Flutter Architecture

CS 442: Mobile App Development

Agenda

– Flutter architecture overview
– Widgets
– Declarative UIs
– Rendering pipeline

Diagrams from Flutter Architectural overview

Flutter Architecture

–

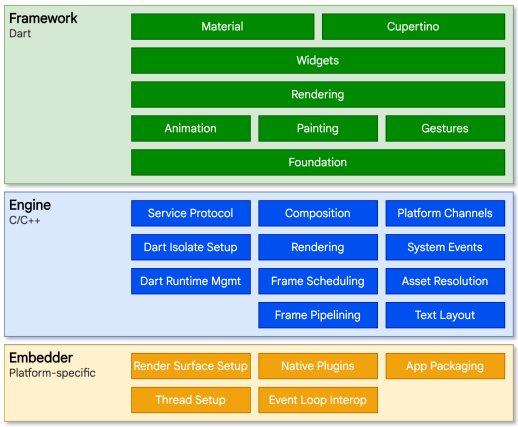
Layered / Ring architecture

–

3 layers: Framework, Engine, Embedder

–

All the layers and pieces are open-source! This is great for folks who want to build their own embedders (e.g., for other platforms) and those working on Flutter apps who want to better understand how base classes work / what they provide.



Embedder

– Integrates with the OS/HW layer to access low-level services (e.g., for drawing, device features, I/O)
– Connects higher-level Flutter code to the system event loop
– E.g., written in Objective-C for iOS, Java for Android

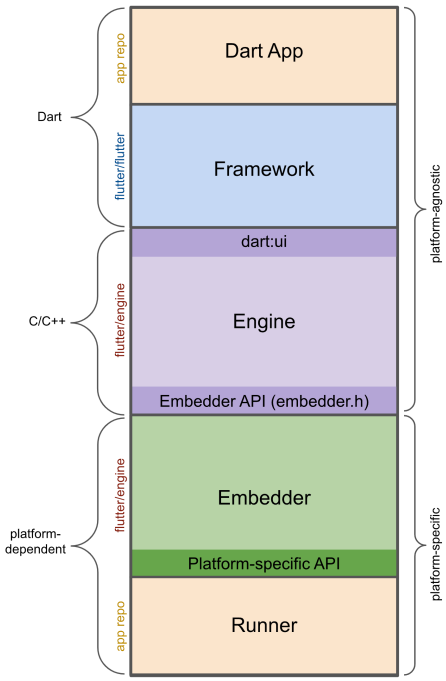
Engine (C/C++)

– Takes higher-level scenes (built from widgets) and rasterizes them (i.e., creates pixel-level renderings)
– Includes efficient implementations of Flutter & Dart APIs
– Communicates with the embedder

Framework (Dart)

– Defines pre-built widgets for composing UIs
– Defines APIs for creating new widgets
– Performs high-level scene compositing
– We will live here most/all of the semester!

Flutter app



– Dart App: our code -- mostly defines and composes widgets
– Framework: pre-built widgets; composites scenes from our definitions
– Engine: low-level API implementations; rasterizes scenes
– Embedder: works with OS
– Runner: synthesizes all components into a runnable "app" package

Widgets all the way down

– Just about all our Flutter code goes towards defining "Widgets" ("components" in React)
– Some inheritance, but primary mechanism we use for building UIs is *composition*
– Widgets that contain widgets that contain widgets, etc.

"Declarative" UI

–

vs. imperative style of UI construction

–

e.g., instantiate a "View" object, then configure it over many lines/method calls (implies mutable UI elements)

–

in the declarative style, *immutable* UI elements are often configured in a single constructor call

```
// Imperative style
var par = WidgetA();
par.setTitle('Some title');
par.setColor(Color.red);

var childWidget = WidgetB();
childWidget.setTitle('Some title');
par.addChild(childWidget);
```

```
// Declarative style
return WidgetA(
  title: 'Some title',
  color: Color.red,
  child: const WidgetB(title: 'Some title')
);
```

What is a widget?

–

A widget is a class that corresponds to some part of the UI

–

Widgets are *immutable*! I.e., once constructed, they cannot be changed

–

A widget describes how its attributes (which are all `final`) are translated into a corresponding element in the UI

–

A UI is based on a hierarchy / tree of immutable widgets

–

Note: because widgets are immutable, "changing" UI attributes necessitate re-constructing parts of the widget tree

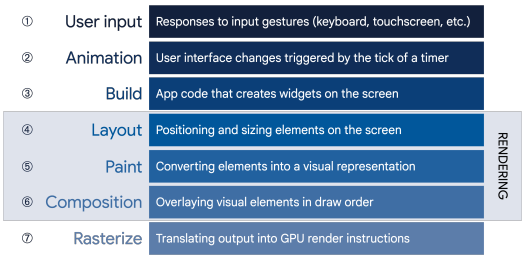
–

But Dart is also good at reusing objects -- especially if they are declared `const`

Flutter is, at its core, a series of mechanisms for efficiently walking the modified parts of trees, converting trees of objects into lower-level trees of objects, and propagating changes across these trees.

How do widgets get rendered?

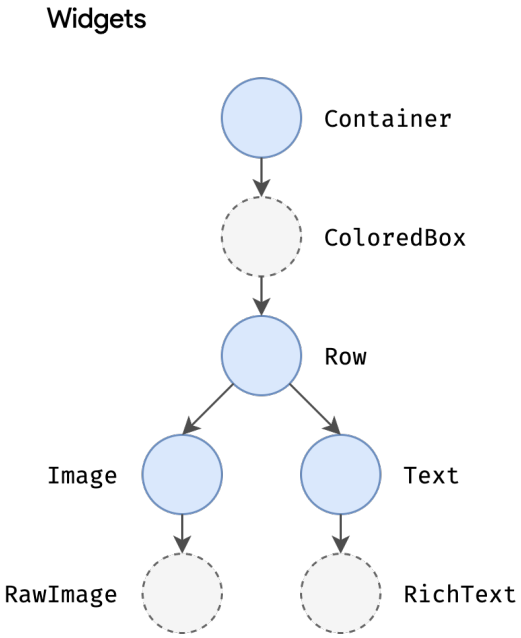
I.e., how does Flutter take an instance of a Widget class and *render* its on-screen analog?




```
Container(
  color: Colors.blue,
  child: Row(
    children: [
      Image.network('https://www.example.com/1.png'),
      const Text('A'),
    ],
  ),
);
```

Example from Flutter Architectural Overview

The widget tree

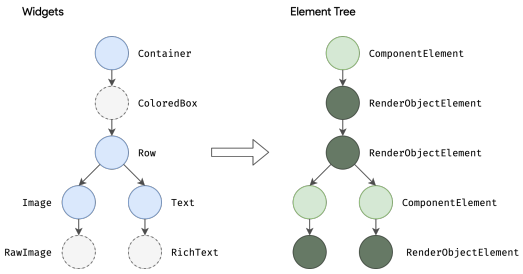



```
Container(

  color: Colors.blue,
  child: Row(
    children: [
      Image.net-
work('https://www.example.-
com/1.png'),
      const Text('A'),
    ],
  ),
);
```

Note that some widgets are composed of other widgets (e.g., `Container`s insert `ColoredBox`s, `Image`s insert `RawImage`s, etc.

Widgets → Elements



In the build phase, Flutter takes the widget tree and translates it into a corresponding *element tree*, with one element per widget.

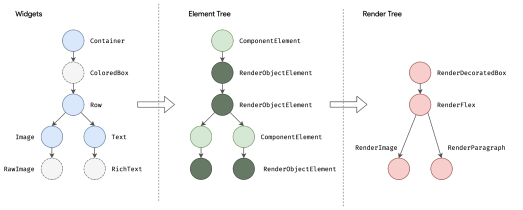
Two types of elements:

– `ComponentElement`: a container for other elements

– `RenderObjectElement`: an element used in layout/painting

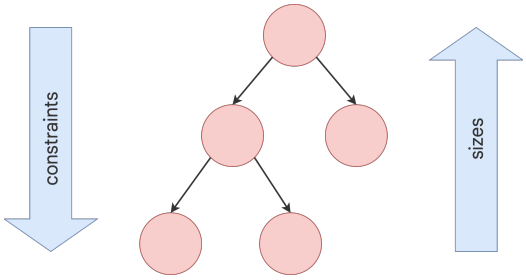
Importantly, even though widgets / parts of the widget tree are frequently reconstructed, Flutter "walks" the widget tree and carefully rebuilds only select parts of the element tree.

Layout & Rendering



In these phases, each `RenderObjectElement` is used to create/update a `RenderObject` subclass. Note: these aren't primitive / low-level / pixel-level representations!

Box constraint model



To perform layout, Flutter walks the render tree in a depth-first traversal and passes down size constraints from parent to child. In determining its size, the child must respect the constraints given to it by its parent. Children respond by passing up a size to their parent object within the constraints the parent established.

This is an efficient algorithm ($O(N)$) for laying out all objects in the render tree.

Ultimately, every `RenderObject` will have a defined size/position, and can be composited and rendered by the Engine & Embedder layers.

> *Flutter is, at its core, a series of mechanisms for efficiently walking the modified parts of trees, converting trees of objects into lower-level trees of objects, and propagating changes across these trees.*