

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

Project setup

Add a single Flutter screen

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 screen to an iOS app

[Docs](#) > [Development](#) > [Add Flutter to existing app](#) > [Adding Flutter to iOS](#) > [Add a Flutter screen](#)

Contents

- [Start a FlutterEngine and FlutterViewController](#)
 - [Create a FlutterEngine](#)
 - [Show a FlutterViewController with your FlutterEngine](#)
 - [Alternatively - Create a FlutterViewController with an implicit FlutterEngine](#)
- [Using the FlutterAppDelegate](#)
- [Launch options](#)
 - [Dart entrypoint](#)
 - [Dart library](#)
 - [Route](#)
 - [Other](#)

This guide describes how to add a single Flutter screen to an existing iOS app.

Start a FlutterEngine and FlutterViewController

To launch a Flutter screen from an existing iOS, you start a [FlutterEngine](#) and a [FlutterViewController](#).

The [FlutterEngine](#) serves as a host to the Dart VM and your Flutter runtime, and the [FlutterViewController](#) attaches to a [FlutterEngine](#) to pass UIKit input events into Flutter and to display frames rendered by the [FlutterEngine](#).

The [FlutterEngine](#) may have the same lifespan as your [FlutterViewController](#) or outlive your [FlutterViewController](#).

Tip: It's generally recommended to pre-warm a long-lived [FlutterEngine](#) for your application because:

- The first frame appears faster when showing the [FlutterViewController](#).
- Your Flutter and Dart state will outlive one [FlutterViewController](#).
- Your application and your plugins can interact with Flutter and your Dart logic before showing the UI.

See [Loading sequence and performance](#) for more analysis on the latency and memory trade-offs of pre-warming an engine.

Create a FlutterEngine

The proper place to create a [FlutterEngine](#) is specific to your host app. As an example, we demonstrate creating a [FlutterEngi](#) exposed as a property, on app startup in the app delegate.

[Objective-C](#) [Swift](#)

In [AppDelegate.h](#):

```
@import UIKit;
#import Flutter;

@interface AppDelegate : FlutterAppDelegate // More on the FlutterAppDelegate below.
@property (nonatomic, strong) FlutterEngine *flutterEngine;
@end
```

In [AppDelegate.m](#):

[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](#)

[Project setup](#)

[Add a single Flutter screen](#)

[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](#)

```
#import <FlutterPluginRegistrant/GeneratedPluginRegistrant.h> // Used to connect plugins.

#import "AppDelegate.h"

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary<UIApplicationLaunchOptionsKey, id> *)launchOptions {
    self.flutterEngine = [[FlutterEngine alloc] initWithName:@"my flutter engine"];
    // Runs the default Dart entrypoint with a default Flutter route.
    [self.flutterEngine run];
    [GeneratedPluginRegistrant registerWithRegistry:self.flutterEngine];
    return [super application:application didFinishLaunchingWithOptions:launchOptions];
}

@end
```

Show a FlutterViewController with your FlutterEngine

The following example shows a generic `ViewController` with a `UIButton` hooked to present a `FlutterViewController`. The `FlutterViewController` uses the `FlutterEngine` instance created in the `AppDelegate`.

[Objective-C](#) [Swift](#)

```
@import Flutter;
#import "AppDelegate.h"
#import "ViewController.h"

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    // Make a button to call the showFlutter function when pressed.
    UIButton *button = [UIButton buttonWithTypeCustom];
    [button addTarget:self
                 action:@selector(showFlutter)
                 forControlEvents:UIControlEventTouchUpInside];
    [button setTitle:@"Show Flutter!" forState:UIControlStateNormal];
    button.backgroundColor = UIColor.blueColor;
    button.frame = CGRectMake(80.0, 210.0, 160.0, 40.0);
    [self.view addSubview:button];
}

- (void)showFlutter {
    FlutterEngine *flutterEngine =
        ((AppDelegate *)UIApplication.sharedApplication.delegate).flutterEngine;
    FlutterViewController *flutterViewController =
        [[FlutterViewController alloc] initWithEngine:flutterEngine nibName:nil bundle:nil];
    [self presentViewController:flutterViewController animated:YES completion:nil];
}

@end
```

Now, you have a Flutter screen embedded in your iOS app.

Note: Using the previous example, the default `main()` entrypoint function of your default Dart library would run when calling `run` on the `FlutterEngine` created in the `AppDelegate`.

Alternatively - Create a FlutterViewController with an implicit FlutterEngine

As an alternative to the previous example, you can let the `FlutterViewController` implicitly create its own `FlutterEngine` witho pre-warming one ahead of time.

This is not usually recommended because creating a `FlutterEngine` on-demand could introduce a noticeable latency between w the `FlutterViewController` is presented and when it renders its first frame. This could, however, be useful if the Flutter screen is rarely shown, when there are no good heuristics to determine when the Dart VM should be started, and when Flutter doesn't need persist state between view controllers.

To let the `FlutterViewController` present without an existing `FlutterEngine`, omit the `FlutterEngine` construction, and create `FlutterViewController` without an engine reference.

[Objective-C](#) [Swift](#)

[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](#)

[Project setup](#)

[Add a single Flutter screen](#)

[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](#)

```
// Existing code omitted.
- (void)showFlutter {
    FlutterViewController *flutterViewController =
        [[FlutterViewController alloc] initWithProject:nil nibName:nil bundle:nil];
    [self presentViewController:flutterViewController animated:YES completion:nil];
}
@end
```

See [Loading sequence and performance](#) for more explorations on latency and memory usage.

Using the FlutterAppDelegate

Letting your application’s UIApplicationDelegate subclass FlutterAppDelegate is recommended but not required.

The FlutterAppDelegate performs functions such as:

- Forwarding application callbacks such as [openURL](#) to plugins such as [local_auth](#).
- Forwarding status bar taps (which can only be detected in the AppDelegate) to Flutter for scroll-to-top behavior.

If your app delegate can’t directly make FlutterAppDelegate a subclass, make your app delegate implement the FlutterAppLifeCycleProvider protocol in order to make sure your plugins receive the necessary callbacks. Otherwise, plugins t depend on these events may have undefined behavior.

For instance:

```
@import Flutter;
@import UIKit;
@import FlutterPluginRegistrant;

@interface AppDelegate : UIResponder <UIApplicationDelegate, FlutterAppLifeCycleProvider>
@property (strong, nonatomic) UIWindow *window;
@property (nonatomic, strong) FlutterEngine *flutterEngine;
@end
```

The implementation should delegate mostly to a FlutterPluginAppLifeCycleDelegate:

[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](#)

[Project setup](#)

[Add a single Flutter screen](#)

[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](#)

```
@interface AppDelegate ()
@property (nonatomic, strong) FlutterPluginAppLifeCycleDelegate* lifeCycleDelegate;
@end

@implementation AppDelegate

- (instancetype)init {
    if (self = [super init]) {
        _lifeCycleDelegate = [[FlutterPluginAppLifeCycleDelegate alloc] init];
    }
    return self;
}

- (BOOL)application:(UIApplication*)application
didFinishLaunchingWithOptions:(NSDictionary<UIApplicationLaunchOptionsKey, id>*)launchOptions {
    self.flutterEngine = [[FlutterEngine alloc] initWithName:@"io.flutter" project:nil];
    [self.flutterEngine runWithEntrypoint:nil];
    [GeneratedPluginRegistrant registerWithRegistry:self.flutterEngine];
    return [_lifeCycleDelegate application:application didFinishLaunchingWithOptions:launchOptions];
}

// Returns the key window's rootViewController, if it's a FlutterViewController.
// Otherwise, returns nil.
- (FlutterViewController*)rootFlutterViewController {
    UIViewController* viewController = [UIApplication sharedApplication].keyWindow.rootViewController;
    if ([viewController isKindOfClass:[FlutterViewController class]]) {
        return (FlutterViewController*)viewController;
    }
    return nil;
}

- (void)touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event {
    [super touchesBegan:touches withEvent:event];

    // Pass status bar taps to key window Flutter rootViewController.
    if (self.rootFlutterViewController != nil) {
        [self.rootFlutterViewController handleStatusBarTouches:event];
    }
}

- (void)application:(UIApplication*)application
didRegisterUserNotificationSettings:(UIUserNotificationSettings*)notificationSettings {
    [_lifeCycleDelegate application:application
didRegisterUserNotificationSettings:notificationSettings];
}

- (void)application:(UIApplication*)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceToken {
    [_lifeCycleDelegate application:application
didRegisterForRemoteNotificationsWithDeviceToken:deviceToken];
}

- (void)application:(UIApplication*)application
didReceiveRemoteNotification:(NSDictionary*)userInfo
fetchCompletionHandler:(void (^)(UIBackgroundFetchResult result))completionHandler {
    [_lifeCycleDelegate application:application
didReceiveRemoteNotification:userInfo
fetchCompletionHandler:completionHandler];
}

- (BOOL)application:(UIApplication*)application
openURL:(NSURL*)url
options:(NSDictionary<UIApplicationOpenURLOptionsKey, id>*)options {
    return [_lifeCycleDelegate application:application openURL:url options:options];
}

- (BOOL)application:(UIApplication*)application handleOpenURL:(NSURL*)url {
    return [_lifeCycleDelegate application:application handleOpenURL:url];
}

- (BOOL)application:(UIApplication*)application
openURL:(NSURL*)url
sourceApplication:(NSString*)sourceApplication
annotation:(id)annotation {
    return [_lifeCycleDelegate application:application
openURL:url
sourceApplication:sourceApplication
annotation:annotation];
}

- (void)application:(UIApplication*)application
performActionForShortcutItem:(UIApplicationShortcutItem*)shortcutItem
completionHandler:(void (^)(BOOL succeeded))completionHandler NS_AVAILABLE_IOS(9_0) {
    [_lifeCycleDelegate application:application
performActionForShortcutItem:shortcutItem
completionHandler:completionHandler];
}
```

[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](#)

[Project setup](#)

[Add a single Flutter screen](#)

[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](#)

```
- (void)application:(UIApplication*)application
handleEventsForBackgroundURLSession:(nonnull NSString*)identifier
completionHandler:(nonnull void (^)(void))completionHandler {
    [_lifeCycleDelegate application:application
handleEventsForBackgroundURLSession:identifier
completionHandler:completionHandler];
}

- (void)application:(UIApplication*)application
performFetchWithCompletionHandler:(void (^)(UIBackgroundFetchResult result))completionHandler {
    [_lifeCycleDelegate application:application performFetchWithCompletionHandler:completionHandler];
}

- (void)addApplicationLifecycleDelegate:(NSObject<FlutterPlugin>*)delegate {
    [_lifeCycleDelegate addDelegate:delegate];
}

@end
```

Launch options

The examples demonstrate running Flutter using the default launch settings.

In order to customize your Flutter runtime, you can also specify the Dart entrypoint, library, and route.

Dart entrypoint

Calling `run` on a `FlutterEngine`, by default, runs the `main()` Dart function of your `lib/main.dart` file.

You can also run a different entrypoint function by using `runWithEntrypoint` with an `NSString` specifying a different Dart functio

Note: Dart entrypoint functions other than `main()` must be annotated with the following in order to not be [tree-shaken](#) away when compiling:

```
@pragma('vm:entry-point')
void myOtherEntrypoint() { ... };
```

Dart library

In addition to specifying a Dart function, you can specify an entrypoint function in a specific file.

For instance the following runs `myOtherEntrypoint()` in `lib/other_file.dart` instead of `main()` in `lib/main.dart`:

[Objective-C](#)

[Swift](#)

```
[flutterEngine runWithEntrypoint:@"myOtherEntrypoint" libraryURI:@"other_file.dart"];
```

Route

An initial route can be set for your Flutter `WidgetsApp` when constructing the engine.

[Objective-C](#)

[Swift](#)

```
FlutterEngine *flutterEngine =
    [[FlutterEngine alloc] initWithName:@"my flutter engine"];
[[flutterEngine navigationChannel] invokeMethod:@"setInitialRoute"
arguments:@" /onboarding"];

[flutterEngine run];
```

This code sets your `dart:ui`'s `window.defaultRouteName` to `" /onboarding"` instead of `" /"`.

Warning: `"setInitialRoute"` on the `navigationChannel` must be called before running your `FlutterEngine` in order for Flutter's first frame to use the desired route.

Specifically, this must be called before running the Dart entrypoint. The entrypoint may lead to a series of events where `runApp` builds a `Material/Cupertino/WidgetsApp` that implicitly creates a `Navigator` that might `window.defaultRouteName` when the `NavigatorState` is first initialized.

Setting the initial route after running the engine doesn't have an effect.

[Get started](#)

[Samples & tutorials](#)

[Development](#)

▶ [User interface](#)

▶ [Data & backend](#)

▲ [Accessibility & internationalization](#)



[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](#).

▶ [Adding to an Android app](#)

▼ [Adding to an iOS app](#)

[Project setup](#)

[Add a single Flutter screen](#)

[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](#)

Tip: In order to imperatively change your current Flutter route from the platform side after the `FlutterEngine` is already running, use `pushRoute()` or `popRoute()` on the `FlutterViewController`.

To pop the iOS route from the Flutter side, call `SystemNavigator.pop()`.

See [Navigation and routing](#) for more about Flutter’s routes.

Other

The previous example only illustrates a few ways to customize how a Flutter instance is initiated. Using [platform channels](#), you’re free to push data or prepare your Flutter environment in any way you’d like, before presenting the Flutter UI using a `FlutterViewController`.