Hybrid composition refers to the ability of composing native views alongside Flutter widgets. For example, displaying the native Webview inside a Flutter app.

Android

Requires API level 19

Starting from Flutter 1.20.0, hybrid composition can be used on Android. This new feature fixes most of the issues with the existing platform view approach. In particular, accessibility and keyboard related issues.

Dart side

To start using this feature, you would need to create a `Widget`, and add the following `build` implementation:

`native_view_example.dart`

1. Add imports:

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

2. Implement `build` method:

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

  return PlatformViewLink(
    viewType: viewType,
    surfaceFactory:
        (BuildContext context, PlatformViewController controller) {
      return AndroidViewSurface(
        controller: controller,
        gestureRecognizers: const <Factory<OneSequenceGestureRecognizer>>{},
        hitTestBehavior: PlatformViewHitTestBehavior.opaque,
      );
    },
    onCreatePlatformView: (PlatformViewCreationParams params) {
      return PlatformViewsService.initSurfaceAndroidView(
        id: params.id,
        viewType: viewType,
        layoutDirection: TextDirection.ltr,
```

```
        creationParams: creationParams,
        creationParamsCodec: StandardMessageCodec(),
      )
        ..addOnPlatformViewCreatedListener(params.onPlatformViewCreated)
        ..create();
    },
  );
}
```

For more documentation see: PlatformViewLink, AndroidViewSurface, PlatformViewsService.

Platform side

Finally, on the platform side, you use the standard `io.flutter.plugin.platform` package in Java or Kotlin:

`NativeView.java`

```java
package dev.flutter.example;

import android.content.Context;
import android.graphics.Color;
import android.view.View;
import android.widget.TextView;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import io.flutter.plugin.platform.PlatformView;

class NativeView implements PlatformView {
   @NonNull private final TextView textView;

    NativeView(@NonNull Context context, int id, @Nullable Map<String, Object> creationParam
        textView = new TextView(context);
        textView.setTextSize(72);
        textView.setBackgroundColor(Color.rgb(255, 255, 255));
        textView.setText("Rendered on a native Android view (id: " + id + ")");
    }

    @NonNull
    @Override
    public View getView() {
        return textView;
    }

    @Override
    public void dispose() {}
```

```
}
```

NativeViewFactory.java

```
package dev.flutter.example;

import android.content.Context;
import android.view.View;
import androidx.annotation.Nullable;
import androidx.annotation.NonNull;
import io.flutter.plugin.common.BinaryMessenger;
import io.flutter.plugin.common.StandardMessageCodec;
import io.flutter.plugin.platform.PlatformView;
import io.flutter.plugin.platform.PlatformViewFactory;
import java.util.Map;

class NativeViewFactory extends PlatformViewFactory {
  @NonNull private final BinaryMessenger messenger;
  @NonNull private final View containerView;

  NativeViewFactory(@NonNull BinaryMessenger messenger, @NonNull View containerView) {
    super(StandardMessageCodec.INSTANCE);
    this.messenger = messenger;
    this.containerView = containerView;
  }

  @NonNull
  @Override
  public PlatformView create(@NonNull Context context, int id, @Nullable Object args) {
    final Map<String, Object> creationParams = (Map<String, Object>) args;
    return new NativeView(context, id, creationParams);
  }
}
```

Register the platform view. This can be done in an app or a plugin.

For app registration, modify the main activity (e.g. `MainActivity.java`):

```
package dev.flutter.example;

import androidx.annotation.NonNull;
import io.flutter.embedding.android.FlutterActivity;
import io.flutter.embedding.engine.FlutterEngine;

public class MainActivity extends FlutterActivity {
    @Override
    public void configureFlutterEngine(@NonNull FlutterEngine flutterEngine) {
        flutterEngine
```

```
            .getPlatformViewsController()
            .getRegistry()
            .registerViewFactory("<platform-view-type>", new NativeViewFactory());
    }
}
```

For plugin registration, modify the main plugin file (e.g. `PlatformViewPlugin.java`):

```
package dev.flutter.plugin.example;

import androidx.annotation.NonNull;
import io.flutter.embedding.engine.plugins.FlutterPlugin;
import io.flutter.plugin.common.BinaryMessenger;

public class PlatformViewPlugin implements FlutterPlugin {
  @Override
  public void onAttachedToEngine(@NonNull FlutterPluginBinding binding) {
    binding
        .getFlutterEngine()
        .getPlatformViewsController()
        .getRegistry()
        .registerViewFactory("<platform-view-type>", new NativeViewFactory());
  }
}
```

For more documentation, see PlatformViewRegistry, PlatformViewFactory, and PlatformView.

Finally, indicate the minimum API Level required for the application to run in `build.gradle`.

```
android {
    defaultConfig {
        minSdkVersion 19
    }
}
```

iOS

In Flutter 1.22, platform views are enabled by default. This means that it's no longer required to add the `io.flutter.embedded_views_preview` flag to `Info.plist`.

To create a platform view on iOS, follow these steps:

On the Dart side

On the Dart side, create a `Widget` and add the following build implementation, as shown in the following steps.

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

1. Add the following imports:

```
import 'package:flutter/widget.dart';
```

2. Implement a `build()` method:

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

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

For more information, see the API docs for: `UIKitView`.

On the platform side

In your native code, implement the following:

`FLNativeView.h`

```
#import <Flutter/Flutter.h>

@interface FLNativeViewFactory : NSObject <FlutterPlatformViewFactory>
- (instancetype)initWithMessenger:(NSObject<FlutterBinaryMessenger>*)messenger;
@end

@interface FLNativeView : NSObject <FlutterPlatformView>

- (instancetype)initWithFrame:(CGRect)frame
               viewIdentifier:(int64_t)viewId
                    arguments:(id _Nullable)args
              binaryMessenger:(NSObject<FlutterBinaryMessenger>*)messenger;

- (UIView*)view;
@end
```

Implement the factory and the platform view in `FLNativeView.m`

```
#import "FLNativeView.h"
```

```
@implementation FLNativeViewFactory {
  NSObject<FlutterBinaryMessenger>* _messenger;
}

- (instancetype)initWithMessenger:(NSObject<FlutterBinaryMessenger>*)messenger {
  self = [super init];
  if (self) {
    _messenger = messenger;
  }
  return self;
}

- (NSObject<FlutterPlatformView>*)createWithFrame:(CGRect)frame
                                   viewIdentifier:(int64_t)viewId
                                        arguments:(id _Nullable)args {
  return [[FLNativeView alloc] initWithFrame:frame
                              viewIdentifier:viewId
                                   arguments:args
                             binaryMessenger:_messenger];
}

@end

@implementation FLNativeView {
   UIView *_view;
}

- (instancetype)initWithFrame:(CGRect)frame
               viewIdentifier:(int64_t)viewId
                    arguments:(id _Nullable)args
              binaryMessenger:(NSObject<FlutterBinaryMessenger>*)messenger {
  if (self = [super init]) {
    _view = [[UIView alloc] init];
  }
  return self;
}

- (UIView*)view {
  return _view;
}

@end
```

Finally, register the platform view. This can be done in an app or a plugin.

For app registration, modify the App's `AppDelegate.m`:

```
#import "AppDelegate.h"
#import "FLNativeView.h"
#import "GeneratedPluginRegistrant.h"

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
  [GeneratedPluginRegistrant registerWithRegistry:self];

   NSObject<FlutterPluginRegistrar>* registrar =
      [self registrarForPlugin:@"plugin-name"];

  FLNativeViewFactory* factory =
      [[FLNativeViewFactory alloc] initWithMessenger:registrar.messenger];
  [[self registrarForPlugin:@"<plugin-name>"] registerViewFactory:factory
                                                           withId:@"<platform-view-type>"];
  return [super application:application didFinishLaunchingWithOptions:launchOptions];
}

@end
```

For plugin registration, modify the main plugin file (e.g. `FLPlugin.m`):

```
#import "FLPlugin.h"
#import "FLNativeView.h"

@implementation FLPlugin

+ (void)registerWithRegistrar:(NSObject<FlutterPluginRegistrar>*)registrar {
  FLNativeViewFactory* factory =
      [[FLNativeViewFactory alloc] initWithMessenger:registrar.messenger];
  [registrar registerViewFactory:factory withId:@"<platform-view-type>"];
}

@end
```

For more information, see the API docs for:

- `FlutterPlatformViewFactory`
- `FlutterPlatformView`
- `PlatformView`

By default, the `UIKitView` widget appends the native `UIView` to the view hierarchy. For more documentation, see UIKitView.

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, the Flutter UI is composed from the platform thread, which competes with other tasks like handling OS or plugin messages, etc.

Prior to Android 10, Hybrid composition copies each Flutter frame out of the graphic memory into main memory and then copied back to a GPU texture. As this copy happens per frame, the performance of the entire Flutter UI may be impacted.

On the other hand, Virtual display 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`