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
  - 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

Load sequence, performance, and memory

Contents

This page describes the breakdown of the steps involved to show a Flutter UI. Knowing this, you can make better, informed decisi about when to pre-warm the Flutter engine, which operations are possible at which stage, and the latency and memory costs of tl operations.

Loading Flutter

Android and iOS apps (the two supported platforms for integrating into existing apps), full Flutter apps, and add-to-app patterns h a similar sequence of conceptual loading steps when displaying the Flutter UI.

Finding the Flutter resources

Flutter's engine runtime and your application's compiled Dart code are both bundled as shared libraries on Android and iOS. The f step of loading Flutter is to find those resources in your .apk/.ipa/.app (along with other Flutter assets such as images, fonts, and code if applicable).

This happens when you construct a FlutterEngine for the first time on both **Android** and **iOS** APIs.

Loading the Flutter library

After it's found, the engine's shared libraries are memory loaded once per process.

On **Android**, this also happens when the `FlutterEngine` is constructed because the JNI connectors need to reference the Flutter library. On **iOS**, this happens when the `FlutterEngine` is first run, such as by running `runWithEntrypoint:`.

Starting the Dart VM

The Dart runtime is responsible for managing Dart memory and concurrency for your Dart code. In JIT mode, it's additionally responsible for compiling the Dart source code into machine code during runtime.

A single Dart runtime exists per application session on Android and iOS.

A one-time Dart VM start is done when constructing the `FlutterEngine` for the first time on **Android** and when running a Dart entrypoint for the first time on **iOS**.

At this point, your Dart code's snapshot is also loaded into memory from your application's files.

This is a generic process that also occurs if you used the Dart SDK directly, without the Flutter engine.

The Dart VM never shuts down after it's started.

Creating and running a Dart Isolate

After the Dart runtime is initialized, the Flutter engine's usage of the Dart runtime is the next step.

This is done by starting a Dart `Isolate` in the Dart runtime. The isolate is Dart's container for memory and threads. A number of auxiliary threads on the host platform are also created at this point to support the isolate, such as a thread for offloading GPU handling and another for image decoding.

One isolate exists per `FlutterEngine` instance, and multiple isolates can be hosted by the same Dart VM.

On **Android**, this happens when you call `DartExecutor.executeDartEntrypoint()` on a `FlutterEngine` instance.

On **iOS**, this happens when you call `runWithEntrypoint:` on a `FlutterEngine`.

At this point, your Dart code's selected entrypoint (the `main()` function of your Dart library's `main.dart` file, by default) is executec you called the Flutter function `runApp()` in your `main()` function, then your Flutter app or your library's widget tree is also created built. If you need to prevent certain functionalities from executing in your Flutter code, then the `AppLifecycleState.detached` en value indicates that the `FlutterEngine` isn't attached to any UI components such as a `FlutterViewController` on iOS or a `FlutterActivity` on Android.

Attaching a UI to the Flutter engine

A standard, full Flutter app moves to reach this state as soon as the app is launched.

In an add-to-app scenario, this happens when you attach a `FlutterEngine` to a UI component such as by calling `startActivity(` with an `Intent` built using `FlutterActivity.withCachedEngine()` on **Android**. Or, by presenting a `FlutterViewController` initialized by using `initWithEngine: nibName: bundle:` on **iOS**.

This is also the case if a Flutter UI component was launched without pre-warming a `FlutterEngine` such as with `FlutterActivity.createDefaultIntent()` on **Android**, or with `FlutterViewController initWithProject: nibName: bundle` **iOS**. An implicit `FlutterEngine` is created in these cases.

Behind the scene, both platform's UI components provide the `FlutterEngine` with a rendering surface such as a `Surface` on **Andı** or a CAEAGLLayer or CAMetalLayer on **iOS**.

At this point, the `Layer` tree generated by your Flutter program, per frame, is converted into OpenGL (or Vulkan or Metal) GPU instructions.

Memory and latency

Showing a Flutter UI has a non-trivial latency cost. This cost can be lessened by starting the Flutter engine ahead of time.

The most relevant choice for add-to-app scenarios is for you to decide when to pre-load a `FlutterEngine` (that is, to load the Flut library, start the Dart VM, and run entrypoint in an isolate), and what the memory and latency cost is of that pre-warm. You also ne to know how the pre-warm affects the memory and latency cost of rendering a first Flutter frame when the UI component is subsequently attached to that `FlutterEngine`.

As of Flutter v1.10.3, and testing on a low-end 2015 class device in release-AOT mode, pre-warming the `FlutterEngine` costs:

- 42 MB and 1530 ms to prewarm on **Android**. 330 ms of it is a blocking call on the main thread.
- 22 MB and 860 ms to prewarm on **iOS**. 260 ms of it is a blocking call on the main thread.

A Flutter UI can be attached during the pre-warm. The remaining time is joined to the time-to-first-frame latency.

Memory-wise, a cost sample (variable, depending on the use case) could be:

- ~4 MB OS's memory usage for creating pthreads.
- ~10 MB GPU driver memory.
- ~1 MB for Dart runtime-managed memory.
- ~5 MB for Dart-loaded font maps.

Latency-wise, a cost sample (variable, depending on the use case) could be:

- ~20 ms to collect the Flutter assets from the application package.
- ~15 ms to dlopen the Flutter engine library.
- ~200 ms to create the Dart VM and load the AOT snapshot.
- ~200 ms to load Flutter-dependent fonts and assets.
- ~400 ms to run the entrypoint, create the first widget tree, and compile the needed GPU shader programs.

The `FlutterEngine` should be pre-warmed late enough to delay the memory consumption needed but early enough to avoid combining the Flutter engine start-up time with the first frame latency of showing Flutter.

The exact timing depends on the app's structure and heuristics. An example would be to load the Flutter engine in the screen bef the screen is drawn by Flutter.

Given an engine pre-warm, the first frame cost on UI attach is:

- 320 ms on **Android** and an additional 12 MB (highly dependent on the screen's physical pixel size).
- 200 ms on **iOS** and an additional 16 MB (highly dependent on the screen's physical pixel size).

Memory-wise, the cost is primarily the graphical memory buffer used for rendering and is dependent on the screen size.

Latency-wise, the cost is primarily waiting for the OS callback to provide Flutter with a rendering surface and compiling the remai shader programs that are not pre-emptively predictable. This is a one-time cost.

When the Flutter UI component is released, the UI-related memory is freed. This doesn't affect the Flutter state, which lives in the `FlutterEngine` (unless the `FlutterEngine` is also released).