Local and Push Notification Programming Guide

Developer

Contents

Figures, Tables, and Listings

About Local Notifications and Push Notifications

Local notifications and push notifications are ways for an application 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 push notifications look and sound the same. They can display an alert message or they can badge the application icon. They can also play a sound when the alert or badge number is shown.

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

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

At a Glance

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

The Problem That Local and Push Notifications Solve

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

Relevant Chapter: "Local and Push Notifications in Depth" (page 9)

Local and Push Notifications Are Different in Origination

Local and push notifications serve different design needs. A local notification is local to an application on an iPhone, iPad, or iPod touch. Push notifications—also known as *remote notifications*—arrive from outside a device. They originate on a remote server—the application's provider—and are pushed to applications 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 9)

You Schedule a Local Notification, Register a Push Notification, and Handle Both

To have iOS deliver a local notification at a later time, an application creates a `UILocalNotification` object, assigns it a delivery date and time, specifies presentation details, and schedules it. To receive push notifications, an application must register to receive the notifications and then pass to its provider a device token it gets from the operating system.

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

Relevant Chapter: "Scheduling, Registering, and Handling Notifications" (page 14)

The Apple Push Notification Service Is the Gateway for Push Notifications

Apple Push Notification service (APNs) propagates push notifications to devices having applications 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 applications. When new data for an application 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 27)

You Must Obtain Security Credentials for Push Notifications

To develop and deploy the provider side of an application for push notifications, you must get SSL certificates from the appropriate Dev Center. Each certificate is limited to a single application, 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 41)

The Provider Communicates with APNs over a Binary Interface

The binary interface is asynchronous and uses a streaming TCP socket design for sending push 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-application list of devices for which there were failed-delivery attempts (that is, APNs was unable to deliver a push notification to an application 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 push notifications to them.

Related Chapters: "Apple Push Notification Service" (page 27), "Provider Communication with Apple Push Notification Service" (page 46)

Prerequisites

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

For local notifications and the client-side implementation of push notifications, familiarity with application 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:

- *iOS Team Administration Guide* describes how to perform a variety of tasks in the iOS Portal, such as provisioning devices for development.
- *Entitlement Key Reference* documents the specific entitlements needed for an app to receive push notifications.

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

- The reference documentation for `UILocalNotification`, `UIApplication`, and `UIApplicationDelegate` describe the local- and push-notification API for client applications in iOS.
- The reference documentation for `NSApplication` and `NSApplicationDelegate Protocol` describe the push-notification API for client applications in OS X.
- *Security Overview* describes the security technologies and techniques used for the iOS and Macs.
- 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 Push Notifications in Depth

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

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

Local and Push Notifications Appear the Same to Users

Users see notifications in the following ways:

- Displaying an alert or banner
- Badging the app's icon
- Playing a sound

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

Users control how the device and specific applications installed on the device should handle notifications. They can also selectively enable or disable push notification types (that is, icon badging, alert messages, and sounds) for specific applications.

Local and Push Notifications Appear Different to Apps

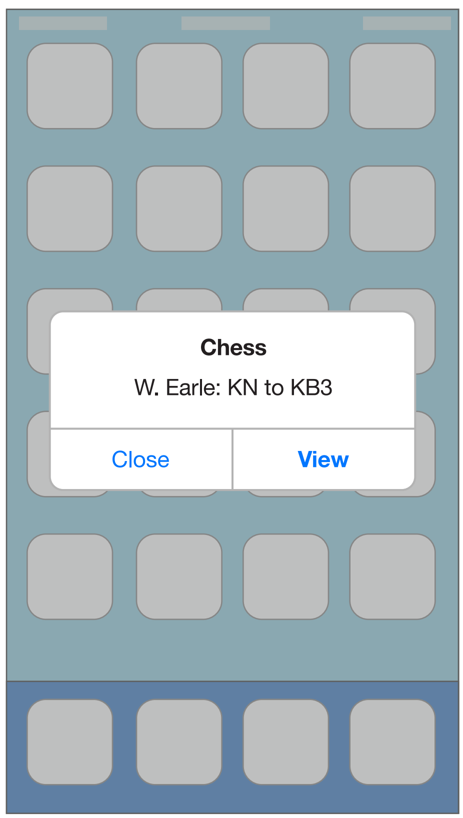
If your app is frontmost, the `application:didReceiveRemoteNotification:` or `application:didReceiveLocalNotification:` method is called on its app delegate. If your app is not frontmost or not running, you handle the notifications by checking the options dictionary passed to the `application:didFinishLaunchingWithOptions:` of your app delegate for either the `UIApplicationLaunchOptionsLocalNotificationKey` or `UIApplicationLaunchOptionsRemoteNotificationKey` key. For more details about handling notifications, see "Scheduling, Registering, and Handling Notifications" (page 14).

Examples of Notification Usage

Suppose you are playing chess with your friend online. While you are in the chess app, your friend's moves appear on the board right away. When you switch to another app—for example, to read email while your friend decides where to move—the app needs a way to notify you when your friend moves. This is a perfect use case for a push notification—something changed outside of the app that is of interest to the user.

The provider for the chess application learns about this move and, seeing that the chess application on your device is no longer connected, sends a push notification to Apple Push Notification service (APNs). Almost immediately, your device—or more precisely, the operating system on your device—receives the notification over the Wi-Fi or cellular connection from APNs. Because your chess application is not currently running, iOS displays an alert similar to Figure 1-1. The message consists of the application name, a short message, and (in this case) two buttons: Close and View. The button on the right is called the *action* button and its default title is "View". An application can customize the title of the action button and can internationalize the button title and the message so that they are in the user's preferred language.

Figure 1-1 A notification alert



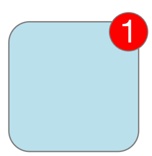
If you tap the View button, the chess application launches, connects with its provider, downloads the new data, and adjusts the chessboard user interface to show your friend's move. (Pressing Close dismisses the alert.)

> **OS X Note:** The only type of push notification in OS X for non-running applications is icon badging. In other words, an application's icon in the Dock is badged only if the application isn't running. If users have not already placed the icon in the Dock, the system inserts the icon into the Dock so that it can badge it (and removes it after the application next terminates). Running applications may examine the notification payload for other types of notifications (alerts and sounds) and handle them appropriately.

Let's consider a type of application with another requirement. This application 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 application to notify it at a specific interval before this due date expires. To effect this, the application schedules a local notification for that date and time. Instead of specifying an alert message, this time the application chooses to specify a badge number (1). At the appointed time, iOS displays a badge number in the upper-right corner of the icon of the application, such as illustrated in Figure 1-2.

For both local and push notifications, the badge number is specific to an application and can indicate any number of things, such as the number of impending calendar events or the number of data items to download or the number of unread (but already downloaded) email messages. The user sees the badge and launches the app which displays the to-do item or whatever else is of interest to the user.

Figure 1-2 An application icon with a badge number (iOS)



In iOS, an application can specify a sound file along with an alert message or badge number. The sound file should contain a short, distinctive sound. At the same moment iOS displays the alert or badges the icon, it plays the sound to alert the user to the incoming notification.

More About Local Notifications

Local notifications (available only in iOS) are ideally suited for applications with time-based behaviors, such as calendar and to-do list applications. Applications that run in the background for the limited period allowed by iOS might also find local notifications useful. For example, applications 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 then present a local notification immediately to inform their users.

A local notification is an instance of `UILocalNotification` 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 may 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**. This category includes the alert message, the title of the action button, the application icon badge number, and a sound to play.
- **Custom data**. Local notifications can include a dictionary of custom data.

"Scheduling Local Notifications" (page 15) describes these properties in programmatic detail. Once an application has created a local-notification object, it can either schedule it with the operating system or present it immediately.

Each application 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 Push Notifications

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

But what if the application 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? Push notifications are the solution to this dilemma. A push notification is a short message that a provider has delivered to the operating system of a device or computer; the operating system, in turn, informs the user of a client application 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 application is properly registered, the notification is delivered to the operating system and possibly to the application. Apple Push Notification service is the primary technology for the push-notification feature.

Push notifications serve much the same purpose as a background application on a desktop system, but without the additional overhead. For an application 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 push notification on behalf of the application and alerts the user. Once alerted, users may choose to launch the application, which then downloads the data from its provider. If an application is running when a notification comes in, the application can choose to handle the notification directly.

As its name suggests, Apple Push Notification service (APNs) uses a push design to deliver 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 push notifications.

Most of a push 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 push notifications is not guaranteed. For more on the payload, see "The Notification Payload" (page 34).

APNs retains the last notification it receives from a provider for an application 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 push notifications over both Wi-Fi and cellular connections; a computer running OS X receives push notifications over both WiFi and Ethernet connections.

> **Important:** In iOS, Wi-Fi is used for push 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 push 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 application requires that you obtain the proper certificates from the Dev Center for either iOS or OS X and then write the requisite code for the client and provider sides of the application. "Provisioning and Development" (page 41) explains the provisioning and setup steps, and "Provider Communication with Apple Push Notification Service" (page 46) and "Scheduling, Registering, and Handling Notifications" (page 14) describe the details of implementation.

Scheduling, Registering, and Handling Notifications

This chapter describes the tasks that an iOS on OS X application should (or might) do to schedule local notifications, register remote notifications, and handle both local and remote notifications. Because the client-side API for push notifications refers to push notifications as *remote notifications*, that terminology is used in this chapter.

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 application. The sound files must be in the main bundle of the client application.

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 application 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 application:

```
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.

Scheduling Local Notifications

Creating and scheduling local notifications in iOS requires that you perform a few simple steps:

1. Allocate and initialize a `UILocalNotification` object.
2. 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).
3. Configure the substance of the notification: alert, icon badge number, and sound.
   - The alert has a property for the message (the `alertBody` property) and for the title of the action button or slider (`alertAction`); both of these string values can be internationalized for the user's current language preference.
   - You set the badge number to display on the application icon through the `applicationIconBadgeNumber` property.
   - You can assign the filename of a nonlocalized custom sound in the application's main bundle to the `soundName` property; to get the default system sound, assign `UILocalNotificationDefaultSoundName`. Sounds should always accompany an alert message or icon badging; they should not be played otherwise.
4. Optionally, you can attach custom data to the notification through the `userInfo` property.

   Keys and values in the `userInfo` dictionary must be property-list objects.
5. Schedule the local notification for delivery.

   You schedule a local notification by calling the `UIApplication` method `scheduleLocalNotification:`. The application 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 2-1 creates and schedules a notification to inform the user of a hypothetical to-do list application about the impending due date of a to-do item. There are a couple things to note about it. For the `alertBody` and `alertAction` 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 2-1 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.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 application object, and you can cancel all scheduled notifications by calling `cancelAllLocalNotifications`. Both of these methods also programmatically dismiss a currently displayed notification alert.

Applications 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, they should present the notification immediately using the `UIApplication` method `presentLocalNotificationNow:` (iOS gives an application a limited time to run in the background). Listing 2-2 illustrates how you might do this.

Listing 2-2 Presenting a local notification immediately while running in the background

```
- (void)applicationDidEnterBackground:(UIApplication *)application {
    NSLog(@"Application entered background state.");
    // bgTask is a property of the class
    NSAssert(self.bgTask == UIInvalidBackgroundTask, nil);

    bgTask = [application beginBackgroundTaskWithExpirationHandler: ^{
        dispatch_async(dispatch_get_main_queue(), ^{
            [application endBackgroundTask:self.bgTask];
            self.bgTask = UIInvalidBackgroundTask;
        });
    }];

    dispatch_async(dispatch_get_main_queue(), ^{
        while ([application backgroundTimeRemaining] > 1.0) {
            NSString *friend = [self checkForIncomingChat];
            if (friend) {
                UILocalNotification *localNotif = [[UILocalNotification alloc]
init];
                if (localNotif) {
                    localNotif.alertBody = [NSString stringWithFormat:
                        NSLocalizedString(@"%@ has a message for you.", nil),
friend];
                    localNotif.alertAction = NSLocalizedString(@"Read Message",
nil);
                    localNotif.soundName = @"alarmsound.caf";
                    localNotif.applicationIconBadgeNumber = 1;
```

```
                    [application presentLocalNotificationNow:localNotif];
                    [localNotif release];
                    friend = nil;
                    break;
                }
            }
        }
        [application endBackgroundTask:self.bgTask];
        self.bgTask = UIInvalidBackgroundTask;
    });

}
```

Registering for Remote Notifications

An application must register with Apple Push Notification service for the operating systems on a device and on a computer to receive remote notifications sent by the application's provider. Registration has three stages:

1. The app registers for remote notifications.
2. The system sets up remote notifications for the app and, if registration is successful, passes a device token to the app delegate.
3. The app sends its device token to the push provider.

The actions that take place during this sequence are illustrated by Figure 3-3 in "Token Generation and Dispersal" (page 31).

Device tokens can change. Your app needs to reregister every time it is launched—in iOS by calling the `registerForRemoteNotificationTypes:` method of `UIApplication`, and in OS X by calling the `registerForRemoteNotificationTypes:` method of `NSApplication`. The parameter passed to this method specifies the initial types of notifications that the application wishes to receive. Users can modify the enabled notification types at any point, using Settings in iOS or System Preferences in OS X. You can query the currently enabled notification types using the `enabledRemoteNotificationTypes` property of `UIApplication` or the `enabledRemoteNotificationTypes` property of `NSApplication`. The system does not badge icons, display alert messages, or play alert sounds if any of these notifications types are not enabled for your app, even if they are specified in the notification payload.

> **OS X Note:** Because the only notification type supported for non-running applications is icon-badging, pass `NSRemoteNotificationTypeBadge` as the parameter of `registerForRemoteNotificationTypes:`.

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 application connects with its provider and pass it this token, encoded in binary format. If there is a problem in obtaining the token, the operating system informs the delegate by calling the `application:didFailToRegisterForRemoteNotificationsWithError:` method. 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. (See "Creating and Installing the Provisioning Profile" (page 43) for details.)

> **iOS Note:** If a cellular or Wi-Fi connection is not available, neither the `application:didRegisterForRemoteNotificationsWithDeviceToken:` method or 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 application launches, you ensure that the provider has the current token for the 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), he or she must launch the application 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 application has previously registered, calling `registerForRemoteNotificationTypes:` 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 2-3 gives a simple example of how you might register for remote notifications in an iOS application. The code would be nearly identical for a Mac app.

Listing 2-3 Registering for remote notifications

```
- (void)applicationDidFinishLaunching:(UIApplication *)app {
```

```
   // other setup tasks here....
    [[UIApplication sharedApplication]
registerForRemoteNotificationTypes:(UIRemoteNotificationTypeBadge |
UIRemoteNotificationTypeSound)];
}


// 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);
}
```

Handling Local and Remote Notifications

Let's review the possible scenarios when the system delivers a local notification or a remote notification for an application.

- The notification is delivered when the application isn't running in the foreground.

  In this case, the system presents the notification, displaying an alert, badging an icon, perhaps playing a sound.

- As a result of the presented notification, the user taps the action button of the alert or taps (or clicks) the application icon.

  If the action button is tapped (on a device running iOS), the system launches the application and the application calls its delegate's `application:didFinishLaunchingWithOptions:` method (if implemented); it passes in the notification payload (for remote notifications) or the local-notification object (for local notifications).

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

> **iOS Note:** The application delegate could implement `applicationDidFinishLaunching:` rather than `application:didFinishLaunchingWithOptions:`, but that is strongly discouraged. The latter method allows the application to receive information related to the reason for its launching, which can include things other than notifications.

- The notification is delivered when the application is running in the foreground.

  The application calls its delegate's `application:didReceiveRemoteNotification:` method (for remote notifications) or `application:didReceiveLocalNotification:` method (for local notifications) and passes in the notification payload or the local-notification object.

> **Note:** The delegate methods cited in this section that have "RemoteNotification" in their name are declared with identical signatures by both `NSApplicationDelegate Protocol` and `UIApplicationDelegate`.

An application 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, it should adopt the `NSApplicationDelegate Protocol` protocol and implement both the `applicationDidFinishLaunching:` method and the `application:didReceiveRemoteNotification:` method.
- For iOS, it should should adopt the `UIApplicationDelegate` protocol and implement both the `application:didFinishLaunchingWithOptions:` method and the `application:didReceiveRemoteNotification:` or `application:didReceiveLocalNotification:` method.

iOS Note: In iOS, you can determine whether an application is launched as a result of the user tapping the action button or whether the notification was delivered to the already-running application by examining the application state. In the delegate's implementation of the `application:didReceiveRemoteNotification:` or `application:didReceiveLocalNotification:` method, get the value of the `applicationState` property and evaluate it. If the value is `UIApplicationStateInactive`, the user tapped the action button; if the value is `UIApplicationStateActive`, the application was frontmost when it received the notification.

The delegate for an iOS application in Listing 2-4 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 application's initial context. As shown in this example, you should appropriately reset the badge number on the application icon—or remove it if there are no outstanding items—as part of handling the notification.

Listing 2-4 Handling a local notification when an application 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 application can use to provide context when setting up the initial user interface. See "The Notification Payload" (page 34) for details about the remote-notification payload.

> **Important:** Do not define custom properties in the notification payload for the purpose of transporting customer data or any other sensitive data. Delivery of remote notifications is not guaranteed.
>
> 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 application 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 application can use this value to gauge how old the notification is.

When handling remote notifications in `application:didFinishLaunchingWithOptions:` or `applicationDidFinishLaunching:`, the application delegate might perform a major additional task. Just after the application launches, the delegate should connect with its provider and fetch the waiting data. Listing 2-5 gives a schematic illustration of this procedure.

Listing 2-5 Downloading data from a provider

```
- (void)application:(UIApplication *)app didFinishLaunchingWithOptions:(NSDictionary
 *)opts {
    // check launchOptions for notification payload and custom data, set UI context
    [self startDownloadingDataFromProvider];  // custom method
    app.applicationIconBadgeNumber = 0;
    // other setup tasks here....
}
```

> **Note:** A client application 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, as you'll recall, is called when application is running in the foreground. Here the application delegate does the same work as it does in Listing 2-4. 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 application 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 application to catch remote notifications that the system delivers while it is running in the foreground, the application delegate should implement the `application:didReceiveRemoteNotification:` 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 application icon. (If your application frequently checks with its provider for new data, implementing this method might not be necessary.) 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 application's current context.

Even though the only supported notification type for nonrunning applications in OS X is icon-badging, the delegate can implement `application:didReceiveRemoteNotification:` to examine the notification payload for other types of notifications and handle them appropriately (that is, display an alert or play a sound).

iOS Note: If the user unlocks the device shortly after a remote-notification alert is displayed, the operating system automatically triggers the action associated with the alert. (This behavior is consistent with SMS and calendar alerts.) This makes it even more important that actions related to remote notifications do not have destructive consequences. A user should always make decisions that result in the destruction of data in the context of the application that stores the data.

Passing the Provider the Current Language Preference (Remote Notifications)

If an application 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 application.) The client application 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 34).

Listing 2-7 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 2-7 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 application 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 application 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 push 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 application arrives when that application is not running, the device alerts the user that the application 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 applications. When new data for an application 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 29) for more information.

"Provider Communication with Apple Push Notification Service" (page 46) and "Scheduling, Registering, and Handling Notifications" (page 14) discuss the specific implementation requirements for providers and iOS applications, respectively.

A Push Notification and Its Path

Apple Push Notification service transports and routes a 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 application 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 application on a device is to be alerted.

> **Note:** For more information about the device token, see "Security Architecture" (page 29); for further information about the notification payload, see "The Notification Payload" (page 34).

The remote-notification data flows in one direction. The provider composes a notification package that includes the device token for a client application 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 application for which it's providing data. The topic is currently the bundle identifier of the target application.

Figure 3-1 A push notification from a provider to a client application

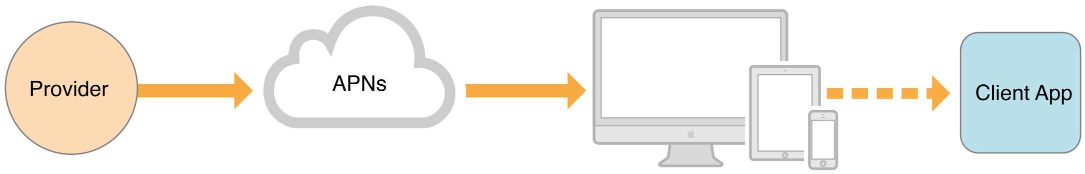
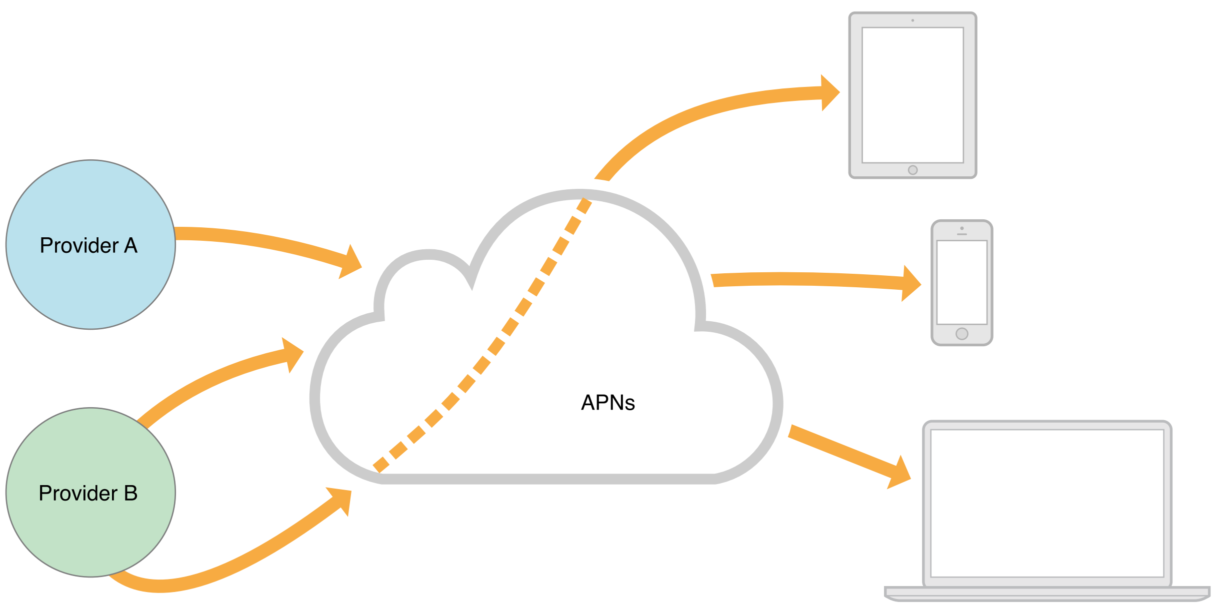


Figure 3-1 is a greatly simplified depiction of the virtual network APNs makes possible among providers and devices. The device-facing and provider-facing sides of APNs both have multiple points of connection; on the provider-facing side, these are called gateways. There are typically multiple providers, 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 applications are installed. Figure 3-2 is a slightly more realistic depiction.

Figure 3-2 Push 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 50).

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 application 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.

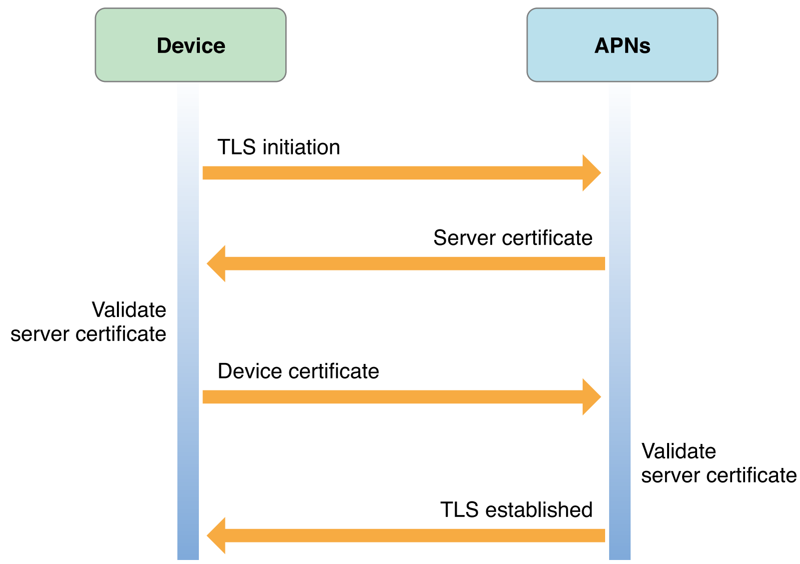
In APNs, assurance of accurate message routing—or **token trust**—is made possible through the device token. A device token is an opaque identifier of a device that APNs gives to the device when it first connects with it. The device shares the device token with its provider. Thereafter, this token accompanies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

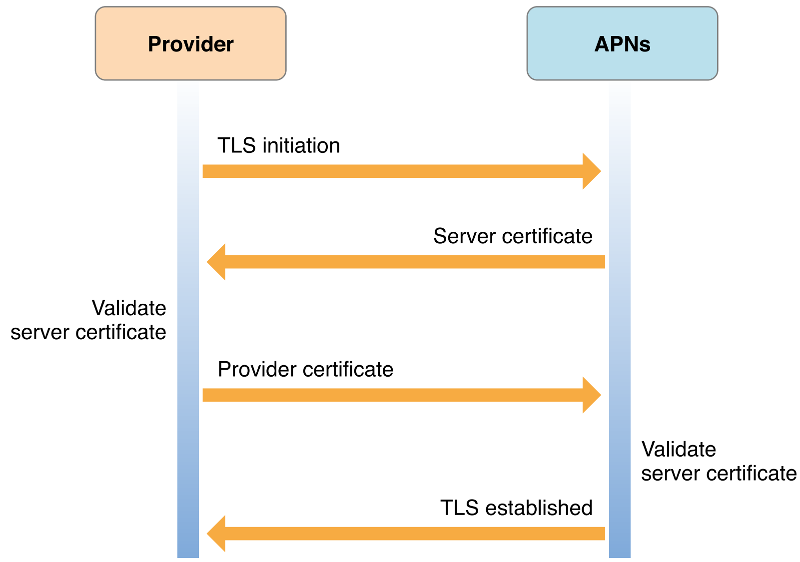
Service-to-Device Connection Trust

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 30). 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.

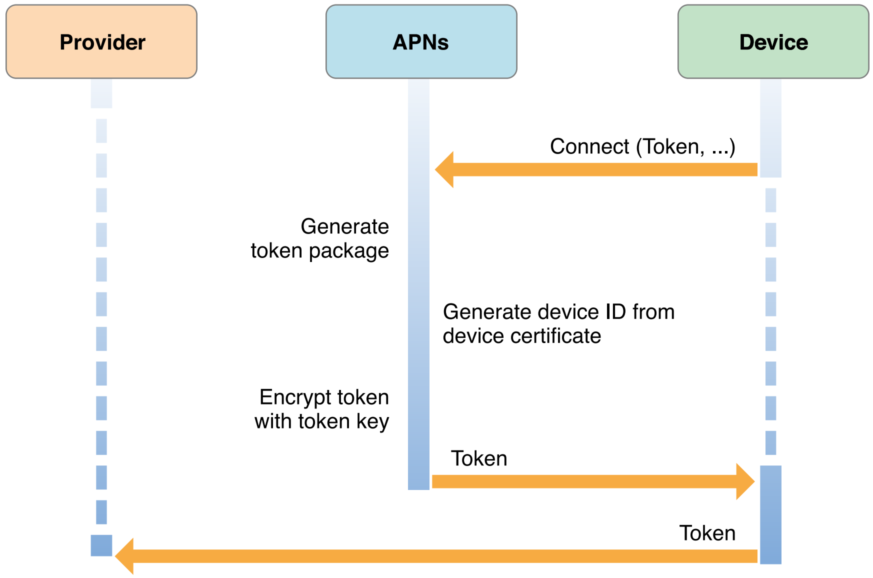


Note that provider connection is valid for delivery to only one specific application, identified by the topic (bundle ID) specified in the certificate. APNs also maintains a certificate revocation list; if a provider's certificate is on this list, APNs may revoke provider trust (that is, refuse the connection).

Token Generation and Dispersal

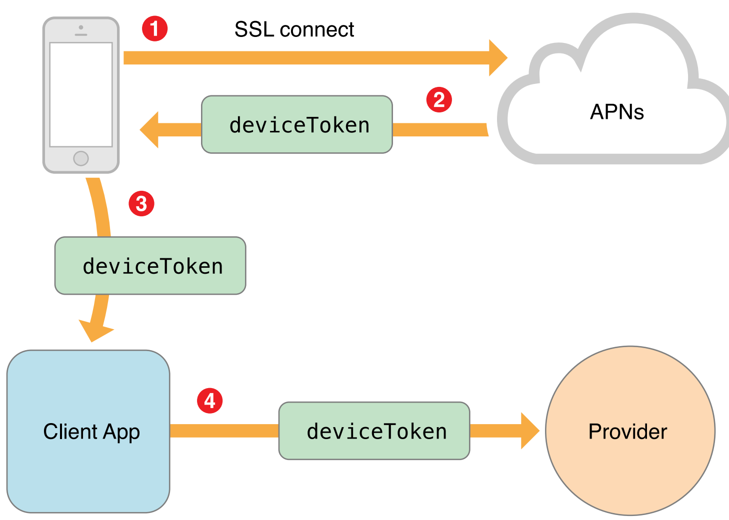
Applications must **register** to receive push notifications; it typically does this right after it is installed on a device. (This procedure is described in "Scheduling, Registering, and Handling Notifications" (page 14).) The system receives the registration request from the application, connects with 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 application as an `NSData` object. The application 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 application in furnishing its provider with the device token.

Figure 3-3 Sharing the device token

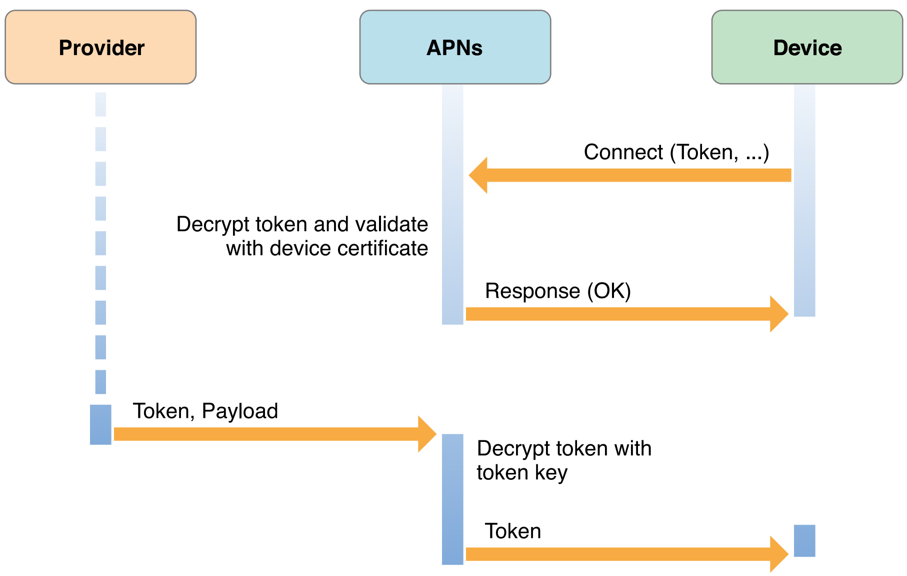


The form of this phase of token trust ensures that only APNs generates the token which it will later honor, and it can assure itself that a token handed to it by a device is the same token that it previously provisioned for that particular device—and only for that device.

Token Trust (Notification)

After the system obtains a device token from APNs, as described in "Token Generation and Dispersal" (page 31), 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 application 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 application. 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 application 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 push notification includes a payload. The payload contains information about how the system should alert the user as well as any custom data you provide. The maximum size allowed for a notification payload is 256 bytes; Apple Push Notification Service refuses any notification that exceeds this limit.

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 contains one or more properties that specify the following actions:

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

> **Note:** Avoid using more than one action per notification. Notifications by their nature interrupt the user, and each action adds additional interruption. Apps that overuse notification actions risk annoying their users.

If the target application isn't running when the notification arrives, the alert message, sound, or badge value is played or shown. If the application is running, the system delivers the notification to the application 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 application 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 application is launched. Alternatively, you can specify a dictionary as the value of `alert`. See Table 3-2 (page 36) for descriptions of the keys of this dictionary.
`badge`	number	The number to display as the badge of the application 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 application 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 14) for details.
`content-available`	number	Provide this key with a value of `1` to indicate that new content is available. This is used to support Newsstand apps and background content downloads. 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
`body`	string	The text of the alert message.
`action-loc-key`	string or `null`	If a string is specified, the system displays an alert with two buttons, whose behavior is described in Table 3-1. 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 36) 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 `loc-args`. See "Localized Formatted Strings" (page 36) 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 36) for more information.
`launch-image`	string	The filename of an image file in the application 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 application'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 25)). Or the client application can store in its bundle the alert-message strings translated for each localization it supports. The provider specifies the `loc-key` and `loc-args` properties in the `aps` dictionary of the notification payload. When the device receives the notification (assuming the application 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 application 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 application 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 application bundle. (The application 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 an `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 Programming Topics* . 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 application icon be badged with the number 5.

```
{
    "aps" : {
        "alert" : {
            "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 device should display an alert message with both Close and View buttons. It also request that the application 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 application'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"
    },
    "acme" : "foo"
}
```

Provisioning and Development

Development and Production Environments

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

- **Development**: Use the development environment for initial development and testing of the provider application. 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 application. Applications 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 application that is the recipient of push notifications; this identifier includes the application'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 push notifications, `<aps-environment>`. The two provisioning profiles are called Development and Distribution. The Distribution provisioning profile is a requirement for submitting your application 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 application ID. As a result this entitlement is built into the application, which enables it to receive push notifications.

Provisioning Procedures

In the iOS Developer Program, each member on a development team has one of three roles: team agent, team admin, and team member. The roles differ in relation to iPhone development certificates and provisioning profiles. The team agent is the only person on the team who can create Development SSL certificates *and* Distribution (Production) SSL certificates. The team admin and the team agent can both create both Development and Distribution provisioning profiles. Team members may only download and install certificates and provisioning profiles. The procedures in the following sections make reference to these roles.

> **Note:** The iOS Provisioning Portal makes available to all iOS Developer Program members a user guide and a series of videos that explain all aspects of certificate creation and provisioning. The following sections focus on APNs-specific aspects of the process and summarize other aspects. To access the portal, iOS Developer Program members should go to the iOS Dev Center (http://developer.apple.com/devcenter/ios), log in, and click then go to the iOS Provisioning Portal page (there's a link in the upper right).

Creating the SSL Certificate and Keys

In the provisioning portal of the iOS Dev Center, the team agent selects the application IDs for APNs. He also completes the following steps to create the SSL certificate:

1. Click App IDs in the sidebar on the left side of the window.

   The next page displays your valid application IDs. An application ID consists of an application's bundle ID prefixed with a ten-character code generated by Apple. The team admin must enter the bundle ID. For a certificate, it must incorporate a specific bundle ID; you cannot use a "wildcard" application ID.

2. Locate the application ID for the development SSL certificate (and that is associated with the Development provisioning profile) and click Configure.

   You must see "Available" under the Apple Push Notification Service column to configure a certificate for this application ID.

3. In the Configure App ID page, check the Enable Push Notification Services box and click the Configure button.

   Clicking this button launches an APNs Assistant, which guides you through the next series of steps.

4. The first step requires that you launch the Keychain Access application and generate a Certificate Signing Request (CSR).

   Follow the instructions presented in the assistant. When you are finished generating a CSR, click Continue in Keychain Access to return to the APNs Assistant.

   When you create a CSR, Keychain Access generates a private and a public cryptographic key pair. The private key is put into your Login keychain by default. The public key is included in the CSR sent to the provisioning authority. When the provisioning authority sends the certificate back to you, one of the items in that certificate is the public key.

5. In the Submit Certificate Signing Request pane, click Choose File. Navigate to the CSR file you created in the previous step and select it.

6. Click the Generate button.

   While displaying the Generate Your Certificate pane, the Assistant configures and generates your Client SSL Certificate. If it succeeds, it displays the message "Your APNs Certificate has been generated." Click Continue to proceed to the next step.

7. In the next pane, click the Download Now button to download the certificate file to your download location. Navigate to that location and double-click the certificate file (which has an extension of `cer`) to install it in your keychain. When you are finished, click Done in the APNs Assistant.

   Double-clicking the file launches Keychain Access. Make sure you install the certificate in your login keychain on the computer you are using for provider development. In Keychain Access, ensure that your certificate user ID matches your application's bundle ID. The APNs SSL certificate should be installed on your notification server.

When you finish these steps you are returned to the Configure App ID page of the iOS Dev Center portal. The certificate should be badged with a green circle and the label "Enabled".

To create a certificate for the production environment, repeat the same procedure but choose the application ID for the production certificate.

Creating and Installing the Provisioning Profile

The Team Admin or Team Agent must next create the provisioning profile (Development or Distribution) used in the server side of remote-notification development. The provisioning profile is a collection of assets that associates developers of an application and their devices with an authorized development team and enables those devices to be used for testing. The profile contains certificates, device identifiers, the application's bundle ID, and all entitlements, including `<aps-environment>`. All team members must install the provisioning profile on the devices on which they will run and test application code.

Note: Refer to the program user guide for the details of creating a provisioning profile.

To download and install the provisioning profile, team members should complete the following steps:

1. Go to the provisioning portal in the iOS Dev Center.
2. Create a new provisioning profile that contains the App ID you registered for APNs.
3. Modify any *existing* profile before you download the new one.

   You have to modify the profile in some minor way (for example, toggle an option) for the portal to generate a new provisioning profile. If the profile isn't so "dirtied," you're given the original profile without the push entitlements.
4. From the download location, drag the profile file (which has an extension of `mobileprovision`) onto the Xcode or iTunes application icons.

   Alternatively, you can move the profile file to `~/Library/MobileDevice/Provisioning Profiles`. Create the directory if it does not exist.
5. Verify that the entitlements in the provisioning-profile file are correct. To do this, open the `.mobileprovision` file in a text editor. The contents of the file are structured in XML. In the Entitlements dictionary locate the `aps-environment` key. For a development provisioning profile, the string value of this key should be `development`; for a distribution provisioning profile, the string value should be `production`.
6. In the Xcode Organizer window, go the Provisioning Profiles section and install the profile on your device.

When you build the project, the binary is now signed by the certificate using the private key.

Installing the SSL Certificate and Key on the Server

You should install the SSL distribution certificate and private cryptographic key you obtained earlier on the server computer on which the provider code runs and from which it connects with the development or production versions of APNs. To do so, complete the following steps:

1. Open Keychain Access utility and click the My Certificates category in the left pane.
2. Find the certificate you want to install and disclose its contents.

   You'll see both a certificate and a private key.
3. Select both the certificate and key, choose File > Export Items, and export them as a Personal Information Exchange (.p12) file.

4. Servers implemented in languages such as Ruby and Perl often are better able to deal with certificates in the Personal Information Exchange format. To convert the certificate to this format, complete the following steps:

   a. In KeyChain Access, select the certificate and choose File > Export Items. Select the Personal Information Exchange (.p12) option, select a save location, and click Save.

   b. Launch the Terminal application and enter the following command after the prompt:

      `openssl pkcs12 -in` *CertificateName*`.p12 -out` *CertificateName*`.pem -nodes`

5. Copy the `.pem` certificate to the new computer and install it in the appropriate place.

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 provisioned through the iOS Provisioning Portal. (See "Provisioning and Development" (page 41) 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 push notifications:

- You must compose the notification payload (see "The Notification Payload" (page 34)).
- You are responsible for supplying the badge number to be displayed on the application 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 applications. See "The Feedback Service" (page 50) 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 application. "Scheduling, Registering, and Handling Notifications" (page 14) suggests an approach for obtaining this information. See "The Notification Payload" (page 34) for information about the `loc-key` and `loc-args` properties.

Best Practices for Managing Connections

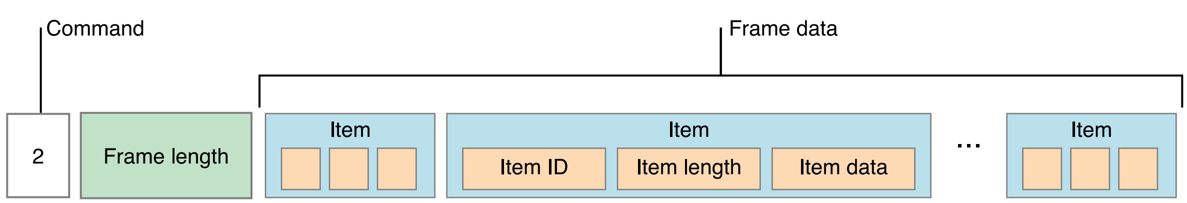
You may establish multiple connections to the same gateway or to multiple gateway instances. If you need to send a large number of push notifications, spread them out over connections to several different gateways. This improves performance compared to using a single connection: it lets you send the push 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 frame contains the body, structured as a series of items.

The items and their identifiers are are as follows:

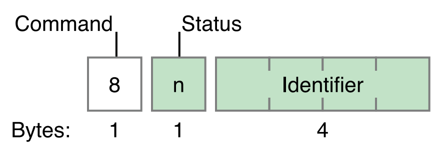
Item ID	Item Name	Length	Data
1	Device token	32 bytes	The device token in binary form, as was registered by the device.
2	Payload	variable length, less than or equal to 256 bytes	The JSON-formatted payload. The payload must not be null-terminated.
3	Notification identifier	4 bytes	An arbitrary, opaque 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 push 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 push notifications. When a push 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. Push 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 push notifications that will fail to be delivered, you reduce unnecessary message overhead and improve overall system performance.

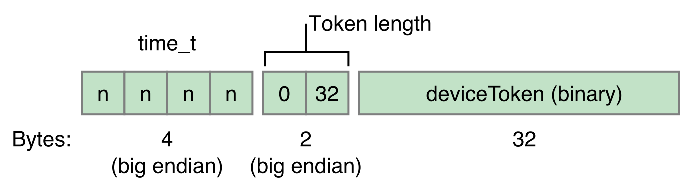
Query the feedback service daily to get the list of device tokens. Use the timestamp to verify that the device tokens haven't been reregistered since the feedback entry was generated. For each device that has not been reregistered, stop sending notifications. APNs monitors providers for their diligence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

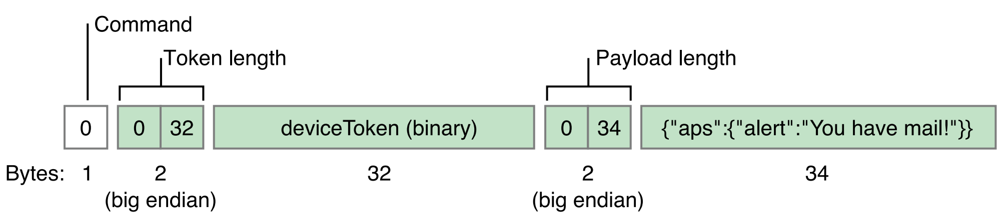
New development should use the modern format to connect to APNs, as described in "The Binary Interface and Notification Format" (page 47).

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 push 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;
}
```

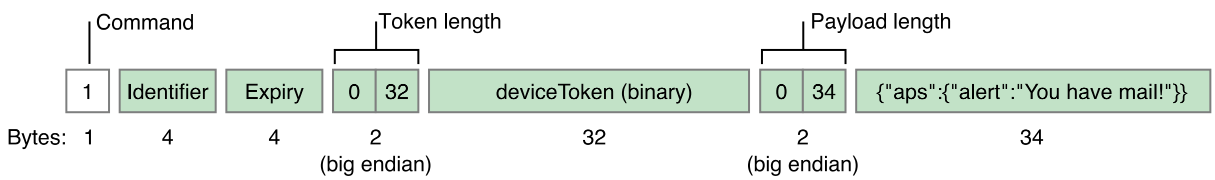
Enhanced Notification Format

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 application 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 push 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 Push Notification Programming Guide* .

Date	Notes
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 47). Added error code 10 to Table 5-1 (page 49). Expanded discussion in "The Feedback Service" (page 50). 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.
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.

Date	Notes
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