Flutter Top to Bottom

Dan Field

Flutter Top to Bottom

- Introduction
- Architecture, Components
- Cross Platform, Engine

Introduction

What Flutter is, etc

Flutter is Google’s UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase.

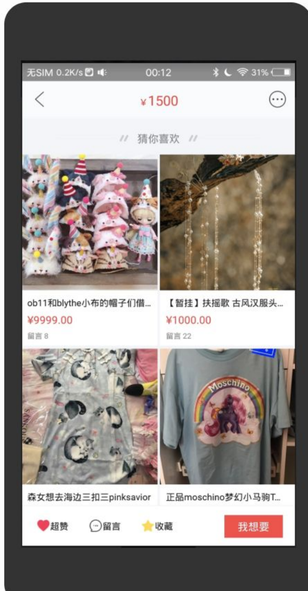
After we're done, find out more at flutter.dev

Flutter is Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase.

An example of what can be built with Flutter

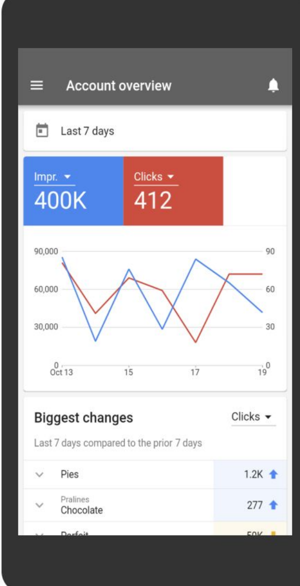
Examples of what the community has built with Flutter

Examples of typical Flutter apps



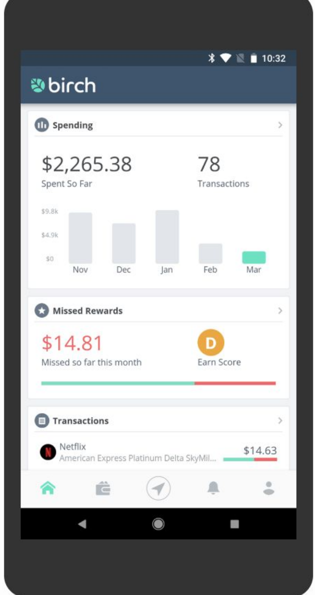
Alibaba's app incorporates Flutter to power parts of their app.

Google Ads



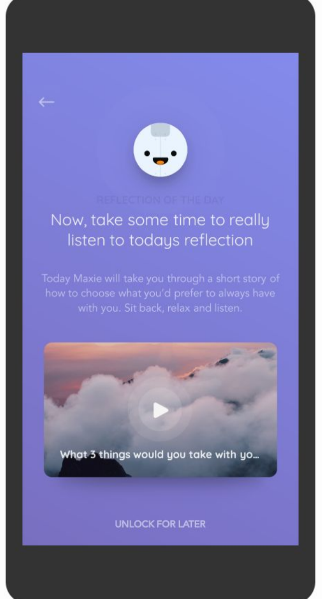
The Google AdWords app helps you keep your ad campaigns running smoothly — no matter where your business takes you.

birch



Credit card rewards app to manage and optimize your existing cards.

reflectly



A beautiful journal and mindfulness app driven by artificial intelligence.

What does a Flutter app look like?

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

void main() {
  runApp(MaterialApp(home: HelloWorld()));
}

class HelloWorld extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final TextStyle style = Theme.of(context).textTheme.headline1;
    return Container(
      color: Colors.white,
      alignment: Alignment.center,
      child: Text('Hello World', style: style),
    );
  }
}
```

Hello World

A few things you should know at the outset

- Flutter is an open source project
- It’s written in the Dart language
- Apps run unchanged on Android, iOS, Web (beta), desktop (technical preview)
- There’s a nice stable of development tools
- “Hot Reload” means that you can try a change in under two seconds

Flutter Origins

- 2014, the “Sky” project, an ever smaller subset of Chrome
- Spring 2015: Dart instead of JavaScript, React architecture, Material Design
- Examples, documentation, tools, I18n, a11y, native L&F, plugins, typography, images, text input, ...
- December 2018: Flutter 1.0
- Current release is 1.12

The “React” Architecture

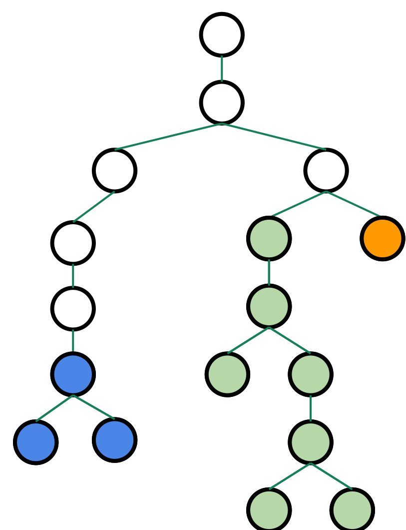
Introduced by FaceBook in 2013

- In an MVC application, it's about keeping the model and the view in sync
- Initially for web applications, later as a framework for native applications
- For the origin story:

  See Pete Hunt's "React: Rethinking best practices" talk at JSConf.EU

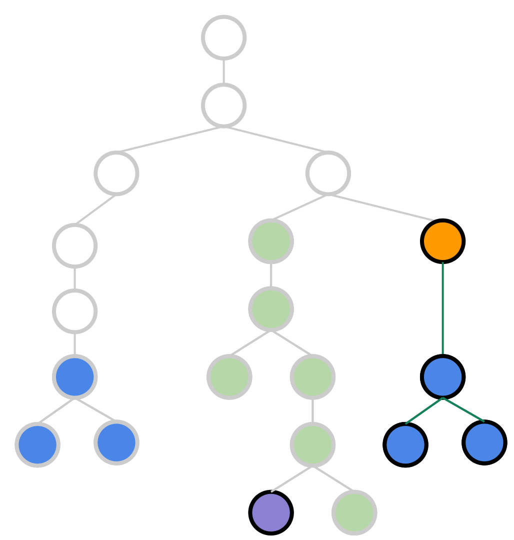
The User Interface Appears

state ⇨ display list



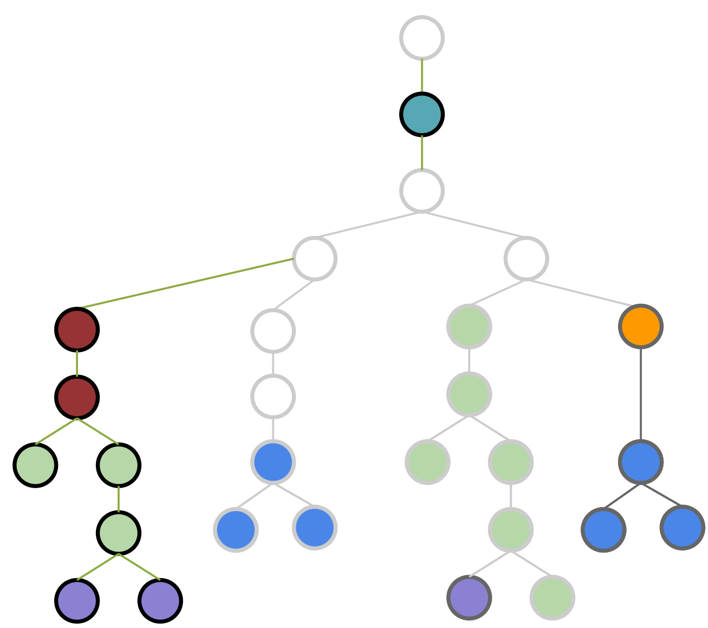
The Application's State changes

Δstate ⇨ Δdisplay list

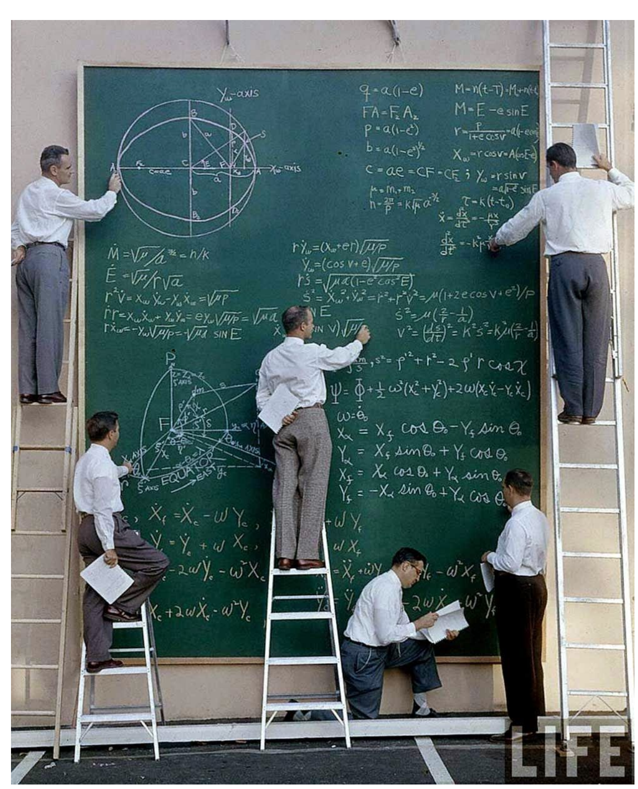


Display List changes cascade

Δstate ⇨ Δdisplay list ⇨ Δdisplay list...



This is where things start to get complicated



The React Idea

f(state) ⇨ display list *model*

The React Implementation

- When the application starts or its state changes:
  - Ask for a new *model* of the entire display list
  - Update the display list to change to match the model*
- The display list is the React system’s responsibility, not the app developer’s
- *Yes: it’s quite surprising that this can be done efficiently

Flutter’s React Implementation

- Display list model
  - Nodes are immutable **Widgets**
  - Widget nodes build subtrees
  - Leaf widget nodes render text, images, graphics

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

void main() {
  runApp(MaterialApp(home: HelloWorld()));
}

class HelloWorld extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final TextStyle style = Theme.of(context).textTheme.headline1;
    return Container(
      color: Colors.white,
      alignment: Alignment.center,
      child: Text('Hello World', style: style),
    );
  }
}
```

StatefulWidgets can ask to be rebuilt

- The `setState()` method just marks a subtree as needing to be rebuilt
- When it's time to build a new frame:
  - The marked widgets are rebuilt with their `build()` methods
  - The updated widget tree is used to update Flutter's display list*
  - *Yes: it's quite surprising that this can be done efficiently

```
class HelloWorldState extends State<HelloWorld> {
  int count = 0; // The widget's state
  @override
  Widget build(BuildContext context) {
    final TextStyle style = Theme.of(context).textTheme.headline1;
    return GestureDetector(
      onTap: () {
        setState(() { // Ask for HelloWorld to be rebuilt
          count += 1; // ... with the updated state
        });
      },
      child: Container(
        color: Colors.white,
        alignment: Alignment.center,
        child: Text('Hello World $count', style: style),
      ),
    );
  }
}
```

How is Flutter's display list updated

- Display list nodes are **Elements**, one per widget
- When a widget is added, removed, or moved the corresponding Element is updated
- Updates are done in one top down pass
  - Starting with elements whose widgets called setState()
  - Stopping when an element is reached whose new widget is identical to the old one

How is Flutter’s display list updated (continued)

- Each Element node also has a **Renderer**
- Renderers are the workhorses
  - Intrinsic geometry and layout
  - Hit testing
  - Painting
- Not every Element has a renderer
  - InheritedWidgets are used to pass along values like visual theme data

How is Flutter’s display list updated (continued)

- Renderers are updated in one pass, starting with the updated element subtree roots
- Renderers are created or updated in place. In place updates happen if:
  - The element’s old widget has the same type as the new one
  - The old widget has the same key as the new one (by default, widget keys are null)
  - Don’t worry about the keys for now
- An updated renderer can mark itself as needing layout or painting
- There are additional passes for layout and painting of the marked renderers
  - Layout is one pass, renderers compute the size and location of each (renderer) child
  - Painting is back to front

How is Flutter’s display list layout updated (continued)

- Only subtrees below renderers that were marked as needing layout are processed
- Renderer layout is governed by a simple BoxConstraints type
  - minWidth, maxWidth
  - minHeight, maxHeight
- The entire process is one pass:
  - Constraints are applied top-down
  - Sizes and positions are computed bottom-up
- Parents indicate if they use the child’s size
  - If not, then the “need layout” flag will not propagate upwards

How is Flutter’s display list updated (continued)

- That was just a quick survey of the process
- The implementation is pretty complicated
- There are other trees which must be updated, like Semantics (a11y) and mouse regions
- On the upside
  - App developers generally don’t need to be aware of any of it
  - The implementation is fast and correct

Flutter’s Widgets

Flutter emphasizes widget composition

- The component widget classes are rarely subclassed
- Instead, apps subclass StatelessWidget and StatefulWidget
    - Compose the UI you want
    - Configure the composition with widget parameters
    - Make the class const because efficiency

Flutter’s API is unlike traditional UI toolkits

- Traditional toolkit base classes, like Android’s View, can get very big
  - Colors and themes
  - Text and icon styles
  - Contents layout, padding, alignment
  - Scrolling
  - Input handling, keyboard focus
  - Lifecycle
  - …

Flutter’s API is unlike traditional UI toolkits (continued)

- Flutter’s widget base classes don’t include anything at all
- They’re essentially just an abstract Widget-valued `build()` method
- Additional specialized widgets are used to add features

```dart
class HelloWorld extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final TextStyle style = Theme.of(context).textTheme.headline1;
    return Center(
      child: DecoratedBox(
        decoration: BoxDecoration(color: Colors.white),
        child: Padding(
          padding: EdgeInsets.all(64),
          child: DefaultTextStyle(
            style: Theme.of(context).textTheme.headline1,
            child: Text('Hello World'),
          ),
        ),
      ),
    );
  }
}
```

Hello World

Flutter's API is unlike traditional UI toolkits (continued)

- Many traditional toolkits, like Flex or Android, combine code and markup
- Flutter uses code as markup
- Non-leaf widgets have
  - A Widget valued child property
  - List<Widget> valued children properties

```dart
class HelloWorld extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final TextStyle style = Theme.of(context).textTheme.headline1;
    return Center(
      child: DecoratedBox(
        decoration: BoxDecoration(color: Colors.white),
        child: Row(
          mainAxisSize: MainAxisSize.min,
          children: [
            Icon(Icons.camera, size: 64, color: Colors.blue),
            Text('Hello World', style: style),
            Icon(Icons.camera, size: 64, color: Colors.orange),
          ],
        ),
      ),
    );
  }
}
```

Hello World

Flutter's "basic" library (nearly 200 widgets)

- A minimal not-opinionated app framework
    - The WidgetsApp singleton
    - Routes and route navigation
- Flutter widgets that are intended to one narrowly defined thing well
    - Layout - for example: Row, Column, Align, Center
    - Rendering - for example: Text, Icon, Image, CustomPaint,
    - Input handling - for example: GestureDetector, FocusNode
- Support for scrolling, accessibility, and internationalization

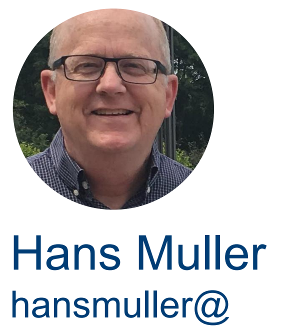
Flutter's "material" library (about 100 widgets)

- An opinionated app framework
  - MaterialApp
  - Scaffold
  - Theme
- Application components per the Material Design specification
- Some components tailor their behavior and appearance to match the underlying platform
  - Respect the user's muscle memory
  - Optionally eliminate some visual differences, notably the default font

Flutter’s “cupertino” library (about 30 widgets)

- Components that look and feel like their native iOS counterparts
  - CupertinoActionSheet
  - CupertinoTabBar
  - CupertinoButton
  - CupertinoContextMenu
  - CupertinoDatePicker, CupertinoTimerPicker
  - CupertinoAlertDialog, CupertinoDialog
  - CupertinoNavigationBar
  - CupertinoScrollbar
  - CupertinoSlider
  - CupertinoSlidingSegmentedControl
  - CupertinoSwitch
  - CupertinoTabView
  - CupertinoTextField

Thanks for Listening



Hans Muller
hansmuller@

Dan Field
dnfield@

Find Flutter at:

flutter.dev

flutter.dev/youtube

@flutterdev

Making a Toolkit Cross Platform

Why not the Web Platform?

- It’s cross platform, and has a variety of excellent UI toolkits available.
- The main problem is around performance and, to a lesser extent, JavaScript.
- The Flutter team basically started by asking: “Can we make a fast (60+fps), beautiful applications for mobile devices using Chrome?”
- What if we get rid of:
    - Web backwards compatibility
    - HTML parsing
    - DOM
    - JavaScript

Fitting Multi-platform Constraints

- No interpreted code in final application (because of iOS).
  - What about JavaScript?
- Support for ARM64, ARMv7, x64 CPU architectures.
- Run using limited RAM and disk space.

Fitting Developer Constraints

- Majority of developers will expect garbage collection.
- Rapid iteration during development cycles.
- Performance that is at least as good as the native platform toolkits.
- Full control over rendering and layout regardless of platform.

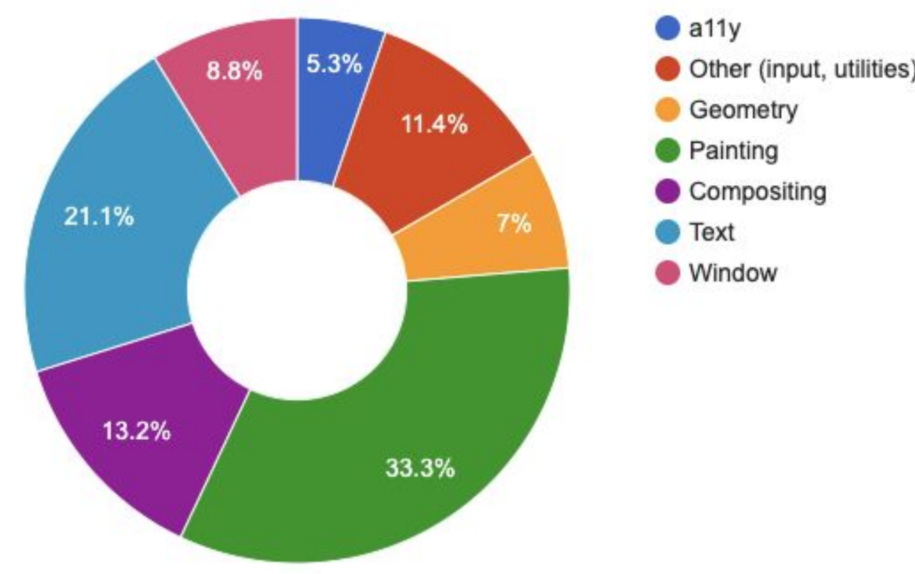
The Basic Ingredients (third-party)

- **Skia:** the GPU accelerated 2D graphics engine that is used by Chrome and Android.
  - Abstracts software based, OpenGL, Vulkan, and Metal rendering.
  - Abstracts font rendering.
- **Harfbuzz:** A glyph shaping library to help with text layout and glyph shaping.
- **Dart:** A multi-platform, strongly typed, garbage collected language that can be run in JIT mode or compiled and assembled into native machine code (“AOT” compiled).
  - Dart feels familiar to people who already know Java, C#, or JavaScript.

Flutter’s Core Library: dart:ui

- Flutter achieves success through an aggressively layered and composed architecture.
- In the Flutter Framework: foundation, scheduling, semantics, gestures, painting, widgets, material, etc.
    - Exports many, many public types (over 200 widgets).
- In dart:ui: semantics, input, painting, compositing, geometry, text.
    - Exports 114 lower level building blocks.
    - Expose the essential and *best* parts of underlying APIs.
- Native platform implementations only have to directly support these types.

API Surface



RenderObjects?
Where we're going, we don't need RenderObjects.
imgflip.com

```dart
import 'dart:ui';

void main() {
  window.onBeginFrame = (Duration duration) {
    final SceneBuilder builder = SceneBuilder();
    final PictureRecorder recorder = PictureRecorder();
    final Canvas canvas = Canvas(recorder);

    canvas.drawRect(
      Rect.fromLTWH(100, 100, 200, 200),
      Paint()..color = Color.fromARGB(255, 50, 50, 50));

    builder.addPicture(Offset.zero, recorder.endRecording());

    final Scene scene = builder.build();
    window.render(scene);
    scene.dispose();
  }
  window.scheduleFrame();
}
```

...maybe we do need Render Objects!

Flutter Engine

- Most* platforms use a shared C++ codebase to manage compositing, rendering, and integration with the Dart runtime, Skia, Harfbuzz, etc.
- Platforms also provide a specific embedding implementation:
  - iOS/macOS: Objective C (public), Objective C++ (private)
  - Android: Java
  - Linux/Windows: C++
  - Embedder API: C
  - Web: Dart!

The Core of the Engine

- Compositor: “Flow”, a simple compositor based on Skia.
- Rasterizer: manages the on screen render surface and the context of the compositor.
- Runtime: manages a Dart Runtime and various Dart Isolates (units of execution).
- Shell: abstracts platform specific details and connects them to the compositor, runtime, rasterizer, etc., and owns the Task Runners for rendering, running Dart code, decoding images, and interacting with the platform.

Platform Embedding Responsibilities

- Create a rendering surface for the Shell to work with, e.g. OpenGL, Metal, Vulkan.
- Create/setup threads, provide event loop interop.
- Inform the shell of system events related to application lifecycle or settings.
- Bridge between Flutter's representation of Semantics with the system Accessibility Services.
- Provide text/pointer input information.
- Host native plugins.
- Produce binaries suitable for use on the platform.

A Minimal Flutter Embedder

- Tell the Engine what rendering API to use, e.g. OpenGL
- Give it a Frame Buffer Object to use.
- Tell the Engine where to find the Flutter Dart blobs.
- Respond to *PresentSurface* calls.
- https://github.com/chinmaygarde/fluttercast - just over 500 C++ SLOC.
- Full featured Flutter embedders range from 15-30k SLOC.

Revisiting the Web Platform

- Flutter now runs in the browser!
- It uses a Dart based implementation of dart:ui.
- Supports both an HTML/Canvas based backend, as well as a WASM compiled version of Skia using WebGL.
- For the browser, Dart is compiled to JavaScript.

Running on Multiple Platforms

Packaging

- Embedders must link with the core of the engine into a product that can be shipped to their respective platforms - e.g. a JAR or AAR file for Android, a .framework for iOS, etc.
- They must also be able to load packaged assets from applications, including compiled Dart binaries and image/font resources for the platform.
- Flutter's tooling further supports creating end binary packages, such as APK/AAB/IPA programs for various platforms.
- Just like the Flutter framework makes it easier to use the primitives in dart:ui, the tooling makes it easier to use the building blocks provided by the Dart SDK and the Engine.

Profiling

- Users want to know: is my app fast?
- Flutter answers this through profiling overlays and traces.
- Dart provides a built in VM Service that can be used to debug or profile running code.
- Tracing information is available from Dart, Skia, as well as specific traces from Flutter code.

Cross Platform Pitfalls and Traps

Memory

- Your application will die if it uses too much memory.
- Platforms do not generally tell you how much memory you can use before they kill you.
- The platform may use interesting accounting tricks that change without warning.
- Getting OOM killed is very confusing for developers, particularly if you're using a GC enabled language.
- Tip: make sure graphics related caches are reasonable multipliers on the screen resolution, which is usually a good proxy for the amount of memory your application will be allowed to use.

Energy

- We often focus on frame rate when thinking of performance, but energy usage is critical on mobile devices.
- Flutter works to be efficient and economical in its usage of CPU and GPU resources.
- Flutter also gives developers fine grained control over every frame:
  - Complicated animations are easier to make with Flutter!
  - But the model requires more work per frame than other simpler animation frameworks.
- Some platforms report energy usage that does not actually match up with battery discharge rates, which causes confusion for developers.

Binary Size

- Mobile Applications must have a small disk footprint.
- Any third-party UI framework will involve binary overhead compared to using built in tools and libraries supplied by the mobile OS.
- Flutter compares better than some other cross-platform mobile solutions, but still requires more space than OS native toolkits.

Incremental Adoption

- Developers may want to use your toolkit for pieces of their application rather than rewriting it whole.
- Flutter supports incremental adoption via normal platform methods, and significant parts of the engine embedding code is devoted to making this work nicely.
- When designing bindings between the toolkit and the OS, assume that developers will need to use them within existing, mature applications.

Thanks for Listening

Hans Muller
hansmuller@

Dan Field
dnfield@

Find Flutter at:

flutter.dev

flutter.dev/youtube

@flutterdev