basysKom

COOL
COMPACT
OSADL
ONLINE
LECTURES

Flutter for embedded systems - a new approach for industrial HMIs

Karsten Herrler

Agenda

basysKom

What is Flutter?

What is Flutter?

- First release was in 2018
- SDK from Google to develop applications for mobile, web, and desktop from a single codebase
- It's a free and open-source SDK (BSD-3 license, permissive)
- Flutter uses Dart as programming language
- Flutter uses Google's open-source Skia graphic library to render UI, instead of relying on platform-specific rendering tools
- Impeller is the new graphic library to render UI for iOS, (currently available also as preview for macOS and Android)

basysKom

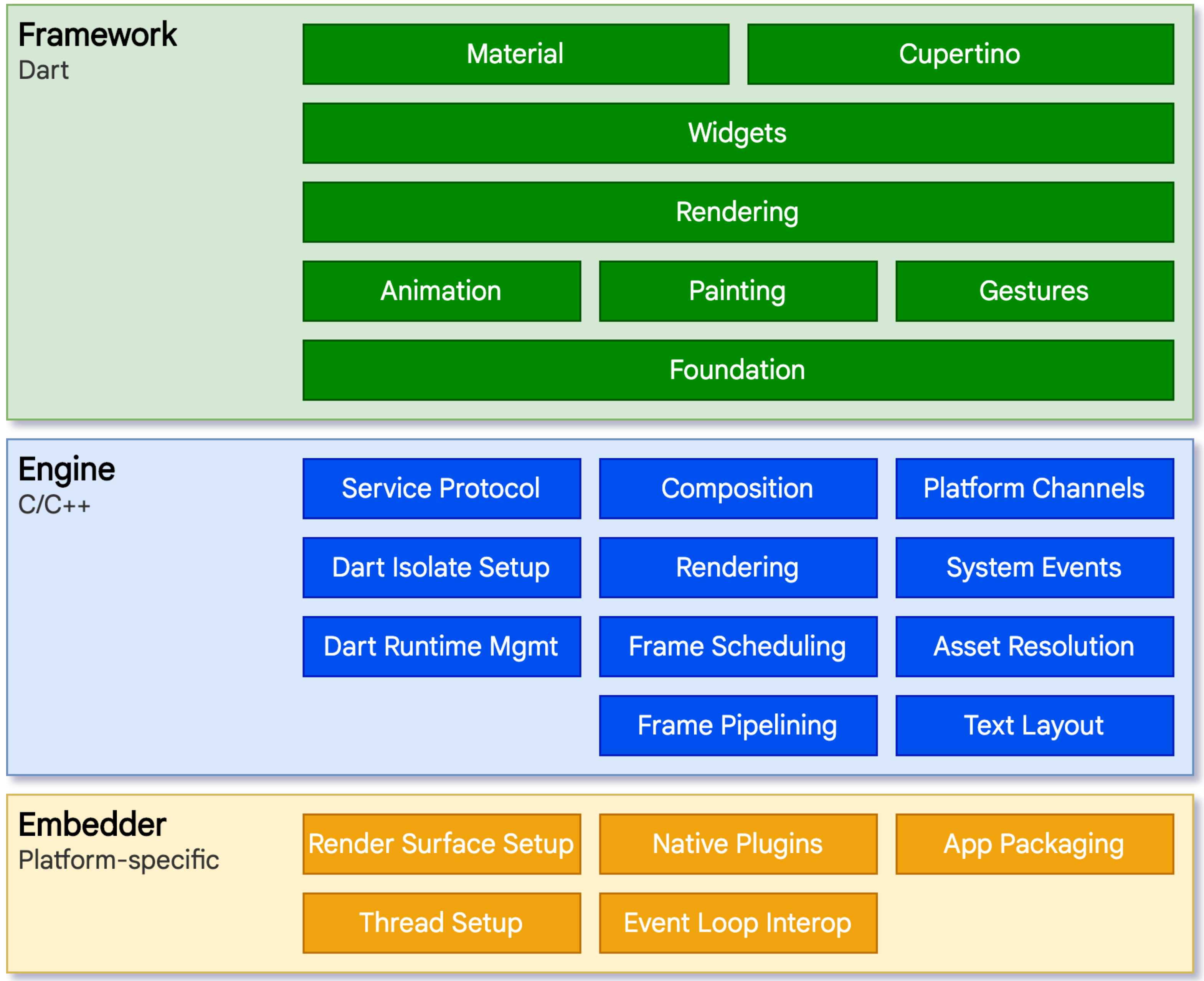
Flutter – Cross-Platform

- Originally intended for app development for Android and iOS
- Flutter provides a clean interface for platform abstraction, called an Embedder
- Since 2021, application development for Windows, macOS, Linux desktops and Web has also been supported
  - For these platforms, a standard Embedder is provided
  - For Embedded Linux there is no standard Embedder provided by Flutter
- Two popular open-source Embedders for the developing of Flutter applications for Embedded Linux are
  - Flutter Pi (MIT license) for Raspberry Pi 2, 3 and 4
  - Sony's Flutter Embedder (BSD-3-Clause license) for Embedded Linux platforms with x64 and Arm64 architectures

basysKom

Flutter architecture

- 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
- Every part of the framework level is designed to be optional and replaceable
- A platform-specific embedder provides an entry point to the underlying operating system



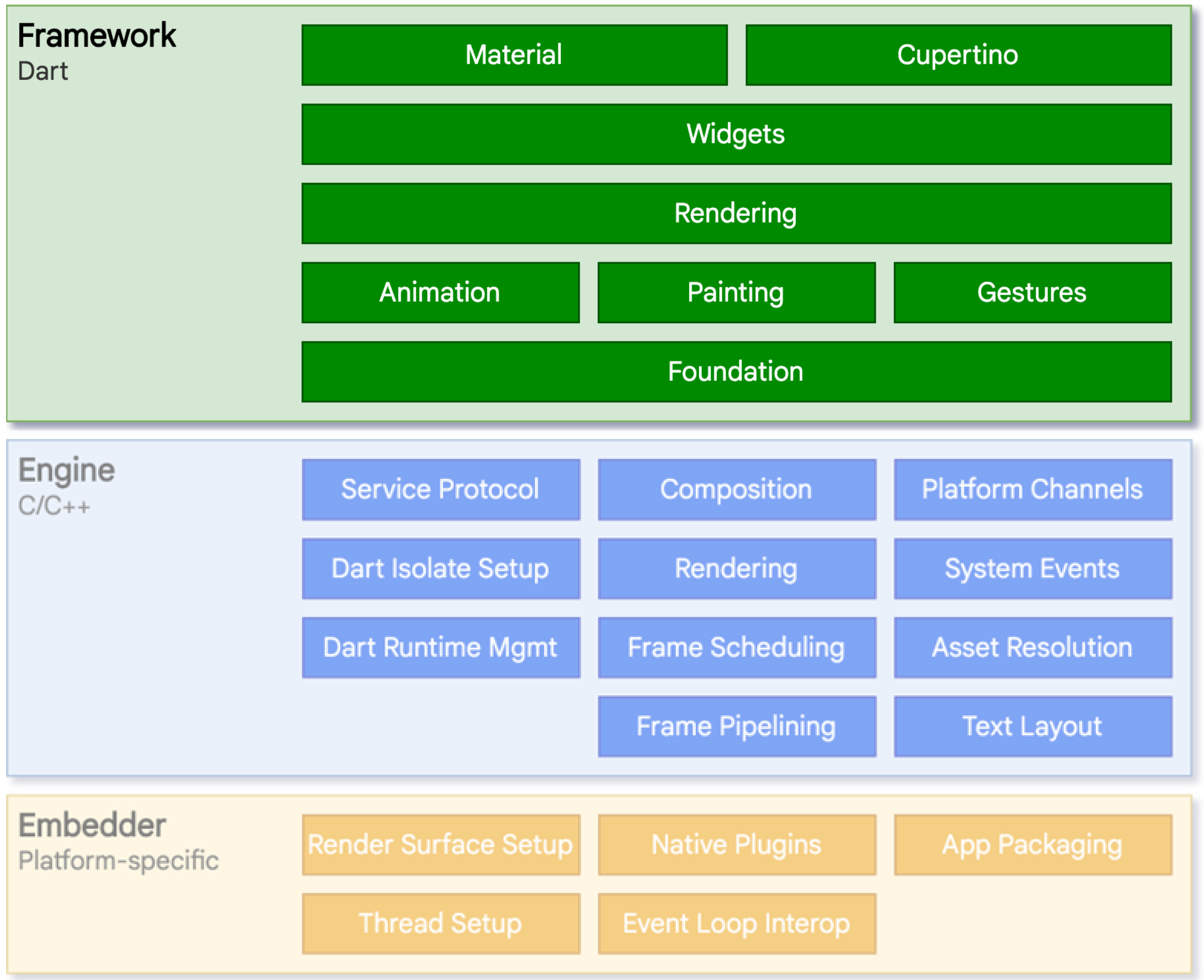
Flutter architecture, source: Flutter.dev

basysKom

Flutter architecture – Framework layer

Consists of three main components:

- **Widget layer** provides a collection of predefined widgets that can be used to create a widget tree. Developers can also create custom widgets using code
- **Rendering layer** converts the widget tree into pixels and renders them on the screen. This layer is called whenever there is any change in the widget's animation, input, or state, and it updates the layout accordingly
- **Foundation layer** provides foundational classes and building block services, such as animations and gestures, which enable developers to abstract away complex implementation details

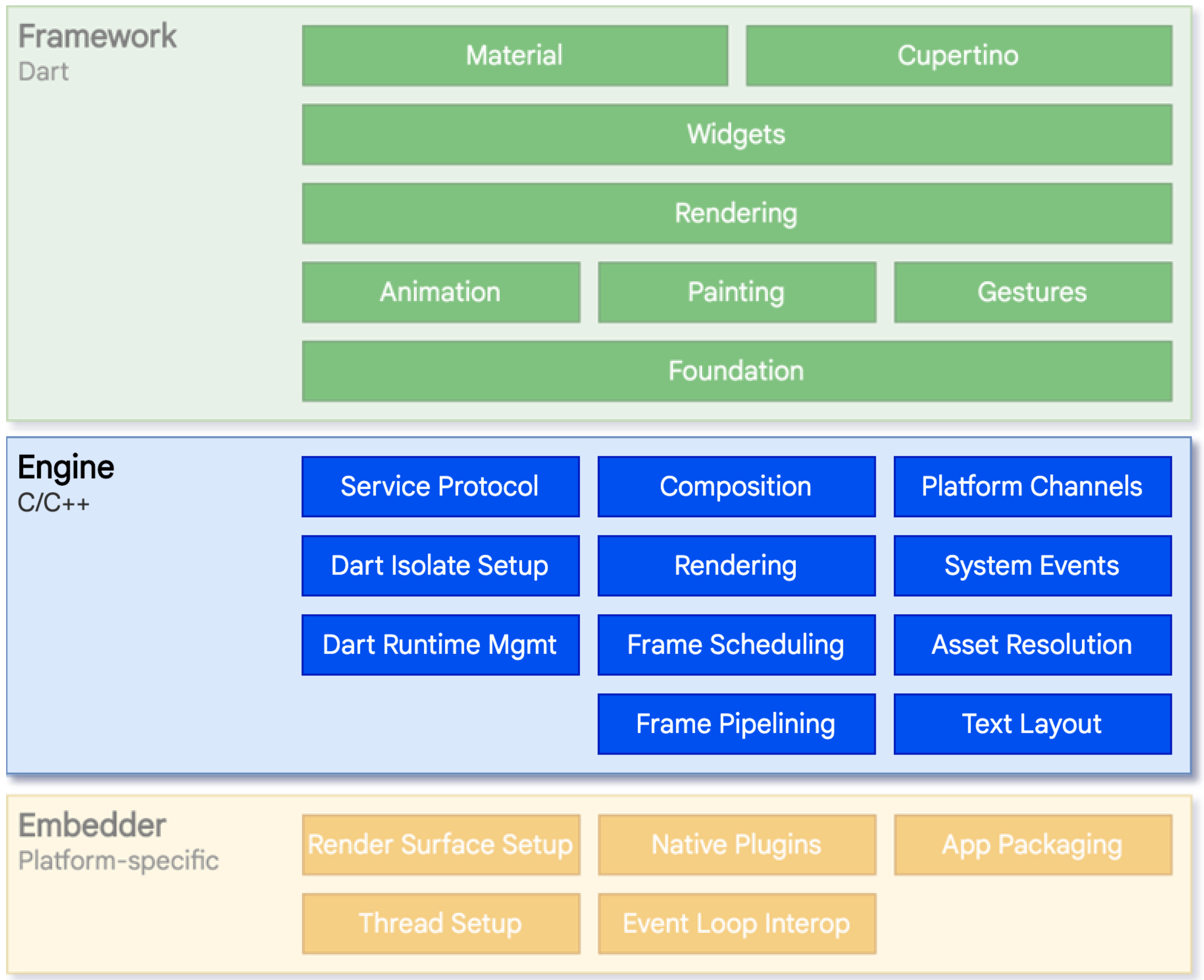


Flutter architecture, source: Flutter.dev

basysKom

Flutter architecture – Engine layer

- Core of the framework and manages the entire application lifecycle
- Is written in C++ and is designed to be platform-independent
- Provides a complete set of low-level services that handle everything from rendering and layout to animation and gesture recognition
- Includes a virtual machine (VM) called Dart VM, which executes the application code written in Dart programming language
- Optimized for high performance and provides a fast and smooth user experience, even on low-end devices

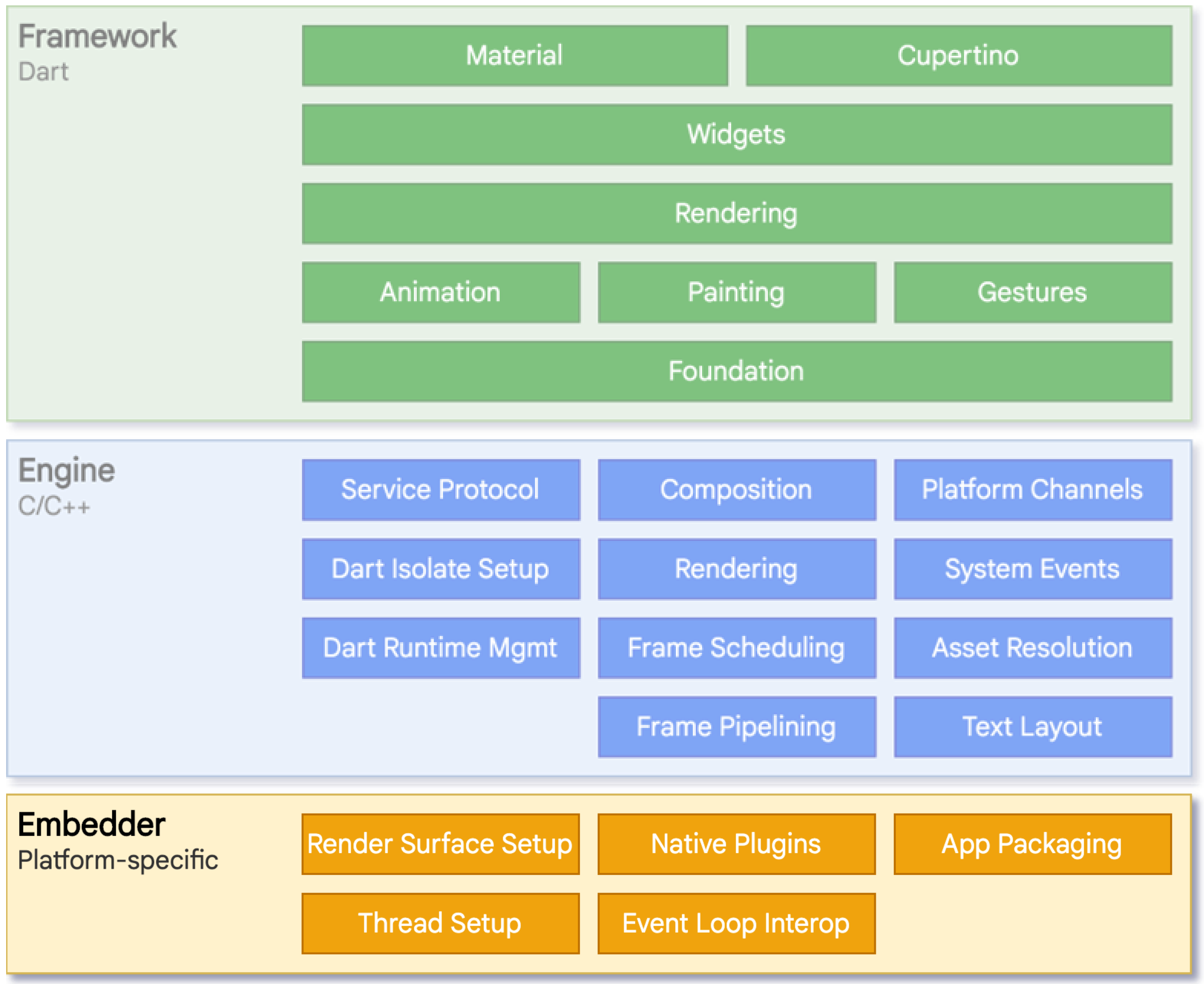


Flutter architecture, source: Flutter.dev

basysKom

Flutter architecture – Embedder layer

- Serves as the interface between the Flutter framework and the host platform
- Integrates the Flutter engine into platform-specific hosts, enabling Flutter applications to run natively on various devices
- Handles platform-specific tasks such as input handling, accessibility, rendering, event loops, and provides APIs for communication between platform-specific code and the Flutter framework
- Is written in a platform-specific language, such as Java and C++ for Android, Objective-C/Objective-C++ for iOS and macOS, and C++ for Windows and Linux



Flutter architecture, source: Flutter.dev

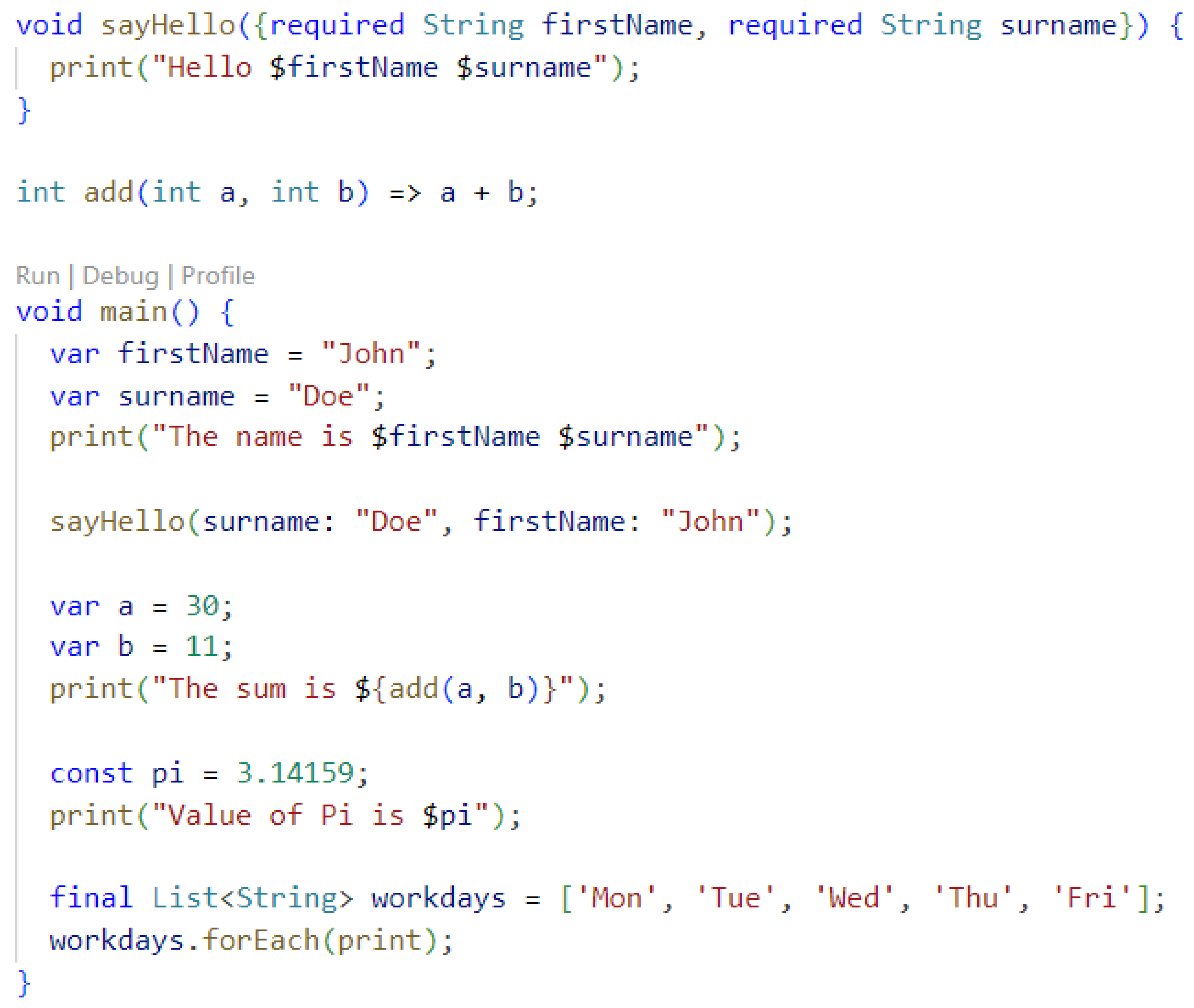
basysKom

Dart - Overview

- First release was in 2013 (developed by Google)
- Open-source
- Object-oriented (class-based)
- Strong Typing
- Garbage Collection
- Asynchronous Programming (Integrated async / wait)
- Optimized for building UIs
- Dart's sound null safety feature enhances code quality by reducing common errors

Dart - Coding

- Coding looks like C-style language
- Entry point of a Dart application is the main() method
- Dart classes only support single inheritance, but a class can have many implementations of Interfaces
- Dart has static type inference; the code treats the variable according to what it contains
- All data types are objects (including numbers), so the default value is null
- A return type of a method is not required in the method signature
- The keyword new used before the constructor for object creation is optional
- Method signatures can include a default value to the parameters passed



```dart
void sayHello({required String firstName, required String surname}) {
  print("Hello $firstName $surname");
}

int add(int a, int b) => a + b;

Run | Debug | Profile
void main() {
  var firstName = "John";
  var surname = "Doe";
  print("The name is $firstName $surname");

  sayHello(surname: "Doe", firstName: "John");

  var a = 30;
  var b = 11;
  print("The sum is ${add(a, b)}");

  const pi = 3.14159;
  print("Value of Pi is $pi");

  final List<String> workdays = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'];
  workdays.forEach(print);
}
```

Folder structure of a Flutter project

- Default structure is created by the CLI with the command: `flutter create <DIRECTORY>`
- android/ios Directory: contains native platform-specific code for Android and iOS
- lib Directory: houses the Dart code that makes up the core functionality of your Flutter app. Your app's main entry point, typically named `main.dart`, resides here. You'll create custom widgets, screens, and logic within this directory
- build Directory: contains compiled code of your project
- asset Directory: used to store static files such as images, fonts, and JSON data that your app may need
- web Directory: contains the web-specific code and assets necessary for running your app in a web browser

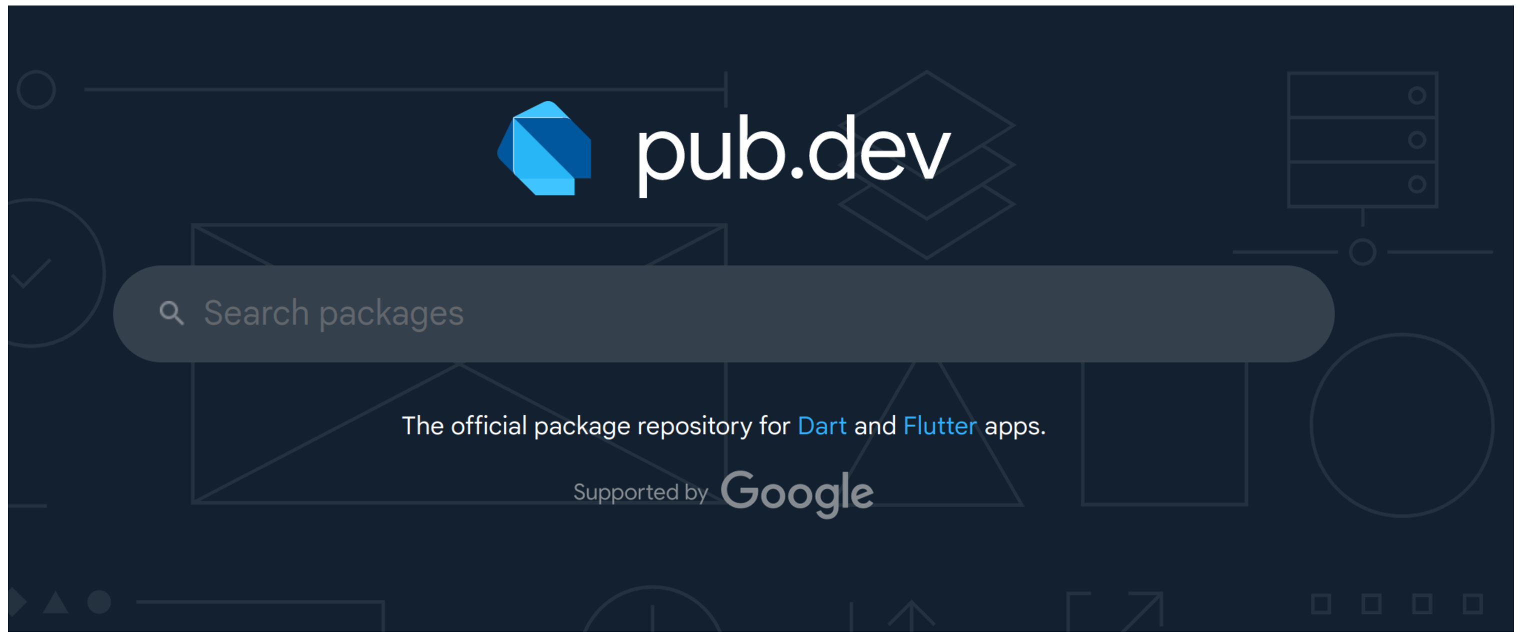
Folder structure of a Flutter project

- pubspec.yaml: manages your project's dependencies, defines metadata about your app, and specifies assets
- analysis_option.yaml: configures Dart Analyzer which will check all your codes and if your codes has any conflict with the configurations than IDE will give an error or warning.
- app.iml: used for the project structure in JetBrains IDEs, it is not specific to Flutter. This file is basically a metadata for IDEs to know how to structure the project and which folders are used for what
- pubspec.lock: contains all package version, dependency and description information used in the project

basysKom

Ecosystem - Toolset

- CLI (Command Line Interface)
  - almost everything can be done in the terminal
  - powerful tool that allows developers to interact with the Flutter framework and manage various aspects of their Flutter projects
- Package Repository pub.dev
  - shared packages contributed by other developers to the Flutter and Dart ecosystems
  - Install package with `flutter pub get`
  - Package dependencies must be added to the pubspec.yaml file

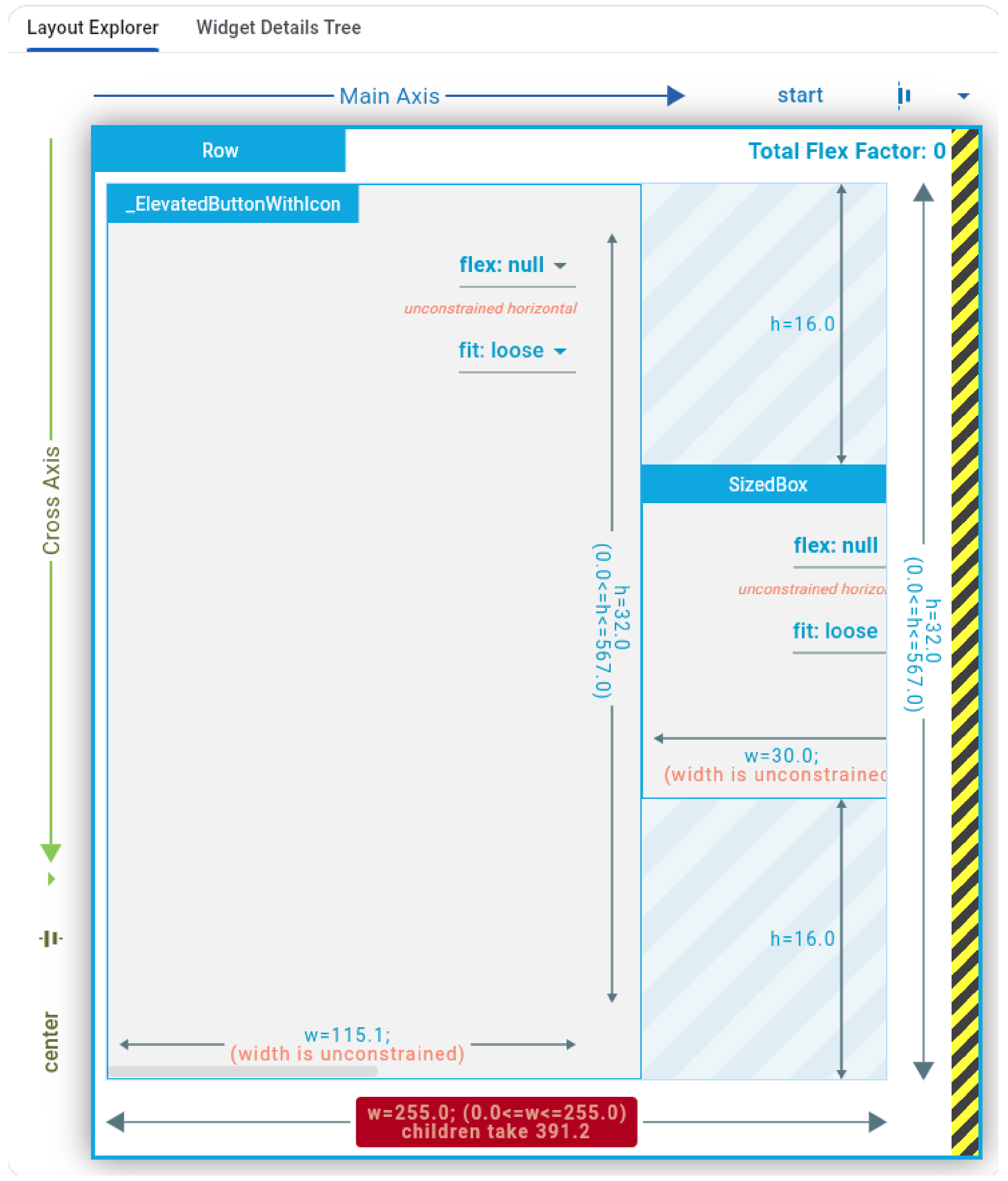


The official package repository for Dart and Flutter apps.

basysKom

Ecosystem - Toolset

- VSCode-Integration
- Hot Reload & Hot Restart
- Flutter inspector (Debugger for Layout issues)
- CPU Profiler
- Just-In-Time (JIT) compilation for debug version
- Ahead-Of-Time (AOT) compilation for release version



Flutter inspector

Hot reload in detail

Functionality

- Works by injecting updated source code files into the running Dart Virtual Machine (only in develop mode)
- Updates classes with new versions of fields and functions and rebuilds the widget tree to reflect changes quickly

Benefits

- **Efficiency:** Enables quick iteration, refinement of app design, and reducing development time significantly
- **Real-time Updates**: Changes to the code are immediately visible on the emulator or device instantly without a full application restart
- **State Preservation**: Hot Reload rebuilds the widget tree but retains the app state as it was before the changes, allowing developers to continue testing without losing their current state

Limitations

- Performance in extremely large projects with thousands of classes might have longer reload times
- Complex changes, for more complex modifications that affect the entire application or involve significant structural alterations, Hot Reload may not be as effective

Ecosystem - IDEs

- Flutter comes with a Language Server Protocol (LSP)
  - provides auto completion, go-to-definition, find all references, and more
- Popular IDEs
  - Visual Studio Code (VSCode)
  - Android Studio
  - IntelliJ IDEA
- Others
  - Kate (KDE Advanced Text Editor)
  - Emacs
  - Atom
  - Sublime Text
  - (Neo-)Vim

basysKom

Why using Flutter?

- Free and open-source (permissive licensing)
- Extensive documentation and tutorials on Flutter's site
- Cross-platform support => Single code base for multiple platforms (Android, iOS, Windows, macOS, Linux, Web)
- High performance (Flutter uses Skia/Impeller graphics engine, just like Chrome and Android)
- Wide range of customizable widgets for UI design
- Hot reload => Fast development cycle
- Easy to learn
- Offers an automated testing toolset for unit test, widget test and integration test
- Strong community support
- Easy integration of 3rd party packages from a central repository (https://pub.dev)

Companies and Apps Using Flutter

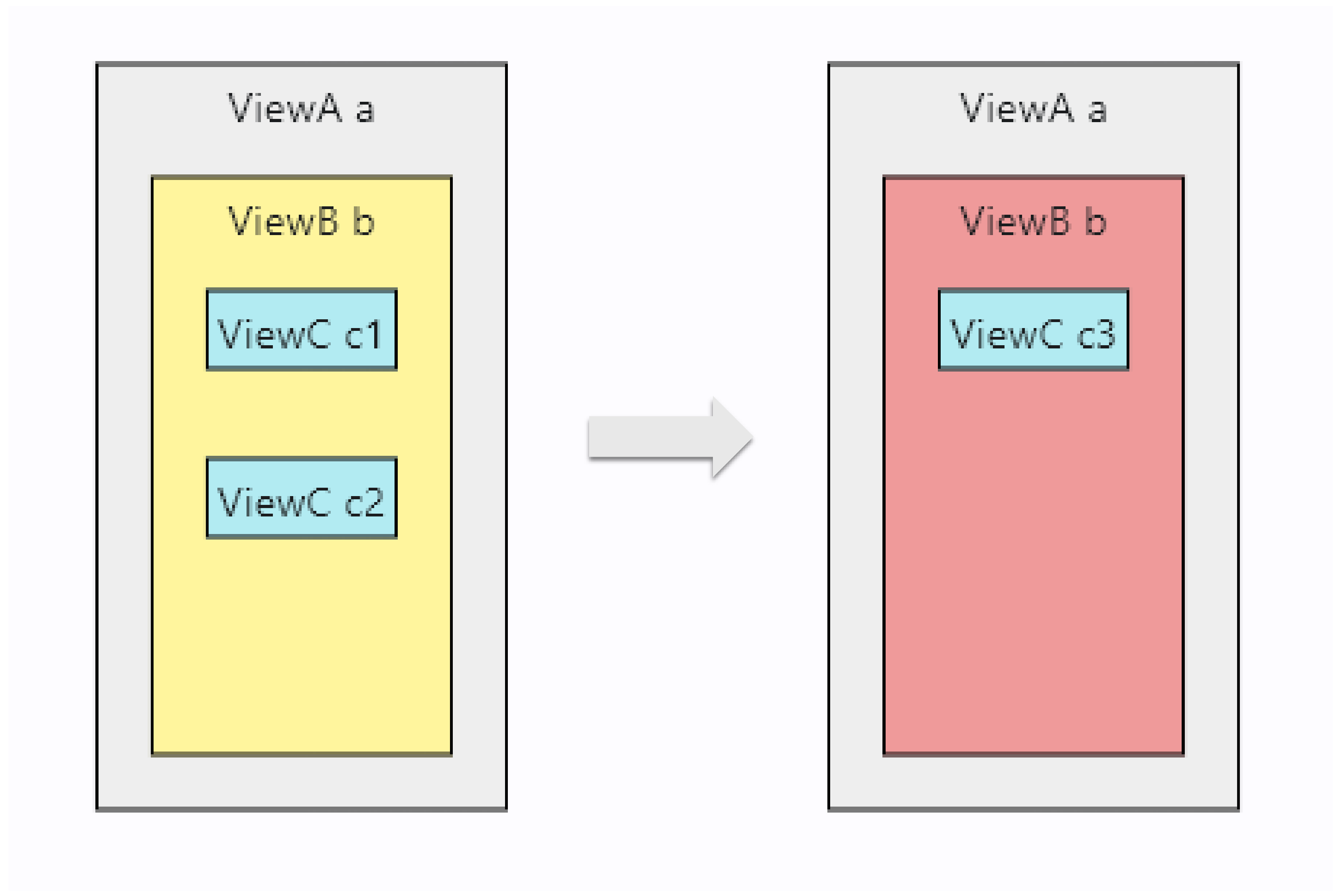
- Web/Mobile
  - Alibaba
  - Google Pay
  - eBay Motors
  - Groupon
  - Etsy
  - Superlist
- Embedded Systems
  - Toyota
  - BMW

basysKom

Principles of Flutter

Flutter – Think declaratively

```
// Imperative style
b.setColor(red)
b.clearChildren()
ViewC c3 = new ViewC(...)
b.add(c3)


// Declarative style
return ViewB(
  color: red,
  child: const ViewC(),
);
```



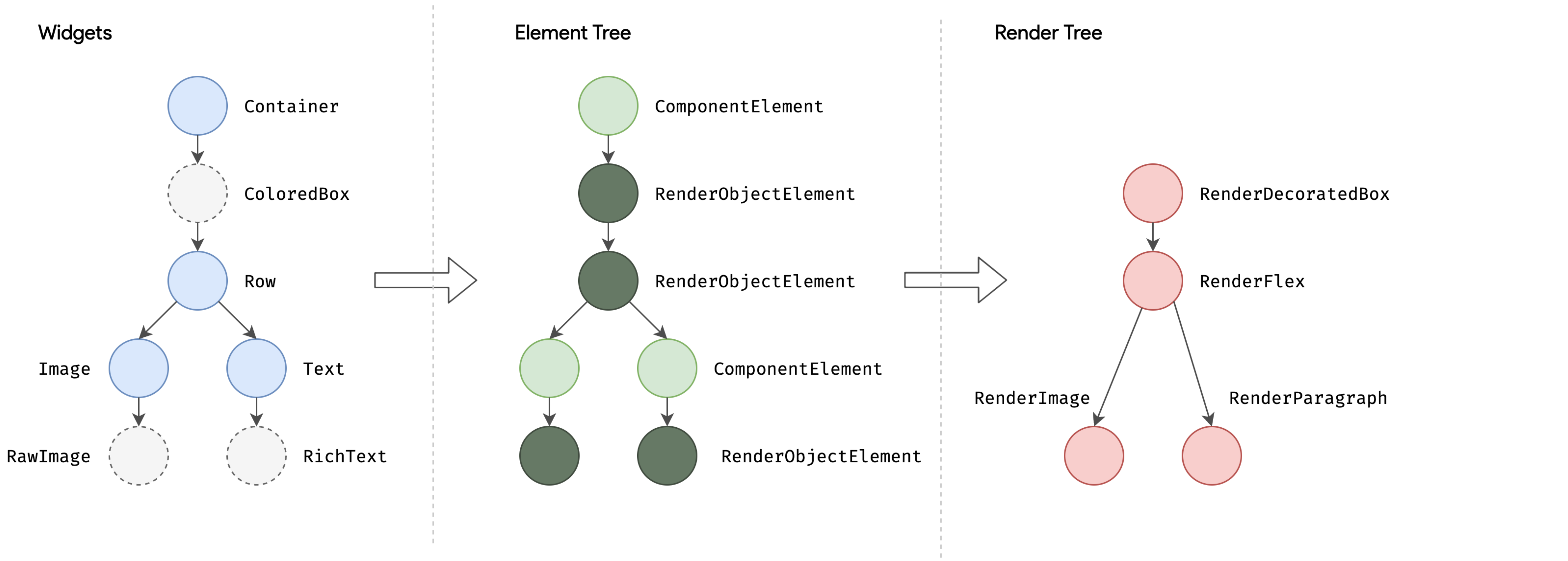
Flutter – Think declaratively

- A change of the state, triggers a redraw of the user interface
- There is only one code path for any state in the UI
- UI is only described once for any given state
- Pros
  - Code readability and conciseness
  - Declarative programs are generally simpler, safer, and more maintainable
  - Immutable objects are easier to work with as they can only be in one state, reducing the risk of bugs related to mutable state
- Cons
  - It can be more difficult to understand for beginner with experience in imperative programming
  - Declarative approach might not seem as intuitive as the imperative style

Widget-based rendering system

- Flutter apps start with a single root widget, building the entire UI as a tree of widgets.
- In Flutter everything is a widget and serves as the basic building blocks for UI elements.
- Flutter utilizes a Widget Tree, Element Tree, and Render Tree to efficiently render UI changes
- These three concepts work together to ensure that only the necessary parts of the UI are updated without rebuilding the entire interface
- The RenderObject is responsible for laying out UI elements based on constraints and painting the visual representation of widgets on the screen
- It participates in the painting pipeline to render the UI efficiently and manages layers for composing the final UI

Widget-based rendering system

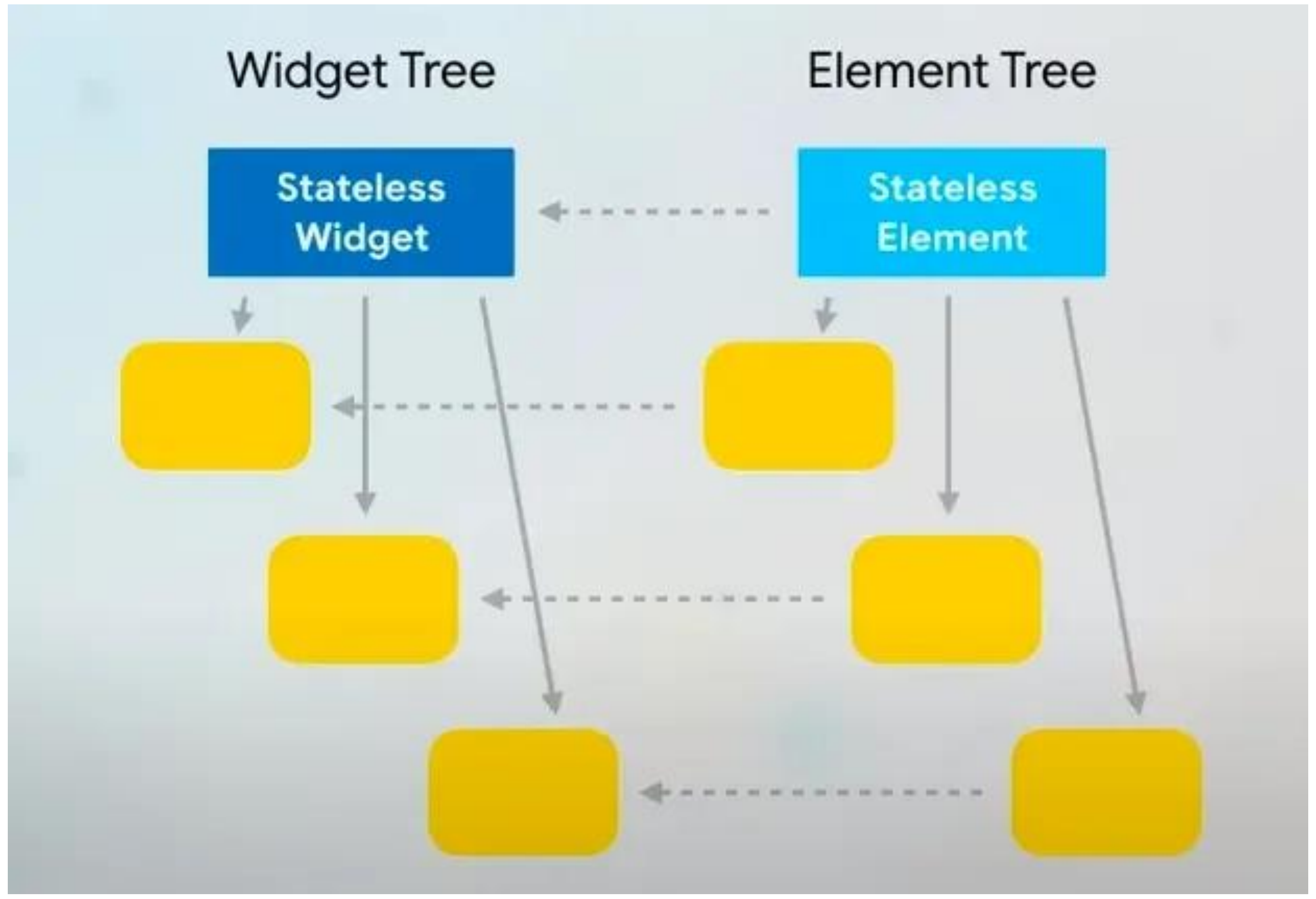


Layout and rendering, source: Flutter.dev

StatelessWidget

- Does not maintain internal state and are immutable once built
- Cannot trigger rebuilds and remain static throughout the application's runtime
- Does not react to user interactions
- Ideal for displaying static content that does not change during runtime like texts, icons, or images
- Lightweight and efficient due to their static nature
- Single class

```
class GreenFrog extends StatelessWidget {
  const GreenFrog({ super.key });

  @override
  Widget build(BuildContext context) {
    return Container(color: const Color(0xFF2DBD3A));
  }
}
```



StatefulWidget class, source: Flutter.dev

basysKom

State management

Internal classes

- Stateful Widget
- InheritedWidget / InheritedModel

3rd party packages

- Provider
- BLoC
- Riverpod
- GetX
- MobX
- Redux
- Fish-Redux

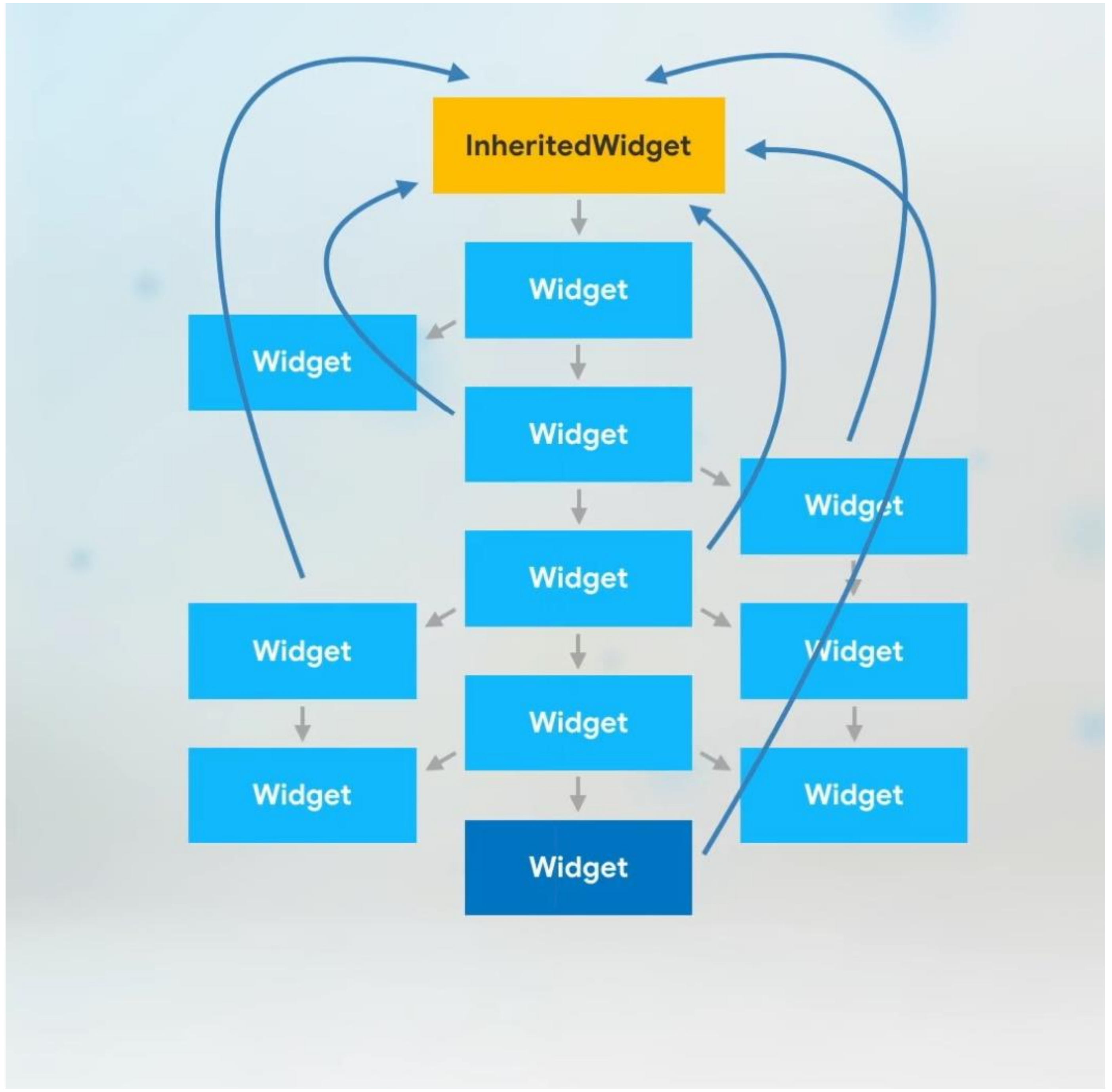
StatefulWidget

- Can update their internal state over time
- Can rebuild when their state changes, allowing for dynamic updates
- Respond to user interactions and can change appearance or behavior based on various factors
- Suitable for handling dynamic data and widgets by managing mutable state
- Slightly more overhead compare to stateless widgets due to managing mutable state
- Two classes: Widget + State
- Whenever you mutate a state object, you must call setState() to signal the framework to update the user interface by calling the State’s build method again

```
class MyCounter extends StatefulWidget {
  const MyCounter({super.key});

  @override
  State<MyCounter> createState() => _MyCounterState();
}

class _MyCounterState extends State<MyCounter> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Text(
          '$_counter',
          style: Theme.of(context).textTheme.headlineMedium,
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

InheritedWidget

- Useful for passing data down the widget tree efficiently without the need for prop drilling
- It's immutable and stateless but allows widget to subscribe to changes in the inherited data
- Simplifies data transfer across the widget tree by providing a centralized way to share data
- Useful for sharing configuration settings, themes, or application-wide data across multiple widgets without passing them explicitly through constructors
- Limited reusability and scalability, especially as the app grows in complexity
- Suitable for small apps due to its limitations in handling larger and more complex applications



InheritedWidget class, source: Flutter.dev

basysKom

Testing and Debugging

Testing Tools

- Flutter Testing library
  - Built-in test library
  - flutter_test package for writing unit tests and widget tests
  - integration_test package for writing integration tests
- Mockito
  - Popular mocking framework for writing unit tests
  - Mock objects are objects that simulate the behavior of real objects
  - Useful for testing code that depends on external services or APIs

Testing Tools

- Flutter Driver
  - Extension by the flutter_driver package
  - Tool for writing integration tests that interact with a running app on a real device or emulator
  - Integration tests are used to verify that different parts of the app work together correctly
  - Useful for testing user interfaces and navigation
  - Provides APIs for interacting with the app’s UI elements, such as tapping buttons, filling out forms, and scrolling through lists
  - Also provides APIs for interacting with the device, such as taking screenshots and retrieving device information
- Code coverage tools
  - Run "flutter test --coverage"
  - Third-party: lcov, codecov

basysKom

Debugging Tools

- Built-in source-level debugger for VSCode, Android Studio and IntelliJ IDEA
- Flutter DevTools
  - UI Inspector: Inspect UI Layout and state of a Flutter app
  - Performance Diagnostics: Diagnosing UI jank performance issues, CPU profiling, network profiling, memory issue debugging, and analyzing code and app size for optimization
  - General Log Information: Provides access to general log and diagnostics information about running Flutter or Dart command-line apps
- Debug-Mode
  - Enables Debugging, Hot Reload, Assertions and Service extensions
- Profile-Mode
  - For performance analysis
  - Similar to Release-Mode, but enables some Service extensions, tracing and source-level debugging can connect to the process
- flutter_gdb: Remote debugging of the Flutter engine running within an Android app process with GDB

basysKom

Flutter linter

- Maintained by the Dart team (https://pub.dev/packages/flutter_lints)
- Helps developers maintain code quality by enforcing coding standards and identifying potential issues
- Provides a set of rules and recommendations to ensure code consistency, readability, and maintainability
- Helps developers with best practice compliance and avoidance of common development pitfalls
- Developers can integrate the Flutter Linter into their projects by configuring the `analyis_options.yaml` file with the desired lint rules
- By running the linter as part of the development process, developers can identify and address issues early on, resulting in cleaner and more efficient code
- Benefits:
  - **Code Consistency**: The linter promotes consistent coding styles and practices across Flutter projects, making it easier for developers to collaborate and maintain codebases
  - **Error Prevention**: By highlighting potential issues and enforcing best practices, the linter helps prevent common errors and improves code quality
  - **Performance Optimization**: The linter rules include performance optimizations that can help developers write more efficient Flutter code for better app performance

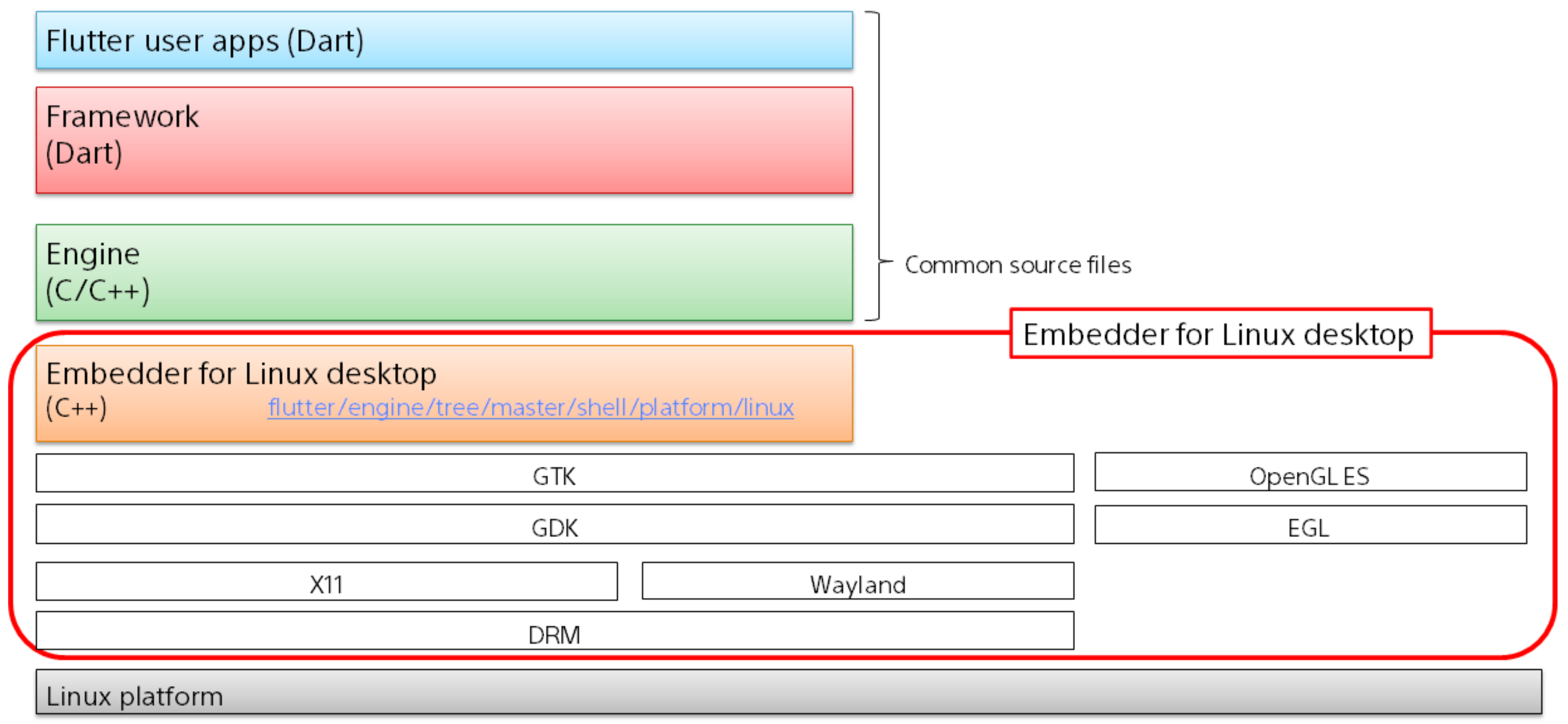
basysKom

Flutter on Embedded Linux

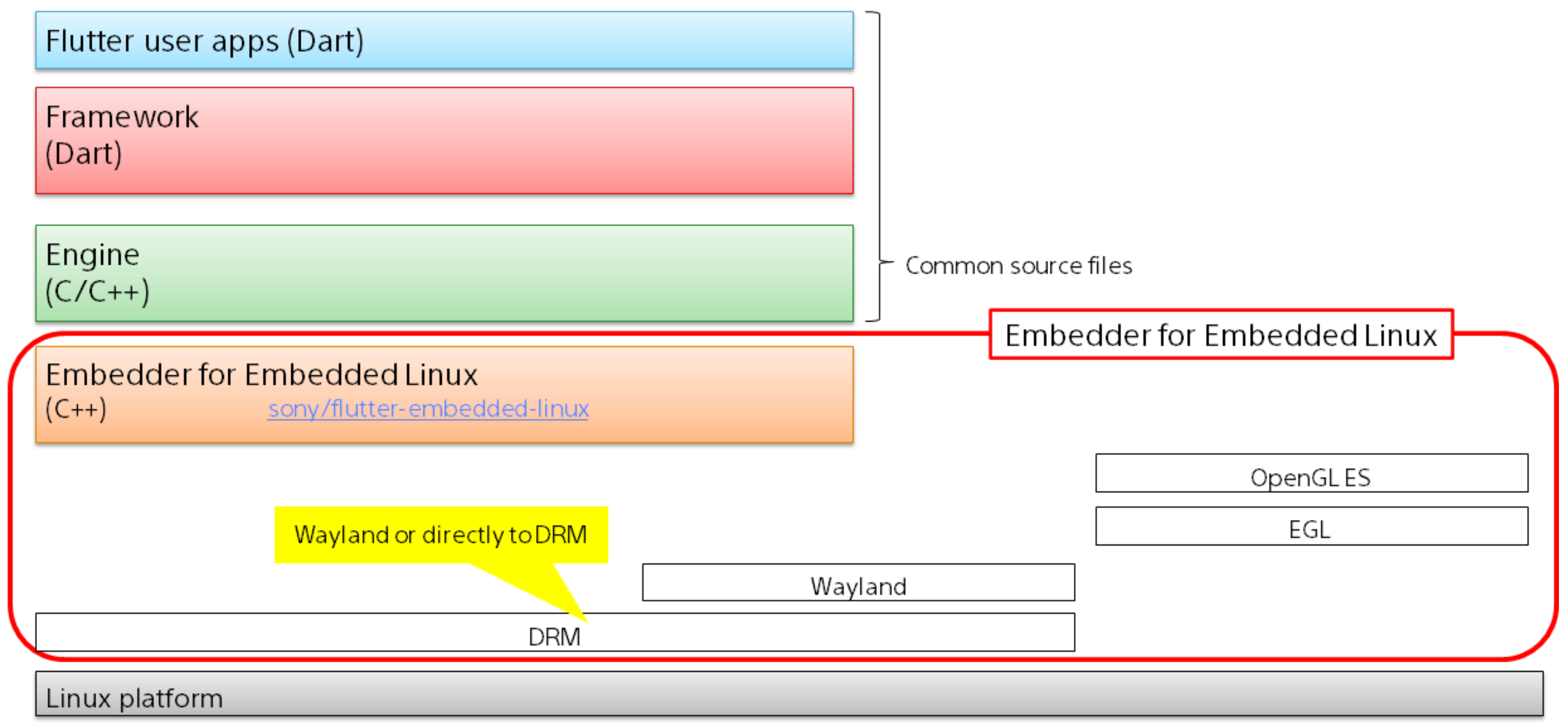
Why did we need to create a new embedder for Embedded Linux?

Requirements for embedded systems are not the same as those for desktop systems

- Flutter desktop for Linux uses GTK/GDK and X11 => requires a lot of dependent libraries
- Embedded devices are generally more limited in CPU, physical memory, storages => only use as many libraries as necessary
- Embedder for Embedded Linux doesn't use GTK/GDK and X11 at all, it uses Wayland or DRM backends



Flutter desktop for Linux
source: https://github.com/sony/flutter-embedded-linux/wiki/What%27s-the-difference-between-Linux-desktop-and-Embedded-Linux


Flutter user apps (Dart)
Framework (Dart)
Engine (C/C++)
Common source files
Embedder for Embedded Linux
Embedder for Embedded Linux (C++) sony/flutter-embedded-linux
Wayland or directly to DRM
Wayland
DRM
OpenGL ES
EGL
Linux platform

Flutter for Embedded Linux
source: https://github.com/sony/flutter-embedded-linux/wiki/What%27s-the-difference-between-Linux-desktop-and-Embedded-Linux

Flutter on Raspberry Pi

- GitHub project: flutter-pi (https://github.com/ardera/flutter-pi)
- Good choice for developing Flutter applications on Raspberry Pi
- Light-weight Flutter Engine Embedder for Raspberry Pi
- Also runs without X11 => boot into command-line is sufficient
- flutter-pi is tested on a Raspberry Pi 4 2GB
- Known working boards
  - Pi 2, 3 and 4 (even the 512MB models)
  - Pi Zero 2 (W)
- Should work on other Linux platforms, with following conditions
  - support for hardware 3D acceleration, more precisely support for kernel-modesetting (KMS) and the direct rendering infrastructure (DRI)
  - CPU architecture is one of ARMv7, ARMv8, x86 or x86 64bit
- flutter-pi won't work on Pi Zero or Pi 1

How to run an application with flutter-pi

Detailed instructions are available under
https://github.com/ardera/flutter-pi?tab=readme-ov-file#-building-flutter-pi-on-the-raspberry-pi

1. Build and install the flutter-pi project and its dependencies on your Raspberry Pi
2. Prepare your host development machine (You can't use your Raspberry Pi as your development machine)
   - Install Flutter SDK (at least version 3.10.5)
   - Install flutterpi_tool: `flutter pub global activate flutterpi_tool`
3. Build the application bundle on your host development machine using the flutterpi_tool and deploy the bundle using rsync or scp to the Raspberry Pi
4. Running your application with flutter-pi on your Raspberry Pi

Performance

Graphical Performance

- Most of the Apps inside the Flutter SDK examples are smooth (50-60 fps) on the Pi 4 2GB and Pi 3 A+.

Touchscreen latency

- Touchscreen driver in the raspbian kernel repeatedly polls the touchscreen at 60 Hz
  => average delay of 17 ms (minimum 0 ms, maximum 34 ms)
- Due to a bug, the Linux side only polls at 25Hz, which makes touch applications look terrible
  => Dragging something in a touch application appears very sluggish

Useful Packages for flutter-pi

Name	Description
flutterpi_tool	Tool to make developing and distributing apps for flutter-pi easier.
flutterpi_gstreamer_video_player	Official video player implementation for flutter-pi.

Flutter for Embedded Linux (eLinux)

- GitHub project: flutter-embedded-linux (https://github.com/sony/flutter-embedded-linux/)
- More general Flutter embedder optimized for embedded Linux systems, not just Raspberry Pi
- The goal is to integrate the embedder into the Flutter engine as the standard embedded Linux embedder
- Features
  - More light-weight than Flutter desktop for Linux (Not using X11 and GTK)
  - Minimal dependent libraries
  - Main target is arm64 devices
- Embedded software development
  - Cross-building from x64 to arm64 support
  - Install/uninstall/debug to remote target devices
- Flutter plugins support
- Display backends: Wayland, X11, Direct rendering module (DRM), Generic Buffer Management (GBM), EGLStream for NVIDIA devices
- Keyboard, mouse and touch inputs support

Useful packages for the Sony Embedder

Name	Description
flutter-embedded-linux	Flutter Embedder for eLinux supports x86 and Arm64 (aarch64, ARMv8) architectures on which supports either Wayland backend or DRM backend
flutter-elinux	Flutter tools for eLinux non-official extension to Flutter SDK to build and debug Flutter apps for embedded Linux devices
flutter-elinux-plugins	Flutter Plugins for eLinux (e.g. video player, camera)
meta-flutter	Yocto recipes of eLinux embedding for Flutter

basysKom

Tested devices

Board / SoC	Vendor	OS /BSP	Backend
Jetson Orin Nano	NVIDIA	JetPack 5.1.1	Wayland / X11
Jetson Nano	NVIDIA	JetPack 4.3	Wayland / X11 / DRM
Raspberry Pi 5	Raspberry Pi Foundation	Raspberry Pi OS	Wayland / X11 / DRM
Raspberry Pi 4 Model B	Raspberry Pi Foundation	Ubuntu 20.10	Wayland / DRM
i.MX 8M Quad EVK	NXP	Sumo (kernel 4.14.98)	Wayland
i.MX 8M Mini EVKB	NXP	Zeus (kernel 5.4.70)	Wayland
RB5 Development Kit	Qualcomm	Ubuntu 18.04.05	DRM
Desktop (x86_64)	Intel	Ubuntu 20.04	Wayland / X11 / DRM
QEMU (x86_64)	QEMU	AGL (Automotive Grade Linux) koi	Wayland / DRM

Useful packages for Embedded Linux

Name	Description
flutter_gpiod	GPIO control support for dart/flutter, uses kernel interfaces directly for more performance.
linux_serial	Serial Port support for dart/flutter, uses kernel interfaces directly for more performance.
linux_spidev	SPI bus support for dart/flutter, uses kernel interfaces directly for more performance.
dart_periphery	All-in-one package GPIO, I2C, SPI, Serial, PWM, Led, MMIO support using c-periphery.
charset_converter	Encode and decode charsets using platform built-in converter.

basysKom

Flutter in comparison to Qt Quick

What is Qt Quick?

- is the current top dog for doing Embedded Linux HMIs
- is part of the Qt framework
  - Qt is widely used in a variety of applications, including desktop software, mobile apps, embedded systems, and even some web applications
  - Qt is a powerful and versatile application framework that enables developers to create multi-platform software applications with a consistent user experience across different operating systems and devices
- is a user interface technology that allows developers to create fluid and dynamic user interfaces and cross-platform deployment
- uses a declarative language called QML (Qt Markup Language)
- provides a QML API for UI development and a C++ API for extending QML applications, allowing for a smooth integration between the declarative UI and the underlying application logic
- can be used in conjunction with other Qt modules to create complete applications
- is optimized for hardware rendering and provides high runtime performance

Flutter in comparison to Qt Quick

Pros:

- Flutter is free with a permissive FOSS license
- Hot reload makes the development and debugging applications less time-consuming
- Dart is easier to learn, C++ has a steep learning curve
- Automatic memory management avoids memory leaks and other errors
- Rich set of predefined widgets and extensive capabilities for creating complex custom widgets
- Flutter has better support for 3rd party packages in a central repository (https://pub.dev)
- Flutter is better optimized for mobile development than Qt
- Google is a big company with a big community

Flutter in comparison to Qt Quick

Cons:

- Flutter applications tend to have larger sizes
- C++ has higher performance and better efficiency than Dart
- C++ delivers the best results for extensive calculations that require significant CPU utilization
- Qt is better integrated into the platform, so it's a little bit easier to communicate with the native platform and the external devices
- Qt is more mature and stable framework with a long history
- Qt has LTS versions
- Qt has a large number of built-in features
- Internationalization is better solved in Qt (Qt Linguist, lupdate, lrelease)
- The Qt company is a European company and is more responsive to customer needs than Google
- Google tends to close projects quickly

basysKom

Useful links

basysKom

Useful links for Flutter and Dart

- Flutter documentation
  https://docs.flutter.dev/
- Flutter YouTube channel
  https://www.youtube.com/flutterdev
- Flutter architectural overview
  https://docs.flutter.dev/resources/architectural-overview
- Performance best practices
  https://docs.flutter.dev/perf/best-practices
- Dart documentation
  https://dart.dev/guides
- Development of a first Flutter app
  https://codelabs.developers.google.com/codelabs/flutter-codelab-first#0

basysKom

Useful links for Embedded Flutter

- Industrial Flutter
  https://www.industrialflutter.com/
- Sony Embedded Linux Embedder for Flutter
  https://github.com/sony/flutter-embedded-linux
- Raspberry Pi Embedder for Flutter
  https://github.com/ardera/flutter-pi

basysKom

Conclusion

Conclusion

- Flutter is a well-established cross-platform UI toolkit from the mobile space
- Flutter is an open-source framework with a large and active community
- Flutter has a rich package ecosystem
- Flutter is easy to learn and provides excellent opportunities for rapid development of custom UIs
- Its excellent tooling and good performance also make it an interesting choice for Embedded Linux applications