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 `registerForRemoteNotificationTypes:` 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 `NSUserNotification` object (which includes a delivery time) and the `NSUserNotificationCenter` is responsible for delivering it appropriately. (An OS X app can also adopt the `NSUserNotificationCenterDelegate` protocol to customize the behavior of the default `NSUserNotificationCenter` object.)

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

1. In iOS 8 and later, register for notification types, as described in Registering 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 `applicationDidFinishLaunching:` method in which the delegate can obtain the remote-notification payload. If the app icon is tapped on a device running iOS, the app calls the same method, but furnishes no information about the notification.

The notification is delivered when the app is running in the foreground. The app calls the `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 `application:didReceiveRemoteNotification:`.

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

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

The delegate for an iOS app in Listing 2-11 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 `NSApplicationLaunchUserNotificationKey` 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.