FAQ

Introduction

This page collects some common questions asked about Flutter. You might also check out the following specialized FAQs:

- Web FAQ
- Performance FAQ

What is Flutter?

Flutter is Google's portable UI toolkit for crafting beautiful, natively compiled applications for mobile, web, and desktop from a single codebase. Flutter works with existing code, is used by developers and organizations around the world, and is free and open source.

Who is Flutter for?

For users, Flutter makes beautiful apps come to life.

For developers, Flutter lowers the bar to entry for building apps. It speeds app development and reduces the cost and complexity of app production across platforms.

For designers, Flutter provides a canvas for high-end user experiences. Fast Company described Flutter as one of the top design ideas of the decade for its ability to turn concepts into production code without the compromises imposed by typical frameworks. It also acts as a productive prototyping tool with drag-and-drop tools like FlutterFlow and web-based IDEs like Zapp!.

For engineering managers and businesses, Flutter allows the unification of app developers into a single *mobile, web, and desktop app team*, building branded apps for multiple platforms out of a single codebase. Flutter speeds feature development and synchronizes release schedules across the entire customer base.

How much development experience do I need to use Flutter?

Flutter is approachable to programmers familiar with object-oriented concepts (classes, methods, variables, etc) and imperative programming concepts (loops, conditionals, etc).

We have seen people with very little programming experience learn and use Flutter for prototyping and app development.

What kinds of apps can I build with Flutter?

Flutter is designed to support mobile apps that run on both Android and iOS, as well as interactive apps that you want to run on your web pages or on the desktop.

Apps that need to deliver highly branded designs are particularly well suited for Flutter. However, you can also create pixel-perfect experiences that match the Android and iOS design languages with Flutter.

Flutter's package ecosystem supports a wide variety of hardware (such as camera, GPS, network, and storage) and services (such as payments, cloud storage, authentication, and ads).

Who makes Flutter?

Flutter is an open source project, with contributions from Google and other companies and individuals.

Who uses Flutter?

Developers inside and outside of Google use Flutter to build beautiful natively-compiled apps for iOS and Android. To learn about some of these apps, visit the showcase.

What makes Flutter unique?

Flutter is different than most other options for building mobile apps because it doesn't rely on web browser technology nor the set of widgets that ship with each device. Instead, Flutter uses its own high-performance rendering engine to draw widgets.

In addition, Flutter is different because it only has a thin layer of C/C++ code. Flutter implements most of its system (compositing, gestures, animation, framework, widgets, etc) in Dart (a modern, concise, object-oriented language) that developers can easily approach read, change, replace, or remove. This gives developers tremendous control over the system, as well as significantly lowers the bar to approachability for the majority of the system.

Should I build my next production app with Flutter?

Flutter 1 was launched on Dec 4th, 2018, Flutter 2 on March 3rd, 2021, and Flutter 3 on May 10th, 2023. As of May 2023, over *one million* apps have shipped using Flutter to many hundreds of millions of devices. Check out some sample apps in the showcase.

Flutter ships updates on a roughly-quarterly cadence that improve stability and performance and address commonly-requested user features.

What does Flutter provide?

What is inside the Flutter SDK?

Flutter includes:

- Heavily optimized, mobile-first 2D rendering engine with excellent support for text
- Modern react-style framework
- Rich set of widgets implementing Material Design and iOS-style
- APIs for unit and integration tests
- Interop and plugin APIs to connect to the system and 3rd-party SDKs
- Headless test runner for running tests on Windows, Linux, and Mac
- Flutter DevTools (also called Dart DevTools) for testing, debugging, and profiling your app
- Command-line tools for creating, building, testing, and compiling your apps

Does Flutter work with any editors or IDEs?

We provide plugins for VS Code, Android Studio, and IntelliJ IDEA. See editor configuration for setup details, and VS Code and Android Studio/IntelliJ for tips on how to use the plugins.

Alternatively, you can use the `flutter` command from a terminal, along with one of the many editors that support editing Dart.

Does Flutter come with a framework?

Yes! Flutter ships with a modern react-style framework. Flutter's framework is designed to be layered and customizable (and optional). Developers can choose to use only parts of the framework, or even replace upper layers of the framework entirely.

Does Flutter come with widgets?

Yes! Flutter ships with a set of high-quality Material Design and Cupertino (iOS-style) widgets, layouts, and themes. Of course, these widgets are only a starting point. Flutter is designed to make it easy to create your own widgets, or customize the existing widgets.

Does Flutter support Material Design?

Yes! The Flutter and Material teams collaborate closely, and Material is fully supported. For more information, check out the Material 2 and Material 3 widgets in the widget catalog.

Does Flutter come with a testing framework?

Yes, Flutter provides APIs for writing unit and integration tests. Learn more about testing with Flutter.

We use our own testing capabilities to test our SDK, and we measure our test coverage on every commit.

Does Flutter come with debugging tools?

Yes, Flutter comes with Flutter DevTools (also called Dart DevTools). For more information, see Debugging with Flutter and the Flutter DevTools docs.

Does Flutter come with a dependency injection framework?

We don't ship with an opinionated solution, but there are a variety of packages that offer dependency injection and service location, such as injectable, get_it, kiwi, and riverpod.

Technology

What technology is Flutter built with?

Flutter is built with C, C++, Dart, Skia (a 2D rendering engine), and Impeller (the default rendering engine on iOS). See this architecture diagram for a better picture of the main components. For a more detailed description of the layered architecture of Flutter, read the architectural overview.

How does Flutter run my code on Android?

The engine's C and C++ code are compiled with Android's NDK. The Dart code (both the SDK's and yours) are ahead-of-time (AOT) compiled into native, ARM, and x86 libraries. Those libraries are included in a "runner" Android project, and the whole thing is built into an `.apk`. When launched, the app loads the Flutter library. Any rendering, input, or event handling, and so on, is delegated to the compiled Flutter and app code. This is similar to the way many game engines work.

During debug mode, Flutter uses a virtual machine (VM) to run its code in order to enable stateful hot reload, a feature that lets you make changes to your running code without recompilation. You'll see a "debug" banner in the top right-hand corner of your app when running in this mode, to remind you that performance is not characteristic of the finished release app.

How does Flutter run my code on iOS?

The engine's C and C++ code are compiled with LLVM. The Dart code (both the SDK's and yours) are ahead-of-time (AOT) compiled into a native, ARM library. That library is included in a “runner” iOS project, and the whole thing is built into an `.ipa`. When launched, the app loads the Flutter library. Any rendering, input or event handling, and so on, are delegated to the compiled Flutter and app code. This is similar to the way many game engines work.

During debug mode, Flutter uses a virtual machine (VM) to run its code in order to enable stateful hot reload, a feature that lets you make changes to your running code without recompilation. You’ll see a “debug” banner in the top right-hand corner of your app when running in this mode, to remind you that performance is not characteristic of the finished release app.

Does Flutter use my operating system’s built-in platform widgets?

No. Instead, Flutter provides a set of widgets (including Material Design and Cupertino (iOS-styled) widgets), managed and rendered by Flutter’s framework and engine. You can browse a catalog of Flutter’s widgets.

We believe that the end result is higher quality apps. If we reused the built-in platform widgets, the quality and performance of Flutter apps would be limited by the flexibility and quality of those widgets.

In Android, for example, there’s a hard-coded set of gestures and fixed rules for disambiguating them. In Flutter, you can write your own gesture recognizer that is a first class participant in the gesture system. Moreover, two widgets authored by different people can coordinate to disambiguate gestures.

Modern app design trends point towards designers and users wanting more motion-rich UIs and brand-first designs. In order to achieve that level of customized, beautiful design, Flutter is architectured to drive pixels instead of the built-in widgets.

By using the same renderer, framework, and set of widgets, it’s easier to publish for multiple platforms from the same codebase, without having to do careful and costly planning to align different feature sets and API characteristics.

By using a single language, a single framework, and a single set of libraries for all of your code (regardless if your UI is different for each platform or not), we also aim to help lower app development and maintenance costs.

What happens when my mobile OS updates and introduces new widgets?

The Flutter team watches the adoption and demand for new mobile widgets from iOS and Android, and aims to work with the community to build support for new widgets. This work might come in the form of lower-level framework features, new composable widgets, or new widget implementations.

Flutter’s layered architecture is designed to support numerous widget libraries, and we encourage and support the community in building and maintaining widget libraries.

What happens when my mobile OS updates and introduces new platform capabilities?

Flutter's interop and plugin system is designed to allow developers to access new mobile OS features and capabilities immediately. Developers don't have to wait for the Flutter team to expose the new mobile OS capability.

What operating systems can I use to build a Flutter app?

Flutter supports development using Linux, macOS, ChromeOS, and Windows.

What language is Flutter written in?

Dart, a fast-growing modern language optimized for client apps. The underlying graphics framework and the Dart virtual machine are implemented in C/C++.

Why did Flutter choose to use Dart?

During the initial development phase, the Flutter team looked at a lot of languages and runtimes, and ultimately adopted Dart for the framework and widgets. Flutter used four primary dimensions for evaluation, and considered the needs of framework authors, developers, and end users. We found many languages met some requirements, but Dart scored highly on all of our evaluation dimensions and met all our requirements and criteria.

Dart runtimes and compilers support the combination of two critical features for Flutter: a JIT-based fast development cycle that allows for shape changing and stateful hot reloads in a language with types, plus an Ahead-of-Time compiler that emits efficient ARM code for fast startup and predictable performance of production deployments.

In addition, we have the opportunity to work closely with the Dart community, which is actively investing resources in improving Dart for use in Flutter. For example, when we adopted Dart, the language didn't have an ahead-of-time toolchain for producing native binaries, which is instrumental in achieving predictable, high performance, but now the language does because the Dart team built it for Flutter. Similarly, the Dart VM has previously been optimized for throughput but the team is now optimizing the VM for latency, which is more important for Flutter's workload.

Dart scores highly for us on the following primary criteria:

Developer productivity

One of Flutter's main value propositions is that it saves engineering resources by letting developers create apps for both iOS and Android with the same codebase. Using a highly productive language accelerates developers further and makes Flutter more attractive. This was very important to both our framework team as well as our developers. The majority of Flutter is built in the same language we give to our users, so we need to stay productive at

100k's lines of code, without sacrificing approachability or readability of the framework and widgets for our developers.

Object-orientation

For Flutter, we want a language that's suited to Flutter's problem domain: creating visual user experiences. The industry has multiple decades of experience building user interface frameworks in object-oriented languages. While we could use a non-object-oriented language, this would mean reinventing the wheel to solve several hard problems. Plus, the vast majority of developers have experience with object-oriented development, making it easier to learn how to develop with Flutter.

Predictable, high performance

With Flutter, we want to empower developers to create fast, fluid user experiences. In order to achieve that, we need to be able to run a significant amount of end-developer code during every animation frame. That means we need a language that both delivers high performance and predictable performance, without periodic pauses that would cause dropped frames.

Fast allocation

The Flutter framework uses a functional-style flow that depends heavily on the underlying memory allocator efficiently handling small, short-lived allocations. This style was developed in languages with this property and doesn't work efficiently in languages that lack this facility.

Can Flutter run any Dart code?

Flutter can run Dart code that doesn't directly or transitively import `dart:mirrors` or `dart:html`.

How big is the Flutter engine?

In March 2021, we measured the download size of a minimal Flutter app (no Material Components, just a single `Center` widget, built with `flutter build apk --split-per-abi`), bundled and compressed as a release APK, to be approximately 4.3 MB for ARM32, and 4.8 MB for ARM64.

On ARM32, the core engine is approximately 3.4 MB (compressed), the framework + app code is approximately 765 KB (compressed), the LICENSE file is 58 KB (compressed), and necessary Java code (`classes.dex`) is 120 KB (compressed).

In ARM64, the core engine is approximately 4.0 MB (compressed), the framework + app code is approximately 659 KB (compressed), the LICENSE file is 58 KB (compressed), and necessary Java code (`classes.dex`) is 120 KB (compressed).

These numbers were measured using apkanalyzer, which is also built into Android Studio.

On iOS, a release IPA of the same app has a download size of 10.9 MB on an iPhone X, as reported by Apple's App Store Connect. The IPA is larger than the APK mainly because Apple encrypts binaries within the IPA, making the compression less efficient (see the iOS App Store Specific Considerations section of Apple's QA1795).

info **Note:** The release engine binary used to include LLVM IR (bitcode). However, Apple deprecated bitcode in Xcode 14 and removed support, so it has been removed from the Flutter 3.7 release.
Of course, we recommend that you measure your own app. To do that, see Measuring your app's size.

How does Flutter define a pixel?

Flutter uses logical pixels, and often refers to them merely as "pixels". Flutter's `devicePixelRatio` expresses the ratio between physical pixels and logical CSS pixels.

Capabilities

What kind of app performance can I expect?

You can expect excellent performance. Flutter is designed to help developers easily achieve a constant 60fps. Flutter apps run using natively compiled code—no interpreters are involved. This means that Flutter apps start quickly.

What kind of developer cycles can I expect? How long between edit and refresh?

Flutter implements a *hot reload* developer cycle. You can expect sub-second reload times, on a device or an emulator/simulator.

Flutter's hot reload is *stateful* so the app state is retained after a reload. This means you can quickly iterate on a screen deeply nested in your app, without starting from the home screen after every reload.

How is *hot reload* different from *hot restart*?

Hot reload works by injecting updated source code files into the running Dart VM (Virtual Machine). This doesn't only add new classes, but also adds methods and fields to existing classes, and changes existing functions. Hot restart resets the state to the app's initial state.

For more information, see Hot reload.

Where can I deploy my Flutter app?

You can compile and deploy your Flutter app to iOS, Android, web, and desktop.

What devices and OS versions does Flutter run on?

- We support and test running Flutter on a variety of low-end to high-end platforms. For a detailed list of the platforms on which we test, see the list of supported platforms.
- Flutter supports building ahead-of-time (AOT) compiled libraries for `x86_64`, `armeabi-v7a`, and `arm64-v8a`.
- Apps built for ARMv7 or ARM64 run fine (using ARM emulation) on many x86 Android devices.
- We support developing Flutter apps on a range of platforms. See the system requirements listed under each development operating system.

Does Flutter run on the web?

Yes, web support is available in the stable channel. For more details, check out the web instructions.

Can I use Flutter to build desktop apps?

Yes, desktop support is in stable for Windows, macOS, and Linux.

Can I use Flutter inside of my existing native app?

Yes, learn more in the add-to-app section of our website.

Can I access platform services and APIs like sensors and local storage?

Yes. Flutter gives developers out-of-the-box access to *some* platform-specific services and APIs from the operating system. However, we want to avoid the “lowest common denominator” problem with most cross-platform APIs, so we don’t intend to build cross-platform APIs for all native services and APIs.

A number of platform services and APIs have ready-made packages available on pub.dev. Using an existing package is easy.

Finally, we encourage developers to use Flutter’s asynchronous message passing system to create your own integrations with platform and third-party APIs. Developers can expose as much or as little of the platform APIs as they need, and build layers of abstractions that are a best fit for their project.

Can I extend and customize the bundled widgets?

Absolutely. Flutter’s widget system was designed to be easily customizable.

Rather than having each widget provide a large number of parameters, Flutter embraces composition. Widgets are built out of smaller widgets that you can reuse and combine in novel ways to make custom widgets. For example, rather than subclassing a generic button widget, `ElevatedButton` combines a Material widget with a `GestureDetector` widget. The Material widget provides the visual design and the `GestureDetector` widget provides the interaction design.

To create a button with a custom visual design, you can combine widgets that implement your visual design with a `GestureDetector`, which provides the interaction design. For example, `CupertinoButton` follows this approach and combines a `GestureDetector` with several other widgets that implement its visual design.

Composition gives you maximum control over the visual and interaction design of your widgets while also allowing a large amount of code reuse. In the framework, we've decomposed complex widgets to pieces that separately implement the visual, interaction, and motion design. You can remix these widgets however you like to make your own custom widgets that have full range of expression.

Why would I want to share layout code across iOS and Android?

You can choose to implement different app layouts for iOS and Android. Developers are free to check the mobile OS at runtime and render different layouts, though we find this practice to be rare.

More and more, we see mobile app layouts and designs evolving to be more brand-driven and unified across platforms. This implies a strong motivation to share layout and UI code across iOS and Android.

The brand identity and customization of the app's aesthetic design is now becoming more important than strictly adhering to traditional platform aesthetics. For example, app designs often require custom fonts, colors, shapes, motion, and more in order to clearly convey their brand identity.

We also see common layout patterns deployed across iOS and Android. For example, the "bottom nav bar" pattern can now be naturally found across iOS and Android. There seems to be a convergence of design ideas across mobile platforms.

Can I interop with my mobile platform's default programming language?

Yes, Flutter supports calling into the platform, including integrating with Java or Kotlin code on Android, and ObjectiveC or Swift code on iOS. This is enabled by a flexible message passing style where a Flutter app might send and receive messages to the mobile platform using a `BasicMessageChannel`.

Learn more about accessing platform and third-party services in Flutter with platform channels.

Here is an example project that shows how to use a platform channel to access battery state information on iOS and Android.

Does Flutter come with a reflection / mirrors system?

No. Dart includes `dart:mirrors`, which provides type reflection. But since Flutter apps are pre-compiled for production, and binary size is always a concern with mobile apps, this library is unavailable for Flutter apps.

Using static analysis we can strip out anything that isn't used ("tree shaking"). If you import a huge Dart library but only use a self-contained two-line method, then you only pay the cost of the two-line method, even if that Dart library itself imports dozens and dozens of other libraries. This guarantee is only secure if Dart can identify the code path at compile time. To date, we've found other approaches for specific needs that offer a better trade-off, such as code generation.

How do I do internationalization (i18n), localization (l10n), and accessibility (a11y) in Flutter?

Learn more about i18n and l10n in the internationalization tutorial.

Learn more about a11y in the accessibility documentation.

How do I write parallel and/or concurrent apps for Flutter?

Flutter supports isolates. Isolates are separate heaps in Flutter's VM, and they are able to run in parallel (usually implemented as separate threads). Isolates communicate by sending and receiving asynchronous messages.

Check out an example of using isolates with Flutter.

Can I run Dart code in the background of a Flutter app?

Yes, you can run Dart code in a background process on both iOS and Android. For more information, see the free Medium article Executing Dart in the Background with Flutter Plugins and Geofencing.

Can I use JSON/XML/protobuffers (and so on) with Flutter?

Absolutely. There are libraries on pub.dev for JSON, XML, protobufs, and many other utilities and formats.

For a detailed writeup on using JSON with Flutter, check out the JSON tutorial.

Can I build 3D (OpenGL) apps with Flutter?

Today we don't support 3D using OpenGL ES or similar. We have long-term plans to expose an optimized 3D API, but right now we're focused on 2D.

Why is my APK or IPA so big?

Usually, assets including images, sound files, fonts, and so on, are the bulk of an APK or IPA. Various tools in the Android and iOS ecosystems can help you understand what's inside of your APK or IPA.

Also, be sure to create a *release build* of your APK or IPA with the Flutter tools. A release build is usually *much* smaller than a *debug build*.

Learn more about creating a release build of your Android app, and creating a release build of your iOS app. Also, check out Measuring your app's size.

Do Flutter apps run on Chromebooks?

We have seen Flutter apps run on some Chromebooks. We are tracking issues related to running Flutter on Chromebooks.

Is Flutter ABI compatible?

Flutter and Dart don't offer application binary interface (ABI) compatibility. Offering ABI compatability is not a current goal for Flutter or Dart.

Framework

Why is the build() method on State, rather than StatefulWidget?

Putting a `Widget build(BuildContext context)` method on `State` rather putting a `Widget build(BuildContext context, State state)` method on `StatefulWidget` gives developers more flexibility when subclassing `StatefulWidget`. You can read a more detailed discussion on the API docs for `State.build`.

Where is Flutter's markup language? Why doesn't Flutter have a markup syntax?

Flutter UIs are built with an imperative, object-oriented language (Dart, the same language used to build Flutter's framework). Flutter doesn't ship with a declarative markup.

We found that UIs dynamically built with code allow for more flexibility. For example, we have found it difficult for a rigid markup system to express and produce customized widgets with bespoke behaviors.

We have also found that our "code-first" approach better allows for features like hot reload and dynamic environment adaptations.

It’s possible to create a custom language that is then converted to widgets on the fly. Because build methods are “just code”, they can do anything, including interpreting markup and turning it into widgets.

My app has a Debug banner/ribbon in the upper right. Why am I seeing that?

By default, the `flutter run` command uses the debug build configuration.

The debug configuration runs your Dart code in a VM (Virtual Machine) enabling a fast development cycle with hot reload (release builds are compiled using the standard Android and iOS toolchains).

The debug configuration also checks all asserts, which helps you catch errors early during development, but imposes a runtime cost. The “Debug” banner indicates that these checks are enabled. You can run your app without these checks by using either the `--profile` or `--release` flag to `flutter run`.

If your IDE uses the Flutter plugin, you can launch the app in profile or release mode. For VS Code, use the **Run > Start debugging** or **Run > Run without debugging** menu entries. For IntelliJ, use the menu entries **Run > Flutter Run in Profile Mode** or **Release Mode**.

What programming paradigm does Flutter’s framework use?

Flutter is a multi-paradigm programming environment. Many programming techniques developed over the past few decades are used in Flutter. We use each one where we believe the strengths of the technique make it particularly well-suited. In no particular order:

Composition

The primary paradigm used by Flutter is that of using small objects with narrow scopes of behavior, composed together to obtain more complicated effects, sometimes called *aggressive composition*. Most widgets in the Flutter widget library are built in this way. For example, the Material `TextButton` class is built using an `IconTheme`, an `InkWell`, a `Padding`, a `Center`, a `Material`, an `AnimatedDefaultTextStyle`, and a `ConstrainedBox`. The `InkWell` is built using a `GestureDetector`. The `Material` is built using an `AnimatedDefaultTextStyle`, a `NotificationListener`, and an `AnimatedPhysicalModel`. And so on. It’s widgets all the way down.

Functional programming

Entire applications can be built with only `StatelessWidget`s, which are essentially functions that describe how arguments map to other functions, bottoming out in primitives that compute layouts or paint graphics. (Such applications can’t easily have state, so are typically non-interactive.) For example, the `Icon` widget is essentially a function that maps its arguments (`color`, `icon`, `size`) into layout primitives. Additionally, heavy use is made of immutable data structures, including the entire `Widget` class hierarchy as well as numerous supporting classes such as `Rect` and `TextStyle`. On a smaller scale, Dart's `Iterable` API, which makes heavy use of the functional style (map, reduce, where, etc), is frequently used to process lists of values in the framework.

Event-driven programming

User interactions are represented by event objects that are dispatched to callbacks registered with event handlers. Screen updates are triggered by a similar callback mechanism. The `Listenable` class, which is used as the basis of the animation system, formalizes a subscription model for events with multiple listeners.

Class-based object-oriented programming

Most of the APIs of the framework are built using classes with inheritance. We use an approach whereby we define very high-level APIs in our base classes, then specialize them iteratively in subclasses. For example, our render objects have a base class (`RenderObject`) that is agnostic regarding the coordinate system, and then we have a subclass (`RenderBox`) that introduces the opinion that the geometry should be based on the Cartesian coordinate system (x/width and y/height).

Prototype-based object-oriented programming

The `ScrollPhysics` class chains instances to compose the physics that apply to scrolling dynamically at runtime. This lets the system compose, for example, paging physics with platform-specific physics, without the platform having to be selected at compile time.

Imperative programming

Straightforward imperative programming, usually paired with state encapsulated within an object, is used where it provides the most intuitive solution. For example, tests are written in an imperative style, first describing the situation under test, then listing the invariants that the test must match, then advancing the clock or inserting events as necessary for the test.

Reactive programming

The widget and element trees are sometimes described as reactive, because new inputs provided in a widget's constructor are immediately propagated as changes to lower-level widgets by the widget's build method, and changes made in the lower widgets (for example, in response to user input) propagate back up the tree using event handlers. Aspects of both functional-reactive and imperative-reactive are present in the framework, depending on the needs of the widgets. Widgets with build methods that consist of just an expression describing how the widget reacts to changes in its configuration are functional reactive widgets (for example, the Material `Divider` class). Widgets whose build methods construct a list of children over several statements, describing how the widget reacts to changes in its configuration, are imperative reactive widgets (for example, the `Chip` class).

Declarative programming

The build methods of widgets are often a single expression with multiple levels of nested constructors, written using a strictly declarative subset of Dart. Such nested expressions could be mechanically transformed to or from any suitably expressive markup language. For example, the `UserAccountsDrawerHeader` widget has a long build method (20+ lines), consisting of a single nested expression. This can also be combined with the imperative style to build UIs that would be harder to describe in a pure-declarative approach.

Generic programming

Types can be used to help developers catch programming errors early. The Flutter framework uses generic programming to help in this regard. For example, the `State` class is parameterized in terms of the type of its associated widget, so that the Dart analyzer can catch mismatches of states and widgets. Similarly, the `GlobalKey` class takes a type parameter so that it can access a remote widget's state in a type-safe manner (using runtime checking), the `Route` interface is parameterized with the type that it is expected to use when popped, and collections such as `List`s, `Map`s, and `Set`s are all parameterized so that mismatched elements can be caught early either during analysis or at runtime during debugging.

Concurrent programming

Flutter makes heavy use of `Future`s and other asynchronous APIs. For example, the animation system reports when an animation is finished by completing a future. The image loading system similarly uses futures to report when a load is complete.

Constraint programming

The layout system in Flutter uses a weak form of constraint programming to determine the geometry of a scene. Constraints (for example, for cartesian boxes, a minimum and maximum width and a minimum and maximum height) are passed from parent to child, and the child selects a resulting geometry (for example, for cartesian boxes, a size, specifically a width and a height) that fulfills those constraints. By using this technique, Flutter can usually lay out an entire scene with a single pass.

Project

Where can I get support?

If you think you've encountered a bug, file it in our issue tracker. You might also use Stack Overflow for "HOWTO" type questions. For discussions, join our mailing list at flutter-dev@googlegroups.com or seek us out on Discord.

For more information, see our Community page.

How do I get involved?

Flutter is open source, and we encourage you to contribute. You can start by simply filing issues for feature requests and bugs in our issue tracker.

We recommend that you join our mailing list at flutter-dev@googlegroups.com and let us know how you're using Flutter and what you'd like to do with it.

If you're interested in contributing code, you can start by reading our Contributing guide, and check out our list of easy starter issues.

Finally, you can connect with helpful Flutter communities. For more information, see the Community page.

Is Flutter open source?

Yes, Flutter is open source technology. You can find the project on GitHub.

Which software license(s) apply to Flutter and its dependencies?

Flutter includes two components: an engine that ships as a dynamically linked binary, and the Dart framework as a separate binary that the engine loads. The engine uses multiple software components with many dependencies; view the complete list in its license file.

The framework is entirely self-contained and requires only one license.

In addition, any Dart packages you use might have their own license requirements.

How can I determine the licenses my Flutter application needs to show?

There's an API to find the list of licenses you need to show:

- If your application has a `Drawer`, add an `AboutListTile`.
- If your application doesn't have a Drawer but does use the Material Components library, call either `showAboutDialog` or `showLicensePage`.
- For a more custom approach, you can get the raw licenses from the `LicenseRegistry`.

Who works on Flutter?

We all do! Flutter is an open source project. Currently, the bulk of the development is done by engineers at Google. If you're excited about Flutter, we encourage you to join the community and contribute to Flutter!

What are Flutter's guiding principles?

We believe the following:

- In order to reach every potential user, developers need to target multiple mobile platforms.
- HTML and WebViews as they exist today make it challenging to consistently hit high frame rates and deliver high-fidelity experiences, due to automatic behavior (scrolling, layout) and legacy support.
- Today, it's too costly to build the same app multiple times: it requires different teams, different code bases, different workflows, different tools, etc.

- Developers want an easier, better way to use a single codebase to build mobile apps for multiple target platforms, and they don't want to sacrifice quality, control, or performance.

We are focused on three things:

Control
Developers deserve access to, and control over, all layers of the system. Which leads to:
Performance
Users deserve perfectly fluid, responsive, jank-free apps. Which leads to:
Fidelity:
Everyone deserves precise, beautiful, delightful app experiences.

Will Apple reject my Flutter app?

We can't speak for Apple, but their App Store contains many apps built with framework technologies such as Flutter. Indeed, Flutter uses the same fundamental architectural model as Unity, the engine that powers many of the most popular games on the Apple store.

Apple has frequently featured well-designed apps that are built with Flutter, including Hamilton and Reflectly.

As with any app submitted to the Apple store, apps built with Flutter should follow Apple's guidelines for App Store submission.

SOURCE: https://docs.flutter.dev/resources/faq