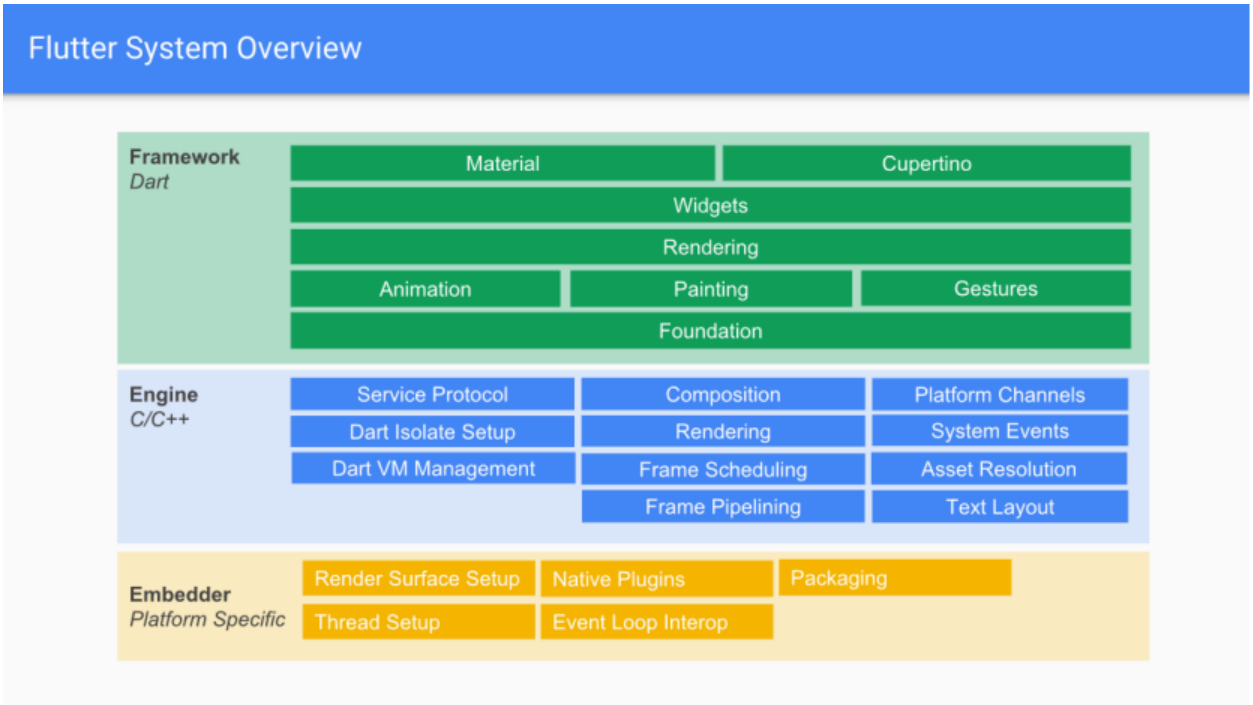
Flutter – Detailed Overview

Flutter's Architecture

This section summarizes Flutter's architecture. First, Flutter technologies will be introduced as well as its architectural design. Then detailed information regarding the framework and its engine will be presented.

Flutter uses two core open-source technologies, Dart and Skia, both developed and maintained by Google. Skia is a 2D graphics rendering library which provides common APIs for a variety of hardware and software platforms. It helps Flutter rendering widgets to devices' screens. Dart on the other hand, is a garbage-collected object-oriented language that provides a VM (Virtual Machine) hosting high-level Flutter framework functionalities such as core libraries. It also implements a library containing UI features that provides access to low-level Skia functionalities. Flutter combines both Dart and Skia in a shell. A shell serves as an interface for running Flutter applications, which contains implementations such as hardware communication for specific platforms. As an example, there is a shell for both Android and iOS and Desktop shells are also under development.



As seen in the diagram above, Flutter's architecture is designed in three layers: Framework, Engine and Embedder. The Framework layer provides high-level libraries for building applications using Dart. Material and Cupertino both hold implementations for styled widgets using Material Design and iOS style guidelines. Widgets provides all available Flutter widgets while Rendering, Animation, Painting and Gestures manage UI functionalities. Finally, Foundation represents framework primitives. Both framework and application code are compiled

in an AOT (Ahead of Time) mode. In Android, it compiles into native ARM (Advanced RISC Machine) and x86 libraries and in iOS it compiles into native ARM libraries. The Engine layer implements core framework libraries such as animations and graphics, file and network I/O (Input/Output), accessibility support and plugin architecture as well as Dart runtime. It is built in C/C++ and compiles to native code depending on the target platform. In Android it compiles using Android NDK (Native Development Kit), whereas in iOS it compiles using LLVM (Low Level Virtual Machine).

Flutter's Performance and Run Modes

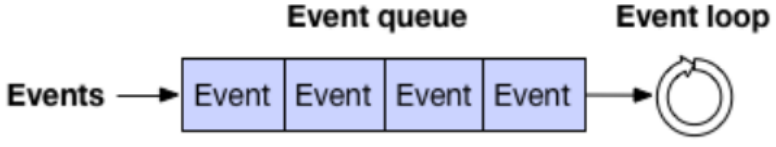
In this section Flutter's performance and run modes will be summarized. First, why Dart has an important role in Flutter. Then topics such as threading will be exposed. Lastly, Flutter run modes will be presented.

Dart's Importance in Flutter

Dart, the primary language which Flutter is built on, provides two critical features for both applications' development and release. Its runtimes and compilers support both JIT (Just in Time) and AOT (Ahead of Time) compilation modes. JIT allows developers to hot-reload the application by injecting source files' updates in Dart VM. After Dart VM updates source files, Flutter rebuilds the Widget tree, reflecting updated changes in the application. AOT on the other hand generates efficient ARM code which is important when releasing the application for production. Dart garbage collection is generational, providing fast small memory allocations on short-lived objects, which is ideal for Flutter as Widgets have a short life span due to the framework rendering the immutable view for every frame. These allocations and garbage collection are also performed without locks which is critical in displaying user interfaces as the application is not stopped until the memory is collected.

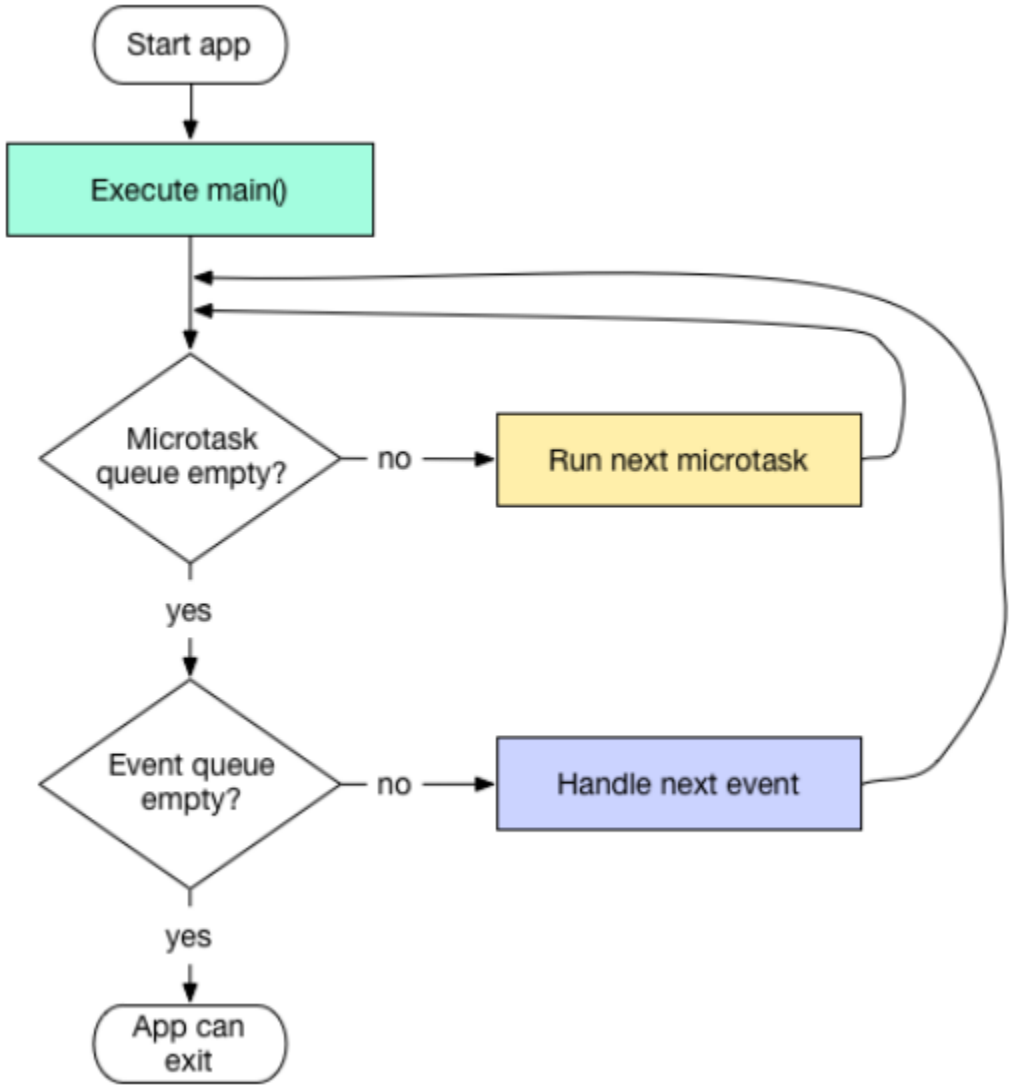
Threading in Flutter

In Flutter there are two types of threads which are respectively used in the engine and framework layer. In the framework layer, threads managed and created by Dart are called Isolates. An isolate is a thread that does not share memory with other threads as every isolate has its own heap, avoiding the need of certain locks such as memory allocation and garbage collection. An isolate is also run on a different process of the main one. As isolates do not share memory, communication between isolates is made via a message channel. Isolates are useful when there is a need to compute an expensive operation, which is convenient in Flutter as this operation will not block the UI renderer. Dart is single threaded as it runs on a single isolate. To control asynchronous calls, Dart uses a mechanism called Event Loops and Queues, as illustrated by the figure below.



The queue holds the operations to be executed while the event loop ensures that each operation is handled one at a time. In this mechanism there is one event loop and two queues. These queues are respectively the event queue and the microtask queue. The event queue primary holds events such as I/O, drawings, timers and isolates messaging. Microtask queue holds operations that are marked as more important than the event queue's, as operations in the microtask queue are executed before the event queue's. To schedule tasks in both queues, Dart provides two APIs, respectively:

- Future which adds an item to the event queue;
- A top-level function named scheduleMicrotask which adds an item to the microtask queue.



As seen in the diagram above, a Dart application starts by executing the main function. After finishing the execution, Dart checks if both microtask and event queues are empty. If not, as the microtask queue is checked first, it handles the next microtask event operation and starts queues

verification again. If both queues are empty the application can successfully exit. Looking into the engine layer, the engine does not have the responsibility to create or manage the threads it uses. Instead, threads are created and managed in the embedded layer, providing them to Flutter's engine. By doing this, Flutter lets the platform which the application is being executed on decide which resources should be provided to the engine. These threads are provided to Flutter Engine as task runners. Each task runner represents a dedicated thread for specific operations, these being:

- Platform Task Runner, which is referred as the main thread of the platform that is serving the Flutter application;
- UI Task Runner, which is represented as the main Dart isolate;
- GPU Task Runner, which executes tasks that requires access to the device's GPU;
- IO Task Runner, which is responsible for managing I/O operations.

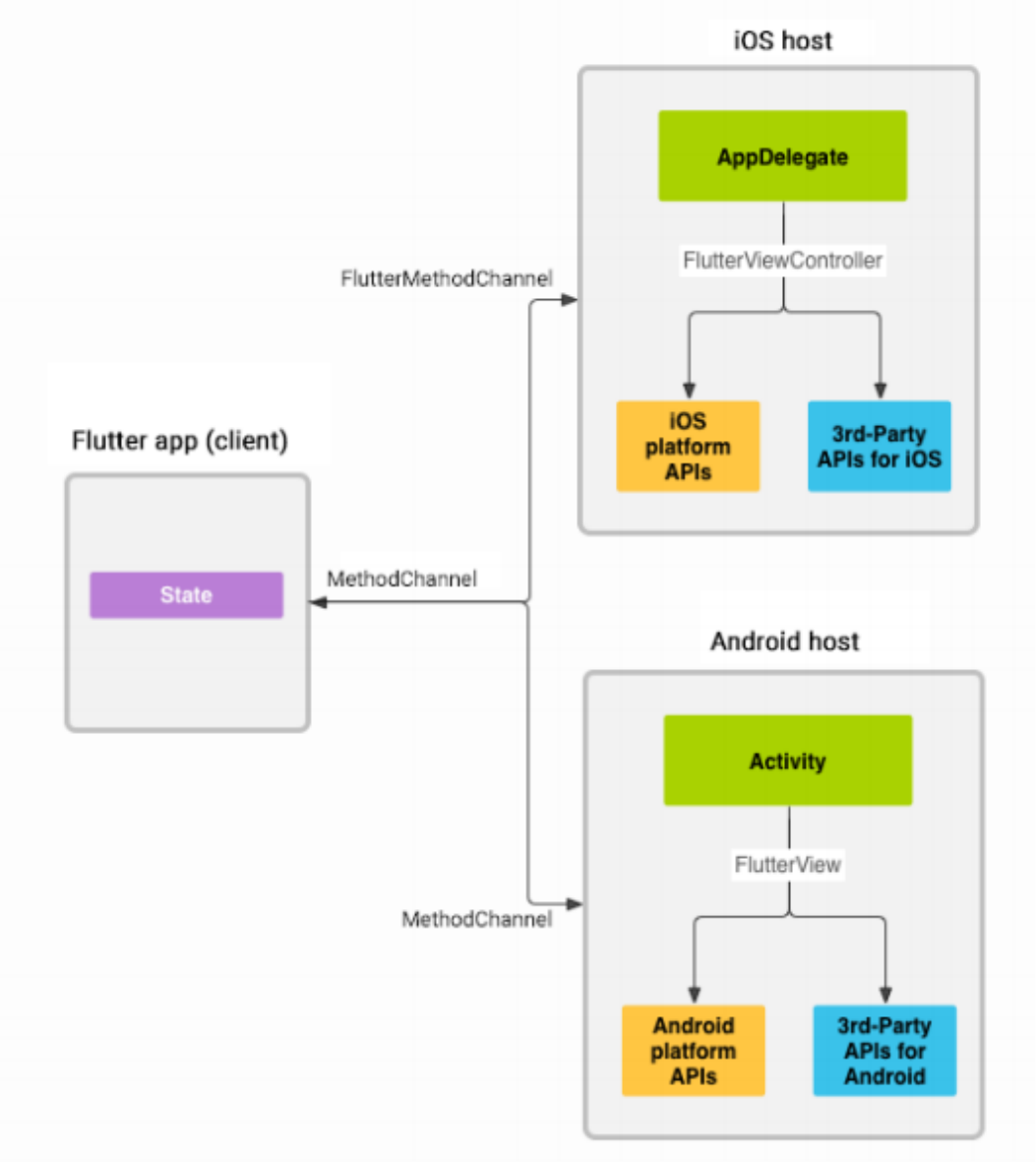
Flutter Run Modes

Flutter applications can be run in four different modes, providing different functionalities for either production or development state. These are:

- Debug Mode, which is the mode that developers use the most when developing Flutter applications. It provides tools for debugging such as the Dart observatory, performance overlay, Widgets' state dump and code assertions. As it compiles the application in JIT mode, it also provides hot-reload to developers;
- Release Mode, which is the state of the application in production, including all performance and package size optimizations as it is compiled in AOT mode, disabling all debugging tools and assertions;
- Profile Mode, which gives developers the possibility to analyze the application performance in the state of an application in production as it is also compiled in AOT mode;
- Headless Test Mode which enables the possibility to debug the application for desktop platforms.

Flutter's Native Communication with Devices

This section will summarize how Flutter and native modules communicate with each other. Even though Flutter provides a variety of features that ease the development of mobile applications, some native device functionalities such as battery level and push notifications are not available. Yet, Flutter allows developers to create plugins that implement accessors to native functionalities. The communication between the Flutter application (client) and the plugin (host) is made using a platform channel. Platform channels allow both communication from client to host or from host to client.



The communication between a Flutter client and both Android and iOS hosts is demonstrated by the figure above. In this example, the client can invoke a native method. The result is communicated as a message via a Method Channel. The method call is encoded into binary before being sent, and the result is decoded in a Dart value. The Dart values which are possible to decode are identified as codecs.

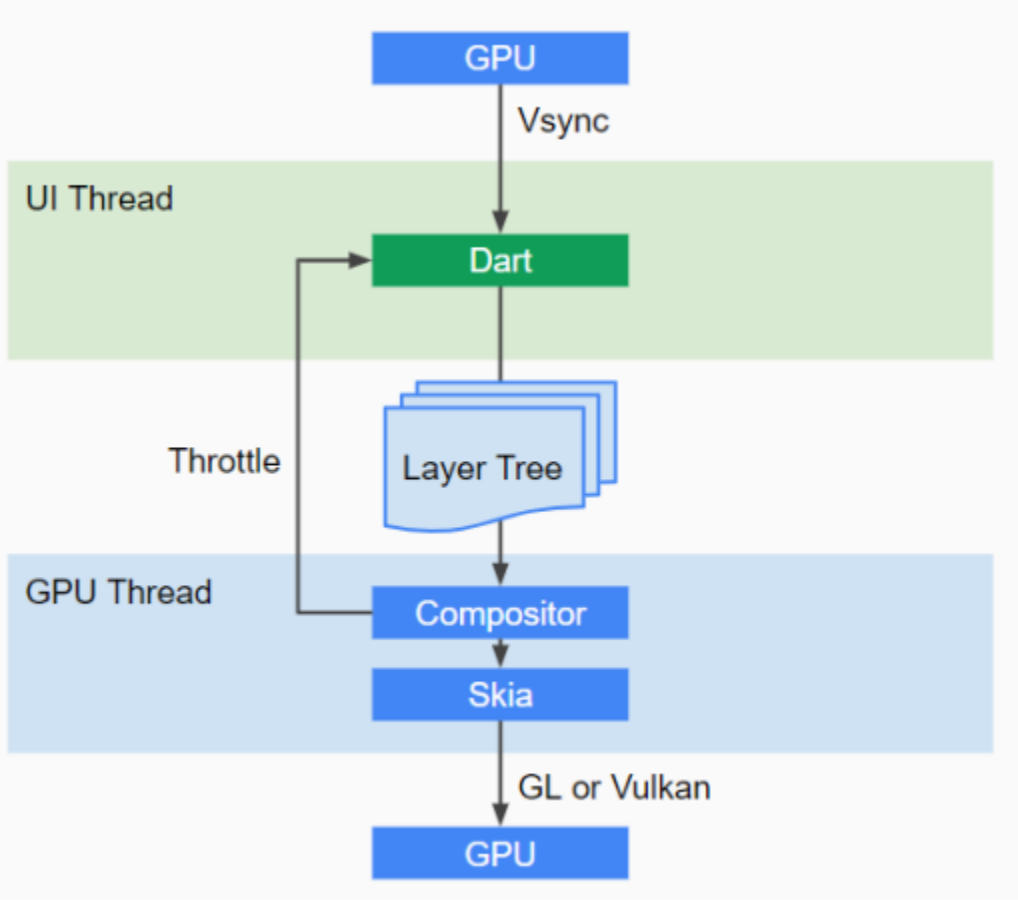
Flutter also allows developers to natively register events and streams using Event Channel and send and receive messages as Strings using a Message Channel. Additionally it is also possible to invoke native methods without developing a plugin using an Optional Method Channel.

How Flutter Renders Graphics

The main topics addressed here are Flutter's graphics and rendering pipelines.

Graphics Pipeline

Flutter's graphics pipeline is illustrated below.



Two main threads exist for the graphics pipeline: the UI Thread and the GPU (Graphics Processing Unit) Thread. The latter is responsible for rasterization and composition, while the first one is primarily responsible for generating rendering instructions for the GUI (Graphical User Interface) in the form of a Layer Tree, i.e. the necessary components to be drawn are stacked in layers. Flutter's engine needs references for 4 task runners. Logically, the ones that run on the UI Thread and the GPU Thread are the UI Task Runner and the GPU Task Runner respectively.

UI Task Runner

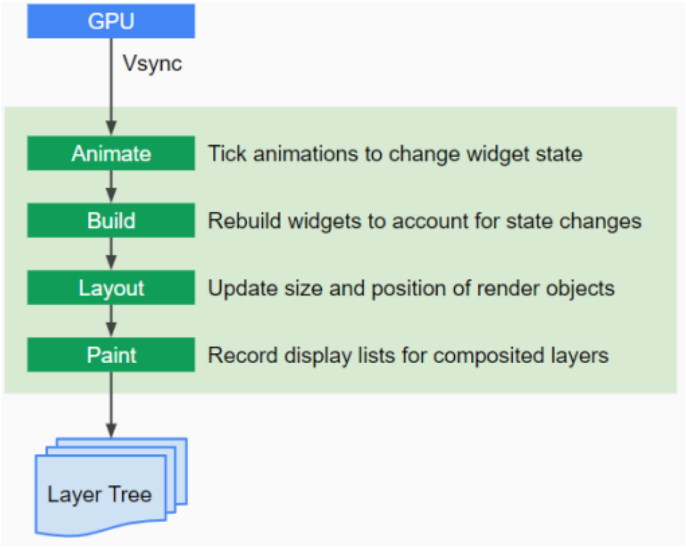
Flutter's engine uses the UI task runner to execute all Dart code for the root isolate. The latter is responsible for running the application's main Dart code. The root isolate has bindings set up by the engine in order to schedule and submit frames. For each frame that Flutter needs to render:

- The root isolate has to tell the engine that a frame needs to be rendered;
- The engine will ask the platform that it should be notified on the next vsync (Vertical Sync);
- The platform waits for the next vsync;
- On vsync, the engine alerts the Dart code and performs the following tasks:
    - Update animation interpolators, i.e. run animations;
    - Check for changes in the Widget tree and rebuild them in a layout phase;
    - Lay out the newly constructed Widgets and paint them into a layer tree that is submitted immediatly to the engine. It should be noted that no rasterization happens here, only a description of what needs to be painted is constructed as part of the paint phase;
    - Build or update a node tree, with each node containing semantic information about Widgets on screen. This is used to update platform specific accessibility components.

GPU Task Runner

The GPU task runner is responsible for executing tasks that require access to the device's GPU. The layer tree created on the UI task runner is client-rendering-API agnostic. This means that the same layer tree can be used to render a frame using OpenGL, Vulkan, or any other configuration for Skia. Components of this task runner take the layer tree and create the appropriate low-level GPU commands. These are also responsible for setting up all GPU resources for a particular frame (e.g. setting up the framebuffer, managing surface lifecycle, ensuring that textures and buffers for a given frame are fully prepared).

Rendering Pipeline



The rendering pipeline's logic is as follows:

1. Animations ran by a GPU's vsync will alter Widgets' States - Animate;
2. State changes will trigger Flutter to build a new Widget tree - Build;
3. Flutter will check for changes in the Render tree and will update it accordingly - Layout;
4. Since Flutter knows the size and position of the changed Widgets, it paints and composites them - Paint;
5. The result of the rendering pipeline is a new layer tree that is submitted to Flutter's engine and passed on to the GPU to be drawn on the device's screen – Layer Tree.

From the steps mentioned above, one can extract, on an abstract level, that Flutter renders applications in four phases, similar to the way web browsers render their graphics:

- Layout Phase: in this phase, Flutter is only concerned with determining how big each object is and where it will be displayed on the screen;
- Painting Phase: this phase focuses on what each object looks like;
- Compositing Phase: at this point, Flutter places everything together into a scene, i.e. the layer tree, and sends it to the GPU for processing;
- Rasterizing Phase: the final phase of Flutter's rendering process has the goal of displaying the mentioned scene as a matrix of pixels.

Layout Phase

Flutter's layout phase is composed of two linear movements: the passing of constraints down the tree and the passing of layout details up the tree. The layout process is as follows:

1. The parent passes certain constraints down to each of its children. These are a set of rules that the children must respect when laying themselves out individually. This could be any type of rule, for example, the parent might pass down a maximum width or minimum height constraint;
2. Receiving its parent's constraints, the child will create new constraints and pass those down to its children. This will keep going until the tree hits a leaf Widget with no children;
3. The leaf Widget will then determine its layout details based on the constraints that were passed down to it. After it defines its own layout details, it passes them back to its parent.
4. Receiving the layout details from its child, the parent will do the same thing as it. Based on the child's layout details it will determine its own and pass them onto its own parent and so on and so forth until the tree's root Widget is reached.

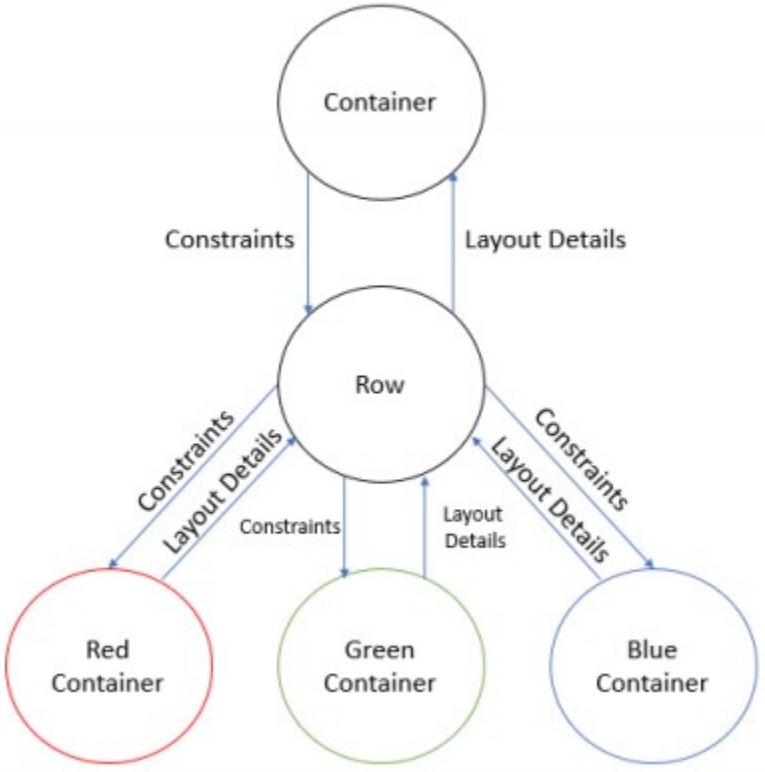
In order to simplify the process of understanding the layout phase, consider the figure below. It is an example of a simple UI built in Flutter of a container with a row of three different sized colored containers. The respective code snippet is shown below the referred Figure.

```
Container(
    color: Colors.white,
    child: Row(
        children: <Widget>[
            Container(
                width: screenSize.width / 2,
                height: screenSize.height / 10,
                color: Colors.red,
            ),
            Container(
                width: screenSize.width / 4,
                height: screenSize.height / 5,
                color: Colors.green,
            ),
            Container(
                width: screenSize.width / 4,
                height: screenSize.height / 15,
                color: Colors.blue,
            ),
        ]
    ),
);
```

Hierarchically, the Widget tree Flutter builds has this structure:

- Container (Parent)
    - Row (Container's child)
        - Red Container (Row's child);
        - Green Container (Row's child);
        - Blue Container (Row's child);

The layout process for this UI is as follows:



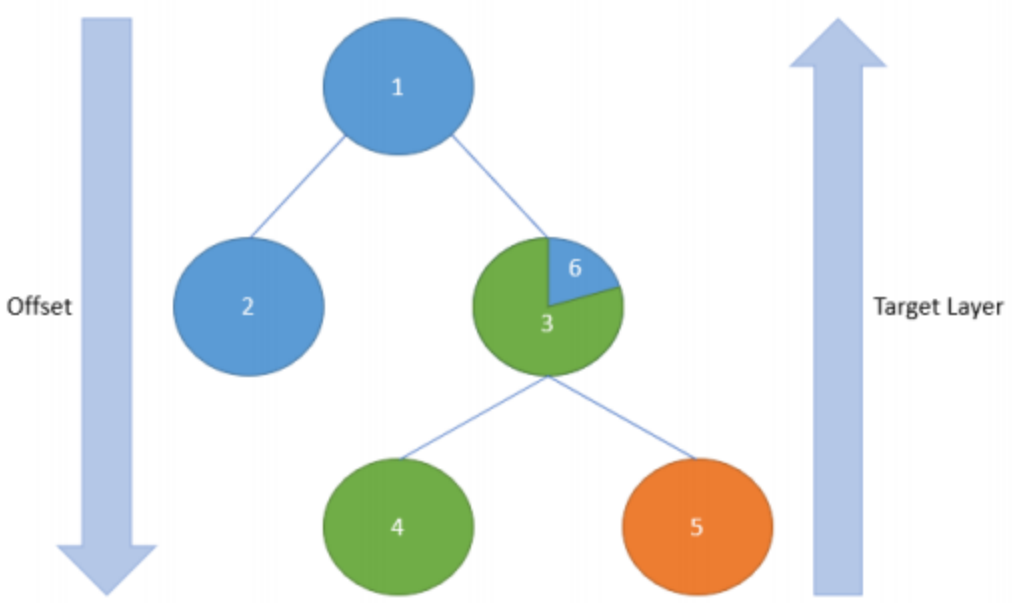
Starting from the Container, the root Widget of the tree, it has no constraints (although one might consider that the only existing constraints in this case are the mobile device's screen height and width). Like so, it tells its child, the Row Widget, that it can use all its available space. The Row Widget then tells to each of its children, the Red, Green and Blue Containers, that it has no constraints either. Likewise, the Row's children are free to use whatever space they want. However, the Red Container has its own constraints, its width is half of the screen's width and its height is a tenth of the screen's height. The Green Container defines its own constraints as well, its width being one fourth of the screen's width and its height being a fifth of the screen's height. The same logic applies to the Blue Container, it having a width of a fourth of the screen's width and a height of one fifteenths of the screen's height. Since there are no more childs, i.e. the tree has reached its leaves, the Red, Green and Blue Containers define their layout details and pass them up to their parent, the Row Widget. On receiving these, the Row Widget defines its layout details based on the ones it received from its children and passes its layout details to the parent Widget, the root Container. Finally, the root Container defines its own layout details and the layout phase is finished.

Because the layout process in Flutter is linear, in contrast to other systems where a multi-pass layout is implemented (i.e. the layout process tree is traversed multiple times to adjust sizes and positions), it makes the method a lot more efficient, achieving O(n) time complexity instead of the $O(n^2)$ or worse time complexity of multi-pass layout process algorithms. Furthermore, if tight constraints (e.g. passing down a specific size to a child Widget) or other scenarios that create

relayout boundaries happen, Flutter's layout process is even more efficient, achieving a sublinear time complexity.

Painting Phase

The second step in Flutter's rendering pipeline is painting. This step is similar to the layout phase. A Widget tree is built, and since the framework knows where each widget is and how big it is due to the layout phase, it only needs to paint the Widgets. However, when there's a need for painting in different layers (e.g. drawing a play button on top of a video that is handled outside of Flutter), the painting phase becomes more complicated. To handle this, a similar logic to the layout phase is used, as illustrated by the figure below.



Like the layout process, for painting Widgets, the tree is traversed in depth order. When going down the tree, each Widget's parent will inform their children about their respective offset, so they know where they have to be drawn. When the painting phase reaches the leaves of the tree, each child Widget will inform their parent about its target painting layer. This happens so that if a given child is painted before or after the parent, the parent knows what layer that painting needs to happen in, i.e. the child is telling its parent where the painting needs to continue. Moreover, a Widget's painting can be split into several layers, like demonstrated in the figure above. In this case, we have three painting layers: blue, green and orange. The process starts out at the root Widget, which only has the blue painting layer. The same thing happens for the first child of the tree's root. The painting process goes through each Widget in a depth first fashion, drawing each tree node into its respective layer. When it is time to go up the tree, the root's second child's child will inform the parent that it should continue painting in another layer (in this case, it was painted first in the green layer and after it should continue painting in the blue layer). This allows Flutter's painting phase, like its layout phase, to be linearly complex.

Furthermore, the concept of having boundaries, in this case, repaint boundaries, is applied again to achieve sublinear complexity in specific scenarios.

Compositing Phase

The third step in Flutter's rendering pipeline is compositing. This phase, as already mentioned, is dedicated to creating a scene out of the Widget tree and sending it to the GPU for processing. Since up until this point Flutter uses the concept of layered Widgets and painting layers, the created scene is made up of composited layers. The latter and how they are managed is what gives Flutter an advantage over other rendering pipelines at this phase. Two ways compositing can be achieved are: (a) directly drawing a vector that represents the graphical component, i.e. executing a set of draw commands to the GPU or (b) record pixels as a texture and render it directly on the screen. Choosing which method to use can have implications on an application's efficiency because of the amount of GPU processing that happens. If a composited layer is used in a lot of frames, it is beneficial to create a texture and use it. On the other hand, if a composited layer is not used in a lot of frames, it is beneficial to always send a set of draw commands to the GPU. In order to find an equilibrium between these two methods, and consequently optimize GPU usage, Flutter has implemented heuristics that will decide whether a composited layer should be transformed into a texture or not.

Rasterizing Phase

The fourth and final phase of Flutter's rendering pipeline is rasterizing. In this phase, the layer tree that was generated is transformed into a vector graphics image and then converted into a matrix of pixels to be displayed on a device's screen.

Flutter's Widget System and State Management

Flutter's Widget library is the primary means of developing software with Google's framework . As such, it is essential to understand what a Widget is, how it behaves with the remaining components of the framework and how to manage an application's state with Stateful Widgets. This section discusses all these points.

What is a Widget?

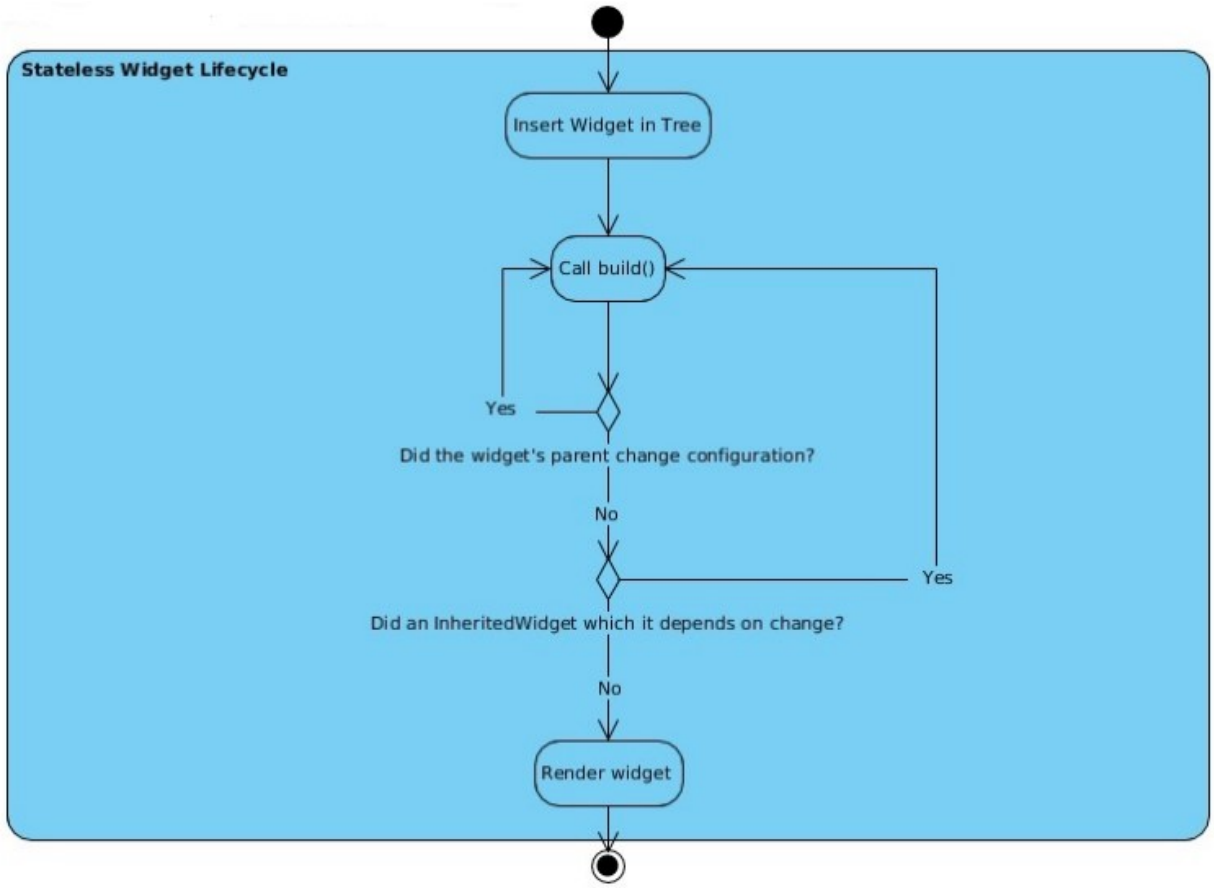
In Flutter, a Widget is an abstract description of a User Interface. Like so, everything in a UI is a Widget. These can be immutable (Stateless Widgets) or mutable (Stateful Widgets) and they describe what their view should look like given their configuration and state at a given moment.

Stateless and Stateful Widgets

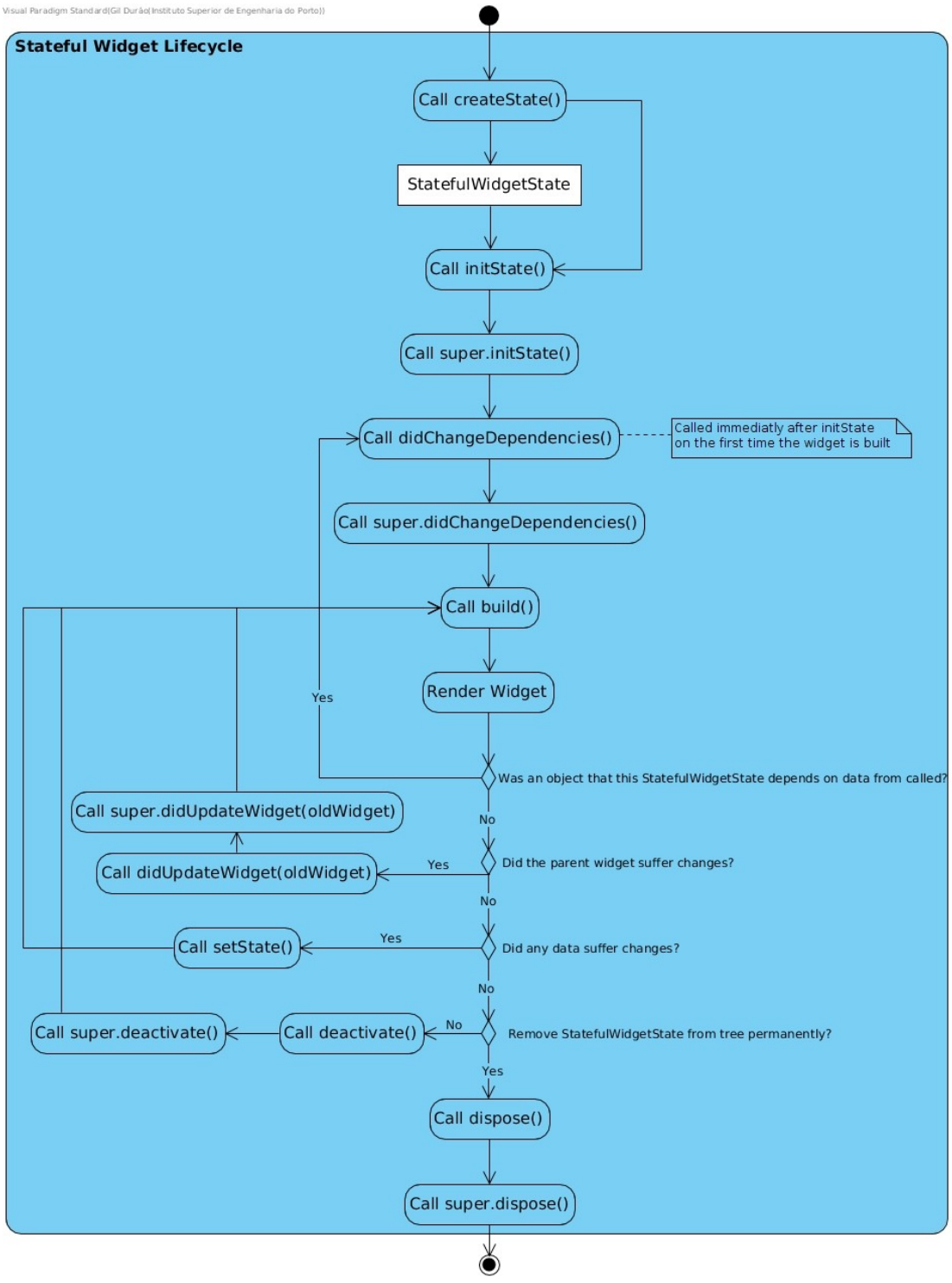
Widgets in Flutter can be Stateless or Stateful. Stateless Widgets are immutable whereas Stateful Widgets are composed by a State object which can change over time and as such will update the Stateful Widget's graphical description.

Even though Stateless and Stateful Widgets are the most commonly used, there are other implementations of Widgets, such as the Inherited Widget that is used for propagating data

efficiently down a Widget tree. Since Stateless and Stateful Widgets are the most used ones, it is important to understand them at a deeper level. To accomplish this, it is necessary to grasp both Widgets' lifecycle.



The Stateless Widget is first inserted into the Widget Tree. After, Flutter builds it in a given BuildContext6 . The next step is to check if the parent of the Stateless Widget changed its configuration. If the parent changed, Flutter needs to build the Stateless Widget again. For example, suppose a Text Widget inside a Container Widget. If the Container changes color, Flutter needs to build the Text Widget again. The same step applies for an Inherited Widget that a Stateless Widget depends on. If the first changes its configuration then the framework needs to build the latter again. In this case we can imagine a Text Widget that receives the text it needs to display from an Inherited Widget. If that text changes, Flutter needs to build the Text Widget again. If neither the parent Widget or an Inherited Widget change, Flutter can render the Stateless Widget directly. The lifecycle of a Stateful Widget is presented next.



The first step in the lifecycle of a Stateful Widget is the creation of a State object. State represents information that can be read synchronously when the Widget is built and that can change during the lifetime of the Widget. The notification of such changes falls on the responsibilty of the implementer, using the *setState* method. As such, every action that follows the creation is on the State object of the Stateful Widget and not the Stateful Widget itself. After

creating the Widget's State, Flutter will initialize it. If should be noted that if the implementer needs to initalize data for the Stateful Widget or subscribe to streams and/or change notifiers, there is a need to override the *initState* method and make a call to *super.initState* in the same override. However, if the implementer does not need to initialize anything, no override is required and Flutter will simply call *initState*. This method is only called one time. Afterwards, Flutter will call *didChangeDependencies*, which, as stated by the method's name, will check if any State dependencies have changed. The mentioned method is always called after *initState*, and usually there is no need to override it. Followed by that, Flutter will build and render the Stateful Widget. If an object that the Stateful Widget depends on data from is called or changes, Flutter will call *didChangeDependencies* again. A scenario where this may happen if the Stateful Widget relies on an Inherited Widget that is updated. Flutter proceeds to check if the Widget's parent changes. If it does, *didUpdateWidget* is called to rebuild the Stateful Widget. Once again, *didUpdateWidget* can be overriden, although most of the times it is not necessary. After this step, Flutter will check if any data of the Stateful Widget's State suffered changes. If it did, then *setState* is called to refresh the Widget with the updated data. A lot of the times, the implementer will call *setState*. This method can be used, for example, to activate a loading indicator on the press of a button. Lastly, if the Widget's State is not removed from the Widget tree permanently, Flutter will call *deactivate*, i.e. the State object will be reinserted from one place in the Widget tree to another before the frame change finishes. This method is rarely called. If the Widget's State is to be removed permanently from the tree, Flutter calls *dispose*. In this method, the implementer should cancel all subscriptions made in *initState*.

Widgets, Elements and Render Objects

Even though the core of Flutter is Widgets, there are two more classes that play an important role in assuring the framework's high performance: Elements and Render Objects. Render Objects contain all the logic for rendering a Widget. As such, it knows where it is going to be and how big it is (layout), what its visual aspect is (painting) and it also handles hit-testing, i.e. gesture detection and how to react to certain user inputs. While the Widget tree is immutable, the Render Object tree is mutable. Thus, for every Widget that is created there is a matching Render Object. Due to their immutability, Widgets can be used to configure multiple subtrees simultaneously. As such, a representation of the use of a Widget at a given location in the tree is required. This is the purpose of Elements. Elements create a bridge between the immutable Widget tree and the Render Object mutable tree. They hold a reference between every Widget and its matching Render Object and when a Widget changes, its Element will compare it to its Render Object and dictate what actions need to be ensued. The three trees exist to enhance Flutter's performance. To illustrate how the framework uses these three trees, another Flutter UI is presented.

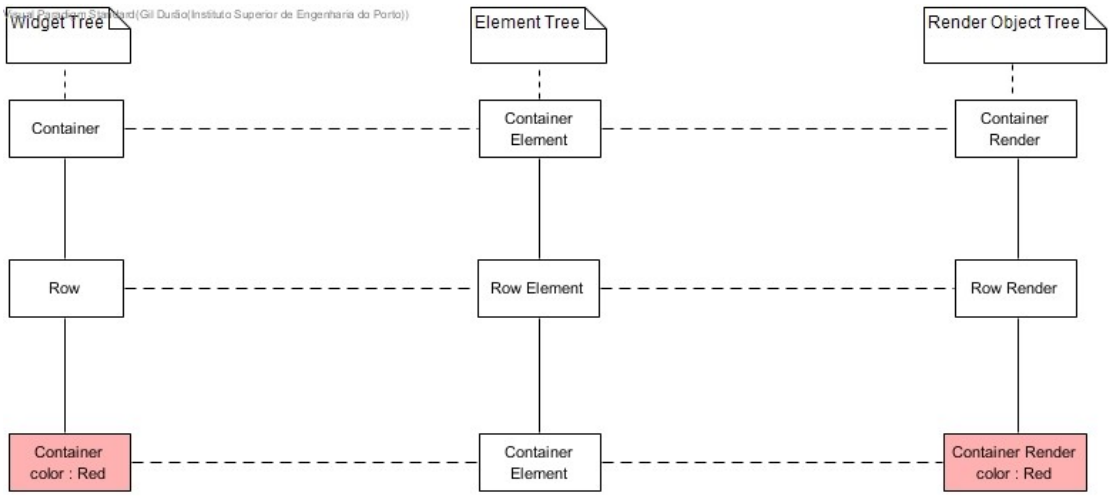
The code for this UI is presented below.

```
Container(
    color: Colors.white,
    child: Row(
        children: <Widget>[
            Container(
                width: screenSize.width,
                height: screenSize.height / 10,
                color: Colors.red,
            ),
        ]
    ),
);
```

When Flutter builds this UI for the first time, the following happens:

1. Flutter builds the Widget tree containing all the UI's Widgets;
2. Secondly, Flutter will create a second tree with Element objects, one for each Widget;
3. Finally, Flutter will build a Render Object tree out of the Element objects.

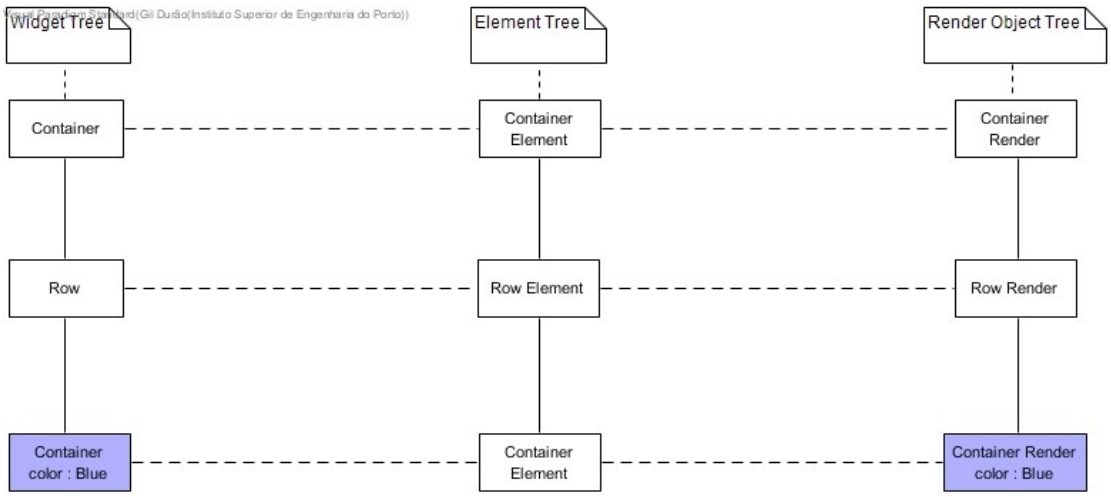
After the UI is built, the framework will hold a forest of trees illustrated in in the next figure.



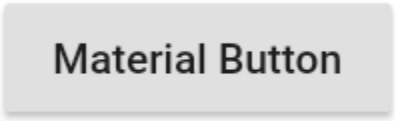
As stated, an Element and a Render Object are created for each Widget in the Widget tree. Elements will hold references of Widgets and Render Objects and compare each other when the framework builds the Widget tree again. If the Red Container's color changes to blue, Flutter will do the following:

1. Rebuild the Widget tree;
2. For each Element in the Element tree, compare the rebuilt Widgets with the Render Objects in the Render Object tree.
3. If the new Widgets are of the same type of the old Widgets (i.e. only the Widget's configuration has changed, such as the example stated above), Flutter will only update the Render Object's configuration. If this condition is not true, then Flutter will remove all of the old objects from the three trees and create new ones.

In the presented scenario, Flutter would start by comparing the root Container. Since no changes happened between the Widget and the Render Object, the framework does not need to do anything. The same logic applies to the Row Widget. For the last Widget, the Red Container, Flutter will detect a change in its configuration between the old Widget (Red Container) and the new Widget (Blue Container). As such, it will update the Red Container's Render Object's configuration to render the Container with the color blue. This results in the updated forest tree in the next figure.



It should be noted that only the Widget tree was rebuilt. All the objects in the Element and Render Object trees are the same. This is one of the aspects that makes Flutter very efficient in rendering UIs. If the blue Container were to be replaced with another Widget type, like a Material Button as seen in the figure below, then the framework would need to rebuild all three trees.



As before, Flutter will not detect any changes made to the root Container and Row Widgets. However, the last Widget type changed from Container to Material Button. Because of this change, the framework will remove the Container's Element and Render Object and instantiate new ones for the Material Button Widget. Then the new tree forest would have the state in the figure below.