Push Notifications Tutor Started

József Vesza on June 5, 2017

Note: This tutorial was updated to Xcode 8.3 and Swift 3.1 by József Vesza. The original tutorial was written by Jack Wu.

iOS developers love to imagine users of their awesome app using the app all day, every day. Unfortunately, the cold hard truth is that users will sometimes have to close the app and perform other activities. Laundry doesn't fold itself, you know :]

Happily, **push notifications** allow developers to reach users and perfor using an app!

Push notifications have become more and more powerful since they we can:

- Display a short text message
- Play a notification sound
- Set a badge number on the app's icon
- Provide actions the user can take without opening the app
- Show a media attachment
- Be silent, allowing the app to wake up in the background and perfo

This push notifications tutorial will go over how push notifications work,

Before you get started, you will need the following to test push notificati

- **An iOS device**. Push notifications do not work in the simulator, so
- **An Apple Developer Program Membership**. Since Xcode 7, you ca membership. However, to configure push notifications you need a which requires the program membership.
- **Pusher**. you'll use this utility app to send notifications to the device

Getting Started

There are three main tasks that must be performed in order to send and receive a push notification:

1. The app must be configured properly and registered with the Apple Push Notification Service (APNS) to receive push notifications upon every start-up.
2. A server must send a push notification to APNS directed to one or more specific devices.
3. The app must receive the push notification; it can then perform tasks or handle user actions using callbacks in the application delegate.

Tasks 1 and 3 will be the main focus of this push notifications tutorial, since they are the responsibility of an iOS

developer.

Task 2 will also be briefly covered, mostly for testing purposes. Sending push notifications is a responsibility of the app's server-component and is usually implemented differently from one app to the next. Many apps use third-parties (you can find some good examples here) to send push notifications, while others use custom solutions and/or popular libraries (ex. Houston).

To get started, download the starter project of **WenderCast**. WenderCast is everyone's go-to source for raywenderlich.com podcasts and breaking news.

Open **WenderCast.xcodeproj** in Xcode and take a peek around. Build and run within the iPhone simulator to see the latest podcasts (you'll use a real device soon!):

The problem with the app is that it doesn't let users know when a new podcast is available. It also doesn't really have any news to display. You'll soon fix all that with the power of push notifications!

Configuring the Push Notifications Tutorial App

Push notifications require a *lot* of security. This is quite important, since you don't want anyone else to send push notifications to your users. Unfortunately, this means there's several required tasks to configure apps for push notifications.

Enabling the Push Notification Service

The first step is to change the App ID. Go to **App Settings -> General** and change **Bundle Identifier** to something unique:

Within **Signing** right below this, select your development **Team**. Again, this must be a paid developer account. If you don't see any teams, you'll first need to add your development team via **Xcode -> Preferences -> Accounts -> +**.

Next, you need to create an App ID in your developer account that has the push notification entitlement enabled. Luckily, Xcode has a simple way to do this. Go to **App Settings -> Capabilities** and flip the switch for **Push Notifications** to **On**.

After some loading, it should look like this:

If any issues occur, visit the Apple Developer Center. You may simply need to agree to a new developer license, which Apple loves to update ;], and try again. Worse case, you may need to manually add the push notifications entitlement by using the **+** and **Edit** buttons.

Behind the scenes, this creates the App ID and then adds the push notifications entitlement to it. You can log into the Apple Developer Center and verify this:

That's all you need to configure for now.

Registering for Push Notifications

There are two steps to register for push notifications. First, you must obtain the user's permission to show *any* kind of notification, after which you can register for remote notifications. If all goes well, the system will then provide you with a **device token**, which you can think of as an "address" to this device.

In WenderCast, you will register for push notifications immediately after the app launches.

Open **AppDelegate.swift** and add the following import to the top of the file:

```
import UserNotifications
```

Then add the following method to the end of **AppDelegate**:

```
func registerForPushNotifications() {
  UNUserNotificationCenter.current().requestAuthorization(options: [.alert, .sound,
.badge]) {
    (granted, error) in
    print("Permission granted: \(granted)")
  }
}
```

Lastly, add a call to **registerForPushNotifications()** at the end of **application(_:didFinishLaunchingWithOptions:)**:

```
func application(
  _ application: UIApplication,
  didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) ->
Bool {
  // ... existing code ...
  registerForPushNotifications()
  return true
}
```

Let's go over the above: **UNUserNotificationCenter** was introduced in iOS 10 within the UserNotifications framework. It's responsible for managing all notification-related activities within the app.

You invoke **requestAuthorization(options:completionHandler:)** to (you guessed it) request authorization for push notifications. Here, you must specify the notification types your app will use. These types (represented by **UNAuthorizationOptions**) can be any combination of the following:

- **.badge** allows the app to display a number on the corner of the app's icon.
- **.sound** allows the app to play a sound.
- **.alert** allows the app to display text.
- **.carPlay** allows the app to display notifications in a CarPlay environment.

You call **registerForPushNotifications** within **application(_:didFinishLaunchingWithOptions:)** to ensure the demo app will attempt to register for push notifications any time it's launched.

Build and run. When the app launches, you should receive a prompt that asks for permission to send you notifications.

Tap **OK** and poof! The app can now display notifications. Great! But what now? What if the user declines the permissions? Add this method inside **AppDelegate**:

```
func getNotificationSettings() {
  UNUserNotificationCenter.current().getNotificationSettings { (settings) in
    print("Notification settings: \(settings)")
  }
}
```

This method is *very different* from the previous one. In the previous method, you specified the settings you *want*, yet this one returns the settings the user has *granted*.

It's important to call **getNotificationSettings(completionHandler:)** within the completion handler on **requestAuthorization**, which happens whenever the app finishes launching. This is because the user can, at any time, go into the Settings app and change the notification permissions.

Update **requestAuthorization** to call **getNotificationSettings()** within the completion closure like this:

```
func registerForPushNotifications() {
```

```
  UNUserNotificationCenter.current().requestAuthorization(options: [.alert, .sound,
.badge]) {
    (granted, error) in
    print("Permission granted: \(granted)")

    guard granted else { return }
    self.getNotificationSettings()
  }
}
```

Step 1 is now complete, and you're now ready to actually register for remote notifications!

Update **`getNotificationSettings()`** with the following:

```
func getNotificationSettings() {
  UNUserNotificationCenter.current().getNotificationSettings { (settings) in
    print("Notification settings: \(settings)")
    guard settings.authorizationStatus == .authorized else { return }
    UIApplication.shared.registerForRemoteNotifications()
  }
}
```

Here you verify the **`authorizationStatus`** is **`.authorized`**, meaning the user has granted notification permissions, and if so, you call **`UIApplication.shared.registerForRemoteNotifications()`**.

Add the following two methods to then end of **`AppDelegate`**; these will be called to inform you about the result of **`registerForRemoteNotifications`**:

```
func application(_ application: UIApplication,
                 didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data) {
  let tokenParts = deviceToken.map { data -> String in
    return String(format: "%02.2hhx", data)
  }

  let token = tokenParts.joined()
  print("Device Token: \(token)")
}

func application(_ application: UIApplication,
                 didFailToRegisterForRemoteNotificationsWithError error: Error) {
  print("Failed to register: \(error)")
}
```

As the names suggest, the system calls **`application(_:didRegisterForRemoteNotificationsWithDeviceToken:)`** if the registration is successful, or else it calls **`application(_:didFailToRegisterForRemoteNotificationsWithError:)`**.

The current implementation of **`application(_:didRegisterForRemoteNotificationsWithDeviceToken:)`** looks cryptic, but it is simply taking **`deviceToken`** and converting it to a string. The device token is the fruit of this process. It is a token provided by APNS that uniquely identifies this app on this particular device. When sending a push notification, the app uses device tokens as "addresses" to deliver notifications to the correct devices.

> **Note:** There are several reasons why registration might fail. Most of the time it's because the app is running on a simulator, or because the App ID configuration was not done properly. The error message generally provides a good hint for what's wrong.

That's it! Build and run. Make sure you are running on a device, and you should receive a device token in the console output. Here's what mine looks like:

Copy this token somewhere handy.

You have a bit more configuration to do before you can send a push notification, so head over to the Apple Developer Member Center and log in.

Creating an SSL Certificate and PEM file

In your member center, go to **Certificates, IDs & Profiles -> Identifiers -> App IDs** and select the App ID for your app. Under **Application Services**, Push Notifications should show as **Configurable**:

Click **Edit** and scroll down to **Push Notifications**:

In **Development SSL Certificate**, click **Create Certificate...** and follow the steps to create a **CSR**. Once you have your CSR, click **continue** and follow the steps to **Generate** your certificate using the CSR. Finally, download the certificate and double-click it, which should add it to your Keychain, paired with a private key:

Back in the member center, your App ID should now have push notifications enabled for development:

Whew! That was a lot to get through, but it was all worth it — with your new certificate file, you are now ready to send your first push notification!

Sending a Push Notification

Sending push notifications requires an SSL connection to APNS, secured by the push certificate you just created. That's where **Pusher** comes in.

Launch **Pusher**. The app will automatically check for push certificates in the Keychain, and list them in a dropdown. Complete the following steps:

- Select your push certificate from the dropdown.
- Paste your device token into the "Device push token" field.
- Modify the request body to look like this:

```
{
  "aps": {
    "alert": "Breaking News!",
    "sound": "default",
    "link_url": "https://raywenderlich.com"
  }
}
```

- On the device you previously ran WenderCast on, background the app or lock the device, or else it won't work*****
- Click the **Push** button in Pusher.

You should receive your first push notification:

***Note:** You won't see anything if the app is open and running in the foreground. The notification is delivered, but there's nothing in the app to handle it yet. Simply close the app and send the notification again.

Common Issues

There are a couple problems that might arise:

Some notifications received but not all: If you're sending multiple push notifications simultaneously and only a few are received, fear not! That is intended behaviour. APNS maintains a QoS (Quality of Service) queue for each device with a push app. The size of this queue is 1, so if you send multiple notifications, the last notification is overridden.

Problem connecting to Push Notification Service: One possibility could be that there is a firewall blocking the ports used by APNS. Make sure you unblock these ports. Another possibility might be that the private key and CSR file are wrong. Remember that each App ID has a unique CSR and private key combination.

Anatomy of a Basic Push Notification

Before you move on to Task 3, handling push notifications, take a look at the body of the notification you've just sent:

```
{
  "aps": {
    "alert": "Breaking News!",
    "sound": "default",
    "link_url": "https://raywenderlich.com"
  }
}
```

For the JSON-uninitiated, a block delimited by curly { } brackets contains a dictionary that consists of key/value pairs (just like a Swift `Dictionary`).

The payload is a dictionary that contains at least one item, **aps**, which itself is also a dictionary. In this example, "aps" contains the fields `alert`, `sound`, and `link_url`. When this push notification is received, it shows an alert view with the text "Breaking News!" and plays the standard sound effect.

`link_url` is actually a custom field. You can add custom fields to the payload like this and they will get delivered to your application. Since you aren't handling it inside the app yet, this key/value pair currently does nothing.

There are six keys you can add to the `aps` dictionary:

- `alert`. This can be a string, like in the previous example, or a dictionary itself. As a dictionary, it can localize the text or change other aspects of the notification.
- `badge`. This is a number that will display in the corner of the app icon. You can remove the badge by setting this to 0.
- `thread-id`. You may use this key for grouping notifications.
- `sound`. By setting this key, you can play custom notification sounds located in the app in place of the default notification sound. Custom notification sounds must be shorter than 30 seconds and have a few restrictions.
- `content-available`. By setting this key to `1`, the push notification becomes a silent one. This will be explored later in this push notifications tutorial.
- `category`. This defines the category of the notification, which is is used to show custom actions on the notification. You will also be exploring this shortly.

Outside of these, you can add as much custom data as you want, as long as the payload does not exceed the maximum size of 4096 bytes.

Once you've had enough fun sending push notifications to your device, move on to the next section. :]

Handling Push Notifications

In this section, you'll learn how to perform actions in your app when push notifications are received and/or when users tap on them.

What Happens When You Receive a Push Notification?

When your app receives a push notification, a method in **`UIApplicationDelegate`** is called.

The notification needs to be handled differently depending on what state your app is in when it's received:

- If your app wasn't running and the user launches it by tapping the push notification, the push notification is passed to your app in the **`launchOptions`** of **`application(_:didFinishLaunchingWithOptions:)`**.
- If your app was running either in the foreground, or the background, **`application(_:didReceiveRemoteNotification:fetchCompletionHandler:)`** will be called. If the user opens the app by tapping the push notification, this method may be called again, so you can update the UI, and display relevant information.

In the first case, WenderCast will create the news item and open up directly to the news section. Add the following code to the end of **`application(_:didFinishLaunchingWithOptions:)`**, before the return statement:

```
// Check if launched from notification
// 1
if let notification = launchOptions?[.remoteNotification] as? [String: AnyObject] {
  // 2
  let aps = notification["aps"] as! [String: AnyObject]
  _ = NewsItem.makeNewsItem(aps)
  // 3
  (window?.rootViewController as? UITabBarController)?.selectedIndex = 1
}
```

This code does three things:

1. It checks whether the value for **`UIApplicationLaunchOptionsKey.remoteNotification`** exists in **`launchOptions`**. If it does, this will be the push notification payload you sent.
2. If it exists, you grab the **`aps`** dictionary and pass it to **`createNewNewsItem(_:)`**, which is a helper method provided to create a **`NewsItem`** from the dictionary and refresh the news table.
3. Lastly, it changes the selected tab of the tab controller to **`1`**, the news section.

To test this, you need to edit the scheme of WenderCast:

Under **Run -> Info**, select **Wait for executable to be launched**:

This option will make the debugger wait for the app to be launched for the first time after installing to attach to it.

Build and run. Once it's done installing, send out some breaking news again. Tap on the notification, and the app should open up to some news:

> **Note:** If you stop receiving push notifications, it is likely that your device token has changed. This can happen if you uninstall and reinstall the app. Double check the device token to make sure.

To handle the other case, add the following method to **`AppDelegate`**:

```
func application(
  _ application: UIApplication,
  didReceiveRemoteNotification userInfo: [AnyHashable : Any],
  fetchCompletionHandler completionHandler: @escaping (UIBackgroundFetchResult) -> Void) {
```

```
  let aps = userInfo["aps"] as! [String: AnyObject]
  _ = NewsItem.makeNewsItem(aps)
}
```

This method directly uses the helper function to create a new **`NewsItem`**. You can now change the scheme back to launching the app automatically if you like.

Build and run. Keep the app running in the foreground and on the News section. Send another news push notification and watch as it magically appears in the feed:

That's it! Your app can now handle breaking news in this basic way.

Something important consider: many times, push notifications may be missed. This is okay for WenderCast, since having the full list of news isn't too important for this app, but in general you should not use push notifications as the only way of delivering content.

Instead, push notifications should signal that there is new content *available* and let the app download the content from the source (e.g. from a REST API). WenderCast is a bit limited in this sense, as it doesn't have a true server-side component.

Actionable Notifications

Actionable notifications let you add custom buttons to the notification itself. You may have noticed this on email notifications or Tweets that let you "reply" or "favorite" on the spot.

Actionable notifications are defined by your app when you register for notifications by using **categories**. Each category of notification can have a few preset custom actions.

Once registered, your server can set the category of a push notification; the corresponding actions will be available to the user when received.

For WenderCast, you will define a "News" category with a custom action named "View" which allows users to directly view the news article in the app if they choose to.

Replace **`registerForPushNotifications()`** in the **`AppDelegate`** with the following:

```
func registerForPushNotifications() {
  UNUserNotificationCenter.current().requestAuthorization(options: [.alert, .sound,
.badge]) {
    (granted, error) in
    print("Permission granted: \(granted)")

    guard granted else { return }

    // 1
    let viewAction = UNNotificationAction(identifier: viewActionIdentifier,
                                          title: "View",
                                          options: [.foreground])

    // 2
    let newsCategory = UNNotificationCategory(identifier: newsCategoryIdentifier,
                                              actions: [viewAction],
                                              intentIdentifiers: [],
                                              options: [])
    // 3
    UNUserNotificationCenter.current().setNotificationCategories([newsCategory])

    self.getNotificationSettings()
  }
}
```

Here's what the new code does:

1. Here you create a new notification action, with the title **View** on the button, that opens the app in the foreground when triggered. The action has a distinct identifier, which is used to differentiate between other actions on the same notification.
2. Next, you define the news category, which will contain the view action. It also has a distinct identifier, which your payload will need to contain to specify, that the push notification belongs to this category.
3. Finally, by invoking **`setNotificationCategories(_:)`**, you register the new actionable notification.

That's it! Build and run the app to register the new notification settings.

Background the app and then send the following payload via **Pusher**:

```
{
  "aps": {
    "alert": "Breaking News!",
    "sound": "default",
    "link_url": "https://raywenderlich.com",
    "category": "NEWS_CATEGORY"
  }
}
```

If all goes well, you should be able to pull down on the notification to reveal the View action:

Nice! Tapping on it will launch WenderCast, but it won't do anything. To get it to display the news item, you need to do some more event handling in the delegate.

Handling Notification Actions

Whenever a notification action is triggered, **`UNUserNotificationCenter`** informs its delegate. Back in **AppDelegate.swift**, add the following class extension to the bottom of the file:

```
extension AppDelegate: UNUserNotificationCenterDelegate {

  func userNotificationCenter(_ center: UNUserNotificationCenter,
                              didReceive response: UNNotificationResponse,
                              withCompletionHandler completionHandler: @escaping () ->
Void) {
    // 1
    let userInfo = response.notification.request.content.userInfo
    let aps = userInfo["aps"] as! [String: AnyObject]

    // 2
    if let newsItem = NewsItem.makeNewsItem(aps) {
      (window?.rootViewController as? UITabBarController)?.selectedIndex = 1

      // 3
      if response.actionIdentifier == viewActionIdentifier,
        let url = URL(string: newsItem.link) {
        let safari = SFSafariViewController(url: url)
        window?.rootViewController?.present(safari, animated: true, completion: nil)
      }
    }

    // 4
    completionHandler()
  }
}
```

This is the callback you get when the app is opened by a custom action. It might look like there's a lot going on, but

there's really not much new here:

1. Get the **`aps`** dictionary.
2. Create the **`NewsItem`** from the dictionary and navigate to the News section.
3. Check the action identifier, which is passed in as **`identifier`**. If it is the "View" action and the link is a valid URL, it displays the link in a **`SFSafariViewController`**.
4. Call the completion handler that is passed to you by the system after handling the action.

There is one last bit: you have to set the delegate on **`UNUserNotificationCenter`**. Add this line to the top of **`application(_:didFinishLaunchingWithOptions:)`**:

```
UNUserNotificationCenter.current().delegate = self
```

Build and run. Close the app again, then send another news notification with the following payload:

```
{
  "aps": {
    "alert": "New Posts!",
    "sound": "default",
    "link_url": "https://raywenderlich.com",
    "category": "NEWS_CATEGORY"
  }
}
```

Tap on the action, and you should see WenderCast present a Safari View Controller right after it launches:

Congratulations, you've just implemented an actionable notification! Send a few more and try opening the notification in different ways to see how it behaves.

Silent Push Notifications

Silent push notifications can wake your app up silently to perform some tasks in the background. WenderCast can use this feature to quietly refresh the podcast list.

As you can imagine, with a proper server-component this can be very efficient. Your app won't need to poll for data constantly — you can send it a silent push notification whenever new data is available.

To get started, go to **App Settings -> Capabilites** and turn on **Background Modes** for WenderCast. Check the last option, **Remote Notifications**:

Now your app will wake up in the background when it receives one of these push notifications.

Inside **`AppDelegate`**, replace **`application(_:didReceiveRemoteNotification:)`** with this more powerful version:

```
func application(
  _ application: UIApplication,
  didReceiveRemoteNotification userInfo: [AnyHashable : Any],
  fetchCompletionHandler completionHandler: @escaping (UIBackgroundFetchResult) -> Void) {

  let aps = userInfo["aps"] as! [String: AnyObject]

  // 1
  if aps["content-available"] as? Int == 1 {
    let podcastStore = PodcastStore.sharedStore
    // Refresh Podcast
    // 2
    podcastStore.refreshItems { didLoadNewItems in
```

```
      // 3
      completionHandler(didLoadNewItems ? .newData : .noData)
    }
  } else  {
    // News
    // 4
    _ = NewsItem.makeNewsItem(aps)
    completionHandler(.newData)
  }
}
```

Let's go over the code:

1. Check to see if **`content-available`** is set to 1, to see whether or not it is a silent notification.
2. Refresh the podcast list, which is a network call and therefore asynchronous.
3. When the list is refreshed, call the completion handler and lets the system know whether any new data was loaded.
4. If it isn't a silent notification, assume it is news again and create a news item.

Be sure to call the completion handler with the honest result. The system measures the battery consumption and time that your app uses in the background, and may throttle your app if needed.

That's all there is to it; to test it, push the following payload via **Pusher**:

```
{
  "aps": {
    "content-available": 1
  }
}
```

If all goes well, nothing should happen! To see the code being run, change the scheme to "Wait for executable to be launched" again and set a breakpoint within **`application(_:didReceiveRemoteNotification:fetchCompletionHandler:)`** to make sure it runs.

Where To Go From Here?

Congratulations! You've completed this push notifications tutorial and made WenderCast a fully-featured app with push notifications!

You can download the completed project here. Remember that you will still need to change the bundle ID and create certificates to make it work.

Even though push notifications are an important part of apps nowadays, it's also very common for users to decline permissions to your app if notifications are sent too often. But with thoughtful design, push notifications can keep your users coming back to your app again and again!

This cat received a push notification that his dinner was ready

---

I hope you've enjoyed this push notifications tutorial; if you have any questions feel free to leave them in the discussion below.

Team

Each tutorial at www.raywenderlich.com is created by a team of dedicated developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Author
József Vesza

Tech Editor
Joshua Greene

Final Pass Editor
James Frost

Team Lead
Andy Obusek

József Vesza

Software developer on multiple platforms, mainly mobile.
Interface Builder and Auto Layout advocate. I have been working with iOS since 2012.
Get in touch via Twitter or LinkedIn.
Browse my code on GitHub and Stack Overflow.