Interface fundamentals

Explore the components that go into building your app's interface, and discover platform-specific features that improve the experience you offer to people.

To build your app's interface, you can use standard system views, draw content yourself, or mix custom drawing with the standard views. Regardless of how you create your content, all interfaces rely on some standard components to present that content:

- *Windows* are the primary containers for your app's content, and they also facilitate system-related interactions. Every SwiftUI, UIKit, and AppKit app has at least one window, and some platforms let your app display multiple windows simultaneously. Windows in macOS, iPadOS, and visionOS can have a visible border and controls to change the size of the window. Windows in iOS, tvOS, and watchOS have no visible appearance of their own.
- *Scenes* manage instances of your app's interface in iOS, iPadOS, tvOS, visionOS, and watchOS. Every SwiftUI and UIKit app has at least one scene, and you can create additional scenes to manage distinct experiences. Each scene manages the data for one instance of your interface and any relevant system behaviors. For example, each scene object handles transitions between the foreground and background execution states for that scene. AppKit apps don't use scenes.
- *Views and controls* display specific types of content in your interface. SwiftUI, UIKit, and AppKit provide views for displaying standard types of content like images, text, collections, pickers, buttons, toggles, and much more. They also define the architecture that you use to create custom views and display any content you want.
- *Volumes* are a specific type of window that you use to showcase 2D and 3D content in visionOS.

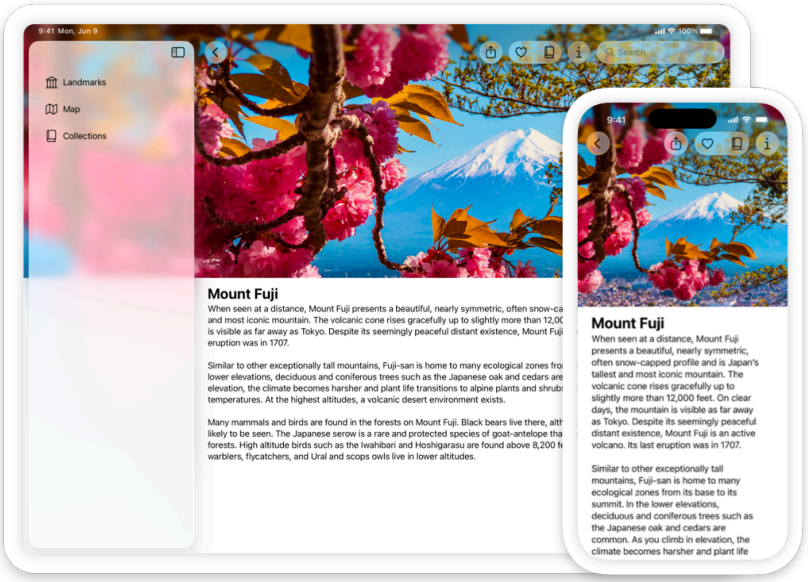
The app-builder technologies you use to create your app maintain a separation between your app's data and the views and other interface elements you use to display that data. Your data model objects are always the source of truth for your app's content. When you configure a view, you pass a copy of your data to the view for display purposes. If interactions with a view cause a value to change, your app's infrastructure is responsible for synchronizing the changes with your

data objects. When you use SwiftUI, this process is more automatic, but with UIKit and AppKit you synchronize the changes yourself.

Choose a design approach for the target platform

The platform you target naturally affects the approach you take to building your interface. You want apps to feel like they belong on a given platform, which might require you to tweak your interface on each platform.

iOS and iPadOS



Design iOS and iPadOS apps as experiences people can take with them anywhere. Apps in iOS fill the screen, and apps in iPadOS need to be flexible enough to fill all or part of the screen. Because space is more constrained, interfaces make greater use of layout and navigation elements to organize content.

iOS and iPadOS apps need to handle different iPhone and iPad sizes and orientations gracefully. Make sure your app's interface adjusts automatically to size changes. In SwiftUI, automatic layout is an integral part of the interface-creation process. In UIKit, adopt Auto Layout constraints to adjust your interface at appropriate times. Take advantage of the Xcode interface editors so you can see how your interface looks in different configurations.

On iPad, don't forget to support features that people can access from Magic Keyboard and Apple Pencil. Magic Keyboard adds pointer-based navigation, menu support, and hover-based interactions. Apple Pencil enables high-precision, low-latency input and also lets you capture altitude, azimuth, and pressure information, which a drawing app can use for content creation.

macOS

Design your macOS app to take advantage of the power, space, and flexibility of a Mac. Mac gives you more space for your content, but that doesn't mean you want a cluttered interface. Take advantage of the same navigation techniques as other platforms to create interfaces that present what people need when they need it.

Make sure your windows support flexible layouts and adapt to different sizes and modes. In SwiftUI, automatic layout is an integral part of the interface-creation process. In AppKit, adopt Auto Layout constraints to adjust your interface at appropriate times. Adopt full-screen mode for windows to give people the option to view your app in a distraction-free environment.

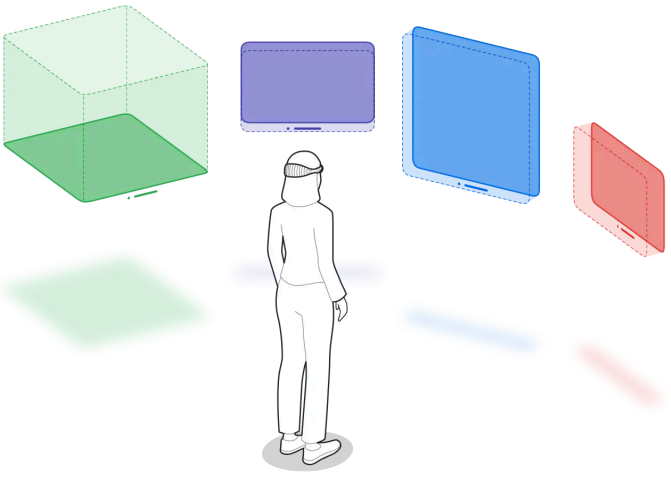
The menu bar along the top of the desktop displays your app's menus. Identify your app's relevant actions, and craft menus that reflect how people interact with your content.

tvOS

Design your tvOS interface with focus-based navigation in mind. Most interactions with your app occur through the Siri Remote. People use the directional buttons on the remote to change focus from one part of your UI to another. They then use the select button to act on the focused item, or the Menu button to navigate back to the previous screen. Make navigation as straightforward as possible, and minimize text input and other complex interactions.

Lockups are one way to simplify navigation and promote consistency among similar items in your UI. A lockup is a group of related views that you combine into a single, selectable element. For example, a movie lockup might include the movie's title, description, cast list, and poster image. When someone selects a movie, tvOS places focus on the entire lockup instead of on individual items.

visionOS



Design your visionOS interface around an initial window to provide a familiar starting point for interactions. Add depth-based offsets to specific views to emphasize parts of your window, or to indicate a change in modality. Incorporate 3D objects directly into your view layouts to place them side by side with your 2D views. Add hover effects to highlight elements when someone looks at them. Place frequently used tools and commands on the outside edge of your windows using ornaments.

Showcase your app's 3D content by adding a volume, or add a Full Space to create an immersive experience. Build and animate your app's 3D content as USD assets and incorporate them into your scenes using RealityKit.

watchOS

Design your watchOS app to deliver only the most relevant content in a timely manner. Be prepared to support different sizes of Apple Watch, ranging from 38mm to 45mm. On Apple Watch models that support the Always-On display, be prepared to keep your app's content up-to-date. In addition to the app you build, support the following additional interfaces:

- *Widgets* elevate a small amount of timely, personally relevant information, and display it where people can see it at a glance. On Apple Watch, widgets appear in Smart Stacks.
- *Complications* are small visual elements that appear on the watch face in predefined areas. Use them to display your app's most crucial information, and to help people engage with your app's content. Build them using the WidgetKit framework and deliver them with your app.
- *Notifications* display time-sensitive information, using haptics, sound, and visual cues to get the person's attention. Create notifications locally from your app, or push notifications to Apple Watch remotely from a server. Provide a custom interface to incorporate custom content, images, or media.

Manage assets for your interface

Most apps incorporate data from external files into their content. For example, an app's interface might incorporate images, icons, audio, textures, and other types of content. An app can also store data files with any content, and use them to configure views or other parts of the app. All of these external files are assets for you to manage.

For images in your app's interface, use SF Symbols whenever possible. The SF Symbols app offers a vast collection of configurable, vector-based images that adapt naturally to appearance and size changes. They also blend well with the San Francisco system font, resulting in a consistent look across Apple platforms.

Store images, colors, and appearance-sensitive items in asset catalogs. Interfaces need to adapt to different appearances, including light and dark appearances and accessibility settings for people with visual impairments. Asset catalogs simplify the process of loading these items into your app. Request the item you want and let the asset catalog return the best variant for the device.

Store critical resources inside your app bundle, and store larger assets on a server and download them later. Your app bundle must contain all of the resources required to launch and run your app. For larger assets that aren't critical to your app's execution, store them on the App Store or your own server and download them using the Background Assets framework.

To load resource files present in your app bundle:

- Load image assets using the `Image` type (SwiftUI), `UIImage` type (UIKit), or `NSImage` type (AppKit). Each type provides methods to load images from a file or asset catalog.
- Load color assets using the `Color` type (SwiftUI), `UIColor` type (UIKit), or `NSColor` type (AppKit).
- Locate other resource files in your bundle using the `Bundle` type. This type returns the URL for the filename you specify.

Apply standard behaviors to your interface

There are a few features you always want to build into your interface:

- *Automatic layout*. Size and position the views of your interface using a rules-based approach, which allows your app to adapt automatically to different device sizes and orientations. Automatic layout is integral to SwiftUI, but you must add specific rules to your UIKit and AppKit interfaces.
- *Internationalization*. Prepare your app for localization using the Foundation framework, which provides code to format strings, dates, times, currencies, and numbers for different languages and regions. Ensure your UI looks good for both left-to-right and right-to-left languages. Localize app resources and add them to your Xcode project.
- *Accessibility*. Apple builds accessibility support into its technologies, and screen readers and other accessibility features use that information to help people navigate your app. Review the default information that SwiftUI, UIKit, and AppKit provide and make improvements based on your content. In addition, review your app's focus-based navigation to make sure it's simple and intuitive.
- *Undo support*. Identify the actions people take in your app and build tasks that you can easily undo or redo.
- *Pasteboard*. Support data exchange in your app and the rest of the system through Cut, Copy, and Paste operations. A pasteboard type in SwiftUI, UIKit, and AppKit manages the content you transfer to it.

Consider platform-specific behaviors in your design

When making decisions about how to build your interface, the platform you target affects some of the decisions you must make. Some features might impact your interface on one platform, but not on others. The following table lists the types of decisions to consider, and the support each platform offers.

Feature	iOS	iPadOS	macOS	tvOS	visionOS	watchOS
Supported app-builder technologies	SwiftUI, UIKit	SwiftUI, UIKit	SwiftUI, AppKit	SwiftUI, UIKit, TVUIKit	SwiftUI, UIKit	SwiftUI
Full-screen content mode	Yes	Yes	Available	Yes	Available	Yes
Dark Mode	Yes	Yes	Yes	Yes	No	No
Dynamic Type	Yes	Yes	No	No	Yes	Yes
Multiple windows support	Yes*	Yes	Yes	No	Yes	No
Menu types	Context	Main, context	Main, context, Dock	None	Main, context	None
Primary interaction type	Touch	Touch, Apple Pencil, Magic Keyboard	Mouse, keyboard	Siri Remote	Eyes, hands	Touch, Digital Crown
Bluetooth keyboard support	Yes	Yes	Yes	No	Yes	No
Background task availability	Limited	Limited	Yes	No	Limited	No
CarPlay support	Yes	No	No	No	No	No

* - iOS supports multiple windows when an external display is connected.