Hosting native Android views in your Flutter app with Platform Views

Platform integration > Android > Android platform-views

Contents

Platform views allow you to embed native views in a Flutter app, so you can apply transforms, clips, and opacity to the native view from Dart.

This allows you, for example, to use the native Google Maps from the Android SDK directly inside your Flutter app.

Note

This page discusses how to host your own native Android views within a Flutter app. If you'd like to embed native iOS views in your Flutter app, see Hosting native iOS views. If you'd like to embed native macOS views in your Flutter app, see Hosting native macOS views.

Platform Views on Android have two implementations. They come with tradeoffs both in terms of performance and fidelity. Platform views require Android API 23+.

Hybrid Composition

Platform Views are rendered as they are normally. Flutter content is rendered into a texture. SurfaceFlinger composes the Flutter content and the platform views.

- + best performance and fidelity of Android views.
- - Flutter performance suffers.
- - FPS of application will be lower.
- - Certain transformations that can be applied to Flutter widgets will not work when applied to platform views.

Texture Layer (or Texture Layer Hybrid Composition)

Platform Views are rendered into a texture. Flutter draws the platform views (via the texture). Flutter content is rendered directly into a Surface.

- + good performance for Android Views
- + best performance for Flutter rendering.
- + all transformations work correctly.
- - quick scrolling (e.g. a web view) will be janky
- - SurfaceViews are problematic in this mode and will be moved into a virtual display (breaking a11y)
- - Text magnifier will break unless Flutter is rendered into a TextureView.

To create a platform view on Android, use the following steps:

On the Dart side

On the Dart side, create a `Widget` and add one of the following build implementations.

Hybrid composition

In your Dart file, for example `native_view_example.dart`, use the following instructions:

1. Add the following imports:

```dart
import 'package:flutter/foundation.dart';
import 'package:flutter/gestures.dart';
import 'package:flutter/material.dart';
import 'package:flutter/rendering.dart';
import 'package:flutter/services.dart';
```

2. Implement a `build()` method:

```dart
Widget build(BuildContext context) {
  // This is used in the platform side to register the view.
  const String viewType = '<platform-view-type>';
  // Pass parameters to the platform side.
  const Map<String, dynamic> creationParams = <String, dynamic>{};

  return PlatformViewLink(
    viewType: viewType,
    surfaceFactory: (context, controller) {
      return AndroidViewSurface(
        controller: controller as AndroidViewController,
        gestureRecognizers: const <Factory<OneSequenceGestureRecognizer>>{},
        hitTestBehavior: PlatformViewHitTestBehavior.opaque,
      );
    },
    onCreatePlatformView: (params) {
      return PlatformViewsService.initSurfaceAndroidView(
        id: params.id,
        viewType: viewType,
        layoutDirection: TextDirection.ltr,
        creationParams: creationParams,
        creationParamsCodec: const StandardMessageCodec(),
        onFocus: () {
          params.onFocusChanged(true);
        },
      )
        ..addOnPlatformViewCreatedListener(params.onPlatformViewCreated)
        ..create();
    },
  );
}
```

For more information, see the API docs for:

- `PlatformViewLink`
- `AndroidViewSurface`
- `PlatformViewsService`

TextureLayerHybridComposition

In your Dart file, for example `native_view_example.dart`, use the following instructions:

1. Add the following imports:

```dart
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
```

2. Implement a `build()` method:

```dart
Widget build(BuildContext context) {
  // This is used in the platform side to register the view.
  const String viewType = '<platform-view-type>';
  // Pass parameters to the platform side.
  final Map<String, dynamic> creationParams = <String, dynamic>{};

  return AndroidView(
    viewType: viewType,
    layoutDirection: TextDirection.ltr,
    creationParams: creationParams,
    creationParamsCodec: const StandardMessageCodec(),
  );
}
```

For more information, see the API docs for:

- `AndroidView`

On the platform side

On the platform side, use the standard `io.flutter.plugin.platform` package in either Kotlin or Java:

Kotlin Java

In your native code, implement the following:

Extend `io.flutter.plugin.platform.PlatformView` to provide a reference to the `android.view.View` (for example, `NativeView.kt`):

```kotlin
package dev.flutter.example

import android.content.Context
import android.graphics.Color
import android.view.View
import android.widget.TextView
import io.flutter.plugin.platform.PlatformView

internal class NativeView(context: Context, id: Int, creationParams: Map<String?, Any?>?) :
PlatformView {
    private val textView: TextView

    override fun getView(): View {
        return textView
    }

    override fun dispose() {}

    init {
        textView = TextView(context)
        textView.textSize = 72f
        textView.setBackgroundColor(Color.rgb(255, 255, 255))
        textView.text = "Rendered on a native Android view (id: $id)"
    }
}
```

Create a factory class that creates an instance of the `NativeView` created earlier (for example, `NativeViewFactory.kt`):

```kotlin
package dev.flutter.example

import android.content.Context
import io.flutter.plugin.common.StandardMessageCodec
import io.flutter.plugin.platform.PlatformView
import io.flutter.plugin.platform.PlatformViewFactory

class NativeViewFactory : PlatformViewFactory(StandardMessageCodec.INSTANCE) {
    override fun create(context: Context, viewId: Int, args: Any?): PlatformView {
        val creationParams = args as Map<String?, Any?>?
        return NativeView(context, viewId, creationParams)
    }
}
```

Finally, register the platform view. You can do this in an app or a plugin.

For app registration, modify the app's main activity (for example, `MainActivity.kt`):

```kotlin
package dev.flutter.example

import io.flutter.embedding.android.FlutterActivity
import io.flutter.embedding.engine.FlutterEngine

class MainActivity : FlutterActivity() {
    override fun configureFlutterEngine(flutterEngine: FlutterEngine) {
        super.configureFlutterEngine(flutterEngine)
        flutterEngine
                .platformViewsController
                .registry
                .registerViewFactory("<platform-view-type>",
                                     NativeViewFactory())
    }
}
```

For plugin registration, modify the plugin's main class (for example, `PlatformViewPlugin.kt`):

```kotlin
package dev.flutter.plugin.example

import io.flutter.embedding.engine.plugins.FlutterPlugin
import io.flutter.embedding.engine.plugins.FlutterPlugin.FlutterPluginBinding

class PlatformViewPlugin : FlutterPlugin {
    override fun onAttachedToEngine(binding: FlutterPluginBinding) {
        binding
                .platformViewRegistry
                .registerViewFactory("<platform-view-type>", NativeViewFactory())
    }

    override fun onDetachedFromEngine(binding: FlutterPluginBinding) {}
}
```

For more information, see the API docs for:

- `FlutterPlugin`
- `PlatformViewRegistry`
- `PlatformViewFactory`
- `PlatformView`

Finally, modify your `build.gradle` file to require one of the minimal Android SDK versions:

```kotlin
android {
    defaultConfig {
        minSdk = 19 // if using hybrid composition
        minSdk = 20 // if using virtual display.
    }
}
```

Surface Views

Handling SurfaceViews is problematic for Flutter and should be avoided when possible.

Manual view invalidation

Certain Android Views do not invalidate themselves when their content changes. Some example views include `SurfaceView` and `SurfaceTexture`. When your Platform View includes these views you are required to manually invalidate the view after they have been drawn to (or more specifically: after the swap chain is flipped). Manual view invalidation is done by calling `invalidate` on the View or one of its parent views.

Issues

Existing Platform View issues

Performance

Platform views in Flutter come with performance trade-offs.

For example, in a typical Flutter app, the Flutter UI is composed on a dedicated raster thread. This allows Flutter apps to be fast, as the main platform thread is rarely blocked.

While a platform view is rendered with hybrid composition, the Flutter UI is composed from the platform thread, which competes with other tasks like handling OS or plugin messages.

Prior to Android 10, hybrid composition copied each Flutter frame out of the graphic memory into main memory, and then copied it back to a GPU texture. As this copy happens per frame, the performance of the entire Flutter UI might be impacted. In Android 10 or above, the graphics memory is copied only once.

Virtual display, on the other hand, makes each pixel of the native view flow through additional intermediate graphic buffers, which cost graphic memory and drawing performance.

For complex cases, there are some techniques that can be used to mitigate these issues.

For example, you could use a placeholder texture while an animation is happening in Dart. In other words, if an animation is slow while a platform view is rendered, then consider taking a screenshot of the native view and rendering it as a texture.

For more information, see:

- `TextureLayer`
- `TextureRegistry`
- `FlutterTextureRegistry`
- `FlutterImageView`