Supporting the new Android plugins APIs

Contents

Note: You might be directed to this page if the framework detects that your app uses a plugin based on the old Android APIs.

If you don't write or maintain an Android Flutter plugin, you can skip this page.

As of the 1.12 release, new plugin APIs are available for the Android platform. The old APIs based on `PluginRegistry.Registra` won't be immediately deprecated, but we encourage you to migrate to the new APIs based on `FlutterPlugin`.

The new API has the advantage of providing a cleaner set of accessors for lifecycle dependent components compared to the old APIs. For instance `PluginRegistry.Registrar.activity()` could return null if Flutter isn't attached to any activities.

In other words, plugins using the old API may produce undefined behaviors when embedding Flutter into an Android app. Most of Flutter plugins provided by the flutter.dev team have been migrated already. (Learn how to become a verified publisher on pub.dev For an example of a plugin that uses the new APIs, see the battery package.

Upgrade steps

The following instructions outline the steps for supporting the new API:

1. Update the main plugin class (`*Plugin.java`) to implement the `FlutterPlugin` interface. For more complex plugins, you ca separate the `FlutterPlugin` and `MethodCallHandler` into two classes. See the next section, Basic plugin, for more details accessing app resources with the latest version (v2) of embedding.

   Also, note that the plugin should still contain the static `registerWith()` method to remain compatible with apps that don't the v2 Android embedding. (See Upgrading pre 1.12 Android projects for details.) The easiest thing to do (if possible) is mo the logic from `registerWith()` into a private method that both `registerWith()` and `onAttachedToEngine()` can call. Eithe `registerWith()` or `onAttachToEngine()` will be called, not both.

   In addition, you should document all non-overridden public members within the plugin. In an add-to-app scenario, these clas are accessible to a developer and require documentation.

2. (Optional) If your plugin needs an `Activity` reference, also implement the `ActivityAware` interface.

3. (Optional) If your plugin is expected to be held in a background Service at any point in time, implement the `ServiceAware` interface.

4. Update the example app's `MainActivity.java` to use the v2 embedding `FlutterActivity`. For details, see Upgrading pre 1 Android projects. You may have to make a public constructor for your plugin class if one didn't exist already. For example:

```
package io.flutter.plugins.firebasecoreexample;

import io.flutter.embedding.android.FlutterActivity;
import io.flutter.embedding.engine.FlutterEngine;
import io.flutter.plugins.firebase.core.FirebaseCorePlugin;

public class MainActivity extends FlutterActivity {
  // You can keep this empty class or remove it. Plugins on the new embedding
  // now automatically registers plugins.
}
```

5. (Optional) If you removed `MainActivity.java`, update the `<plugin_name>/example/android/app/src/main/AndroidManifest.xml` to use `io.flutter.embedding.android.FlutterActivity`. For example:

```
  <activity android:name="io.flutter.embedding.android.FlutterActivity"
          android:theme="@style/LaunchTheme"
android:configChanges="orientation|keyboardHidden|keyboard|screenSize|locale|layoutDirection|fontScale
          android:hardwareAccelerated="true"
          android:windowSoftInputMode="adjustResize">
          <meta-data
              android:name="io.flutter.app.android.SplashScreenUntilFirstFrame"
              android:value="true" />
          <intent-filter>
              <action android:name="android.intent.action.MAIN"/>
              <category android:name="android.intent.category.LAUNCHER"/>
          </intent-filter>
      </activity>
```

6. (Optional) Create an `EmbeddingV1Activity.java` file] that uses the v1 embedding for the example project in the same folde `MainActivity` to keep testing the v1 embedding's compatibility with your plugin. Note that you have to manually register all plugins instead of using `GeneratedPluginRegistrant`. For example:

```
package io.flutter.plugins.batteryexample;

import android.os.Bundle;
import dev.flutter.plugins.e2e.E2EPlugin;
import io.flutter.app.FlutterActivity;
import io.flutter.plugins.battery.BatteryPlugin;

public class EmbeddingV1Activity extends FlutterActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    BatteryPlugin.registerWith(registrarFor("io.flutter.plugins.battery.BatteryPlugin"));
    E2EPlugin.registerWith(registrarFor("dev.flutter.plugins.e2e.E2EPlugin"));
  }
}
```

7. Add `<meta-data android:name="flutterEmbedding" android:value="2"/>` to the `<plugin_name>/example/android/app/src/main/AndroidManifest.xml`. This sets the example app to use the v2 embedd

8. (Optional) If you created an `EmbeddingV1Activity` in the previoous step, add the `EmbeddingV1Activity` to the `<plugin_name>/example/android/app/src/main/AndroidManifest.xml` file. For example:

```
 <activity
     android:name=".EmbeddingV1Activity"
     android:theme="@style/LaunchTheme"

android:configChanges="orientation|keyboardHidden|keyboard|screenSize|locale|layoutDirection|fontScale
     android:hardwareAccelerated="true"
     android:windowSoftInputMode="adjustResize">
 </activity>
```

Testing your plugin

The remaining steps address testing your plugin, which we encourage, but aren't required.

1. Update `<plugin_name>/example/android/app/build.gradle` to replace references to `android.support.test` with `androidx.test`:

```
defaultConfig {
  ...
  testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
  ...
}
```

```
dependencies {
...
androidTestImplementation 'androidx.test:runner:1.2.0'
androidTestImplementation 'androidx.test:rules:1.2.0'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
...
}
```

2. Add tests files for `MainActivity` and `EmbeddingV1Activity` in `<plugin_name>/example/android/app/src/androidTest/java/<plugin_path>/`. You will need to create these directories example:

```
package io.flutter.plugins.firebase.core;

import androidx.test.rule.ActivityTestRule;
import dev.flutter.plugins.e2e.FlutterRunner;
import io.flutter.plugins.firebasecoreexample.MainActivity;
import org.junit.Rule;
import org.junit.runner.RunWith;

@RunWith(FlutterRunner.class)
public class MainActivityTest {
  // Replace `MainActivity` with `io.flutter.embedding.android.FlutterActivity` if you removed
`MainActivity`.
  @Rule public ActivityTestRule<MainActivity> rule = new ActivityTestRule<>(MainActivity.class);
}
```

```
package io.flutter.plugins.firebase.core;

import androidx.test.rule.ActivityTestRule;
import dev.flutter.plugins.e2e.FlutterRunner;
import io.flutter.plugins.firebasecoreexample.EmbeddingV1Activity;
import org.junit.Rule;
import org.junit.runner.RunWith;

@RunWith(FlutterRunner.class)
public class EmbeddingV1ActivityTest {
  @Rule
  public ActivityTestRule<EmbeddingV1Activity> rule =
      new ActivityTestRule<>(EmbeddingV1Activity.class);
}
```

3. Add `e2e` and `flutter_driver` dev_dependencies to `<plugin_name>/pubspec.yaml` and `<plugin_name>/example/pubspec.yaml`.

```
e2e: ^0.2.1
flutter_driver:
  sdk: flutter
```

4. Update minimum Flutter version of environment in `<plugin_name>/pubspec.yaml`. All plugins moving forward will set the minimum version to 1.12.13+hotfix.6 which is the minimum version for which we can guarantee support. For example:

```
environment:
  sdk: ">=2.0.0-dev.28.0 <3.0.0"
  flutter: ">=1.12.13+hotfix.6 <2.0.0"
```

5. Create a simple test in `<plugin_name>/test/<plugin_name>_e2e.dart`. For the purpose of testing the PR that adds the v2 embedding support, we're trying to test some very basic functionality of the plugin. This is a smoke test to ensure that the plugin properly registers with the new embedder. For example:

```
import 'package:flutter_test/flutter_test.dart';
import 'package:battery/battery.dart';
import 'package:e2e/e2e.dart';

void main() {
  E2EWidgetsFlutterBinding.ensureInitialized();

  testWidgets('Can get battery level', (WidgetTester tester) async {
    final Battery battery = Battery();
    final int batteryLevel = await battery.batteryLevel;
    expect(batteryLevel, isNotNull);
  });
}
```

6. Test run the e2e tests locally. In a terminal, do the following:

```
cd <plugin_name>/example
flutter build apk
cd android
./gradlew app:connectedAndroidTest -Ptarget=`pwd`/../../test/<plugin_name>_e2e.dart
```

Basic plugin

To get started with a Flutter Android plugin in code, start by implementing `FlutterPlugin`.

```
public class MyPlugin implements FlutterPlugin {
  @Override
  public void onAttachedToEngine(@NonNull FlutterPluginBinding binding) {
    // TODO: your plugin is now attached to a Flutter experience.
  }

  @Override
  public void onDetachedFromEngine(@NonNull FlutterPluginBinding binding) {
    // TODO: your plugin is no longer attached to a Flutter experience.
  }
}
```

As shown above, your plugin may or may not be associated with a given Flutter experience at any given moment in time. You sho take care to initialize your plugin's behavior in `onAttachedToEngine()`, and then cleanup your plugin's references in `onDetachedFromEngine()`.

The FlutterPluginBinding provides your plugin with a few important references:

binding.getFlutterEngine()
Returns the `FlutterEngine` that your plugin is attached to, providing access to components like the `DartExecutor`, `FlutterRenderer`, and more.

binding.getApplicationContext()
Returns the Android application's `Context` for the running app.

UI/Activity plugin

If your plugin needs to interact with the UI, such as requesting permissions, or altering Android UI chrome, then you need to take additional steps to define your plugin. You must implement the `ActivityAware` interface.

```
public class MyPlugin implements FlutterPlugin, ActivityAware {
  //...normal plugin behavior is hidden...

  @Override
  public void onAttachedToActivity(ActivityPluginBinding activityPluginBinding) {
    // TODO: your plugin is now attached to an Activity
  }

  @Override
  public void onDetachedFromActivityForConfigChanges() {
    // TODO: the Activity your plugin was attached to was
    // destroyed to change configuration.
    // This call will be followed by onReattachedToActivityForConfigChanges().
  }

  @Override
  public void onReattachedToActivityForConfigChanges(ActivityPluginBinding activityPluginBinding) {
    // TODO: your plugin is now attached to a new Activity
    // after a configuration change.
  }

  @Override
  public void onDetachedFromActivity() {
    // TODO: your plugin is no longer associated with an Activity.
    // Clean up references.
  }
}
```

To interact with an `Activity`, your `ActivityAware` plugin must implement appropriate behavior at 4 stages. First, your plugin is attached to an `Activity`. You can access that `Activity` and a number of its callbacks through the provided `ActivityPluginBinding`.

Since `Activity`s can be destroyed during configuration changes, you must cleanup any references to the given `Activity` in `onDetachedFromActivityForConfigChanges()`, and then re-establish those references in `onReattachedToActivityForConfigChanges()`.

Finally, in `onDetachedFromActivity()` your plugin should clean up all references related to `Activity` behavior and return to a nor configuration.