iOS Human Interface Guidelines

Developer

Contents

Tables

UI Design Basics

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

Designing for iOS 7

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

iOS 7 embodies the following themes:

- **Deference.** The UI helps users understand and interact with the content, but never competes with it.
- **Clarity.** Text is legible at every size, icons are precise and lucid, adornments are subtle and appropriate, and a sharpened focus on functionality motivates the design.
- **Depth.** Visual layers and realistic motion impart vitality and heighten users' delight and understanding.

Weather in iOS 7

Weather in iOS 6

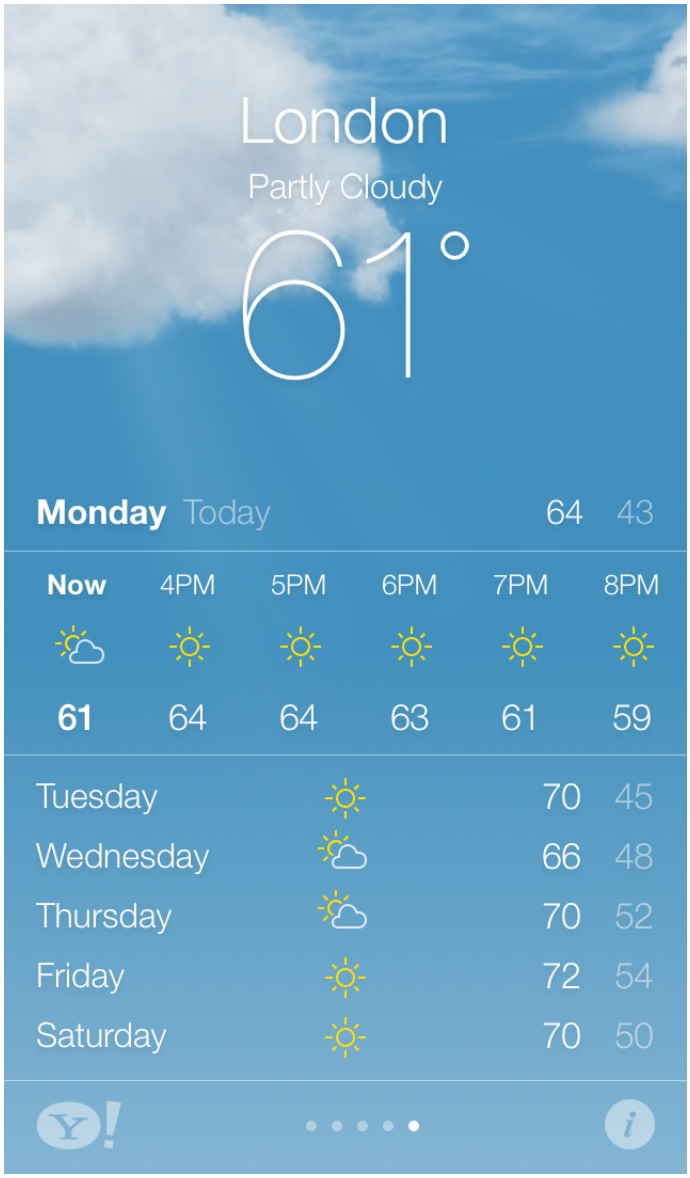
Whether you're redesigning an existing app or creating a new one, consider approaching the job in the way that Apple approached the redesign of the built-in apps:

- First, strip away the UI to expose the app's core functionality and reaffirm its relevance.
- Then, use the themes of iOS 7 to inform the design of the UI and the user experience.
- Throughout, be prepared to defy precedent, question assumptions, and let a focus on content and functionality motivate every design decision.

Defer to Content

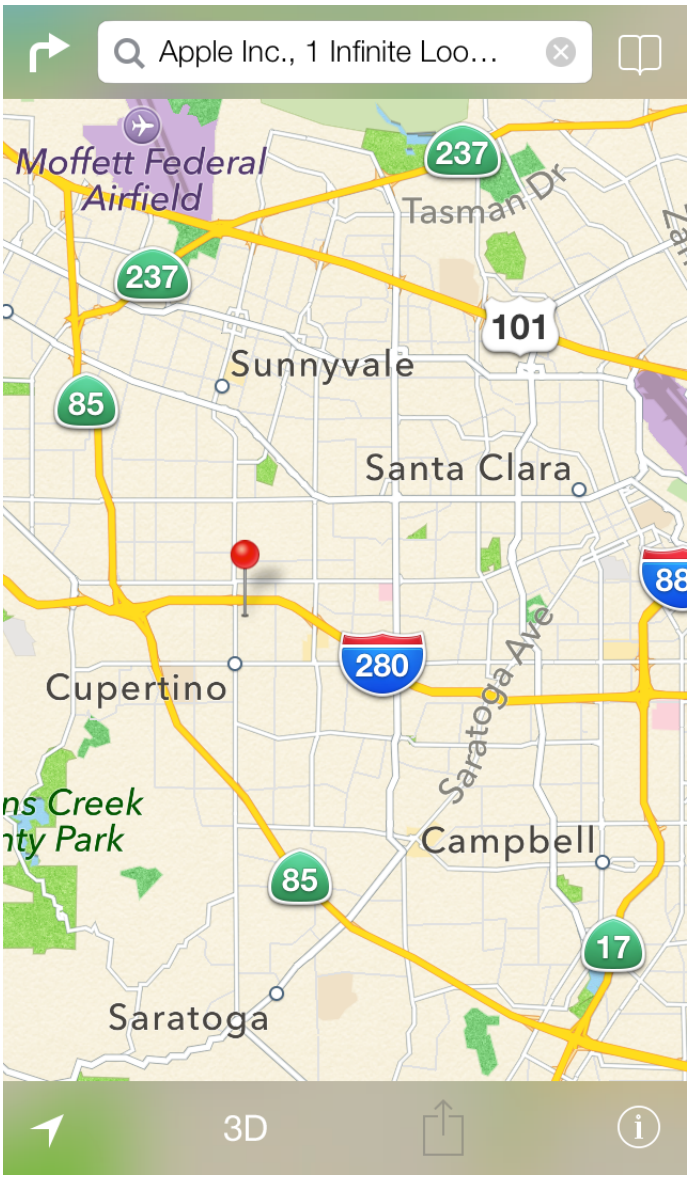
Although crisp, beautiful UI and fluid motion are highlights of the iOS 7 experience, the user's content is at its heart.

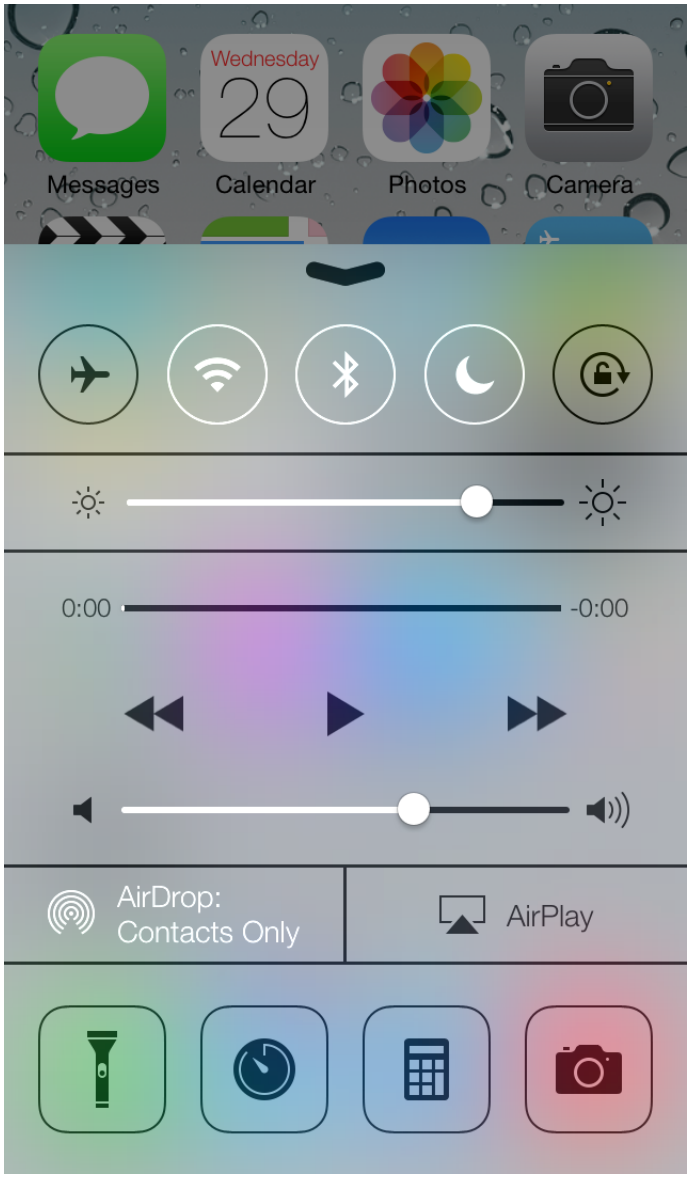
Here are some ways to make sure that your designs elevate functionality and defer to the user's content.



Take advantage of the whole screen. Reconsider the use of insets and visual frames and—instead—let the content extend to the edges of the screen. Weather is a great example of this approach: The beautiful, full-screen depiction of the current weather instantly conveys the most important information, with room to spare for hourly data.

Reconsider visual indicators of physicality and realism. Bezels, gradients, and drop shadows sometimes lead to heavier UI elements that can overpower the content. Instead, focus on the content and let the UI play a supporting role.

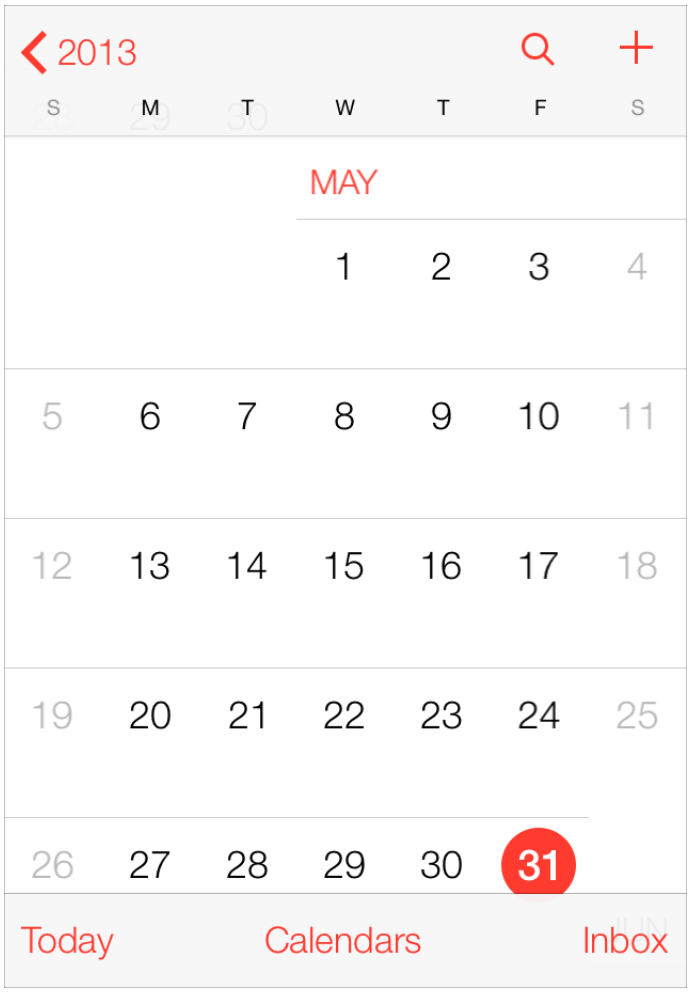




Let translucent UI elements hint at the content behind them. Translucency can provide context and help users see that more content is available.

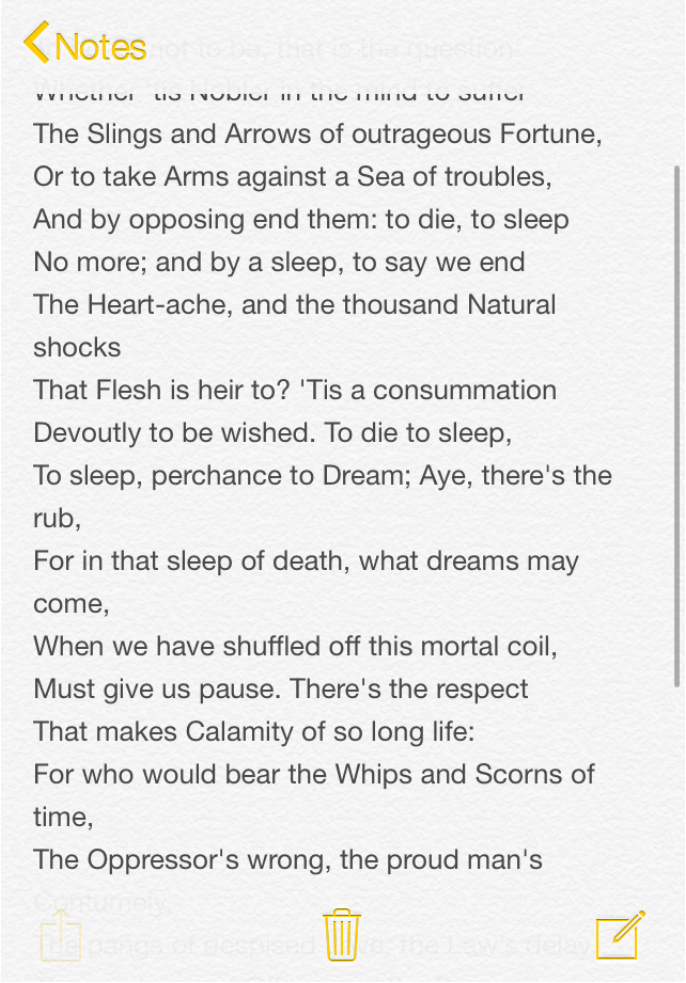
Provide Clarity

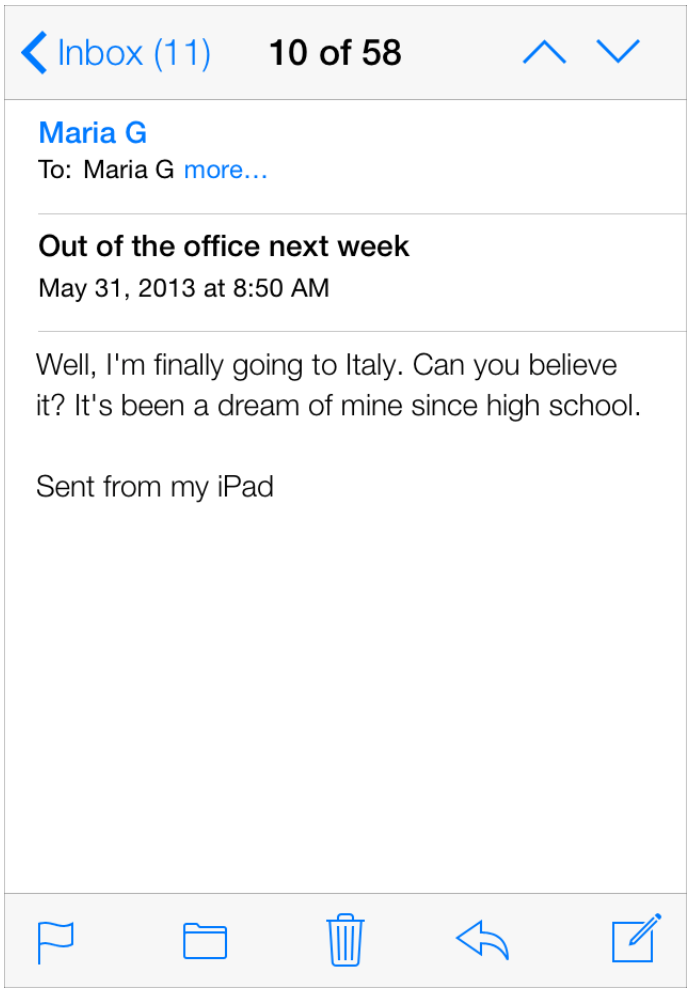
Providing clarity is another way to ensure that content is paramount in your app. Here are some ways to make the most important content and functionality clear and easy to interact with.



Use plenty of white space. White space makes important content and functionality more noticeable. Also, white space can impart a sense of calm and tranquility, and it can make an app look more focused and efficient.

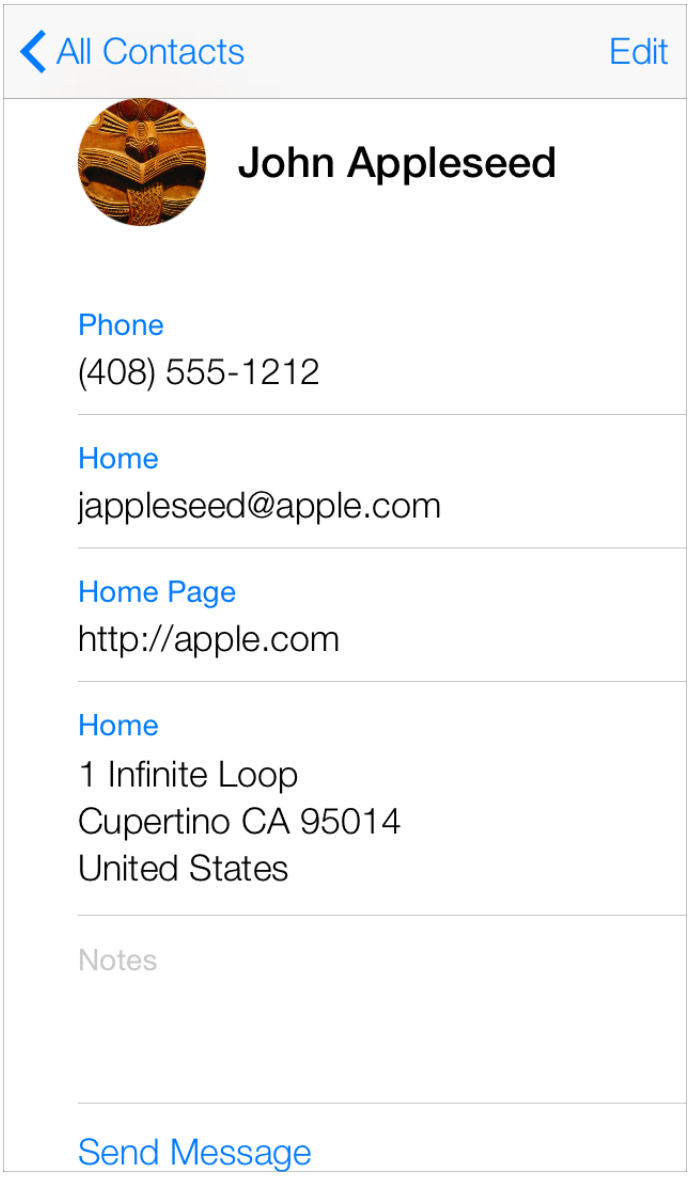
Let color simplify the UI. A key color—such as yellow in Notes—highlights important state and subtly indicates interactivity. It also gives an app a consistent visual theme.





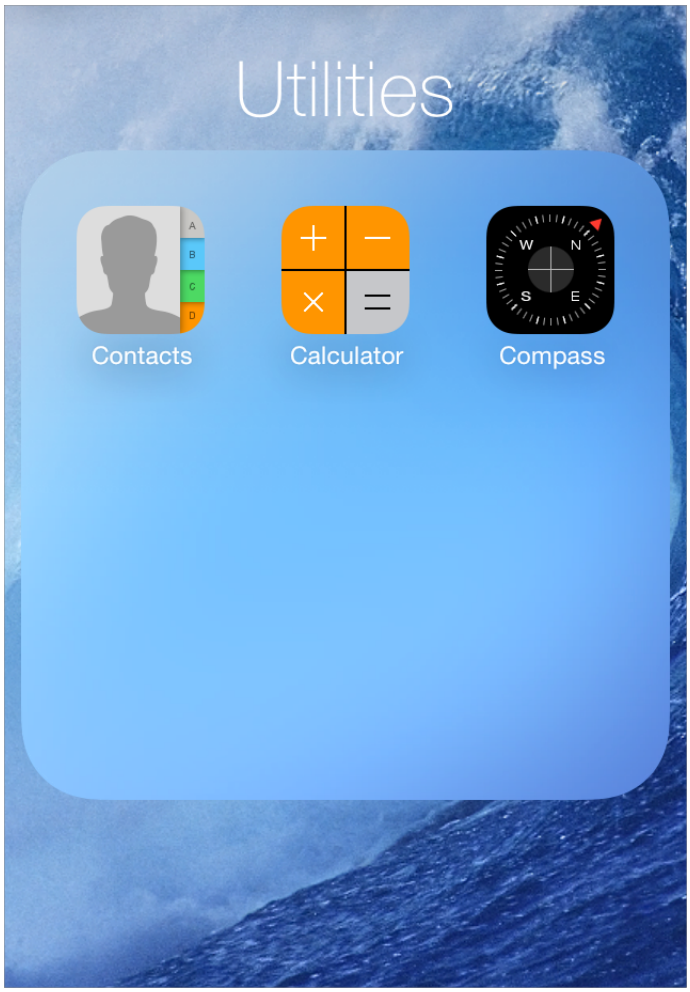
Ensure legibility by using the system fonts. iOS 7 system fonts automatically adjust letter spacing and line height so that text is easy to read and looks great at every size the user chooses.

Embrace borderless buttons. Instead of relying on a bezel to indicate interactivity, craft meaningful button titles and use a key or system color to show users that elements are interactive. The contact view uses the system blue color to show users that most onscreen items respond to interaction.



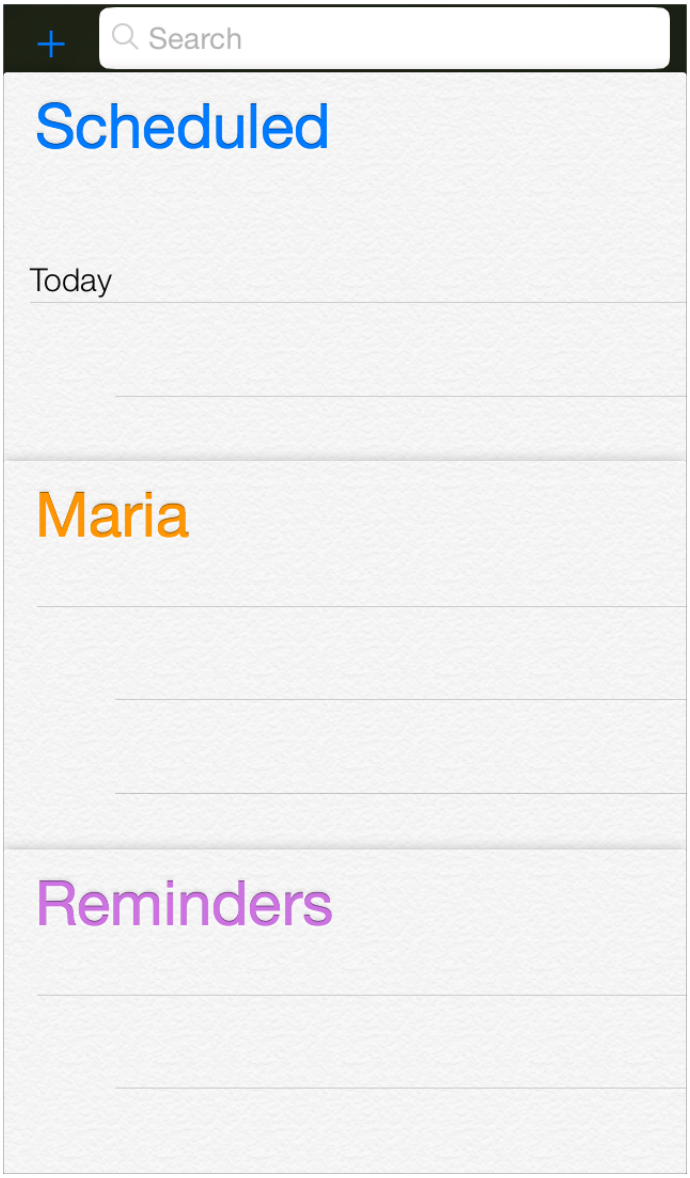
Use Depth to Communicate

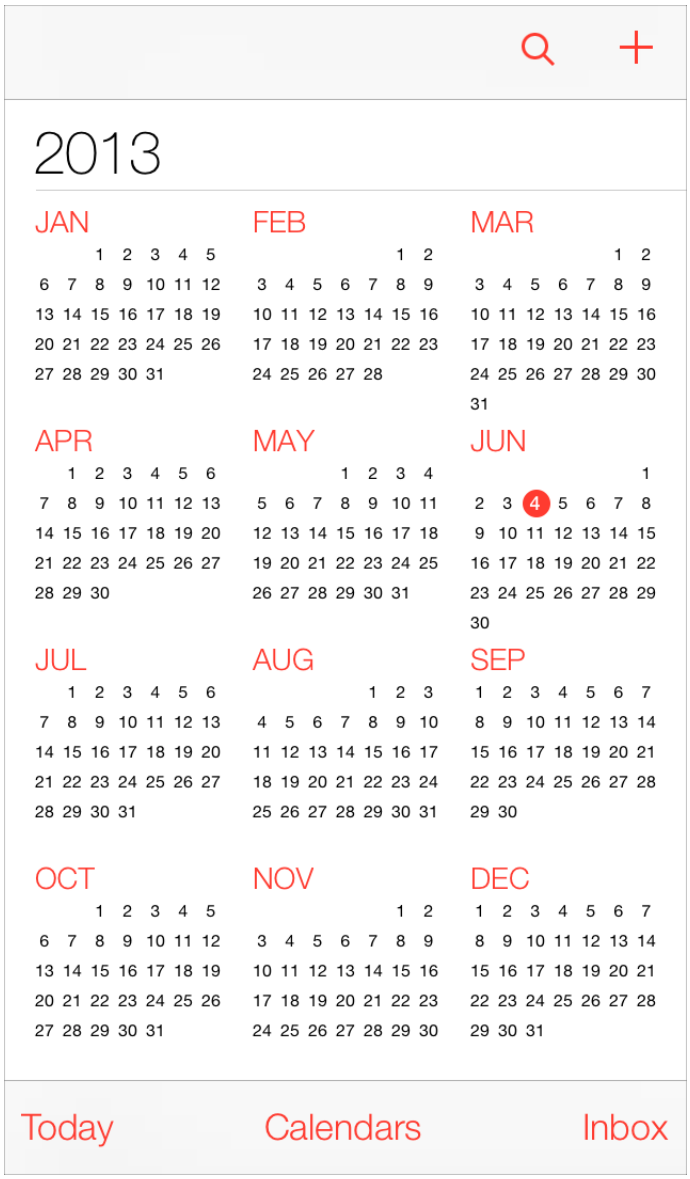
iOS 7 often displays content in distinct layers that convey hierarchy and position, and that help users understand the relationships among onscreen objects.



By using a translucent background and appearing to float above the Home screen, folders separate their content from the rest of the screen.

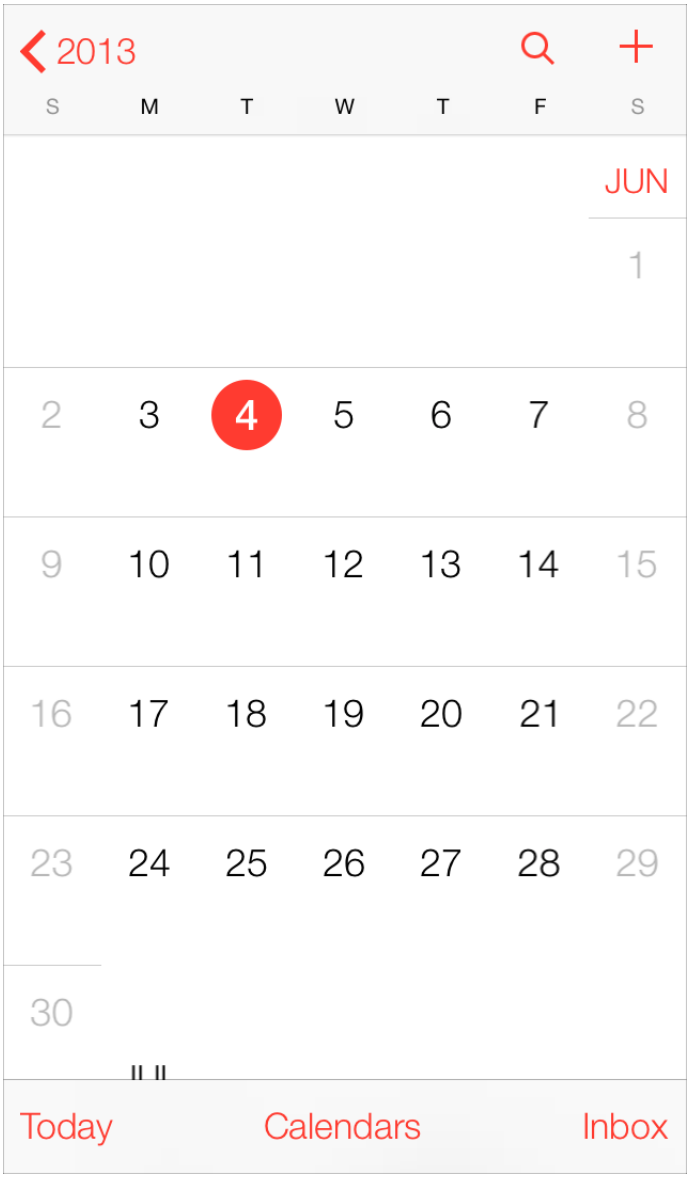
While users work with one item in Reminders, additional items are displayed in layers at the bottom of the screen. To view all items—as shown here—users expand the layers.

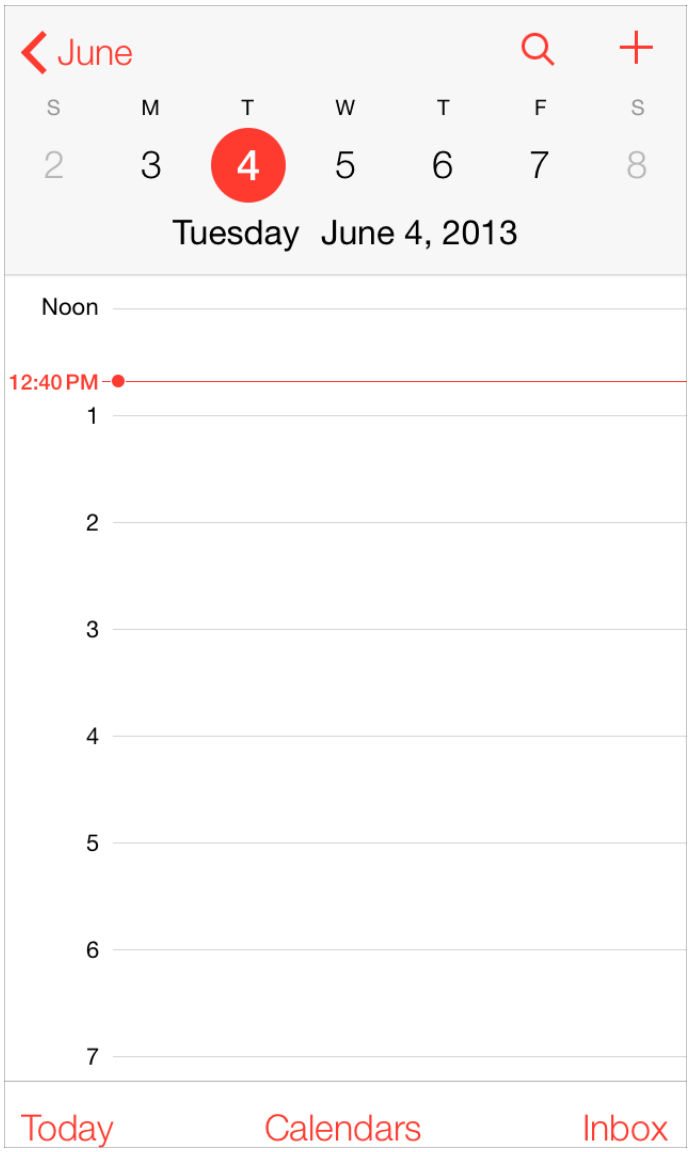




Calendar uses enhanced transitions to give users a sense of depth as they move between viewing years, months, and days. In the scrolling year view shown here, users can instantly see today's date and perform other calendar tasks.

When users select a month, the year view appears to zoom out and the month view is revealed behind it.

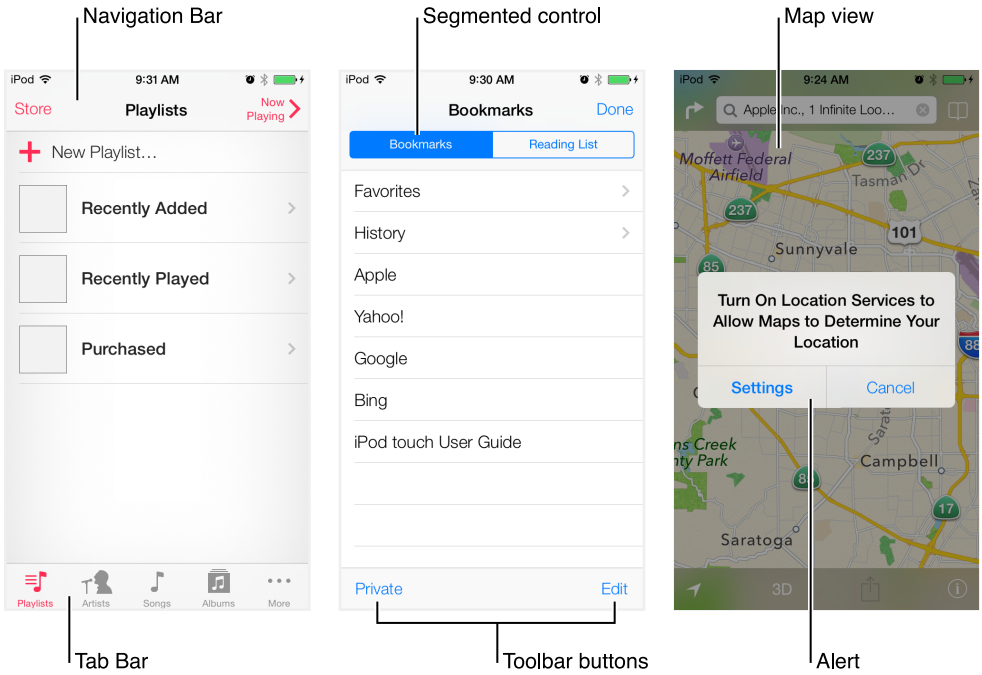




A similar transition happens when users select a day: The month view expands outward, revealing the day view.

iOS App Anatomy

Almost all iOS apps use at least some of the UI components defined by the UIKit framework. Knowing the names, roles, and capabilities of these basic components helps you make informed decisions as you design the UI of your app.



The UI elements provided by UIKit fall into four broad categories:

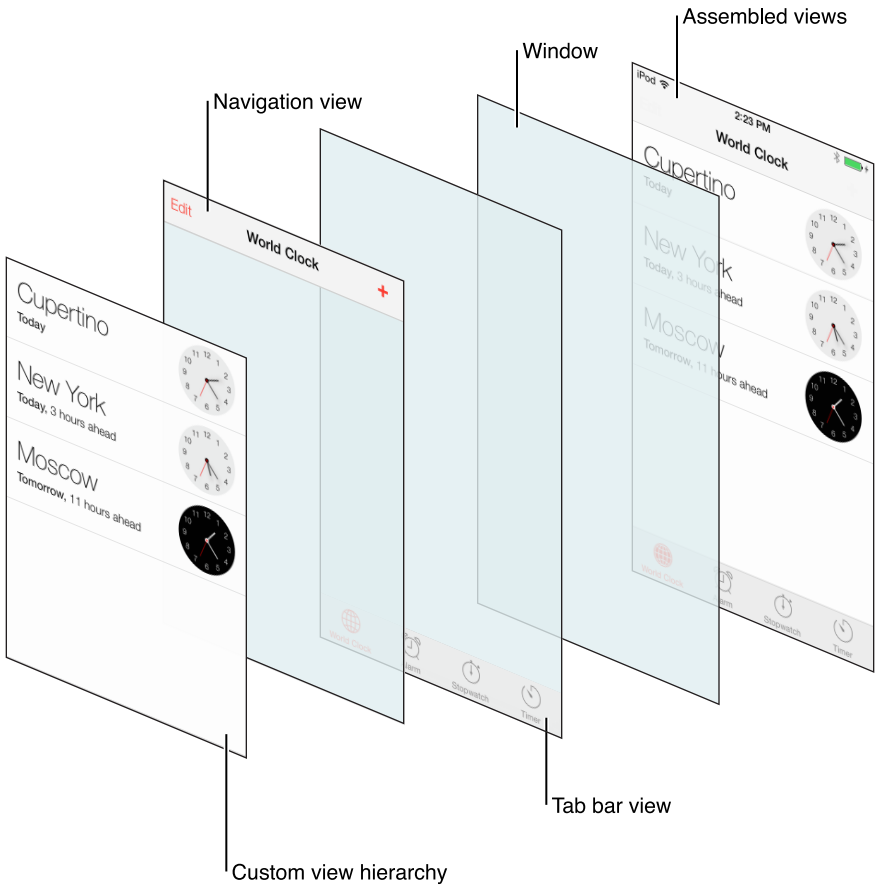
- **Bars** contain contextual information that tell users where they are and controls that help users navigate or initiate actions.
- **Content views** contain app-specific content and can enable behaviors, such as scrolling, insertion, deletion, and rearrangement of items.
- **Controls** perform actions or display information.
- **Temporary views** appear briefly to give users important information or additional choices and functionality.

In addition to defining UI elements, UIKit defines objects that implement functionality such as gesture recognition, drawing, accessibility, and printing support.

Programmatically, UI elements are considered to be types of views because they inherit from `UIView`. A view knows how to draw itself onscreen and it knows when a user touches within its bounds.

To manage a set or hierarchy of views in your app, you typically use a **view controller**. A view controller coordinates the display of views, implements the functionality behind user interactions, and can manage transitions from one screen to another.

Here's an example of how views and view controllers can combine to present the UI of an iOS app.



Although developers think in terms of views and view controllers, users tend to experience an iOS app as a collection of screens. From this perspective, a *screen* generally corresponds to a distinct visual state or mode in an app.

> **Note:** An iOS app includes a window. But—unlike a window in a computer app—an iOS window has no visible parts and it can't be moved to another location on the display. Most iOS apps contain only one window; apps that support an external display can have more than one.

In *iOS Human Interface Guidelines*, the word "screen" is used as it's understood by most users. As a developer, you might also read about screens in different contexts, where the term refers to the `UIScreen` object you can use to access an external display screen.

Starting and Stopping

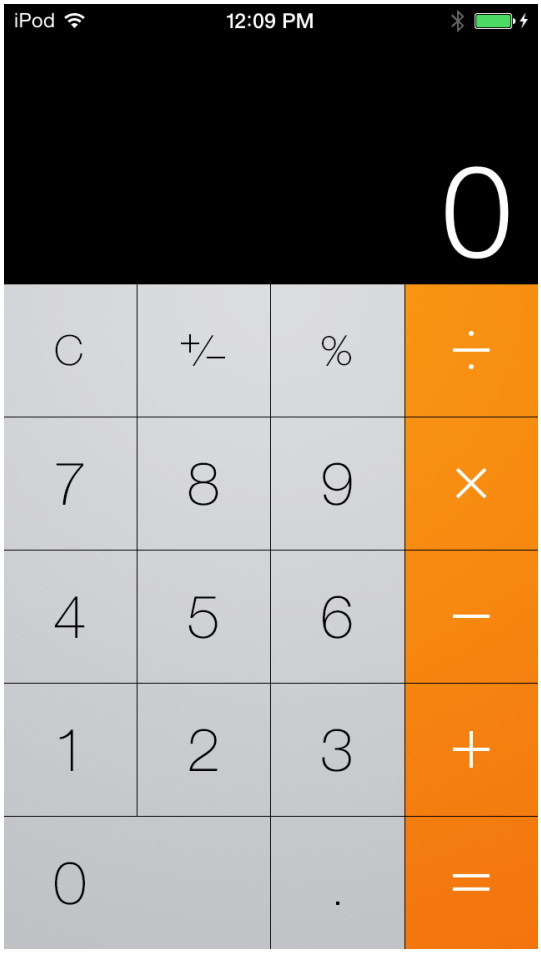
Start Instantly

It's often said that people spend no more than a minute or two evaluating a new app. When you make the most of this brief period by presenting useful content immediately, you pique the interest of new users and give all users a superior experience.

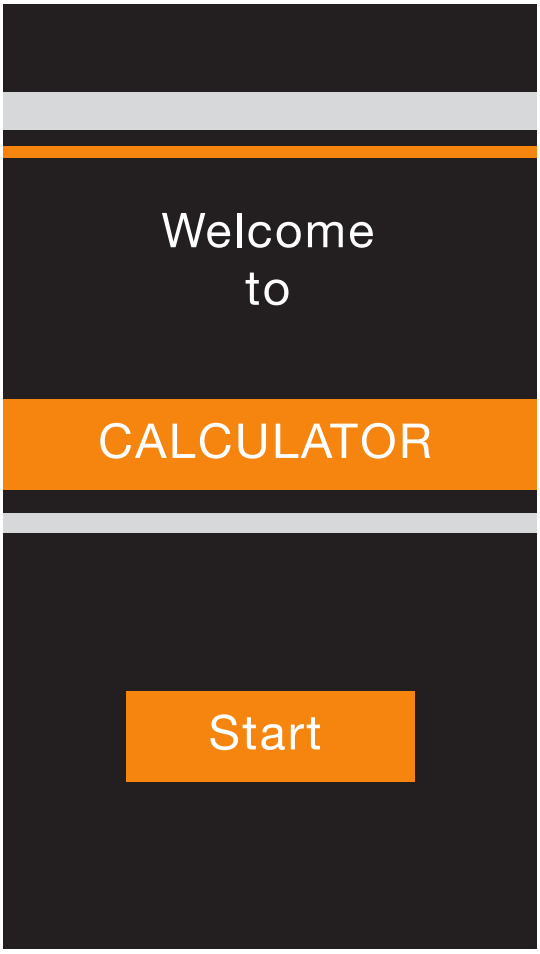
> **Important:** Don't tell people to reboot or restart their devices after installing your app. Restarting takes time and can make your app seem unreliable and hard to use.
>
> If your app has memory-usage or other issues that make it difficult to run unless the system has just booted, you need to address those issues. For some guidance on developing a well-tuned app, see "Using Memory Efficiently" in *iOS App Programming Guide*.

As much as possible, avoid displaying a splash screen or other startup experience. It's best when users can begin using your app immediately.

Recommended



Not Recommended



Avoid asking people to supply setup information. Instead:

- **Focus your solution on the needs of 80 percent of your users.** When you do this, most people won't have to supply any settings, because your app is already set up to behave the way they expect. If there is functionality that only a few users might want—or that most users might want only once—leave it out.
- **Get as much information as possible from other sources.** If you can use any of the information people supply in built-in app or device settings, query the system for these values; don't ask people to enter them again.
- **If you must ask for setup information, prompt people to enter it within your app.** Then, store this information as soon as possible (potentially, in your app's settings). This way, people aren't forced to switch to Settings before they get the chance to enjoy your app. If people need to make changes to this information later, they can go to your app's settings at any time.

Delay a login requirement for as long as possible. Ideally, users can navigate through much of your app and use some of its functionality without logging in. When you ask users to log in before they can begin using your app, it can make the startup process seem burdensome.

In general, launch in the device's default orientation. On iPhone, the default orientation is portrait; on iPad, it's the current device orientation. If your app runs *only* in landscape orientation, you should always launch in landscape and let users rotate the device if necessary.

It's best when a landscape-only app supports both variants of landscape orientation—that is, with the Home button on the right or on the left. If the device is already in landscape, a landscape-only app should launch in that variant, unless there's a very good reason not to. Otherwise, launch a landscape-only app in the variant with the Home button on the right.

Supply a launch image that closely resembles the first screen of the app. iOS displays the launch image the moment your app starts—giving users the impression that your app is fast and giving your app enough time to load content. Learn how to create a launch image in "Launch Images" (page 196).

If possible, avoid requiring users to read a disclaimer or agree to an end-user license agreement when they first start your app. Instead, you can let the App Store display your disclaimer or end-user license agreement (EULA) so that people can access it before they get your app. Although reading a disclaimer or agreeing to a EULA in the App Store causes the least inconvenience to users, it might not be feasible in all cases. If you must provide these items within your app, be sure to integrate them in a way that harmonizes with your UI and balances business requirements with user experience needs.

When your app restarts, restore its state so that users can continue where they left off. People shouldn't have to remember the steps they took to reach their previous location in your app. To learn more about efficient ways to preserve and restore your app's state, see "State Preservation and Restoration".

Always Be Prepared to Stop

An iOS app never displays a Close or Quit option. People stop using an app when they switch to a different app, return to the Home screen, or put their devices in sleep mode.

When people switch away from your app, iOS multitasking transitions it to the background and replaces its UI with the UI of the new app. To prepare for this situation, your app should:

- **Save user data as soon as possible and as often as reasonable** because an app in the background can be told to exit or terminate at any time.

- **Save the current state when stopping** at the finest level of detail possible so that people don't lose their context when they switch back to your app. For example, if your app displays scrolling data, save the current scroll position. You can learn more about efficient ways to preserve and restore your app's state in "State Preservation and Restoration".

Some apps might need to keep running in the background while users run another app in the foreground. For example, users might want to keep listening to the song that's playing in one app while they're using a different app to check their to-do list or play a game. Learn how to handle multitasking correctly and gracefully in "Multitasking" (page 88).

Never quit an iOS app programmatically because people tend to interpret this as a crash. If something prevents your app from functioning as intended, you need to tell users about the situation and explain what they can do about it. Depending on how severe the problem is, you have two choices.

- **Display an attractive screen that describes the problem and suggests a correction.** A screen gives feedback to users and reassures them that there's nothing wrong with your app. It also puts users in control, letting them decide whether they want to take corrective action and continue using your app or switch to a different app.
- **If only some app features are unavailable, display either a screen or an alert when people use the feature.** Display the alert *only* when people try to access the feature that isn't functioning.

Layout

Layout concerns much more than how UI elements look on an app screen. With your layout, you show users what's most important, what their choices are, and how content is related. Depending on the device your app is running on—and on the device's current orientation—your layout may vary.

Make it easy for people to interact with content and controls by giving each interactive element ample spacing. Give tappable controls a hit target of about 44 x 44 points.

Make it easy to focus on the main task by elevating important content or functionality. Some good ways to do this are to place principal items in the upper half of the screen and—in left-to-right cultures—near the left side of the screen.

Use visual weight and balance to show users the relative importance of onscreen elements. Large items—and those that look heavier in weight—catch the eye and tend to appear more important than smaller ones.

In general, avoid inconsistent appearances in your UI. As much as possible, elements that have similar functions should also look similar. People often assume that there must be a reason for the inconsistencies they notice, and they're apt to spend time trying to figure it out.

Make sure that users can understand primary content at its default size. For example, users shouldn't have to scroll horizontally to read text or zoom to see primary images.

Navigation

People tend to be unaware of the navigation experience in an app unless it doesn't meet their expectations. Your job is to implement navigation in a way that supports the structure and purpose of your app without calling attention to itself.

Broadly speaking, there are three main styles of navigation, each of which is well suited to a specific app structure: hierarchical, flat, and content- or experience-driven.

In some cases, it works well to combine more than one navigation style in an app. For example, the items in one category of a flat information structure might best be displayed in a hierarchy.

Regardless of the navigation style that suits the structure of your app, the most important thing is that the user's path through the content is logical, predictable, and easy to follow. Users should always know where they are in your app and how to get to their next destination.

UIKit defines some standard UI elements that make it easy to implement hierarchical and flat navigation styles, in addition to some elements that help you enable content-centric navigation, such as in a book-style or media-viewing app. An app that provides an experience-driven navigation style—such as a game—typically relies on custom elements and behaviors.

Use a navigation bar to give users an easy way to traverse a hierarchy of data. The navigation bar's title shows users their current position in the hierarchy; the Back button makes it easy to return to the previous level. To learn more, see "Navigation Bar" (page 120).

Use a tab bar to display several peer categories of content or functionality. A tab bar is a good way to support a flat information architecture because it lets people switch between categories regardless of their current location. To learn more, see "Tab Bar" (page 124).

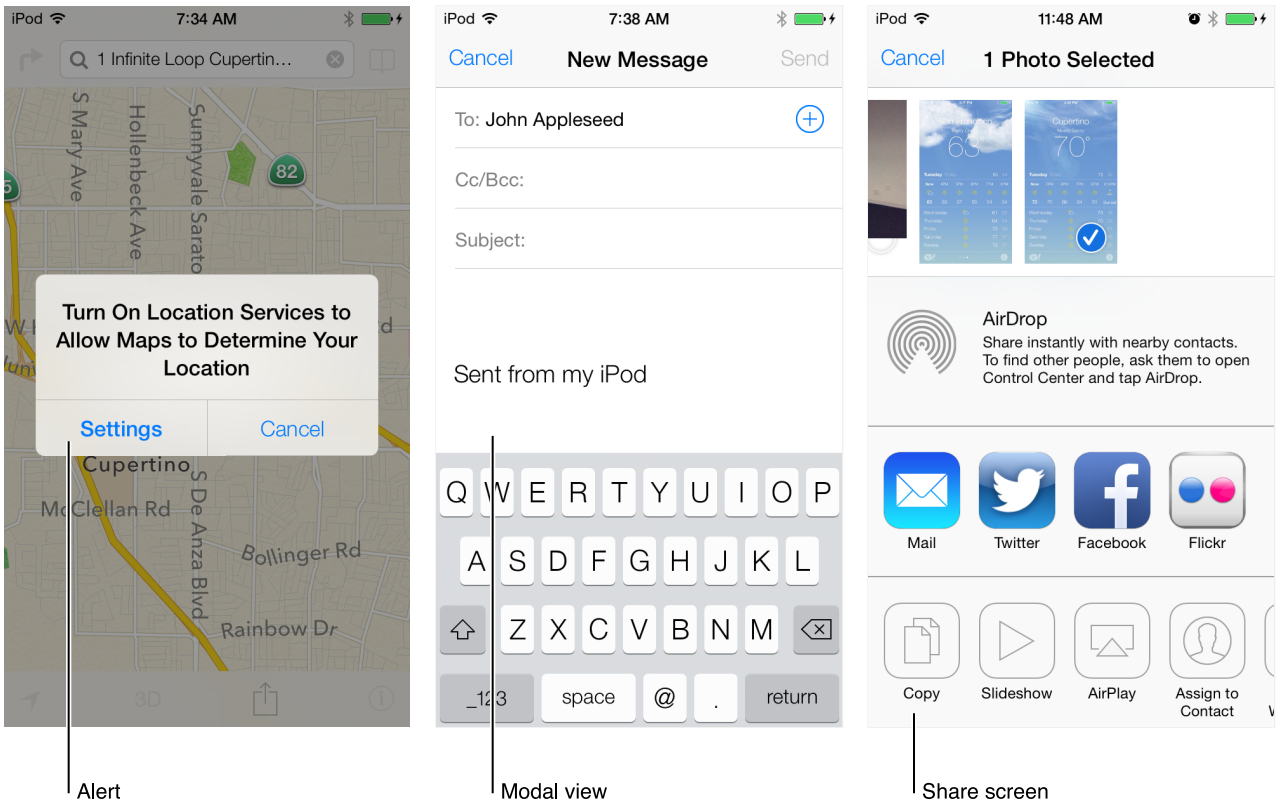
Use a page control to indicate that there are multiple peer items or content screens. A page control is good for showing users how many pages are available and which one is currently open. To learn more, see "Page Control" (page 169).

> **Note:** Although a toolbar looks similar to a navigation bar or a tab bar, it doesn't implement navigation. Instead, a toolbar gives users controls that act on the contents of the current screen. To learn more, see "Toolbar" (page 122).

In general, it's best to give users one path to each screen. If there's one screen that users need to see in more than one context, consider using a temporary view, such as a modal view, action sheet, or alert. To learn more, see "Modal View" (page 185), "Action Sheet" (page 183), and "Alert" (page 179).

Modal Contexts

Modality—that is, a mode in which something exists or is experienced—has advantages and disadvantages. It can give users a way to complete a task or get information without distractions, but it does so by temporarily preventing them from interacting with the rest of the app.



Ideally, people can interact with iOS apps in nonlinear ways, so it's best when you can minimize the number of modal experiences in your app. In general, consider creating a modal context only when:

- It's critical to get the user's attention
- A task must be completed—or explicitly abandoned—to avoid leaving the user's data in an ambiguous state

Keep modal tasks simple, short, and narrowly focused. You don't want your users to experience a modal view as a mini app within your app. If a subtask is too complex, people can lose sight of the main task they suspended when they entered the modal context. Be especially wary of creating a modal task that involves a

hierarchy of views, because people can get lost and forget how to retrace their steps. If a modal task must contain subtasks in separate views, be sure to give users a single, clear path through the hierarchy, and avoid circularities. For guidelines on modal views, see "Modal View" (page 185).

Always provide an obvious and safe way to exit a modal task. People should always be able to predict the fate of their work when they dismiss a modal view.

If the task requires a hierarchy of modal views, make sure your users understand what happens if they tap a Done button in a view that's below the top level. Examine the task to decide whether a Done button in a lower-level view should complete only the part of the task in that view or the entire task. Because of this potential for confusion, avoid adding a Done button to a subordinate view as much as possible.

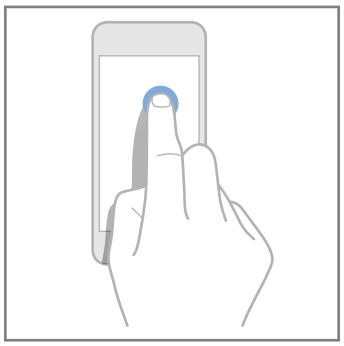
Reserve alerts for delivering essential—and ideally actionable—information. An alert interrupts the user's experience and requires a tap to dismiss, so it's important for users to feel that the alert's message warrants the intrusion. To learn more, see "Alert" (page 179).

Respect the user's preferences for receiving notifications. In Settings, users indicate how they want to receive notifications from your app. Be sure to abide by these preferences so that users aren't tempted to turn off all notifications from your app.

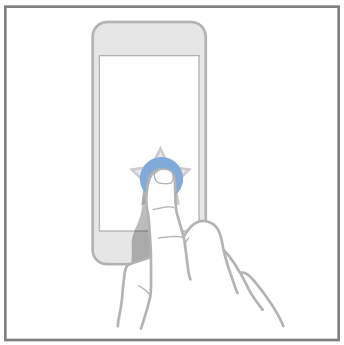
Interactivity and Feedback

Users Are Comfortable with the Standard Gestures

People use gestures—such as tap, drag, and pinch—to interact with apps and their iOS devices. Using gestures gives people a close personal connection to their devices and enhances their sense of direct manipulation of onscreen objects. People generally expect gestures to work the same in all the apps they use.

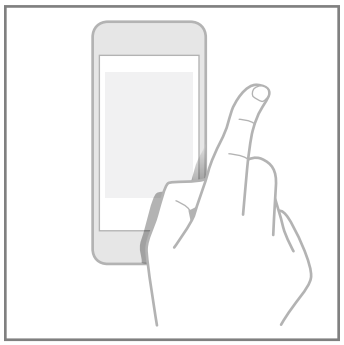


Tap To press or select a control or item.

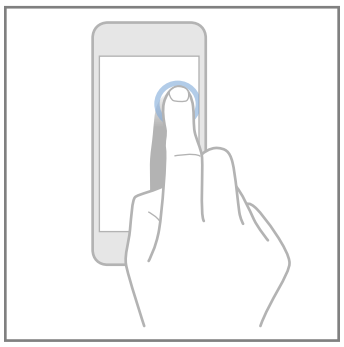


Drag To scroll or pan—that is, move side to side.

To drag an element.

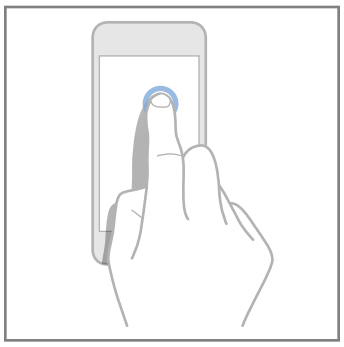


Flick To scroll or pan quickly.



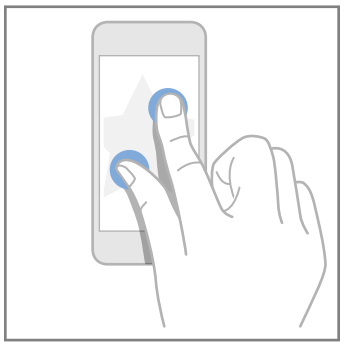
Swipe With one finger, to reveal the Delete button in a table-view row, the hidden view in a split view (iPad only), or Notification Center (from the top edge of the screen).

With four fingers, to switch between apps on iPad.

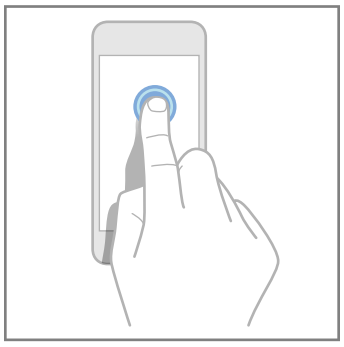


Double tap To zoom in and center a block of content or an image.

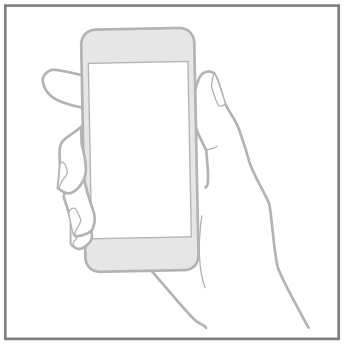
To zoom out (if already zoomed in).



Pinch Pinch open to zoom in; pinch close to zoom out.



Touch and hold In editable or selectable text, to display a magnified view for cursor positioning.



Shake To initiate an undo or redo action.

In addition to the standard gestures users know, iOS defines a few gestures that invoke systemwide actions, such as revealing Control Center or Notification Center. Users rely on these gestures to work regardless of the app they're using.

Avoid associating different actions with the standard gestures. Unless your app is a game, redefining the meaning of a standard gesture may confuse people and make your app harder to use.

Avoid creating custom gestures that invoke the same actions as the standard gestures. People are used to the behavior of the standard gestures and they don't appreciate being expected to learn different ways to do the same thing.

Use complex gestures as shortcuts to expedite a task, not as the only way to perform it. As much as possible, always give users a simple, straightforward way to perform an action, even if it means an extra tap or two. Simple gestures let users focus on the experience and the content, not the interaction.

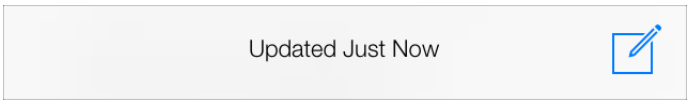
Avoid defining new gestures unless your app is a game. In games and other immersive apps, custom gestures can be a fun part of the experience. But in apps that help people do things that are important to them, it's best to use standard gestures because people don't have to think about them.

For iPad, consider using multifinger gestures. The large iPad screen provides great scope for custom multifinger gestures, including gestures made by more than one person. Although complex gestures aren't appropriate for every app, they can enrich the experience in apps that people spend a lot of time in, such as games or content-creation environments. Always bear in mind that nonstandard gestures aren't discoverable and should rarely, if ever, be the only way to perform an action.

Feedback Aids Understanding

iOS users are accustomed to getting feedback that helps them know what an app is doing, discover what they can do next, and understand the results of their actions. UIKit controls and views provide many kinds of feedback.

As much as possible, integrate status or other relevant feedback information into your UI. For example, Mail displays the update status in the toolbar.

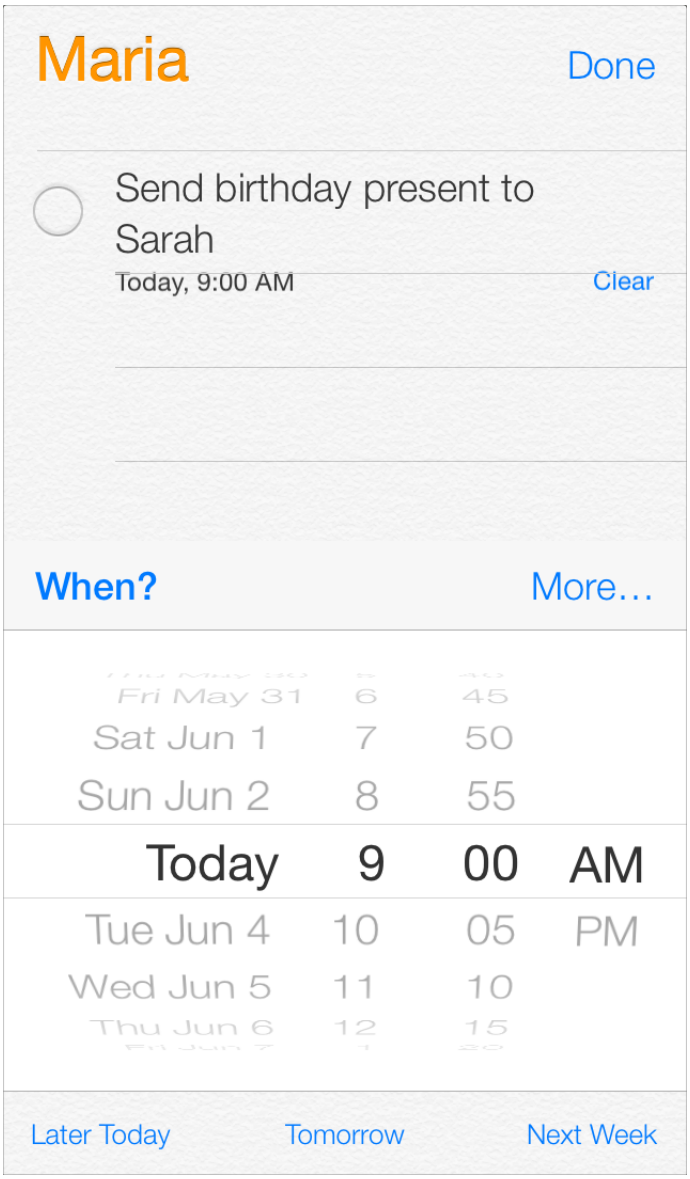


Avoid unnecessary alerts. An alert is a powerful feedback mechanism, but it should be used only to deliver important—and ideally actionable—information. If users see too many alerts that don't contain essential information, they quickly learn to ignore all alerts. To learn more about alerts, see "Alert" (page 179).

Inputting Information Should Be Simple and Easy

Inputting information takes time and attention, whether people tap controls or use the keyboard. And if your app slows people down by asking for a lot of user input before anything useful happens, people may feel discouraged from using it.

Make it easy for users to make choices. For example, you can use a table view or a picker instead of a text field, because most people find it easier to select an item than to type words.



Get information from iOS, when appropriate. People store lots of information on their devices. When it makes sense, don't force people to give you information you can easily find for yourself, such as their contacts or calendar information.

Balance a request for input by giving users something useful in return. A sense of give and take helps people feel they're making progress as they move through your app.

Terminology and Wording

Every word you display in an app is part of a conversation you have with users. Use this conversation as an opportunity to help people feel comfortable in your app.

Use terminology you're sure users understand. Use what you know about your users to determine whether the words and phrases you plan to use are appropriate. For example, technical jargon is rarely helpful in an app aimed at unsophisticated users, but in an app designed for technically savvy users, it might be appreciated.

Use a tone that's informal and friendly, but not too familiar. You want to avoid being stilted or too formal, but you don't want to risk sounding falsely jovial or patronizing. Remember that users are likely to read the text in your UI many times, and what might seem clever at first can become irritating when repeated.

Think like a newspaper editor, and watch out for redundant or unnecessary words. When your UI text is short and direct, users can absorb it quickly and easily. Identify the most important information, express it concisely, and display it prominently so that people don't have to read too many words to find what they're looking for or to figure out what to do next.

Give controls short labels or use well-understood icons, so people can tell what they do at a glance.

Take care to be accurate when describing dates. It's often appropriate to use friendly terms such as "today" and "tomorrow" when you display date information in your UI, but it can be confusing if you don't account for the user's current locale. For example, consider an event that starts just before midnight. To users in the same time zone, the event starts today, but to users in an earlier time zone, the same event may have started yesterday.

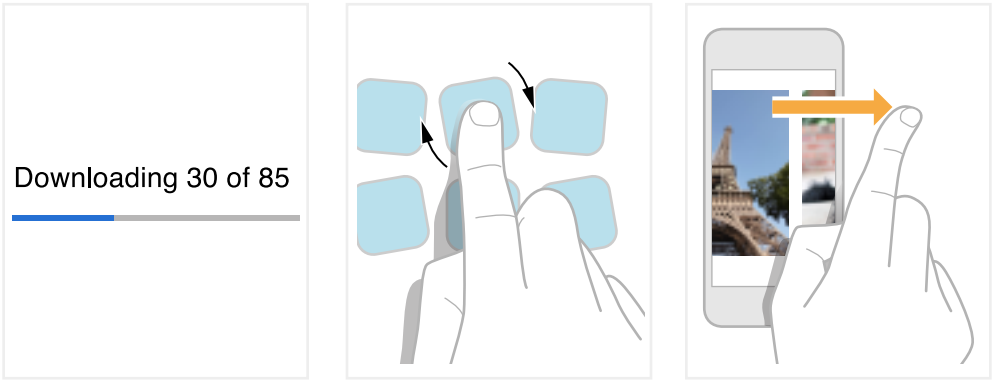
Make the most of the opportunity to communicate with potential users by writing a great App Store description. In addition to describing your app accurately and highlighting the qualities you think people are most likely to appreciate, be sure to:

- **Correct all spelling, grammatical, and punctuation errors.** Although such errors don't bother everyone, in some people they can create a negative impression of your app's quality.
- **Keep all-capital words to a minimum.** Occasional all-capital words help draw people's attention, but when an entire passage is capitalized, it's difficult to read and it can be interpreted as shouting.
- **Consider describing specific bug fixes.** If a new version of your app contains bug fixes that customers have been waiting for, it can be a good idea to mention this in your description.

Animation

Beautiful, subtle animation pervades the iOS UI and makes the app experience more engaging and dynamic. Subtle and appropriate animation can:

- Communicate status
- Enhance the sense of direct manipulation
- Help people visualize the results of their actions



Add animation cautiously, especially in apps that don't provide an immersive experience. In apps that are focused on serious or productive tasks, animation that seems excessive or gratuitous can obstruct app flow, decrease performance, and distract users from their task.

Make animation consistent with built-in apps when appropriate. People are accustomed to the subtle animation used in the built-in iOS apps. In fact, people tend to regard the smooth transitions between views, the fluid response to changes in device orientation, and the physics-based scrolling as an expected part of the iOS experience. Unless you're creating an app that enables an immersive experience—such as a game—custom animation should be comparable to the built-in animations.

Use animation consistently throughout your app. As with other types of customization, it's important to use custom animation consistently so that users can build on the experience they gain as they use your app.

In most cases, it's appropriate to strive for realism in custom animation. People tend to be willing to accept artistic license in appearance, but they can feel disoriented when they experience movement that appears to defy physical laws.

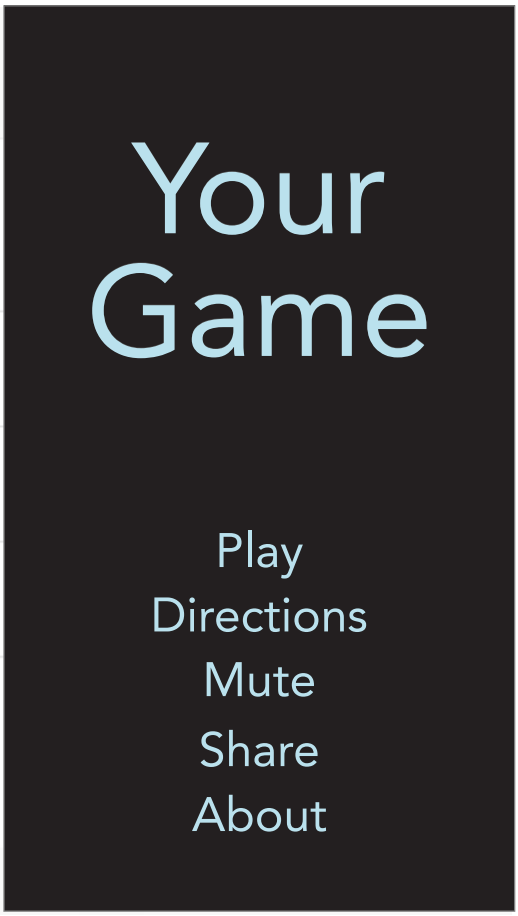
Typography and Color

Typography Should Always Be Legible

Above all, typography must be legible. If users can't read the words in your app, it doesn't matter how beautiful the text looks. When you adopt Dynamic Type in an iOS 7 app, you get:

- Automatic adjustments to weight, letter spacing, and line height for every font size.
- The ability to specify different text styles for semantically distinct blocks of text, such as `Body`, `Footnote`, or `Headline1`.
- Text that responds appropriately to changes in both the Dynamic Type and accessibility settings for user-specified text sizes.

In general, use a single font throughout your app. Mixing several different fonts can make your app seem fragmented and sloppy. Instead, use one font and just a few styles. Use the `UIFont` text styles API to define different areas of text according to semantic usage, such as body or headline.



Color Can Enhance Communication

Consider defining a key color. The built-in apps use key colors—such as yellow in Notes—to indicate interactivity and element state.

Color communicates, but not always in the way you intend. Everyone sees color differently and many cultures differ in how they assign meanings to colors. Spend the time to research how your use of color might be perceived in other countries and cultures. As much as possible, you want to be sure that the colors in your app send the appropriate message.

In most cases, don't let color distract users. Unless color is essential to your app's purpose, it usually works well to use color as a subtle enhancement.

Icons and Graphics

The App Icon

Every app needs a beautiful app icon. It's not unusual for people to base their initial opinions about your app's quality, purpose, and reliability solely on the look of your app icon.

Here are a few of the things you should keep in mind as you think about your app icon. When you're ready to start creating it, see "App Icon" (page 192) for detailed guidance and specifications.

- Your app icon is an important part of your app's brand. Approach the design as an opportunity to tell your app's story and build an emotional connection with users.
- The best app icons are unique, uncluttered, and engaging.
- An app icon needs to look good at many different sizes and on different backgrounds. Details that might enrich an icon at large sizes can make it look muddy at small sizes.

Other Icons

iOS provides a lot of small icons that represent common tasks and types of content in tab bars, toolbars, and navigation bars. It's a good idea to use the built-in icons as much as possible because users already know what they mean.

You can create custom bar icons if you need to represent custom actions or content types. Designing these small, streamlined icons is very different from designing your app icon. If you need to create custom bar icons, see "Bar Button Icons" (page 198) to learn how.

Graphics

iOS apps tend to be graphically rich. Whether you're displaying the user's photos or supplying custom artwork, here are a few guidelines you should follow.

Support the Retina display. Make sure that you supply @2x assets for all artwork and graphics in your app.

Display photos and graphics in their original aspect ratio and don't scale them greater than 100%. You don't want the artwork or graphics in your app to look skewed or blown up. Let users choose whether they want to zoom images in or out.

Don't use images that replicate Apple products in your designs. These symbols are copyrighted and product designs can change frequently.

Branding

Branding isn't just about displaying a brand's colors or logo in an app. Ideally, you develop an app-specific brand by creating a distinctive look and feel that gives users a memorable experience.

When you do need to remind users of an existing brand within your app, follow these guidelines.

Incorporate a brand's colors or images in a refined, unobtrusive way. People use your app to get things done or to be entertained; they don't want to feel as if they're being forced to watch an advertisement. For the best user experience, you want to quietly remind users of your identity. iOS 7 makes it easy to define a key color that indicates interactivity and selection state throughout your app.

Avoid taking space away from the content people care about. For example, displaying a second, persistent bar at the top of the screen that does nothing but display branding assets means that there's less room for content. Consider other, less intrusive ways to display pervasive branding, such as subtly customizing the background of a screen.

Important: The app icon is an exception to these guidelines, because it should be completely focused on your brand. Because users see your app icon frequently, it's important to spend time designing an icon that balances eye appeal with brand recognition.

Integrating with iOS

Use Standard UI Elements

As much as possible, it's a good idea to use the standard UI elements that UIKit provides. When you use standard elements instead of creating custom ones, both you and your users benefit:

- Standard UI elements automatically receive updates if iOS introduces a redesigned appearance—custom elements don't get updated.
- People are comfortable with the standard UI elements so that they instantly understand how to use them in your app.

To take advantage of the benefits of using standard UI elements, it's crucial that you:

Follow the guidelines for every UI element. When a UI element looks and works they way people expect it to, they can depend on their prior experience to help them use it in your app. You can find UI element guidelines in "Bars" (page 119), "Content Views" (page 130), "Controls" (page 164), and "Temporary Views" (page 179).

In general, avoid creating a custom UI element that performs a standard action. First, ask yourself why you're creating a custom UI element that behaves exactly like a standard one. If you just want a custom look, consider changing the look of a standard element by using the UIKit appearance customization APIs or tint color. If you want a slightly different behavior, be sure to find out whether a standard element might do what you want when you adjust its properties and attributes. If you need completely custom behavior, it's best to design a custom element that doesn't look too similar to the standard ones.

Tip: Interface Builder makes it easy to get the standard UI elements, use the appearance customization APIs, access properties and attributes, and apply custom and system-provided icons to your controls. To learn more about Interface Builder, see *Xcode User Guide*.

Don't use system-defined buttons and icons to mean something else. iOS provides several buttons and many icons that you can use in your app. Be sure you understand the documented, semantic meaning of these buttons and icons; don't rely on your interpretation of their appearance.

If you can't find a system-provided button or icon that has the appropriate meaning for a function in your app, you can create your own. For some guidelines to help you design custom icons, see "Bar Button Icons" (page 198).

If your app enables an immersive task or experience, it may be reasonable to create completely custom controls. This is because you're creating a unique environment, and discovering how to control that environment is an experience users expect in such apps.

Respond to Changes in Device Orientation

People generally expect to use their iOS devices in any orientation, so it's best when your app responds appropriately.

Maintain focus on the primary content in all orientations. This is your highest priority. People use your app to view and interact with the content they care about. Changing the focus when the device rotates can disorient people and make them feel that they've lost control over the app.

In general, run in all orientations. People expect to use your app in different orientations, and it's best when you can fulfill that expectation. iPad users, in particular, expect to use your app in whichever orientation they're currently holding their device. But some apps need to run in portrait only or in landscape only. If it's essential that your app run in only one orientation, you should:

- **Launch your app in the supported orientation, regardless of the current device orientation.** For example, if a game or media-viewing app runs in landscape only, it's appropriate to launch the app in landscape, even if the device is currently in portrait. This way, if people start the app in portrait, they know to rotate the device to landscape to view the content.
- **Avoid displaying a UI element that tells people to rotate the device.** Running in the supported orientation clearly tells people to rotate the device, if required, without adding unnecessary clutter to the UI.
- **Support both variants of an orientation.** For example, if an app runs only in landscape, people should be able to use it whether they're holding the device with the Home button on the right or on the left. And if people rotate the device 180 degrees while using the app, it's best if the app responds by rotating its content 180 degrees.

If your app interprets changes in device orientation as user input, you can handle rotation in app-specific ways. For example, a game that lets people move game pieces by rotating the device can't respond to device rotation by rotating the screen. In a case like this, you should launch in both variants of the required orientation and allow people to switch between the variants until they start the main task of the app. Then, as soon as people begin the main task, you can begin responding to device movement in app-specific ways.

On iPhone, anticipate users' needs when you respond to a change in device orientation. Users often rotate their devices to landscape orientation because they want to "see more." If you respond by merely scaling up the content, you fail to meet users' expectations. Instead, respond by rewrapping lines of text and—if necessary—rearranging the layout of the UI so that more content fits on the screen.

On iPad, strive to satisfy users' expectations by supporting all orientations. The iPad screen mitigates people's desire to rotate the device to landscape to "see more." And because people don't pay much attention to the minimal frame of the device or the location of the Home button, they don't view the device as having a default orientation. As much as possible, your app should encourage people to interact with iPad from any side by providing a great experience in all orientations.

Follow these guidelines as you design how your iPad app should handle rotation:

- **Consider changing how you display auxiliary information or functionality.** Although you should make sure that the most important content is always in focus, you can respond to rotation by changing the way you provide secondary content.

  In Mail on iPad, for example, the lists of accounts and mailboxes are secondary content (the main content is the selected message). In landscape, secondary content is displayed in the left pane of a split view; in portrait, it's displayed in a popover.

  Or consider an iPad game that displays a rectangular game board in landscape. In portrait, the game needs to redraw the board to fit well on the screen, which might result in additional space above or below the board. Instead of vertically stretching the game board to fit the space or leaving the space empty, the game could display supplemental information or objects in the additional space.

- **Avoid gratuitous changes in layout.** As much as possible, provide a consistent experience in all orientations. A comparable experience in all orientations allows people to maintain their usage patterns when they rotate the device. For example, if your iPad app displays images in a grid while in landscape, it's not necessary to display the same information in a list while in portrait (although you might adjust the dimensions of the grid).

- **When possible, avoid reformatting information and rewrapping text on rotation.** Strive to maintain a similar format in all orientations. Especially if people are reading text, it's important to avoid causing them to lose their place when they rotate the device.

  If some reformatting is unavoidable, use animation to help people track the changes. For example, if you must add or remove a column of text in different orientations, you might choose to hide the movement of columns and simply fade in the new arrangement. To help you design appropriate rotation behavior, think about how you'd expect your content to behave if you were physically interacting with it in the real world.

- **Provide a unique launch image for each orientation.** When each orientation has a unique launch image, people experience a smooth app start regardless of the current device orientation. In contrast with the Home screen on iPhone, the iPad Home screen supports all orientations, and so people are likely to start your app in the same orientation in which they quit the previous app.

Downplay File and Document Handling

iOS apps can help people create and manipulate files, but this doesn't mean that people should have to think about the file system on an iOS device.

There is no iOS app analogous to the OS X Finder, and people shouldn't be asked to interact with files as they do on a computer. In particular, people shouldn't be faced with anything that encourages them to think about file metadata or locations, such as:

- An open or save dialog that exposes a file hierarchy
- Information about the permissions status of files

As much as possible, allow people to manage documents without opening iTunes on their computer. Consider using iCloud to help users access their content on all of their devices. For some tips on how to provide a great iCloud experience in your app, see "iCloud" (page 81).

If your app helps people create and edit documents, it's appropriate to provide some sort of document picker that lets them to open an existing document or create a new one. Ideally, such a document picker:

- **Is highly graphical.** People should be able to easily identify the document they want by looking at visual representations of the documents onscreen.
- **Lets people make the fewest possible gestures to do what they want.** For example, people might scroll horizontally through a carousel or grid of existing documents and open the desired one with a tap.
- **Includes a new document function.** Instead of making people go somewhere else to create a new document, a document picker can allow them to tap a placeholder image to create a new document.

Tip: You can use the Quick Look Preview feature to allow people to preview documents within your app, even if your app can't open them. To learn how to provide this feature in your app, see "Quick Look" (page 99).

Give people confidence that their work is always preserved unless they explicitly cancel or delete it. If your app helps people create and edit documents, make sure that they don't have to take an explicit save action. iOS apps should take responsibility for saving people's input, both periodically and when they open a different document or switch away from the app.

If the main function of your app isn't content creation—but you allow people to switch between viewing information and editing it—it can make sense to ask them to save their changes. In this scenario, it often works well to provide an Edit button in the view that displays the information. When people tap the Edit button, you can replace it with a Save button and add a Cancel button. The transformation of the Edit button helps remind people that they're in an editing mode and might need to save changes, and the Cancel button gives them the opportunity to exit without saving their changes.

Be Configurable If Necessary

Some apps might need to give users a way to make setup or configuration choices, but most apps can avoid or delay doing this. Successful apps work well for most people right away and offer some ways to adjust the user experience within the main UI.

If possible, avoid sending users to Settings. It's important to remember that users can't open the Settings app without first switching away from your app, and you don't want to encourage this action.

When you design your app to function the way most of your users expect, you decrease the need for settings. If you need information about the user, query the system for it instead of asking users to provide it. If you decide you must provide settings in your iOS app, see "The Settings Bundle" in *iOS App Programming Guide* to learn how to support them in your code.

If necessary, let users set behaviors they want within your app. Integrating configuration options into your app lets you react dynamically to changes because people don't have to leave your app to set them.

As much as possible, offer configuration options in the main UI. Putting options in the main UI can make sense if they represent a primary task and if people might want to change them frequently. If people are likely to change an app's configuration only occasionally, it makes sense to put them in a separate view.

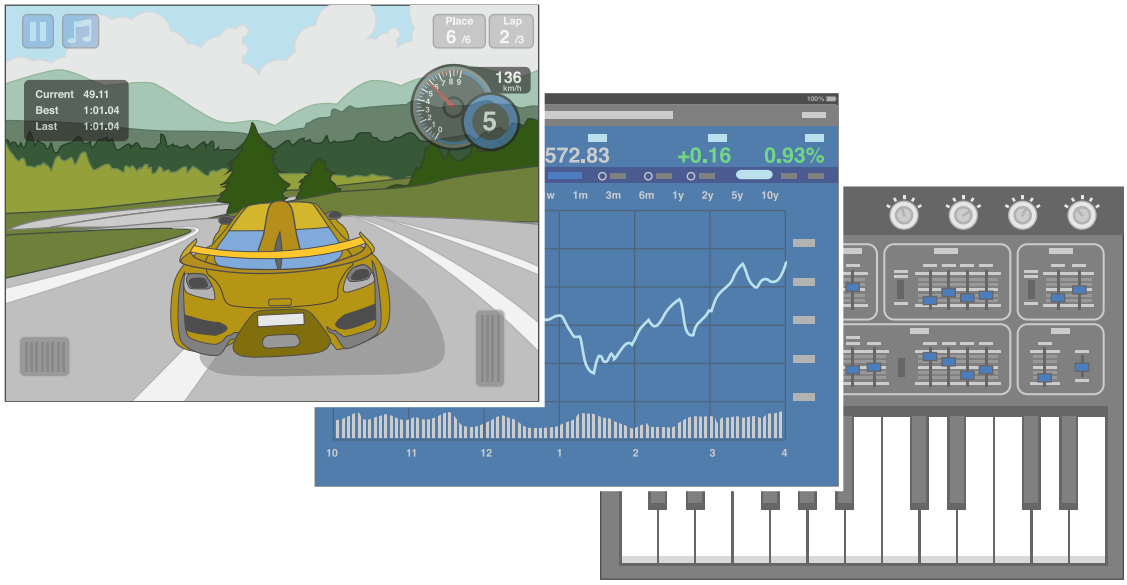
Design Strategies

Design Principles

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

Aesthetic Integrity

Aesthetic integrity doesn't measure the beauty of an app's artwork or characterize its style; rather, it represents how well an app's appearance and behavior integrates with its function to send a coherent message.



People care about whether an app delivers the functionality it promises, but they're also affected by the app's appearance and behavior in strong—sometimes subliminal—ways. For example, an app that helps people perform a serious task can put the focus on the task by keeping decorative elements subtle and unobtrusive

and by using standard controls and predictable behaviors. This app sends a clear, unified message about its purpose and its identity that helps people trust it. But if the app sends mixed signals by presenting the task in a UI that's intrusive, frivolous, or arbitrary, people might question the app's reliability or trustworthiness.

On the other hand, in an app that encourages an immersive task—such as a game—users expect a captivating appearance that promises fun and excitement and encourages discovery. People don't expect to accomplish a serious or productive task in a game, but they expect the game's appearance and behavior to integrate with its purpose.

Consistency

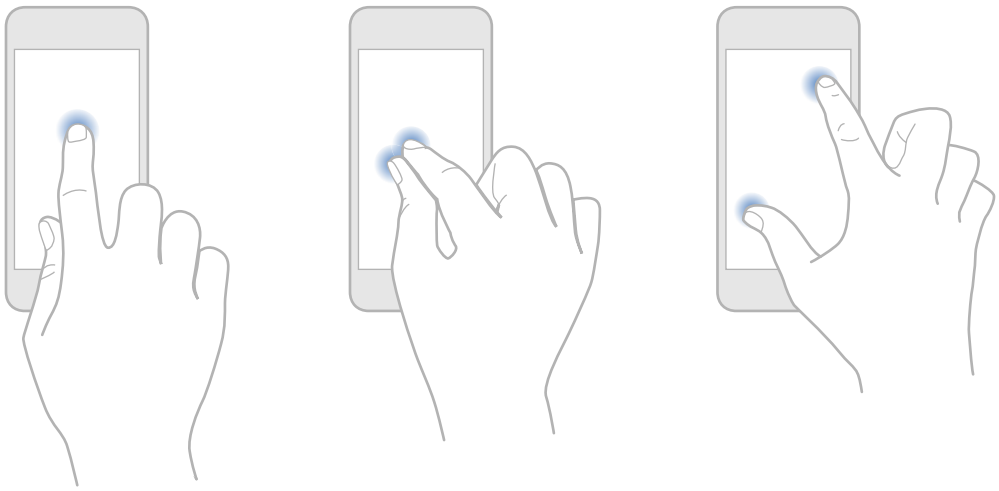
Consistency lets people transfer their knowledge and skills from one part of an app's UI to another and from one app to another app. A consistent app isn't a slavish copy of other apps and it isn't stylistically stagnant; rather, it pays attention to the standards and paradigms people are comfortable with.

To determine whether an iOS app follows the principle of consistency, think about these questions:

- Is the app consistent with iOS standards? Does it use system-provided controls, views, and icons correctly? Does it incorporate device features in a reliable way?
- Is the app consistent within itself? Does text use uniform terminology and style? Do the same icons always mean the same thing? Can people predict what will happen when they perform the same action in different places? Do custom UI elements look and behave the same throughout the app?
- Within reason, is the app consistent with its earlier versions? Have the terms and meanings remained the same? Are the fundamental concepts and primary functionality essentially unchanged?

Direct Manipulation

When people directly manipulate onscreen objects instead of using separate controls to manipulate them, they're more engaged with their task and it's easier for them to understand the results of their actions.



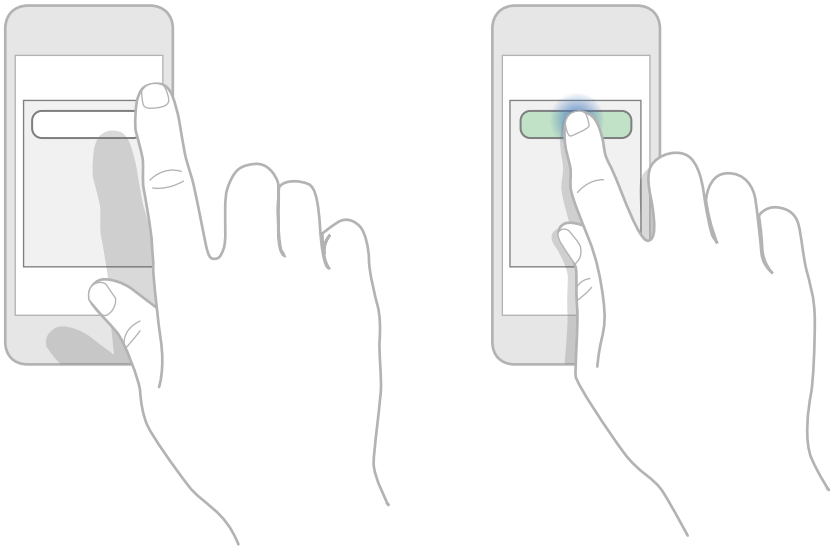
Using the Multi-Touch interface, people can pinch to directly expand or contract an image or content area. And in a game, players move and interact directly with onscreen objects—for example, a game might display a combination lock that users can spin to open.

In an iOS app, people experience direct manipulation when they:

- Rotate or otherwise move the device to affect onscreen objects
- Use gestures to manipulate onscreen objects
- Can see that their actions have immediate, visible results

Feedback

Feedback acknowledges people's actions, shows them the results, and updates them on the progress of their task.



The built-in iOS apps provide perceptible feedback in response to every user action. List items and controls highlight briefly when people tap them and—during operations that last more than a few seconds—a control shows elapsing progress.

Subtle animation can give people meaningful feedback that helps clarify the results of their actions. For example, lists can animate the addition of a new row to help people track the change visually.

Sound can also give people useful feedback, but it shouldn't be the only feedback mechanism because people can't always hear their devices.

Metaphors

When virtual objects and actions in an app are metaphors for familiar experiences—whether these experiences are rooted in the real world or the digital world—users quickly grasp how to use the app.

It's best when an app uses a metaphor to suggest a usage or experience without letting the metaphor enforce the limitations of the object or action on which it's based.

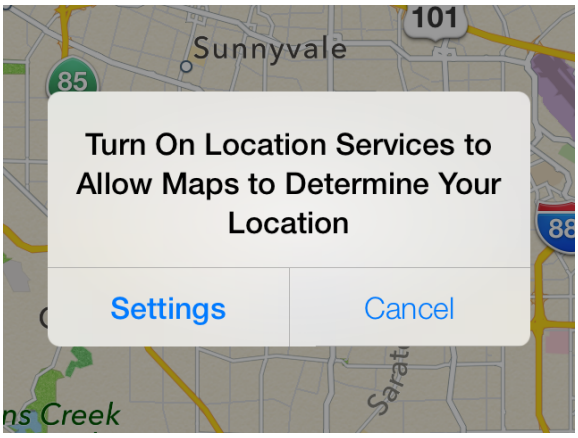
iOS apps have great scope for metaphors because people physically interact with the screen. Metaphors in iOS include:

- Moving layered views to expose content beneath them

- Dragging, flicking, or swiping objects in a game
- Tapping switches, sliding sliders, and spinning pickers
- Flicking through pages of a book or magazine

User Control

People—not apps—should initiate and control actions. An app can suggest a course of action or warn about dangerous consequences, but it's usually a mistake for the app to take decision-making away from the user. The best apps find the correct balance between giving people the capabilities they need while helping them avoid unwanted outcomes.



Users feel more in control of an app when behaviors and controls are familiar and predictable. And when actions are simple and straightforward, users can easily understand and remember them.

People expect to have ample opportunity to cancel an operation before it begins, and they expect to get a chance to confirm their intention to perform a potentially destructive action. Finally, people expect to be able to gracefully stop an operation that's underway.

From Concept to Product

Define Your App

An **app definition statement** is a concise, concrete declaration of an app's main purpose and its intended audience.

Create an app definition statement early in your development effort to help you turn an idea and a list of features into a coherent product that people want to own. Throughout development, use the definition statement to decide if potential features and behaviors make sense. Take the following steps to create a robust app definition statement.

1. List All the Features You Think Users Might Like

Go ahead and brainstorm here. At this point, you're trying to capture all the tasks related to your main product idea. Don't worry if your list is long; you'll narrow it down later.

Imagine that your initial idea is to develop an app that helps people shop for groceries. As you think about this activity, you come up with a list of related tasks—that is, potential features—that users might be interested in. For example:

- Creating lists
- Getting recipes
- Comparing prices
- Locating stores
- Annotating recipes
- Getting and using coupons
- Viewing cooking demos
- Exploring different cuisines
- Finding ingredient substitutions

2. Determine Who Your Users Are

Now you need to figure out what distinguishes your app's users from all other iOS users. In the context of your main idea, what's most important to them? Using the grocery-shopping example, you might ask whether your users:

- Usually cook at home or prefer ready-made meals
- Are committed coupon-users or think that coupons aren't worth the effort
- Enjoy hunting for speciality ingredients or seldom venture beyond the basics
- Follow recipes strictly or use recipes as inspiration
- Buy small amounts frequently or buy in bulk infrequently
- Want to keep several in-progress lists for different purposes or just want to remember a few things to buy on the way home
- Insist on specific brands or make do with the most convenient alternatives
- Tend to buy a similar set of items on each shopping trip or buy items listed in a recipe

After musing on these questions, imagine that you decide on three characteristics that best describe your target audience: Love to experiment with recipes, are often in a hurry, and are thrifty if it doesn't take too much effort.

3. Filter the Feature List Through the Audience Definition

If, after deciding on some audience characteristics, you end up with just a few app features, you're on the right track: Great iOS apps have a laser focus on the task they help users accomplish.

For example, consider the long list of possible features you came up with in Step 1. Even though these are all useful features, not all of them are likely to be appreciated by the audience you defined in Step 2.

When you examine your feature list in the context of your target audience, you conclude that your app should focus on three main features: Creating lists, getting and using coupons, and getting recipes.

Now you can craft your app definition statement, concretely summarizing what the app does and for whom. A good app definition statement for this grocery-shopping app might be:

"A shopping list creation tool for thrifty people who love to cook."

4. Don't Stop There

Use your app definition statement throughout the development process to determine the suitability of features, controls, and terminology. For example:

As you consider adding a new feature, ask yourself whether it's essential to the main purpose of your app and to your target audience. If it isn't, set it aside; it might form the basis of a different app. For example, you've decided that your users are interested in adventurous cooking, so emphasizing boxed cake mixes and ready-made meals would probably not be appreciated.

As you consider the look and behavior of the UI, ask yourself whether your users appreciate a simple, streamlined style or a more overtly thematic style. Be guided by what people might expect to accomplish with your app, such as the ability to accomplish a serious task, to get a quick answer, to delve into comprehensive content, or to be entertained. For example, although your grocery list app needs to be easy to understand and quick to use, your audience would probably appreciate a themed UI that displays plenty of beautiful pictures of ingredients and meals.

As you consider the terminology to use, strive to match your audience's expertise with the subject. For example, even though your audience might not be made up of expert chefs, you're fairly confident that they want to see the proper terms for ingredients and techniques.

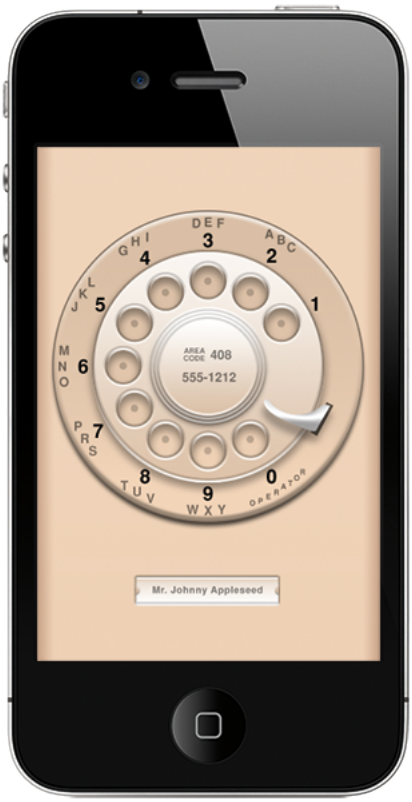
Tailor Customization to the Task

The best iOS apps balance UI customization with clarity of purpose and ease of use. To achieve this balance in your app, be sure to consider customization early in the design process. Because concerns about branding, originality, and marketability often influence customization decisions, it can be challenging to stay focused on how customization impacts the user experience.

Start by considering the tasks in your app: How often do users perform them and under what circumstances?

For example, imagine an app that enables phone calls. Now imagine that instead of a keypad, the app displays a beautiful, realistic rotary dial. The dial is meticulously rendered, so users appreciate its quality. The dial behaves realistically, so users delight in making the old-fashioned dialing gesture and hearing the distinctive sounds.

But for users who often need to dial phone numbers, initial appreciation of the experience soon gives way to frustration, because using a rotary dial is much less efficient than using a keypad. In an app that is designed to help people make phone calls, this beautiful custom UI is a hindrance.



On the other hand, consider the BubbleLevel sample app, which displays a realistic rendition of a carpenter's level. People know how to use the physical tool so they instantly know how to use the app. The app could have displayed its information without the rendition of the bubble vial, but this would have made the app less intuitive and perhaps harder to use. In this case, the custom UI not only shows people how to use the app, it also makes the task easier to accomplish.

As you think about how customization might enhance or detract from the task your app enables, keep these guidelines in mind.

Always have a reason for customization. Ideally, UI customization facilitates the task people want to perform and enhances their experience. As much as possible, you need to let your app's task drive your customization decisions.

As much as possible, avoid increasing the user's cognitive burden. Users are familiar with the appearance and behavior of the standard UI elements, so they don't have to stop and think about how to use them. When faced with elements that don't look or behave at all like standard ones, users lose the advantage of their prior experience. Unless your unique elements make performing the task easier, users might dislike being forced to learn new procedures that don't transfer to any other apps.

Be internally consistent. The more custom your UI is, the more important it is for the appearance and behavior of your custom elements to be consistent within your app. If users take the time to learn how to use the unfamiliar controls you create, they expect to be able to rely on that knowledge throughout your app.

Always defer to the content. Because the standard elements are so familiar, they don't compete with the content for people's attention. As you customize your UI, take care to ensure that it doesn't overshadow the content people care about. For example, if your app allows people to watch videos, you might choose to design custom playback controls. But whether you use custom or standard playback controls is less important than whether the controls fade out after the user begins watching the video and reappear with a tap.

Think twice before you redesign a standard control. If you plan on doing more than customizing a standard control, make sure your redesigned control provides as much information as the standard one. For example, if you create a switch control that doesn't indicate the presence of the opposite value, people might not realize that it's a two-state control.

Be sure to thoroughly user-test custom UI elements. During testing, closely observe users to see if they can predict what your elements do and if they can interact with them easily. If, for example, you create a control that has a hit target smaller than 44 x 44 points, people will have trouble activating it. Or if you create a view that responds differently to a tap than it does to a swipe, be sure the functionality the view provides is worth the extra care people have to take when interacting with it.

Prototype & Iterate

Before you invest significant engineering resources into the implementation of your design, it's a really good idea to create prototypes for user testing. Even if you can get only a few colleagues to test the prototypes, you'll benefit from their fresh perspectives on your app's functionality and user experience.

In the very early stages of your design you can use paper prototypes or wireframes to lay out the main views and controls, and to map the flow among screens. You can get some useful feedback from testing wireframes, but their sparseness may mislead testers. This is because it's difficult for people to imagine how the experience of an app will change when wireframes are filled in with real content.

You'll get more valuable feedback if you can put together a fleshed-out prototype that runs on a device. When people can interact with your prototype on a device, they're more likely to uncover places where the app doesn't function as they expect, or where the user experience is too complex.

The easiest way to create a credible prototype is to use a storyboard-based Xcode template to build a basic app, and populate it with some appropriate placeholder content. (A storyboard file captures the entire UI of your app, including the transitions among different screens.) Then, install the prototype on a device so that your testers can have as realistic an experience as possible.

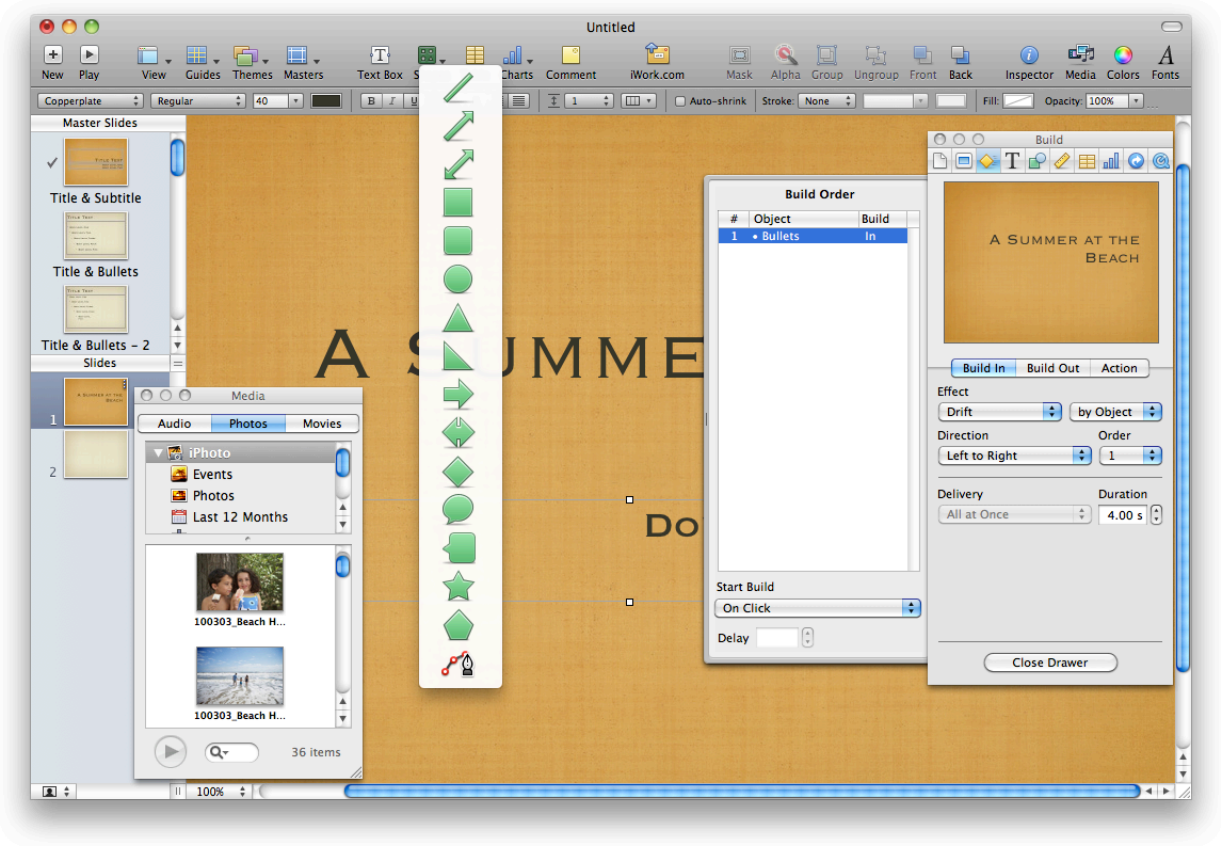
You don't need to supply a large amount of content or enable every control in your prototype app, but you do need to provide enough context to suggest a realistic experience. Aim for a balance between the typical user experience and the more unusual edge cases. For example, if it's likely that your app will handle long lists of items, you should avoid creating a prototype that displays only one or two list items. And for testing user interactions, as long as testers can tap an area of the screen to advance to the next logical view or to perform the main task, they'll be able to provide constructive feedback.

When you base your prototype on an Xcode app template, you get lots of functionality for free and it's relatively easy to make design adjustments in response to feedback. With a short turnaround time, you should be able to test several iterations of your prototype before you solidify your design and commit resources to its implementation. To get started learning about Xcode, see *Xcode User Guide* .

Case Study: From Desktop to iOS

Keynote on iPad

Keynote on the desktop is a powerful, flexible app for creating world-class slide presentations. People love how Keynote combines ease of use with fine-grained control over myriad precise details, such as animations and text attributes.



Keynote on iPad captures the essence of Keynote on the desktop, and makes it feel at home on iPad by creating a user experience that:

- Focuses on the user's content
- Reduces complexity without diluting capability
- Provides shortcuts that empower and delight
- Adapts familiar hallmarks of the desktop experience

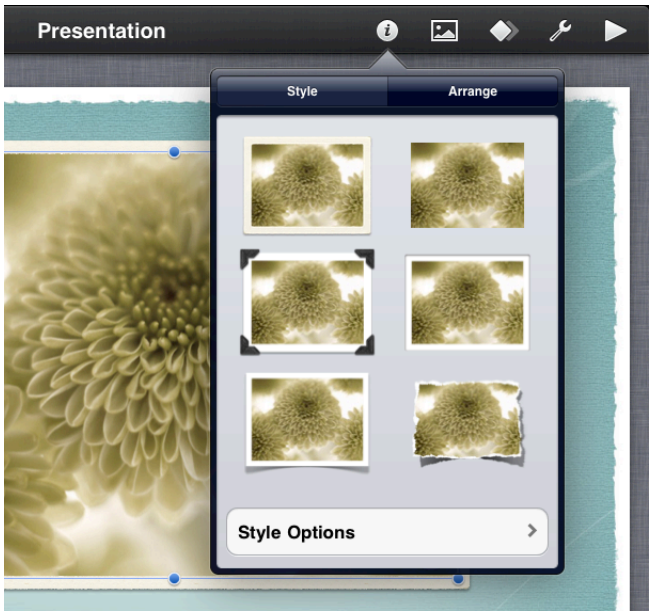
- Provides feedback and communication via eloquent animation

Keynote users instantly understand how to use the app on iPad because it delivers expected functionality using native iPad paradigms. New users easily learn how to use Keynote on iPad because they can directly manipulate their content in simple, natural ways.

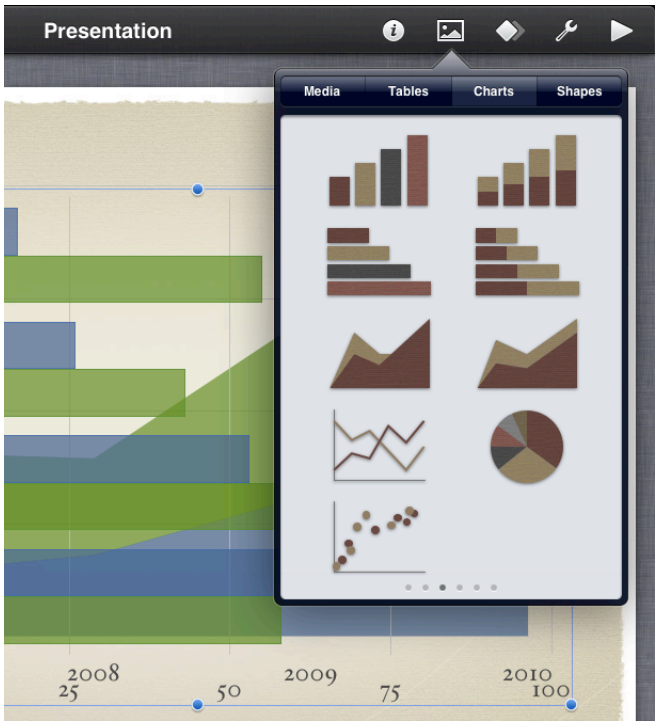
The transformation of Keynote from the desktop to iPad is based on myriad modifications and redesigns that range from subtle to profound. These are some of the most visible adaptations:

A streamlined toolbar. Only seven items are in the toolbar, but they give users consistent access to all the functions and tools they need to create their content.

A simplified, prioritized inspector that responds to the user's focus. The Keynote on iPad inspector automatically contains the tools and attributes people need to modify the selected object. Often, people can make all the modifications they need in the first inspector view. If they need to modify less frequently changed attributes, they can drill down to other inspector views.



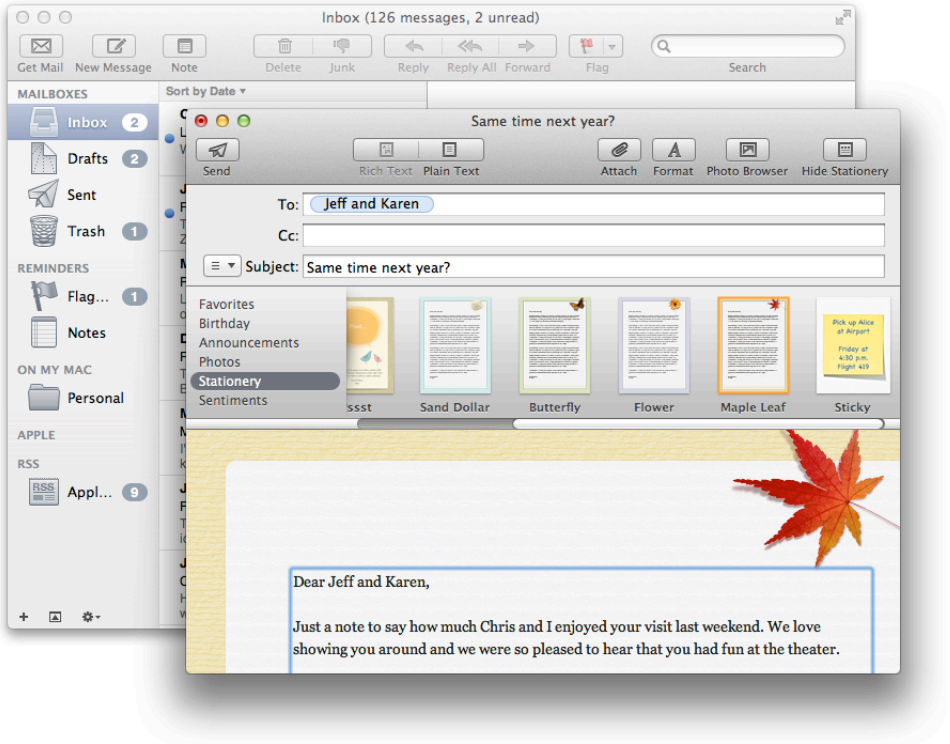
Lots of prebuilt style collections. People can easily change the look and feel of objects such as charts and tables by taking advantage of the prebuilt styles. In addition to color scheme, each collection includes prestyled attributes, such as table headings and axis-division marks, that are designed to coordinate with the overall theme.



Direct manipulation of content, enriched with meaningful animation. In Keynote on iPad, a user drags a slide to a new position, twists an object to rotate it, and taps an image to select it. The impression of direct manipulation is enhanced by the responsive animations Keynote on iPad performs. For example, a slide pulses gently as users move it and, when they place it in a new location, the surrounding slides ripple outward to make room for it.

Mail on iPhone

Mail is one of the most highly visible, well-used, and appreciated apps in OS X. It is also a very powerful program that allows users to create, receive, prioritize, and store email, track action items and events, and create notes and invitations. Mail on the desktop offers this powerful functionality in a couple of windows.



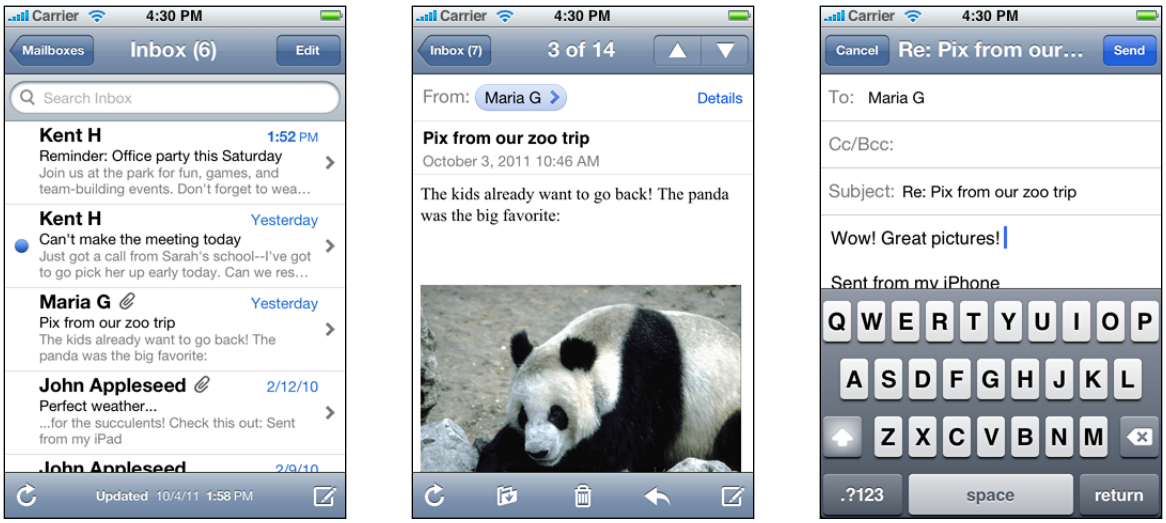
Mail on iPhone focuses on the core functionality of Mail on the desktop, helping people to receive, create, send, and organize their messages. Mail on iPhone delivers this condensed functionality in a UI tailored for the mobile experience that includes:

- A streamlined appearance that puts people's content front and center
- Different views designed to facilitate different tasks
- An intuitive information structure that scales effortlessly
- Powerful editing and organizing tools that are available when they're needed
- Subtle but expressive animation that communicates actions and provides feedback

It's important to realize that Mail on iPhone isn't a better app than Mail on the desktop; rather, it's Mail, redesigned for mobile users. By concentrating on a subset of desktop features and presenting them in an attractively lean UI, Mail on iPhone gives people the core of the Mail experience while they're mobile.

To adapt the Mail experience to the mobile context, Mail on iPhone innovates the UI in several key ways.

Distinct, highly focused screens. Each screen displays one aspect of the Mail experience: account list, mailbox list, message list, message view, and composition view. Within a screen, people scroll to see the entire contents.



Easy, predictable navigation. Making one tap per screen, people drill down from the general (the list of accounts) to the specific (a message). Each screen displays a title that shows people where they are, and a back button that makes it easy for them to retrace their steps.

Simple, tappable controls, available when needed. Because composing a message and checking for new email are primary actions people might want to take in any context, Mail on iPhone makes them accessible in multiple screens. When people are viewing a message, functions such as reply, move, and trash are available because they act upon a message.

Different types of feedback for different tasks. When people delete a message, it animates into the trash icon. When people send a message, they can see its progress; when the send finishes, they can hear a distinctive sound. By looking at the subtle text in the message list toolbar, people can see at a glance when their mailbox was last updated.

Web Content in iOS

Safari on iOS provides a preeminent mobile web-viewing experience on iOS devices. People appreciate the crisp text and sharp images and the ability to adjust their view by rotating the device or pinching and tapping the screen.

Standards-based websites display well on iOS devices. In particular, websites that detect the device and do not use plug-ins look great on both iPhone and iPad with little, if any, modification.

In addition, the most successful websites typically:

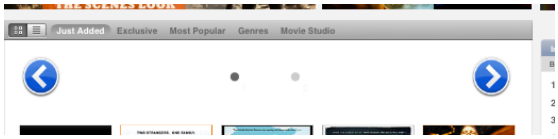
- Set the viewport appropriately for the device, if the page width needs to match the device width
- Avoid CSS fixed positioning, so that content does not move offscreen when users zoom or pan the page
- Enable a touch-based UI that does not rely on pointer-based interactions

Sometimes, other modifications can be appropriate. For example, web apps always set the viewport width appropriately and often hide the UI of Safari on iOS. To learn more about how to make these modifications, see "Configuring the Viewport" and "Configuring Web Applications" in *Safari Web Content Guide*.

Websites adapt the desktop web experience to Safari on iOS in other ways, too:

Using custom CSS to provide adaptable UI. Different UI elements can be suitable for different environments. For example, the Apple website includes a page that displays movie trailers users can watch. When viewed in Safari on the desktop, users can click the numbers or the previous and next controls to see additional trailers:

When viewed on iPhone, the next, previous, and number controls are replaced by prominent, easy-to-use buttons with symbols that echo the style of built-in controls (such as the detail disclosure indicator and the page indicator):



Accommodating the keyboard in Safari on iOS. When a keyboard and the form assistant are visible, Safari on iPhone displays your webpage in the area below the URL text field and above the keyboard and form assistant.

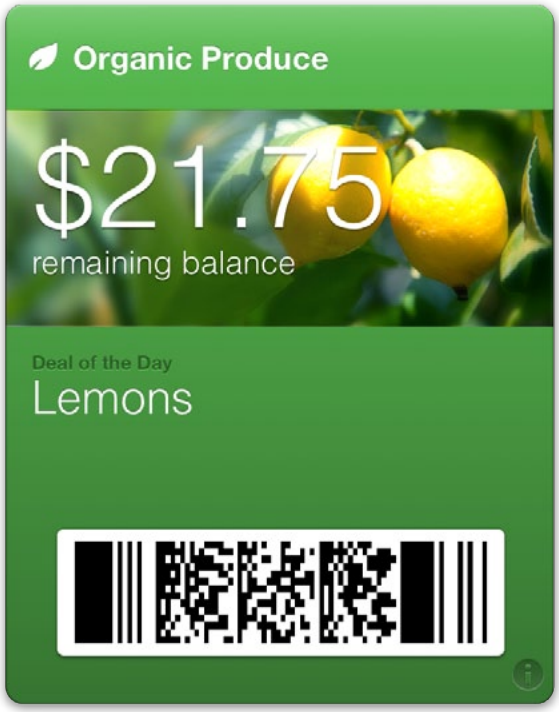
Accommodating the pop-up menu control in Safari on iOS. In Safari on the desktop, a pop-up menu that contains a large number of items displays as it does in an OS X app; that is, the menu opens to display all items, extending past the window boundaries, if necessary. In Safari on iOS, a pop-up menu is displayed using native elements, which provides a much better user experience. For example, on iPhone, the pop-up menu appears in a **picker**, a list of choices from which the user can pick. (To learn more about the picker control, see "Picker" (page 170).)

iOS Technologies

Passbook

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

The Passbook app helps people view and manage passes, which are digital representations of physical items such as boarding passes, coupons, membership cards, and tickets. In your app, you can create a pass, distribute it to users, and update it when things change.



The Pass Kit framework makes it easy to use custom content to assemble a pass and to access a pass when it's in the user's pass library. (To learn about the key concepts of Passbook technology and how to use the Pass Kit APIs in your app, see *Passbook Programming Guide*.) The following guidelines can help you create a pass that people appreciate having in their pass library and enjoy using.

As much as possible, avoid simply reproducing an existing physical pass. Passbook has an established design aesthetic and passes that coordinate with this aesthetic tend to look best. Instead of replicating the appearance of a physical item, take this opportunity to design a clean, simple pass that follows the form and function of Passbook.

Be selective about the information you put on the front of a pass. People expect to be able to glance at a pass and quickly get the information they need, so the front of a pass should be uncluttered and easy to read. If there's additional information that you think people might need, it's better to put it on the back of the pass than to squeeze it onto the front.

In general, avoid using a plain white background. A pass tends to look best when its background is a vivid, solid color or displays an image that uses strong, vibrant colors. As you design the background, always make sure that it doesn't interfere with the readability of the content.

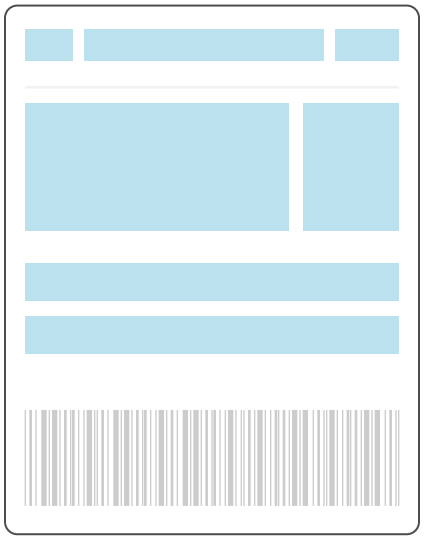
Use the logo text field for your company name. Text in the logo text field is rendered in a consistent font on all passes. To avoid clashing with other passes in the user's pass library, it's recommended that you enter text into the logo text field instead of using a custom font.

> **Note:** It's best to use the appropriate pass fields for all of the text in your pass and avoid embedding text in images or using custom fonts. Using the fields benefits you in two important ways: It allows VoiceOver users to get all the information in your pass and it gives your pass a consistent appearance.

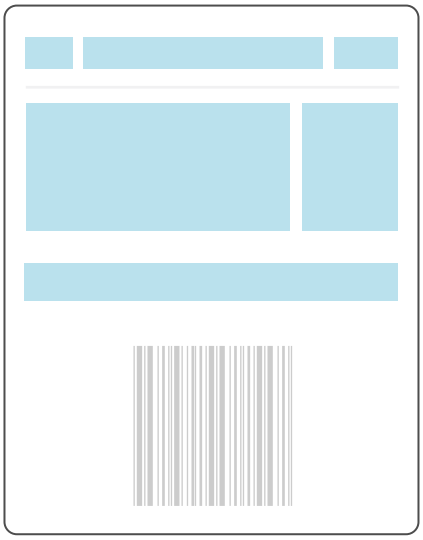
Use a white company logo. The logo image is placed in the upper-left corner of the pass, next to your company name. For best results, supply a white, monochrome version of your logo that doesn't include text. If you want to engrave the logo so that it matches the rendered logo text, add a black drop shadow with a 1 pixel y offset, a 1 pixel blur, and 35% opacity.

Use a rectangular barcode when possible. Because of the layout of a pass, a rectangular barcode—such as PDF417—tends to look better than a square barcode. As shown below on the right, a square barcode creates empty gutters on both sides and can vertically crowd the fields above and below it.

A rectangular barcode fits well in the overall layout



A square barcode can crowd other fields

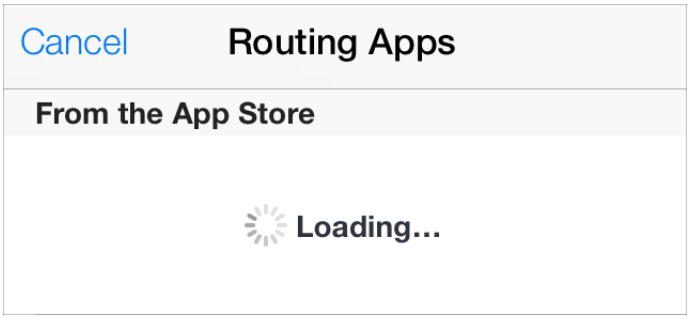


Optimize images for performance. Because users often receive passes via email or Safari, it's important to make downloads as fast as possible. To improve the user experience, use the smallest image files that achieve the desired visual appearance.

Enhance the utility of a pass by updating it when appropriate. Even though a pass represents a physical item that doesn't typically change, your digital pass can provide a better experience by reflecting real-world events. For example, you can update an airline boarding pass when a flight is delayed so that people always get current information when they check the pass.

Routing

In iOS 6 and later, Maps displays a list of routing apps—including apps installed on the device and in the App Store—when people want transit information for a route.



A **routing app** provides information about transit options for the currently selected route. People expect routing apps to be quick, easy to use, and—above all—accurate. Following the guidelines in this section helps you give users transit information they can trust and a user experience they appreciate.

> **Important:** Maps gives people driving and walking directions for their route. Routing apps provide **transit information**, which focuses on step-by-step directions that use alternate modes of transportation—such as bus, train, subway, ferry, bike, pedestrian, shuttle, and so on.
>
> If your app doesn't provide transit information for the routes that people specify, don't identify it as a routing app.

Deliver the functionality your app promises. When people see your app in the transit list, they assume that it can help them reach their destination. But if your app can't provide information about the selected route—or it doesn't include the type of transit it appears to include—people are unlikely to give it a second chance. It's essential to represent your app's capabilities accurately; otherwise, your app can look like it's intentionally misleading users.

There are two main ways you can give users confidence in your routing app:

- Define the geographic regions you support as precisely as possible. For example, if your app helps people get information about bus routes in Paris, your supported region should be Paris, not Île-de-France, and not France.
- Be specific about the types of transit that you support. For example, if you specialize in subway information, don't imply that you provide information about all rail-based transit types.

> **Note:** Although accurately reporting your supported region can mean that your app appears in the transit list less often, doing so helps users trust it more.

Streamline the UI for ease of use. Ease of use is especially important for routing apps because people tend to use them under challenging conditions—such as in bright sunlight or in the dim interior of a train, on a bumpy ride, and when they're in a hurry. Make sure that your text is easy to read in any light and that buttons are easy to tap accurately even when the ride is not smooth.

Focus on the route. Although auxiliary information can be useful, your app should focus on giving users step-by-step directions they can follow to their destination. In particular, you want users to know which step they're in and how to get to the next step. You can provide additional data—such as timetables and system maps—but don't make this data more prominent than the transit information.

Provide information for every step of a route. People should never feel abandoned by your app. But even when you accurately report your supported region, you can't assume that users are already at the first transit stop in a route, or that the last transit stop is at the same location as their destination. To handle this situation, first examine the distances at the beginning and end of the route. If the distances are short enough, provide walking directions from the user's current location to the first transit stop and from the last transit stop to the user's final destination. If walking is not a reasonable choice, try to describe the user's other options. If necessary, you can give users a way to open Maps to get walking or driving directions for these portions of the route.

When users transition to your app from Maps, don't ask them to reenter information. If users are coming from Maps, you already know the start and end points of the route they're interested in, so you can present the appropriate transit information as soon as your app opens. If users start your app from the Home screen, provide an easy way for them to enter route details.

Display transit information both graphically and textually. A map view helps people see their complete route in a larger, physical context; a list of steps helps people focus on the actions they must take to arrive at their destination. It's best when you support both of these tasks and make it easy for users to switch between them.

> **Note:** Regardless of format, it's crucial that you always display the same transit information for the user's route. For example, if a route consists of five steps, both the map and the list view of the route must describe the same five steps.

When your app is chosen from the transit list, it works well to start by displaying the complete route—including walking paths to and from the transit stops, if appropriate—in a map view. A map view gives users an overview of the various steps in their journey and shows them how their route fits into the surrounding geographical area.

Enrich map views with additional information. People expect the maps in your app to behave similarly to other maps they've used. In addition to letting users zoom and pan, you should display annotations that give users the information they need. For example, you could display pins that represent the user's current location, the destination, and transfers or points of interest along the way. Be sure to avoid displaying only a single pin, because it's hard for users to tell what it represents if there's no additional context. For more information about using map views in your app, see "Map View" (page 139).

As much as possible, integrate static maps—such as a subway system map—with a map view. A good way to do this is to overlay the static image on the map view so that users can see how their route and their current location relate to the larger transit system.

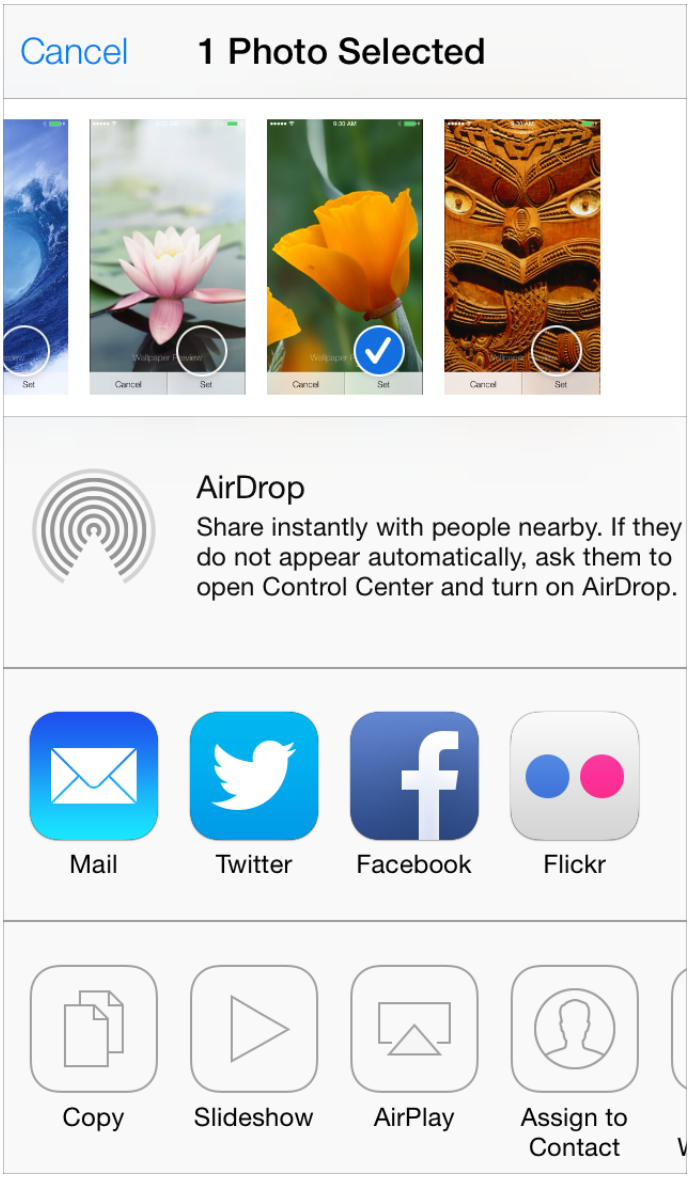
> **Note:** If you decide to display a static map image by itself, be sure to use a high-resolution image that maintains good quality when users zoom into it.

Give users different ways to sort multiple transit options. Lots of factors influence people's transit decisions—such as time of day, weather, and how much they're carrying—so it's important to make it easy to compare transit options. For example, you could let users sort transit options by start or end time, amount of walking required, number of stops along the way, or number of transfers or different transit types required. Regardless of the order in which you display multiple transit options, make sure that users can instantly distinguish the differences between the options.

Consider using push notifications to give people important information about their route. As much as possible, let people know when transit information changes, so that they can adjust their plans. For example, if a train is delayed or a bus line is temporarily unavailable people might need to choose a different route to their destination. And in a route that includes long stops between steps, people might appreciate being notified when their transport is about to depart for the next part of the journey.

Social Media

People expect to have access to their favorite social media accounts regardless of their current context. iOS makes it easy to integrate social media interactions into your app in ways that people appreciate.



Give users a convenient way to compose a post without leaving your app. As much as possible, you want to integrate social media support into your app so users can post content to their account without switching to another app to do so. The Social framework provides a compose view controller that allows you to present users with a view in which they can edit a post. Optionally, you can prepopulate the compose view with custom content before you present it to users for editing (after you present the view to users, only they can edit the content). To learn about the programming interfaces of the Social framework—including the `SLComposeViewController` class—see *Social Framework Reference*.

When possible, avoid asking users to sign into a social media account. The Social framework works with the Accounts framework to support a single sign-on model, so you can get authorization to access the user's account without asking them to reauthenticate. If the user hasn't already signed into an account, you can present UI that allows them to do so.

Consider using an activity view controller to help users choose one of their social media accounts. By default, an activity view controller—that is, a `UIActivityViewController` object—lists several system-provided services that act upon the currently selected content, including sending the content via Mail or Messages and posting the content to social media accounts. When you use an activity view controller, you don't have to provide a custom service that interacts with a social media account and you benefit from the user's familiarity with the Share button that reveals the list of services. For guidelines on how to use an activity view controller in your app, see "Activity View Controller" (page 133).

iCloud

iCloud lets people access the content they care about regardless of which device they're currently using. When you integrate iCloud into your app, users can use different instances of your app on different devices to view and edit their personal content without performing explicit synchronization.

To provide this user experience, it's likely that you'll need to reexamine the ways in which you store, access, and present information—especially user-created content—in your app. To learn how to enable iCloud in your app, see *iCloud Design Guide*.

A fundamental aspect of the iCloud user experience is transparency: Ideally, users don't need to know where their content is located and they should seldom have to think about which version of the content they're currently viewing. The following guidelines can help you give users the iCloud experience they're expecting.

If appropriate, make it easy for users to enable iCloud for your app. On their iOS devices, users log into their iCloud account in iCloud Settings, and for the most part, they expect their apps to work with iCloud automatically. But if you think users might want to choose whether to use iCloud with your app, you can provide a simple option that they can set when they open your app for the first time. In most cases, this option should provide a choice between using iCloud with all the content that users access in your app or not at all.

Respect the user's iCloud space. It's important to remember that iCloud is a finite resource for which users pay. You should use iCloud for storing information that users create and understand, and avoid using it to store app resources or content that you can regenerate. Also, note that when the user's iCloud account is active, iCloud automatically backs up the contents of your app's Documents folder. To avoid using up too much of the user's space, it's best to be picky about the content you place in the Documents folder.

Avoid asking users to choose which documents to store in iCloud. Typically, users expect all the content they care about to be available via iCloud. Most users don't need to manage the storage of individual documents, so you shouldn't assume that your app needs to support this experience. To provide a good user experience, you might want to rearchitect the way your app handles and exposes content so that you can perform more file-management tasks for the user.

Determine which types of information to store in iCloud. In addition to storing user-created documents and other content, you can also store small amounts of data such as the user's current state in your app or their preferences. To store this type of information you use iCloud key-value storage. For example, if people use your app to read a magazine, you might use iCloud key-value storage to store the last page they viewed so that when they reopen the issue on a different device, they can continue reading from where they left off.

If you use iCloud key-value storage to store preferences, be sure that the preferences are ones that users are likely to want to have applied to all their devices. For example, some preferences are more useful in a work environment than they are in a home environment. In some cases, it can make sense to store preferences on your app's server, instead of in the user's iCloud account, so that the preferences are available regardless of whether iCloud is enabled.

Make sure that your app behaves reasonably when iCloud is unavailable. For example, if users log out of their iCloud account, turn off iCloud usage for your app, or enter Airplane mode, iCloud becomes unavailable. In these cases, users performed an action that turned off access to iCloud, so your app does not need to tell them about it. However, it can be appropriate to show users that the changes they make will not be visible on their other devices until they restore access to iCloud.

Avoid giving users the option to create a "local" document. Regardless of whether you support iCloud in your app, you should not encourage users to think in terms of a device-specific file system. Instead, you want users to focus on the pervasive availability of their content through iCloud.

When appropriate, update content automatically. It's best when users don't have to take any action to ensure that they're accessing the most up-to-date content in your app. However, you need to balance this experience with respect for the user's device space and bandwidth constraints. If your users work with very large documents, it can be appropriate to give them control over whether to download an update from iCloud. If you need to do this, design a way to indicate that a more recent version of the document is available in iCloud. When the user chooses to update the document, be sure to provide subtle feedback if the download takes more than a few seconds.

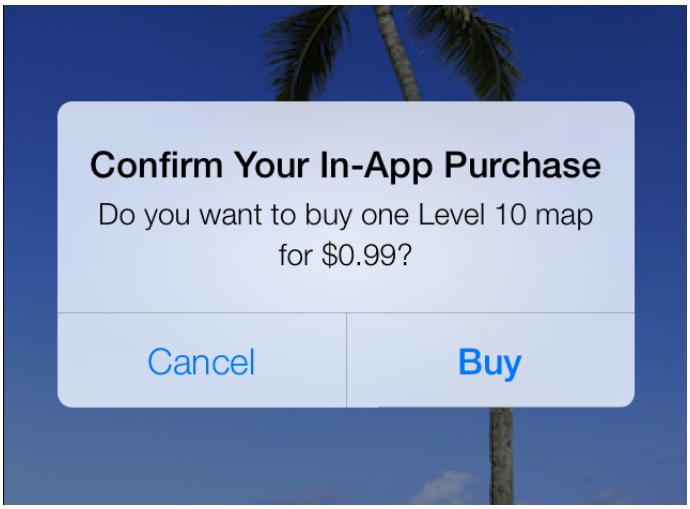
Warn users about the consequences of deleting a document. When a user deletes a document in an iCloud-enabled app, the document is removed from the user's iCloud account and all other devices. It's appropriate to display an alert that describes this result and to get confirmation before you perform the deletion.

Tell users about conflicts as soon as possible, but only when necessary. Using the iCloud programming interfaces, you should be able to resolve most conflicts between different versions of a document without involving the user. In cases where this is not possible, make sure that you detect conflicts as soon as possible so that you can help users avoid wasting time on the wrong version of their content. You need to design an unobtrusive way to show users that a conflict exists; then, make it easy for users to differentiate between versions and make a decision.

Be sure to include the user's iCloud content in searches. Users with iCloud accounts tend to think of their content as being universally available, and they expect search results to reflect this perspective. If your app allows people to search their content, make sure you use the appropriate APIs to extend search to their iCloud accounts. (To learn how to search content in iCloud, see "Incorporating Search into Your Infrastructure" in *iOS App Programming Guide* .)

In-App Purchase

In-App Purchase lets people buy digital products within your app, in a store that you design.



For example, users might:

- Upgrade a basic version of an app to a premium version
- Renew a subscription for new monthly content
- Purchase virtual items, such as a new level or weapon in a game
- Buy and download new books

You use the Store Kit framework to embed a store in your app and support In-App Purchase. When a user makes a purchase, Store Kit connects to the App Store to securely process the payment and then notifies your app so that it can provide the purchased item.

Important: In-App Purchase only collects payment—you provide additional functionality, such as presenting your store to users, unlocking built-in features, and downloading content from your own servers. Also, all products you sell through In-App Purchase must be registered in the App Store.

To learn about the technical requirements of adding a store to your app, see *In-App Purchase Programming Guide*. For more information on the business requirements of using In-App Purchase, visit the App Store Resource Center. You should also read your licensing agreement for definitive information about what you may sell and how you are required to provide those products in your app.

The following guidelines can help you design a purchasing experience that users appreciate.

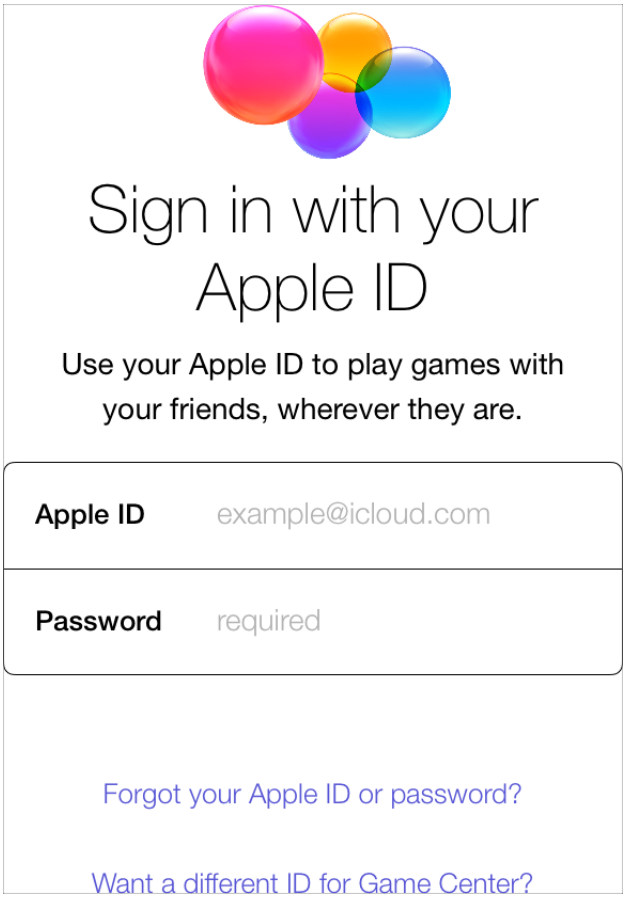
Elegantly integrate the store experience into your app. When presenting products and handling user transactions, create an experience that feels at home in your app. You don't want users to think that they've entered a different app when they visit your store.

Use simple, succinct titles and descriptions. It's best when people can scan a set of items and quickly find the ones they're interested in. When you use plain, direct language and titles that don't truncate or wrap, it's easier for people to understand the items you're offering.

Don't alter the default confirmation alert. When users buy a product, Store Kit presents a confirmation alert (shown above). You shouldn't modify this alert because it helps users avoid accidental purchases.

Game Center

Game Center lets people play games, organize online multiplayer games, and more. Players use the built-in Game Center app to sign in to an account, discover new games, add new friends, and browse leaderboards and achievements.



As a game developer, you use the Game Kit APIs to post scores and achievements to the Game Center service, display leaderboards in the game UI, and help users find other players. To learn how to integrate Game Center into your app, see *Game Center Programming Guide*.

The following guidelines can help you give people a great Game Center experience in your app.

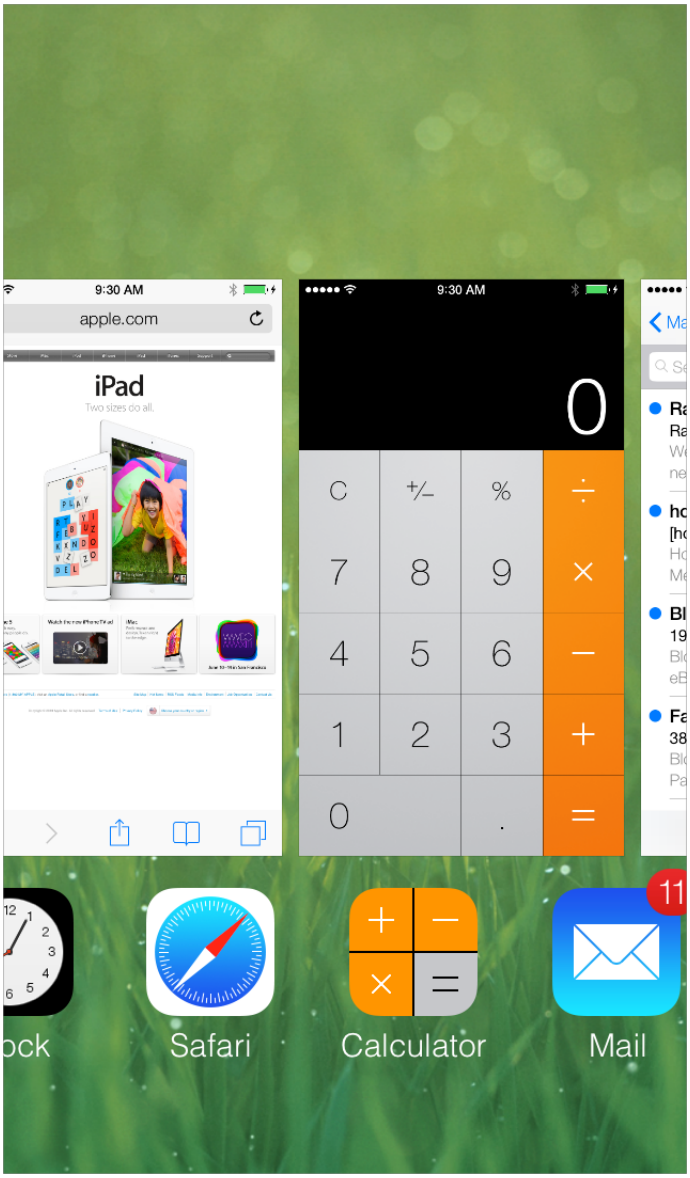
Don't create custom UI that prompts users to sign into Game Center. When people start your Game Center-enabled app—and they're not already signed into Game Center on their device—the system automatically prompts them to sign in. Displaying custom sign-in UI is unnecessary and might confuse users.

In general, use the standard Game Center UI. In rare cases, it might make sense for a game to customize the Game Center UI, but doing so risks confusing people. The standard Game Center UI—which is familiar to both iOS and OS X users—promotes the sense of belonging to a larger gaming community.

Give users the ability to turn off voice chat. Some users might not want voice chat to be on automatically when they start your game, and most users appreciate the ability to turn off voice chat in certain situations.

Multitasking

Multitasking allows people to switch quickly among recently used apps.



To support this experience, multitasking allows an app to enter a suspended state in the background when users switch away from it. When users switch back to the app, the app can resume quickly because it doesn't have to reload its UI. People use the **multitasking UI** to choose a recently used app.

Thriving in a multitasking environment hinges on achieving a harmonious coexistence with other apps on the device. At a high level, this means that all apps should:

- Handle interruptions or audio from other apps gracefully
- Stop and restart—that is, transition to and from the background—quickly and smoothly
- Behave responsibly when not in the foreground

The following specific guidelines help your app succeed in the multitasking environment.

Be prepared for interruptions, and be ready to resume. Multitasking increases the probability that a background app will interrupt your app. Other features, such as the presence of ads and faster app-switching, can also cause more frequent interruptions. The more quickly and precisely you can save the current state of your app, the faster people can relaunch it and continue from where they left off. To give users a seamless restart experience, take advantage of UIKit's state preservation and restoration functionality (for more information, see "State Preservation and Restoration").

Make sure your UI can handle the double-high status bar. The double-high status bar appears during events such as in-progress phone calls, audio recording, and tethering. In unprepared apps the extra height of this bar can cause layout problems. For example, the UI can become pushed down or covered. In a multitasking environment, it's especially important to be able to handle the double-high status bar properly because there are likely to be more apps that can cause it to appear.

Be ready to pause activities that require people's attention or active participation. For example, if your app is a game or a media-viewing app, make sure your users don't miss any content or events when they switch away from your app. When people switch back to a game or media viewer, they want to continue the experience as if they'd never left it.

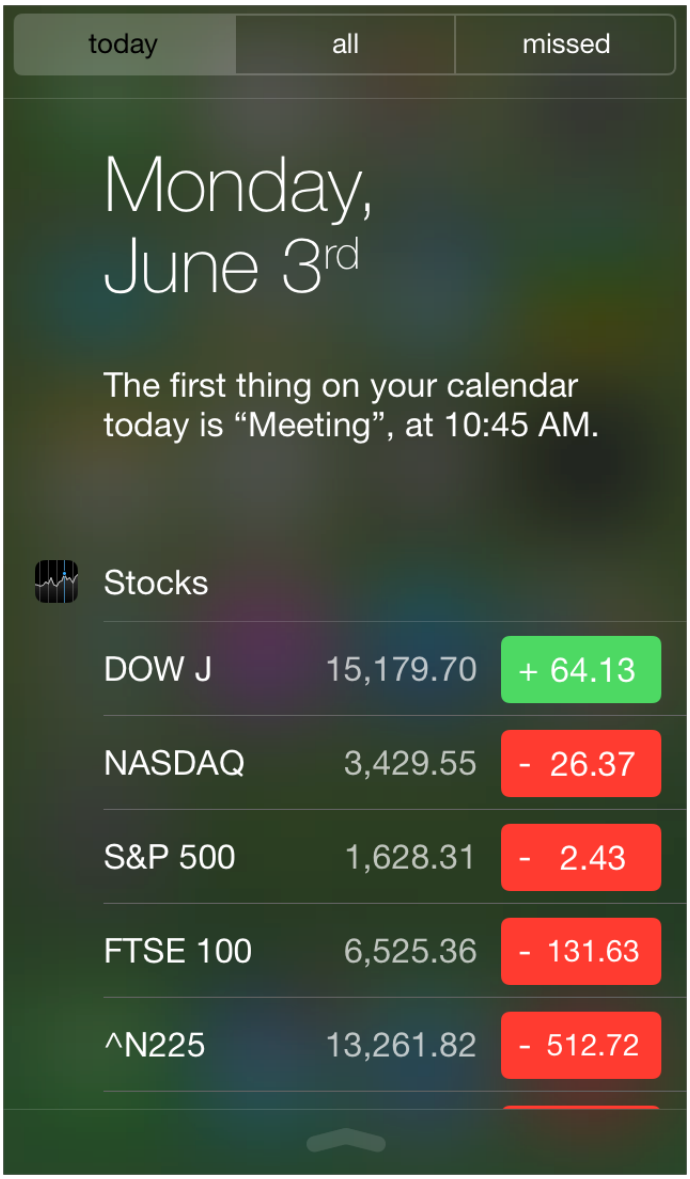
Ensure that your audio behaves appropriately. Multitasking makes it more likely that other media activity is occurring while your app is running. It also makes it more likely that your audio will have to pause and resume to handle interruptions. For specific guidelines that help you make sure your audio meets people's expectations and coexists properly with other audio on the device, see "Sound" (page 101).

Use local notifications sparingly. An app can arrange for local notifications to be sent at specific times, whether the app is suspended, running in the background, or not running at all. For the best user experience, avoid pestering people with too many notifications, and follow the guidelines for creating notification content described in "Notification Center" (page 90).

When appropriate, finish user-initiated tasks in the background. When people initiate a task, they usually expect it to finish even if they switch away from your app. If your app is in the middle of performing a user-initiated task that does not require additional user interaction, you should complete it in the background before suspending.

Notification Center

Notification Center gives users a single, convenient place in which to view notifications from their apps. Users appreciate the unobtrusive interface of Notification Center and they value the ability to customize the way each app can present its notifications.



Notification Center uses a sectioned list to display recent notification items from the apps that users are interested in. In addition to notifications, users can also choose to see weather and stock market information in Notification Center.

iOS apps can use local or push notifications to let people know when interesting things happen, such as:

- A message has arrived
- An event is about to occur
- New data is available for download
- The status of something has changed

A **local notification** is scheduled by an app and delivered by iOS on the same device, regardless of whether the app is currently running in the foreground. For example, a calendar or to-do app can schedule a local notification to alert people of an upcoming meeting or due date.

A **push notification** is sent by an app's remote server to the Apple Push Notification service, which pushes the notification to all devices that have the app installed. For example, a game that a user can play against remote opponents can update all players with the latest move.

You can still receive local and push notifications when your app is running in the foreground, but you pass the information to your users in an app-specific way.

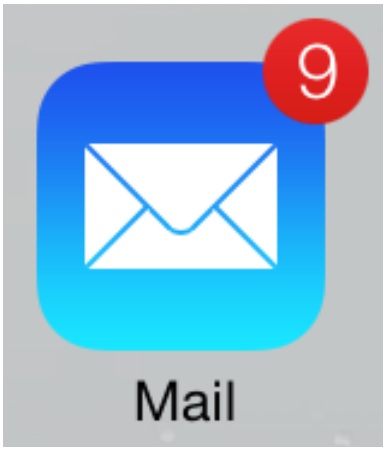
iOS apps that support local or push notifications can participate in Notification Center in various ways, depending on the user's preferences. To ensure that users can customize their notification experience, you should support as many as possible of the following notification styles:

- Banner
- Alert
- Badge
- Sound

A **banner** is a small translucent view that appears onscreen and then disappears after a few seconds. In addition to your notification message, iOS displays the small version of your app icon in a banner, so that people can see at a glance which app is notifying them (to learn more about the small app icon, see "App Icon" (page 192)).

An **alert** is a standard alert view that appears onscreen and requires user interaction to dismiss. You supply the notification message and, optionally, a title for the action button in the alert. You have no control over the background appearance of the alert or the buttons.

A **badge** is a small red oval that displays the number of pending notification items (a badge appears over the upper-right corner of an app's icon). You have no control over the size or color of the badge.



A custom or system-provided **sound** can accompany any of the other three notification delivery styles.

> **Note:** Apps that use local notifications can provide banners, alerts, badges, and sounds. But an app that uses push notifications instead of local notifications can provide only the notification types that correspond to the push categories for which the app is registered. For example, if a push-notification app registers for alerts only, users aren't given the choice to receive badges or sounds when a notification arrives.

As you design the content that your notifications can deliver, be sure to observe the following guidelines.

Keep badge contents up to date. It's especially important to update the badge as soon as users have attended to the new information, so that they don't think additional notifications have arrived. Note that setting the badge contents to zero also removes the related notification items from Notification Center.

> **Important:** Don't use a badge for purposes other than notifications. Remember that users can turn off badging for your app, so you can't be sure that they will see the content in a badge.

Don't send multiple notifications for the same event. Users can attend to notification items when they choose; the items don't disappear until users handle them in some way. If you send multiple notifications for the same event, you fill up the Notification Center list and users are likely to turn off notifications from your app.

Provide a custom message that does not include your app name. Your custom message is displayed in alerts and banners, and in Notification Center list items. You should not include your app's name in your custom message because iOS automatically displays the name with your message.

To be useful, a local or push notification message should:

- Focus on the information, not user actions. Avoid telling people which alert button to tap or how to open your app.
- Be short enough to display on one or two lines. Long messages are difficult for users to read quickly, and they can force alerts to scroll.
- Use sentence-style capitalization and appropriate ending punctuation. When possible, use a complete sentence.

> **Note:** In general, a Notification Center item can display more of a notification message than a banner can. If necessary, iOS truncates your message so that it fits well in each notification delivery style; for best results, you shouldn't truncate your message.

Optionally, provide a custom title for the action button in an alert. An alert can contain one or two buttons. In a two-button alert, the Close button is on the left and the action button (titled View by default) is on the right. If you specify one button, the alert displays an OK button.

Tapping the action button dismisses the alert and launches your app simultaneously. Tapping either the Close button or the OK button dismisses the alert without opening your app.

If you want to use a custom title for the action button, be sure to create a title that clearly describes the action that occurs when your app launches. For example, a game might use the title Play to indicate that tapping the button opens the app to a place where the user can take their turn. Make sure the title:

- Uses title-style capitalization
- Is short enough to fit in the button without truncation (be sure to test the length of localized titles, too)

> **Note:** Your custom button title can also be displayed in the "slide to view" message people see when a notification arrives while the device is locked. When this happens, your custom title is automatically converted to lowercase and replaces the word "view" in the message.

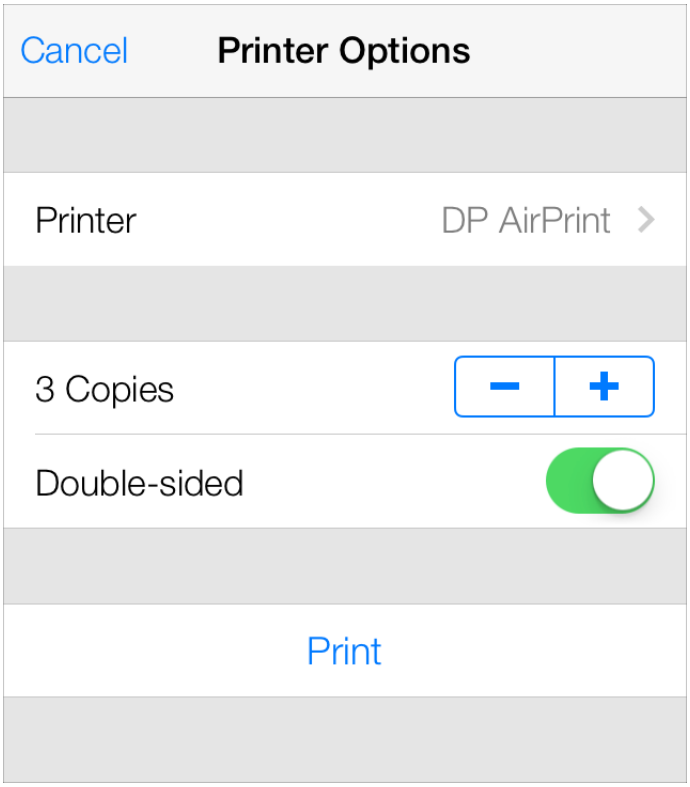
Provide a sound that users can choose to hear when a notification arrives. A sound can get people's attention when they're not looking at the device screen. Users might want to enable sounds when they're expecting a notification that they consider important. For example, a calendar app might play a sound with an alert that reminds people about an imminent event. Or, a collaborative task management app might play a sound with a badge update to signal that a remote colleague has completed an assignment.

You can supply a custom sound, or you can use a built-in alert sound. If you create a custom sound, be sure it's short, distinctive, and professionally produced. (To learn about the technical requirements for this sound, see "Preparing Custom Alert Sounds" in *Local and Push Notification Programming Guide*.) Note that you can't programmatically cause the device to vibrate when a notification is delivered, because the user has control over whether alerts are accompanied by vibration.

Optionally, provide a launch image. In addition to displaying your existing launch images, you can supply a different launch image to display when people start your app in response to a notification. For example, a game might specify a launch image that's similar to a screen within the game, instead of an image that's similar to the opening menu screen. If you don't supply this launch image, iOS displays either the previous snapshot or one of your other launch images. (To learn how to create a launch image, see "Launch Images" (page 196).)

AirPrint

Using AirPrint, people can wirelessly print content from your app and use Print Center app to check on a print job.



You can take advantage of built-in support for printing images and PDF content, or you can use printing-specific programming interfaces to do custom formatting and rendering. iOS handles printer discovery and the scheduling and execution of print jobs on the selected printer.

Typically, users tap the standard Share button in your app when they want to print something. When they choose the Print item in the view that appears, they can then select a printer, set available printing options, and tap the Print button to start the job. On iPhone, this view appears in an action sheet that slides up from the bottom of the screen; on iPad, the view appears in a popover that emerges from the button.

Users can check on the print job they requested in the Print Center app, which is a background system app that is available only while a print job is in progress. In Print Center, users can view the current print queue, get details about a specific print job, and even cancel the job.

You can support basic printing in your app with comparatively little additional code (to learn about adding print support to your code, see *Drawing and Printing Guide for iOS*). To ensure that users appreciate the printing experience in your app, follow these guidelines:

Use the system-provided Share button. Users are familiar with the meaning and behavior of this button, so it's a good idea to use it when possible. The main exception to this is if your app does not contain a toolbar or navigation bar. When this is the case, you need to design a custom print button that can appear in the main UI of your app, because the Share button can only be used in a toolbar or navigation bar.

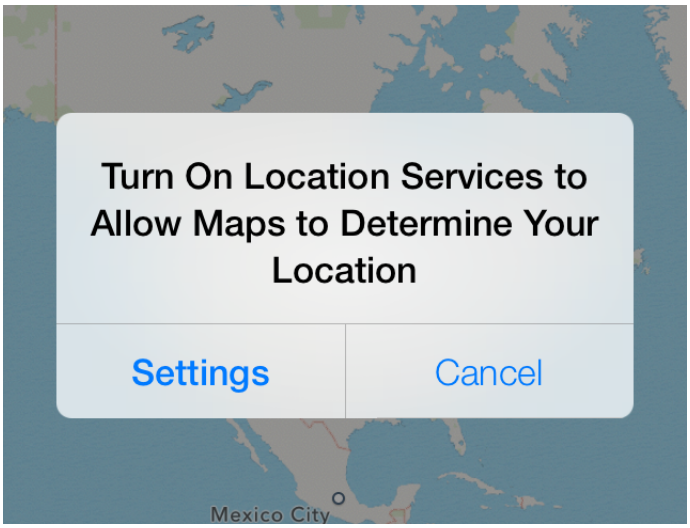
Display the Print item when printing is a primary function in the current context. If printing is inappropriate in the current context, or if users are not likely to want to print, don't include the Print item in the view revealed by the Share button.

If appropriate, provide additional printing options to users. For example, you might allow users to choose a page range or to request multiple copies.

Don't display print-specific UI if users can't print. Be sure to check whether the user's device supports printing before you display UI that offers printing as an option. To learn how to do this in your code, see *UIPrintInteractionController Class Reference* .

Location Services

Location Services allows apps to determine people's approximate location geographically, the direction they're pointing their device, and the direction in which they're moving. In iOS 6.0 and later, other system services—such as Contacts, Calendar, Reminders, and Photo Library—also allow apps to access the data people store in them.



Although people appreciate the convenience of using an app that already knows a lot about them, they also expect to have the option of keeping their data private. For example, people like being able to automatically tag content with their physical location or find friends that are currently nearby, but they also want to be able to disable such features when they don't choose to share their location with others. (To learn more about how to make your app location-aware, see *Location Awareness Programming Guide* .)

The following guidelines can help you ask for user data in ways that help people feel comfortable.

Make sure users understand why they're being asked to share their personal data. It's natural for people to be suspicious of a request for their personal information if they don't see an obvious need for it. To avoid making users uncomfortable, make sure the alert appears only when they attempt to use a feature that clearly needs to know their information. For example, people can use Maps when Location Services is off, but they see an alert when they access the feature that finds and tracks their current location.

Describe why your app needs the information, if it's not obvious. You can provide text that appears in the alert, below a system-provided title such as ""App Name" Would Like to Access Your Contacts". You want this text to be specific and polite so that people understand why you're asking for access to their information and don't feel pressured. Your reason text should:

- Not include your app name. The system-provided alert title already includes your app name.
- Clearly describe why your app needs the data. If appropriate, you might also explain ways in which your app will not use the data.
- Use user-centric terminology and be localizable.
- Be as short as possible, while still being easy to understand. As much as possible, avoid supplying more than one sentence.
- Use sentence-style capitalization. (Sentence-style capitalization means that the first word is capitalized, and the rest of the words are lowercase unless they are proper nouns or proper adjectives.)

Ask permission at app startup only if your app can't perform its primary function without the user's data. People will not be bothered by this if it's obvious that the main function of your app depends on knowing their personal information.

Avoid making programmatic calls that trigger the alert before the user actually selects the feature that needs the data. This way, you avoid causing people to wonder why your app wants their personal information when they're doing something that doesn't appear to need it. (Note that getting the user's Location Services preference does not trigger the alert.)

For location data, check the Location Services preference to avoid triggering the alert unnecessarily. You can use Core Location programming interfaces to get this setting (to learn how to do this, see *Core Location Framework Reference*). With this knowledge, you can trigger the alert as closely as possible to the feature that requires location information, or perhaps avoid an alert altogether.

Quick Look

Using Quick Look, users can preview a document within your app, even if your app can't open the document. For example, you might allow users to preview documents that they download from the web or receive from other sources.

To learn more about how to support Quick Look document preview in your app, see *Document Interaction Programming Topics for iOS*.

Before users preview a document in your app, they can see information about the document in a custom view that you create. For example, after users download a document attached to an email message, Mail on iPad displays the document's icon, title, and size in a custom view within the message. Users can tap this view to preview the document.

You can present a document preview in a new view in your app, or in a full-screen, modal view. The presentation method you choose depends on which device your app runs on.

On iPad, display a document preview modally. The large iPad screen is appropriate for displaying a document preview in an immersive environment that users can easily leave. The zoom transition is especially well-suited to reveal the preview.

On iPhone, display a document preview in a dedicated view, preferably a navigation view. Doing this allows users to navigate to and from the document preview without losing context in your app. Although it's possible to display a document preview modally in an iPhone app, it's not recommended. (Note that the zoom transition is not available on iPhone.)

Also, note that displaying a document preview in a navigation view allows Quick Look to place preview-specific navigation controls in the navigation bar. (If your view already contains a toolbar, Quick Look places the preview navigation controls in the toolbar, instead.)

Sound

Whether sound is an essential part of your app's user experience or only an incidental enhancement, you need to know how users expect sound to behave and how to meet those expectations.

Understand User Expectations

People can use device controls to affect sound, and they might use wired or wireless headsets and headphones. People also have various expectations for how their actions impact the sound they hear. Although you might find some of these expectations surprising, they all follow the principle of user control in that the user, not the device, decides when it's appropriate to hear sound.

Users switch their devices to silent when they want to:

- Avoid being interrupted by unexpected sounds, such as phone ringtones and incoming message sounds
- Avoid hearing sounds that are the byproducts of user actions, such as keyboard or other feedback sounds, incidental sounds, or app startup sounds
- Avoid hearing game sounds that are not essential to using the game, such as incidental sounds and soundtracks

> **Note:** People switch their devices to silent using either the Ring/Silent switch (on iPhone) or the Silent switch (on iPad).

For example, in a theater users switch their devices to silent to avoid bothering other people in the theater. In this situation, users still want to be able to use apps on their devices, but they don't want to be surprised by sounds they don't expect or explicitly request, such as ringtones or new message sounds.

The Ring/Silent (or Silent) switch does *not* silence sounds that result from user actions that are solely and explicitly intended to produce sound. For example:

- Media playback in a media-only app is not silenced because the media playback was explicitly requested by the user.
- A Clock alarm is not silenced because the alarm was explicitly set by the user.
- A sound clip in a language-learning app is not silenced because the user took explicit action to hear it.

- Conversation in an audio chat app is not silenced because the user started the app for the sole purpose of having an audio chat.

Users use the device's volume buttons to adjust the volume of all sounds their devices can play, including songs, app sounds, and device sounds. Users can use the volume buttons to quiet any sound, regardless of the position of the Ring/Silent (or Silent) switch. Using the volume buttons to adjust an app's currently playing audio also adjusts the overall system volume, with the exception of the ringer volume.

> **iPhone:** Using the volume buttons when no audio is currently playing adjusts the ringer volume.

Users use headsets and headphones to hear sounds privately and to free their hands. Regardless of whether these accessories are wired or wireless, users have specific expectations for the user experience.

When users plug in a headset or headphones, or connect to a wireless audio device, they intend to continue listening to the current audio, but privately. For this reason, they expect an app that is currently playing audio to continue playing without pause.

When users unplug a headset or headphones, or disconnect from a wireless device (or the device goes out of range or turns off), they don't want to automatically share what they've been listening to with others. For this reason, they expect an app that is currently playing audio to pause, allowing them to explicitly restart playback when they're ready.

Define the Audio Behavior of Your App

If necessary, you can adjust relative, independent volume levels to produce the best mix in your app's audio output. But the volume of the final audio output should always be governed by the system volume, whether it's adjusted by the volume buttons or a volume slider. This means that control over an app's audio output remains in users' hands, where it belongs.

Ensure that your app can display the audio route picker, if appropriate. (An **audio route** is an electronic pathway for audio signals, such as from a device to headphone or from a device to speakers.) Even though people don't physically plug in or unplug a wireless audio device, they still expect to be able to choose a different audio route. To handle this, iOS automatically displays a control that allows users to pick an output audio route (use the `MPVolumeView` class to allow the control to display in your app). Because choosing a different audio route is a user-initiated action, users expect currently playing audio to continue without pause.

If you need to display a volume slider, be sure to use the system-provided volume slider available when you use the `MPVolumeView` class. Note that when the currently active audio output device does not support volume control, the volume slider is replaced by the appropriate device name.

If your app produces only UI sound effects that are incidental to its functionality, use System Sound Services. System Sound Services is the iOS technology that produces alerts and UI sounds and invokes vibration; it is unsuitable for any other purpose. When you use System Sound Services to produce sound, you cannot influence how your audio interacts with audio on the device, or how it should respond to interruptions and changes in device configuration. For a sample project that demonstrates how to use this technology, see *Audio UI Sounds (SysSound)* .

If sound plays an important role in your app, use Audio Session Services or the `AVAudioSession` class. These programming interfaces do not produce sound; instead, they help you express how your audio should interact with audio on the device and respond to interruptions and changes in device configuration.

> **iPhone:** No matter what technology you use to produce audio or how you define its behavior, the phone can always interrupt the currently running app. This is because no app should prevent people from receiving an incoming call.

In Audio Session Services, the **audio session** functions as an intermediary for audio between your app and the system. One of the most important facets of the audio session is the **category**, which defines the audio behavior of your app.

To realize the benefits of Audio Session Services and provide the audio experience users expect, you need to select the category that best describes the audio behavior of your app. This is the case whether your app plays only audio in the foreground or can also play audio in the background. Follow these guidelines as you make this selection:

- **Select an audio session category based on its semantic meaning, not its precise set of behaviors.** By selecting a category whose purpose is clear, you ensure that your app behaves according to users' expectations. In addition, it gives your app the best chance of working properly if the exact set of behaviors is refined in the future.
- **In rare cases, add a property to the audio session to modify a category's standard behavior.** A category's standard behavior represents what most users expect, so you should consider carefully before you change that behavior. For example, you might add the ducking property to make sure your audio is louder than all other audio (except phone audio), if that's what users expect from your app. (To learn more about audio session properties, see "Fine-Tuning the Category" in *Audio Session Programming Guide* .)
- **Consider basing your category selection on the current audio environment of the device.** This might make sense if, for example, users can use your app while listening to other audio instead of to your soundtrack. If you do this, be sure to avoid forcing users to stop listening to their music or make an explicit soundtrack choice when your app starts.

- **In general, avoid changing categories while your app is running.** The primary reason for changing the category is if your app needs to support recording and playback at different times. In this case, it can be better to switch between the Record category and the Playback category as needed, than to select the Play and Record category. This is because selecting the Record category ensures that no alerts—such as an incoming text message alert—will sound while the recording is in progress.

"Audio session categories and their associated behaviors" lists the audio session categories you can use. Different categories allow sounds to be silenced by the Ring/Silent or Silent switch (or device locking), to mix with other audio, or to play while the app is in the background. (For the actual category and property names as they appear in the programming interfaces, see *Audio Session Programming Guide*.)

Table 28-1 Audio session categories and their associated behaviors

Category	Meaning	Silenced	Mixes	In Background
Solo Ambient	Sounds enhance app functionality, and should silence other audio.	Yes	No	No
Ambient	Sounds enhance app functionality but should not silence other audio.	Yes	Yes	No
Playback	Sounds are essential to app functionality and might mix with other audio.	No	No (default) Yes (when the Mix With Others property is added)	Yes
Record	Audio is user-recorded.	No	No	Yes
Play and Record	Sounds represent audio input and output, sequentially or simultaneously.	No	No (default) Yes (when the Mix With Others property is added)	Yes
Audio Processing	App performs hardware-assisted audio encoding (it does not play or record).	N/A	No	Yes *

* If you select the Audio Processing category and you want to perform audio processing in the background, you need to prevent your app from suspending before you're finished with the audio processing. To learn how to do this, see "Implementing Long-Running Background Tasks" in *iOS App Programming Guide*.

Here are some scenarios that illustrate how to choose the audio session category that provides an audio experience users appreciate.

Scenario 1: An educational app that helps people learn a new language. You provide:

- Feedback sounds that play when users tap specific controls
- Recordings of words and phrases that play when users want to hear examples of correct pronunciation

In this app, sound is essential to the primary functionality. People use this app to hear words and phrases in the language they're learning, so the sound should play even when the device locks or is switched to silent. Because users need to hear the sounds clearly, they expect other audio they might be playing to be silenced.

To produce the audio experience users expect for this app, you'd use the Playback category. Although this category can be refined to allow mixing with other audio, this app should use the default behavior to ensure that other audio does not compete with the educational content the user has explicitly chosen to hear.

Scenario 2: A Voice over Internet Protocol (VoIP) app. You provide:

- The ability to accept audio input
- The ability to play audio

In this app, sound is essential to the primary functionality. People use this app to communicate with others, often while they're currently using a different app. Users expect to be able to receive calls when they've switched their device to silent or the device is locked, and they expect other audio to be silent for the duration of a call. They also expect to be able to continue calls when the app is in the background.

To produce the expected user experience for this app, you'd use the Play and Record category, and you'd be sure to activate your audio session only when you need it so that users can use other audio between calls.

Scenario 3: A game that allows users to guide a character through different tasks. You provide:

- Various gameplay sound effects
- A musical soundtrack

In this app, sound greatly enhances the user experience, but isn't essential to the main task. Also, users are likely to appreciate being able to play the game silently or while listening to songs in their music library instead of to the game soundtrack.

The best strategy is to find out if users are listening to other audio when your app starts. Don't ask users to choose whether they want to listen to other audio or listen to your soundtrack. Instead, use the Audio Session Services function `AudioSessionGetProperty` to query the state of the `kAudioSessionProperty_OtherAudioIsPlaying` property. Based on the answer to this query, you can choose either the Ambient or Solo Ambient categories (both categories allow users to play the game silently):

- If users are listening to other audio, you should assume that they'd like to continue listening and wouldn't appreciate being forced to listen to the game soundtrack instead. In this situation, you'd choose the Ambient category.
- If users aren't listening to any other audio when your app starts, you'd choose the Solo Ambient category.

Scenario 4: An app that provides precise, real-time navigation instructions to the user's destination. You provide:

- Spoken directions for every step of the journey
- A few feedback sounds
- The ability for users to continue to listen to their own audio

In this app, the spoken navigation instructions represent the primary task, regardless of whether the app is in the background. For this reason, you'd use the Playback category, which allows your audio to play when the device is locked or switched to silent, and while the app is in the background.

To allow people to listen to other audio while they use your app, you can add the `kAudioSessionProperty_OverrideCategoryMixWithOthers` property. However, you also want to make sure that users can hear the spoken instructions above the audio they're currently playing. To do this, you can apply the `kAudioSessionProperty_OtherMixableAudioShouldDuck` property to the audio session to ensures that your audio is louder than all currently playing audio, with the exception of phone audio on iPhone. These settings allow the app to reactivate its audio session while the app is in the background, which ensures that users get navigation updates in real time.

Scenario 5: A blogging app that allows users to upload their text and graphics to a website. You provide:

- A short startup sound file
- Various short sound effects that accompany user actions (such as a sound that plays when a post has been uploaded)
- An alert sound that plays when a posting fails

In this app, sound enhances the user experience, but it's not essential. The main task has nothing to do with audio and users don't need to hear any sounds to successfully use the app. In this scenario, you'd use System Sound Services to produce sound. This is because the audio context of all sound in the app conforms to the intended purpose of this technology, which is to produce UI sound effects and alert sounds that obey device locking and the Ring/Silent (or Silent) switch in the way that users expect.

Manage Audio Interruptions

Sometimes, currently playing audio is interrupted by audio from a different app. On iPhone, for example, an incoming phone call interrupts the current app's audio for the duration of the call. In a multitasking environment, the frequency of such audio interruptions can be high.

To provide an audio experience users appreciate, iOS relies on you to:

- Identify the type of audio interruption your app can cause
- Respond appropriately when your app continues after an audio interruption ends

Every app needs to identify the type of audio interruption it can cause, but not every app needs to determine how to respond to the end of an audio interruption. This is because most types of apps should respond to the end of an audio interruption by resuming audio. Only apps that are primarily or partly media playback apps—and that provide media playback controls—have to take an extra step to determine the appropriate response.

Conceptually, there are two types of audio interruptions, based on the type of audio that's doing the interrupting and the way users expect the particular app to respond when the interruption ends:

- A **resumable interruption** is caused by audio that users view as a temporary interlude in their primary listening experience.

  After a resumable interruption ends, an app that displays controls for media playback should resume what it was doing when the interruption occurred, whether this is playing audio or remaining paused. An app that doesn't have media playback controls should resume playing audio.

  For example, consider a user listening to an app for music playback on iPhone when a VoIP call arrives in the middle of a song. The user answers the call, expecting the playback app to be silent while they talk. After the call ends, the user expects the playback app to automatically resume playing the song, because the music—not the call—constitutes their primary listening experience *and* they had not paused the music before the call arrived. On the other hand, if the user had paused music playback before the call arrived, they would expect the music to remain paused after the call ends.

  Other examples of apps that can cause resumable interruptions are apps that play alarms, audio prompts (such as spoken driving directions), or other intermittent audio.

- A **nonresumable interruption** is caused by audio that users view as a primary listening experience, such as audio from a media playback app.

  After a nonresumable interruption ends, an app that displays media playback controls should not resume playing audio. An app that doesn't have media playback controls should resume playing audio.

  For example, consider a user listening to a music playback app (music app 1) when a different music playback app (music app 2) interrupts. In response, the user decides to listen to music app 2 for some period of time. After quitting music app 2, the user wouldn't expect music app 1 to automatically resume playing because they'd deliberately made music app 2 their primary listening experience.

The following guidelines help you decide what information to supply and how to continue after an audio interruption ends.

Identify the type of audio interruption your app caused. You do this by deactivating your audio session in one of the following two ways when your audio is finished:

- If your app caused a resumable interruption, deactivate your audio session with the `AVAudioSessionSetActiveFlags_NotifyOthersOnDeactivation` flag.
- If your app caused a nonresumable interruption, deactivate your audio session without any flags.

Providing, or not providing, the flags allows iOS to give interrupted apps the ability to resume playing their audio automatically, if appropriate.

Determine whether you should resume audio when an audio interruption ends. You base this decision on the audio user experience you provide in your app.

- If your app displays media playback controls that people use to play or pause audio, you need to check the `AVAudioSessionInterruptionFlags_ShouldResume` flag when an audio interruption ends.

  If your app receives the Should Resume flag, your app should:

  - Resume playing audio if your app was actively playing audio when it was interrupted
  - Not resume playing audio if your app was *not* actively playing audio when it was interrupted

- If your app doesn't display any media playback controls that people can use to play or pause audio, your app should always resume previously playing audio when an audio interruption ends, regardless of whether the Should Resume flag is present.

  For example, a game that plays a soundtrack should automatically resume playing the soundtrack after an interruption.

Handle Media Remote Control Events, if Appropriate

Apps can receive remote control events when people use iOS media controls or accessory controls, such as headset controls. This allows your app to accept user input that doesn't come through your UI, whether your app is currently playing audio in the foreground or in the background.

Apps can send video to AirPlay-enabled hardware—such as Apple TV—and transition to the background while playback continues. Such an app can accept user input via remote control events, so that users can control video playback while the app is in the background. In addition, this type of app can also reactivate an audio session after an interruption while it's in the background.

A media playback app, in particular, needs to respond appropriately to media remote control events, especially if it plays audio or video while it's in the background.

To meet the responsibilities associated with the privilege of playing media while your app is in the background, be sure to follow these guidelines:

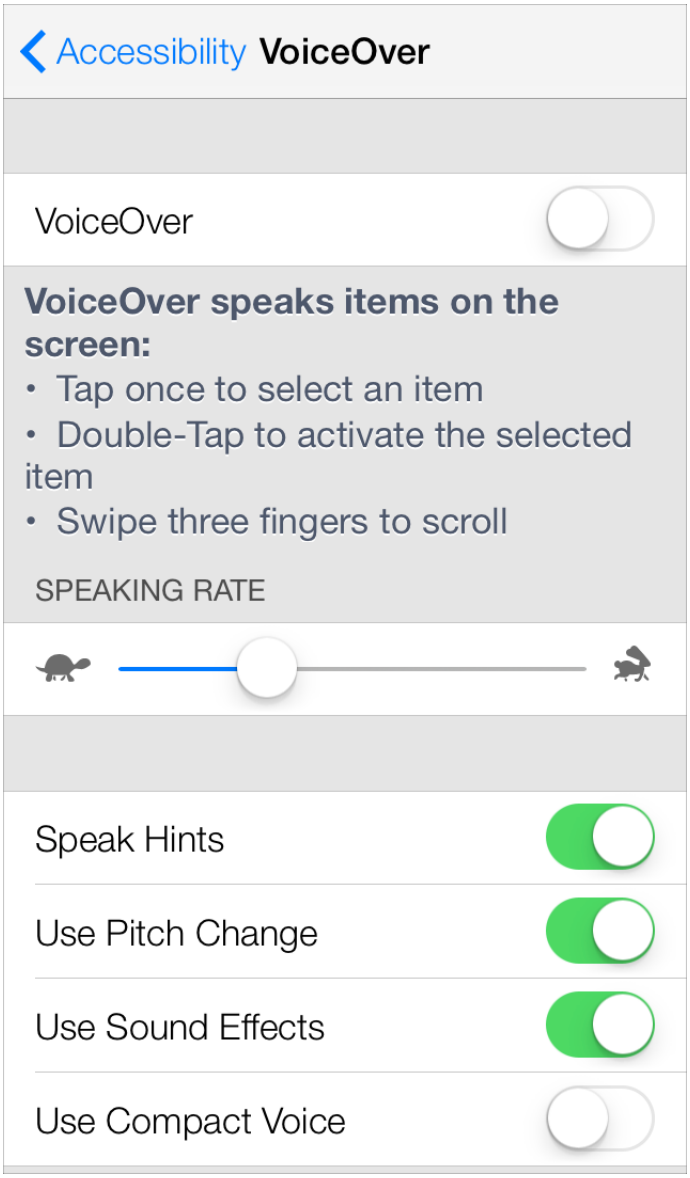
Limit your app's eligibility to receive remote control events to times when it makes sense. For example, if your app helps users read content, search for information, and listen to audio, it should accept remote control events only while the user is in the audio context. When the user leaves the audio context, you should relinquish the ability to receive the events. If your app lets users play audio or video on an AirPlay-enabled device, it should accept remote control events for the duration of media playback. Following these guidelines allows users to consume a different app's media—and control it with headset controls—when they're in the nonmedia contexts of your app.

As much as possible, use system-provided controls to offer AirPlay support. When you use the `MPMoviePlayerController` class to enable AirPlay playback, you can take advantage of a standard control that allows users to choose an AirPlay-enabled device that is currently in range. Or you can use the `MPVolumeView` class to display AirPlay-enabled audio or video devices from which users can choose. Users are accustomed to the appearance and behavior of these standard controls, so they'll know how to use them in your app.

Don't repurpose an event, even if the event has no meaning in your app. Users expect the iOS media controls and accessory controls to function consistently in all apps. You do not have to handle the events that your app doesn't need, but the events that you do handle must result in the experience users expect. If you redefine the meaning of an event, you confuse users and risk leading them into an unknown state from which they can't escape without quitting your app.

VoiceOver and Accessibility

VoiceOver increases accessibility for blind and low-vision users, and for users with certain learning challenges.



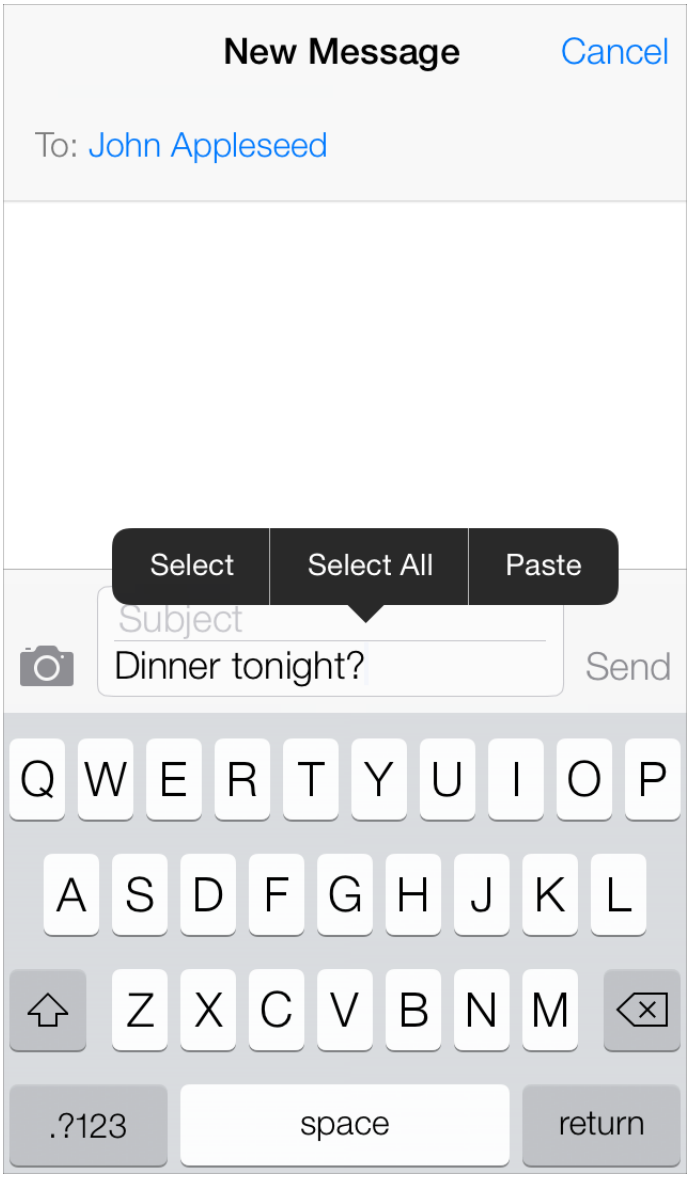
To make sure VoiceOver users can use your app, you might need to supply some descriptive information about the views and controls in your user interface. Supporting VoiceOver does *not* require you to change the visual design of your UI in any way.

When you use standard UI elements in a completely standard way, you have little (if any) additional work to do. The more custom your UI is, the more custom information you need to provide so that VoiceOver can accurately describe your app.

Making your iOS app accessible to VoiceOver users can increase your user base and help you enter new markets. Supporting VoiceOver can also help you address accessibility guidelines created by various governing bodies.

Edit Menu

Users can reveal an edit menu to perform operations such as Cut, Paste, and Select in a text view, web view, or image view.



You can adjust some of the behaviors of the menu to give users more control over the content in your app. For example, you can:

- Specify which of the standard menu commands are appropriate for the current context
- Determine the position of the menu before it appears so that you can prevent important parts of your app's UI from being obscured
- Define the object that is selected by default when users double-tap to reveal the menu

You can't change the color or shape of the menu itself.

For information on how to implement these behaviors in code, see "Copy and Paste Operations" in *iOS App Programming Guide*.

To ensure that the edit menu behaves as users expect in your app, you should:

Display commands that make sense in the current context. For example, if nothing is selected, the menu should not contain Copy or Cut because these commands act on a selection. Similarly, if something is selected, the menu should not contain Select. If you support an edit menu in a custom view, you're responsible for making sure that the commands the menu displays are appropriate for the current context.

Accommodate the menu display in your layout. iOS displays the edit menu above or below the insertion point or selection, depending on available space, and places the menu pointer so that users can see how the menu commands relate to the content. You can programmatically determine the position of the menu before it appears so that you can prevent important parts of your UI from being obscured, if necessary.

Support both gestures that people can use to invoke the menu. Although the touch and hold gesture is the primary way users reveal the edit menu, they can also double-tap a word in a text view to select the word and reveal the menu at the same time. If you support the menu in a custom view, be sure to respond to both gestures. In addition, you can define the object that is selected by default when the user double taps.

Avoid creating a button in your UI that performs a command that's available in the edit menu. For example, it's better to allow users to perform a copy operation using the edit menu than to provide a Copy button, because users will wonder why there are two ways to do the same thing in your app.

Consider enabling the selection of static text if it's useful to the user. For example, a user might want to copy the caption of an image, but they're not likely to want to copy the label of a tab item or a screen title, such as Accounts. In a text view, selection by word should be the default.

Don't make button titles selectable. A selectable button title makes it difficult for users to reveal the edit menu without activating the button. In general, elements that behave as buttons don't need to be selectable.

Combine support for undo and redo with your support of copy and paste. People often expect to be able to undo recent operations if they change their minds. Because the edit menu doesn't require confirmation before its actions are performed, you should give users the opportunity to undo or redo these actions.

> **Note:** If you need to enable actions that use the selected text or object in a way that's external to the current context, it's better to use an action sheet. For example, if you want to allow people to share their selection with others, you might display an action sheet that lists social networking sites to allow the actions of selecting and sending to a particular site. To learn about the usage guidelines for action sheets, see "Action Sheet" (page 183).

Follow these guidelines if you need to create custom edit menu items.

Create edit menu items that edit, alter, or otherwise act directly upon the user's selection. People expect the standard edit menu items to act upon text or objects within the current context, and it's best when your custom menu items behave similarly.

List custom items together after all system-provided items. Don't intersperse your custom items with the system-provided ones.

Keep the number of custom menu items reasonable. You don't want to overwhelm your users with too many choices.

Use succinct names for your custom menu items and make sure the names precisely describe what the commands do. In general, item names should be verbs that describe the action to be performed. Although you should generally use a single capitalized word for an item name, use title-style capitalization if you must use a short phrase. (Briefly, title-style capitalization means to capitalize every word except articles, coordinating conjunctions, and prepositions of four or fewer letters.)

Undo and Redo

Users initiate an Undo operation by shaking the device, which displays an alert that allows them to:

- Undo what they just typed
- Redo previously undone typing
- Cancel the undo operation

You can support the Undo operation in a more general way in your app by specifying:

- The actions users can undo or redo
- The circumstances under which your app should interpret a shake event as the shake-to-undo gesture
- How many levels of undo to support

To learn how to implement this behavior in code, see *Undo Architecture*. If you support undo and redo in your app, follow these guidelines to provide a good user experience.

Supply brief descriptive phrases that tell users precisely what they're undoing or redoing. iOS automatically supplies the strings "Undo " and "Redo " (including a space after the word) for the undo alert button titles, but you need to provide a word or two that describes the action users can undo or redo. For example, you might supply the text Delete Name or Address Change, to create button titles such as "Undo Delete Name" or "Redo Address Change." (Note that the Cancel button in the alert cannot be changed or removed.)

Avoid supplying text that is too long. A button title that is too long is truncated and is difficult for users to decipher. And because this text is in a button title, use title-style capitalization and do not add punctuation.

Avoid overloading the shake gesture. Even though you can programmatically set when your app interprets a shake event as shake to undo, you run the risk of confusing users if they also use shake to perform a different action. Analyze user interaction in your app and avoid creating situations in which users can't reliably predict the result of the shake gesture.

Use the system-provided Undo and Redo buttons only if undo and redo are fundamental tasks in your app. Remember that the shake gesture is the primary way users initiate undo and redo, and that it can be confusing to offer two different ways to perform the same task. If you decide it's important to provide explicit, dedicated controls for undo and redo, you can place the system-provided buttons in the navigation bar. (To learn more about these buttons, see "Toolbar and Navigation Bar Buttons" (page 123).)

Clearly relate undo and redo capability to the user's immediate context, and not to an earlier context. Consider the context of the actions you allow to be undone or redone. In general, users expect their changes and actions to take effect immediately.

Keyboards and Input Views

If appropriate, you can design a custom input view to replace the system-provided onscreen keyboard.

If you provide a custom input view, be sure its function is obvious to people. Also, be sure to make the controls in your input view look tappable.

You can also provide a custom input accessory view, which is a separate view that appears above the keyboard (or your custom input view).

Use the standard keyboard click sound to provide audible feedback when people tap the custom controls in your input view. To learn how to enable this sound in your code, see the documentation for `playInputClick` in *UIDevice Class Reference*.

> **Note:** The standard click sound is available only for custom input views that are currently onscreen. People can turn off all keyboard clicks (including ones that come from your custom input view) in Settings > Sounds.

UI Elements

Bars

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

The Status Bar

The **status bar** displays important information about the device and the current environment.

Default status bar

Light-content status bar

You can set the style of the status bar globally for the entire app or you can let individual view controllers set the style as appropriate. To learn more, read *UIApplication Class Reference* for information about the `UIStatusBarStyle` constant and *UIViewController Class Reference* for information about the `preferredStatusBarStyle` property.

Appearance and Behavior

The status bar is transparent. In all orientations, the status bar appears at the upper edge of the device screen and contains information people need, such as the network connection, the time of day, and the battery charge.

Guidelines

Although you don't use the status bar in the same way that you use other UI elements, it's important to understand its function in your app.

Think twice before hiding the status bar. Because the status bar is transparent, it's not usually necessary to hide it. Permanently hiding the status bar means that users must switch away from your app to get the time or find out whether they have a Wi-Fi connection.

Consider hiding the status bar—and all other app UI—while people are actively viewing full-screen media. If you do this, be sure to allow people to retrieve the status bar and appropriate app UI with a single tap. Unless you have a very compelling reason to do so, it's best to avoid defining a custom gesture to redisplay the status bar because users are unlikely to discover such a gesture or remember it.

Don't create a custom status bar. Users depend on the consistency of the system-provided status bar. Although you might hide the status bar in your app, it's not appropriate to create custom UI that takes its place.

Choose a status bar content color that coordinates with your app. The default appearance displays dark content, which looks good on top of light-colored app content. The light status bar content looks good on top of dark-colored app content.

As much as possible, avoid putting distracting content behind the status bar. In particular, you don't want to imply that users should tap the status bar to access content or activate controls in your app.

When appropriate, display the network activity indicator. The network activity indicator can appear in the status bar to show users that lengthy network access is occurring. To learn how to implement this indicator in your code, see "Network Activity Indicator" (page 168).

Navigation Bar

A **navigation bar** enables navigation through an information hierarchy and, optionally, management of screen contents.

A navigation bar is contained in a navigation controller, which is a programmatic object that manages the display of a hierarchy of custom views. To learn more about defining a navigation bar in your code, see "Navigation Controllers" and "Navigation Bars".

Appearance and Behavior

A navigation bar generally appears at the upper edge of an app screen, just below the status bar. A navigation bar can display the title of the current screen or view, centered along its length. When navigating through a hierarchy of information, users tap the Back button—or swipe from the edge of the device—to return to the previous screen. Otherwise, users can tap content-specific controls in the navigation bar to manage the contents of the screen.

A navigation bar is translucent and all controls in the bar are borderless.

On iPhone, a navigation bar always displays across the full width of the screen, and changing the device orientation can change the height of the navigation bar automatically.

Guidelines

You can use a navigation bar to enable navigation among different views, or provide controls that manage the items in a view.

When it adds value, use the title of the current view as the title of the navigation bar. If titling a navigation bar seems redundant, you can leave the title empty.

When the user navigates to a new level, two things should happen:

- The bar title should change to the new level's title, if appropriate.
- A back button should appear to the left of the title, and it should be labeled with the previous level's title.

Make sure it's easy to read the text in the navigation bar. The system font provides maximum readability, but you can specify a different font, if appropriate.

Consider putting a segmented control in a navigation bar at the top level of an app. This is especially useful if doing so helps to flatten your information hierarchy and make it easier for people to find what they're looking for. If you use a segmented control in a navigation bar, be sure to choose accurate back-button titles. (For usage guidelines, see "Segmented Control" (page 173).)

Avoid crowding a navigation bar with additional controls, even if there appears to be enough space. The navigation bar should contain no more than a view's current title, the Back button, and one control that manages the view's contents. If instead, you use a segmented control in the navigation bar, the bar shouldn't display a title and it shouldn't contain any controls other than the segmented control.

Use system-provided buttons according to their documented meaning. For more information, see "Toolbar and Navigation Bar Buttons" (page 123). If you decide to create your own navigation bar controls, see "Bar Button Icons" (page 198) for advice on how to design them.

If appropriate, customize the appearance of a navigation bar to coordinate with the look of your app. For example, you can supply a custom background image or tint for the bar and you can specify translucency. In some cases, it can be a good idea to supply a resizable background image; to learn more about creating a resizable image, see "Creating Resizable Images" (page 188). Take care to provide a background image of the right height in an iOS 7 app; for more information, see "Navigation Bar" in *iOS 7 UI Transition Guide*.

Make sure that the look of your customized navigation bar is consistent with your app's appearance and style. For example, don't combine an opaque navigation bar with a translucent toolbar. Also, it's best to avoid changing the image, color, or translucency of the navigation bar in different screens in the same orientation.

Make sure that a customized back button still looks like a back button. Users know that the standard back button allows them to retrace their steps through a hierarchy of information. If you decide to replace the system-provided chevron with a custom image, be sure to supply a custom mask image, too. iOS 7 uses the mask to make the button title appear to emerge from—or disappear into—the chevron during transitions.

On iPhone, be prepared for the change in navigation bar height that occurs on device rotation. In particular, make sure that your custom navigation bar icons fit well in the thinner bar that appears in landscape orientation. Don't specify the height of a navigation bar programmatically; instead, you can take advantage of the `UIBarMetrics` constants to ensure that your content fits well.

Toolbar

A **toolbar** contains controls that perform actions related to objects in the screen or view.

A toolbar is typically contained in a navigation controller, which is an object that manages the display of a hierarchy of custom views. To learn more about defining a toolbar in your code, see "Displaying a Navigation Toolbar" in *View Controller Catalog for iOS* and "Toolbar".

Appearance and Behavior

On iPhone, a toolbar always appears at the bottom edge of a screen or view, but on iPad it can instead appear at the top edge.

A toolbar is translucent and its items are displayed equally spaced across its width. The precise set of toolbar items can change from view to view, because the items are always specific to the context of the current view.

On iPhone, changing the device orientation from portrait to landscape can change the height of the toolbar automatically.

Guidelines

Use a toolbar to provide a set of actions users can take in the current context.

Use a toolbar to give people a selection of frequently used commands that make sense in the current context. An alternative is to put a segmented control in a toolbar to give people access to different perspectives on your app's data or to different app modes (for usage guidelines, see "Segmented Control" (page 173)).

Maintain a hit target area of at least 44 x 44 points for each toolbar item. If you crowd toolbar items too closely together, it's hard for people to tap the one they want.

Use system-provided toolbar items according to their documented meaning. See "Toolbar and Navigation Bar Buttons" (page 123) for more information. If you decide to create your own toolbar items, see "Bar Button Icons" (page 198) for advice on how to design them.

On iPhone, be prepared for the change in toolbar height that occurs on device rotation. In particular, make sure your custom toolbar icons fit well in the thinner bar that appears in landscape orientation. Don't specify the height of a toolbar programmatically; instead, you can take advantage of the `UIBarMetrics` constants to ensure that your content fits well.

Toolbar and Navigation Bar Buttons

iOS makes available many of the standard buttons users see in toolbars and navigation bars.

As with all system-provided buttons, you should avoid using the buttons described in Table 33-1 to represent actions other than those for which they are designed. In particular, avoid choosing a button based on its appearance, without regard for its documented meaning.

To find out which symbol names to use to specify these buttons, see the documentation for `UIBarButtonSystemItem` in *UIBarButtonItem Class Reference*.

Table 33-1 Standard buttons available for toolbars and navigation bars

Button	Name	Meaning
	Share	Open an action sheet that lists system-provided and app-specific services that act on the specified content
	Camera	Open an action sheet that displays a photo picker in camera mode

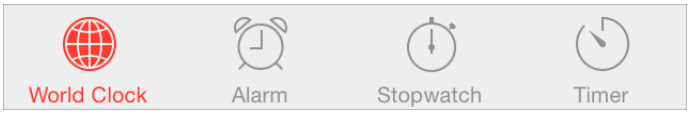
Button	Name	Meaning
	AirPlay	Open an action sheet that displays nearby AirPlay enabled devices.
	Locate	Use Location Services to display the user's current location
	Compose	Open a new message view in edit mode
	Bookmarks	Show app-specific bookmarks
	Search	Display a search field
	Add	Create a new item
	Trash	Delete current item
	Organize	Move or route an item to a destination within the app, such as a folder
	Reply	Send or route an item to another location
	Refresh	Refresh contents (use only when necessary; otherwise, refresh automatically)

In addition to the buttons shown in Table 33-1, you can also use the system-provided Edit, Cancel, Save, Done, Redo, and Undo buttons to support editing or other types of content manipulation in your app. The appearance of each of these buttons is provided by its text title. To find out which symbol names to use to specify these buttons, see the documentation for `UIBarButtonSystemItem` in *UIBarButtonItem Class Reference*.

In addition, you can use the system-provided Info button in a toolbar:

Tab Bar

A **tab bar** gives people the ability to switch between different subtasks, views, or modes.



A tab bar is contained in a tab bar controller, which is an object that manages the display of a set of custom views. To learn more about defining a tab bar in your code, see “Tab Bar Controllers” and “Tab Bars”.

Appearance and Behavior

A tab bar appears at the bottom edge of the screen and should be accessible from every location in the app. A tab bar is translucent and it displays icons and text in tabs, all of which are equal in width. When users select a tab, the icon receives the appropriate tint color.

On iPhone, a tab bar can display no more than five tabs at one time; if the app has more tabs, the tab bar displays four of them and adds the More tab, which reveals the additional tabs in a list. On iPad, a tab bar can display more than five tabs.

A tab can display a badge—which is a red oval that contains white text and either a number or exclamation point—that communicates app-specific information.

A tab bar does not change its height when the device changes its orientation.

Guidelines

Use a tab bar to give users access to different perspectives on the same set of data or different subtasks related to the overall function of your app. When you use a tab bar, follow these guidelines:

Don’t use a tab bar to give users controls that act on elements in the current mode or screen. If you need to provide controls for your users, use a toolbar instead (for usage guidelines, see “Toolbar” (page 122)).

In general, use a tab bar to organize information at the app level. A tab bar is well-suited for use in the main app view because it’s a good way to flatten your information hierarchy and provide access to several peer information categories or modes at one time.

Don’t remove a tab when its function is unavailable. If a tab represents a part of your app that is unavailable in the current context, it’s better to display a disabled tab than to remove the tab altogether. If you remove a tab in some cases but not in others, you make your app’s UI unstable and unpredictable. The best solution is to ensure that all tabs are enabled, but explain why a tab’s content is unavailable. For example, if the user doesn’t have any songs on an iOS device, the Songs tab in the Music app displays a screen that explains how to download songs.

Consider badging a tab bar icon to communicate unobtrusively. You can display a badge on a tab bar icon to indicate that there is new information associated with that view or mode.

Use system-provided tab bar icons according to their documented meaning. For more information, see “Tab Bar Icons” (page 126). If you decide to create your own tab bar icons, see “Bar Button Icons” (page 198) for advice on how to design them.

If appropriate, customize the appearance of a tab bar. For example, you can supply a custom tint for the tab bar and its icons, as long as the icons are either system-provided or custom template images. You can also supply a background image for the tab bar (note that it's often a good idea to supply a resizable background image; to learn more about creating a resizable image, see "Creating Resizable Images" (page 188)).

On iPad, you might use a tab bar in a split view pane or a popover if the tabs switch or filter the content within that view. However, it often works better to use a segmented control at the bottom edge of a popover or split view pane, because the appearance of a segmented control coordinates better with the popover or split view appearance. (For more information on using a segmented control, see "Segmented Control" (page 173).)

On iPad, avoid crowding the tab bar with too many tabs. Putting too many tabs in a tab bar can make it physically difficult for people to tap the one they want. Also, with each additional tab you display, you increase the complexity of your app. In general, try to limit the number of tabs in the main view or in the right pane of a split view to about seven. In a popover or in the left pane of a split view, up to about five tabs fit well.

On iPad, avoid creating a More tab. In an iPad app, a screen devoted solely to a list of additional tabs is a poor use of space.

On iPad, display the same tabs in each orientation to increase the visual stability of your app. In portrait, the recommended seven tabs fit well across the width of the screen. In landscape orientation, you should center the same tabs along the width of the screen. This guidance also applies to the usage of a tab bar within a split view pane or a popover. For example, if you use a tab bar in a popover in portrait, it works well to display the same tabs in the left pane of a split view in landscape.

Tab Bar Icons

iOS provides the standard icons described in Table 33-2 for use in tab bars.

To find out which symbol names to use to specify these icons, see the documentation for `UITabBarSystemItem` in *UITabBarItem Class Reference*.

Table 33-2 Standard icons for use in the tabs of a tab bar

Icon	Name	Meaning
	Bookmarks	Show app-specific bookmarks
	Contacts	Show contacts
	Downloads	Show downloads

Icon	Name	Meaning
	Favorites	Show user-determined favorites
	Featured	Show content featured by the app
	History	Show history of user actions
	More	Show additional tab bar items
	MostRecent	Show the most recent item
	MostViewed	Show items most popular with all users
	Recents	Show the items accessed by the user within an app-defined period
	Search	Enter a search mode
	TopRated	Show the highest-rated items, as determined by the user

As with all standard buttons and icons, it's essential to use the tab bar icons in accordance with their documented meanings. In particular, take care to base your usage of an icon on its semantic meaning, not its appearance. This will help your app's UI make sense even if the icon associated with a specific meaning changes its appearance.

Search Bar

A **search bar** accepts text from users, which can be used as input for a search (shown here with placeholder text).

To learn how to define a search bar in your code, see "Search Bars".

Appearance and Behavior

A search bar looks similar to a text field. By default, a search bar displays the search icon on the left side. When the user taps a search bar, a keyboard appears; when the user is finished typing search terms, the input is handled in an app-specific way.

In addition, a search bar can display a few optional elements, such as:

- **Placeholder** text. This text might state the function of the control—for example, "Search"—or remind users in what context they are searching—for example, "Google".
- The **Bookmarks** button. This button can provide a shortcut to information users want to easily find again. For example, the Bookmarks button in the Maps search mode gives access to bookmarked locations, recent searches, and contacts.

  The Bookmarks button is visible only when there is no user-supplied or nonplaceholder text in the search bar. When the search bar contains such text, the Clear button appears so that users can erase the text.
- The **Clear** button. Most search bars include a Clear button that lets users to erase the contents of the search bar with one tap.

  When the search bar contains any nonplaceholder text, the Clear button is visible so users can erase the text. If there is no user-supplied or nonplaceholder text in the search bar, the Clear button is hidden.
- The **results list** icon. This icon indicates the presence of search results. When users tap the results list icon, an app can display the results of their most recent search.
- A descriptive title, called a **prompt**, that appears above the search bar. For example, a prompt can be a short phrase that provides introductory or app-specific context for the search bar.

Guidelines

Use a search bar to enable search in your app. Don't use a text field to enable search because it doesn't have the standard search-bar appearance that users expect.

You can customize a search bar by tinting it or specifying a custom background appearance and by providing custom accessory images. In iOS 7, you can put a search bar in a navigation bar (for more information, see `UISearchDisplayController`).

If you decide to supply a background image, it can be a good idea to supply a resizable image; to learn more about creating one, see "Creating Resizable Images" (page 188).

Scope Bar

A **scope bar**—which is available only in conjunction with a search bar—allows users to define the scope of a search.

To learn more about defining a search bar and scope bar in your code, see “Search Bars”.

Appearance and Behavior

When a search bar is present, a scope bar can appear near it. Regardless of orientation, a scope bar displays below a search bar, unless you use a search display controller in your code (for more information on the way this works, see *UISearchDisplayController Class Reference*). When you use a search display controller, the scope bar is displayed within the search bar to the right of the search field when the device is in landscape orientation and below the search bar when the device is in portrait.

Guidelines

It can be useful to display a scope bar when there are clearly defined or typical categories in which users might want to search. For example, users often want to narrow their search to one field in an email message.

You can customize a scope bar by supplying a background image. In addition, you can define different appearances for the enabled and disabled states of the scope bar buttons and the dividers between them.

Content Views

Activity

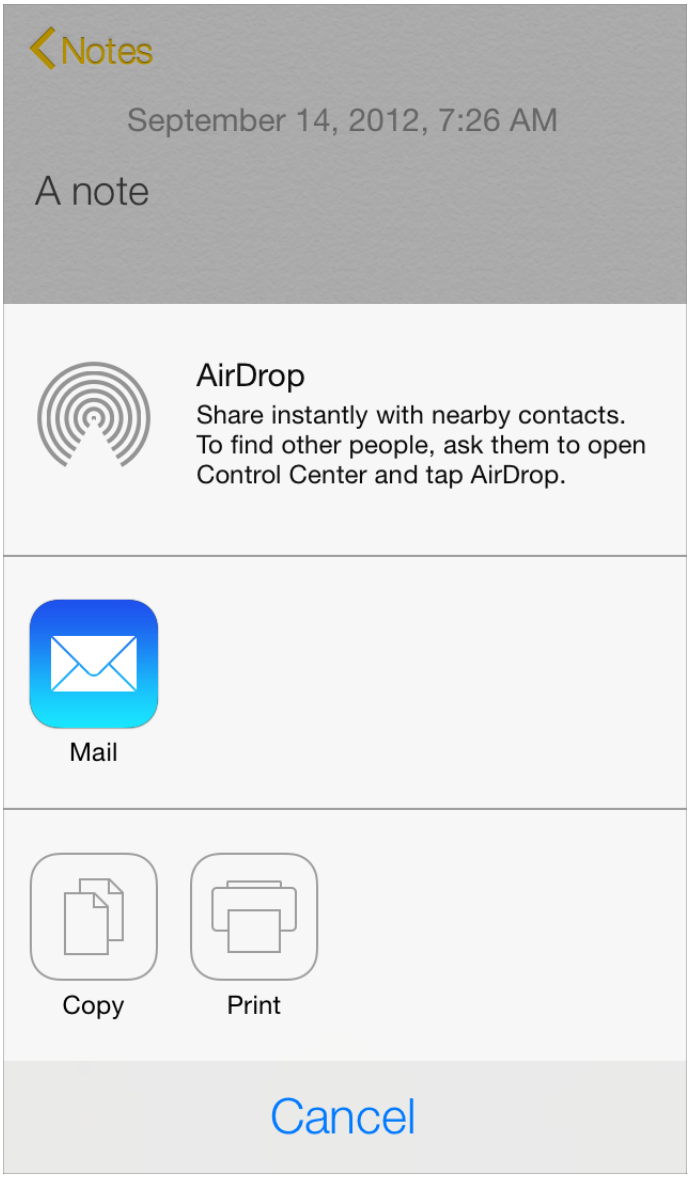
An **activity** represents a system-provided or custom service—accessible via an activity view controller—that can act on some specified content.

To learn more about defining an activity in your code, see *UIActivity Class Reference*.

Appearance and Behavior

An **activity** is a customizable object that represents a service that an app can perform while users are in the app. When users tap the Share button, a set of activities is presented by an activity view controller (to learn how incorporate an activity view controller into your app, see "Activity View Controller" (page 133)).

Each activity is represented by an icon and a title that appears below the icon. System-provided activities can use either of two icon styles: a style that looks like an app icon or a style that looks similar to a bar button icon. A third-party activity is always represented by an icon that uses the second style. You can see both icon styles in the activity view controller shown below.



Users initiate a service by tapping its activity icon in the activity view controller. In response, the activity either performs the service immediately or—if the service is complicated—it can request more information before performing the service.

Guidelines

Use an activity to give users access to a custom service that your app can perform. Note that iOS provides several built-in services, such as Print, Twitter, Message, and AirPlay. You don't need to create a custom activity that performs a built-in service.

Create a streamlined template image that represents your service. A template image is an image that iOS uses as a mask to create the final image that users see. To create a template image that looks good in the final icon, follow these guidelines:

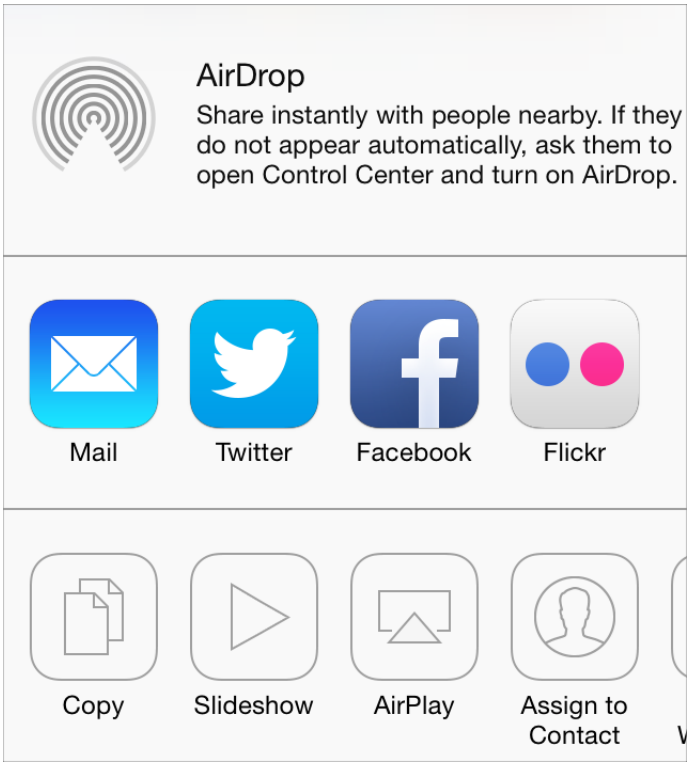
- Use black or white with appropriate alpha transparency.
- Don't include a drop shadow.
- Use anti-aliasing.

An activity template image should be centered in an area that measures about 70 x 70 pixels (high resolution).

Craft an activity title that succinctly describes your service. The title is displayed below the activity's icon in the activity view controller. A short title is best, because it looks better onscreen and it's easier to localize. If your title is too long, iOS first shrinks the text and then—if the title is still too long—truncates it. In general, it's a good idea to avoid including your company or product name in the activity title.

Activity View Controller

An **activity view controller** presents a transient view that lists system-provided and custom services that can act on some specified content.



To learn more about defining an activity view controller in your code, see *UIActivityViewController Class Reference*.

Appearance and Behavior

An activity view controller displays a configurable list of services that the user can perform on the specified content. Users tap the Share button to reveal the contents of an activity view controller.

An activity view controller works with a set of activities, each of which represents a specific service. To learn how to design an activity to provide a custom service, see "Activity" (page 130).

On iPhone and iPod touch, an activity view controller appears in an action sheet; on iPad, it appears in a popover.

Guidelines

Use an activity view controller to give people a list of services they can perform on content that is specified in some way. The services can be system-provided—such as Copy, Twitter, and Print—or custom. A common way to use an activity view controller is to allow users to post content they've selected to a social media account.

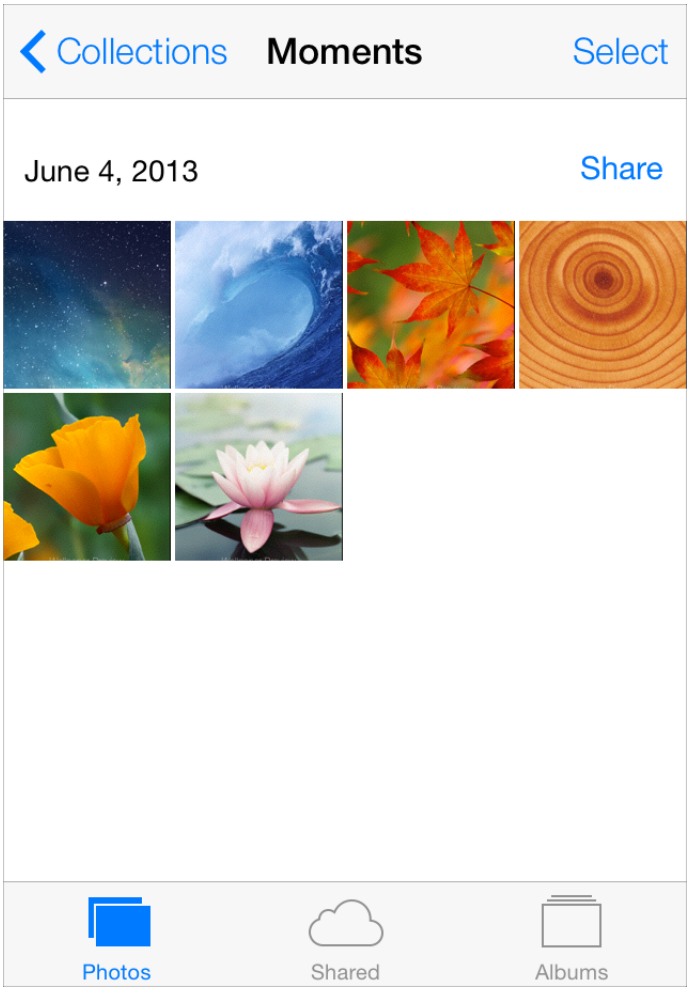
Don't create a custom button that reveals an activity view controller. People are accustomed to accessing system-provided services when they tap the Share button. You want to take advantage of this learned behavior and avoid confusing users by providing an alternate way to do the same thing.

Ensure that the listed services are appropriate in the current context. You can change the services listed in an activity view controller by specifying system-provided services to exclude and by identifying custom services to include. For example, if you wanted to prevent users from printing an image, you would exclude the Print activity from the activity view controller.

Note: You can't change the order in which the system-provided services are listed in an activity view controller. Also, all system-provided services appear before any custom services.

Collection View

A **collection view** manages an ordered collection of items and presents them in a customizable layout.



To learn more about defining a collection view in your code, see *Collection View Programming Guide for iOS*.

Appearance and Behavior

A collection view is a customizable scrolling view that displays a set of items your app provides. People use gestures to interact with the items in a collection view and they can modify the collection by inserting, moving, and deleting items.

A collection view works with several other objects in your code to define the overall layout and appearance of items. Chief among these objects is the layout object—that is, a standard or custom subclass of `UICollectionViewLayout`—which specifies the placement and visual attributes of the individual items in the collection. For convenience, UIKit provides the `UICollectionViewFlowLayout` object, which defines an adjustable linear ordering that can display a grid of items.

Within a collection view, optional supplementary views can visually distinguish subsets of items. A collection view also supports optional decoration views that can provide custom background and other appearances.

When users insert, move, or delete items, a collection view animates the action by default. Collection views also support the addition of gesture recognizers to perform custom actions; by default, a collection view recognizes the tap and touch-and-hold gestures to handle item selection and editing, respectively.

In iOS 7, collection views support custom animated transitions between layouts. To learn more, see *UICollectionViewTransitionLayout Class Reference*.

Guidelines

Use a collection view to give users a way to view and manipulate a set of items that don't need to be displayed in a list. Because a collection view doesn't enforce a strictly linear layout, it's particularly well suited to display items that differ in size.

A collection view supports extensive customization, so it's essential to avoid getting distracted by the ability to create radical new designs. You want a collection view to enhance the user's task; you don't want a collection view to become the focus of the user experience. The following guidelines can help you create collection views that people appreciate.

Don't use a collection view when a table view is a better choice. In some cases, it's easier for people to view and understand information when it's presented in a list. For example, it can be simpler and more efficient for people to view and interact with textual information when it's in a scrolling list.

Make it easy for people to select an item. If it's hard for users to tap an item in your collection view, they're less likely to enjoy using your app. As with all UI objects that users might want to tap, ensure that the minimum target area for each item in a collection view is 44 x 44 points. If you're using a collection view in an iPhone app, it's especially important to make items easy to tap.

Use caution if you make dynamic layout changes. A collection view allows you to change the layout of items while users are viewing and interacting with them. If you decide to dynamically adjust a collection view's layout, be sure that the change makes sense and is easy for users to track. Changing a collection view's layout without an obvious motivation can give people the impression that your app is unpredictable and hard to use. And, if the current focus or context is lost during a dynamic layout change, users are likely to feel that they're no longer in control of your app.

Container View Controller

A **container view controller** manages and presents its set of child views—or view controllers—in a custom way. Examples of system-defined container view controllers are tab bar view controller and navigation view controller (you can learn more about these components in "Tab Bar" (page 124) and "Navigation Bar" (page 120)).

To learn more about defining a custom container view controller in your code, see *UIViewController Class Reference*.

Appearance and Behavior

As you might expect, a custom container view controller doesn't have any predefined appearance or behavior. When you subclass `UIViewController` to create a custom container view controller object, you decide how many child view controllers it contains and how they should be presented.

Guidelines

Use a container view controller to present content through which users navigate in a custom way.

Ask yourself whether a custom container view controller is really necessary. Users are comfortable with the appearance and behavior of standard container view controllers, such as split view and tab bar view. You need to be sure that the potential advantages of your custom container view outweigh the fact that users won't recognize it or instantly know how it works.

Make sure that your custom container view controller works in both orientations. You need to design a container view controller that gives users a consistent experience in both portrait and landscape.

In general, avoid flashy view transitions. When you use storyboards to design a custom view controller, it's easy to define custom animations for the transitions between content views. But in most cases, flamboyant view transitions distract people from their purpose and often decrease the aesthetic appeal of your app.

Image View

An **image view** displays one image or an animated series of images.

To learn more about defining an image view in your code, see "Image Views".

Appearance and Behavior

An image view doesn't have any predefined appearance and it doesn't enable user interaction by default. An image view examines properties of both the image and its parent view to determine whether the image should be stretched, scaled, sized to fit, or pinned to a specific location.

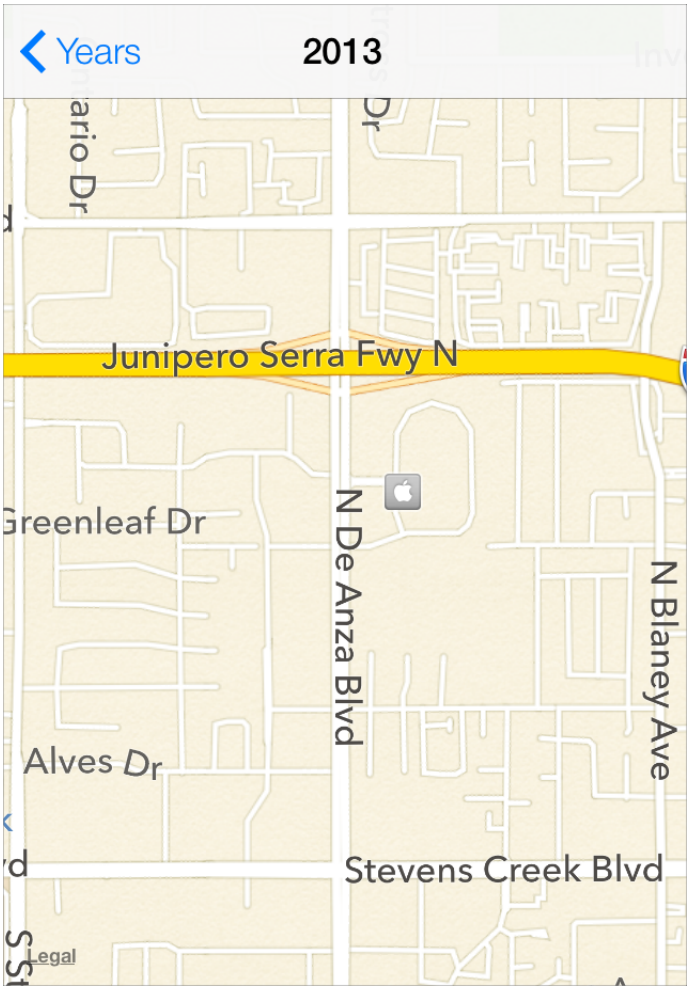
In iOS 7, an image view that contains a template image applies the current tint color to the image.

Guidelines

As much as possible, ensure that all images in an image view have the same size and use the same scale. If your images have different sizes, the image view will adjust them separately; if your images use different scale factors, they may render incorrectly.

Map View

A **map view** presents geographical data and supports most of the functionality provided by the built-in Maps app (shown below in Photos).



To learn more about defining a map view in your code, see *Map Kit Framework Reference*.

Appearance and Behavior

A map view displays a geographical area using standard map data, satellite imagery, or a combination of both. A map view can also display annotations—which mark single points—and overlays, which delineate paths or two-dimensional areas.

Users can zoom and pan a map view—unless you disallow these actions—and you can zoom and pan the map programmatically.

Guidelines

Use a map view to give users an interactive view of a geographical area. If you're developing a routing app, use a map view to display the user's route (for more information about creating a routing app, see "Routing" (page 76)).

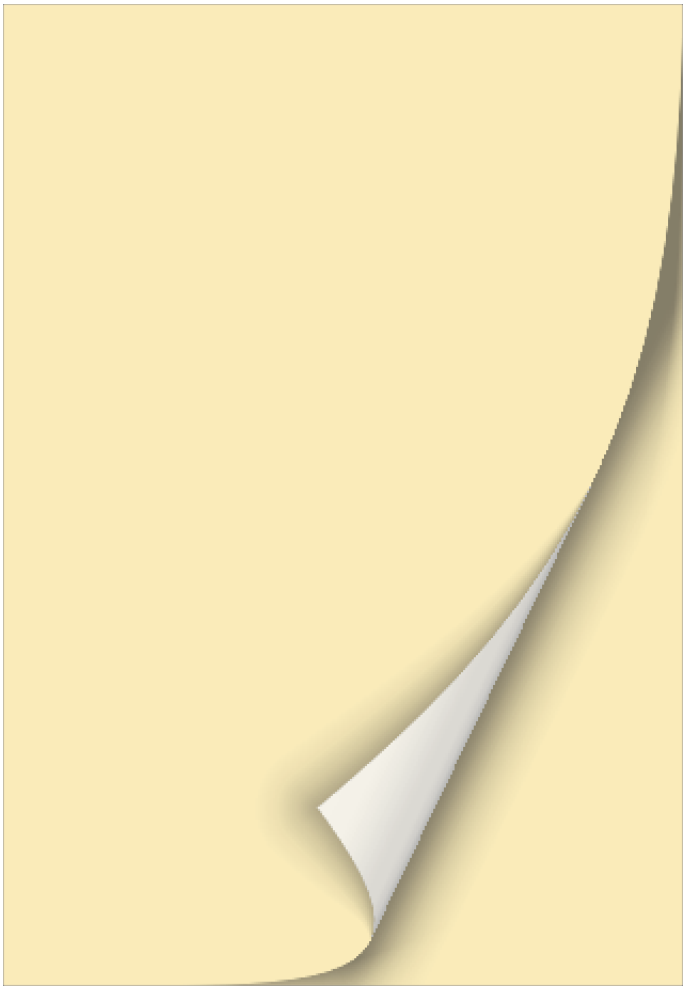
In general, let users interact with the map. People are accustomed to interacting with the built-in Maps app, and they expect to be able to interact with your map in similar ways.

Use the standard pin colors in a consistent way. A map pin shows the location of a point of interest in your map. People are familiar with the pin colors in the built-in Maps app, so it's best to avoid redefining the meaning of these colors in your app. When you use the standard pin colors, be sure to use them in the following ways:

- Red—a destination point
- Green—a starting point
- Purple—a user-specified point

Page View Controller

A **page view controller** manages multipage content using either a scrolling or page-curl transition style (the page-curl style as it appears in iOS 7 Simulator is shown below).



To learn more about defining a page view controller in your code, see "Page View Controllers".

Appearance and Behavior

A scroll-style page view controller has no default appearance. A page-curl-style page view controller can add the appearance of the inside of a book spine between pairs of pages and—while the user is turning a page—it displays a page-curl appearance.

A page view controller animates the transition from one page to another, according to the specified transition style. If the specified style is scroll, the current page scrolls to the next page; if the specified style is page-curl, the current page appears to turn like a page in a book or a notepad.

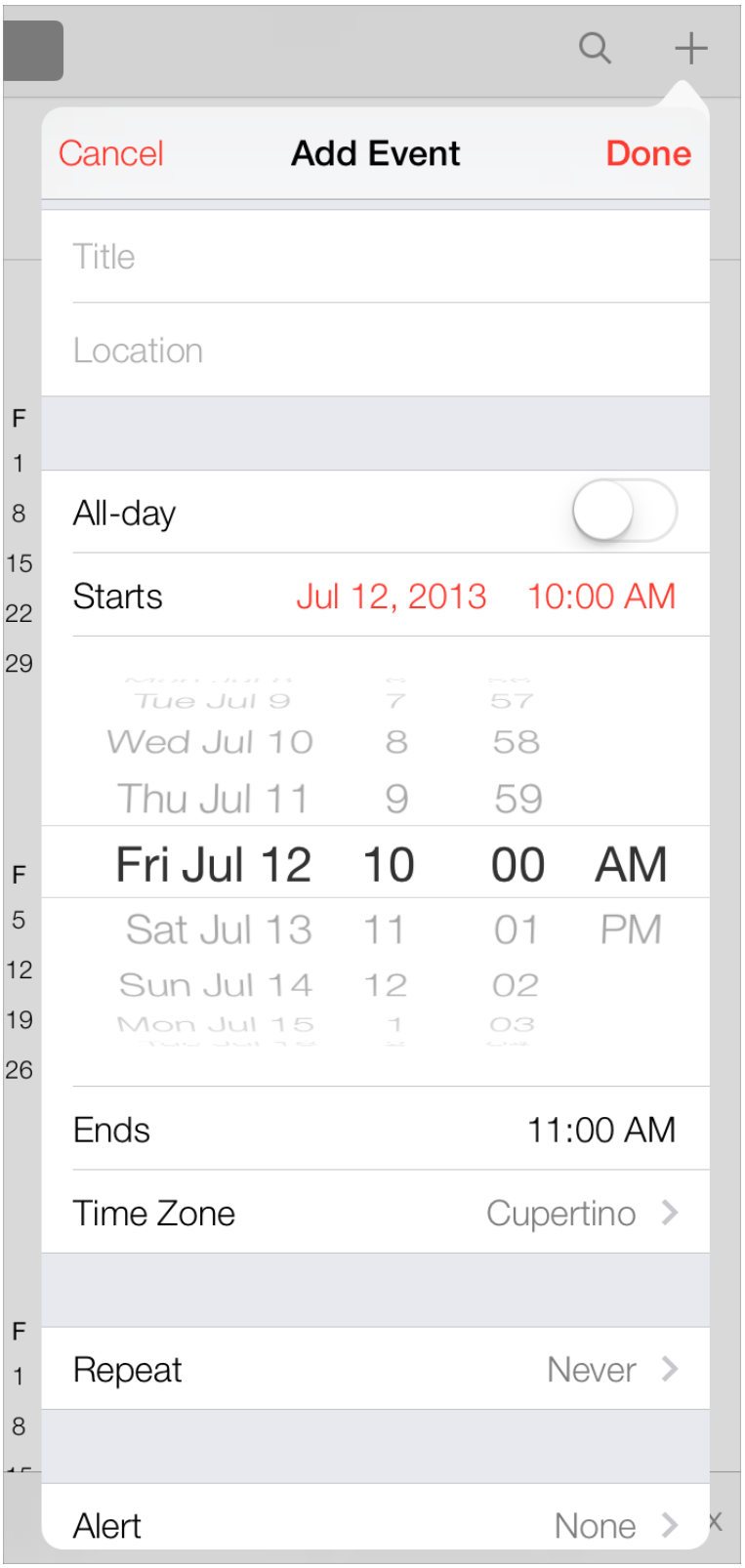
Guidelines

Use a page view controller to present content that users access in a linear fashion—such as the text of a story—or content that naturally breaks into chunks—such as a calendar.

A page view controller lets users move from one page to the next or previous page; it doesn't give users a way to jump among nonadjoining pages. If you want to use a page view controller to present content that users might access in a nonlinear fashion—such as a dictionary or the table of contents in a book—you must implement a custom way to let users move to different areas of the content.

Popover (iPad Only)

A **popover** is a transient view that can be revealed when people tap a control or an onscreen area.


Cancel
Add Event
Done
Title
Location
All-day
Starts
Jul 12, 2013
10:00 AM
Tue Jul 9
Wed Jul 10
Thu Jul 11
Fri Jul 12
Sat Jul 13
Sun Jul 14
Mon Jul 15
7
8
9
10
11
12
1
57
58
59
00
01
02
03
AM
PM
Ends
11:00 AM
Time Zone
Cupertino
Repeat
Never
Alert
None

To learn more about defining a popover in your code, see *UIPopoverController Class Reference*.

Important: Popovers are available in iPad apps only.

Appearance and Behavior

A popover is a self-contained view that hovers above the contents of a screen. It always displays an arrow that indicates the point from which it emerged. A popover can contain a wide variety of objects and views, such as:

- Table, image, map, text, web, or custom views
- Navigation bars, toolbars, or tab bars
- Controls or objects that act upon objects in the current app view

In iPad apps, an action sheet always appears inside a popover.

Guidelines

You can use a popover to:

- Display additional information or a list of items related to the focused or selected object.
- Display an action sheet that contains a short list of options that are closely related to something on the screen.
- Display the contents of the left pane when a split view–based app is in portrait. If you do this, you either provide an appropriately titled button that displays the popover—preferably in a navigation bar or toolbar at the top of the screen—or allow people to make the swipe gesture to hide and reveal it.

Avoid providing a "dismiss popover" button. A popover should close automatically when its presence is no longer necessary. To determine when a popover's presence is no longer necessary, consider the following scenarios:

- When a popover's only function is to provide a set of options or items that have an effect on the main view, it should close as soon as people make a choice. This behavior is very similar to that of a menu in a computer application. Note that this behavior also applies to a popover that contains only an action sheet: As soon as people tap a button in the action sheet, the popover should close.

  Occasionally, it can make sense to provide a popover that contains items that affect the main view, but that does *not* close when people make a choice. You might want to do this if you implement an inspector in a popover, because people might want to make an additional choice or change the attributes of the current choice.

A popover that provides menu or inspector functionality should close when people tap anywhere outside its bounds, including the control that reveals the popover. In a popover that provides a menu of choices, this gesture means that the user has decided not to make a choice (so the main view remains unaffected). In a popover that contains an action sheet, this gesture takes the place of tapping a Cancel button.

- If a popover enables a task, it can be appropriate to display buttons that complete or cancel the task and simultaneously dismiss the popover. In general, popovers that enable an editing task don't close when people tap outside of them; instead, they display a Done button and a Cancel button. These buttons help remind people that they're in an editing environment and allow them to explicitly keep or discard their input. When people tap either button, the popover should close.

  In general, you should prevent people from closing a task-enabling popover when they tap outside its borders, especially when it's important that people finish—or explicitly abandon—the task. Otherwise, you should save people's input when they tap outside a popover's borders, just as you would if they tapped Done.

In general, save users' work when they tap outside a popover's borders. Not every popover requires an explicit dismissal, so people might dismiss them mistakenly. You should discard the work people do in a popover only if they tap a Cancel button.

Make the popover arrow point as directly as possible to the element that revealed it. Doing this helps people remember where the popover came from and what task or object it's associated with.

Make sure people can use a popover without seeing the app content behind it. A popover obscures the content behind it, and people can't drag a popover to another location.

Ensure that only one popover is visible onscreen at a time. You shouldn't display more than one popover (or custom view designed to look and behave like a popover) at the same time. In particular, you should avoid displaying a cascade or hierarchy of popovers simultaneously, in which one popover emerges from another.

Don't display a modal view on top of a popover. Except for an alert, nothing should display on top of a popover.

When possible, allow people to close one popover and open a new one with one tap. This behavior is especially desirable when several different bar buttons each open a popover, because it prevents people from having to make extra taps.

Avoid making a popover too big. A popover shouldn't appear to take over the entire screen. Instead, it should be just big enough to display its contents and still point to the place it came from.

Ideally, the width of a popover should be at least 320 points, but no greater than 600 points. The height of a popover is not constrained, so you can use it to display a long list of items. In general, though, you should try to avoid scrolling in a popover that enables a task or that presents an action sheet. Note that the system might adjust both the height and the width of a popover to ensure that it fits well on the screen.

Generally, use standard UI controls and views within a popover. In general, popovers look best, and are easier for users to understand, when they contain standard controls and views.

If appropriate, customize the appearance of a popover. Although it's easy to customize many of the visual aspects of a popover by using the `UIPopoverBackgroundView` APIs, it's important to avoid creating a design that people might not recognize as a popover. If you change the appearance of a popover too much, users can't rely on their prior experience to help them understand how to use it in your app. If you decide to supply a resizable background image, see "Creating Resizable Images" (page 188) for more information on how to do this. If it looks good in your app, you can supply a translucent background image to take advantage of the popover's translucency.

If appropriate, change a popover's size while it remains visible. You might want to change a popover's size if you use it to display both a minimal and an expanded view of the same information. If you adjust the size of a popover while it's visible, you can choose to animate the change. Animating the change is usually a good idea because it avoids making people think that a new popover has replaced the old one.

Scroll View

A **scroll view** helps people see content that is larger than the scroll view's boundaries (the image shown below is both taller and wider than the scroll view that contains it).

To learn more about defining a scroll view in your code, see "Scroll Views".

Appearance and Behavior

When a scroll view first appears—or when users interact with it—vertical or horizontal scroll indicators flash briefly to show users that there is more content they can reveal. Other than the transient scroll indicators, a scroll view has no predefined appearance.

A scroll view responds to the speed and direction of gestures to reveal content in a way that feels natural to people. When users drag content in a scroll view, the content follows the touch; when users flick content, the scroll view reveals the content quickly and stops scrolling when the user touches the screen or when the end of the content is reached. A scroll view can also operate in paging mode, in which each drag or flick gesture reveals one app-defined page of content.

Guidelines

You can use a scroll view to give people access to large views—or to large numbers of views—in a constrained space. Because people are accustomed to using scroll views throughout iOS, make sure the scroll views in your app behave as people expect.

Support zoom behavior appropriately. If it makes sense in your app, you can let users pinch or double-tap to zoom into and out of a scroll view. When you enable zoom, you should also set maximum and minimum scale values that make sense in the context of the user's task. For example, letting users zoom in on text until one character fills the screen is unlikely to make it easier for them to read the content.

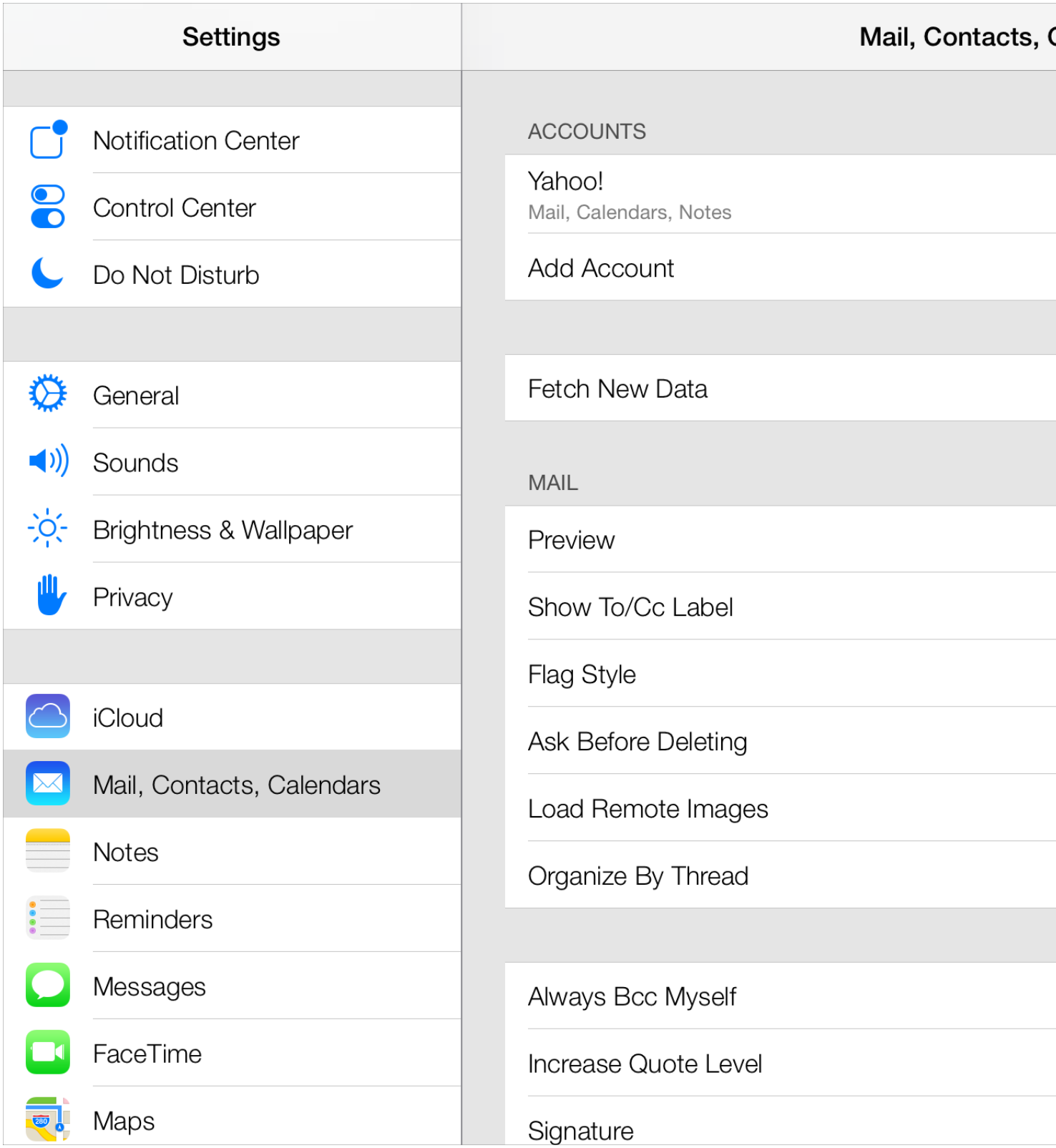
Consider using a page control with a paging-mode scroll view. When you want to display content that's been divided into pages, screenfuls, or other chunks, it's a good idea to give users a way to tell how many chunks are available and which one they're currently viewing. A page control displays dots that give people this information and—because page controls are used in Safari, Stocks, Weather, and other built-in apps—people already understand how to use them.

When you combine a page control with a paging-mode scroll view, it's a good idea to disable the scroll indicator that's on the same axis as the page control. People then have one unambiguous way to page through the content. For more information about using a page control in your app, see "Web View" (page 162).

In general, display only one scroll view at a time. People often make large swipe gestures when they scroll, so it can be difficult for them to avoid interacting with a neighboring scroll view on the same screen. If you decide to put two scroll views on one screen, consider allowing them to scroll in different directions so that one gesture is less likely to scroll both views. For example, Stocks in portrait orientation on iPhone displays stock quotes in a vertically scrolling view above company-specific information in a horizontally scrolling view.

Split View Controller (iPad Only)

A **split view controller** is a full-screen view controller that manages the presentation of two side-by-side view controllers.



Each child view controller of a split view controller is responsible for managing the display of one pane. The split view controller itself presents these child view controllers and manages transitions between different orientations. To learn more about defining a split view controller in your code, see *UISplitViewController Class Reference*.

Important: Split view controllers are available in iPad apps only.

Appearance and Behavior

A split view controller contains two panes. The width of the left pane is fixed at 320 points in all orientations. Users can't resize either pane of a split view controller.

Note: Even though the left pane is often called the *master pane* and the right pane is often called the *detail pane*, this relationship is not enforced in code.

Both panes can contain a wide variety of objects and views, such as:

- Table, image, map, text, web, or custom views.
- Navigation bars, toolbars, or tab bars.

Guidelines

Use a split view controller to display persistent information in the left pane and related details or subordinate information in the right pane. In this design pattern, when people select an item in the left pane, the right pane should display the information related to that item. (You're responsible for making this happen in code.)

Sometimes, when an app uses a split view controller in landscape, the contents of the left pane is displayed in a popover when the device rotates to portrait. However, you don't have to follow this pattern. If it makes sense in your app, you can design your UI to display side-by-side views in all orientations.

Avoid creating a right pane that is narrower than the left pane. Although the width of the right pane is up to you, it doesn't look good to use a width of less than 320 points (which is the width of the left pane).

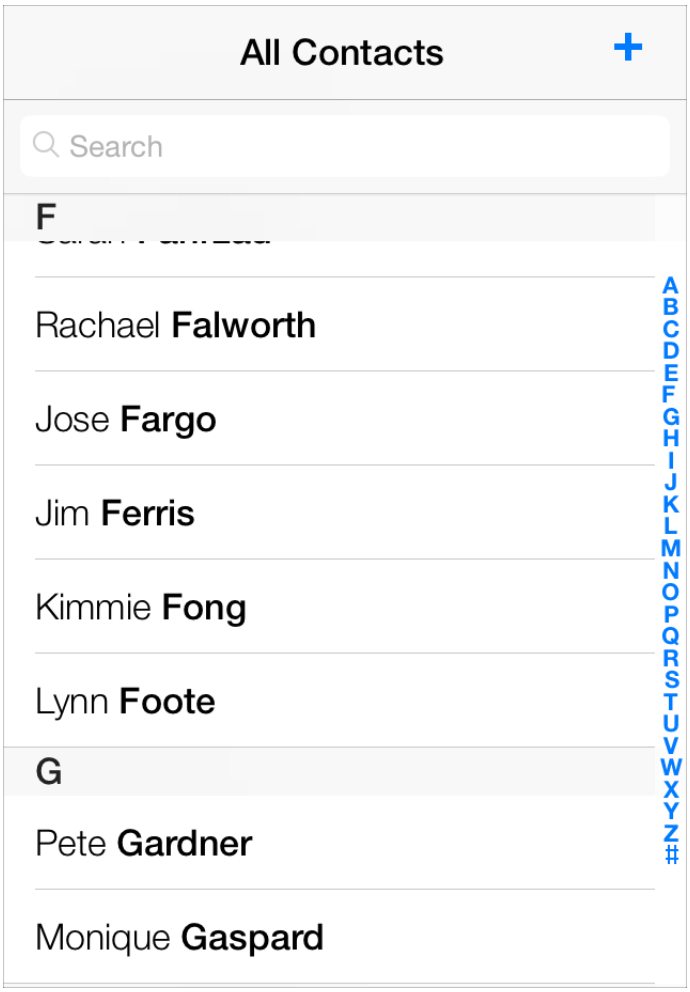
Avoid displaying a navigation bar in both panes at the same time. Doing this would make it very difficult for users to discern the relationship between the two panes.

In general, indicate the current selection in the left pane in a persistent way. This behavior helps people understand the relationship between the item in the left pane and the contents of the right pane. This is important because the content of the right pane can change, but it should always remain related to the item selected in the left pane.

Give people alternative ways to access the left pane, if appropriate. By default, only the right pane is displayed in portrait orientation and you provide users with a button—typically located in a navigation bar—to reveal and hide the left pane. The split view controller also supports the swipe gesture to perform the reveal/hide action. Unless your app uses the swipe gesture to perform other functions, you should let people use it to access the left pane.

Table View

A **table view** presents data in a single-column list of multiple rows.



To learn more about defining a table view in your code, see *Table View Programming Guide for iOS* and "Table Views".

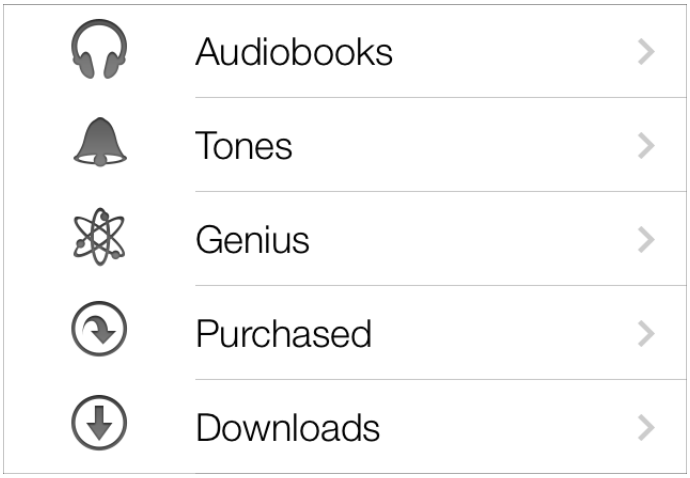
Appearance and Behavior

A table view displays data in rows that can be divided by section or separated into groups. Users flick or drag to scroll through rows or groups of rows. Users tap a table row to select it and use table view controls to add or remove rows, select multiple rows, see more information about a row item, or reveal another table view. A table row highlights briefly when the user taps a selectable item.

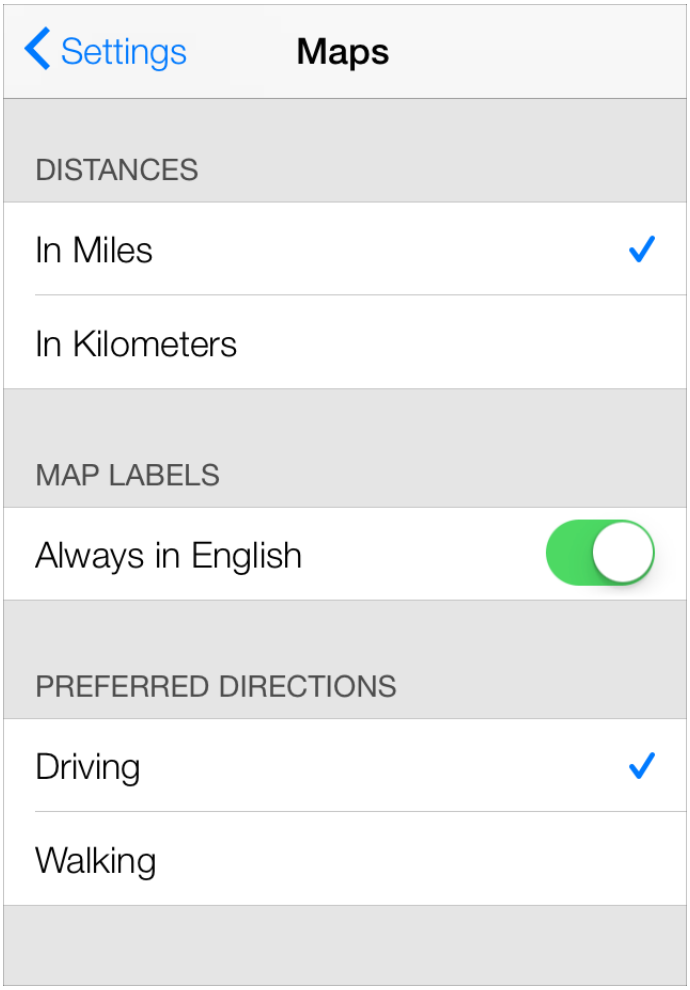
If a row selection results in navigation to a new screen, the selected row highlights briefly as the new screen slides into place. When the user navigates back to the previous screen, the originally selected row again highlights briefly to remind the user of their earlier selection (it doesn't remain highlighted).

iOS defines two styles of table views.

Plain tables display rows that can be separated into labeled sections. An optional index can appear vertically along the right edge of the view. A header can appear before the first item in a section and a footer can appear after the last item.



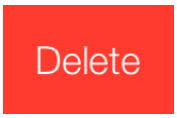
Grouped tables display groups of rows. A grouped table view always contains at least one group of list items—one list item per row—and each group always contains at least one item. A group can be preceded by a header and followed by a footer. A grouped table view doesn't include an index.



iOS includes some **table-view elements** that can extend the functionality of table views. Unless noted otherwise, these elements are suitable for use with table views only.

Table 34-1 Table-view elements

Table-view element	Name	Description
✓	Checkmark	Indicates that the row is selected
›	Disclosure indicator	Displays another table associated with the row

Table-view element	Name	Description
	Detail Disclosure button	Displays additional details about the row in a new view (for information on how to use this element outside of a table, see "Detail Disclosure Button" (page ?))
	Row reorder	Indicates that the row can be dragged to another location in the table
	Row insert	Adds a new row to the table
	Delete button control	In an editing context, reveals and hides the Delete button for a row
	Delete button	Deletes the row

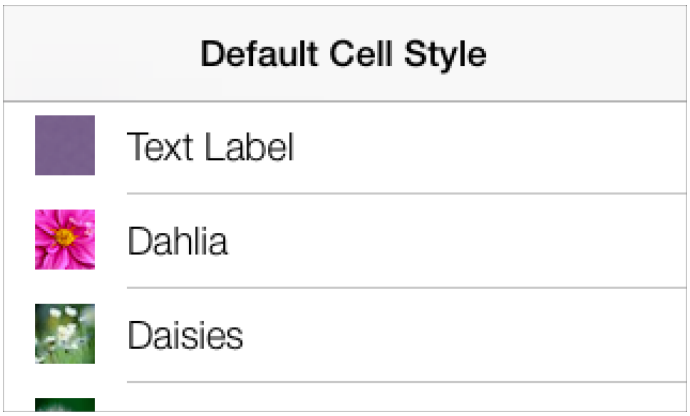
In addition to the table-specific elements listed in "Table-view elements," iOS defines the refresh control, which gives users the ability to refresh a table's contents. To learn more about using a refresh control with a table in your app, see "Refresh Control" (page 172).

iOS defines four table-cell styles that implement the most common layouts for table rows in both plain and grouped tables. Each cell style is best suited to display a different type of information.

> **Note:** Programmatically, these styles are applied to a table view's cell, which is an object that tells the table how to draw its rows.

Default (`UITableViewCellStyleDefault`). The default cell style includes an optional image in the left end of the row, followed by a left-aligned title.

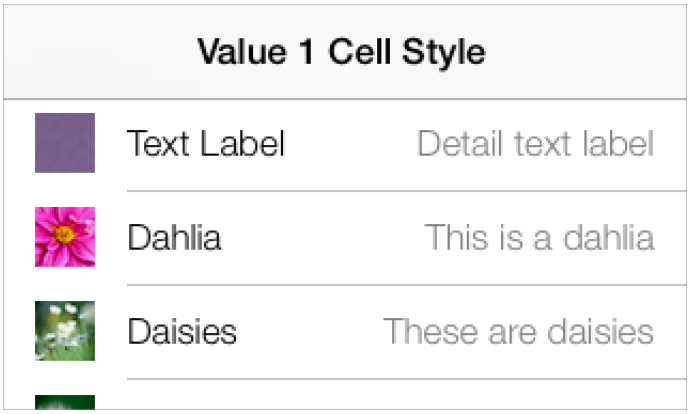
The default style is good for displaying a list of items that don't need to be differentiated by supplementary information.



Subtitle (`UITableViewCellStyleSubtitle`). The subtitle style includes an optional image in the left end of the row, followed by a left-aligned title on one line and a left-aligned subtitle on the line below.

The left-alignment of the text labels makes the list easy to scan. This table-cell style works well when list items look similar, because users can use the additional information in the detail text labels to help distinguish items named in the text labels.

Value 1 (`UITableViewCellStyleValue1`). The value 1 style displays a left-aligned title on the same line with a right-aligned subtitle in a lighter font.



Value 2 (`UITableViewCellStyleValue2`). The value 2 style displays a right-aligned title in a blue font, followed on the same line by a left-aligned subtitle in a black font. Images don't fit well in this style.

In the value 2 layout, the crisp, vertical margin between the text and the detail text helps users focus on the first words of the detail text label.

Note: All four standard table-cell styles also allow the addition of a table-view element, such as the checkmark or the disclosure indicator. Adding these elements decreases the width of the cell available for the title and subtitle.

Guidelines

Use a table view to display large or small amounts of information cleanly and efficiently. For example, you can use a table view to:

Provide a list of options from which users can select. You can use the checkmark to show users the currently selected options in the list.

Display hierarchical information. The plain table style is well-suited to display a hierarchy of information. Each list item can lead to a different subset of information displayed in another list. Users follow a path through the hierarchy by selecting one item in each successive list. The disclosure indicator tells users that tapping anywhere in the row reveals the subset of information in a new list.

Display conceptually grouped information. Both table view styles allow you to provide context by supplying header and footer views between sections of information.

In iOS 6.0 and later, you can use a header-footer view—that is, an instance of `UITableViewHeaderFooterView`—to display text or a custom view in a header or footer. To learn how to use a header-footer view in your code, see *UITableViewHeaderFooterView Class Reference*.

Display an index to facilitate lookup. The plain style supports an index view that helps users quickly find what they need. The index consists of a column of entries—usually letters in an alphabet—that floats on the right edge of the screen.

If you include an index, avoid using table-view elements that display on the right edge of the table—such as the disclosure indicator—because these elements interfere with the index.

Follow these guidelines when you use table views:

Always provide feedback when users select a list item. Users expect a table row to highlight briefly when they tap a selectable item in it. After tapping, users expect a new view to appear or the row to display a checkmark to indicate that the item has been selected or enabled.

If table content is extensive or complex, avoid waiting until all the data is available before displaying anything. Instead, fill the onscreen rows with textual data immediately and display more complex data—such as images—as they become available. This technique gives users useful information right away and increases the perceived responsiveness of your app.

Consider displaying "stale" data while waiting for new data to arrive. Although this technique isn't recommended for apps that handle frequently changing data, it can help more static apps give users something useful right away. Before you decide to do this, gauge how often the data changes and how much users depend on seeing fresh data quickly.

If the data is slow-loading or complex, tell users that processing is continuing. If a table contains only complex data, it might be difficult to display anything useful right away. In these rare cases, it's important to avoid displaying empty rows, because this can imply that your app has stalled. Instead, the table should display a spinning activity indicator along with an informative label, such as "Loading...", centered in the screen. Doing this reassures users that processing is continuing.

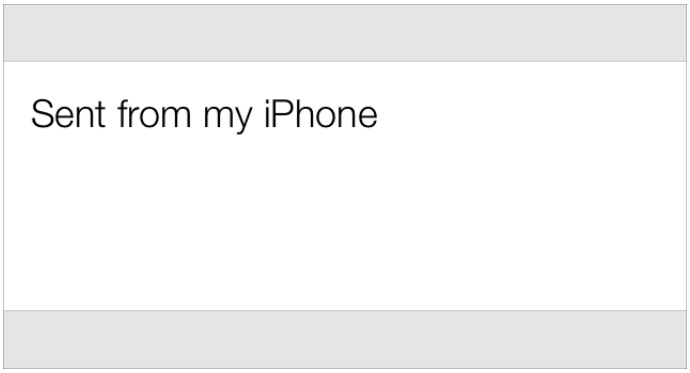
If appropriate, use a custom title for the Delete button. If it helps users to better understand the way your app works, you can create a title to replace the system-provided Delete title.

As much as possible, use succinct text labels to avoid truncation. Truncated words and phrases can be difficult for users to scan and understand. Text truncation is automatic in all table-cell styles, but it can present more or less of a problem, depending on which cell style you use and on where truncation occurs.

If you want to lay out your table rows in a nonstandard way, it's better to create a custom table-cell style than to significantly alter a standard one. To learn how to create your own cells, see "Customizing Cells" in *Table View Programming Guide for iOS*.

Text View

A **text view** accepts and displays multiple lines of text.



To learn more about defining a text view in your code, see "Text Views".

Appearance and Behavior

A text view is a rectangle of any height. A text view supports scrolling when the content is too large to fit inside its bounds.

If the text view supports user editing, a keyboard appears when the user taps inside the text view. The keyboard's input method and layout are determined by the user's language settings.

Guidelines

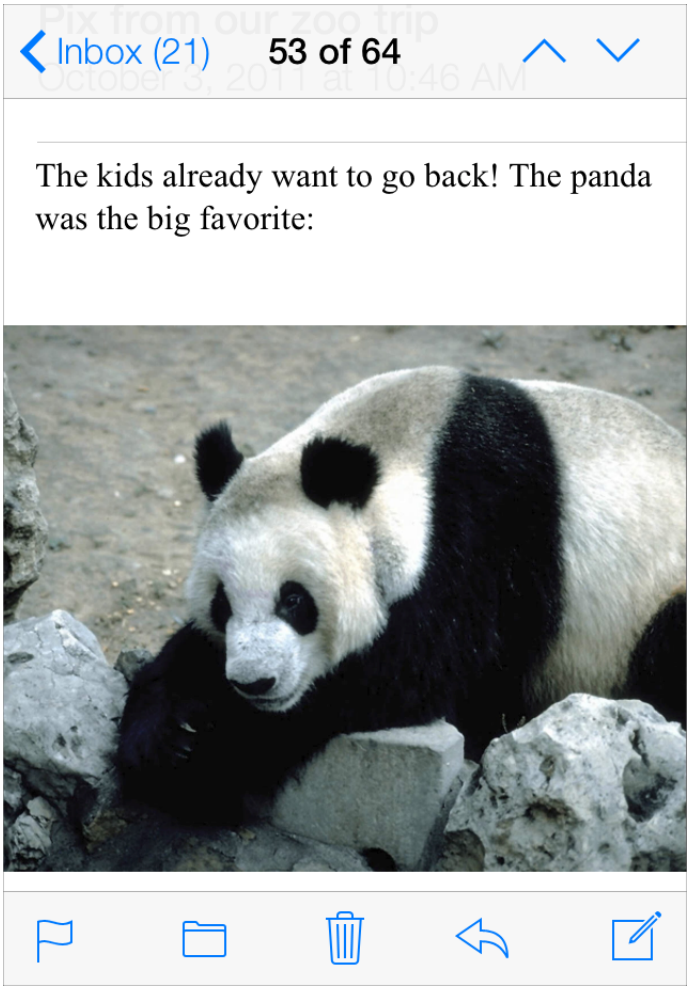
You have control over the font, color, and alignment of the text in a text view. The default font is the system font and the default color is black. The default for the alignment property is left (you can change it to center or right).

Always make sure the text is easy to read. Although you can use attributed strings to combine multiple fonts, colors, and alignments in creative ways, it's essential to maintain the readability of the text. It's a good idea to support Dynamic Type and use the `UIFont` method `preferredFontForTextStyle` to get the text for display in a text view.

Specify different keyboard types to accommodate different types of content you expect users to enter. For example, you might want to make it easy for users to enter a URL, a PIN, or a phone number. Note, however, that you have no control over the keyboard's input method and layout, which are determined by the user's language settings. iOS provides several different keyboard types, each designed to facilitate a different type of input. To learn about the keyboard types that are available, see the documentation for `UIKeyboardType`. To learn more about managing the keyboard in your app, read "Managing the Keyboard" in *iOS App Programming Guide* .

Web View

A **web view** is a region that can display rich HTML content (shown here between the navigation bar and toolbar in Mail on iPhone).



To learn more about defining a web view in your code, see “Web Views”.

Appearance and Behavior

In addition to displaying web content, a web view performs some automatic processing on web content, such as converting a phone number to a phone link.

Guidelines

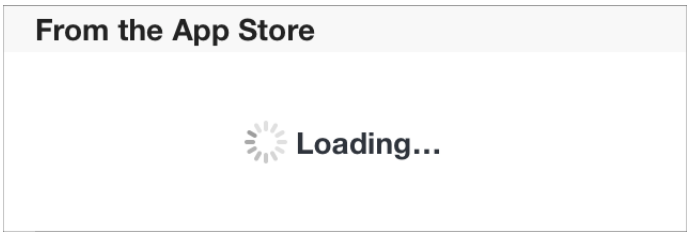
If you have a webpage or web app, you might decide to use a web view to implement a simple iOS app that provides a wrapper for your webpage or web app. If you plan to use a web view to access web content that you control, be sure to read *Safari Web Content Guide*.

Avoid using a web view to create an app that looks and behaves like a mini web browser. People expect to use Safari on iOS to browse web content, so replicating this broad functionality within your app is not recommended.

Controls

Activity Indicator

An **activity indicator** shows that a task or process is progressing (shown below with text labels).



To learn how to define an activity indicator in your code, see *UIActivityIndicatorView Class Reference*.

Appearance and Behavior

An activity indicator spins while a task is progressing and disappears when the task completes. Users don't interact with an activity indicator.

Guidelines

Use an activity indicator in a toolbar or a main view to show that processing is occurring, without suggesting when it will finish.

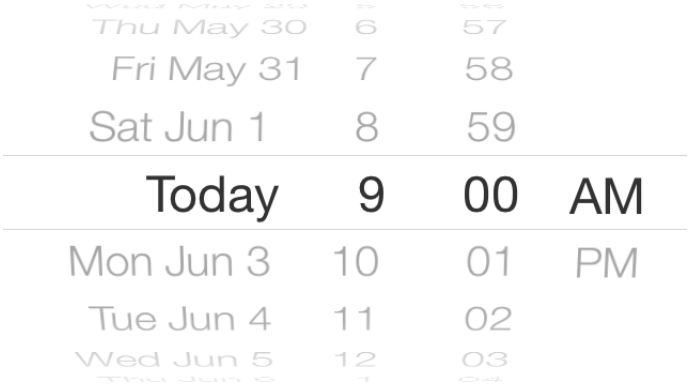
Don't display a stationary activity indicator, because users associate this with a stalled process.

Use an activity indicator when it's more important to reassure users that their task or process has not stalled than it is to suggest when processing will finish.

If appropriate, customize the size and color of an activity indicator to coordinate with the background of the view in which it appears.

Date Picker

A **date picker** displays components of date and time, such as hours, minutes, days, and years.



To learn how to define a date picker in your code, see “Date Pickers”.

Appearance and Behavior

A date picker can have up to four independent wheels, each of which displays values in a single category, such as month or hour. Users flick or drag to spin each wheel until it displays the desired value horizontally in the middle of the picker. The final value comprises the values displayed in each wheel.

The overall size of a date picker is fixed at the same size as the iPhone keyboard.

A date picker has four modes, each of which displays a different number of wheels that contain a set of different values.

- **Date and time.** The date and time mode—which is the default mode—displays wheels for the calendar date, hour, and minute values, plus an optional wheel for the AM/PM designation.
- **Time.** The time mode displays wheels for the hour and minute values, plus an optional wheel for the AM/PM designation.
- **Date.** The date mode displays wheels for the month, day, and year values.
- **Countdown timer.** The countdown timer mode displays wheels for the hour and minute. You can specify the total duration of a countdown, up to a maximum of 23 hours and 59 minutes.

Guidelines

Use a date picker to let users pick—instead of type—a date or time value that consists of multiple parts, such as the day, month, and year. A date picker is easy to use because the values in each part have a relatively small range and users already know what the values are.

As much as possible, display a date picker inline with the content. It's best when users can avoid navigating to a different view to use a date picker.

If it makes sense in your app, change the interval in the minutes wheel. By default, a minutes wheel displays 60 values (0 to 59). If you need to display a coarser granularity of choices, you can set a minutes wheel to display a larger minute interval, as long as the interval divides evenly into 60. For example, you might want to display the quarter-hour intervals 0, 15, 30, and 45.

Contact Add Button

A **Contact Add button** lets the user add an existing contact to a text field or other text-based view.

To learn how to define a Contact Add button in your code, see "Buttons".

Appearance and Behavior

When users tap a Contact Add button, a list of their contacts is revealed. When users choose a contact from the list, the list closes and the contact is added to the view that contains the Contact Add button.

Guidelines

Use a Contact Add button to give users an easy way to access a contact without using the keyboard. For example, users can tap the Contact Add button in the To field of the Mail compose view instead of typing a recipient's name.

Because the Contact Add button functions as an alternative to typing contact information, it's not appropriate to use the button in a view that doesn't accept keyboard input.

Detail Disclosure Button

A **Detail Disclosure button** reveals additional details or functionality related to an item (shown here inside a map annotation view).

To learn how to define a Detail Disclosure button in your code, see *UITableViewCell Class Reference* and "Buttons".

Appearance and Behavior

Users tap a Detail Disclosure button to reveal additional information or functionality related to a specific item. The additional details or functionality are revealed in a separate view.

When a Detail Disclosure button appears in a table row, tapping elsewhere in the row doesn't activate the Detail Disclosure button; instead, it selects the row item or results in app-defined behavior.

Guidelines

Typically, you use a Detail Disclosure button in a table view to give users a way to see more details or functionality related to a list item. However, you can also use this element in other types of views to provide a way to reveal more information or functionality related to an item in that view.

Info Button

An **Info button** reveals configuration details about an app, sometimes on the back of the current view.

To learn more about defining an Info button in your code, see "Buttons".

Appearance and Behavior

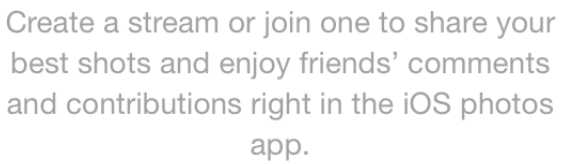
iOS includes two styles of Info button: a dark-colored button that looks good on light content and a light-colored button that looks good on dark content.

Guidelines

Use an Info button to reveal configuration details or options about an app. You can use the style of Info button that coordinates best with the UI of your app.

Label

A **label** displays static text.



To learn more about defining labels in your code, see *UILabel Class Reference*.

Appearance and Behavior

A label displays any amount of static text. Users don't interact with labels except—potentially—to copy the text.

Guidelines

Use a label to name or describe parts of your UI or to provide short messages to the user. A label is best suited to display a relatively small amount of text.

Take care to make your labels legible. It's best to support Dynamic Type and use the `UIFont` method `preferredFontForTextStyle` to get the text for display in a label. If you choose to use custom fonts, don't sacrifice clarity for fancy lettering or showy colors.

Network Activity Indicator

A **network activity indicator** appears in the status bar and shows that network activity is occurring.

In your code, use the `UIApplication` method `networkActivityIndicatorVisible` to control the indicator's visibility.

Appearance and Behavior

The network activity indicator spins in the status bar while network activity proceeds and it disappears when network activity stops. Users don't interact with the network activity indicator.

Guidelines

Display the network activity indicator to provide feedback when your app accesses the network for more than a couple of seconds. If the operation finishes sooner than that, you don't have to show the network activity indicator, because the indicator would be likely to disappear before users notice its presence.

Page Control

A **page control** indicates how many views are open and which one is currently visible (shown here in Weather).

To learn more about defining a page control in your code, see "Page Controls".

Appearance and Behavior

A page control displays an indicator dot for each currently open peer view in an app. From left to right, the dots represent the order in which the views were opened (the leftmost dot represents the first view). By default, the dot that represents the currently visible view is opaque and the dots that represent all other views are translucent. Users tap to the left or the right of the current dot to see the previous or next open view.

The dots of a page control don't shrink or squeeze together as more appear. A screen in portrait orientation can accommodate approximately 20 dots; if you try to display more dots than will fit in the screen, they will be clipped.

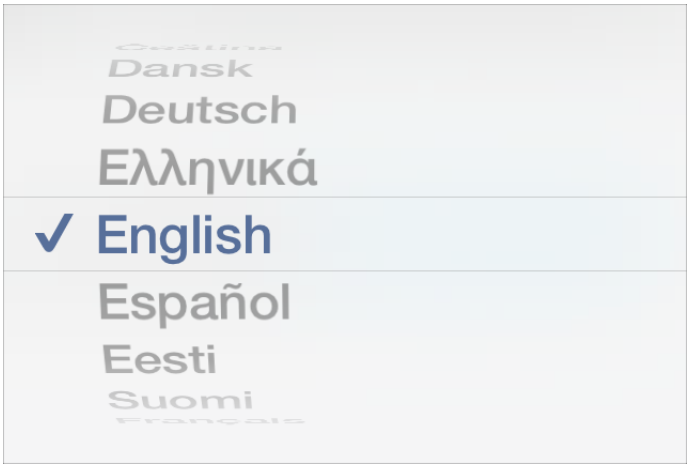
Guidelines

Use a page control when each view in your app is a peer of every other view. Don't use a page control if your app displays information in a hierarchy of views, because a page control doesn't help users retrace their steps through a specific path.

Vertically center a page control between the bottom edge of an open view and the bottom edge of the screen, where it's always visible without getting in users' way. Avoid trying to display too many dots for the current screen orientation.

Picker

A **picker** displays a set of values from which a user picks one.



To learn more about defining a picker in your code, see *UIPickerView Class Reference*.

Appearance and Behavior

A picker is a generic version of the date picker. As with a date picker, users spin the wheel (or wheels) of a picker until the value they want appears. The overall size of a picker, including its background, is fixed at the same size as the keyboard on iPhone. (For more information about the date picker, see "Date Picker" (page ?).)

Guidelines

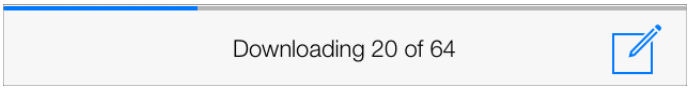
Use a picker to make it easy for people to choose from a set of values. It's often best to use a picker when people are familiar with the entire set of values. This is because many of the values are hidden when the wheel is stationary. If you need to provide a large set of choices that aren't well known to your users, a picker might not be the appropriate control.

As much as possible, display a picker inline with the content. It's best when users can avoid navigating to a different view to use a picker.

Consider using a table view, instead of a picker, if you need to display a very large number of values. This is because the greater height of a table view makes scrolling faster.

Progress View

A **progress view** shows the progress of a task or process that has a known duration (shown here in the Mail toolbar).



To learn more about defining a progress view in your code, see *UIProgressView Class Reference*.

Appearance and Behavior

iOS provides two styles of progress view. The appearance of each style is very similar, except for height.

- The **default** style has a weight that makes it suitable for use in an app's main content area.
- The **bar** style is thinner than the default style, which makes it well-suited for use in a toolbar.

As the task or process proceeds, the track of the progress view is filled from left to right. At any given time, the proportion of filled to unfilled area in the progress view gives people an indication of how soon the task or process will finish. Users don't interact with a progress view.

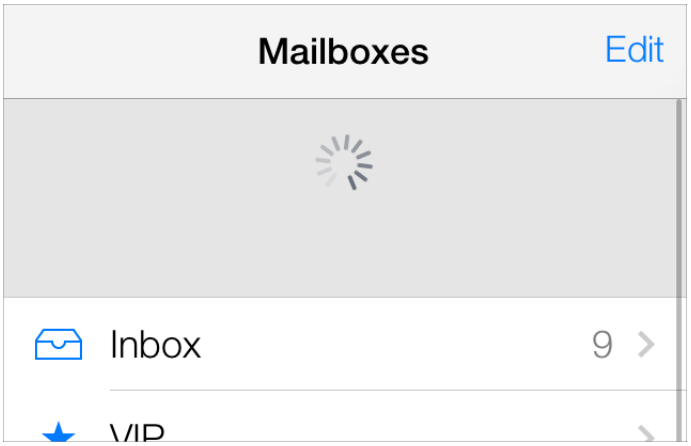
Guidelines

Use a progress view to give feedback on a task that has a well-defined duration, especially when it's important to show approximately how long the task will take. When you display a progress view, you tell the user that their task is being performed and you give them enough information to decide if they want to wait until the task is complete or cancel it.

If appropriate, customize the appearance of a progress view to coordinate with the style of your app. You can specify a custom tint or image for both the track and the fill of a progress view.

Refresh Control

A **refresh control** performs a user-initiated content refresh—typically in a table (shown here above the mailbox list).



To learn more about defining a refresh control in your code, see *UIRefreshControl Class Reference*.

Appearance and Behavior

By default, a refresh control is hidden until the user initiates a refresh action by dragging down from the top edge of a table. The refresh control looks similar to an activity indicator.

A refresh control can also display a title and use a custom tint.

Guidelines

Use a refresh control to give users a consistent way to tell a table or other view to update its contents immediately, without waiting for the next automatic update. The following guidelines can help you ensure that your refresh control enhances the user's experience.

Don't stop performing automatic content updates just because you provide a refresh control. Even though users appreciate being able to request that an update be performed *now*, they still appreciate content that refreshes itself automatically. And if you rely on users to initiate all refreshes, users who are unaware of the refresh control are likely to wonder why your app displays stale data. In general, you want to give users the option to refresh contents immediately; you don't want to make users responsible for every update.

Supply a short title only if it adds value. In particular, don't use the title to describe how to use the refresh control.

Rounded Rectangle Button

The rounded rectangle button has been deprecated in iOS 7. Instead, use the system button—that is, a `UIButton` of type `UIButtonTypeSystem`. For guidelines, see "System Button" (page 176).

Segmented Control

A **segmented control** is a linear set of segments, each of which functions as a button that can display a different view.

To learn more about defining a segmented control in your code, see "Segmented Controls".

Appearance and Behavior

The length of a segmented control is determined by the number of its segments; the height of a segmented control is fixed. The width of each segment is proportional, based on the total number of segments. When users tap a segment, the segment displays a selected appearance.

Guidelines

Use a segmented control to offer closely related, but mutually exclusive choices.

Make sure that each segment is easy to tap. To maintain a comfortable hit region of 44 x 44 points for each segment, you need to limit the number of segments. On iPhone, a segmented control should have five or fewer segments.

As much as possible, maintain consistency in the size of each segment's contents. Because all segments in a segmented control have equal width, it doesn't look good if the content fills some segments, but not others.

Avoid mixing text and images in a single segmented control. A segmented control can contain text or images. An individual segment can contain either text or an image, but not both. In general, it's best to avoid putting text in some segments and images in other segments of a single segmented control.

If appropriate, customize the appearance of a segmented control. For example, you can supply a custom background tint or image. If you supply a background image, you can also specify a different background image to use for a segment's selected appearance and a custom appearance for the dividers between segments. (In some cases, it can be a good idea to supply a resizable background image; to learn more about creating one, see "Creating Resizable Images" (page 188).)

If you customize the background appearance of a segmented control, you should make sure that the automatic centering of the control's text or image content still looks good. If necessary, you can use the bar metrics APIs to adjust the positioning of the content inside the segmented control (to learn more about specifying bar metrics, see the appearance-customization APIs described in UISegmentedControl).

Slider

A **slider** allows users to make adjustments to a value or process throughout a range of allowed values (shown here with custom images on the left and the right).

To learn more about defining a slider in your code, see "Sliders".

Appearance and Behavior

A slider consists of a horizontal track and a thumb (a circular control that the user can slide) and can include optional images that convey the meaning of the right and left values. When people drag the thumb along the slider, the value or process is updated continuously and is displayed in the track.

Guidelines

Use a slider to give users fine-grained control over values they can choose or the operation of the current process.

If appropriate, customize the appearance of a slider. For example, you can do any of the following:

- Set the width of a slider to fit in with the UI of your app.
- Define the appearance of the thumb, so that users can see at a glance whether the slider is active.
- Supply images to appear at both ends of the slider to help users understand what the slider does.

  Typically, these custom images correspond to the minimum and maximum values of the value range that the slider controls. A slider that controls image size, for example, could display a very small image at the minimum end and a very large image at the maximum end.
- Define a different appearance for the track, depending on which side of the thumb it is on and which state the control is in.

Stepper

A **stepper** increases or decreases a value by a constant amount.

To learn more about defining a stepper in your code, see “Steppers”.

Appearance and Behavior

A stepper is a two-segment control in which one segment displays a plus symbol and the other segment displays a minus symbol. Users tap a segment to increase or decrease a value. A stepper doesn’t display the value that the user changes.

Guidelines

In general, use a stepper when users might need to make small adjustments to a value. For example, it makes sense to use a stepper to set the number of copies in the Printer Options action sheet, because users rarely change this value by very much. On the other hand, it wouldn’t make sense to use a stepper to help users choose a page range, because those values can vary a lot.

Make it obvious which value the stepper affects. A stepper doesn’t display any values, so you need to make sure that users know which value they’re changing when they use a stepper.

If appropriate, customize the appearance of a stepper to coordinate with the style of your app. You can specify a custom tint for the control or you can supply custom images for the background and divider and for the increment and decrement symbols.

Switch

A **switch** presents two mutually exclusive choices or states (used in table views only).

To learn more about defining a switch in your code, see “Switches”.

Appearance and Behavior

A switch displays the value that is currently in effect; users slide the control to select the other value. Users can also tap the control to switch between choices.

Guidelines

Use a switch in a table row to give users two simple, diametrically opposed choices that determine the state of something, such as yes/no or on/off.

You can use a switch control to change the state of other UI elements in the view. Depending on the choice users make, new list items might appear or disappear, or list items might become active or inactive.

System Button

A system button performs an app-specific action.

If you need to display a button that includes a bezel, use a button of type `UIButtonTypeCustom` and supply a custom background image.

To learn more about defining a system button in your code, see “Buttons”.

Appearance and Behavior

iOS 7 system buttons don’t include a bezel or a background appearance. A system button can contain an icon or a text title, and it can specify a tint color or receive its parent’s color.

> **Note:** In iOS 7, `UIButtonTypeRoundedRect` has been redefined as `UIButtonTypeSystem`. An app that uses a rounded rectangle button in iOS 6 automatically gets the system button appearance when it links against iOS 7.

Guidelines

Use a system button to initiate an action. When you supply a title for a system button, follow this approach:

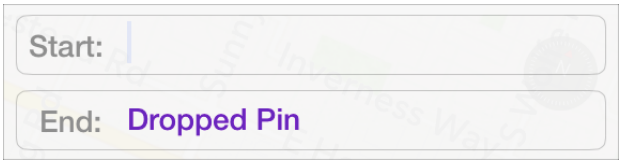
- Use title-style capitalization—that is, capitalize every word except articles, coordinating conjunctions, and prepositions of four or fewer letters

- Avoid creating a title that is too long. Overly long text gets truncated, which can make it difficult for users to understand it.

You can specify the title or image to display in a system button, and you can tell the button to highlight when it's tapped and specify how the title or image should look when the button highlights. You can also supply a tint for the button's content.

Text Field

A **text field** accepts a single line of user input (shown here with a purpose description and placeholder text).



To learn more about defining a text field and customizing it to display images and buttons, see "Text Fields".

Appearance and Behavior

A text field is a fixed-height field with rounded corners. When users tap a text field a keyboard appears; when users dismiss the keyboard, the text field handles the input in an app-specific way.

Guidelines

Use a text field to get a small amount of information from the user. Before you decide to use a text field, consider whether there are other controls that might make inputting the information easier, such as a picker or a list.

Customize a text field if it helps users understand how they should use it. For example, you can display custom images in the left or right sides of the text field, or add a system-provided button, such as the Bookmarks button. In general, you should use the left end of a text field to indicate its purpose and the right end to indicate the presence of additional features, such as bookmarks.

Display the Clear button in the right end of a text field when appropriate. When this element is present, tapping it clears the contents of the text field, regardless of any other image you might display over it.

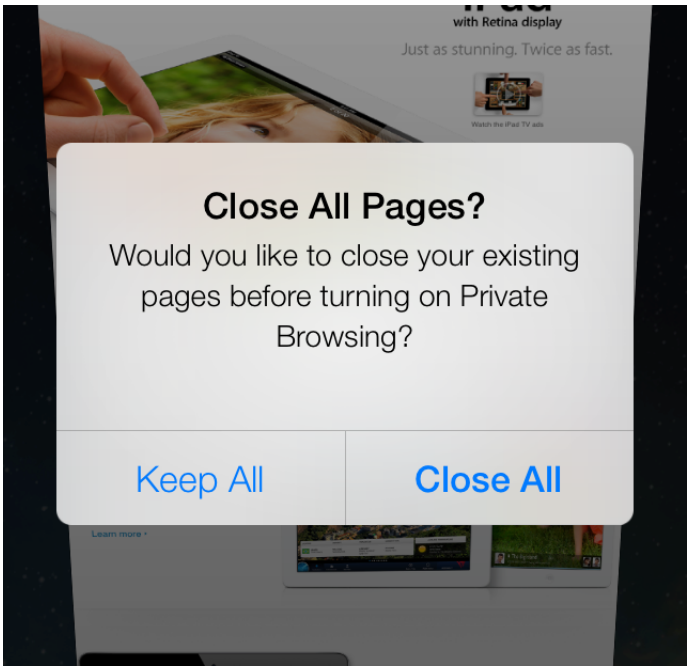
Display a hint in the text field if it helps users understand its purpose, such as "Name" or "Address." A text field can display such placeholder text when there is no other text in the field.

Specify a keyboard type that's appropriate for the type of content you expect users to enter. For example, you might want to make it easy for users to enter a URL, a PIN, or a phone number. iOS provides several different keyboard types, each designed to facilitate a different type of input. To learn about the keyboard types that are available, see the documentation for `UIKeyboardType`. To learn more about managing the keyboard in your app, read "Managing the Keyboard" in *iOS App Programming Guide* . Note that you have no control over the keyboard's input method and layout, because these attributes are determined by the user's language settings.

Temporary Views

Alert

An **alert** gives people important information that affects their use of the app or the device.



To learn about using an alert in your code, see *UIAlertView Class Reference*.

Appearance and Behavior

An alert pops up in the middle of the app screen and floats above its views. The unattached appearance of an alert emphasizes the fact that its arrival is due to some change in the app or the device, not necessarily as the result of the user's most recent action. Users must dismiss the alert before they can continue using the currently running app.

An alert always contains at least one button, which users tap to dismiss the alert. By default, an alert displays a title and might also display a message that provides additional information. An alert can contain one or two text fields, one of which can be a secure text-input field. The background appearance of an alert is system-defined and can't be changed.

> **Note:** A local or push notification might use an alert to communicate with users, although this is rare. To learn more about local and push notifications, see "Notification Center" (page $@).

Guidelines

The infrequency with which alerts appear helps users take them seriously. Be sure to minimize the number of alerts your app displays and ensure that each one offers critical information and useful choices.

Avoid creating unnecessary alerts. In general, alerts are unnecessary if they:

- Merely increase the visibility of some information, especially information that's related to the standard functioning of your app.

  Instead, design an eye-catching way to display the information that harmonizes with your app's style.

- Update users on tasks that are progressing normally.

  Instead, consider using a progress view or an activity indicator to provide progress-related feedback to users (these methods of feedback are described in "Progress View" (page 171) and "Activity Indicator" (page 164)).

- Ask for confirmation of user-initiated actions.

  To get confirmation for an action the user initiated—even a potentially risky action such as deleting a contact—you should use an action sheet (described in "Action Sheet" (page 183)).

- Inform users of errors or problems about which they can do nothing.

  Although it might be necessary to use an alert to tell users about a critical problem they can't fix, it's better to integrate such information into the UI, if possible. For example, instead of telling users every time a server connection fails, display the time of the last successful connection.

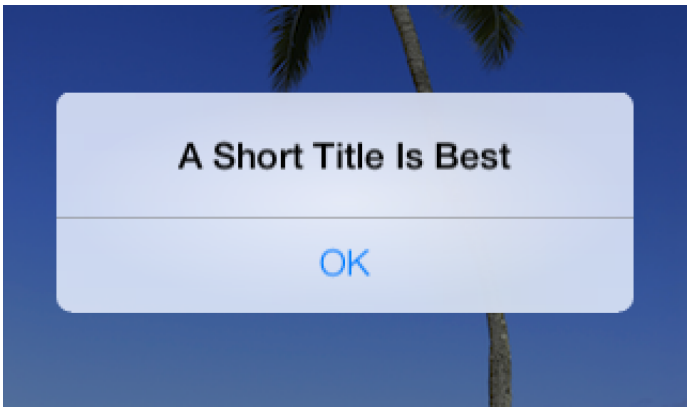
You can specify the text of the required title and optional message, the number of buttons, and the button contents in an alert. You can't customize the width or the background appearance of the alert view itself, or the alignment of the text (it's center-aligned).

As you read the alert-text design guidelines, it's useful to know the following definitions:

- **Title-style capitalization** means that every word is capitalized, except articles, coordinating conjunctions, and prepositions of four or fewer letters when they aren't the first word.
- **Sentence-style capitalization** means that the first word is capitalized, and the rest of the words are lowercase unless they are proper nouns or proper adjectives.

Succinctly describe the situation and explain what people can do about it. Ideally, the text you write gives people enough context to understand why the alert has appeared and to decide which button to tap.

Keep the title short enough to display on a single line, if possible. A long alert title is difficult for people to read quickly, and it might get truncated or force the alert message to scroll.



Avoid single-word titles that don't provide any useful information, such as "Error" or "Warning."

When possible, use a sentence fragment. A short, informative statement is often easier to understand than a complete sentence.

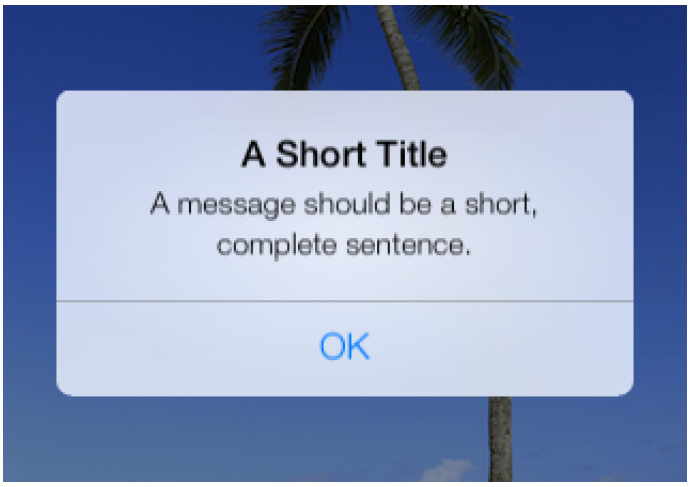
Don't hesitate to be negative. People understand that most alerts tell them about problems or warn them about dangerous situations. It's better to be negative and direct than it is to be positive but oblique.

As much as possible, avoid using "you," "your," "me," and "my". Sometimes, text that identifies people directly can be ambiguous and can even be interpreted as an insult.

Use capitalization and punctuation appropriately. Specifically:

- Use title-style capitalization and no ending punctuation when the title is a sentence fragment or it consists of a single sentence that is not a question.
- If the title consists of a single sentence that is a question, use sentence-style capitalization and an ending question mark. In general, consider using a question for an alert title if it allows you to avoid adding a message.
- If the title consists of two or more sentences, use sentence-style capitalization and appropriate ending punctuation for each sentence. A two-sentence alert title should seldom be necessary, although you might consider it if it allows you to avoid adding a message.

If you provide an optional alert message, create a short, complete sentence. Use sentence-style capitalization and appropriate ending punctuation.



Avoid creating an alert message that is too long. If possible, keep the message short enough to display on one or two lines. If the message is too long it will scroll, which is not a good user experience.

Avoid lengthening your alert text with descriptions of which button to tap, such as "Tap View to see the information." Ideally, the combination of unambiguous alert text and logical button labels gives people enough information to understand the situation and their choices. However, if you must provide detailed guidance, follow these guidelines:

- Be sure to use the word "tap" (not "touch" or "click" or "choose") to describe the selection action.
- Don't enclose a button title in quotation marks, but do preserve its capitalization.

Be sure to test the appearance of your alert in both orientations. Because the height of an alert is constrained in landscape, it might look different from the way it looks in portrait. It's recommended that you optimize the length of the alert text so that it looks good and avoids scrolling in both orientations.

Generally, use a two-button alert. A two-button alert is often the most useful, because it's easiest for people to choose between two alternatives. A single button alert is rarely helpful because it informs without giving people any control over the situation. An alert that contains three or more buttons is significantly more complex than a two-button alert and should be avoided if possible. In fact, if you find that you need to offer people more than two choices, you should consider using an action sheet instead (to learn how to use an action sheet, see "Action Sheet" (page 183)).

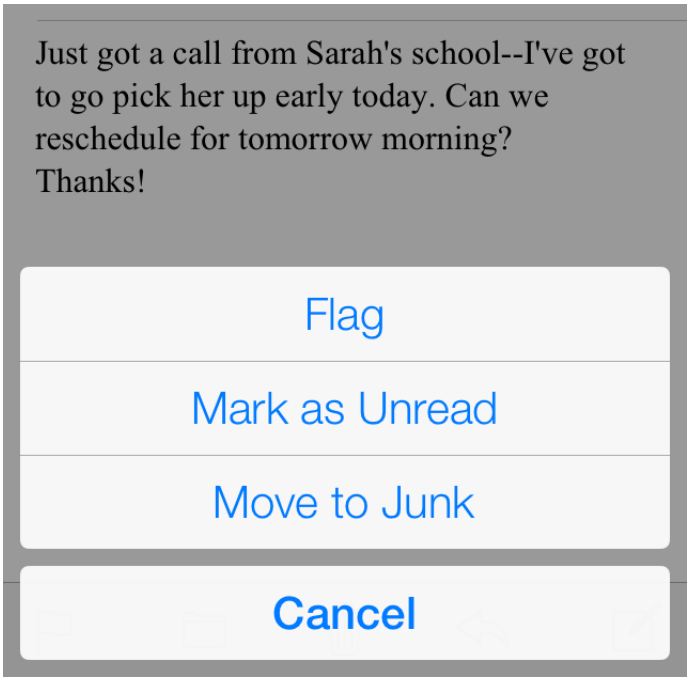
Give alert buttons short, logical titles. The best titles consist of one or two words that make sense in the context of the alert text. Follow these guidelines as you create titles for alert buttons:

- As with all button titles, use title-style capitalization and no ending punctuation.

- Use verbs and verb phrases, such as “Cancel,” “Allow,” “Reply,” or “Ignore” that relate directly to the alert text.
- Use “OK” for a simple acceptance option if there is no better alternative. Avoid using “Yes” or “No.”
- Avoid “you,” “your,” “me,” and “my” as much as possible. Button titles that use these words are often both ambiguous and patronizing.

Action Sheet

An **action sheet** displays a set of choices related to a task the user initiates (shown below on iPhone).



To learn how to define an action sheet in your code, see “Action Sheets”.

Appearance and Behavior

On iPhone, an action sheet always emerges from the bottom of the screen. While an action sheet is visible, iOS dims all other onscreen content.

An action sheet always contains at least two buttons that allow users to choose how to complete their task. When users tap a button, the action sheet disappears. An action sheet doesn’t include a title or explanatory text, because it appears in immediate response to a user action.

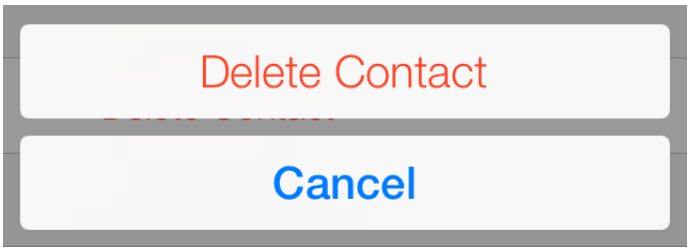
Guidelines

Use an action sheet to:

- Provide alternate ways to complete a task. An action sheet lets you to provide a range of choices that make sense in the context of the current task, without giving these choices a permanent place in the UI.
- Get confirmation before completing a potentially dangerous task. An action sheet prompts users to think about the potentially dangerous effects of the step they're about to take and gives them some alternatives. This type of communication is particularly important on iOS devices because sometimes users tap controls without meaning to.

On iPhone, include a Cancel button so that users can easily and safely abandon the task. Place the Cancel button at the bottom of the action sheet to encourage users to read through all the alternatives before making a choice.

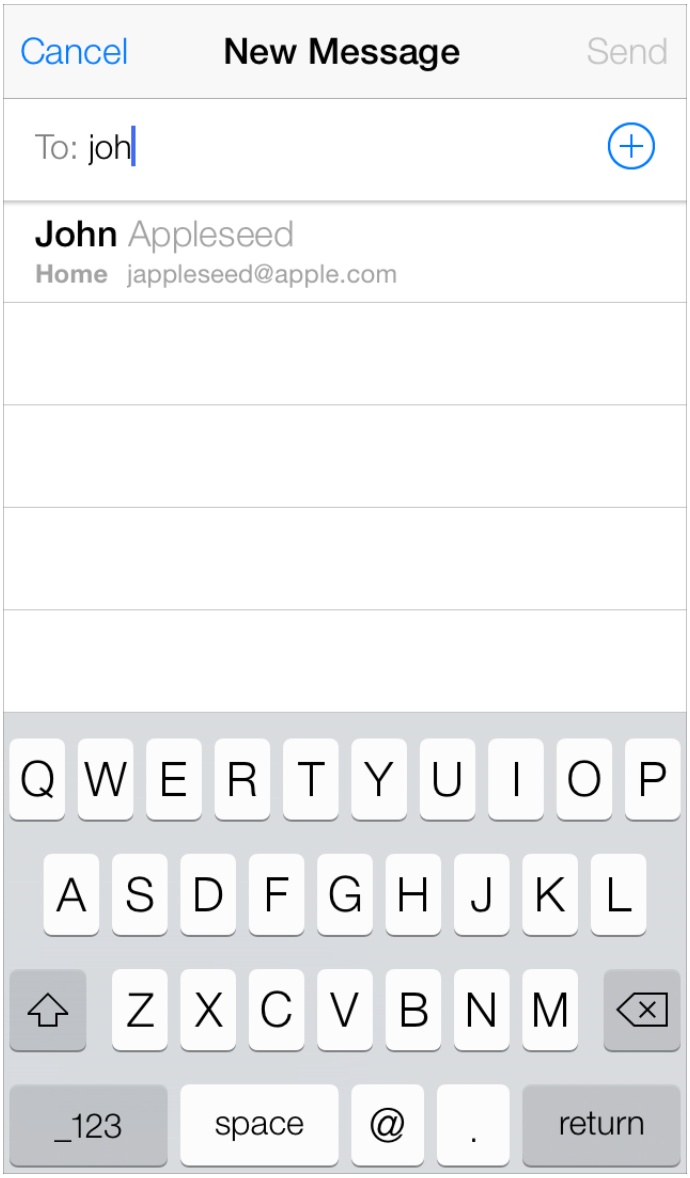
On both devices, use the red button color if a potentially destructive action can be performed. Display a red button at the top of the action sheet, because the closer to the top of the action sheet a button is, the more eye-catching it is. And on iPhone, the farther a destructive button is from the bottom of an action sheet, the less likely users are to tap it when they're aiming for the Home button.



Avoid making users scroll through an action sheet. If you include too many buttons in an action sheet, users must scroll to see all actions. This is a disconcerting experience for users, because they must spend extra time considering each choice. Also, it can be very difficult for users to scroll without inadvertently tapping a button.

Modal View

A **modal view**—that is, a view presented modally—provides self-contained functionality in the context of the current task or workflow.



To learn more about defining a modal view in your code, see *UIViewController Class Reference*.

Appearance and Behavior

A modal view occupies the entire screen, which strengthens the user's perception of entering a separate, transient mode in which they can accomplish something. On iPad, a modal view might also occupy the entire area of a parent view, such as a popover.

A modal view can display text if appropriate, and contains the controls necessary to perform the task. A modal view generally displays a button that completes the task and dismisses the view, and a Cancel button users can tap to abandon the task.

Guidelines

Use a modal view when you need to offer the ability to accomplish a self-contained task related to your app's primary function. A modal view is especially appropriate for a multistep subtask that requires UI elements that don't belong in the main app UI all the time.

On iPhone, coordinate the overall look of a modal view with the appearance of your app. For example, a modal view often includes a navigation bar that contains a title and buttons that cancel or complete the modal view's task. When this is the case, the navigation bar should use the same appearance as the navigation bar in the app.

On all devices, display a title that identifies the task, if appropriate. You might also display text in other areas of the view that more fully describes the task or provides some guidance.

On both devices, choose an appropriate transition style for revealing the modal view. Use a style that coordinates with your app and enhances the user's awareness of the temporary context shift that the modal view represents. To do this, you can specify one of the following transition styles:

- **Vertical.** The modal view slides up from the bottom edge of the screen and slides back down when dismissed. (This is the default transition style.)
- **Flip.** The current view flips horizontally from right to left to reveal the modal view. Visually, the modal view looks as if it is the back of the current view. When the modal view is dismissed, it flips horizontally from left to right, revealing the previous view.

If you decide to vary the transition styles of the modal views in your app, avoid doing so merely for the sake of variety. Users are quick to notice such differences and will assume that they mean something. For this reason, it's best to establish a logical, consistent pattern that users can easily detect and remember, and avoid changing transition styles without a good reason.

Icon and Image Design

Creating Resizable Images

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

You can create a resizable image to customize the background of several standard UI elements, such as popovers, buttons, navigation bars, tab bars, and toolbars (including the items on these bars). Providing resizable images for these elements can result in better app performance.

For many UI elements, you can also specify end caps in addition to a background appearance. An **end cap** defines an area of the image that should not be resized. For example, you might create a resizable image that includes four end caps that define the four corners of a button. When the image is resized to fill the button's background area, the portions defined by the end caps are drawn unchanged.

Depending on the dimensions of the resizable image you supply, iOS either stretches or tiles it as appropriate to fill a UI element's background area. To **stretch** an image means to scale up the image, without regard for its original aspect ratio. Stretching is performant, but it isn't usually desirable for a multipixel image that can distort. To **tile** an image is to repeat the original image as many times as necessary to fill the target area. Tiling is less performant than stretching, but it's the only way to achieve a textured or patterned effect.

As a general rule, you should supply the smallest image (excluding end caps) that will result in the look you want. For example:

- If you want a solid color with no gradient, create a 1 x 1 pixel image.
- If you want a vertical gradient, create an image that has a width of 1 pixel and a height that matches the height of the UI element's background.
- If you want to provide a repeating textured appearance, you need to create an image with dimensions that match the dimensions of the repeating portion of the texture.
- If you want to provide a nonrepeating textured appearance, you need to create a static image with dimensions that match the dimensions of the UI element's background area.

Note: If you're creating resizable images to draw on a Retina display, you also need to supply high-resolution versions of your images. For example, you would also supply a solid-color 2 x 2 pixel image, or a gradient image that has a width of 2 pixels.

Icon and Image Sizes

Every app needs an app icon and a launch image. In addition, some apps need custom icons to represent app-specific content, functions, or modes in navigation bars, toolbars, and tab bars.

Unlike other custom artwork in your app, the icons and images listed in Table 38-1 must meet specific criteria so that iOS can display them properly. In addition, some icon and image files have naming requirements. (If you need to support standard-resolution iPhone or iPod touch devices, divide by 2 the high-resolution sizes listed below.)

Table 38-1 Size (in pixels) of custom icons and images

Description	Size for iPhone 5 and iPod touch (high resolution)	Size for iPhone and iPod touch (high resolution)	Size for iPad (high resolution)	Size for iPad 2 and iPad mini (standard resolution)
App Icon (**required** for all apps)	120 x 120	120 x 120	144 x 144	72 x 72
App Icon for the App Store (**required** for all apps)	1024 x 1024	1024 x 1024	1024 x 1024	1024 x 1024
Launch image (**required** for all apps)	640 x 1136	640 x 960	1536 x 2048 (portrait) 2048 x 1536 (landscape)	768 x 1024 (portrait) 1024 x 768 (landscape)
Small icon for Spotlight search results (recommended)	80 x 80	80 x 80	80 x 80	40 x 40
Small icon for Settings (recommended)	58 x 58	58 x 58	58 x 58	29 x 29
Toolbar and navigation bar icon (optional)	About 44 x 44	About 44 x 44	About 44 x 44	About 22 x 22

Description	Size for iPhone 5 and iPod touch (high resolution)	Size for iPhone and iPod touch (high resolution)	Size for iPad (high resolution)	Size for iPad 2 and iPad mini (standard resolution)
Tab bar icon (optional)	About 60 x 60 (maximum: 96 x 64)	About 60 x 60 (maximum: 96 x 64)	About 60 x 60 (maximum: 96 x 64)	About 30 x 30 (maximum: 48 x 32)
Default Newsstand cover icon for the App Store (**required** for Newsstand apps)	At least 1024 pixels on the longest edge	At least 1024 pixels on the longest edge	At least 1024 pixels on the longest edge	At least 1024 pixels on the longest edge
Web clip icon (recommended for web apps and websites)	120 x 120	120 x 120	144 x 144	72 x 72

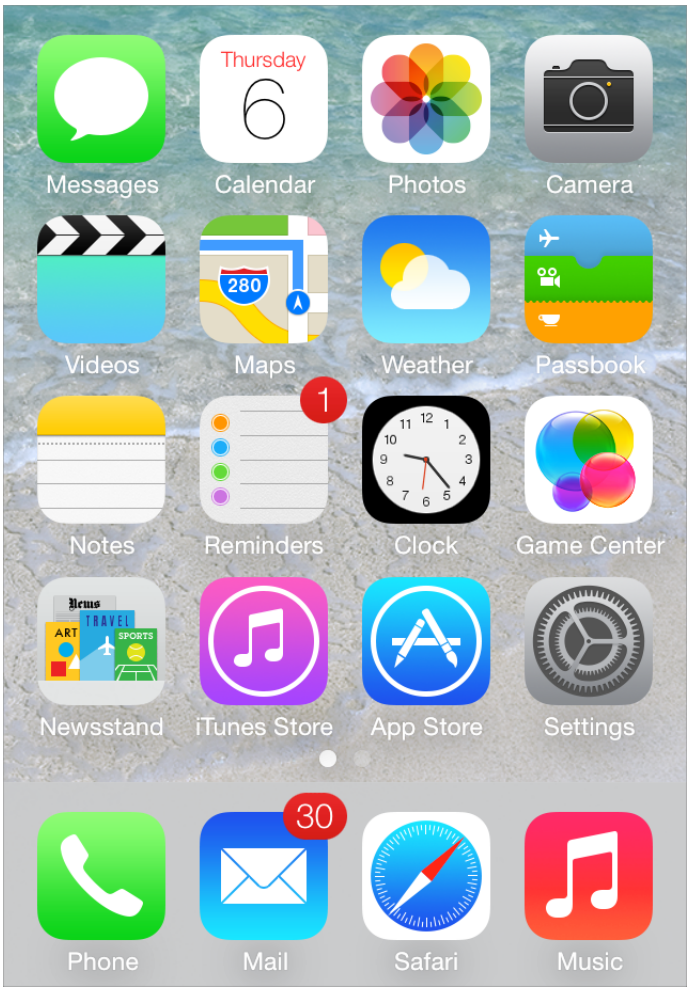
For all images and icons, the PNG format is recommended. You should avoid using interlaced PNGs.

The standard bit depth for icons and images is 24 bits—that is, 8 bits each for red, green, and blue—plus an 8-bit alpha channel.

You don't need to constrain your palette to web-safe colors.

App Icon

Every app needs a beautiful, memorable app icon that attracts people in the App Store and stands out on their Home screen. iOS can use versions of the app icon in Game Center, search results, Settings, and to represent app-created documents.



The **app icon** represents an app in the App Store and on the Home screen. In this icon, branding and strong visual design should come together into a compact, instantly recognizable, attractive package.

For the best results, enlist the help of a professional graphic designer. An experienced graphic designer can help you develop an overall visual style for your app and apply that style to all the icons and images in it.

Use universal imagery that people will easily recognize. In general, avoid focusing on a secondary or obscure aspect of an element.

Embrace simplicity. In particular, avoid cramming lots of different images into your icon. Try to find a single element that expresses the essence of your app. Start with a basic shape and add details cautiously. If an icon's content or shape is overly complex, the details can become confusing and may appear muddy at smaller sizes.

In general, avoid using "greek" text or wavy lines to suggest text. If you want to show text in your icon, but you don't want to draw attention to the words themselves, start with actual text and make it hard to read by shrinking it or doubling the layers.

Consider creating an abstract interpretation of your app's main idea. Creating an artistically enhanced version of an idea or a real object can help you emphasize the aspects of the subject that you want users to notice.

Don't use iOS interface elements in your artwork. You don't want users to confuse your icons or images with the iOS UI.

Don't use replicas of Apple hardware products in your artwork. The symbols that represent Apple products are copyrighted and can't be reproduced in your icons or images. In general, it's a good idea to avoid replicas of any specific devices in your artwork, because these designs change frequently and icons or images that are based on them can quickly look dated.

Don't reuse iOS app icons in your interface. It can be confusing to users to see the same icon used to mean slightly different things in multiple locations throughout the system.

If you want to portray real substances, do it accurately. Icons or images that represent real objects should also look as though they are made of real materials and have real mass. Realistic icons accurately replicate the characteristics of substances such as fabric, glass, paper, and metal, and convey an object's weight and feel.

Avoid transparency. Transparency in an image can help depict glass or plastic, but it can be tricky to use convincingly in an icon. Overall, the app icon should be opaque.

With the exception of the App Store icon—which must be named `iTunesArtwork`—you can name an app icon anything you want. As long as you use the `CFBundleIcons` key to declare the names and you add the `@2x` suffix to the names of all high-resolution icons, iOS chooses an icon based on whether its size is appropriate for the intended usage. To learn more about icon naming, see "App Icons" in *iOS App Programming Guide*.

Create different sizes of the app icon for different devices. If you're creating a universal app, you need to supply app icons in all four sizes.

For iPhone and iPod touch both of these sizes are required:

- 120 x 120 pixels

- 60 x 60 pixels (standard resolution)

For iPad, both of these sizes are required:

- 144 x 144
- 72 x 72 pixels (standard resolution)

Give your app icon a discernible background. Icons with visible backgrounds look best on the Home screen primarily because of the rounded corners iOS adds. This is because uniformly rounded corners ensure that all the icons on a user's Home screen have a consistent appearance that invites tapping. If you create an icon with a background that disappears when it's viewed on the Home screen, users don't see the rounded corners. Such icons often don't look tappable and tend to interfere with the orderly symmetry of the Home screen that users appreciate.

Be sure your image completely fills the required area. If your image boundaries are smaller than the recommended sizes, or you use transparency to create “see-through” areas, your icon can appear to float on a black background with rounded corners.

An icon that appears to float on a visible black background looks especially unattractive on a Home screen that displays a custom picture.

Create a large version of your app icon for display in the App Store. Although it’s important that this version be instantly recognizable as your app icon, it can be subtly richer and more detailed. There are no visual effects added to this version of your app icon.

For the App Store, create a large version of your app icon in two sizes so that it looks good on all devices:

- 1024 x 1024 pixels
- 512 x 512 pixels (standard resolution)

Be sure to name this version of your app icon `iTunesArtwork` and `iTunesArtwork@2x`, respectively.

If you’re developing an app for ad-hoc distribution (that is, to be distributed in-house only, not through the App Store), you must also provide the large versions of your app icon. This icon identifies your app in iTunes.

iOS might also use the large image in other ways. In an iPad app, for example, iOS uses the large image to generate the large document icon.

Document Icons

If your iOS app creates documents of a custom type, you want users to be able to recognize these documents at a glance. You don't need to design a custom icon for this purpose because iOS uses your app icon to create document icons for you.

Small Icons

Every app should supply a small icon that iOS can display when the app name matches a term in a Spotlight search. Apps that supply settings should also supply this icon to identify them in the built-in Settings app.

This icon should clearly identify your app so that people can recognize it in a list of search results or in Settings.

You can name your icon anything you want as long as you use the `CFBundleIcons` key to declare the names and you add the `@2x` suffix to the names of all high-resolution icons. You can use custom names because iOS chooses an icon based on whether its size is appropriate for the intended usage. To learn more about icon naming, see “App Icons” in *iOS App Programming Guide*.

For all devices, supply separate icons for Settings and Spotlight search results. If you don’t provide this icon, iOS might shrink your app icon for display in these locations.

For Settings on iPhone, iPod touch, and iPad create two icons that measure:

- 58 x 58 pixels
- 29 x 29 pixels (standard resolution)

For Spotlight search results on iPhone, iPod touch, and iPad create two icons that measure:

- 80 x 80 pixels
- 40 x 40 pixels (standard resolution)

Launch Images

The launch image is a simple placeholder image that iOS displays when your app starts up. Because the launch image appears so quickly, users get the impression that your app is fast and responsive.

To enhance the user's experience at app launch, you must provide at least one launch image.

> **Note:** In general, an iPhone app should include a launch image in portrait orientation; an iPad app should include one launch image in portrait orientation and one launch image in landscape orientation.

Because iOS lets you supply different launch images for different usages, you give each image a name that specifies how it should be used. The format of the launch image filename includes modifiers you use to specify the device, resolution, and orientation of the image. To learn how to name launch images appropriately, see "App Launch (Default) Images" in *iOS App Programming Guide*.

Supply a launch image to improve user experience.

The launch image isn't an opportunity to provide:

- An "app entry experience," such as a splash screen
- An About window
- Branding elements, unless they are a static part of your app's first screen

Because users are likely to switch among apps frequently, you should make every effort to cut launch time to a minimum, and you should design a launch image that downplays the experience rather than drawing attention to it.

Generally, design a launch image that is identical to the first screen of the app.

Exceptions:

Text. The launch image is static, so any text you display in it will not be localized.

UI elements that might change. Avoid including elements that might look different when the app finishes launching, so that users don't experience a flash between the launch image and the first app screen.

For iPhone and iPod touch launch images, include the status bar region. Create launch images of the following sizes.

For iPhone 5 and iPod touch (5th generation):

- 640 x 1136 pixels

For other iPhone and iPod touch devices:

- 640 x 960 pixels
- 320 x 480 pixels (standard resolution)

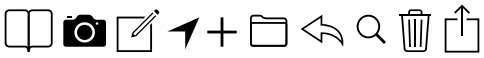
For iPad launch images, don't include the status bar region. Create launch images of these sizes (most iPad apps should supply a launch image for each orientation):

- For portrait:
  - 1536 x 2008 pixels
  - 768 x 1004 pixels (standard resolution)
- For landscape:
  - 2048 x 1496 pixels
  - 1024 x 748 pixels (standard resolution)

If you think that following these guidelines will result in a plain, boring launch image, you're right. Remember, the launch image is not meant to provide an opportunity for artistic expression. It's solely intended to enhance the user's perception of your app as quick to launch and immediately ready for use.

Bar Button Icons

iOS defines lots of standard bar-button icons, such as Refresh, Share, Add, and Favorites. If you need to represent custom topics or actions, follow these guidelines to design your own icons.



As much as possible, you should use the system-provided buttons and icons to represent standard tasks in your app. For a complete list of standard buttons and icons, and guidelines on how to use them, see "Toolbar and Navigation Bar Buttons" (page 123) and "Tab Bar Icons" (page 126).

Of course, not every task or mode in your app can be represented by a standard icon. As you think about the best way to convey the task or mode, aim for a design that is:

- **Simple and streamlined.** Too many details can make an icon appear sloppy or indecipherable.
- **Not easily mistaken for one of the system-provided icons.** Users should be able to distinguish your custom icon from the standard icons at a glance.
- **Readily understood and widely acceptable.** Strive to create a symbol that most users will interpret correctly and that no users will find offensive.

> **Important:** Be sure to avoid using images that replicate Apple products in your designs. These symbols are copyrighted and product designs can change frequently.

A custom icon that you provide for a toolbar, navigation bar, or tab bar is also known as a **template** image, because iOS uses it as a mask to create the icon you see in your app. If you ceate a full-color template image, iOS ignores the color.

After you've decided on the appearance of your icon, follow these guidelines as you create it:

- Use pure white with appropriate alpha transparency.
- Don't include a drop shadow.
- Use anti-aliasing.
- If you decide to add a bevel, be sure that it's 90° (to help you do this, imagine a light source positioned at the top of the icon).

For toolbar and navigation bar icons on iPhone, iPod touch, and iPad, create an icon in the following sizes:

- About 44 x 44 pixels
- About 22 x 22 pixels (standard resolution)

For tab bar icons on iPhone, iPod touch, and iPad, create an icon in the following sizes:

- About 60 x 60 pixels (96 x 64 pixels maximum)
- About 30 x 30 pixels (48 x 32 pixels maximum) for standard resolution

Don't include text in a custom tab bar icon. Instead, use the tab bar item APIs to set the title for each tab (for example, `initWithTitle:image:tag:`). If you need to adjust the automatic layout of the title, you can use the title adjustment APIs (such as `setTitlePositionAdjustment:`).

> **Note:** For a tab bar icon, you can also provide a set of two fixed images, one for the unselected appearance and one for the selected appearance.

Give all icons in a bar a similar visual weight. Aim to balance the overall size, level of detail, and use of solid regions across all icons that can appear in a specific bar. In general, it doesn't look good to combine in the same bar icons that are large and blocky, and completely filled, with icons that are small, detailed, and unfilled.

Newsstand Icons

If your app uses Newsstand Kit to publish subscription-based periodical content, you need to provide icons for display in the App Store and on people's devices.

Important: The aspect ratio of all Newsstand icons should be between 1:2 and 2:1.

All Newsstand icons must be flat and have 90° corners.

Don't add perspective to any of your Newsstand icons.

All Newsstand apps need to supply a Newsstand cover icon that represents the default cover art in the App Store. The long edge of this icon should measure at least 1024 pixels (512 pixels for standard-resolution devices). Note that this icon is separate from the app icon that all iOS apps must provide.

A default Newsstand cover icon should be a generalized facsimile of the cover of a typical issue, which focuses on the parts of the cover that are fairly consistent from issue to issue. For example:

- Avoid adding to the default cover icon elements that users would never see on an actual cover, such as a message to "tap here for the latest issue".
- Avoid using artwork or headlines that are seasonal or topical, such as images related to holidays or headlines that refer to current events.

In particular, don't reuse the cover of a previous issue for your default Newsstand cover icon, because users might confuse your app with a specific issue.

In addition to the default Newsstand cover icon, you also need to supply a separate icon that accurately represents each new issue so it can appear on the Newsstand shelf and in the multitasking UI on an iOS device. Unlike the default cover icon, each per-issue icon should display details about the contents of a specific issue.

It's recommended that you create a single large icon for each issue, and allow iOS to scale it for display in both places (the icon displayed in Newsstand is larger than the icon displayed in the multitasking UI). Depending on the precise location of an issue on a Newsstand shelf, iOS might also add perspective to the icon so that it matches the realistic look of the shelf.

Specifically, you should create a per-issue icon whose long edge measures at least 1024 pixels (512 pixels for standard-resolution devices). To display the current issue's icon on the Newsstand shelf and in the multitasking UI, iOS scales your large icon to the following sizes:

Table 42-1 Maximum scaled sizes for the long edges of per-issue icons

Device	Scaled long-edge size (Newsstand shelf)	Scaled long-edge size (multitasking UI)
iPhone and iPod touch	180 pixels (90 pixels for standard resolution)	114 pixels (57 pixels for standard resolution)
iPad	252 pixels (126 pixels for standard resolution)	144 pixels (72 pixels for standard resolution)

In addition to providing icons, you use keys in your app's `Info.plist` file to define how the icons should appear on iOS devices.

- First, use the binding type key to indicate whether your content is a magazine or a newspaper.
- Then, use the binding edge key to specify the visual enhancements that iOS should add to the icon, such as the fold at the bottom edge of a standard newspaper.

For more information about these keys and their values, see "Contents of the UINewsstandIcon Dictionary" in *Information Property List Key Reference* .

When you specify the magazine binding type, iOS adds the appearance of multiple pages and a shadow that suggests thickness. You must also specify a left or right binding edge for a magazine icon, to indicate the edge that should receive the stapled binding appearance.

For example, suppose that you supply a per-issue icon similar to this:

If you specify the left binding edge, iOS adds the stapled binding appearance to the left edge and the multiple-page appearance to the right edge.

When you specify the newspaper binding type, iOS adds the appearance of additional copies of the paper that are stacked beneath the current issue.

If your newspaper is standard size, you can specify a bottom binding edge to give your icon the appearance of a fold at its bottom edge. If your newspaper is tabloid-size—that is, approximately half the size of a broadsheet—you can specify left, right, or no binding edge to avoid the addition of the fold appearance.

For example, suppose that your per-issue newspaper icon looks similar to this:

If you specify the bottom binding edge, iOS adds the appearance of a fold to the bottom edge. (iOS adds the stacked-paper appearance regardless of the value you supply for the binding edge key.)

For additional information about setting up a Newsstand app, see *iTunes Connect Developer Guide* .

Web Clip Icons

If you have a web app or a website, you can provide a custom icon that users can display on their Home screens using the web clip feature. Users tap the icon to reach your web content in one easy step. You can create an icon that represents your website as a whole or an icon that represents a single webpage.

If your web content is distinguished by a familiar image or recognizable color scheme, it makes sense to incorporate it in your icon. However, to ensure that your icon looks great on the device, you should also follow the guidelines in this section. (To learn how to add code to your web content to provide a custom icon, see *Safari Web Content Guide*.)

For iPhone and iPod touch, create icons that measure:

- 120 x 120 pixels
- 60 x 60 pixels (standard resolution)

For iPad, create icons that measure:

- 144 x 144 pixels
- 72 x 72 pixels (standard resolution)

> **Note:** You can prevent the addition of any effects by naming your icon `apple-touch-icon-precomposed.png`.

Document Revision History

This table describes the changes to *iOS Human Interface Guidelines* .

Date	Notes
2013-06-20	Reorganized and updated for iOS 7.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010