App Programming Guide for iOS

Developer

Contents

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Figures, Tables, and Listings

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

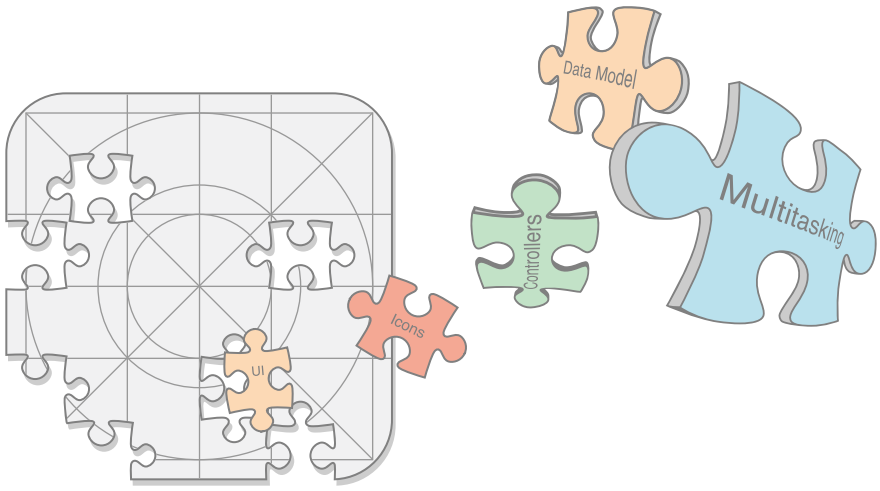
2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

SwiftObjective-C

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

About iOS App Architecture

Apps need to work with the iOS to ensure that they deliver a great user experience. Beyond just a good design for your app's design and user interface, a great user experience encompasses many other factors. Users expect iOS apps to be fast and responsive while expecting the app to use as little power as possible. Apps need to support all of the latest iOS devices while still appearing as if the app was tailored for the current device. Implementing all of these behaviors can seem daunting at first but iOS provides the help you need to make it happen.



This document highlights the core behaviors that make your app work well on iOS. You might not implement every feature described in this document but you should consider these features for every project you create.

> **Note:** Development of iOS apps requires an Intel-based Macintosh computer with the iOS SDK installed. For information about how to get the iOS SDK, go to the iOS Dev Center.

At a Glance

When you are ready to take your ideas and turn them into an app, you need to understand the interactions that occur between the system and your app.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Apps Are Expected to Support Key Features

The system expects every app to have some specific resources and configuration data, such as an app icon and information about the capabilities of the app. Xcode provides some information with every new project but you must any resource files and you must make sure the information in your project is correct before submitting your app.

Relevant Chapter: Expected App Behaviors (page 11)

Apps Follow Well-Defined Execution Paths

From the time the user launches an app to the time it quits, apps follow a well-defined execution path. During the life of an app, it can transition between foreground and background execution, it can be terminated and relaunched, and it can go to sleep temporarily. Each time it transitions to a new state, the expectations for the app change. A foreground app can do almost anything but background apps must do as little as possible. You use the state transitions to adjust your app's behaviors accordingly.

Relevant Chapter: The App Life Cycle (page 22), Strategies for Handling App State Transitions (page 48)

Apps Must Run Efficiently in a Multitasking Environment

Battery life is important for users, as is performance, responsiveness, and a great user experience. Minimizing your app's usage of the battery ensures that the user can run your app all day without having to recharge the device, but launching and being ready to run quickly are also important. The iOS multitasking implementation offers good battery life without sacrificing the responsiveness and user experience that users expect, but the implementation requires apps to adopt system-provided behaviors.

Relevant Chapters: Background Execution (page 32), Strategies for Handling App State Transitions (page 48)

Communication Between Apps Follows Specific Pathways

For security, iOS apps run in a sandbox and have limited interactions with other apps. When you want to communicate with other apps on the system, there are specific ways to do so.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Relevant Chapter: Inter-App Communication (page 97)

Performance Tuning is Important for Apps

Every task performed by an app has a power cost associated with it. Apps that drain the user's battery create a negative user experience and are more likely to be deleted than those that appear to run for days on a single charge. So be aware of the cost of different operations and take advantage of power-saving measures offered by the system.

Relevant Chapter: Performance Tips (page 106)

How to Use This Document

This document is not a beginner's guide to creating iOS apps. It is for developers who are ready to polish their app before putting it in the App Store. Use this document as a guide to understanding how your app interacts with the system and what it must do to make those interactions happen smoothly.

Prerequisites

This document provides detailed information about iOS app architecture and shows you how to implement many app-level features. This book assumes that you have already installed the iOS SDK, configured your development environment, and understand the basics of creating and implementing an app in Xcode.

If you are new to iOS app development, read *Start Developing iOS Apps Today*. That document offers a step-by-step introduction to the development process to help you get up to speed quickly. It also includes a hands-on tutorial that walks you through the app-creation process from start to finish, showing you how to create a simple app and get it running quickly.

See Also

If you are learning about iOS, read *iOS Technology Overview* to learn about the technologies and features you can incorporate into your iOS apps.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Expected App Behaviors

SwiftObjective-C

Every new Xcode project comes configured to run right away in iOS Simulator or on a device. But simply being able to run on a device does not mean that your app is ready to ship on the App Store. Every app requires some amount of customization to ensure a good experience for the user. Customizations can range from providing an icon for your app to making architectural-level decisions about how your app presents and uses information. This chapter describes the behaviors that all apps are expected to handle and that you should consider early in the planning process.

Providing the Required Resources

Every app you create must have the following set of resources and metadata so that it can be displayed properly on iOS devices:

- **An information property-list file.** The `Info.plist` file contains metadata about your app, which the system uses to interact with your app. Xcode creates this file for you automatically based on your project's configuration and settings. If you want to view or modify the contents of this file directly, you can do so from the Info tab of your project. For information about editing this file and for recommendations about what keys you should include, see The Information Property List File (page 15).
- **A declaration of the app's required capabilities.** Every app must declare the hardware capabilities or features that it requires to run. The App Store uses this information to determine whether or not a user can run your app on a specific device. You can edit your app's list of requirements using the Required device capabilities entry in the Info tab of your project. For information on how to configure this key, see Declaring the Required Device Capabilities (page 16).
- **One or more icons.** The system displays your app icon on the home screen of a user's device. The system may also use other versions of your icon in the Settings app or when displaying the results of a search. For information about how to specify app icons, see App Icons (page 16).
- **One or more launch images.** When an app is launched, the system displays a temporary image until the app is able to present its user interface. This temporary image is your app's launch image and it provides the user with immediate feedback that your app is launching and will be ready soon. You must provide at least one launch image for your app and you may provide additional launch images to address specific scenarios. For information about creating your launch images, see App Launch (Default) Images (page 17).

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

These resources are required for all apps but are not the only ones you should include. There are many keys that Xcode does not include in your app's `Info.plist` file by default. Most of the additional keys are important only if you incorporate specific features into your app. For example, an app that uses the microphone should include the `NSMicrophoneUsageDescription` key and provide the user with information about how the app intends to use it.

The App Bundle

When you build your iOS app, Xcode packages it as a bundle. A *bundle* is a directory in the file system that groups related resources together in one place. An iOS app bundle contains the app executable file and supporting resource files such as app icons, image files, and localized content. Table 1-1 lists the contents of a typical iOS app bundle, which for demonstration purposes is called `MyApp`. This example is for illustrative purposes only. Some of the files listed in this table may not appear in your own app bundles.

Table 1-1 A typical app bundle

File	Example	Description
App executable	`MyApp`	The executable file contains your app's compiled code. The name of your app's executable file is the same as your app name minus the `.app` extension. This file is required.
The information property list file	`Info.plist`	The `Info.plist` file contains configuration data for the app. The system uses this data to determine how to interact with the app. This file is required and must be called `Info.plist`. For more information, see The Information Property List File (page 15).

App icons	`Icon.png`
	`Icon@2x.png`
	`Icon-Small.png`
	`Icon-Small@2x.png`
		Your app icon is used to represent your app on the device's Home screen. Other icons are used by the system in appropriate places. Icons with `@2x` in their filename are intended for devices with Retina displays.
		An app icon is required. For information about specifying icon image files, see App Icons (page 16).

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

File	Example	Description

Launch images	`Default.png`
	`Default-Portrait.png`
	`Default-Landscape.png`
		The system uses this file as a temporary background while your app is launching. It is removed as soon as your app is ready to display its user interface.
		At least one launch image is required. For information about specifying launch images, see App Launch (Default) Images (page 17).

Storyboard files (or nib files)	`MainBoard.storyboard`	Storyboards contain the views and view controllers that the app presents on screen. Views in a storyboard are organized according to the view controller that presents them. Storyboards also identify the transitions (called segues) that take the user from one set of views to another. The name of the main storyboard file is set by Xcode when you create your project. You can change the name by assigning a different value to the `UIMainStoryboardFile` key in the `Info.plist` file.) Apps that use nib files instead of storyboards can replace the `UIMainStoryboardFile` key with the `NSMainNibFile` key and use that key to specify their main nib file. The use of storyboards (or nib files) is optional but recommended.
Ad hoc distribution icon	`iTunesArtwork`	If you are distributing your app ad hoc, include a 512 x 512 pixel version of your app icon. This icon is normally provided by the App Store from the materials you submit to iTunes Connect. However, because apps distributed ad hoc do not go through the App Store, your icon must be present in your app bundle instead. iTunes uses this icon to represent your app. (The file you specify should be the same one you would have submitted to the App Store, if you were distributing your app that way.) The filename of this icon must be `iTunesArtwork` and must not include a filename extension. This file is required for ad hoc distribution but is optional otherwise.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

File	Example	Description
Settings bundle	`Settings.bundle`	If you want to expose custom app preferences through the Settings app, you must include a settings bundle. This bundle contains the property list data and other resource files that define your app preferences. The Settings app uses the information in this bundle to assemble the interface elements required by your app. This bundle is optional. For more information about preferences and specifying a settings bundle, see *Preferences and Settings Programming Guide*.
Nonlocalized resource files	`sun.png` `mydata.plist`	Nonlocalized resources include things like images, sound files, movies, and custom data files that your app uses. All of these files should be placed at the top level of your app bundle.

Subdirectories for localized resources	`en.lproj`
	`fr.lproj`
	`es.lproj`
		Localized resources must be placed in language-specific project directories, the names for which consist of an ISO 639-1 language abbreviation plus the `.lproj` suffix. (For example, the `en.lproj`, `fr.lproj`, and `es.lproj` directories contain resources localized for English, French, and Spanish.)
		
		An iOS app should be internationalized and have a *language* `.lproj` directory for each language it supports. In addition to providing localized versions of your app's custom resources, you can also localize your app icon, launch images, and Settings icon by placing files with the same name in your language-specific project directories.
		
		For more information, see Internationalizing Your App (page 20).

For more information about the structure of an iOS app bundle, see *Bundle Programming Guide*. For information about how to load resource files from your bundle, see *Resource Programming Guide*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

The Information Property List File

Xcode uses information from the General, Capabilities, and Info tabs of your project to generate an information property list (`Info.plist`) file for your app at compile time. The `Info.plist` file is a structured file that contains critical information about your app's configuration. It is used by the App Store and by iOS to determine your app's capabilities and to locate key resources. Every app must include this file.

Although the `Info.plist` file provided by Xcode includes default values for all of the required entries, most apps require some changes or additions. Whenever possible, use the General and Capabilities tabs to specify the configuration information for your app. Those tabs contain the most common configuration options available for apps. If you do not see a specific option on either of those tabs, use the Info tab.

For options where Xcode does not provide a custom configuration interface, you must provide appropriate keys and values. The Custom iOS Target Properties section of the Info tab contains a summary of the entries to be included in the `Info.plist` file. By default, Xcode displays human-readable descriptions of the intended feature but each feature actually corresponds to a unique key in the `Info.plist` file. Most keys are optional and used infrequently, but there are a handful of keys that you should consider when defining any new project:

- **Declare your app's required capabilities in the Info tab.** The Required device capabilities section contains information about the device-level features that your app requires to run. The App Store uses the information in this entry to determine the capabilities of your app and to prevent it from being installed on devices that do not support features your app requires. For more information, see Declaring the Required Device Capabilities (page 16).
- **Apps that require a persistent Wi-Fi connection must declare that fact.** If your app talks to a server across the network, you can add the Application uses Wi-Fi entry to the Info tab of your project. This entry corresponds to the `UIRequiresPersistentWiFi` key in the `Info.plist` file. Setting this key to `YES` prevents iOS from closing the active Wi-Fi connection when it has been inactive for an extended period of time. This key is recommended for all apps that use the network to communicate with a server.
- **Newsstand apps must declare themselves as such.** Include the `UINewsstandApp` key to indicate that your app presents content from the Newsstand app.
- **Apps that define custom document types must declare those types.** Use the Document Types section of the Info tab to specify icons and UTI information for the document formats that you support. The system uses this information to identify apps capable of handling specific file types. For more information about adding document support to your app, see *Document-Based App Programming Guide for iOS*.
- **Apps can declare any custom URL schemes they support.** Use the URL Types section of the Info tab to specify the custom URL schemes that your app handles. Apps can use custom URL schemes to communicate with each other. For more information about how to implement support for this feature, see Using URL Schemes to Communicate with Apps (page 99).

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- **Apps should provide usage descriptions for certain app features.** Whenever there is a privacy concern about an app accessing a user's data or a device's capabilities, iOS frameworks prompt the user and request permission for your app to use the feature. Apps that use these features should provide privacy usage descriptions that explain what your app plans to do with the corresponding data. For information about the features that require user permission, see Table 1-2 (page 19).

For detailed information about the keys and values you can include in the `Info.plist` file, see *Information Property List Key Reference*.

Declaring the Required Device Capabilities

All apps must declare the device-specific capabilities they need to run. Xcode includes a Required device capabilities entry in your project's Info tab and populates it with some minimum requirements. You can add values to this entry to specify additional requirements for your app. The Required device capabilities entry corresponds to the `UIRequiredDeviceCapabilities` key in your app's `Info.plist` file.

The value of the `UIRequiredDeviceCapabilities` key is either an array or dictionary that contains additional keys identifying features your app requires (or specifically prohibits). If you specify the value of the key using an array, the presence of a key indicates that the feature is required; the absence of a key indicates that the feature is not required and that the app can run without it. If you specify a dictionary instead, each key in the dictionary must have a Boolean value that indicates whether the feature is required or prohibited. A value of `true` indicates the feature is required and a value of `false` indicates that the feature must *not* be present on the device. If a given capability is optional for your app, do not include the corresponding key in the dictionary.

For detailed information on the values you can include for the `UIRequiredDeviceCapabilities` key, see *Information Property List Key Reference*.

App Icons

Every app must provide an icon to be displayed on a device's Home screen and in the App Store. An app may actually specify several different icons for use in different situations. For example, an app can provide a small icon to use when displaying search results and can provide a high-resolution icon for devices with Retina displays.

New Xcode projects include image asset entries for your app's icon images. To add icons, assign the corresponding image files to the image assets of your project. At build time, Xcode adds the appropriate keys to your app's `Info.plist` file and places the images in your app bundle.

For information about designing your app icons, including the sizes of those icons, see *iOS Human Interface Guidelines*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

App Launch (Default) Images

When the system launches an app for the first time on a device, it temporarily displays a static launch image on the screen. This image is your app's launch image and it is a resource that you specify in your Xcode project. Launch images provide the user with immediate feedback that your app has launched while giving your app time to prepare its initial user interface. When your app's window is configured and ready to be displayed, the system swaps out the launch image for that window.

When a recent snapshot of your app's user interface is available, the system prefers the use of that image over the use of your app's launch images. The system takes a snapshot of your app's user interface when your app transitions from the foreground to the background. When your app returns to the foreground, it uses that image instead of a launch image whenever possible. In cases where the user has killed your app or your app has not run for a long time, the system discards the snapshot and relies once again on your launch images.

New Xcode projects include image asset entries for your app's launch images. To add launch images, add the corresponding image files to the image assets of your project. At build time, Xcode adds the appropriate keys to your app's `Info.plist` file and places the images in your app bundle.

For information about designing your app's launch images, including the sizes of those images, see *iOS Human Interface Guidelines* .

Supporting User Privacy

Maintaining user privacy should be an important consideration when designing your app. Most iOS devices contain user and device data that users might not want to expose to apps or external entities. Remember that the user might delete your app if it uses data in an inappropriate way.

Access user or device data only with the user's informed consent obtained in accordance with applicable law. In addition, take appropriate steps to protect user and device data and be transparent about how you use it. Here are some best practices that you can take:

- Review guidelines from government or industry sources, including the following documents:
  - The Federal Trade Commission's report on mobile privacy: Mobile Privacy Disclosures: Building Trust Through Transparency.
  - The EU Data Protection Commissioners' Opinion on data protection for Mobile Apps: http://ec.europa.eu/justice/data-protection/article-29/documentation/opinion-recommendation/files/2013/wp202_en.pdf
  - The Japanese Ministry of Internal Affairs and Communications' Smartphone Privacy Initiatives:
    - Smartphone Privacy Initiative (2012):

      English: http://www.soumu.go.jp/main_sosiki/joho_tsusin/eng/presentation/pdf/Initiative.pdf

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Japanese: http://www.soumu.go.jp/main_content/000171225.pdf

- Smartphone Privacy Initiative II (2013):

  English: http://www.soumu.go.jp/main_sosiki/joho_tsusin/eng/presentation/pdf/Summary_II.pdf

  Japanese: http://www.soumu.go.jp/main_content/000247654.pdf

- The California State Attorney General's recommendations for mobile privacy: Privacy on the Go: Recommendations for the Mobile Ecosystem

These reports provide helpful recommendations for protecting user privacy. You should also review these documents with your company's legal counsel.

- Request access to user or device data that is protected by the iOS system authorization settings at the time the data is needed. Consider supplying a usage description string in your app's `Info.plist` file explaining why your app needs that data. Data protected by iOS system authorization settings includes location data, contacts, calendar events, reminders, photos, and media; see Table 1-2 (page 19). Provide reasonable fallback behavior in situations where the user does not grant access to the requested data.
- Be transparent with users about how their data is going to be used. For example, you should specify a URL for your privacy policy or statement with your iTunes Connect metadata when you submit your app, and you might also want to summarize that policy in your app description.

  For more information about providing your app's privacy policy in iTunes Connect, see Adding an App in iTunes Connect.
- Give the user control over their user or device data. Provide settings so that the user can disable access to certain types of sensitive information as needed.
- Request and use the minimum amount of user or device data needed to accomplish a given task. Do not seek access to or collect data for non obvious reasons, for unnecessary reasons, or because you think it might be useful later.
- Take reasonable steps to protect the user and device data that you collect in your apps. When storing such information locally, try to use the iOS data protection feature (described in Protecting Data Using On-Disk Encryption (page 65)) to store it in an encrypted format. And try to use HTTPS when sending user or device data over the network.
- If your app uses the `ASIdentifierManager` class, you must respect the value of its `advertisingTrackingEnabled` property. And if that property is set to `NO` by the user, then use the `ASIdentifierManager` class only for Limited Advertising Purposes. "Limited Advertising Purposes" means frequency capping, attribution, conversion events, estimating the number of unique users, advertising fraud detection, debugging for advertising purposes only, and other uses for advertising that may be permitted by Apple in Documentation for the Ad Support APIs.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- If you have not already done so, stop using the unique device identifier (UDID) provided by the `uniqueIdentifier` property of the `UIDevice` class. That property was deprecated in iOS 5.0, and the App Store does not accept new apps or app updates that use that identifier. Instead, apps should use the `identifierForVendor` property of the `UIDevice` class or the `advertisingIdentifier` property of the `ASIdentifierManager` class, as appropriate.
- If your app supports audio input, configure your audio session for recording only at the point where you actually plan to begin recording. Do not configure your audio session for recording at launch time if you do not plan to record right away. In iOS 7, the system alerts users when apps configure their audio session for recording and gives the user the option to disable recording for your app.

Table 1-2 lists the types of data authorizations supported by iOS. Using the services listed in this table causes an alert to be displayed to the user requesting permission to do so. You can determine if the user authorized your app for a service using the API listed for each item. You should view this table as a starting point for your app's own privacy behaviors and not as a finite checklist. The contents of this table may evolve over time.

Table 1-2 Data protected by system authorization settings

Data	System authorization support
Location	The current authorization status for location data is available from the `authorizationStatus` class method of `CLLocationManager`. In requesting authorization in iOS 8 and later, you must use the `requestWhenInUse–Authorization` or `requestAlwaysAuthorization` method and include the `NSLocationWhenInUseUsageDescription` or `NSLocationAlways–UsageDescription` key in your `Info.plist` file to indicate the level of authorization you require.
Photos	The authorization status for photo data is available from the `authorizationStatus` method of `ALAssetsLibrary`. To inform the user about how you intend to use this information, include the `NSPhotoLibraryUsageDescription` key in your `Info.plist` file.
Music, video, and other media assets	The authorization status for media assets is available from the `authorizationStatus` method of `ALAssetsLibrary`.
Contacts	The authorization status for contact data is available from the `ABAddressBookGetAuthorizationStatus` function. To inform the user about how you intend to use this information, include the `NSContactsUsageDescription` key in your `Info.plist` file.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Data	System authorization support
Calendar data	The authorization status for calendar data is available from the `authorizationStatusForEntityType:` method of `EKEventStore`. To inform the user about how you intend to use this information, include the `NSCalendarsUsageDescription` key in your `Info.plist` file.
Reminders	The authorization status for reminder data is available from the `authorizationStatusForEntityType:` method of `EKEventStore`. To inform the user about how you intend to use this information, include the `NSRemindersUsageDescription` key in your `Info.plist` file.
Bluetooth peripherals	The authorization status for Bluetooth peripherals is available from the `state` property of `CBCentralManager`. To inform the user about how you intend to use Bluetooth, include the `NSBluetoothPeripheralUsageDescription` key in your `Info.plist` file.
Microphone	In iOS 7 and later, the authorization status for the microphone is available from the `requestRecordPermission:` method of `AVAudioSession`. To inform the user about how you intend to use the microphone, include the `NSMicrophoneUsage–Description` key in your `Info.plist` file.
Camera	In iOS 7 and later, the authorization status for the camera is available in `deviceInputWithDevice: error:` method of `AVCaptureDeviceInput`. To inform the user about how you intend to use the camera, include the `NSCameraUsageDescription` key in your `Info.plist` file.

Internationalizing Your App

Because iOS apps are distributed in many countries, localizing your app's content can help you reach many more customers. Users are much more likely to use an app when it is localized for their native language. When you factor your user-facing content into resource files, localizing that content is a relatively simple process.

Before you can localize your content, you must internationalize your app in order to facilitate the localization process. Internationalizing your app involves factoring out any user-facing content into localizable resource files and providing language-specific project (`.lproj`) directories for storing that content. It also means using appropriate technologies (such as date and number formatters) when working with language-specific and locale-specific content.

For a fully internationalized app, the localization process creates new sets of language-specific resource files for you to add to your project. A typical iOS app requires localized versions of the following types of resource files:

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- Storyboard files (or nib files)—Storyboards can contain text labels and other content that need to be localized. You might also want to adjust the position of interface items to accommodate changes in text length. (Similarly, nib files can contain text that needs to be localized or layout that needs to be updated.)
- Strings files—Strings files (so named because of their `.strings` filename extension) contain localized versions of the static text that your app displays.
- Image files—You should avoid localizing images unless the images contain culture-specific content. Whenever possible, you should avoid storing text directly in your image files. For images that you load and use from within your app, store text in a strings file and composite that text with your image-based content at runtime.
- Video and audio files—You should avoid localizing multimedia files unless they contain language-specific or culture-specific content. For example, you would want to localize a video file that contained a voice-over track.

For information about the internationalization and localization process, see *Internationalization and Localization Guide*. For information about the proper way to use resource files in your app, see *Resource Programming Guide*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

The App Life Cycle

Apps are a sophisticated interplay between your custom code and the system frameworks. The system frameworks provide the basic infrastructure that all apps need to run, and you provide the code required to customize that infrastructure and give the app the look and feel you want. To do that effectively, it helps to understand a little bit about the iOS infrastructure and how it works.

iOS frameworks rely on design patterns such as model-view-controller and delegation in their implementation. Understanding those design patterns is crucial to the successful creation of an app. It also helps to be familiar with the Objective-C language and its features. If you are new to iOS programming, read *Start Developing iOS Apps Today* for an introduction to iOS apps and the Objective-C language.

The Main Function

The entry point for every C-based app is the `main` function and iOS apps are no different. What is different is that for iOS apps you do not write the `main` function yourself. Instead, Xcode creates this function as part of your basic project. Listing 2-1 shows an example of this function. With few exceptions, you should never change the implementation of the `main` function that Xcode provides.

Listing 2-1 The `main` function of an iOS app

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"

int main(int argc, char * argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate
class]));
    }
}
```

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

The only thing to mention about the `main` function is that its job is to hand control off to the UIKit framework. The `UIApplicationMain` function handles this process by creating the core objects of your app, loading your app's user interface from the available storyboard files, calling your custom code so that you have a chance to do some initial setup, and putting the app's run loop in motion. The only pieces that you have to provide are the storyboard files and the custom initialization code.

The Structure of an App

During startup, the `UIApplicationMain` function sets up several key objects and starts the app running. At the heart of every iOS app is the `UIApplication` object, whose job is to facilitate the interactions between the system and other objects in the app. Figure 2-1 shows the objects commonly found in most apps, while Table 2-1 lists the roles each of those objects plays. The first thing to notice is that iOS apps use a model-view-controller architecture. This pattern separates the app's data and business logic from the visual presentation of that data. This architecture is crucial to creating apps that can run on different devices with different screen sizes.

Figure 2-1 Key objects in an iOS app

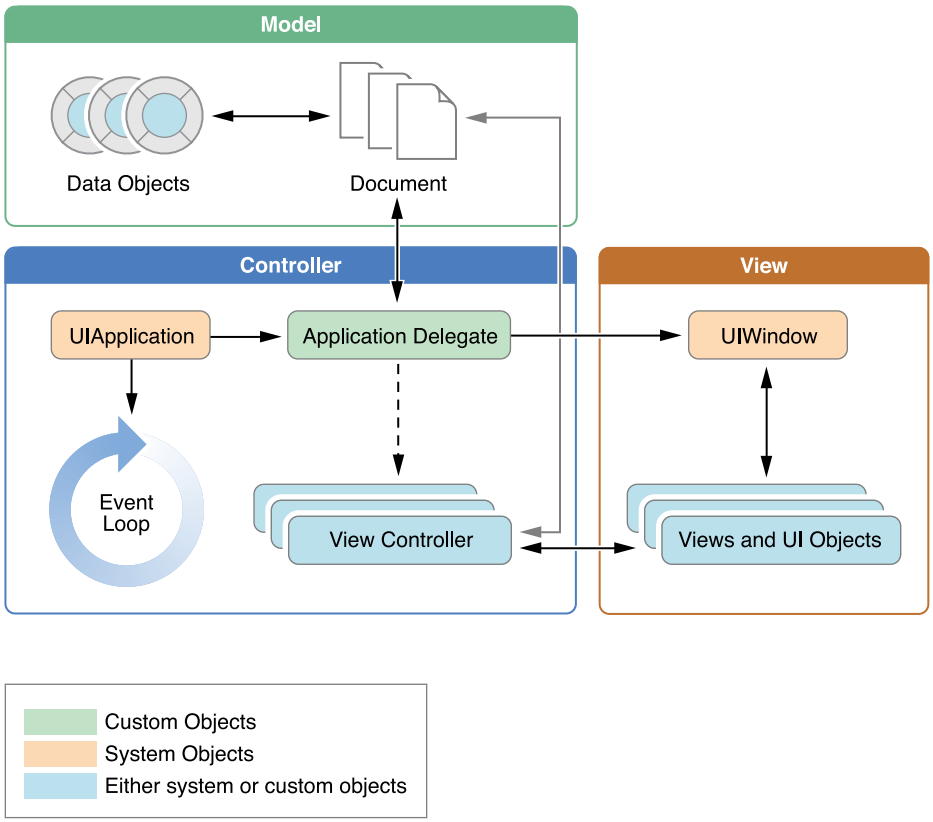


Table 2-1 The role of objects in an iOS app

Object	Description
`UIApplication` object	The `UIApplication` object manages the event loop and other high-level app behaviors. It also reports key app transitions and some special events (such as incoming push notifications) to its delegate, which is a custom object you define. Use the `UIApplication` object as is—that is, without subclassing.
App delegate object	The app delegate is the heart of your custom code. This object works in tandem with the `UIApplication` object to handle app initialization, state transitions, and many high-level app events. This object is also the only one guaranteed to be present in every app, so it is often used to set up the app's initial data structures.
Documents and data model objects	*Data model objects* store your app's content and are specific to your app. For example, a banking app might store a database containing financial transactions, whereas a painting app might store an image object or even the sequence of drawing commands that led to the creation of that image. (In the latter case, an image object is still a data object because it is just a container for the image data.) Apps can also use *document objects* (custom subclasses of `UIDocument`) to manage some or all of their data model objects. Document objects are not required but offer a convenient way to group data that belongs in a single file or file package. For more information about documents, see *Document-Based App Programming Guide for iOS*.
View controller objects	*View controller objects* manage the presentation of your app's content on screen. A view controller manages a single view and its collection of subviews. When presented, the view controller makes its views visible by installing them in the app's window. The `UIViewController` class is the base class for all view controller objects. It provides default functionality for loading views, presenting them, rotating them in response to device rotations, and several other standard system behaviors. UIKit and other frameworks define additional view controller classes to implement standard system interfaces such as the image picker, tab bar interface, and navigation interface. For detailed information about how to use view controllers, see *View Controller Programming Guide for iOS*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Object	Description
`UIWindow` object	A `UIWindow` object coordinates the presentation of one or more views on a screen. Most apps have only one window, which presents content on the main screen, but apps may have an additional window for content displayed on an external display. To change the content of your app, you use a view controller to change the views displayed in the corresponding window. You never replace the window itself. In addition to hosting views, windows work with the `UIApplication` object to deliver events to your views and view controllers.
View objects, control objects, and layer objects	Views and controls provide the visual representation of your app's content. A *view* is an object that draws content in a designated rectangular area and responds to events within that area. *Controls* are a specialized type of view responsible for implementing familiar interface objects such as buttons, text fields, and toggle switches. The UIKit framework provides standard views for presenting many different types of content. You can also define your own custom views by subclassing `UIView` (or its descendants) directly. In addition to incorporating views and controls, apps can also incorporate Core Animation layers into their view and control hierarchies. *Layer objects* are actually data objects that represent visual content. Views use layer objects intensively behind the scenes to render their content. You can also add custom layer objects to your interface to implement complex animations and other types of sophisticated visual effects.

What distinguishes one iOS app from another is the data it manages (and the corresponding business logic) and how it presents that data to the user. Most interactions with UIKit objects do not define your app but help you to refine its behavior. For example, the methods of your app delegate let you know when the app is changing states so that your custom code can respond appropriately.

The Main Run Loop

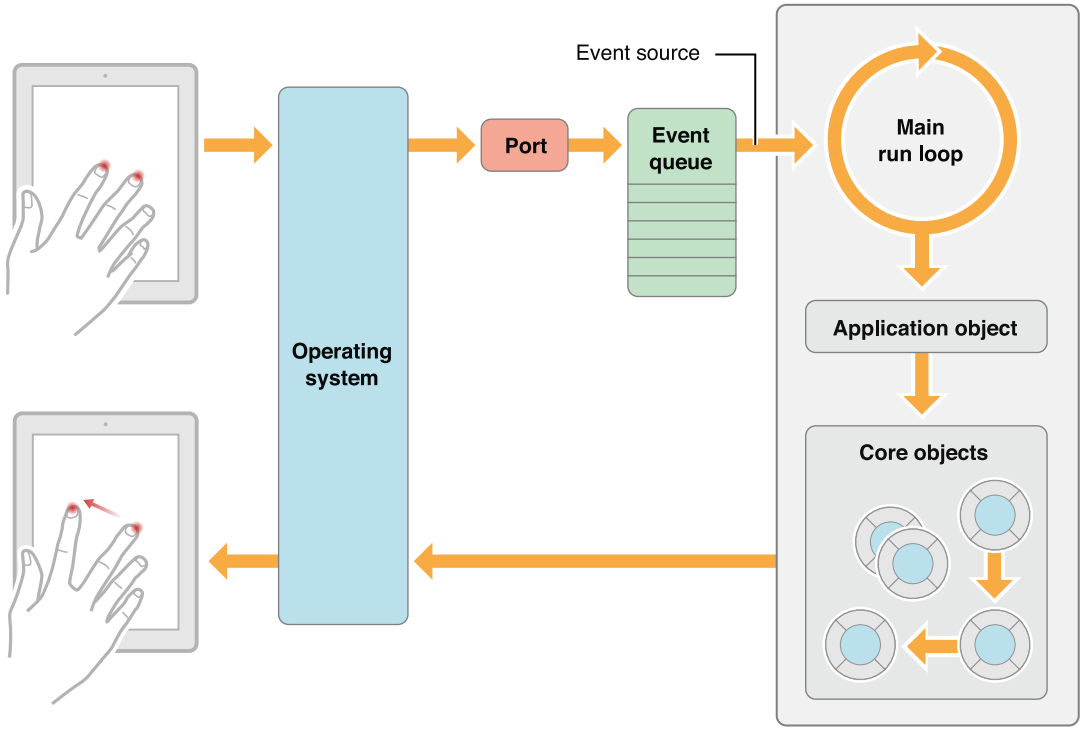
An app's *main run loop* processes all user-related events. The `UIApplication` object sets up the main run loop at launch time and uses it to process events and handle updates to view-based interfaces. As the name suggests, the main run loop executes on the app's main thread. This behavior ensures that user-related events are processed serially in the order in which they were received.

Figure 2-2 shows the architecture of the main run loop and how user events result in actions taken by your app. As the user interacts with a device, events related to those interactions are generated by the system and delivered to the app via a special port set up by UIKit. Events are queued internally by the app and dispatched

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

one-by-one to the main run loop for execution. The `UIApplication` object is the first object to receive the event and make the decision about what needs to be done. A touch event is usually dispatched to the main window object, which in turn dispatches it to the view in which the touch occurred. Other events might take slightly different paths through various app objects.

Figure 2-2 Processing events in the main run loop



Many types of events can be delivered in an iOS app. The most common ones are listed in Table 2-2. Many of these event types are delivered using the main run loop of your app, but some are not. Some events are sent to a delegate object or are passed to a block that you provide. For information about how to handle most types of events—including touch, remote control, motion, accelerometer, and gyroscopic events—see *Event Handling Guide for iOS*.

Table 2-2 Common types of events for iOS apps

Event type	Delivered to…	Notes
Touch	The view object in which the event occurred	Views are responder objects. Any touch events not handled by the view are forwarded down the responder chain for processing.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Event type	Delivered to…	Notes
Remote control Shake motion events	First responder object	Remote control events are for controlling media playback and are generated by headphones and other accessories.
Accelerometer Magnetometer Gyroscope	The object you designate	Events related to the accelerometer, magnetometer, and gyroscope hardware are delivered to the object you designate.
Location	The object you designate	You register to receive location events using the Core Location framework. For more information about using Core Location, see *Location and Maps Programming Guide*.
Redraw	The view that needs the update	Redraw events do not involve an event object but are simply calls to the view to draw itself. The drawing architecture for iOS is described in *Drawing and Printing Guide for iOS*.

Some events, such as touch and remote control events, are handled by your app's responder objects. Responder objects are everywhere in your app. (The `UIApplication` object, your view objects, and your view controller objects are all examples of responder objects.) Most events target a specific responder object but can be passed to other responder objects (via the responder chain) if needed to handle an event. For example, a view that does not handle an event can pass the event to its superview or to a view controller.

Touch events occurring in controls (such as buttons) are handled differently than touch events occurring in many other types of views. There are typically only a limited number of interactions possible with a control, and so those interactions are repackaged into action messages and delivered to an appropriate target object. This target-action design pattern makes it easy to use controls to trigger the execution of custom code in your app.

Execution States for Apps

At any given moment, your app is in one of the states listed in Table 2-3. The system moves your app from state to state in response to actions happening throughout the system. For example, when the user presses the Home button, a phone call comes in, or any of several other interruptions occurs, the currently running apps change state in response. Figure 2-3 (page 29) shows the paths that an app takes when moving from state to state.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

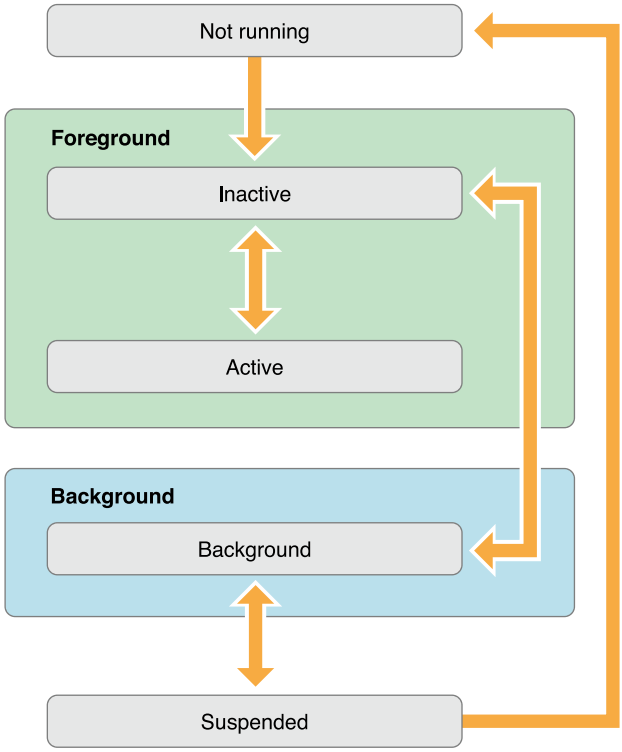
Table 2-3 App states

State	Description
Not running	The app has not been launched or was running but was terminated by the system.
Inactive	The app is running in the foreground but is currently not receiving events. (It may be executing other code though.) An app usually stays in this state only briefly as it transitions to a different state.
Active	The app is running in the foreground and is receiving events. This is the normal mode for foreground apps.
Background	The app is in the background and executing code. Most apps enter this state briefly on their way to being suspended. However, an app that requests extra execution time may remain in this state for a period of time. In addition, an app being launched directly into the background enters this state instead of the inactive state. For information about how to execute code while in the background, see Background Execution (page 32).

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

State	Description
Suspended	The app is in the background but is not executing code. The system moves apps to this state automatically and does not notify them before doing so. While suspended, an app remains in memory but does not execute any code. When a low-memory condition occurs, the system may purge suspended apps without notice to make more space for the foreground app.

Figure 2-3 State changes in an iOS app



Most state transitions are accompanied by a corresponding call to the methods of your app delegate object. These methods are your chance to respond to state changes in an appropriate way. These methods are listed below, along with a summary of how you might use them.

- `application:willFinishLaunchingWithOptions:`—This method is your app's first chance to execute code at launch time.
- `application:didFinishLaunchingWithOptions:`—This method allows you to perform any final initialization before your app is displayed to the user.
- `applicationDidBecomeActive:`—Lets your app know that it is about to become the foreground app. Use this method for any last minute preparation.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- `applicationWillResignActive:`—Lets you know that your app is transitioning away from being the foreground app. Use this method to put your app into a quiescent state.
- `applicationDidEnterBackground:`—Lets you know that your app is now running in the background and may be suspended at any time.
- `applicationWillEnterForeground:`—Lets you know that your app is moving out of the background and back into the foreground, but that it is not yet active.
- `applicationWillTerminate:`—Lets you know that your app is being terminated. This method is not called if your app is suspended.

App Termination

Apps must be prepared for termination to happen at any time and should not wait to save user data or perform other critical tasks. System-initiated termination is a normal part of an app's life cycle. The system usually terminates apps so that it can reclaim memory and make room for other apps being launched by the user, but the system may also terminate apps that are misbehaving or not responding to events in a timely manner.

Suspended apps receive no notification when they are terminated; the system kills the process and reclaims the corresponding memory. If an app is currently running in the background and not suspended, the system calls the `applicationWillTerminate:` of its app delegate prior to termination. The system does not call this method when the device reboots.

In addition to the system terminating your app, the user can terminate your app explicitly using the multitasking UI. User-initiated termination has the same effect as terminating a suspended app. The app's process is killed and no notification is sent to the app.

Threads and Concurrency

The system creates your app's main thread and you can create additional threads, as needed, to perform other tasks. For iOS apps, the preferred technique is to use Grand Central Dispatch (GCD), operation objects, and other asynchronous programming interfaces rather than creating and managing threads yourself. Technologies such as GCD let you define the work you want to do and the order you want to do it in, but let the system decide how best to execute that work on the available CPUs. Letting the system handle the thread management simplifies the code you must write, makes it easier to ensure the correctness of that code, and offers better overall performance.

When thinking about threads and concurrency, consider the following:

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- Work involving views, Core Animation, and many other UIKit classes usually must occur on the app's main thread. There are some exceptions to this rule—for example, image-based manipulations can often occur on background threads—but when in doubt, assume that work needs to happen on the main thread.
- Lengthy tasks (or potentially length tasks) should always be performed on a background thread. Any tasks involving network access, file access, or large amounts of data processing should all be performed asynchronously using GCD or operation objects.
- At launch time, move tasks off the main thread whenever possible. At launch time, your app should use the available time to set up its user interface as quickly as possible. Only tasks that contribute to setting up the user interface should be performed on the main thread. All other tasks should be executed asynchronously, with the results displayed to the user as soon as they are ready.

For more information about using GCD and operation objects to execute tasks, see *Concurrency Programming Guide*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Background Execution

Objective-CSwift

When the user is not actively using your app, the system moves it to the background state. For many apps, the background state is just a brief stop on the way to the app being suspended. Suspending apps is a way of improving battery life it also allows the system to devote important system resources to the new foreground app that has drawn the user's attention.

Most apps can move to the extended state easily enough but there are also legitimate reasons for apps to continue running in the background. A hiking app might want to track the user's position over time so that it can display that course overlaid on top of a hiking map. An audio app might need to continue playing music over the lock screen. Other apps might want to download content in the background so that it can minimize the delay in presenting that content to the user. When you find it necessary to keep your app running in the background, iOS helps you do so efficiently and without draining system resources or the user's battery. The techniques offered by iOS fall into three categories:

- Apps that start a short task in the foreground can ask for time to finish that task when the app moves to the background.
- Apps that initiate downloads in the foreground can hand off management of those downloads to the system, thereby allowing the app to be suspended or terminated while the download continues.
- Apps that need to run in the background to support specific types of tasks can declare their support for one or more background execution modes.

Always try to avoid doing any background work unless doing so improves the overall user experience. An app might move to the background because the user launched a different app or because the user locked the device and is not using it right now. In both situations, the user is signaling that your app does not need to be doing any meaningful work right now. Continuing to run in such conditions will only drain the device's battery and might lead the user to force quit your app altogether. So be mindful about the work you do in the background and avoid it when you can.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Executing Finite-Length Tasks

Apps moving to the background are expected to put themselves into a quiescent state as quickly as possible so that they can be suspended by the system. If your app is in the middle of a task and needs a little extra time to complete that task, it can call the `beginBackgroundTaskWithName:expirationHandler:` or `beginBackgroundTaskWithExpirationHandler:` method of the `UIApplication` object to request some additional execution time. Calling either of these methods delays the suspension of your app temporarily, giving it a little extra time to finish its work. Upon completion of that work, your app must call the `endBackgroundTask:` method to let the system know that it is finished and can be suspended.

Each call to the `beginBackgroundTaskWithName:expirationHandler:` or `beginBackgroundTaskWithExpirationHandler:` method generates a unique token to associate with the corresponding task. When your app completes a task, it must call the `endBackgroundTask:` method with the corresponding token to let the system know that the task is complete. Failure to call the `endBackgroundTask:` method for a background task will result in the termination of your app. If you provided an expiration handler when starting the task, the system calls that handler and gives you one last chance to end the task and avoid termination.

You do not need to wait until your app moves to the background to designate background tasks. A more useful design is to call the `beginBackgroundTaskWithName:expirationHandler:` or `beginBackgroundTaskWithExpirationHandler:` method before starting a task and call the `endBackgroundTask:` method as soon as you finish. You can even follow this pattern while your app is executing in the foreground.

Listing 3-1 shows how to start a long-running task when your app transitions to the background. In this example, the request to start a background task includes an expiration handler just in case the task takes too long. The task itself is then submitted to a dispatch queue for asynchronous execution so that the `applicationDidEnterBackground:` method can return normally. The use of blocks simplifies the code needed to maintain references to any important variables, such as the background task identifier. The `bgTask` variable is a member variable of the class that stores a pointer to the current background task identifier and is initialized prior to its use in this method.

Listing 3-1 Starting a background task at quit time

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    bgTask = [application beginBackgroundTaskWithName:@"MyTask" expirationHandler:^{
        // Clean up any unfinished task business by marking where you
        // stopped or ending the task outright.
        [application endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
```

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

```
    }];

    // Start the long-running task and return immediately.
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
 ^{

        // Do the work associated with the task, preferably in chunks.

        [application endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
    });
}
```

> **Note:** Always provide an expiration handler when starting a task, but if you want to know how much time your app has left to run, get the value of the `backgroundTimeRemaining` property of `UIApplication`.

In your own expiration handlers, you can include additional code needed to close out your task. However, any code you include must not take too long to execute because, by the time your expiration handler is called, your app is already very close to its time limit. For this reason, perform only minimal cleanup of your state information and end the task.

Downloading Content in the Background

When downloading files, apps should use an `NSURLSession` object to start the downloads so that the system can take control of the download process in case the app is suspended or terminated. When you configure an `NSURLSession` object for background transfers, the system manages those transfers in a separate process and reports status back to your app in the usual way. If your app is terminated while transfers are ongoing, the system continues the transfers in the background and launches your app (as appropriate) when the transfers finish or when one or more tasks need your app's attention.

To support background transfers, you must configure your `NSURLSession` object appropriately. To configure the session, you must first create a `NSURLSessionConfiguration` object and set several properties to appropriate values. You then pass that configuration object to the appropriate initialization method of `NSURLSession` when creating your session.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

The process for creating a configuration object that supports background downloads is as follows:

1. Create the configuration object using the `backgroundSessionConfigurationWithIdentifier:` method of `NSURLSessionConfiguration`.
2. Set the value of the configuration object's `sessionSendsLaunchEvents` property to `YES`.
3. if your app starts transfers while it is in the foreground, it is recommend that you also set the `discretionary` property of the configuration object to `YES`.
4. Configure any other properties of the configuration object as appropriate.
5. Use the configuration object to create your `NSURLSession` object.

Once configured, your `NSURLSession` object seamlessly hands off upload and download tasks to the system at appropriate times. If tasks finish while your app is still running (either in the foreground or the background), the session object notifies its delegate in the usual way. If tasks have not yet finished and the system terminates your app, the system automatically continues managing the tasks in the background. If the user terminates your app, the system cancels any pending tasks.

When all of the tasks associated with a background session are complete, the system relaunches a terminated app (assuming that the `sessionSendsLaunchEvents` property was set to `YES` and that the user did not force quit the app) and calls the app delegate's `application:handleEventsForBackgroundURLSession:completionHandler:` method. (The system may also relaunch the app to handle authentication challenges or other task-related events that require your app's attention.) In your implementation of that delegate method, use the provided identifier to create a new `NSURLSessionConfiguration` and `NSURLSession` object with the same configuration as before. The system reconnects your new session object to the previous tasks and reports their status to the session object's delegate.

Implementing Long-Running Tasks

For tasks that require more execution time to implement, you must request specific permissions to run them in the background without their being suspended. In iOS, only specific app types are allowed to run in the background:

- Apps that play audible content to the user while in the background, such as a music player app
- Apps that record audio content while in the background
- Apps that keep users informed of their location at all times, such as a navigation app
- Apps that support Voice over Internet Protocol (VoIP)
- Apps that need to download and process new content regularly

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- Apps that receive regular updates from external accessories

Apps that implement these services must declare the services they support and use system frameworks to implement the relevant aspects of those services. Declaring the services lets the system know which services you use, but in some cases it is the system frameworks that actually prevent your application from being suspended.

Declaring Your App's Supported Background Tasks

Support for some types of background execution must be declared in advance by the app that uses them. In Xcode 5 and later, you declare the background modes your app supports from the Capabilities tab of your project settings. Enabling the Background Modes option adds the `UIBackgroundModes` key to your app's `Info.plist` file. Selecting one or more checkboxes adds the corresponding background mode values to that key. Table 3-1 lists the background modes you can specify and the values that Xcode assigns to the `UIBackgroundModes` key in your app's `Info.plist` file.

Table 3-1 Background modes for apps

Xcode background mode	UIBackgroundModes value	Description
Audio and AirPlay	`audio`	The app plays audible content to the user or records audio while in the background. (This content includes streaming audio or video content using AirPlay.) The user must grant permission for apps to use the microphone prior to the first use; for more information, see Supporting User Privacy (page 17).
Location updates	`location`	The app keeps users informed of their location, even while it is running in the background.
Voice over IP	`voip`	The app provides the ability for the user to make phone calls using an Internet connection.
Newsstand downloads	`newsstand-content`	The app is a Newsstand app that downloads and processes magazine or newspaper content in the background.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Xcode background mode	UIBackgroundModes value	Description
External accessory communication	`external-accessory`	The app works with a hardware accessory that needs to deliver updates on a regular schedule through the External Accessory framework.
Uses Bluetooth LE accessories	`bluetooth-central`	The app works with a Bluetooth accessory that needs to deliver updates on a regular schedule through the Core Bluetooth framework.
Acts as a Bluetooth LE accessory	`bluetooth-peripheral`	The app supports Bluetooth communication in peripheral mode through the Core Bluetooth framework. Using this mode requires user authorization; for more information, see Supporting User Privacy (page 17).
Background fetch	`fetch`	The app regularly downloads and processes small amounts of content from the network.
Remote notifications	`remote-notification`	The app wants to start downloading content when a push notification arrives. Use this notification to minimize the delay in showing content related to the push notification.

Each of the preceding modes lets the system know that your app should be woken up or launched at appropriate times to respond to relevant events. For example, an app that begins playing music and then moves to the background still needs execution time to fill the audio output buffers. Enabling the Audio mode tells the system frameworks that they should continue to make the necessary callbacks to the app at appropriate intervals. If the app does not select this mode, any audio being played or recorded by the app stops when the app moves to the background.

Tracking the User's Location

There are several ways to track the user's location in the background, most of which do not actually require your app to run continuously in the background:

- The significant-change location service (Recommended)
- Foreground-only location services
- Background location services

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

The significant-change location service is highly recommended for apps that do not need high-precision location data. With this service, location updates are generated only when the user's location changes significantly; thus, it is ideal for social apps or apps that provide the user with noncritical, location-relevant information. If the app is suspended when an update occurs, the system wakes it up in the background to handle the update. If the app starts this service and is then terminated, the system relaunches the app automatically when a new location becomes available. This service is available in iOS 4 and later, and it is available only on devices that contain a cellular radio.

The foreground-only and background location services both use the standard location Core Location service to retrieve location data. The only difference is that the foreground-only location services stop delivering updates if the app is ever suspended, which is likely to happen if the app does not support other background services or tasks. Foreground-only location services are intended for apps that only need location data while they are in the foreground.

You enable location support from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `location` value in your app's `Info.plist` file.) Enabling this mode does not prevent the system from suspending the app, but it does tell the system that it should wake up the app whenever there is new location data to deliver. Thus, this key effectively lets the app run in the background to process location updates whenever they occur.

> **Important:** You are encouraged to use the standard services sparingly or use the significant location change service instead. Location services require the active use of an iOS device's onboard radio hardware. Running this hardware continuously can consume a significant amount of power. If your app does not need to provide precise and continuous location information to the user, it is best to minimize the use of location services.

For information about how to use each of the different location services in your app, see *Location and Maps Programming Guide* .

Playing and Recording Background Audio

An app that plays or records audio continuously (even while the app is running in the background) can register to perform those tasks in the background. You enable audio support from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `audio` value in your app's `Info.plist` file.) Apps that play audio content in the background must play audible content and not silence.

Typical examples of background audio apps include:

- Music player apps
- Audio recording apps

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- Apps that support audio or video playback over AirPlay
- VoIP apps

When the `UIBackgroundModes` key contains the `audio` value, the system's media frameworks automatically prevent the corresponding app from being suspended when it moves to the background. As long as it is playing audio or video content or recording audio content, the app continues to run in the background. However, if recording or playback stops, the system suspends the app.

You can use any of the system audio frameworks to work with background audio content, and the process for using those frameworks is unchanged. (For video playback over AirPlay, you can use the Media Player or AV Foundation framework to present your video.) Because your app is not suspended while playing media files, callbacks operate normally while your app is in the background. In your callbacks, though, you should do only the work necessary to provide data for playback. For example, a streaming audio app would need to download the music stream data from its server and push the current audio samples out for playback. Apps should not perform any extraneous tasks that are unrelated to playback.

Because more than one app may support audio, the system determines which app is allowed to play or record audio at any given time. The foreground app always has priority for audio operations. It is possible for more than one background app to be allowed to play audio and such determinations are based on the configuration of each app's audio session objects. You should always configure your app's audio session object appropriately and work carefully with the system frameworks to handle interruptions and other types of audio-related notifications. For information on how to configure audio session objects for background execution, see *Audio Session Programming Guide*.

Implementing a VoIP App

A *Voice over Internet Protocol (VoIP)* app allows the user to make phone calls using an Internet connection instead of the device's cellular service. Such an app needs to maintain a persistent network connection to its associated service so that it can receive incoming calls and other relevant data. Rather than keep VoIP apps awake all the time, the system allows them to be suspended and provides facilities for monitoring their sockets for them. When incoming traffic is detected, the system wakes up the VoIP app and returns control of its sockets to it.

To configure a VoIP app, you must do the following:

1. Enable support for Voice over IP from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `voip` value in your app's `Info.plist` file.)
2. Configure one of the app's sockets for VoIP usage.
3. Before moving to the background, call the `setKeepAliveTimeout:handler:` method to install a handler to be executed periodically. Your app can use this handler to maintain its service connection.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

4. Configure your audio session to handle transitions to and from active use.

Including the `voip` value in the `UIBackgroundModes` key lets the system know that it should allow the app to run in the background as needed to manage its network sockets. An app with this key is also relaunched in the background immediately after system boot to ensure that the VoIP services are always available.

Most VoIP apps also need to be configured as background audio apps to deliver audio while in the background. Therefore, you should include both the `audio` and `voip` values to the `UIBackgroundModes` key. If you do not do this, your app cannot play or record audio while it is in the background. For more information about the `UIBackgroundModes` key, see *Information Property List Key Reference*.

For specific information about the steps you must take to implement a VoIP app, see Tips for Developing a VoIP App (page 92).

Fetching Small Amounts of Content Opportunistically

Apps that need to check for new content periodically can ask the system to wake them up so that they can initiate a fetch operation for that content. To support this mode, enable the Background fetch option from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `fetch` value in your app's `Info.plist` file.) Enabling this mode is not a guarantee that the system will give your app any time to perform background fetches. The system must balance your app's need to fetch content with the needs of other apps and the system itself. After assessing that information, the system gives time to apps when there are good opportunities to do so.

When a good opportunity arises, the system wakes or launches your app into the background and calls the app delegate's `application:performFetchWithCompletionHandler:` method. Use that method to check for new content and initiate a download operation if content is available. As soon as you finish downloading the new content, you must execute the provided completion handler block, passing a result that indicates whether content was available. Executing this block tells the system that it can move your app back to the suspended state and evaluate its power usage. Apps that download small amounts of content quickly, and accurately reflect when they had content available to download, are more likely to receive execution time in the future than apps that take a long time to download their content or that claim content was available but then do not download anything.

When downloading any content, it is recommended that you use the `NSURLSession` class to initiate and manage your downloads. For information about how to use this class to manage upload and download tasks, see *URL Loading System Programming Guide*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Using Push Notifications to Initiate a Download

If your server sends push notifications to a user's device when new content is available for your app, you can ask the system to run your app in the background so that it can begin downloading the new content right away. The intent of this background mode is to minimize the amount of time that elapses between when a user sees a push notification and when your app is able to able to display the associated content. Apps are typically woken up at roughly the same time that the user sees the notification but that still gives you more time than you might have otherwise.

To support this background mode, enable the Remote notifications option from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `remote-notification` value in your app's `Info.plist` file.)

For a push notification to trigger a download operation, the notification's payload must include the `content-available` key with its value set to `1`. When that key is present, the system wakes the app in the background (or launches it into the background) and calls the app delegate's `application:didReceiveRemoteNotification:fetchCompletionHandler:` method. Your implementation of that method should download the relevant content and integrate it into your app.

When downloading any content, it is recommended that you use the `NSURLSession` class to initiate and manage your downloads. For information about how to use this class to manage upload and download tasks, see *URL Loading System Programming Guide*.

Downloading Newsstand Content in the Background

A Newsstand app that downloads new magazine or newspaper issues can register to perform those downloads in the background. You enable support for newsstand downloads from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `newsstand-content` value in your app's `Info.plist` file.) When this key is present, the system launches your app, if it is not already running, so that it can initiate the downloading of the new issue.

When you use the Newsstand Kit framework to initiate a download, the system handles the download process for your app. The system continues to download the file even if your app is suspended or terminated. When the download operation is complete, the system transfers the file to your app sandbox and notifies your app. If the app is not running, this notification wakes it up and gives it a chance to process the newly downloaded file. If there are errors during the download process, your app is similarly woken up to handle them.

For information about how to download content using the Newsstand Kit framework, see *NewsstandKit Framework Reference*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Communicating with an External Accessory

Apps that work with external accessories can ask to be woken up if the accessory delivers an update when the app is suspended. This support is important for some types of accessories that deliver data at regular intervals, such as heart-rate monitors. You enable support for external accessory communication from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `external-accessory` value in your app's `Info.plist` file.) When you enable this mode, the external accessory framework does not close active sessions with accessories. (In iOS 4 and earlier, these sessions are closed automatically when the app is suspended.) When new data arrives from the accessory, the framework wakes your app so that it can process that data. The system also wakes the app to process accessory connection and disconnection notifications.

Any app that supports the background processing of accessory updates must follow a few basic guidelines:

- Apps must provide an interface that allows the user to start and stop the delivery of accessory update events. That interface should then open or close the accessory session as appropriate.
- Upon being woken up, the app has around 10 seconds to process the data. Ideally, it should process the data as fast as possible and allow itself to be suspended again. However, if more time is needed, the app can use the `beginBackgroundTaskWithExpirationHandler:` method to request additional time; it should do so only when absolutely necessary, though.

Communicating with a Bluetooth Accessory

Apps that work with Bluetooth peripherals can ask to be woken up if the peripheral delivers an update when the app is suspended. This support is important for Bluetooth-LE accessories that deliver data at regular intervals, such as a Bluetooth heart rate belt. You enable support for using bluetooth accessories from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `bluetooth-central` value in your app's `Info.plist` file.) When you enable this mode, the Core Bluetooth framework keeps open any active sessions for the corresponding peripheral. In addition, new data arriving from the peripheral causes the system to wake up the app so that it can process the data. The system also wakes up the app to process accessory connection and disconnection notifications.

In iOS 6, an app can also operate in peripheral mode with Bluetooth accessories. To act as a Bluetooth accessory, you must enable support for that mode from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `bluetooth-peripheral` value in your app's `Info.plist` file.) Enabling this mode lets the Core Bluetooth framework wake the app up briefly in the background so that it can handle accessory-related requests. Apps woken up for these events should process them and return as quickly as possible so that the app can be suspended again.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Any app that supports the background processing of Bluetooth data must be session-based and follow a few basic guidelines:

- Apps must provide an interface that allows the user to start and stop the delivery of Bluetooth events. That interface should then open or close the session as appropriate.
- Upon being woken up, the app has around 10 seconds to process the data. Ideally, it should process the data as fast as possible and allow itself to be suspended again. However, if more time is needed, the app can use the `beginBackgroundTaskWithExpirationHandler:` method to request additional time; it should do so only when absolutely necessary, though.

Getting the User's Attention While in the Background

Notifications are a way for an app that is suspended, is in the background, or is not running to get the user's attention. Apps can use local notifications to display alerts, play sounds, badge the app's icon, or a combination of the three. For example, an alarm clock app might use local notifications to play an alarm sound and display an alert to disable the alarm. When a notification is delivered to the user, the user must decide if the information warrants bringing the app back to the foreground. (If the app is already running in the foreground, local notifications are delivered quietly to the app and not to the user.)

To schedule the delivery of a local notification, create an instance of the `UILocalNotification` class, configure the notification parameters, and schedule it using the methods of the `UIApplication` class. The local notification object contains information about the type of notification to deliver (sound, alert, or badge) and the time (when applicable) at which to deliver it. The methods of the `UIApplication` class provide options for delivering notifications immediately or at the scheduled time.

Listing 3-2 shows an example that schedules a single alarm using a date and time that is set by the user. This example configures only one alarm at a time and cancels the previous alarm before scheduling a new one. (Your own apps can have no more than 128 local notifications active at any given time, any of which can be configured to repeat at a specified interval.) The alarm itself consists of an alert box and a sound file that is played if the app is not running or is in the background when the alarm fires. If the app is active and therefore running in the foreground, the app delegate's `application:didReceiveLocalNotification:` method is called instead.

Listing 3-2 Scheduling an alarm notification

```
- (void)scheduleAlarmForDate:(NSDate*)theDate
{
    UIApplication* app = [UIApplication sharedApplication];
    NSArray*    oldNotifications = [app scheduledLocalNotifications];
```

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

```
    // Clear out the old notification before scheduling a new one.
    if ([oldNotifications count] > 0)
        [app cancelAllLocalNotifications];

    // Create a new notification.
    UILocalNotification* alarm = [[UILocalNotification alloc] init];
    if (alarm)
    {
        alarm.fireDate = theDate;
        alarm.timeZone = [NSTimeZone defaultTimeZone];
        alarm.repeatInterval = 0;
        alarm.soundName = @"alarmsound.caf";
        alarm.alertBody = @"Time to wake up!";

        [app scheduleLocalNotification:alarm];
    }
}
```

Sound files used with local notifications have the same requirements as those used for push notifications. Custom sound files must be located inside your app's main bundle and support one of the following formats: Linear PCM, MA4, µ-Law, or a-Law. You can also specify the `UILocalNotificationDefaultSoundName` constant to play the default alert sound for the device. When the notification is sent and the sound is played, the system also triggers a vibration on devices that support it.

You can cancel scheduled notifications or get a list of notifications using the methods of the `UIApplication` class. For more information about these methods, see *UIApplication Class Reference*. For additional information about configuring local notifications, see *Local and Remote Notification Programming Guide*.

Understanding When Your App Gets Launched into the Background

Apps that support background execution may be relaunched by the system to handle incoming events. If an app is terminated for any reason other than the user force quitting it, the system launches the app when one of the following events happens:

- For location apps:

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

    - The system receives a location update that meets the app's configured criteria for delivery.
    - The device entered or exited a registered region. (Regions can be geographic regions or iBeacon regions.)
- For audio apps, the audio framework needs the app to process some data. (Audio apps include those that play audio or use the microphone.)
- For Bluetooth apps:
    - An app acting in the central role receives data from a connected peripheral.
    - An app acting in the peripheral role receives commands from a connected central.
- For background download apps:
    - A push notification arrives for an app and the payload of the notification contains the `content-available` key with a value of `1`.
    - The system wakes the app at opportunistic moments to begin downloading new content.
    - For apps downloading content in the background using the `NSURLSession` class, all tasks associated with that session object either completed successfully or received an error.
    - A download initiated by a Newsstand app finishes.

In most cases, the system does not relaunch apps after they are force quit by the user. One exception is location apps, which in iOS 8 and later are relaunched after being force quit by the user. In other cases, though, the user must launch the app explicitly or reboot the device before the app can be launched automatically into the background by the system.

Being a Responsible Background App

The foreground app always has precedence over background apps when it comes to the use of system resources and hardware. Apps running in the background need to be prepared for this discrepancy and adjust their behavior when running in the background. Specifically, apps moving to the background should follow these guidelines:

- **Do not make any OpenGL ES calls from your code.** You must not create an `EAGLContext` object or issue any OpenGL ES drawing commands of any kind while running in the background. Using these calls causes your app to be killed immediately. Apps must also ensure that any previously submitted commands have completed before moving to the background. For information about how to handle OpenGL ES when moving to and from the background, see Implementing a Multitasking-aware OpenGL ES Application in *OpenGL ES Programming Guide for iOS*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- **Cancel any Bonjour-related services before being suspended.** When your app moves to the background, and before it is suspended, it should unregister from Bonjour and close listening sockets associated with any network services. A suspended app cannot respond to incoming service requests anyway. Closing out those services prevents them from appearing to be available when they actually are not. If you do not close out Bonjour services yourself, the system closes out those services automatically when your app is suspended.
- **Be prepared to handle connection failures in your network-based sockets.** The system may tear down socket connections while your app is suspended for any number of reasons. As long as your socket-based code is prepared for other types of network failures, such as a lost signal or network transition, this should not lead to any unusual problems. When your app resumes, if it encounters a failure upon using a socket, simply reestablish the connection.
- **Save your app state before moving to the background.** During low-memory conditions, background apps may be purged from memory to free up space. Suspended apps are purged first, and no notice is given to the app before it is purged. As a result, apps should take advantage of the state preservation mechanism in iOS 6 and later to save their interface state to disk. For information about how to support this feature, see Preserving Your App's Visual Appearance Across Launches (page 68).
- **Remove strong references to unneeded objects when moving to the background.** If your app maintains a large in-memory cache of objects (especially images), remove all strong references to those caches when moving to the background. For more information, see Reduce Your Memory Footprint (page 63).
- **Stop using shared system resources before being suspended.** Apps that interact with shared system resources such as the Address Book or calendar databases should stop using those resources before being suspended. Priority for such resources always goes to the foreground app. When your app is suspended, if it is found to be using a shared resource, the app is killed.
- **Avoid updating your windows and views.** Because your app's windows and views are not visible when your app is in the background, you should avoid updating them. The exception is in cases where you need to update the contents of a window prior to having a snapshot of your app taken.
- **Respond to connect and disconnect notifications for external accessories.** For apps that communicate with external accessories, the system automatically sends a disconnection notification when the app moves to the background. The app must register for this notification and use it to close out the current accessory session. When the app moves back to the foreground, a matching connection notification is sent, giving the app a chance to reconnect. For more information on handling accessory connection and disconnection notifications, see *External Accessory Programming Topics*.
- **Clean up resources for active alerts when moving to the background.** In order to preserve context when switching between apps, the system does not automatically dismiss action sheets (`UIActionSheet`) or alert views (`UIAlertView`) when your app moves to the background. It is up to you to provide the

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

appropriate cleanup behavior prior to moving to the background. For example, you might want to cancel the action sheet or alert view programmatically or save enough contextual information to restore the view later (in cases where your app is terminated).

- **Remove sensitive information from views before moving to the background.** When an app transitions to the background, the system takes a snapshot of the app's main window, which it then presents briefly when transitioning your app back to the foreground. Before returning from your `applicationDidEnterBackground:` method, you should hide or obscure passwords and other sensitive personal information that might be captured as part of the snapshot.
- **Do minimal work while running in the background.** The execution time given to background apps is more constrained than the amount of time given to the foreground app. Apps that spend too much time executing in the background can be throttled back by the system or terminated.

If you are implementing a background audio app, or any other type of app that is allowed to run in the background, your app responds to incoming messages in the usual way. In other words, the system may notify your app of low-memory warnings when they occur. And in situations where the system needs to terminate apps to free even more memory, the app calls its delegate's `applicationWillTerminate:` method to perform any final tasks before exiting.

Opting Out of Background Execution

If you do not want your app to run in the background at all, you can explicitly opt out of background by adding the `UIApplicationExitsOnSuspend` key (with the value `YES`) to your app's `Info.plist` file. When an app opts out, it cycles between the not-running, inactive, and active states and never enters the background or suspended states. When the user presses the Home button to quit the app, the `applicationWillTerminate:` method of the app delegate is called and the app has approximately 5 seconds to clean up and exit before it is terminated and moved back to the not-running state.

Opting out of background execution is strongly discouraged but may be the preferred option under certain conditions. Specifically, if coding for background execution adds significant complexity to your app, terminating the app might be a simpler solution. Also, if your app consumes a large amount of memory and cannot easily release any of it, the system might kill your app quickly anyway to make room for other apps. Thus, opting to terminate, instead of switching to the background, might yield the same results and save you development time and effort.

For more information about the keys you can include in your app's `Info.plist` file, see *Information Property List Key Reference*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Strategies for Handling App State Transitions

Objective-CSwift

For each of the possible runtime states of an app, the system has different expectations while your app is in that state. When state transitions occur, the system notifies the app object, which in turn notifies its app delegate. You can use the state transition methods of the `UIApplicationDelegate` protocol to detect these state changes and respond appropriately. For example, when transitioning from the foreground to the background, you might write out any unsaved data and stop any ongoing tasks. The following sections offer tips and guidance for how to implement your state transition code.

What to Do at Launch Time

When your app is launched (either into the foreground or background), use your app delegate's `application:willFinishLaunchingWithOptions:` and `application:didFinishLaunchingWithOptions:` methods to do the following:

- Check the contents of the launch options dictionary for information about why the app was launched, and respond appropriately.
- Initialize your app's critical data structures.
- Prepare your app's window and views for display:
    - Apps that use OpenGL ES for drawing must not use these methods to prepare their drawing environment. Instead, defer any OpenGL ES drawing calls to the `applicationDidBecomeActive:` method.
    - Show your app window from your `application:willFinishLaunchingWithOptions:` method. UIKit delays making the window visible until after the `application:didFinishLaunchingWithOptions:` method returns.

At launch time, the system automatically loads your app's main storyboard file and loads the initial view controller. For apps that support state restoration, the state restoration machinery restores your interface to its previous state between calls to the `application:willFinishLaunchingWithOptions:` and `application:didFinishLaunchingWithOptions:` methods. Use the `application:willFinishLaunchingWithOptions:` method to show your app window and to determine if state restoration should happen at all. Use the `application:didFinishLaunchingWithOptions:` method to make any final adjustments to your app's user interface.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Your `application:willFinishLaunchingWithOptions:` and `application:didFinishLaunchingWithOptions:` methods should always be as lightweight as possible to reduce your app's launch time. Apps are expected to launch, initialize themselves, and start handling events in less than 5 seconds. If an app does not finish its launch cycle in a timely manner, the system kills it for being unresponsive. Thus, any tasks that might slow down your launch (such as accessing the network) should be scheduled performed on a secondary thread.

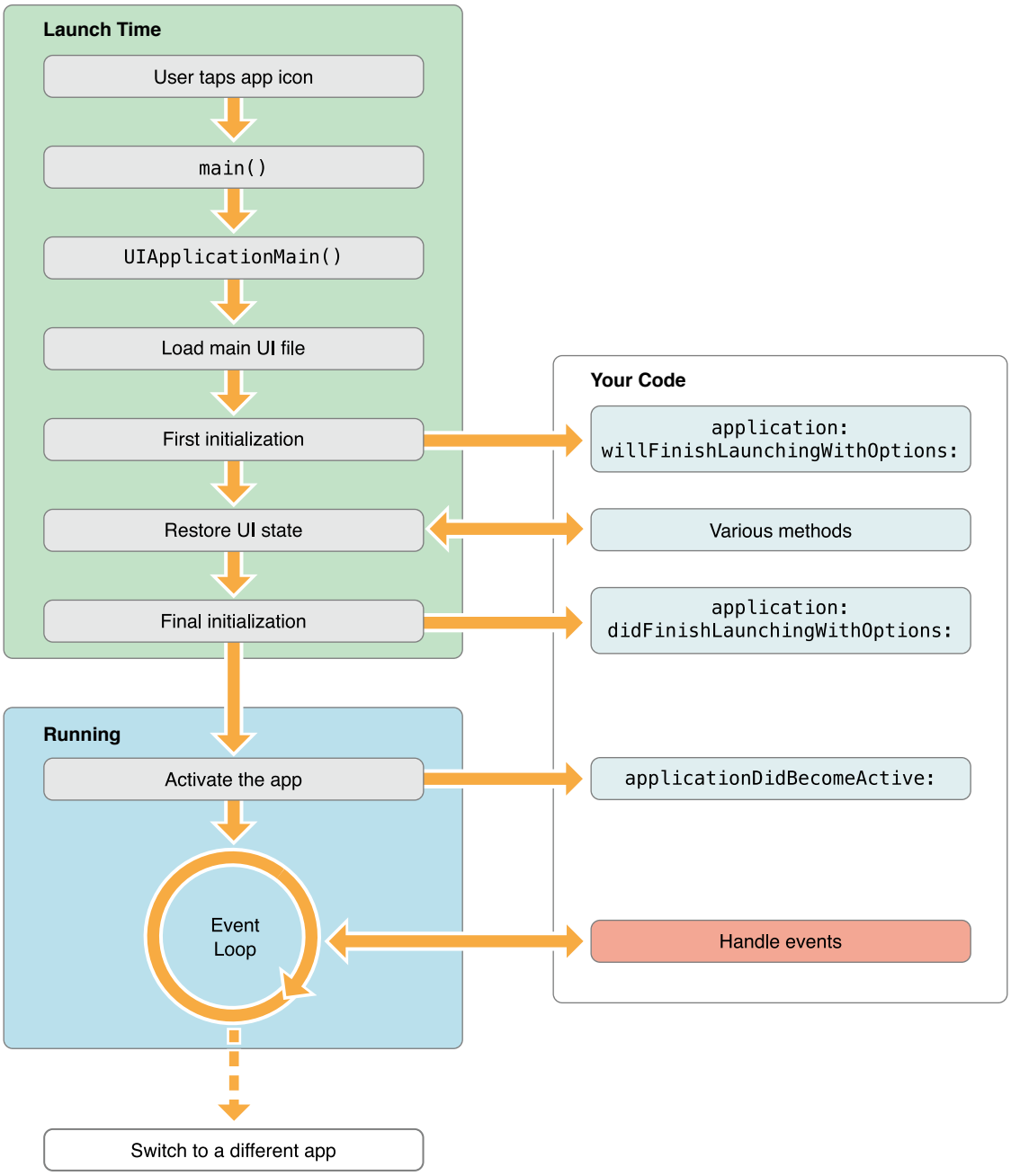
The Launch Cycle

When your app is launched, it moves from the not running state to the active or background state, transitioning briefly through the inactive state. As part of the launch cycle, the system creates a process and main thread for your app and calls your app's `main` function on that main thread. The default `main` function that comes with your Xcode project promptly hands control over to the UIKit framework, which does most of the work in initializing your app and preparing it to run.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Figure 4-1 shows the sequence of events that occurs when an app is launched into the foreground, including the app delegate methods that are called.

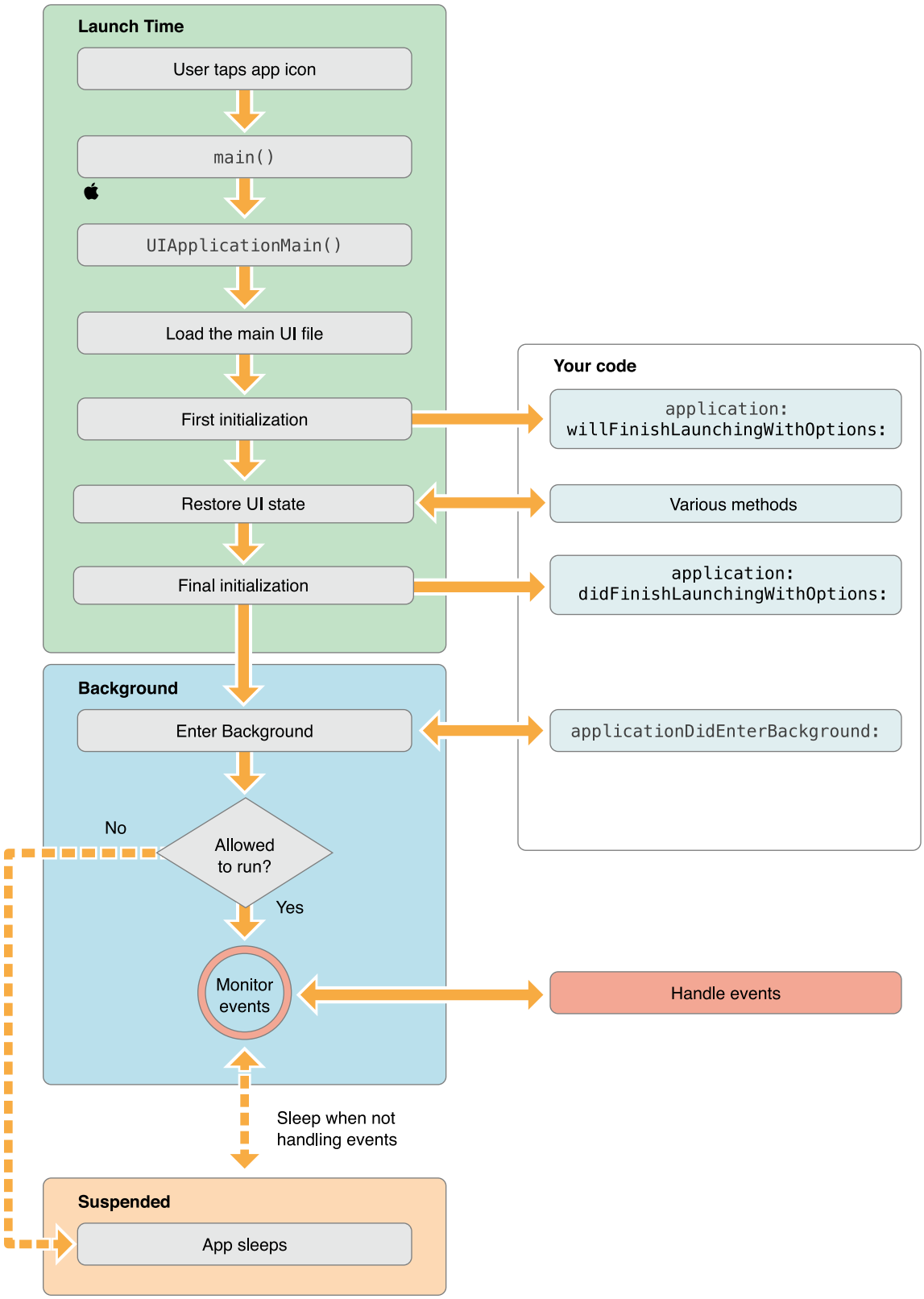
Figure 4-1 Launching an app into the foreground



When your app is launched into the background—usually to handle some type of background event—the launch cycle changes slightly to the one shown in Figure 4-2. The main difference is that instead of your app being made active, it enters the background state to handle the event and may be suspended at some point after that. When launching into the background, the system still loads your app's user interface files but it does not display the app's window.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Figure 4-2 Launching an app into the background



To determine whether your app is launching into the foreground or background, check the `applicationState` property of the shared `UIApplication` object in your `application:willFinishLaunchingWithOptions:` or `application:didFinishLaunchingWithOptions:` delegate method. When the app is launched into the foreground, this property contains the value `UIApplicationStateInactive`. When the app is launched into the background, the property contains the value `UIApplicationStateBackground` instead. You can use this difference to adjust the launch-time behavior of your delegate methods accordingly.

> **Note:** When an app is launched so that it can open a URL, the sequence of startup events is slightly different from those shown in Figure 4-1 (page 50) and Figure 4-2 (page 52). For information about the startup sequences that occur when opening a URL, see Handling URL Requests (page 100).

Launching in Landscape Mode

Apps that uses only landscape orientations for their interface must explicitly ask the system to launch the app in that orientation. Normally, apps launch in portrait mode and rotate their interface to match the device orientation as needed. For apps that support both portrait and landscape orientations, always configure your views for portrait mode and then let your view controllers handle any rotations. If, however, your app supports landscape but not portrait orientations, perform the following tasks to make it launch in landscape mode initially:

- Add the `UIInterfaceOrientation` key to your app's `Info.plist` file and set the value of this key to either `UIInterfaceOrientationLandscapeLeft` or `UIInterfaceOrientationLandscapeRight`.
- Lay out your views in landscape mode and make sure that their layout or autosizing options are set correctly.
- Override your view controller's `shouldAutorotateToInterfaceOrientation:` method and return `YES` for the left or right landscape orientations and `NO` for portrait orientations.

> **Important:** Apps should always use view controllers to manage their window-based content.

The `UIInterfaceOrientation` key in the `Info.plist` file tells iOS that it should configure the orientation of the app status bar (if one is displayed) as well as the orientation of views managed by any view controllers at launch time. View controllers respect this key and set their view's initial orientation to match. Using this key is equivalent to calling the `setStatusBarOrientation:animated:` method of `UIApplication` early in the execution of your `applicationDidFinishLaunching:` method.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Installing App-Specific Data Files at First Launch

You can use your app's first launch cycle to set up any data or configuration files required to run. App-specific data files should be created in the `Library/Application Support/<bundleID>/` directory of your app sandbox, where *<bundleID>* is your app's bundle identifier. You can further subdivide this directory to organize your data files as needed. You can also create files in other directories, such as to your app's iCloud container directory or to the local `Documents` directory, depending on your needs.

If your app's bundle contains data files that you plan to modify, copy those files out of the app bundle and modify the copies. You must not modify any files inside your app bundle. Because iOS apps are code signed, modifying files inside your app bundle invalidates your app's signature and will prevent your app from launching in the future. Copying those files to the `Application Support` directory (or another writable directory in your sandbox) and modifying them there is the only way to use such files safely.

For more information about where to put app-related data files, see *File System Programming Guide* .

What to Do When Your App Is Interrupted Temporarily

Alert-based interruptions result in a temporary loss of control by your app. Your app continues to run in the foreground, but it does not receive touch events from the system. (It does continue to receive notifications and other types of events, such as accelerometer events, though.) In response to this change, your app should do the following in its `applicationWillResignActive:` method:

- Save data and any relevant state information.
- Stop timers and other periodic tasks.
- Stop any running metadata queries.
- Do not initiate any new tasks.
- Pause movie playback (except when playing back over AirPlay).
- Enter into a pause state if your app is a game.
- Throttle back OpenGL ES frame rates.
- Suspend any dispatch queues or operation queues executing non-critical code. (You can continue processing network requests and other time-sensitive background tasks while inactive.)

When your app is moved back to the active state, its `applicationDidBecomeActive:` method should reverse any of the steps taken in the `applicationWillResignActive:` method. Thus, upon reactivation, your app should restart timers, resume dispatch queues, and throttle up OpenGL ES frame rates again. However, games should not resume automatically; they should remain paused until the user chooses to resume them.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

When the user presses the Sleep/Wake button, apps with files protected by the `NSFileProtectionComplete` protection option must close any references to those files. For devices configured with an appropriate password, pressing the Sleep/Wake button locks the screen and forces the system to throw away the decryption keys for files with complete protection enabled. While the screen is locked, any attempts to access the corresponding files will fail. So if you have such files, you should close any references to them in your `applicationWillResignActive:` method and open new references in your `applicationDidBecomeActive:` method.

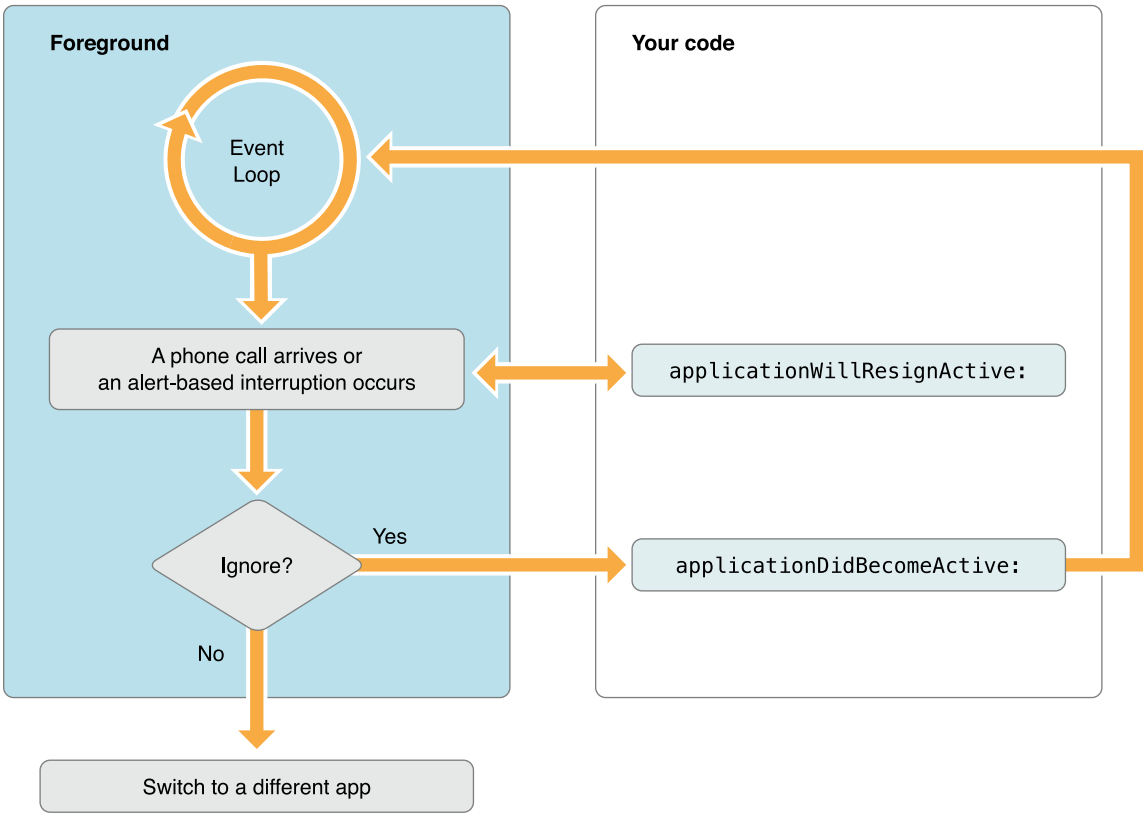
2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Important: Always save user data at appropriate checkpoints in your app. Although you can use app state transitions to force objects to write unsaved changes to disk, never wait for an app state transition to save data. For example, a view controller that manages user data should save its data when it is dismissed.

Responding to Temporary Interruptions

When an alert-based interruption occurs, such as an incoming phone call, the app moves temporarily to the inactive state so that the system can prompt the user about how to proceed. The app remains in this state until the user dismisses the alert. At this point, the app either returns to the active state or moves to the background state. Figure 4-3 shows the flow of events through your app when an alert-based interruption occurs.

Figure 4-3 Handling alert-based interruptions



Notifications that display a banner do not deactivate your app in the way that alert-based notifications do. Instead, the banner is laid along the top edge of your app window and your app continues receive touch events as before. However, if the user pulls down the banner to reveal the notification center, your app moves to the inactive state just as if an alert-based interruption had occurred. Your app remains in the inactive state

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

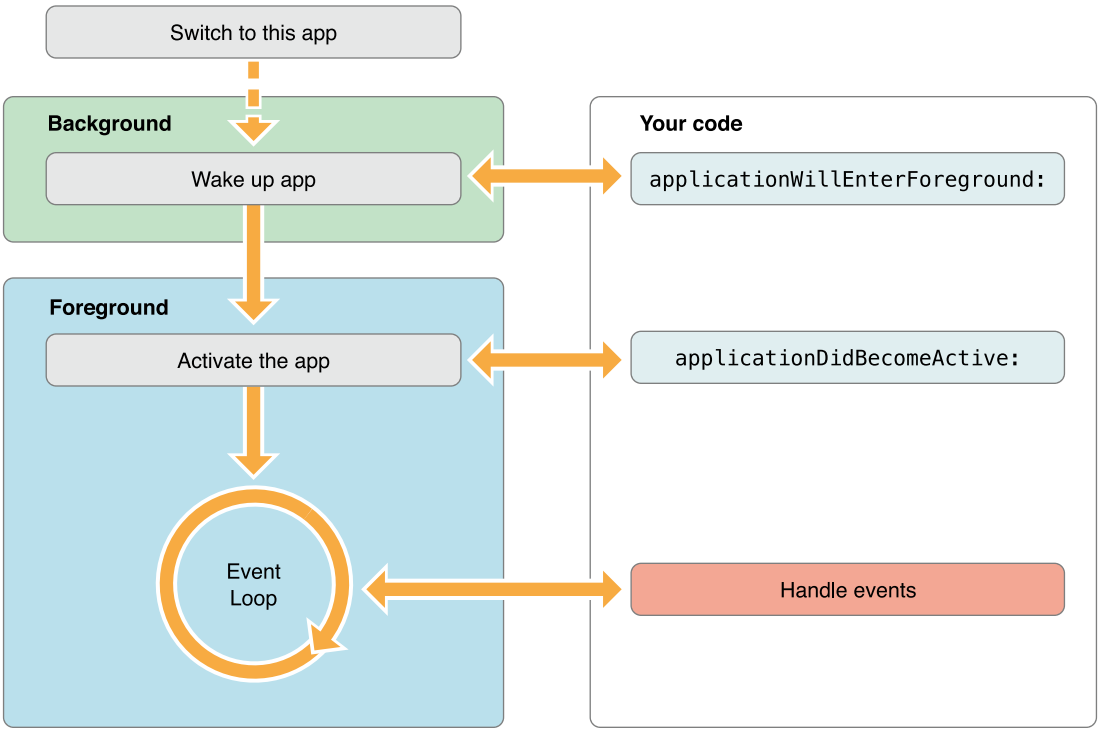
until the user dismisses the notification center or launches another app. At this point, your app moves to the appropriate active or background state. The user can use the Settings app to configure which notifications display a banner and which display an alert.

Pressing the Sleep/Wake button is another type of interruption that causes your app to be deactivated temporarily. When the user presses this button, the system disables touch events, moves the app to the background, sets the value of the app's `applicationState` property to `UIApplicationStateBackground`, and locks the screen. A locked screen has additional consequences for apps that use data protection to encrypt files. Those consequences are described in What to Do When Your App Is Interrupted Temporarily (page 54).

What to Do When Your App Enters the Foreground

Returning to the foreground is your app's chance to restart the tasks that it stopped when it moved to the background. The steps that occur when moving to the foreground are shown in Figure 4-4. The `applicationWillEnterForeground:` method should undo anything that was done in your `applicationDidEnterBackground:` method, and the `applicationDidBecomeActive:` method should continue to perform the same activation tasks that it would at launch time.

Figure 4-4 Transitioning from the background to the foreground



> **Note:** The `UIApplicationWillEnterForegroundNotification` notification is also available for tracking when your app reenters the foreground. Objects in your app can use the default notification center to register for this notification.

Be Prepared to Process Queued Notifications

An app in the suspended state must be ready to handle any queued notifications when it returns to a foreground or background execution state. A suspended app does not execute any code and therefore cannot process notifications related to orientation changes, time changes, preferences changes, and many others that would affect the app's appearance or state. To make sure these changes are not lost, the system queues many relevant notifications and delivers them to the app as soon as it starts executing code again (either in the foreground or background). To prevent your app from becoming overloaded with notifications when it resumes, the system coalesces events and delivers a single notification (of each relevant type) that reflects the net change since your app was suspended.

Table 4-1 lists the notifications that can be coalesced and delivered to your app. Most of these notifications are delivered directly to the registered observers. Some, like those related to device orientation changes, are typically intercepted by a system framework and delivered to your app in another way.

Table 4-1 Notifications delivered to waking apps

Event	Notifications
An accessory is connected or disconnected.	`EAAccessoryDidConnectNotification` `EAAccessoryDidDisconnectNotification`
The device orientation changes.	`UIDeviceOrientationDidChangeNotification` In addition to this notification, view controllers update their interface orientations automatically.
There is a significant time change.	`UIApplicationSignificantTimeChangeNotification`
The battery level or battery state changes.	`UIDeviceBatteryLevelDidChangeNotification` `UIDeviceBatteryStateDidChangeNotification`
The proximity state changes.	`UIDeviceProximityStateDidChangeNotification`
The status of protected files changes.	`UIApplicationProtectedDataWillBecomeUnavailable` `UIApplicationProtectedDataDidBecomeAvailable`

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Event	Notifications
An external display is connected or disconnected.	`UIScreenDidConnectNotification` `UIScreenDidDisconnectNotification`
The screen mode of a display changes.	`UIScreenModeDidChangeNotification`
Preferences that your app exposes through the Settings app changed.	`NSUserDefaultsDidChangeNotification`
The current language or locale settings changed.	`NSCurrentLocaleDidChangeNotification`
The status of the user's iCloud account changed.	`NSUbiquityIdentityDidChangeNotification`

Queued notifications are delivered on your app's main run loop and are typically delivered before any touch events or other user input. Most apps should be able to handle these events quickly enough that they would not cause any noticeable lag when resumed. However, if your app appears sluggish when it returns from the background state, use Instruments to determine whether your notification handler code is causing the delay.

An app returning to the foreground also receives view-update notifications for any views that were marked dirty since the last update. An app running in the background can still call the `setNeedsDisplay` or `setNeedsDisplayInRect:` methods to request an update for its views. However, because the views are not visible, the system coalesces the requests and updates the views only after the app returns to the foreground.

Handle iCloud Changes

If the status of iCloud changes for any reason, the system delivers a `NSUbiquityIdentityDidChangeNotification` notification to your app. The state of iCloud changes when the user logs into or out of an iCloud account or enables or disables the syncing of documents and data. This notification is your app's cue to update caches and any iCloud-related user interface elements to accommodate the change. For example, when the user logs out of iCloud, you should remove references to all iCloud–based files or data.

If your app has already prompted the user about whether to store files in iCloud, do not prompt again when the status of iCloud changes. After prompting the user the first time, store the user's choice in your app's local preferences. You might then want to expose that preference using a Settings bundle or as an option in your app. But do not repeat the prompt again unless that preference is not currently in the user defaults database.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Handle Locale Changes

If a user changes the current locale while your app is suspended, you can use the `NSCurrentLocaleDidChangeNotification` notification to force updates to any views containing locale-sensitive information, such as dates, times, and numbers when your app returns to the foreground. Of course, the best way to avoid locale-related issues is to write your code in ways that make it easy to update views. For example:

- Use the `autoupdatingCurrentLocale` class method when retrieving `NSLocale` objects. This method returns a locale object that updates itself automatically in response to changes, so you never need to recreate it. However, when the locale changes, you still need to refresh views that contain content derived from the current locale.
- Re-create any cached date and number formatter objects whenever the current locale information changes.

For more information about internationalizing your code to handle locale changes, see *Internationalization and Localization Guide*.

Handle Changes to Your App's Settings

If your app has settings that are managed by the Settings app, it should observe the `NSUserDefaultsDidChangeNotification` notification. Because the user can modify settings while your app is suspended or in the background, you can use this notification to respond to any important changes in those settings. In some cases, responding to this notification can help close a potential security hole. For example, an email program should respond to changes in the user's account information. Failure to monitor these changes could cause privacy or security issues. Specifically, the current user might be able to send email using the old account information, even if the account no longer belongs to that person.

Upon receiving the `NSUserDefaultsDidChangeNotification` notification, your app should reload any relevant settings and, if necessary, reset its user interface appropriately. In cases where passwords or other security-related information has changed, you should also hide any previously displayed information and force the user to enter the new password.

What to Do When Your App Enters the Background

When moving from foreground to background execution, use the `applicationDidEnterBackground:` method of your app delegate to do the following:

- **Prepare to have your app's picture taken.** When your `applicationDidEnterBackground:` method returns, the system takes a picture of your app's user interface and uses the resulting image for transition animations. If any views in your interface contain sensitive information, you should hide or modify those

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

views before the `applicationDidEnterBackground:` method returns. If you add new views to your view hierarchy as part of this process, you must force those views to draw themselves, as described in Prepare for the App Snapshot (page 63).

- **Save any relevant app state information.** Prior to entering the background, your app should already have saved all critical user data. Use the transition to the background to save any last minute changes to your app's state.
- **Free up memory as needed.** Release any cached data that you do not need and do any simple cleanup that might reduce your app's memory footprint. Apps with large memory footprints are the first to be terminated by the system, so release image resources, data caches, and any other objects that you no longer need. For more information, see Reduce Your Memory Footprint (page 63).

Your app delegate's `applicationDidEnterBackground:` method has approximately 5 seconds to finish any tasks and return. In practice, this method should return as quickly as possible. If the method does not return before time runs out, your app is killed and purged from memory. If you still need more time to perform tasks, call the `beginBackgroundTaskWithExpirationHandler:` method to request background execution time and then start any long-running tasks in a secondary thread. Regardless of whether you start any background tasks, the `applicationDidEnterBackground:` method must still exit within 5 seconds.

> **Note:** The system sends the `UIApplicationDidEnterBackgroundNotification` notification in addition to calling the `applicationDidEnterBackground:` method. You can use that notification to distribute cleanup tasks to other objects of your app.

Depending on the features of your app, there are other things your app should do when moving to the background. For example, any active Bonjour services should be suspended and the app should stop calling OpenGL ES functions. For a list of things your app should do when moving to the background, see Being a Responsible Background App (page 45).

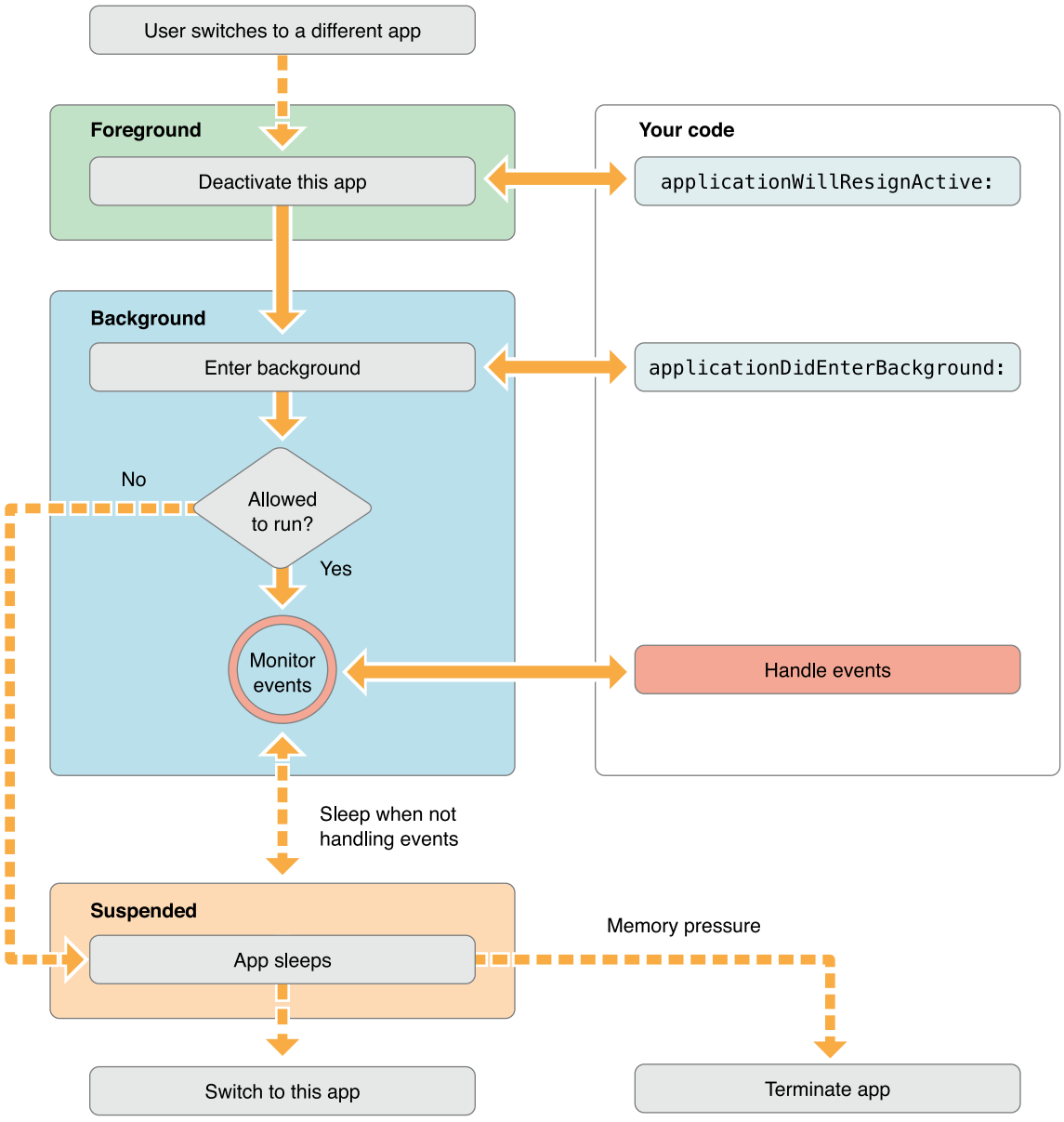
The Background Transition Cycle

When the user presses the Home button, presses the Sleep/Wake button, or the system launches another app, the foreground app transitions to the inactive state and then to the background state. These transitions result in calls to the app delegate's `applicationWillResignActive:` and `applicationDidEnterBackground:` methods, as shown in Figure 4-5. After returning from the `applicationDidEnterBackground:` method,

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

most apps move to the suspended state shortly afterward. Apps that request specific background tasks (such as playing music) or that request a little extra execution time from the system may continue to run for a while longer.

Figure 4-5 Moving from the foreground to the background



Prepare for the App Snapshot

Shortly after an app delegate's `applicationDidEnterBackground:` method returns, the system takes a snapshot of the app's windows. Similarly, when an app is woken up to perform background tasks, the system may take a new snapshot to reflect any relevant changes. For example, when an app is woken to process downloaded items, the system takes a new snapshot so that can reflect any changes caused by the incorporation of the items. The system uses these snapshot images in the multitasking UI to show the state of your app.

If you make changes to your views upon entering the background, you can call the `snapshotViewAfterScreenUpdates:` method of your main view to force those changes to be rendered. Calling the setNeedsDisplay method on a view is ineffective for snapshots because the snapshot is taken before the next drawing cycle, thus preventing any changes from being rendered. Calling the `snapshotViewAfterScreenUpdates:` method with a value of `YES` forces an immediate update to the underlying buffers that the snapshot machinery uses.

Reduce Your Memory Footprint

Every app should free up as much memory as is practical upon entering the background. The system tries to keep as many apps in memory at the same time as it can, but when memory runs low it terminates suspended apps to reclaim that memory. Apps that consume large amounts of memory while in the background are the first apps to be terminated.

Practically speaking, your app should remove strong references to objects as soon as they are no longer needed. Removing strong references gives the compiler the ability to release the objects right away so that the corresponding memory can be reclaimed. However, if you want to cache some objects to improve performance, you can wait until the app transitions to the background before removing references to them.

Some examples of objects that you should remove strong references to as soon as possible include:

- Image objects you created. (Some methods of `UIImage` return images whose underlying image data is purged automatically by the system. For more information, see the discussion in the overview of *UIImage Class Reference*.)
- Large media or data files that you can load again from disk
- Any other objects that your app does not need and can recreate easily later

To help reduce your app's memory footprint, the system automatically purges some data allocated on behalf of your app when your app moves to the background.

- The system purges the backing store for all Core Animation layers. This effort does not remove your app's layer objects from memory, nor does it change the current layer properties. It simply prevents the contents of those layers from appearing onscreen, which given that the app is in the background should not happen anyway.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- It removes any system references to cached images.
- It removes strong references to some other system-managed data caches.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Strategies for Implementing Specific App Features

SwiftObjective-C

Different apps have different needs but some behaviors are common to many types of app. The following sections provide guidance about how to implement specific types of features in your app.

Privacy Strategies

Protecting a user's privacy is an important consideration in the design of an app. Privacy protection includes protecting the user's data, including the user's identity and personal information. The system frameworks already provide privacy controls for managing data such as contacts but your app should take steps to protect the data that you use locally.

Protecting Data Using On-Disk Encryption

Data protection uses built-in hardware to store files in an encrypted format on disk and to decrypt them on demand. While the user's device is locked, protected files are inaccessible, even to the app that created them. The user must unlock the device (by entering the appropriate passcode) before an app can access one of its protected files.

Data protection is available on most iOS devices and is subject to the following requirements:

- The file system on the user's device must support data protection. Most devices support this behavior.
- The user must have an active passcode lock set for the device.

To protect a file, you add an attribute to the file indicating the desired level of protection. Add this attribute using either the `NSData` class or the `NSFileManager` class. When writing new files, you can use the `writeToFile:options:error:` method of `NSData` with the appropriate protection value as one of the write options. For existing files, you can use the `setAttributes:ofItemAtPath:error:` method of `NSFileManager` to set or change the value of the `NSFileProtectionKey`. When using these methods, specify one of the following protection levels for the file:

- No protection—The file is encrypted but is not protected by the passcode and is available when the device is locked. Specify the `NSDataWritingFileProtectionNone` option (`NSData`) or the `NSFileProtectionNone` attribute (`NSFileManager`).

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- Complete—The file is encrypted and inaccessible while the device is locked. Specify the `NSDataWritingFileProtectionComplete` option (`NSData`) or the `NSFileProtectionComplete` attribute (`NSFileManager`).
- Complete unless already open—The file is encrypted. A closed file is inaccessible while the device is locked. After the user unlocks the device, your app can open the file and use it. If the user locks the device while the file is open, though, your app can continue to access it. Specify the `NSDataWritingFileProtectionCompleteUnlessOpen` option (`NSData`) or the `NSFileProtectionCompleteUnlessOpen` attribute (`NSFileManager`).
- Complete until first login—The file is encrypted and inaccessible until after the device has booted and the user has unlocked it once. Specify the `NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication` option (`NSData`) or the `NSFileProtectionCompleteUntilFirstUserAuthentication` attribute (`NSFileManager`).

If you protect a file, your app must be prepared to lose access to that file. When complete file protection is enabled, your app loses the ability to read and write the file's contents when the user locks the device. You can track changes to the state of protected files using one of the following techniques:

- The app delegate can implement the `applicationProtectedDataWillBecomeUnavailable:` and `applicationProtectedDataDidBecomeAvailable:` methods.
- Any object can register for the `UIApplicationProtectedDataWillBecomeUnavailable` and `UIApplicationProtectedDataDidBecomeAvailable` notifications.
- Any object can check the value of the `protectedDataAvailable` property of the shared `UIApplication` object to determine whether files are currently accessible.

For new files, it is recommended that you enable data protection before writing any data to them. If you are using the `writeToFile:options:error:` method to write the contents of an `NSData` object to disk, this happens automatically. For existing files, adding data protection replaces an unprotected file with a new protected version.

Identifying Unique Users of Your App

You should identify a user of your app only when doing so offers a clear benefit to that user. In cases where you only need to differentiate one user of your app from another, iOS provides identifiers that can help you do that. However, if you need a higher level of security, you might need to do more work on your own. For example, an app that provides financial services would likely want to prompt the user for login credentials to ensure that the user is authorized to access a specific account.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Important: When identifying a user, always be transparent about what you intend to do with any information you obtain. It is not acceptable to identify a user so that you can track them surreptitiously.

Here are some common scenarios that might require you to identify a user, along with solutions for how to implement them.

- **You want to link a user to a specific account on your server.** Include a login screen that requires the user to enter their account information securely. Always protect the account information you gather from the user by storing it in an encrypted form.
- **You want to differentiate instances of your app running on different devices.** Use the `identifierForVendor` property of the `UIDevice` class to obtain an ID that differentiates a user on one device from users on other devices. This technique does now allow you to identify specific users. A single user can have multiple devices, each with a different ID value.
- **You want to identify a user for the purposes of advertising.** Use the `advertisingIdentifier` property of the `ASIdentifierManager` class to obtain an ID for the user.

Because users are allowed to run apps on all of their iOS devices, Apple does not provide a way to identify the same user on multiple devices. If you need to identify a specific user, you must provide your own solution using universally unique IDs (UUIDs), a login account, or some other type of identification system.

Supporting Multiple Versions of iOS

An app that supports the latest version of iOS plus one or more earlier versions must use runtime checks to prevent the use of newer APIs on older versions of iOS. Runtime checks prevent your app from crashing when it tries to use a feature that is not available on the current operating system.

There are several types of checks that you can make:

- To determine whether a class exists, see if its `Class` object is `nil`. The linker returns `nil` for any unknown class objects, making it possible to use a conditional check similar to the following:

```
if ([UIPrintInteractionController class]) {
   // Create an instance of the class and use it.
}
else {
   // The print interaction controller is not available so use an
alternative technique.
}
```

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- To determine whether a method is available on an existing class, use the `instancesRespondToSelector:` class method or the `respondsToSelector:` instance method.
- To determine whether a C-based function is available, perform a Boolean comparison of the function name to `NULL`. If the symbol is not `NULL`, you can call the function. For example:

```
if (UIGraphicsBeginPDFPage != NULL) {
    UIGraphicsBeginPDFPage();
}
```

For more information and examples of how to write code that supports multiple deployment targets, see *SDK Compatibility Guide*.

Preserving Your App's Visual Appearance Across Launches

Even if your app supports background execution, it cannot run forever. At some point, the system might need to terminate your app to free up memory for the current foreground app. However, the user should never have to care if an app is already running or was terminated. From the user's perspective, quitting an app should just seem like a temporary interruption. When the user returns to an app, that app should always return the user to the last point of use, so that the user can continue with whatever task was in progress. This behavior provides a better experience for the user and with the state restoration support built in to UIKit is relatively easy to achieve.

The state preservation system in UIKit provides a simple but flexible infrastructure for preserving and restoring the state of your app's view controllers and views. The job of the infrastructure is to drive the preservation and restoration processes at the appropriate times. To do that, UIKit needs help from your app. Only you understand the content of your app, and so only you can write the code needed to save and restore that content. And when you update your app's UI, only you know how to map older preserved content to the newer objects in your interface.

There are three places where you have to think about state preservation in your app:

- Your app delegate object, which manages the app's top-level state
- Your app's view controller objects, which manage the overall state for your app's user interface
- Your app's custom views, which might have some custom data that needs to be preserved

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

UIKit allows you to choose which parts of your user interface you want to preserve. And if you already have custom code for handling state preservation, you can continue to use that code and migrate portions to the UIKit state preservation system as needed.

Enabling State Preservation and Restoration in Your App

State preservation and restoration is not an automatic feature and apps must opt-in to use it. Apps indicate their support for the feature by implementing the following methods in their app delegate:

```
application:shouldSaveApplicationState:
application:shouldRestoreApplicationState:
```

Normally, your implementations of these methods just return `YES` to indicate that state preservation and restoration can occur. However, apps that want to preserve and restore their state conditionally can return `NO` in situations where the operations should not occur. For example, after releasing an update to your app, you might want to return `NO` from your `application:shouldRestoreApplicationState:` method if your app is unable to usefully restore the state from a previous version.

The Preservation and Restoration Process

State preservation and restoration is an opt-in feature and works with the help of your app. Your app identifies objects that should be preserved and UIKit does the work of preserving and restoring those objects at appropriate times. Because UIKit handles so much of the process, it helps to understand what it does behind the scenes so that you know how your custom code fits into the overall scheme.

When thinking about state preservation and restoration, it helps to separate the two processes first. UIKit preserves your app's state at appropriate times, such as when your app moves from the foreground to the background. When UIKit determines new state information is needed, it looks at your app's views and view controllers to see which ones should be preserved. For each of those objects, UIKit writes preservation-related data to an encrypted on-disk file. The next time your app launches from scratch, UIKit looks for that file and, if it is present, uses it to try and restore your app's state. Because the file is encrypted, state preservation and restoration only happens when the device is unlocked.

During the restoration process, UIKit uses the preserved data to reconstitute your interface but the creation of actual objects is handled by your code. Because your app might load objects from a storyboard file automatically, only your code knows which objects need to be created and which might already exist and can simply be returned. After creating each object, UIKit initializes them with the preserved state information.

During the preservation and restoration process, your app has a handful of responsibilities.

- During preservation, your app is responsible for:
    - Telling UIKit that it supports state preservation.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

    - Telling UIKit which view controllers and views should be preserved.
    - Encoding relevant data for any preserved objects.
- During restoration, your app is responsible for:
    - Telling UIKit that it supports state restoration.
    - Providing (or creating) the objects that are requested by UIKit.
    - Decoding the state of your preserved objects and using it to return the object to its previous state.

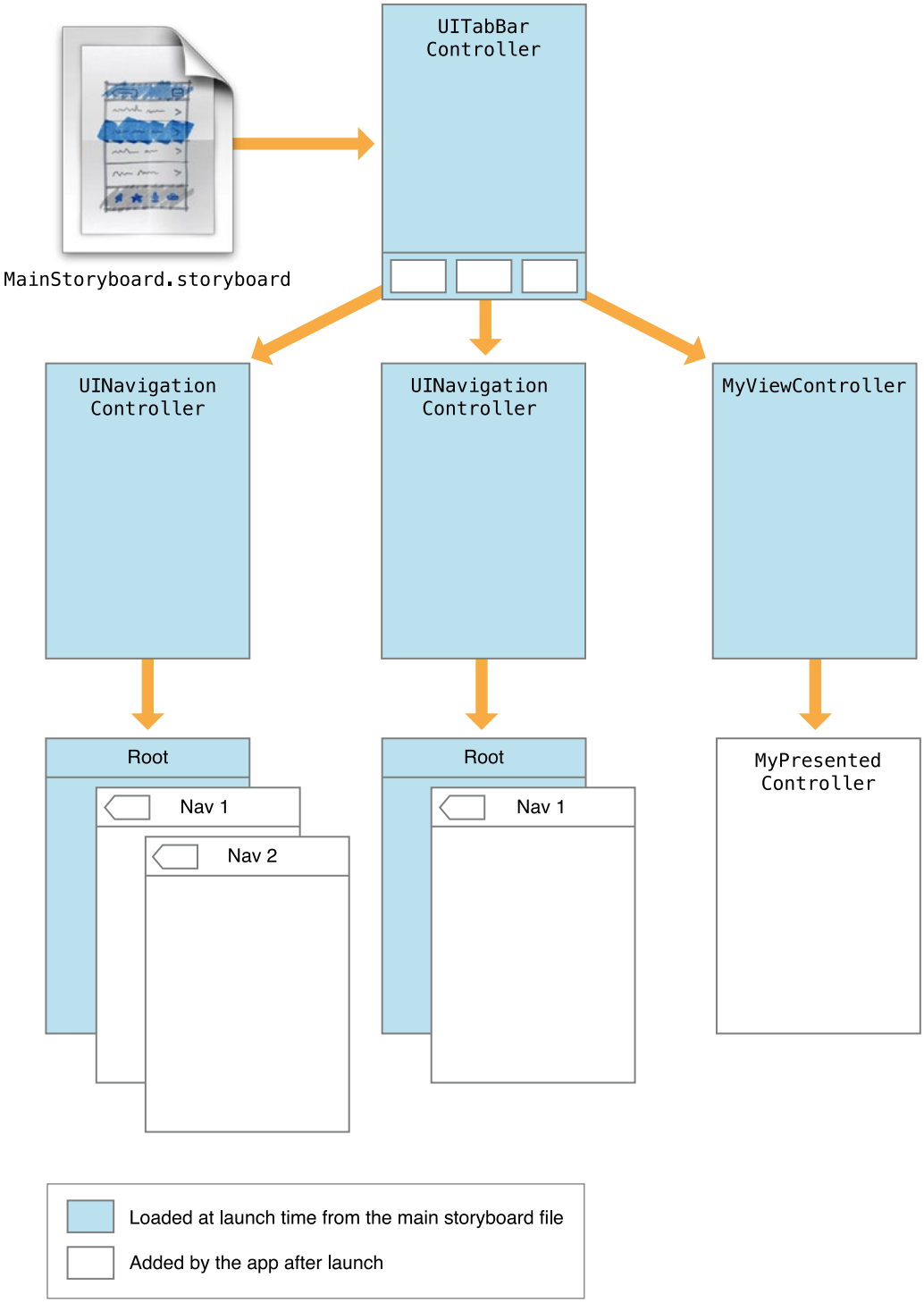
Of your app's responsibilities, the most significant are telling UIKit which objects to preserve and providing those objects during subsequent launches. Those two behaviors are where you should spend most of your time when designing your app's preservation and restoration code. They are also where you have the most control over the actual process. To understand why that is the case, it helps to look at an example.

Figure 5-1 shows the view controller hierarchy of a tab bar interface after the user has interacted with several of the tabs. As you can see, some of the view controllers are loaded automatically as part of the app's main storyboard file but some of the view controllers were presented or pushed onto the view controllers in different

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

tabs. Without state restoration, only the view controllers from the main storyboard file would be restored during subsequent launches. By adding support for state restoration to your app, you can preserve all of the view controllers.

Figure 5-1 A sample view controller hierarchy



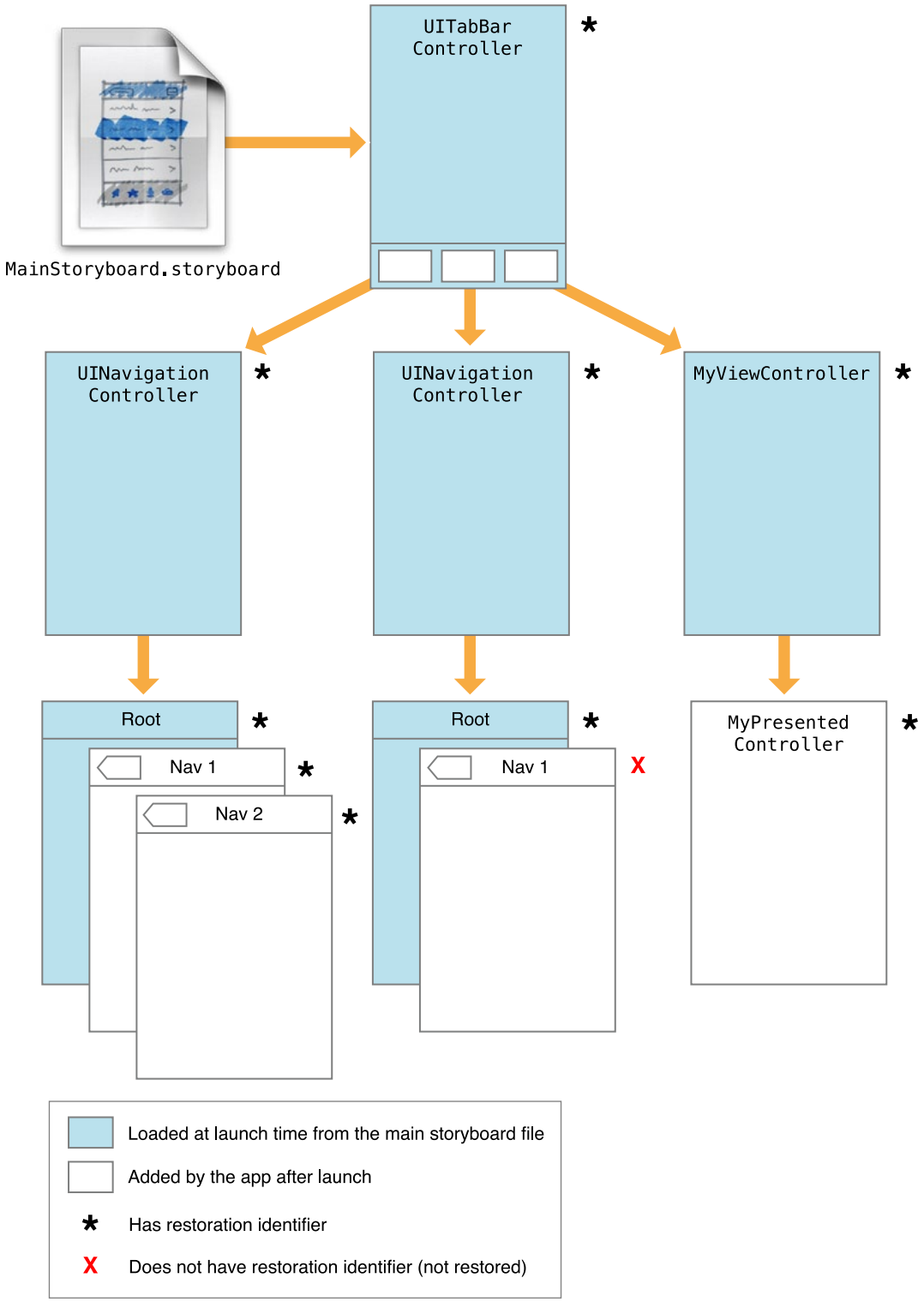
UIKit preserves only those objects that have an assigned restoration identifier. A *restoration identifier* is a string that identifies the view or view controller to UIKit and your app. The value of this string is significant only to your code but the presence of this string tells UIKit that it needs to preserve the tagged object. During the preservation process, UIKit walks your app's view controller hierarchy and preserves all objects that have a

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

restoration identifier. If a view controller does not have a restoration identifier, that view controller and all of its views and child view controllers are not preserved. Figure 5-2 shows an updated version of the previous view hierarchy, now with restoration identifies applied to most (but not all) of the view controllers.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Figure 5-2 Adding restoration identifies to view controllers



Depending on your app, it might or might not make sense to preserve every view controller. If a view controller presents transitory information, you might not want to return to that same point on restore, opting instead to return the user to a more stable point in your interface.

For each view controller you choose to preserve, you also need to decide how you want to restore it later. UIKit offers two ways to recreate objects. You can let your app delegate recreate it or you can assign a restoration class to the view controller and let that class recreate it. A *restoration class* implements the `UIViewControllerRestoration` protocol and is responsible for finding or creating a designated object at restore time. Here are some tips for when to use each one:

- **If the view controller is always loaded from your app's main storyboard file at launch time, do not assign a restoration class.** Instead, let your app delegate find the object or take advantage of UIKit's support for implicitly finding restored objects.
- **For view controllers that are not loaded from your main storyboard file at launch time, assign a restoration class.** The simplest option is to make each view controller its own restoration class.

During the preservation process, UIKit identifies the objects to save and writes each affected object's state to disk. Each view controller object is given a chance to write out any data it wants to save. For example, a tab view controller saves the identity of the selected tab. UIKit also saves information such as the view controller's restoration class to disk. And if any of the view controller's views has a restoration identifier, UIKit asks them to save their state information too.

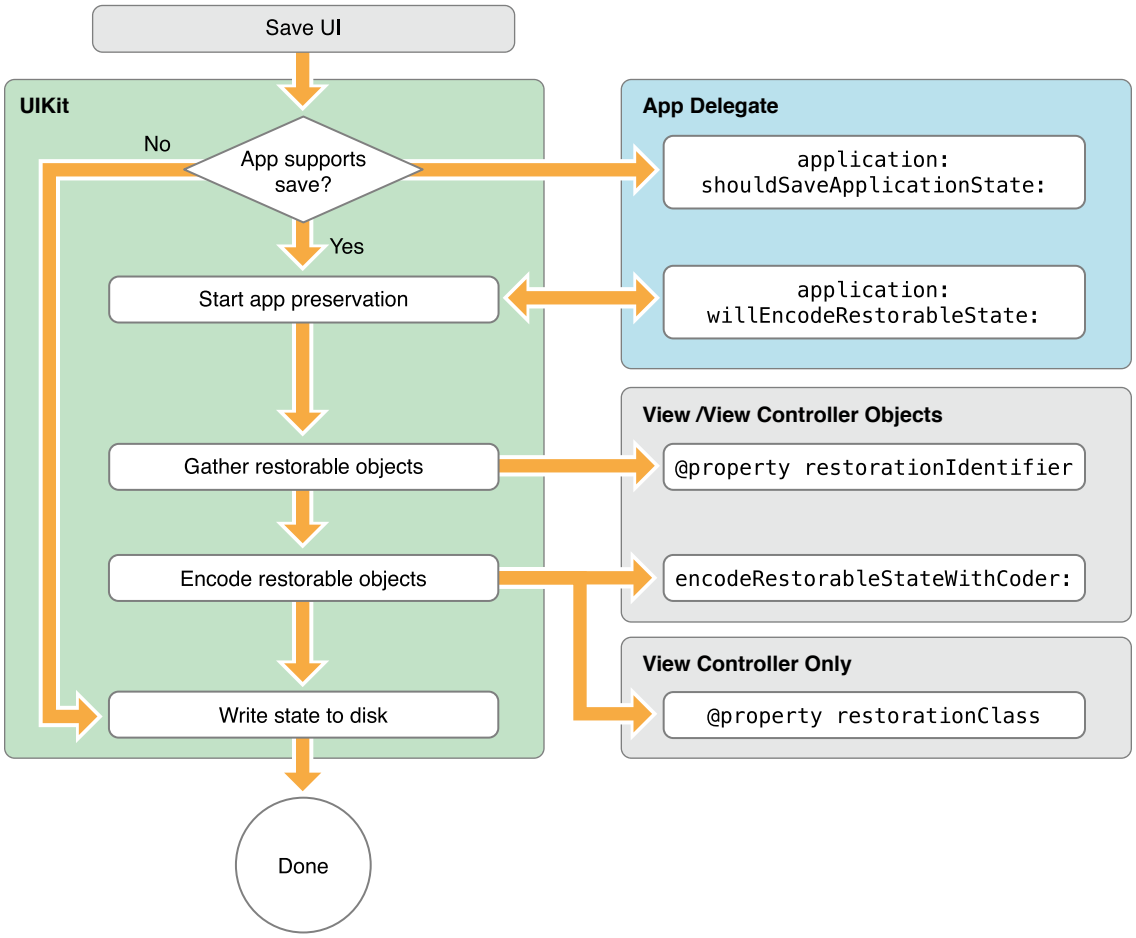
The next time the app is launched, UIKit loads the app's main storyboard or nib file as usual, calls the app delegate's `application:willFinishLaunchingWithOptions:` method, and then tries to restore the app's previous state. The first thing it does is ask your app to provide the set of view controller objects that match the ones that were preserved. If a given view controller had an assigned restoration class, that class is asked to provide the object; otherwise, the app delegate is asked to provide it.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Flow of the Preservation Process

Figure 5-3 shows the high-level events that happen during state preservation and shows how the objects of your app are affected. Before preservation even occurs, UIKit asks your app delegate if it *should* occur by calling the `application:shouldSaveApplicationState:` method. If that method returns `YES`, UIKit begins gathering and encoding your app's views and view controllers. When it is finished, it writes the encoded data to disk.

Figure 5-3 High-level flow interface preservation



The next time your app launches, the system automatically looks for a preserved state file, and if present, uses it to restore your interface. Because this state information is only relevant between the previous and current launch cycles of your app, the file is typically discarded after your app finishes launching. The file is also discarded any time there is an error restoring your app. For example, if your app crashes during the restoration process, the system automatically throws away the state information during the next launch cycle to avoid another crash.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

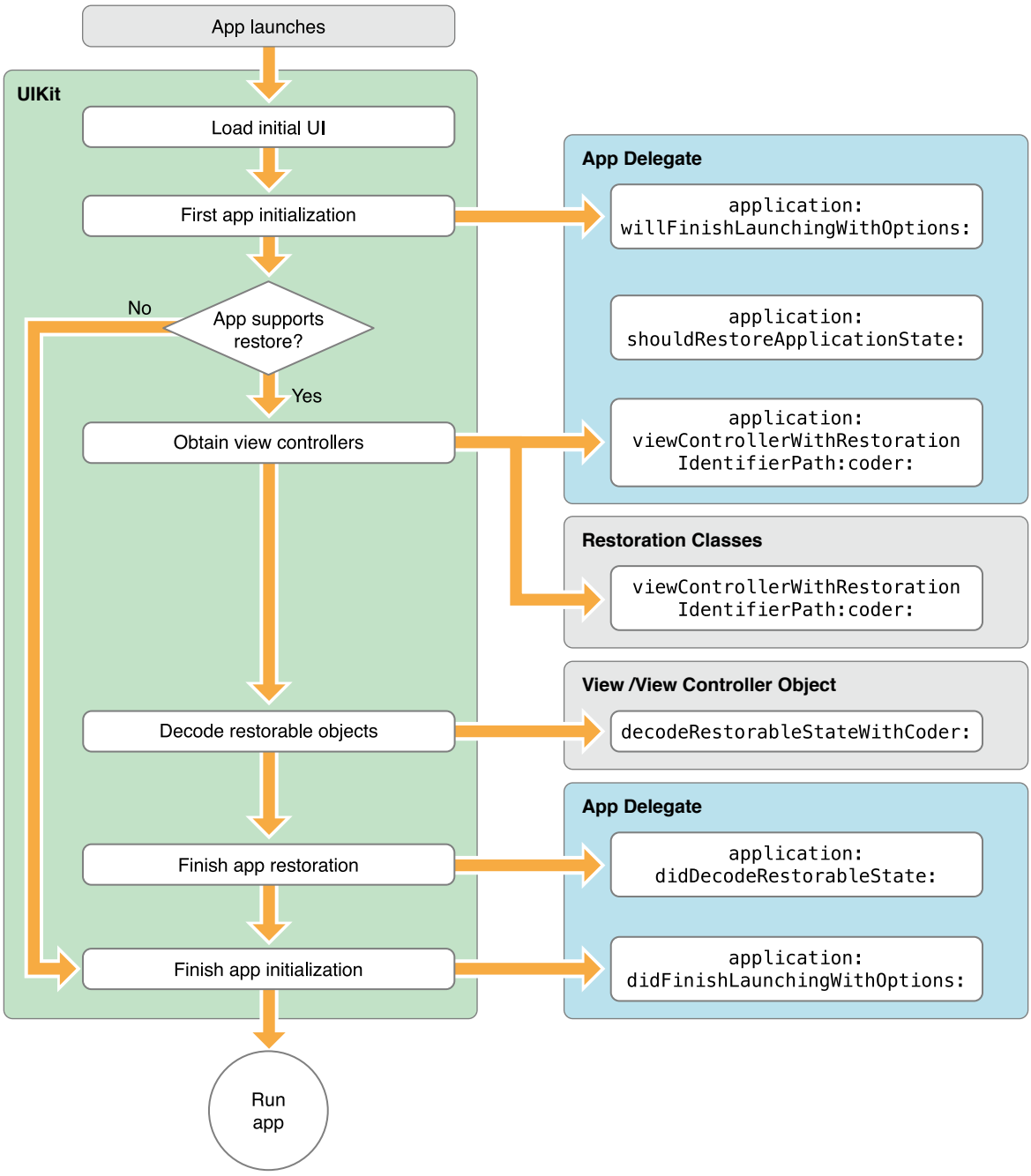
Flow of the Restoration Process

Figure 5-4 shows the high-level events that happen during state restoration and shows how the objects of your app are affected. After the standard initialization and UI loading is complete, UIKit asks your app delegate if state restoration should occur at all by calling the `application:shouldRestoreApplicationState:`

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

method. This is your app delegate's opportunity to examine the preserved data and determine if state restoration is possible. If it is, UIKit uses the app delegate and restoration classes to obtain references to your app's view controllers. Each object is then provided with the data it needs to restore itself to its previous state.

Figure 5-4 High-level flow for restoring your user interface



Although UIKit helps restore the individual view controllers, it does not automatically restore the relationships between those view controllers. Instead, each view controller is responsible for encoding enough state information to return itself to its previous state. For example, a navigation controller encodes information about the order of the view controllers on its navigation stack. It then uses this information later to return those view controllers to their previous positions on the stack. Other view controllers that have embedded child view controllers are similarly responsible for encoding any information they need to restore their children later.

> **Note:** Not all view controllers need to encode their child view controllers. For example, tab bar controllers do not encode information about their child view controllers. Instead, it is assumed that your app follows the usual pattern of creating the appropriate child view controllers prior to creating the tab bar controller itself.

Because you are responsible for recreating your app's view controllers, you have some flexibility to change your interface during the restoration process. For example, you could reorder the tabs in a tab bar controller and still use the preserved data to return each tab to its previous state. Of course, if you make dramatic changes to your view controller hierarchy, such as during an app update, you might not be able to use the preserved data.

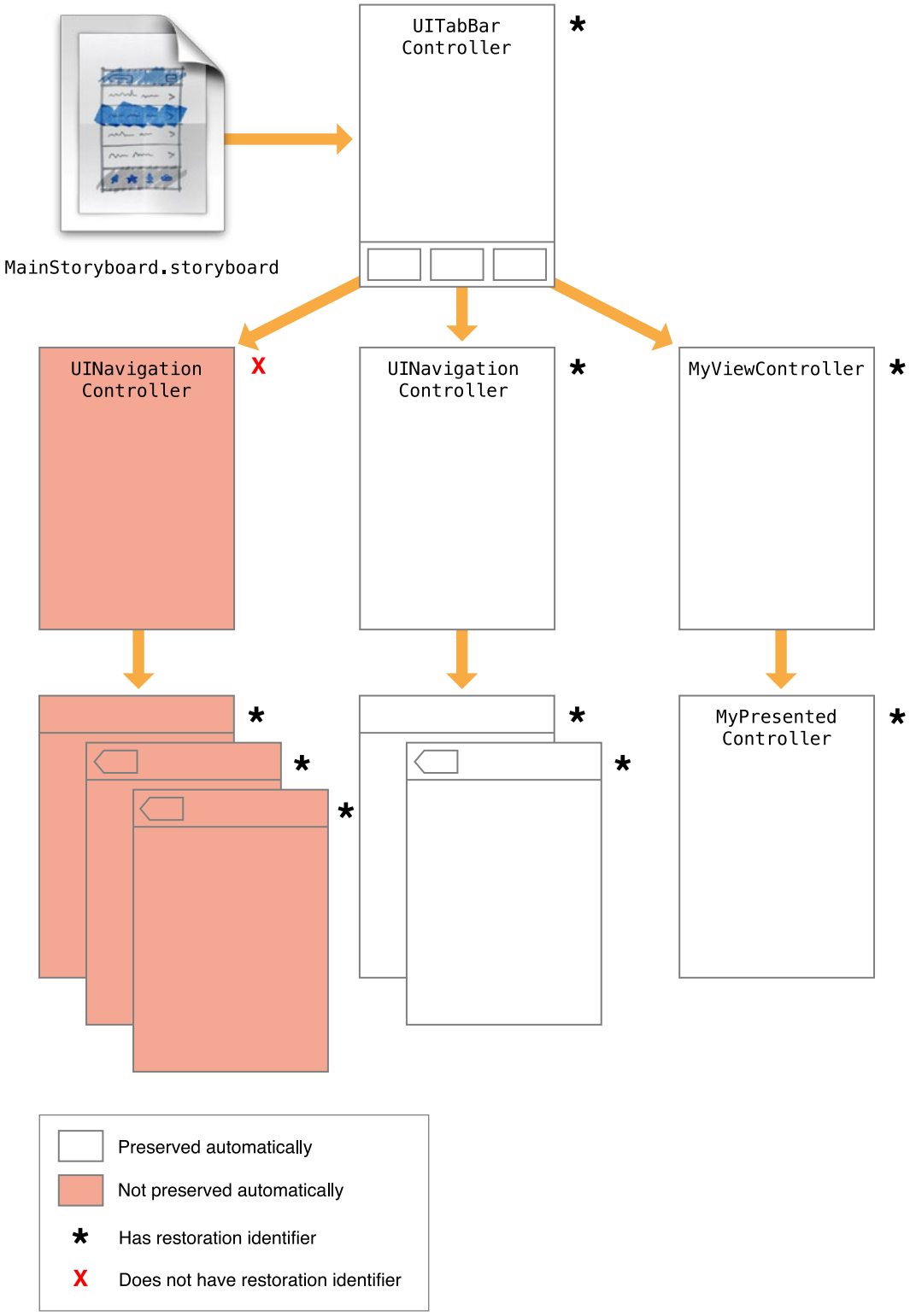
2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

What Happens When You Exclude Groups of View Controllers?

When the restoration identifier of a view controller is `nil`, that view controller and any child view controllers it manages are not preserved automatically. For example, in Figure 5-5, because a navigation controller did not have a restoration identifier, it and all of its child view controllers and views are omitted from the preserved data.

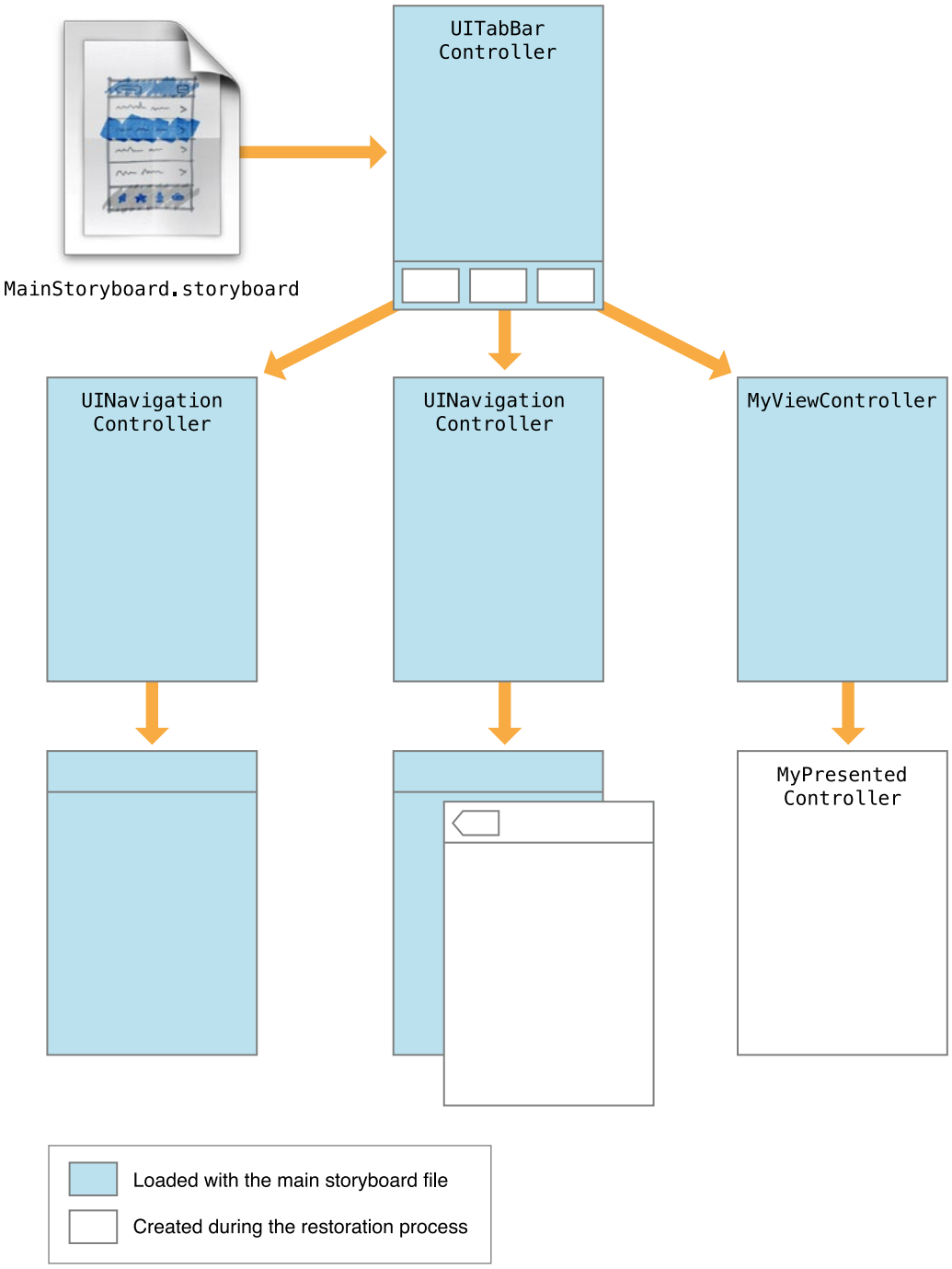
2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Figure 5-5 Excluding view controllers from the automatic preservation process



Even if you decide not to preserve view controllers, that does not mean all of those view controllers disappear from the view hierarchy altogether. At launch time, your app might still create the view controllers as part of its default setup. For example, if any view controllers are loaded automatically from your app's storyboard file, they would still appear, albeit in their default configuration, as shown in Figure 5-6.

Figure 5-6 Loading the default set of view controllers



Something else to realize is that even if a view controller is not preserved automatically, you can still encode a reference to that view controller and preserve it manually. In Figure 5-5 (page 81), the three child view controllers of the first navigation controller have restoration identifiers, even though there parent navigation controller does not. If your app delegate (or any preserved object) encodes a reference to those view controllers, their state is preserved. Even though their order in the navigation controller is not saved, you could still use those references to recreate the view controllers and install them in the navigation controller during subsequent launch cycles.

Checklist for Implementing State Preservation and Restoration

Supporting state preservation and restoration requires modifying your app delegate and view controller objects to encode and decode the state information. If your app has any custom views that also have preservable state information, you need to modify those objects too.

When adding state preservation and restoration to your code, use the following list to remind you of the code you need to write.

- (Required) Implement the `application:shouldSaveApplicationState:` and `application:shouldRestoreApplicationState:` methods in your app delegate; see Enabling State Preservation and Restoration in Your App (page 84).
- (Required) Assign restoration identifiers to each view controller you want to preserve by assigning a non empty string to their `restorationIdentifier` property; see Marking Your View Controllers for Preservation (page 85).

  If you want to save the state of specific views too, assign non empty strings to their `restorationIdentifier` properties; see Preserving the State of Your Views (page 88).
- (Required) Show your app's window from the `application:willFinishLaunchingWithOptions:` method of your app delegate. The state restoration machinery needs the window so that it can restore scroll positions and other relevant bits of your app's interface.
- Assign restoration classes to the appropriate view controllers. (If you do not do this, your app delegate is asked to provide the corresponding view controller at restore time.) See Restoring Your View Controllers at Launch Time (page 85).
- (Recommended) Encode and decode the state of your views and view controllers using the `encodeRestorableStateWithCoder:` and `decodeRestorableStateWithCoder:` methods of those objects; see Encoding and Decoding Your View Controller's State (page 87).
- Encode and decode any version information or additional state information for your app using the `application:willEncodeRestorableStateWithCoder:` and `application:didDecodeRestorableStateWithCoder:` methods of your app delegate; see Preserving Your App's High-Level State (page 91).

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- Objects that act as data sources for table views and collection views should implement the `UIDataSourceModelAssociation` protocol. Although not required, this protocol helps preserve the selected and visible items in those types of views. See Implementing Preservation-Friendly Data Sources (page 90).

Enabling State Preservation and Restoration in Your App

State preservation and restoration is not an automatic feature and apps must opt-in to use it. Apps indicate their support for the feature by implementing the following methods in their app delegate:

```
application:shouldSaveApplicationState:
application:shouldRestoreApplicationState:
```

Normally, your implementations of these methods just return `YES` to indicate that state preservation and restoration can occur. However, apps that want to preserve and restore their state conditionally can return `NO` in situations where the operations should not occur. For example, after releasing an update to your app, you might want to return `NO` from your `application:shouldRestoreApplicationState:` method if your app is unable to usefully restore the state from a previous version.

Preserving the State of Your View Controllers

Preserving the state of your app's view controllers should be your main goal. View controllers define the structure of your user interface. They manage the views needed to present that interface and they coordinate the getting and setting of the data that backs those views. To preserve the state of a single view controller, you must do the following:

- (Required) Assign a restoration identifier to the view controller; see Marking Your View Controllers for Preservation (page 85).
- (Required) Provide code to create or locate new view controller objects at launch time; see Restoring Your View Controllers at Launch Time (page 85).
- (Optional) Implement the `encodeRestorableStateWithCoder:` and `decodeRestorableStateWithCoder:` methods to encode and restore any state information that cannot be recreated during a subsequent launch; see Encoding and Decoding Your View Controller's State (page 87).

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Marking Your View Controllers for Preservation

UIKit preserves only those view controllers whose `restorationIdentifier` property contains a valid string object. For view controllers that you know you want to preserve, set the value of this property when you initialize the view controller object. If you load the view controller from a storyboard or nib file, you can set the restoration identifier there.

Choosing an appropriate value for restoration identifiers is important. During the restoration process, your code uses the restoration identifier to determine which view controller to retrieve or create. If every view controller object is based on a different class, you can use the class name for the restoration identifier. However, if your view controller hierarchy contains multiple instances of the same class, you might need to choose different names based on each view usage.

When it asks you to provide a view controller, UIKit provides you with the restoration path of the view controller object. A *restoration path* is the sequence of restoration identifiers starting at the root view controller and walking down the view controller hierarchy to the current object. For example, imagine you have a tab bar controller whose restoration identifier is `TabBarControllerID`, and the first tab contains a navigation controller whose identifier is `NavControllerID` and whose root view controller's identifier is `MyViewController`. The full restoration path for the root view controller would be `TabBarControllerID/NavControllerID/MyViewController`.

The restoration path for every object must be unique. If a view controller has two child view controllers, each child must have a different restoration identifier. However, two view controllers with different parent objects may use the same restoration identifier because the rest of the restoration path provides the needed uniqueness. Some UIKit view controllers, such as navigation controllers, automatically disambiguate their child view controllers, allowing you to use the same restoration identifiers for each child. For more information about the behavior of a given view controller, see the corresponding class reference.

At restore time, you use the provided restoration path to determine which view controller to return to UIKit. For more information on how you use restoration identifiers and restoration paths to restore view controllers, see Restoring Your View Controllers at Launch Time (page 85).

Restoring Your View Controllers at Launch Time

During the restoration process, UIKit asks your app to create (or locate) the view controller objects that comprise your preserved user interface. UIKit adheres to the following process when trying to locate view controllers:

1. **If the view controller had a restoration class, UIKit asks that class to provide the view controller.** UIKit calls the `viewControllerWithRestorationIdentifierPath:coder:` method of the associated restoration class to retrieve the view controller. If that method returns `nil`, it is assumed that the app does not want to recreate the view controller and UIKit stops looking for it.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

2. **If the view controller did not have a restoration class, UIKit asks the app delegate to provide the view controller.** UIKit calls the `application:viewControllerWithRestorationIdentifierPath:coder:` method of your app delegate to look for view controllers without a restoration class. If that method returns `nil`, UIKit tries to find the view controller implicitly.
3. **If a view controller with the correct restoration path already exists, UIKit uses that object.** If your app creates view controllers at launch time (either programmatically or by loading them from a resource file) and assigns restoration identifiers to them, UIKit finds them implicitly through their restoration paths.
4. **If the view controller was originally loaded from a storyboard file, UIKit uses the saved storyboard information to locate and create it.** UIKit saves information about a view controller's storyboard inside the restoration archive. At restore time, it uses that information to locate the same storyboard file and instantiate the corresponding view controller if the view controller was not found by any other means.

It is worth noting that if you specify a restoration class for a view controller, UIKit does not try to find your view controller implicitly. If the `viewControllerWithRestorationIdentifierPath:coder:` method of your restoration class returns `nil`, UIKit stops trying to locate your view controller. This gives you control over whether you really want to create the view controller. If you do not specify a restoration class, UIKit does everything it can to find the view controller for you, creating it as necessary from your app's storyboard files.

If you choose to use a restoration class, the implementation of your `viewControllerWithRestorationIdentifierPath:coder:` method should create a new instance of the class, perform some minimal initialization, and return the resulting object. Listing 5-1 shows an example of how you might use this method to load a view controller from a storyboard. Because the view controller was originally loaded from a storyboard, this method uses the `UIStateRestorationViewControllerStoryboardKey` key to get the storyboard from the archive. Note that this method does not try to configure the view controller's data fields. That step occurs later when the view controller's state is decoded.

Listing 5-1 Creating a new view controller during restoration

```
+ (UIViewController*) viewControllerWithRestorationIdentifierPath:(NSArray
*)identifierComponents
                      coder:(NSCoder *)coder {
   MyViewController* vc;
   UIStoryboard* sb = [coder
decodeObjectForKey:UIStateRestorationViewControllerStoryboardKey];
   if (sb) {
      vc = (PushViewController*)[sb
instantiateViewControllerWithIdentifier:@"MyViewController"];
      vc.restorationIdentifier = [identifierComponents lastObject];
```

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

```
        vc.restorationClass = [MyViewController class];
    }
     return vc;
}
```

Reassigning the restoration identifier and restoration class, as in the preceding example, is a good habit to adopt when creating new view controllers. The simplest way to restore the restoration identifier is to grab the last item in the `identifierComponents` array and assign it to your view controller.

For objects that were already loaded from your app's main storyboard file at launch time, do not create a new instance of each object. Instead, implement the `application:viewControllerWithRestorationIdentifierPath:coder:` method of your app delegate and use it to return the appropriate objects or let UIKit find those objects implicitly.

Encoding and Decoding Your View Controller's State

For each object slated for preservation, UIKit calls the object's `encodeRestorableStateWithCoder:` method to give it a chance to save its state. During the decode process, a matching call to the `decodeRestorableStateWithCoder:` method is made to decode that state and apply it to the object. The implementation of these methods is optional, but recommended, for your view controllers. You can use them to save and restore the following types of information:

- References to any data being displayed (not the data itself)
- For a container view controller, references to its child view controllers
- Information about the current selection
- For view controllers with a user-configurable view, information about the current configuration of that view.

In your encode and decode methods, you can encode any values supported by the coder, including other objects. For all objects except views and view controllers, the object must adopt the `NSCoding` protocol and use the methods of that protocol to write its state. For views and view controllers, the coder does not use the methods of the `NSCoding` protocol to save the object's state. Instead, the coder saves the restoration identifier of the object and adds it to the list of preservable objects, which results in that object's `encodeRestorableStateWithCoder:` method being called.

The `encodeRestorableStateWithCoder:` and `decodeRestorableStateWithCoder:` methods of your view controllers should always call `super` at some point in their implementation. Calling `super` gives the parent class a chance to save and restore any additional information. Listing 5-2 shows a sample implementation of these methods that save a numerical value used to identify the specified view controller.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Listing 5-2 Encoding and decoding a view controller's state.

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder {
    [super encodeRestorableStateWithCoder:coder];

    [coder encodeInt:self.number forKey:MyViewControllerNumber];
}

- (void)decodeRestorableStateWithCoder:(NSCoder *)coder {
    [super decodeRestorableStateWithCoder:coder];

    self.number = [coder decodeIntForKey:MyViewControllerNumber];
}
```

Coder objects are not shared during the encode and decode process. Each object with preservable state receives its own coder that it can use to read or write data. The use of unique coders means that you do not have to worry about key namespace collisions among your own objects. However, you must still avoid using some special key names that UIKit provides. Specifically, each coder contains the `UIApplicationStateRestorationBundleVersionKey` and `UIApplicationStateRestorationUserInterfaceIdiomKey` keys, which provide information about the bundle version and current user interface idiom. Coders associated with view controllers may also contain the `UIStateRestorationViewControllerStoryboardKey` key, which identifies the storyboard from which that view controller originated.

For more information about implementing your encode and decode methods for your view controllers, see *UIViewController Class Reference*.

Preserving the State of Your Views

If a view has state information worth preserving, you can save that state with the rest of your app's view controllers. Because they are usually configured by their owning view controller, most views do not need to save state information. The only time you need to save a view's state is when the view itself can be altered by the user in a way that is independent of its data or the owning view controller. For example, scroll views save the current scroll position, which is information that is not interesting to the view controller but which does affect how the view presents itself.

To designate that a view's state should be saved, you do the following:

- Assign a valid string to the view's `restorationIdentifier` property.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- Use the view from a view controller that also has a valid restoration identifier.
- For table views and collection views, assign a data source that adopts the `UIDataSourceModelAssociation` protocol.

As with view controllers, assigning a restoration identifier to a view tells the system that the view object has state that your app wants to save. The restoration identifier can also be used to locate the view later.

Like view controllers, views define methods for encoding and decoding their custom state. If you create a view with state worth saving, you can use these methods to read and write any relevant data.

UIKit Views with Preservable State

In order to save the state of any view, including both custom and standard system views, you must assign a restoration identifier to the view. Views without a restoration identifier are not added to the list of preservable objects by UIKit.

The following UIKit views have state information that can be preserved:

- `UICollectionView`
- `UIImageView`
- `UIScrollView`
- `UITableView`
- `UITextField`
- `UITextView`
- `UIWebView`

Other frameworks may also have views with preservable state. For information about whether a view saves state information and what state it saves, see the reference for the corresponding class.

Preserving the State of a Custom View

If you are implementing a custom view that has restorable state, implement the `encodeRestorableStateWithCoder:` and `decodeRestorableStateWithCoder:` methods and use them to encode and decode that state. Use those methods to save only the data that cannot be easily reconfigured by other means. For example, use these methods to save data that is modified by user interactions with the view. Do not use these methods to save the data being presented by the view or any data that the owning view controller can configure easily.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Listing 5-3 shows an example of how to preserve and restore the selection for a custom view that contains editable text. In the example, the range is accessible using the `selectionRange` and `setSelectionRange:` methods, which are custom methods the view uses to manage the selection. Encoding the data only requires writing it to the provided coder object. Restoring the data requires reading it and applying it to the view.

Listing 5-3 Preserving the selection of a custom text view

```
// Preserve the text selection
- (void) encodeRestorableStateWithCoder:(NSCoder *)coder {
    [super encodeRestorableStateWithCoder:coder];

    NSRange range = [self selectionRange];
    [coder encodeInt:range.length forKey:kMyTextViewSelectionRangeLength];
    [coder encodeInt:range.location forKey:kMyTextViewSelectionRangeLocation];
}

// Restore the text selection.
- (void) decodeRestorableStateWithCoder:(NSCoder *)coder {
   [super decodeRestorableStateWithCoder:coder];
   if ([coder containsValueForKey:kMyTextViewSelectionRangeLength] &&
            [coder containsValueForKey:kMyTextViewSelectionRangeLocation]) {
      NSRange range;
      range.length = [coder decodeIntForKey:kMyTextViewSelectionRangeLength];
      range.location = [coder decodeIntForKey:kMyTextViewSelectionRangeLocation];
      if (range.length > 0)
         [self setSelectionRange:range];
   }
}
```

Implementing Preservation-Friendly Data Sources

Because the data displayed by a table or collection view can change, both classes save information about the current selection and visible cells only if their data source implements the `UIDataSourceModelAssociation` protocol. This protocol provides a way for a table or collection view to identify the content it contains without relying on the index path of that content. Thus, regardless of where the data source places an item during the next launch cycle, the view still has all the information it needs to locate that item.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

In order to implement the `UIDataSourceModelAssociation` protocol successfully, your data source object must be able to identify items between subsequent launches of the app. This means that any identification scheme you devise must be invariant for a given piece of data. This is essential because the data source must be able to retrieve the same piece of data for the same identifier each time it is requested. Implementing the protocol itself is a matter of mapping from a data item to its unique ID and back again.

Apps that use Core Data can implement the protocol by taking advantage of object identifiers. Each object in a Core Data store has a unique object identifier that can be converted into a URI and used to locate the object later. If your app does not use Core Data, you need to devise your own form of unique identifiers if you want to support state preservation for your views.

> **Note:** Remember that implementing the `UIDataSourceModelAssociation` protocol is only necessary to preserve attributes such as the current selection in a table or collection view. This protocol is not used to preserve the actual data managed by your data source. It is your app's responsibility to ensure that its data is saved at appropriate times.

Preserving Your App's High-Level State

In addition to the data preserved by your app's view controllers and views, UIKit provides hooks for you to save any miscellaneous data needed by your app. Specifically, the `UIApplicationDelegate` protocol includes the following methods for you to override:

- `application:willEncodeRestorableStateWithCoder:`
- `application:didDecodeRestorableStateWithCoder:`

If your app contains state that does not live in a view controller, but that needs to be preserved, you can use the preceding methods to save and restore it. The `application:willEncodeRestorableStateWithCoder:` method is called at the very beginning of the preservation process so that you can write out any high-level app state, such as the current version of your user interface. The `application:didDecodeRestorableStateWithCoder:` method is called at the end of the restoration state so that you can decode any data and perform any final cleanup that your app requires.

Tips for Saving and Restoring State Information

As you add support for state preservation and restoration to your app, consider the following guidelines:

- **Encode version information along with the rest of your app's state.** During the preservation process, it is recommended that you encode a version string or number that identifies the current revision of your app's user interface. You can encode this state in the

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

`application:willEncodeRestorableStateWithCoder:` method of your app delegate. When your app delegate's `application:shouldRestoreApplicationState:` method is called, you can retrieve this information from the provided coder and use it to determine if state preservation is possible.

- **Do not include objects from your data model in your app's state.** Apps should continue to save their data separately in iCloud or to local files on disk. Never use the state restoration mechanism to save that data. Preserved interface data may be deleted if problems occur during a restore operation. Therefore, any preservation-related data you write to disk should be considered purgeable.
- **The state preservation system expects you to use view controllers in the ways they were designed to be used.** The view controller hierarchy is created through a combination of view controller containment and by presenting one view controller from another. If your app displays the view of a view controller by another means—for example, by adding it to another view without creating a containment relationship between the corresponding view controllers—the preservation system will not be able to find your view controller to preserve it.
- **Remember that you might not want to preserve all view controllers.** In some cases, it might not make sense to preserve a view controller. For example, if the user left your app while it was displaying a view controller to change the user's password, you might want to cancel the operation and restore the app to the previous screen. In such a case, you would not preserve the view controller that asks for the new password information.
- **Avoid swapping view controller classes during the restoration process.** The state preservation system encodes the class of the view controllers it preserves. During restoration, if your app returns an object whose class does not match (or is not a subclass of) the original object, the system does not ask the view controller to decode any state information. Thus, swapping out the old view controller for a completely different one does not restore the full state of the object.
- **The system automatically deletes an app's preserved state when the user force quits the app.** Deleting the preserved state information when the app is killed is a safety precaution. (As a safety precaution, the system also deletes preserved state if the app crashes twice during launch.) If you want to test your app's ability to restore its state, you should not use the multitasking bar to kill the app during debugging. Instead, use Xcode to kill the app or kill the app programmatically by installing a temporary command or gesture to call `exit` on demand.

Tips for Developing a VoIP App

A *Voice over Internet Protocol (VoIP)* app allows the user to make phone calls using an Internet connection instead of the device's cellular service. Such an app needs to maintain a persistent network connection to its associated service so that it can receive incoming calls and other relevant data. Rather than keep VoIP apps

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

awake all the time, the system allows them to be suspended and provides facilities for monitoring their sockets for them. When incoming traffic is detected, the system wakes up the VoIP app and returns control of its sockets to it.

There are several requirements for implementing a VoIP app:

1. Enable the Voice over IP background mode for your app. (Because VoIP apps involve audio content, it is recommended that you also enable the Audio and AirPlay background mode.) You enable background modes in the Capabilities tab of your Xcode project.
2. Configure one of the app's sockets for VoIP usage.
3. Before moving to the background, call the `setKeepAliveTimeout:handler:` method to install a handler to be executed periodically. Your app can use this handler to maintain its service connection.
4. Configure your audio session to handle transitions to and from active use.
5. To ensure a better user experience on iPhone, use the Core Telephony framework to adjust your behavior in relation to cell-based phone calls; see *Core Telephony Framework Reference*.
6. To ensure good performance for your VoIP app, use the System Configuration framework to detect network changes and allow your app to sleep as much as possible.

Enabling the VoIP background mode lets the system know that it should allow the app to run in the background as needed to manage its network sockets. This key also permits your app to play background audio (although enabling the Audio and AirPlay mode is still encouraged). An app that supports this mode is also relaunched in the background immediately after system boot to ensure that the VoIP services are always available.

Configuring Sockets for VoIP Usage

In order for your app to maintain a persistent connection while it is in the background, you must tag your app's main communication socket specifically for VoIP usage. Tagging this socket tells the system that it should take over management of the socket when your app is suspended. The handoff itself is totally transparent to your app. And when new data arrives on the socket, the system wakes up the app and returns control of the socket so that the app can process the incoming data.

You need to tag only the socket you use for communicating with your VoIP service. This is the socket you use to receive incoming calls or other data relevant to maintaining your VoIP service connection. Upon receipt of incoming data, the handler for this socket needs to decide what to do. For an incoming call, you likely want to post a local notification to alert the user to the call. For other noncritical data, though, you might just process the data quietly and allow the system to put your app back into the suspended state.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

In iOS, most sockets are managed using streams or other high-level constructs. To configure a socket for VoIP usage, the only thing you have to do beyond the normal configuration is add a special key that tags the interface as being associated with a VoIP service. Table 5-1 lists the stream interfaces and the configuration for each.

Table 5-1 Configuring stream interfaces for VoIP usage

Interface	Configuration
`NSInputStream` and `NSOutputStream`	For Cocoa streams, use the `setProperty:forKey:` method to add the `NSStreamNetworkServiceType` property to the stream. The value of this property should be set to `NSStreamNetworkServiceTypeVoIP`.
`NSURLRequest`	When using the URL loading system, use the `setNetworkServiceType:` method of your `NSMutableURLRequest` object to set the network service type of the request. The service type should be set to `NSURLNetworkServiceTypeVoIP`.
`CFReadStreamRef` and `CFWriteStreamRef`	For Core Foundation streams, use the `CFReadStreamSetProperty` or `CFWriteStreamSetProperty` function to add the `kCFStreamNetworkServiceType` property to the stream. The value for this property should be set to `kCFStreamNetworkServiceTypeVoIP`.

Note: When configuring your sockets, you need to configure only your main signaling channel with the appropriate service type key. You do not need to include this key when configuring your voice channels.

Because VoIP apps need to stay running in order to receive incoming calls, the system automatically relaunches the app if it exits with a nonzero exit code. (This type of exit could happen when there is memory pressure and your app is terminated as a result.) However, terminating the app also releases all of its sockets, including the one used to maintain the VoIP service connection. Therefore, when the app is launched, it always needs to create its sockets from scratch.

For more information about configuring Cocoa stream objects, see *Stream Programming Guide* . For information about using URL requests, see *URL Loading System Programming Guide* . And for information about configuring streams using the CFNetwork interfaces, see *CFNetwork Programming Guide* .

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Installing a Keep-Alive Handler

To prevent the loss of its connection, a VoIP app typically needs to wake up periodically and check in with its server. To facilitate this behavior, iOS lets you install a special handler using the `setKeepAliveTimeout:handler:` method of `UIApplication`. You typically install this handler in the `applicationDidEnterBackground:` method of your app delegate. Once installed, the system calls your handler at least once before the timeout interval expires, waking up your app as needed to do so.

Your keep-alive handler executes in the background and should return as quickly as possible. Handlers are given a maximum of 10 seconds to perform any needed tasks and return. If a handler has not returned after 10 seconds, or has not requested extra execution time before that interval expires, the system suspends the app.

When installing your handler, specify the largest timeout value that is practical for your app's needs. The minimum allowable interval for running your handler is 600 seconds, and attempting to install a handler with a smaller timeout value will fail. Although the system promises to call your handler block before the timeout value expires, it does not guarantee the exact call time. To improve battery life, the system typically groups the execution of your handler with other periodic system tasks, thereby processing all tasks in one quick burst. As a result, your handler code must be prepared to run earlier than the actual timeout period you specified.

Configuring Your App's Audio Session

As with any background audio app, the audio session for a VoIP app must be configured properly to ensure the app works smoothly with other audio-based apps. Because audio playback and recording for a VoIP app are not used all the time, it is especially important that you create and configure your app's audio session object only when it is needed. For example, you would create the audio session to notify the user of an incoming call or while the user was actually on a call. As soon as the call ends, you would then remove strong references to the audio session and give other audio apps the opportunity to play their audio.

For information about how to configure and manage an audio session for a VoIP app, see *Audio Session Programming Guide*.

Using the Reachability Interfaces to Improve the User Experience

Because VoIP apps rely heavily on the network, they should use the reachability interfaces of the System Configuration framework to track network availability and adjust their behavior accordingly. The reachability interfaces allow an app to be notified whenever network conditions change. For example, a VoIP app could close its network connections when the network becomes unavailable and recreate them when it becomes available again. The app could also use those kinds of changes to keep the user apprised about the state of the VoIP connection.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

To use the reachability interfaces, you must register a callback function with the framework and use it to track changes. To register a callback function:

1. Create a `SCNetworkReachabilityRef` structure for your target remote host.
2. Assign a callback function to your structure (using the `SCNetworkReachabilitySetCallback` function) that processes changes in your target's reachability status.
3. Add that target to an active run loop of your app (such as the main run loop) using the `SCNetworkReachabilityScheduleWithRunLoop` function.

Adjusting your app's behavior based on the availability of the network can also help improve the battery life of the underlying device. Letting the system track the network changes means that your app can let itself go to sleep more often.

For more information about the reachability interfaces, see *System Configuration Framework Reference* .

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Inter-App Communication

SwiftObjective-C

Apps communicate only indirectly with other apps on a device. You can use AirDrop to share files and data with other apps. You can also define a custom URL scheme so that apps can send information to your app using URLs.

> **Note:** You can also send files between apps using a `UIDocumentInteractionController` object or a document picker. For information about adding support for a document interaction controller, see *Document Interaction Programming Topics for iOS*. For information about using a document picker to open files, see *Document Picker Programming Guide*.

Supporting AirDrop

AirDrop lets you share photos, documents, URLs, and other types of data with nearby devices. AirDrop takes advantage of peer-to-peer networking to find nearby devices and connect to them.

Sending Files and Data to Another App

To send files and data using AirDrop, use a `UIActivityViewController` object to display an activity sheet from your user interface using. When creating this view controller, you specify the data objects that you want to share. The view controller displays only those activities that support the specified data. For AirDrop, you can specify images, strings, URLs, and several other types of data. You can also pass custom objects that adopt the `UIActivityItemSource` protocol.

To display an activity view controller, you can use code similar to that shown in Listing 6-1. The activity view controller automatically uses the type of the specified object to determine what activities to display in the activity sheet. You do not have to specify the AirDrop activity explicitly. However, you can prevent the sheet from displaying specific types using the view controller's `excludedActivityTypes` property. When displaying an activity view controller on iPad, you must use a popover.

Listing 6-1 Displaying an activity sheet on iPhone

```
- (void)displayActivityControllerWithDataObject:(id)obj {
```

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

```
   UIActivityViewController* vc = [[UIActivityViewController alloc]
                                   initWithActivityItems:@[obj]
applicationActivities:nil];
    [self presentViewController:vc animated:YES completion:nil];
}
```

For more information about using the activity view controller, see *UIActivityViewController Class Reference*. For a complete list of activities and the data types they support, see *UIActivity Class Reference*.

Receiving Files and Data Sent to Your App

To receive files sent to your app using AirDrop, do the following:

- In Xcode, declare support for the document types your app is capable of opening.
- In your app delegate, implement the `application:openURL:sourceApplication:annotation:` method. Use that method to receive the data that was sent by the other app.
- Be prepared to look for files in your app's `Documents/Inbox` directory and move them out of that directory as needed.

The Info tab of your Xcode project contains a Document Types section for specifying the document types your app supports. At a minimum, you must specify a name for your document type and one or more UTIs that represent the data type. For example, to declare support for PNG files, you would include `public.png` as the UTI string. iOS uses the specified UTIs to determine if your app is eligible to open a given document.

After transferring an eligible document to your app's container, iOS launches your app (if needed) and calls the `application:openURL:sourceApplication:annotation:` method of its app delegate. If your app is in the foreground, you should use this method to open the file and display it to the user. If your app is in the background, you might decide only to note that the file is there so that you can open it later. Because files transferred via AirDrop are encrypted using data protection, you cannot open files unless the device is currently unlocked.

Files transferred to your app using AirDrop are placed in your app's `Documents/Inbox` directory. Your app has permission to read and delete files in this directory but it does not have permission to write to files. If you plan to modify the file, you must move it out of the `Inbox` directory before doing so. It is recommended that you delete files from the `Inbox` directory when you no longer need them.

For more information about supporting document types in your app, see *Document-Based App Programming Guide for iOS*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Using URL Schemes to Communicate with Apps

A URL scheme lets you communicate with other apps through a protocol that you define. To communicate with an app that implements such a scheme, you must create an appropriately formatted URL and ask the system to open it. To implement support for a custom scheme, you must declare support for the scheme and handle incoming URLs that use the scheme.

> **Note:** Apple provides built-in support for the `http`, `mailto`, `tel`, and `sms` URL schemes among others. It also supports `http`–based URLs targeted at the Maps, YouTube, and iPod apps. The handlers for these schemes are fixed and cannot be changed. If your URL type includes a scheme that is identical to one defined by Apple, the Apple-provided app is launched instead of your app. For information about the schemes supported by apple, see *Apple URL Scheme Reference*.

Sending a URL to Another App

When you want to send data to an app that implements a custom URL scheme, create an appropriately formatted URL and call the `openURL:` method of the app object. The `openURL:` method launches the app with the registered scheme and passes your URL to it. At that point, control passes to the new app.

The following code fragment illustrates how one app can request the services of another app ("todolist" in this example is a hypothetical custom scheme registered by an app):

```
NSURL *myURL = [NSURL
URLWithString:@"todolist://www.acme.com?Quarterly%20Report#200806231300"];
[[UIApplication sharedApplication] openURL:myURL];
```

If your app defines a custom URL scheme, it should implement a handler for that scheme as described in Implementing Custom URL Schemes (page 99). For more information about the system-supported URL schemes, including information about how to format the URLs, see *Apple URL Scheme Reference*.

Implementing Custom URL Schemes

If your app can receive specially formatted URLs, you should register the corresponding URL schemes with the system. Apps often use custom URL schemes to vend services to other apps. For example, the Maps app supports URLs for displaying specific map locations.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Registering Custom URL Schemes

To register a URL type for your app, include the `CFBundleURLTypes` key in your app's `Info.plist` file. The `CFBundleURLTypes` key contains an array of dictionaries, each of which defines a URL scheme the app supports. Table 6-1 describes the keys and values to include in each dictionary.

Table 6-1 Keys and values of the `CFBundleURLTypes` property

Key	Value
`CFBundleURLName`	A string containing the abstract name of the URL scheme. To ensure uniqueness, it is recommended that you specify a reverse-DNS style of identifier, for example, `com.acme.myscheme`. The string you specify is also used as a key in your app's `InfoPlist.strings` file. The value of the key is the human-readable scheme name.
`CFBundleURLSchemes`	An array of strings containing the URL scheme names—for example, `http`, `mailto`, `tel`, and `sms`.

> **Note:** If more than one third-party app registers to handle the same URL scheme, there is currently no process for determining which app will be given that scheme.

Handling URL Requests

An app that has its own custom URL scheme must be able to handle URLs passed to it. All URLs are passed to your app delegate, either at launch time or while your app is running or in the background. To handle incoming URLs, your delegate should implement the following methods:

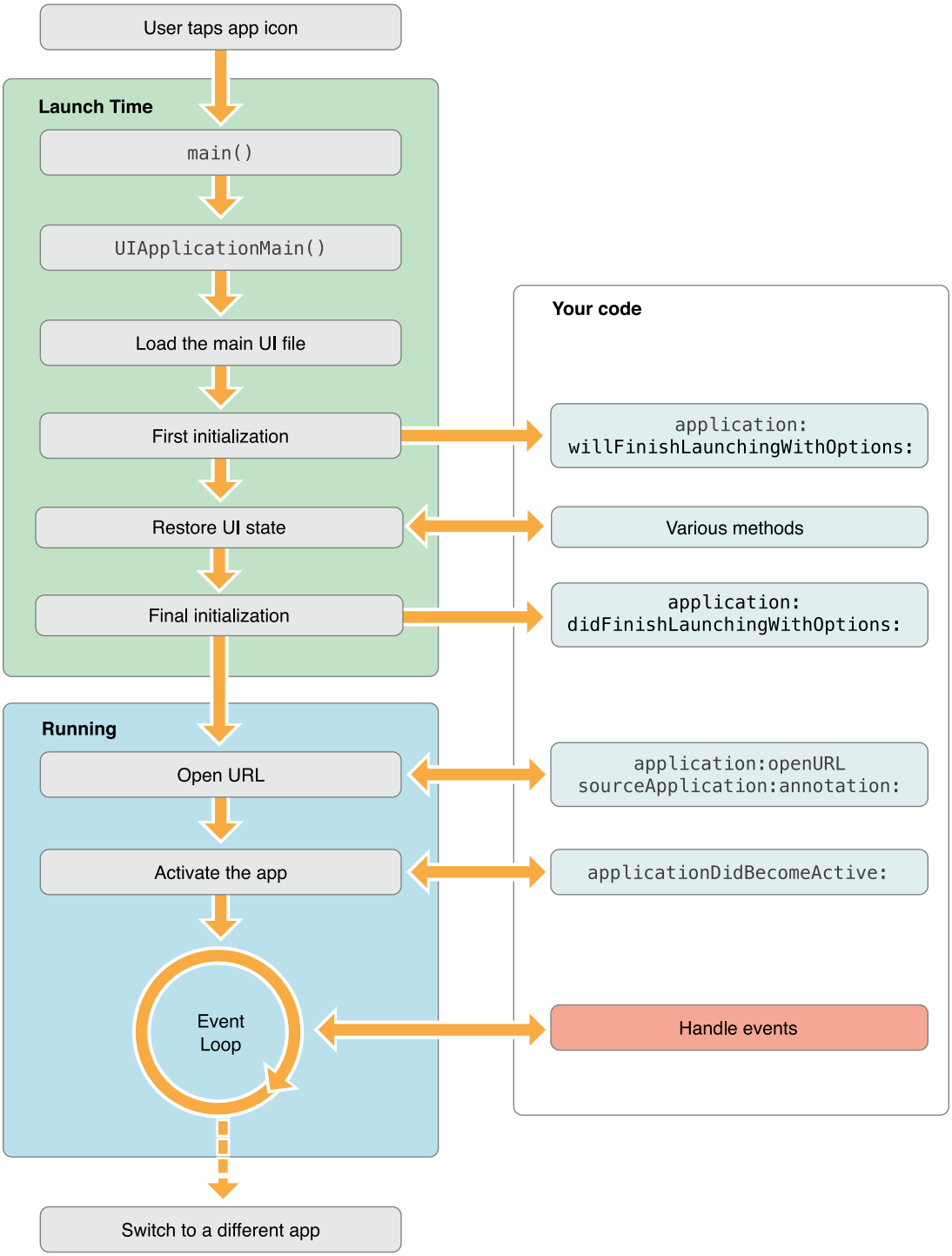
- Use the `application:willFinishLaunchingWithOptions:` and `application:didFinishLaunchingWithOptions:` methods to retrieve information about the URL and decide whether you want to open it. If either method returns `NO`, your app's URL handling code is not called.
- Use the `application:openURL:sourceApplication:annotation:` method to open the file.

If your app is not running when a URL request arrives, it is launched and moved to the foreground so that it can open the URL. The implementation of your `application:willFinishLaunchingWithOptions:` or `application:didFinishLaunchingWithOptions:` method should retrieve the URL from its options dictionary and determine whether the app can open it. If it can, return `YES` and let your `application:openURL:sourceApplication:annotation:` (or `application:handleOpenURL:`)

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

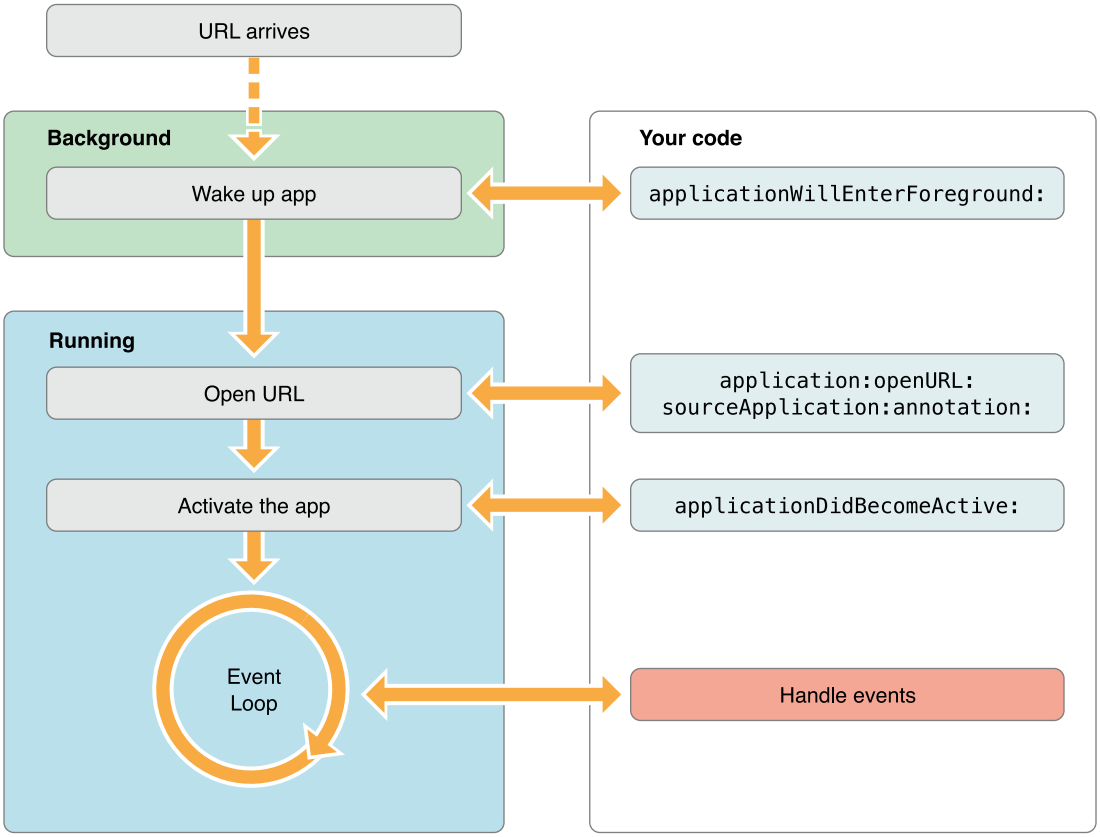
method handle the actual opening of the URL. (If you implement both methods, both must return YES before the URL can be opened.) Figure 6-1 shows the modified launch sequence for an app that is asked to open a URL.

Figure 6-1 Launching an app to open a URL



If your app is running but is in the background or suspended when a URL request arrives, it is moved to the foreground to open the URL. Shortly thereafter, the system calls the delegate's `application:openURL:sourceApplication:annotation:` to check the URL and open it. Figure 6-2 shows the modified process for moving an app to the foreground to open a URL.

Figure 6-2 Waking a background app to open a URL



Note: Apps that support custom URL schemes can specify different launch images to be displayed when launching the app to handle a URL. For more information about how to specify these launch images, see Displaying a Custom Launch Image When a URL is Opened (page 104).

All URLs are passed to your app in an `NSURL` object. It is up to you to define the format of the URL, but the `NSURL` class conforms to the RFC 1808 specification and therefore supports most URL formatting conventions. Specifically, the class includes methods that return the various parts of a URL as defined by RFC 1808, including the user, password, query, fragment, and parameter strings. The "protocol" for your custom scheme can use these URL parts for conveying various kinds of information.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

In the implementation of `application:openURL:sourceApplication:annotation:` shown in Listing 6-2, the passed-in URL object conveys app-specific information in its query and fragment parts. The delegate extracts this information—in this case, the name of a to-do task and the date the task is due—and with it creates a model object of the app. This example assumes that the user is using a Gregorian calendar. If your app supports non-Gregorian calendars, you need to design your URL scheme accordingly and be prepared to handle those other calendar types in your code.

Listing 6-2 Handling a URL request based on a custom scheme

```
- (BOOL)application:(UIApplication *)application openURL:(NSURL *)url
        sourceApplication:(NSString *)sourceApplication annotation:(id)annotation
 {
    if ([[url scheme] isEqualToString:@"todolist"]) {
        ToDoItem *item = [[ToDoItem alloc] init];
        NSString *taskName = [url query];
        if (!taskName || ![self isValidTaskString:taskName]) { // must have a task
 name
            return NO;
        }
        taskName = [taskName
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];

        item.toDoTask = taskName;
        NSString *dateString = [url fragment];
        if (!dateString || [dateString isEqualToString:@"today"]) {
            item.dateDue = [NSDate date];
        } else {
            if (![self isValidDateString:dateString]) {
                return NO;
            }
            // format: yyyymmddhhmm (24-hour clock)
            NSString *curStr = [dateString substringWithRange:NSMakeRange(0, 4)];
            NSInteger yeardigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(4, 2)];
            NSInteger monthdigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(6, 2)];
            NSInteger daydigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(8, 2)];
```

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

```
            NSInteger hourdigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(10, 2)];
            NSInteger minutedigit = [curStr integerValue];

            NSDateComponents *dateComps = [[NSDateComponents alloc] init];
            [dateComps setYear:yeardigit];
            [dateComps setMonth:monthdigit];
            [dateComps setDay:daydigit];
            [dateComps setHour:hourdigit];
            [dateComps setMinute:minutedigit];
            NSCalendar *calendar = [s[NSCalendar alloc]
initWithCalendarIdentifier:NSGregorianCalendar];
            NSDate *itemDate = [calendar dateFromComponents:dateComps];
            if (!itemDate) {
                return NO;
            }
            item.dateDue = itemDate;
        }

        [(NSMutableArray *)self.list addObject:item];
        return YES;
    }
    return NO;
}
```

Be sure to validate the input you get from URLs passed to your app; see Validating Input and Interprocess Communication in *Secure Coding Guide* to find out how to avoid problems related to URL handling. To learn about URL schemes defined by Apple, see *Apple URL Scheme Reference*.

Displaying a Custom Launch Image When a URL is Opened

Apps that support custom URL schemes can provide a custom launch image for each scheme. When the system launches your app to handle a URL and no relevant snapshot is available, it displays the launch image you specify. To specify a launch image, provide a PNG image whose name uses the following naming conventions:

<basename>`-`*<url_scheme>* *<other_modifiers>*`.png`

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

In this naming convention, basename represents the base image name specified by the `UILaunchImageFile` key in your app's `Info.plist` file. If you do not specify a custom base name, use the string `Default`. The *<url_scheme>* portion of the name is your URL scheme name. To specify a generic launch image for the `myapp` URL scheme, you would include an image file with the name `Default-myapp@2x.png` in the app's bundle. (The @2x modifier signifies that the image is intended for Retina displays. If your app also supports standard resolution displays, you would also provide a `Default-myapp.png` image.)

For information about the other modifiers you can include in launch image names, see the description of the `UILaunchImageFile` name key in *Information Property List Key Reference*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Performance Tips

Objective-CSwift

At each step in the development of your app, consider the implications of your design choices on the overall performance of your app. Power usage and memory consumption are extremely important considerations for iOS apps, and there are many other considerations as well. The following sections describe the factors you should consider throughout the development process.

Reduce Your App's Power Consumption

Power consumption on mobile devices is always an issue. The power management system in iOS conserves power by shutting down any hardware features that are not currently being used. You can help improve battery life by optimizing your use of the following features:

- The CPU
- Wi-Fi, Bluetooth, and baseband (EDGE, 3G) radios
- The Core Location framework
- The accelerometers
- The disk

The goal of your optimizations should be to do the most work you can in the most efficient way possible. You should always optimize your app's algorithms using Instruments. But even the most optimized algorithm can still have a negative impact on a device's battery life. You should therefore consider the following guidelines when writing your code:

- Avoid doing work that requires polling. Polling prevents the CPU from going to sleep. Instead of polling, use the `NSRunLoop` or `NSTimer` classes to schedule work as needed.
- Leave the `idleTimerDisabled` property of the shared `UIApplication` object set to `NO` whenever possible. The idle timer turns off the device's screen after a specified period of inactivity. If your app does not need the screen to stay on, let the system turn it off. If your app experiences side effects as a result of the screen being turned off, you should modify your code to eliminate the side effects rather than disable the idle timer unnecessarily.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- Coalesce work whenever possible to maximize idle time. It generally takes less power to perform a set of calculations all at once than it does to perform them in small chunks over an extended period of time. Doing small bits of work periodically requires waking up the CPU more often and getting it into a state where it can perform your tasks.
- Avoid accessing the disk too frequently. For example, if your app saves state information to the disk, do so only when that state information changes, and coalesce changes whenever possible to avoid writing small changes at frequent intervals.
- Do not draw to the screen faster than is needed. Drawing is an expensive operation when it comes to power. Do not rely on the hardware to throttle your frame rates. Draw only as many frames as your app actually needs.
- If you use the `UIAccelerometer` class to receive regular accelerometer events, disable the delivery of those events when you do not need them. Similarly, set the frequency of event delivery to the smallest value that is suitable for your needs. For more information, see *Event Handling Guide for iOS* .

The more data you transmit to the network, the more power must be used to run the radios. In fact, accessing the network is the most power-intensive operation you can perform. You can minimize that time by following these guidelines:

- Connect to external network servers only when needed, and do not poll those servers.
- When you must connect to the network, transmit the smallest amount of data needed to do the job. Use compact data formats, and do not include excess content that simply is ignored.
- Transmit data in bursts rather than spreading out transmission packets over time. The system turns off the Wi-Fi and cell radios when it detects a lack of activity. When it transmits data over a longer period of time, your app uses much more power than when it transmits the same amount of data in a shorter amount of time.

  When using the `NSURLSession` class to enqueue multiple upload or download tasks, enqueue those items together rather than waiting for one to finish before starting the next one. The system manages automatically executes queued tasks when it is most efficient to do so.
- Connect to the network using the Wi-Fi radios whenever possible. Wi-Fi uses less power and is preferred over cellular radios.
- If you use the Core Location framework to gather location data, disable location updates as soon as you can and set the distance filter and accuracy levels to appropriate values. Core Location uses the available GPS, cell, and Wi-Fi networks to determine the user's location. Although Core Location works hard to minimize the use of these radios, setting the accuracy and filter values gives Core Location the option to turn off hardware altogether in situations where it is not needed. For more information, see *Location and Maps Programming Guide* .

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

The Instruments app includes several instruments for gathering power-related information. You can use these instruments to gather general information about power consumption and to gather specific measurements for hardware such as the Wi-Fi and Bluetooth radios, GPS receiver, display, and CPU. You can also enable Energy Diagnostics Logging on a device to gather information. For information about using Instruments to gather power-related data, see *Instruments User Guide*. For information about how to enable Energy Diagnostics Logging on a device, see *Instruments Help*.

Use Memory Efficiently

Apps are encouraged to use as little memory as possible so that the system may keep more apps in memory or dedicate more memory to foreground apps that truly need it. There is a direct correlation between the amount of free memory available to the system and the relative performance of your app. Less free memory means that the system is more likely to have trouble fulfilling future memory requests.

To ensure there is always enough free memory available, you should minimize your app's memory usage and be responsive when the system asks you to free up memory.

Observe Low-Memory Warnings

When the system dispatches a low-memory warning to your app, *respond immediately*. Low-memory warnings are your opportunity to remove references to objects that you do not need. Responding to these warnings is crucial because apps that fail to do so are more likely to be terminated. The system delivers memory warnings to your app using the following APIs:

- The `applicationDidReceiveMemoryWarning:` method of your app delegate.
- The `didReceiveMemoryWarning` method of your `UIViewController` classes.
- The `UIApplicationDidReceiveMemoryWarningNotification`notification.
- Dispatch sources of type `DISPATCH_SOURCE_TYPE_MEMORYPRESSURE`. This technique is the only one that you can use to distinguish the severity of the memory pressure.

Upon receiving any of these warnings, your handler method should respond by immediately freeing up any unneeded memory. Use the warnings to clear out caches and release images. If you have large data structures that are not being used, write those structures to disk and release the in-memory copies of the data.

If your data model includes known purgeable resources, you can have a corresponding manager object register for the `UIApplicationDidReceiveMemoryWarningNotification` notification and remove strong references to its purgeable resources directly. Handling this notification directly avoids the need to route all memory warning calls through the app delegate.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Note: You can test your app's behavior under low-memory conditions using the Simulate Memory Warning command in iOS Simulator.

Reduce Your App's Memory Footprint

Starting off with a low footprint gives you more room for expanding your app later. Table 7-1 lists some tips on how to reduce your app's overall memory footprint.

Table 7-1 Tips for reducing your app's memory footprint

Tip	Actions to take
Eliminate memory leaks.	Because memory is a critical resource in iOS, your app should never have memory leaks. Use the Instruments app to track down leaks in your code, both in Simulator and on actual devices. For more information on using Instruments, see *Instruments User Guide*.
Make resource files as small as possible.	Files reside on disk but must be loaded into memory before they can be used. Compress all image files to make them as small as possible. (To compress PNG images—the preferred image format for iOS apps—use the `pngcrush` tool.) You can make property list files smaller by writing them out in a binary format using the `NSPropertyListSerialization` class.
Use Core Data or SQLite for large data sets.	If your app manipulates large amounts of structured data, store it in a Core Data persistent store or in a SQLite database instead of in a flat file. Both Core Data and SQLite provides efficient ways to manage large data sets without requiring the entire set to be in memory all at once.
Load resources lazily.	You should never load a resource file until it is actually needed. Prefetching resource files may seem like a way to save time, but this practice actually slows down your app right away. In addition, if you end up not using the resource, loading it wastes memory for no good purpose.

Allocate Memory Wisely

Table 7-2 lists tips for improving memory usage in your app.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Table 7-2 Tips for allocating memory

Tip	Actions to take
Impose size limits on resources.	Avoid loading a large resource file when a smaller one will do. Instead of using a high-resolution image, use one that is appropriately sized for iOS-based devices. If you must use large resource files, find ways to load only the portion of the file that you need at any given time. For example, rather than load the entire file into memory, use the `mmap` and `munmap` functions to map portions of the file into and out of memory. For more information about mapping files into memory, see *File-System Performance Guidelines*.
Avoid unbounded problem sets.	Unbounded problem sets might require an arbitrarily large amount of data to compute. If the set requires more memory than is available, your app may be unable to complete the calculations. Your apps should avoid such sets whenever possible and work on problems with known memory limits.

For detailed information about ARC and memory management, see *Transitioning to ARC Release Notes*.

Tune Your Networking Code

The networking stack in iOS includes several interfaces for communicating over the radio hardware of iOS devices. The main programming interface is the CFNetwork framework, which builds on top of BSD sockets and opaque types in the Core Foundation framework to communicate with network entities. You can also use the `NSStream` classes in the Foundation framework and the low-level BSD sockets found in the Core OS layer of the system.

For information about how to use the CFNetwork framework for network communication, see *CFNetwork Programming Guide* and *CFNetwork Framework Reference*. For information about using the `NSStream` class, see *Foundation Framework Reference*.

Tips for Efficient Networking

Implementing code to receive or transmit data across the network is one of the most power-intensive operations on a device. Minimizing the amount of time spent transmitting or receiving data helps improve battery life. To that end, you should consider the following tips when writing your network-related code:

- For protocols you control, define your data formats to be as compact as possible.
- Avoid using chatty protocols.
- Transmit data packets in bursts whenever you can.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Cellular and Wi-Fi radios are designed to power down when there is no activity. Depending on the radio, though, doing so can take several seconds. If your app transmits small bursts of data every few seconds, the radios may stay powered up and continue to consume power, even when they are not actually doing anything. Rather than transmit small amounts of data more often, it is better to transmit a larger amount of data once or at relatively large intervals.

When communicating over the network, packets can be lost at any time. Therefore, when writing your networking code, you should be sure to make it as robust as possible when it comes to failure handling. It is perfectly reasonable to implement handlers that respond to changes in network conditions, but do not be surprised if those handlers are not called consistently. For example, the Bonjour networking callbacks may not always be called immediately in response to the disappearance of a network service. The Bonjour system service immediately invokes browsing callbacks when it receives a notification that a service is going away, but network services can disappear without notification. This situation might occur if the device providing the network service unexpectedly loses network connectivity or the notification is lost in transit.

Using Wi-Fi

If your app accesses the network using the Wi-Fi radios, you must notify the system of that fact by including the `UIRequiresPersistentWiFi` key in the app's `Info.plist` file. The inclusion of this key lets the system know that it should display the network selection dialog if it detects any active Wi-Fi hot spots. It also lets the system know that it should not attempt to shut down the Wi-Fi hardware while your app is running.

To prevent the Wi-Fi hardware from using too much power, iOS has a built-in timer that turns off the hardware completely after 30 minutes if no running app has requested its use through the `UIRequiresPersistentWiFi` key. If the user launches an app that includes the key, iOS effectively disables the timer for the duration of the app's life cycle. As soon as that app quits or is suspended, however, the system reenables the timer.

> **Note:** Note that even when `UIRequiresPersistentWiFi` has a value of `true`, it has no effect when the device is idle (that is, screen-locked). The app is considered inactive, and although it may function on some levels, it has no Wi-Fi connection.

For more information on the `UIRequiresPersistentWiFi` key and the keys of the `Info.plist` file, see The Information Property List File (page 15).

The Airplane Mode Alert

If your app launches while the device is in airplane mode, the system may display an alert to notify the user of that fact. The system displays this alert only when all of the following conditions are met:

- Your app's information property list (`Info.plist`) file contains the `UIRequiresPersistentWiFi` key and the value of that key is set to `true`.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- Your app launches while the device is currently in airplane mode.
- Wi-Fi on the device has not been manually reenabled after the switch to airplane mode.

Improve Your File Management

Minimize the amount of data you write to the disk. File operations are relatively slow and involve writing to the flash drive, which has a limited lifespan. Some specific tips to help you minimize file-related operations include:

- Write only the portions of the file that changed, and aggregate changes when you can. Avoid writing out the entire file just to change a few bytes.
- When defining your file format, group frequently modified content together to minimize the overall number of blocks that need to be written to disk each time.
- If your data consists of structured content that is randomly accessed, store it in a Core Data persistent store or a SQLite database, especially if the amount of data you are manipulating could grow to more than a few megabytes.

Avoid writing cache files to disk. The only exception to this rule is when your app quits and you need to write state information that can be used to put your app back into the same state when it is next launched.

Make App Backups More Efficient

Backups occur wirelessly via iCloud or when the user syncs the device with iTunes. During backups, files are transferred from the device to the user's computer or iCloud account. The location of files in your app sandbox determines whether or not those files are backed up and restored. If your application creates many large files that change regularly and puts them in a location that is backed up, backups could be slowed down as a result. As you write your file-management code, you need to be mindful of this fact.

App Backup Best Practices

You do not have to prepare your app in any way for backup and restore operations. Devices with an active iCloud account have their app data backed up to iCloud at appropriate times. For devices that are plugged into a computer, iTunes performs an incremental backup of the app's data files. However, iCloud and iTunes do not back up the contents of the following directories:

- *<Application_Home>*`/`*AppName*`.app`
- *<Application_Data>*`/Library/Caches`

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

- *<Application_Data>*`/tmp`

To prevent the syncing process from taking a long time, be selective about where you place files inside your app's home directory. Apps that store large files can slow down the process of backing up to iTunes or iCloud. These apps can also consume a large amount of a user's available storage, which may encourage the user to delete the app or disable backup of that app's data to iCloud. With this in mind, you should store app data according to the following guidelines:

- Critical data should be stored in the *<Application_Data>*`/Documents` directory. Critical data is any data that cannot be recreated by your app, such as user documents and other user-generated content.
- Support files include files your application downloads or generates and that your application can recreate as needed. The location for storing your application's support files depends on the current iOS version.
    - In iOS 5.1 and later, store support files in the *<Application_Data>*`/Library/Application Support` directory and add the `NSURLIsExcludedFromBackupKey` attribute to the corresponding `NSURL` object using the `setResourceValue:forKey:error:` method. (If you are using Core Foundation, add the `kCFURLIsExcludedFromBackupKey` key to your `CFURLRef` object using the `CFURLSetResourcePropertyForKey` function.) Applying this attribute prevents the files from being backed up to iTunes or iCloud. If you have a large number of support files, you may store them in a custom subdirectory and apply the extended attribute to just the directory.
    - In iOS 5.0 and earlier, store support files in the *<Application_Data>*`/Library/Caches` directory to prevent them from being backed up. If you are targeting iOS 5.0.1, see *How do I prevent files from being backed up to iCloud and iTunes?* for information about how to exclude files from backups.
- Cached data should be stored in the *<Application_Data>*`/Library/Caches` directory. Examples of files you should put in the `Caches` directory include (but are not limited to) database cache files and downloadable content, such as that used by magazine, newspaper, and map apps. Your app should be able to gracefully handle situations where cached data is deleted by the system to free up disk space.
- Temporary data should be stored in the *<Application_Data>*`/tmp` directory. Temporary data comprises any data that you do not need to persist for an extended period of time. Remember to delete those files when you are done with them so that they do not continue to consume space on the user's device.

Although iTunes backs up the app bundle itself, it does not do this during every sync operation. Apps purchased directly from a device are backed up when that device is next synced with iTunes. Apps are not backed up during subsequent sync operations, though, unless the app bundle itself has changed (because the app was updated, for example).

For additional guidance about how you should use the directories in your app, see *File System Programming Guide*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Files Saved During App Updates

When a user downloads an app update, iTunes installs the update in a new app directory. It then moves the user's data files from the old installation over to the new app directory before deleting the old installation. Files in the following directories are guaranteed to be preserved during the update process:

- <*Application_Data*>`/Documents`
- <*Application_Data*>`/Library`

Although files in other user directories may also be moved over, you should not rely on them being present after an update.

Move Work off the Main Thread

Be sure to limit the type of work you do on the main thread of your app. The main thread is where your app handles touch events and other user input. To ensure that your app is always responsive to the user, you should never use the main thread to perform long-running or potentially unbounded tasks, such as tasks that access the network. Instead, you should always move those tasks onto background threads. The preferred way to do so is to use Grand Central Dispatch (GCD) or `NSOperation` objects to perform tasks asynchronously.

Moving tasks into the background leaves your main thread free to continue processing user input, which is especially important when your app is starting up or quitting. During these times, your app is expected to respond to events in a timely manner. If your app's main thread is blocked at launch time, the system could kill the app before it even finishes launching. If the main thread is blocked at quitting time, the system could similarly kill the app before it has a chance to write out crucial user data.

For more information about using GCD, operation objects, and threads, see *Concurrency Programming Guide*.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Document Revision History

This table describes the changes to *App Programming Guide for iOS*.

Date	Notes
2014-09-17	Reorganized the content and removed outdated information. Changed the name of the book from *iOS App Programming Guide*. Removed information about legacy techniques for specifying app icons and launch images. Relevant information moved to the corresponding key descriptions in *Information Property List Key Reference*. Added the Understanding When Your App Gets Launched into the Background (page 44) section to provide guidance about what technologies cause an app to be launched automatically. Fixed numerous bugs in the existing content.
2013-10-23	Added links to the Japanese smartphone privacy initiatives.
2013-09-18	Added information about new background execution modes and about app icon sizes in iOS 7.
2013-04-23	Added a section about privacy best practices.
2013-01-28	Added explicit information about how to support iPhone 5.
2012-09-19	Contains information about new features in iOS 6.
2012-03-07	Added information about the NSURL and CFURL keys used to prevent a file from being backed up.
2012-01-09	Updated the section that describes the behavior of apps in the background.
2011-10-12	Added information about features introduced in iOS 5.0.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Date	Notes
	Reorganized book and added more design-level information. Added high-level information about iCloud and how it impacts the design of applications.
2011-02-24	Added information about using AirPlay in the background.
2010-12-13	Made minor editorial changes.
2010-11-15	Incorporated additional iPad-related design guidelines into this document. Updated the information about how keychain data is preserved and restored.
2010-08-20	Fixed several typographical errors and updated the code sample on initiating background tasks.
2010-06-30	Updated the guidance related to specifying application icons and launch images. Changed the title from *iPhone Application Programming Guide*.
2010-06-14	Reorganized the book so that it focuses on the design of the core parts of your application. Added information about how to support multitasking in iOS 4 and later. For more information, see Core App Objects (page 16). Updated the section describing how to determine what hardware is available. Added information about how to support devices with high-resolution screens. Incorporated iPad-related information.
2010-02-24	Made minor corrections.
2010-01-20	Updated the “Multimedia Support” chapter with improved descriptions of audio formats and codecs.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Date	Notes
2009-10-19	Moved the iPhone specific `Info.plist` keys to *Information Property List Key Reference*. Updated the “Multimedia Support” chapter for iOS 3.1.
2009-06-17	Added information about using the compass interfaces. Moved information about OpenGL support to *OpenGL ES Programming Guide for iOS*. Updated the list of supported `Info.plist` keys.
2009-03-12	Updated for iOS 3.0 Added code examples to "Copy and Paste Operations" in the Event Handling chapter. Added a section on keychain data to the Files and Networking chapter. Added information about how to display map and email interfaces. Made various small corrections.
2009-01-06	Fixed several typos and clarified the creation process for child pages in the Settings application.
2008-11-12	Added guidance about floating-point math considerations Updated information related to what is backed up by iTunes.
2008-10-15	Reorganized the contents of the book. Moved the high-level iOS information to *iOS Technology Overview*. Moved information about the standard system URL schemes to *Apple URL Scheme Reference*. Moved information about the development tools and how to configure devices to *Tools Workflow Guide for iOS*. Created the Core Application chapter, which now introduces the application architecture and covers much of the guidance for creating iPhone applications.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Date	Notes
	Added a Text and Web chapter to cover the use of text and web classes and the manipulation of the onscreen keyboard. Created a separate chapter for Files and Networking and moved existing information into it. Changed the title from *iPhone OS Programming Guide*.
2008-07-08	New document that describes iOS and the development process for iPhone applications.

2014-09-17 | Copyright © 2014 Apple Inc. All Rights Reserved.

Apple Inc.
Copyright © 2014 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer or device for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-branded products.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, AirPlay, Bonjour, Cocoa, Instruments, iPad, iPhone, iPod, iTunes, Mac, Macintosh, Objective-C, Shake, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

AirDrop and Retina are trademarks of Apple Inc.

iCloud is a service mark of Apple Inc., registered in the U.S. and other countries.

App Store is a service mark of Apple Inc.

IOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Intel and Intel Core are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

OpenGL is a registered trademark of Silicon Graphics, Inc.

APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT, ERROR OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

Some jurisdictions do not allow the exclusion of implied warranties or liability, so the above exclusion may not apply to you.