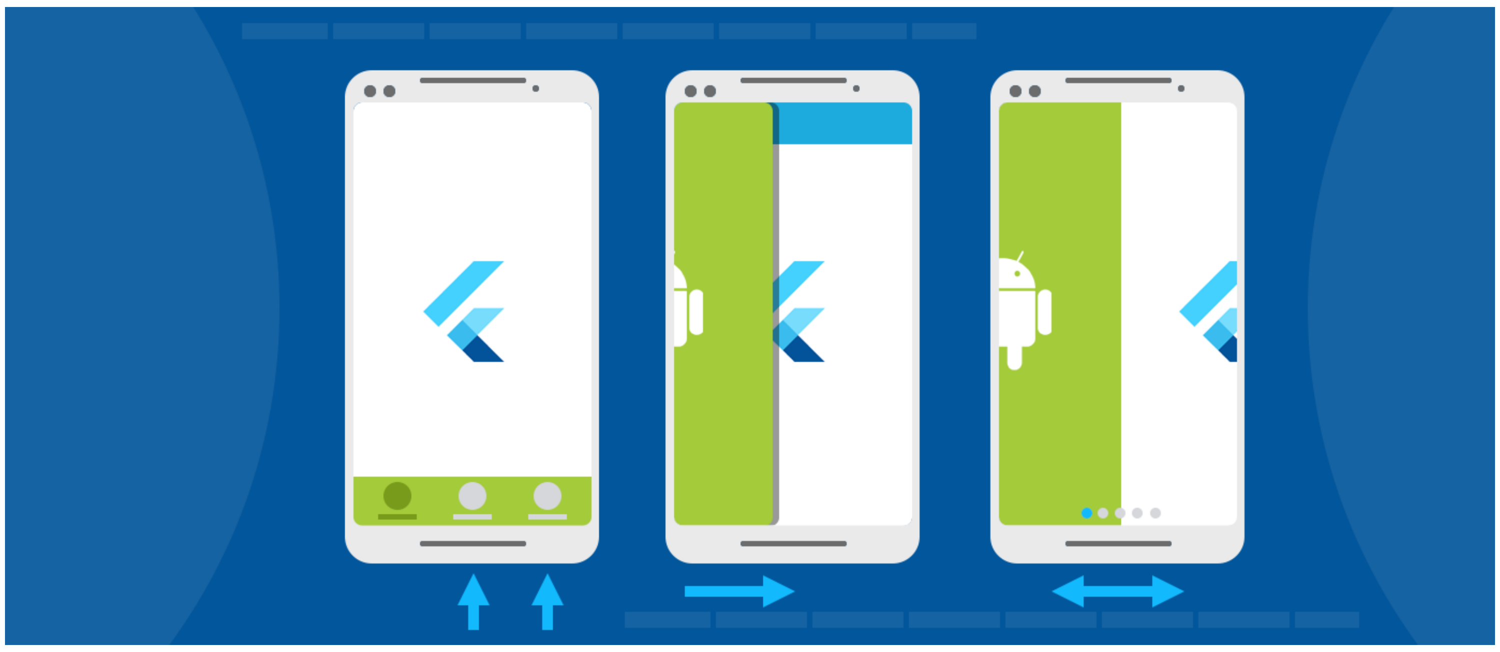
Get started
Samples & tutorials
Development
- User interface
- Data & backend
- Accessibility & internationalization
- Platform integration
- Packages & plugins
- Add Flutter to existing app
  - Introduction
  - Adding to an Android app
    - Project setup
    - Add a single Flutter screen
    - Add 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

Adding a Flutter Fragment to an Android app

Contents



This guide describes how to add a Flutter `Fragment` to an existing Android app. In Android, a `Fragment` represents a modular piec a larger UI. A `Fragment` might be used to present a sliding drawer, tabbed content, a page in a `ViewPager`, or it might simply repres a normal screen in a single-`Activity` app. Flutter provides a `FlutterFragment` so that developers can present a Flutter experienc any place that they can use a regular `Fragment`.

If an `Activity` is equally applicable for your application needs, consider using a `FlutterActivity` instead of a `FlutterFragment` which is quicker and easier to use.

`FlutterFragment` allows developers to control the following details of the Flutter experience within the `Fragment`:

- Initial Flutter route.
- Dart entrypoint to execute.
- Opaque vs translucent background.
- Whether `FlutterFragment` should control its surrounding `Activity`.
- Whether a new `FlutterEngine` or a cached `FlutterEngine` should be used.

`FlutterFragment` also comes with a number of calls that must be forwarded from its surrounding `Activity`. These calls allow Flutter to react appropriately to OS events.

All varieties of `FlutterFragment`, and its requirements, are described in this guide.

Add a `FlutterFragment` to an `Activity` with a new `FlutterEngine`

The first thing to do to use a `FlutterFragment` is to add it to a host `Activity`.

To add a `FlutterFragment` to a host `Activity`, instantiate and attach an instance of `FlutterFragment` in `onCreate()` within the `Activity`, or at another time that works for your app:

Java Kotlin

```java
public class MyActivity extends FragmentActivity {
    // Define a tag String to represent the FlutterFragment within this
    // Activity's FragmentManager. This value can be whatever you'd like.
    private static final String TAG_FLUTTER_FRAGMENT = "flutter_fragment";

    // Declare a local variable to reference the FlutterFragment so that you
    // can forward calls to it later.
    private FlutterFragment flutterFragment;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Inflate a layout that has a container for your FlutterFragment. For
        // this example, assume that a FrameLayout exists with an ID of
        // R.id.fragment_container.
        setContentView(R.layout.my_activity_layout);

        // Get a reference to the Activity's FragmentManager to add a new
        // FlutterFragment, or find an existing one.
        FragmentManager fragmentManager = getSupportFragmentManager();

        // Attempt to find an existing FlutterFragment, in case this is not the
        // first time that onCreate() was run.
        flutterFragment = (FlutterFragment) fragmentManager
            .findFragmentByTag(TAG_FLUTTER_FRAGMENT);

        // Create and attach a FlutterFragment if one does not exist.
        if (flutterFragment == null) {
            flutterFragment = FlutterFragment.createDefault();

            fragmentManager
                .beginTransaction()
                .add(
                    R.id.fragment_container,
                    flutterFragment,
                    TAG_FLUTTER_FRAGMENT
                )
                .commit();
        }
    }
}
```

The previous code is sufficient to render a Flutter UI that begins with a call to your `main()` Dart entrypoint, an initial Flutter route o and a new `FlutterEngine`. However, this code is not sufficient to achieve all expected Flutter behavior. Flutter depends on variou signals that must be forwarded from your host `Activity` to `FlutterFragment`. These calls are shown in the following example:

Java Kotlin

```java
public class MyActivity extends FragmentActivity {
    @Override
    public void onPostResume() {
        super.onPostResume();
        flutterFragment.onPostResume();
    }

    @Override
    protected void onNewIntent(@NonNull Intent intent) {
        flutterFragment.onNewIntent(intent);
    }

    @Override
    public void onBackPressed() {
        flutterFragment.onBackPressed();
    }

    @Override
    public void onRequestPermissionsResult(
        int requestCode,
        @NonNull String[] permissions,
        @NonNull int[] grantResults
    ) {
        flutterFragment.onRequestPermissionsResult(
            requestCode,
            permissions,
            grantResults
        );
    }

    @Override
    public void onUserLeaveHint() {
        flutterFragment.onUserLeaveHint();
    }

    @Override
    public void onTrimMemory(int level) {
        super.onTrimMemory(level);
        flutterFragment.onTrimMemory(level);
    }
}
```

With the OS signals forwarded to Flutter, your `FlutterFragment` works as expected. You have now added a `FlutterFragment` to y existing Android app.

The simplest integration path uses a new `FlutterEngine`, which comes with a non-trivial initialization time, leading to a blank UI u Flutter is initialized and rendered the first time. Most of this time overhead can be avoided by using a cached, pre-warmed `FlutterEngine`, which is discussed next.

Using a pre-warmed `FlutterEngine`

By default, a `FlutterFragment` creates its own instance of a `FlutterEngine`, which requires non-trivial warm-up time. This means your user sees a blank `Fragment` for a brief moment. You can mitigate most of this warm-up time by using an existing, pre-warme instance of `FlutterEngine`.

To use a pre-warmed `FlutterEngine` in a `FlutterFragment`, instantiate a `FlutterFragment` with the `withCachedEngine()` factor method.

Java Kotlin

```java
// Somewhere in your app, before your FlutterFragment is needed, like in the
// Application class ...
// Instantiate a FlutterEngine.
FlutterEngine flutterEngine = new FlutterEngine(context);

// Start executing Dart code in the FlutterEngine.
flutterEngine.getDartExecutor().executeDartEntrypoint(
    DartEntrypoint.createDefault()
);

// Cache the pre-warmed FlutterEngine to be used later by FlutterFragment.
FlutterEngineCache
  .getInstance()
  .put("my_engine_id", flutterEngine);
```

```java
flutterFragment.withCachedEngine("my_engine_id").build();
```

`FlutterFragment` internally knows about `FlutterEngineCache` and retrieves the pre-warmed `FlutterEngine` based on the ID giv to `withCachedEngine()`.

By providing a pre-warmed `FlutterEngine`, as previously shown, your app renders the first Flutter frame as quickly as possible.

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.

Display a splash screen

The initial display of Flutter content requires some wait time, even if a pre-warmed `FlutterEngine` is used. To help improve the u experience around this brief waiting period, Flutter supports the display of a splash screen until Flutter renders its first frame. For instructions about how to show a splash screen, see the Android splash screen guide.

Run Flutter with a specified initial route

An Android app might contain many independent Flutter experiences, running in different `FlutterFragment`s, with different `FlutterEngine`s. In these scenarios, it's common for each Flutter experience to begin with different initial routes (routes other th `/`). To facilitate this, `FlutterFragment`'s `Builder` allows you to specify a desired initial route, as shown:

Java Kotlin

```
// With a new FlutterEngine.
FlutterFragment flutterFragment = FlutterFragment.withNewEngine()
    .initialRoute("myInitialRoute/")
    .build();
```

Note: `FlutterFragment`'s initial route property has no effect when a pre-warmed `FlutterEngine` is used because the pre-warmed `FlutterEngine` already chose an initial route. The initial route can be chosen explicitly when pre-warming a `FlutterEngine`.

Run Flutter from a specified entrypoint

Similar to varying initial routes, different `FlutterFragments` may want to execute different Dart entrypoints. In a typical Flutter ap there is only one Dart entrypoint: `main()`, but you can define other entrypoints.

`FlutterFragment` supports specification of the desired Dart entrypoint to execute for the given Flutter experience. To specify an entrypoint, build `FlutterFragment`, as shown:

Java Kotlin

```
FlutterFragment flutterFragment = FlutterFragment.withNewEngine()
    .dartEntrypoint("mySpecialEntrypoint")
    .build();
```

The `FlutterFragment` configuration results in the execution of a Dart entrypoint called `mySpecialEntrypoint()`. Notice that the parentheses `()` are not included in the `dartEntrypoint` `String` name.

Note: `FlutterFragment`'s Dart entrypoint property has no effect when a pre-warmed `FlutterEngine` is used because the pre-warmed `FlutterEngine` already executed a Dart entrypoint. The Dart entrypoint can be chosen explicitly when pre-warming a `FlutterEngine`.

Control `FlutterFragment`'s render mode

`FlutterFragment` can either use a `SurfaceView` to render its Flutter content, or it can use a `TextureView`. The default is `SurfaceView`, which is significantly better for performance than `TextureView`. However, `SurfaceView` can't be interleaved in the middle of an Android `View` hierarchy. A `SurfaceView` must either be the bottommost `View` in the hierarchy, or the topmost `View` in hierarchy. Additionally, on Android versions before Android N, `SurfaceView`s can't be animated becuase their layout and rendering aren't synchronized with the rest of the `View` hierarchy. If either of these use cases are requirements for your app, then you need t use `TextureView` instead of `SurfaceView`. Select a `TextureView` by building a `FlutterFragment` with a `texture` `RenderMode`:

Java Kotlin

```
// With a new FlutterEngine.
FlutterFragment flutterFragment = FlutterFragment.withNewEngine()
    .renderMode(FlutterView.RenderMode.texture)
    .build();

// With a cached FlutterEngine.
FlutterFragment flutterFragment = FlutterFragment.withCachedEngine("my_engine_id")
    .renderMode(FlutterView.RenderMode.texture)
    .build();
```

Using the configuration shown, the resulting `FlutterFragment` renders its UI to a `TextureView`.

Display a `FlutterFragment` with transparency

By default, `FlutterFragment` renders with an opaque background, using a `SurfaceView`. (See "Control `FlutterFragment`'s render mode.") That background is black for any pixels that aren't painted by Flutter. Rendering with an opaque background is the preferr rendering mode for performance reasons. Flutter rendering with transparency on Android negatively affects performance. Howev there are many designs that require transparent pixels in the Flutter experience that show through to the underlying Android UI. Fo this reason, Flutter supports translucency in a `FlutterFragment`.

Note: Both `SurfaceView` and `TextureView` support transparency. However, when a `SurfaceView` is instructed to render wit transparency, it positions itself at a higher z-index than all other Android `View`s, which means it appears above all other `View`s. This is a limitation of `SurfaceView`. If it's acceptable to render your Flutter experience on top of all other content, then `FlutterFragment`'s default `RenderMode` of `surface` is the `RenderMode` that you should use. However, if you need to display Android `View`s both above and below your Flutter experience, then you must specify a `RenderMode` of `texture`. See "Control `FlutterFragment`'s render mode" for information about controlling the `RenderMode`.

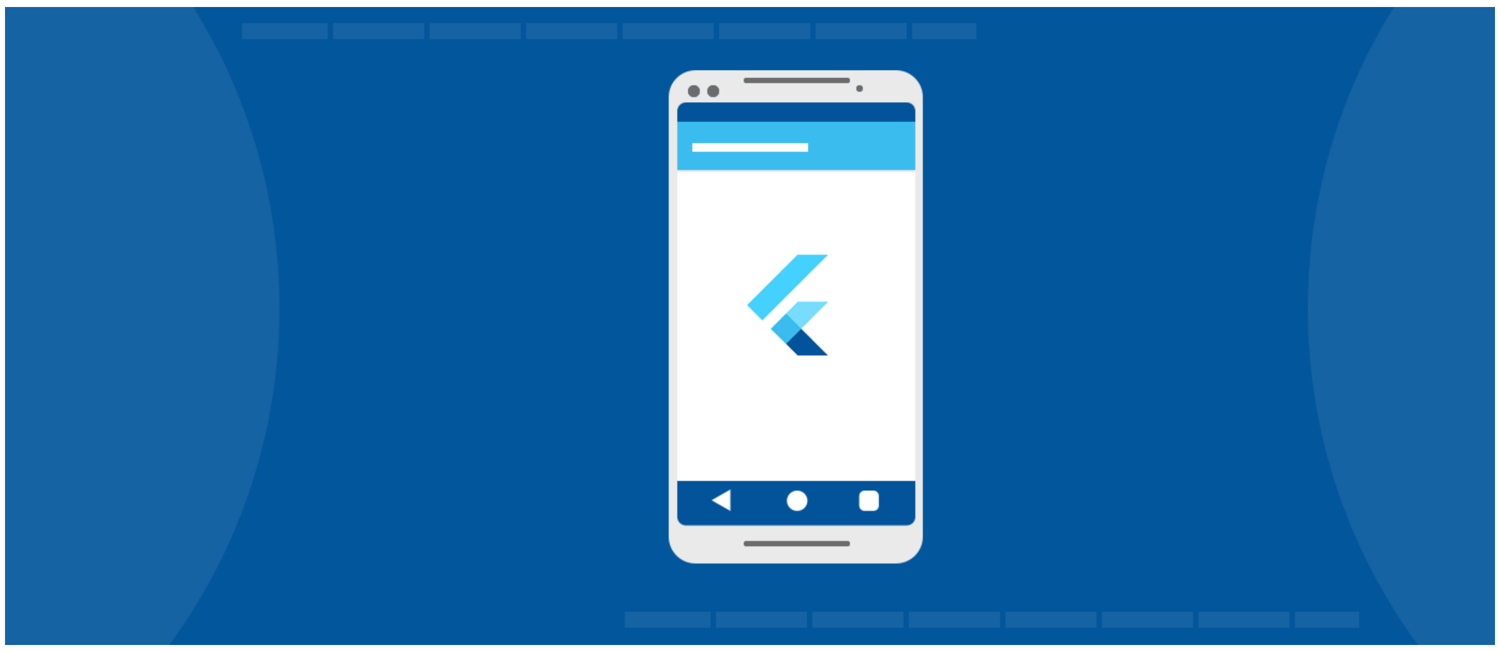
To enable transparency for a `FlutterFragment`, build it with the following configuration:

Java Kotlin

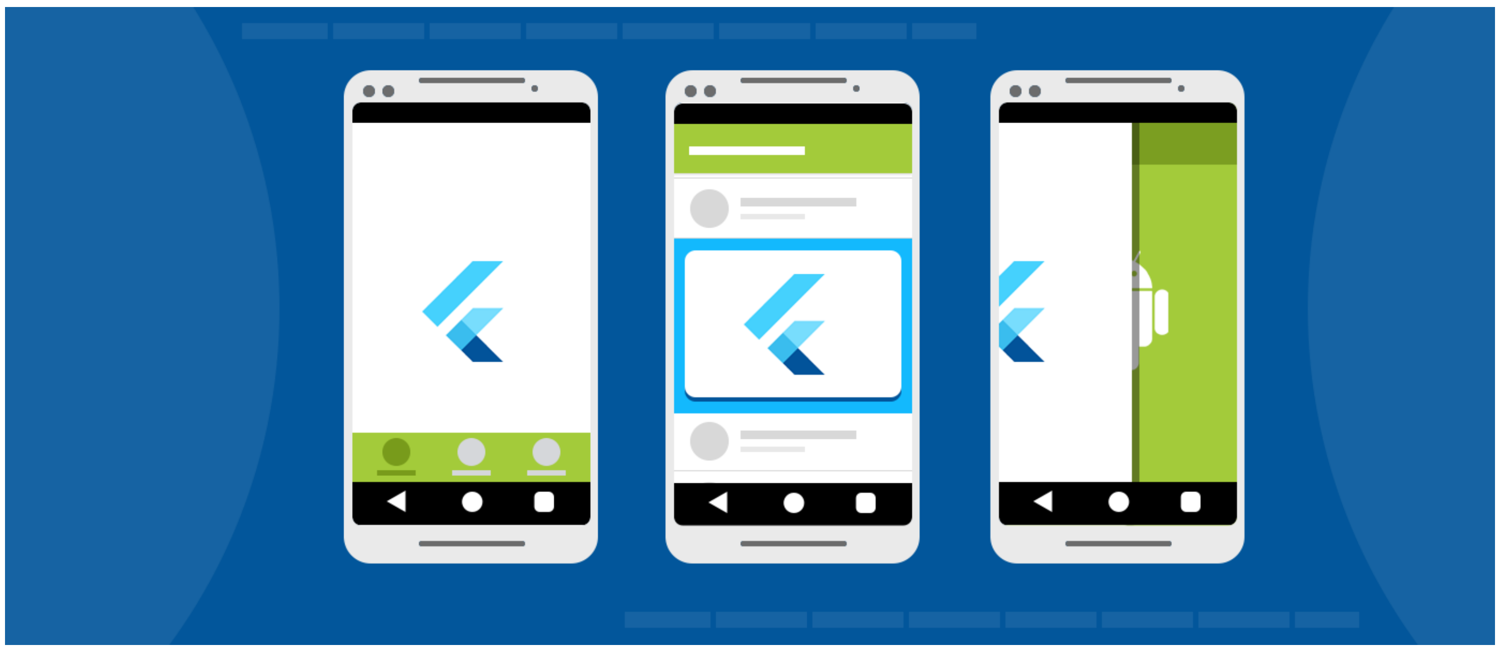
```
// Using a new FlutterEngine.
FlutterFragment flutterFragment = FlutterFragment.withNewEngine()
    .transparencyMode(FlutterView.TransparencyMode.transparent)
    .build();

// Using a cached FlutterEngine.
FlutterFragment flutterFragment = FlutterFragment.withCachedEngine("my_engine_id")
    .transparencyMode(FlutterView.TransparencyMode.transparent)
    .build();
```

The relationship beween `FlutterFragment` and its `Activity`

Some apps choose to use `Fragment`s as entire Android screens. In these apps, it would be reasonable for a `Fragment` to control system chrome like Android's status bar, navigation bar, and orientation.



In other apps, `Fragment`s are used to represent only a portion of a UI. A `FlutterFragment` might be used to implement the inside drawer, a video player, or a single card. In these situations, it would be inappropriate for the `FlutterFragment` to affect Android's system chrome because there are other UI pieces within the same `Window`.



`FlutterFragment` comes with a concept that helps differentiate between the case when a `FlutterFragment` should be able to control its host `Activity`, and the cases when a `FlutterFragment` should only affect its own behavior. To prevent a `FlutterFragment` from exposing its `Activity` to Flutter plugins, and to prevent Flutter from controlling the `Activity`'s system UI, the `shouldAttachEngineToActivity()` method in `FlutterFragment`'s `Builder`, as shown: 9

Java   Kotlin

```
// Using a new FlutterEngine.
FlutterFragment flutterFragment = FlutterFragment.withNewEngine()
    .shouldAttachEngineToActivity(false)
    .build();

// Using a cached FlutterEngine.
FlutterFragment flutterFragment = FlutterFragment.withCachedEngine("my_engine_id")
    .shouldAttachEngineToActivity(false)
    .build();
```

Passing `false` to the `shouldAttachEngineToActivity()` `Builder` method prevents Flutter from interacting with the surrounding `Activity`. The default value is `true`, which allows Flutter and Flutter plugins to interact with the surrounding `Activity`.

Note: Some plugins may expect or require an `Activity` reference. Ensure that none of your plugins require an `Activity` before you disable access.

flutter-dev@ • terms • security • privacy • español • 社区中文资源

Except as otherwise noted, this work is licensed under a Creative Commons Attribution 4.0 International License, and code samples are licensed under the BSD License.