Article

Making your existing app compatible with visionOS

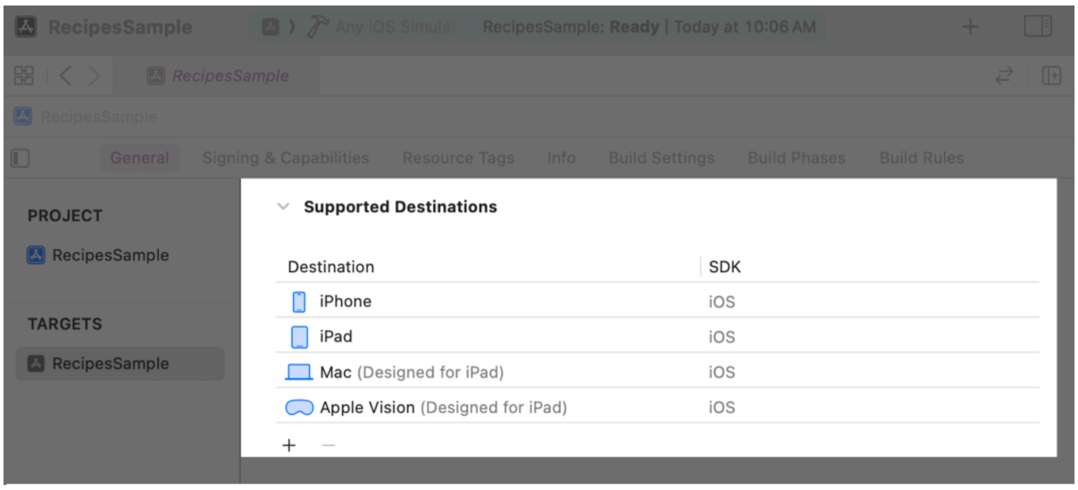
Modify your iPadOS or iOS app to run successfully in visionOS as a compatible app.

Overview

A compatible iPadOS or iOS app links against the iOS SDK and runs in visionOS without needing to add a visionOS destination. Although visionOS provides a complete set of iOS frameworks for linking, some features of those frameworks might be unavailable due to hardware or usage differences. Read Determining whether to bring your app to visionOS for details on behavior changes, API availability checks, and deprecated frameworks. To ensure your app runs correctly in visionOS, handle any unavailable features gracefully and provide workarounds wherever possible.

Run as a compatible app in visionOS

The App Store makes compatible iPad and iPhone apps available in visionOS automatically after you sign the updated Apple Developer Program License Agreement. If you have an app in the iOS App Store, try downloading it on Apple Vision Pro and run it. If you built your app using the iOS SDK, Xcode 15 and later automatically adds a Designed for iPad runtime destination to your project.



Supported Destinations

Destination	SDK
iPhone	iOS
iPad	iOS
Mac (Designed for iPad)	iOS
Apple Vision (Designed for iPad)	iOS

Use this destination to run your app and test its compatibility in visionOS. Depending on your app, you might need to make additional changes to account for features that are only found in the iOS SDK. You can test most of your app's core functionality in Simulator, but some features are available only on a device.

Handle SDK differences gracefully

visionOS contains most of the same technologies as iPadOS and iOS, but there are differences. In some cases, a feature you use in your app might not be available because of hardware differences or because of differences in how people use a visionOS device. As part of your testing, consider the impact of any unavailable features on your app's overall experience. Whenever possible, work around unavailable features by disabling them or providing alternate ways to access the same content.

The following features aren't available in compatible iPad and iPhone apps in visionOS. Use framework APIs listed in Determining whether to bring your app to visionOS to determine when the features are available.

- Core Motion services
- Barometer and magnetometer data
- All location services except the standard service
- HealthKit data
- Video or still-photo capture
- Camera features like Auto Focus or flash
- Rear-facing (selfie) cameras

Note

Although device cameras are unavailable on Apple Vision Pro, compatible apps can capture photos of personas using `AVCaptureDevice` with the position of `.front` or use continuity camera to capture photo and video from iPhone and iPad.

If your app uses an unsupported feature but can function without it, you can still bring your app to visionOS. Remove features that aren't available and focus on bringing the rest of your content to the platform. For example, if you have an app that lets people write down notes and take pictures to include with those notes, disable the picture-taking ability in visionOS but let people add text and incorporate images from their library.

If your app relies on frameworks that behave differently in visionOS, update your code to handle those differences. Throughout your code, make sure you respond to unusual situations:

- **Use availability checks.** Availability checks give you a clear indication when you can't use a feature, but some frameworks might have more subtle behavior. Read Determining whether to bring your app to visionOS for a list of frameworks and features with availability checks.
- **Remove deprecated code.** If your app currently uses deprecated APIs or frameworks, update your code to use appropriate replacements. Read Determining whether to bring your app to visionOS for a list of deprecated frameworks.
- **Handle error conditions.** If a function throws an exception or returns an error, handle the error. Use error information to adjust your app's behavior or provide an explanation of why it can't perform certain operations.
- **Handle `nil` or empty values gracefully.** Validate objects and return values before you try to use them.
- **Update your user interface.** Provide appropriate messaging in your interface when a feature is unavailable, or remove feature-specific views entirely if you can do so cleanly. Don't leave empty views where the feature was.

Adapt to device differences

Apple frameworks take a device-agnostic approach whenever possible to minimize issues when you use them on different device types. Apple devices come in a variety of shapes and sizes, with different sets of features. Rather than build your app for a specific device, make sure it adapts to any device and can gracefully handle differences.

Build robustness into your app during the design process. Avoid assumptions that might cause your app to break when it runs on a new device, and make sure your app adapts dynamically to different conditions. For example:

- **Don’t assume the device type or idiom is always iPhone, iPad, or iPod Touch.** Avoid decisions based on the current idiom. If you do rely on the current idiom, provide reasonable defaults for unknown idioms.
- **Design your app to handle unavailable hardware or features.** Specific hardware and features might be unavailable for many different reasons. For example, a feature might be unavailable when your app runs in Simulator. Perform availability checks whenever possible, and handle unavailable features gracefully.
- **Design your windows and views to adapt dynamically.** Build your interface to adapt dynamically to any size using SwiftUI or Auto Layout. Assume the size of your app can change dynamically.
- **Don’t assume the device has a specific number of displays.** People can connect iPad and iPhone to an external display, and visionOS devices use two displays to create a stereoscopic version of your app’s content.
- **Don’t make assumptions based on the available frameworks or symbols.** The presence or absence of frameworks or code symbols is an unreliable way to identify a device type and can change in later software updates. See Determining whether to bring your app to visionOS for information on using availability checks to identify which features are available for a given framework.
- **Don’t assume your app runs in the background.** visionOS doesn’t support location, external accessory, or Bluetooth-peripheral background execution modes.
- **Don’t assume that background apps are hidden.** In visionOS, the windows of background apps remain visible, but are dimmed when no one looks at them. The only time app windows disappear is when one app presents an immersive space.

Audit your interface code

To minimize disruptions, visionOS runs your compatible iPad or iPhone app in an environment that matches an iPad as much as possible. Windows and views retain the same appearance that they have in iPadOS or iOS, and the system sizes your app’s window to fit an iPad whenever possible.

When building your app’s interface, make choices that ensure your app runs well in visionOS too. Adopt the following best practices for your interface-related code:

- **Support iPad and iPhone in the same app.** Create one app that supports both device types, rather than separate apps for each device. SwiftUI and UIKit support adaptable interfaces, and Xcode provides tools to help you visualize your interface at different supported sizes.
- **Organize your interface using scenes.** Scenes are a fundamental tool for managing your app’s interface. Use the scene types in SwiftUI and UIKit to assemble and manage the views you display in windows.

- **Adapt your interface to any size.** Design your interface to adapt naturally to different sizes. For an introduction to SwiftUI views and layout, see Declaring a custom view. For information about laying out views in UIKit, see View layout.
- **Don't access screen details.** visionOS provides reasonable values for `UIScreen` objects, but don't use those values to make decisions. Instead, create an adaptive layout or use Auto Layout to make your app look good on all platforms.
- **Don't rely on the status bar for layouts.** `statusBarFrame` returns `CGRectZero` on visionOS. Use safe areas to calculate layout instead.
- **Specify the supported interface orientations.** Add the `UISupportedInterfaceOrientations` key to your app's `Info.plist` file to specify the interface orientations it supports. Support all interface orientations whenever possible. visionOS adds an interface rotation for your app button only when this key is present. To display your app's interface in a particular orientation at launch, add the `UIPreferredDefaultInterfaceOrientation` key to your app's `Info.plist` file. Set the value of the key to one of the values in your app's `UISupportedInterfaceOrientations` key. Add `~ipad` or `~iphone` to the key name, for example, `UIInterfaceOrientationPortrait~ipad`, to specify device-specific orientation preferences.
- **Adopt vector-based images when possible.** Vector-based images scale well to different sizes while retaining a crisp appearance. If you use bitmap-based assets, make them the exact size you need. Don't use oversized assets, which require extra work to display at the correct size.
- **Update hover effects in custom views.** Hover effects convey the focused view or control in your interface. Standard system views apply hover effects as needed. For custom views and controls, verify that the hover effects look appropriate in visionOS. Add or update the content shape for your hover effects if needed. The following example uses SwiftUI to add a rectangle-shaped hover effect to a rectangle in a view:

```
Rectangle()
  .contentShape(.hoverEffect, .rectangle)
  .hoverEffect()
```

Create adaptive layouts in UIKit

If your UIKit app uses hardcoded values or relies on `UIScreen` for layout, the first step to migrating your app to visionOS is to use an adaptable layout. When you make decisions using device details, your app might produce inconsistent or erroneous results on an unknown device type, or it might fail altogether. Find solutions that rely on environmental information, rather than the device type. For example, SwiftUI and UIKit start layout using the app's window size, which isn't necessarily the same size as the device's display.

Note

Device-specific information is available when you absolutely need it, but validate the information you receive and provide reasonable default behavior for unexpected values.

Think about ways to create adaptive layouts using the following techniques:

- **Use stack views.** `UIStackView` objects adjust the position of their contained views automatically when interface dimensions change. Alternatively, Auto Layout constraints let you specify the rules that determine the size and position of the views in your interface.
- **Stay within layout margins.** Read Positioning content within layout margins to set up constraints that respect layout margins and don't crowd other content.
- **Respect the safe area.** Place views so they're not obstructed by other content. Each view has a layout guide that helps you create constraints to position your views within the safe area, which adapt to the current device automatically. Read Positioning content relative to the safe area for guidance.
- **Adapt based on changes in UITraitCollection.** Write code to adjust your app's layout according to changes in elements of the iOS user interface, such as size class, display scale, and layout direction. Read UITraitCollection for more information.

Test specific scenarios before uploading your app

The following App Store features for iOS continue to work when your app runs in visionOS:

- In-app purchases and subscriptions
- App capabilities and entitlements
- On-demand resources
- App thinning

When you use app thinning to optimize your app for different devices and operating systems, the App Store selects the resources and content that offer the best fit for visionOS devices. It then removes any other resources to create a streamlined installation of your app. When you export your app from Xcode 15 or later, you can test the thinning support using the visionOS virtual thinning target.

See Also

iOS migration and compatibility

Determining whether to bring your app to visionOS

Decide whether to bring your existing iPadOS or iOS app to visionOS.

Bringing your existing apps to visionOS

Build a version of your iPadOS or iOS app using the visionOS SDK, and update your code for platform differences.

Bringing your ARKit app to visionOS

Update an iPadOS or iOS app that uses ARKit, and provide an equivalent experience in visionOS.