Local and Remote Notification Programming Guide

Developer

Contents

Figures, Tables, and Listings

Objective-CSwift

About Local Notifications and Remote Notifications

Local notifications and remote notifications are the two types of user notifications. (Remote notifications are also known as *push* notifications.) Both types of user notifications enable an app that isn't running in the foreground to let its users know it has information for them. The information could be a message, an impending calendar event, or new data on a remote server. When presented by the operating system, local and remote user notifications look and sound the same. They can display an alert message or they can badge the app icon. They can also play a sound when the alert or badge number is shown.

When users are notified that the app has a message, event, or other data for them, they can launch the app and see the details. They can also choose to ignore the notification, in which case the app is not activated.

> **Note:** Remote notifications and local notifications are *not* related to broadcast notifications (`NSNotificationCenter`) or key-value observing notifications.

At a Glance

Local notifications and remote notifications have several important aspects you should be aware of.

Local and Remote Notifications Solve Similar Problems

Only one app can be active in the foreground at any time. Many apps operate in a time-based or interconnected environment where events of interest to users can occur when the app is not in the foreground. Local and remote notifications allow these apps to notify their users when these events occur.

> **Relevant Chapter:** Local and Push Notifications in Depth (page 11)

Local and Remote Notifications Are Different in Origination

Local and remote notifications serve different design needs. A local notification is scheduled and sent by the app itself. Remote notifications—also known as *push* notifications—arrive from outside a device or a Mac. They originate on a remote server—the app's provider—and are pushed to apps on devices (via the Apple Push Notification service) when there are messages to see or data to download.

Relevant Chapter: Local and Push Notifications in Depth (page 11)

You Register, Schedule, and Handle Both Local and Remote Notifications

To have the system deliver a local notification at a later time, an app registers notification types (in iOS 8 and later), creates a local notification object (using either `UILocalNotification` or `NSUserNotification`), assigns it a delivery date and time, specifies presentation details, and schedules it for delivery. To receive remote notifications, an app must register notification types, then pass to its provider a device token it gets from the operating system.

When the operating system delivers a local notification or remote notification and the target app is not running in the foreground, it can present the notification to the user through an alert, icon badge number, or sound. If there is a notification alert and the user taps or clicks an action button (or moves the action slider), the app launches and calls a method to pass in the local-notification object or remote-notification payload. If the app is running in the foreground when the notification is delivered, the app delegate receives a local or remote notification.

In iOS 8 and later, user notifications can include custom actions. Also, location-based notifications can be sent whenever the user arrives at a particular geographic location.

Relevant Chapter: Scheduling, Registering, and Handling Notifications (page 15)

The Apple Push Notification Service Is the Gateway for Remote Notifications

Apple Push Notification service (APNs) propagates remote notifications to devices having apps registered to receive those notifications. Each device establishes an accredited and encrypted IP connection with the service and receives notifications over this persistent connection. Providers connect with APNs through a persistent and secure channel while monitoring incoming data intended for their client apps. When new data for an app arrives, the provider prepares and sends a notification through the channel to APNs, which pushes the notification to the target device.

Related Chapter: Apple Push Notification Service (page 36)

You Must Obtain Security Credentials for Remote Notifications

To develop and deploy the provider side of an app for remote notifications, you must get SSL certificates from Member Center. Each certificate is limited to a single app, identified by its bundle ID; it is also limited to one of two environments, one for development and one for production. These environments have their own assigned IP address and require their own certificates. You must also obtain provisioning profiles for each of these environments.

Related Chapter: Provisioning and Development (page 52)

The Provider Communicates with APNs over a Binary Interface

The binary interface is asynchronous and uses a streaming TCP socket design for sending remote notifications as binary content to APNs. There is a separate interface for the development and production environments, each with its own address and port. For each interface, you need to use TLS (or SSL) and the SSL certificate you obtained to establish a secured communications channel. The provider composes each outgoing notification and sends it over this channel to APNs.

APNs has a feedback service that maintains a per-app list of devices for which there were failed-delivery attempts (that is, APNs was unable to deliver a remote notification to an app on a device). Periodically, the provider should connect with the feedback service to see what devices have persistent failures so that it can refrain from sending remote notifications to them.

Related Chapters: Apple Push Notification Service (page 36), Provider Communication with Apple Push Notification Service (page 54)

Prerequisites

App Programming Guide for iOS describes the high level patterns for writing iOS apps.

For local notifications and the client-side implementation of remote notifications, familiarity with app development for iOS is assumed. For the provider side of the implementation, knowledge of TLS/SSL and streaming sockets is helpful.

See Also

The following documents provide background information:

- *App Distribution Quick Start* teaches how to create a team provisioning profile in Xcode before you enable APNs.
- *App Distribution Guide* describes how to perform a variety of tasks in Xcode and Member Center, such as configuring APNs.
- *Entitlement Key Reference* documents the specific entitlements needed for an app to receive remote notifications.

You might find these additional sources of information useful for understanding and implementing local and remote notifications:

- The reference documentation for `UILocalNotification`, `UIApplication`, and `UIApplicationDelegate` describe the local- and remote-notification API for client apps in iOS.
- The reference documentation for `NSApplication` and `NSApplicationDelegate Protocol` describe the remote-notification API for client apps in OS X.
- *Security Overview* describes the security technologies and techniques used for the iOS and OS X systems.
- RFC 5246 is the standard for the TLS protocol.

  Secure communication between data providers and Apple Push Notification service requires knowledge of Transport Layer Security (TLS) or its predecessor, Secure Sockets Layer (SSL). Refer to one of the many online or printed descriptions of these cryptographic protocols for further information.

For information on how to send push notifications to your website visitors using OS X, read Configuring Safari Push Notifications in *Notification Programming Guide for Websites* .

Local and Remote Notifications in Depth

The essential purpose of both local and remote notifications is to enable an app to inform its users that it has something for them—for example, a message or an upcoming appointment—when the app isn't running in the foreground. The essential difference between local notifications and remote notifications is simple:

- Local notifications are scheduled by an app and delivered on the same device.
- Remote notifications, also known as *push* notifications, are sent by your server to the Apple Push Notification service, which pushes the notification to devices.

Local and Remote Notifications Appear the Same to Users

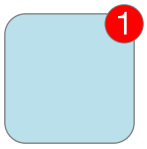
Users can get notified in the following ways:

- An onscreen alert or banner
- A badge on the app's icon
- A sound that accompanies an alert, banner, or badge

From a user's perspective, both local and remote notifications indicate that there is something of interest in the app.

For example, consider an app that manages a to-do list, and each item in the list has a date and time when the item must be completed. The user can request the app to notify it at a specific interval before this due date expires. To effect this, the app schedules a local notification for that date and time. Instead of specifying an alert message, the app chooses to specify a badge number (1) and a sound. At the appointed time, iOS plays the sound ad displays the badge number in the upper-right corner of the icon of the app, such as illustrated in Figure 1-1.

Figure 1-1 An app icon with a badge number (iOS)



The user hears the sound and sees the badge and responds by launching the app to see the to-do item. Users control how the device and specific apps installed on the device should handle notifications. They can also selectively enable or disable remote notification types (that is, icon badging, alert messages, and sounds) for specific apps.

Local and Remote Notifications Appear Different to Apps

If your app is frontmost when a local or remote notification arrives, the `application:didReceiveRemoteNotification:` or `application:didReceiveLocalNotification:` method is called on its app delegate. If your app is not frontmost or not running, you handle the notifications by checking the options dictionary passed to the `application:didFinishLaunchingWithOptions:` of your app delegate for either the `UIApplicationLaunchOptionsLocalNotificationKey` or `UIApplicationLaunchOptionsRemoteNotificationKey` key. For more details about handling notifications, see Scheduling, Registering, and Handling Notifications (page 15).

More About Local Notifications

Local notifications are ideally suited for apps with time-based behaviors, such as calendar and to-do list apps. Apps that run in the background for the limited period allowed by iOS might also find local notifications useful. For example, apps that depend on servers for messages or data can poll their servers for incoming items while running in the background; if a message is ready to view or an update is ready to download, they can handle the data as needed, and notify users if appropriate.

A local notification is an instance of `UILocalNotification` or `NSUserNotification` with three general kinds of properties:

- **Scheduled time.** You must specify the date and time the operating system delivers the notification; this is known as the *fire date*. You can qualify the fire date with a specific time zone so that the system can make adjustments to the fire date when the user travels. You can also request the operating system to reschedule the notification at a regular interval (weekly, monthly, and so on).
- **Notification type.** These properties include the alert message, the title of the default action button, the app icon badge number, a sound to play, and optionally in iOS 8 and later, a category of custom actions.
- **Custom data.** Local notifications can include a user info dictionary of custom data.

Listing 2-9 (page 27) describes these properties in programmatic detail. After an app has created a local-notification object, it can either schedule it with the operating system or present it immediately.

Each app on a device is limited to 64 scheduled local notifications. The system discards scheduled notifications in excess of this limit, keeping only the 64 notifications that will fire the soonest. Recurring notifications are treated as a single notification.

More About Remote Notifications

An iOS or Mac app is often a part of a larger application based on the client/server model. The client side of the app is installed on the device or computer; the server side of the app has the main function of providing data to its client apps, and hence is termed a *provider*. A client app periodically connects with its provider and downloads any data that is waiting for it. Email and social-networking apps are examples of this client/server model.

But what if the app is not connected to its provider or even running on the device or computer when the provider has new data for it to download? How does it learn about this waiting data? Remote (or push) notifications are the solution to this dilemma. A remote notification is a short message that a provider has delivered to the operating system of a device or computer; the operating system, in turn, can inform the user of a client app that there is data to be downloaded, a message to be viewed, and so on. If the user enables this feature (on iOS) and the app is properly registered, the notification is delivered to the operating system and possibly to the app. Apple Push Notification service (APNs) is the primary technology for the remote-notification feature.

Remote notifications serve much the same purpose as a background app on a desktop system, but without the additional overhead. For an app that is not currently running—or, in the case of iOS, not running in the foreground—the notification occurs indirectly. The operating system receives a remote notification on behalf of the app and alerts the user. Once alerted, users may choose to launch the app, which then downloads the data from its provider. If an app is running when a notification comes in, the app can choose to handle the notification directly.

As its name suggests, Apple Push Notification service uses a remote design to deliver remote notifications to devices and computers. A push design differs from its opposite, a pull design, in that the recipient of the notification passively listens for updates rather than actively polling for them. A push design makes possible a wide and timely dissemination of information with few of the scalability problems inherent with pull designs. APNs uses a persistent IP connection for implementing remote notifications.

Most of a remote notification consists of a payload: a property list containing APNs-defined properties specifying how the user is to be notified. For performance reasons, the payload is deliberately small. Although you may define custom properties for the payload, you should never use the remote-notification mechanism for data transport because delivery of remote notifications is not guaranteed. For more on the payload, see The Notification Payload (page 44).

APNs retains the last notification it receives from a provider for an app on a device; so, if a device or computer comes online and has not received the notification, APNs pushes the stored notification to it. A device running iOS receives remote notifications over both Wi-Fi and cellular connections; a computer running OS X receives remote notifications over both Wi-Fi and Ethernet connections.

Important: In iOS, Wi-Fi is used for remote notifications only if there is no cellular connection or if the device is an iPod touch. For some devices to receive notifications via Wi-Fi, the device's display must be on (that is, it cannot be sleeping) or it must be plugged in. The iPad, on the other hand, remains associated with the Wi-Fi access point while asleep, thus permitting the delivery of remote notifications. The Wi-Fi radio wakes the host processor for any incoming traffic.

Sending notifications too frequently negatively impacts the device's battery life—devices must access the network to receive notifications.

Adding the remote-notification feature to your app requires that you obtain the proper certificates from Member Center for either the iOS Developer Program or Mac Developer Program and then write the requisite code for the client and provider sides of the app. Provisioning and Development (page 52) explains the provisioning and setup steps, and Provider Communication with Apple Push Notification Service (page 54) and Scheduling, Registering, and Handling Notifications (page 15) describe the details of implementation.

Registering, Scheduling, and Handling User Notifications

Objective-CSwift

The are several tasks that an iOS or OS X app should do to register, schedule, and handle both local and remote notifications.

Registering for Notification Types in iOS

In iOS 8 and later, apps that use either local or remote notifications must register the types of notifications they intend to deliver. The system then gives the user the ability to limit the types of notifications your app displays. The system does not badge icons, display alert messages, or play alert sounds if any of these notification types are not enabled for your app, even if they are specified in the notification payload.

In iOS, use the `registerUserNotificationSettings:` method of `UIApplication` to register notification types. The notification types represent the user interface elements the app displays when it receives a notification: badging the app's icon, playing a sound, and displaying an alert. If you don't register any notification types, the system pushes all remote notifications to your app silently, that is, without displaying any user interface. Listing 2-1 shows how an app registers notification types.

Listing 2-1 Registering notification types

```
UIUserNotificationType types = UIUserNotificationTypeBadge |
            UIUserNotificationTypeSound | UIUserNotificationTypeAlert;

UIUserNotificationSettings *mySettings =
            [UIUserNotificationSettings settingsForTypes:types categories:nil];

[[UIApplication sharedApplication] registerUserNotificationSettings:mySettings];
```

Notice the use of the `categories` parameter in Listing 2-1. A category is a group of actions that can be displayed in conjunction with a single notification. You can learn more about categories in Using Notification Actions in iOS (page 25).

The first time you call the `registerUserNotificationSettings:` method, iOS presents a dialog that asks the user for permission to present the types of notifications the app registered. After the user replies, iOS asynchronously calls back to the `UIApplicationDelegate` object with the `application:didRegisterUserNotificationSettings:` method, passing a `UIUserNotificationType` object that specifies the types of notifications the user allows.

Users can change their notification settings at any time using the Settings app. Your app is added to the Settings app as soon as you call `registerUserNotificationSettings:`. Users can enable or disable notifications, as well as modify where and how notifications are presented. Because the user can change their initial setting at any time, call `currentUserNotificationSettings` before you do any work preparing a notification for presentation.

In iOS 8 and later, calling `registerUserNotificationSettings:` applies to remote notifications as well as local notifications. Because doing so specifies the types of remote notifications the app displays, the `registerForRemoteNotificationTypes:` is deprecated in iOS 8. You can learn more about remote notifications in Registering for Remote Notifications (page 19).

Scheduling Local Notifications

In iOS, you create a `UILocalNotification` object and schedule its delivery using the `scheduleLocalNotification:` method of `UIApplication`. In OS X, you create an `NSUserNotification` object (which includes a delivery time) and the `NSUserNotificationCenter` is responsible for delivering it appropriately. (An OS X app can also adopt the `NSUserNotificationCenterDelegate` protocol to customize the behavior of the default `NSUserNotificationCenter` object.)

Creating and scheduling local notifications in iOS requires that you perform the following steps:

1. In iOS 8 and later, register for notification types, as described in Registering for Notification Types in iOS (page 15). (In OS X and earlier versions of iOS, you need register only for remote notifications.) If you already registered notification types, call `currentUserNotificationSettings` to get the types of notifications the user accepts from your app.
2. Allocate and initialize a `UILocalNotification` object.
3. Set the date and time that the operating system should deliver the notification. This is the `fireDate` property.

   If you set the `timeZone` property to the `NSTimeZone` object for the current locale, the system automatically adjusts the fire date when the device travels across (and is reset for) different time zones. (Time zones affect the values of date components—that is, day, month, hour, year, and minute—that the system calculates for a given calendar and date value.)

   You can also schedule the notification for delivery on a recurring basis (daily, weekly, monthly, and so on).

4. As appropriate, configure an alert, icon badge, or sound so that the notification can be delivered to users according to their preferences. (To learn about when different notification types are appropriate, see Notifications.)
    - An alert has a property for the message (the `alertBody` property) and for the title of the action button or slider (`alertAction`). Specify strings that are localized for the user's current language preference. If your notifications can be displayed on Apple Watch, assign a value to the `alertTitle` property.
    - To display a number in a badge on the app icon, use the `applicationIconBadgeNumber` property.
    - To play a sound, assign a sound to the `soundName` property. You can assign the filename of a nonlocalized custom sound in the app's main bundle or you can assign `UILocalNotificationDefaultSoundName` to get the default system sound. A sound should always accompany the display of an alert message or the badging of an icon; a sound should not be played in the absence of other notification types.
5. Optionally, you can attach custom data to the notification through the `userInfo` property. For example, a notification that's sent when a CloudKit record changes includes the identifier of the record, so that a handler can get the record and update it.
6. Optionally, in iOS 8 and later, your local notification can present custom actions that your app can perform in response to user interaction, as described in Using Notification Actions in iOS (page 25).
7. Schedule the local notification for delivery.

    You schedule a local notification by calling `scheduleLocalNotification:`. The app uses the fire date specified in the `UILocalNotification` object for the moment of delivery. Alternatively, you can present the notification immediately by calling the `presentLocalNotificationNow:` method.

The method in Listing 2-2 creates and schedules a notification to inform the user of a hypothetical to-do list app about the impending due date of a to-do item. There are a couple things to note about it. For the `alertBody`, `alertAction`, and `alertTitle` properties, it fetches from the main bundle (via the `NSLocalizedString` macro) strings localized to the user's preferred language. It also adds the name of the relevant to-do item to a dictionary assigned to the `userInfo` property.

Listing 2-2 Creating, configuring, and scheduling a local notification

```
- (void)scheduleNotificationWithItem:(ToDoItem *)item interval:(int)minutesBefore
 {
    NSCalendar *calendar = [NSCalendar autoupdatingCurrentCalendar];
    NSDateComponents *dateComps = [[NSDateComponents alloc] init];
    [dateComps setDay:item.day];
    [dateComps setMonth:item.month];
```

```
    [dateComps setYear:item.year];
    [dateComps setHour:item.hour];
    [dateComps setMinute:item.minute];
    NSDate *itemDate = [calendar dateFromComponents:dateComps];

    UILocalNotification *localNotif = [[UILocalNotification alloc] init];
    if (localNotif == nil)
        return;
    localNotif.fireDate = [itemDate
dateByAddingTimeIntervalInterval:-(minutesBefore*60)];
    localNotif.timeZone = [NSTimeZone defaultTimeZone];

    localNotif.alertBody = [NSString stringWithFormat:NSLocalizedString(@"%@ in
%i minutes.", nil),
         item.eventName, minutesBefore];
    localNotif.alertAction = NSLocalizedString(@"View Details", nil);
    localNotif.alerttitle = NSLocalizedString(@"Item Due", nil);

    localNotif.soundName = UILocalNotificationDefaultSoundName;
    localNotif.applicationIconBadgeNumber = 1;

    NSDictionary *infoDict = [NSDictionary dictionaryWithObject:item.eventName
forKey:ToDoItemKey];
    localNotif.userInfo = infoDict;

    [[UIApplication sharedApplication] scheduleLocalNotification:localNotif];
}
```

You can cancel a specific scheduled notification by calling `cancelLocalNotification:` on the app object, and you can cancel all scheduled notifications by calling `cancelAllLocalNotifications`. Both of these methods also programmatically dismiss a currently displayed notification alert. For example, you might want to cancel a notification that's associated with a reminder the user no longer wants.

Apps might also find local notifications useful when they run in the background and some message, data, or other item arrives that might be of interest to the user. In this case, an app can present the notification immediately using the `UIApplication` method `presentLocalNotificationNow:` (iOS gives an app a limited time to run in the background).

In OS X, you might write code like that shown in Listing 2-3 to create a local notification and schedule it for delivery. Note that OS X doesn't deliver a local notification if your app is currently frontmost. Also, OS X users can change their preferences for receiving notifications in System Preferences.

Listing 2-3 Creating and scheduling a local notification in OS X

```
//Create a new local notification
NSUserNotification *notification = [[NSUserNotification alloc] init];
//Set the title of the notification
notification.title = @"My Title";
//Set the text of the notification
notification.informativeText = @"My Text";
//Schedule the notification to be delivered 20 seconds after execution
notification.deliveryDate = [NSDate dateWithTimeIntervalSinceNow:20];

//Get the default notification center and schedule delivery
[[NSUserNotificationCenter defaultUserNotificationCenter]
scheduleNotification:notification];
```

Registering for Remote Notifications

An app must register with Apple Push Notification service (APNs) to receive remote notifications sent by the app's push provider. In iOS 8 and later, registration has four stages:

1. Register the notification types your app supports using `registerUserNotificationSettings:`.
2. Register to receive push notifications via APNs by calling your app's `registerForRemoteNotifications` method.
3. Store the device token returned to the app delegate by the server for a successful registration, or handle registration failure gracefully.
4. Forward the device token to the app's push provider.

(In iOS 7, instead of the first two steps, you register by calling the `registerForRemoteNotificationTypes:` method of `UIApplication`, and in OS X by calling the `registerForRemoteNotificationTypes:` method of `NSApplication`.) The actions that take place during the registration sequence are illustrated by Figure 3-3 (page 42) in Token Generation and Dispersal (page 41).

Device tokens can change, so your app needs to reregister every time it is launched.

If registration is successful, APNs returns a device token to the device and iOS passes the token to the app delegate in the `application:didRegisterForRemoteNotificationsWithDeviceToken:` method. The app passes this token, encoded in binary format, to its provider. If there is a problem in obtaining the token, the operating system informs the delegate by calling the `application:didFailToRegisterForRemoteNotificationsWithError:` method (or the `application:didFailToRegisterForRemoteNotificationsWithError:` method in OS X). The `NSError` object passed into this method clearly describes the cause of the error. The error might be, for instance, an erroneous `aps-environment` value in the provisioning profile. You should view the error as a transient state and not attempt to parse it.

> **iOS Note:** If a cellular or Wi-Fi connection is not available, neither the `application:didRegisterForRemoteNotificationsWithDeviceToken:` method nor the `application:didFailToRegisterForRemoteNotificationsWithError:` method is called. For Wi-Fi connections, this sometimes occurs when the device cannot connect with APNs over port 5223. If this happens, the user can move to another Wi-Fi network that isn't blocking this port or, on an iPhone or iPad, wait until the cellular data service becomes available. In either case, the device should be able to make the connection, and then one of the delegation methods is called.

By requesting the device token and passing it to the provider every time your app launches, you ensure that the provider has the current token for the device. Otherwise, pushes may not make their way to the user's device. If a user restores a backup to a device or computer other than the one that the backup was created for (for example, the user migrates data to a new device or computer), the user must launch the app at least once for it to receive notifications again. If the user restores backup data to a new device or computer, or reinstalls the operating system, the device token changes. Moreover, never cache a device token and give that to your provider; always get the token from the system whenever you need it. If your app has previously registered, calling `registerForRemoteNotifications` results in the operating system passing the device token to the delegate immediately without incurring additional overhead. Also note that the delegate method may be called any time the device token changes, not just in response to your app registering or re-registering.

Listing 2-4 gives a simple example of how you might register for remote notifications in an iOS app. The code would be similar for a Mac app.

Listing 2-4 Registering for remote notifications

```
- (void)applicationDidFinishLaunching:(UIApplication *)app {
   // other setup tasks here....
    UIUserNotificationType types = UIUserNotificationTypeBadge |
                UIUserNotificationTypeSound | UIUserNotificationTypeAlert;
```

```
    UIUserNotificationSettings *mySettings =
                [UIUserNotificationSettings settingsForTypes:types categories:nil];

    [[UIApplication sharedApplication] registerUserNotificationSettings:mySettings];
    [app.registerForRemoteNotifications];
}

// Delegation methods
- (void)application:(UIApplication *)app
didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)devToken {
    const void *devTokenBytes = [devToken bytes];
    self.registered = YES;
    [self sendProviderDeviceToken:devTokenBytes]; // custom method
}

- (void)application:(UIApplication *)app
didFailToRegisterForRemoteNotificationsWithError:(NSError *)err {
    NSLog(@"Error in registration. Error: %@", err);
}
```

In your `application:didFailToRegisterForRemoteNotificationsWithError:` implementation, you should process that error object appropriately and make sure you disable any logic within your app that depends on receiving remote notifications. You don't want do any unnecessary processing within your app for notifications that aren't going to be coming in. Just gracefully degrade.

Handling Local and Remote Notifications

Let's review the possible scenarios that can arise when the system delivers a local notification or a remote notification for an app.

The notification is delivered when the app isn't running in the foreground. In this case, the system presents the notification, displaying an alert, badging an icon, perhaps playing a sound, and perhaps displaying one or more action buttons for the user to tap.

The user taps a custom action button in an iOS 8 notification. In this case, iOS calls either `application:handleActionWithIdentifier:forRemoteNotification:completionHandler:` or `application:handleActionWithIdentifier:forLocalNotification:completionHandler:`. In both methods, you get the identifier of the action so that you can determine which button the user tapped. You also get either the remote or local notification object, so that you can retrieve any information you need to handle the action.

The user taps the default button in the alert or taps (or clicks) the app icon. If the default action button is tapped (on a device running iOS), the system launches the app and the app calls its delegate's `application:didFinishLaunchingWithOptions:` method, passing in the notification payload (for remote notifications) or the local-notification object (for local notifications). Although `application:didFinishLaunchingWithOptions:` isn't the best place to handle the notification, getting the payload at this point gives you the opportunity to start the update process before your handler method is called.

If the notification is remote, the system also calls `application:didReceiveRemoteNotification:fetchCompletionHandler:`.

If the app icon is clicked on a computer running OS X, the app calls the delegate's `applicationDidFinishLaunching:` method in which the delegate can obtain the remote-notification payload. If the app icon is tapped on a device running iOS, the app calls the same method, but furnishes no information about the notification.

The notification is delivered when the app is running in the foreground. The app calls the `UIApplicationDelegate` method `application:didReceiveLocalNotification:` or `application:didReceiveRemoteNotification:fetchCompletionHandler:`. (If `application:didReceiveRemoteNotification:fetchCompletionHandler:` isn't implemented, the system calls `application:didReceiveRemoteNotification:`.) In OS X, the system calls `application:didReceiveRemoteNotification:`.

An app can use the passed-in remote-notification payload or, in iOS, the `UILocalNotification` object to help set the context for processing the item related to the notification. Ideally, the delegate does the following on each platform to handle the delivery of remote and local notifications in all situations:

- For OS X, the delegate should adopt the `NSApplicationDelegate` protocol and implement the `application:didReceiveRemoteNotification:` method.
- For iOS, the delegate should should adopt the `UIApplicationDelegate` protocol and implement the `application:didReceiveRemoteNotification:fetchCompletionHandler:` or `application:didReceiveLocalNotification:` methods. To handle notification actions, implement

the `application:handleActionWithIdentifier:forLocalNotification:completionHandler:` or `application:handleActionWithIdentifier:forRemoteNotification:completionHandler:` methods.

The delegate for an iOS app in Listing 2-5 implements the `application:didFinishLaunchingWithOptions:` method to handle a local notification. It gets the associated `UILocalNotification` object from the launch-options dictionary using the `UIApplicationLaunchOptionsLocalNotificationKey` key. From the `UILocalNotification` object's `userInfo` dictionary, it accesses the to-do item that is the reason for the notification and uses it to set the app's initial context. As shown in this example, you might appropriately reset the badge number on the app icon—or remove it if there are no outstanding items—as part of handling the notification.

Listing 2-5 Handling a local notification when an app is launched

```
- (BOOL)application:(UIApplication *)app didFinishLaunchingWithOptions:(NSDictionary
 *)launchOptions {
    UILocalNotification *localNotif =
       [launchOptions objectForKey:UIApplicationLaunchOptionsLocalNotificationKey];
    if (localNotif) {
        NSString *itemName = [localNotif.userInfo objectForKey:ToDoItemKey];
        [viewController displayItem:itemName];  // custom method
        app.applicationIconBadgeNumber = localNotif.applicationIconBadgeNumber-1;
    }
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
    return YES;
}
```

The implementation for a remote notification would be similar, except that you would use a specially declared constant in each platform as a key to access the notification payload:

- In iOS, the delegate, in its implementation of the `application:didFinishLaunchingWithOptions:` method, uses the `UIApplicationLaunchOptionsRemoteNotificationKey` key to access the payload from the launch-options dictionary.
- In OS X, the delegate, in its implementation of the `applicationDidFinishLaunching:` method, uses the `NSApplicationLaunchRemoteNotificationKey` key to access the payload dictionary from the `userInfo` dictionary of the `NSNotification` object that is passed into the method.

The payload itself is an `NSDictionary` object that contains the elements of the notification—alert message, badge number, sound, and so on. It can also contain custom data the app can use to provide context when setting up the initial user interface. See The Notification Payload (page 44) for details about the remote-notification payload.

> **Important:** Delivery of remote notifications is not guaranteed, so you should not use the notification payload to deliver sensitive data or data that can't be retrieved by other means.
>
> One example of an appropriate usage for a custom payload property is a string identifying an email account from which messages are downloaded to an email client; the app can incorporate this string in its download user-interface. Another example of custom payload property is a timestamp for when the provider first sent the notification; the client app can use this value to gauge how old the notification is.

When handling remote notifications in your notification handling methods, the app delegate might perform a major additional task. Just after the app launches, the delegate should connect with its provider and fetch the waiting data.

> **Note:** A client app should always communicate with its provider asynchronously or on a secondary thread.

The code in Listing 2-6 shows an implementation of the `application:didReceiveLocalNotification:` method which is called when app is running in the foreground. Here the app delegate does the same work as it does in Listing 2-5. It can access the `UILocalNotification` object directly this time because this object is an argument of the method.

Listing 2-6 Handling a local notification when an app is already running

```
- (void)application:(UIApplication *)app
didReceiveLocalNotification:(UILocalNotification *)notif {

    NSString *itemName = [notif.userInfo objectForKey:ToDoItemKey];

    [viewController displayItem:itemName];  // custom method

    app.applicationIconBadgeNumber = notification.applicationIconBadgeNumber - 1;

}
```

If you want your app to catch remote notifications that the system delivers while it is running in the foreground, the app delegate must implement the `application:didReceiveRemoteNotification:fetchCompletionHandler:` method. The delegate should begin the procedure for downloading the waiting data, message, or other item and, after this concludes,

it should remove the badge from the app icon. The dictionary passed in the second parameter of this method is the notification payload; you should not use any custom properties it contains to alter your app's current context.

Using Notification Actions in iOS

In OS X and iOS versions prior to iOS 8, user notifications can have only one default action. In iOS 8 and later, user notifications can have additional custom actions. Two actions can be displayed on the lock screen, in a banner, and in Notification Center. In modal alerts, notifications can display up to four actions when the user taps the Options button. To use notification actions in your app, you need to register the actions, schedule a local notification or push a remote notification, and handle the action chosen by the user.

Registering Notification Actions

To use notification actions in your app, you must define the actions, group them into categories, and then register them with your app's shared `UIApplication` instance.

To define a notification action, first you must create and initialize an instance of a notification action class, typically `UIMutableUserNotificationAction`. Then you define an identifier, passed back to your app when it handles the action, and a localized string displayed to the user on the action button. Next, you set the action's activation mode to foreground if the action needs to interrupt the user or background if not. Finally, you declare whether the action is destructive, meaning its button displays red, and whether choosing the action requires the user to enter their passcode. Listing 2-7 illustrates these steps.

Listing 2-7 Defining a notification action

```
UIMutableUserNotificationAction *acceptAction =
            [[UIMutableUserNotificationAction alloc] init];

// Define an ID string to be passed back to your app when you handle the action
acceptAction.identifier = @"ACCEPT_IDENTIFIER";

// Localized string displayed in the action button
acceptAction.title = @"Accept";

// If you need to show UI, choose foreground
acceptAction.activationMode = UIUserNotificationActivationModeBackground;
```

```
// Destructive actions display in red
acceptAction.destructive = NO;

// Set whether the action requires the user to authenticate
acceptAction.authenticationRequired = NO;
```

The `activationMode` property determines whether iOS launches your app in the foreground or background when the user responds to the notification. If you set it to `UIUserNotificationActivationModeBackground`, your app is given seconds to run. If the `destructive` property is `NO`, the action's button appears blue; if it's `YES`, the button is red. If you set the action's `authenticationRequired` property to `YES` and the device is locked when the user responds to the notification, the user must enter a passcode when choosing the action. However, this does not unlock the device, so if your app needs to access files, make sure the files are in the right data protection class. When the value of the `activationMode` property is `UIUserNotificationActivationModeForeground`, the value of the `authenticationRequired` property is assumed to be `YES` regardless of its actual value.

For example, to configure actions for a calendar app, an Accept action needs no additional user interaction after the user taps the Accept button, so its `activationMode` can be background. Also, the Accept action is not destructive, so it doesn't appear in red on the notification and lock screen, and it doesn't need authentication because accepting an invitation is relatively harmless. As another example, a Trash action to delete a message in a Mail app also needs no further user interaction, so it can run in the background, but it is destructive, so its `destructive` property should be set to `YES`, and it requires authentication because you don't want someone else deleting your messages. On the other hand, a Reply action requires user interaction, so the activation mode should be foreground. It's not destructive, but the user must unlock the device because foreground actions always require authentication, regardless of the value in the `authenticationRequired` property.

After you have defined your actions, you need to group each of them into a category, which associates a type of notification with a set of related actions. For example, an Invite category could have Accept, Maybe, and Decline actions. A New mail category could have Mark as Read and Trash, and a Tagged category could have Like, Comment, and Untag actions. When crafting local or remote notifications for a user's device, you specify the category that contains the actions you want to display with that notification. When the notification is displayed, iOS uses the category information to determine which buttons to display in the notification alert and to notify you of which action the user selected.

To group actions into a category, create and initialize an instance of a notification category class, typically `UIMutableUserNotificationCategory`. Then define an identifier for the category, which you include in local notifications and the push payload of remote notifications.

Next you add actions to the category and set their action context. There are two user notification action contexts: the default context, which supports four actions, and the minimal context, which displays two. The context relates to the part of the user interface in which the notification is presented—the lock screen only has room to display two actions, so the minimal context applies, whereas a modal alert has room for a full set of actions, and the default context applies. Listing 2-8 shows how these steps can be coded.

Listing 2-8 Grouping actions into categories

```
// First create the category
UIMutableUserNotificationCategory *inviteCategory =
        [[UIMutableUserNotificationCategory alloc] init];

// Identifier to include in your push payload and local notification
inviteCategory.identifier = @"INVITE_CATEGORY";

// Add the actions to the category and set the action context
[inviteCategory setActions:@[acceptAction, maybeAction, declineAction]
    forContext:UIUserNotificationActionContextDefault];

// Set the actions to present in a minimal context
[inviteCategory setActions:@[acceptAction, declineAction]
    forContext:UIUserNotificationActionContextMinimal];
```

The two `setActions:forContext:` messages in Listing 2-8 (page 27) ensure that the actions are presented in the correct order in the default context, and that the most important actions are presented in a minimal context. That is, in a modal alert, the actions displayed are Accept, Maybe, and Decline (in that order), but on the lock screen the two actions displayed are Accept and Decline. If the second `setActions:forContext:` were not specified, only the first two actions of the default context would be displayed on the lock screen: Accept and Maybe.

After you define your notification action categories, you need to register them. You do this by grouping them together in a set, providing them to your user notification settings, and then registering those settings with your shared app instance. Listing 2-9 illustrates these steps.

Listing 2-9 Registering notification categories

```
NSSet *categories = [NSSet setWithObjects:inviteCategory, alarmCategory, ...
```

```
UIUserNotificationSettings *settings =
       [UIUserNotificationSettings settingsForTypes:types categories:categories];

[[UIApplication sharedApplication] registerUserNotificationSettings:settings];
```

The `UIUserNotificationSettings` class method `settingsForTypes:categories:` method is the same one shown in Listing 2-1 (page 15) which passed `nil` for the `categories` parameter, and the notification settings are registered in the same way with the app instance. In this case, the notification categories, as well as the notification types, are included in the app's notification settings.

Pushing a Remote Notification or Scheduling a Local Notification with Custom Actions

To show the notification actions that you defined, categorized, and registered, you must push a remote notification or schedule a local notification. In the remote notification case, you need to include the category identifier in your push payload, as shown in Listing 2-10. Support for categories is a collaboration between your iOS app and your push notification server. When your push server wants to send a notification to a user, it can add a category key with an appropriate value to the notification's payload. When iOS sees a push notification with a category key, it looks up the categories that were registered by the app. If iOS finds a match, it displays the corresponding actions with the notification.

In iOS 8, the previous size limit of 256 bytes for a push payload has been increased to 2 kilobytes. See The Notification Payload (page 44) for details about the remote-notification payload.

Listing 2-10 Push payload including category identifier

```
{
    "aps" :  {
        "alert" : "You're invited!",
        "category" : "INVITE_CATEGORY",
    }
}
```

In the case of a local notification, you create the notification as usual, then set the category of the actions to be presented, and finally, schedule the notification as usual, as shown in Listing 2-11.

Listing 2-11 Defining a category of actions for a local notification

```
UILocalNotification *notification = [[UILocalNotification alloc] init];
. . .
notification.category = @"INVITE_CATEGORY";
[[UIApplication sharedApplication] scheduleLocalNotification:notification];
```

Handling Notification Actions

If your app is not running in the foreground, to handle the default action when a user just swipes or taps on a notification, iOS launches your app in the foreground and calls the `UIApplicationDelegate` method `application:didFinishLaunchingWithOptions:` passing in the local notification or the remote notification in the `options` dictionary. In the remote notification case, the system also calls `application:didReceiveRemoteNotification:fetchCompletionHandler:`.

If your app is already in the foreground, iOS does not show the notification. Instead, to handle the default action, it calls one of the `UIApplicationDelegate` methods `application:didReceiveLocalNotification:` or `application:didReceiveRemoteNotification:fetchCompletionHandler:`. (If you don't implement `application:didReceiveRemoteNotification:fetchCompletionHandler:`, iOS calls `application:didReceiveRemoteNotification:`.)

Finally, to handle the custom actions available in iOS 8, you need to implement at least one of two new methods on your app delegate, `application:handleActionWithIdentifier:forRemoteNotification:completionHandler:` or `application:handleActionWithIdentifier:forLocalNotification:completionHandler:`. In either case, you receive the action identifier, which you can use to determine what action was tapped. You also receive the notification, remote or local, which you can use to retrieve any information you need to handle that action. Finally, the system passes you the completion handler, which you must call when you finish handling the action. Listing 2-12 shows an example implementation that calls a self-defined action handler method.

Listing 2-12 Handling a custom notification action

```
- (void)application:(UIApplication *) application
              handleActionWithIdentifier: (NSString *) identifier
          // either forLocalNotification: (NSDictionary *) notification or
                    forRemoteNotification: (NSDictionary *) notification
                        completionHandler: (void (^)()) completionHandler {

    if ([identifier isEqualToString: @"ACCEPT_IDENTIFIER"]) {
```

```
        [self handleAcceptActionWithNotification:notification];
    }

    // Must be called when finished
    completionHandler();
}
```

Using Location-Based Notifications

In iOS 8 and later, you can send the user a notification whenever they arrive at a particular geographic location. This feature uses Core Location and is implemented through simple API additions to the `UILocalNotification` class. You define Core Location region objects and attach them to a notification so that the notification fires when the user comes near, enters, or exits a region. You can make it so that the notification is presented only the first time that the user enters this region, or you could have the notifications fire continuously if that makes sense for your app.

Registering for Location-Based Notifications

Before you can schedule a location-based notification, you must register with Core Location. To register, create a `CLLocationManager` instance and set your app as the delegate on this manager. The delegate receives callbacks that tell your app whether it is allowed to track the user's location. Finally, you must send the location manager instance a `requestWhenInUseAuthorization` message, as shown in Listing 2-13. The first time your app calls this method, it displays an alert that asks the user to allow or disallow your app's tracking of the user's whereabouts. In addition to asking the user for permission for your app to access their location, the alert also displays some explanatory text that you provide, such as "Enabling location tracking allows friends to see where you are." *This explanatory string is required to use location services.* Your app defines the string in its `Info.plist` file under the `NSLocationWhenInUseUsageDescription` key. If your app runs in locales with different languages, make sure you localize the string appropriately in your `Info.plist` strings file. If the user agrees to allow access, your app can track the user's location when your app is running in the foreground.

Listing 2-13 Getting authorization for tracking the user's location

```
CLLocationManager *locMan = [[CLLocationManager alloc] init];

// Set a delegate that receives callbacks that specify if your app is allowed to
track the user's location

locMan.delegate = self;
```

```
// Request authorization to track the user's location and enable location-based
notifications
[locMan requestWhenInUseAuthorization];
```

Note that users may see location-based notification alerts even when your app is in the background or suspended. However, an app does not receive any callbacks until users interact with the alert and the app is allowed to access their location.

Handling Core Location Callbacks

At startup, you should check the authorization status and store the state information you need to allow or disallow location-based notifications. The first delegate callback from the Core Location manager that you must handle is `locationManager:didChangeAuthorizationStatus:`, which reports changes to the authorization status. First, check that the status passed with the callback is `kCLAuthorizationStatusAuthorizedWhenInUse`, as shown in Listing 2-14, meaning that your app is authorized to track the user's location. Then you can begin scheduling location-based notifications.

Listing 2-14 Handling the Core Location authorization callback

```
- (void)locationManager:(CLLocationManager *)manager
               didChangeAuthorizationStatus:(CLAuthorizationStatus)status {


    // Check status to see if the app is authorized
    BOOL canUseLocationNotifications = (status ==
kCLAuthorizationStatusAuthorizedWhenInUse);


    if (canUseLocationNotifications) {
        [self startShowingLocationNotifications]; // Custom method defined below
    }
}
```

Listing 2-15 shows how to schedule a notification that triggers when the user enters a region. The first thing you must do, as with a local notification triggered by a date or a time, is to create an instance of `UILocalNotification` and define its type, in this case an alert.

Listing 2-15 Scheduling a location-based notification

```
- (void)startShowingNotifications {
```

```
    UILocalNotification *locNotification = [[UILocalNotification alloc] init];
    locNotification.alertBody = @"You have arrived!";
    locNotification.regionTriggersOnce = YES;

    locNotification.region = [[CLCircularRegion alloc]
                        initWithCenter:LOC_COORDINATE
                                radius:LOC_RADIUS
                            identifier:LOC_IDENTIFIER];

    [[UIApplication sharedApplication] scheduleLocalNotification:locNotification];
}
```

When the user enters the region defined in Listing 2-15, assuming the app isn't running in the foreground, the app displays an alert saying: "You have arrived!" The next line specifies that this notification triggers only once, the first time the user enters or exits this region. This is actually the default behavior, so it's superfluous to specify `YES`, but you could set this property to `NO` if that makes sense for your users and for your app.

Next, you create a `CLCircularRegion` instance and set it on the region property of the `UILocalNotification` instance. In this case we're giving it an app-defined location coordinate with some radius so that when the user enters this circle, this notification is triggered. This example uses a `CLCircularRegion` property, but you could also use `CLBeaconRegion` or any other type of `CLRegion` subclass.

Finally, call `scheduleLocalNotification:` on your `UIApplication` shared instance, passing this notification just like you would do for any other local user notification.

Handling Location-Based Notifications

Assuming that your app is suspended when the user enters the region defined in Listing 2-15 (page 31), an alert is displayed that says: "You have arrived." Your app can handle that local notification in the `application:didFinishLaunchingWithOptions:` app delegate method callback. Alternatively, if your app is executing in the foreground when the user enters that region, your app delegate is called back with `application:didReceiveLocalNotification:` message.

The logic for handling a location-based notification is very similar for both the `application:didFinishLaunchingWithOptions:` and `application:didReceiveLocalNotification:` methods. Both methods provide the notification, an instance of `UILocalNotification`, which has a `region`

property. If that property is not `nil`, then the notification is a location-based notification, and you can do whatever makes sense for your app. The example code in Listing 2-16 calls a hypothetical method of the app delegate named `tellFriendsUserArrivedAtRegion:`.

Listing 2-16 Handling a location-based notification

```
- (void)application:(UIApplication *)application
                        didReceiveLocalNotification: (UILocalNotification
*)notification {


    CLRegion *region = notification.region;


    if (region) {
            [self tellFriendsUserArrivedAtRegion:region];
    }
}
```

Finally, remember that the `application:didReceiveLocalNotification:` method is not called if the user disables Core Location, which they can do at any time in the Settings app under Privacy > Location Services.

Preparing Custom Alert Sounds

For remote notifications in iOS, you can specify a custom sound that iOS plays when it presents a local or remote notification for an app. The sound files must be in the main bundle of the client app.

Custom alert sounds are played by the iOS system-sound facility, so they must be in one of the following audio data formats:

- Linear PCM
- MA4 (IMA/ADPCM)
- µLaw
- aLaw

You can package the audio data in an `aiff`, `wav`, or `caf` file. Then, in Xcode, add the sound file to your project as a nonlocalized resource of the app bundle.

You may use the `afconvert` tool to convert sounds. For example, to convert the 16-bit linear PCM system sound `Submarine.aiff` to IMA4 audio in a CAF file, use the following command in the Terminal app:

```
afconvert /System/Library/Sounds/Submarine.aiff ~/Desktop/sub.caf –d ima4 –f caff
 –v
```

You can inspect a sound to determine its data format by opening it in QuickTime Player and choosing Show Movie Inspector from the Movie menu.

Custom sounds must be under 30 seconds when played. If a custom sound is over that limit, the default system sound is played instead.

Passing the Provider the Current Language Preference (Remote Notifications)

If an app doesn't use the `loc-key` and `loc-args` properties of the `aps` dictionary for client-side fetching of localized alert messages, the provider needs to localize the text of alert messages it puts in the notification payload. To do this, however, the provider needs to know the language that the device user has selected as the preferred language. (The user sets this preference in the General > International > Language view of the Settings app.) The client app should send its provider an identifier of the preferred language; this could be a canonicalized IETF BCP 47 language identifier such as "en" or "fr".

> **Note:** For more information about the `loc-key` and `loc-args` properties and client-side message localizations, see The Notification Payload (page 44).

Listing 2-17 illustrates a technique for obtaining the currently selected language and communicating it to the provider. In iOS, the array returned by the `preferredLanguages` property of `NSLocale` contains one object: an `NSString` object encapsulating the language code identifying the preferred language. The `UTF8String` coverts the string object to a C string encoded as UTF8.

Listing 2-17 Getting the current supported language and sending it to the provider

```
NSString *preferredLang = [[NSLocale preferredLanguages] objectAtIndex:0];
const char *langStr = [preferredLang UTF8String];
[self sendProviderCurrentLanguage:langStr]; // custom method
}
```

The app might send its provider the preferred language every time the user changes something in the current locale. To do this, you can listen for the notification named `NSCurrentLocaleDidChangeNotification` and, in your notification-handling method, get the code identifying the preferred language and send that to your provider.

If the preferred language is not one the app supports, the provider should localize the message text in a widely spoken fallback language such as English or Spanish.

Apple Push Notification Service

Apple Push Notification service (APNs for short) is the centerpiece of the remote notifications feature. It is a robust and highly efficient service for propagating information to iOS and OS X devices. Each device establishes an accredited and encrypted IP connection with the service and receives notifications over this persistent connection. If a notification for an app arrives when that app is not running, the device alerts the user that the app has data waiting for it.

Software developers ("providers") originate the notifications in their server software. The provider connects with APNs through a persistent and secure channel while monitoring incoming data intended for their client apps. When new data for an app arrives, the provider prepares and sends a notification through the channel to APNs, which pushes the notification to the target device.

In addition to being a simple but efficient and high-capacity transport service, APNs includes a default quality-of-service component that provides store-and-forward capabilities. See Quality of Service (page 38) for more information.

Provider Communication with Apple Push Notification Service (page 54) and Scheduling, Registering, and Handling Notifications (page 15) discuss the specific implementation requirements for providers and iOS apps, respectively.

A Remote Notification and Its Path

Apple Push Notification service transports and routes a remote notification from a given provider to a given device. A notification is a short message consisting of two major pieces of data: the device token and the payload. The device token is analogous to a phone number; it contains information that enables APNs to locate the device on which the client app is installed. APNs also uses it to authenticate the routing of a notification. The payload is a JSON-defined property list that specifies how the user of an app on a device is to be alerted.

> **Note:** For more information about the device token, see Security Architecture (page 38); for further information about the notification payload, see The Notification Payload (page 44).

The remote-notification data flows in one direction. The provider composes a notification package that includes the device token for a client app and the payload. The provider sends the notification to APNs which in turn pushes the notification to the device.

When a provider authenticates itself to APNs, it sends its topic to the APNs server, which identifies the app for which it's providing data. The topic is currently the bundle identifier of the target app.

Figure 3-1 Pushing a remote notification from a provider to a client app

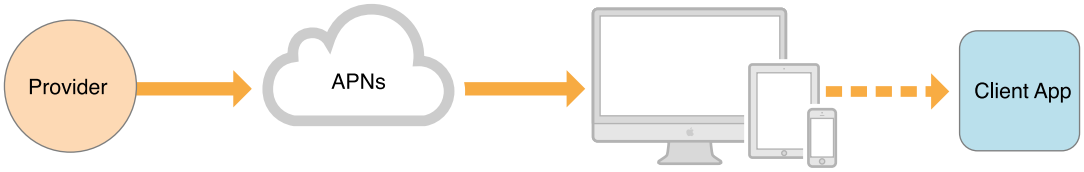
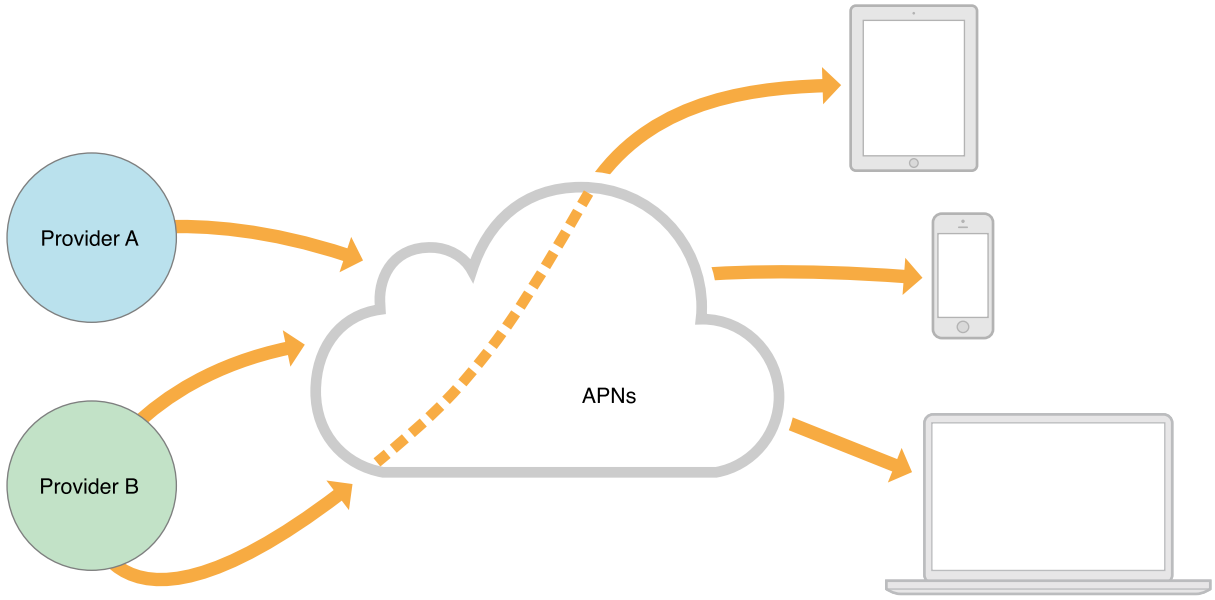


Figure 3-1 is a greatly simplified depiction of the virtual network APNs makes possible among providers and devices. The device-facing and provider-facing sides of APNs both have multiple points of connection; on the provider-facing side, these are called gateways. There are typically multiple providers, each making one or more persistent and secure connections with APNs through these gateways. And these providers are sending notifications through APNs to many devices on which their client apps are installed. Figure 3-2 is a slightly more realistic depiction.

Figure 3-2 Pushing remote notifications from multiple providers to multiple devices



The feedback service gives providers information about notifications that could not be delivered—for example, because the target app is no longer installed on that device. For more information, see The Feedback Service (page 58).

Quality of Service

Apple Push Notification service includes a default Quality of Service (QoS) component that performs a store-and-forward function.

If APNs attempts to deliver a notification but the device is offline, the notification is stored for a limited period of time, and delivered to the device when it becomes available.

Only one recent notification for a particular app is stored. If multiple notifications are sent while the device is offline, each new notification causes the prior notification to be discarded. This behavior of keeping only the newest notification is referred to as *coalescing* notifications.

If the device remains offline for a long time, any notifications that were being stored for it are discarded.

Security Architecture

To enable communication between a provider and a device, Apple Push Notification service must expose certain entry points to them. But then to ensure security, it must also regulate access to these entry points. For this purpose, APNs requires two different levels of trust for providers, devices, and their communications. These are known as connection trust and token trust.

Connection trust establishes certainty that, on one side, the APNs connection is with an authorized provider with whom Apple has agreed to deliver notifications. At the device side of the connection, APNs must validate that the connection is with a legitimate device.

After APNs has established trust at the entry points, it must then ensure that it conveys notifications to legitimate end points only. To do this, it must validate the routing of messages traveling through the transport; only the device that is the intended target of a notification should receive it.

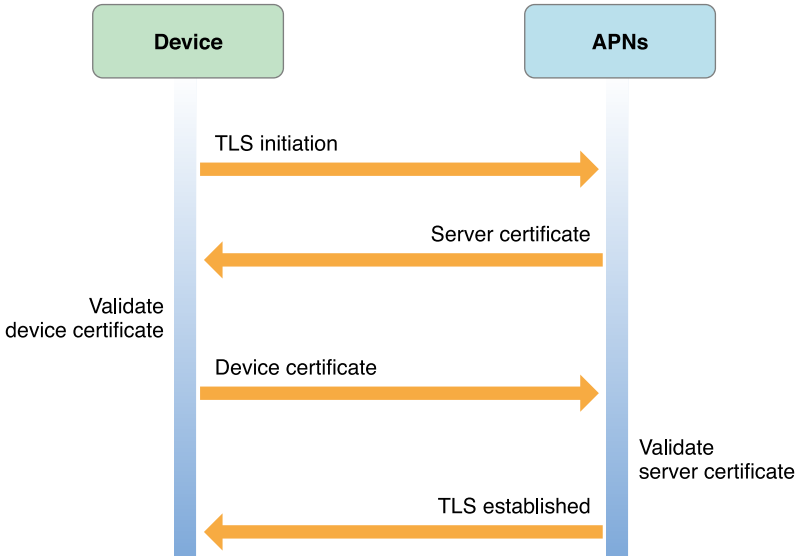
In APNs, assurance of accurate message routing—or *token trust*—is made possible through the device token. A device token is an opaque identifier of a device that APNs gives to the device when it first connects with it. The device shares the device token with its provider. Thereafter, this token accompanies each notification from the provider. It is the basis for establishing trust that the routing of a particular notification is legitimate.

> **Note:** A device token is not the same thing as the device UDID returned by the `identifierForVendor` or `uniqueIdentifier` property of `UIDevice` or any other similar properties such as the `advertisingIdentifier` property of `ASIdentifierManager`.

The following sections discuss the requisite components for connection trust and token trust as well as the four procedures for establishing trust.

Service-to-Device Connection Trust

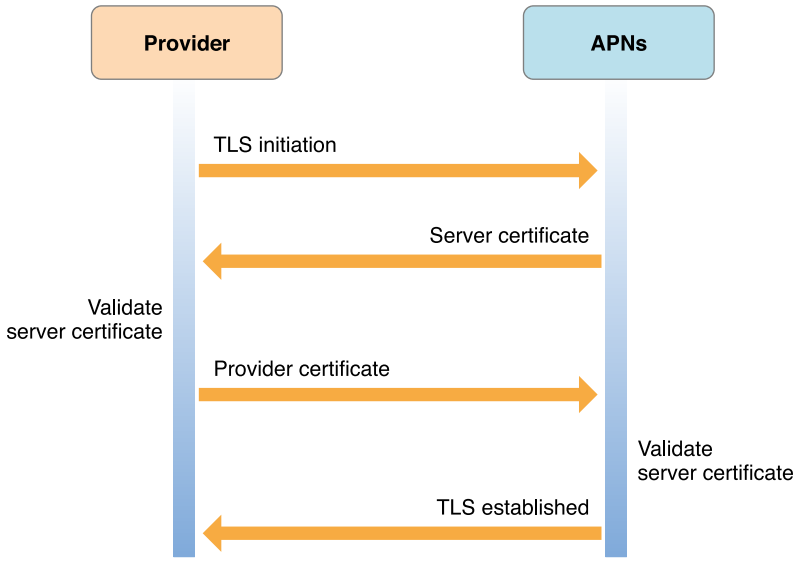
APNs establishes the identity of a connecting device through TLS peer-to-peer authentication. (Note that the system takes care of this stage of connection trust; you do not need to implement anything yourself.) In the course of this procedure, a device initiates a TLS connection with APNs, which returns its server certificate. The device validates this certificate and then sends its device certificate to APNs, which validates that certificate.



Provider-to-Service Connection Trust

Connection trust between a provider and APNs is also established through TLS peer-to-peer authentication. The procedure is similar to that described in Service-to-Device Connection Trust (page 39). The provider initiates a TLS connection, gets the server certificate from APNs, and validates that certificate. Then the provider

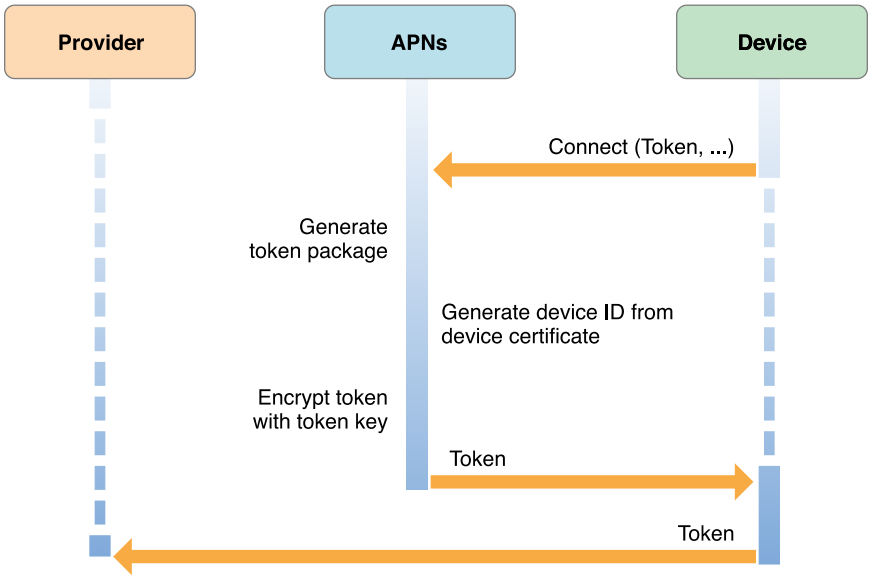
sends its provider certificate to APNs, which validates it on its end. Once this procedure is complete, a secure TLS connection has been established; APNs is now satisfied that the connection has been made by a legitimate provider.



Note that provider connection is valid for delivery to only one specific app, identified by the topic (bundle ID) specified in the certificate. APNs also maintains a certificate revocation list; if a provider's certificate is on this list, APNs may revoke provider trust (that is, refuse the connection).

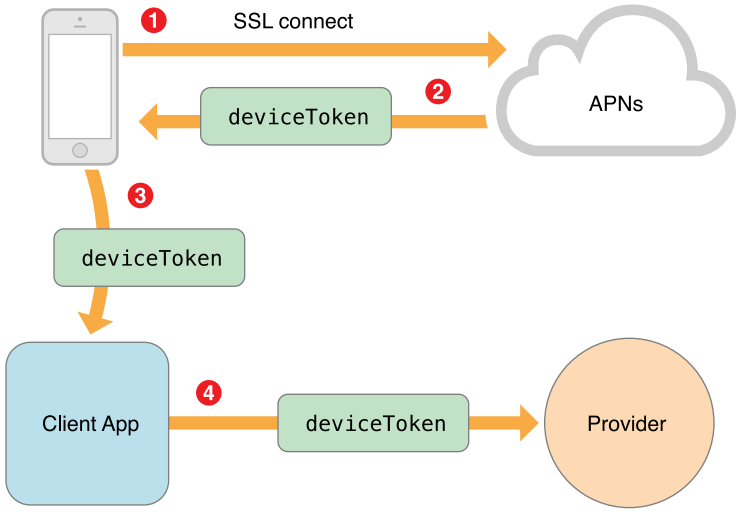
Token Generation and Dispersal

Apps must *register* to receive remote notifications; it typically does this right after it is installed on a device. (This procedure is described in Scheduling, Registering, and Handling Notifications (page 15).) The system receives the registration request from the app, connects with APNs, and forwards the request. APNs generates a device token using information contained in the unique device certificate. The device token contains an identifier of the device. It then encrypts the device token with a token key and returns it to the device.



The device returns the device token to the requesting app as an `NSData` object. The app must then deliver the device token to its provider in either binary or hexadecimal format. Figure 3-3 also illustrates the token generation and dispersal sequence, but in addition shows the role of the client app in furnishing its provider with the device token.

Figure 3-3 Sharing the device token

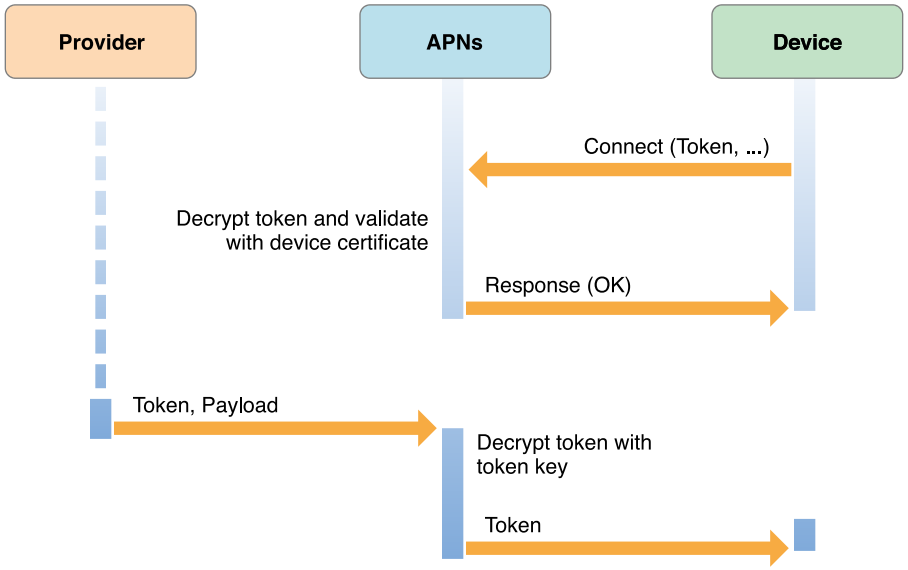


The form of this phase of token trust ensures that only APNs generates the token which it will later honor, and it can assure itself that a token handed to it by a device is the same token that it previously provisioned for that particular device—and only for that device.

Token Trust (Notification)

After the system obtains a device token from APNs, as described in Token Generation and Dispersal (page 41), it must provide APNs with the token every time it connects with it. APNs decrypts the device token and validates that the token was generated for the connecting device. To validate, APNs ensures that the device identifier contained in the token matches the device identifier in the device certificate.

Every notification that a provider sends to APNs for delivery to a device must be accompanied by the device token it obtained from an app on that device. APNs decrypts the token using the token key, thereby ensuring that the notification is valid. It then uses the device ID contained in the device token to determine the destination device for the notification.



Trust Components

To support the security model for APNs, providers and devices must possess certain certificates, certificate authority (CA) certificates, or tokens.

- **Provider**: Each provider requires a unique provider certificate and private cryptographic key for validating their connection with APNs. This certificate, provisioned by Apple, must identify the particular topic published by the provider; the topic is the bundle ID of the client app. For each notification, the provider must furnish APNs with a device token identifying the target device. The provider may optionally wish to validate the service it is connecting to using the public server certificate provided by the APNs server.
- **Device**: The system uses the public server certificate passed to it by APNs to authenticate the service that it has connected to. It has a unique private key and certificate that it uses to authenticate itself to the service and establish the TLS connection. It obtains the device certificate and key during device activation and stores them in the keychain. The system also holds its particular device token, which it receives during the service connection process. Each registered client app is responsible for delivering this token to its content provider.

APNs servers also have the necessary certificates, CA certificates, and cryptographic keys (private and public) for validating connections and the identities of providers and devices.

The Notification Payload

Each remote notification includes a payload. The payload contains information about how the system should alert the user as well as any custom data you provide. In iOS 8 and later, the maximum size allowed for a notification payload is 2 kilobytes; Apple Push Notification service refuses any notification that exceeds this limit. (Prior to iOS 8 and in OS X, the maximum payload size is 256 bytes.)

For each notification, compose a JSON dictionary object (as defined by RFC 4627). This dictionary must contain another dictionary identified by the key `aps`. The `aps` dictionary can contain one or more properties that specify the following user notification types:

- An alert message to display to the user
- A number to badge the app icon with
- A sound to play

The `aps` dictionary can also contain the `content-available` property. The `content-available` property with a value of 1 lets the remote notification act as a "silent" notification. When a silent notification arrives, iOS wakes up your app in the background so that you can get new data from your server or do background information processing. Users aren't told about the new or changed information that results from a silent notification, but they can find out about it the next time they open your app.

To support silent remote notifications, add the `remote-notification` value to the `UIBackgroundModes` array in your `Info.plist` file. To learn more about this array, see UIBackgroundModes.

If the target app isn't running when the notification arrives, the alert message, sound, or badge value is played or shown. If the app is running, the system delivers the notification to the app delegate as an `NSDictionary` object. The dictionary contains the corresponding Cocoa property-list objects (plus `NSNull`).

Providers can specify custom payload values outside the Apple-reserved `aps` namespace. Custom values must use the JSON structured and primitive types: dictionary (object), array, string, number, and Boolean. You should not include customer information (or any sensitive data) as custom payload data. Instead, use it for such purposes as setting context (for the user interface) or internal metrics. For example, a custom payload value might be a conversation identifier for use by an instant-message client app or a timestamp identifying when the provider sent the notification. Any action associated with an alert message should not be destructive—for example, it should not delete data on the device.

> **Important:** Delivery of notifications is a "best effort", not guaranteed. It is not intended to deliver data to your app, only to *notify* the user that there is new data available.

Table 3-1 lists the keys and expected values of the `aps` payload.

Table 3-1 Keys and values of the `aps` dictionary

Key	Value type	Comment
`alert`	string or dictionary	If this property is included, the system displays a standard alert. You may specify a string as the value of `alert` or a dictionary as its value. If you specify a string, it becomes the message text of an alert with two buttons: Close and View. If the user taps View, the app is launched. Alternatively, you can specify a dictionary as the value of `alert`. See Table 3-2 (page 45) for descriptions of the keys of this dictionary.
`badge`	number	The number to display as the badge of the app icon. If this property is absent, the badge is not changed. To remove the badge, set the value of this property to `0`.
`sound`	string	The name of a sound file in the app bundle. The sound in this file is played as an alert. If the sound file doesn't exist or `default` is specified as the value, the default alert sound is played. The audio must be in one of the audio data formats that are compatible with system sounds; see Preparing Custom Alert Sounds (page 33) for details.
`content-available`	number	Provide this key with a value of `1` to indicate that new content is available. Including this key and value means that when your app is launched in the background or resumed, `application: didReceiveRemoteNotification: fetchCompletionHandler:` is called. (Newsstand apps are guaranteed to be able to receive at least one push with this key per 24-hour window.)

Table 3-2 lists the keys and expected values for the `alert` dictionary.

Table 3-2 Child properties of the `alert` property

Key	Value type	Comment
`title`	string	A short string describing the purpose of the notification. Apple Watch displays this string as part of the notification interface. This string is displayed only briefly and should be crafted so that it can be understood quickly. This key was added in iOS 8.2.

Key	Value type	Comment
`body`	string	The text of the alert message.
`title-loc-key`	string or `null`	The key to a title string in the `Localizable.strings` file for the current localization. The key string can be formatted with `%@` and `%n$@` specifiers to take the variables specified in the `title-loc-args` array. See Localized Formatted Strings (page 47) for more information. This key was added in iOS 8.2.
`title-loc-args`	array of strings or `null`	Variable string values to appear in place of the format specifiers in `title-loc-key`. See Localized Formatted Strings (page 47) for more information. This key was added in iOS 8.2.
`action-loc-key`	string or `null`	If a string is specified, the system displays an alert that includes the Close and View buttons. The string is used as a key to get a localized string in the current localization to use for the right button's title instead of "View". See Localized Formatted Strings (page 47) for more information.
`loc-key`	string	A key to an alert-message string in a `Localizable.strings` file for the current localization (which is set by the user's language preference). The key string can be formatted with `%@` and `%n$@` specifiers to take the variables specified in the `loc-args` array. See Localized Formatted Strings (page 47) for more information.
`loc-args`	array of strings	Variable string values to appear in place of the format specifiers in `loc-key`. See Localized Formatted Strings (page 47) for more information.
`launch-image`	string	The filename of an image file in the app bundle; it may include the extension or omit it. The image is used as the launch image when users tap the action button or move the action slider. If this property is not specified, the system either uses the previous snapshot,uses the image identified by the `UILaunchImageFile` key in the app's `Info.plist` file, or falls back to `Default.png`. This property was added in iOS 4.0.

Note: If you want the device to display the message text as-is in an alert that has both the Close and View buttons, then specify a string as the direct value of alert. *Don't* specify a dictionary as the value of `alert` if the dictionary only has the `body` property.

Localized Formatted Strings

You can display localized alert messages in two ways:

- The server originating the notification can localize the text; to do this, it must discover the current language preference selected for the device (see Passing the Provider the Current Language Preference (Remote Notifications) (page 34)).
- The client app can store the alert-message strings in its bundle, translated for each localization it supports. The provider includes the `loc-key` and `loc-args` keys in the `aps` dictionary of the notification payload. (For title strings, it includes the `title-loc-key` and `title-loc-args` keys in the aps dictionary.) When the device receives the notification (assuming the app isn't running), it uses these `aps`-dictionary properties to find and format the string localized for the current language, which it then displays to the user.

Here's how that second option works in a little more detail.

An app can internationalize resources such as images, sounds, and text for each language that it supports, Internationalization collects the resources and puts them in a subdirectory of the bundle with a two-part name: a language code and an extension of `.lproj` (for example, `fr.lproj`). Localized strings that are programmatically displayed are put in a file called `Localizable.strings`. Each entry in this file has a key and a localized string value; the string can have format specifiers for the substitution of variable values. When an app asks for a particular resource—say a localized string—it gets the resource that is localized for the language currently selected by the user. For example, if the preferred language is French, the corresponding string value for an alert message would be fetched from `Localizable.strings` in the `fr.lproj` directory in the app bundle. (The app makes this request through the `NSLocalizedString` macro.)

Note: This general pattern is also followed when the value of the `action-loc-key` property is a string. This string is a key into the `Localizable.strings` in the localization directory for the currently selected language. iOS uses this key to get the title of the button on the right side of an alert message (the "action" button).

To make this clearer, let's consider an example. The provider specifies the following dictionary as the value of the alert property:

```
"alert" : {
    "loc-key" : "GAME_PLAY_REQUEST_FORMAT",
```

```
"loc-args" : [ "Jenna", "Frank"]
}
```

When the device receives the notification, it uses `"GAME_PLAY_REQUEST_FORMAT"` as a key to look up the associated string value in the `Localizable.strings` file in the `.lproj` directory for the current language. Assuming the current localization has a `Localizable.strings` entry such as this:

```
"GAME_PLAY_REQUEST_FORMAT" = "%@ and %@ have invited you to play Monopoly";
```

the device displays an alert with the message "Jenna and Frank have invited you to play Monopoly".

In addition to the format specifier `%@`, you can `%`*n*`$@` format specifiers for positional substitution of string variables. The *n* is the index (starting with 1) of the array value in `loc-args` to substitute. (There's also the `%%` specifier for expressing a percentage sign (%).) So if the entry in `Localizable.strings` is this:

```
"GAME_PLAY_REQUEST_FORMAT" = "%2$@ and %1$@ have invited you to play Monopoly";
```

the device displays an alert with the message "Frank and Jenna have invited you to play Monopoly".

For a full example of a notification payload that uses the `loc-key` and `loc-arg` properties, see Examples of JSON Payloads. To learn more about internationalization, see *Internationalization and Localization Guide* . String formatting is discussed in Formatting String Objects in *String Programming Guide* .

> **Note:** You should use the `loc-key` and `loc-args` properties—and the `alert` dictionary in general—only if you absolutely need to. The values of these properties, especially if they are long strings, might use up more bandwidth than is good for performance. Many apps don't need these properties because their message strings are originated by users.

Examples of JSON Payloads

The following examples of the payload portion of notifications illustrate the practical use of the properties listed in Table 3-1. Properties with "acme" in the key name are examples of custom payload data.

Note: The examples are formatted with whitespace and line breaks for readability. In practice, omit whitespace and line breaks to reduce the size of the payload, improving network performance.

Example 1. The following payload has an `aps` dictionary with a simple, recommended form for alert messages with the default alert buttons (Close and View). It uses a string as the value of `alert` rather than a dictionary. This payload also has a custom array property.

```
{
    "aps" : { "alert" : "Message received from Bob" },
    "acme2" : [ "bang",  "whiz" ]
}
```

Example 2. The payload in the example uses an `aps` dictionary to request that the device display an alert message with a Close button on the left and a localized title for the "action" button on the right side of the alert. In this case, "PLAY" is used as a key into the `Localizable.strings` file for the currently selected language to get the localized equivalent of "Play". The `aps` dictionary also requests that the app icon be badged with the number 5. On Apple Watch, the title key alerts the user to the new request.

```
{
    "aps" : {
        "alert" : {
            "title" : "Game Request",
            "body" : "Bob wants to play poker",
            "action-loc-key" : "PLAY"
        },
        "badge" : 5,
    },
    "acme1" : "bar",
    "acme2" : [ "bang",  "whiz" ]
}
```

Example 3. The payload in this example specifies that the device should display an alert message with both Close and View buttons. It also requests that the app icon be badged with the number 9 and that a bundled alert sound be played when the notification is delivered.

```
{
```

```
    "aps" : {
        "alert" : "You got your emails.",
        "badge" : 9,
        "sound" : "bingbong.aiff"
    },
    "acme1" : "bar",
    "acme2" : 42
}
```

Example 4. The payload in this example uses the `loc-key` and `loc-args` child properties of the `alert` dictionary to fetch a formatted localized string from the app's bundle and substitute the variable string values (`loc-args`) in the appropriate places. It also specifies a custom sound and includes a custom property.

```
{
    "aps" : {
        "alert" : {
            "loc-key" : "GAME_PLAY_REQUEST_FORMAT",
            "loc-args" : [ "Jenna", "Frank"]
        },
        "sound" : "chime.aiff"
    },
    "acme" : "foo"
}
```

Example 5. The payload in this example includes custom notification actions. Note that the presence of the additional `actions` array does not affect the default action associated with the alert.

```
{
   "aps" : {
      "alert" : {
         “body” : "Acme message received from Johnny Appleseed”,
         “action-loc-key” : “VIEW”,
         "actions" : [
            {
               “id" : “delete",
```

```
                "title" : "Delete"
            },
            {
                “id" : “reply-to”,
                "loc-key" : “REPLYTO”,
                "loc-args" : [“Jane"]
            }
         ]
      }
      "badge" : 3,
      "sound" : “chime.aiff"
   },
   "acme-account" : "jane.appleseed@apple.com",
   "acme-message" : "message123456"
}
```

Provisioning and Development

Development and Production Environments

To develop and deploy the provider side of a client/server app, you must get SSL certificates from Member Center. Each certificate is limited to a single app, identified by its bundle ID. Each certificate is also limited to one of two development environments, each with its own assigned hostname:

- **Development**: Use the development environment for initial development and testing of the provider app. It provides the same set of services as the production environment, although with a smaller number of server units. The development environment also acts as a virtual device, enabling simulated end-to-end testing.

  You access the development environment at `gateway.sandbox.push.apple.com`, outbound TCP port 2195.

- **Production**: Use the production environment when building the production version of the provider app. Apps using the production environment must meet Apple's reliability requirements.

  You access the production environment at `gateway.push.apple.com`, outbound TCP port 2195.

You must get separate certificates for the development environment and the production environment. The certificates are associated with an identifier of the app that is the recipient of remote notifications; this identifier includes the app's bundle ID. When you create a provisioning profile for one of the environments, the requisite entitlements are automatically added to the profile, including the entitlement specific to remote notifications, `<aps-environment>`. The two provisioning profiles are called Development and Distribution. The Distribution provisioning profile is a requirement for submitting your app to the App Store.

> **OS X Note:** The entitlement for the OS X provisioning profile is `com.apple.developer.aps-environment`, which scopes it to the platform.

You can determine in Xcode which environment you are in by the selection of a code-signing identity. If you see an "iPhone Developer: *Firstname Lastname*" certificate/provisioning profile pair, you are in the development environment. If you see an "iPhone Distribution: *Companyname*" certificate/provisioning profile pair, you are in the production environment. It is a good idea to create a Distribution release configuration in Xcode to help you further differentiate the environments.

Although an SSL certificate is not put into a provisioning profile, the `<aps-environment>` is added to the profile because of the association of the certificate and a particular App ID. As a result this entitlement is built into the app, which enables it to receive remote notifications.

Provisioning Procedures

Apple Push Notification service (APNs) is available only to apps distributed through the iOS App Store or Mac App Store. Your app must be provisioned and code signed to use app services. If you are a company, most of these configuration steps can be performed only by a team agent or admin.

To learn how to code sign and provision your app during development, read *App Distribution Quick Start*. For how to enable APNs, read Configuring Push Notifications in *App Distribution Guide*.

However, APNs is not fully enabled until you create the client SSL certificates in Member Center, as described in Creating Push Notification Client SSL Certificates. Create a development client SSL certificate if you are developing and testing your app. The signing identities for client SSL certificates with the private keys are stored in your keychain. To export a client SSL signing identity and install it on your server, read Installing Client SSL Certificates.

When you are ready to create your production client SSL certificate, follow the same steps in Creating Push Notification Client SSL Certificates but select the SSL certificate under Production, not Development. You must create the production client SSL certificate before you submit your app to the store.

After you configure your app to use APNs, Xcode automatically creates the necessary distribution provisioning profiles when you export your iOS app for beta testing or submit your app to the store.

Provider Communication with Apple Push Notification Service

This chapter describes the interfaces that providers use for communication with Apple Push Notification service (APNs) and discusses some of the functions that providers are expected to fulfill.

General Provider Requirements

As a provider you communicate with Apple Push Notification service over a binary interface. This interface is a high-speed, high-capacity interface for providers; it uses a streaming TCP socket design in conjunction with binary content. The binary interface is asynchronous.

The binary interface of the production environment is available through `gateway.push.apple.com`, port 2195; the binary interface of the development environment is available through `gateway.sandbox.push.apple.com`, port 2195.

For each interface, use TLS (or SSL) to establish a secured communications channel. The SSL certificate required for these connections is obtained from Member Center. (See Provisioning and Development (page 52) for details.) To establish a trusted provider identity, present this certificate to APNs at connection time using peer-to-peer authentication.

> **Note:** To establish a TLS session with APNs, an Entrust Secure CA root certificate must be installed on the provider's server. If the server is running OS X, this root certificate is already in the keychain. On other systems, the certificate might not be available. You can download this certificate from the Entrust SSL Certificates website.

As a provider, you are responsible for the following aspects of remote notifications:

- You must compose the notification payload (see The Notification Payload (page 44)).
- You are responsible for supplying the badge number to be displayed on the app icon.
- Connect regularly with the feedback service and fetch the current list of those devices that have repeatedly reported failed-delivery attempts. Then stop sending notifications to the devices associated with those apps. See The Feedback Service (page 58) for more information.

If you intend to support notification messages in multiple languages, but do not use the `loc-key` and `loc-args` properties of the `aps` payload dictionary for client-side fetching of localized alert strings, you need to localize the text of alert messages on the server side. To do this, you need to find out the current language preference from the client app. Scheduling, Registering, and Handling Notifications (page 15) suggests an approach for obtaining this information. See The Notification Payload (page 44) for information about the `loc-key` and `loc-args` properties.

Best Practices for Managing Connections

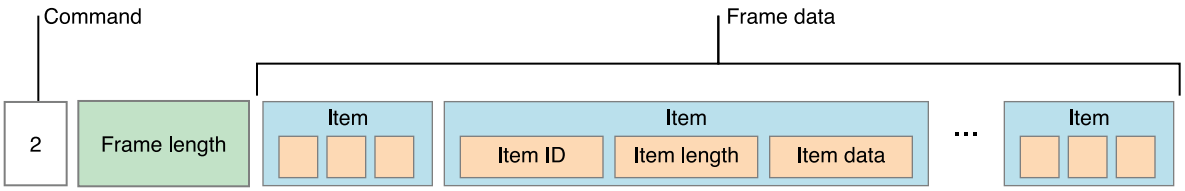
You may establish multiple connections to the same gateway or to multiple gateway instances. If you need to send a large number of remote notifications, spread them out over connections to several different gateways. This improves performance compared to using a single connection: it lets you send the remote notifications faster, and it lets APNs deliver them faster.

Keep your connections with APNs open across multiple notifications; don't repeatedly open and close connections. APNs treats rapid connection and disconnection as a denial-of-service attack. You should leave a connection open unless you know it will be idle for an extended period of time—for example, if you only send notifications to your users once a day it is ok to use a new connection each day.

The Binary Interface and Notification Format

The binary interface employs a plain TCP socket for binary content that is streaming in nature. For optimum performance, batch multiple notifications in a single transmission over the interface, either explicitly or using a TCP/IP Nagle algorithm. The format of notifications is shown in Figure 5-1.

Figure 5-1 Notification format



Note: All data is specified in network order, that is big endian.

The top level of the notification format is made up of the following, in order:

Field name	Length	Discussion
Command	1 byte	Populate with the number 2.
Frame length	4 bytes	The size of the frame data.
Frame data	variable length	The frame contains the body, structured as a series of items.

The frame data is made up of a series of items. Each item is made up of the following, in order:

Field name	Length	Discussion
Item ID	1 byte	The item identifier. For example, the item number of the payload is 2.
Item data length	2 bytes	The size of the item data.
Item data	variable length	The value for the item.

The items and their identifiers are are as follows:

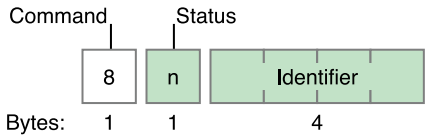
Item ID	Item Name	Length	Data
1	Device token	32 bytes	The device token in binary form, as was registered by the device.
2	Payload	variable length, less than or equal to 2 kilobytes	The JSON-formatted payload. The payload must not be null-terminated.
3	Notification identifier	4 bytes	An arbitrary, opaque value that identifies this notification. This identifier is used for reporting errors to your server.
4	Expiration date	4 bytes	A UNIX epoch date expressed in seconds (UTC) that identifies when the notification is no longer valid and can be discarded. If this value is non-zero, APNs stores the notification tries to deliver the notification at least once. Specify zero to indicate that the notification expires immediately and that APNs should not store the notification at all.

Item ID	Item Name	Length	Data
5	Priority	1 byte	The notification's priority. Provide one of the following values: • `10` The push message is sent immediately. The remote notification must trigger an alert, sound, or badge on the device. It is an error to use this priority for a push that contains only the `content-available` key. • `5` The push message is sent at a time that conserves power on the device receiving it.

If you send a notification that is accepted by APNs, nothing is returned.

If you send a notification that is malformed or otherwise unintelligible, APNs returns an error-response packet and closes the connection. Any notifications that you sent after the malformed notification using the same connection are discarded, and must be resent. Figure 5-2 shows the format of the error-response packet.

Figure 5-2 Format of error-response packet



The packet has a command value of 8 followed by a one-byte status code and the notification identifier of the malformed notification. Table 5-1 lists the possible status codes and their meanings.

Table 5-1 Codes in error-response packet

Status code	Description
0	No errors encountered
1	Processing error
2	Missing device token
3	Missing topic
4	Missing payload

Status code	Description
5	Invalid token size
6	Invalid topic size
7	Invalid payload size
8	Invalid token
10	Shutdown
255	None (unknown)

A status code of 10 indicates that the APNs server closed the connection (for example, to perform maintenance). The notification identifier in the error response indicates the last notification that was successfully sent. Any notifications you sent after it have been discarded and must be resent. When you receive this status code, stop using this connection and open a new connection.

Take note that the device token in the production environment and the device token in the development environment are not the same value.

The Feedback Service

The Apple Push Notification service includes a feedback service to give you information about failed remote notifications. When a remote notification cannot be delivered because the intended app does not exist on the device, the feedback service adds that device's token to its list. Remote notifications that expire before being delivered are not considered a failed delivery and don't impact the feedback service. By using this information to stop sending remote notifications that will fail to be delivered, you reduce unnecessary message overhead and improve overall system performance.

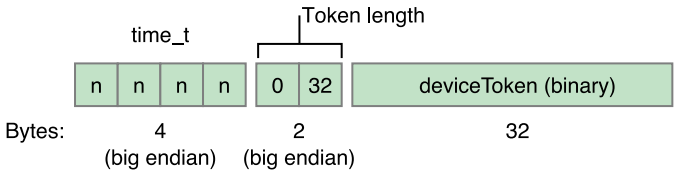
Query the feedback service daily to get the list of device tokens. Use the timestamp to verify that the device tokens haven't been reregistered since the feedback entry was generated. For each device that has not been reregistered, stop sending notifications. APNs monitors providers for their diligence in checking the feedback service and refraining from sending remote notifications to nonexistent apps on devices.

> **Note:** The feedback service maintains a separate list for each push topic. If you have multiple apps, you must connect to the feedback service once for each app, using the corresponding certificate, in order to receive all feedback.

The feedback service has a binary interface similar to the interface used for sending remote notifications. You access the production feedback service via `feedback.push.apple.com` on port 2196 and the development feedback service via `feedback.sandbox.push.apple.com` on port 2196. As with the binary interface for remote notifications, use TLS (or SSL) to establish a secured communications channel. You use the same SSL certificate for connecting to the feedback service as you use for sending notifications. To establish a trusted provider identity, present this certificate to APNs at connection time using peer-to-peer authentication.

Once you are connected, transmission begins immediately; you do not need to send any command to APNs. Read the stream from the feedback service until there is no more data to read. The received data is in tuples with the following format:

Figure 5-3 Binary format of a feedback tuple



Timestamp	A timestamp (as a four-byte `time_t` value) indicating when APNs determined that the app no longer exists on the device. This value, which is in network order, represents the seconds since 12:00 midnight on January 1, 1970 UTC.
Token length	The length of the device token as a two-byte integer value in network order.
Device token	The device token in binary format.

The feedback service's list is cleared after you read it. Each time you connect to the feedback service, the information it returns lists only the failures that have happened since you last connected.

Legacy Information

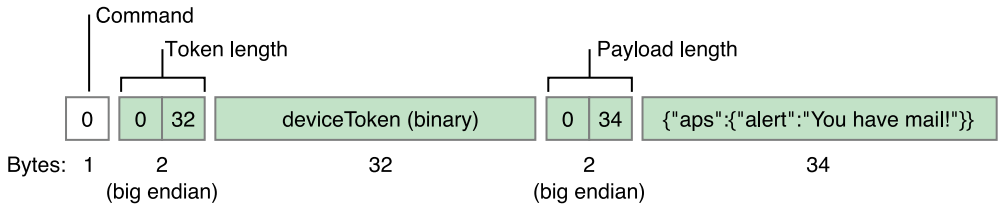
New development should use the modern format to connect to APNs, as described in The Binary Interface and Notification Format (page 55).

These formats do not include a priority; a priority of `10` is assumed.

Simple Notification Format

Figure A-1 shows this format.

Figure A-1 Simple notification format



The first byte in the legacy format is a command value of 0 (zero). The other fields are the same as the enhanced format. Listing A-1 gives an example of a function that sends a remote notification to APNs over the binary interface using the simple notification format. The example assumes prior SSL connection to `gateway.push.apple.com` (or `gateway.sandbox.push.apple.com`) and peer-exchange authentication.

Listing A-1 Sending a notification in the simple format via the binary interface

```
static bool sendPayload(SSL *sslPtr, char *deviceTokenBinary, char *payloadBuff,
size_t payloadLength)
{
    bool rtn = false;
    if (sslPtr && deviceTokenBinary && payloadBuff && payloadLength)
    {
        uint8_t command = 0; /* command number */
        char binaryMessageBuff[sizeof(uint8_t) + sizeof(uint16_t) +
            DEVICE_BINARY_SIZE + sizeof(uint16_t) + MAXPAYLOAD_SIZE];
```

```
        /* message format is, |COMMAND|TOKENLEN|TOKEN|PAYLOADLEN|PAYLOAD| */
        char *binaryMessagePt = binaryMessageBuff;
        uint16_t networkOrderTokenLength = htons(DEVICE_BINARY_SIZE);
        uint16_t networkOrderPayloadLength = htons(payloadLength);

        /* command */
        *binaryMessagePt++ = command;

       /* token length network order */
        memcpy(binaryMessagePt, &networkOrderTokenLength, sizeof(uint16_t));
        binaryMessagePt += sizeof(uint16_t);

        /* device token */
        memcpy(binaryMessagePt, deviceTokenBinary, DEVICE_BINARY_SIZE);
        binaryMessagePt += DEVICE_BINARY_SIZE;

        /* payload length network order */
        memcpy(binaryMessagePt, &networkOrderPayloadLength, sizeof(uint16_t));
        binaryMessagePt += sizeof(uint16_t);

        /* payload */
        memcpy(binaryMessagePt, payloadBuff, payloadLength);
        binaryMessagePt += payloadLength;
        if (SSL_write(sslPtr, binaryMessageBuff, (binaryMessagePt -
binaryMessageBuff)) > 0)
            rtn = true;
    }
    return rtn;
}
```

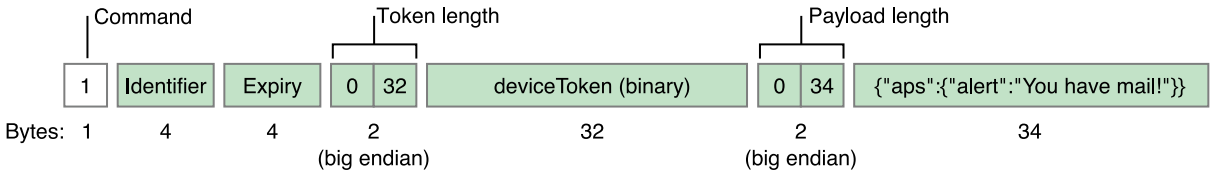
Enhanced Notification Format

The enhanced format has several improvements over the simple format:

- **Error response**. With the simple format, if you send a notification packet that is malformed in some way—for example, the payload exceeds the stipulated limit—APNs responds by severing the connection. It gives no indication why it rejected the notification. The enhanced format lets a provider tag a notification with an arbitrary identifier. If there is an error, APNs returns a packet that associates an error code with the identifier. This response enables the provider to locate and correct the malformed notification.
- **Notification expiration**. APNs has a store-and-forward feature that keeps the most recent notification sent to an app on a device. If the device is offline at time of delivery, APNs delivers the notification when the device next comes online. With the simple format, the notification is delivered regardless of the pertinence of the notification. In other words, the notification can become "stale" over time. The enhanced format includes an expiry value that indicates the period of validity for a notification. APNs discards a notification in store-and-forward when this period expires.

Figure A-2 depicts the format for notification packets.

Figure A-2 Enhanced notification format



The first byte in the notification format is a command value of 1. The remaining fields are as follows:

- **Identifier**—An arbitrary value that identifies this notification. This same identifier is returned in a error-response packet if APNs cannot interpret a notification.
- **Expiry**—A fixed UNIX epoch date expressed in seconds (UTC) that identifies when the notification is no longer valid and can be discarded. The expiry value uses network byte order (big endian). If the expiry value is non-zero, APNs tries to deliver the notification at least once. Specify zero to request that APNs not store the notification at all.
- **Token length**—The length of the device token in network order (that is, big endian)
- **Device token**—The device token in binary form.
- **Payload length**—The length of the payload in network order (that is, big endian). The payload must not exceed 256 bytes and must *not* be null-terminated.
- **Payload**—The notification payload.

Listing A-2 composes a remote notification in the enhanced format before sending it to APNs. It assumes prior SSL connection to `gateway.push.apple.com` (or `gateway.sandbox.push.apple.com`) and peer-exchange authentication.

Listing A-2 Sending a notification in the enhanced format via the binary interface

```
static bool sendPayload(SSL *sslPtr, char *deviceTokenBinary, char *payloadBuff,
size_t payloadLength)

{

  bool rtn = false;

  if (sslPtr && deviceTokenBinary && payloadBuff && payloadLength)

  {

      uint8_t command = 1; /* command number */

     char binaryMessageBuff[sizeof(uint8_t) + sizeof(uint32_t) + sizeof(uint32_t)
 + sizeof(uint16_t) +

          DEVICE_BINARY_SIZE + sizeof(uint16_t) + MAXPAYLOAD_SIZE];

      /* message format is, |COMMAND|ID|EXPIRY|TOKENLEN|TOKEN|PAYLOADLEN|PAYLOAD|
 */

      char *binaryMessagePt = binaryMessageBuff;

      uint32_t whicheverOrderIWantToGetBackInAErrorResponse_ID = 1234;

      uint32_t networkOrderExpiryEpochUTC = htonl(time(NULL)+86400); // expire
message if not delivered in 1 day

      uint16_t networkOrderTokenLength = htons(DEVICE_BINARY_SIZE);

      uint16_t networkOrderPayloadLength = htons(payloadLength);


      /* command */

      *binaryMessagePt++ = command;


     /* provider preference ordered ID */

     memcpy(binaryMessagePt, &whicheverOrderIWantToGetBackInAErrorResponse_ID,
sizeof(uint32_t));

     binaryMessagePt += sizeof(uint32_t);


     /* expiry date network order */

     memcpy(binaryMessagePt, &networkOrderExpiryEpochUTC, sizeof(uint32_t));

     binaryMessagePt += sizeof(uint32_t);


     /* token length network order */

      memcpy(binaryMessagePt, &networkOrderTokenLength, sizeof(uint16_t));

      binaryMessagePt += sizeof(uint16_t);


      /* device token */
```

```
      memcpy(binaryMessagePt, deviceTokenBinary, DEVICE_BINARY_SIZE);
      binaryMessagePt += DEVICE_BINARY_SIZE;

      /* payload length network order */
      memcpy(binaryMessagePt, &networkOrderPayloadLength, sizeof(uint16_t));
      binaryMessagePt += sizeof(uint16_t);

      /* payload */
      memcpy(binaryMessagePt, payloadBuff, payloadLength);
      binaryMessagePt += payloadLength;
     if (SSL_write(sslPtr, binaryMessageBuff, (binaryMessagePt - binaryMessageBuff))
 > 0)
          rtn = true;
  }
  return rtn;
}
```

Document Revision History

This table describes the changes to *Local and Remote Notification Programming Guide*.

Date	Notes
2015-03-09	Added information about users changing their notification settings in the Settings app.
2014-10-31	Made minor corrections. Made minor corrections.
2014-10-16	Added information about custom notification actions and location-based notifications. Removed note stating that not all notification types are available in OS X—all types are delivered. Changed title from *Local and Push Notification Programming Guide*.
2014-09-17	Added note about content-available key. Added information about notification custom actions and location-based notifications. Changed title from *Local and Push Notification Programming Guide*.
2014-02-11	Added note about content-available key.
2013-09-18	Added information about setting priority for a push notification.
2013-04-23	Updated chapter "Provider Communication with Apple Push Notification Service". Minor changes throughout other chapters. Added section Best Practices for Managing Connections (page 55). Added error code 10 to Table 5-1 (page 57). Expanded discussion in The Feedback Service (page 58). Moved discussion of a legacy protocol to an appendix.
2011-08-09	Added information about implementing push notifications on an OS X desktop client. Unified the guide for iOS and OS X.

Date	Notes
2010-08-03	Describes how to determine if an application is launched because the user tapped the notification alert's action button.
2010-07-08	Changed occurrences of "iPhone OS" to "iOS."
2010-05-27	Updated and reorganized to describe local notifications, a feature introduced in iOS 4.0. Also describes a new format for push notifications sent to APNs.
2010-01-28	Made many small corrections.
2009-08-14	Made minor corrections and linked to short inline articles on Cocoa concepts.
2009-05-22	Added notes about Wi-Fi and frequency of registration, and gateway address for the development environment. Updated with various clarifications and enhancements.
2009-03-15	First version of a document that explains how providers can send push notifications to client applications using Apple Push Notification Service.

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