Mobile Application Architecture

Written by Thapanapong Rukkanchanunt

Outline

Common Architectural Patterns

- Model-View-Controller (MVC)
  - Model = data and business logic
  - View = user interface and presentation
  - Controller = intermediary handling user input, processing it to model and updating view
- Model-View-Presenter (MVP)
  - Presenter = update view based on changes in the model and handling user input
- Model-View-ViewModel (MVVM)
  - ViewModel = mediator between view and model

Client-Server Architecture in Mobile Apps

- The client, often the mobile device or application running on it, is responsible for the user interface, presentation logic, and user interactions.
- The server handles data storage, business logic, and the processing of requests from clients. It manages the application's core functionality and often interacts with databases, external services, and other resources.
- Application Programming Interfaces (APIs) define how the client and server interact. RESTful APIs or GraphQL are commonly used in mobile app development.

GraphQL

- GraphQL is a query language and runtime for APIs (Application Programming Interfaces) that was developed by Facebook.
- It provides a more efficient, powerful, and flexible alternative to traditional REST APIs by enabling clients to request only the specific data they need, and no more.
- GraphQL allows clients to retrieve multiple types of data in a single request and is designed to address some of the limitations and challenges associated with RESTful API development.

Example of GraphQL

- In this example, the client is requesting information about a user with ID 123, including the user's ID, name, email, and a list of their posts with titles and bodies.

```
query {
  user(id: 123) {
    id
    name
    email
    posts {
      title
      body
    }
  }
}
```

Restful APIs vs GraphQL

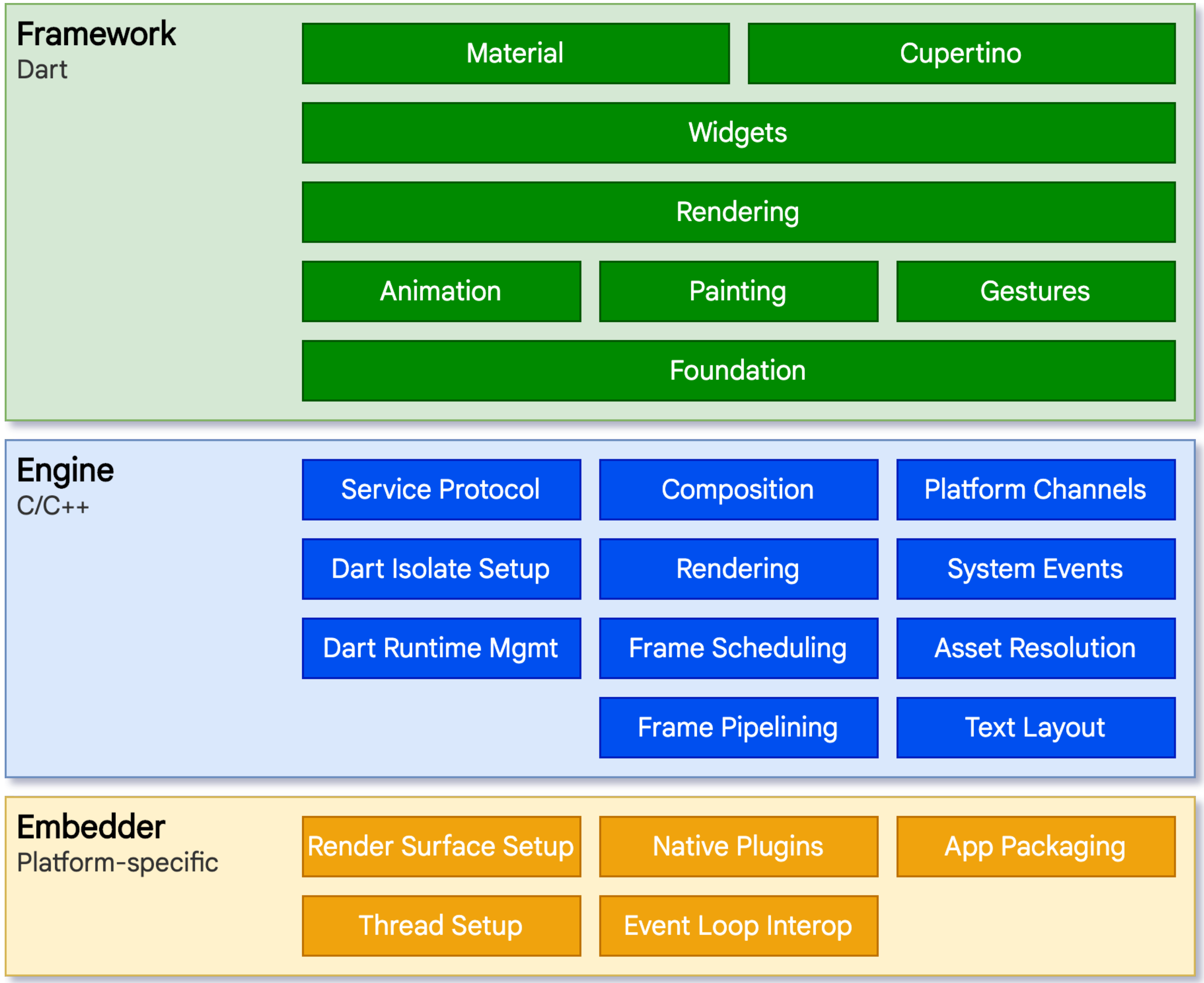
- Flexibility:
  - GraphQL offers more flexibility as clients can request precisely the data they need.
  - RESTful APIs may result in over-fetching or under-fetching as clients receive fixed responses.
- Efficiency:
  - GraphQL can be more efficient in terms of data transfer, especially when dealing with mobile devices or limited bandwidth.
  - RESTful APIs might transfer more data than necessary, impacting performance.
- Adaptability:
  - GraphQL is well-suited for scenarios where the data requirements are complex and may change over time.
  - RESTful APIs are suitable for simpler scenarios where the data requirements are stable.

Flutter Concepts

- Since we will use Flutter in development, we should understand the concepts behind Flutter.
- Flutter is a cross-platform UI toolkit that is designed to allow code reuse across operating systems.
- During development, Flutter apps run in a VM that offers stateful hot reload of changes without needing a full recompile.
  - For release, Flutter apps are compiled directly to machine code, whether Intel x64 or ARM instructions, or to JavaScript if targeting the web.

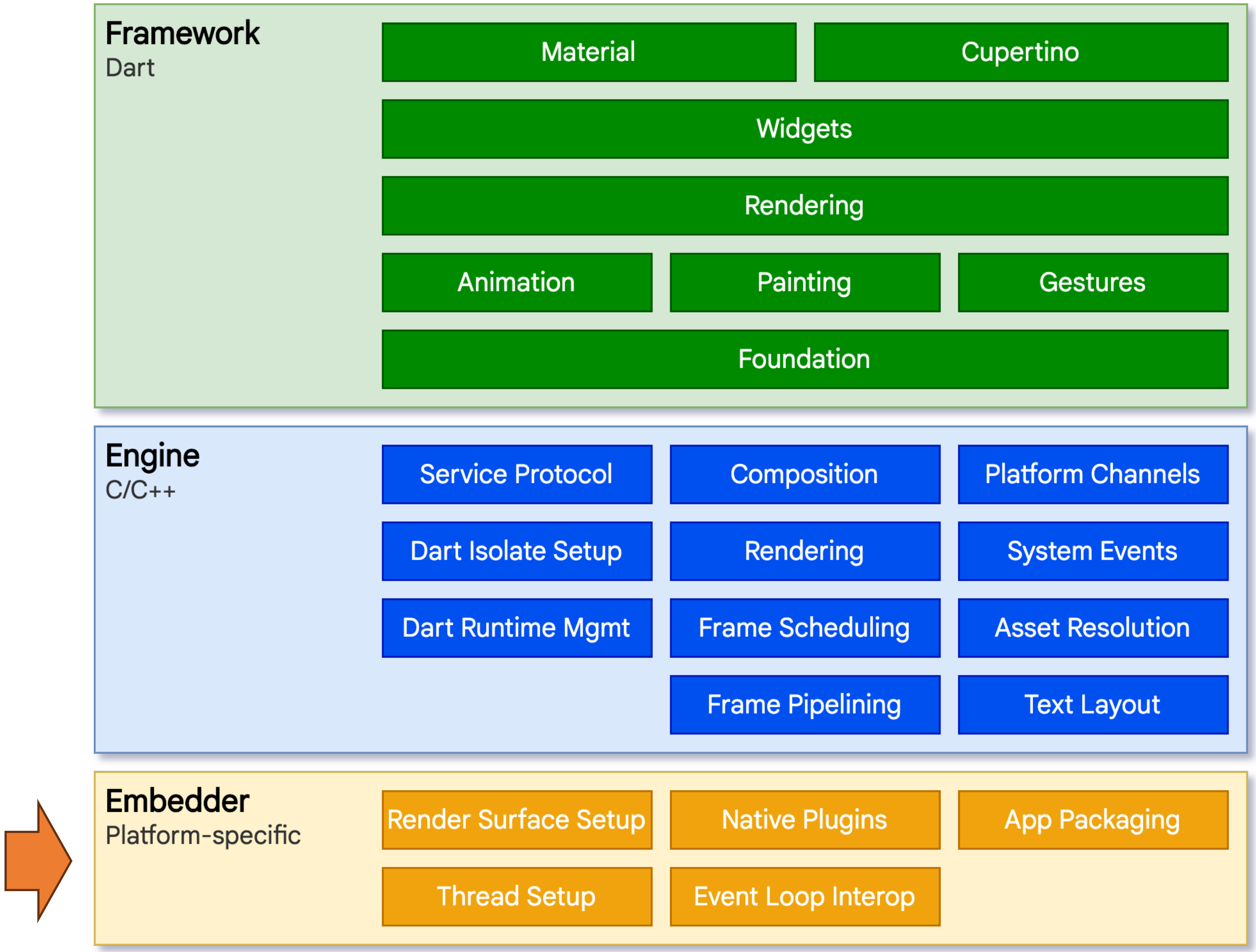
Architectural Layer

- Flutter is designed as an extensible, layered system.
- It exists as a series of independent libraries that each depend on the underlying layer.
- No layer has privileged access to the layer below, and every part of the framework level is designed to be optional and replaceable.



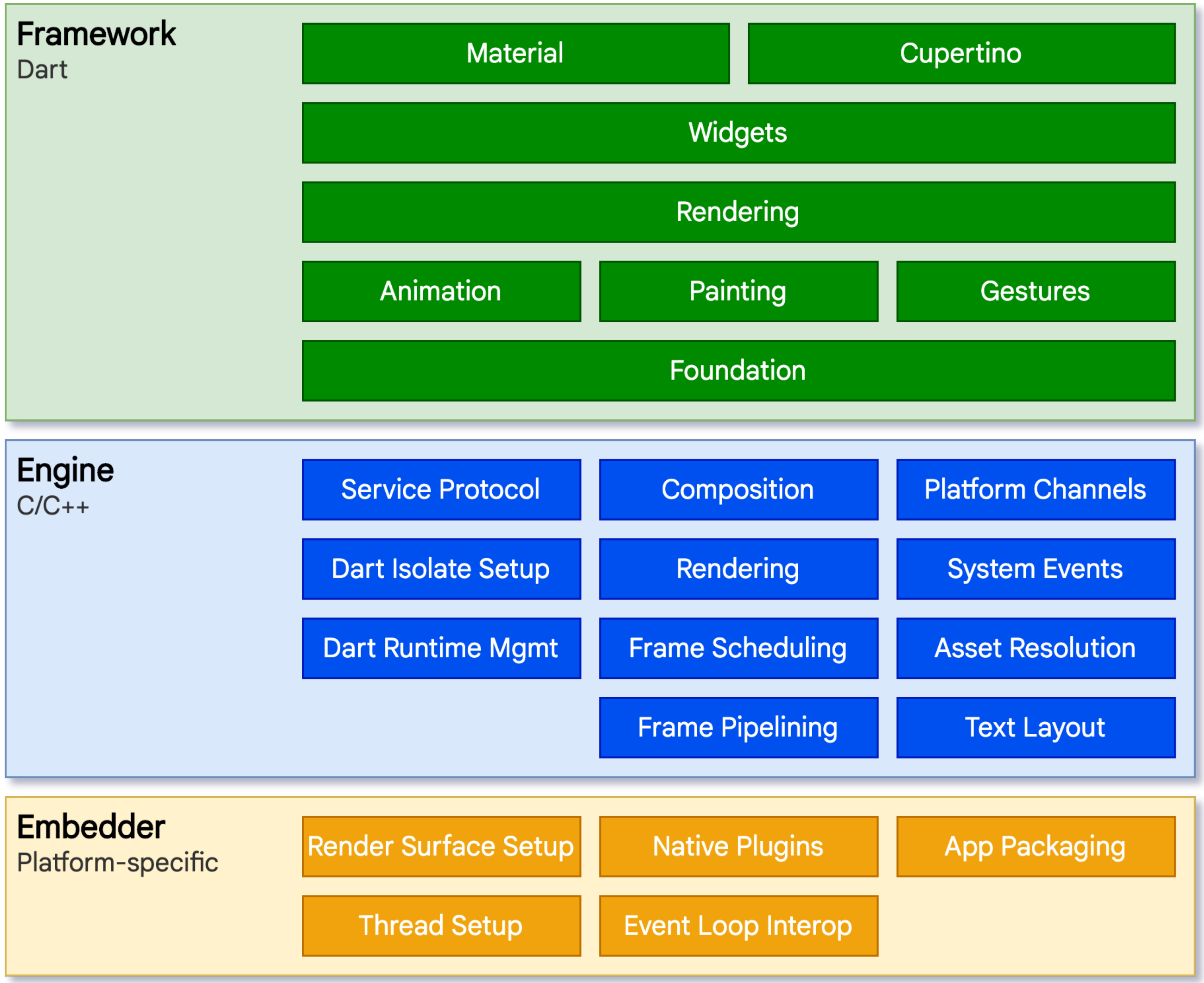
Embedder

- The **Embedder** is platform-specific and coordinates with the underlying operating system for access to services like rendering surfaces, accessibility, input, permission settings, intent sharing, etc.
- It is written in a language appropriate for the platform; Java and C++ for Android, Objective-C and Objective C++ for iOS.



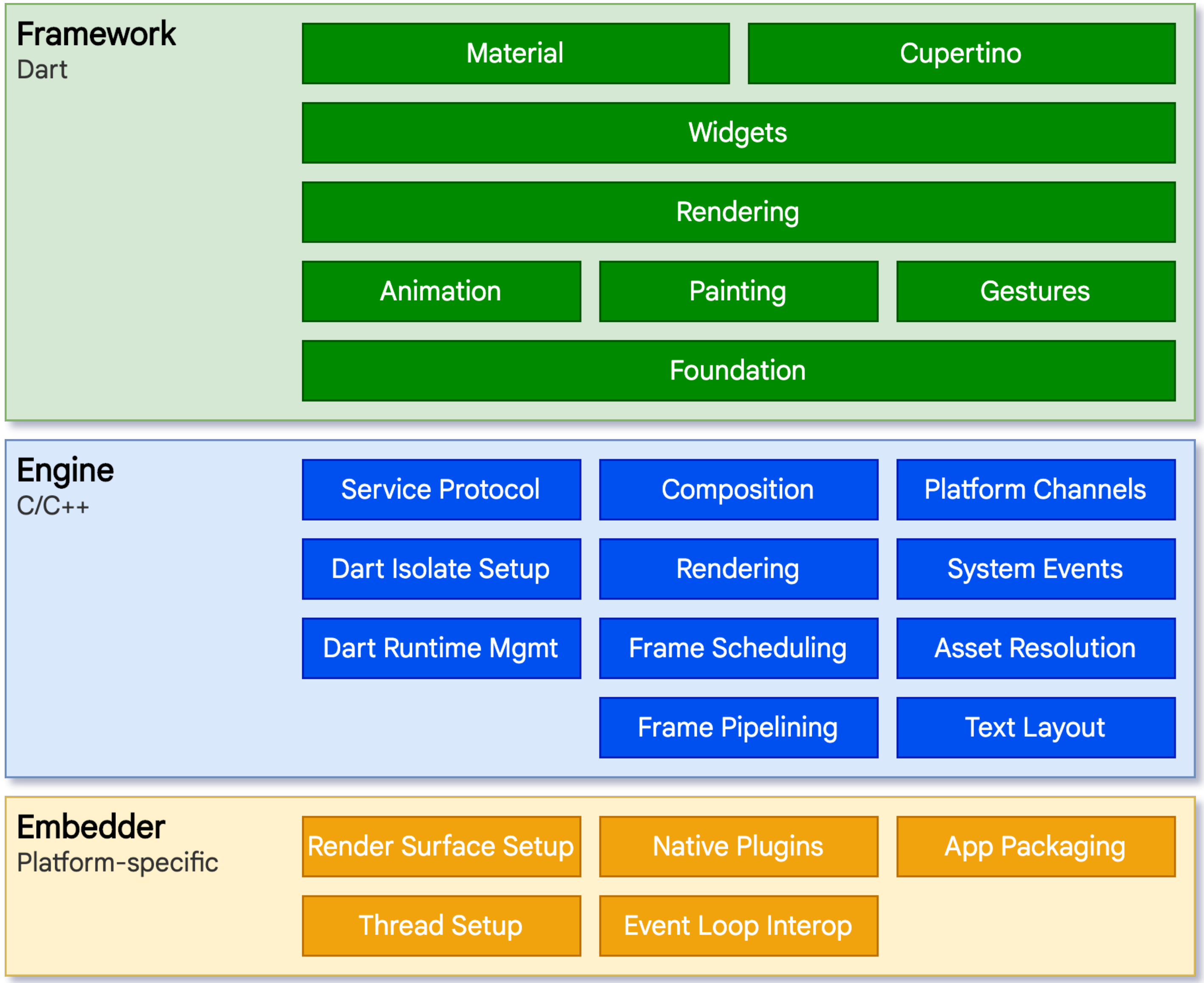
Engine

- **Flutter engine** provides a low-level implementation of Flutter's core API which includes animations and graphics (Impeller, Skia), text layout, file and network I/O, accessibility support, etc.
- It is written in C++ and exposed to flutter framework through `dart:ui` library.



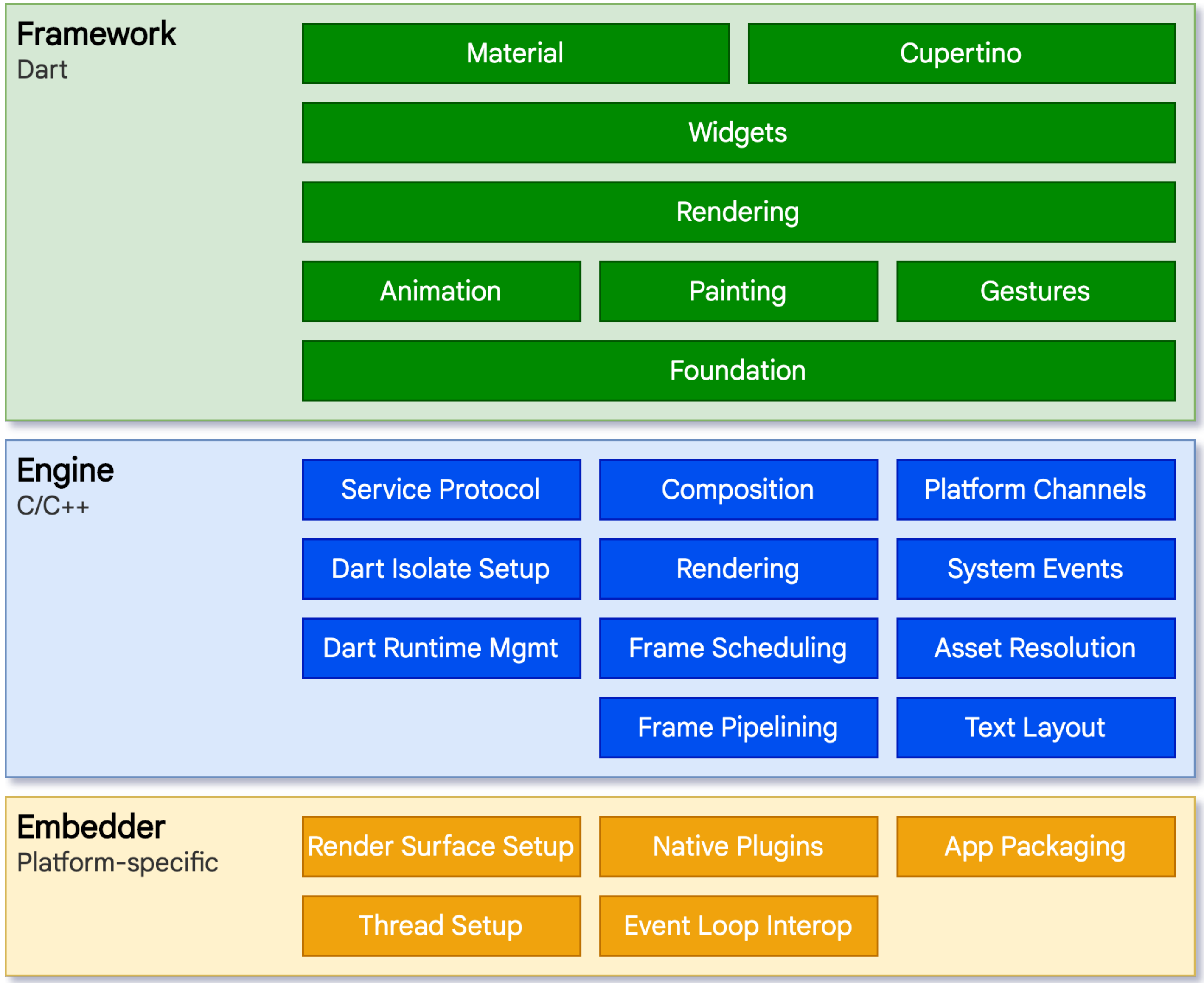
Framework

- **Foundation** and building block services such as **animation, painting** and **gestures** act as abstractions for the layers above.
- The **rendering layer** builds a tree of renderable objects and dynamically updates the layout when the widgets are changed.



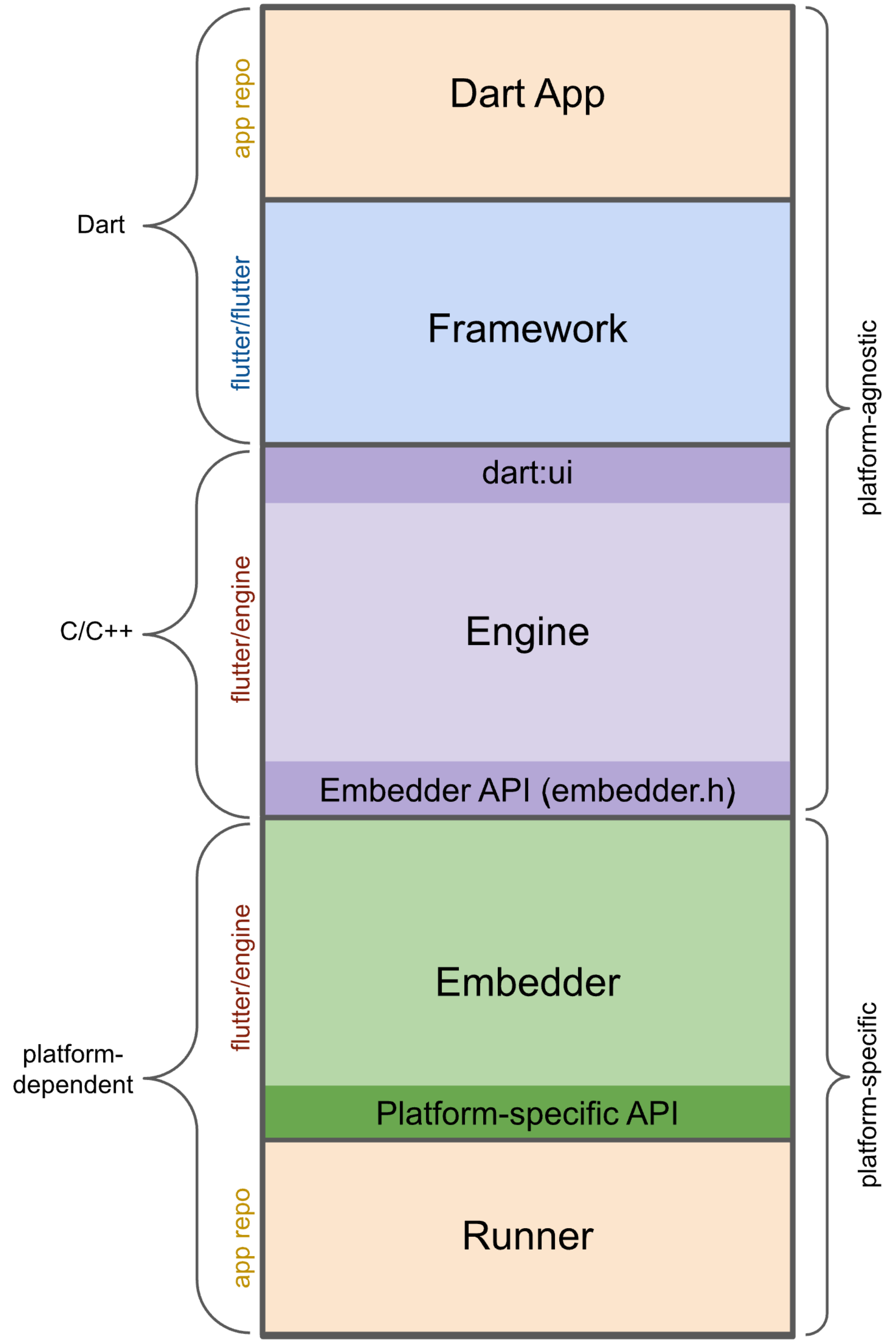
Framework

- The **widgets layer** is a composition abstraction. Each render object in the rendering layer has a corresponding class in the widgets layer.
- The **Material** and **Cupertino** lib raries offer comprehensive sets of controls that use the widget layer's composition primitives to implement the Material or iOS design languages.



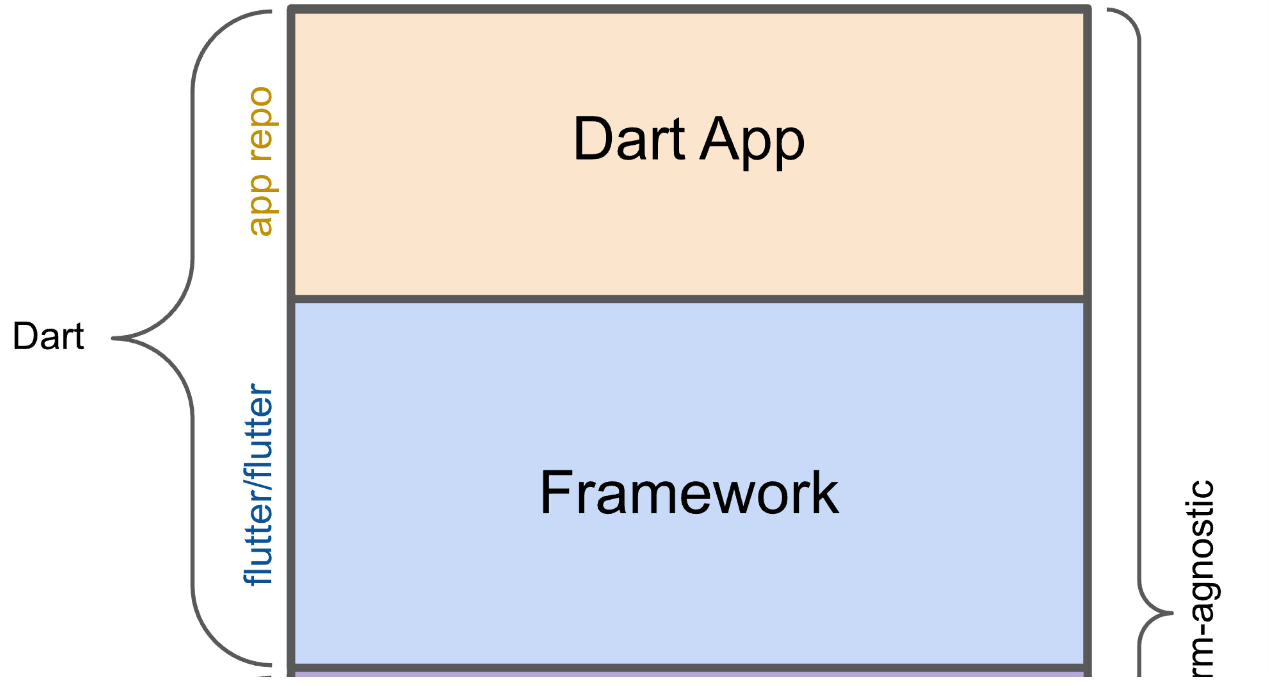
Anatomy of an App

- The following diagram gives an overview of the pieces that make up a regular Flutter app generated by `flutter create`.



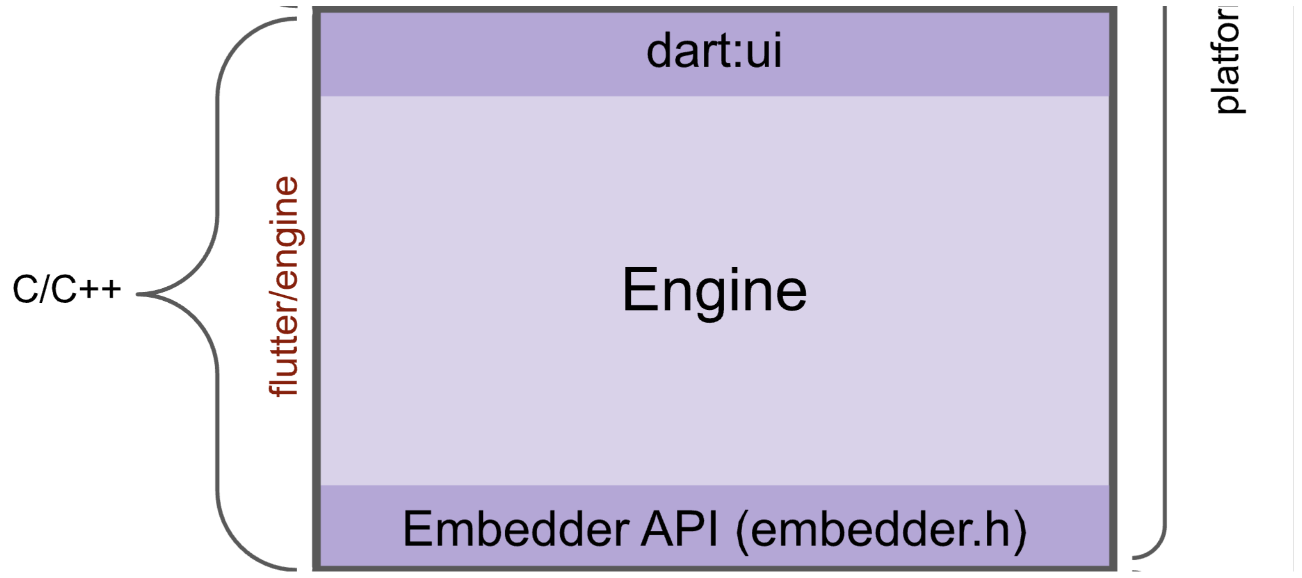
Dart

- Dart App
  - Composes widgets into the desired UI.
  - Implements business logic.
  - Owned by app developer.
- Framework
  - Provides higher-level API to build high-quality apps (for example, widgets, hit-testing, gesture detection, accessibility, text input).
  - Composites the app’s widget tree into a scene.



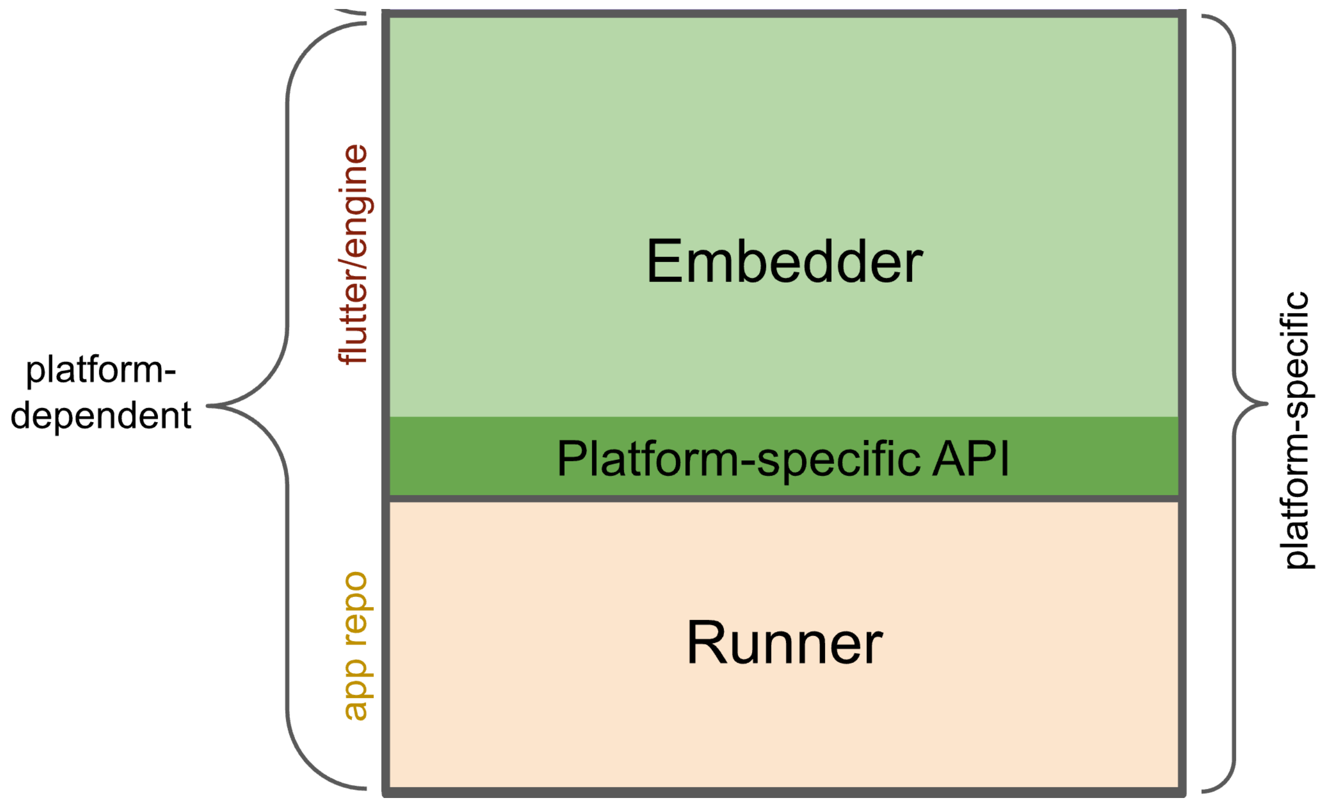
Engine

- Responsible for rasterizing composited scenes.
- Provides low-level implementation of Flutter’s core APIs (for example, graphics, text layout, Dart runtime).
- Exposes its functionality to the framework using the **dart:ui API**.
- Integrates with a specific platform using the Engine’s **Embedder API**.



Embedder

- Coordinates with the underlying operating system for access to services like rendering surfaces, accessibility, and input.
- Manages the event loop.
- Exposes platform-specific API to integrate the Embedder into apps.

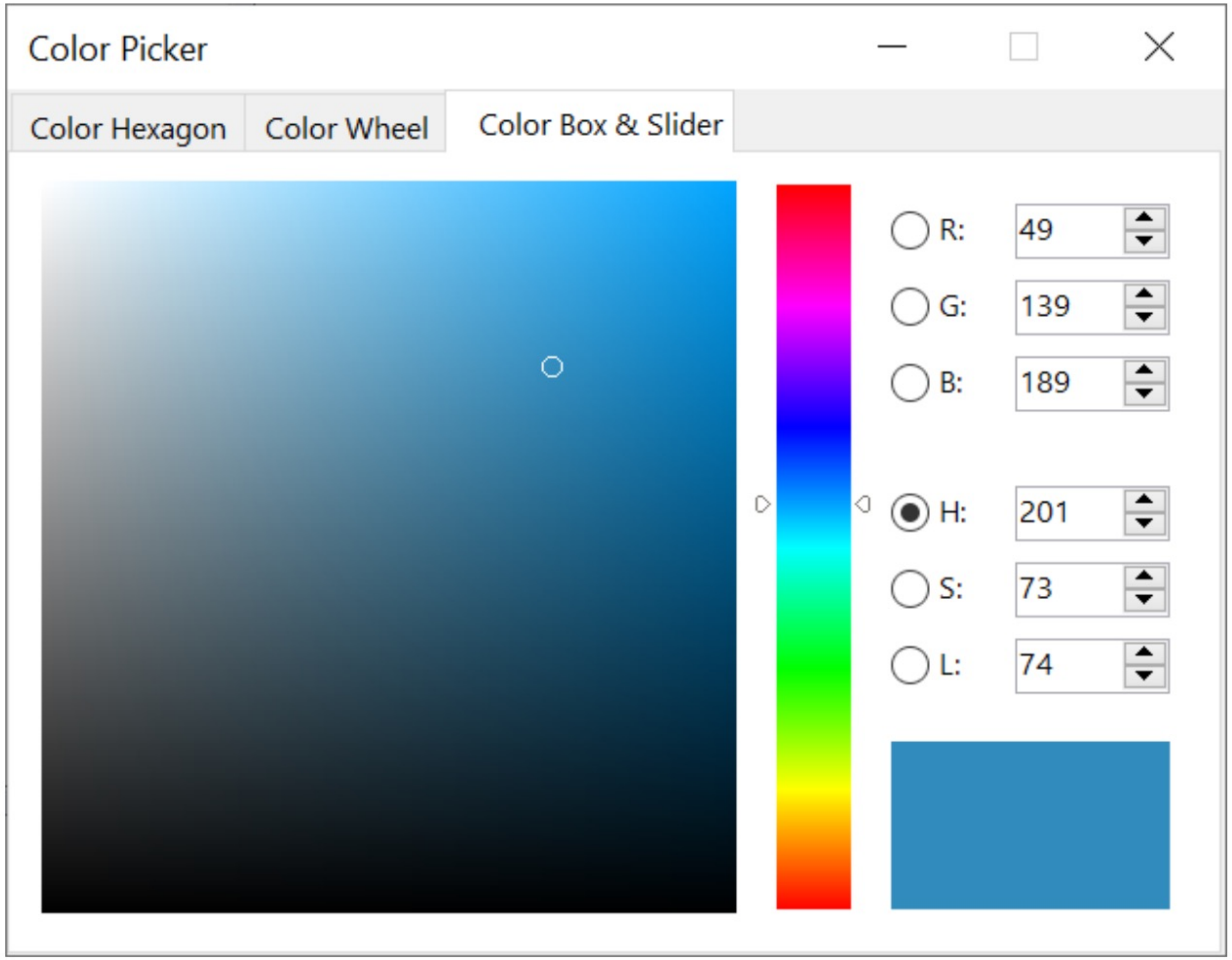


Reactive User Interfaces

- Flutter is a reactive, declarative UI framework, in which the developer provides a mapping from application state to interface state, and the framework takes on the task of updating the interface at runtime when the application state changes.
- This model is inspired by work that came from Facebook for their own React framework, which includes a rethinking of many traditional design principles.
  - https://www.youtube.com/watch?v=x7cQ3mrcKaY (10 years ago)

Color Picker UI

- Consider the following UI:
- What can be changed?



( ) R: 49
( ) G: 139
( ) B: 189
(●) H: 201
( ) S: 73
( ) L: 74

MVC Approach

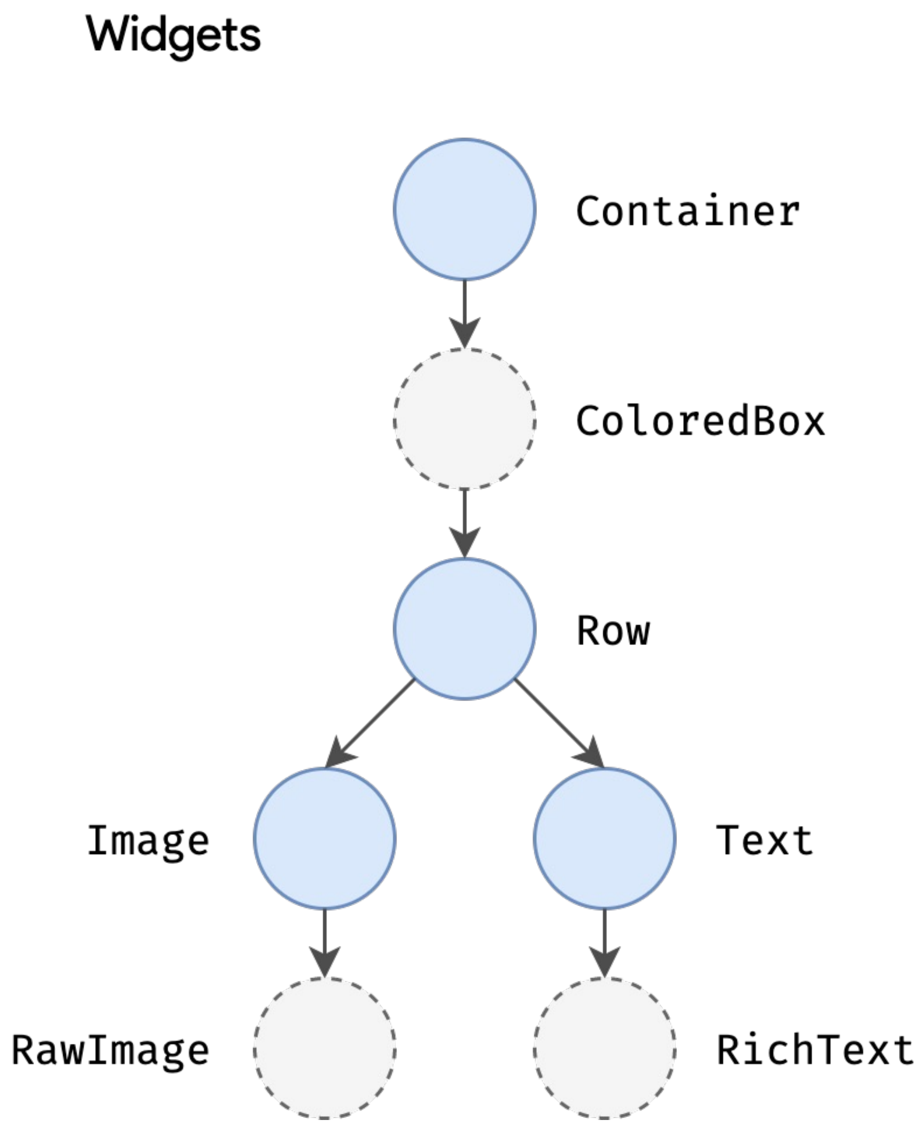
- In most traditional UI frameworks, the user interface’s initial state is described once and then separately updated by user code at runtime, in response to events.
- You push data changes to the model via the controller, and then the model pushes the new state to the view via the controller.
- However, this also is problematic, since creating and updating UI elements are two separate steps that can easily get out of sync.
  - Race condition can occur when there are many updates to model and view.

React-Style APIs

- Flutter, along with other reactive frameworks, takes an alternative approach to this problem, by explicitly decoupling the user interface from its underlying state.
- With React-style APIs, you only create the UI description, and the framework takes care of using that one configuration to both create and/or update the user interface as appropriate.
- In Flutter, widgets are represented by immutable classes that are used to configure a tree of objects. It declares its user interface by overriding the build() method, which is a function that converts state to UI: UI = f(state)

Widgets

- Widgets are the building blocks of a Flutter app's user interface, and each widget is an immutable declaration of part of the user interface.
- Widgets form a hierarchy based on composition. Each widget nests inside its parent and can receive context from the parent. This structure carries all the way up to the root widget (the container that hosts the Flutter app, typically MaterialApp or CupertinoApp)
- Apps update their user interface in response to events (such as a user interaction) by telling the framework to replace a widget in the hierarchy with another widget.



Widget State

- The framework introduces two major classes of widget: stateful and stateless widgets.
- Many widgets have no mutable state: they don’t have any properties that change over time (for example, an icon or a label). These widgets subclass StatelessWidget.
- If the unique characteristics of a widget need to change based on user interaction or other factors, that widget is stateful. These widgets subclass StatefulWidget, and because the widget itself is immutable, they store mutable state in a separate class that subclasses State.
- StatefulWidgets don’t have a build method; instead, their user interface is built through their State object.

Example of Stateful Widget

- For example, if a widget has a counter that increments whenever the user taps a button, then the value of the counter is the state for that widget. When that value changes, the widget needs to be rebuilt to update its part of the UI.
- Whenever you mutate a State object (for example, by incrementing the counter), you must call setState() to signal the framework to update the user interface by calling the State's build method again.
- Having separate state and widget objects lets other widgets treat both stateless and stateful widgets in the same way, without being concerned about losing state.

Rendering and Layout

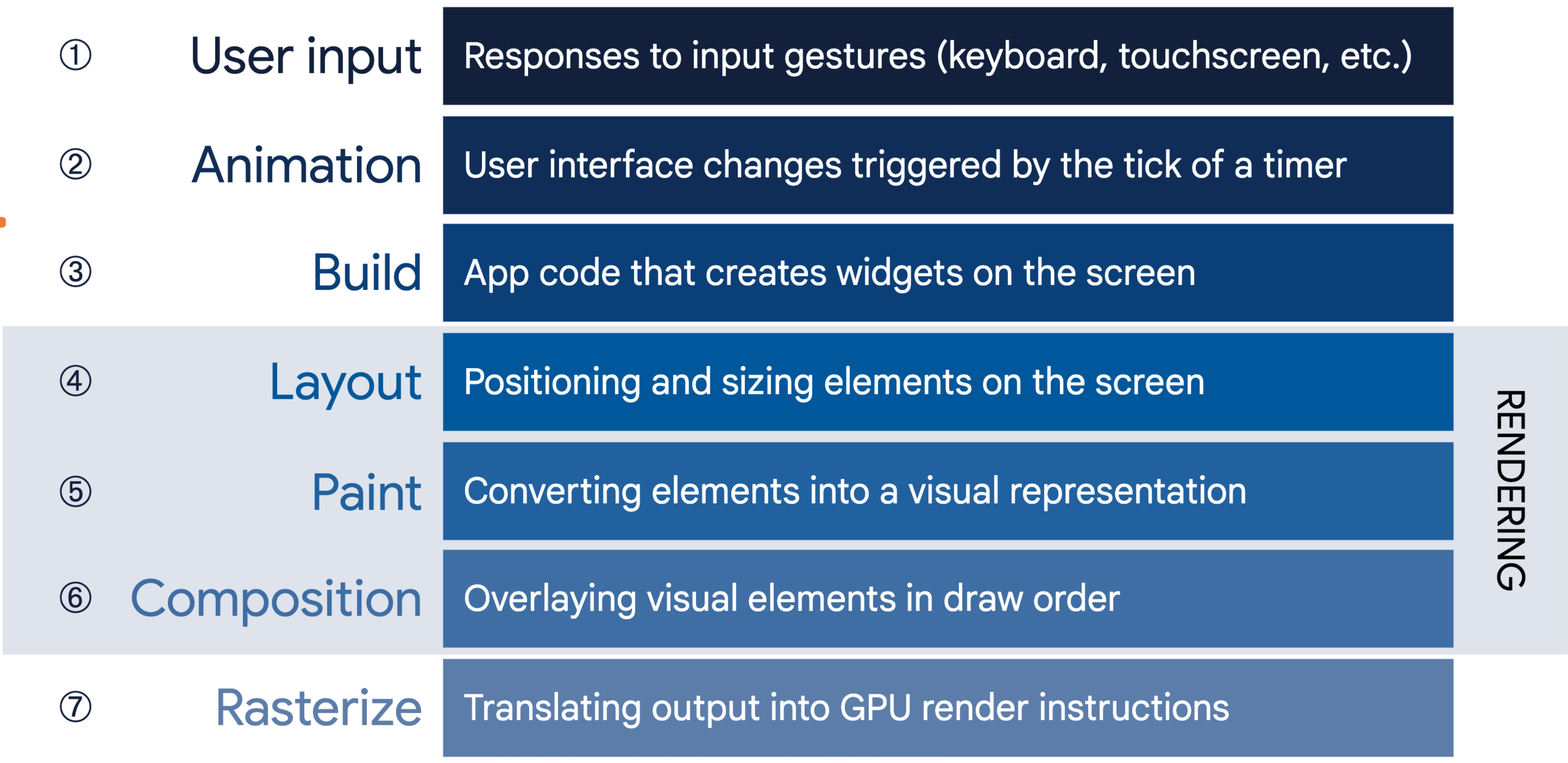
- If Flutter is a cross-platform framework, then how can it offer comparable performance to single-platform frameworks?
- Let’s look at how Android renders their app:
  - When drawing, you first call the Java code of the Android framework.
  - The Android system libraries provide the components responsible for drawing themselves to a Canvas object, which Android can then render with Skia, a graphics engine written in C/C++ that calls the CPU or GPU to complete the drawing on the device.

Flutter Rendering

- Flutter bypasses the system UI widget libraries in favor of its own widget set.
- The Dart code that paints Flutter’s visuals is compiled into native code, which uses Skia (or, in future, Impeller) for rendering.
- Flutter also embeds its own copy of Skia as part of the engine, allowing the developer to upgrade their app to stay updated with the latest performance improvements even if the phone hasn’t been updated with a new Android version.
- The same is true for Flutter on other native platforms, such as Windows or macOS.

From User Input to GPU

- The overriding principle that Flutter applies to its rendering pipeline is that **simple is fast**. Flutter has a straightforward pipeline for how data flows to the system, as shown in the following sequencing diagram:



①	User input	Responses to input gestures (keyboard, touchscreen, etc.)
②	Animation	User interface changes triggered by the tick of a timer
③	Build	App code that creates widgets on the screen
④	Layout	Positioning and sizing elements on the screen
⑤	Paint	Converting elements into a visual representation
⑥	Composition	Overlaying visual elements in draw order
⑦	Rasterize	Translating output into GPU render instructions

RENDERING

Platform Embedding

- As we’ve seen, rather than being translated into the equivalent OS widgets, Flutter user interfaces are built, laid out, composited, and painted by Flutter itself.
- The platform embedder is the native OS application that hosts all Flutter content and acts as the glue between the host operating system and Flutter.
- When you start a Flutter app, the embedder provides the entry point, initializes the Flutter engine, obtains threads for UI and rastering, and creates a texture that Flutter can write to.