Next

About Local and Remote Notifications

Local notifications and remote notifications are the two types of so-called user notifications, as distinguished from broadcast notifications (managed by the `NSNotificationCenter` class) and key-value observing notifications. User notifications enable an app that isn't running in the foreground to let its users know it has information for them. The information could be a message, an impending calendar event, or new data on a remote server, for example. When presented by the operating system, user notifications, whether local or remote in origin, look and sound the same. They can display an alert message or they can badge the app icon. They can also play a sound when the alert or badge number is shown. Upon receiving a user notification, the user can tap it to launch the associated app and see the details. They can also choose to ignore the notification, in which case the app is not activated.

At a Glance

Local and remote notifications appear the same to a user, but they serve different use cases and you configure and manage them differently.

Local and Remote Notifications Contrasted

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

Local and remote notifications serve different design needs, as follows:

- A local notification is scheduled and sent by the app itself, without necessary involvement of the Internet.
- A remote notification, also called a push notification, arrives from outside the device. It originates on a remote server that you manage—the app's provider—and is pushed to your app on a user's device via the Apple Push Notification service (APNs).

Relevant Chapter: Local and Remote Notifications in Depth

Registering, Scheduling, and Handling User Notifications

For the system deliver a local notification at a later time, an app registers notification types (in iOS 8 and later), creates a local notification object (using either `UILocalNotification` or `a target="_self" NSUserNotification/a`), assigns it a delivery date and time, specifies presentation details, and schedules it for delivery. To receive remote

notifications, an app must register notification types, then pass to its provider an app-specific device token it gets from the operating system. When the operating system delivers a local notification or remote notification and the target app is not running in the foreground, it can present the notification to the user through an alert, icon badge number, or sound. If there is a notification alert and the user taps or clicks an action button (or moves the action slider), the app launches and calls a method to pass in the local-notification object or remote-notification payload. If the app is running in the foreground when the notification is delivered, the app delegate receives a local or remote notification.
In iOS 8 and later, user notifications can include custom actions. Also, location-based local notifications can be sent whenever the user arrives at a particular geographic location.
Relevant Chapter: Registering, Scheduling, and Handling User Notifications

Apple Push Notification Service

Apple Push Notification service (APNs) propagates remote notifications to devices having apps registered to receive those notifications. Each app on a device establishes an accredited and encrypted IP connection with the service and receives notifications over this persistent connection. Providers connect with APNs through a persistent and secure channel while monitoring incoming data intended for their client apps. When new data for an app arrives, the provider prepares and sends a notification through the channel to APNs, which pushes the notification to the target device.
The APNs Provider API is asynchronous and, starting in December 2015, uses HTTP/2 to send remote notification requests from your provider server to APNs. The provider composes each outgoing notification and sends it over this channel to APNs.
Related Chapters: Apple Push Notification Service, APNs Provider API

Security Credentials for Remote Notifications

To develop and deploy the remote notification provider server for your app, you must get one or more SSL certificates from Member Center. Starting in December 2015, the HTTP/2-based provider API lets you obtain a single certificate to use for both your development and production environments. In addition, the single certificate can be used to send notifications to not only the primary app (as identified by bundle ID) but also to associated Apple Watch complications and backgrounded VoIP services.
Related Chapter: Provisioning and Development

Prerequisites

App Programming Guide for iOS describes the high level patterns for writing iOS apps.
For local notifications and the client-side implementation of remote notifications, familiarity with app development for iOS is assumed. For the provider side of the implementation, knowledge of TLS/SSL and streaming sockets is helpful.

See Also

The following documents provide background information:

- App Distribution Quick Start teaches how to create a team provisioning profile in Xcode before you enable APNs.
- Configuring Push Notifications in App Distribution Guide explains the steps for obtaining APNs certificates from Member Center.
- Entitlement Key Reference documents the specific entitlements needed for an app to receive remote notifications.

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

- Notification Essentials in App Programming Guide for watchOS explains how Apple Watch supports user notifications.
- The reference documentation for `UILocalNotification`, `UIApplication`, and `UIApplicationDelegate` describe the local- and remote-notification API for client apps in iOS.
- The reference documentation for `a target="_self" NSApplication/a` and `a target="_self" NSApplicationDelegate Protocol/a` describe the remote-notification API for client apps in OS X.
- Security Overview describes the security technologies and techniques used for the iOS and OS X systems.
- RFC 5246 is the standard for the TLS protocol.Secure communication between data providers and Apple Push Notification service requires knowledge of Transport Layer Security (TLS) or its predecessor, Secure Sockets Layer (SSL). Refer to one of the many online or printed descriptions of these cryptographic protocols for further information.

For information on how to send push notifications to your website visitors using OS X, read a target="_self" Configuring Safari Push Notifications/a in a target="_self" Notification Programming Guide for Websites/a.
For help with issues you encounter with sending or receiving remote (push) notifications, read Technical Note TN2265, Troubleshooting Push Notifications.
Next

Local and Remote Notifications in Depth

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

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

Local and Remote Notifications Appear the Same to Users

Users can get notified in the following ways:

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

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

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

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

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

Local and Remote Notifications Appear Different to Apps

When your app is frontmost, UIKit delivers local and remote notifications directly to your app delegate object without displaying any system UI. UIKit calls the `application:didReceiveLocalNotification:` method for incoming local notifications and the `application:didReceiveRemoteNotification:fetchCompletionHandler:` method for incoming remote notifications. Use the provided notification dictionary to update your app accordingly. Because your app is running, you can incorporate the notification data quietly or update your user interface and let the user know that new information is available.
When your app must be launched to receive a notification, UIKit includes the `UIApplicationLaunchOptionsLocalNotificationKey` or `UIApplicationLaunchOptionsRemoteNotificationKey` key in the launch options dictionary passed to your app delegate's `application:willFinishLaunchingWithOptions:` and `application:didFinishLaunchingWithOptions:` methods. The presence of those keys lets you know that there is notification data waiting to be handled and gives you a chance to configure your app's interface appropriately. You do not need to handle the notification in these methods, though. After your app is running, UIKit calls other methods of your app delegate, such as the `application:didReceiveLocalNotification:` method, to give you an opportunity to process the notification data. Which methods are called depends on which methods you implemented and whether the user interacted with the system UI for the message.
When your app is running but not frontmost, UIKit displays the system UI and then delivers the results to your app in the background when possible. Similar to when your app is running, UIKit calls methods of your app delegate to receive the notification data and perform any actions. If it is unable to deliver the notification data in the background, UIKit waits until the next time your app runs to deliver it.
For more information about handling notifications, see Registering, Scheduling, and Handling User Notifications.

More About Local Notifications

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

message is ready to view or an update is ready to download, they can handle the data as needed, and notify users if appropriate.
A local notification is an instance of `UILocalNotification` or `a target="_self" NSUserNotification/a` with three general kinds of properties:

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

Listing 2-5 describes these properties in programmatic detail. After an app has created a local-notification object, it can either schedule it with the operating system or present it immediately.
Each app on a device is limited to 64 scheduled local notifications. The system discards scheduled notifications in excess of this limit, keeping only the 64 notifications that will fire the soonest. Recurring notifications are treated as a single notification.

More About Remote Notifications

An iOS or Mac app is often a part of a larger application based on the client/server model. The client side of the app is installed on the device or computer; the server side of the app has the main function of providing data to its client apps, and hence is termed a provider. A client app periodically connects with its provider and downloads any data that is waiting for it. Email and social-networking apps are examples of this client/server model.
But what if the app is not connected to its provider or even running on the device or computer when the provider has new data for it to download? How does it learn about this waiting data? Remote (or push) notifications are the solution to this dilemma. A remote notification is a short message that a provider has delivered to the operating system of a device or computer; the operating system, in turn, can inform the user of a client app that there is data to be downloaded, a message to be viewed, and so on. If the user enables this feature (on iOS) and the app is properly registered, the notification is delivered to the operating system and possibly to the app. Apple Push Notification service (APNs) is the primary technology for the remote-notification feature.

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

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

APNs uses a persistent IP connection for implementing remote notifications. Most of a remote notification consists of a payload: a JSON dictionary containing APNs-defined properties specifying how the user is to be notified. The smaller you make the payload, the better the performance of your notifications. Although you can define custom properties, do not use the remote-notification mechanism for data transport. Delivery of remote notifications is best-effort with a high success rate but is not guaranteed. For more on the payload, see The Remote Notification Payload.

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

Note: In iOS, remote notifications use Wi-Fi if it is turned on and connected. If Wi-Fi connection fails, remote notifications attempt to use the device's cellular connection.

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. iPad remains associated with its Wi-Fi access point while asleep, permitting the delivery of remote notifications. The iPad Wi-Fi radio wakes the host processor for incoming traffic as needed.

Sending notifications too frequently negatively impacts device battery life because a device must access the network to receive a notification.

Adding the remote-notification feature to your app requires that you obtain the proper certificate from Member Center and then write the requisite code for the client and provider sides of the app. Provisioning and Development explains the provisioning and setup steps, and APNs Provider API and Registering, Scheduling, and Handling User Notifications describe the details of implementation.

For details on using Member Center for obtaining an APNs certificate, read Configuring Push Notifications in App Distribution Guide.
Next
Previous

Next
Previous

Registering, Scheduling, and Handling User Notifications

Apps must be configured appropriately before they can receive local or remote notifications. The configuration process differs slightly on iOS and OS X, but the basic principles are the same. At launch time, your app registers to receive notifications and works with the system to configure that notification support. Once registration is complete, you can start creating notifications for delivery to your app. Your app then handles these incoming notifications and provides an appropriate response.
In iOS and tvOS, registration is divided into two parts: registering the supported user interactions and registering for the notifications themselves. Registering your app's supported user interaction types tells the operating system how you want to notify the user when a notification arrives. This step is required for both local or remote notifications. For remote notifications, you must perform a second registration step to obtain the app-specific device token used by the APNs server to deliver notifications. (For local notifications, there is no second registration step.)
In OS X, registration is necessary only for apps that support remote notifications.
For help with issues you encounter with sending or receiving remote (push) notifications, read Technical Note TN2265, Troubleshooting Push Notifications.

Registering Your iOS App's Supported User Interaction Types

In iOS 8 and later, apps that use either local or remote notifications must register the types of user interactions the app supports. Apps can ask to badge icons, display alert messages, or play sounds. When you request any of these interaction types, the checks to see what types of interactions the user has allowed for your app. If the user has disallowed a particular type of interaction, the system ignores attempts to interact with the user in that

way. For example, if a notification wants to display an alert message and play a sound, and the user has disallowed sounds, the system displays the alert message but does not play the sound.

To register your app's supported interaction types, call the `registerUserNotificationSettings:` method of the shared `UIApplication` object. Use the settings object to specify whether your app badges its icon, displays alert messages, or plays sounds. If you do not request any interaction types, the system pushes all notifications to your app silently. Listing 2-1 shows a short code snippet for an app that supports displaying alert messages and playing sounds. (The cast to the `UIUserNotificationType` type, in the first line of the snippet, is required by the notifications API.)

Listing 2-1 Registering notification types

```
UIUserNotificationType types = (UIUserNotificationType)
(UIUserNotificationTypeBadge |

             UIUserNotificationTypeSound |
UIUserNotificationTypeAlert);



UIUserNotificationSettings *mySettings =

            [UIUserNotificationSettings settingsForTypes:types
categories:nil];



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

In addition to registering your app's interaction types, apps can register one or more categories. Categories are supported for both local and remote notifications, and you use them to identify the purpose of the notification. Your iOS app can use the category identifier to decide how to handle the notification. In watchOS, categories are also used to customize the notification interface displayed to the user.

Starting in iOS 8, you can optionally create actionable notifications by registering custom actions for a notification type. When an actionable notification arrives, the system creates a button for each registered action and adds those buttons to the notification interface. These action buttons give the user a quick way to perform tasks related to the notification. For example, a remote notification for a meeting invite might offer actions to accept or decline the meeting. When the user taps one of your action buttons, the system notifies your app, giving you an opportunity to perform the corresponding task. For information about how to configure actionable notifications, see Registering Your Actionable Notification Types.

The first time an app calls the `registerUserNotificationSettings:` method, iOS prompts the user to allow the specified interactions. On subsequent launches, calling this method does not prompt the user. After you call the method, iOS reports the results asynchronously to the `application:didRegisterUserNotificationSettings:` method of your app delegate. The first time you register your settings, iOS waits for the user's response before calling this method, but on subsequent calls it returns the existing user settings.

The user can change the notification settings for your app at any time using the Settings app. Because settings can change, always call the `registerUserNotificationSettings:` at launch time and use the `application:didRegisterUserNotificationSettings:` method to get the response. If the user disallows specific notification types, avoid using those types when configuring local and remote notifications for your app.

Registering for Remote Notifications

An app that wants to receive remote notifications must register with Apple Push Notification service (APNs) to get an appropriate device token. In iOS 8 and later, registration involves the following steps:

1. Register your app's supported interaction types as described in Registering Your iOS App's Supported User Interaction Types.
2. Call the `registerForRemoteNotifications` method to register your app for remote notifications. (In OS X, you use the `a target="_self" registerForRemoteNotificationTypes:/a` method to register your app's interaction types and register for remote notifications in one step.)
3. Use your app delegate's `application:didRegisterForRemoteNotificationsWithDeviceToken:` method to receive the device token needed to deliver remote notifications. Use the `application:didFailToRegisterForRemoteNotificationsWithError:` method to process errors.
4. If registration was successful, send the device token to the server you use to generate remote notifications.

iOS Note: If a cellular or Wi-Fi connection is not available, neither the `application:didRegisterForRemoteNotificationsWithDeviceToken:` method nor the `application:didFailToRegisterForRemoteNotificationsWithError:` method is called. For Wi-Fi connections, this sometimes occurs when the device cannot connect with APNs over the configured port. If this happens, the user can move to another Wi-Fi network that isn't blocking the required 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 delegate methods is called.

The device token is your key to sending push notifications to your app on a specific device. Device tokens can change, so your app needs to reregister every time it is launched and pass the received token back to your server. If you fail to update the device token, remote notifications might not make their way to the user's device. Device tokens always change when the user restores backup data to a new device or computer or reinstalls the operating system. When migrating data to a new device or computer, the user must launch your app once before remote notifications can be delivered to that device.
Never cache a device token; always get the token from the system whenever you need it. If your app previously registered for remote notifications, calling the `registerForRemoteNotifications` method again does not incur any additional overhead, and iOS returns the existing device token to your app delegate immediately. In addition, iOS calls your delegate method any time the device token changes, not just in response to your app registering or re-registering.
Listing 2-2 shows how to register for remote notifications in an iOS app. After registering the app's supported action types, the method calls the registerForRemoteNotifications method of the shared app object. Upon receiving the device token, the delegate method calls custom code to deliver that token to its parent server. In OS X, the method you use to register your interaction types is different, but the delegate methods you use to process registration are similar.

Listing 2-2 Registering for remote notifications

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

    // other setup tasks here....



    // Register the supported interaction types.

     UIUserNotificationType types = UIUserNotificationTypeBadge |

                     UIUserNotificationTypeSound |
UIUserNotificationTypeAlert;

     UIUserNotificationSettings *mySettings =

                   [UIUserNotificationSettings settingsForTypes:types
categories:nil];

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

```
   // Register for remote notifications.

    [[UIApplication sharedApplication]
registerForRemoteNotifications];

}



// Handle remote notification registration.

- (void)application:(UIApplication *)app

        didRegisterForRemoteNotificationsWithDeviceToken:(NSData
*)devToken {

    const void *devTokenBytes = [devToken bytes];

    self.registered = YES;

    [self sendProviderDeviceToken:devTokenBytes]; // custom method

}



- (void)application:(UIApplication *)app

        didFailToRegisterForRemoteNotificationsWithError:(NSError
*)err {

    NSLog(@"Error in registration. Error: %@", err);

}
```

In your `application:didFailToRegisterForRemoteNotificationsWithError:` implementation, you should process the error object appropriately and disable any features related to remote notifications. Because notifications are not going to be arriving anyway, it is usually better to degrade gracefully and avoid any unnecessary work needed to process or display those notifications.

Registering Your Actionable Notification Types

Actionable notifications let you add custom action buttons to the standard iOS interfaces for local and push notifications. Actionable notifications give the user a quick and easy way to perform relevant tasks in response to a notification. Prior to iOS 8, user notifications had only one default action. In iOS 8 and later, the lock screen, notification banners, and notification entries in Notification Center can display one or two custom actions. Modal alerts can display up to four. When the user selects a custom action, iOS notifies your app so that you can perform the task associated with that action.
Note: OS X does not support actionable notifications.

Defining Your Actionable Notifications

The configuration of custom actions depends on defining one or more categories for your notifications. Each category represents a type of notification that your app might receive, and you are responsible for defining the categories your app supports. For each category, you define the actions that a user might take when receiving a notification of that type. You then register your categories and actions with iOS using the same `registerUserNotificationSettings:` method you use to register the interaction types your app supports.
Each custom action consists of a button title and the information that iOS needs to notify your app when the action is selected. To create an action, create an instance of the `UIMutableUserNotificationAction` class and configure its properties appropriately. Listing 2-3 shows a code snippet for creating a single “accept” action. You create separate action objects for distinct action your app supports.
Listing 2-3 Defining a notification action

```
UIMutableUserNotificationAction *acceptAction =

            [[UIMutableUserNotificationAction alloc] init];



// The identifier that you use internally to handle the action.

acceptAction.identifier = @"ACCEPT_IDENTIFIER";



// The localized title of the action button.

acceptAction.title = @"Accept";
```

```
// Specifies whether the app must be in the foreground to perform
the action.

acceptAction.activationMode =
UIUserNotificationActivationModeBackground;



// Destructive actions are highlighted appropriately to indicate
their nature.

acceptAction.destructive = NO;



// Indicates whether user authentication is required to perform
the action.

acceptAction.authenticationRequired = NO;
```

After creating any custom action objects, assign those objects to the `UIUserNotificationCategory` objects you use to define your app's notification categories. When configuring categories at launch time, you typically create an instance of the `UIMutableUserNotificationCategory` class. Assign the category identifier to your new instance and use the `setActions:forContext:` method to associate any custom actions with that category. The context parameter lets you specify different sets of actions for different types of system interfaces. The default context supports four actions and is used when displaying messages in modal alerts. The minimal context supports only two actions and is used by the lock screen, notification banners, and Notification Center.
Listing 2-4 shows the creation of a an invitation category that includes the Accept action from Listing 2-3 and two additional actions whose implementation is not shown. In this example, the minimal context displays only the accept and decline buttons. If you do not specify actions for the minimal context, the first two actions of the default context are shown. The order in which you specify the buttons determines the order in which they are displayed onscreen.

Listing 2-4 Grouping actions into categories

```
// First create the category

UIMutableUserNotificationCategory *inviteCategory =

        [[UIMutableUserNotificationCategory alloc] init];



// Identifier to include in your push payload and local
notification
```

```
inviteCategory.identifier = @"INVITE_CATEGORY";


// Set the actions to display in the default context

[inviteCategory setActions:@[acceptAction, maybeAction,
declineAction]

    forContext:UIUserNotificationActionContextDefault];


// Set the actions to display in a minimal context

[inviteCategory setActions:@[acceptAction, declineAction]

    forContext:UIUserNotificationActionContextMinimal];
```

After you define your notification action categories, you register them as shown in Listing 2-5. Apps can define any number of categories, but each category must be unique.

Listing 2-5 Registering notification categories

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


UIUserNotificationSettings *settings =

       [UIUserNotificationSettings settingsForTypes:types
categories:categories];


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

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

Scheduling an Actionable Notification

To show the notification actions that you defined, categorized, and registered, you must push a remote notification or schedule a local

notification. In the remote notification case, you need to include the category identifier in your push payload, as shown in Listing 2-6. Support for categories is a collaboration between your iOS app and your push notification server. When your push server wants to send a notification to a user, it can add a category key with an appropriate value to the notification's payload. When iOS sees a push notification with a category key, it looks up the categories that were registered by the app. If iOS finds a match, it displays the corresponding actions with the notification.
The push payload size limit for the HTTP/2 provider API, available starting in December 2015, is 4KB. (In 2014, Apple increased the push payload size limit from 256 bytes to 2KB in the now-legacy binary interface.) See The Remote Notification Payload for details about the remote-notification payload.

Listing 2-6 Push payload including category identifier

```
{

    "aps" :  {

        "alert" : "You're invited!",

        "category" : "INVITE_CATEGORY"

    }

}
```

In the case of a local notification, you create the notification as usual, then set the category of the actions to be presented, and finally, schedule the notification as usual, as shown in Listing 2-7.

Listing 2-7 Defining a category of actions for a local notification

```
UILocalNotification *notification = [[UILocalNotification alloc]
init];

. . .

notification.category = @"INVITE_CATEGORY";

[[UIApplication sharedApplication]
scheduleLocalNotification:notification];
```

Handling an Actionable Notification

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

remote notification case, the system also calls `application:didReceiveRemoteNotification:fetchCompletionHandler:`.
If your app is already in the foreground, iOS does not show the notification. Instead, to handle the default action, it calls one of the `UIApplicationDelegate` methods `application:didReceiveLocalNotification:` or `application:didReceiveRemoteNotification:fetchCompletionHandler:`. (If you don't implement `application:didReceiveRemoteNotification:fetchCompletionHandler:`, iOS calls `application:didReceiveRemoteNotification:`.)
Finally, to handle the custom actions available in iOS 8 or newer , you need to implement at least one of two new methods on your app delegate, `application:handleActionWithIdentifier:forRemoteNotification:completionHandler:` or `application:handleActionWithIdentifier:forLocalNotification:completionHandler:`. In either case, you receive the action identifier, which you can use to determine what action was tapped. You also receive the notification, remote or local, which you can use to retrieve any information you need to handle that action. Finally, the system passes you the completion handler, which you must call when you finish handling the action. Listing 2-8 shows an example implementation that calls a self-defined action handler method.

Listing 2-8 Handling a custom notification action

```
- (void)application:(UIApplication *) application

              handleActionWithIdentifier: (NSString *) identifier

          // either forLocalNotification: (NSDictionary *)
notification or

                    forRemoteNotification: (NSDictionary *)
notification

                        completionHandler: (void (^)())
completionHandler {



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

        [self handleAcceptActionWithNotification:notification];

    }



    // Must be called when finished
```

```
    completionHandler();

}
```

Scheduling Local Notifications

In iOS, you create a `UILocalNotification` object and schedule its delivery using the `scheduleLocalNotification:` method of `UIApplication`. In OS X, you create an `a target="_self" NSUserNotification/a` object (which includes a delivery time) and the `a target="_self" NSUserNotificationCenter/a` is responsible for delivering it appropriately. (An OS X app can also adopt the `a target="_self" NSUserNotificationCenterDelegate/a` protocol to customize the behavior of the default `NSUserNotificationCenter` object.)
Creating and scheduling local notifications in iOS requires that you perform the following steps:

1. In iOS 8 and later, register for notification types, as described in Registering Your iOS App's Supported User Interaction Types. (In OS X and earlier versions of iOS, you need register only for remote notifications.) If you already registered notification types, call `currentUserNotificationSettings` to get the types of notifications the user accepts from your app.
2. Allocate and initialize a `UILocalNotification` object.
3. Set the date and time that the operating system should deliver the notification. This is the `fireDate` property. If you set the `timeZone` property to the `NSTimeZone` object for the current locale, the system automatically adjusts the fire date when the device travels across (and is reset for) different time zones. (Time zones affect the values of date components—that is, day, month, hour, year, and minute—that the system calculates for a given calendar and date value.)
   You can also schedule the notification for delivery on a recurring basis (daily, weekly, monthly, and so on).
4. As appropriate, configure an alert, icon badge, or sound so that the notification can be delivered to users according to their preferences. (To learn about when different notification types are appropriate, see Notifications.)
   - An alert has a property for the message (the `alertBody` property) and for the title of the action button or slider (`alertAction`). Specify strings that are localized for the user's current language preference. If your notifications can be displayed on Apple Watch, assign a value to the `alertTitle` property.

- To display a number in a badge on the app icon, use the `applicationIconBadgeNumber` property.
- To play a sound, assign a sound to the `soundName` property. You can assign the filename of a nonlocalized custom sound in the app's main bundle (or data container) or you can assign `UILocalNotificationDefaultSoundName` to get the default system sound. A sound should always accompany the display of an alert message or the badging of an icon; a sound should not be played in the absence of other notification types.

5. Optionally, you can attach custom data to the notification through the `userInfo` property. For example, a notification that's sent when a CloudKit record changes includes the identifier of the record, so that a handler can get the record and update it.
6. Optionally, in iOS 8 and later, your local notification can present custom actions that your app can perform in response to user interaction, as described in Registering Your Actionable Notification Types.
7. Schedule the local notification for delivery. You schedule a local notification by calling `scheduleLocalNotification:`. The app uses the fire date specified in the `UILocalNotification` object for the moment of delivery. Alternatively, you can present the notification immediately by calling the `presentLocalNotificationNow:` method.

The method in Listing 2-9 creates and schedules a notification to inform the user of a hypothetical to-do list app about the impending due date of a to-do item. There are a couple things to note about it. For the `alertBody`, `alertAction`, and `alertTitle` properties, it fetches from the main bundle (via the `NSLocalizedString` macro) strings localized to the user's preferred language. It also adds the name of the relevant to-do item to a dictionary assigned to the `userInfo` property.

Listing 2-9 Creating, configuring, and scheduling a local notification

```
- (void)scheduleNotificationWithItem:(ToDoItem *)item interval:
(int)minutesBefore {

    NSCalendar *calendar = [NSCalendar
autoupdatingCurrentCalendar];

    NSDateComponents *dateComps = [[NSDateComponents alloc] init];

    [dateComps setDay:item.day];

    [dateComps setMonth:item.month];

    [dateComps setYear:item.year];

    [dateComps setHour:item.hour];

    [dateComps setMinute:item.minute];
```

```
    NSDate *itemDate = [calendar dateFromComponents:dateComps];


    UILocalNotification *localNotif = [[UILocalNotification alloc]
init];

    if (localNotif == nil)

        return;

    localNotif.fireDate = [itemDate
dateByAddingTimeIntervalInterval:-(minutesBefore*60)];

    localNotif.timeZone = [NSTimeZone defaultTimeZone];



    localNotif.alertBody = [NSString
stringWithFormat:NSLocalizedString(@"%@ in %i minutes.", nil),

         item.eventName, minutesBefore];

    localNotif.alertAction = NSLocalizedString(@"View Details",
nil);

    localNotif.alertTitle = NSLocalizedString(@"Item Due", nil);



    localNotif.soundName = UILocalNotificationDefaultSoundName;

    localNotif.applicationIconBadgeNumber = 1;



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

    localNotif.userInfo = infoDict;



    [[UIApplication sharedApplication]
scheduleLocalNotification:localNotif];

}
```

You can cancel a specific scheduled notification by calling `cancelLocalNotification:` on the app object, and you can cancel all

scheduled notifications by calling `cancelAllLocalNotifications`. Both of these methods also programmatically dismiss a currently displayed notification alert. For example, you might want to cancel a notification that's associated with a reminder the user no longer wants.

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

In OS X, you might write code like that shown in Listing 2-10 to create a local notification and schedule it for delivery. Note that OS X doesn't deliver a local notification if your app is currently frontmost. Also, OS X users can change their preferences for receiving notifications in System Preferences.

Listing 2-10 Creating and scheduling a local notification in OS X

```
//Create a new local notification

NSUserNotification *notification = [[NSUserNotification alloc]
init];

//Set the title of the notification

notification.title = @"My Title";

//Set the text of the notification

notification.informativeText = @"My Text";

//Schedule the notification to be delivered 20 seconds after
execution

notification.deliveryDate = [NSDate dateWithTimeIntervalSinceNow:
20];



//Get the default notification center and schedule delivery

[[NSUserNotificationCenter defaultUserNotificationCenter]
scheduleNotification:notification];
```

Handling Local and Remote Notifications

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

The notification is delivered when the app isn't running in the foreground. In this case, the system presents the notification, displaying an

alert, badging an icon, perhaps playing a sound, and perhaps displaying one or more action buttons for the user to tap.

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

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

For remote notifications, the system also calls the `application:didReceiveRemoteNotification:fetchCompletionHandler:` method of the app delegate.

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

The notification is delivered when the app is running in the foreground. The app calls the `application:didReceiveRemoteNotification:fetchCompletionHandler:` or `application:didReceiveLocalNotification:` method of the app delegate. (If `application:didReceiveRemoteNotification:fetchCompletionHandler:` isn't implemented, the system calls `application:didReceiveRemoteNotification:`.) In OS X, the system calls `a target="_self" application:didReceiveRemoteNotification:/a`.

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

- For OS X, the delegate should adopt the a target="_self" NSApplicationDelegate/a protocol and implement the a target="_self" application:didReceiveRemoteNotification:/a method.
- For iOS, the delegate should should adopt the UIApplicationDelegate protocol and implement the application:didReceiveRemoteNotification:fetchCompletionHandler: or application:didReceiveLocalNotification: methods. To handle notification actions, implement the application:handleActionWithIdentifier:forLocalNotification:completionHandler: or application:handleActionWithIdentifier:forRemoteNotification:completionHandler: methods.

The delegate for an iOS app in Listing 2-11 implements the application:didFinishLaunchingWithOptions: method to handle a local notification. It gets the associated UILocalNotification object from the launch-options dictionary using the UIApplicationLaunchOptionsLocalNotificationKey key. From the UILocalNotification object's userInfo dictionary, it accesses the to-do item that is the reason for the notification and uses it to set the app's initial context. As shown in this example, you might appropriately reset the badge number on the app icon—or remove it if there are no outstanding items—as part of handling the notification.

Listing 2-11 Handling a local notification when an app is launched

```
- (BOOL)application:(UIApplication *)app
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    UILocalNotification *localNotif =

        [launchOptions
objectForKey:UIApplicationLaunchOptionsLocalNotificationKey];

    if (localNotif) {

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

        [viewController displayItem:itemName];  // custom method

        app.applicationIconBadgeNumber =
localNotif.applicationIconBadgeNumber-1;

    }

    [window addSubview:viewController.view];

    [window makeKeyAndVisible];
```

```
    return YES;

}
```

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

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

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

Important: Delivery of remote notifications is not guaranteed, so you should not use the notification payload to deliver sensitive data or data that can't be retrieved by other means.

One example of an appropriate usage for a custom payload property is a string identifying an email account from which messages are downloaded to an email client; the app can incorporate this string in its download user-interface. Another example of custom payload property is a timestamp for when the provider first sent the notification; the client app can use this value to gauge how old the notification is.

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

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

The code in Listing 2-12 shows an implementation of the `application:didReceiveLocalNotification:` method which is called when app is running in the foreground. Here the app delegate does the same work as it does in Listing 2-11. It can access the `UILocalNotification` object directly this time because this object is an argument of the method.

Listing 2-12 Handling a local notification when an app is already running

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

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

    [viewController displayItem:itemName];  // custom method

    app.applicationIconBadgeNumber =
notification.applicationIconBadgeNumber - 1;

}
```

If you want your app to catch remote notifications that the system delivers while it is running in the foreground, the app delegate must implement the `application:didReceiveRemoteNotification:fetchCompletionHandler:` method. The delegate should begin the procedure for downloading the waiting data, message, or other item and, after this concludes, it should remove the badge from the app icon. The dictionary passed in the second parameter of this method is the notification payload; you should not use any custom properties it contains to alter your app's current context.

Scheduling Location-Based Local Notifications

In iOS 8 and later, you can create location-based local notifications, which are delivered when the user arrives at a particular geographic location. A `UILocalNotification` object can now be configured with a Core Location region object. When the user comes enters or exits the specified region, iOS delivers the notification. You can configure notifications to be delivered once or delivered every time the user crosses the region boundary.

Registering for Location-Based Local Notifications

The use of location-based local notifications requires configuring your app to support Core Location. You must configure a `CLLocationManager` object and provide a delegate object, and you must request authorization to use location services. At a minimum, request in-use authorization by calling the `requestWhenInUseAuthorization` method, as shown in Listing 2-13.

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

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

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

locMan.delegate = self;
```

```
// Request authorization to track the user's location and enable
location-based local notifications

[locMan requestWhenInUseAuthorization];
```

The first time you request authorization for location services, iOS asks the user to allow or deny your request. When prompting the user, iOS displays the explanatory text you provided in the `NSLocationWhenInUseUsageDescription` key of your app's `Info.plist` file. Inclusion of this key is mandatory and location services cannot be started if the key is not present.
Users might see location-based notification alerts even when your app is in the background or suspended. However, an app does not receive any callbacks until users interact with the alert and the app is allowed to access their location.

Handling Core Location Callbacks

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

Listing 2-14 Handling the Core Location authorization callback

```
- (void)locationManager:(CLLocationManager *)manager

                didChangeAuthorizationStatus:
(CLAuthorizationStatus)status {



    // Check status to see if the app is authorized

    BOOL canUseLocationNotifications = (status ==
kCLAuthorizationStatusAuthorizedWhenInUse);



    if (canUseLocationNotifications) {
```

```
        [self startShowingLocationNotifications]; // Custom method
defined below

    }

}
```

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

Listing 2-15 Scheduling a location-based notification

```
- (void)startShowingNotifications {



    UILocalNotification *locNotification = [[UILocalNotification
alloc] init];

    locNotification.alertBody = @“You have arrived!”;

    locNotification.regionTriggersOnce = YES;



    locNotification.region = [[CLCircularRegion alloc]

                        initWithCenter:LOC_COORDINATE

                                radius:LOC_RADIUS

                            identifier:LOC_IDENTIFIER];



    [[UIApplication sharedApplication]
scheduleLocalNotification:locNotification];

}
```

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

Next, you create a `CLCircularRegion` instance and set it on the region property of the `UILocalNotification` instance. In this case we're giving it an app-defined location coordinate with some radius so that when the user

enters this circle, this notification is triggered. This example uses a `CLCircularRegion` property, but you could also use `CLBeaconRegion` or any other type of `CLRegion` subclass.
Finally, call `scheduleLocalNotification:` on your `UIApplication` shared instance, passing this notification just like you would do for any other local user notification.

Handling Location-Based Local Notifications

Assuming that your app is suspended when the user enters the region defined in Listing 2-15, an alert is displayed that says: "You have arrived." Your app can handle that local notification in the `application:didFinishLaunchingWithOptions:` app delegate method callback. Alternatively, if your app is executing in the foreground when the user enters that region, your app delegate is called back with `application:didReceiveLocalNotification:` message.
The logic for handling a location-based notification is very similar for both the `application:didFinishLaunchingWithOptions:` and `application:didReceiveLocalNotification:` methods. Both methods provide the notification, an instance of `UILocalNotification`, which has a `region` property. If that property is not `nil`, then the notification is a location-based notification, and you can do whatever makes sense for your app. The example code in Listing 2-16 calls a hypothetical method of the app delegate named `tellFriendsUserArrivedAtRegion:`.

Listing 2-16 Handling a location-based notification

```
- (void)application:(UIApplication *)application

                        didReceiveLocalNotification:
(UILocalNotification *)notification {



    CLRegion *region = notification.region;



    if (region) {

            [self tellFriendsUserArrivedAtRegion:region];

    }

}
```

Finally, remember that the `application:didReceiveLocalNotification:` method is not called if

the user disables Core Location, which they can do at any time in the Settings app under Privacy > Location Services.

Preparing Custom Alert Sounds

For remote notifications in iOS, you can specify a custom sound that iOS plays when it presents a local or remote notification for an app. The sound files can be in the main bundle of the client app or in the `Library/Sounds` folder of the app's data container.

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

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

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

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

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

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

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

Passing the Provider the Current Language Preference (Remote Notifications)

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

Note: For more information about the `loc-key` and `loc-args` properties and client-side message localizations, see The Remote Notification Payload.

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

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

```
NSString *preferredLang = [[NSLocale preferredLanguages]
objectAtIndex:0];

const char *langStr = [preferredLang UTF8String];

[self sendProviderCurrentLanguage:langStr]; // custom method

}
```

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

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

Next
Previous

Next
Previous

Apple Push Notification Service

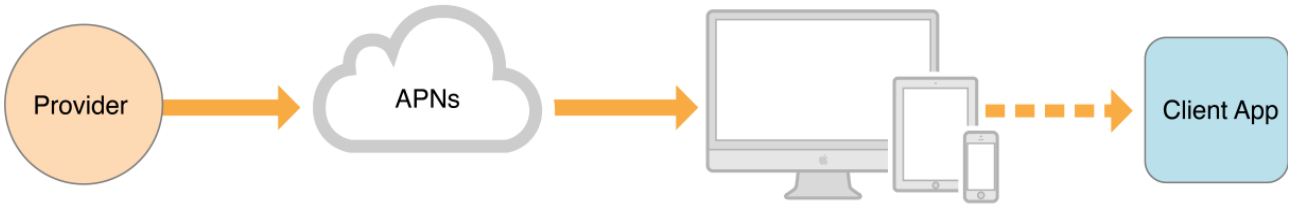
Apple Push Notification service (APNs) is the centerpiece of the remote notifications feature. It is a robust and highly efficient service for propagating information to iOS (and, indirectly, watchOS), tvOS, and OS X devices. Each device establishes an accredited and encrypted IP connection with APNs and receives notifications over this persistent connection. If a

notification for an app arrives when that app is not running, the device alerts the user that the app has data waiting for it.
You provide your own server to generate the remote notifications for your users. This server, known as the provider, gathers data for your users and decides when a notification needs to be sent. For each notification, the provider generates the notification payload and attaches that payload to an HTTP/2 request, which it then sends to APNs using a persistent and secure channel using the HTTP/2 multiplex protocol. Upon receipt of your request, APNs handles the delivery of your notification payload to your app on the user's device.
For information about the format of the requests that you send to APNs, and the responses and errors you can receive, see APNs Provider API. For information about how to implement notification support in your app, see Registering, Scheduling, and Handling User Notifications.

The Path of a Remote Notification

Apple Push Notification service transports and routes remote notifications for your apps from your provider to each user's device. Figure 3-1 shows the path each notification takes. When your provider determines that a notification is needed, you send the notification and a device token to the APNs servers. The APNs servers handle the routing of that notification to the correct user device, and the operating system handles the deliver of the notification to your client app.

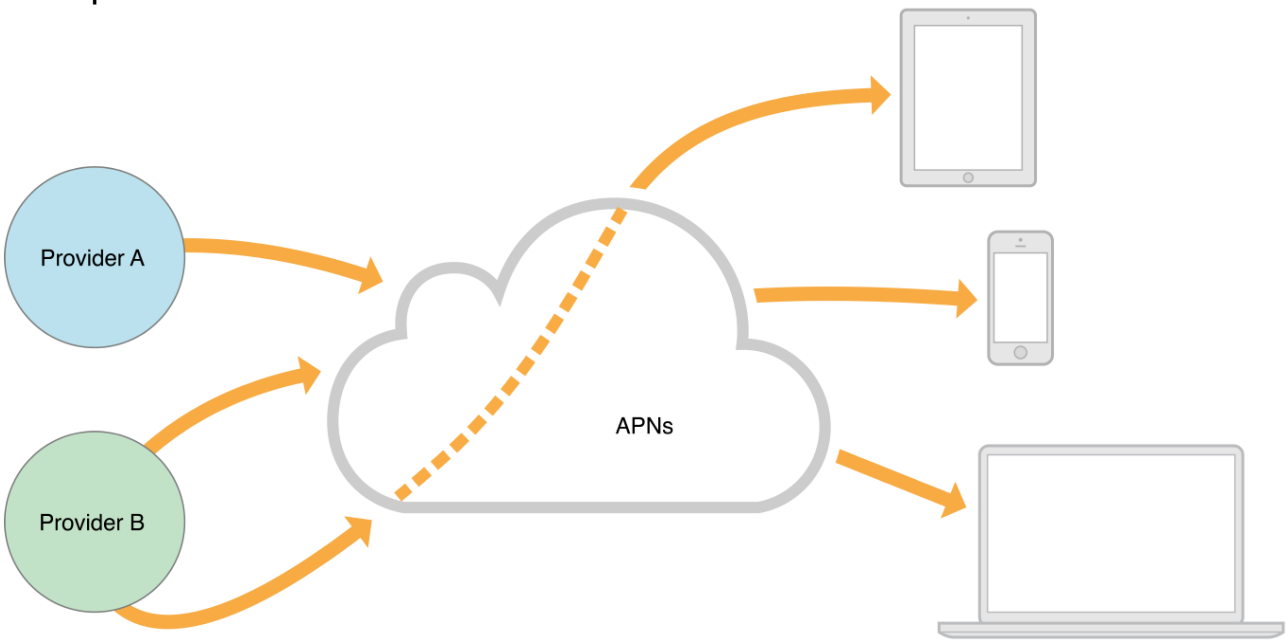
Figure 3-1 Pushing a remote notification from a provider to a client app



The device token you provide to the server is analogous to a phone number; it contains information that enables APNs to locate the device on which your client app is installed. APNs also uses it to authenticate the routing of a notification. The device token is provided to you by your client app, which receives the token after registering itself with the remote notification service.
The notification payload is a JSON dictionary containing the data you want sent to the device. The payload contains information about how you want to notify the user, such as using an alert, badge or sound. It can also contain custom data that you define.
Figure 3-2 shows a more realistic depiction of the virtual network APNs makes possible among providers and devices. The device-facing and provider-facing sides of APNs both have multiple points of connection; on the provider-facing side, these are called gateways. There are typically

multiple providers, each making one or more persistent and secure connections with APNs through these gateways. And these providers are sending notifications through APNs to many devices on which their client apps are installed.

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



For information about getting the device token, see Token Generation and Dispersal. For information about the notification payload, see The Remote Notification Payload.

Quality of Service

Apple Push Notification service includes a default Quality of Service (QoS) component that performs a store-and-forward function. If APNs attempts to deliver a notification but the device is offline, the notification is stored for a limited period of time, and delivered to the device when it becomes available. Only one recent notification for a particular app is stored. If multiple notifications are sent while the device is offline, the 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

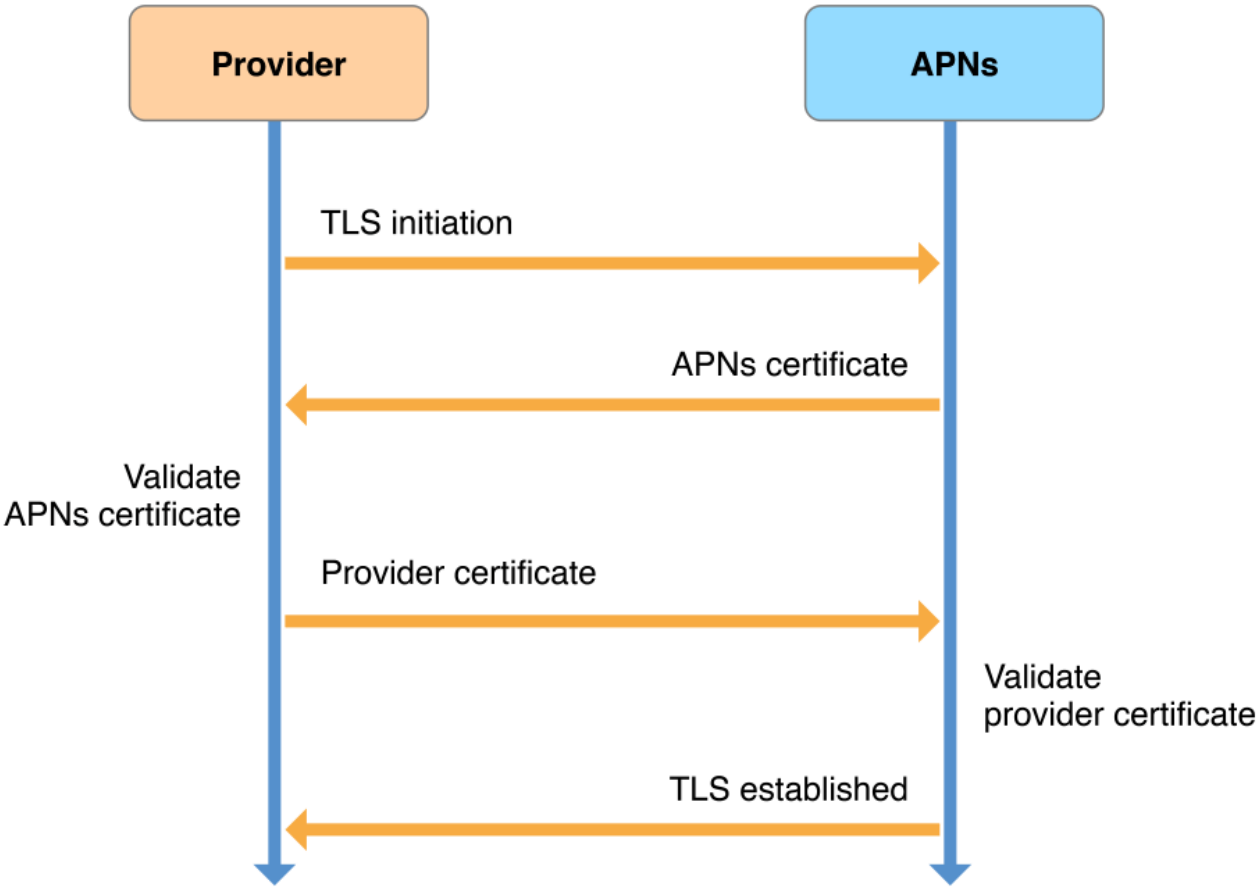
To ensure secure communication, APNs regulates the entry points between providers and devices using two different levels of trust: connection trust and token trust.

Connection trust establishes certainty that APNs is connected to an authorized provider for whom Apple has agreed to deliver notifications. APNs also uses connection trust with the device to ensure the legitimacy of that device. Connection trust with the device is handled automatically by APNs but you must take steps to ensure connection trust exists between your provider and APNs.
Token trust ensures that notifications are routed only between legitimate start and end points. Token trust involves the use of a device token, which is an opaque identifier assigned to a specific app on a specific device. Each app instance receives its unique token when it registers with APNs and must share this token with its provider. Thereafter, the token must accompany each notification sent by your provider. Providing the token ensures that the notification is delivered only to the app/device combination for which it is intended.
Note: A device token is not a unique ID that you can use to identify a device. Device tokens can change after updating the operating system on a device. As a result, apps should send their device token

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.

Provider-to-APNs Connection Trust

Each provider must have a unique provider certificate and private cryptographic key, which are used to validate the provider's connection with APNs. The provider certificate (which is provisioned by Apple) identifies the topics supported by the provider. (A topic is the bundle ID associated with one of your apps.)
Your provider establishes connection trust with APNs through TLS peer-to-peer authentication. After the TLS connection is initiated, you get the server certificate from APNs and validate that certificate on your end. Then you send your provider certificate to APNs, which validates that certificate on its end. After this procedure is complete, a secure TLS connection is established; APNs is now satisfied that the connection has been made by a legitimate provider.
Figure 3-3 Establishing connection trust between a provider and APNs



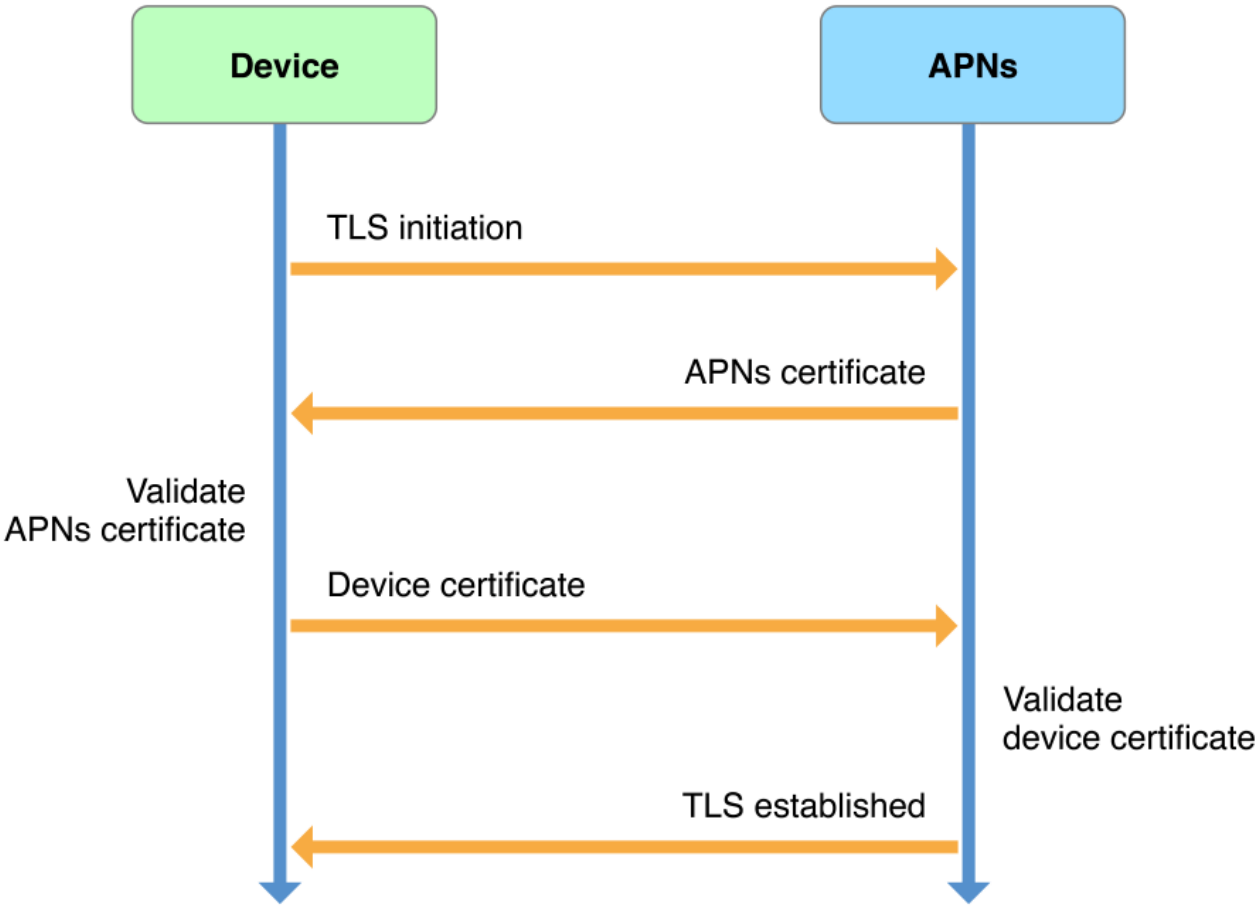
The HTTP/2 provider connection is valid for delivery to one specific app, identified by the topic (bundle ID) specified in the certificate. Depending on how you configure and provision your APNs SSL certificate, the connection can also be valid for delivery of remote notifications to any Apple Watch complications or backgrounded VoIP services associated with that primary app. See APNs Provider API for details. APNs also maintains a certificate revocation list; if a provider's certificate is on this list, APNs can revoke provider trust (that is, refuse the connection).

APNs-to-Device Connection Trust

Each device has a device certificate and private cryptographic key, which are used to validate the device's connection with APNs. The device obtains its certificate and key at device activation time and stores them in the keychain.

You do not have to do anything to establish connection trust between APNs and a device. APNs establishes the identity of a connecting device through TLS peer-to-peer authentication. In the course of this procedure, the 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.

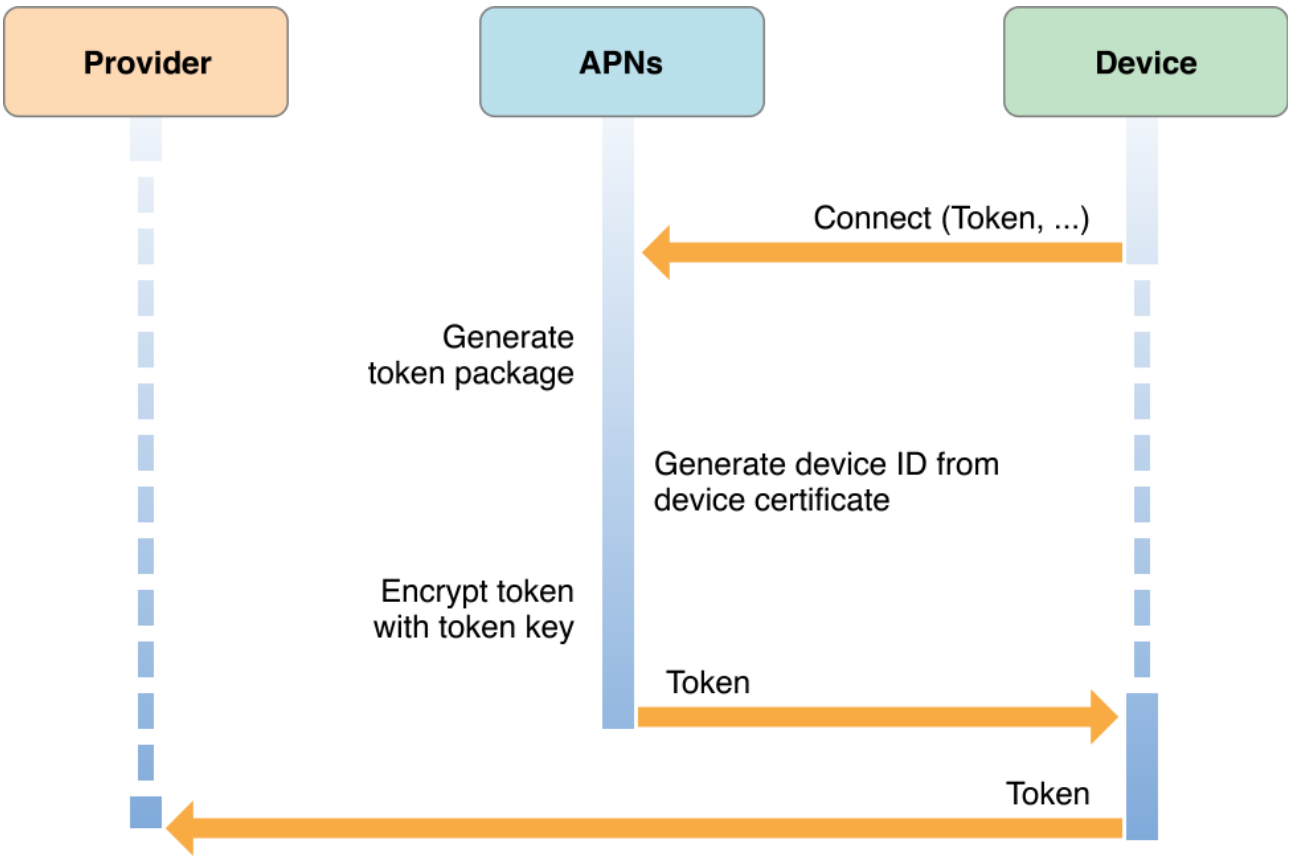
Figure 3-4 Establishing connection trust between a device and APNs



Token Generation and Dispersal

An app must register with the system to receive remote notifications, as described in Registering for Remote Notifications. Upon receiving a registration request, the system forwards the request to APNs, which generates a unique device token, for the app, using information contained in the device's certificate. It then encrypts the token using a token key and returns it to the device, as shown in Figure 3-5. The system delivers the device token to your app as an `NSData` object. Upon receiving this token, your app must forward it to your provider in either binary or hexadecimal format. Your provider cannot send notifications to the device without this token.

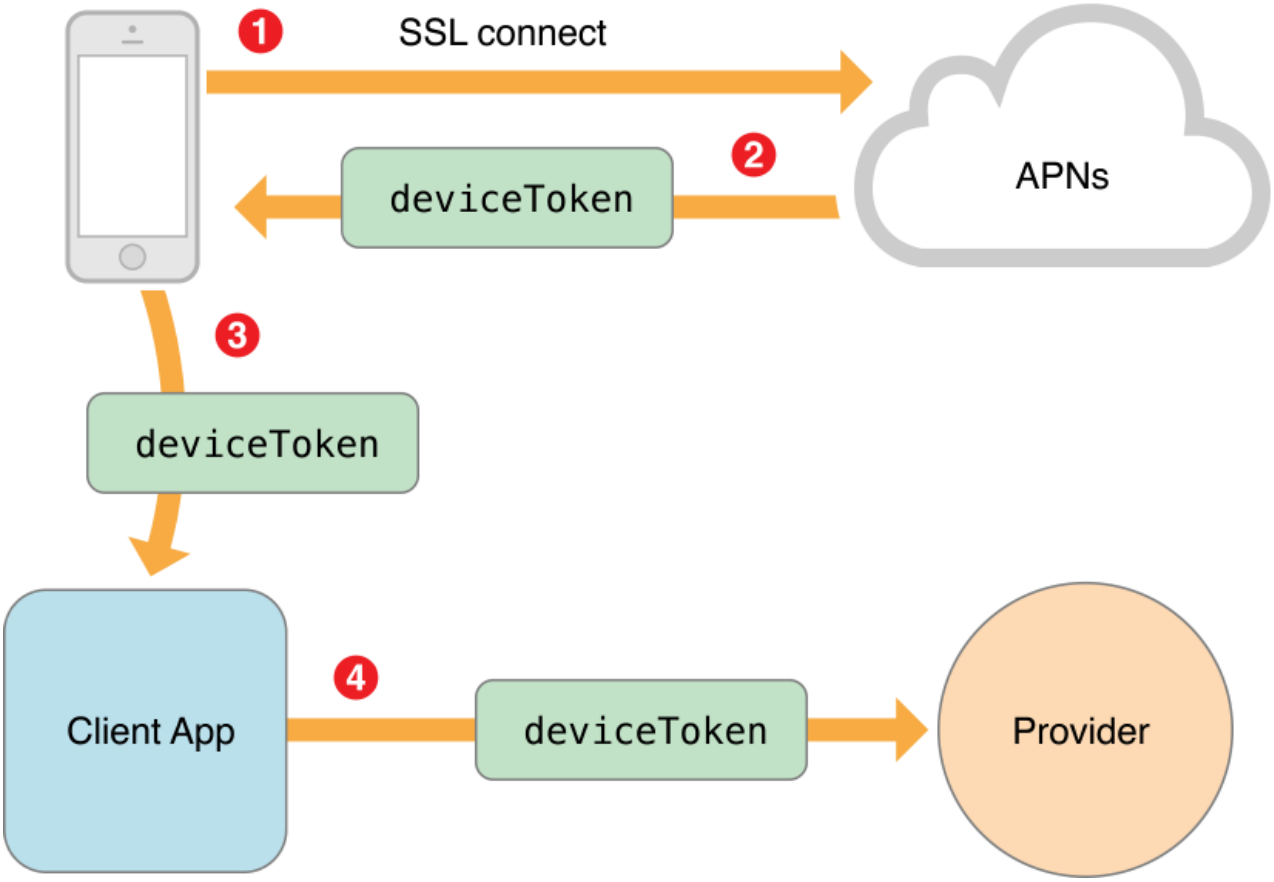
Figure 3-5 Managing the device token



Important: APNs device tokens are of variable length. Do not hardcode their size.

Figure 3-6 illustrates the token generation and dispersal sequence, but in addition shows the path of the token as it is returned by APNs and subsequently forwarded to your custom provider.

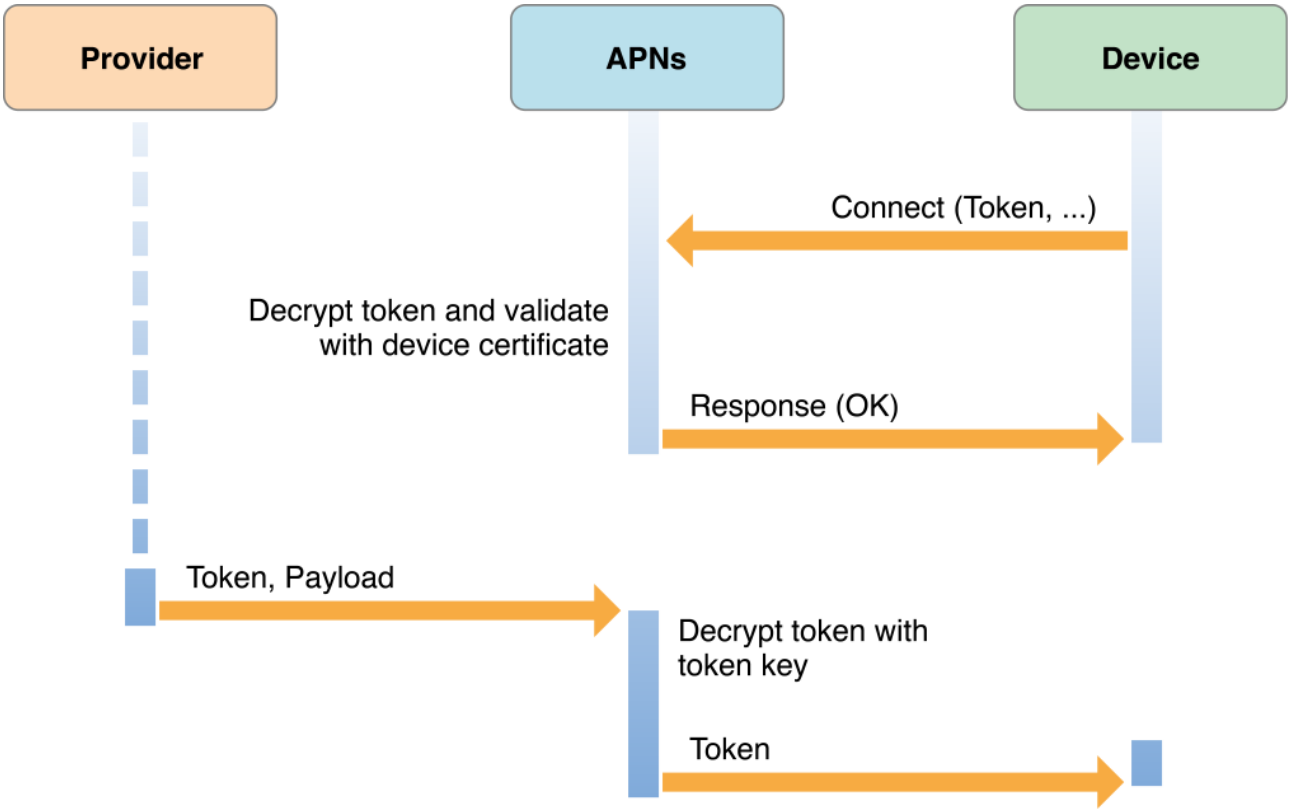
Figure 3-6 Sharing the device token



Token Trust (Notification)

Every notification that your provider sends to APNs must be accompanied by the device token associated of the device for which the notification is intended. APNs decrypts the token using its token key to ensure the validity of the notification source—that is, your provider. APNs uses the device ID contained in the device token to determine the identity of the target device. It then sends the notification to that device, as shown in Figure 3-7.

Figure 3-7 Identifying a device using the device token



Next
Previous

Provisioning and Development

Development and Production Environments

To develop and deploy the provider side of a client/server app, you must get app SSL certificates from Member Center. The HTTP/2-based provider API lets you use a single certificate for both development and production environments.
For details on obtaining an Apple Push Notification service (APNs) certificate that works in both environments, read Creating a Universal Push Notification Client SSL Certificate in App Distribution Guide.

Provisioning Procedures

APNs is available to apps distributed through the iOS App Store, tvOS App Store, and Mac App Store, as well as to enterprise apps. Your app must be provisioned and code signed to use app services. If you are working in a company, most of these configuration steps can be performed only by a team agent or admin.
To learn how to code sign and provision your app during development, read App Distribution Quick Start. For how to enable APNs, read a target="_self" Configuring Push Notifications/a in App Distribution Guide.
After you configure your app to use APNs, Xcode automatically creates the necessary distribution provisioning profiles when you export your iOS app for beta testing or submit your app to the store.

Next
Previous

The Remote Notification Payload

Each remote notification includes a payload. The payload contains information about how the system should alert the user as well as any custom data you provide. The maximum size allowed for a notification payload depends on which provider API you employ. When using the HTTP/2 provider API, maximum payload size is 4096 bytes. Using the legacy binary interface, maximum payload size is 2048 bytes. Apple Push Notification service (APNs) refuses any notification that exceeds the maximum size.

Payload Keys

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

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

To support silent remote notifications, add the `remote-notification` value to the `UIBackgroundModes` array in your `Info.plist` file. To learn more about this array, see UIBackgroundModes.
If the target app isn't running when the notification arrives, the alert message, sound, or badge value is played or shown. If the app is running, the system delivers the notification to the app delegate as an

`NSDictionary` object. The dictionary contains the corresponding Cocoa property-list objects (plus `NSNull`).
Providers can specify custom payload values outside the Apple-reserved `aps` dictionary. Custom values must use the JSON structured and primitive types: dictionary (object), array, string, number, and Boolean. You should not include customer information (or any sensitive data) as custom payload data. Instead, use it for such purposes as setting context (for the user interface) or internal metrics. For example, a custom payload value might be a conversation identifier for use by an instant-message client app or a timestamp identifying when the provider sent the notification. Any action associated with an alert message should not be destructive—for example, it should not delete data on the device.
Important: Delivery of notifications is a "best effort", not guaranteed. It is not intended to deliver data to your app, only to notify the user that there is new data available.

Table 5-1 lists the keys and expected values of the `aps` payload.
Table 5-1 Keys and values of the `aps` dictionary

`alert`
string or dictionary
If this property is included, the system displays a standard alert or a banner, based on the user's setting.
You can specify a string or a dictionary as the value of `alert`.

- If you specify a string, it becomes the message text of an alert with two buttons: Close and View. If the user taps View, the app launches.
- If you specify a dictionary, refer to Table 5-2 for descriptions of the keys of this dictionary.

The JSON `\U` notation is not supported. Put the actual UTF-8 character in the alert text instead.
`badge`
number
The number to display as the badge of the app icon.
If this property is absent, the badge is not changed. To remove the badge, set the value of this property to `0`.
`sound`
string
The name of a sound file in the app bundle or in the `Library/Sounds` folder of the app's data container. 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 for details.

`content-available`
number
Provide this key with a value of `1` to indicate that new content is available. Including this key and value means that when your app is launched in the background or resumed, `application:didReceiveRemoteNotification:fetchCompletionHandler:` is called.

`category`
string
Provide this key with a string value that represents the `identifier` property of the `UIMutableUserNotificationCategory` object you created to define custom actions. To learn more about using custom actions, see Registering Your Actionable Notification Types.
Table 5-2 lists the keys and expected values for the `alert` dictionary.
Table 5-2 Child properties of the `alert` property

`title`
string
A short string describing the purpose of the notification. Apple Watch displays this string as part of the notification interface. This string is displayed only briefly and should be crafted so that it can be understood quickly. This key was added in iOS 8.2.
`body`
string
The text of the alert message.
`title-loc-key`
string or `null`
The key to a title string in the `Localizable.strings` file for the current localization. The key string can be formatted with `%@` and `%n$@` specifiers to take the variables specified in the `title-loc-args` array. See Localized Formatted Strings for more information. This key was added in iOS 8.2.
`title-loc-args`
array of strings or `null`
Variable string values to appear in place of the format specifiers in `title-loc-key`. See Localized Formatted Strings for more information. This key was added in iOS 8.2.
`action-loc-key`
string or `null`
If a string is specified, the system displays an alert that includes the Close and View buttons. The string is used as a key to get a localized string in the current localization to use for the right button's title instead of "View". See Localized Formatted Strings for more information.

`loc-key`
string
A key to an alert-message string in a `Localizable.strings` file for the current localization (which is set by the user's language preference). The key string can be formatted with `%@` and `%n$@` specifiers to take the variables specified in the `loc-args` array. See Localized Formatted Strings 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 for more information.
`launch-image`
string
The filename of an image file in the app bundle, with or without the filename extension. The image is used as the launch image when users tap the action button or move the action slider. If this property is not specified, the system either uses the previous snapshot,uses the image identified by the `UILaunchImageFile` key in the app's `Info.plist` file, or falls back to `Default.png`.
This property was added in iOS 4.0.
Note: If you want the device to display the message text as-is in an alert that has both the Close and View buttons, then specify a string as the direct value of alert. Don't specify a dictionary as the value of `alert` if the dictionary only has the `body` property.

Configuring a Silent Notification

The `aps` dictionary can also contain the `content-available` property. The `content-available` property with a value of `1` lets the remote notification act as a silent notification. When a silent notification arrives, iOS wakes up your app in the background so that you can get new data from your server or do background information processing. Users aren't told about the new or changed information that results from a silent notification, but they can find out about it the next time they open your app.
For a silent notification, take care to ensure there is no `alert`, `sound`, or `badge` payload in the `aps` dictionary. If you don't follow this guidance, the incorrectly-configured notification might be throttled and not delivered to the app in the background, and instead of being silent is displayed to the user.

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)).
- The client app can store the alert-message strings in its bundle, translated for each localization it supports. The provider includes the `loc-key` and `loc-args` keys in the `aps` dictionary of the notification payload. (For title strings, it includes the `title-loc-key` and `title-loc-args` keys in the aps dictionary.) When the device receives the notification (assuming the app isn't running), it uses these `aps`-dictionary properties to find and format the string localized for the current language, which it then displays to the user.

Here's how that second option works in a little more detail.
An app can internationalize resources such as images, sounds, and text for each language that it supports, Internationalization collects the resources and puts them in a subdirectory of the bundle with a two-part name: a language code and an extension of `.lproj` (for example, `fr.lproj`). Localized strings that are programmatically displayed are put in a file called `Localizable.strings`. Each entry in this file has a key and a localized string value; the string can have format specifiers for the substitution of variable values. When an app asks for a particular resource—say a localized string—it gets the resource that is localized for the language currently selected by the user. For example, if the preferred language is French, the corresponding string value for an alert message would be fetched from `Localizable.strings` in the `fr.lproj` directory in the app bundle. (The app makes this request through the `NSLocalizedString` macro.)
Note: This general pattern is also followed when the value of the `action-loc-key` property is a string. This string is a key into the `Localizable.strings` in the localization directory for the currently selected language. iOS uses this key to get the title of the button on the right side of an alert message (the "action" button).

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

```
"alert" : {

    "loc-key" : "GAME_PLAY_REQUEST_FORMAT",

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

}
```

When the device receives the notification, it uses `"GAME_PLAY_REQUEST_FORMAT"` as a key to look up the associated string value in the `Localizable.strings` file in the `.lproj` directory for the

current language. Assuming the current localization has a `Localizable.strings` entry such as this:

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

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

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

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

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

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

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

Examples of JSON Payloads

The following examples of the payload portion of notifications illustrate the practical use of the properties listed in Table 5-1. Properties with “acme” in the key name are examples of custom payload data.

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

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

```
{
    "aps" : { "alert" : "Message received from Bob" },
```

```
    "acme2" : [ "bang",  "whiz" ]

}
```

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

```
{

    "aps" : {

        "alert" : {

            "title" : "Game Request",

            "body" : "Bob wants to play poker",

            "action-loc-key" : "PLAY"

        },

        "badge" : 5

    },

    "acme1" : "bar",

    "acme2" : [ "bang",  "whiz" ]

}
```

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

```
{

    "aps" : {

        "alert" : "You got your emails.",

        "badge" : 9,

        "sound" : "bingbong.aiff"

    },
```

```
    "acme1" : "bar",
    "acme2" : 42
}
```

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

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

Example 5. The payload in this example uses the `category` key to specify a group of notification actions (to learn more about categories, see Registering Your Actionable Notification Types).

```
{
   "aps" : {
      "category" : "NEW_MESSAGE_CATEGORY"
      "alert" : {
         "body" : "Acme message received from Johnny Appleseed",
         "action-loc-key" : "VIEW"
      },
      "badge" : 3,
```

```
      "sound" : “chime.aiff"

   },

   "acme-account" : "jane.appleseed@apple.com",

   "acme-message" : "message123456"

}
```

Next
Previous

Next
Previous

APNs Provider API

Apple Push Notification service includes the APNs Provider API that allows you to send remote notifications to your app on iOS, tvOS, and OS X devices, and to Apple Watch via iOS. This API is based on the HTTP/2 network protocol. Each interaction starts with a POST request, containing a JSON payload, that you send from your provider server to APNs. APNs then forwards the notification to your app on a specific user device.
Your APNs certificate, which you obtain as explained in Creating a Universal Push Notification Client SSL Certificate in App Distribution Guide, enables connection to both the APNs Production and Development environments. You can use your APNs certificate to send notifications to your primary app, as identified by its bundle ID, as well as to any Apple Watch complications or backgrounded VoIP services associated with that app. Use the `( 1.2.840.113635.100.6.3.6 )` extension in the certificate to identify the topics for your push notifications. For example, if you provide an app with the bundle ID `com.yourcompany.yourexampleapp`, you can specify the following topics in the certificate:

```
Extension ( 1.2.840.113635.100.6.3.6 )

Critical NO

Data com.yourcompany.yourexampleapp

Data app

Data com.yourcompany.yourexampleapp.voip

Data voip
```

`Data com.yourcompany.yourexampleapp.complication`

`Data complication`

Connections

The first step in sending a remote notification is to establish a connection with the appropriate APNs server:

- **Development server:** `api.development.push.apple.com:443`
- **Production server:** `api.push.apple.com:443`

Note: You can alternatively use port `2197` when communicating with APNs. You might do this, for example, to allow APNs traffic through your firewall but to block other HTTPS traffic.

Your provider server must support TLS 1.2 or higher when creating the connection to the APNs servers. In addition, the connection must be authenticated with your provider client certificate, which you obtain from Member Center, as described in Creating a Universal Push Notification Client SSL Certificate.

The APNs server allows multiple concurrent streams for each connection. The exact number of streams is based on server load, so do not assume a specific number of streams. The APNs servers ignore HTTP/2 `PRIORITY` frames, so do not send them on your streams.

Best Practices for Managing Connections

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.

You can establish multiple connections to APNs servers to improve performance. When you send a large number of remote notifications, distribute them across connections to several server endpoints. This improves performance, compared to using a single connection, by letting you send remote notifications faster and by letting APNs deliver them faster.

You can check the health of your connection using an HTTP/2 `PING` frame.

Termination

If APNs decides to terminate an established HTTP/2 connection, it sends a `GOAWAY` frame. The `GOAWAY` frame includes JSON data in its payload with a

`reason` key, whose value indicates the reason for the connection termination. For a list of possible values for the `reason` key, see Table 6-6. Normal request failures do not result in termination of a connection.

Notification API

The APNs Provider API consists of a request and a response that you configure and send using an HTTP/2 `POST` command. You use the request to send a push notification to the APNs server and use the response to determine the results of that request.

Request

Use a request to send a notification to a specific user device.

Table 6-1 HTTP/2 request

`:method`	
`POST`	
`:path`	
`/3/device/`<device-token>	

For the <device-token> parameter, specify the hexadecimal bytes of the device token for the target device.

APNs requires the use of HPACK (header compression for HTTP/2), which prevents repeated header keys and values. APNs maintains a small dynamic table for HPACK. To help avoid filling up the APNs HPACK table and necessitating the discarding of table data, encode headers in the following way—especially when sending a large number of streams:

- The `:path` header should be encoded as a literal header field without indexing,
- The `apns-id` and `apns-expiration` headers should be encoded differently depending on initial or subsequent POST operation, as follows:
    - The first time you send these headers, encode them with incremental indexing to allow the header names to be added to the dynamic table
    - Subsequent times you send these headers, encode them as literal header fields without indexing

Encode all other headers as literal header fields with incremental indexing. For specifics on header encoding, see tools.ietf.org/html/rfc7541#section-6.2.1 and tools.ietf.org/html/rfc7541#section-6.2.2.

APNs ignores headers other than the ones in Table 6-2.

Table 6-2 Request headers

`apns-id`
A canonical UUID that identifies the notification. If there is an error sending the notification, APNs uses this value to identify the notification to your server.
The canonical form is 32 lowercase hexadecimal digits, displayed in five groups separated by hyphens in the form 8-4-4-4-12. An example UUID is as follows:
`123e4567-e89b-12d3-a456-42665544000`
If you omit this header, a new UUID is created by APNs and returned in the response.
`apns-expiration`
A UNIX epoch date expressed in seconds (UTC). This header identifies the date when the notification is no longer valid and can be discarded.
If this value is nonzero, APNs stores the notification and tries to deliver it at least once, repeating the attempt as needed if it is unable to deliver the notification the first time. If the value is `0`, APNs treats the notification as if it expires immediately and does not store the notification or attempt to redeliver it.
`apns-priority`
The priority of the notification. Specify one of the following values:

- `10`–Send the push message immediately. Notifications with this priority must trigger an alert, sound, or badge on the target device. It is an error to use this priority for a push notification that contains only the `content-available` key.
- `5`—Send the push message at a time that takes into account power considerations for the device. Notifications with this priority might be grouped and delivered in bursts. They are throttled, and in some cases are not delivered.

If you omit this header, the APNs server sets the priority to `10`.
`apns-topic`
The topic of the remote notification, which is typically the bundle ID for your app. The certificate you create in Member Center must include the capability for this topic.
If your certificate includes multiple topics, you must specify a value for this header.
If you omit this header and your APNs certificate does not specify multiple topics, the APNs server uses the certificate's Subject as the default topic.
The body content of your message is the JSON dictionary object containing the notification data. The body data must not be compressed and its maximum size is 4KB (4096 bytes). For information about the keys and values to include in the body content, see The Remote Notification Payload.

Response

The response to a request has the format listed in Table 6-3.

Table 6-3 Response headers

`apns-id`
The `apns-id` value from the request. If no value was included in the request, the server creates a new UUID and returns it in this header.
`:status`
The HTTP status code. For a list of possible status codes, see Table 6-4.
Table 6-4 lists the possible status codes for a request. These values are included in the `:status` header of the response.
Table 6-4 Status codes for a response

200
Success
400
Bad request
403
There was an error with the certificate.
405
The request used a bad `:method` value. Only `POST` requests are supported.
410
The device token is no longer active for the topic.
413
The notification payload was too large.
429
The server received too many requests for the same device token.
500
Internal server error
503
The server is shutting down and unavailable.
For a successful request, the body of the response is empty. On failure, the response body contains a JSON dictionary with the keys listed in Table 6-5. This JSON data might also be included in the `GOAWAY` frame when a connection is terminated.
Table 6-5 JSON data keys

`reason`
The error indicating the reason for the failure. The error code is specified as a string. For a list of possible values, see Table 6-6.
`timestamp`
If the value in the `:status` header is `410`, the value of this key is the last time at which APNs confirmed that the device token was no longer valid for the topic.

Stop pushing notifications until the device registers a token with a later timestamp with your provider.
Table 6-6 lists the possible error codes included in the `reason` key of a response's JSON payload.

Table 6-6 Values for the `reason` key

`PayloadEmpty`
The message payload was empty.
Expected HTTP/2 status code is `400`; see Table 6-4.

`PayloadTooLarge`
The message payload was too large. The maximum payload size is `4096` bytes.

`BadTopic`
The `apns-topic` was invalid.

`TopicDisallowed`
Pushing to this topic is not allowed.

`BadMessageId`
The `apns-id` value is bad.

`BadExpirationDate`
The `apns-expiration` value is bad.

`BadPriority`
The `apns-priority` value is bad.

`MissingDeviceToken`
The device token is not specified in the request `:path`. Verify that the `:path` header contains the device token.

`BadDeviceToken`
The specified device token was bad. Verify that the request contains a valid token and that the token matches the environment.

`DeviceTokenNotForTopic`
The device token does not match the specified topic.

`Unregistered`
The device token is inactive for the specified topic.
Expected HTTP/2 status code is `410`; see Table 6-4.

`DuplicateHeaders`
One or more headers were repeated.

`BadCertificateEnvironment`
The client certificate was for the wrong environment.

`BadCertificate`
The certificate was bad.

`Forbidden`
The specified action is not allowed.

`BadPath`
The request contained a bad `:path` value.

`MethodNotAllowed`

The specified `:method` was not `POST`.

`TooManyRequests`

Too many requests were made consecutively to the same device token.

`IdleTimeout`

Idle time out.

`Shutdown`

The server is shutting down.

`InternalServerError`

An internal server error occurred.

`ServiceUnavailable`

The service is unavailable.

`MissingTopic`

The `apns-topic` header of the request was not specified and was required. The apns-topic header is mandatory when the client is connected using a certificate that supports multiple topics.

Examples

Listing 6-1 shows a sample request constructed for a certificate.

Listing 6-1 Sample request for a certificate with a single topic

```
HEADERS

  - END_STREAM

  + END_HEADERS

  :method = POST

  :scheme = https

  :path = /3/device/
00fc13adff785122b4ad28809a3420982341241421348097878e577c991de8f0

  host = api.development.push.apple.com

  apns-id = eabeae54-14a8-11e5-b60b-1697f925ec7b

  apns-expiration = 0

  apns-priority = 10

DATA

  + END_STREAM

    { "aps" : { "alert" : "Hello" } }
```

Listing 6-2 shows a sample request constructed for a certificate that contains multiple topics.

Listing 6-2 Sample request for a certificate with multiple topics

```
HEADERS

  - END_STREAM

  + END_HEADERS

  :method = POST

  :scheme = https

  :path = /3/device/
00fc13adff785122b4ad28809a3420982341241421348097878e577c991de8f0

  host = api.development.push.apple.com

  apns-id = eabeae54-14a8-11e5-b60b-1697f925ec7b

  apns-expiration = 0

  apns-priority = 10

  apns-topic = <MyAppTopic>

DATA

  + END_STREAM

    { "aps" : { "alert" : "Hello" } }
```

Listing 6-3 shows a sample response for a successful push request.

Listing 6-3 Sample response for a successful request

```
HEADERS

  + END_STREAM

  + END_HEADERS

  :status = 200
```

Listing 6-4 shows a sample response when an error occurs.

Listing 6-4 Sample response for a request that encountered an error

```
HEADERS

  - END_STREAM

  + END_HEADERS

  :status = 400

  content-type = application/json
```

```
    apns-id: <a_UUID>

DATA

  + END_STREAM

  { "reason" : "BadDeviceToken" }
```

Next
Previous

Next
Previous

Binary Provider API

The legacy binary interface required your provider server to employ the binary API described in this appendix. All developers should migrate their remote notification provider servers to the more capable and more efficient HTTP/2-based API described in APNs Provider API.

General Provider Requirements

As a provider, you can 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 interface is supported, but you should prefer the use of the APNs Provider API if possible.

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

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

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

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

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

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

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 A-1.

Figure A-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:

Table A-1 Top-level fields for remote notifications

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:
Table A-2 Fields for remote notification frames

Item ID	1 byte The item identifier, as listed in Table A-3. For example, the item identifier of the payload is 2.
Item data length	2 bytes The size of the item data.
Item data	variable length The value for the item.

The items and their identifiers are as follows:
Table A-3 Item identifiers for remote notifications

1	Device token	32 bytes The device token in binary form, as was registered by the device. A remote notification must have exactly one device token.
2	Payload	variable length, less than or equal to 2 kilobytes The JSON-formatted payload. A remote notification must have exactly one 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. Although this item is not required, you should include it to allow APNs to restart the sending of notifications upon encountering an error.
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.
5
Priority
1 byte
The notification's priority. Provide one of the following values:

- `10` The push message is sent immediately.The remote notification must trigger an alert, sound, or badge on the device. It is an error to use this priority for a push that contains only the `content-available` key.
- `5` The push message is sent at a time that conserves power on the device receiving it.Notifications with this priority might be grouped and delivered in bursts. They are throttled, and in some cases are not delivered.

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 A-2 shows the format of the error-response packet.
Figure A-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 A-4 lists the possible status codes and their meanings.
Table A-4 Codes in error-response packet

0
No errors encountered
1
Processing error
2
Missing device token
3
Missing topic
4
Missing payload
5
Invalid token size
6
Invalid topic size

7
Invalid payload size
8
Invalid token
10
Shutdown
128
Protocol error (APNs could not parse the notification)
255
None (unknown)
A status code of 10 indicates that the APNs server closed the connection (for example, to perform maintenance). The notification identifier in the error response indicates the last notification that was successfully sent. Any notifications you sent after it have been discarded and must be resent. When you receive this status code, stop using this connection and open a new connection.
Take note that the device token in the production environment and the device token in the development environment are not the same value.

The Feedback Service

The Apple Push Notification service includes a feedback service to give you information about failed remote notifications. When a remote notification cannot be delivered because the intended app does not exist on the device, the feedback service adds that device's token to its list. Remote notifications that expire before being delivered are not considered a failed delivery and don't impact the feedback service. By using this information to stop sending remote notifications that will fail to be delivered, you reduce unnecessary message overhead and improve overall system performance. Query the feedback service daily to get the list of device tokens. Use the timestamp to verify that the device tokens haven't been reregistered since the feedback entry was generated. For each device that has not been reregistered, stop sending notifications. APNs monitors providers for their diligence in checking the feedback service and refraining from sending remote notifications to nonexistent apps on devices.
Note: The feedback service maintains a separate list for each push topic. If you have multiple apps, you must connect to the feedback service once for each app, using the corresponding certificate, in order to receive all feedback.

The feedback service has a binary interface similar to the interface used for sending remote notifications. You access the production feedback service via `feedback.push.apple.com` on port 2196 and the development feedback service via `feedback.sandbox.push.apple.com` on port 2196. As with the binary interface for remote notifications, use TLS (or SSL) to

establish a secured communications channel. You use the same SSL certificate for connecting to the feedback service as you use for sending notifications. To establish a trusted provider identity, present this certificate to APNs at connection time using peer-to-peer authentication.
Once you are connected, transmission begins immediately; you do not need to send any command to APNs. Read the stream from the feedback service until there is no more data to read. The received data is in tuples with the following format:
Figure A-3 Binary format of a feedback tuple

Table A-5 Table
Timestamp
A timestamp (as a four-byte `time_t` value) indicating when APNs determined that the app no longer exists on the device. This value, which is in network order, represents the seconds since 12:00 midnight on January 1, 1970 UTC.
Token length
The length of the device token as a two-byte integer value in network order.
Device token
The device token in binary format.
The feedback service's list is cleared after you read it. Each time you connect to the feedback service, the information it returns lists only the failures that have happened since you last connected.
Next
Previous

Next
Previous
Legacy Notification Format

New development should use the modern format to connect to APNs, as described in APNs Provider API.

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

Figure B-1 shows this format.
Figure B-1 Legacy 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 B-1 gives an example of a function that sends a remote notification to APNs over the binary interface using the legacy notification format. The example assumes prior SSL connection to `gateway.push.apple.com` (or `gateway.sandbox.push.apple.com`) and peer-exchange authentication.

Listing B-1 Sending a notification in the legacy 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 legacy format:

- **Error response**. With the legacy format, if you send a notification packet that is malformed in some way—for example, the payload exceeds the stipulated limit—APNs responds by severing the connection. It gives no indication why it rejected the notification. The

enhanced format lets a provider tag a notification with an arbitrary identifier. If there is an error, APNs returns a packet that associates an error code with the identifier. This response enables the provider to locate and correct the malformed notification.

- **Notification expiration**. APNs has a store-and-forward feature that keeps the most recent notification sent to an app on a device. If the device is offline at time of delivery, APNs delivers the notification when the device next comes online. With the legacy 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 B-2 depicts the format for notification packets.

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

Listing B-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;

}
```

Next
Previous