If you `flutter create`d your project prior to version 1.12, this may apply to your project

Background

In order to better support the execution environments of adding Flutter to an existing project, the old Android platform-side wrappers hosting the Flutter runtime at `io.flutter.app.FlutterActivity` and their associated classes are now deprecated. New wrappers at `io.flutter.embedding.android.FlutterActivity` and associated classes now replace them.

Those classes better support real world scenarios where the FlutterActivity isn't the first and only Android Activity in an application.

Motivation

Your existing full-Flutter projects aren't immediately affected and will continue to work as before for the foreseeable future.

However, the new Android wrappers also introduce a new set of Android plugin development APIs. Plugins developed exclusively on the new plugins API will not work on older pre-1.12 Android projects. Building a pre-1.12 Android project that uses plugins created after 1.12 will yield a build-time error unless the plugin developer explicitly opted to create a second backward compatible implementation.

Add-to-app was not officially supported on the old Android APIs. If you followed the experimental instructions in the wiki for add-to-app prior to 1.12, you should follow the migration steps under the add-to-app section below.

Full-Flutter app migration

This guide assumes you haven't manually modified your Android host project for your Flutter project. If you did, consult the add-to-app migration guide below.

If you opt to migrate your standard `flutter create`d project, follow the following steps:

1a. If you don't have any of your own added code to `android/app/src/main/java/[your/package/name]/MainActivity.java` - remove the body of your `MainActivity.java` and change the `FlutterActivity` import. The new `FlutterActivity` no longer requires manually registering your plugins. It will now perform the registration automatically when the underlaying `FlutterEngine` is created.

```
// MainActivity.java
-import android.os.Bundle;
-import io.flutter.app.FlutterActivity;
+import io.flutter.embedding.android.FlutterActivity;
-import io.flutter.plugins.GeneratedPluginRegistrant;

public class MainActivity extends FlutterActivity {
-  @Override
-  protected void onCreate(Bundle savedInstanceState) {
-    super.onCreate(savedInstanceState);
-    GeneratedPluginRegistrant.registerWith(this);
```

```
-   }
 }
```

```
// MainActivity.kt
-import android.os.Bundle
-import io.flutter.app.FlutterActivity
+import io.flutter.embedding.android.FlutterActivity
-import io.flutter.plugins.GeneratedPluginRegistrant

class MainActivity: FlutterActivity() {
-  override fun onCreate(savedInstanceState: Bundle?) {
-    super.onCreate(savedInstanceState)
-    GeneratedPluginRegistrant.registerWith(this)
-  }
}
```

Since the body of the `MainActivity` is now empty, you can also optionally delete the `MainActivity.java/kt` file if you'd like. If you do, you need to change your `AndroidManifest.xml`'s reference to `.MainActivity` to `io.flutter.embedding.android.FlutterActivity`.

1b. If you had existing custom platform channel handling code in your `MainActivity.java`, below is an example of the change you can make to adopt the new embedding API:

```
-import io.flutter.app.FlutterActivity;
-import io.flutter.plugin.common.MethodCall;
+import androidx.annotation.NonNull;
+import io.flutter.embedding.android.FlutterActivity;
+import io.flutter.embedding.engine.FlutterEngine;
 import io.flutter.plugin.common.MethodChannel;
-import io.flutter.plugin.common.MethodChannel.MethodCallHandler;
-import io.flutter.plugin.common.MethodChannel.Result;
+import io.flutter.plugins.GeneratedPluginRegistrant;

 public class MainActivity extends FlutterActivity {
     private static final String CHANNEL = "samples.flutter.dev/battery";
-
-    @Override
-    public void onCreate(Bundle savedInstanceState) {
-
-        super.onCreate(savedInstanceState);
-        GeneratedPluginRegistrant.registerWith(this);
-
-        new MethodChannel(getFlutterView(), CHANNEL).setMethodCallHandler(
-                new MethodCallHandler() {
-                    @Override
-                    public void onMethodCall(MethodCall call, Result result) {
-                        // Your existing code
-                    }
-                });
-    }
```

```
+
+    @Override
+    public void configureFlutterEngine(@NonNull FlutterEngine flutterEngine) {
+        GeneratedPluginRegistrant.registerWith(flutterEngine);
+        new MethodChannel(flutterEngine.getDartExecutor().getBinaryMessenger(),
CHANNEL)
+                .setMethodCallHandler(
+                    (call, result) -> {
+                        // Your existing code
+                }
+        );
+    }
 }
```

In other words, move the channel registration part of the code in your `onCreate` into the `configureFlutterEngine` override of the FlutterActivity subclass and use `flutterEngine.getDartExecutor().getBinaryMessenger()` as the binary messenger rather than `getFlutterView()`.

2. Open `android/app/src/main/AndroidManifest.xml`.

3. Replace the reference to `FlutterApplication` in the application tag with `${applicationName}`.

Previous configuration:

```
<application
  android:name="io.flutter.app.FlutterApplication"
  >
  <!-- code omitted -->
</application>
```

New configuration:

```
<application
  android:name="${applicationName}"
  >
  <!-- code omitted -->
</application>
```

4. Update splash screen behavior (if splash behavior is desired).

Remove all `<meta-data>` tags with key `android:name="io.flutter.app.android.SplashScreenUntilFirstFrame"`.

Add a launch theme to `styles.xml` that configures the desired launch screen as a background `Drawable`:

```
<!-- You can name this style whatever you'd like -->
<style name="LaunchTheme" parent="@android:style/Theme.Black.NoTitleBar">
    <item name="android:windowBackground">@drawable/[your_launch_drawable_here]
</item>
</style>
```

If you created your Flutter project with `flutter create` then you likely already have a `LaunchTheme` defined, along with a drawable called `launch_background`. You can re-use that configuration and adjust it as desired.

Add a normal theme that to `styles.xml` that should replace the launch screen when the Android process is fully initialized:

```
<!-- You can name this style whatever you'd like -->
<style name="NormalTheme" parent="@android:style/Theme.Black.NoTitleBar">
    <item
name="android:windowBackground">@drawable/[your_normal_background_drawable]</item>
</style>
```

The "normal theme" draws the background behind your Flutter experience. That background is typically seen for a brief moment just before the first Flutter frame renders. The "normal theme" also controls Android's status bar and navigation bar visual properties for the duration of your Flutter experience.

Configure `MainActivity` to start with your launch theme and then shift to your normal theme. Also specify that you want your launch screen to continue being displayed until Flutter renders its first frame:

```
<activity android:name=".MainActivity"
  android:theme="@style/LaunchTheme"
  // some code omitted
  >
  <!-- Specify that the launch screen should continue being displayed -->
  <!-- until Flutter renders its first frame. -->
  <meta-data
    android:name="io.flutter.embedding.android.SplashScreenDrawable"
    android:resource="@drawable/launch_background" />

  <!-- Theme to apply as soon as Flutter begins rendering frames -->
  <meta-data
    android:name="io.flutter.embedding.android.NormalTheme"
    android:resource="@style/NormalTheme"
    />

  <!-- some code omitted -->
</activity>
```

5. Add a new `<meta-data>` tag under `<application>`.

```
<meta-data
    android:name="flutterEmbedding"
    android:value="2" />
```

Once you make this declaration in your AndroidManifest and also use plugins, the new GeneratedPluginRegistrants created by the Flutter tools during build will now be able to also use plugins that uses the new Android embedding (which registers the plugins against a `FlutterEngine` instead of a `PluginRegistry.Registrar`).

Your app should still build as normal (such as via `flutter build apk`) but you're now using the new Android classes.

Add-to-app migration

This section details how to take add-to-app scenarios that were built using Flutter's experimental embedding, and transition that code to Flutter's new stable embedding.

Same steps as full-Flutter

Some instructions from the "Full-Flutter app migration" section above still apply. Follow the above steps for:

3. Remove the reference to `FlutterApplication` from the application tag.
4. Update splash screen behavior (if splash behavior is desired).
5. Add a new `<meta-data>` tag under `<application>`.

Changes specific to add-to-app

If you invoke `FlutterMain.startInitialization(...)` or `FlutterMain.ensureInitializationComplete(...)` anywhere in your code, you should remove those calls. Flutter now initializes itself at the appropriate time.

Migrating FlutterActivity Uses

Add-to-app scenarios often involve modifications to subclasses of `FlutterActivity`. For example, such a scenario might introduce new `MethodChannel`s, a custom `FlutterEngine` instance, custom splash screen behavior, or other behaviors that require overriding existing methods. Therefore, whereas full-Flutter apps can delete their `MainActivity` and replace it with a standard `FlutterActivity`, you will need to retain your subclass so that you can keep your behavior overrides.

If your add-to-app use-cases do not modify behavior within `FlutterActivity`, you should delete your subclasses and replace them with standard `FlutterActivity`s as described in the previous section.

If your add-to-app use-cases do require modifying behavior within `FlutterActivity`, you need to migrate your code from the old `io.flutter.app.FlutterActivity` to the new `io.flutter.embedding.android.FlutterActivity`.

From:

```
package [your.package.name];

import android.os.Bundle;
import io.flutter.app.FlutterActivity;
import io.flutter.plugins.GeneratedPluginRegistrant;

public class MainActivity extends FlutterActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    GeneratedPluginRegistrant.registerWith(this);
  }
```

```
  // ...some amount of custom code for your app is here.
}
```

To:

```
package [your.package.name];

import io.flutter.embedding.android.FlutterActivity;

public class MainActivity extends FlutterActivity {
  // You do not need to override onCreate() in order to invoke
  // GeneratedPluginRegistrant. Flutter now does that on your behalf.

  // ...retain whatever custom code you had from before (if any).
}
```

Some apps may have required pre-warming a Flutter experience. It is now recommended that all add-to-app use-cases pre-warm Flutter experiences to achieve optimal visual performance when initially rendering a Flutter UI. Please refer to the Use a cached FlutterEngine to update your code for pre-warming Flutter.

Your `FlutterActivity` subclass is now up to date with the new, stable Android embedding.

Migrating FlutterFragment uses

The experimental embedding provides a class called `io.flutter.facade.FlutterFragment`, along with other classes in the `io.flutter.facade` package. The entire `io.flutter.facade` package is deprecated and you should not use any classes in that package.

The experimental `io.flutter.facade.FlutterFragment` has been replaced by `io.flutter.embedding.android.FlutterFragment`, which was designed for a much broader set of use-cases than the original `FlutterFragment`.

If you are instantiating a `io.flutter.facade.FlutterFragment` via `Flutter.createFragment(...)`, you should delete any such calls and instantiate the new `io.flutter.embedding.android.FlutterFragment` via one of the following factory methods:

- `FlutterFragment.createDefault()`
- `FlutterFragment.withNewEngine()`
- `FlutterFragment.withCachedEngine(...)`

The use of these factory methods are discussed in depth in website guides at http://flutter.dev.

Migrating FlutterView uses

The deprecated `io.flutter.facade.Flutter` class has a factory method called `createView(...)`. This method is deprecated, along with all other code in the `io.flutter.facade` package.

Flutter does not currently provide convenient APIs for utilizing Flutter at the `View` level, so the use of a `FlutterView` should be avoided, if possible. However, it is technically feasible to display a `FlutterView`, if required. Be sure to use `io.flutter.embedding.android.FlutterView` instead of

`io.flutter.view.FlutterView`. You can instantiate the new `FlutterView` just like any other Android `View`. Then, follow instructions in the associated Javadocs to display Flutter via a `FlutterView`.