Article

Placing content in a bundle

Place bundle content in the correct location based on its type.

Overview

A *bundle* is a directory with a standardized hierarchical structure that typically contains executable code and the resources used by that code. Bundles fulfill many different roles: apps, app extensions, frameworks, and plug-ins are all bundles. Bundles can also contain other bundles; for example, an app may contain an app extension.

A bundle has a set of standard locations to hold content. The correct location to place content within a bundle depends on the content type. For example, you must place an app extension in the location reserved for plug-ins and a storyboard in the location reserved for resources. Not all locations are appropriate on all platforms and, similarly, not all platforms support all content types.

Xcode understands the bundle structure and, if you use Xcode to build your bundle, it places content correctly based on its type. If you don't use Xcode to build your software product, use the information below to place your bundled content in the right location. Even if you use Xcode, you might find this information useful as you check the final structure of your product.

Important

If you put content in the wrong location, you may encounter hard-to-debug code signing and distribution problems. These problems aren't always immediately obvious. For example, when building a Mac app, incorrectly placed code might work during day-to-day development, but might cause problems during notarization.

Place content based on type and platform

Bundled content includes the bundle's `Info.plist`, code content and resources:

- The `Info.plist` is a property list file stored at a location that identifies a directory hierarchy as a bundle. For a list of `Info.plist` keys, see `Information Property List`.
- Code content is either executable code, like a helper tool, or another bundle that contains executable code, like an app extension. In this context, executable code means a Mach-O image. It doesn't include things like shell scripts, Python scripts, and AppleScripts (unless you save the AppleScript as an application). Although you execute a script, it has no place to hold a code signature, so you treat it as a resource.
- Resources are everything that's not code.

When adding content to a bundle, place it according to the rules in the following table:

- If the location ends with a slash (`/`), place the item within that directory. Otherwise, place a single item of that type at that location.
- A location of `/` indicates the root of the bundle.
- The macOS platform covers both Mac and Mac Catalyst apps.

Content type	Platform	Location
`Info.plist`	macOS	`Contents/Info.plist`
	macOS framework	`Versions/A/Resources/Info.plist`
	iOS, watchOS, tvOS, visionOS	`Info.plist`
main executable	macOS	`Contents/MacOS/`
	macOS framework	`Versions/A/`
	iOS, watchOS, tvOS, visionOS	`/`
resource	macOS	`Contents/Resources/`
	macOS framework	`Versions/A/Resources/`
	iOS, watchOS, tvOS, visionOS	`/`

Content type	Platform	Location
privacy manifest	macOS	`Contents/Resources/`
	macOS framework	`Versions/A/Resources/`
	iOS, watchOS, tvOS, visionOS	`/`
framework, dynamic library	macOS	`Contents/Frameworks/`
	macOS framework	`Versions/A/Frameworks/`
	iOS, tvOS, visionOS	`Frameworks/`
	watchOS	See Handle frameworks and Swift system libraries on watchOS section below.
app extension	macOS	`Contents/PlugIns/`
	iOS, watchOS, tvOS	`PlugIns/`
plug-in	macOS	`Contents/PlugIns/`
	macOS framework	`Versions/A/PlugIns/`
	iOS, watchOS, tvOS, visionOS	`PlugIns/`
provisioning profile	macOS	`Contents/embedded.provisionprofile`
	iOS, watchOS, tvOS, visionOS	`embedded.mobileprovision`
help app, helper tool	macOS	`Contents/MacOS/`
		`Contents/Helpers/`

Content type	Platform	Location
	macOS framework	`Versions/A/Helpers/`
XPC Service	macOS	`Contents/XPCServices/`
Automator action	macOS	`Contents/Library/Automator/`
QuickLook generator	macOS	`Contents/Library/QuickLook/`
privileged helper tool	macOS	`Contents/Library/LaunchServices/`
Service Management login item	macOS	`Contents/Library/LoginItems/`
Spotlight importer	macOS	`Contents/Library/Spotlight/`
system extension	macOS	`Contents/Library/SystemExtensions/`
App Clip	iOS	`AppClips/`
watchOS app	iOS	`Watch/`

iOS, watchOS, and tvOS support third-party frameworks but don't support third-party standalone dynamic libraries, which are those outside a framework bundle, typically with the `.dylib` filename extension. The only exception to this rule is the Swift system libraries provided by Xcode.

iOS and tvOS support frameworks and Swift system libraries at the topmost app level; a nested bundle, like an app extension, can't include a framework.

For more information about embedding frameworks and Swift system libraries on watchOS, see Handle frameworks and Swift system libraries on watchOS.

For information on localizing resources, see Localized Resources in Bundles.

Handle frameworks and Swift system libraries on watchOS

A watchOS app consists of an iOS app that contains a watchOS app, that then contains a WatchKit extension. This nesting is present even for watch-only apps. When embedding frameworks and Swift system libraries in a watchOS app, you need to:

- Place iOS frameworks and Swift system libraries in the iOS app's `Frameworks` directory. For example, for an iOS app called MyApp, place iOS frameworks and Swift system libraries in `MyApp.app/Frameworks/`.
- Place the watchOS Swift system libraries in the WatchKit app's `Frameworks` directory, for example, `MyApp.app/Watch/MyApp WatchKit App.app/Frameworks/`.
- Place watchOS frameworks in the WatchKit extension's `Frameworks` directory, for example, `MyApp.app/Watch/MyApp WatchKit App.app/PlugIns/MyApp WatchKit Extension.appex/Frameworks/`.

Support a single framework version on macOS

A macOS framework is a bundle that uses a unique format. The framework's root contains a `Versions` directory that holds one or more versions of the framework, each of which has its own bundle-like structure. On ancestor platforms to macOS, multiple versions of a framework can coexist within this versioned bundle. On macOS, however, best practice is to use a single version named `A`.

If you build your macOS framework in Xcode, it generates the correct structure automatically. If you're working outside of Xcode, follow these rules to structure your framework for maximum compatibility:

1. Create a single `Versions` directory at the framework's root.
2. Within that, create a directory name `A`.
3. Populate `Versions/A` with your framework content.
4. Create a symlink within `Versions` called `Current` that targets `A`.
5. Create a symlink in the framework's root that target's the framework's executable through `Versions/Current`. For example, if you're creating the CoreWaffleVarnishing framework, create a symlink called `CoreWaffleVarnishing` that targets the framework's executable at `Versions/Current/CoreWaffleVarnishing`.
6. Create a symlink in the framework's root that target's the framework's `Resources` directory through `Versions/Current`.
7. Optionally, create other symlinks in the root that target similarly named items through `Versions/Current`.

The final structure looks like this:

```
CoreWaffleVarnishing.framework/
  CoreWaffleVarnishing -> Versions/Current/CoreWaffleVarnishing
  Resources -> Versions/Current/Resources
```

```
  Versions/
    Current -> A
    A/
      CoreWaffleVarnishing
      Resources/
        Info.plist
        … other resources …
```

Important

The framework's root must contain only the `Versions` directory and symlinks. Don't place any other content there. Doing so causes code-signing problems. Similarly, the `Versions` directory must contain only version directories — ideally just one, called `A` — and a symlink called `Current` that targets one version directory.

Place code content directly in its location

Each code location must contain a flat list of code content. If you have a lot of code content, you might be tempted to group it in nested directories, for example:

```
WaffleVarnisher.app/
  Contents/
    …
    PlugIns/
      Waffles/
        Belgian.plugin
        Buttermilk.plugin
        BananaCaramel.plugin
        …
      Varnishes/
        Gloss.plugin
        Satin.plugin
        Matte.plugin
    …
```

Don't group your nested code in this way. Although it might work in some situations, it might fail later in hard-to-debug ways.

If you ignore this recommendation and use this structure, don't use a dot (`.`) in your directory names, such as `Waffles` and `Varnishes` in the example above. The code-signing machinery

assumes that any directory with a name that contains a dot is a bundle, and then fails when signing that directory because it's not a well-formed bundle.

Codeless bundles

A codeless bundle has no executable code. For example, some apps use a codeless bundle as part of their plug-in support. A codeless bundle can hold a code signature, so you can sign it either as code or a resource, depending on the circumstances:

- If you distribute a codeless bundle independently, sign it as code.
- If you embed a codeless bundle within another bundle, sign it as code if the embedding location typically holds code, for example, `Contents/PlugIns`. Otherwise, sign the codeless bundle as a resource.