Hello Flutter

tl;dr

- What the heck is Flutter?
- The pros and cons of using Flutter vs. ten other options.
- Why Flutter is a better choice and why it is a worse choice.
- The three barriers to learning Flutter and how we'll tackle them.


We need a web app. It'll be so much easier for people to do business with us.
I have to hire a web developer.


We love your web app, but we'd have been here earlier if you were in the App Store.
Umm, how do I get into the App Store?
Hire an iOS developer who knows Swift or Objective C.


Hey, most of us have Android devices. Why aren't you in the Play Store?
How do I get into the Play Store?
Hire an Android dev who knows Kotlin or Java.

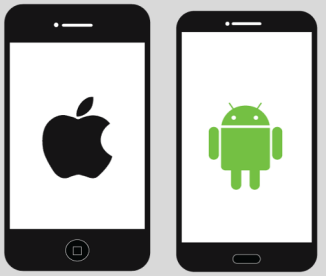

But I can't afford three developers!
Isn't there some way I can hire one person who can develop on all platforms?

Cross-platform options

Progressive Web Apps

Hybrid

Native Solutions



100% written in HTML, CSS, and JavaScript. Runs in a browser

Pros

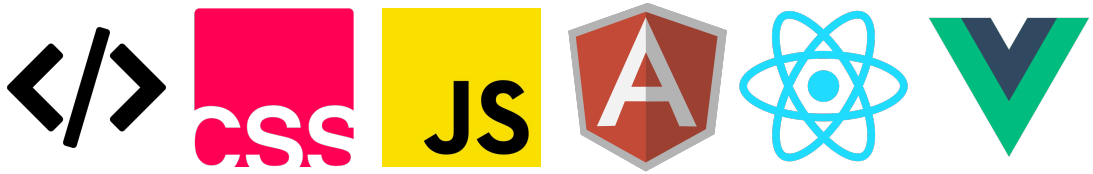
Easy to write

Cons

- ✖ Not a real app.
- ✖ Not available in app stores.
- ✖ Hard to create a desktop shortcut.
- ✖ Cannot access many of the device's resources like accelerometer, compass, proximity sensor, bluetooth, NFC, and more

Progressive Web Apps

Some tech



Frameworks create a native app with a WebView where your HTML/CSS/JavaScript run

Pros

Easier for web devs to learn because it uses HTML and JavaScript
It is in the stores

Cons

- ✖ Runs in a WebView so it can be slow

Hybrid

Some tech

You're writing a truly native app, indistinguishable from one written in Swift or Kotlin.

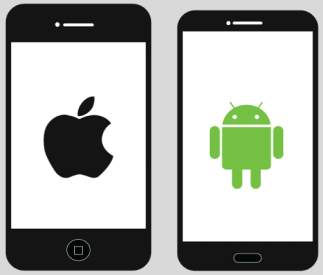
Pros

HIghest quality choice
Real apps that can be found in the stores
Runs fast

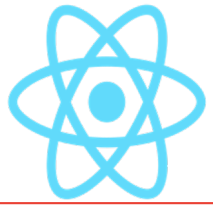
Cons

- ✖ Learning a framework may be difficult.
- ✖ Mastering the toolchain definitely is!

Native Solutions



Some tech

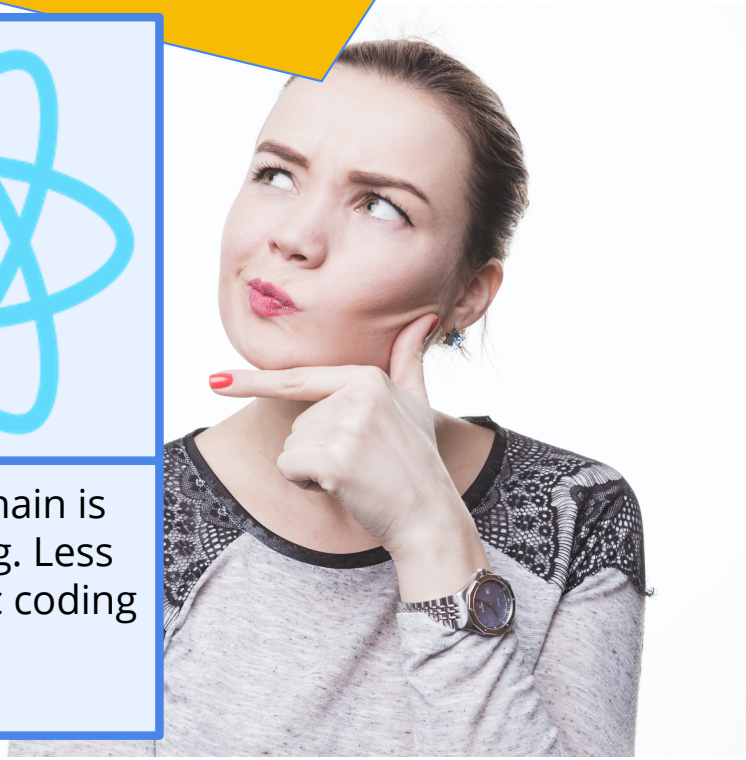
Warning: This is a very opinionated slide. YMMV

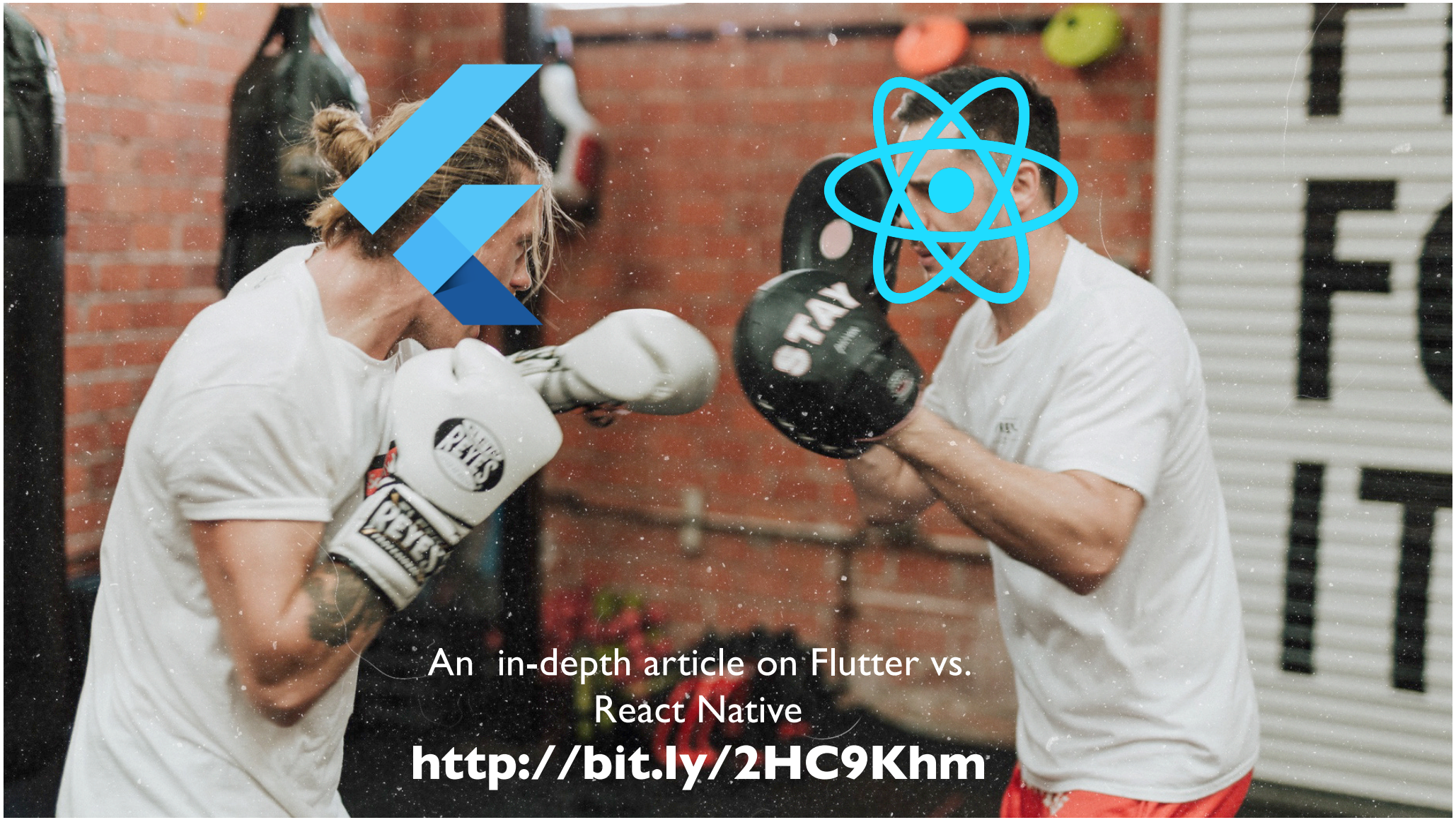
But NativeScript, Xamarin and React Native are cross-platform. What does Flutter do better?

Flutter has more community help and is backed by Google

Flutter has a better development experience. Less device-specific coding

Flutter's toolchain is less frustrating. Less device-specific coding




An in-depth article on Flutter vs.
React Native
http://bit.ly/2HC9Khm


Everyone has these three barriers to learning Flutter
Setup
Learning Dart
Flutter widgets

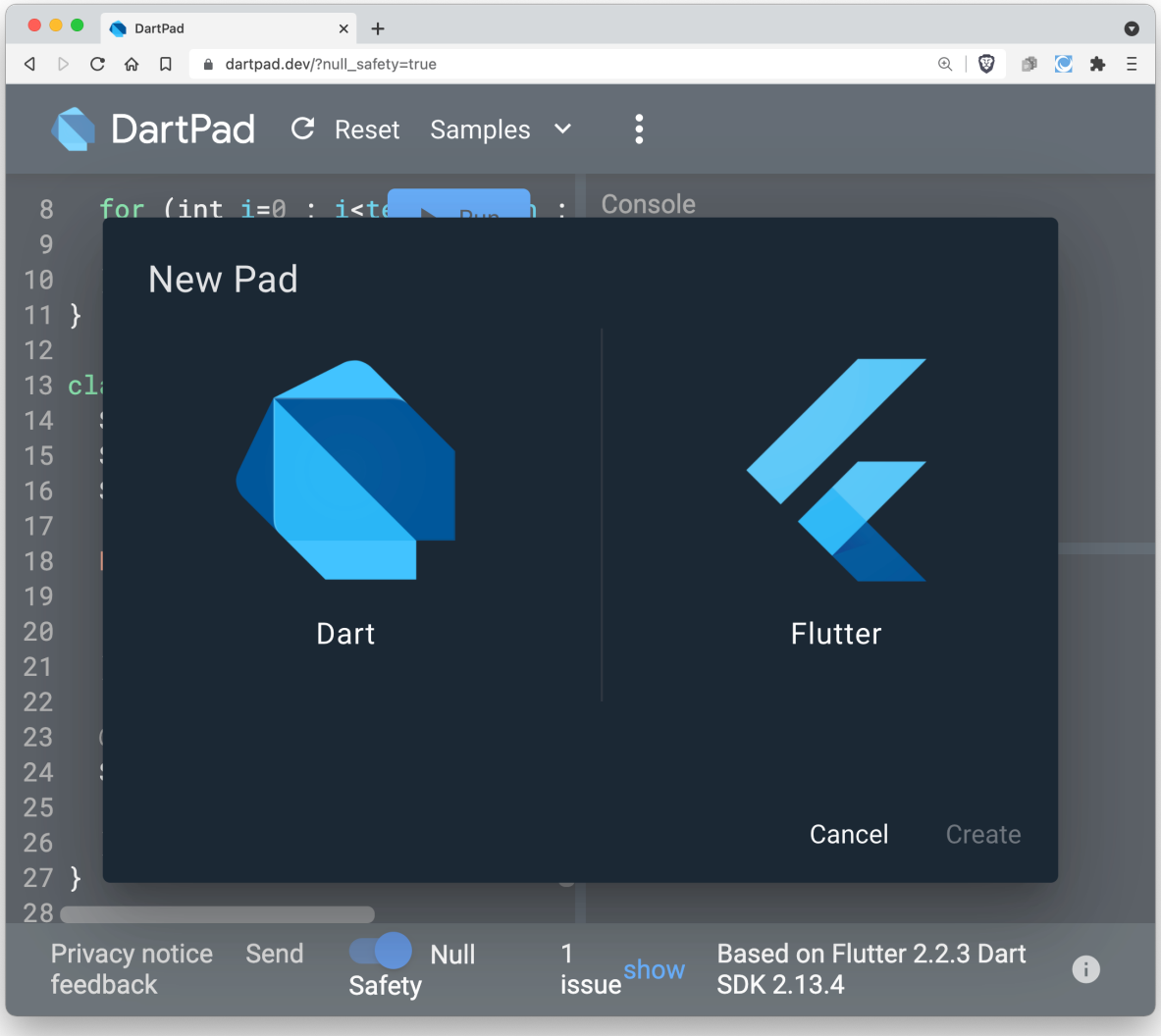
1. Setup of the dev environment

2. Learning Dart

If you know Java, C#, or JavaScript, it will be easy!

3. Learning the Flutter widgets

- That's the bulk of the course.
- Let's look at a sneak peak, though



tl;dr

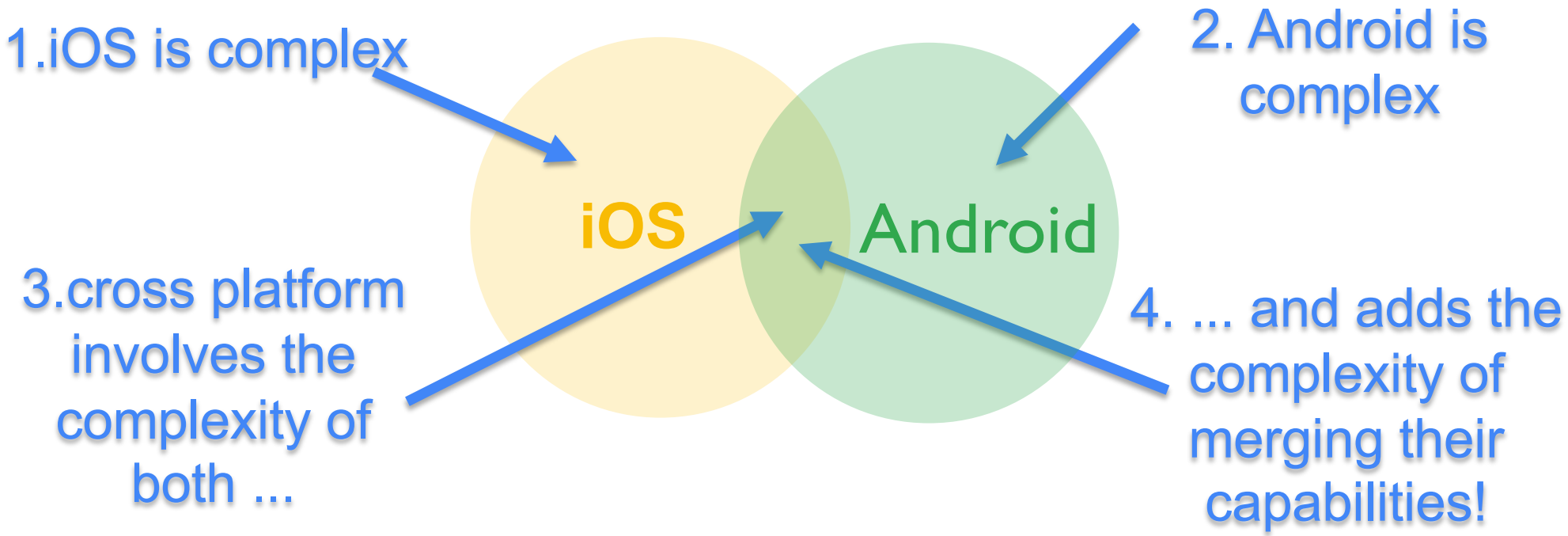
- What the heck is Flutter?
- The pros and cons of using Flutter vs. ten other options.
- Why Flutter is a better choice and why it is a worse choice.
- The three barriers to learning Flutter and how we'll tackle them.

Developing in Flutter

tl;dr

- Why is cross-platform development so difficult?
- What tools are needed for it?
- Where does the flutter SDK fit in?
- Is there a tool to help manage the complexity?

Developing cross-platform is VERY complex



What tools do I need?

What hardware need?

How do I run my app?

Where do I get them?

What files are ne

How do I test it?

How create an app?

You want to build a Flutter app but ...

- **Flutter SDK**
- **IDEs (VS Code, Android Studio)**
- **IDE plug ins**
- **Underlying requirements**
  - **Things to compile to iOS**
  - **Things to compile to Android**
- **Emulators/simulators**
- **Profilers**
- **Visual layout debuggers**
- **... and more!**

We'll **need** some of these

and

want others

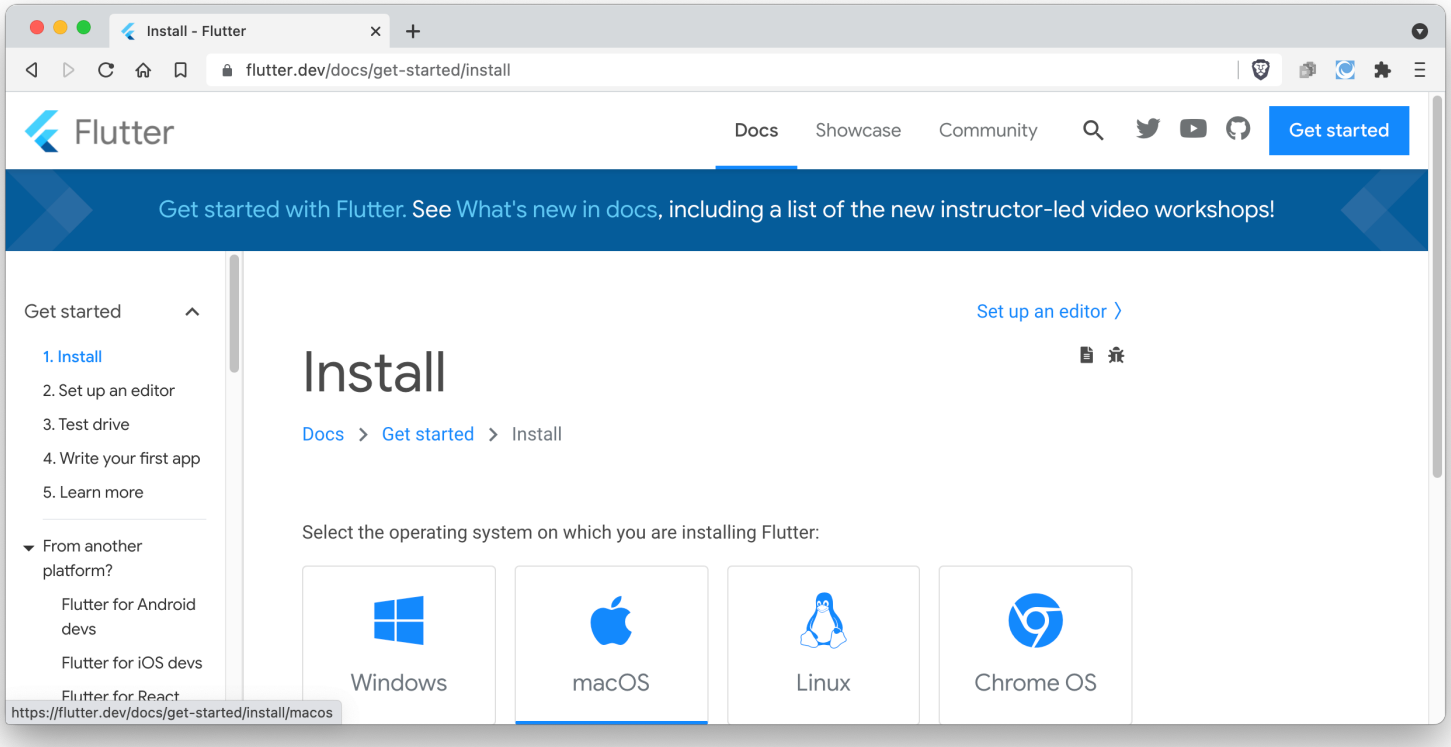
Flutter SDK

The Flutter SDK is the only thing that is 100% required

- All the widgets
- Compiler to ipa
- Compiler to apk
- Compiler to browser app
- Compiler to desktop apps
- apis for unit tests
- Headless test runner
- dart command line tool
- flutter command line tool
- interop for 3rd party SDKs

Okay, okay! How do I get it?

Download it from flutter.dev



The flutter command

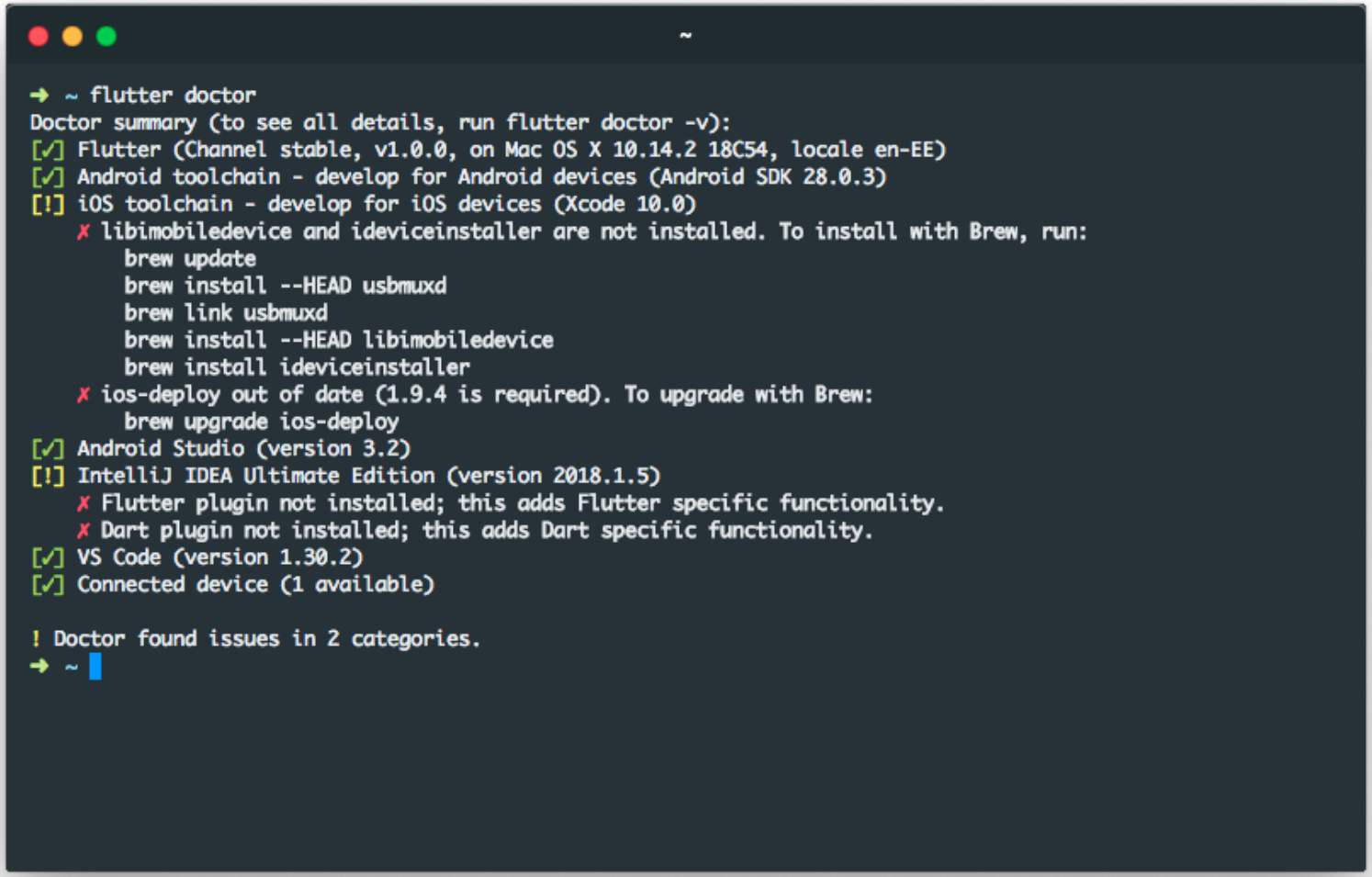
analyze	flutter analyze -d <DEVICE_ID>	Analyzes the project's Dart source code.
build	flutter build <DIRECTORY>	Flutter build commands.
channel	flutter channel <CHANNEL_NAME>	List or switch flutter channels.
config	flutter config	Configure Flutter settings. To remove a setting, configure it to an empty string.
create	flutter create <DIRECTORY>	Creates a new project.
devices	flutter devices -d <DEVICE_ID>	List all connected devices.
doctor	flutter doctor	Show information about the installed tooling.
downgrade	flutter downgrade	Downgrade Flutter to the last active version for the current channel.
emulators	flutter emulators	List, launch and create emulators.
format	flutter format <DIRECTORY>	Formats Flutter source code.
gen-l10n	flutter gen-l10n <DIRECTORY>	Generate localizations for the Flutter project.
install	flutter install -d <DEVICE_ID>	Install a Flutter app on an attached device.
logs	flutter logs	Show log output for running Flutter apps.
pub	flutter pub <PUB_COMMAND>	Works with packages.
run	flutter run <DART_FILE>	Runs a Flutter program.
test	flutter test <DIRECTORY>	Runs tests in this package.
upgrade	flutter upgrade	Upgrade your copy of Flutter.

flutter doctor

Analyzes your entire toolchain and prescribes solutions for broken parts.

```
$ flutter doctor
Doctor summary (to see all details, run flutter doctor -
v):
[✓] Flutter (Channel stable, X.Y.Z, on macOS X.Y 20F71
darwin-x64, locale en-US)
[✓] Android toolchain - develop for Android devices
(Android SDK version A.B.C)
[✓] Xcode - develop for iOS and macOS
[✓] Chrome - develop for the web
[✓] Android Studio (version M.N)
[✓] VS Code (version Q.R.S)
[✓] Connected device (3 available)

• No issues found!
$
```

flutter doctor


~
➜ ~ flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, v1.0.0, on Mac OS X 10.14.2 18C54, locale en-EE)
[✓] Android toolchain - develop for Android devices (Android SDK 28.0.3)
[!] iOS toolchain - develop for iOS devices (Xcode 10.0)
✗ libimobiledevice and ideviceinstaller are not installed. To install with Brew, run:
brew update
brew install --HEAD usbmuxd
brew link usbmuxd
brew install --HEAD libimobiledevice
brew install ideviceinstaller
✗ ios-deploy out of date (1.9.4 is required). To upgrade with Brew:
brew upgrade ios-deploy
[✓] Android Studio (version 3.2)
[!] IntelliJ IDEA Ultimate Edition (version 2018.1.5)
✗ Flutter plugin not installed; this adds Flutter specific functionality.
✗ Dart plugin not installed; this adds Dart specific functionality.
[✓] VS Code (version 1.30.2)
[✓] Connected device (1 available)
! Doctor found issues in 2 categories.
➜ ~

```
Sookie@SILENCE_HILL C:\Users\init_
> flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, v1.9.1+hotfix.4, on Microsoft Windows [Version 10.0.18362.418], locale zh-CN)
[√] Android toolchain - develop for Android devices (Android SDK version 29.0.2)
[√] Android Studio (version 3.5)
[!] IntelliJ IDEA Ultimate Edition (version 2016.3)
    X Flutter plugin not installed; this adds Flutter specific functionality.
    X Dart plugin not installed; this adds Dart specific functionality.
    X This install is older than the minimum recommended version of 2017.1.0.
[√] VS Code, 64-bit edition (version 1.37.1)
[!] Connected device
    ! No devices available

! Doctor found issues in 2 categories.
```

```
>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, v1.12.13+hotfix.8, on Microsoft Windows [Version 10.0.18363.720], locale
    zh-CN)
[!] Android toolchain - develop for Android devices (Android SDK version 29.0.3)
    X Android license status unknown.
      Try re-installing or updating your Android SDK Manager.
      See https://developer.android.com/studio/#downloads or visit https://flutter.dev/setup/#android-setup
      for detailed instructions.
[√] Android Studio (version 3.6)
[√] Connected device (1 available)

! Doctor found issues in 1 category.
```

```
!] Android Studio (not installed)
    • Android Studio not found; download from https://developer.android.com/studio/index.html
      (or visit https://flutter.io/setup/#android-setup for detailed instructions).

√] VS Code (version 1.30.2)
    • VS Code at C:\Users\annur.arya\AppData\Local\Programs\Microsoft VS Code
    • Flutter extension version 2.22.1

!] Connected device
    ! No devices available

 Doctor found issues in 3 categories.

:\Users\annur.arya>flutter config --android-sdk C:\Users\annur.arya\AppData\android-sdk
etting "android-sdk" value to "C:\Users\annur.arya\AppData\android-sdk".

:\Users\annur.arya>flutter doctor
octor summary (to see all details, run flutter doctor -v):
√] Flutter (Channel stable, v1.0.0, on Microsoft Windows [Version 10.0.17134.556], locale en-IN)
!] Android toolchain - develop for Android devices (Android SDK 23.0.3)
    X Android license status unknown.
!] Android Studio (not installed)
√] VS Code (version 1.30.2)
√] Connected device (1 available)

 Doctor found issues in 2 categories.
```

tl;dr

- Why is cross-platform development so difficult?
- What tools are needed for it?
- Where does the flutter SDK fit in?
- Is there a tool to help manage the complexity?

Development tools

IDEs and the dev tools you run from within them

Where do I write a Flutter app?

VS Code

Android Studio

¯_(ツ)_/¯

Anywhere

	Android Studio	VS Code	Everything else
From	JetBrains	Microsoft	Various
Can write	check	check	check
Can compile	check	check	no
Can run/debug	check	check	no
Best for	Former Android devs	Former web devs	No one

Android Studio

Installing Android Studio

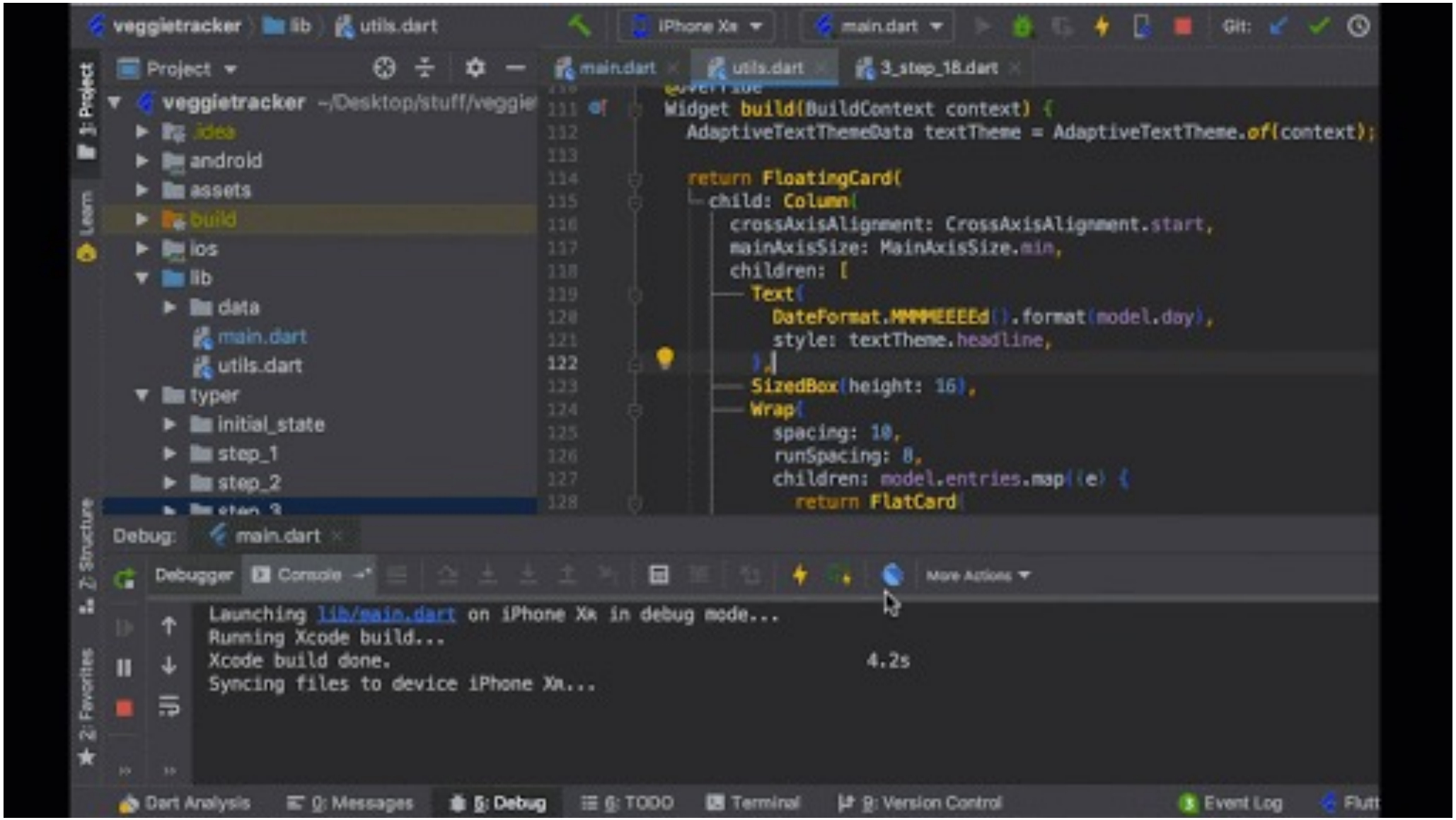
- https://developer.android.com/studio

VS Code

Installing VS Code

- https://code.visualstudio.com/

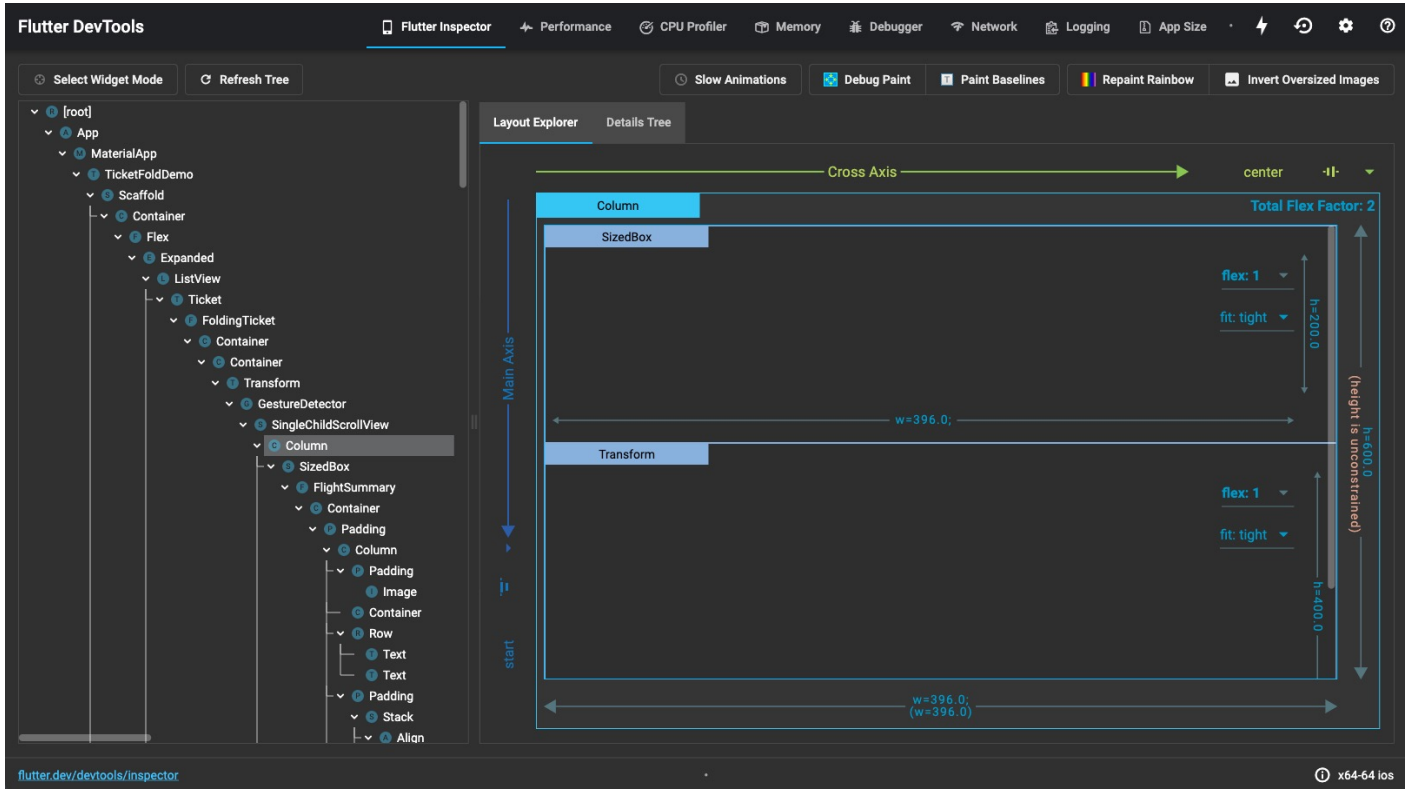
IDE Dev Tools


veggietracker
lib
utils.dart
iPhone Xs
main.dart
Git:
Project
veggietracker ~/Desktop/stuff/veggie
.idea
android
assets
build
ios
lib
data
main.dart
utils.dart
typer
initial_state
step_1
step_2
main.dart
utils.dart
3_step_18.dart
Widget build(BuildContext context) {
AdaptiveTextThemeData textTheme = AdaptiveTextTheme.of(context);
return FloatingCard(
child: Column(
crossAxisAlignment: CrossAxisAlignment.start,
mainAxisSize: MainAxisSize.min,
children: [
Text(
DateFormat.MMMMEEEEd().format(model.day),
style: textTheme.headline,
),
SizedBox(height: 16),
Wrap(
spacing: 10,
runSpacing: 8,
children: model.entries.map((e) {
return FlatCard
Debug:
main.dart
Debugger
Console
More Actions
Launching lib/main.dart on iPhone Xs in debug mode...
Running Xcode build...
Xcode build done.
4.2s
Syncing files to device iPhone Xs...
Dart Analysis
0: Messages
5: Debug
6: TODO
Terminal
9: Version Control
Event Log
Flutt
1: Project
Learn
7: Structure
2: Favorites

- Dart analyzer
- Widget inspector
- Memory profiler
- Timeline view
- logging view

widget inspector

- https://flutter.dev/docs/development/tools/devtools/inspector
- Can examine the widget tree and troubleshoot layout issues



tl;dr

- You have two IDE options, VS Code and Android Studio.
- Within those, there are some additional tools called the Dart dev tools.

Creating and running

How to create a Flutter app and then run it in the simplest way

tl;dr

- Create a flutter app from the command line with flutter create
- Run and debug from within the IDE
- Set breakpoints, step through, examine variables

To create a new project

```
flutter create cool_app
```

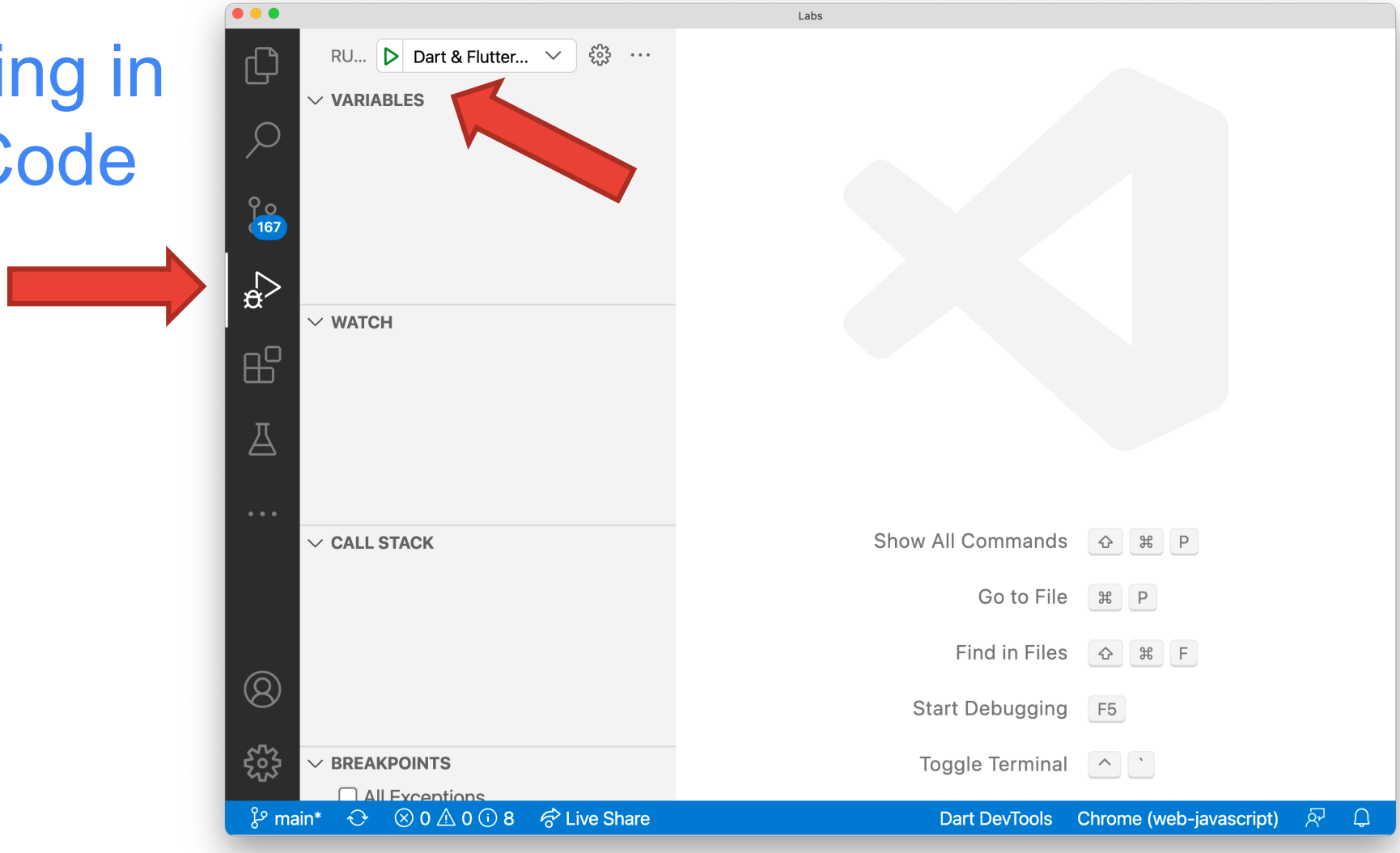
A typical Flutter project has ...

these folders	these files
• build - For compiled files • android - Android-specific config • ios - iOS-specific config • web - Web-specific config • test - Unit-testing files go here • lib - All of your source code	• README.md • .gitignore • pubspec.yaml • main.dart (under lib)

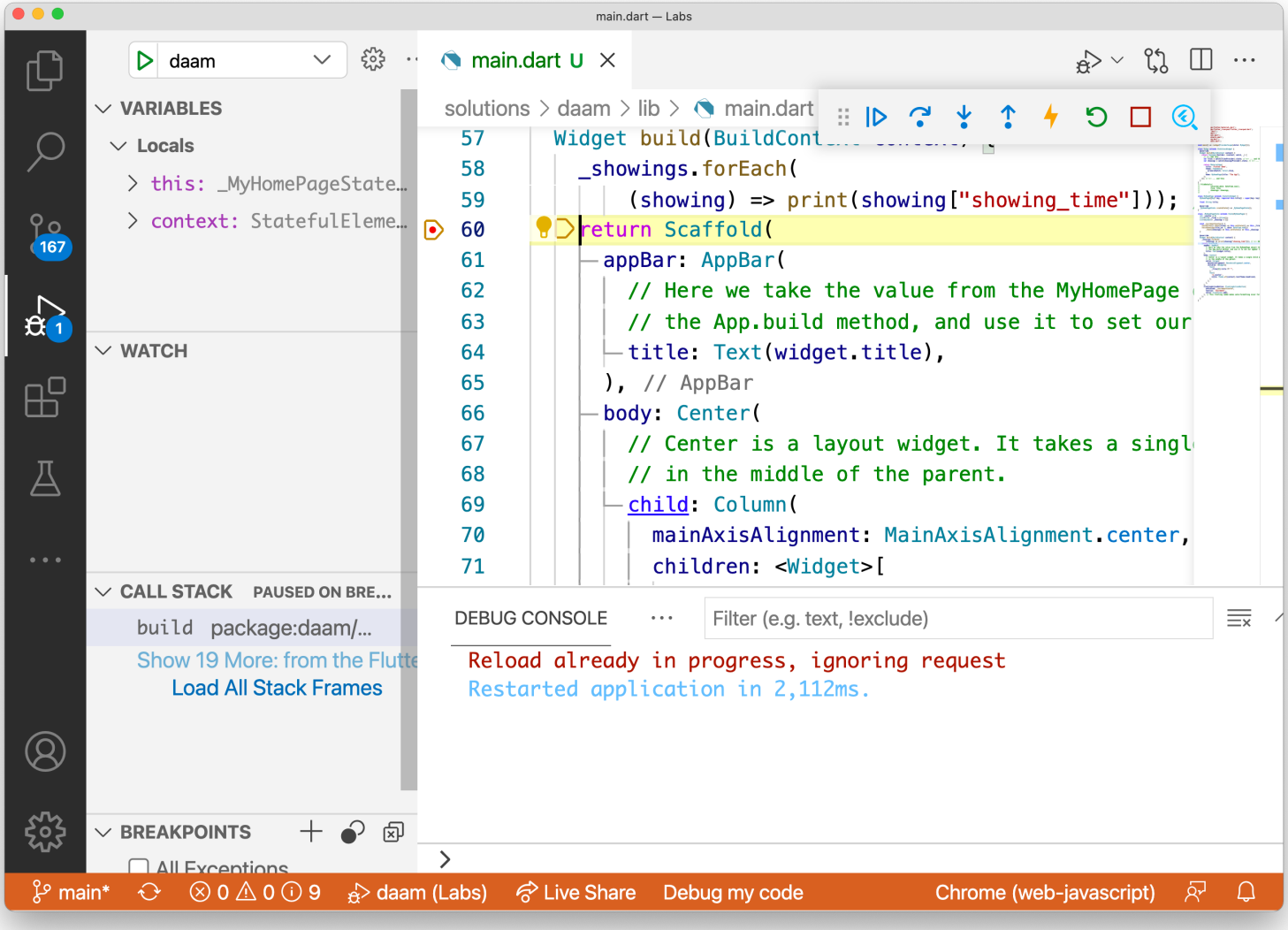
Running the app

```
flutter run
```

Running in VS Code



Debugging


main.dart — Labs
daam
main.dart U
VARIABLES
Locals
this: _MyHomePageState...
context: StatefulEleme...
WATCH
CALL STACK
PAUSED ON BRE...
build package:daam/...
Show 19 More: from the Flutte
Load All Stack Frames
BREAKPOINTS
All Exceptions
solutions > daam > lib > main.dart
57 Widget build(BuildCont
58 _showings.forEach(
59 (showing) => print(showing["showing_time"]));
60 return Scaffold(
61 appBar: AppBar(
62 // Here we take the value from the MyHomePage
63 // the App.build method, and use it to set our
64 title: Text(widget.title),
65), // AppBar
66 body: Center(
67 // Center is a layout widget. It takes a singl
68 // in the middle of the parent.
69 child: Column(
70 mainAxisAlignment: MainAxisAlignment.center,
71 children: <Widget>[
DEBUG CONSOLE
Filter (e.g. text, !exclude)
Reload already in progress, ignoring request
Restarted application in 2,112ms.
main*
0 0 9
daam (Labs)
Live Share
Debug my code
Chrome (web-javascript)

tl;dr

- Create a flutter app from the command line with flutter create
- Run and debug from within the IDE
- Set breakpoints, step through, examine variables

Running on a virtual device

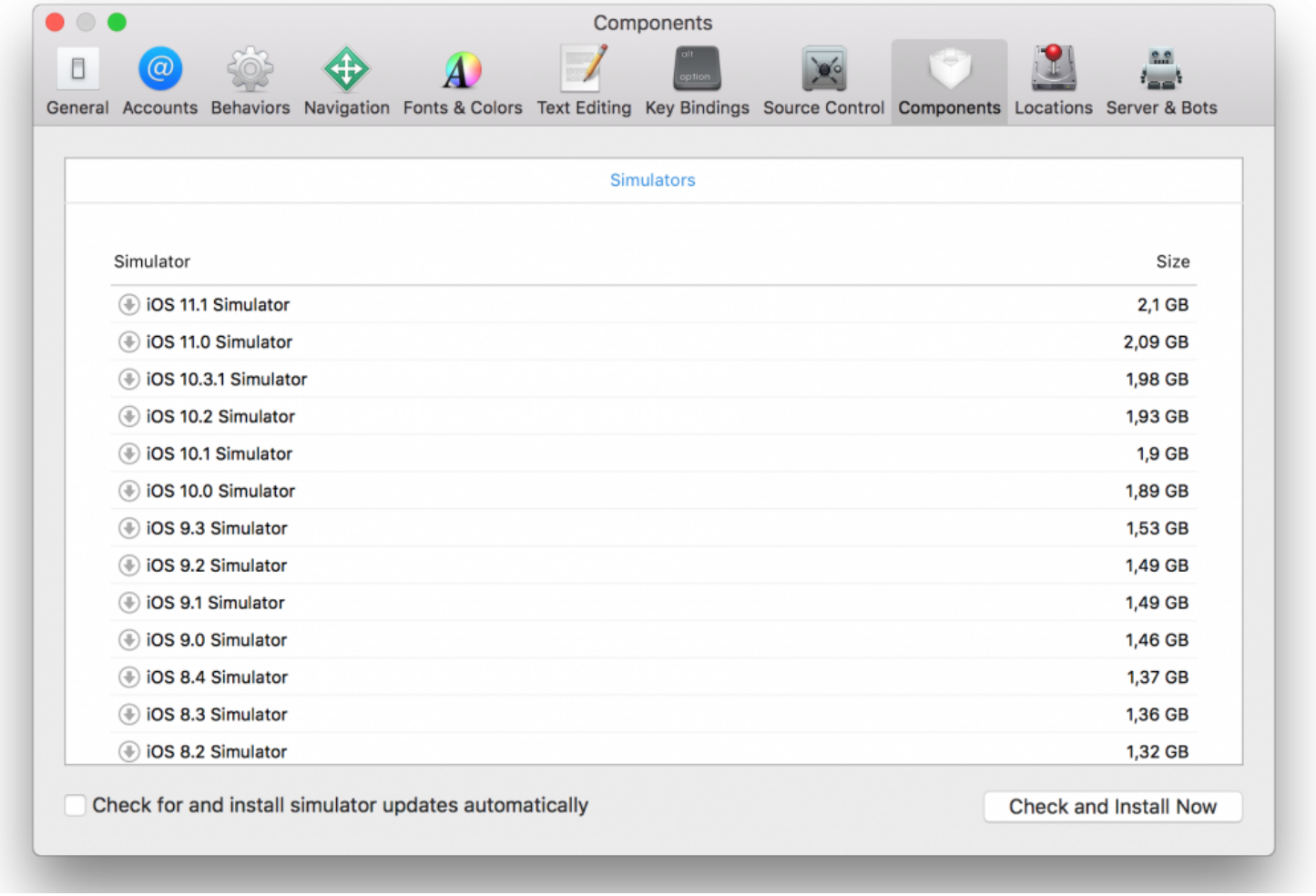
Debugging on the iOS simulator and the Android emulator

tl;dr

Steps to run on the iOS simulator

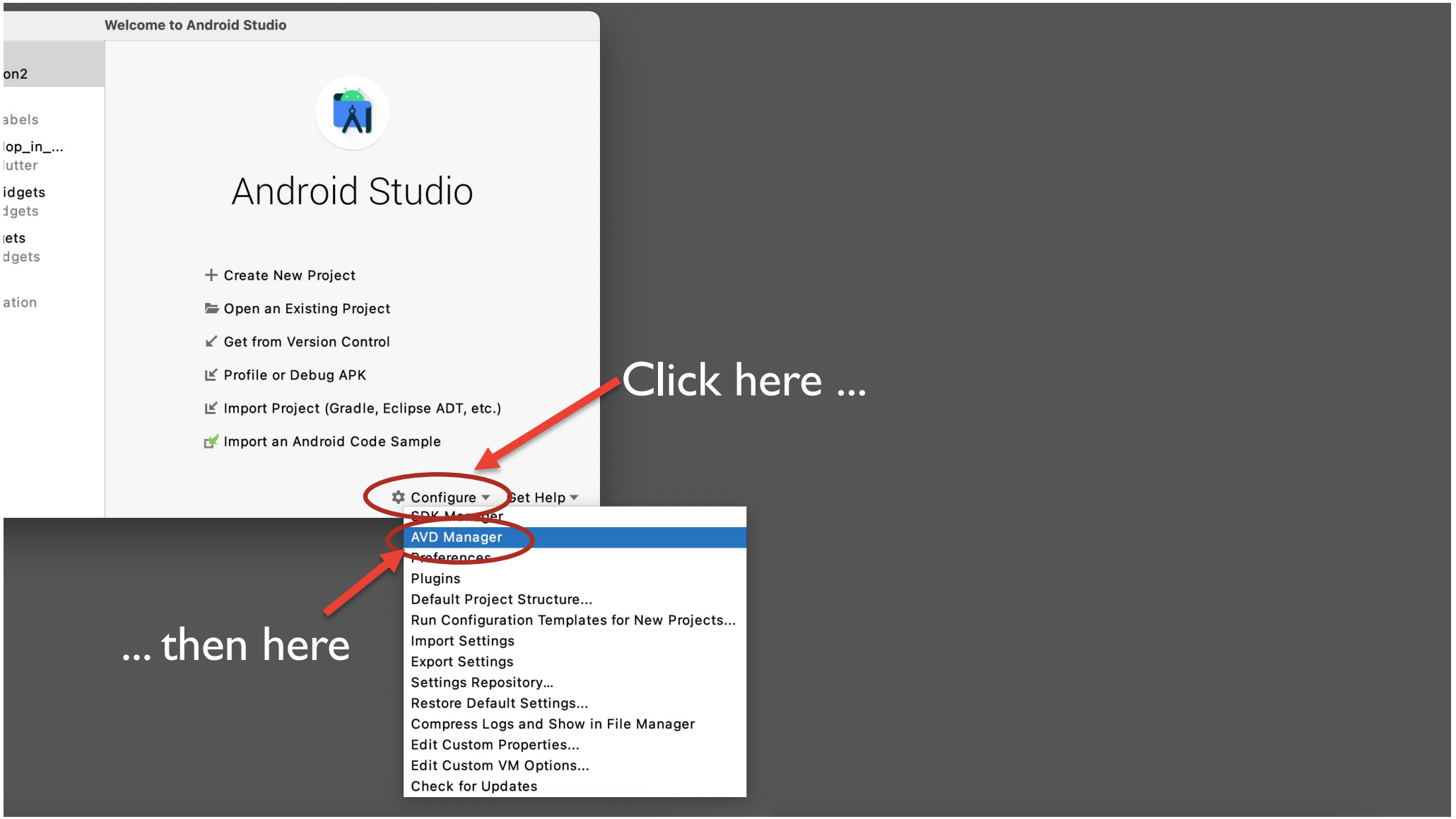
To install the iOS simulator ...

- Open Xcode
- Go preferences - components - simulators - Pick one.

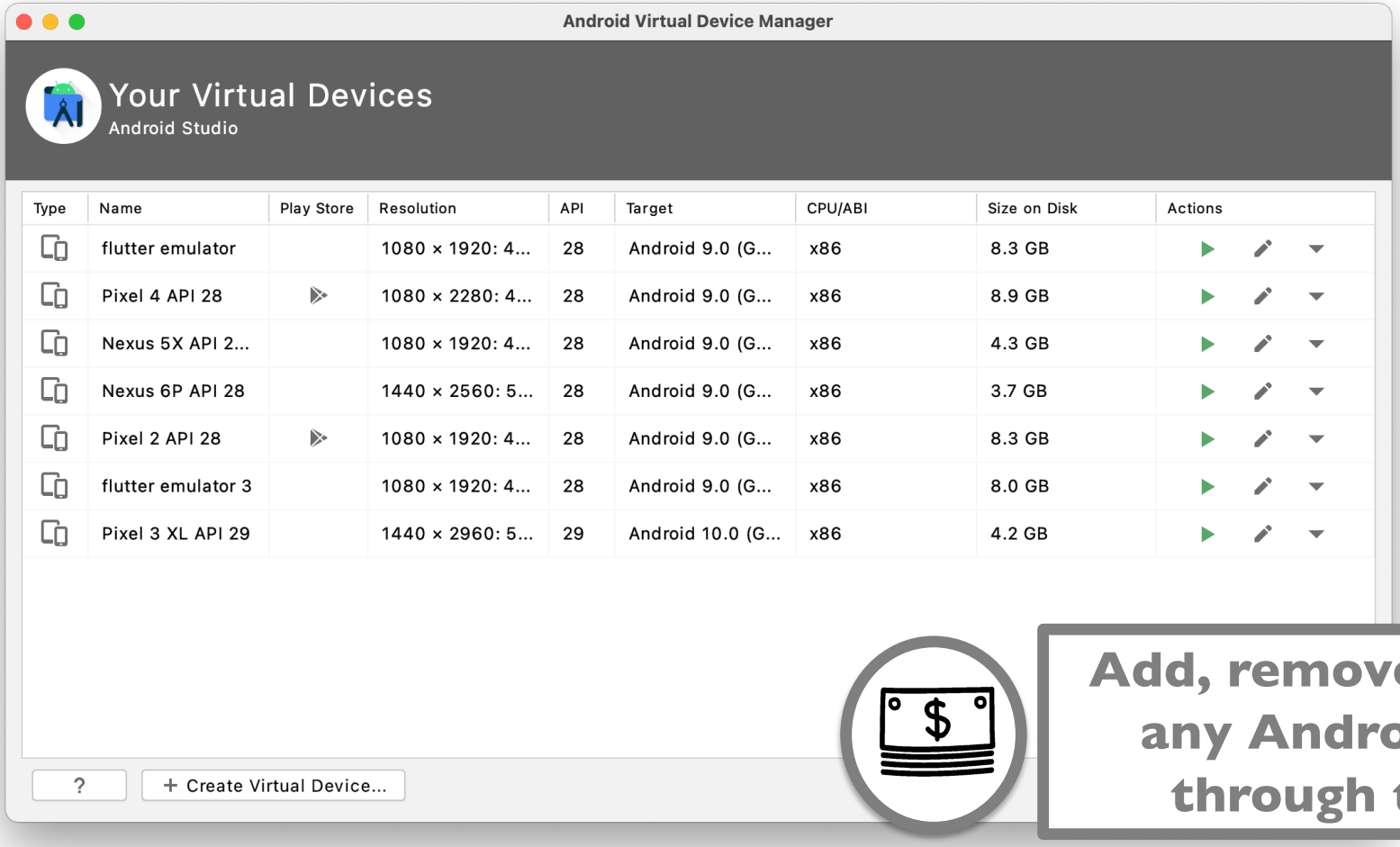


Steps to run on the android emulator


You mangage
Android emulators
in AVD Manager
android studio
Powered by the IntelliJ® Platform
... wh
is par
Andr
Stu

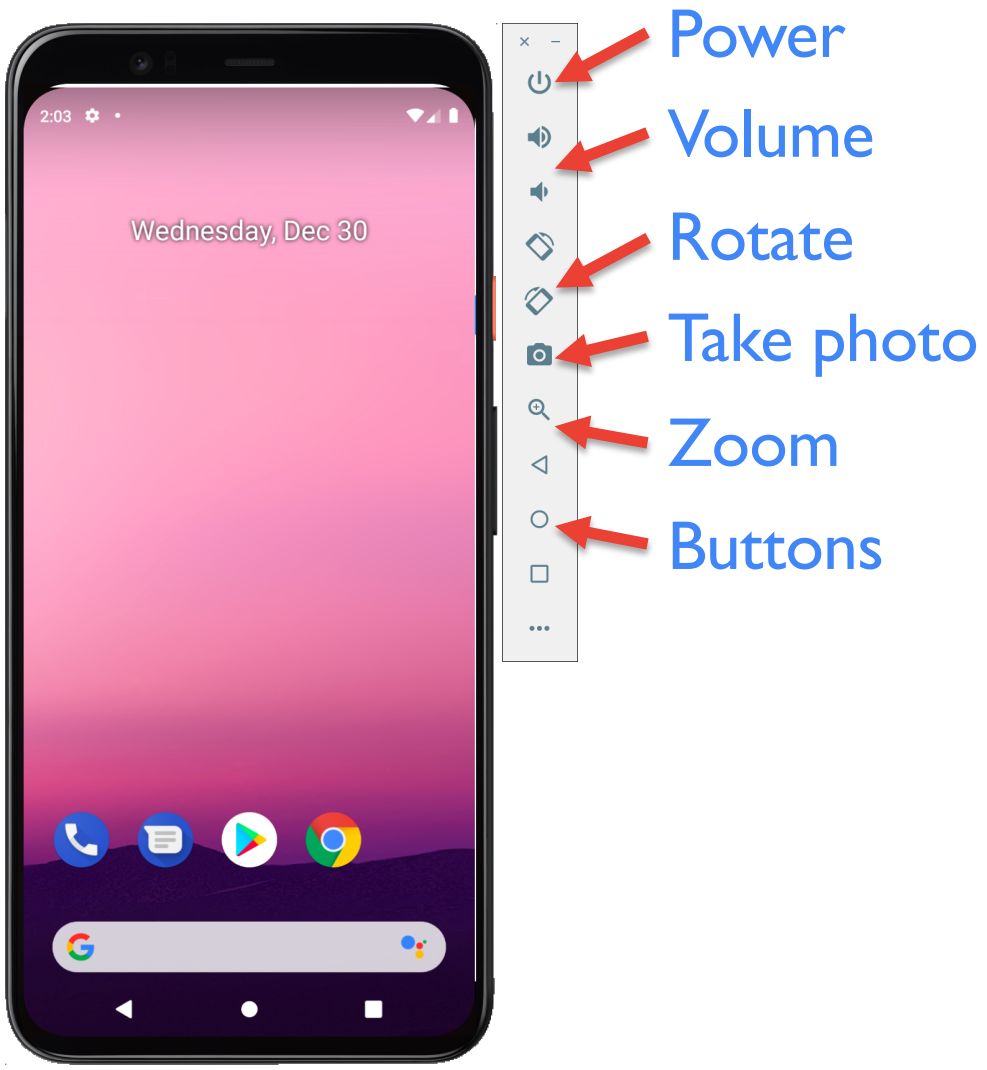

Welcome to Android Studio
Android Studio
Create New Project
Open an Existing Project
Get from Version Control
Profile or Debug APK
Import Project (Gradle, Eclipse ADT, etc.)
Import an Android Code Sample
Configure
Get Help
AVD Manager
Plugins
Default Project Structure...
Run Configuration Templates for New Projects...
Import Settings
Export Settings
Settings Repository...
Restore Default Settings...
Compress Logs and Show in File Manager
Edit Custom Properties...
Edit Custom VM Options...
Check for Updates
Click here ...
... then here

Pick a device and hit the play button



Add, remove or modify any Android device through this tool

- Once installed, open the emulator and then hit "a"
- Your app opens in the emulator



Debugging tools

List of debugging tools here

- Only those tools that run on the emulator.
- Overlays for layout
- Performance mode? Or something?

Expected things in Dart

The Dart language overview part 1

Dart overview

Dart is a compiled, statically-typed, object-oriented, procedural programming language. It has a very mainstream structure much like other OO languages, making it awfully easy to pick up for folks who have experience with Java, C#, C++, or other OO, C-like languages. And it adds some features that developers in those other languages would not expect but are very cool nonetheless and make the language more than elegant.

Dart has things you'd expect.

Dart is an easy, elegant language

Dart has features that are surprising.

Dart has the things that experienced devs expect

- Variables
- Arrays
- Conditional statements
- Looping
- Classes
- Class constructors

Disclaimer

We're making no attempt to teach you everything about Dart. Our goal here is to get you juuuust enough Dart to be effective as you write Flutter.

Expected things

DataTypes.dart

```
int x = 10;           // Integers
double y = 2.0;       // IEEE754 floating point
numbers
bool z = true;        // Booleans
String s = "hello"; // Strings

// Dynamic variables can change types
// at any time. Use sparingly!
dynamic d;

d = x;
d = y;
d = z;
```

ConvertingTypes.dart

```
// Converting strings to numbers
int x = int.tryParse('7');
double y = double.tryParse(x);

// Converting ints to doubles
double z = x.toDouble();

// Casting objects - the thing must be a
subtype
Shape thing = triangle1 as Shape
```

Arrays_or_lists.dart

```
// Square brackets means a list/array

// In Dart, arrays and lists are the same thing.
List<dynamic> list = [1, "two", 3];
// Dart supports Generics

// How to iterate a list
for (var d in list) {
  print(d);
}

// Another way to iterate a list
list.forEach((d) => print(d));
```

Conditionals.dart

```
// Traditional if/else statement
int x = 10;
if (x < 100) {
  print('Yes');
} else {
  print('No');
}

// Dart also supports ternaries
String response = (x < 100) ? 'Yes' : 'No';
```

Looping.dart

```
// A for loop
for (int i=1 ; i<10 ; i++) {
  print(i);
}

// A while loop
int i=0;
while(i<10) {
  print(i++);
}
```

Classes.dart

```
class Name {
  String first;
  String last;
  String suffix;
}

class Person {
  // Classes have properties
  int id;
  Name name;    // Another class can be used as a type
  String email;
  String phone;
  // Classes have methods
  void save() {
    // Write to a database somehow.
  }
}
```

ClassConstructors.dart

```
class Person {
  Name name;

  // Typical constructor
  Person() {
    this.name = new Name();
    this.name.first = "";
    this.name.last = "";
  }

}
```

ClassConstructor.dart

```
class Person {
  Name name;

  // Typical constructor
  Person(String first, String last) {
    this.name = new Name();
    this.name.first = first;
    this.name.last = last;
  }
}
```

Unexpected things in Dart

The Dart language overview part 2

tl;dr

- var, final, const
- Null safety
- String interpolation
- Array spread
- Maps (aka. hashes)
- Big arrow/lambdas
- Named parameters
- Omitting this and new
- Private properties
- Mixins
- Cascade operator
- Named constructors

Dart has things you'd expect.

Dart is an easy, elegant language

Dart has features that are surprising.

Dart has the things that experienced devs expect

- Variables
- Arrays
- Conditional statements
- Looping
- Classes
- Class constructors

But Dart has surprises!

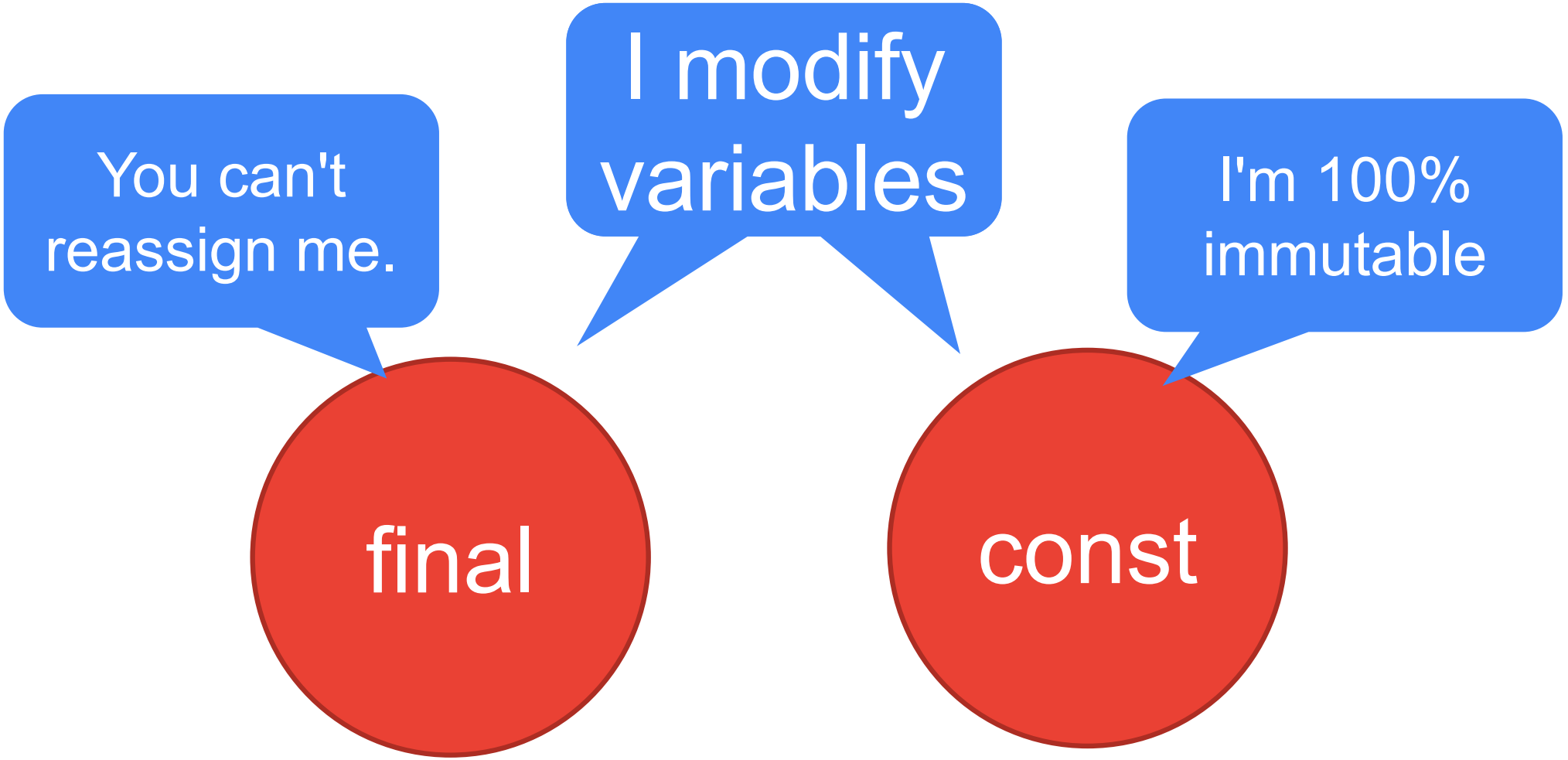
Nifty shortcuts and features that devs love

- private variables begin with "_"
- Functions are objects
- Anonymous functions and the fat arrow
- Named function parameters
- Omitting this and new
- Class constructor parameter shorthand

- There are many more like named constructors, type inference and Maps

Type_inference.dart

```
var i = 10;           // i is now defined as an int.

i = 12;               // Works, because 12 is an int.

i = "twelve";         // No! "twelve" is a String and

var str = "ten";      // str is now defined as a String.

str = "a million";    // Yep, works great.

str = 1000000.0;      // Nope! 1000000.0 is a double,
```


You can't reassign me.
I modify variables
I'm 100% immutable
final
const

const: the value is set at compile time. It is embedded in the installation bundle!

final: the variable can change but it can't be *reassigned*.

Foo.dart

```
final int x = 10;
const double y = 2.0;

final Employee e = Employee();
e.employer = "The Bluth Company";
// e changed, but it wasn't reassigned so that's okay.

// This, however, is not allowed:
const Employee e = Employee();
```

dynamic vs var vs final vs const

dynamic	Can store any datatype. The datatype can change at any time.
var	The datatype is inferred from the value on the right side of the "=". The datatype does not change.
final	The variable, once set cannot be reassigned.
const	The value is set at compile time, not runtime.

null safety

Dart has sound null safety

Sometimes a value cannot ever be null	Sometimes you want a value to be nullable
• A person's name • A passenger's ticket number • A circle's radius • A showing's film or theater	• A person's driver license # • A customer's frequent shopper number • A person's favorite food

```
String name;
double radius;
Film film;
```

```
String? name;
double? radius;
Film? film;
```

- Null coalescing

```
int x = tryParse(someString) ?? 0;
```

- Null propagation

```
var foo = person?.address?.street?.name;
```

Classes with properties

- To make properties nullable, use "?"
- You must initialize required props in all constructors
- That, or mark them as required

Foo.dart

```
class Foo {
  Foo():id=0{}
  int id;
  String? title;
}
```

Foo.dart

```
class Foo {
 Foo({required this.id});
 int id;
 String? title;
}
```

String interpolation with $

- Interpolation saves devs from writing string concatenations. This:

```
String fullName = '$first $last, $suffix';
```

- Is effectively the same thing as this:

```
String fullName = first + " " + last + ", " + suffix;
```

- When the variable is part of a map or an object, the compiler can get confused, so you should wrap the interpolation in curly braces.

```
String fullName = '${name['first']} ${name['last']}';
```

Multiline strings

- You can create multiline strings with three single- or double-quotes.

Multiline.dart

```
String introduction = """
Now the story of a wealthy family
who lost everything
And the one son who had no choice
but to keep them all together.
""";
```

Spread operator

- The "..." operator will spread out the elements of an array, flattening them. This will be very familiar to JavaScript developers.

Spread.dart

```
List fiveTo10 = [ 5, 6, 7, 8, 9, 10, ];
// Spreading the inner array with "...":
List numbers = [ 1, 2, 3, 4,
                 ...fiveTo10, 11, 12];
// numbers now has [1, 2, 3, 4, 5, 6, 7, 8, 9,
// 10, 11, 12]
```

Map<foo, bar>

- Maps are like a hash or dictionary. They're merely an object with a set of key-value pairs. The keys and values can be of any type.

Spread.dart

```
// You set the value of a Map with curly braces:
Map<String, dynamic> person = {
  "first": "George",
  "last": "Bluth",
  "dob": DateTime.parse("1972-07-16"),
  "email": "amazingGob@gmail.com",
};

// You reference a map member with square brackets:
print(person['first'] + " was born "+ person['dob'].toString());
```

Generics on a Map set the data types. Not required but are a good practice

Functions are objects

Multiline.dart

```
var sayHi=(name)=>print('Hello, $name');


// You can pass sayHi around like data;
// it's an object!
Function meToo = sayHi;
meToo("Tobias");
```

Big arrow/Fat arrow

These are all the same.

Multiline.dart

```
int triple(int val) {
  return val * 3;
}
Function triple = (int val) {
  return val * 3;
};
Function triple = (int val) => val * 3;
```

Positional parameters are great but it can be less error-prone (albeit more typing) to have **named parameters**.

Foo.dart

```
// Instead of calling a function like this:
sendEmail('ceo@bluthcompany.com',
  'Popcorn in the breakroom');

// You can call it like this:
sendEmail(subject:'Popcorn in the breakroom',
  toAddress:'ceo@bluthcompany.com');
// Now the order of parameters is unimportant.
```

To create named function parameters

Use curly braces around the parameters.

Foo.dart

```
void sendEmail({String toAddress,
                String subject}) {
  // send the email here
}
```

Named function parameters in constructors

Named parameters also work great with class constructors where they are very commonly used in Flutter.

Person.dart

```
class Person {
 Name name;
 // Named parameters
 Person({String firstName, String lastName}) {
  name = Name()..first=firstName..last=lastName;
 }
}
```

Omitting "new"

In Dart, it is possible -- and encouraged -- to avoid the use of the *new* keyword when instantiating a class.

Foo.dart

```
// No. Avoid.
Person p = new Person();
// Yes
Person p = Person();
```

Name.dart

```
class Name {
 String first;
 String last;
 String suffix;
 String getFullName() {
  // No. Avoid "this.":
  String full = this.first + " " + this.last +
    ", " + this.suffix;
  // Better.
  String full = first + " " + last + ", "+suffix;
  return full;
  }
}
```

Inside of a class, the use of "this." is also discouraged.

When constructors receive "this.something" and have a class-scoped property with the same name, the compiler writes the assignments so you don't have to.

Foo.dart

```
class Person {
 String email;
 String phone;
 Person(this.email,
        this.phone) {}
}
```

Foo.dart

```
class Person {
 String email;
 String phone;
 Person(String email,
        String phone) {
  this.email = email;
  this.phone = phone;
 }
}
```

Person.dart

```
class Person {
  int id;
  String email;
  String phone;
  String _password;

  set password(String value) {
    _password = value;
  }
  String get hashedPassword {
    return sha512
     .convert(utf8.encode(_password)).toString();
  }
}
```

To make a class member private, put an underscore in front of the name.

Private class members

Mixins are baskets of methods and properties

Can be added to any class but can't be instantiated

Employment.dart

```
mixin Employment {
  String employer;
  String businessPhone;
  void callBoss() {
    print('Calling my boss');
  }
}
```

A mixin is added to a class when it uses the "with" keyword

Employee.dart

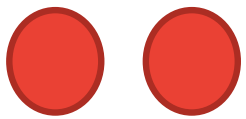
```
class Employee extends Person with Employment {
  String position;
}
```

This Employee class now has employer and businessPhone properties and a callBoss() method.

main.dart

```
Employee e = Employee();
e.employer = "The Bluth Company";
e.callBoss();   // An employee can call its boss.
```

The cascade operator (..)



Return this object but before you do, do something with a property or method.

Foo.dart

```
Person p = Person()..id=100
  ..email='gob@bluth.com'..save();
```

These two are equivalent

Foo.dart

```
Person p = Person();
p.id=100;
p.email='gob@bluth.com';
p.save();
```

No overloading

Dart does not support overloading methods. This includes constructors.

Named constructors

Write a typical constructor but tack on a dot and another name.

Foo.dart

```
Person p = Person();
Person p = Person.withName(
             firstName:"Lindsay",
             lastName:"Fünke");
Person p = Person.byId(100);
```

Person.dart

```
class Person {
 // Typical constructor
 Person() {
  name = Name()..first=""..last="";
 }
 // A named constructor
 Person.withName({String firstName, String lastName}) {
  name = Name()..first = firstName..last = lastName;
 }
 // Another named constructor
 Person.byId(int id) {
  // Maybe go fetch from a service by the provided id
 }
}
```

tl;dr

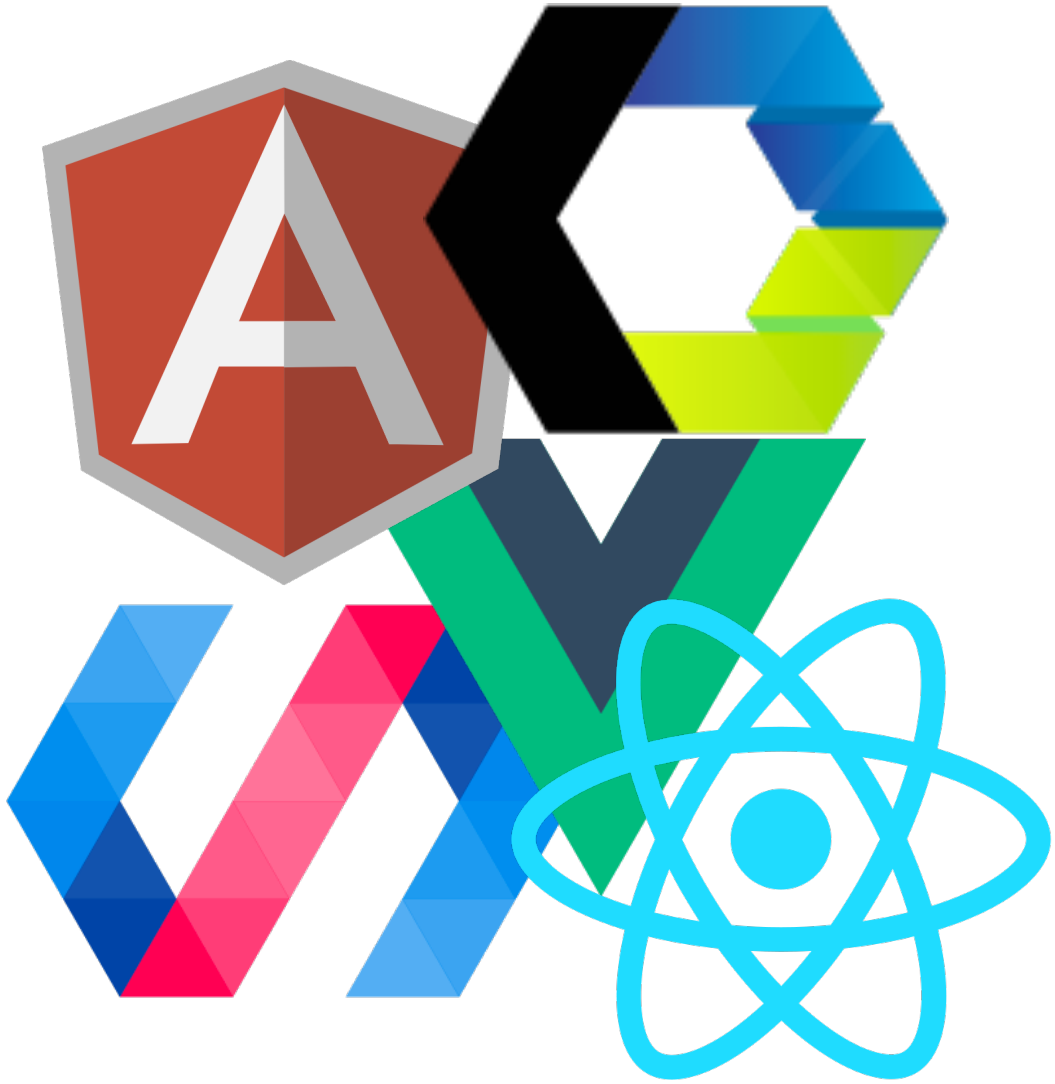
- var, final, const
- Null safety
- String interpolation
- Array spread
- Maps (aka. hashes)
- Big arrow/lambdas
- Named parameters
- Omitting this and new
- Private properties
- Mixins
- Cascade operator
- Named constructors

"Everything is widgets"

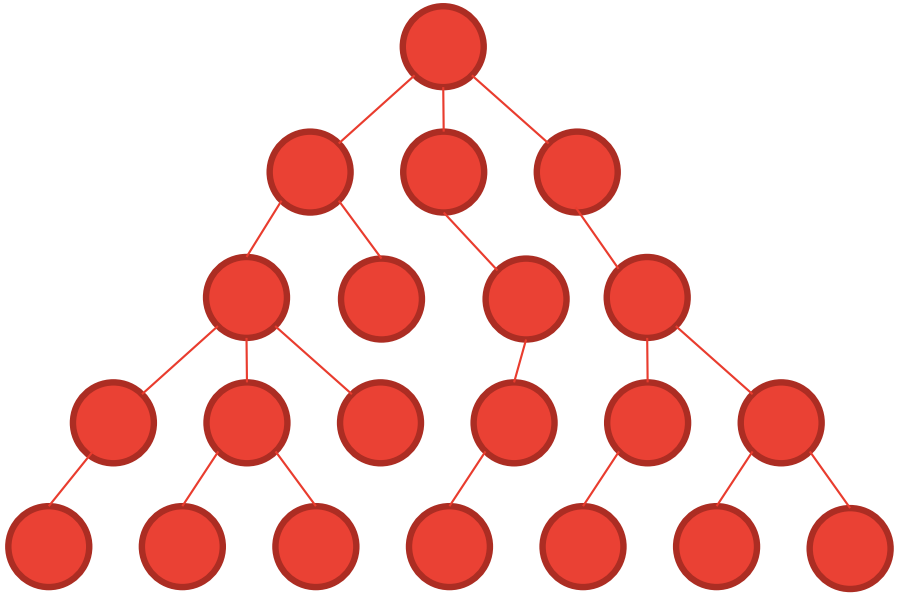
-- The Flutter Team

The web has gone to components

Nearly 100% of modern web app development is done using components



Everything is Widgets



There's 4 types of widgets

	Built-in to Flutter	Built by you
Stateless	Text, Image, Buttons	Easier to write
Stateful	TextField, Radio, Slider, Switch	More capable

- Simpler for devs to read and write (slightly)
- More efficient to render (slightly)

Stateless widgets

Yo, nothing in me can change so don't waste time keeping track of the data in me or any of my descendents.

- More capable

Stateful widgets

Umm, Displayed data inside me can change when I invoke setState() so be ready to re-draw me and all of my descendents.

UI as Code

Other development frameworks have proven componentization to be the way to go. The Flutter team has openly stated that they were heavily inspired by React which is based on componentization. In fact, all framework makers seem to borrow heavily from one another. But Flutter is unique in the way that the user interface is expressed. We developers use the same Dart language to express our app's graphical user interface as well as the behavior. We call this "UI as code".

Framework	Behavior expressed in ...	UI expressed in ...
Xamarin	C#	XAML
React Native	JavaScript	JSX
NativeScript	JavaScript	XML
Flutter	Dart	Dart

So how does this UI get created?

- Like many other frameworks and languages, A flutter app starts with a *main* function.
- In Flutter, main will call a function called runApp(). This runApp() receives one widget; the root widget which can be named anything, but it should be a class that extends a Flutter StatelessWidget. It looks like this:

main.dart

```
// import the Dart package needed for all Flutter apps
import 'package:flutter/material.dart';

// Here is main calling runApp
void main() => runApp(RootWidget());

// And here is your root widget
class RootWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text("Hello world");
  }
}
```

- And that's all you need to create a "Hello world" in Flutter.
- But wait ... what is this Text() thing? It's a built-in Flutter widget. Since these built-in widgets are so important, we need to take a look at them.

Built-in Flutter widgets

- Flutter's foundational widgets are the building blocks of everything we create and there are tons of them-- about 160 at last count. This is a lot of widgets for you and I to keep track. But if you mentally organize them, it becomes much more manageable.
- They fall into these major categories:
- Value widgets
- Layout widgets
- Navigation widgets
- Other widgets
- You can find a list of them all here: https://flutter.dev/docs/reference/widgets

Value widgets

- Certain widgets hold a value, maybe values that came from local storage, a service on the Internet, or from the user themselves. These are used to display values to the user and to get values from the user into the app. The seminal example is the Text widget which displays a little bit of text. Another is the Image widget which displays a .jpg, .png, or another picture.

- Checkbox
- CircularProgressIndicator
- Date & Time Pickers
- DataTable
- DropdownButton
- FlatButton
- FloatingActionButton
- FlutterLogo
- Form
- FormField
- Icon
- IconButton
- Image
- LinearProgressIndicator
- PopupMenuButton
- Radio
- RaisedButon
- RawImage
- RefreshIndicator
- RichText
- Slider
- Switch
- Text
- TextField
- Tooltip
-

Layout widgets

- Layout widgets give us tons of control in making our scene lay out properly -- placing widgets side-by-side or above-and-beneath, making them scrollable, making them wrap, determining the space around widgets so they don't feel crowded, and so on.
-

- Align
- AppBar
- AspectRatio
- Baseline
- BottomSheet
- ButtonBar
- Card
- Center
- Column
- ConstrainedBox
- Container
- CustomMultiChildLayout
- Divider
- Expanded
- ExpansionPanel
- FittedBox
- Flow
- FractionallySizedBox
- GridView
- IndexedStack
- IntrinsicHeight
- IntrinsicWidth
- LayoutBuilder
- LimitedBox
- ListBody
- ListTile
- ListView
- MediaQuery
- NestedScrollview
- OverflowBox
- Padding
- PageView
- Placeholder
- Row
- Scaffold
- Scrollable
- Scrollbar
- SingleChildScrollView
- SizedBox
- SizedOverflowBox
- SliverAppBar
- SnackBar
- Stack
- Table
- Wrap

Navigation widgets

- When you app has multiple scenes ("screens", "pages", whatever you want to call them), you'll need some way to move between them. That's where Navigation widgets come in. These will control how your user sees one scene and then moves to the next. Usually this is done when the user taps a button. And sometimes the navigation button is located on a tab bar or in a drawer that slides in from the left side of the screen. Here are some navigation widgets.

- AlertDialog
- BottomNavigationBar
- Drawer
- MaterialApp
- Navigator
- SimpleDialog
- TabBar
- TabBarView

Other widgets

- And no, not all widgets fall into these neat categories. Let's lump the rest into a miscellaneous category. Here are some miscellaneous widgets:
- GestureDector
- Dismissible
- Cupertino
- Theme
- Transitions
- Transforms

How to create your own stateless widgets

- So we know that we will be composing these built-in widgets to form our own custom widgets which will then be composed with other built-in widgets to eventually form an app.
- Widgets are masterfully designed because each widget is easy to understand and therefore easy to maintain. Widgets are abstract from the outside while being logical and predictable on the inside. They are a dream to work with.

- Every widget is a class that can have properties and methods. Every widget can have a constructor with zero or more parameters. And most importantly, every widget has a build method which receives a BuildContext and returns a single Flutter Widget. If you're ever wondering how a widget got to look the way it does, locate its build method.
- Don't get distracted by the BuildContext. It's used by the framework and we do occasionaly refer to it but we'll save those examples for later in the book. For now, just think of it as part of the recipe to write a custom widget.

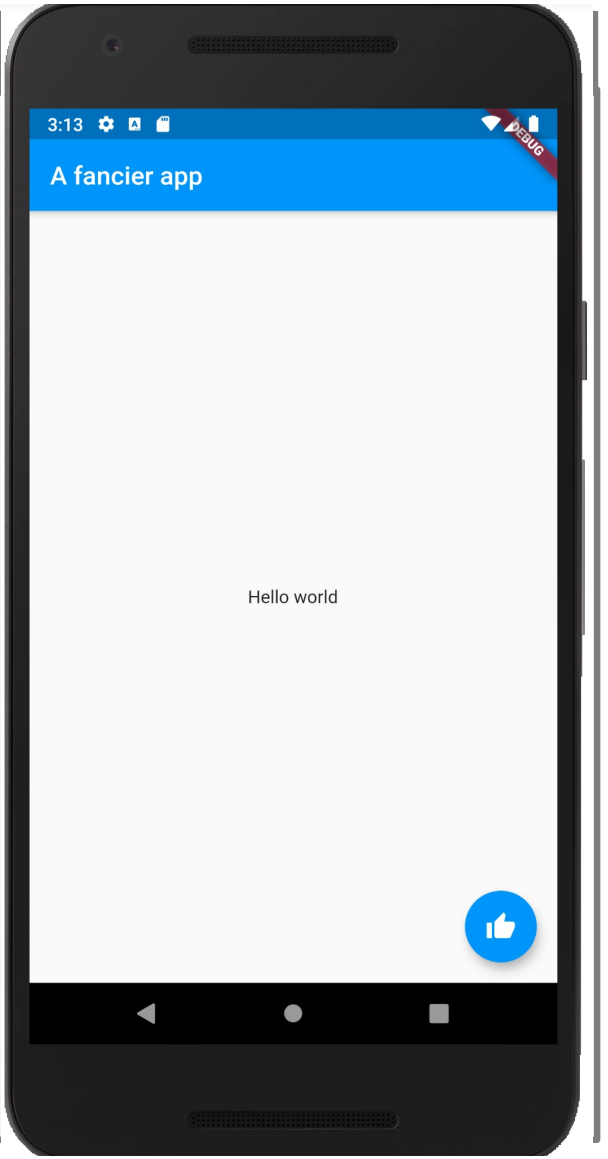
Foo.dart

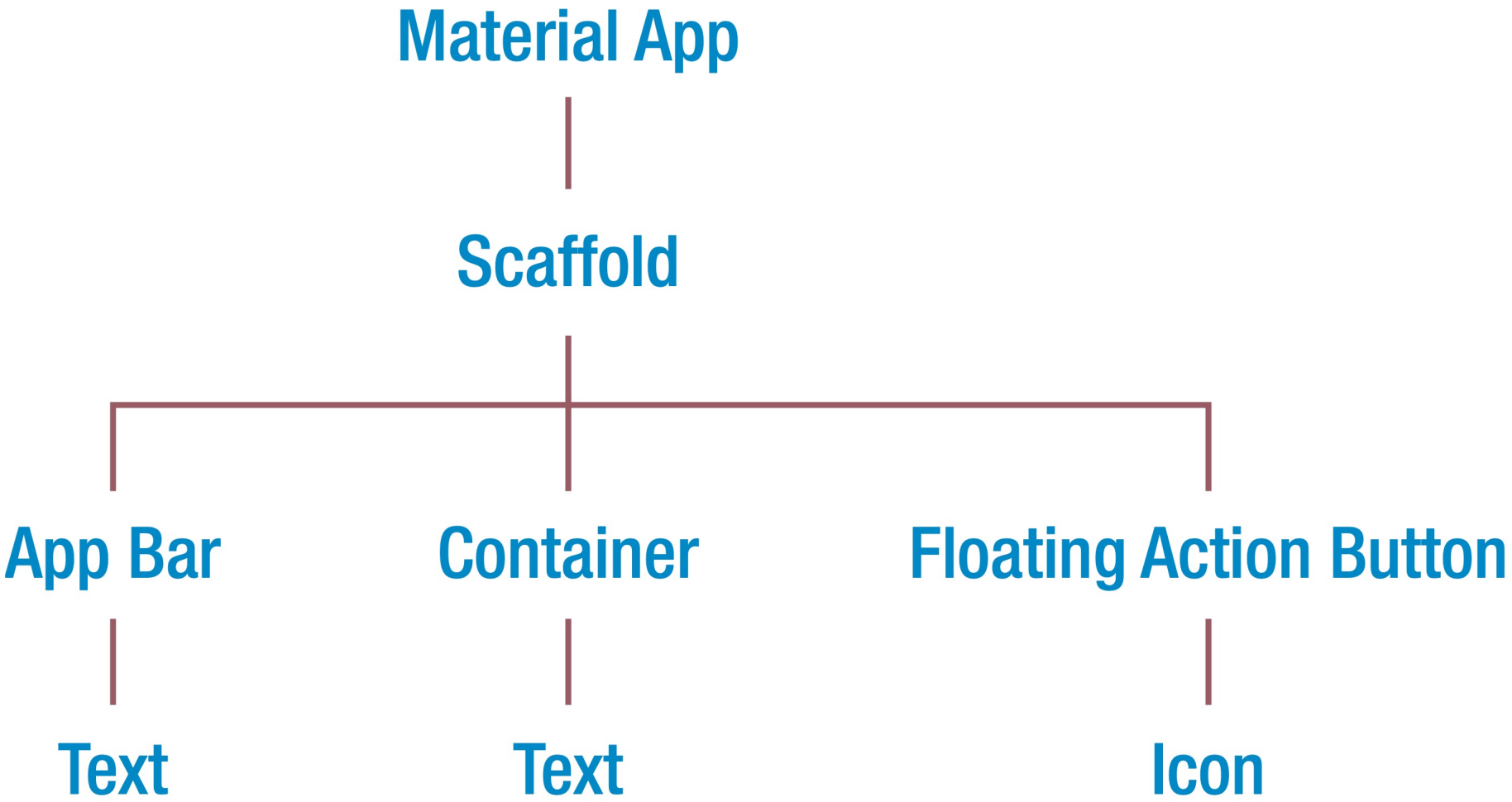
```
class Foo extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Text('Hello world');
  }
}
```

- In this hello world example which we repeated from earlier in the chapter, we're displaying a Text widget. A single inner widget works but real-world apps will be a whole lot more complex. The root widget could be composed of many other sub-widgets.

Foo.dart

```
class FancyHelloWidget extends StatelessWidget {

  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("A fancier app"),
        ),
        body: Container(
          alignment: Alignment.center,
          child: Text("Hello world"),
        ),
        floatingActionButton: FloatingActionButton(
          child: Icon(Icons.thumb_up),
          onPressed: () => {},
        ),
      ),
    );
  }
}
```




Material App
Scaffold
App Bar
Container
Floating Action Button
Text
Text
Icon

Passing a value into your widget

$$y = f(x)$$

$$Scene = f(Data)$$

- Now how might that data change? There's two ways.
- The widget can be re-rendered with new data passed from outside
- Data can be maintained within certain widgets

Let's talk about the first. To pass data into a widget, you'll send it in as a constructor parameter like this:

```
Widget build(BuildContext context) {
  return Person("Sarah");
}
```

Foo.dart

```
class Person extends StatelessWidget {
  final String firstName;
  Person(this.firstName) {}

  Widget build(BuildContext context) {
    return Text(firstName);
  }

}
```

Quiz! How would I write the class to handle two named parameters?

Foo.dart

```
Widget build(BuildContext context) {

  return Person(firstName:"Sarah", lastName:"Ali");

}
```

Person.dart

```
class Person extends StatelessWidget {
  final String firstName;
  final String lastName;
  Person({this.firstName,
          this.lastName}) {}

  Widget build(BuildContext context) {
    return Container(child:
          Text('$firstName $lastName'));
  }
}
```

Stateless and Stateful widgets

Person.dart

```
class Person extends StatefulWidget {
 final String firstName;
 final String lastName;
 Person({this.firstName, this.lastName}) {}
 @override
 _PersonState createState() => _PersonState();
}

// The file is continued below ...
```

Person.dart

```
// ... file continued from above.

class _PersonState extends State<Person> {
 @override
 Widget build(BuildContext context) {
  return Column(
   children: [
    Text(widget.firstName),
    Text(widget.lastName),
    FlatButton(child: Text("Go"),
     onPressed: () => setState(()=>widget.firstName="Jo"),
   ],
  );
 }
}
```

Three types of widgets

Stateful

Stateless

Inherited

So which one should I create?

- The short answer is create a stateless widget. Never use a stateful widget until you must. Assume all widgets you make will be stateless and start them out that way. Refactor them into stateful widgets when you're sure you really do need state. But recognize that state can be avoided more often than developers think. Avoid it when you can to make widgets simpler and therefore easier to write, to maintain, and to extend. Your team members will thank you for it.

Value widgets

Widgets that present data

tl;dr

- Text() widgets display strings, styled or not.
- Flutter provides 100s of icons that can be displayed in Icon() widgets
- Image() widgets show two kinds of pictures -- preloaded ones and ones downloaded from the Internet live.

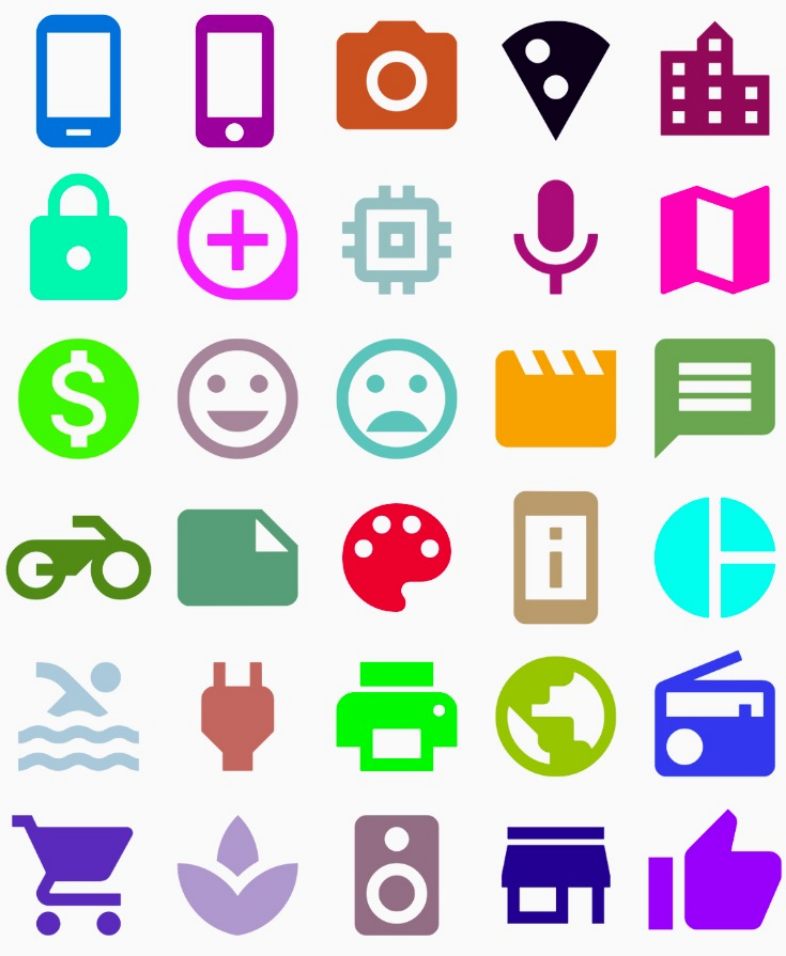
The Text() widget displays a string to the screen.

```
Text('Hello world'),
```

If your Text is a literal, put *const* in front of it and the widget will run faster on the device.

The Icon widget

- Flutter has built-in icons.
- Over 7,000!
- A full list list is here: https://api.flutter.dev/flutter/material/Icons-class.html
- Get 1,000s more here: https://github.com/flutter-studio/flutter-icons



Here's how you'd put a big red birthday cake on your app.

Foo.dart

```
Icon(
  Icons.cake,
  color: Colors.red,
  size: 200,
)
```

Images

Getting images into your app

Two ways to present images

Embedded images

Static images
Faster
Larger compiled file

Network images

Runtime images
Slower!
Smaller compiled file

To embedded images

1. Put the image file in your project folder
2. Add it to pubspec.yaml
3. Run flutter pub get
4. Use Image.asset()

pubspec.yaml

```
flutter:
  ...
  assets:
   - assets/images/photo1.png
   - assets/images/photo2.jpg
```

The pubspec.yaml file holds all kinds of great information about your project.

Project metadata

- name
- description
- author(s)
- license
- repository location
- version number

- Dependencies
- libraries
- dev tools
- fonts
- Images.

It is the go-to location for other developers new to your project. For any of you JavaScript developers, it is the package.json file of your Dart project.

- Save the file and run "flutter pub get" from the command line to have your project process the file.

Foo.dart

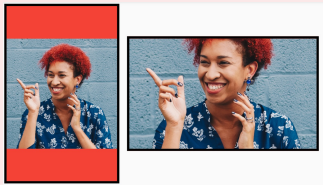
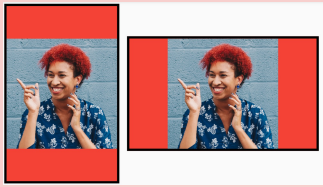
```
Image.asset('assets/images/photo1.jpg',),
```

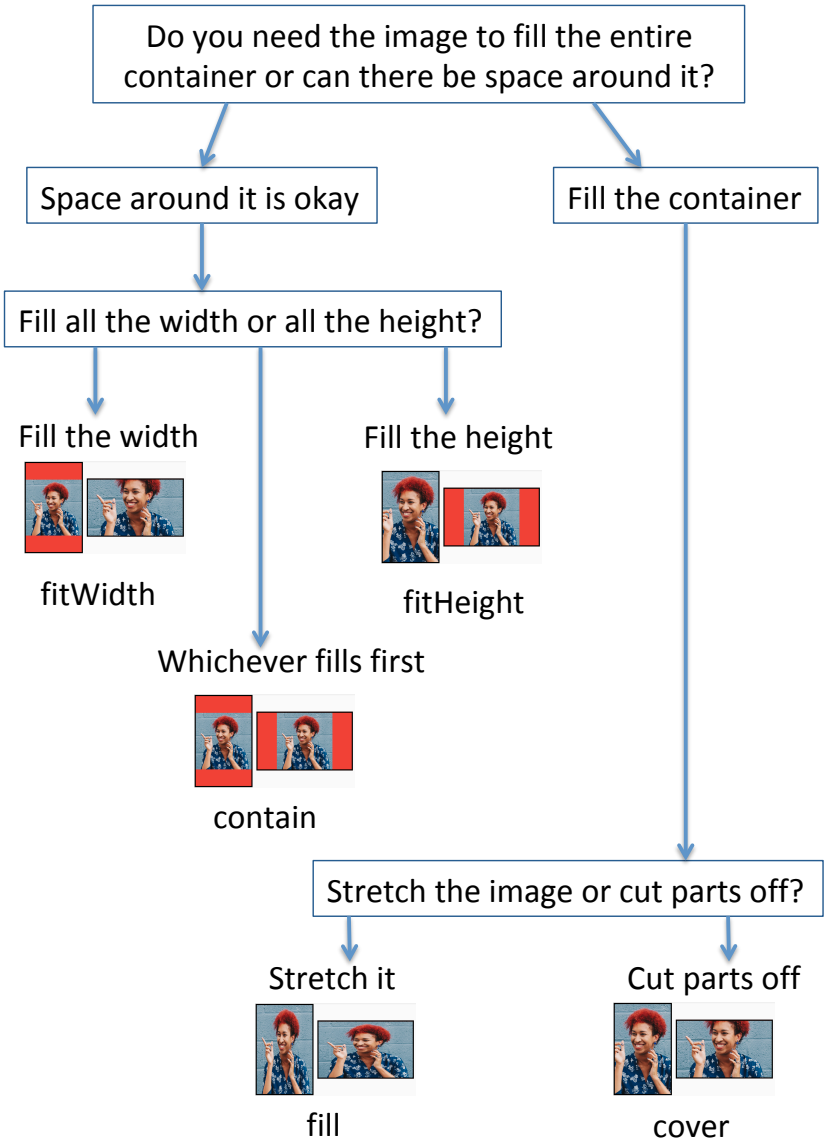
Network images

- Like what web devs do.
- Fetching an image over the Internet via HTTP.
- You'll use the network constructor and pass in a URL as a string.

```
Image.network(imageUrl),
```

BoxFit options

fill	Stretch it so that both the width and the height fit exactly. Distorts the image.	
cover	Shrink or grow until the space is filled. The top/bottom or sides will be clipped.	
fitHeight	Make the height fit exactly. Clip the width or add extra space as needed.	
fitWidth	Make the width fit. Clip the height or add extra space as needed.	
contain	Shrink until both the height and the width fit. There will be extra space on the top/bottom or sides.	


Do you need the image to fill the entire container or can there be space around it?
Space around it is okay
Fill the container
Fill all the width or all the height?
Fill the width
fitWidth
Fill the height
fitHeight
Whichever fills first
contain
Stretch the image or cut parts off?
Stretch it
fill
Cut parts off
cover

Foo.dart

```
class Foo extends StatefulWidget {
  ...
  // To specify the fit, you'll set
  // the fit property.
  Image.asset('assets/img/woman.jpg',
              fit: BoxFit.contain,),
}
```

tl;dr

- Text() widgets display strings, styled or not.
- Flutter provides 100s of icons that can be displayed in Icon() widgets
- Image() widgets show two kinds of pictures -- preloaded ones and ones downloaded from the Internet live.

User inputs

How to get user input into your apps

tl;dr

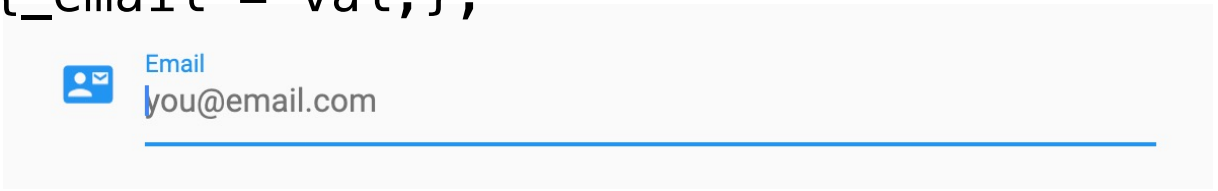
- Form fields ...
  - Textfields
  - DropdownButtons
  - CheckBoxes
  - Radios
  - Switches
  - Sliders

Text fields

The TextField() widget

```
 TextField(
  onChanged: (String val) {_ email = val;}
 )
```


```
TextField(
 decoration: InputDecoration(
  labelText: 'Email',
  hintText: 'you@email.com',
  icon: Icon(Icons.contact_mail),
 ),
 onChanged: (String val) {_email = val;},
)
```



```
TextField(
  onChanged: (String val) => _searchTerm = val,
),
```

- The onChanged event handler fires after every keystroke.
- It receives a single value -- the value that the user is typing.

Controller

To provide an initial value with a TextField, you need the unnecessarily complex TextInputController.

```
var _controller = TextEditingController(
                    text: "Some initial value");
TextField(
  controller: _controller,
  onChanged: (String val) => _searchTerm = val,
),
```

You can also use that _controller.text property to retrieve the value that the user is typing into the box.

TextField decorations

Making your TextField fancy

There's a ton of options to make your TextField more useful -- not infinite options, but lots. And they're all available through the InputDecoration widget.

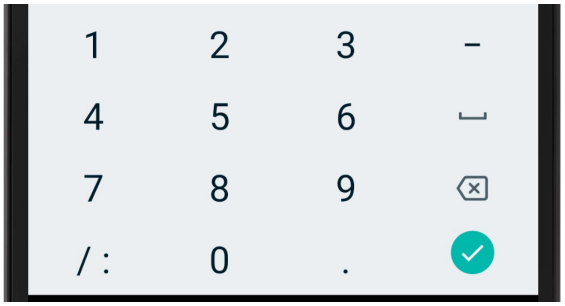
```
return TextField(
  controller: _emailController,
  decoration: InputDecoration(
    labelText: 'Email',
    hintText: 'you@email.com',
    icon: Icon(Icons.contact_mail),
  ),
),
```

Property	Description
labelText	Appears above the TextField. Tells the user what this TextField is for.
hintText	Light ghost text inside the TextField. Disappears as the user begins typing.
errorText	Error message that appears below the TextField. Usually in red. It is set automatically by validation (covered later) but you can set it manually if you need to.
prefixText	Text in the TextField to the left of the stuff the user types in.
suffixText	Same as above but to the far right.
icon	Draws an icon to the left of the entire TextField
prefixIcon	Draws one inside the TextField to the left.
suffixIcon	Same as above but to the far right.

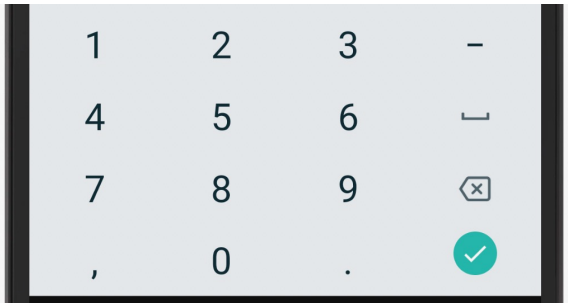
Passwords

```
return TextField(
  obscureText: true,
  decoration: InputDecoration(
    labelText: 'Password',
  ),
);
```

```
return TextField(
  keyboardType: TextInputType.number,
);
```

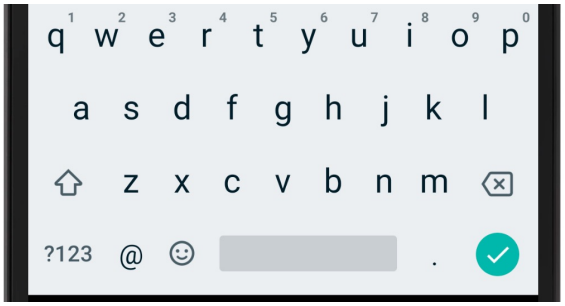


TextInputType.datetime

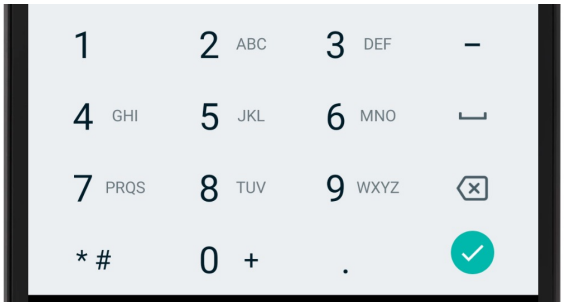


TextInputType.number

TextInputType.email



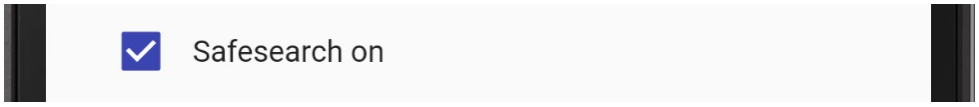
TextInputType.phone



Checkboxes

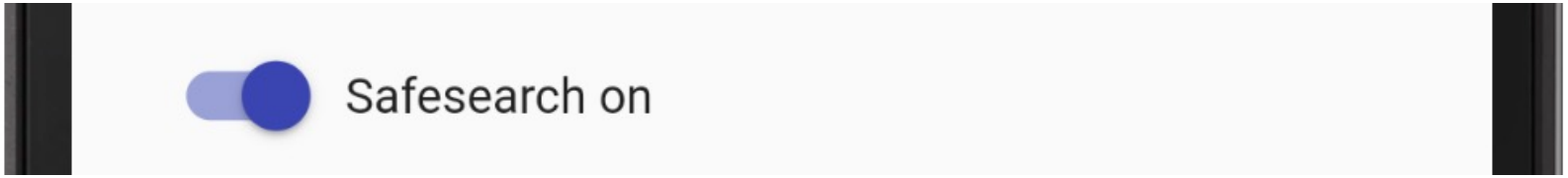
Checkboxes have a boolean value property and an onChanged method. Like all of the other input widgets, the onChanged method receives the value that the user set. Therefore in the case of Checkboxes, that value is a bool.

```
Checkbox(
  value: true,
  onChanged: (bool val) => print(val)),
```

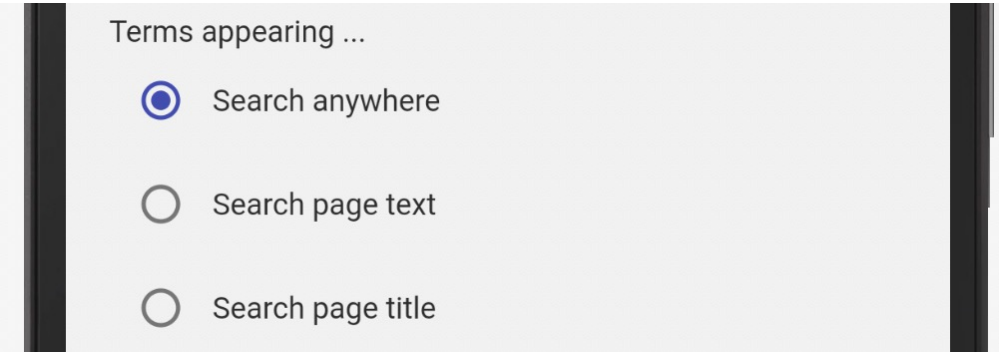


Switch

A Switch serves the same purpose as a Checkbox -- on or off. So the Switch widget has the same options and works in the same way. It just looks different.



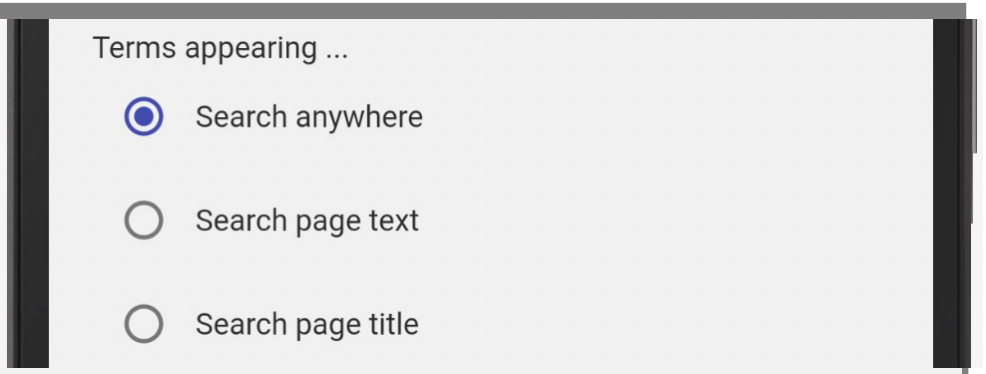
Radio buttons



- The groupValue property holds the value of the one Radio that is currently turned on. (and it groups them together)
- Each Radio also has its own value property; the value associated with that particular widget whether it is selected or not.
- In the onChanged method, you'll set the groupValue variable to the radio's value.

Foo.dart

```
SearchType _searchType;
//Other code goes here
Radio<SearchType>(
  groupValue: _searchType,
  value: SearchType.anywhere,
  onChanged: (SearchType val) => _searchType = val),
const Text('Search anywhere'),
Radio<SearchType>(
  groupValue: _searchType,
  value: SearchType.text,
  onChanged: (SearchType val) => _searchType = val),
const Text('Search page text'),
Radio<SearchType>(
  groupValue: _searchType,
  value: SearchType.title,
  onChanged: (SearchType val) => _searchType = val),
const Text('Search page title'),
```



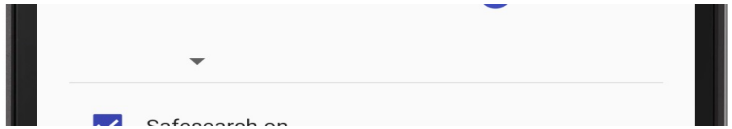
Slider()

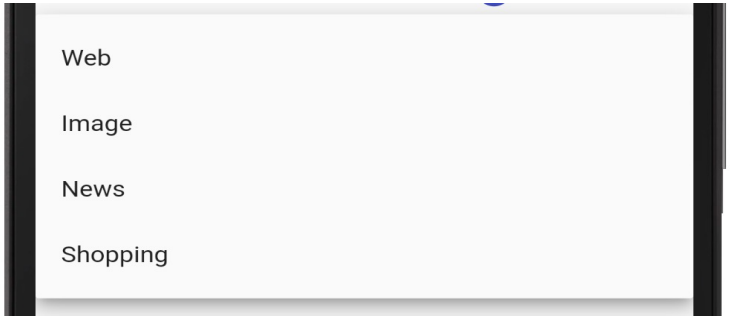
```
Slider(
  label: _value.toString(),
  min: 0, max: 100,
  divisions: 100,
  value: _value,
  onChanged: (double val) => _value = val,
),
```

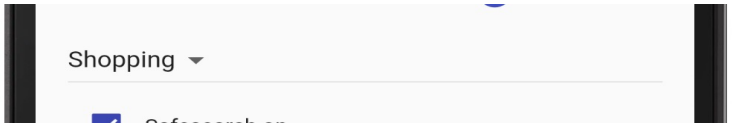
DropdownButton()

Dropdowns are great for picking one of a small number of things, like in an enumeration. Let's say we have an enum like this:

```
enum SearchType { web, image, news, shopping }
```







Foo.dart

```
SearchType _searchType = SearchType.web;
//Other code goes here
DropdownButton<SearchType>(
  value: _searchType,
  items: const <DropdownMenuItem<SearchType>>[
    DropdownMenuItem<SearchType>(
      child:Text('Web'), value: SearchType.web,),
    DropdownMenuItem<SearchType>(
      child:Text('Image'), value: SearchType.image,),
    DropdownMenuItem<SearchType>(
      child:Text('News'), value: SearchType.news,),
    DropdownMenuItem<SearchType>(
      child:Text('Shopping'), value: SearchType.shopping,),
  ],
  onChanged: (SearchType val) => _searchType = val,
),
```

tl;dr

- Form fields ...
  - Textfields
  - DropdownButtons
  - CheckBoxes
  - Radios
  - Switches
  - Sliders

Forms

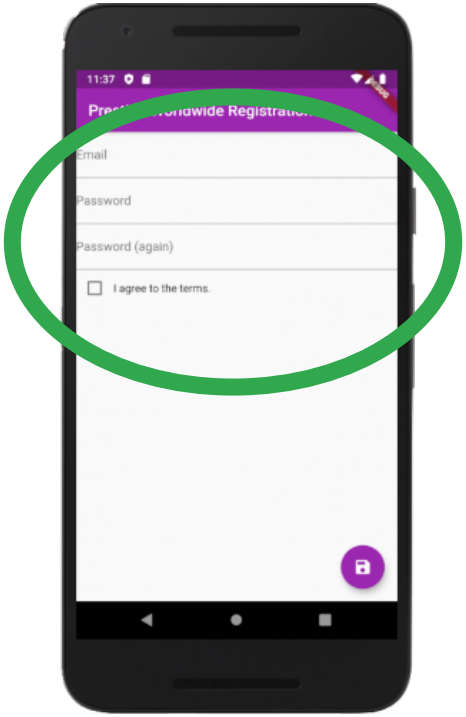
Grouping the fields so we can validate and save them together

tl;dr

- The need for a Form(), a Key and FormField() widgets
- Validating fields
- Submitting the form

Putting the form widgets together

The Form() widget groups our fields using a *key*



You don't need a Form widget... until you need to validate fields or group data together.

Its not rendered.

Foo.dart

```
GlobalKey<FormState> _key = GlobalKey<FormState>();
```

Foo.dart

```
@override
Widget build(BuildContext context) {
 return Form(
  key: _key,
  child: // form fields go here
 );
}
```

_key.*currentState*

- **.save()** - Saves all fields inside the form by calling each's onSaved
- **.validate()** - Runs each field's validator function
- **.reset()** - Resets each field inside the form back to its initialValue


But if _key.currentState.save() is calling a field's onSaved() we need to provide an onSaved method.
Same with validate()
But the TextField, Dropdown, Radio, Checkbox, and Slider widgets themselves don't have those methods.

FormField widget

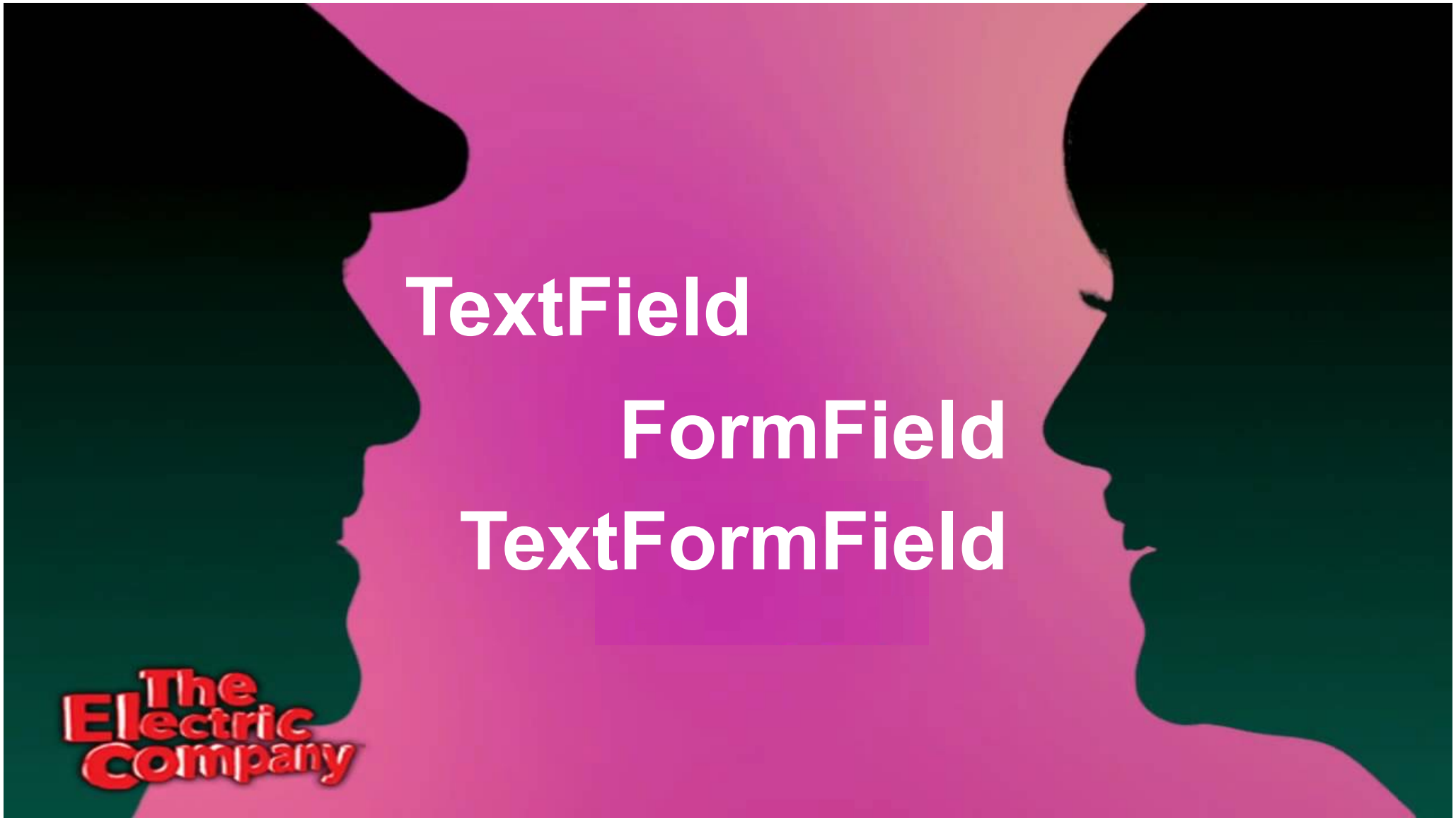
- This widget's entire purpose in life is to provide save, reset, and validator event handlers to an inner widget.

- So we first wrap a FormField widget around each input widget and we do so in a method called *builder*. Then we can add the onSaved and validator methods.

Foo.dart

```
FormField<String>(
  builder: (FormFieldState<String> state) {
    return TextField();
  },
  onSaved: (String initialValue) {
    // Push values to a repo or something here.
  },
  validator: (String val) {
    // Put validation logic here
  },
),
```

Any field widget ie. Radio, DropDownButton, Checkbox, Slider


TextField
FormField
TextFormField
The Electric Company

onSaved

- Please remember that your Form has a key which has a currentState which has a save() method.
- On a "Save" button press, you will write your code to call ...

```
_key.currentState.save();
```

- ... and it in turn invokes the onSaved method for each FormField that has one.

validator

- Similarly, you probably guessed that you can call ...

```
_key.currentState.validate();
```

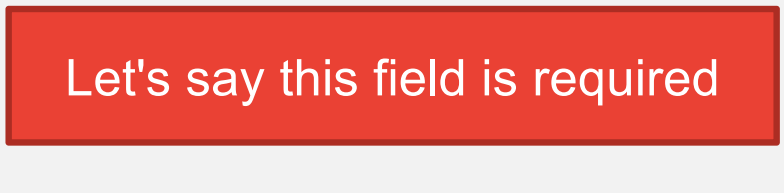
- ... and Flutter will call each FormField's *validator* method.

The validator returns a String or null

- String => Invalid data ... and here's the reason why.
- null => Yay! Valid data

Foo.dart

```
return Form(
 child: TextFormField(
  validator: (String val) {
   if (val.isEmpty)
    return 'We need something to search for';
   return null;
  },
 ),
);
```



Validate while typing

- Remember that the way to perform instant validation is to set Form.autovalidate to true and write a validator for your TextFormField.

Foo.dart

```
return Form(
 autovalidate: true,
 child: TextFormField(
  validator: (String val) {
   if (val.isEmpty)
    return 'We need something to search for';
   return null;
  },
 ),
);
```

Don't want your code to validate until the user has finished entering data? Set autovalidate to false. Then call validate() in the button's pressed event:

Foo.dart

```
RaisedButton(
 child: Text('Submit'),
 onPressed: () {
  if (_key.currentState.validate()) {
   _key.currentState.save();
   print('Successfully saved the state.')
  }
 },
)
```

If every field passes validation, run their save methods.

tl;dr

- The need for a Form(), a Key and FormField() widgets
- Validating fields
- Submitting the form