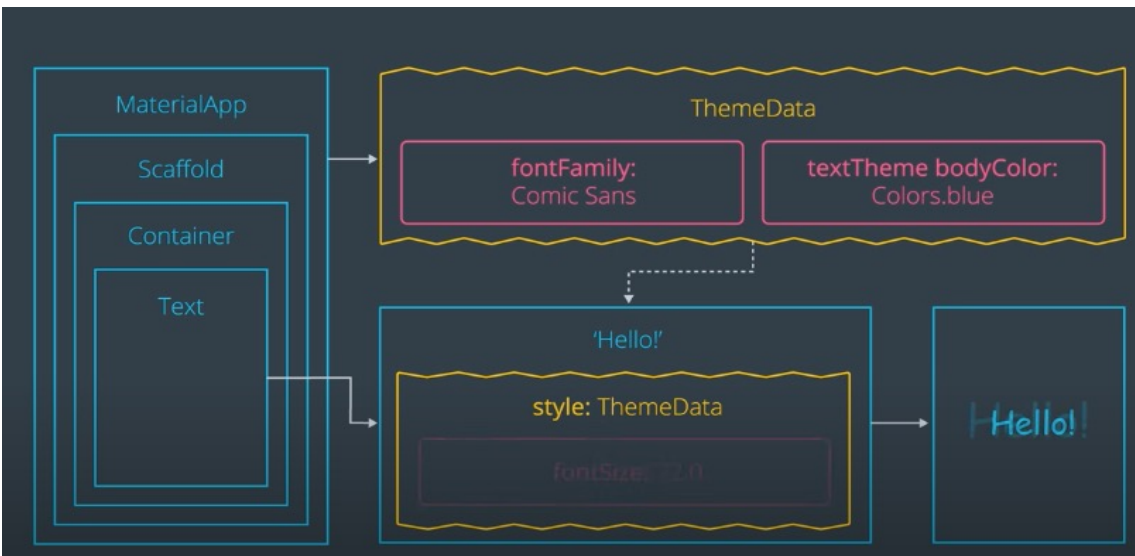
BUILDING PLATFORM SPECIFIC UI (IOS & ANDROID)

- To incorporate specific native widgets, we can use the themes platform property to determine the platform and build a relevant widgets accordingly in either the **Material** or **Cupertino** style.
- Here is a **Material** switch when we toggle between device platforms via the Flutter inspector, the switch style does not change.
- To use **Cupertino** widgets, we import the **Cupertino.dart** package.
- This allows us to use the Cupertino switch.
- We will check if the platform is **iOS**, and if it is, we show the **CupertinoSwitch**. Now, when we toggle between devices, the iOS version shows the default **iOS** toggle.



BUILDING PLATFORM SPECIFIC UI (IOS & ANDROID)

- You can customize the master theme and its properties.
- This master theme is propagated down the widget tree. Child widgets are able to inherit the master theme's styling.
- The child widget can also override the theme and customize their styling.



```
body: HelloSwitch(),
),
),
);
}

class HelloSwitch extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'Hello!',
            style: TextStyle(fontSize: 36.0),
          ),
          Theme.of(context).platform == TargetPlatform.iOS
            ? CupertinoSwitch(
                value: true,
                onChanged: (bool toggled) {},
              )
            : Switch(
                value: true,
                onChanged: (bool toggled) {},
              ),
        ],
      ),
    );
  }
}
```

CREATE AND STYLE A TEXT FIELD

- Text fields allow users to type text into an app. They are used to build forms, send messages, create search experiences, and more. In this recipe, explore how to create and style text fields.
- Flutter provides two text fields: [TextField](#) and [TextFormField](#).

TextField

`TextField` is the most commonly used text input widget.

By default, a `TextField` is decorated with an underline. You can add a label, icon, inline hint text, and error text by supplying an `InputDecoration` as the `decoration` property of the `TextField`. To remove the decoration entirely (including the underline and the space reserved for the label), set the `decoration` to null.

```
TextField(  
  decoration: const InputDecoration(  
    border: OutlineInputBorder(),  
    hintText: 'Enter a search term'  
  ),  
);
```



CREATE AND STYLE A TEXT FIELD

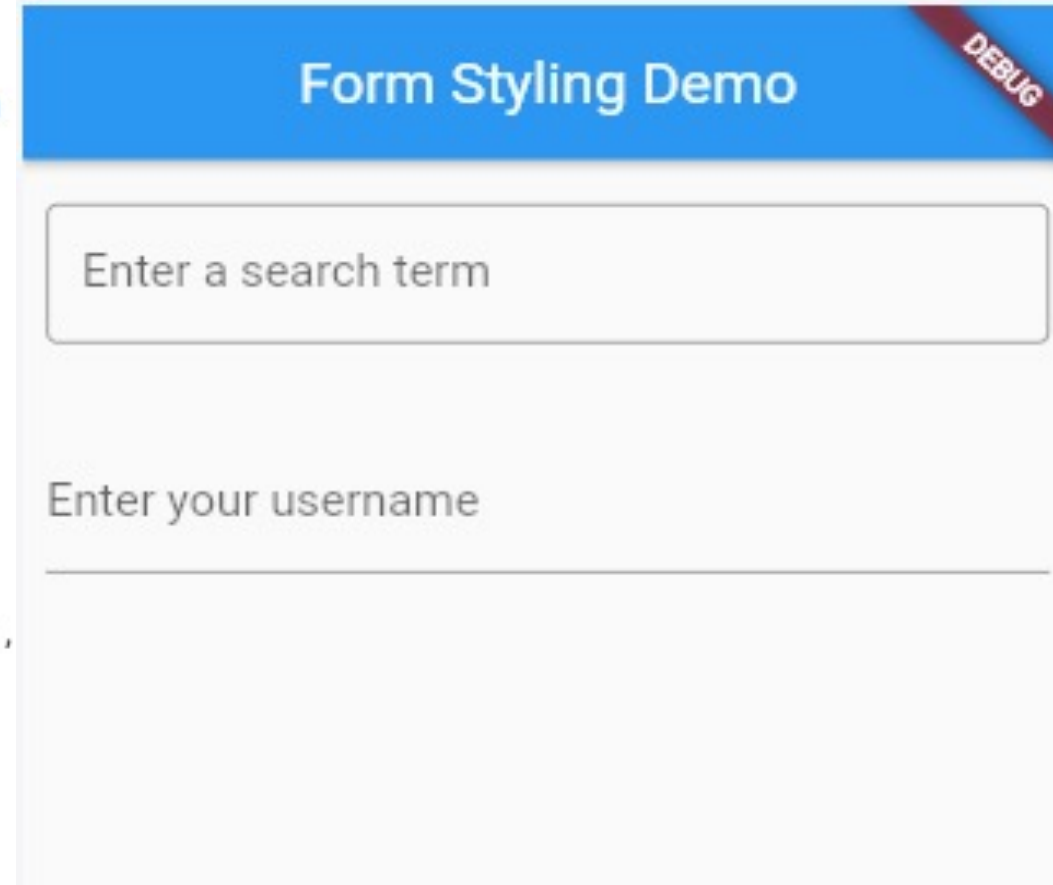
TextFormField

`TextFormField` wraps a `TextField` and integrates it with the enclosing `Form`. This provides additional functionality, such as validation and integration with other `FormField` widgets.

```
TextFormField(  
  decoration: const InputDecoration(  
    border: UnderlineInputBorder(),  
    labelText: 'Enter your username'  
  ),  
);
```

INTERACTIVE EXAMPLE

```
@override
Widget build(BuildContext context) {
  return Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: <Widget>[
      const Padding(
        padding: EdgeInsets.symmetric(horizontal: 8, vertical: 16),
        child: TextField(
          decoration: InputDecoration(
            border: OutlineInputBorder(),
            hintText: 'Enter a search term',
          ),
        ),
      ),
      Padding(
        padding: const EdgeInsets.symmetric(horizontal: 8, vertical: 16),
        child: TextFormField(
          decoration: const InputDecoration(
            border: UnderlineInputBorder(),
            labelText: 'Enter your username',
          ),
        ),
      ),
    ],
  );
}
```



HANDLE CHANGES TO A TEXT FIELD

- In some cases, it's useful to run a callback function every time the text in a text field changes. For example, you might want to build a search screen with autocomplete functionality where you want to update the results as the user types.

How do you run a callback function every time the text changes? With Flutter, you have two options:

1. Supply an `onSubmitted` **or** `onChanged()` callback to a `TextField` or a `TextFormField`.
2. Use a `TextEditingController`.

1. Supply an `onChanged()` callback to a `TextField` or a `TextFormField`

The simplest approach is to supply an `onChanged()` callback to a `TextField` or a `TextFormField`. Whenever the text changes, the callback is invoked.

In this example, print the current value of the text field to the console every time the text changes.

```
TextField(  
  onChanged: (text) {  
    print('First text field: $text');  
  },  
);
```

Retrieving Text

- `onChanged`
- `onSubmitted`
- `controller`

HANDLE CHANGES TO A TEXT FIELD

2. Use a `TextEditingController`

A more powerful, but more elaborate approach, is to supply a `TextEditingController` as the `controller` property of the `TextField` or a `TextFormField`.

To be notified when the text changes, listen to the controller using the `addListener()` method using the following steps:

1. Create a `TextEditingController`.
2. Connect the `TextEditingController` to a text field.
3. Create a function to print the latest value.
4. Listen to the controller for changes.

```
class _MyCustomFormState extends State<MyCustomForm> {  
  // Create a text controller. Later, use it to retrieve the  
  // current value of the TextField.  
  final myController = TextEditingController();  
}
```

Connect the `TextEditingController` to a text field

Supply the `TextEditingController` to either a `TextField` or a `TextFormField`. Once you wire these two classes together, you can begin listening for changes to the text field.

```
TextField(  
  controller: myController,  
),
```

```
@override  
void initState() {  
  super.initState();  
  
  // Start listening to changes.  
  myController.addListener(_printLatestValue);  
}
```

Create a function to print the latest value

You need a function to run every time the text changes. Create a method in the `_MyCustomFormState` class that prints out the current value of the text field.

```
void _printLatestValue() {  
  print('Second text field: ${myController.text}');  
}
```

GESTURES

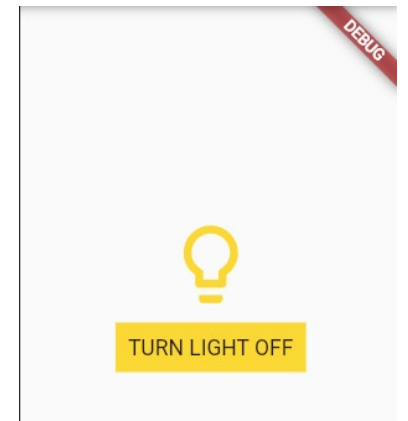
- Responding to gestures is vital to an interactive app. Examples of gestures include **taps**, **drags** and **scaling**.
- If we want to add our own interactivity to any widget, we can wrap the widget in a **GestureDetector**. **Gestures** can make an app experience feel much more seamless.
- Material design applications typically react to touches with ink splash effects. The [InkWell](#) class implements this effect and can be used in place of a [GestureDetector](#) for handling taps.
- If this widget has a child, it defers to that child for its sizing behavior. If it does not have a child, it grows to fit the parent instead.
- **GestureDetector widget** Attempts to recognize gestures that correspond to its non-null callbacks.

Gestures

- onTapDown
- onTap
- onDoubleTap
- onLongPress
- onVerticalDragStart
- onHorizontalDragUpdate

EXAMPLE

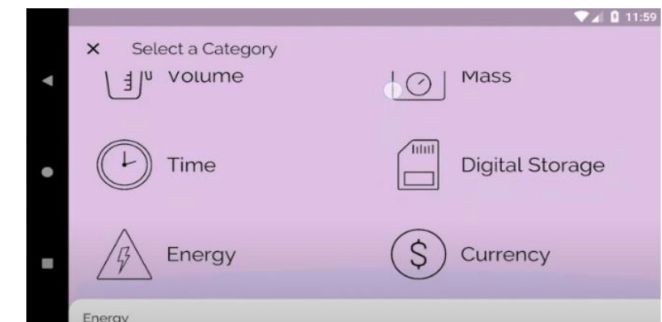
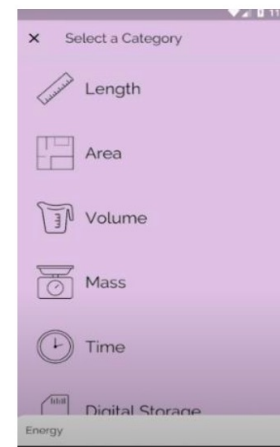
```
/// This is the private State class that goes with MyStatefulWidget.  
class _MyStatefulWidgetState extends State<MyStatefulWidget> {  
  bool _lightIsOn = false;  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: Container(  
        alignment: FractionalOffset.center,  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.center,  
          children: <Widget>[  
            Padding(  
              padding: const EdgeInsets.all(8.0),  
              child: Icon(  
                Icons.lightbulb_outline,  
                color: _lightIsOn ? Colors.yellow.shade600 : Colors.black,  
                size: 60,  
              ),  
            ),  
          ],  
        ),  
      ),  
    );  
  }  
}
```



```
GestureDetector(  
  onTap: () {  
    setState(() {  
      // Toggle light when tapped.  
      _lightIsOn = !_lightIsOn;  
    });  
  },  
  child: Container(  
    color: Colors.yellow.shade600,  
    padding: const EdgeInsets.all(8),  
    // Change button text when light changes state.  
    child: Text(_lightIsOn ? 'TURN LIGHT OFF' : 'TURN LIGHT ON'),  
  ),  
),  
),  
),  
),  
),  
);  
}
```

CREATING RESPONSIVE AND ADAPTIVE APPS

- One of Flutter's primary goals is to create a framework that allows you to develop apps from a single codebase that look and feel great on any platform.
- This means that your app may appear on screens of many different sizes, from a watch, to a foldable phone with two screens, to a high def monitor.
- Two terms that describe concepts for this scenario are ***adaptive*** and ***responsive***.
- Ideally, you'd want your app to be *both* but what, exactly, does this mean? These terms are similar, but they are not the same.



THE DIFFERENCE BETWEEN AN ADAPTIVE AND A RESPONSIVE APP

Adaptive and *responsive* can be viewed as separate dimensions of an app:

HOWEVER, you can have an adaptive app that is not responsive, or vice versa.

And, of course, an app can be both, or neither.

- **Responsive :**

Typically, a *responsive* app has had its layout tuned for the available screen size. Often this means (for example), re-laying out the UI if the user resizes the window, or changes the device's orientation. This is especially necessary when the same app can run on a variety of devices, from a watch, phone, tablet, to a laptop or desktop computer.

- **Adaptive**

Adapting an app to run on different device types, such as mobile and desktop, requires dealing with mouse and keyboard input, as well as touch input. It also means there are different expectations about the app's visual density, how component selection works (cascading menus vs bottom sheets, for example), using platform-specific features (such as top-level windows), and more.

CREATING A RESPONSIVE FLUTTER APP

- Flutter allows you to create apps that self-adapt to the device's screen size and orientation.

There are two basic approaches to creating Flutter apps with responsive design:

- Use the [LayoutBuilder](#) class

From its [builder](#) property, you get a [BoxConstraints](#) object. Examine the constraint's properties to decide what to display. For example, if your [maxWidth](#) is greater than your **width** breakpoint, return a [Scaffold](#) object with a row that has a list on the left. If it's narrower, return a [Scaffold](#) object with a drawer containing that list.

You can also adjust your display based on the device's height, the aspect ratio, or some other property. When the constraints change (for example, the user rotates the phone, or puts your app into a tile UI in Nougat), the build function runs.

- Use the [MediaQuery.of\(\)](#) method in your build functions

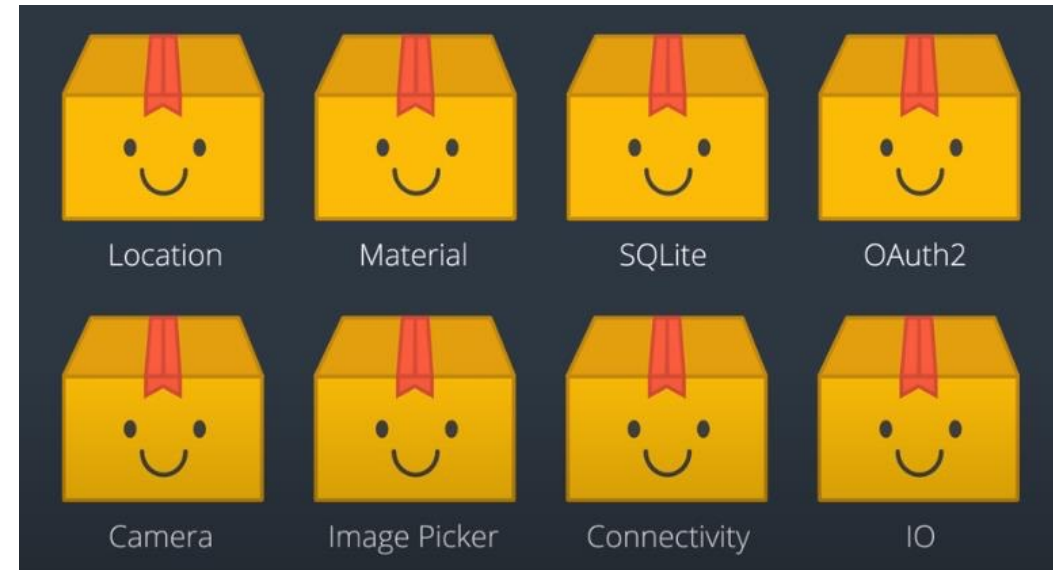
This gives you the size, orientation, etc, of your current app. This is more useful if you want to make decisions based on the complete context rather than on just the size of your particular widget. Again, if you use this, then your build function automatically runs if the user somehow changes the app's size.

Other useful widgets and classes for creating a responsive UI:

[AspectRatio](#), [CustomSingleChildLayout](#), [CustomMultiChildLayout](#), [FittedBoxFractionallySizedBox](#), [LayoutBuilder](#), [MediaQuery](#), [MediaQueryData](#), [OrientationBuilder](#).

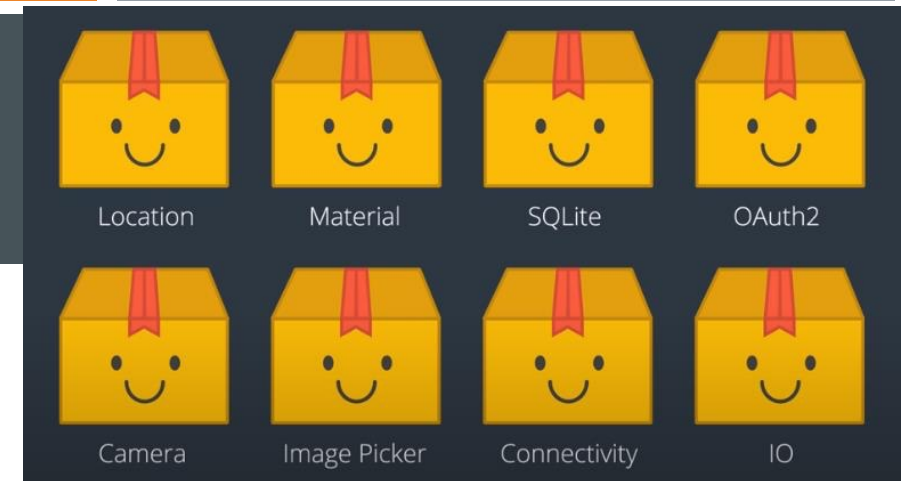
PACKAGES, PLUGINS, AND PUBSPEC.YAML

- Just like with **widgets**, we don't want to reinvent the wheel each time we start to add a big feature to our app. **Login, authentication, network calls**, what should we use?
- The answer is **packages**.
- **Package** is a library of functions that can be shared easily.
- **Packages** enable the creation of **modular code** that can be shared easily.
- We can use both **Flutter and Dart packages**.
- Some integrate with device **APIs** such as the **Battery package**. These are called **plug-ins** as they interface with either the **iOS or Android** platforms.
- Others, such as the **Firestore package**, are just called **regular packages**.
- We already used the **material package** and we will also be using the **Dart IO package to connect to our API**. You can also write and publish your own **packages**.



PACKAGES, PLUGINS, AND PUBSPEC.YAML

- In order to use **packages**, we have to specify our **dependencies** in a **pubspec.yaml** file which contains **metadata** for our app. Information that you include into
- **Pubspec** include the **name, version, description, authors, dependencies, and more.**
- For Flutter applications, a **dependency would be the Flutter SDK.** We can also specify **assets** and **fonts** inside **Pubspec.**
- When you make changes to **Pubspec.yaml**, you'll want to **run a flutter packages PUB GET** this will *gets or updates the required packages that your app depends on.*
- Even though you import a large **package**, only the functions you use end up being compiled down to code in **release mode.** This is because Flutter uses **tree shaking** to **remove redundant and unused code** in the **compilation process** for *the binary used in production.*
- You can search for **packages** on the **Dart packages site.** Using **packages** and **plugins** can make your development that much more efficient.



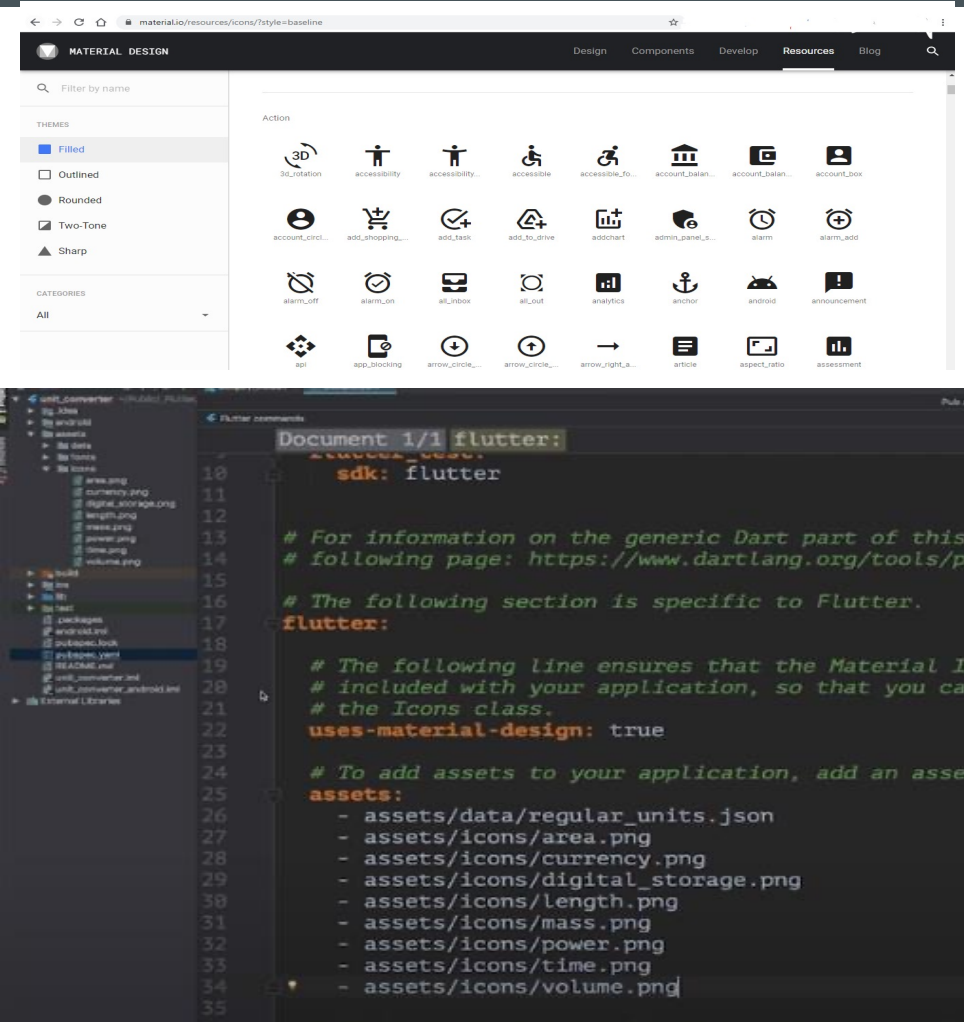
Tree-shaking

Process where redundant and unused code is removed during code compilation.



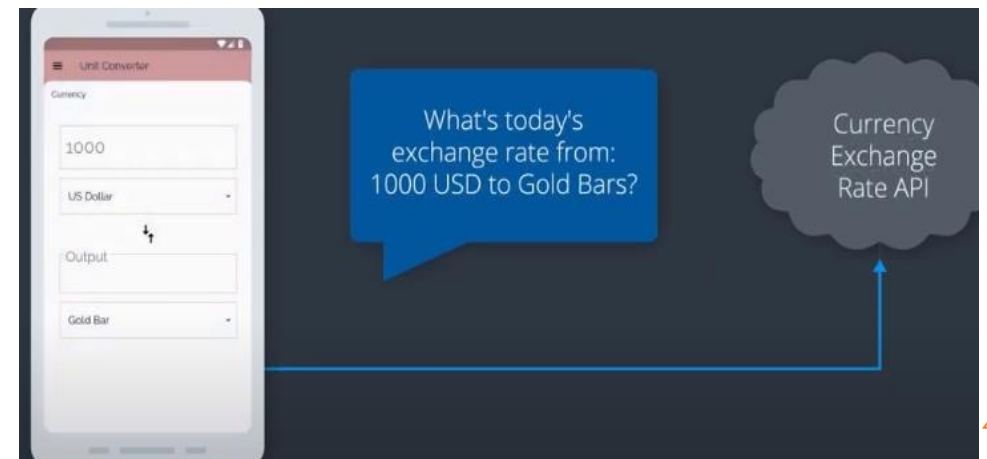
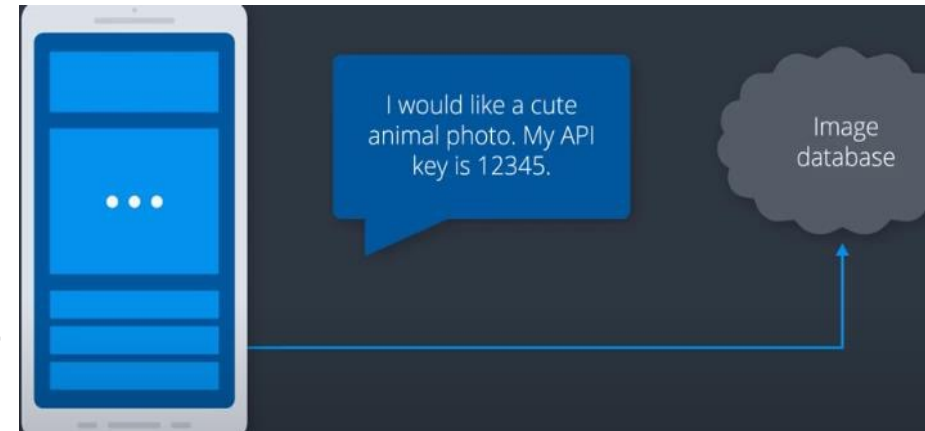
IMAGE AND ICON ASSETS

- **Icons** and **images** are also saved and managed within the **assets** directory and **pubspec.yaml** file.
- **Material design** provides over **built-in icons**, such as **play, refresh, alarm, pets, insert photo, and more**. The material components also incorporate these icons.
- You can use them by setting "**Use material icon to true**" in your apps **pubspec.yaml** file.
- These icons can be used in icon buttons which let you specify a function to run when the icon is tapped.
- The flutter **image widget** has separate **constructors** based on whether your **path** points to an **asset, local file, or from the Web**.



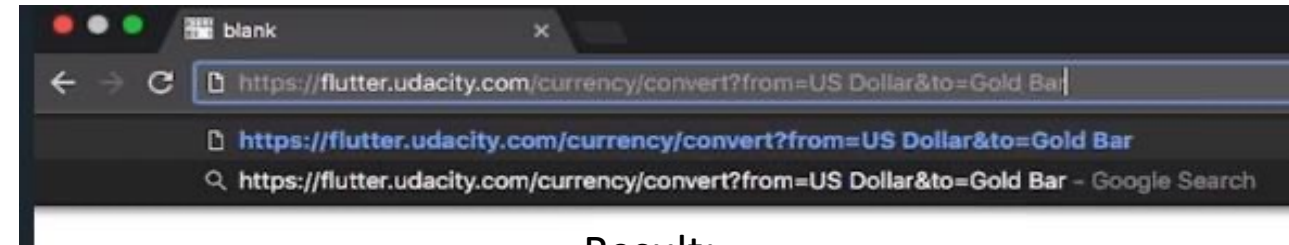
NO ASSETS? USE AN API!

- Sometimes, the data you want to save changes. So, you can't save it in an **asset**.
- For example, you might want to randomly select a **cute animal picture** from an **API** *each day or for this app*, you might want to know this moment's **exchange rate for currencies**.
- For this, we will call an **API** which **retrieves real-time data**. Connecting to **APIs** can be done with the **HTTP client in dart:io**.
- We create a **HTTP client** that points to our **endpoint**. Our current app does not require **authentication** or an **API key**.
- So we just hit the **endpoint** with our **amount query parameters** to get the **unit conversion** that we want.



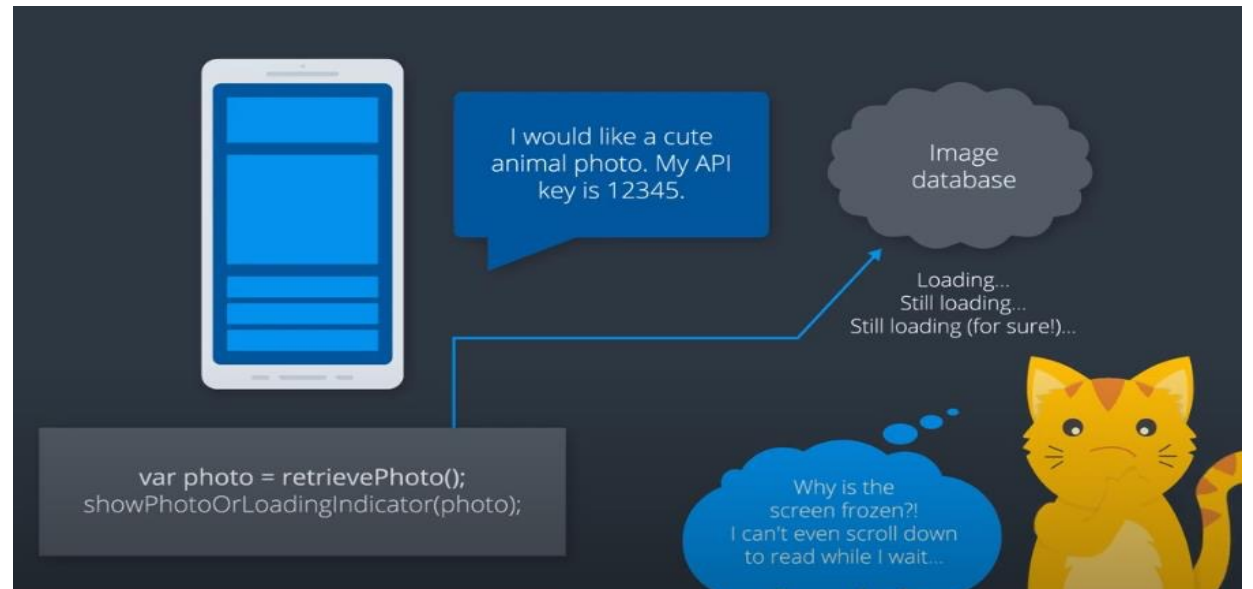
NO ASSETS? USE AN API!

- You can try out some **queries** by going to flutter.udacity.com/currency in your **browser**.
- An API doesn't immediately return your data the way a local asset would because Dart runs in a single thread.
- If we solely wait for the **API call to return**, we would see **a frozen screen** and we wouldn't be able to interact with the app.
- The **API call** may take some time to return based on the **server's speed, your Internet connection, and other factors**.



Result:

```
{"status": "ok", "conversion": 0.40439439630352586}
```

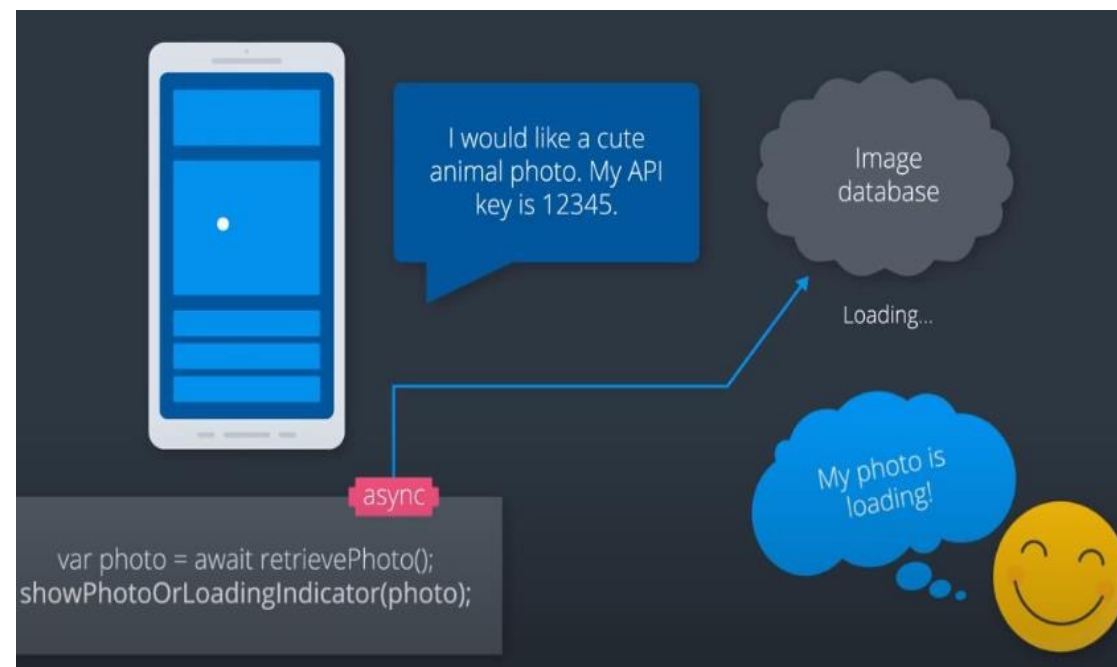


NO ASSETS? USE AN API!

- Rather than wait for it, we wrap the API in an asynchronous operation.
- This lets your app continue to run without getting blocked.
- **Dart uses *future objects* to represent *asynchronous operations*.**
- When a **function** that returns a **future** is invoked, two things happen.
- *First, the function cues up work to be done and returns an incomplete future object.*
- *Later, when a value is available, the future object completes with that value or with an error. We'll discuss **errors** later.*

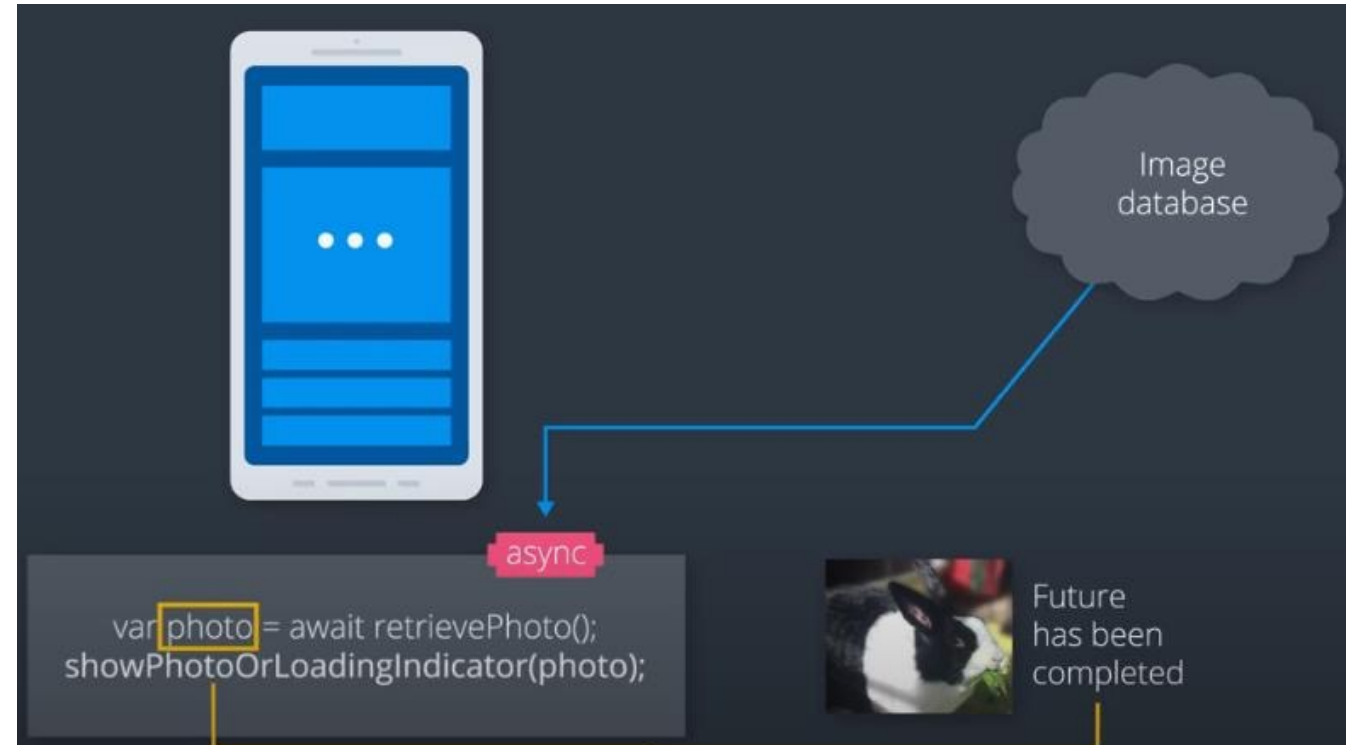
Future

A Future represents a means for getting a value sometime in the future, used in asynchronous operations.



NO ASSETS? USE AN API!

- We save the **value** that the **future** returns into a **variable** and **call await on the function**.
- We need to wrap the function in which this is called with an **async keyword**.
- Let's look at how we do this for the **convert API call**. First, we make a **request** to our **URI**.
- Note that it **returns a future object a double**. We also add the **async keyword** to our convert function. Now let's get our response. This is also an **asynchronous operation**.
- So we add the **await keyword**.
- We have to **decode our JSON response body** too before we **parse** it and **return** it to the **units**.
- This is also an **async operation**. Then, we return the conversion as a **double**.



Now you know both how to retrieve live data from an API and how to use **asynchronous functions**.

KEY POINTS

- Material design is very useful and has interesting online tools to show you best practices Regarding UI and UX design.
- Material Design color system can help you create a color theme that reflects your brand or style.
- Sometimes you need to get a specific native widget or behavior then you can check the device Platform then customize your content using Cupertino and Material libraries.
- You learned about getting user text input and how to handle it.
- Images, Icons, and Fonts are useful assets that you can use or import into your app.
- If your App data changes! , then you need to use an API to get data from Internet.