Extracted from:

Developing for Apple Watch

Your App on Their Wrists

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Developing for Apple Watch

Your App on Their Wrists

Jeff Kelley

edited by Rebecca Gulick

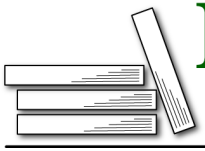
Developing for Apple Watch

Your App on Their Wrists

Jeff Kelley

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Rebecca Gulick (editor)
Candace Cunningham (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-068-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—July 2015

Responding to User Actions in Push Notifications

Given that the ideal watch app allows your users to unlock their iPhones *less*, one of the best things you can offer them is actionable push notifications. Sometimes all you need from the user is a single tap of a button. Consider a social networking app where users can send invitations to events. If your app sends a push notification saying that your user has received an invitation, you could offer Accept and Decline buttons right in the notification. Then, without needing to open your app at all, the user has taken care of the task right then and there.

Registering Categories

The key to actionable notifications is the concept of *categories*. You'll give your notifications categories—arbitrary strings you create—to differentiate them. It's up to you to make as many or as few categories as you need, but each notification for a given category will have the same options. When your app starts, you'll register these categories with the system. Once they're registered, any push notifications that come in with a category set will automatically show the relevant options. Registering the category is a laborious process. First, you'll create the actions:

ActionableNotifications/ActionableNotifications/AppDelegate.swift

```
let acceptAction = UIMutableUserNotificationAction()

acceptAction.activationMode = .Background
acceptAction.authenticationRequired = true
acceptAction.identifier = "Accept"
acceptAction.title = "Accept"

let declineAction = UIMutableUserNotificationAction()

declineAction.activationMode = .Background
declineAction.authenticationRequired = true
declineAction.identifier = "Decline"
declineAction.title = "Decline"

let maybeAction = UIMutableUserNotificationAction()

maybeAction.activationMode = .Background
maybeAction.authenticationRequired = true
maybeAction.identifier = "Maybe"
maybeAction.title = "Maybe"

let deleteAction = UIMutableUserNotificationAction()

deleteAction.activationMode = .Background
```

```
deleteAction.authenticationRequired = true
deleteAction.destructive = true
deleteAction.identifier = "Delete"
deleteAction.title = "Delete"
```

For every action you want to offer, you'll create a UIMutableUserNotificationAction to represent it, giving it an identifier to identify it later, a title to show the user, and some other options, such as whether authentication is required. You'll use that for sensitive actions, such as replying to this calendar invitation. You can also set destructive, which will color the option red to inform users that their action will have some destructive consequences: deleting a message, blocking a user, etc. Once you've created your actions, you can create a category to contain them:

ActionableNotifications/ActionableNotifications/AppDelegate.swift

```
Line 1 let category = UIMutableUserNotificationCategory()
  2
  3 category.identifier = "Invitation"
  4
  5 category.setActions([acceptAction, declineAction], forContext: .Minimal)
  6
  7 category.setActions([acceptAction, maybeAction, declineAction, deleteAction],
  8     forContext: .Default)
```

You'll notice that you need to define the list of actions *twice*: first on line 5 and again on line 7. That's because these notifications can appear in multiple locations. When the notification is a banner on the user's iPhone and he drags it down to interact with it, only two buttons appear. If you'd only set up the default context by passing UIUserNotificationActionContextDefault for the forContext: parameter, the first two would be used: Accept and Maybe. Since we want Accept and Decline to appear if there are only two choices, we pass in those for the UIUserNotificationActionContextMinimal context.

Once you've created your category, you need to tell iOS that it exists. This is just a few lines of code:

ActionableNotifications/ActionableNotifications/AppDelegate.swift

```
let categories: Set<NSObject> = [category]

let settings = UIUserNotificationSettings(forTypes: .Alert,
    categories: categories)

UIApplication.sharedApplication().registerUserNotificationSettings(settings)
```

First, you create a set containing all of your app's categories. Then you register a UIUserNotificationSettings object, which you create for an alert type. In this case we'll use .Alert since badges and sounds don't allow us to present buttons.

Once that's complete, we're done! Future push notifications or local notifications using that category will present these options.

Including Category Information in Notifications

For push notifications, setting the category of an alert is as easy as adding a category key to the JSON payload your notification server sends to Apple. A sample alert JSON payload would be as follows:

```
{
  "aps": {
    "alert": "Jane invited you to Launch Party!",
    "category": "Invitation"
  }
}
```

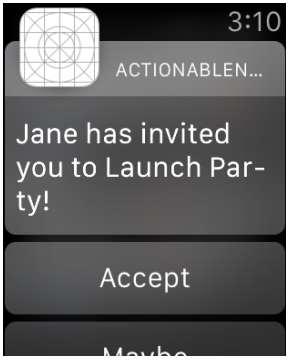
Note that the value we gave for the category key is the same as the identifier we used when creating the category in code to register it. For a local notification, simply set the category property:

ActionableNotifications/ActionableNotifications/ViewController.swift

```
let notification = UILocalNotification()

notification.alertBody = "Jane has invited you to Launch Party!"
notification.category = "Invitation"

UIApplication.sharedApplication().scheduleLocalNotification(notification)
```



Once you have this all set up, notifications like this will appear on the watch.

When your user taps one of these actions, whether it's on Apple Watch or iPhone, you'll need to write some code to handle the response.

Responding to Actions

To respond to these actions inside of notifications, the UIApplicationDelegate has two slightly different methods for your app delegate: application(_:handleActionWithIdentifier:forRemoteNotification:completionHandler:) and application(_:handleActionWithIdentifier:forLocalNotification:completionHandler:). The former handles push notifications and the latter handles local notifications. The final completionHandler argument is a closure for you to call when your work is finished; it's how the system knows you've handled the action. In our invitation example, responding is easy:

ActionableNotifications/ActionableNotifications/AppDelegate.swift

```
func application(application: UIApplication,
```

```
    handleActionWithIdentifier identifier: String?,
    forLocalNotification notification: UILocalNotification,
    completionHandler: () -> Void) {
    if notification.category == "Invitation" {
        if identifier == "Accept" {
            // Handle accepted invitation
        }
        else if identifier == "Maybe" {
            // Handle tentative response
        }
        else if identifier == "Decline" {
            // Handle declined invitation
        }
        else if identifier == "Delete" {
            // Delete invitation
        }
    }

    completionHandler()
}
```

First, examine the notification itself to see its category. In an app where you have multiple categories, it's important that the right code is running. Next, use the identifier argument to see which action the user selected. Finally, after doing whatever you need to do to handle the action, call the completion handler to tell iOS you're done. That's it! Just like that, you've run some code in response to a user selecting a notification action on the watch.

By using actionable notifications, you've future-proofed your app for Apple Watch and any other connected device. Instead of writing code for a specific screen or device, you've written code that tells iOS what the valid options are, allowing the iPhone or Apple Watch device to determine the best way to get your user's feedback.