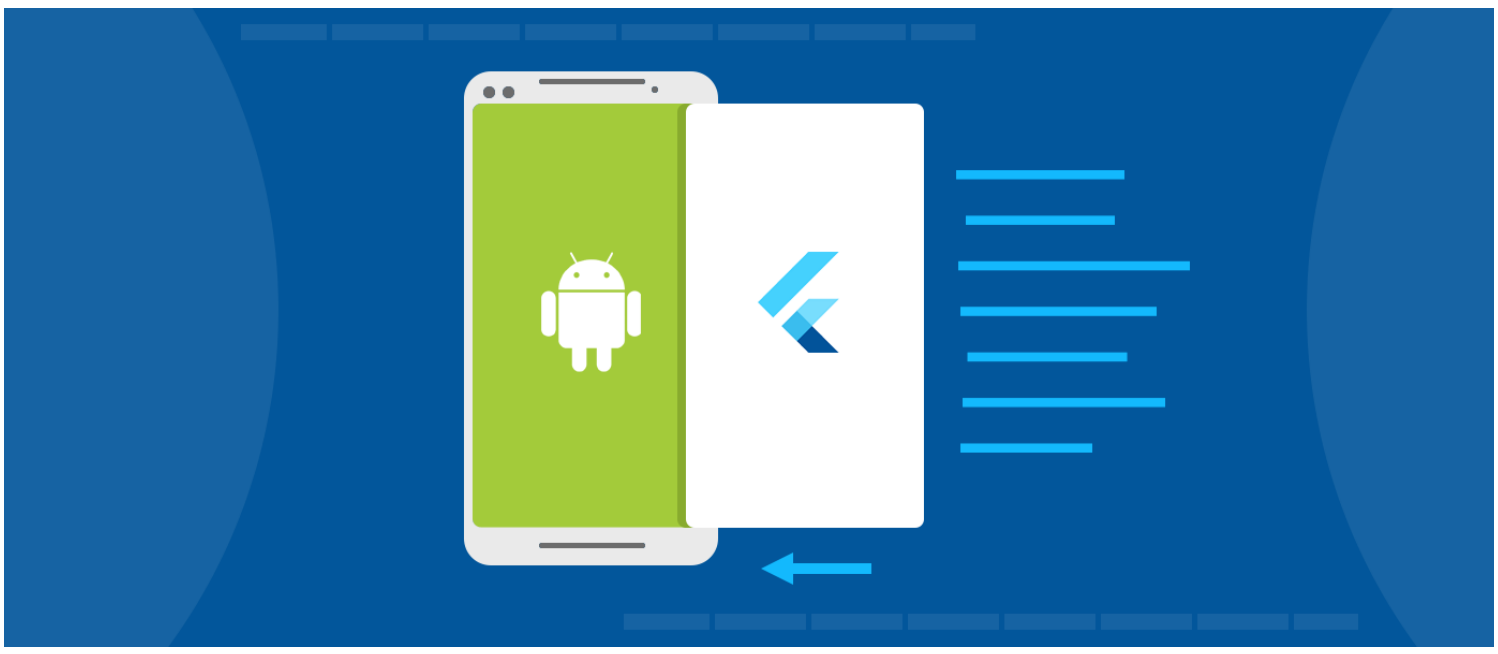
Adding a Flutter screen to an Android app

Contents

- Add a normal Flutter screen
  - Step 1: Add FlutterActivity to AndroidManifest.xml
  - Step 2: Launch FlutterActivity
  - Step 3: (Optional) Use a cached FlutterEngine
    - Initial route with a cached engine
- Add a translucent Flutter screen
  - Step 1: Use a theme with translucency
  - Step 2: Start FlutterActivity with transparency

This guide describes how to add a single Flutter screen to an existing Android app. A Flutter screen can be added as a normal, opaque screen, or as a see-through, translucent screen. Both options are described in this guide.

Add a normal Flutter screen



Step 1: Add FlutterActivity to AndroidManifest.xml

Flutter provides `FlutterActivity` to display a Flutter experience within an Android app. Like any other `Activity`, `FlutterActivi` must be registered in your `AndroidManifest.xml`. Add the following XML to your `AndroidManifestxml` file under your `applicati` tag:

```
<activity
  android:name="io.flutter.embedding.android.FlutterActivity"
  android:theme="@style/LaunchTheme"

android:configChanges="orientation|keyboardHidden|keyboard|screenSize|locale|layoutDirection|fontScale|scr
enLayout|density|uiMode"
  android:hardwareAccelerated="true"
  android:windowSoftInputMode="adjustResize"
  />
```

The reference to `@style/LaunchTheme` can be replaced by any Android theme that want to apply to your `FlutterActivity`. The choice of theme dictates the colors applied to Android's system chrome, like Android's navigation bar, and to the background colo the `FlutterActivity` just before the Flutter UI renders itself for the first time.

Step 2: Launch FlutterActivity

With `FlutterActivity` registered in your manifest file, add code to launch `FlutterActivity` from whatever point in your app tha you'd like. The following example shows `FlutterActivity` being launched from an `OnClickListener`.

Java Kotlin

Get started
Samples & tutorials
Development
User interface
Data & backend
Accessibility & internationalization
Platform integration
Packages & plugins
Add Flutter to existing app
Introduction
Adding to an Android app
Project setup
Add a single Flutter screen
Add a Flutter Fragment
Adding to an iOS app
Running, debugging & hot reload
Loading sequence and performance
Tools & techniques
Migration notes
Testing & debugging
Performance & optimization
Deployment
Resources
Reference
Widget index
API reference
Package site

```
myButton.setOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View v) {
    startActivity(
      FlutterActivity.createDefaultIntent(currentActivity)
    );
  }
});
```

The previous example assumes that your Dart entrypoint is called `main()`, and your initial Flutter route is '/'. The Dart entrypoint c be changed using `Intent`, but the initial route can be changed using `Intent`. The following example demonstrates how to launch `FlutterActivity` that initially renders a custom route in Flutter.

Java Kotlin

```
myButton.addOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View v) {
    startActivity(
      FlutterActivity
        .withNewEngine()
        .initialRoute("/my_route")
        .build(currentActivity)
    );
  }
});
```

Replace `"/my_route"` with your desired initial route.

The use of the `withNewEngine()` factory method configures a `FlutterActivity` that internally create its own `FlutterEngine` instance. This comes with a non-trivial initialization time. The alternative approach is to instruct `FlutterActivity` to use a pre-warmed, cached `FlutterEngine`, which minimizes Flutter's initialization time. That approach is discussed next.

Step 3: (Optional) Use a cached FlutterEngine

Every `FlutterActivity` creates its own `FlutterEngine` by default. Each `FlutterEngine` has a non-trivial warm-up time. This mea that launching a standard `FlutterActivity` comes with a brief delay before your Flutter experience becomes visible. To minimiz this delay, you can warm up a `FlutterEngine` before arriving at your `FlutterActivity`, and then you can use your pre-warmed `FlutterEngine` instead.

To pre-warm a `FlutterEngine`, find a reasonable location in your app to instantiate a `FlutterEngine`. The following example arbitrarily pre-warms a `FlutterEngine` in the `Application` class:

Java Kotlin

```
public class MyApplication extends Application {
  @Override
  public void onCreate() {
    super.onCreate();
    // Instantiate a FlutterEngine.
    flutterEngine = new FlutterEngine(this);

    // Start executing Dart code to pre-warm the FlutterEngine.
    flutterEngine.getDartExecutor().executeDartEntrypoint(
      DartEntrypoint.createDefault()
    );

    // Cache the FlutterEngine to be used by FlutterActivity.
    FlutterEngineCache
      .getInstance()
      .put("my_engine_id", flutterEngine);
  }
}
```

The ID passed to the `FlutterEngineCache` can be whatever you want. Make sure that you pass the same ID to any `FlutterActii` or `FlutterFragment` that should use the cached `FlutterEngine`. Using `FlutterActivity` with a cached `FlutterEngine` is discus next.

> **Note:** To warm up a `FlutterEngine`, you must execute a Dart entrypoint. Keep in mind that the moment `executeDartEntrypoint()` is invoked, your Dart entrypoint method begins executing. If your Dart entrypoint invokes `runApp(` to run a Flutter app, then your Flutter app behaves as if it were running in a window of zero size until this `FlutterEngine` is attached to a `FlutterActivity`, `FlutterFragment`, or `FlutterView`. Make sure that your app behaves appropriately between the time you warm it up and the time you display Flutter content.

With a pre-warmed, cached `FlutterEngine`, you now need to instruct your `FlutterActivity` to use the cached `FlutterEngine` instead of creating a new one. To accomplish this, use `FlutterActivity`'s `withCachedEngine()` builder:

Get started
Samples & tutorials
Development
User interface
Data & backend
Accessibility & internationalization
Platform integration
Packages & plugins
Add Flutter to existing app
Introduction
Adding to an Android app
Project setup
Add a single Flutter screen
Add a Flutter Fragment
Adding to an iOS app
Running, debugging & hot reload
Loading sequence and performance
Tools & techniques
Migration notes
Testing & debugging
Performance & optimization
Deployment
Resources
Reference
Widget index
API reference
Package site

Java Kotlin

```
myButton.addOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View v) {
    startActivity(
      FlutterActivity
        .withCachedEngine("my_engine_id")
        .build(currentActivity)
    );
  }
});
```

When using the `withCachedEngine()` factory method, pass the same ID that you used when caching the desired `FlutterEngine`.

Now, when you launch `FlutterActivity`, there is significantly less delay in the display of Flutter content.

Note: When using a cached `FlutterEngine`, that `FlutterEngine` outlives any `FlutterActivity` or `FlutterFragment` that displays it. Keep in mind that Dart code begins executing as soon as you pre-warm the `FlutterEngine`, and continues executing after the destruction of your `FlutterActivity`/`FlutterFragment`. To stop executing and clear resources, obtain your `FlutterEngine` from the `FlutterEngineCache` and destroy the `FlutterEngine` with `FlutterEngine.destroy()`.

Note: Runtime performance isn't the only reason that you might pre-warm and cache a `FlutterEngine`. A pre-warmed `FlutterEngine` executes Dart code independent from a `FlutterActivity`, which allows such a `FlutterEngine` to be used to execute arbitrary Dart code at any moment. Non-UI application logic can be executed in a `FlutterEngine`, like networking and data caching, and in background behavior within a `Service` or elsewhere. When using a `FlutterEngine` to execute behavior in the background, be sure to adhere to all Android restrictions on background execution.

Note: Flutter's debug/release builds have drastically different performance characteristics. To evaluate the performance of Flutter, use a release build.

Initial route with a cached engine

The concept of an initial route is available when configuring a `FlutterActivity` or a `FlutterFragment` with a new `FlutterEngin` However, `FlutterActivity` and `FlutterFragment` don't offer the concept of an initial route when using a cached engine. This is because a cached engine is expected to already be running Dart code, which means it's too late to configure the initial route.

Developers that would like their cached engine to begin with a custom initial route can configure their cached `FlutterEngine` to u a custom initial route just before executing the Dart entrypoint. The following example demonstrates the use of an initial route wi cached engine:

Java Kotlin

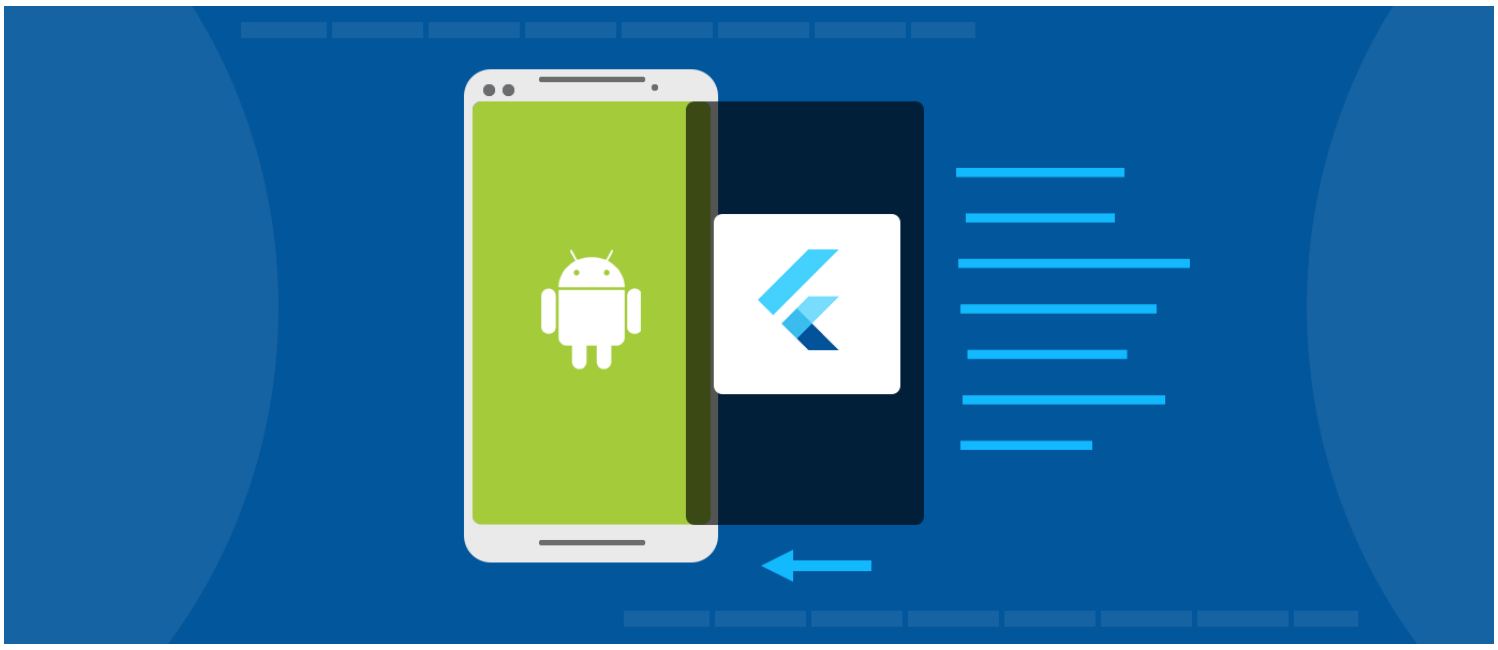
```
public class MyApplication extends Application {
  @Override
  public void onCreate() {
    super.onCreate();
    // Instantiate a FlutterEngine.
    flutterEngine = new FlutterEngine(this);
    // Configure an initial route.
    flutterEngine.getNavigationChannel().setInitialRoute("your/route/here");
    // Start executing Dart code to pre-warm the FlutterEngine.
    flutterEngine.getDartExecutor().executeDartEntrypoint(
      DartEntrypoint.createDefault()
    );
    // Cache the FlutterEngine to be used by FlutterActivity or FlutterFragment.
    FlutterEngineCache
      .getInstance()
      .put("my_engine_id", flutterEngine);
  }
}
```

By setting the initial route of the navigation channel, the associated `FlutterEngine` displays the desired route upon initial executi of the `runApp()` Dart function.

Changing the initial route property of the navigation channel after the initial execution of `runApp()` has no effect. Developers who would like to use the same `FlutterEngine` between different `Activity`s and `Fragment`s and switch the route between those disp need to setup a method channel and explicitly instruct their Dart code to change `Navigator` routes.

Add a translucent Flutter screen

Get started
Samples & tutorials
Development
User interface
Data & backend
Accessibility & internationalization
Platform integration
Packages & plugins
Add Flutter to existing app
Introduction
Adding to an Android app
Project setup
Add a single Flutter screen
Add a Flutter Fragment
Adding to an iOS app
Running, debugging & hot reload
Loading sequence and performance
Tools & techniques
Migration notes
Testing & debugging
Performance & optimization
Deployment
Resources
Reference
Widget index
API reference
Package site



Most full-screen Flutter experiences are opaque. However, some apps would like to deploy a Flutter screen that looks like a moda for example, a dialog or bottom sheet. Flutter supports translucent `FlutterActivity`s out of the box.

To make your `FlutterActivity` translucent, make the following changes to the regular process of creating and launching a `FlutterActivity`.

Step 1: Use a theme with translucency

Android requires a special theme property for `Activity`s that render with a translucent background. Create or update an Android theme with the following property:

```
<style name="MyTheme" parent="@style/MyParentTheme">
  <item name="android:windowIsTranslucent">true</item>
</style>
```

Then, apply the translucent theme to your `FlutterActivity`.

```
<activity
  android:name="io.flutter.embedding.android.FlutterActivity"
  android:theme="@style/MyTheme"

android:configChanges="orientation|keyboardHidden|keyboard|screenSize|locale|layoutDirection|fontScale|scr
enLayout|density|uiMode"
  android:hardwareAccelerated="true"
  android:windowSoftInputMode="adjustResize"
  />
```

Your `FlutterActivity` now supports translucency. Next, you need to launch your `FlutterActivity` with explicit transparency support.

Step 2: Start FlutterActivity with transparency

To launch your `FlutterActivity` with a transparent background, pass the appropriate `BackgroundMode` to the `IntentBuilder`:

Java Kotlin

```
// Using a new FlutterEngine.
startActivity(
  FlutterActivity
    .withNewEngine()
    .backgroundMode(FlutterActivity.BackgroundMode.transparent)
    .build(context)
);

// Using a cached FlutterEngine.
startActivity(
  FlutterActivity
    .withCachedEngine("my_engine_id")
    .backgroundMode(FlutterActivity.BackgroundMode.transparent)
    .build(context)
);
```

You now have a `FlutterActivity` with a transparent background.

Note: Make sure that your Flutter content also includes a translucent background. If your Flutter UI paints a solid background color, then it still appears as though your `FlutterActivity` has an opaque background.

Get started

Samples & tutorials

Development

- User interface
- Data & backend
- Accessibility & internationalization
- Platform integration
- Packages & plugins
- Add Flutter to existing app
  - Introduction
  - Adding to an Android app
    - Project setup
    - Add a single Flutter screen
    - Add a Flutter Fragment
  - Adding to an iOS app
  - Running, debugging & hot reload
  - Loading sequence and performance
- Tools & techniques
- Migration notes

Testing & debugging

Performance & optimization

Deployment

Resources

Reference

- Widget index
- API reference
- Package site