Strategies for Handling App State Transitions

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.

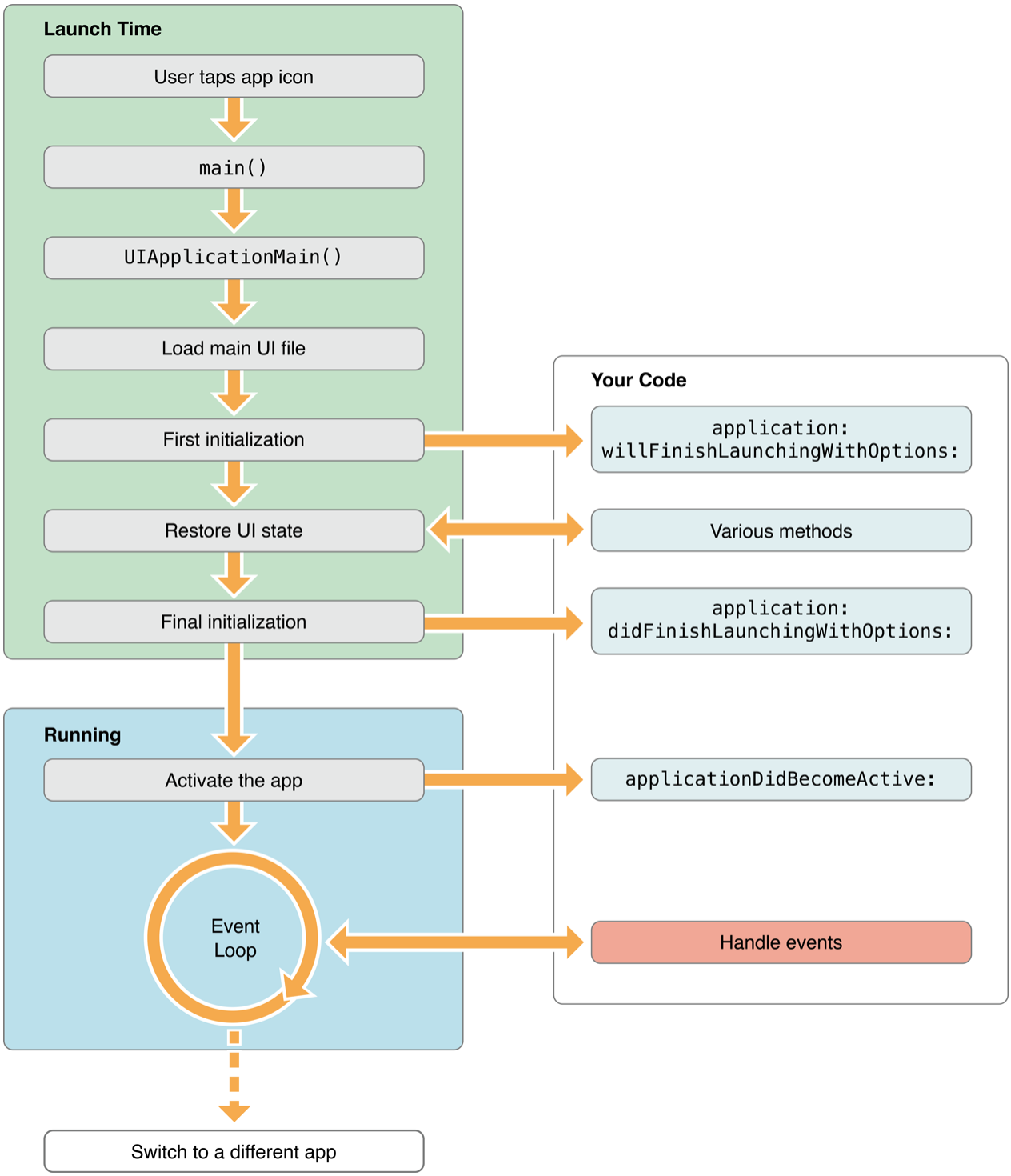
Your `application:willFinishLaunchingWithOptions:` and `application:didFinishLaunchingWithOptions:` methods should always be as lightweight as possible to reduce your app's launch time. Apps are expected to launch, initialize themselves, and start handling events in less than 5 seconds. If an app does 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.

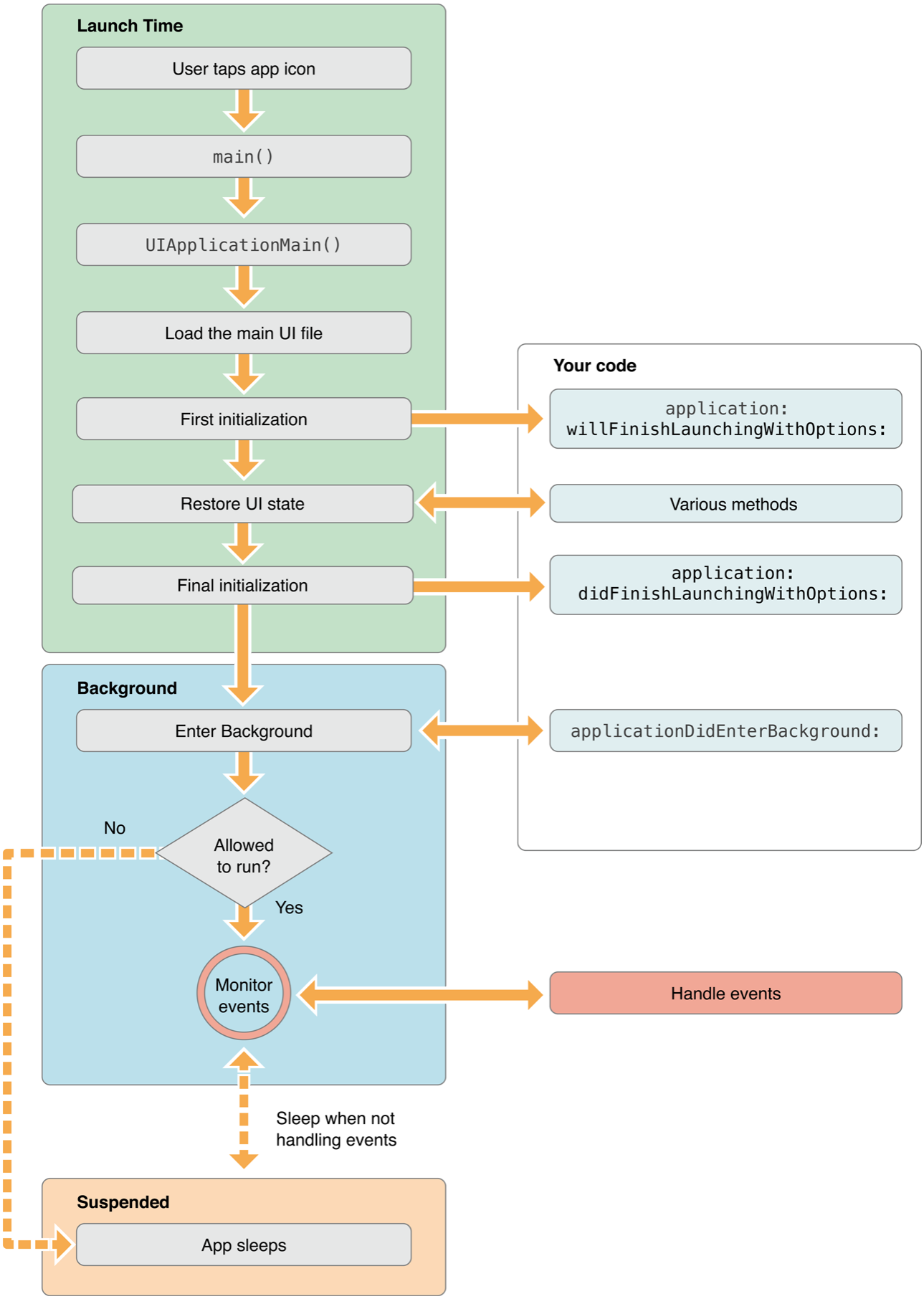
Figure 4-1 shows the sequence of events that occurs when an app is launched into the foreground, including the app delegate methods that are called.

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.

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 and Figure 4-2. For information about the startup sequences that occur when opening a URL, see Handling URL Requests.

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.

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.

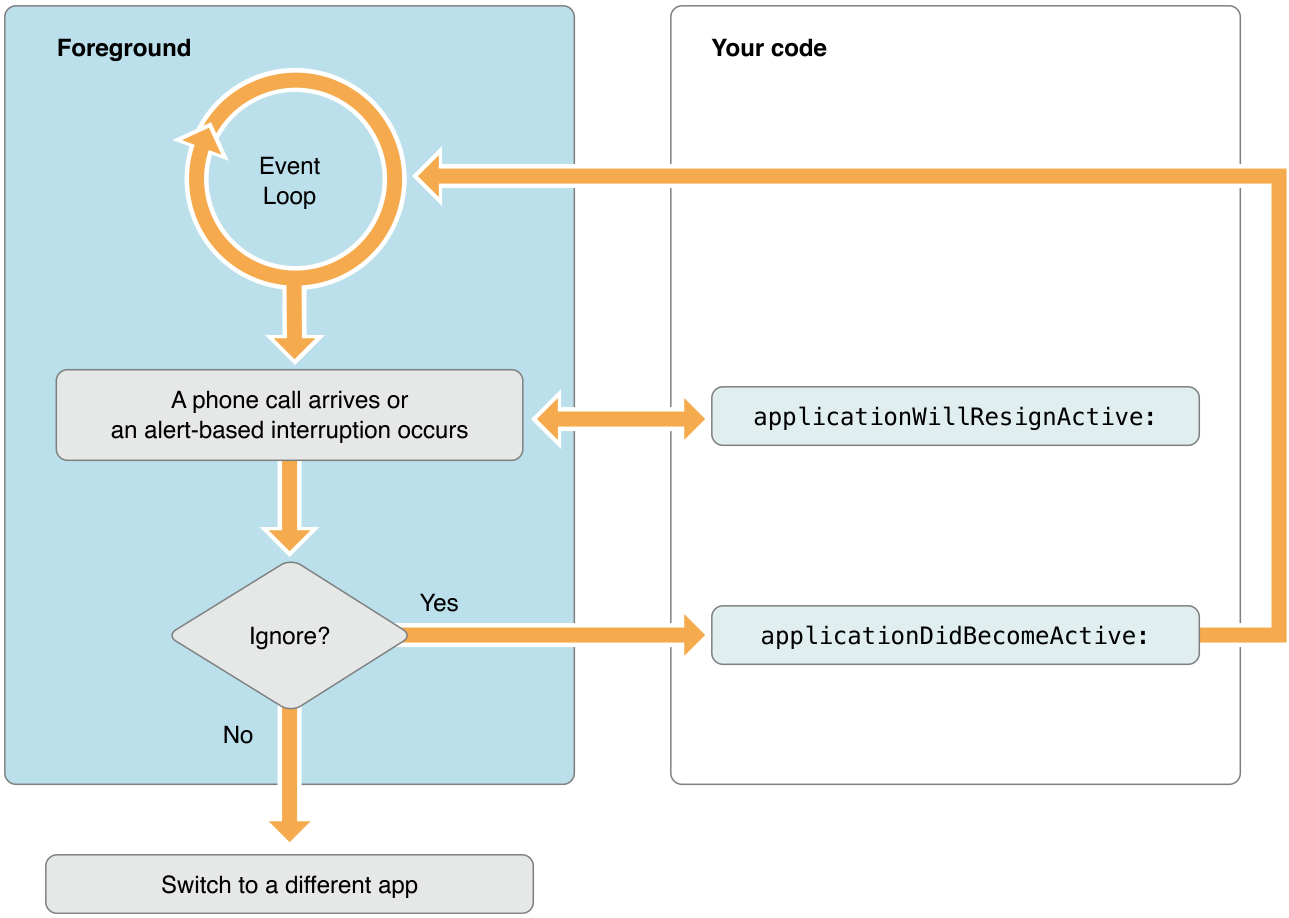
When the user presses the Sleep/Wake button, apps with files protected by the `NSFileProtectionComplete` protection option must close any references to those files. For devices configured with an appropriate password, pressing the Sleep/Wake button locks the screen and forces the system to throw away 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.

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



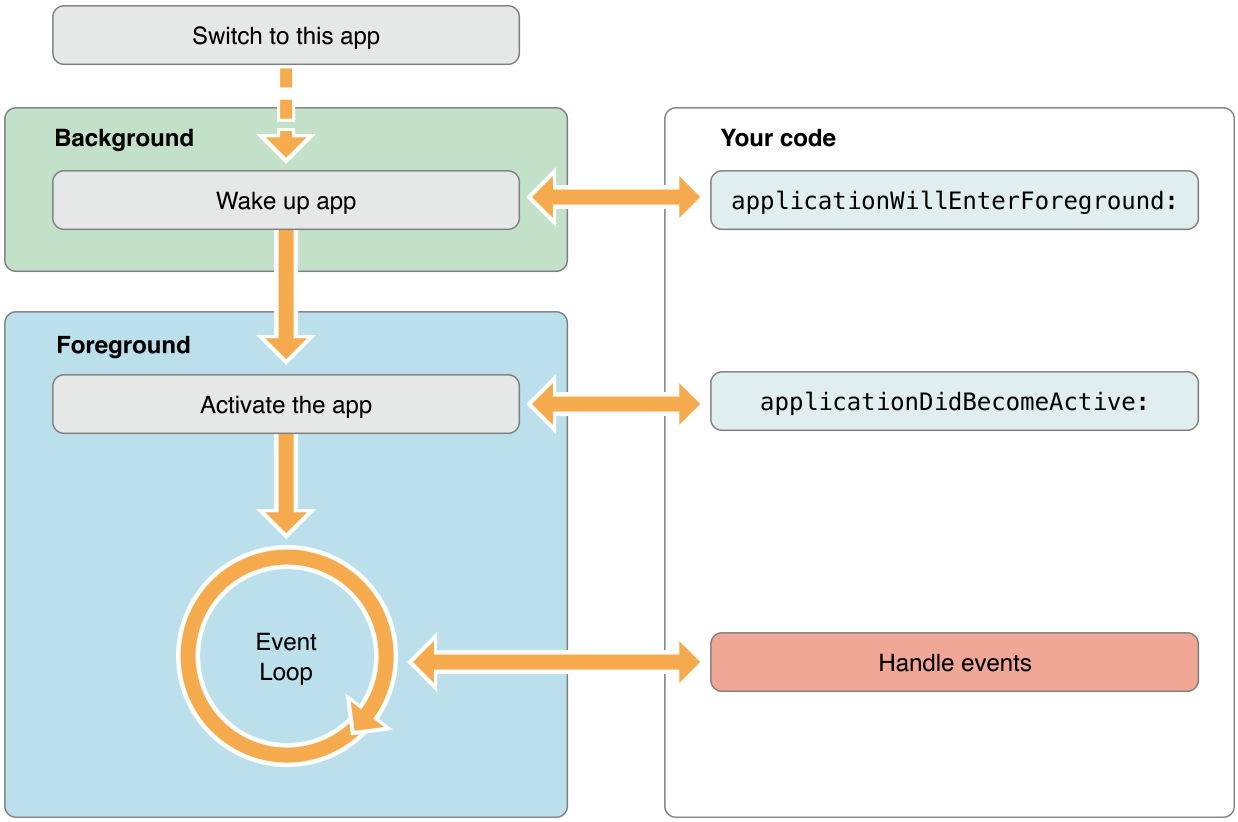
Notifications that display a banner do not deactivate your app in the way that alert-based notifications do. Instead, the banner is laid along the top edge of your app window and your app continues receive touch events as before. However, if the user pulls down the banner to reveal the notification center,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

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`
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.

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 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.
- **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.

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.

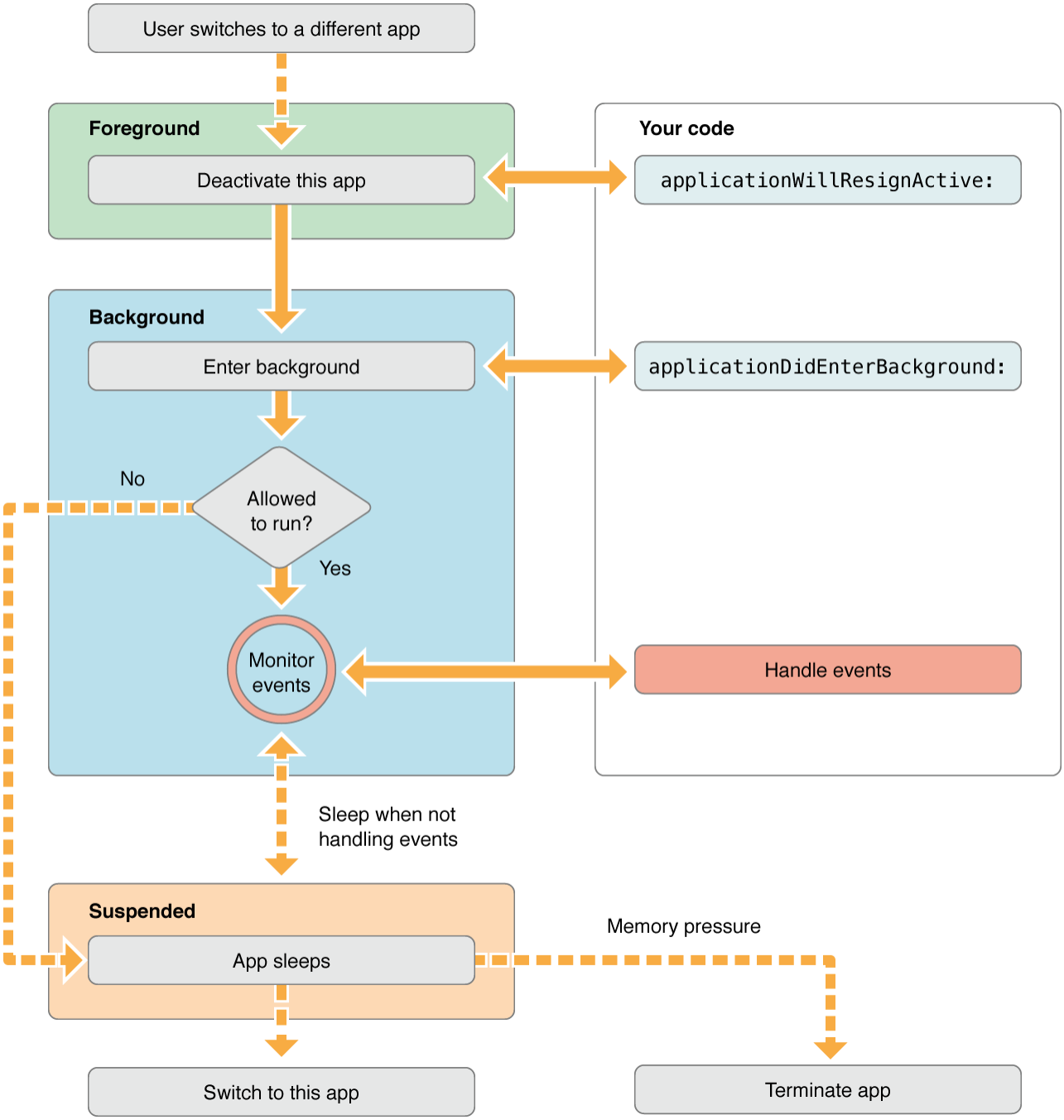
Depending on the features of your app, there are other things your app should do when moving to

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.

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