## Background

Flutter UIs function conceptually similar to a WebView on Android. The Flutter Framework takes a widget tree described by the app developer and translates that into an internal widget hierarchy, and then from that decides what pixels to actually render. Web developers can think of this as analogous to how a browser takes HTML/CSS, creates a Document Object Model ("DOM"), and then uses that DOM to actually render pixels each frame. Also like a WebView, Flutter UIs aren't translated to a set of Android View widgets and composited by the Android OS itself. Flutter controls a texture (generally through a SurfaceView) and uses Skia to render directly to the texture without ever using any kind of Android View hierarchy to represent its internal Flutter widget hierarchy or UI.

This means that like a WebView, by default a Flutter UI could never contain an Android View within its widget hierarchy. Since the Flutter UI is just being drawn to a texture and its widget tree is entirely internal, there's no place for the View to "fit" within Flutter's internal model or render interleaved within Flutter widgets. That's a problem for developers that would like to include complex pre-existing Android views in their Flutter apps, like WebViews themselves, or maps.

To solve this problem Flutter created an AndroidView widget that Flutter developers can use to visually embed actual Android View components within their Flutter UI.

## The approach

`AndroidView` widgets need to be composited within the Flutter UI and interleaved between Flutter widgets. However the entire Flutter UI is rendered to a single texture as mentioned in the previous section.

In order to solve this problem, Flutter inflates and renders `AndroidView` widgets inside of VirtualDisplays instead of attempting to add it to the Android View hierarchy alongside Flutter's texture directly. The `VirtualDisplay` renders its output to a raw graphical buffer (accessed through `getSurface()`), and not to any actual real display(s) of the device. This allows Flutter to graphically interleave the Android View inside of its own Flutter widget tree by taking the texture from the `VirtualDisplay` output and treating it as a texture associated with any other Flutter widget in its internal hierarchy. Then the `VirtualDisplay's Surface` output is composited with the rest of the Flutter widget hierarchy and rendered as part of Flutter's larger texture output on Android.

## Are there any alternatives?

Yes! See Hybrid Composition.

## Associated problems and workarounds

While this approach unblocks the core problem with embedding Android views in a Flutter UI visually, the `VirtualDisplay` choice causes a long tail of issues that we've needed to work around.

The heart of the problem with this approach is that the Android View inside of the `VirtualDisplay` is, as far as Android is concerned, inside of its own View hierarchy in a completely invisible Display and completely unconnected and unrelated to the View hierarchy that contains the Flutter texture output.

Much of the internal functionality to Android depends on walking the View hierarchy and querying information about the given View in its current hierarchy and window in order to function. Since the information for the embedded view is "wrong" here this internal functionality tends to break down. In addition to that this logic can change based on Android version, so we've needed to case some of our logic based on the runtime version of the OS.

Some of our issues with this approach and extended workarounds are described below.

### Touch events

By default touch events would not work with PlatformViews. The Android View is inflated inside of a `VirtualDisplay`. Whenever the user taps what they *see* as the Android View on their primary display, they're "really" tapping the part of Flutter's texture output that we're rendering to look like the actual Android View. The touch event created by their input is getting sent straight to Flutter's View, not to the Android View they're actually trying to tap on.

#### The workaround

- We detect whether or not the touch from the user should collide with the `AndroidView` Flutter widget using hit testing logic within the Flutter Framework.
- When the touch is a hit, we dispatch a message to the Android engine embedding containing the details of the touch event.
- Within the Android embedding, we convert the coordinates of this event from the ones describing the touch inside of the larger Flutter texture to ones that would match the real Android View inside of its `VirtualDisplay`. We then create a new `MotionEvent` describing the touch and forward it to the real Android View inside of its `VirtualDisplay`.

#### Limitations

- We're dispatching the new `MotionEvent` directly to the known Android View embedded by the user. In cases where the View spawns other Views this dispatch may be sent to the wrong place.

- We're synthesizing a new `MotionEvent` based on the information handed to us by the Flutter framework. It's possible that there's data being lost and not totally carried over to our new `MotionEvent`, or some other general problem with synthesizing `MotionEvent`s.

## Accessibility

### Background

Explaining our problems with accessbility (or "a11y") requires a brief detour into explaining a11y itself on Android, and how a11y typically works on Flutter.

Android itself builds up a tree of a11y nodes for each Android View rendered to the screen. Each a11y node contains semantic information about what information is represented by a UI element: for example, whether it's a button, and what the text on button is. Typically the OS builds this information itself by directly walking the View hierarchy. A11y services on the device then use this tree of a11y nodes to deliver the semantic information in an accessible way to the user, by example drawing highlights over interactable UI elements and reading the semantic information described by the UI aloud to the user. Users can then use a11y services to interact with a UI, for example by using it to activate the button without needing to press once directly on where the button is graphically rendering on screen.

Some UIs, like Flutter and WebViews, don't have a tree of Android Views to describe their UIs. For these Android has a concept of a "virtual" a11y node hierarchy. Instead of walking the View hierarchy, Android Views may implement an AccessibilityNodeProvider that returns a tree of a11y nodes to describe whatever is being rendered within its own View subtree. The Flutter Android embedding uses this concept to generate an a11y node tree that matches the `Semantics` tree created by the Flutter framework.

### The problem

Theoretically we should have been able to attach the a11y tree from the embedded Android View to our `AccessibilityNodeProvider` in Flutter, but in reality we've found that any attempts to do so directly fail. The reality of the situation appears to be that a11y code in Android frequently relies on the View hierarchy for state management and querying extra information, so attempting to parent the child's hierarchy from the embedded view to our virtual hierarchy fails to really correctly associate the underlying view hierarchy in a functional way for a11y. In other words, reparenting the a11y node trees only "fixes" the a11y node tree itself, but there are still multiple places where Android a11y code relies on being able to query the "real" view hierarchy for required info and ends up failing with bugs in our reparented case.

**The workaround**

- Create a mirror copy of the Android View's a11y nodes as a subtree within the Flutter Android embedding. This mirror copy consists of virtual nodes and is part of the larger virtual a11y node tree created by Flutter's `AccessibilityNodeProvider`.
- (P and above only): blacklisted APIs and reflection are usually used to create a mirror copy of the Android View's a11y nodes. The critical private information are the node IDs of the parent and children of the a11y hierarchy, which we need to construct our mirror correctly but which are private data to the nodes. On newer Android versions we read the data we need from the serialized binary form of the node instead.
- We forward all events sent to mirror a11y nodes to the "real" a11y nodes in the Android View's subtree as well. This forwarding involves translating any coordinates on the event, similar to forwarding touch events. This means users can use a11y services to still interact with the embedded Android UI.

**Limitations**

- Only pre-existing virtual hierarchies can be duplicated and mirrored to our own virtual subtree in this way. That means that embedded views like WebView are accessible through this mechanism, but things like `Button` are not.
- The use of reflection and serialization to read private a11y node data in order to make an accurate mirror tree is extremely fragile. This could easily be broken by Android in a future release.

Tracked in flutter/flutter#19418. Better support for reparenting a11y hierarchies is also an Android SDK issue.

## Text input

Normally the embedded Android view wouldn't be able to get any text input because it's inside of a `VirtualDisplay` that is always reported as being unfocused. Android doesn't offer any APIs to dynamically set or change the focus of a `Window`. The focused `Window` for a Flutter app would normally be the one actually holding the "real" Flutter texture and UI, and directly visible to the user. `InputConnections` (how text is entered in Android) to unfocused Views are normally discarded.

**Workaround**

- We override checkInputConnectionProxy so that Android treats the Flutter View as a proxy for the Input Method Editor (IME) when it tries to talk to the embedded Android View. Basically, Android asks the Flutter View for an `InputConnection` when the embedded Android View wants one.

- Q changed the `InputMethodManager` (IMM) to be instantiated per `Window` instead of being a global singleton. So our naive attempt at setting up a proxy no longer worked on Q. To work around this further we created a subclass of `Context` that returned the same `IMM` as the Flutter View instead of the real `IMM` for the window when `getSystemService` is queried. This means whenever Android tries to use the `IMM` in our `VirtualDisplay` it still sees and uses the `IMM` from the real display that has been set to use the Flutter View as a proxy.
- When the Flutter Android embedding is asked for an `InputConnection`, it checks to see if an embedded Android View is "really" the target of the input. If so it internally goes to the embedded Android View, gets the `InputConnection` from there, and returns it to Android as if the embedded Android View's `InputConnection` is its own.
- Android sees that the Flutter view is focused and available, so it takes the `InputConnection` that's ultimately from the embedded Android View and uses it successfully.

**In embedded WebViews**

WebViews running on Android versions pre N have additional complications because they have their own internal logic for creating and setting up input connections that don't completely defer to Android. Within the `flutter_webview` plugin we've also needed to add extra workarounds in order to enable text input there.

- Set a proxy view that listens for input connections on the same thread as WebView.. Without this WebView would internally consume all `InputConnection` calls without the Flutter View proxy ever being notified.
- Within the proxy thread, refer back to the Flutter View for input creation..
- Reset the input connection back to the Flutter thread when the WebView is unfocused. This prevents the text input from being "stuck" inside of the WebView.

**Limitations**

- In general this depends on internal Android behavior and is brittle. The Q workaround was a surprise for us as we were broken by what was supposed to be an internal Android refactoring.
- Some text functionality is still broken. For example the "Copy" and "Share" dialogue is currently not usable.