

Up to date for iOS 12,
Xcode 10 & Swift 4.2



Push Notifications

by Tutorials

FIRST EDITION

Mastering push notifications on iOS

By Scott Grosch

Push Notifications by Tutorials

By Scott Grosch

Copyright ©2018 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This book and all corresponding materials (such as source code) are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

About the Authors



Scott Grosch is the author of this book. He has been involved with iOS app development since the first release of the public SDK from Apple. He mostly works with a small set of clients on a couple large apps. During the day, Scott is a Solutions Architect at a Fortune 500 company in the Pacific Northwest. At night, he's still working on figuring out how to be a good parent to a toddler with his wife.

About the Editors



Marin Bencevic is the tech editor of this book. He is a Swift and Unity developer who likes to work on cool iOS apps and games, nerd out about programming, learn new things and then blog about it. Mostly, though, he just causes SourceKit crashes. He also has a chubby cat.



Shai Mishali is the Final Pass Editor of this book. He's the iOS Tech Lead for Gett, the global on-demand mobility company; as well as an international speaker, and a highly active open-source contributor and maintainer on several high-profile projects - namely, the RxSwift Community and RxSwift projects. As an avid enthusiast of hackathons, Shai took 1st place at BattleHack Tel-Aviv 2014, BattleHack World Finals San Jose 2014, and Ford's Developer Challenge Tel-Aviv 2015. You can find him on GitHub and Twitter @freak4pc.



Manda Frederick is the editor of this book. She has been involved in publishing for over 10 years through various creative, educational, medical and technical print and digital publications, and is thrilled to bring her experience to the raywenderlich.com family as Managing Editor. In her free time, you can find her at the climbing gym, backpacking in the backcountry, hanging with her dog, working on poems, playing guitar and exploring breweries.

About the Artist



Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Dedication

"This book is dedicated to my wife and daughter, both of whom gave up many a night so that I could work on it, as well as to my parents who always made sure a good education was a priority."

— Scott Grosch

Table of Contents: Overview

Introduction	12
Book License	13
Book Source Code & Forums.....	14
What You Need	15
Book Updates	16
About the Cover	17
Chapter 1: Introduction.....	18
Chapter 2: Push Notifications.....	20
Chapter 3: Remote Notification Payload.....	25
Chapter 4: Xcode Project Setup.....	34
Chapter 5: Apple Push Notification Servers	40
Chapter 6: Server Side Pushes.....	45
Chapter 7: Expanding the Application.....	61
Chapter 8: Handling Common Scenarios	68
Chapter 9: Custom Actions	79
Chapter 10: Modifying the Payload.....	86
Chapter 11: Custom Interfaces	102
Chapter 12: Putting It All Together.....	124
Chapter 13: Local Notifications.....	144

Conclusion.....	163
More Books You Might Enjoy	164

Table of Contents: Extended

Introduction	12
Book License	13
Book Source Code & Forums.....	14
What You Need	15
Book Updates	16
About the Cover	17
Chapter 1: Introduction.....	18
Getting started	19
Chapter 2: Push Notifications.....	20
What are they good for?.....	20
Remote notifications	21
Local notifications.....	23
Location aware.....	23
Key points	23
Where to go from here?	24
Chapter 3: Remote Notification Payload.....	25
The aps dictionary key	26
Your custom data	31
Collapsing notifications.....	31
Key points	32
Where to go from here?	33
Chapter 4: Xcode Project Setup.....	34
Adding capabilities	34
Registering for notifications.....	35
Getting the device token.....	37

Key points	39
Where to go from here?	39
Chapter 5: Apple Push Notification Servers	40
Token types	40
Getting your Authentication Token.....	41
Sending a push	42
Key points	44
Where to go from here?	44
Chapter 6: Server Side Pushes.....	45
Using third-party services.....	45
Setting up Docker	46
Setting up a SQL server	46
Setting up Vapor.....	47
But they disabled push!	59
Key points	60
Where to go from here?	60
Chapter 7: Expanding the Application.....	61
Setting the team and bundle identifier.....	61
Updating the server	62
Extending AppDelegate	65
Key points	66
Where to go from here?	67
Chapter 8: Handling Common Scenarios	68
Tapping the notification.....	71
Silent notifications.....	74
Method routing	78
Key points	78
Where to go from here?	78
Chapter 9: Custom Actions	79
Categories	79
Extending Foundation's notification	82

Responding to the action.....	84
Key points	85
Chapter 10: Modifying the Payload.....	86
Configuring Xcode for a service extension.....	86
Decrypting the payload	88
Downloading a video	91
Service extension payloads.....	93
Sharing data with your main target	94
Badging the app icon	95
Accessing Core Data	97
Localization	99
Debugging.....	99
Key points.....	100
Where to go from here?.....	101
Chapter 11: Custom Interfaces	102
Configuring Xcode for custom UI	102
Designing the interface	104
Accepting text input.....	107
Changing actions.....	110
Attachments.....	113
Custom user input	116
Hiding default content	119
Interactive UI	120
Debugging.....	121
Key points.....	122
Where to go from here?.....	123
Chapter 12: Putting It All Together.....	124
Setting up the Xcode project	124
AppDelegate code	125
Requesting calendar permissions.....	127
The payload	129
Notification Service Extension	130

Push Notifications

Content Service Extension.....	135
Final cleanups	141
Where to go from here?.....	143
Chapter 13: Local Notifications.....	144
You still need permission!.....	144
Objects versus payloads.....	145
Foreground notifications	149
The sample platter	150
Key points.....	162
Where to go from here?.....	162
Conclusion.....	163
More Books You Might Enjoy	164
New to iOS or Swift?.....	164
Experienced iOS developer?.....	166
Want to make games?.....	179
Want to learn Android or Kotlin?.....	183



Introduction

Push Notifications by Tutorials provides a beginner-to-master path for developers who wish to learn everything there is about push notifications. This book teaches the basic building blocks of delivering push notifications, as well as how these notifications are constructed and delivered to your end user.

As you work through the book, you'll deliver basic push notifications and expand your knowledge from chapter to chapter — super-charging your notifications with additional abilities as you progress through the book: rich custom UI for notifications, custom actions, notifications with special triggers such as time or location and much more. You'll also learn how to build a Vapor-based web service to deliver your very own push notifications, without the need for a third-party provider.

This book uses the Swift language. If you want to brush up on your Swift knowledge before diving in, be sure to check out our classic Swift beginner books — *Swift Apprentice* <https://bit.ly/2ue5EH3> and *iOS Apprentice* <https://bit.ly/2JdWTjD>.

As always, we appreciate that the By Tutorials team is your resource for beginner and advanced development skills!

— The *Push Notifications by Tutorials* team

Book License

By purchasing *Push Notifications by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *Push Notifications by Tutorials* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Push Notifications by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: “Artwork/images/designs: from *Push Notifications by Tutorials*, available at www.raywenderlich.com.”
- The source code included in *Push Notifications by Tutorials* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Push Notifications by Tutorials* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

Book Source Code & Forums

This book comes with the source code for the starter and completed projects for each chapter. These resources are shipped with the digital edition you downloaded from store.raywenderlich.com.

We've also set up an official forum for the book at forums.raywenderlich.com. This is a great place to ask questions about the book or to submit any errors you may find.



What You Need

To follow along with this book, you'll need the following:

- **Xcode 10 and Swift 4.2:** Xcode is the main development tool for writing code in Swift. You'll need Xcode 10 at a minimum, since that version includes Swift 4.2. You can download the latest version of Xcode for free from the Mac App Store.
- **Apple Developer Program membership:** Sending push notifications requires having the ability to build, sign and run your app on a **physical** device with a Push Notification certificate — a capability reserved for paid members of the Apple Developer Program. The Apple Developer Program annual fee is \$99 USD, but the exact amount might change based on local currency. More information on these memberships can be found at <https://developer.apple.com/programs/>. The information in this book will still prove an invaluable reference and resource to a developer without a paid membership, though.

If you haven't installed the latest version of Xcode, be sure to do that before continuing with the book. The code covered in this book requires Swift 4.2 and Xcode 10 — you may get lost if you try to work with an older version.

The only two prerequisites for this book are an intermediate understanding of Swift and iOS development, along with a paid Apple Developer Program membership.

Book Updates

Since you've purchased the digital edition version of this book, you get free access to any updates we may make to the book!

The best way to get update notifications is to sign up for our monthly newsletter. This includes a list of the tutorials that came out on raywenderlich.com that month, any important news like book updates or new books, and a list of our favorite iOS development links for that month. You can sign up, here: www.raywenderlich.com/newsletter.





About the Cover

Whether you call a group of these animals a “family,” a “bevy,” a “lodge,” a “romp” or a “raft,” there’s a lot to love about otters!

Despite their propensity for living near water and eating a diet made up mostly of fish and shellfish, most species of otter prefer to spend most of their time on land; otherwise, their dense fur would become waterlogged. Only sea otters spend the majority of their lives in the ocean.

Otters have a fascinating vocabulary; they use all kinds of vocalizations to communicate with their group or to draw notice to themselves. From coos, to hums, to growls, screams and high-intensity snorting to a “ha!” sound, otters really know how to get their point across.

In fact, you can take the communication stylings of otters as inspiration during your work with push notifications; you need to figure out how to get just the right information, to just the right people, at just the right time. Otherwise, your users will end up interpreting your app’s notifications as *otter* nonsense!

Chapter 1: Introduction

Push notifications are one of the most important interaction points of your app with your users. Simply put, a push notification is a way to send any type of data to your user's app, even if the user isn't actively using it. The user will normally see the push notification appear as a banner alert on the device, a badge on the app icon and/or a sound. Push notifications are a direct line of communication to your user. You can alert the user of new content, new messages from friends or any other interesting piece of information. Notifications also provide users with a quick way to interact with your app and allow for faster interaction via background data downloads.

Conversely, notifications can be a bane to app retention — meaning how likely a user is to continue using your app — if you send them too frequently or use them in a way that's not useful to your customers. For example, if you send notifications about app version upgrades or messages just telling them new content is available, that will lead to bad user experiences.

Push notifications may seem simple and straightforward at first because they aren't hard to use and almost everyone is familiar with them; however, knowing how and when to use them may prove challenging. With advancements in the latest iOS releases bringing some exciting advanced features — such as Rich Media Notifications, Notification Actions, Grouped Notifications and more — you may quickly realize that you need a book to help you out. Well, here's that book!

In this book, you'll learn everything you need in order to create, send and receive push notifications, meaning notifications that come from an external service as opposed to locally from the device. You'll also cover how to handle local notifications because, sometimes, you don't need all the overhead of a remote notification; rather, it's enough to simply schedule a notification to appear at a specific point in the future or when you enter a specific location.

Once you've worked your way through this book, you'll be a master of push notifications and well on your way to implementing them inside your own apps!

However, as helpful as this book may be and as great as push notifications can be, it's critical that you always keep in mind that the user may never receive your notification. Not only can your users opt-out of them at any point in time, *there is no guarantee your push notifications will be delivered*. What this means, as a developer, is that you can't depend on push notifications for your app to function properly — but this doesn't mean your push notifications shouldn't be well-made and used responsibly, which is what will be covered in this book.

Getting started

To follow along with the tutorials in this book, you'll need a Mac computer capable of running Xcode. You can get the latest version of Xcode for free from the Mac App Store (<https://apple.co/1f2E3nY>). While there are other platforms for developing iOS apps, none are officially supported by Apple and will not be covered in this book.

For the purposes of following these tutorials, please note that Apple's iOS Simulator is **not** capable of receiving push notifications, meaning that you'll need a physical iOS device — such as an iPhone or iPad — along with a paid Apple Developer account, in order to create a Push Notifications certificate and run any of the apps included in this book.

You will also need an intermediate level of knowledge of Swift and iOS development. This book makes the assumption that you are already an experienced iOS developer and are simply looking for details on implementing push notifications in your apps, or looking for a great reference when working on your app's notifications.

If you need to brush up on your Swift or iOS skills, you may be interested in the following resources:

- *Swift Apprentice* (<https://bit.ly/2ue5EH3>)
- *iOS Apprentice* (<https://bit.ly/2JdWTjD>)
- “Programming in Swift” video tutorials (<https://bit.ly/2Ja34VP>)
- “Your First Swift 4 & iOS 11 App” video tutorials (<https://bit.ly/2fjVcDH>)

Chapter 2: Push Notifications

Push notifications are a useful feature that allow you to interact with your users outside of the normal flow of your app. A notification can be scheduled locally based on conditions such as time or location, or scheduled from a remote service and “pushed” to your device. Regardless of whether you are utilizing a local or remote notification, the general process for handling one is the same:

- Ask your user for permission to receive notifications.
- Optionally make changes to the message before display.
- Optionally add custom buttons for the user to interact with.
- Optionally configure a custom user interface to display the notification.
- Optionally take action based on what the user did with the notification.

What are they good for?

You’d be hard pressed in this day and age to *not* have seen a push notification at some point. They are capable of many actions:

- Displaying a message.
- Playing a sound.
- Updating the badge icon on your app.
- Showing an image or playing a movie.
- Giving the user a way to pick from a few options.

- *Anything* that a `UIViewController` can implement.

While you can technically show any type of user interface as long as it fits within the bounds of a notification window, that doesn't mean you *should* do so. Always keep user experience in the forefront of your mind when designing a notification. Will your users want to see it, hear it or interact with it?

Remote notifications

By far, the most common type of notification used is a **remote notification**, in which a Cloud service, usually a web server, is utilized to determine that a notification should be built and sent to the device.

A remote notification can be a great fit for multiplayer games that are turn-based. Once an opponent has made his or her move, the user is sent a notification stating that it's now their turn. If the user has any type of data feed, such as a news app, then a silent remote notification can be used to proactively send data to the user's device so that the content is already there when they run the app the next time, versus having to wait for a network download.

Clearly, you don't want just anyone to be able to send messages to your users! Apple has built its **Apple Push Notification service** (APNs) using **Transport Layer Security** (TLS). TLS provides privacy and data integrity, which ensures that you, and only you, control your app's notifications.

Security

APNs utilizes cryptographic validation and authentication to ensure security of your messages.

Your server, called a **provider**, utilizes TLS to send notification requests to Apple, and the device uses an opaque `Data` instance — referred to as a **device token** — which contains a unique identifier that the APNs is able to decode. The iOS device receives a (possibly new) token when it authenticates with the APNs; the token is then sent to your provider so that a notification can be received in the future. You should never cache a device token on the user's device as there are multiple instances in which APNs will need to generate a new token, such as installing the app on a new device or performing a restore from a backup.

The device token is now the **address** that a provider uses to reference a user's specific device. When the provider service wishes to send a notification, it will tell APNs the

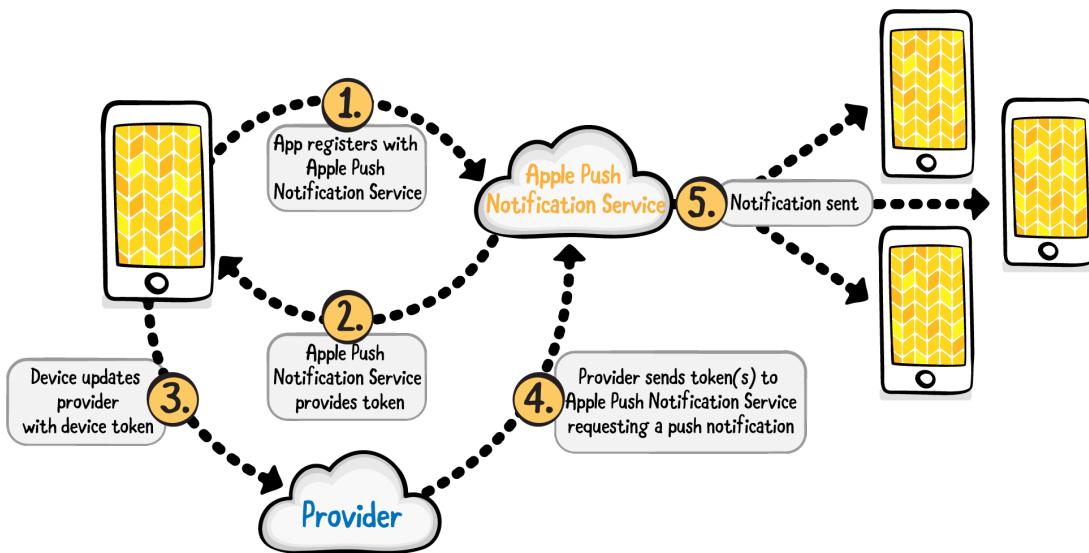
specific device token(s) that need to be sent a message. The device then receives the message and can take appropriate action based on the content of the notification. You can either build your own provider service, as discussed in Chapter 6, “Server Side Pushes,” or you can use one of the many third-party providers that already exist.

Notification message flow

It is important to understand the steps between registering your app with the Apple Push Notification Service and the user actually receiving a notification.

1. During `application(_:didFinishLaunchingWithOptions:)`, a request is sent to APNs for a device token via `registerForRemoteNotifications`.
2. APNs will return a device token to your app and call `application(_:didRegisterForRemoteNotificationsWithDeviceToken:)` or emit an error message to `application(_:didFailToRegisterForRemoteNotificationsWithError:)`.
3. The device sends the token to a provider in either binary or hexadecimal format. The provider will keep track of the token.
4. The provider sends a notification request, including one or more tokens, to APNs.
5. APNs sends a notification to each device for which a valid token was provided.

You can see these steps reflected in the image below:



There are multiple ways a notification can materialize on a device once the notification has actually been pushed, depending on the state of the app and what features have been configured. Those will be discussed in greater detail in the chapters of this book.

Local notifications

A **local notification** is created and scheduled on the device, as opposed to being sent to the device from a remote provider. A local notification allows all the same features as a remote notification. The only difference is that a local notification is triggered based on a set amount of time passing, or entering/exiting a geographical area, as opposed to being pushed to the device.

You might want to use a local notification similar to a timer. If your app teaches people how to cook in a step-by-step process, you may have a notification appear when the food has been marinating long enough and is now ready to go into the oven, with a new notification when it's time to take the food out of the oven.

Location aware

While it's easy to think of notifications as being in a type of sandbox of their own, there's no reason to exclude other iOS-provided features to enhance your app, such as location services. You can employ location services by tying a remote notification to a user's location. You may decide to send coupons to your customers, but only in a specific geographical area. Perhaps a guest author is reading at the local bookstore and you want to let your app's users know about it, but only if they live close enough to make it worthwhile.

Key points

- Push notifications allow you to interact with your users outside of the normal flow of your app.
- A notification can be scheduled locally based on conditions or from a remote service and “pushed” to your device.
- **Remote notifications** are the most common type, and they use a Cloud service to determine that a notification should be built and sent to the device.
- Notification messages remain secure because APNs utilizes cryptographic validation

and authentication.

- A **local notification** is created and scheduled on the device, as opposed to being sent to the device from a remote provider.

Where to go from here?

This chapter has been the first step in your journey to understanding the many facets, opportunities and challenges of leveraging push notifications.

Now that you know the basic terminology, it's time for you to actually learn how a notification request, known as a **payload**, is constructed.

Chapter 3: Remote Notification Payload

As you learned in Chapter 2, “Push Notifications,” a **push** happens by sending data across the internet. That data is referred to as a **payload**, and it contains all of the information necessary to tell your app what to do when the push notification arrives. The Cloud service is responsible for constructing that payload and sending it, along with one or more unique device tokens, to APNs.

Originally, notifications used a packed-binary interface, where each bit of the packet had a specific meaning. Using bitfields is much harder to handle and was confusing for many developers. Apple changed the payload structure and decided to use a single, simple, JSON structure. By using JSON, Apple ensured that the payload is simple to parse and construct in any language, while also providing the flexibility needed for the future, as new capabilities are added to push notifications.

There are a few keys specifically defined by Apple, some of which are mandatory, but the rest of the keys and values are up to you to define as needed. This chapter will cover those predefined keys.

For regular remote notifications, the maximum payload size is currently 4KB (4,096 bytes). If your notification is too large, Apple will simply refuse it and you’ll get an error from APNs. If you’re suddenly worried that your push notifications won’t work because you’ve got a sizable image to send, for example, don’t worry! You’ll handle how to download attachments in Chapter 10, “Modifying the Payload.”

In this chapter, you’ll learn how to construct the payload, what the various payload keys mean, how to supply custom data and how to handle collapsed and grouped notifications.

The `aps` dictionary key

The aforementioned JSON object is well-structured to hold all of the key pieces of data necessary to describe a push notification. The rest of this chapter will describe each key in detail.

The `aps` dictionary key is the main hub of the notification payload wherein everything defined and owned by Apple lives. Within the object at this key, you'll configure such items as:

- The message to be displayed to the end user.
- What the app badge number should be set to.
- What sound, if any, should be played when the notification arrives.
- Whether the notification happens without user interaction.
- Whether the notification triggers custom actions or user interfaces.

There are several keys you can use, each with their own considerations.

Alert

The key you'll use most often is the `alert` key. This key allows you to specify the message that will be displayed to your user. When push notifications were first released, the `alert` key simply took a string with the message. While you can, for legacy reasons, continue to set the value to a string, it's preferred that you instead use a dictionary. The most common payload for a message would include a simple `title` and `body`:

```
{  
  "aps": {  
    "alert": {  
      "title": "Your food is done.",  
      "body": "Be careful, it's really hot!"  
    }  
  }  
}
```

Using a dictionary, instead of the legacy string, enables you to utilize both the `title` and `body` data points for your message. If you don't want a `title`, for example, simply leave that key/value pair out.

You may, however, run into some issues with this because of **localization**.

Localization

Yes, the localization monster rears its ugly head, again! If the whole world could just agree on a single language, life would be so much simpler for us developers. You can quickly tell how using a dictionary isn't going to work for your non-English speaking users. There are two options to work around this issue:

1. Call `Locale.preferredLanguages` at registration and send the list of languages your user speaks to your server.
2. Store localized versions of all your notifications in your app bundle.

There are pros and cons to each approach, and it really depends on the quantity and type of notifications you'll be sending. If you keep everything on the server, and send each person the proper translation, you'll never have to push a new version of your app when you add new notification messages.

Conversely, that means more work on the server side and more customized push notification code versus just letting iOS handle the translations for you.

If you decide to handle localization on the app side, instead of passing along `title` and `body` keys, you can use `title-loc-key` and `title-loc-args` for the title, and `loc-key` and `loc-args` for the body.

For example, your payload might end up looking like this:

```
{  
  "aps": {  
    "alert": {  
      "title-loc-key": "FOOD_ORDERED",  
      "loc-key": "FOOD_PICKUP_TIME",  
      "loc-args": ["2018-05-02T19:32:41Z"]  
    }  
  }  
}
```

When iOS gets the notification, it'll look in the proper **Localizable.strings** file inside your app to automatically get the correct translation, and then substitute the date and time into the proper location. This might result in an English language speaker seeing:

You can pick up your order at 5:32 p.m.

Whereas a person reading Mandarin would see this instead:

你可以五點半領取

To keep the rest of the examples in this chapter simple, only the `title` and `body` keys will be used.

Grouping notifications

Starting with iOS 12, adding the `thread_identifier` key to the `alert` dictionary will let iOS combine all notifications with the same identifier value into a single group in the notification center. Try to use a guaranteed unique value representing some thread of messages, such as the primary key from a database or a UUID.

In a game app, you might use this feature so that all notifications related to a specific game session are grouped together and not merged in with all other game sessions. If you leave this key out, iOS will default to grouping everything from your app together into one group. Keep in mind “grouping” notifications is different from “collapsing” notifications.

Be aware that your users are able to turn off notification grouping in the iOS Settings app, if they so desire!



Badge

Since your users might not have had their phones handy when the alert message came through, you can also badge the app icon. If you'd like your app icon to display the numerical badge number, simply specify it using the `badge` key. To clear the badge and remove it, set the value to `0`.

Note: The badge number is not a mathematical addition or subtraction. It's an absolute value that will be set on your app icon.

What this means is that your server is going to have to know what number to display to the end user, which sometimes makes this key more trouble than it's worth. In Chapter 10, “Modifying the Payload,” when we discuss service extensions, you’ll learn a trick to work around this issue.

```
{  
  "aps": {  
    "alert": {  
      "title": "Your food is done.",  
      "body": "Be careful, it's really hot!"  
    },  
    "badge": 12  
  }  
}
```

Sound

When alerts arrive, it’s possible to have a notification sound play. The most common value is simply the string `default`, which tells iOS to play the standard alert sound. If you want to use a custom sound included in your app’s bundle, you can instead specify the name of a sound file in your app’s main bundle.

Sounds must be 30 seconds or less. If they’re any longer than 30 seconds, iOS will ignore your custom sound and fall back to the default sound.

Note: Be very careful with custom or long sounds! It seems like a great idea when developing your app, but ask yourself this — will your end users appreciate your unique sound or the length of the sound? Be sure to do some user acceptance testing before deploying to the App Store.

I had a client, for example, who provided a sports team management app. When a notification was delivered, it played the sound of a baseball being hit by a bat and the crowd roaring. Everybody thought it was pretty cool for about two days, and then he started getting bug reports begging him to remove it.

You can use the `afconvert` tool on your Mac to convert your custom sound to one of the four acceptable formats:

- Linear PCM
- MA4 (IMA/ADPCM)

- μ Law
- aLaw

For example, if you have an existing .mp3 file you would run a command like so:

```
$ afconvert -f caff -d LEI16 filename.mp3 filename.caf
```

Then, you can just add that new **filename.caf** to your Xcode project and include its name in your payload:

```
{
  "aps": {
    "alert": {
      "title": "Your food is done.",
      "body": "Be careful, it's really hot!"
    },
    "badge": 12,
    "sound": "filename.caf"
  }
}
```

Critical alert sounds

If your app needs to display a critical alert, which will be discussed in Chapter 4, “Xcode Project Setup,” you’ll need to use a dictionary as the value of the sound key, instead of just a string:

```
{
  "aps": {
    "alert": {
      "title": "Your food is done.",
      "body": "Be careful, it's really hot!"
    },
    "badge": 12,
    "sound": {
      "critical": 1,
      "name": "filename.caf",
      "volume": 0.75
    }
  }
}
```

Notice the three keys used in the sound dictionary above:

- **critical**: Setting this to 1 will specify this sound is a critical alert.
- **name**: The sound file in the app’s main bundle, as explained above.
- **volume**: A value between 0.0 (silent) and 1.0 (full volume).

Other predefined keys

Apple defines three other keys as part of the `aps` dictionary, which will be discussed in greater detail in later chapters. These can be used for background update notifications, custom notification types, user interfaces and groupings of notifications.

Your custom data

Everything outside of the `aps` key is for your personal use. You'll frequently find that you need to pass extra data to your app along with a push notification, and this is where you can do so. For example, if you're writing a geocaching app, you might want to send the user the next set of coordinates to investigate. You will, therefore, send a payload like so:

```
{  
    "aps": {  
        "alert": {  
            "title": "Save The Princess!"  
        }  
    },  
    "coords": {  
        "latitude": 37.33182,  
        "longitude": -122.03118  
    }  
}
```

In Chapter 8, “Handling Common Scenarios,” you’ll learn more about how to retrieve this data inside your app. As long as all of your custom data is kept outside of the `aps` dictionary, you’ll never have to worry about conflicting with Apple.

Collapsing notifications

As discussed earlier, the payload is only one of a few things your server sends to APNs. Aside from a unique device token, you can send an additional HTTP **header** field to specify how Apple should handle your notification and how it is delivered to the user’s device. One of those headers is the `apns-collapse-id` HTTP header field. It’s unclear why Apple chose to put this as a header, instead of part of the payload, but since it changes the content of the notification displayed, we’ll discuss it, here.

Apple recently added the ability to collapse multiple notifications down to one when a newer notification supersedes an older one.

For example, if you’re using a notification to alert users as to how many people have completed the scavenger hunt so far, you really only need to know the current total.

While you’re still diligently searching for that elusive item, three other people might have completed the game. Each time another person finds all their items, a push notification is sent to you. When you get the time to check on the status, you really don’t want to see three separate notifications saying somebody has finished. Wouldn’t you rather see a single notification saying three people are done? That’s exactly what this header field is for. You can put any unique identifier into the field, up to 64 bytes. When a notification is delivered, and this value is set, iOS will remove any other notification previously delivered that has the same value.

In the previous example of the scavenger hunt, it would make sense to use the unique ID from your database that represents that specific game. Shy away from using things like the name of the hunt to avoid inadvertently collapsing notifications that don’t relate. Try to use guaranteed unique values instead. Any type of primary key from a database or a UUID are good examples of values to use.

Note: If you’re using a third-party delivery service, they’ll have to provide a specific location for you to identify the `apns-collapse-id`. If this is a feature you think you’ll utilize, be sure to look for it explicitly when you’re shopping for a vendor.

Key points

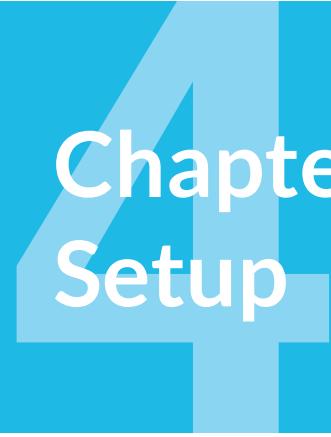
In this chapter, you covered the basics of the remote notification payload. Some key things to remember:

- Prefer to use a dictionary instead of a string for the `alert` key.
- Consider how you’re going to deal with localization issues: server-side vs. client-side.
- Utilize the `apns-collapse-id` HTTP header field when “overriding” or “updating” your notification is more appropriate than sending an additional notification.
- Place all of your custom data outside of the `aps` key to future-proof your custom keys.
- Think about whether grouping and/or collapsing your notifications makes sense.

Where to go from here?

If you want to learn more about notification payloads, you might be interested in reviewing Apple’s documentation at <https://apple.co/2Ia9iUf>.

With remote notification payloads covered, you’re now ready to set up your app to receive notifications in Chapter 4, “Xcode Project Setup.”



Chapter 4: Xcode Project Setup

Before you start sending and receiving push notifications, you first need to make sure your project is set up to do so!

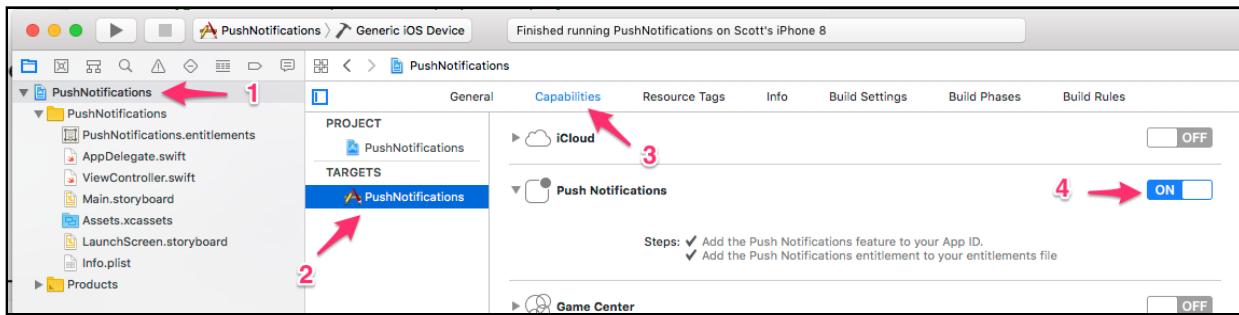
Open Xcode and create a new “Single View App” project. You may leave the checkmarks at the bottom of the project creation screen unchecked, as you won’t need Core Data or any tests in your project. In the Bad Ol’ Days, this is the point in which you’d have to set up a custom profile with Apple to enable push notifications. Fortunately, with the current toolchains, this is all automated now.

Adding capabilities

To tell Xcode that you’ll be using push notifications in this project, just follow these four simple steps so that it can handle the registration for you:

1. Press **⌘ + 1** (or **View ▾ Navigators ▾ Show Project Navigator**) to open the **Project Navigator** and click on the top-most item (e.g. your project).
2. Select the target, *not* the project.
3. Select the **Capabilities** tab.
4. Toggle the selector for **Push Notifications** to **ON**.

5. Notice the **Steps** below the **Push Notifications** selections and wait for them to be done and marked with a checkmark.



If you were to go back to the Member Center now and look at your provisioning profiles, you'd see that one has been generated specifically for this project with push notifications enabled. Well, that was so easy that it makes you wonder why Apple didn't make it this easy from day one!

Registering for notifications

You've told Apple that you're going to send push notifications. Next, you'll have to add the required code to prepare your app for receiving push notifications. As push notifications are an opt-in feature, you'll have to request permissions from the user to enable them.

Because users can turn off notifications at any point, you need to check for whether or not they are enabled *every time* the app starts. The very first time, and only the very first time that your app requests access to push notifications, iOS will display an alert asking the user to accept or reject notifications. If the user accepts, or has previously accepted, you can tell your app to register for notifications.

Open up **AppDelegate.swift** and replace its contents with the following code:

```
import UIKit
import UserNotifications

@UIApplicationMain
class AppDelegate: UIResponder {
    var window: UIWindow?
}

extension AppDelegate: UIApplicationDelegate {
    func application(_ application: UIApplication,
                    didFinishLaunchingWithOptions launchOptions:
                        [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        UNUserNotificationCenter.current().requestAuthorization(options: [
            UNAuthorizationOption.Alert, UNAuthorizationOption.Sound])
    }
}
```

```
    .badge, .sound, .alert
]) { granted, _ in
    guard granted else { return }

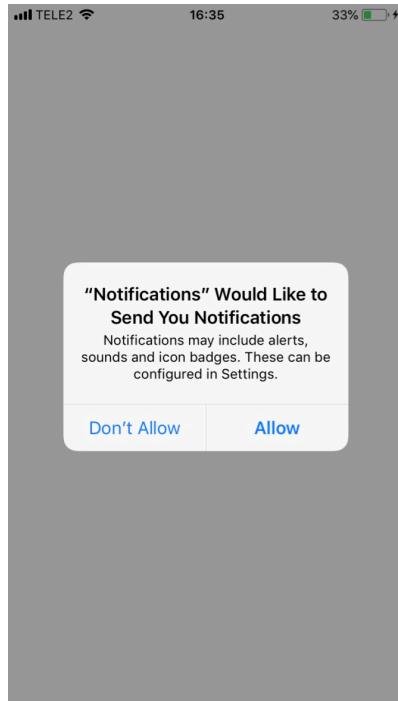
    DispatchQueue.main.async {
        application.registerForRemoteNotifications()
    }
}

return true
}
```

Notice the addition of a new import statement to pull in the `UserNotifications` system framework. It's a bit hard to read due to the long length of the function name but, essentially, whenever the app starts, you request authorization from the user to send badges, sounds and alerts to the user. If any of those items are granted by the user, you register the app for notifications.

Note: The notification closure is *not* run on the main thread, so you must dispatch the actual registration method to the main thread of your app using the main `DispatchQueue`.

In order to test push notifications, you'll have to run the app *on an actual device*, and *not* in the simulator. If you build and run your app now, you'll see the request to allow notifications.



Since you're reading this book, you must want to see the notifications, so click on **Allow** in the alert.

Provisional authorization

An alert like the one above can be somewhat jarring to a user when the app starts up the first time. Why are they being asked for this? What type of data are you going to send? If you talk to your friends and colleagues, you'll likely find that a surprising number of people, especially older people, simply reject all notifications.

To work around this issue, Apple provides another useful case for the `UNAuthorizationOptions` enum that you can pass to `requestAuthorization` during setup. If you include `.provisional` in the `options` argument, notifications will automatically be delivered silently to the user's Notification Center, without asking for permission – there will be no sound or alerts for these provisional notifications.

The benefit of this option is that users who look in Notification Center can see your notifications and decide if they're interested in them or not. If they are, they simply authorize them from there, resulting in future notifications appearing as regular push notifications. That's quite a nice feature to include via a simple flag!

Critical alerts

There's another type of authorization that you might need to request, depending on the type of app you're building. If your app has something to do with health and medical, home and security, public safety or anything else that may have the need to present a notification even if the user declined alerts, you can ask Apple to configure critical alerts via the `.criticalAlert` enum case. Critical alerts will bypass the Do Not Disturb and ringer switch settings as well as always play a sound... even a custom sound.

Due to the disruptive nature of critical alerts, you must apply for a special entitlement from Apple to enable them. You can do that through the Apple Developer Portal (<https://apple.co/2JwRvby>)

Getting the device token

If your app successfully registered for notifications, iOS will call another delegate method providing a device token to your app. The token is an opaque data type which is globally unique and identifies one app-device combination with the APNs.

Unfortunately, iOS provides this to you as a `Data` type instead of a `String`, so you'll have to convert it since most push service providers expect a string.

Paste the following code into your `AppDelegate` extension:

```
func application(_ application: UIApplication,  
didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data) {  
    let token = deviceToken.reduce("") { $0 + String(format: "%02x", $1) }  
    print(token)  
}
```

The token is basically a set of hex characters; the above code simply turns the token into a hex string. There are multiple methods that you'll see on the internet for how to convert the `Data` type to a `String`. Use whatever method seems most natural to you. `reduce` is a method that combines the elements of a sequence into a single value using the given closure. So, in this case, you're simply taking each byte and converting it to hex, then appending it to the accumulated value.

Note: Never make an assumption about the length of the token. Many tutorials you find will hardcode the length of a token for efficiency. Apple has already increased the token length once before from 32 to 64 characters. When you store your device tokens in something like a SQL database, be sure to not hardcode a length or you may have issues in the future.

Also, keep in mind that the device token itself can change. Apple will issue a new device token when the user installs the app on another device, restores from an old backup, reinstalls iOS and in some other cases. You should never try and link a token to a specific user.

Build and run the app. You should see a device token (a string of random characters) in the Xcode console window:



Key points

- You must tell Xcode that push notifications will be part of your project; follow the steps in this chapter so that Xcode can handle the registration for you.
- You must add the required code to prepare your app for receiving push notifications.
- Push notifications are an opt-in feature, so you must request permissions from the user to enable them.
- To avoid jarring notifications the first time a user opens an app, use **provisional authorization** so that notifications are delivered silently to the user's Notification Center, without asking for permission.
- For **critical alerts** that override a user's declined alerts, you must apply for a special entitlement from Apple to enable them, due to their disruptive nature.
- Once you have successfully registered your app for notifications, iOS will call another delegate method providing a device token to your app. Never make assumptions about the length of your token or try to link the token to a specific user.

Where to go from here?

At this point, you've technically done everything necessary to make your app capable of receiving and displaying a push notification. In the next chapter, you'll get your **Authentication Token** from Apple so that Apple's servers will allow you to send notifications and you'll finally send your first push notification!

Chapter 5: Apple Push Notification Servers

In the last chapter, you set up your app to be able to receive push notifications. The last piece that you'll need in order to have your app receive a push notification is an **Authentication Token** used by Apple's servers to trust your app. This token validates your server and makes sure that there's always a secure connection between your backend and APNs.

Token types

When Apple first started allowing sending push notifications, it used the **PKCS #12** archive file format, also commonly known as the **PFX** format.

If you've ever worked with push notifications in the past, this file ends with the `.p12` file extension.

This type of format was quite cumbersome to work with for multiple reasons:

- They are only valid for a single year, requiring yearly “maintenance” of your certificates.
- You need separate certificates for both production and development distributions.
- You need separate certificates for every app you publish.
- Apple does not provide the certificate in the “final” format you’ll actually need to send notifications, requiring you to run multiple `openssl` commands from Terminal for the multiple conversions; usually requiring a bit of research to remember how.

Around 2016, in order to work around the above problems, Apple started supporting the industry standard **RFC 7519**, better known as **JSON Web Tokens**, or **JWT** (<https://jwt.io/>). These tokens use the newer **.p8** file extension.

Apple, of course, likes its own names and so all of its documentation on push notifications refers to these as **Authentication Tokens**.

Changing to the newer format alleviated all the issues of the **PFX** file format as they do not need to be renewed, don't differentiate between production and development, and can be used by **all** of your apps.

Unfortunately, Apple did very little other than say, “There it is!”, when it released it. Unless you already had experience with HTTP/2 and JWT, you were stuck. We'll remedy that now!

Getting your Authentication Token

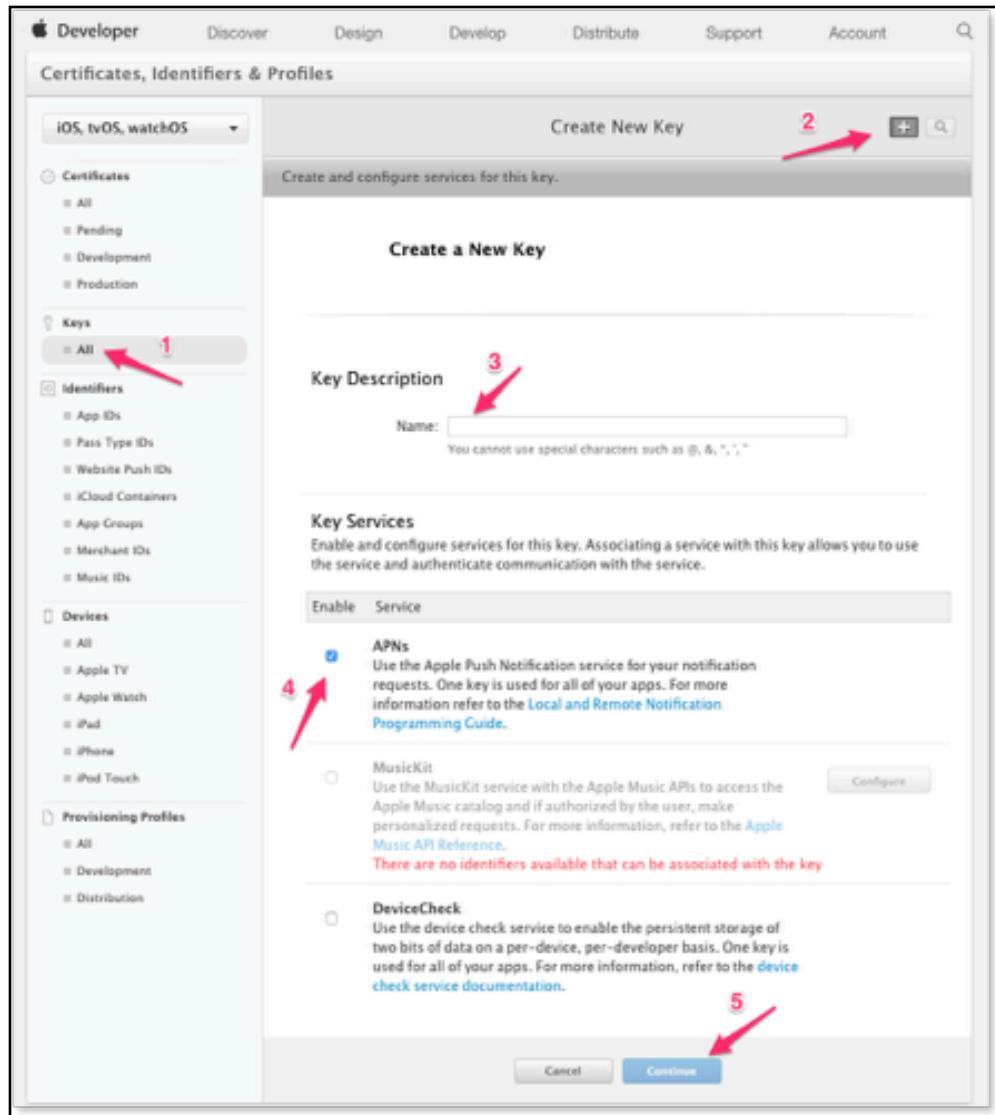
Creating the Authentication Token is a simple process that you only have to do **once**. In a browser of your choice, go to the **Member Center** (<https://apple.co/2HRPzxx>), and sign in with your Apple ID.

1. In the side-bar, under the **Keys** section, click on **All**.
2. In the upper-right corner, click on the **+** button to create a new key.
3. Name the key **Push Notification Key**.
4. Under **Key Services**, select the checkbox next to **APNs**.
5. Press **Continue**.

After these steps, simply confirm your key creation and download the file. By default, it will go to your **Downloads** directory and will be named something like **AuthKey_689R3WVN5F.p8**. The **689R3WVN5F** part is your **Key ID**, which you'll need when you're ready to send a push.

You'll also need to know your **Team ID**, so grab that now. Still in the Member Center, at the very top-right, click on **Account**. On the page that loads, in the sidebar, click on **Membership**. Your Team ID will be displayed on the details page.

The steps just described are illustrated, here:



Sending a push

At this point, you have everything you need to send a push notification to your app. You'll need some way to actually configure the push and send it manually. There are many free, open-source, projects on GitHub to provide this functionality; consider using **PushNotifications** (<https://bit.ly/2jvEUtK>) as it supports the newer Authentication Keys, which some of the other apps don't. You can use any of the apps out there as long as they support Authentication Keys.

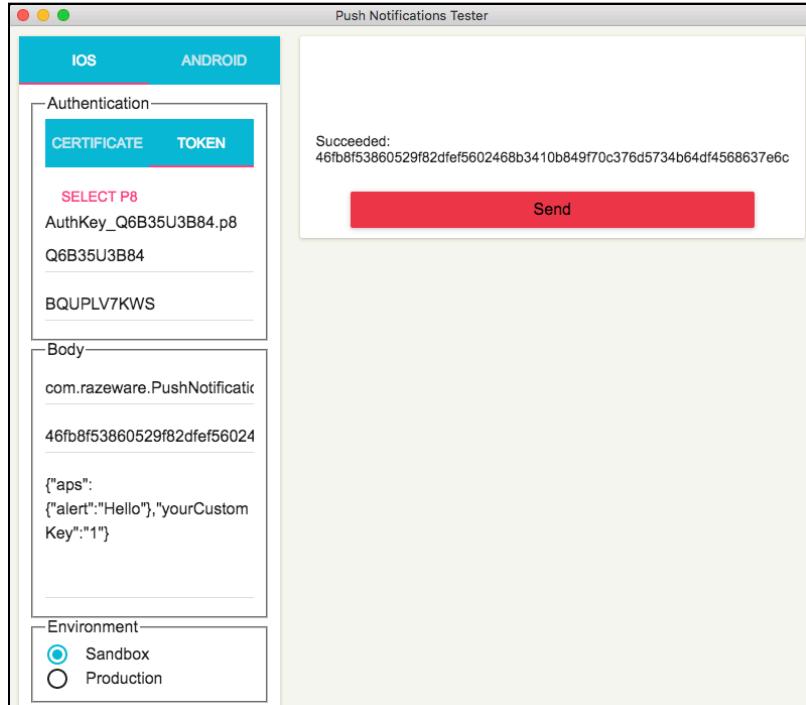
Note: The simulator does not support push notifications.

Build and run the app from Chapter 4, “Xcode Project Setup,” in Xcode, being sure to run against a physical device as the simulator does not support push notifications. Once the app has successfully launched, move the app to background, by either switching to your home screen or locking the device. With the way your notifications are configured right now, you’ll only see them if you are *not* currently running the app in the foreground. You’ll fix that later on in the book, in Chapter 8, “Handling Common Scenarios.”

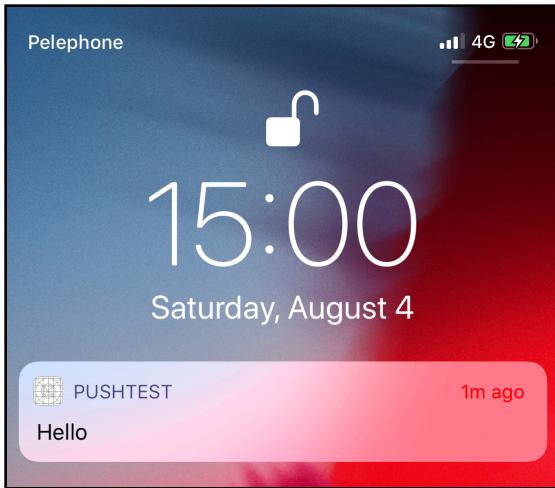
In your Xcode **console window** ($\Delta + \text{⌘} + \text{C}$), you’ll see a long hex string printed out, which is the token from your print call during registration. Copy that string into your clipboard.

Now, launch the **PushNotifications** app you downloaded from GitHub (or any other similar tool).

You’ll need to be sure to select the **TOKEN** authentication option and then select the p8 file you downloaded from the Developer Portal, and fill in your Key ID, Team ID, Bundle ID and Device Token.



You don't need to change the payload at all. Once you press **Send**, you should see a notification appear on your device shortly thereafter!



Key points

- For your app to receive push notifications, you must have an **Authentication Token** used by Apple's servers to trust your app.
- The authentication token validates your server and makes sure that there's always a secure connection between your backend and APNs.
- Creating the Authentication Token is a simple process that you're only required to do once; follow the steps in the chapter to create yours.
- To configure the push and send it manually, there are many free, open-source, projects on GitHub to provide this functionality — just make sure whatever you use supports Authentication Keys.

Where to go from here?

Congratulations! You've sent and received your first push notification! In the next chapter, Chapter 6, “Server Side Pushes,” you will configure a web and SQL server to handle the API necessary for storing device tokens and sending push notifications from your own server, instead of using a manual tool like in this chapter.

Chapter 6: Server Side Pushes

While you've successfully sent yourself a notification, doing this manually won't be very useful. As customers run your app and register for receiving notifications, you'll need to somehow store their device tokens so you can send them notifications at a later date.

Using third-party services

There are a slew of services online that will handle the server side for you. You can simply search Google for something along the lines of "Apple push notification companies" and you'll find multiple examples. Some of the most popular one are:

- Amazon Simple Notification Service (SNS) (<https://amzn.to/2rk6VIQ>)
- Braze (<https://bit.ly/2yM4hx7>)
- Firebase Cloud Messaging (<http://bit.ly/2Nq4b5x>)
- Kumulos (<https://bit.ly/2FIQ8Dy>)
- OneSignal (<https://bit.ly/1Ukk3WL>)
- Urban Airship (<https://bit.ly/1QymqCY>)

Each company will vary in its pricing and API, so discussing any specific service is beyond the scope of this book. If you want to get running quickly or don't want to deal with anything on the server side, then solutions like the above may be perfect for you.

You may find, however, that you prefer avoiding third-party services, as you can run into issues if the service changes how its API works or if the company goes out of business. These services will usually also charge a fee based on how many notifications you send.

As an iOS developer, you might already be paying for a web hosting service for your website, which gives you the tools you need to do this work yourself — and you can find multiple vendors that charge \$10 or less per month. Most web hosting services provide SSH access and the ability to run a database. Since handling the server side only requires a single database table, a couple of REST endpoints and a few easy-to-write pieces of code, you may want to do this work yourself.

If you have no interest in running your own server, you can skip to Chapter 7, “Expanding the Application.”

Setting up Docker

If you’d like to follow along with this chapter but don’t have a server readily available to use, don’t fret! Utilizing **Docker**, you can run a local SQL server without modifying your system. If you don’t already have Docker installed, you can go to the Docker for Mac (<https://dockr.ly/2JOzJ31>) site and follow the installation instructions. Since you’ll be using the Docker CLI tools, you might need to use the `docker login` command for the initial setup.

Setting up a SQL server

The first step is to set up a SQL server to store the device tokens that your users send to you. **PostgreSQL** is a great service that is readily available, but any SQL server you wish to use will do.

This book assumes basic database knowledge, so this won’t be covered in this book. You might use the `psql` command directly or maybe use a tool like **Navicat** for PostgreSQL; however, since all of your interactions will be through your app, you technically don’t need to know anything about it!

You’ll need to create a new database user called **apns**, and then create a new database named **apns** that’s owned by your newly-created **apns** user. If you’re not using Docker for this, you’ll want to run commands similar to the following:

```
$ createuser -P -h yourServer -U postgres -W apns
$ createdb -O apns -h yourServer -U postgres -W apns
```

You’ll need to replace the `-h` argument with the name of the host running your SQL server.

If you *are* using Docker, then the startup command will create the user and database for you. In that case, you'll want to run this command from **Terminal**:

```
$ docker run --name postgres \
-e POSTGRES_DB=apns \
-e POSTGRES_USER=apns \
-e POSTGRES_PASSWORD=password \
-p 5432:5432 \
-d postgres
```

Those \ characters allow you to run multiline commands in your shell. The Docker command will spin up a process that's running PostgreSQL and map port 5432 on your local machine to port 5432 in the Docker image, effectively mapping the default PostgreSQL port for you.

Next, you'll set up your server-side application, which will already handle the creation of the table for you. But, in essence, it'll run something similar to this SQL command:

```
CREATE TABLE tokens (
    id UUID PRIMARY KEY,
    token TEXT UNIQUE NOT NULL,
    debug BOOLEAN NOT NULL
);
```

Setting up Vapor

Now that you have somewhere to store your device tokens, you need *something* to handle your web connections. For this tutorial, you'll use **Vapor** for this purpose. Vapor is a very well supported implementation of server side development using Swift. Without too much code you can use it to control your SQL database as well as your RESTful API. In order to use Vapor, though, there's a little bit of setup that needs to happen.

If you're not familiar with Vapor, you can find a list of resources at the end of this chapter.

It's time to create your server app, which will allow you to store your web tokens!

Open the **projects/starter** folder, where you'll find a starter project called **PushNotifications** which has the base configuration already taken care of for Vapor. It has been modified to use PostgreSQL as the database instead of SQLite and contains empty files for your model and a controller so that Vapor knows about them.

If you don't have Vapor installed, run the following command in **Terminal**:

```
$ brew install vapor/tap/vapor
```

Note: If you don't have Homebrew already installed, install it by following the instructions at <http://brew.sh>.

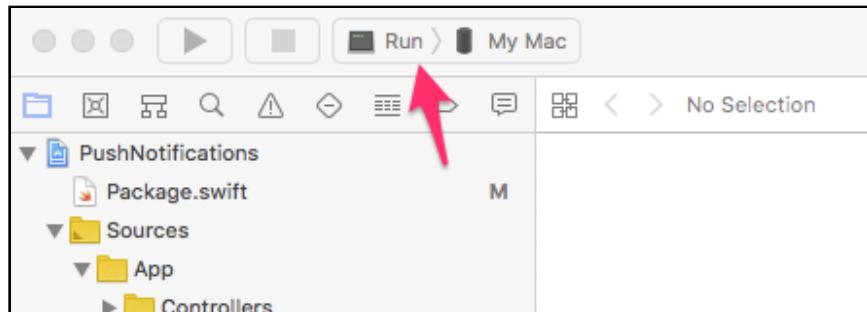
The way Vapor works, under the hood, is by generating the Xcode project for you using numerous dependencies. So the very first task is let Vapor create that. In your **Terminal** window, navigate to the root folder of the starter project and run the following command:

```
$ vapor xcode -y
```

The above will take a bit to download all of the project's dependencies and create **PushNotifications.xcodeproj** for you. Once it's done, Xcode will automatically start up.

Creating the model

Vapor projects include numerous Xcode schemes and, if you're not using the proper one, the compile/run phases will fail with an overwhelming number of compiler errors! Always make sure that the active scheme is set to **Run** and **My Mac**.



The device token you receive from Apple is the *model* that you'll store. Edit the **Sources/App/Models/Token.swift** file and add the following code into it:

```
import FluentPostgreSQL
import Vapor

final class Token: PostgreSQLUUIDModel {
    var id: UUID?
    let token: String
    let debug: Bool

    init(token: String, debug: Bool) {
```

```
        self.token = token
        self.debug = debug
    }

extension Token: Migration {
    static func prepare(on connection: PostgreSQLConnection) ->
        Future<Void> {
        return Database.create(self, on: connection) { builder in
            try addProperties(to: builder)
            builder.unique(on: \.token)
        }
    }
}

extension Token: Content {}
extension Token: Parameter {}
```

As you can see, this is a simple model with three properties. When you’re creating a new token, you’ll obviously not have an ID to specify, which is why that has to be specified as an optional value. Because APNs tokens are unique, you add a unique constraint to that field in your `prepare(on:)` method.

The `Migration` code is what Vapor uses in order to properly create the database schema. This simply tells PostgreSQL to create the table if it doesn’t already exist, make a column for each property in the `Token` class and then ensure that the `token` column has a `UNIQUE` constraint assigned to it.

If you take a minute to think about this structure, you’ll realize that you could, in fact, use this same model for **all** of your iOS apps.

Note: Since you can use the same model for all of your iOS apps, there’s technically no reason to have a separate table for each app. You could, for example, expand the `Token` class to include an `appIdentifier` property and then you’d just need one database and one table for everything.

Creating the controller

Now that you’ve got a model, you’ll need to create the controller that will respond to your HTTP POST and DELETE requests. Edit the `Sources/App/Controllers/TokenController.swift` file to use the model. Navigate to the file and add the following code:

```
import FluentPostgreSQL
import Vapor

final class TokenController: RouteCollection {
```

```
func boot(router: Router) throws {
    let routes = router.grouped("api", "token")
    routes.post(Token.self, use: storeToken)
    routes.delete(String.parameter, use: removeToken)
}
```

Here, you create a controller, which determines what will happen when somebody sends a request to your server. You create two new routes:

1. POST /api/token, which calls the `storeToken` function. The client will send the new token as JSON inside the request.
2. DELETE /api/token/`token_id`, which calls the `removeToken` function.

Now you need to actually implement these functions! Add the following method to the end of the class, below the `boot(router:)` method:

```
func storeToken(_ req: Request, token: Token) throws -> Future<Token> {
    return token.save(on: req)
}
```

When storing a token, Vapor will decode the request's JSON body as long as it matches your `Token` object. If it does, then the data is saved to the database and the new JSON representation of the stored token is returned.

Next, add this method to the end of the class to delete a token:

```
func removeToken(_ req: Request) throws -> Future<HTTPStatus> {
    let tokenStr = try req.parameters.next(String.self)

    return Token.query(on: req)
        .filter(\.token == tokenStr)
        .delete()
        .transform(to: .ok)
}
```

When sending a DELETE request, the token is sent as part of the URL's query string and later used to remove the given token from the database.

Notice how it doesn't really matter whether or not the token existed in the database during a deletion; you always return a successful response. Since Vapor works asynchronously with futures, you'll actually *return* from the method before the database operation has completed. What this means is that the API method could finish before the database has actually deleted your token. For a better understanding of asynchronous database access and futures, please refer to the resources listed at the end of this chapter.

Also notice that, in your `removeToken` method, you're actually taking the token value itself and not the ID of the token. The normal case is that you'll try to send a push notification, get a failure and want to remove the failed token. There's no reason to force your callers to store the ID of the token itself.

You might be wondering why there are no methods to *get* a token. If you consider the usage of the API, you need to store and delete tokens, but there's never a case in which you would want to let someone query your tokens via the API, since your app is the one that has access to the user's push token.

Note: Are you still thinking about the challenge to support all your apps mentioned during the model creation? The only change you'll have to make here is accepting a second parameter in the `DELETE` request that identifies the `appIdentifier`.

Updating the routes

In order to tell the app how to route requests to your new controller, you'll need to make some changes to the **Sources/App/routes.swift** file.

You'll notice that the default template has these two lines commented out:

```
//let tokenController = TokenController()  
//try router.register(collection: tokenController)
```

Uncomment those now. Vapor 3 adds syntax to allow you to embed all the routing for a controller into the controller itself so that you only need to edit the **Sources/App/routes.swift** file when you add or remove a controller from your project.

Configuring the app

Because you're running the server locally, you'll have to take an extra step to tell Vapor that it should respond to more than just local connections. You *only* have to do this during development and it's as simple as telling Vapor what IP address it should use when listening for network connections.

Copying your IP address

Click on the Apple icon in your Mac's menubar and then choose the **System Preferences...** option.

Select the Network option and then choose the **Advanced...** button. Finally, select **TCP/IP** and you'll see your **IPv4 Address**. Copy your IP address and close those preference windows.

Back in your Xcode project, open up **Sources/App/configure.swift** and find the line that currently says this:

```
severConfig.hostname = "192.168.1.1"
```

Replace that IP address with your IP address. Now Vapor knows to accept connections from outside of your mac. Throughout the rest of this book, whenever an example refers to 192.168.1.1 you should replace that with *your machine's* IP address.

Note: This is your internal IP address, not what's visible outside your network. Do not try to use a webpage like www.whatismyip.org to get this value! The most common way to get this value is using `ifconfig` from **Terminal**. You may further filter this command by piping `grep`, like so: `ifconfig | grep "inet "`.

Running the migrations

There's just one step left to make everything work. You have to tell Vapor that it should run the migrations for the `Token` class. While still in **configure.swift**, find this commented out line at the bottom of the file.

```
//migrations.add(model: Token.self, database: .psql)
```

Uncomment that line and run your project. As long as you have a PostgreSQL server running on port 5432 of `localhost`, your output should be similar to the following:

```
[ INFO ] Migrating 'psql' database (FluentProvider.swift:28)
[ INFO ] Preparing migration 'Token' (MigrationContainer.swift:50)
[ INFO ] Migrations complete (FluentProvider.swift:32)
Running default command: /Users/scott/Library/Developer/Xcode/
DerivedData/dts-fdkxzqveujzmdycmravnzrydjvwn/Build/Products/Debug/Run
serve
Server starting on http://192.168.1.1:8080
```

If you're getting error messages related to NIO, that usually means there's a problem connecting to your database. Some common items you may want to look into if you get errors:

- Is another webserver running on port 8080? Try `lsof -i :8080`.
- Is the Docker instance running? Try `docker ps`.

- You might need to use `docker ps` to find your container, and then stop it with `docker stop <container-id>`, and re-run the docker setup command from earlier in this chapter.
- Are the database, user and password all configured correctly?

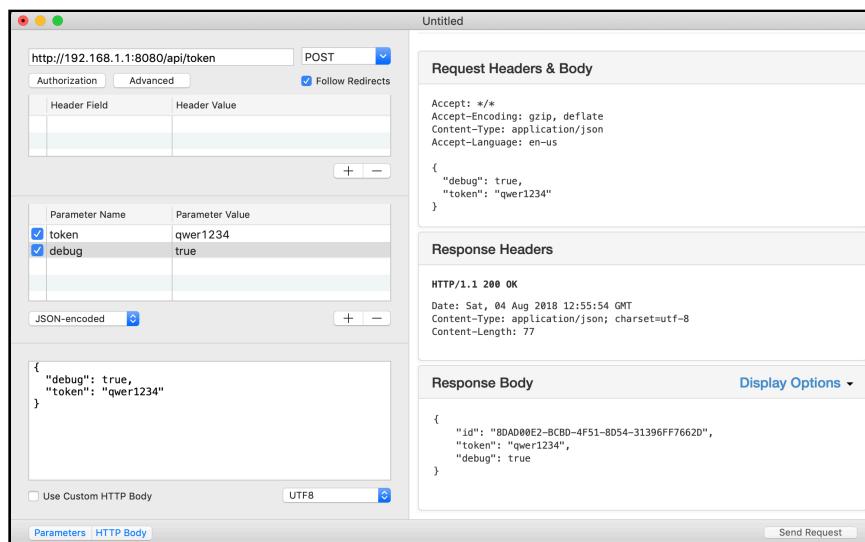
Testing your API

At this point, you can use any REST-capable app to test out your endpoints. A good choice is Rested, which is available as a free download from the Mac App Store at <https://apple.co/2HP0IEH>.

To test your POST endpoint, set up the request as follows:

- **URL:** `http://192.168.1.1:8080/api/token` (Use your IP address).
- **METHOD:** POST.
- Add a parameter called **token** and put any value you like.
- Add a parameter called **debug** and put the value `true`
- Select **JSON-encoded** as the request type. This ensures that the data is sent as JSON and that the Content-Type header is set to `application/json`.

Your request will look similar to the following:



Press the **Send Request** button. You should see in the **Response Body** section at the lower-right of the image that your token was stored in the database and given a unique identifier.

Sending pushes

As surprising as it is, Apple has not provided any way for a Swift app to natively send a push notification. Sending a push notification uses HTTP/2 now, but the standard Foundation classes don't support it well. While you *could* use a `URLSession` method to send an HTTP/2 packet, there's no way to tell the session to stay open. What this means is that every push you send creates a new connection to the APNs and Apple will therefore eventually consider you to be attempting a denial-of-service attack.

Note: The **SwiftNIO** group is working on adding HTTP/2 support to their package. Once that is done, you'll be able to write a native Swift app to send push notifications. Until then, you'll have to use something else. Once Vapor is able to send pushes, we'll release an updated version of this book showing that solution.

The current workaround is to use `libcurl` to send your pushes. Before doing anything though, you'll need to make sure that the `curl` command built for your system supports HTTP2. Run it with the `-V` flag and ensure you see **HTTP2** in the output:

```
$ curl -V
curl 7.48.0 (x86_64-pc-linux-gnu) libcurl/7.48.0 OpenSSL/1.0.2h zlib/
1.2.7 libidn/1.28 libssh2/1.4.3 nghttp2/1.11.1
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps
pop3 pop3s rtsp scp sftp smb smbs smtp smtps telnet tftp
Features: IDN IPv6 Largefile NTLM NTLM_WB SSL libz TLS-SRP **HTTP2**
UnixSockets
```

On to the script! Create a new file using your favorite editor called **sendPushes.php**. This isn't part of your Xcode project so store it wherever you're keeping your webserver's source files. You'll create a small PHP script that will send a HTTP/2 network request to APNs.

Firstly, you'll need to specify your Auth Key details and what the payload will be.

```
<?php

const AUTH_KEY_PATH = '/full/path/to/AuthKey_keyid.p8';
const AUTH_KEY_ID = '<your auth key id here>';
const TEAM_ID = '<your team id here>';
const BUNDLE_ID = 'com.razeware.APNS';

	payload = [
	'aps' => [
```

```
'alert' => [
    'title' => 'This is the notification.',
],
'sound'=> 'default',
],
];
```

Fill in those values based on your specific details. Recall that the AUTH_KEY_ID is the middle part of the filename that you downloaded from Apple, and your TEAM_ID comes from your developer account’s Membership page (<https://apple.co/2tXpJ2m>). Be sure that you specify a fully qualified path to the Auth Key file!

Next, create a method to get your list of tokens. This will obviously be very app-specific, but as a simple example, you can just get all the registered tokens in the database.

Add the following code below your \$payload variable:

```
$db = new
PDO('pgsql:host=localhost;dbname=apns;user=apns;password=password');

function tokensToReceiveNotification($debug)
{
    $sql = 'SELECT DISTINCT token FROM tokens WHERE debug = :debug';
    $stmt = $GLOBALS['db']->prepare($sql);
    $stmt->execute(['debug' => $debug ? 't' : 'f']);

    return $stmt->fetchAll(PDO::FETCH_COLUMN, 0);
}
```

Notice how you’re differentiating between debug and production tokens.

Note: Any app that was installed directly via Xcode is considered a debugging app and must be sent to a different server than apps installed via TestFlight or the App Store. More on this in a moment.

The only tricky part to sending a push notification using the newer HTTP/2 protocol is getting the authentication header right. This is the part that Apple didn’t provide much guidance on when it released its Authentication token implementation. Append the following code:

```
function generateAuthenticationHeader()
{
    // 1
    $header = base64_encode(json_encode([
        'alg' => 'ES256',
        'kid' => AUTH_KEY_ID
    ]));

    // 2
}
```

```
$claims = base64_encode(json_encode([
    'iss' => TEAM_ID,
    'iat' => time()
]));

// 3
$pkey = openssl_pkey_get_private('file://' . AUTH_KEY_PATH);
openssl_sign("$header.$claims", $signature, $pkey, 'sha256');

// 4
$signed = base64_encode($signature);

// 5
return "$header.$claims.$signed";
}
```

The above code takes care of generating the needed JWT authentication header. Let's break it down:

1. You specify that the encryption algorithm (`alg`) is using the SHA-256 hash algorithm and that the key identifier (`kid`) is the 10-character identifier from your p8 file.
2. Next, you'll generate the claims payload by specifying the issuer (`iss`) using your 10-character Team ID, obtained from your developer account (<https://apple.co/2tXpJ2m>), along with the issue time (`iat`), e.g. when the JWTs was generated, in terms of the number of seconds since the epoch, in UTC.
3. You read your p8 auth key file and digitally sign the header and claim into `$signature`.
4. You take your digitally signed `$signature` and encode is using Base 64.
5. Finally, you wrap it up by concatenating all 3 pieces, which you'll pass down to the Authentication header.

The only signature algorithm that Apple accepts is the ES256 algorithm. Don't try to sign the payload with any other algorithm or Apple will send a `InvalidProviderToken` (403) response to your request.

You should generate a new authentication header at the start of every group of pushes that you'll be sending. Additionally, these generated tokens last for about an hour; any request sent with a token older than an hour will be rejected by Apple with a `ExpiredProviderToken` (403) error.

You'll notice that nothing here actually encrypts the header. JWTs are signed and encoded, but they do nothing to provide security for sensitive data.

Now that you know what tokens you need to send and how to sign your request, you'll open an HTTP/2 session to the APNs. Add the following function to the file:

```
function sendNotifications($debug) {
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_2_0);
    curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($GLOBALS['payload']));
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_HTTPHEADER, [
        'apns-topic: ' . BUNDLE_ID,
        'authorization: bearer ' . generateAuthenticationHeader(),
    ]);
}
```

Notice how you're explicitly telling `libcurl` that it should use the HTTP/2 protocol for this connection while passing your JWT as the authentication header. At this point, the session is open and signed, so you just need to loop through each token and send your payload across. Add the following code to the end of the `sendNotifications` function:

```
$removeToken = $GLOBALS['db']->prepare('DELETE FROM apns WHERE token
= ?');
$server = $debug ? 'api.development' : 'api';
$tokens = tokensToReceiveNotification($debug);
```

This creates a PDO statement to remove a single token from the database, determines which APNs to connect to and then queries all of the tokens using your previously defined function. You're almost done, keep going! Add this final piece of PHP code inside your `sendNotification` function:

```
foreach ($tokens as $token) {
    // 1
    $url = "https://$server.push.apple.com/3/device/$token";
    curl_setopt($ch, CURLOPT_URL, "{$url}");

    // 2
    $response = curl_exec($ch);
    if ($response === false) {
        echo("curl_exec failed: " . curl_error($ch));
        continue;
    }

    // 3
    $code = curl_getinfo($ch, CURLINFO_HTTP_CODE);
    if ($code === 400 || $code === 410) {
        $json = @json_decode($response);
        if ($json->reason === 'BadDeviceToken') {
            $removeToken->execute([$token]);
        }
    }
}

curl_close($ch);
```

Here's what's happening in the above code:

1. You construct the actual URL to be used to send a notification to this token.
2. You try to submit the request to Apple over the cURL HTTP/2 Session you previously opened.
3. If Apple said something was wrong, and the reason was that the token was bad (`BadDeviceToken`), you remove this token from your database using the `$removeToken` PDO statement you prepared earlier. A token will become invalid if the user uninstalls your app.

Now all that you need to do is call the function! Add this code to the end of the file:

```
sendNotifications(true); // Development (Sandbox)
sendNotifications(false); // Production
?>
```

Depending on the way your development cycle works, you'll need to determine which type of tokens you're sending your push notifications to. At the start of development, when you're the only user, you'll just call `sendNotifications(true)`. Once you have some beta testers, you'll have to start calling it again with `false` so they get notifications. There will then be a period of time where both have to go out.

What happens when you finally push your app to the App Store? That's again dependent on your development flow. While you continue to develop some other awesome features, you'll probably continue to send both Sandbox and Production notifications during your development and release cycle.

A PHP solution should support most server types. Another option would be using Node.js for your server, in which case you're not forced to add a PHP solution. There are multiple options on GitHub that you can use. For example, if you install the `apn` and `pg` modules using **Terminal**:

```
$ npm install apn --save
$ npm install pg --save
```

Your Node.js server could look a lot like this:

```
#!/usr/bin/env node

var apn = require('apn');
const { Client } = require('pg')

const options = {
  token: {
    key: '/full/path/to/AuthKey_keyid.p8',
    keyId: '',
  }
}
```

```
        teamId: '',
    },
    production: false
}

const apnProvider = new apn.Provider(options);

var note = new apn.Notification();
note.expiry = Math.floor(Date.now() / 1000) + 3600; // 1 hour
note.badge = 3;
note.sound = "default";
note.alert = "Your alert here";
note.topic = "com.razeware.PushNotifications";

const client = new Client({
  user: 'apns',
  host: 'localhost',
  database: 'apns',
  password: 'apns',
  port: 5433
})

client.connect()

client.query('SELECT DISTINCT token FROM tokens WHERE debug = true',
  (err, res) => {
  client.end()

  const tokens = res.rows.map(row => row.token)

  apnProvider.send(note, tokens).then( (response) => {
    // response.sent has successful pushes
    // response.failed has error details
  });
})
}
```

But they disabled push!

You'll notice that you remove tokens from your database when a failure occurs. There's nothing there to handle the case in which your user disables push notifications, nor should there be. Your user can toggle the status of push notifications at any time, and nothing requires them to go into the app to do that, since it's done from their device's Settings. Even if push notifications are disabled, it's still valid for Apple to send the push. The device simply ignores the push when it arrives.

Note: Do *not* try detecting when pushes are off and removing the token! If the end user goes into Settings and turns them back on, but doesn't run your app again for a while, they'll miss all the notifications they are expecting to receive!

Key points

- You'll need to have a SQL server available to store device tokens.
- You'll need an API available to your iOS app to store and delete tokens.
- Do not use native Swift network commands to send push notifications until HTTP/2 becomes available, as it will appear to Apple as a denial of service attack due to repetitive opening and closing of connections.
- There are many options available for building your push server. Choose the one(s) that work best for your skillset.

Where to go from here?

As stated, if you are interested in learning more about the Vapor framework, you can check out our great set of videos at <https://bit.ly/2JTxX0B> as well as our recent book, *Server Side Swift with Vapor* at <https://bit.ly/2FI9wAR>.

In the next chapter, “Expanding the Application,” you’ll configure your iOS app to talk to the server that you just configured.

Chapter 7: Expanding the Application

Now that you've got a database up and running, you need to tell your app how to connect to it. As you saw in the previous chapter, "Server Side Pushes," Vapor will run a local server for you at **http://192.168.1.1:8080** (change with your own IP address).

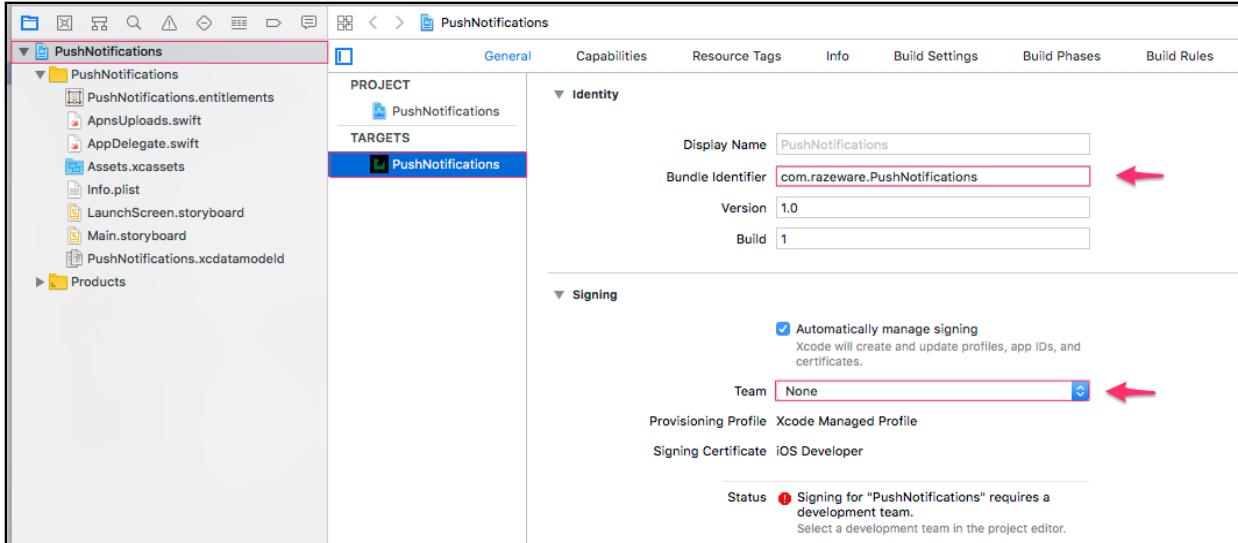
This is the URL that your app will need to talk to if you successfully registered for push notifications. Of course, remember to substitute *your* IP address in the URL.

Setting the team and bundle identifier

If you take a look at the starter project for this chapter, you'll see that it's a standard app that uses push notifications, utilizing the code that you used in previous chapters. Before you can use the project, however, you'll have to configure the target so that the certificates are created for you. To do so:

1. Select **PushNotifications** from the Project navigator (**⌘ + 1**).
2. Select the **PushNotifications** target.
3. Change the **Bundle Identifier**.
4. Choose your **Team**.

Follow steps 1–4 as shown below:



Remember, also, to turn on the Push Notifications capability as discussed in Chapter 4, “Xcode Project Setup.”

Updating the server

In your Xcode project, you can now modify **AppDelegate.swift** to contact the endpoint you just set up to register your device token. Return to your Xcode project. Modify `application(_:didRegisterForRemoteNotificationsWithDeviceToken:)` method. After you print the token, add a few lines to call your new web service:

```
// 1
let url = URL(string: "http://192.168.1.1:8080/api/token")!

// 2
var obj: [String: Any] = [
    "token": token,
    "debug": false
]

// 3
#if DEBUG
obj["debug"] = true
#endif

// 4
var request = URLRequest(url: url)
request.addValue("application/json",
                 forHTTPHeaderField: "Content-Type")
request.httpMethod = "POST"
request.httpBody = try! JSONSerialization.data(withJSONObject: obj)
```

```
// 5  
URLSession.shared.dataTask(with: request).resume()
```

Here's what's happening in the code above:

1. Remember to update the IP address with the IP of your Mac that you discovered in Chapter 6, "Server Side Pushes."
2. The **POST** method that you configured to receive your token expects a JSON object, and it will always need to know the token, so you start with that. Use the **Any** type as the value of the dictionary because you'll pass along both strings and Booleans.
3. Set debug to true if you're running directly via Xcode.
4. Set up an HTTP POST request with the request body you built previously.
5. Finally, send along the request.

Notice that you're not checking the status of the request. If the registration with your website or database were to fail for any reason, you wouldn't tell your end user this as there isn't anything they can do about it anyway. Hopefully, the next time they run your app, you would've fixed the server-side issue and the registration will complete successfully.

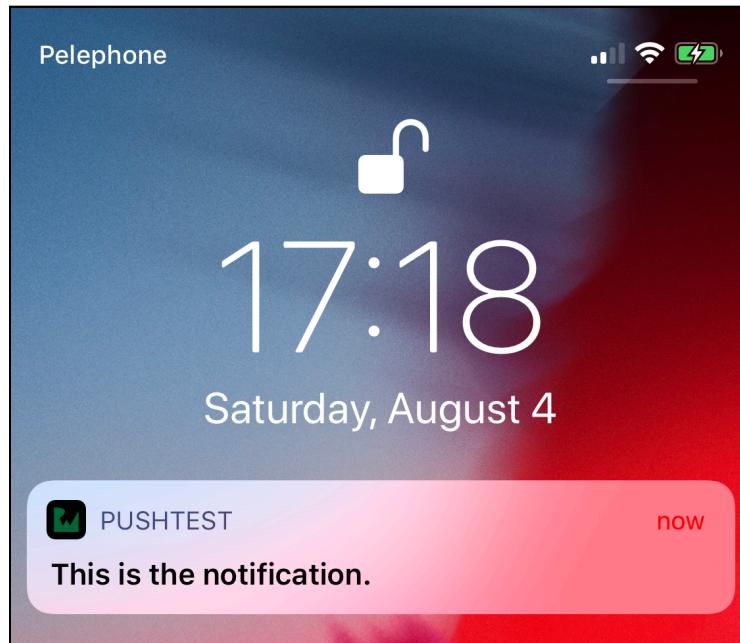
If you try to run this now, you'll get errors as Apple blocks the URL due to App Transport Security, or ATS. Since this is only a development example, you can disable that security measurement, but for a production app, you'd probably never want to do this:

1. Edit the **Info.plist** file.
2. Right-click in the empty whitespace and choose **Add Row**.
3. Set the key to be **App Transport Security Settings**.
4. Expand your newly created **App Transport Security Settings** using the left triangle.
5. Right-click on **App Transport Security Settings** and choose **Add Row**.

6. Set the key to be **Allow Arbitrary Loads** and the value to be **YES**.

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
Localization native development re...	String	\$(DEVELOPMENT_LANGUAGE)
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
► Required device capabilities	Array	(1 item)
► Supported interface orientations	Array	(3 items)
► Supported interface orientations (i...)	Array	(4 items)
▼ App Transport Security Settings	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES

Build and run your app. Put the app into the background by going to your home screen or locking your phone. At this point, if you run the **sendPushes.php** script that you created in Chapter 6, “Server Side Pushes,” you should get a push notification!



Note: For this to work, you need to make sure an instance of your Vapor server is running and configured to run on your IP address, as well as make sure your database is running. You also need to set up the **sendPushes.php** script to use your APNs token. This is all described in Chapter 6, “Server Side Pushes.”

Extending AppDelegate

You can probably already see how the push notification code is going to be almost exactly the same in every project you create. Do a little cleanup by moving this common code to an extension. Create a new file in your Xcode project called **ApnsUploads.swift** and then move your notification code over.

When you move this code, you'll also add some additional lines at the end to display the body of the JSON request in a “pretty” format that's easy for you to read. Your **ApnsUploads.swift** file should look like this:

```
import UIKit
import UserNotifications

extension AppDelegate {
    func registerForPushNotifications(application: UIApplication) {
        let center = UNUserNotificationCenter.current()
        center.requestAuthorization(options: [.badge, .sound, .alert]) {
            granted, _ in
            guard granted else { return }

            DispatchQueue.main.async {
                application.registerForRemoteNotifications()
            }
        }
    }

    func sendPushNotificationDetails(to urlString: String,
                                     using deviceToken: Data) {
        guard let url = URL(string: urlString) else {
            fatalError("Invalid URL string")
        }

        let token = deviceToken.reduce("") {
            $0 + String(format: "%02x", $1)
        }

        var obj: [String: Any] = [
            "token": token,
            "debug": false
        ]

        #if DEBUG
        obj["debug"] = true
        #endif

        var request = URLRequest(url: url)
        request.addValue("application/json",
                         forHTTPHeaderField: "Content-Type")
        request.httpMethod = "POST"
        request.httpBody = try! JSONSerialization.data(withJSONObject: obj)

        #if DEBUG

```

```
        print("Device Token: \(token)")

        let pretty = try! JSONSerialization.data(withJSONObject: obj,
                                                options: .prettyPrinted)
        print(String(data: pretty, encoding: .utf8)!)
    #endif

    URLSession.shared.dataTask(with: request).resume()
}
}
```

As you add more items to the data that you send during registration, these “pretty” lines become a life saver when debugging. You might, for example, want to store the users’ language preferences or the version of the app they are running.

Now, you have a nice extension that you can simply copy into any app where you want to use push notifications. Once that’s done, your `AppDelegate.swift`’s extension becomes nice and clean, and should look like this:

```
extension AppDelegate: UIApplicationDelegate {
    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    registerForPushNotifications(application: application)

    return true
}

    func application(_ application: UIApplication,
didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data) {
        sendPushNotificationDetails(to: "http://192.168.1.1:8080/api/token",
                                    using: deviceToken)
    }
}
```

Key points

- Once you have your database established, you need to tell your app how to connect to it. Vapor will allow you to run a server written with Swift.
- Take the time to add some additional lines at the end of notification registration to display the body of the JSON request in a “pretty,” easy-to-read format, which can help in the future with debugging.

Where to go from here?

And there you have it! You've successfully built an API that saves device tokens and an app that consumes that API. This is the basic skeleton you will build upon when using push notifications. In Chapter 8, “Handling Common Scenarios,” you will start handling common push notification scenarios, such as displaying a notification while the app is in the foreground... so keep reading!

Chapter 8: Handling Common Scenarios

As you noticed in previous projects in this book, iOS will automatically handle presenting your notifications as long as your app is in the background or terminated. But what happens when it is actively running? In that case, you need to decide what it is that you want to happen. By default, iOS simply eats the notification and never displays it. That's pretty much always what you want to happen, right? No? Didn't think so!

In the download materials for this chapter, you'll find possibly the coolest starter project that's ever been created.

```
sarcasm  
'sär-, kə-zəm  
noun  
the use of irony to mock or convey contempt
```

If you'd like to have iOS display your notification while your app is running in the foreground, you'll need to implement the `UNUserNotificationCenterDelegate` method `userNotificationCenter(_:willPresent:withCompletionHandler:)`, which is called when a notification is delivered to your app while it's in the foreground. The only requirement of this method is calling the completion handler before it returns. Here, you can identify what you want to happen when the notification comes in.

Note: After opening up the starter project for this chapter, remember to turn on the Push Notifications capability as discussed in Chapter 4, “Xcode Project Setup,” and set the team signing as discussed in Chapter 7, “Expanding the Application.”

Conform to the aforementioned protocol in your AppDelegate. At the bottom of **AppDelegate.swift**, write the following:

```
extension AppDelegate: UNUserNotificationCenterDelegate {
    func userNotificationCenter(
        _ center: UNUserNotificationCenter,
        willPresent notification: UNNotification,
        withCompletionHandler completionHandler:
            @escaping (UNNotificationPresentationOptions) -> Void) {
        completionHandler([.alert, .sound, .badge])
    }
}
```

Probably one of the most complex methods you've ever written, right?

You're simply telling the app that you want the normal alert to be displayed, the sound played and the badge updated. If the notification doesn't have one of these components, or the user has disabled any of them, that part is simply ignored.

If you want no action to happen, you can simply pass an empty array to the completion closure. Depending on the logic that pertains to your app, you may want to investigate the `notification.request` property of type `UNNotificationRequest` and make the decision about which components to show based on the notification that was sent to you.

In order for the delegate to be called, you have to tell the notification center that the `AppDelegate` is the actual delegate to use.

Make a couple changes to your `registerForPushNotifications(application:)` back in **ApnsUploads.swift**:

```
func registerForPushNotifications(application: UIApplication) {
    let center = UNUserNotificationCenter.current()
    center.requestAuthorization(options: [.badge, .sound, .alert]) {
        // 1
        [weak self] granted, _ in

        // 2
        guard granted else {
            return
        }

        // 3
        center.delegate = self

        DispatchQueue.main.async {
            application.registerForRemoteNotifications()
        }
    }
}
```

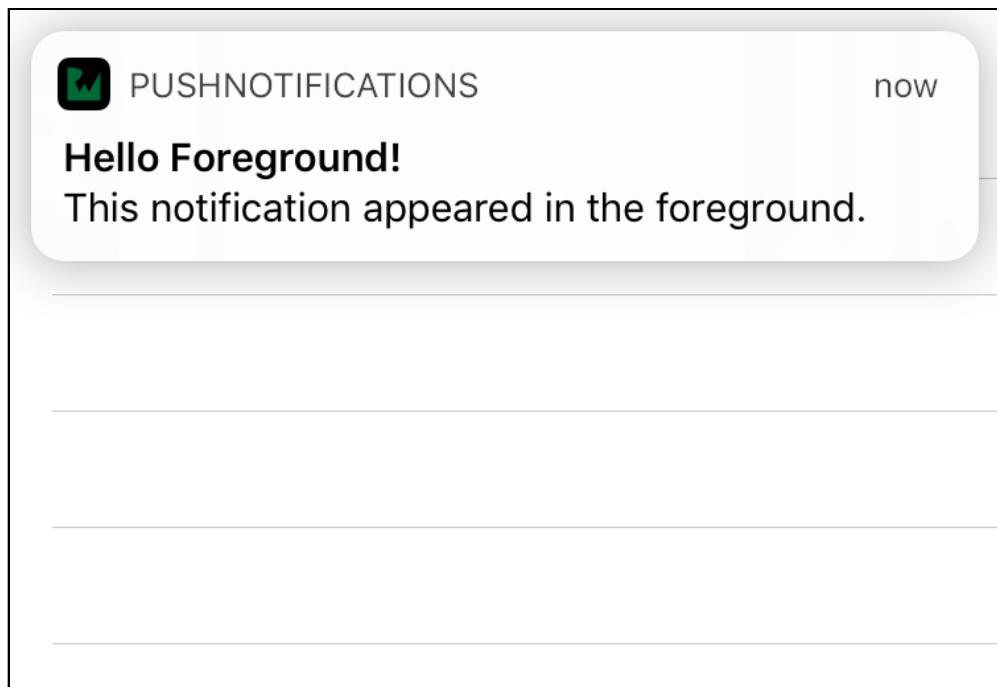
There are three simple changes made:

1. Capture a weak reference to `self` in the completion handler.
2. Then, make sure you have been granted the proper authorization to register for notifications.
3. Finally, you just need to set the `UNUserNotificationCenter`'s delegate to be the `AppDelegate` object.

Build and run your app. Now, use the tester app (as described in Chapter 5, “Apple Push Notifications Servers”) to send a push notification while you’re in the foreground. You should see it displayed this time! You can use the following simple payload for testing purposes:

```
{  
    "aps": {  
        "alert": {  
            "title": "Hello Foreground!",  
            "body": "This notification appeared in the foreground."  
        }  
    }  
}
```

You should get a notification on your device with your app still in the foreground!



Tapping the notification

The vast majority of the time when a push notification arrives, your end users won't do anything except glance at it. Good notifications don't **require** interaction, and your user gets what they need at a glance. However, that's not always the case. Sometimes your users actually tap on the notification, which will trigger your app to be launched.

Go back into your **AppDelegate.swift** file and add the following **UNUserNotificationCenterDelegate** method at the bottom of your extension:

```
func userNotificationCenter(  
    center: UNUserNotificationCenter,  
    didReceive response: UNNotificationResponse,  
    withCompletionHandler completionHandler: @escaping () -> Void) {  
  
    defer { completionHandler() }  
  
    guard response.actionIdentifier ==  
        UNNotificationDefaultActionIdentifier else {  
        return  
    }  
  
    // Perform actions here  
}
```

Notice again that there is a completion handler that **must** be called before exiting the method. This is a great use case for Swift's `defer` keyword as you're ensuring the block of code will be run no matter how you leave the method.

Right now, this method doesn't make much sense as-is. In the next chapter, when you read about custom actions, you'll come back to expand on this. If you don't need to take any custom actions when the user dismisses or taps on your notifications, you can simply omit this method as it's optional in the delegate definition.

Note: There is an `actionIdentifier` called `UNNotificationDismissActionIdentifier`. Don't be fooled into thinking this method will be called if the user simply dismisses the notification!

Handle user interaction

By default, tapping on the notification simply opens up your app to whatever the "current" screen was — or the default startup screen, if the app was launched from a terminated state.

Sometimes, that's not what you want though, as the notification should take you to a specific view controller within your app. This delegate method is exactly where you'll handle that routing.

Since your delegate is growing at this point, you should get it out of the **AppDelegate.swift** file. Obviously, this is a matter of personal style and preference, but keeping a clear separation of duties is always a good idea.

Create a new Swift file called **NotificationDelegate.swift** and then move your delegate methods to that new file. Since `UNUserNotificationCenterDelegate` depends on `NSObjectProtocol`, you'll have to define your class as inheriting from `NSObject`.

```
import UIKit
import UserNotifications

final class NotificationDelegate: NSObject,
    UNUserNotificationCenterDelegate {

    func userNotificationCenter(
        _ center: UNUserNotificationCenter,
        willPresent notification: UNNotification,
        withCompletionHandler completionHandler:
            @escaping (UNNotificationPresentationOptions) -> Void) {
        completionHandler([.alert, .sound, .badge])
    }

    func userNotificationCenter(
        _ center: UNUserNotificationCenter,
        didReceive response: UNNotificationResponse,
        withCompletionHandler completionHandler: @escaping () -> Void) {
        defer { completionHandler() }

        guard
            response.actionIdentifier == UNNotificationDefaultActionIdentifier
        else {
            return
        }

        // Perform actions here
    }
}
```

Back in **AppDelegate.swift**, there are just three quick steps to take:

1. Remove the entire `UNUserNotificationCenterDelegate` extension.
2. Remove the `import UserNotifications` line.
3. Add a delegate property to the `AppDelegate` class: `let notificationDelegate = NotificationDelegate()`.

Then hop over to **ApnsUploads.swift** and change the assignment of the delegate to use your new object:

```
center.delegate = self?.notificationDelegate
```

Do a quick build of your project just to make sure you didn't miss any steps. There should be no warnings or errors at this point.

For the example, if your payload contains a beach key, then you want to land directly on the BeachViewController location of your app. To keep everything simple, the starter project already includes the view controller and a pretty beach image. Normally, your payload would specify the image URL to download and display in the view controller itself.

In **NotificationDelegate.swift**'s

`userNotificationCenter(_:_:didReceive:withCompletionHandler:)`, examine the payload and take them to the right spot if the key exists. Below the `// Perform actions here` comment, add:

```
let payload = response.notification.request.content
guard let _ = payload.userInfo["beach"] else { return }

let storyboard = UIStoryboard(name: "Main", bundle: nil)
let vc = storyboard.instantiateViewController(withIdentifier: "beach")

let appDelegate = UIApplication.shared.delegate as! AppDelegate
appDelegate.window!.rootViewController!.present(vc, animated: false)
```

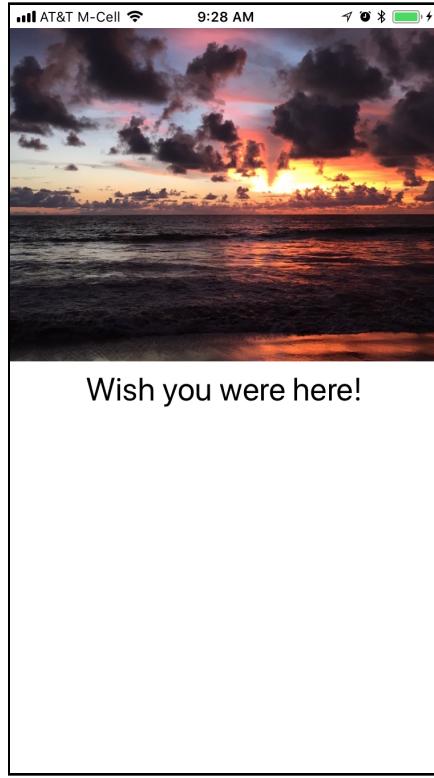
The keys sent with the push notification are inside the `userInfo` property, a simple Swift dictionary. If you find a value with the key "beach", instantiate the BeachViewController from a storyboard and present it on top of the root view controller of the window.

You can see how, in a more dynamic setup, the `userInfo` might contain a URL to an image that you may configure on the view controller itself.

Build and run your app, then send yourself a test push with the following payload:

```
{
  "beach": true,
  "aps": {
    "alert": {
      "body": "Tap me!"
    }
  }
}
```

Once the notification is presented, tap it. If all goes well, you should be presented with the BeachViewController instantiated above:



Silent notifications

Sometimes, when you send a notification, you don't want the user to actually get a visual cue when it comes in. No alert or sound, for example. These are generally referred to as **silent notifications**, but what they really mean is, “Hey app, there’s new content available on the server you might need to do something with”.

If you’ve written an RSS reader app, for example, you might send a silent notification when a new post is submitted so that the app can prefetch the data. This makes the user’s app experience much quicker as the data is there as soon as the app is opened, versus the end user watching an activity indicator while the article is being downloaded.

There are three distinct steps you have to take in order to enable silent notifications:

1. Update the payload.
2. Add the **Background Modes** capability.

3. Implement a new UIApplicationDelegate method.

Updating the payload

The first step to take is simply adding a new key-value pair to your payload. Inside of the `aps` dictionary, add a new key of `content-available` with a value of `1`. This will tell iOS to wake your app when it receives a push notification, so it can prefetch any content related to the notification.

In this case, you're going to have your app prefetch an image. To start, create a payload like so:

```
{  
    "aps": {  
        "content-available": 1  
    },  
    "image": "https://bit.ly/2Iodl06",  
    "text": "A nice picture of the Earth"  
}
```

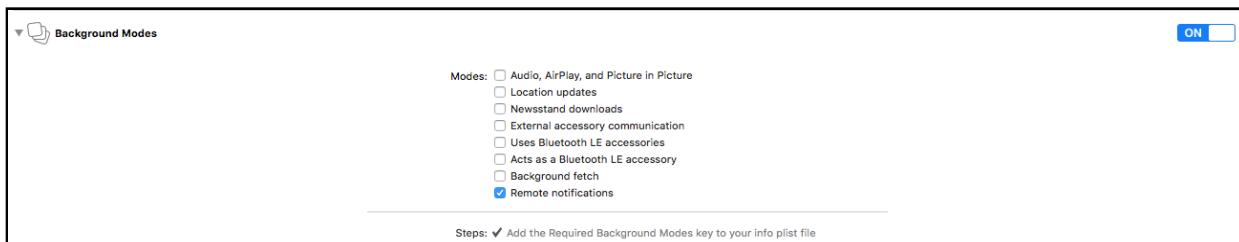
You can use any image URL you'd like. The above is just a known image that should always resolve.

Note: Don't set the value to `0` thinking you've disabled this. If you don't want a silent notification — do not include the `content-available` key!

Adding background modes capability

Next, back in Xcode, you'll need to add a new capability just as you did at project creation.

Open the project navigator (`⌘ + 1`), select your project and then select your app target. Now, on the **Capabilities** tab, toggle on the **Background Modes** option and then choose the **Remote notifications** checkbox from the bottom of the list.



App delegate updates

When a silent notification comes in, you'll want to make sure that it contains the data you're expecting, updates your Core Data model, and then tells iOS you're done processing.

You'll need to implement a new AppDelegate method by adding following code in your UIApplicationDelegate extension in **AppDelegate.swift**:

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

    guard let text = userInfo["text"] as? String,
          let image = userInfo["image"] as? String,
          let url = URL(string: image) else {
        completionHandler(.noData)
        return
    }
}
```

You are expecting both `text` and an `image` as part of the payload, and you need to ensure that the `image` specified is actually something you can turn into a `URL`. If there is any issue, you can just tell iOS that you don't have the needed data by passing `.noData` to your `completionHandler`. You probably don't want to specify `.failed` since technically this just wasn't a payload for an image.

Next, add the following code below the `guard` statement in the method:

```
// 1
let context = persistentContainer.viewContext
context.perform {
    do {
        // 2
        let message = Message(context: context)
        message.image = try Data(contentsOf: url)
        message.received = Date()
        message.text = text

        try context.save()

        // 3
        completionHandler(.newData)
    } catch {
        // 4
        completionHandler(.failed)
    }
}
```

Here's what's going on in the code:

1. Which thread is your notification running on? Not sure? Play it safe and make sure the Core Data operations run on the proper thread of your Core Data persistent container.
2. You create a new Message object and download the image.
3. Since you did, in fact, receive data, you'll want to tell iOS that you got new data from this notification, and that you were able to successfully process the notification.
4. If anything went wrong, you'll tell iOS that processing the notification failed.

Note: iOS will wake up your app in the background and give it up to 30 seconds to complete whatever actions you need to take. Make sure you perform the minimal amount of work necessary so that your action can complete in time.

Build and run the project. Go ahead and send yourself a few more silent push notifications using different images and text, and you should see your table updating appropriately.



Method routing

The following table shows you which methods are called, and in what order, depending on whether your app is in the foreground or background, and whether or not the `content-available` flag (i.e., silent notification) is present with a value of 1.

Foreground	content-available	Action Taken
Yes	No	<code>userNotificationCenter(_:willPresent:withCompletionHandler:)</code>
Yes	Yes	<code>userNotificationCenter(_:willPresent:withCompletionHandler:)</code> <code>application(_:didReceiveRemoteNotification:fetchCompletionHandler:)</code>
No	No	iOS displays the alert as appropriate
No	Yes	<code>application(_:didReceiveRemoteNotification:fetchCompletionHandler:)</code>

Key points

- For iOS to display your notification while your app is running in the foreground, you'll need to implement a `UNUserNotificationCenterDelegate` method, which is called when a notification is delivered to your app while it's in the foreground.
- Good notifications don't require interaction, and your user gets what they need at a glance. Some notifications are tapped, however, which triggers an app launch. You will need to add an additional method in your `AppDelegate.swift` file.
- Sometimes, you want a tapped notification to open a specific view controller within your app. You will need to add an additional method to handle this routing.
- Silent notifications** give no visual or audible cue. To enable silent notifications, you'll need to update the payload, add the Background Modes capability, and implement a new `UIApplicationDelegate` method.

Where to go from here?

In this chapter, you've reviewed a number of common scenarios when dealing with push notifications. In the next chapter, you'll expand more on these scenarios by defining and handling custom actions!



Chapter 9: Custom Actions

At this point, you've implemented as much of push notifications as most app developers will ever want or need to do.

Don't give up now! There are still some really amazing features you can add to your app to make it shine, should you so desire.

In the previous chapter, you built an app that triggers an action when the user taps on a received notification. Sometimes, a simple tap is not enough. Maybe your friend is asking you to grab coffee, and you want an easy way to accept the offer. Or maybe another friend posted a funny tweet and you want to favorite it right from the notification.

Thankfully, iOS gives you a way to attach buttons to a push notification so that the user can provide a meaningful response to the received notification without having to open your app! In this chapter, you'll learn how to make your notifications actionable.

After opening up the starter project for this chapter, remember to turn on the Push Notifications capability as discussed in Chapter 4, "Xcode Project Setup," and set the team signing as discussed in Chapter 7, "Expanding the Application."

Categories

Notification categories allow you to specify up to four custom actions per category that will be displayed with your push notification. Keep in mind that the system will only display the first two actions if your notification appears in a banner, so you always want to configure the most relevant actions first.

To enable the user to decide what action to take, you'll add **Accept** and **Reject** buttons to your push notifications. You'll first add an enum to the top of `AppDelegate.swift`, right below the import statements, to identify your buttons.

```
private let categoryIdentifier = "AcceptOrReject"

private enum ActionIdentifier: String {
    case accept, reject
}
```

Use an enum to ensure you aren't hardcoding strings for identifiers as you won't ever display them to an end user.

Once that's done, just add a method at the bottom of `AppDelegate` to perform the registration.

```
private func registerCustomActions() {
    let accept = UNNotificationAction(identifier: ActionIdentifier.accept.rawValue,
                                       title: "Accept")
    let reject = UNNotificationAction(identifier: ActionIdentifier.reject.rawValue,
                                       title: "Reject")

    let category = UNNotificationCategory(identifier: categoryIdentifier,
                                           actions: [accept, reject],
                                           intentIdentifiers: [])
    UNUserNotificationCenter.current()
        .setNotificationCategories([category])
}
```

Here, you create a notification category with two buttons. When a push notification arrives with a category set to `AcceptOrReject`, your custom actions will be triggered, and iOS will include the two buttons at the bottom of your push notification. While I've simply hardcoded the titles here for brevity, you should always use a localized string, via the `NSLocalizedString` method.

Note: Even if you don't think you're going to localize your app, it's better to get in the habit now than have to go back and find every single user visible string later if plans change!

You only need to register your actions if you're actually accepting push notifications, so add a call to `registerCustomActions()` once you've successfully registered for remote notifications.

```
func application(
    _ application: UIApplication,
    didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data) {
```

```
    sendPushNotificationDetails(to: "http://192.168.1.1:8080/api/token",
                                using: deviceToken)
    registerCustomActions()
}
```

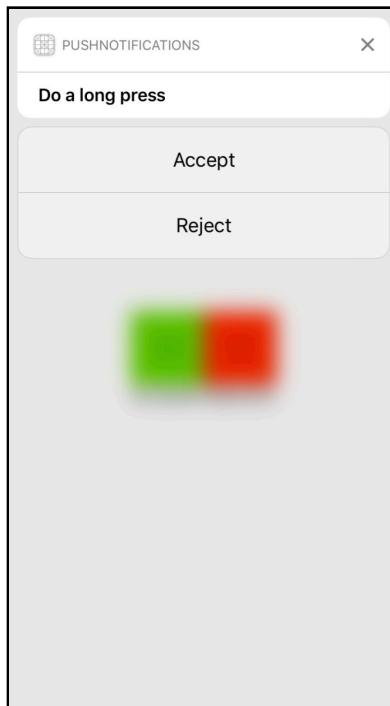
Build and run your app.

Now, go back into the push notification tester app (as described in Chapter 5, “Apple Push Notifications Servers”) and use the following payload:

```
{
  "aps": {
    "alert": {
      "title": "3D Touch this notification (long-press)"
    },
    "category": "AcceptOrReject",
    "sound": "default"
  }
}
```

The critical part of the payload is making sure the `category` value **exactly** matches what you specified during your registration with `UNUserNotificationCenter`.

Send another push to yourself now. See your action buttons? No? Don’t worry, you didn’t mess anything up! The trick is you need to 3D Touch the notification to reveal the buttons. Once you do that, the custom buttons appear and you can select one.



Go back into your **AppDelegate.swift** and add the `userNotificationCenter(_:_:didReceive:withCompletionHandler:)` delegate method to your `UNUserNotificationCenterDelegate` extension:

```
func userNotificationCenter(  
    _ center: UNUserNotificationCenter,  
    didReceive response: UNNotificationResponse,  
    withCompletionHandler completionHandler: @escaping () -> Void) {  
  
    defer { completionHandler() }  
  
    let identity = response.notification.request.content.categoryIdentifier  
    guard identity == categoryIdentifier,  
        let action = ActionIdentifier(rawValue: response.actionIdentifier)  
    else {  
        return  
    }  
  
    print("You pressed \(response.actionIdentifier)")  
}
```

Remember that this method will be called when you tap on the notification, so you need to do a quick check to make sure you're handling your own category, and then you can grab the button that was pressed. This method will be called even when your app is not in the foreground, so be careful of what you do here!

Build and run your code again, and send yourself another notification from the Tester app. 3D Touch the notification and select one of the buttons; you should see a similar message in Xcode's console:

```
You pressed accept
```

Extending Foundation's notification

The `AppDelegate` is definitely not where you want to take action when a push notification arrives. A better idea is to send a Foundation notification that lets you know what happened, providing your other view controllers with the opportunity to take appropriate action. The SDK makes Foundation's notifications a bit awkward to work with, but you're going to fix that with a simple protocol extension.

Create a new Swift file called **Notification.swift** and replace its contents with the following:

```
import Foundation  
  
extension Notification.Name {  
    // 1
```

```
static let acceptButton = Notification.Name("acceptTapped")
static let rejectButton = Notification.Name("rejectTapped")

// 2
func post(center: NotificationCenter = NotificationCenter.default,
          object: Any? = nil,
          userInfo: [AnyHashable : Any]? = nil) {
    center.post(name: self, object: object, userInfo: userInfo)
}

// 3
@discardableResult
func onPost(center: NotificationCenter = NotificationCenter.default,
           object: Any? = nil,
           queue: OperationQueue? = nil,
           using: @escaping (Notification) -> Void
           -> NSObjectProtocol {
    return center.addObserver(forName: self, object: object,
                               queue: queue, using: using)
}
}
```

There's not a lot to the code, but it provides a ton of functionality!

1. First, you're defining two custom `Notification.Name` static properties to represent each of your custom actions. You'll post the appropriate one based on what happened. The string you use for the name doesn't matter; it just has to be unique.
2. Any `Notification.Name`, like the two that you just defined, now has a `post` method with the most commonly used defaults set for its arguments.
3. You've also provided a simple way to take an action when a post occurs. Again, defaults allow for the normal operation without boilerplate code.

Back in your `AppDelegate`'s

`userNotificationCenter(_:_:didReceive:withCompletionHandler:)` you can now replace your `print` statement with the following:

```
switch action {
case .accept:
    Notification.Name.acceptButton.post()
case .reject:
    Notification.Name.rejectButton.post()
}
```

By adding that simple protocol extension to `Notification.Name`, you have made the rest of your code much easier to understand.

If your payload contains custom keys-values that you also need to pass along, you can simply change the `post` method to accept the data from the push notification, like so:

```
let userInfo = response.notification.request.content.userInfo

switch action {
case .accept:
    Notification.Name.acceptButton.post(userInfo: userInfo)
case .reject:
    Notification.Name.rejectButton.post(userInfo: userInfo)
}
```

Responding to the action

Now that you've properly detected that your end user has selected a custom action, and you've posted the notification, you need to actually *do* something. Return to **ViewController.swift**. You'll see a little bit of existing code that simply keeps the label display value in sync with the counters.

Override `viewDidLoad` with the code below by simply adding it below your outlets. This code responds to the Foundation notifications you post when the user interacts with your notification, and updates the appropriate counter based on the user's selection:

```
override func viewDidLoad() {
    super.viewDidLoad()

    Notification.Name.acceptButton.onPost { [weak self] _ in
        self?.numAccepted += 1
    }

    Notification.Name.rejectButton.onPost { [weak self] _ in
        self?.numRejected += 1
    }
}
```

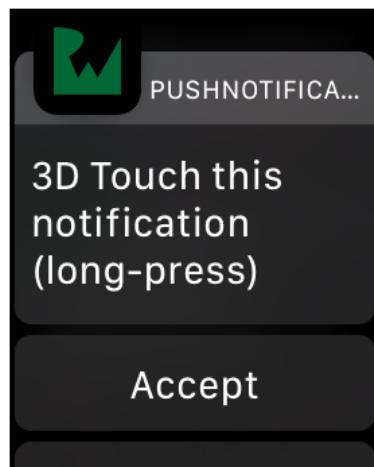
Creating your quick extension on `Notification.Name` has definitely paid off, hasn't it? If the user presses the **Accept** button, you'll increment the `numAccepted` counter; if they press the **Reject** button, you'll increment the `numRejected` counter instead.

Build and run the app one final time. Send yourself a bunch of notifications, alternating between which button you tap each time. The display should keep a count for you!



By using Foundation's notification service, you've kept the logic of your app encapsulated into the right areas, and you haven't gone through convolutions to let the `UNUserNotificationCenterDelegate` methods know anything about the view controllers, which might or might not want to take action based on a notification.

Making notifications actionable is a good idea whenever it makes sense, as it streamlines the experience for the user and makes their life a little bit easier. Another added benefit is the fact these actions are also automatically displayed for users of the Apple Watch!



Key points

- You can make a push notification actionable by attaching a button to a notification.
- Notification categories allow you to specify up to four custom actions per category that will be displayed with your push notification.
- Since you do not want the `AppDelegate` to be where you take action when a push notification arrives, send a Foundation notification that reports what happened instead. That allows your other view controllers to take an appropriate action.

Chapter 10: Modifying the Payload

Sometimes, you'll need to take extra steps before a notification is presented to the user. For example, you may wish to download an image or change the text of a notification.

In the **DidIWin** lottery app, for example, you'd want the notification to tell the user exactly how much money he or she has won. Given the push notification simply contains today's drawing numbers, you'll be using a **Notification Service Extension** to intercept those numbers and apply logic to them.

You can think of a Notification Service Extension as middleware between APNs and your UI. With it, you can receive a remote notification and modify its content before it's presented to the user. Considering the fact notification payloads are limited in size, this can be a very useful trick! Another common use case for modifying the payload is if you're sending encrypted data to your app. The service extension is where you'd decrypt the data so that it's properly displayed to your end user.

In this chapter, you'll go over what it takes to build a Notification Service app extension and how to implement some of its most common use cases.

Configuring Xcode for a service extension

Due to your proven track record of writing amazing apps, your country's spy agency has contracted you to write the app that its field agents will use to receive updates from headquarters. Of course, the agency sends all of its data using massive encryption, so you'll need to handle the decryption for the agents. Nobody wants to read a gobbledegook text!

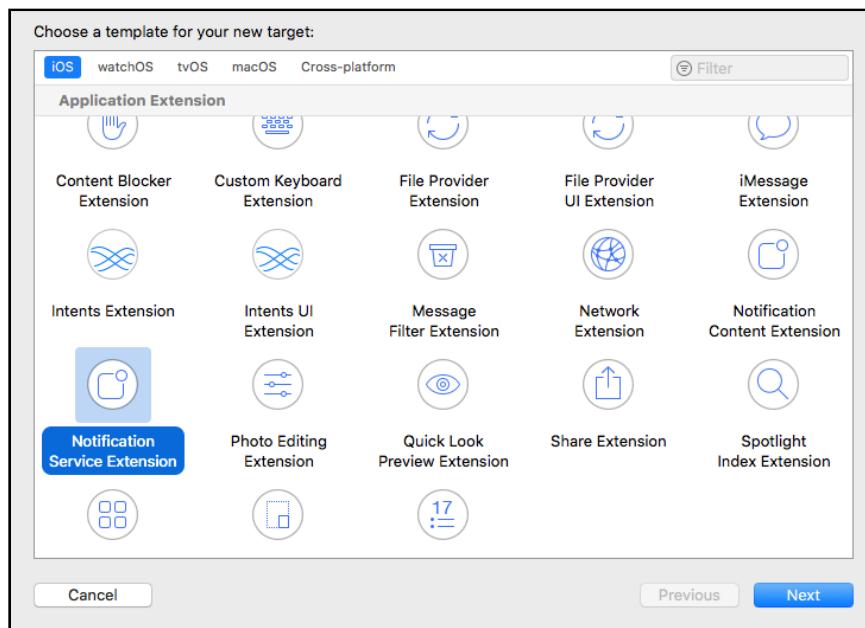
Open the starter project for this chapter. Remember to turn on the Push Notifications capability as discussed in Chapter 4, “Xcode Project Setup,” and set the team signing as discussed in Chapter 7, “Expanding the Application.”

Edit **AppDelegate.swift** to contain your IP address in the call to `sendPushNotificationDetails(to:using:)`.

In order to find out your IP address, go into **System Preferences** ▶ **Network** ▶ **Advanced** ▶ **TCP/IP** and copy the value under **IPv4 Address**. Paste this value between `http://` and `:8080` in the code, like so: `http://YOUR_IP_HERE:8080/api/token`.

Now, you need to add your extension target so that you can handle the encryption being used.

1. In Xcode, select **File** ▶ **New** ▶ **Target....**
2. Make sure **iOS** is selected and choose the **Notification Service Extension**.
3. For the product name specify **Payload Modification**.
4. Press **Finish**.
5. If asked about scheme activation, select **Cancel**.



Note: You don't actually *run* a service extension so that's why you didn't let it make the new target your active scheme.

You can name the new target anything that makes sense for you, but it can be helpful to use the above name because, when you glance at your project, you will immediately know what that target is doing.

If you look in the Project navigator ($\mathbf{\#} + 1$), you'll see you now have a new folder group called **Payload Modification**. You'll notice that there's a **NotificationService.swift** file but no storyboard. This is because service extensions don't present any type of UI. They are called *before* the UI is presented, be it yours or the one Apple displays for you. You'll get into UI modifications in the next chapter.

Edit the **NotificationService.swift** file and you'll see a bit of content already provided for you by Apple. The first method in this file, `didReceive(_:withContentHandler:)` is called when your notification arrives. You have *roughly* 30 seconds to perform whatever actions you need to take. If you run out of time, iOS will call the second method, `serviceExtensionTimeWillExpire` to give you one last chance to hurry up and finish.

If you're using a restartable network connection, the second method might give you just enough time to finish. Don't try to perform the same actions again in the `serviceExtensionTimeWillExpire` method though. The intent of this method is that you perform a much smaller change that can happen quickly. You may have a slow network connection, for example, so there's no point in trying yet another network download. Instead, it might be a good idea to tell the user that they got a new image or a new video, even if you didn't get a chance to download it.

Note: If you haven't called the completion handler before time runs out, iOS will continue on with the original payload.

You may make any modification to the payload you want — except for one. You may not remove the alert text. If you don't have alert text, then iOS will ignore your modifications and proceed with the original payload.

Decrypting the payload

As mentioned at the start of the chapter, the payload you receive has encrypted the data. Your country is a little bit behind the times though, and it is still using the **ROT13** letter substitution cipher in which each letter is simply replaced by the letter 13 places further along in the alphabet, wrapping back to the beginning of the alphabet if necessary.

Implementing the ROT13 cipher

In your **Payload Modification** target create a new Swift file named **ROT13.swift** and paste this code into it:

```
import Foundation

struct ROT13 {
    static let shared = ROT13()

    private let upper = Array("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
    private let lower = Array("abcdefghijklmnopqrstuvwxyz")
    private var mapped: [Character: Character] = [:]

    private init() {
        for i in 0 ..< 26 {
            let idx = (i + 13) % 26
            mapped[upper[i]] = upper[idx]
            mapped[lower[i]] = lower[idx]
        }
    }

    public func decrypt(_ str: String) -> String {
        return String(str.map { mapped[$0] ?? $0 })
    }
}
```

You can find many different ways of implementing this cipher in Swift. The above is just a quick and dirty way to handle the American English alphabet.

Obviously, the above code makes a nice sample for a book as it doesn't require downloads and configuration. However, for real security, you should look to something like the CryptoSwift (<https://cryptoswift.io/>) library.

Modifying the payload

Run your app on a device, taking note of the device token that gets printed to the console window. In the starter materials for this project, you'll find a **sendEncrypted.php** file. Edit this file with your favorite text editor and specify your token and other details at the top of the file. When done, run it from **Terminal**:

```
$ php sendEncrypted.php
```

Note: If you haven't already set up the **sendEncrypted.php** script, you can find instructions on how to set it up in Chapter 6, "Server Side Pushes."

If everything goes correctly, you should see a notification on your device. However, this notification is encrypted by the agency, and you need to decrypt the contents before displaying the notification on the device.



Now, back in your **NotificationService.swift** file, find the lines in `didReceive` that show an example modification:

```
// Modify the notification content here...
bestAttemptContent.title = "\(bestAttemptContent.title) [modified]"
```

Replace them with code to decrypt the data:

```
bestAttemptContent.title = ROT13.shared.decrypt(bestAttemptContent.title)
bestAttemptContent.body = ROT13.shared.decrypt(bestAttemptContent.body)
```

Build and run your app again, and then go back to **Terminal** and run the PHP script again:

```
$ php sendEncrypted.php
```

If everything worked correctly, you should see a decrypted push notification appear on your phone.



Downloading a video

Service extensions are also the place in which you can download videos or other content from the internet. First, you need to find the URL of the attached media. Once you have that, you can try to download it into a temporary directory somewhere on the user's device. Once you have the data, you can create a `UNNotificationAttachment` object, which you can attach to the actual notification.

Go back to `NotificationService.swift` and, right after you decrypt the body of the message, before the call to the `contentHandler` closure, add the following code to download any video they might have sent along:

```
// 1
if let urlPath = request.content.userInfo["media-url"] as? String,
    let url = URL(string: ROT13.shared.decrypt(urlPath)) {

// 2
let destination = URL(fileURLWithPath: NSTemporaryDirectory())
.appendingPathComponent(url.lastPathComponent)

do {
// 3
let data = try Data(contentsOf: url)
```

```
try data.write(to: destination)

// 4
let attachment = try UNNotificationAttachment(identifier: "",  
                                              url: destination)

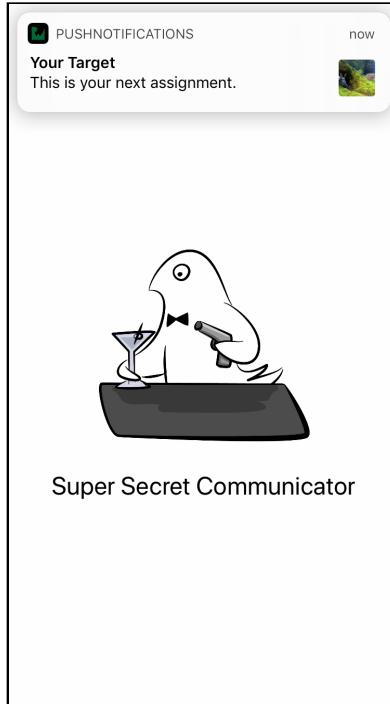
// 5
bestAttemptContent.attachments = [attachment]
} catch {
    // 6
}
}
```

Here's what the above code is doing:

1. You first have to make sure that not only did they send along a **media-url** key but that you can turn it into a valid URL. Don't forget to also decrypt the URL! You don't want those foreign operatives knowing your URLs!
2. You'll also make a local file URL where you'll write the data to. Ensure that your filename stays the same so that iOS knows what type of file you're working with. If you save to a file with a random extension, your download isn't going to work the way you are expecting.
3. Use the `Data(contentsOf:)` method to perform a **synchronous** data download. You can't use an asynchronous method or the function will exit before your data has been retrieved.
4. Once you've written the data to disk, you just create a `UNNotificationAttachment`. iOS will generate a unique identifier for you if you leave it empty.
5. Finally, add the attachment to the content of the push notification.
6. There's not really anything you can do if the download fails, so you'll just be leaving the error case empty.

Build and run your app again, and then rerun the `sendEncrypted.php` script.

You should get a push notification that has a small image on the right-hand side. 3D Touch the notification and you'll see a video with your next target!



Service extension payloads

You don't necessarily always want an extension to run every time you receive a push notification — just when it needs to be modified. In the above example, you'd obviously use it 100% of the time as you're decrypting data. But what if you were just downloading a video? You don't **always** send videos.

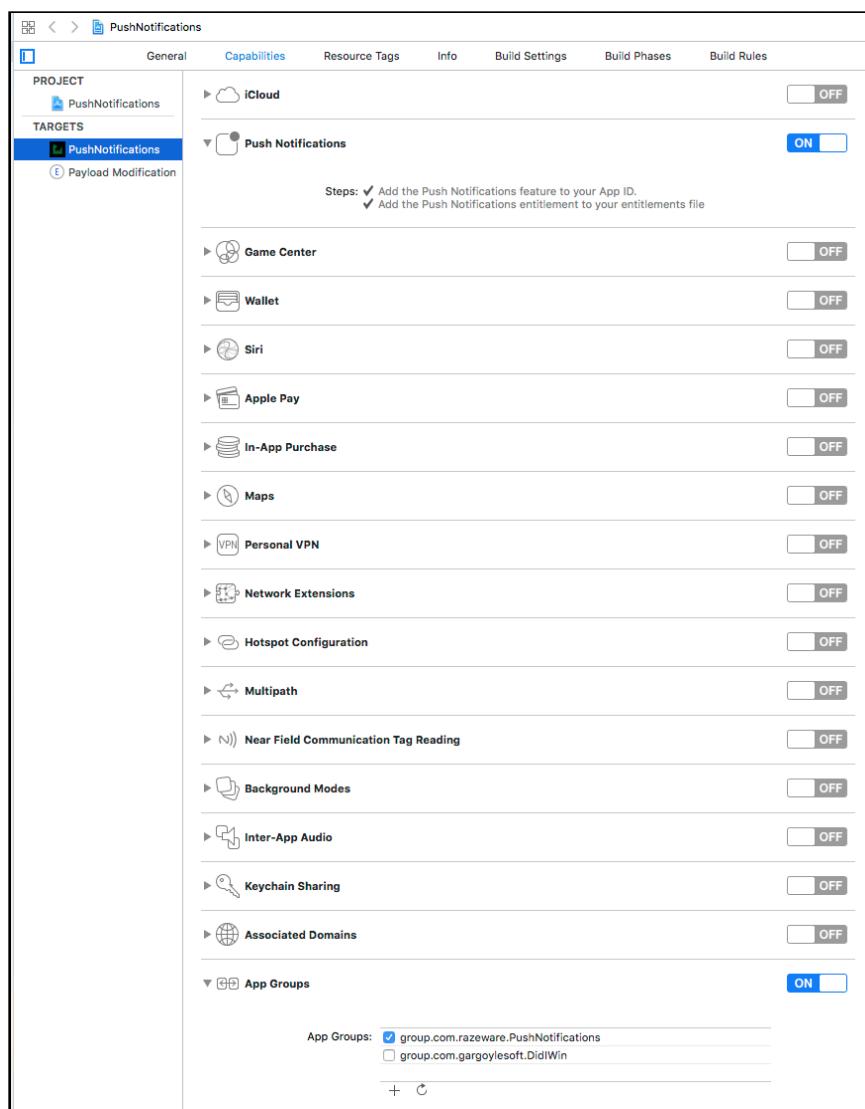
To tell iOS that the service extension should be used, simply add a `mutable-content` key to the `aps` dictionary with an integer value of 1. You'll notice that the provided `sendEncrypted.php` already includes this key for you.

Note: If you forget to add this key, your service extension will never be called. You're most likely going to forget to do this and have a heck of a time figuring out why your code doesn't work!

Sharing data with your main target

Your primary app target and your extension are two separate processes. They can't share data between them by default. If you do more than the most simplistic of things with your extension, you'll quickly find yourself wanting to be able to pass data back and forth. This is easily accomplished via **Application Groups**, which allows access to group containers that are shared between multiple related apps and extensions.

To enable this capability, press **⌘ + 1** to go back to the **Project navigator** and click on your main target. Next, navigate to the **Capabilities** tab again. Near the top of the list, you'll see a section for **App Groups**. Turn that on, press the **+** button and then set the name you wish to use. Generally, you'll want the same name as your bundle identifier, just prefixed with **group**:



You'll notice, in this image, that an App Group has already been created for another project, so that's also shown in the image. Be sure that you only select the one group you want if there are multiple listed.

Now, go into your **Payload Modification** target's capabilities tab and enable the App Groups there as well, selecting the same app group you selected for your app target.

Badging the app icon

A great use for service extensions is to handle the app badge. As discussed in Chapter 3, “Remote Notification Payload,” iOS will set the badge to *exactly* what you specify in the payload, if you provide a number. What happens if the end user has ignored your notifications so far? Maybe you’ve sent them three new items at this point. You’d rather the badge said 3 and not 1, right?

Historically, app developers have sent information back to the server as to how many badges the app icon is currently displaying, and then the push notification would increment that number by one. While that’s doable, it’s quite a bit of extra overhead to deal with on your server. By utilizing a service extension, you can now just pretend that the badge key being there means to increment the badge count by that number. You’re now just storing locally how many items are unread versus having to send those details back to your server for tracking.

As this is just an integer value, you can make use of the `UserDefaults` class with one small change — assuming you’ve already enabled App Groups. You have to specify the suite that is used to enable it to span targets. To do so, add a new Swift file to your **primary target**, not the extension, called **UserDefaults.swift**:

```
import Foundation

extension UserDefaults {
    // 1
    static let suiteName = "group.com.razeware.PushNotifications"
    static let extensions = UserDefaults(suiteName: suiteName)!

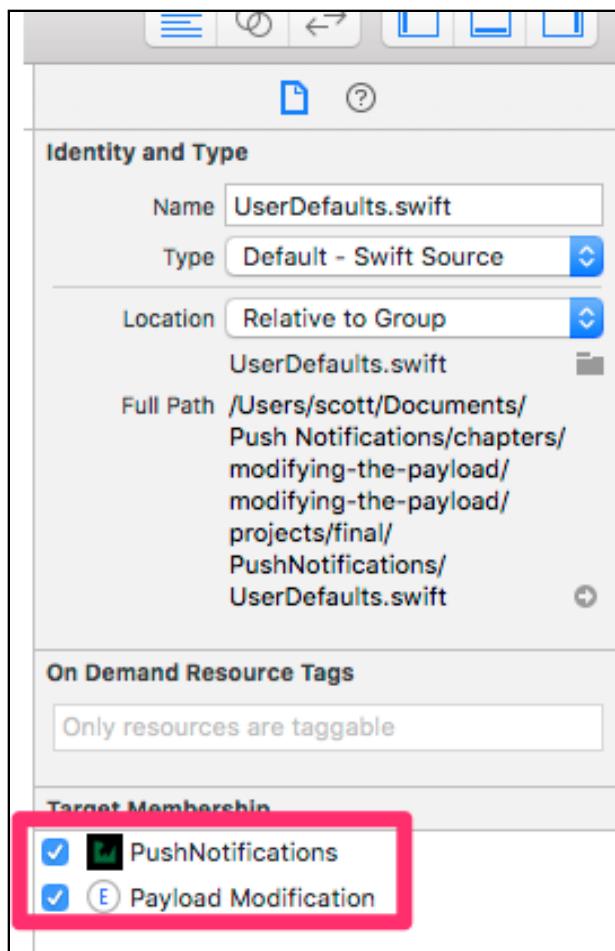
    // 2
    private enum Keys {
        static let badge = "badge"
    }

    // 3
    var badge: Int {
        get {
            return UserDefaults.extensions.integer(forKey: Keys.badge)
        }
    }
}
```

```
    set {
        UserDefaults.extensions.set(newValue, forKey: Keys.badge)
    }
}
```

1. First, you define a new extensions property, providing a UserDefaults object you'd use when you want to share your defaults between targets.
2. Hardcoding strings is a bad idea, so you create an enum with a static let so that you only have to do it once. A struct would work here just as well. The reason you want to use an enum is that you can't accidentally instantiate it.
3. Finally, you wrap up by creating a computed property for badge that handles the get/set. Again, this is just good coding style to make life easier on the caller.

Right now, this file is only accessible from the main target though. Bring up the **File inspector** by pressing ⌘ + ⌥ + 1 and in the **Target Membership** section check the boxes next to your service extension as well as the primary target:



Now, back in **NotificationService.swift**, edit the `didReceive(_:withContentHandler:)` method. You can check for badging information by placing the following code just after the section where you download the video, right before the call to `contentHandler`:

```
if let incr = bestAttemptContent.badge as? Int {
    switch incr {
        case 0:
            UserDefaults.extensions.badge = 0
            bestAttemptContent.badge = 0
        default:
            let current = UserDefaults.extensions.badge
            let new = current + incr

            UserDefaults.extensions.badge = new
            bestAttemptContent.badge = NSNumber(value: new)
    }
}
```

It's important to store the value to a `UserDefault`s type structure so you modify that value in your primary target as well. When your user accesses the part of your app that the badge refers to, you'll want to decrement the badge count so that the app icon is updated.

Build and run the app. From **Terminal**, run the `sendEncrypted.php` script a few times. The badge number should increase for each notification you receive:



Accessing Core Data

Writing to a `UserDefault`s key can be incredibly useful, but isn't normally good enough. Sometimes, you really just need access to your actual app's data store in your extension. Most commonly, you'll look for a way to access Core Data. It's easy enough to do once you've enabled App Groups.

First, select your data model (**PushNotifications.xcdatamodeld**). Then, in the **Target Membership** section of the **File inspector**, add a checkmark next to your service notification target. If you created any `NSManagedObject` subclasses that you need to use, do the same thing with them.

Second, edit your **AppDelegate.swift** file, making a small change to the `persistentContainer` lazy variable. You've got to tell the container exactly where to store the data. So modify the default like so:

```
lazy private var persistentContainer: NSPersistentContainer = {
    // 1
    let groupName = "group.YOUR_BUNDLE_ID"
    let url = FileManager.default
        .containerURL(forSecurityApplicationGroupIdentifier: groupName)!
        .appendingPathComponent("PushNotifications.sqlite")

    // 2
    let container = NSPersistentContainer(name: "PushNotifications")

    // 3
    container.persistentStoreDescriptions = [
        NSPersistentStoreDescription(url: url)
    ]

    // 4
    container.loadPersistentStores(completionHandler: { _, error in
        if let error = error as NSError? {
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })

    return container
}()
```

Using app groups with Core Data really only requires two changes to the default setup.

1. The first difference is that you have to tell iOS exactly where to write the internal **.sqlite** file since the default doesn't work with app groups. Be sure you use the exact name that you gave the App Group!
2. Creating the container is no different than the default setup.
3. However, the container has to know that it's using your custom location.
4. Then, just load the store like normal and return the container.

Now, your main app knows exactly where to write the Core Data database to, but your extension still doesn't have this information.

Copy the lazily computed property code you just used exactly as is and then paste it into **Payload Modification/NotificationService.swift** file, inside of the **NotificationService** class. Remember to add an `import CoreData` statement or you'll get a build error. Now, you can access your data model in the extension

Localization

If you're modifying the content of your payload, you might be modifying the text as well. Always keep in mind that not everyone speaks the same language you do, so you still need to follow all the localization rules you normally would.

Note: There's currently a bug in Xcode in which your base language will not always be used in an extension. To work around this bug, simply make sure that you have a **Localizable.strings** for your base language defined.

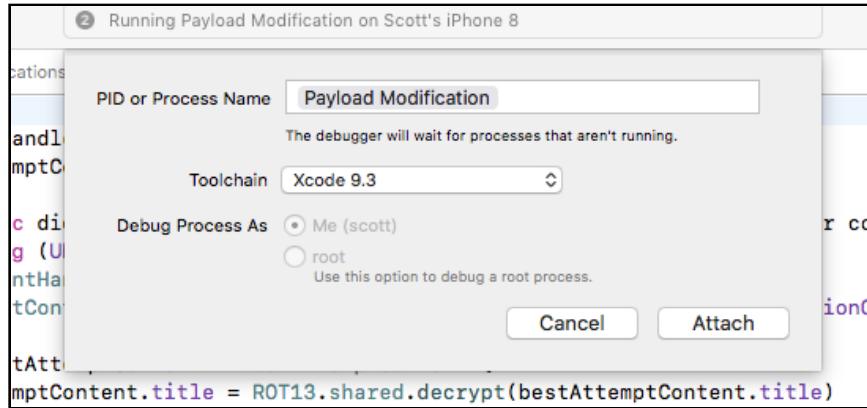
If the only reason you're using an extension is to perform localizations on text, you should instead look at the keys of the `aps alert` dictionary, as explained back in Chapter 3, "Remote Notification Payload," as there are multiple items there to perform this action for you.

Debugging

Sometimes, no matter how hard you try, things just don't go right. Debugging a service extension works almost the same as any other Xcode project. However, because it's a target and not an app, you have to take a few extra steps.

1. Open up your **NotificationService.swift** file and set a breakpoint on the line where you decode the title.
2. Build and run your app.
3. In Xcode's menu bar, choose **Debug > Attach to Process by PID or Name....**
4. In the dialog window that appears, enter **Payload Modification** — or whatever you named your target.

5. Press the **Attach** button.



If you send yourself another push notification, Xcode should stop execution at the breakpoint you set. Be aware that debugging service extensions is a bit finicky and sometimes it just plain doesn't work. If you aren't able to find your process listed, you might have to go through a full restart of Xcode and possibly even a reboot of your device.

Key points

- A Notification Service Extension is a sort of middleware between APNs and your UI. With it, you can receive a remote notification and modify its content before it's presented to the user.
- You may make any modification to the payload you want — except for one. You may not remove the alert text. If you don't have alert text, then iOS will ignore your modifications and proceed with the original payload.
- You can use service extensions so that you can download videos or other content from the internet; you will create a `UNNotificationAttachment` object that you attach to the push notification.
- Your primary app target and your extension are two separate processes and cannot share data between them by default. You can overcome this using Application Groups.
- Service extensions can be used to handle your app's badge so that the badge reflects the number of unseen notifications without having to involve server side storage.

- You can access your app’s data store in your extension once you have Application Groups set up.
- When modifying the content of your payload, if your text is also changed, follow localization rules to account for different languages.

Where to go from here?

You covered quite a bit of material in this chapter. You can now download data in response to a push notification, modify the incoming payload and easily handle app badging. You also learned how to attach a debugger to a running extension process to debug it.

Everything, so far, has been behind-the-scenes work that’s not necessarily visible to the end user of your app. In the next chapter, you’ll tackle the task of modifying the UI that’s displayed to your users.

Chapter 11: Custom Interfaces

In the last few chapters, you worked through most types of notifications, including those that present an attachment, such as an image or video, alongside the banner message; but if you really want to go hog wild, you can even customize the way the notification itself looks to your heart's content! This can get quite complex, but it is worth the time to make an app that really shines. Custom interfaces are implemented as separate targets in your Xcode project, just like the service extension.

Your top-secret agency wants to send you the locations of your targets, so you'll need to build a way to do that. In this chapter, you'll create a notification that displays a location on the map, with the ability to comment on that location right from the notification, all without opening the app.

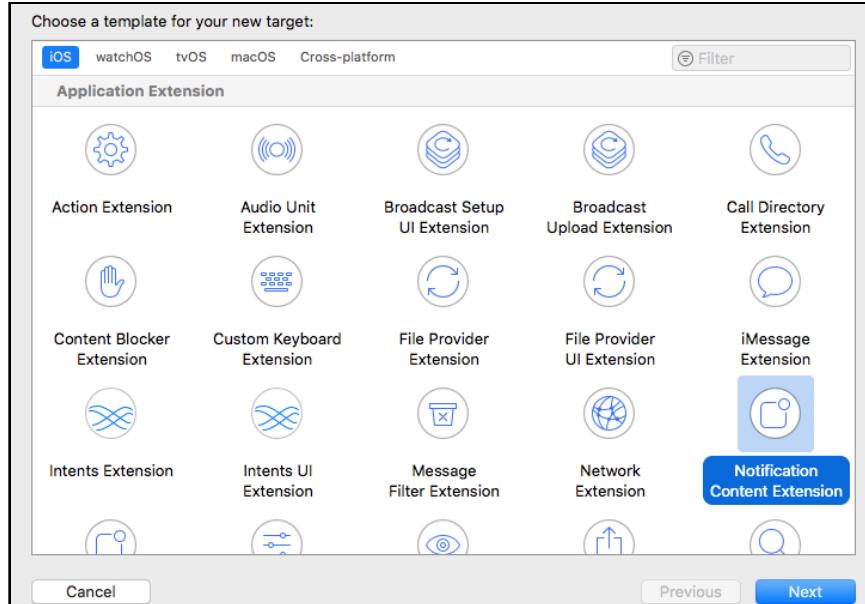
Configuring Xcode for custom UI

After opening up the starter project for this chapter, remember to turn on the Push Notifications capability as discussed in Chapter 4, “Xcode Project Set Up,” and set the team signing as discussed in Chapter 7, “Expanding the Application.” Don’t forget to also set the team signing for the Payload Modification target just as you did in the previous chapter, Chapter 10, “Modifying the Payload.”

First, you’ll create a new **Notification Content Extension** that will handle showing your custom UI.

1. In Xcode, select **File** ▶ **New** ▶ **Target....**
2. Makes sure **iOS** is selected and choose the **Notification Content Extension**.
3. Press **Next**.

4. For the **Product Name** field type **Custom UI**.
5. Press **Finish**.
6. If asked about scheme activation, select **Cancel**.



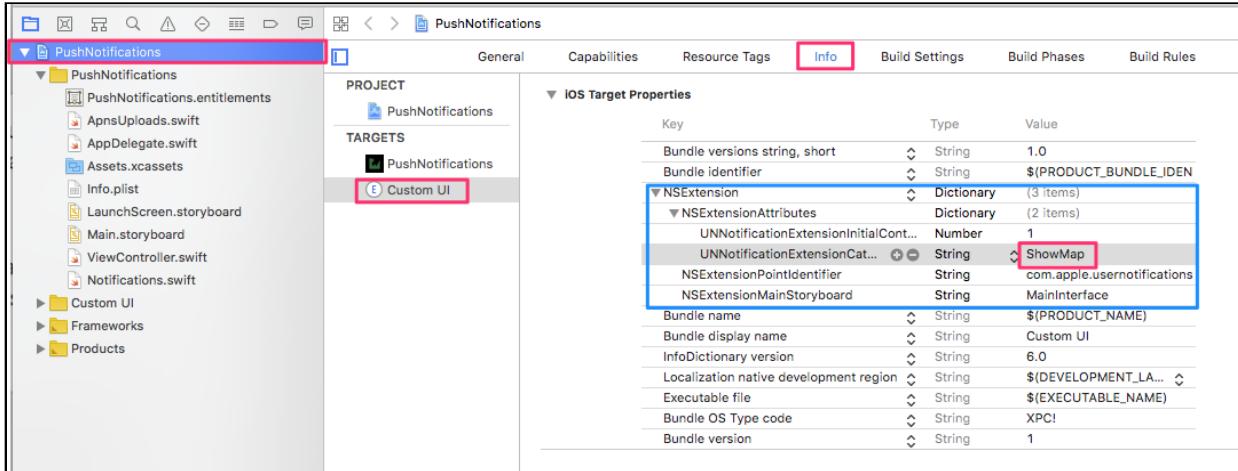
Note: You don't actually *run* a **Notification Content Extension**, so that's why you didn't let it make the new target your active scheme.

You can name the new target anything that makes sense for you, but it can be helpful to use the above name because, when you glance at your project, you will immediately know what that target is doing.

Custom interfaces are triggered by specifying a category, just as you learned about with custom actions in Chapter 9, “Custom Actions.” The sample project has already registered a category for you in `AppDelegate.swift`’s `registerCustomActions()`, with a category identifier of **ShowMap**.

Every custom UI must have its own unique category identifier. Bring up the Project navigator ($\mathbf{\#} + 1$) and select your project. Then, select the newly created target and go to the **Info** tab. You’ll see an item labeled **NSExtension**. Expand that *all the way* out and find a key labeled **UNNotificationExtensionCategory**. This identifier connects your main target, registering the identifier, with the correct content extension.

If your push notification contains a category key that matches this, the UI in your content extension will be used. Update this value to ShowMap.



If you have multiple category types that will all use the same UI, simply change the type of `UNNotificationExtensionCategory` from `String` to `Array` and list each category name that you'd like to support.

Designing the interface

You'll notice that your new target includes a storyboard and view controller for you to utilize. You're going to present your users with a map of the coordinates that you send them via a push notification.

Open up the **MainInterface.storyboard** from your UI target (not the **Main.storyboard** from your app itself) and make the following changes:

1. Select the **View** and in the **Size inspector** change the view's height to be 320.
2. Remove the **Label**.
3. Drag an **MKMapView** onto the view.
4. Constrain it to all four edges of the superview with a constant of 0.
5. In **NotificationViewController.swift**, add to the top of the file:

```
import MapKit
```

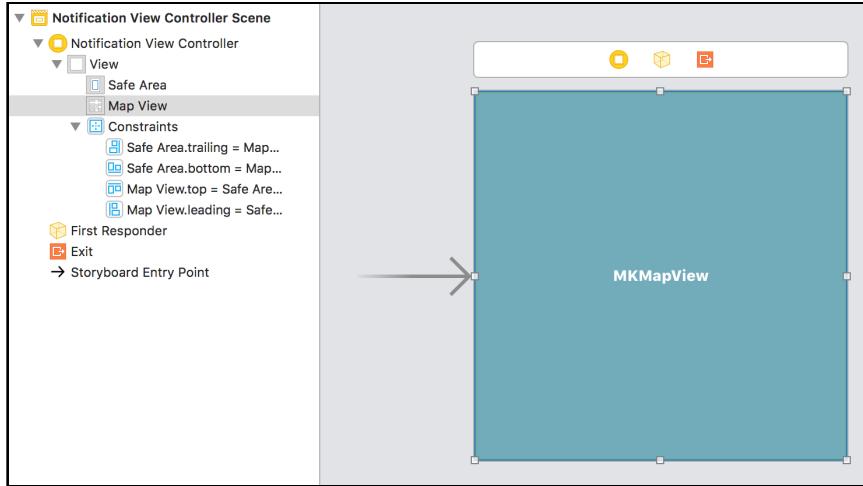
And then, replace:

```
@IBOutlet var label: UILabel?
```

With:

```
@IBOutlet var mapView: MKMapView!
```

6. Back in **MainInterface.storyboard**, connect your **MKMapView** outlet.



That's all you have to do in your storyboard. Now, open up **NotificationViewController.swift** and replace the `didReceive(_:)` method with:

```
func didReceive(_ notification: UNNotification) {
    let userInfo = notification.request.content.userInfo

    guard let latitude = userInfo["latitude"] as? CLLocationDistance,
          let longitude = userInfo["longitude"] as? CLLocationDistance,
          let radius = userInfo["radius"] as? CLLocationDistance else {
        return
    }

    let location = CLLocation(latitude: latitude, longitude: longitude)
    let region = MKCoordinateRegion(center: location.coordinate,
                                    latitudinalMeters: radius,
                                    longitudinalMeters: radius)

    mapView.setRegion(region, animated: false)
}
```

Your view controller has access to the full payload that was sent over by accessing the `userInfo` property of the `UNNotification` instance. You're simply pulling the `latitude`, `longitude` and `radius` from your payload, constructing the appropriate `CoreLocation` objects, and end up by telling the map to display that region.

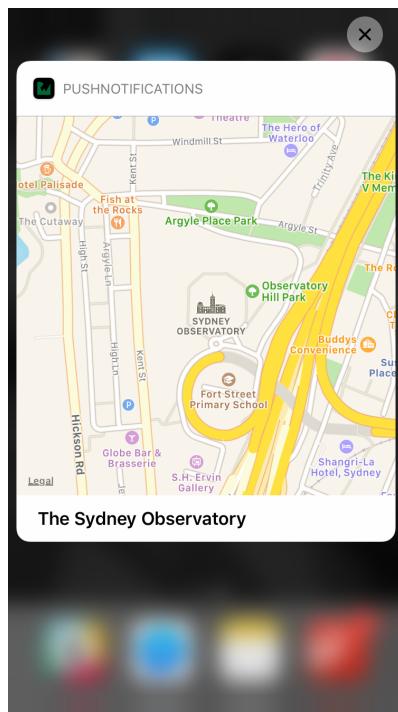
Build and run your app so that you can test everything. There shouldn't be any warnings or errors from the build.

If you haven't set up your **PushNotifications** tester app, do so now as described in Chapter 5, "Apple Push Notification Servers."

Make sure you change your payload to the following JSON:

```
{  
    "aps": {  
        "alert" : {  
            "title" : "The Sydney Observatory"  
        },  
        "category" : "ShowMap",  
        "sound": "default"  
    },  
    "latitude" : -33.859574,  
    "longitude" : 151.204576,  
    "radius" : 500  
}
```

Now, send the push notification. You should see a notification come in and, by using 3D Touch on it, you should see the location on a map right inside the notification!



You'll quickly notice, if you try to pan or zoom the map, the custom UI view controller, while fully functional, does not accept any type of user input. Keep this in mind while designing your interface. In a map example, it probably doesn't make sense to place any pins on the view as the end user won't be able to touch them to get more information, which could lead to confusion.

Keep in mind that your custom interface is still just an iOS target. This means that you can easily share properly encapsulated `UIViews` between your main target and the content extension. Just add the `UIView` to the content extension target in the **File Inspector** (`⌘ + ⌘ + 1`), and you can use it like any other view! You can refer back to Chapter 10, “Modifying the Payload,” in which you added the `UserDefaults.swift` file to the service extension, if you need a reminder of how this works.

Resizing the initial view

If you watch really closely while your custom UI comes into place, you’ll probably notice that it might start a bit too big and then shrink down to the proper size.

Apple, without explaining why, implemented the initial height of the view as a percentage of the width, instead of letting you specify a specific size. In the `Info.plist` of your target extension, you can expand the **NSExtension** row again, where you’ll see a setting for `UNNotificationExtensionInitialContentSizeRatio`, which defaults to 1. You should set this to a decimal value less than or equal to 1, representing the ratio of the height to the width. If you specify `0.8`, for example, the UI will start with a height that is 80% as tall as the width.

Trial and error are your friend in getting this just right.

Accepting text input

At times, you may want to allow your users to type some text in response to a push notification. With the previous map push, people may want to tell you how jealous they are that you’re there or the awesome things they saw last time they went themselves. Or, in your spy app, you might want to request additional information about your target.

Head over to the `AppDelegate.swift` file. First, add the following `enum` to the top of the file, right underneath the `import` statements:

```
private enum ActionIdentifier: String {
    case comment
}
```

Even though it’s just a single action, you should still use an `enum` so that additions are easier in the future with less code refactoring.

You'll need to update your `registerCustomActions()` method in the `AppDelegate.swift` file to include an action button. This time, though, you'll use the `UNTextInputNotificationAction` type. Replace `registerCustomActions()` with:

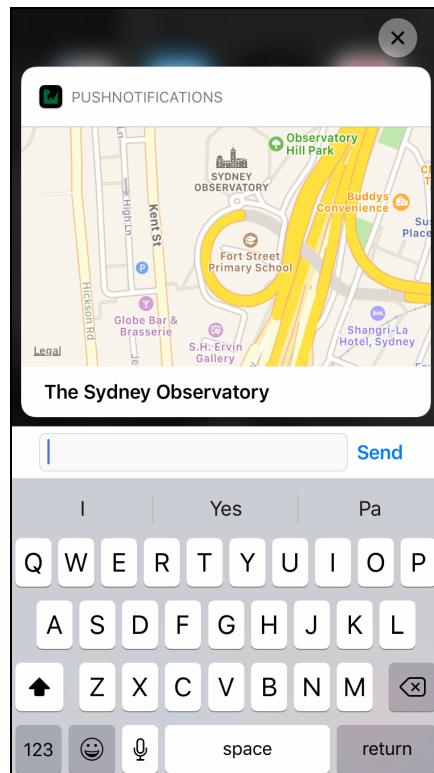
```
private func registerCustomActions() {
    // 1
    let ident = ActionIdentifier.comment.rawValue
    let comment = UNTextInputNotificationAction(identifier: ident,
                                                title: "Comment")

    // 2
    let category = UNNotificationCategory(identifier: categoryIdentifier,
                                           actions: [comment],
                                           intentIdentifiers: [])
    UNUserNotificationCenter.current()
        .setNotificationCategories([category])
}
```

A couple points to note in the code from above:

1. You're asking for *text* input instead of a button click, so be sure you use the `UNTextInputNotificationAction` action type.
2. The only change here is remembering to pass in your action.

If you build and run the app, then send that same push notification to yourself again, you should now have a keyboard on screen!



You'll notice that you received a keyboard directly and not a **Comment** button. iOS is smart enough to realize that, if your only action is a keyboard action, it should just show the keyboard by default. If you were to add another action, however, you'd instead get an actual button labeled **Comment** that you'd tap to open the keyboard.

Showing a keyboard is great, but now you've got to know what was said! To get the text that was typed by the user, you must implement a new delegate method, `didReceive(_:completionHandler:)`. In your UI extension, in **NotificationViewController.swift**, add:

```
func didReceive(  
    _ response: UNNotificationResponse,  
    completionHandler completion:  
    @escaping (UNNotificationContentExtensionResponseOption) -> Void) {  
  
    // 1  
    defer { completion(.dismiss) }  
  
    // 2  
    guard let response = response as? UNTextInputNotificationResponse  
    else {  
        return  
    }  
  
    // 3  
    let text = response.userText  
    print("The user responded with: \(text)")  
}
```

Here's what's going on in the code above:

1. As with most of the notification delegates, you must call the completion handler no matter how you exit the method. See below for an explanation of the parameter.
2. By looking at the type of response, you can determine whether or not you've received text from the end user to process.
3. All that's left to do is grab the text the user typed and process it. Frequently, this will mean calling some web service that you've implemented to store the response and possibly send it back out to other users.

Note: `didReceive(_:)` is called when the notification is displayed to configure the UI itself. `didReceive(_:completionHandler:)` is called in response to tapping an action button or by pressing **Send** on the keyboard.

When using a custom UI, it's not immediately obvious what to do with the notification after you've tapped a button or sent text. Is that it? Should iOS now dismiss the notification? Usually, the answer is yes, but sometimes you'll want to send text *and* be able to hit a social media *like*-type button. In the latter case, you wouldn't want the Notification window to go away.

If multiple interactions with your UI are possible, you'd instead want to pass `.doNotDismiss` to the completion handler.

There is a third, not normally used, possibility. You can specify `.dismissAndForwardAction` to simply dismiss the custom UI and send the notification straight to your main app.

Changing actions

It's also possible to modify the action buttons dynamically inside of your **Notification Content Extension**. If you're sending a social media notification, for example, you may want to provide a button to let the end-user "like" your content. Once you've tapped the **Like** button, it only makes sense to now provide an "Unlike" button in its place. In the case of your spy app, you'll add an "Accept" and "Cancel" buttons, to accept your next target and cancel the mission if anything goes wrong.

By simply modifying the `notificationActions` property on the `extensionContext` variable you can do just that!

First, in **NotificationViewController.swift**, add the following `enum` to the top of the class:

```
enum ActionIdentifier: String {
    case accept
    case cancel
}
```

These are identifiers for your **Accept** and **Cancel** actions. Next, update the `didReceive(_:_completionHandler:)` with the following code:

```
func didReceive(
    _ response: UNNotificationResponse,
    completionHandler completion:
        @escaping (UNNotificationContentExtensionResponseOption) -> Void) {
    let accept = ActionIdentifier.accept.rawValue
    let cancel = ActionIdentifier.cancel.rawValue

    switch response.actionIdentifier {
```

```
case accept:
    let cancelAction = UNNotificationAction(identifier: cancel,
                                              title: "Cancel")
    let currentActions = extensionContext?.notificationActions ?? []
    extensionContext?.notificationActions = currentActions.map {
        $0.identifier == accept ? cancelAction : $0
    }
case cancel:
    let acceptAction = UNNotificationAction(identifier: accept,
                                              title: "Accept")
    let currentActions = extensionContext?.notificationActions ?? []
    extensionContext?.notificationActions = currentActions.map {
        $0.identifier == cancel ? acceptAction : $0
    }
default:
    break
}

completion(.doNotDismiss)
}
```

The `actionIdentifier` property inside the notification response tells you which button was tapped. If the user tapped the **Accept** button, you'll create a **Cancel** button and replace the existing **Accept** button with it. Similarly, if the user tapped the **Cancel** button, you'll replace it with the **Accept** button.

While you could make this a tiny bit easier to read by simply replacing the index of the button directly, as opposed to using the `map`, this is definitely much more future-proof. This way, you don't have to worry if you decide to add new buttons that change the order of your actions.

Still in **NotificationViewController.swift**, add the following lines to the bottom of the `didReceive(_:)` method, right after you set the region on the map view:

```
let acceptAction = UNNotificationAction(
    identifier: ActionIdentifier.accept.rawValue,
    title: "Accept")
extensionContext?.notificationActions = [acceptAction]
```

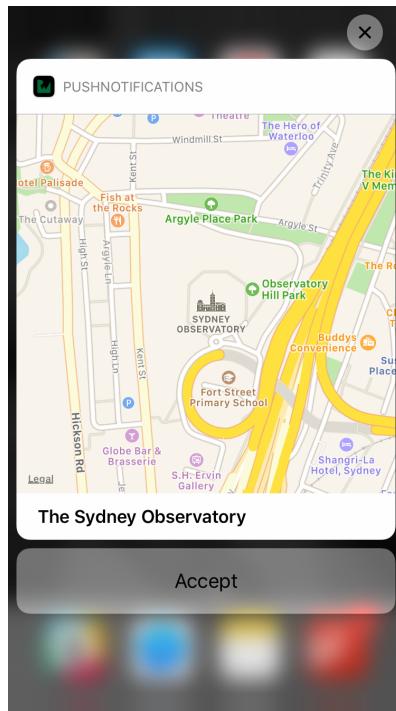
This will make sure the **Accept** action shows up when you receive a notification.

Finally, you have to remove the comment action. Head to **AppDelegate.swift** and modify the contents of the `registerCustomActions` method to the following:

```
private func registerCustomActions() {
    let category = UNNotificationCategory(identifier: categoryIdentifier,
                                           actions: [],
                                           intentIdentifiers: [])
    UNUserNotificationCenter.current()
        .setNotificationCategories([category])
}
```

Since you’re setting the actions inside the UI extension, there’s no need to set them in `AppDelegate`.

Build and run your app. You should see an **Accept** button on the notification and, when you tap it, it should change into a **Cancel** button.



By using this in conjunction with modifying the custom UI in response to tapping on an action button, you can now present a very rich user experience.

The fact that Apple now makes the action buttons dynamic also means that you’re no longer required to set up all of your actions when you register your category. You might, for example, simply register the category to trigger the extension and then dynamically generate all of your buttons based on the content of the payload, which provides major flexibility benefits.

You’re also able to present layered actions, but you need to again think very carefully about your user experience doing this. For example, the “Like” button may replace all the existing buttons with something like Love, Like, Kind of Like, and Meh. However, just because you *can* do something doesn’t mean that you *should* do something!

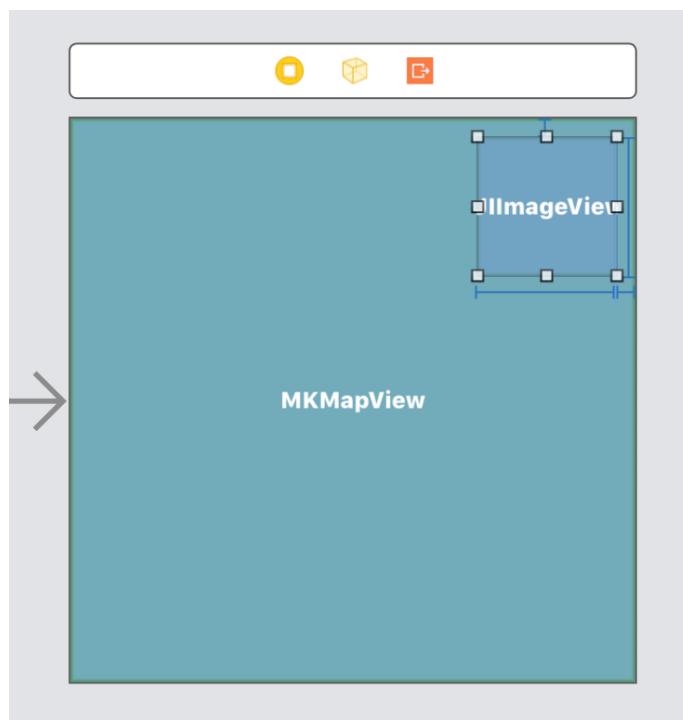
Attachments

If your project also includes a Service Notification Extension, it will be executed before your Notification Content Extension. A frequent reason you'd have both extensions is that the former will download an attachment that the latter wants to use. It's not enough to just know where your mission's target is. You also need to know what they look like; that's why you'll add a small image of your target's headshot to your notification.

In the previous chapter, Chapter 10, “Modifying the Payload,” you used a notification service extension to download a video. A similar service extension is already included in your starter project. It will try to download an image and a video, and then add them as attachments to the notification. This lets you use those attachments in your content extension.

Head over to **MainInterface.storyboard** and drag an **Image View** into the **View**. Add the following constraints to the image view:

1. A **width** constraint with a constant value of **80**.
2. A **height** constraint with a constant value of **80**.
3. **Trailing Space to Safe Area** from the image view to the **View** with a constant of **8**.
4. **Top Space to Safe Area** from the image view to the **View** with a constant of **8**.



Next, add another outlet in **NotificationViewController.swift**:

```
@IBOutlet var imageView: UIImageView!
```

Go back to **MainInterface.storyboard** and connect your new **Image View** to your new outlet.

Now, it's time to set the image on the image view. In **NotificationViewController.swift**, add the following code to the bottom of `didReceive(_:`), after setting the map's location and the notification actions:

```
var images: [UIImage] = []

notification.request.content.attachments.forEach { attachment in
    if attachment.url.startAccessingSecurityScopedResource() {
        if let data = try? Data(contentsOf: attachment.url),
           let image = UIImage(data: data) {
            images.append(image)
        }
        attachment.url.stopAccessingSecurityScopedResource()
    }
}

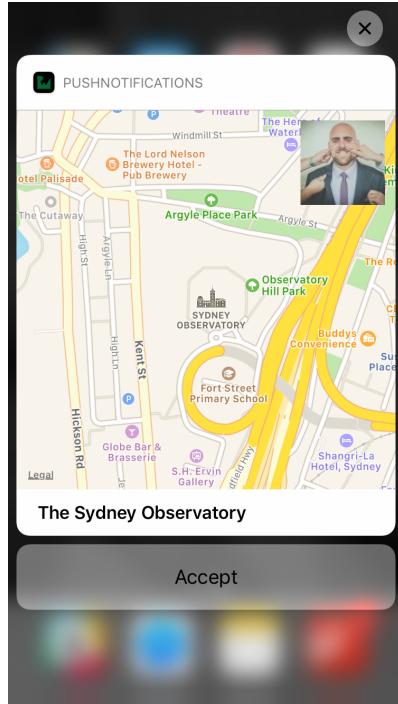
imageView.image = images.first
```

Here, you fetch the image from the notification content. Due to the way iOS performs its sandboxing, for security reasons, you can't just directly access the attachment. You must wrap access to the attachments in calls to start and stop accessing scoped resources.

Build and run the app. In the push notification tester app, set the payload to the following JSON:

```
{
  "aps": {
    "alert" : {
      "title" : "The Sydney Observatory"
    },
    "category" : "ShowMap",
    "sound": "default",
    "mutable-content": 1
  },
  "latitude" : -33.859574,
  "longitude" : 151.204576,
  "radius" : 500,
  "media-url": "https://www.gravatar.com/avatar/8477f7be4418a0ce325b2b41e5298e4c.jpg"
}
```

Send the push notification. You should see an attached image on the notification and, when you press into it, you should see an image of your next target:



Whoa! Looks like Shai is in for some big trouble.

Video attachments

Things get more complicated when your attachment is a video file, however. While this is out-of-scope for your spy app, it's still a valuable feature to know about.

As you remember, custom UI notifications are *not* interactive by default, meaning you can't simply tap on a media player to start and stop the video like you normally would.

Given you have a video player as part of your custom notification UI, you'll need to implement at least two of the three optional delegate properties:

```
// 1
var mediaPlayPauseButtonType:
UNNotificationContentExtensionMediaPlayPauseButtonType {
    return .overlay
}

// 2
var mediaPlayPauseButtonFrame: CGRect {
    return CGRect(x: 0, y: 0, width: 44, height: 44)
}

// 3
```

```
var mediaPlayPauseButtonTintColor: UIColor {  
    return .purple  
}
```

Here's what these lines of code are for:

1. You ask iOS to draw a button that either disappears on play (`.overlay`) or stays onscreen (`.default`).
2. You must tell iOS exactly what `CGRect` to use for positioning and sizing the **Play** button.
3. Optionally, you can specify the tinting of the button to match your theme.

The button iOS draws for you will be tappable. When tapped, the `UNNotificationContentExtension` delegate methods `mediaPlay()` and `mediaPause()` will be called so that you can take action on your video player controller.

Custom user input

While action buttons and the keyboard are great, sometimes you really just want your own custom interface for user input – a grid of buttons, sliders, etc...

Note: If you provide a custom input, you can't also have an option for the keyboard to appear. You need to pick one or the other.

Adding a payment action

Agents need to get paid! You'll add a slider that the agents can use to select how much they want to get paid for the job. Head back into your app's `AppDelegate.swift` file and change your text action back to a normal action to represent the payment.

Add a new case to the `ActionIdentifier` enum:

```
case payment
```

Next, replace `registerCustomActions()` with the following:

```
private func registerCustomActions() {  
    let ident = ActionIdentifier.payment.rawValue  
    let payment = UNNotificationAction(identifier: ident,  
                                         title: "Payment")  
  
    let category = UNNotificationCategory(identifier: categoryIdentifier,  
                                         actions: [payment],
```

```
        intentIdentifiers: []  
    UNUserNotificationCenter.current()  
    .setNotificationCategories([category])  
}
```

Here, you set the new action as a category on the notification, as you did before.

Next, in **NotificationViewController.swift**, delete the following lines from `didReceive(_:)`:

```
let acceptAction = UNNotificationAction(  
    identifier: ActionIdentifier.accept.rawValue,  
    title: "Accept")  
extensionContext!.notificationActions = [acceptAction]
```

This will make sure the new payment action shows up below the notification.

The first responder

Remember way back when you first learned iOS programming, there was that pesky responder chain that never made much sense? Well, it's finally time to do something useful with it!

Still in **NotificationViewController.swift**, add an override to the top of the class to tell the system that you can, in fact, become the first responder:

```
override var canBecomeFirstResponder: Bool {  
    return true  
}
```

When the user taps on your **Payment** button, you want to become the first responder so that you can present a custom user interaction view. Replace your `didReceive(_:completionHandler:)` method with the following to make that happen!

```
func didReceive(  
    _ response: UNNotificationResponse,  
    completionHandler completion:  
    @escaping (UNNotificationContentExtensionResponseOption) -> Void) {  
  
    becomeFirstResponder()  
    completion(.doNotDismiss)  
}
```

The user input

If you become the first responder, iOS will expect you to return a view via the `inputView` property that contains your custom user interaction view. The sample project already includes a `PaymentView` for you that will display a slider for selecting payments.

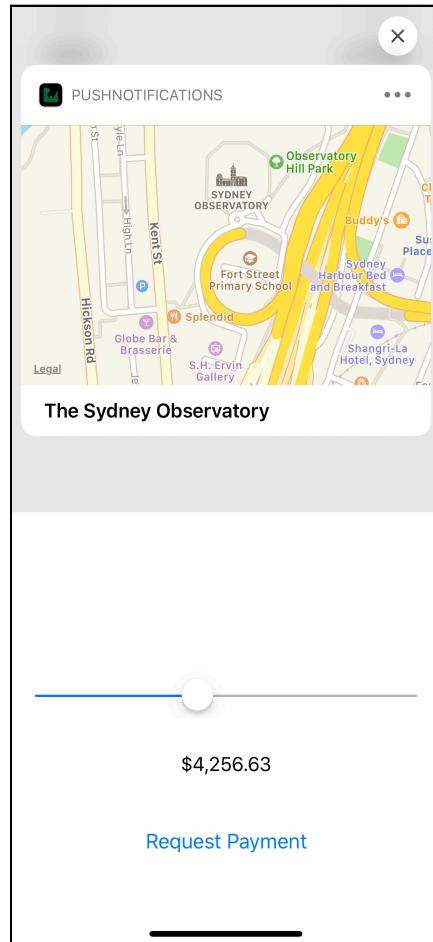
Drag the **PaymentView** folder from the starter project folder into the **Custom UI** group in **Xcode**. Make sure **Copy items if needed** is checked, and also that the **Custom UI** target is checked.

Back in **NotificationViewController.swift**, add the following properties to the top of the class to tell the system to use your new view:

```
private lazy var paymentView: PaymentView = {
    let paymentView = PaymentView()
    paymentView.onPaymentRequested = { [weak self] payment in
        self?.resignFirstResponder()
    }
    return paymentView
}()

override var inputView: UIView? {
    return paymentView
}
```

When the view controller becomes the first responder, iOS will ask it for the input view to display. Build and run the app and send yourself another push notification. After tapping on the **Payment** button, you'll see a slider to select your payment:

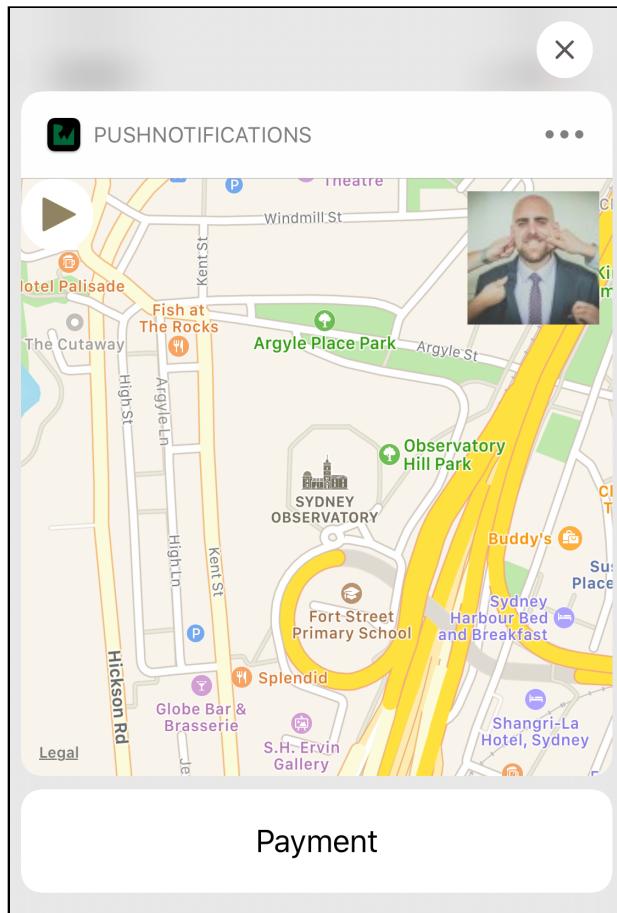


Hiding default content

If you’re creating a custom UI, odds are that you’re already presenting the title and body of the notification somewhere in your UI. If that’s the case, you can tell iOS to not present that default data under your view by editing the content extension’s **Info.plist**. Expand the **NSExtension** property again. This time, under **NSExtensionAttributes**, add a new Boolean key called **UNNotificationExtensionDefaultContentHidden** and set its value to YES.

Key	Type	Value
Bundle versions string, short	String	1.0
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
▼ NSExtension	Dictionary	(3 items)
▼ NSExtensionAttributes	Dictionary	(3 items)
UNNotificationExtensionInitialContentSizeRatio	Number	1
UNNotificationExtensionCategory	String	ShowMap
New UNNotificationExtensionDefaultContentHidden	Boolean	YES
NSExtensionPointIdentifier	String	com.apple.usernotifications.content-extension
NSExtensionMainStoryboard	String	MainInterface

Delivering a notification with this setting will show the same custom UI, without the title text:



Interactive UI

If you want to support interactive touches on your custom user interface, you need to edit the `Info.plist` of your extension and add the

`UNNotificationExtensionUserInteractionEnabled` attribute key with a value of YES.

Key	Type	Value
▼ Information Property List	Dictionary	(10 items)
Localization native development region	String	\$(DEVELOPMENT_LANGUAGE)
Bundle display name	String	Custom UI
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	XPC!
Bundle versions string, short	String	1.0
Bundle version	String	1
▼ NSExtension	Dictionary	(3 items)
▼ NSExtensionAttributes	Dictionary	(4 items)
New UNNotificationExtensionDefaultContentHidden	Boolean	YES
UNNotificationExtensionCategory	String	ShowMap
UNNotificationExtensionInitialContentSizeRatio	Number	1
UNNotificationExtensionUserInteractionEnabled	Boolean	YES
NSExtensionMainStoryboard	String	MainInterface
NSExtensionPointIdentifier	String	com.apple.usernotifications.content-extension

At this point, you can create an `IBOutlet` like you would on a normal view controller and link appropriate actions to them. It's important to remember that you are responsible for handling all of the actions and callbacks once you've done this. Tapping on the UI will no longer open your app, for example.

Launching the app

Depending on the content of your UI, it may make sense to have a button tap launch your app. This is as simple as calling a single method:

```
extensionContext?.performNotificationDefaultAction()
```

Once that's called, your app's `userNotificationCenter(_:_:didReceive:withCompletionHandler:)` delegate method will be called, and the identifier will be set to `UNNotificationDefaultActionIdentifier`.

Dismissing the UI

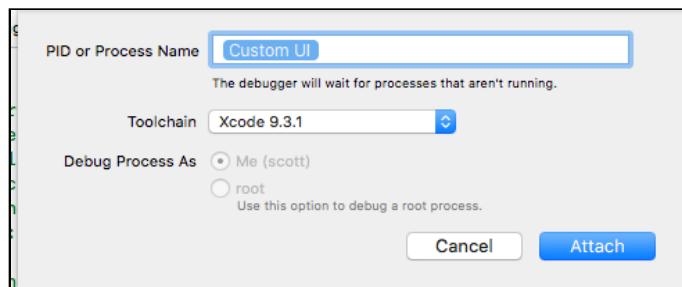
Similarly to being able to launch your app, you can also dismiss the UI based on a button tap. As usual, you'll want to call a method on the `extensionContext`:

```
extensionContext?.dismissNotificationContentExtension()
```

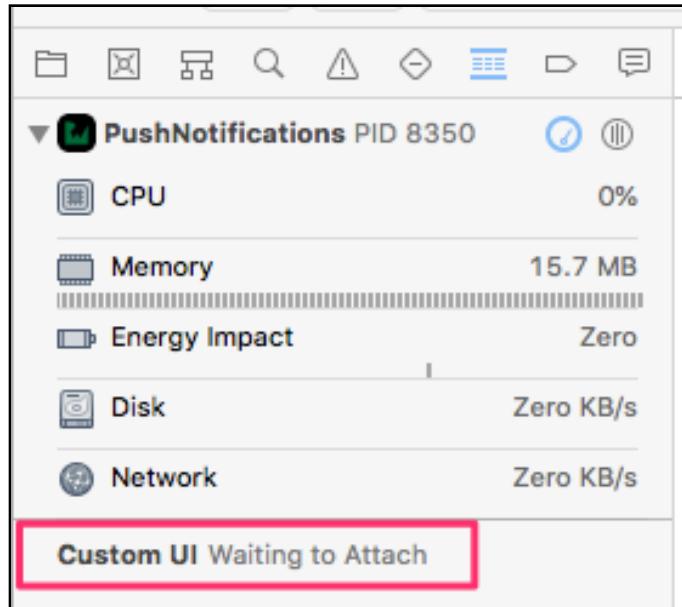
Debugging

Debugging a UI extension works almost the same as any other Xcode project. However, because it's a target and not an app, you have to take a few extra steps.

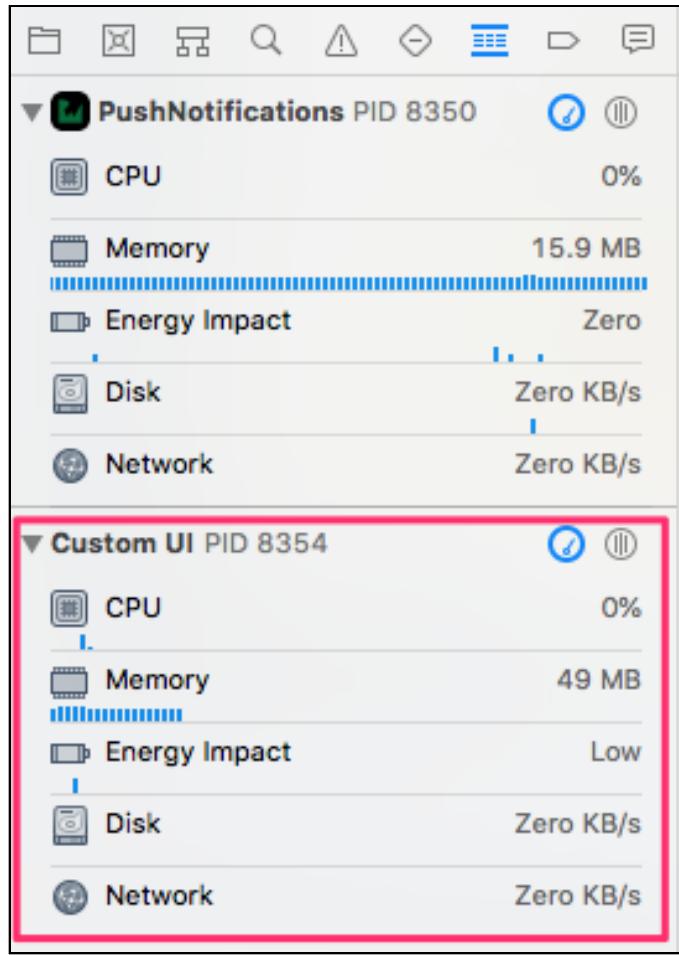
1. Open up your **NotificationViewController.swift** file and set a breakpoint where you need to start debugging.
2. Build and run your app.
3. In Xcode's menu bar choose **Debug > Attach to Process by PID or Name....**
4. In the dialog window that appears, enter **Custom UI**, or whatever you named your target.
5. Press the **Attach** button.



If you switch over to the **Debug Navigator** ($\text{⌘} + 7$) you'll see that Xcode is waiting for your target to start before it can attach to it.



If you send yourself another push and open up the custom UI, Xcode will show that it's attached to your process.



It's important to point this out as you need to wait for the process to be attached before you interact with your user interface beyond the initial 3D Touch to open the UI. If you tap on anything before Xcode has attached, you won't actually hit your breakpoint.

Key points

- You can customize the look of a push notification; custom interfaces are implemented as separate targets in your Xcode project, just like the service extension.
- Custom interfaces are triggered by specifying a category and every custom UI must have its own unique category identifier.

- There are a number of customizations you can make such as allowing your user to respond to a push notification with text, changing action buttons, allowing attachments and tailoring your interface for user input like payment actions. You can also hide default content and create an interactive UI. All of these features will enhance your user experience and make your app really stand out.

Where to go from here?

Phew, that was a lot of information to take in! Take a moment to applaud yourself.

You now possess more knowledge about crafting high-end, customized notifications, than most iOS developers. The knowledge you've gained in this chapter opens a gate to endless creative and personalized opportunities for users of your app. Take a big breath and get ready for the next chapter: "Putting It All Together," where you'll use all of the knowledge gained throughout this book and, well... put it all together!

Chapter 12: Putting It All Together

With 11 chapters behind you, you've become quite the master of everything related to Push Notifications!

This chapter is all about leveraging all that you've learned in this book into a single app, titled **CoolCalendar**.

When somebody sends you a calendar invite, it will be pushed to your device via a remote notification. You'll have the ability to see how the new event relates to your existing calendars, be able to accept/reject right from the notification and have the option of sending a comment back.

Setting up the Xcode project

The starter project for this chapter uses a third-party library called **CalendarKit**, which comes pre-installed via CocoaPods, to simplify the presentation of a Calendar UI in your app. Be sure to open **CoolCalendar.xcworkspace** and *not* **CoolCalendar.xcodeproj**. If you accidentally open the latter, you'll get multiple compiler errors.

First, it's time to set up your workspace:

1. Open **CoolCalendar.xcworkspace** from the starter project.
2. Set the team signing for both the **CoolCalendar** target and the **Custom UI** extension, as discussed in Chapter 7, “Expanding the Application.”
3. Enable the **Push Notifications** capability as discussed in Chapter 4, “Xcode Project Setup.”

4. Enable **Remote notifications** as part of **Background Modes** as discussed in Chapter 8, “Handling Common Scenarios.”
5. Add a **Notification Service Extension** as discussed in Chapter 10, “Modifying the Payload.”

A **Notification Content Extension** was already included in the starter project, as some sample code was provided for you. In your own projects, you’d have to create that target yourself.

AppDelegate code

Take a minute to set up your AppDelegate code the way you think it should be. Keep in mind all the items discussed in the preceding chapters and don’t be afraid to flip back to one or more for help!

Some features that you’ll want to be sure to handle:

- You’ll need action identifiers to know which custom action buttons were selected. Plan to have buttons for **Accept**, **Decline** and **Comment**.
- You’ll need to register all of your custom actions.
- You’ll need to register for push notifications.
- The payload’s custom category will be called `CalendarInvite`.

Try to do this yourself. You can use the `registerForPushNotifications` method found in **ApnsUploads.swift** — just make sure to uncomment it first!

When you’re ready, turn the page to see one potential solution. The code that was already provided in the starter project has been removed for clarity.

Here’s the solution:

Start by creating a new file called **ActionIdentifier.swift** where you’ll define your Action Identifiers. Add the following enum to the file:

```
enum ActionIdentifier: String {  
    case accept  
    case decline  
    case comment  
}
```

Select the newly created file and, in the **File Inspector**, make sure the **Target Membership** is checked for **Custom UI**.

Back in **AppDelegate.swift**, add the following code inside your AppDelegate:

```
private let categoryIdentifier = "CalendarInvite"

private func registerCustomActions() {
    let accept = UNNotificationAction(
        identifier: ActionIdentifier.accept.rawValue,
        title: "Accept")

    let decline = UNNotificationAction(
        identifier: ActionIdentifier.decline.rawValue,
        title: "Decline")

    let comment = UNTTextInputNotificationAction(
        identifier: ActionIdentifier.comment.rawValue,
        title: "Comment", options: [])

    let category = UNNotificationCategory(
        identifier: categoryIdentifier,
        actions: [accept, decline, comment],
        intentIdentifiers: [])
}

UNUserNotificationCenter
    .current()
    .setNotificationCategories([category])
}
```

Next, add this code to the UIApplicationDelegate extension inside the same file:

```
func application(
    _ application: UIApplication,
    didRegisterForRemoteNotificationsWithDeviceToken
    deviceToken: Data) {

    registerCustomActions()
    sendPushNotificationDetails(
        to: "http://192.168.1.1:8080/api/token",
        using: deviceToken)
}
```

Also, add the following **UNUserNotificationCenterDelegate** extension at the bottom of the file:

```
extension AppDelegate: UNUserNotificationCenterDelegate {
    func userNotificationCenter(
        _ center: UNUserNotificationCenter,
        willPresent notification: UNNotification,
        withCompletionHandler completionHandler:
            @escaping (UNNotificationPresentationOptions) -> Void) {

        completionHandler([.badge, .sound, .alert])
    }
}
```

ApnsUploads.swift already has the `registerForPushNotifications` method ready and waiting for you. Simply uncomment it.

Finally, all that's left is calling the `registerForPushNotifications` method. Back in your `AppDelegate`, Look for the `application(_:didFinishLaunchingWithOptions:)` method, and add the following line right before the `return` clause.

```
registerForPushNotifications(application: application)
```

If you build your app, it should compile cleanly at this point. Be sure not to move on until you are left with neither warnings nor errors.

Requesting calendar permissions

Accessing the user's calendar is a privacy concern, and so you'll have to first request permission of your end users. Apple kindly ensured that the same authorization status is shared by all targets of your app. This means that your extensions can simply look at the status and not have to ask for it, as the primary target already takes care of that. Like all good iOS apps, you'll have to tell your end users *why* you want to get into their calendars, so go back to the **CoolCalendar** target's **Info** panel and add a **Privacy – Calendars Usage Description** key. You can use any text that explains why you need access to the Calendar, such as, "We need access to the Calendar to import your events and important dates".

Now, edit **ViewController.swift** and request permission to the user's calendar when the view appears. This is boilerplate code that you'll use in any calendar app, but it's important to get it right.

First, add an event store to the top of the class. The event store is the way your app will communicate with the calendar data:

```
private let eventStore = EKEventStore()
```

Next, add the following method underneath the event store property:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    let status = EKEventStore.authorizationStatus(for: .event)

    switch (status) {
    case .notDetermined:
        eventStore.requestAccess(to: .event) {
            [weak self] granted, _ in
            guard !granted, let `self` = self else { return }
    }
```

```
    self.askForAccess()
}

case .authorized:
    break

default:
    askForAccess()
}
}
```

When the screen appears, ask the `EKEventStore` if the user has authorized calendar access. If the status is not yet determined, request access from the event store, which will trigger a system alert asking the user for permission. If the user has denied permission, you will show a new alert with a handy button to open the settings screen.

As you might've noticed, you're using `askForAccess`, which is still not implemented. Replace it with the following code:

```
private func askForAccess() {
    let alert = UIAlertController(
        title: nil,
        message: "This application requires calendar access",
        preferredStyle: .actionSheet)

    alert.addAction(UIAlertAction(
        title: "Settings",
        style: .default) { _ in
        let url = URL(
            string: UIApplication.openSettingsURLString)!
        UIApplication.shared.open(url)
    })

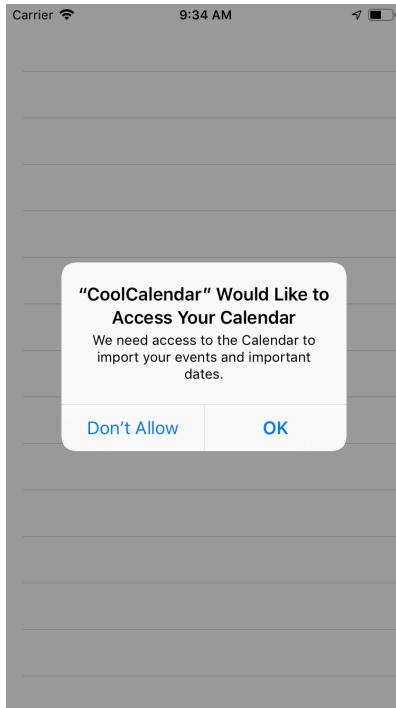
    alert.addAction(UIAlertAction(
        title: "Cancel",
        style: .default))

    present(alert, animated: true)
}
```

This will show a new alert in which one of the buttons opens up your application's **Settings**.

Note: It's a nice touch to give the user a simple way to get to your app settings if permissions aren't currently granted. However, you should be sure *where* you're doing this from. In the case of this app, it's done every time the view appears if permission has been denied or not yet requested. If your app *requires* calendar access, this makes sense. However, if it's optional, and not 100% necessary to your app's functionality, then you'll just annoy the end user if you ask every single time.

Build and run the app to verify that you're asked to grant calendar permissions and to allow push notifications.



Be sure to tap **OK** on both! If the app crashes at this point, you probably added the privacy policy to one of the extension targets instead of the main app.

The payload

When it's time to invite somebody to an event, you'll send a remote notification with a payload that looks like this:

```
{  
    "aps" : {  
        "alert" : {  
            "title" : "New Calendar Invitation"  
        },  
        "badge" : 1,  
        "mutable-content" : 1,  
        "category" : "CalendarInvite"  
    },  
    "title" : "Family Reunion",  
    "start" : "2018-04-10T08:00:00-08:00",  
    "end" : "2018-04-10T12:00:00-08:00",  
    "id" : 12  
}
```

Notice that you're setting the `mutable-content` key to 1 so that your service extension runs, as well as a category so that your custom UI extension is triggered. The last four fields simply specify the details of the event. This packet structure also assumes that your server is tracking the calendar invitations to know who accepted and rejected them, which is why there is an `id` key which uniquely identifies this invitation in your database. To make life easy, the dates use the **ISO8601** date format.

Notification Service Extension

The goal of your remote push notification is to provide a custom user interface to accept/reject/comment on the calendar invitation that's sent to the end user. What happens if the end user doesn't allow calendar access? It would be pretty strange to pop up the UI asking them to take action.

Even though you can't stop a notification from going through, you *can* change the notification. In this case, that means that you should check if calendar permissions are granted. What action do you think you could take if permission is denied? The simplest solution is to remove the category from the payload, which would prevent the custom UI from appearing at all!

The pieces to be implemented in the service extension are threefold:

1. Remove the category if calendar permissions aren't granted.
2. Update the app icon badge.
3. Update the body of the notification.

Spend some time trying to implement those three items yourself and then come back to see the way the goals are accomplished, here.

Validate calendar permissions

Modify `NotificationService.swift` to include an import of `EventKit`:

```
import EventKit
```

Then update the `didReceive` method to blank out the `category` field of the payload if calendar permissions are denied by adding this check immediately before your call to the `contentHandler` closure:

```
if EKEventStore.authorizationStatus(for: .event) != .authorized {  
    bestAttemptContent.categoryIdentifier = ""  
}
```

Setting `categoryIdentifier` to an empty string will ensure the Content UI doesn't display.

App badging

It's time to add a badge to your app's icon for when a new notification comes in. The first step is to create an **App Group**. If you don't remember how to do this, follow the steps shown in Chapter 10, "Modifying the Payload."

Badging the app is handled by using the App Group you created and the `UserDefaults` class. Create a new Swift file called **UserDefaults.swift** in the **CoolCalendar** target with code to set an integer in the proper App Group. Be sure you update the name of the suite to match what you called the App Group!

```
extension UserDefaults {
    static let appGroup = UserDefaults(
       (suiteName: "group.com.razeware.CoolCalendar")!

    private enum Keys {
        static let badge = "badge"
    }

    var badge: Int {
        get {
            return integer(forKey: Keys.badge)
        } set {
            set(newValue, forKey: Keys.badge)
        }
    }
}
```

Since you're going to use this file in both the primary target and the service extension, click on the newly created file and, in the **File Inspector**, check **Payload Modification** inside **Target membership**.

With that done, you can now add a method to **NotificationService.swift** to update the badge:

```
private func updateBadge() {
    guard
        let bestAttemptContent = bestAttemptContent,
        let increment = bestAttemptContent.badge as? Int
    else { return }

    switch increment {
        case 0:
            UserDefaults.appGroup.badge = 0
            bestAttemptContent.badge = 0
        default:
            let current = UserDefaults.appGroup.badge
            let new = current + increment
    }
}
```

```
        UserDefaults.appGroup.badge = new
        bestAttemptContent.badge = NSNumber(integerLiteral: new)
    }
```

When the service extension completes, iOS will set the badge on your app icon to the value specified in the payload. You've handled the update here by incrementing the *existing* badge count based on what was sent in the payload so that the count properly increments after each invitation is received.

When the user runs the app, you'll of course want to blank the badge count out. To do so, return to **AppDelegate.swift** and blank out the badge count once the app starts by adding the following method to the **UIApplicationDelegate** extension:

```
func applicationDidBecomeActive(_ application: UIApplication) {
    UserDefaults.appGroup.badge = 0
}
```

You'll notice that the **UserDefault**s extension wasn't technically necessary here as you're not decrementing based on which invitations you've seen. In a normal production app, however, you'd only want to decrement the count when the user actually sees the specific invitations that were new.

Notification body

The final task is parsing your custom payload data and updating the body of the notification message to something more user friendly. Add the following method inside the class in **NotificationService.swift**:

```
private func updateText(request: UNNotificationRequest) {
    let formatter = ISO8601DateFormatter()

    guard
        let bestAttemptContent = bestAttemptContent
    else { return }

    guard
        EKEventStore.authorizationStatus(for: .event)
    == .authorized,
        let userInfo = request.content.userInfo as? [String : Any],
        let title = userInfo["title"] as? String, !title.isEmpty,
        let start = userInfo["start"] as? String,
        let startDate = formatter.date(from: start),
        let end = userInfo["end"] as? String,
        let endDate = formatter.date(from: end),
        let _ = userInfo["id"] as? Int
    else {
        bestAttemptContent.categoryIdentifier = ""
        return
    }
}
```

```
let rangeFormatter = DateIntervalFormatter()
rangeFormatter.dateStyle = .short
rangeFormatter.timeStyle = .short

let range = rangeFormatter.string(
    from: startDate,
    to: endDate)
bestAttemptContent.body = "\(title)\n\(range)"
}
```

Notice how, if any piece of the required payload is missing, or in an incorrect format, the `categoryIdentifier` is blanked out. It doesn't make sense to let the custom UI code get called when it would simply fail. Two separate guard clauses are required, as Swift will not allow access to `bestAttemptContent` in the failure condition of a guard clause where that variable is checked.

Using ISO8601 dates is very convenient, as Apple has provided a parser explicitly for that format! As long as the payload contains all the expected keys in the proper date formats, the body is updated to include the title and the date range. With dates, you'll always want to utilize the provided classes, such as `DateIntervalFormatter` to ensure that the user's locale is properly respected.

While nothing needs to be done with the invitation id, you still want to ensure that it exists in the payload so that you know there's valid content to pass to the Content Service Extension.

All that's left to do in this file is to call both of the methods that you just implemented from `didReceive` just before the completion handler is called. Your final `didReceive` method should now look like this:

```
override func didReceive(
    _ request: UNNotificationRequest,
    withContentHandler contentHandler:
        @escaping (UNNotificationContent) -> Void) {

    self.contentHandler = contentHandler
    bestAttemptContent = request.content.mutableCopy()
        as? UNMutableNotificationContent
    guard
        let bestAttemptContent = bestAttemptContent
    else { return }

    if EKEventStore.authorizationStatus(for: .event)
        != .authorized {
        bestAttemptContent.categoryIdentifier = ""
    }

    updateBadge()
    updateText(request: request)
}
```

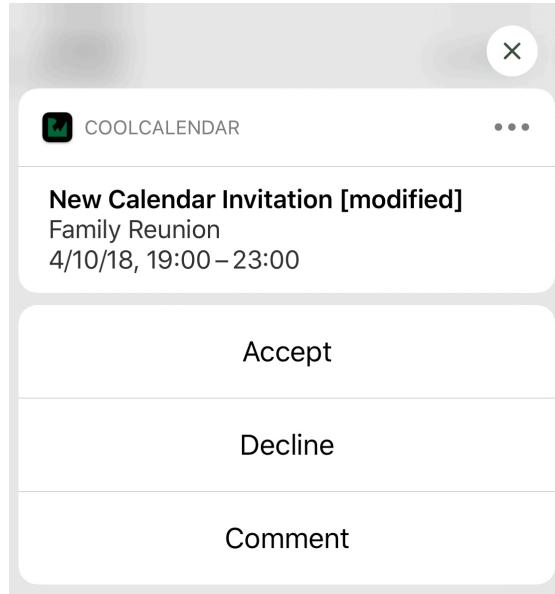
```
    contentHandler(bestAttemptContent)
}
```

Phew! There's almost more text explaining what to do than it actually takes to do it! You can see how modifying the payload might seem daunting at first, but you can take significant action with very little code. The net benefit to your end users is a much better experience, which always makes the little bit of extra effort worth it.

In a production app there are other considerations you might want to take, such as:

- What about all-day events?
- What about recurring events?
- What if the title is an empty string?
- What happens if you send a start date that comes after an end date?

This is a great time to build and run the app again and send yourself a push notification. You can do this with the **PushNotifications** tester app as described in Chapter 5, “Apple Push Notifications Servers.” You can find the payload at the start of this chapter.



Has the body of the text message been updated properly? If there's no change, make sure you remembered to set `mutable-content` to 1 in the `aps` part of the payload. If it's still not working, refer back to Chapter 10, “Modifying the Payload,” for help.

Those goodies in your kitchen aren't going to eat themselves. You've done some great work so grab yourself a snack, take a quick break and then it'll be time to work on the user interface.

Content Service Extension

Instead of just asking for a response, wouldn't it be nicer to show your users what their calendars look like for the time period related to the new event that they were invited to? To do this, you'll use a library called **CalendarKit** that the starter project has included. As the goal here isn't to teach you how to use CalendarKit, the starter project already includes the code related to that library for you.

Considering what the goals of the UI will be leads to the following five tasks:

1. Set the **Info.plist** details related to the category that you're using.
2. Add the newly arrived invitation to CalendarKit.
3. Display all events happening at the same time as the new event.
4. If the user accepts the invite, add it to iOS's calendar.
5. If the user comments, update the server.

It probably seems a bit silly to list out such simple tasks, but thinking of the UI tasks ahead of time helps to break down what, at first, seems like a daunting challenge into manageable pieces that you can focus on.

Updating the Info.plist

Open up **Info.plist** inside of the **Custom UI** target folder and expand out the **NSExtension** key all the way, as you learned to do in Chapter 11, "Custom Interfaces." You'll need to update the **UNNotificationExtensionCategory** value to match what you set the **categoryIdentifier** to be in **AppDelegate.swift**. If you've used the same category name as the book example, that means you'll need to put **CalendarInvite** as the value.

Since the calendar itself will contain the details of the notification, it's definitely not desirable to have iOS display the body of the notification in the UI. I know, I know... right now you're thinking to yourself, "What?! Then why did I *just* edit the body of the notification to be human readable?" Remember that you might have had to disable the custom UI portion. If it gets disabled, you'd still want a nice text message. If it's not, then you want the visual UI.

Create a new Boolean key, under **NSExtensionAttributes**, named **UNNotificationExtensionDefaultContentHidden** and set the value to YES.

Adding information to CalendarKit

You'll have to do the same extraction from the payload that you did in the Notification Service Extension, but, this time, there's no need to check for calendar access because, if you get here, it's guaranteed to be "on" as you just checked it in the Notification Service Extension.

In **NotificationViewController.swift**'s `didReceive(_:)`, after you do the parsing, you'll want to send the invitation details into CalendarKit. Add the below code just after the line that adds the timeline container as a subview:

```
let formatter = ISO8601DateFormatter()

guard
    let userInfo = notification.request.content.userInfo as? [String: Any],
    let title = userInfo["title"] as? String, !title.isEmpty,
    let start = userInfo["start"] as? String,
    let startDate = formatter.date(from: start),
    let end = userInfo["end"] as? String,
    let endDate = formatter.date(from: end),
    let id = userInfo["id"] as? Int
else {
    return
}

var appointments = [
    addCalendarKitEvent(
        start: startDate,
        end: endDate,
        title: title)
]
```

Getting nearby calendar items

Now that the new invitation is squared away, you'll need to find the events in the users' existing calendars that will occur around the same date/time. It's probably a good idea to consider a couple of hours before and after the event so that your users can plan for drive times, doctors always being late to the start of an appointment or other buffers of time needed.

Start off by creating a class-level property for the event store to the top of **NotificationViewController**:

```
private let eventStore = EKEventStore()
```

Determine what time is two hours before the invitation and two hours after.

In a production app, you'd need to do some extra checks to see how long the appointment is, for example, or whether it's an all-day event; you'd then need to modify the times accordingly. Never just add seconds to a date thinking it's the right thing to do. Always use the built-in calendrical calculations that Foundation provides so that you don't get caught by leap years, leap seconds, missing midnight hours and a slew of other time-related issues.

Add the following code to the bottom of `didReceive`, before the commented out lines:

```
let calendar = Calendar.current
let displayStart = calendar.date(
    byAdding: .hour,
    value: -2,
    to: startDate)!

let displayEnd = calendar.date(
    byAdding: .hour,
    value: 2,
    to: endDate)!

let predicate = eventStore.predicateForEvents(
    withStart: displayStart,
    end: displayEnd,
    calendars: nil)

appointments += eventStore
    .events(matching: predicate)
    .map {
        addCalendarKitEvent(
            start: $0.startDate,
            end: $0.endDate,
            title: $0.title,
            cgColor: $0.calendar.cgColor)
    }
```

After adding the above code, uncomment the commented-out lines related to the `timelineContainer`. Those three lines are necessary to make CalendarKit work properly but, until `displayStart` and `displayEnd` were defined, they would have resulted in confusing compiler errors.

Build and run the app, and send yourself another push notification. When you 3D Touch into the notification, you should see a UI showing the time slot for the new event, as well as any events you might have planned at the same time.

You can play with the time in the payload to test out different appointments.



Accepting and declining

The UI is now displaying a snapshot of part of the calendar so that the end user can make an informed decision about whether or not to accept the invitation. What happens when they accept or reject, though? You'll want to determine which option was chosen and then take some action, such as connecting to a REST endpoint to store the response.

If the invitation is declined, you'll probably want to update your server so it can process the event, and eventually you'll dismiss the UI. There's now an issue to consider: The `didReceive(_:completionHandler:)` method, which responds to the action buttons, has no idea what the event is. You'll fix that by adding another class-level property:

```
private var calendarIdentifier: Int?
```

Then set that in `didReceive(_:)` just after decoding the payload.

```
calendarIdentifier = id
```

You can now implement the `didReceive(_:completionHandler:)` method to start handling the actions by adding the following to the bottom of `NotificationViewController`:

```
func didReceive(
    _ response: UNNotificationResponse,
    completionHandler completion:
    @escaping (UNNotificationContentExtensionResponseOption)
-> Void) {

    guard let choice =
        ActionIdentifier(rawValue: response.actionIdentifier)
    else {
        // This shouldn't happen but definitely don't crash.
        // Let the users report a bug that nothing happens
        // for this choice so you can fix it.
        completion(.doNotDismiss)
        return
    }

    switch choice {
    case .accept, .decline:
        completion(.dismissAndForwardAction)

    case .comment:
        completion(.doNotDismiss)
    }
}
```

Are you getting a compiler error that `ActionIdentifier` is unknown? You know the drill! Add **Custom UI** to its target membership.

If the user chooses to enter a comment, bring up the keyboard and tell the completion handler that the UI window should stay active. If they accept or decline the invitation, the window can simply be dismissed. You're using a new option here, called `dismissAndForward`, which tells the UI to dismiss, while also forwarding the notification onto your primary app, triggering the `userNotificationCenter(_:didReceive:withCompletionHandler:)` method.

Because this app wants to display the responses to each invite in a table, it's necessary to store the response in a Core Data entity. While it's entirely possible to create a new entity in an extension target, it's *not* easy to know that it happened in the main target. Both targets would use the same root context from the `NSPersistentContainer`, and Foundation's notifications don't cross app targets. For this reason, it's simpler to leave the Core Data work to the primary app target. You'll handle that in just a bit.

Commenting on the invitation

This one takes a little more work to get properly. You want to be able to comment without the notification being dismissed as soon as you do. In order to make that happen, you've got to tell iOS that you're willing to become the first responder (i.e., provide a custom keyboard) and handle input by overriding `canBecomeFirstResponder`.

Add the following override to the top of `NotificationViewController`:

```
override var canBecomeFirstResponder: Bool {  
    return true  
}
```

When a keyboard appears from iOS, there's no way for you to get access to the `UITextField` that is presented. Since you need to know when the **Return** button is pressed, it's therefore necessary to replace the `UITextField` Apple provides with one of your own. The starter project has already created the `keyboardInputAccessoryView` for you for just this purpose.

By embedding the text field inside of another view, you can give some shading to the outer view, making the text field easier to see. Remember that you're using full UIKit-based controls here, so you can add as many features as you need such as buttons and date pickers to suggest a new times. Just always keep the user experience in mind as you add more controls.

The delegate has to be set on the `keyboardTextField` so that you can catch when the user taps the **Return** button on the keyboard to dismiss it. Add the following to the top of `textFieldShouldReturn`:

```
guard  
    textField == keyboardTextField,  
    let text = textField.text,  
    let calendarIdentifier = calendarIdentifier  
else {  
    return true  
}  
  
Server.shared.commentOnInvitation(with: calendarIdentifier,  
                                    comment: text)  
textField.text = nil  
  
keyboardTextField.resignFirstResponder()  
resignFirstResponder()
```

Something that's not immediately obvious until after you've done some UI testing is that the comment text the end user typed won't automatically disappear as the same `UITextField` is utilized each time the keyboard appears. That's why it's necessary to set the text field's `text` property to `nil` to properly clear it when you're done.

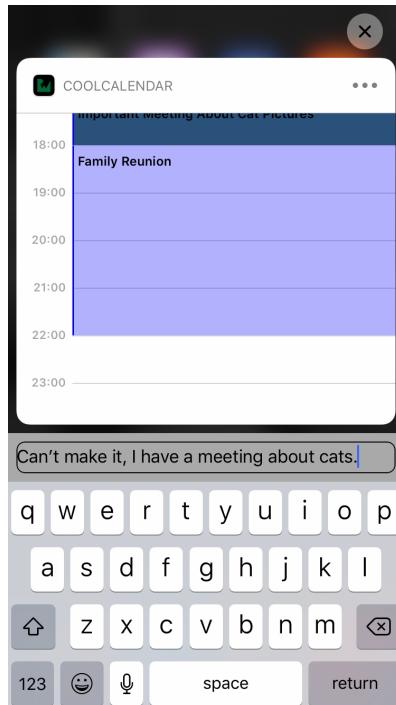
Of course, for iOS to know that you want to actually do something with the `UIView` that you just created, you've got to tell it so! Add this override to the top of the class:

```
override var inputAccessoryView: UIView? {
    return keyboardInputAccessoryView
}
```

All that's left to do is to display the keyboard when the **Comment** button is tapped inside the switch in `didReceive(_:completionHandler:)`:

```
case .comment:
    becomeFirstResponder()
    keyboardTextField.becomeFirstResponder()
    completion(.doNotDismiss)
```

Build and run. Send yourself some events and you should see your snazzy new custom UI with usable buttons!



Final cleanups

After sending some notifications, it'll quickly become apparent that nothing is happening in the main app when you accept or reject a notification. There is already code in `ViewController.swift` that shows the data, but you never actually create a Core Data entity anywhere. Time to resolve that issue!

Head over to `AppDelegate.swift` one last time. As mentioned, you'll generate the Core Data entities from one of `UNUserNotificationCenterDelegate`'s methods. Add the following method to the `UNUserNotificationCenterDelegate` delegate:

```
func userNotificationCenter(_ center: UNUserNotificationCenter, didReceive response: UNNotificationResponse, withCompletionHandler completionHandler: @escaping () -> Void) {  
    defer { completionHandler() }  
  
    let formatter = ISO8601DateFormatter()  
    let content = response.notification.request.content  
  
    guard  
        let choice = ActionIdentifier(rawValue: response.actionIdentifier),  
        let userInfo = content.userInfo as? [String : Any],  
        let title = userInfo["title"] as? String, !title.isEmpty,  
        let start = userInfo["start"] as? String,  
        let startDate = formatter.date(from: start),  
        let end = userInfo["end"] as? String,  
        let endDate = formatter.date(from: end),  
        let calendarIdentifier = userInfo["id"] as? Int  
    else {  
        return  
    }  
}
```

Remember that the completion handler *must* be called, which is why it's passed right into a `defer` block. The parsing is no different than before, but notice that there are a couple extra "dot walks" that have to happen to get to the `userInfo` property. Even though you know for a fact that everything will parse properly, it's never a good idea to use force unwrapping if there's another way; the `guard` syntax is a better choice here.

All that's left to do is to update the server with the user's decision and create a Core Data entity, which will then automatically be displayed in the table on the main view. You can do this by adding the following code to the bottom of the above method:

```
switch choice {  
case .accept:  
    Server.shared.acceptInvitation(with: calendarIdentifier)  
    createInvite(  
        with: title,  
        starting: startDate,  
        ending: endDate,  
        accepted: true)  
  
case .decline:  
    Server.shared.declineInvitation(with: calendarIdentifier)  
    createInvite(  
        with: title,
```

```
        starting: startDate,  
        ending: endDate,  
        accepted: false)  
  
    default:  
        break  
    }  
}
```

Build and run the app. Send yourself a few invites. When you open the app, you should see a list of accepted and declined appointments.



Where to go from here?

And, with that, your project is finished! Even so, there is still one final category of notifications to cover: local notifications, which you will explore in the next and final chapter.

Chapter 13: Local Notifications

Although you've put together the key concepts up to this point, there is one more category of notifications to cover: local notifications.

While the vast majority of notifications displayed on your device are remote notifications, it's also possible to display notifications originating from the user's device, locally. There are three distinct types of local notifications:

1. **Calendar:** Notification occurs on a specific date.
2. **Interval:** Notification occurs after a specific amount of time.
3. **Location:** Notification occurs when entering a specific area.

While less frequently used, local notifications still play an important role for many apps. You should also challenge the immediate notion of using a remote notification. For example, if you provide a food-ordering app, it might want to tell the user that the food is ready to pick up. Will the restaurant really take action when the food is ready or could you, instead, use an interval-based local notification to send the alert after a 10-minute waiting period?

You still need permission!

Even though the notification is created and delivered locally, on the user's device, you must still obtain permission to display local notifications. Just like remote notifications, the user can grant or remove permissions at any time.

The only difference when requesting permissions locally is that you do not call the `registerForRemoteNotifications` method on success:

```
func registerForLocalNotifications(application: UIApplication) {  
    let center = UNUserNotificationCenter.current()  
    center.requestAuthorization(  
        options: [.badge, .sound, .alert]) {  
        [weak center, weak self] granted, _ in  
        guard granted, let center = center, let `self` = self  
        else { return }  
  
        // Take action here  
    }  
}
```

Note: Since the user may revoke permissions at any time, view controllers creating a local notification must check for permission in `viewDidAppear`.

Objects versus payloads

The primary difference between remote and local notifications is how they are triggered. You've seen that remote notifications require some type of external service to send a JSON payload through APNs. Local notifications use all the same type of data that you provide in a JSON payload but they instead use class objects to define what is delivered to the user.

Creating a trigger

Local notifications utilize what is referred to as a **trigger**, which is the condition under which the notification will be delivered to the user. There are three possible triggers, each corresponding to one of the notification types:

1. `UNCalendarNotificationTrigger`
2. `UNTimeIntervalNotificationTrigger`
3. `UNLocationNotificationTrigger`

All three triggers contain a `repeats` property, which allows you to have the trigger fire more than once.

UNCalendarNotificationTrigger

Not surprisingly, this trigger occurs at specific points in time. While you might assume that you'd be using a Date to specify when the trigger goes off, you'll actually use DateComponents. A Date distinctly specifies one specific point in time, which isn't always helpful for a trigger.

If you're using a calendar trigger, it's more likely that you only have *parts* of a date.

For example, you might want to trigger at 8:30 in the morning, or just on a Monday. Using DateComponents lets you specify as much of the requirements as necessary without being too explicit about the rest.

To have an alarm go off every Monday at 8:30 a.m., you'd write code like this:

```
let components = DateComponents(hour: 8, minute: 30, weekday: 2)
let trigger = UNCalendarNotificationTrigger(dateMatching: components,
                                            repeats: true)
```

UNTimeIntervalNotificationTrigger

This trigger is perfect for timers. You might want to display a notification after 10 minutes, rather than at a specific time. You just tell iOS how many seconds in the future the notification should be delivered. If you need the trigger to happen at a *specific* time, like 2 p.m., you should be using the UNCalendarNotificationTrigger instead to avoid numerous time zone issues related to dates.

In this example, after ordering food from an online service, you'll want to let the end user know to head out in 10 minutes to pick it up:

```
let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 10 * 60,
                                                repeats: false)
```

UNLocationNotificationTrigger

If you're a fan of geocaching, this one's for you! Utilizing this trigger allows you to specify a CLCircularRegion that you wish to monitor. When the device enters said area, the notification will fire. You need to know the latitude and longitude of the center of your target location as well as the radius that should be used. Those three items define a circular region on the map, which iOS will monitor for entry.

Note: You must have authorization to use **Core Location** and must have permission to monitor the user's location while they're using the app. You do not need to request to always have permission as just regions are being monitored.

You'll also need to let iOS know whether you care if the user is entering, exiting or both.

Please see “Core Location Tutorial for iOS: Tracking Visited Locations” (<https://bit.ly/2MLc1GG>) for more information on Core Location, privacy concerns and requesting permissions if you’re not already familiar with that framework.

If, for example, you’d like to schedule a notification whenever the user enters a 1 mile radius around 1 Infinite Loop, Cupertino, California, you’d use code similar to the following:

```
let oneMile = Measurement(value: 1, unit: UnitLength.miles)
let radius = oneMile.converted(to: .meters).value
let coordinate = CLLocationCoordinate2D(latitude: 37.33182,
                                         longitude: -122.03118)
let region = CLCircularRegion(center: coordinate,
                               radius: radius,
                               identifier: UUID().uuidString)

region.notifyOnExit = false
region.notifyOnEntry = true

let trigger = UNLocationNotificationTrigger(region: region,
                                             repeats: false)
```

Defining content

Excellent; you now know *when* the trigger is going to go off. It’s time to tell iOS *what* should be presented in the notification. This is where the `UNMutableNotificationContent` class comes into play. Be sure to note the “`Mutable`” in that class’s name. There’s also a class called `UNNotificationContent`, which you won’t use here or you’ll end up with compiler errors.

You can think of this class as the equivalent of the JSON payload used in remote notifications. The elements from the `aps` dictionary exist as properties right on the object. For your custom content, you simply add that to the `userInfo` dictionary.

If you worked through Chapter 12, “Putting It All Together,” then you’ll remember working with a payload like so:

```
{ "aps" : {
    "alert" : {
        "title" : "New Calendar Invitation"
    },
    "badge" : 1,
    "mutable-content" : 1,
    "category" : "CalendarInvite"
},
"title" : "Family Reunion",
```

```
"start" : "2018-04-10T08:00:00-08:00",
"end" : "2018-04-10T12:00:00-08:00",
"id" : 12
}
```

You'd exactly mimic that same data with a local notification using the following code:

```
let content = UNMutableNotificationContent()
content.title = "New Calendar Invitation"
content.badge = 1
content.categoryIdentifier = "CalendarInvite"
content.userInfo = [
    "title": "Family Reunion",
    "start": "2018-04-10T08:00:00-08:00",
    "end": "2018-04-10T12:00:00-08:00",
    "id": 12
]
```

Notice how everything outside of your `aps` dictionary, meaning - your custom content, falls under the `userInfo` dictionary.

Sounds

If you'd like your notification to play a sound when it's delivered, you must either store the file in your app's main bundle, or you must download it and store it in the **Library/Sounds** subdirectory of your app's container directory. Generally, you'll just want to use the default sound:

```
content.sound = UNNotificationSound.default()
```

Please refer back to Chapter 3, “Remote Notification Payload,” for full details on the requirements around playing sounds and which formats are supported.

Localization

There's one small “gotcha” when working with localization and local notifications. Consider the case wherein the user's device is set to English, and you set the content to a localized value. Then, you create a trigger to fire in three hours. An hour from then, the user switches their device back to Arabic. Suddenly, you're showing the wrong language!

The solution to the above problem is to not use the normal `NSLocalizedString` methods. Instead, you should use `localizedUserNotificationString(forKey:arguments:)` from `NSString`. The difference is that the latter method delays loading the localized string until the notification is actually delivered, thus ensuring the localization is correct.

Note: Always use `localizedUserNotificationString(forKey:arguments:)` when localizing local notifications.

Scheduling

Now that you've defined *when* the notification should occur and *what* to display, you simply need to ask iOS to take care of it for you:

```
let identifier = UUID().uuidString
let request = UNNotificationRequest(identifier: identifier,
                                      content: content,
                                      trigger: trigger)

UNUserNotificationCenter.current().add(request) { error in
    if let error = error {
        // Handle unfortunate error if one occurs.
    }
}
```

Each request needs to have a *unique* identifier so that you can refer to it later on if you wish to cancel the notification before it's actually fired. A UUID is unique by definition, so it's a great choice to use.

Foreground notifications

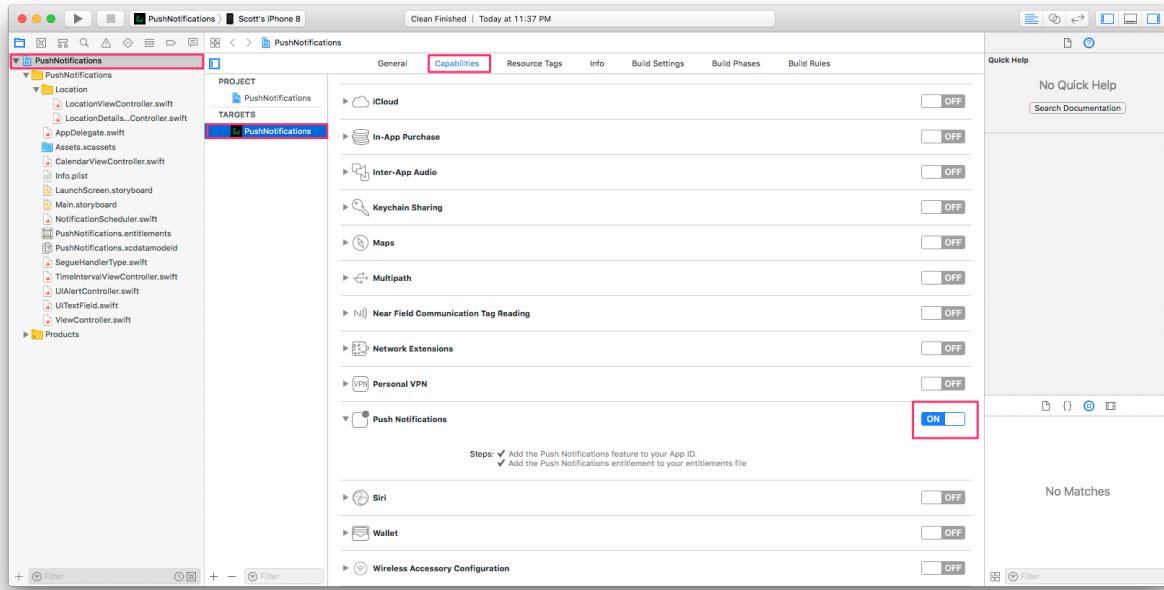
Just like with remote notifications, you'll need to take an extra step to allow local notifications to be displayed when the app is running in the foreground. It's the exact same code that remote notifications use. Simply conform your `AppDelegate` to `UNUserNotificationCenterDelegate` and implement the following method:

```
func userNotificationCenter(
    _ center: UNUserNotificationCenter,
    willPresent notification: UNNotification,
    withCompletionHandler completionHandler:
        @escaping (UNNotificationPresentationOptions) -> Void) {
    completionHandler([.badge, .sound, .alert])
}
```

You can, of course, take other actions here such as updating the user interface directly based on what the notification is for!

The sample platter

That seems like quite enough reference material. Time to write some code! Please open up the starter project and enable push notifications as described in Chapter 4, “Xcode Project Set Up,” and set your team ID as discussed in Chapter 7, “Expanding the Application.”



You'll notice that there's an awful lot of code in the starter project, but don't let that scare you. The intent of this chapter is for you to learn about local notifications, not have you spend a ton of time building an iOS app to handle all three types of local notifications.

If you build the app right now, you'll get four warnings from Xcode about values being defined but never used. Don't worry as those will all go away as you build out the app.

The general goal for this app is to allow the user to pick one of the three types of notifications, configure it and then see on the main view whether or not it's been delivered. The user will also be able to cancel any notifications that are still pending to be delivered.

Configuring the main UITableView

Just like with remote notifications, the first task you'll need to take care of is getting your user's permission to send them notifications.

Open up **ViewController.swift** and take care of that in `viewDidAppear(_:)`, by adding the following code to the method:

```
center.requestAuthorization(
    options: [.alert,.sound,.badge],
    completionHandler: { [weak self] (granted, error) in
        guard let `self` = self else { return }

        if granted {
            self.refreshNotificationList()
            self.center.delegate = self
        }

        self.addButton.isEnabled = granted
        self.refreshButton.isEnabled = granted
    })
}
```

You'll want to be able to show the notifications that have already been delivered as well as those that are still pending. As the state of notifications will change, you'll also want the end user to be able to update that list. In the `refreshNotificationList` method, add the following code to ask iOS to tell you about all of the notifications that are still pending:

```
center.getPendingNotificationRequests { [weak self] requests in
    guard let `self` = self else { return }

    self.pending = requests
    DispatchQueue.main.async {
        self.tableView.reloadData()
    }
}
```

Note: You must dispatch to the main queue to make any UI changes as the completion handler is *not* guaranteed to run on the main thread.

You're simply storing the current list of notifications that are queued, but not yet delivered, and then asking the table to reload its content. You should also grab the notifications that are already delivered. Add the following to the bottom of the method:

```
center.getDeliveredNotifications { [weak self] requests in
    guard let `self` = self else { return }

    self.delivered = requests
    DispatchQueue.main.async {
```

```
        self.tableView.reloadData()  
    }  
}
```

Note: Once a user deletes a notification from the Notification Center on their device, it will no longer appear in the list of delivered notifications.

While the code for both pending and delivered notifications looks almost exactly the same, take note of the fact that pending notifications are of type `UNNotificationRequest`, whereas delivered notifications are `UNNotification`. The `UNNotification` has a `request` property that lets you get at the `UNNotificationRequest` details.

There's just one more modification that you'll need to make in this file: allowing notifications to be removed.

There are two very similar methods on `UNNotificationCenter` to handle that. Look in the `tableView(_:commit:forRowAt:)` method and you'll see most of it has been stubbed out for you. This method will get called if the user swipes to delete the table view cell. You'll handle cancelling a pending request in the code block for section zero. Just after the `let request = pending[indexPath.row]` is where you'll want to add the following code to cancel the notification:

```
let identifiers = [request.identifier]  
center.removePendingNotificationRequests(  
    withIdentifiers: identifiers)
```

As discussed earlier, each notification is created with a unique identifier so that you can cancel it if necessary.

The delivered notifications in the first section are handled basically the same way. Call just a slightly different method on `UNNotificationCenter`, and you'll notice that to get the request you'll need to access the `request` property discussed earlier. Add the following code inside the `else` block, just below the `request` declaration:

```
let identifiers = [request.identifier]  
center.removeDeliveredNotifications(  
    withIdentifiers: identifiers)
```

You don't need to remove anything from the pending or delivered arrays, nor do you need to remove rows from the table as the `refreshNotificationList` method will handle that for you.

Scheduling

While there are more options available on the content of a notification, for the sample app, you'll only be using the `title`, `sound` and `badge` properties of the `UNMutableNotificationContent`. As a good programmer, you always follow the KISS principle, right?

Open up `NotificationScheduler.swift`. Multiple great features of Swift are used here. Instead of sub-classing to use a common method, a protocol is used to describe the required method for scheduling notifications. Then, a default implementation is provided via the extension shown. Finally, you'll notice there is a condition on the extension so that it only applies to classes that inherit from `UIViewController`. The `where` clause is necessary, as you'll need to access the `navigationController` that all `UIViewControllers` have.

Creating content

You'll create the content in `scheduleNotification`. You've already been passed the appropriate `UNNotificationTrigger`, so now you'll need to generate the content that goes with it. Add the following code to the bottom of the method:

```
let content = UNMutableNotificationContent()
content.title = title

if sound {
    content.sound = UNNotificationSound.default()
}

if let badge = badge, let number = Int(badge) {
    content.badge = NSNumber(value: number)
}
```

Pretty straightforward, right?

Adding the request

Now that the content and trigger are in place, all that's left to do is create the request and hand it off to `UNUserNotificationCenter`. You've already seen the code for this, so it shouldn't be anything too shocking. Add the following to the end of the method:

```
let identifier = UUID().uuidString
let request = UNNotificationRequest(identifier: identifier,
                                    content: content,
                                    trigger: trigger)

UNUserNotificationCenter.current().add(request) {
    [weak self] error in
```

```
guard let `self` = self else { return }

if let error = error {
    DispatchQueue.main.async {
        let message = """
            Failed to schedule notification.
            \(error.localizedDescription)
        """
        UIAlertController.okWithMessage(message,
            presentingViewController: self)
    }
} else {
    DispatchQueue.main.async {
        self.navigationController?.popToRootViewController(
            animated: true)
    }
}
```

While the request has to have a unique identifier, you don't really have a need to know what it is, so using a UUID is a great choice here. If the request wasn't successfully added to the list of pending local notifications, then you'll warn the user about that using another provided helper method from **Extensions/UIAlertController+Ext.swift**. If it was added, which should always be the case with valid content, then you pop everything off of the UI's navigation stack and take the user back to the main table view.

Note: You must dispatch to the main queue to make any UI changes as the completion handler is *not* guaranteed to run on the main thread.

If you're not using extensions in Swift yet, then hopefully the two that were provided for you have helped show the great power they provide, as well as the cleanliness that they add to the rest of your code!

Time interval notifications

You're almost ready to run the app and see something! The first location notification trigger to implement is the `UNTimeIntervalNotificationTrigger`. With the extension you just created, you'll only need two lines of code now to set up a time-interval trigger. Open `TimeIntervalViewController.swift` and take a look at the `doneButtonTouched` method. Once the number of seconds to wait is known, you need to create the trigger, just like you learned about a few pages back. Add this code to the end of the method:

```
let trigger = UNTimeIntervalNotificationTrigger(
    timeInterval: interval,
    repeats: repeats.isOn)

scheduleNotification()
```

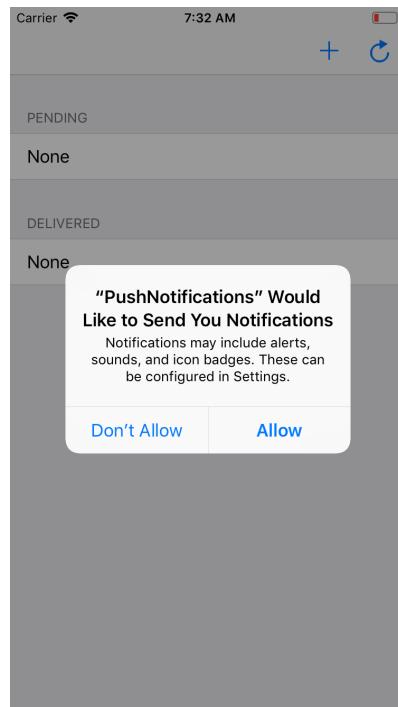
```
trigger: trigger,
titleTextField: notificationTitle,
sound: sound.isOn,
badge: badge.text)
```

As you'll need to repetitively get the integer value from a `UITextField`, the starter project provides a few helper methods for that in `Extensions/UITextField+Ext.swift`.

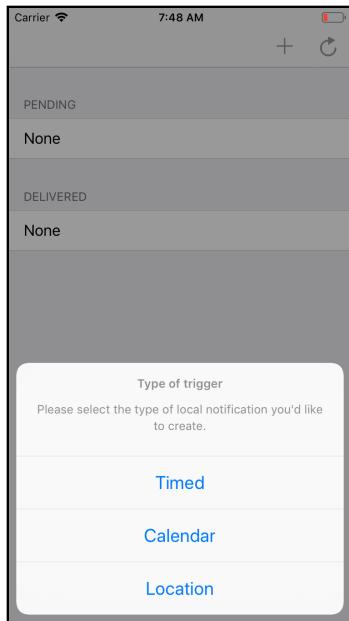
The values are all taken from the UI controls and then handed over to your extension. The method is kept nice and clean, and it's immediately obvious what is happening when the **Done** button is pressed.

It's finally time to try things out! Build and run the app. You should have no errors and only one warning left. As expected, you're asked right away to grant permissions. Say yes...you know you want to.

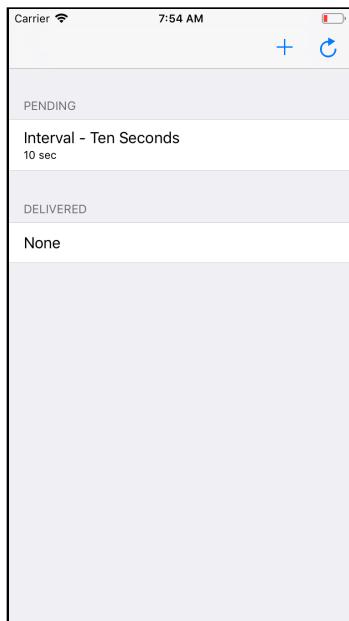
Note: When working on your local notification, as opposed to remote notifications, you don't need a physical device when testing your notification code, and may simply use your Simulator! Super helpful.



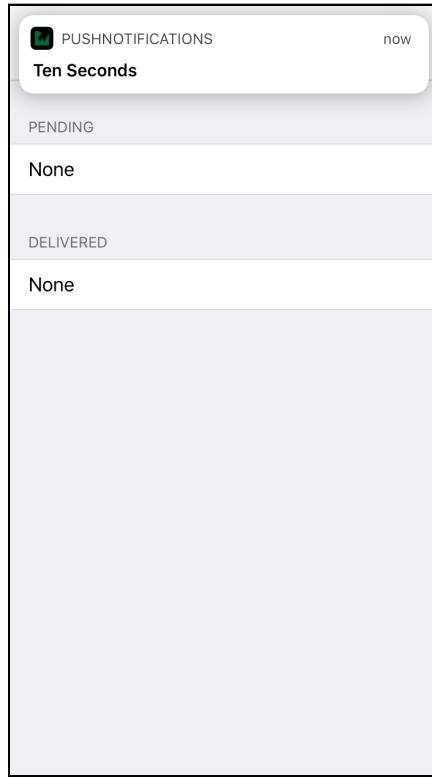
Tap the + button in the navigation bar and choose to add a timed trigger.



You'll be presented with a simple screen where you can specify how many seconds in the future the notification should trigger. While you must specify a title, the badge is optional. If you include a numeric value, then the app icon will be badged appropriately. If you specify a 10-second wait period and tap the **Done** button, you'll be returned to the home screen with a view like the following:



And, then, 10 seconds later you'll be notified!



Beautiful work! Moving on!

Location notifications

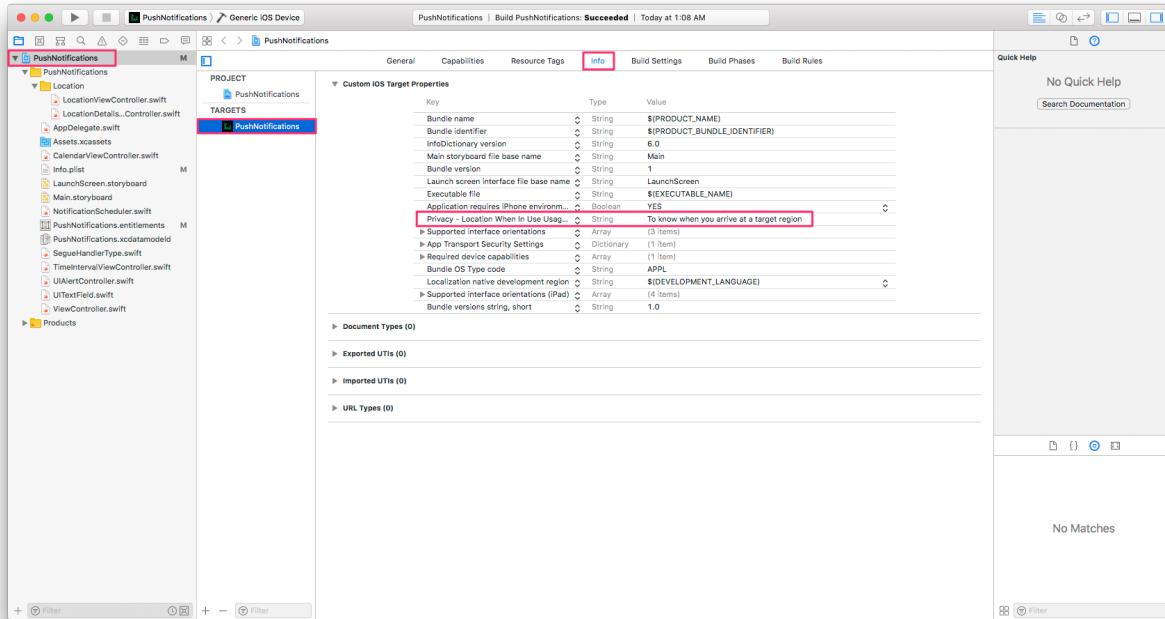
Handling locations takes just a bit more work.

Location permissions

In order to get the location to trigger, you need to know the user's location. This means you first need to ask the user's permission to access their location.

Click on the **PushNotifications** project, and then the **PushNotification** target's Info tab. Add the privacy key for access to the user's location. The key's name is "**Privacy - Location When In Use Usage Description**", or you may simply paste in the `NSLocationWhenInUseUsageDescription` key.

Set its value to the string: “To know when you arrive at the target region.”



Now, you’ll need to ensure that you ask for those permissions in **LocationViewController.swift** when the view appears. Replace the `viewDidAppear(_:`)

 method with the following code:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    address.isEnabled = false
    doneButton.isEnabled = false

    switch CLLocationManager.authorizationStatus() {
    case .notDetermined:
        locationManager.requestWhenInUseAuthorization()

    case .restricted:
        let message =
            "This device is not allowed to use location services."
        UIAlertController.okWithMessage(message,
                                        presentingViewController: self)

    case .denied:
        let message = "Location services must be enabled."
        UIAlertController.okWithMessage(
            message,
            presentingViewController: self)

    case .authorizedWhenInUse:
        address.becomeFirstResponder()
        address.isEnabled = true
    }
}
```

```
doneButton.isEnabled = true

default:
    break
}

locationManager.startUpdatingLocation()
}
```

If permissions are not granted, then you don't want the users to be able to interact with the **Address** or **Done** buttons. When they do have authorization, however, you can re-enable them and bring up the keyboard for the address.

Now, you can start working on the actual trigger. In **Location/LocationDetailsViewController.swift** you'll again edit the `doneButtonTouched` action to create the trigger. Add to the end of the method:

```
let region = CLCircularRegion(
    center: coordinate,
    radius: distance,
    identifier: UUID().uuidString)
region.notifyOnExit = notifyOnExit.isOn
region.notifyOnEntry = notifyOnEntry.isOn

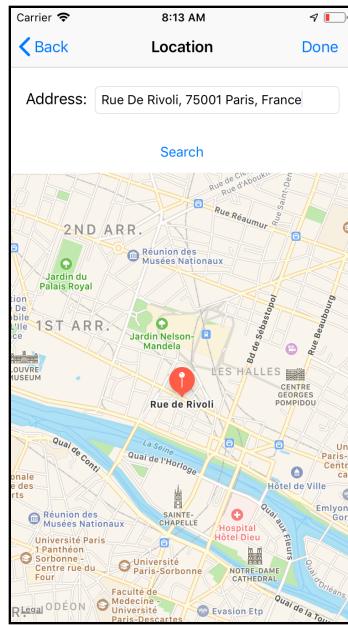
let trigger = UNLocationNotificationTrigger(
    region: region,
    repeats: repeats.isOn)

scheduleNotification(
    trigger: trigger,
    titleTextField: notificationTitle,
    sound: sound.isOn,
    badge: badge.text)
```

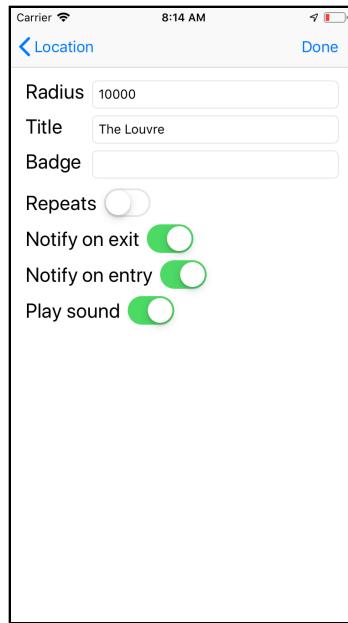
Similar to a timed notification, you're simply pulling values from the UI and then creating the trigger. Scheduling is handled by your extension method just like before.

Build and run a second time but, this time around, after tapping the **+** button, choose a location notification. The first screen you see allows you to specify an address and see a view of it on the map. Enter any address you like and tap the **Search** button. If you gave a valid address, you should see your destination.

I don't know about you, but I'm headed to the Louvre in Paris!



After you've entered a valid address, tap on the **Done** button in the navigation bar and you'll see another beautiful data entry screen.



Location notifications are based on a circular radius, so you'll have to specify how many *meters* you'd like to use and provide a title. The badge is again optional but, this time, you can also identify if you want a notification when you enter the area, leave the area or both.

After tapping **Done**, you should see your trigger in the Pending section. To complete this chapter, you'll have to book a flight to Paris and head over to the Louvre.

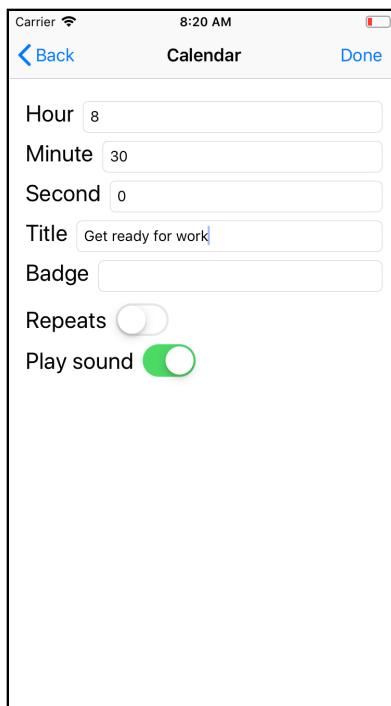
Calendar notifications

Just one notification to go! Calendar-based local notifications, as discussed earlier, use a `DateComponents` struct to specify exactly when the notification will trigger. If you've worked with `DateComponents` before, you know how many different properties are available to you. For the sample app, to keep things simple, you're just using hours, minutes and seconds.

In `CalendarViewController.swift`, you'll see that the `doneButtonTouched` method has pulled out the details of the time for you already. All you've got to do is create the trigger to fire at the right time. Add the following code at the end of the method:

```
let trigger = UNCalendarNotificationTrigger(  
    dateMatching: components,  
    repeats: repeats.isOn)  
  
scheduleNotification(  
    trigger: trigger,  
    titleTextField: notificationTitle,  
    sound: sound.isOn,  
    badge: badge.text)
```

Build and run your app one final time, and you'll be able to schedule a calendar-based local notification.



You've managed to utilize all the local notification types in a sample app. Hopefully, you've seen how easy the notification-related code is to implement.

Key points

- While most push notifications displayed on your device are remote notifications, it's also possible to display notifications originating from the user's device, locally.
- Local notifications are less frequently used but they still merit your understanding. There may be times when a user needs a notification (like a reminder) free of any action being taken.
- **Calendar** notifications occur on a specific date.
- **Interval** notifications occur after a specific amount of time.
- **Location** notifications occur when entering a specific area.
- Even though the notification is created and delivered locally, on the user's device, you must still obtain permission to display notifications.

Where to go from here?

In your own apps, you'll likely want to explore other concepts such as custom sounds, more options around calendar selection, and even custom actions and user interfaces. Refer back to each of the following chapters for information on how to add each feature to your local notifications.

- Chapter 9, "Custom Actions."
- Chapter 10, "Modifying the Payload."
- Chapter 11, "Custom Interfaces."



Conclusion

We hope you're excited about the knowledge that you've gained throughout this book, learning how to create highly professional push notifications with the latest features and abilities.

As you've learned in this book, push notifications are conceptually simple. But what eventually makes an app stand out in the crowd comes down to all of the advanced features and abilities you've mastered in this book: rich notification with custom UI, location and time-based notifications, custom actions, grouped notifications and much more!

You've also briefly touched on how to create your very own server and what are the customization options you're faced with when building and delivering your own payload.

We've spent an enormous amount of time, love and effort to make this book the best and most accessible resource for iOS push notifications. If you have any questions, comments or suggestions, please stop by our forums at <https://forums.raywenderlich.com>.

Thank you, again, for purchasing this book. Your continued support is what makes the books, tutorials, videos and other things we do at raywenderlich.com possible. We truly appreciate it!

– Scott, Marin, Shai and Manda

More Books You Might Enjoy

We hope you enjoyed this book! If you're looking for more, we have a whole library of books waiting for you at <https://store.raywenderlich.com>.

New to iOS or Swift?

Learn how to develop iOS apps in Swift with our classic, beginner editions.

iOS Apprentice

<https://store.raywenderlich.com/products/ios-apprentice>



The iOS Apprentice is a series of epic-length tutorials for beginners where you'll learn how to build 4 complete apps from scratch.

Each new app will be a little more advanced than the one before, and together they cover everything you need to know to make your own apps. By the end of the series you'll be experienced enough to turn your ideas into real apps that you can sell on the App Store.

These tutorials have easy to follow step-by-step instructions, and consist of more than 900 pages and 500 illustrations! You also get full source code, image files, and other resources you can re-use for your own projects.

Swift Apprentice

<https://store.raywenderlich.com/products/swift-apprentice>



This is a book for complete beginners to Apple's brand new programming language — Swift 4.

Everything can be done in a playground, so you can stay focused on the core Swift 4 language concepts like classes, protocols, and generics.

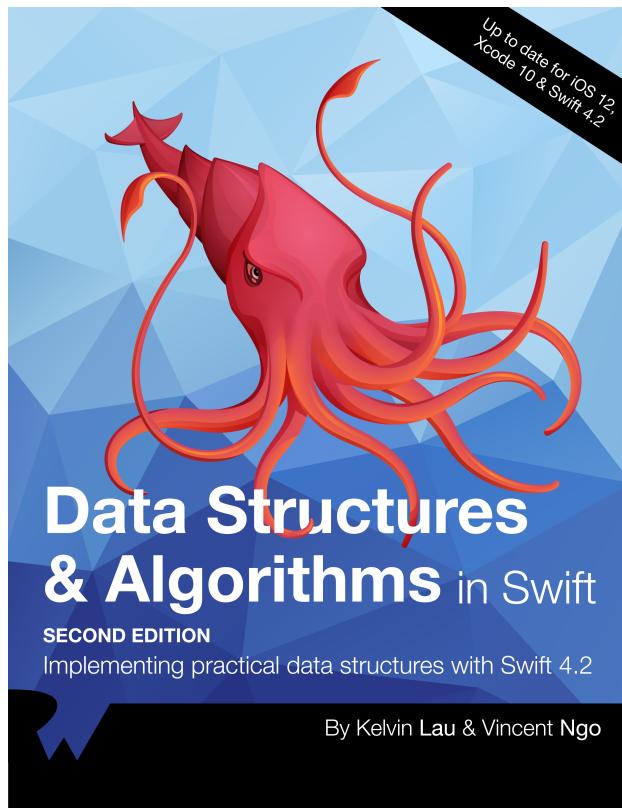
This is a sister book to the iOS Apprentice; the iOS Apprentice focuses on making apps, while Swift Apprentice focuses on the Swift 4 language itself.

Experienced iOS developer?

Level up your development skills with a deep dive into our many intermediate to advanced editions.

Data Structures and Algorithms in Swift

<https://store.raywenderlich.com/products/data-structures-and-algorithms-in-swift>



Understanding how data structures and algorithms work in code is crucial for creating efficient and scalable apps. Swift's Standard Library has a small set of general purpose collection types, yet they definitely don't cover every case!

In Data Structures and Algorithms in Swift, you'll learn how to implement the most popular and useful data structures, and when and why you should use one particular datastructure or algorithm over another. This set of basic data structures and algorithms will serve as an excellent foundation for building more complex and special-purpose constructs. As well, the high-level expressiveness of Swift makes it an ideal choice for learning these core concepts without sacrificing performance.

Realm: Building Modern Swift Apps with Realm Database

<https://store.raywenderlich.com/products/realm-building-modern-swift-apps-with-realm-database>



Realm Platform is a relatively new commercial product which allows developers to automatically synchronize data not only across Apple devices but also between any combination of Android, iPhone, Windows, or macOS apps. Realm Platform allows you to run the server software on your own infrastructure and keep your data in-house which more often suits large enterprises. Alternatively you can use Realm Cloud which runs a Platform for you and you start syncing data very quickly and only pay for what you use.

In this book, you'll take a deep dive into the Realm Database, learn how to set up your first Realm database, see how to persist and read data, find out how to perform migrations and more. In the last chapter of this book, you'll take a look at the synchronization features of Realm Cloud to perform real-time sync of your data across all devices.

Design Patterns by Tutorials

<https://store.raywenderlich.com/products/design-patterns-by-tutorials>



Design patterns are incredibly useful, no matter what language or platform you develop for. Using the right pattern for the right job can save you time, create less maintenance work for your team and ultimately let you create more great things with less effort. Every developer should absolutely know about design patterns, and how and when to apply them. That's what you're going to learn in this book!

Move from the basic building blocks of patterns such as MVC, Delegate and Strategy, into more advanced patterns such as the Factory, Prototype and Multicast Delegate pattern, and finish off with some less-common but still incredibly useful patterns including Flyweight, Command and Chain of Responsibility.

Server Side Swift with Vapor

<https://store.raywenderlich.com/products/server-side-swift-with-vapor>



If you're a beginner to web development, but have worked with Swift for some time, you'll find it's easy to create robust, fully-featured web apps and web APIs with Vapor 3.

Whether you're looking to create a backend for your iOS app, or want to create fully-featured web apps, Vapor is the perfect platform for you.

This book starts with the basics of web development and introduces the basics of Vapor; it then walks you through creating APIs and web backends; creating and configuring databases; deploying to Heroku, AWS, or Docker; testing your creations and more!

iOS 11 by Tutorials

<https://store.raywenderlich.com/products/ios-11-by-tutorials>

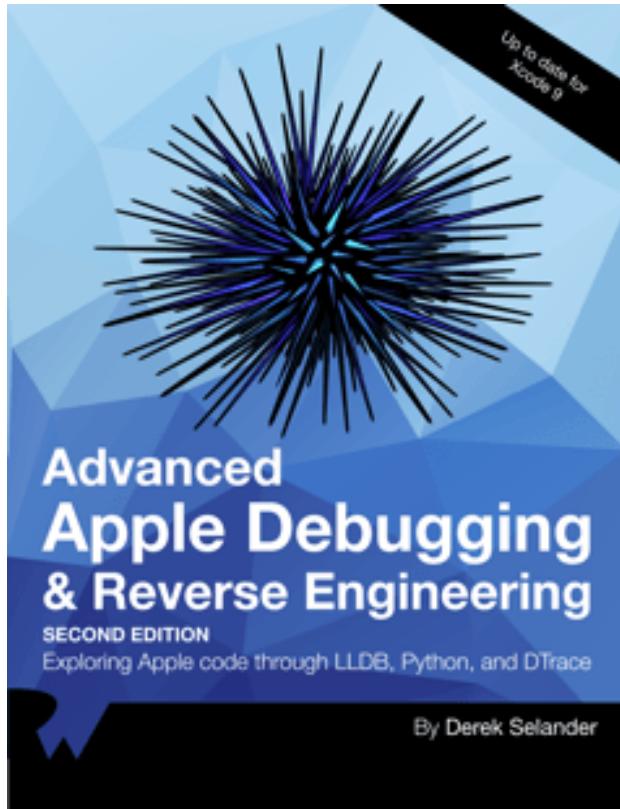


This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn the new APIs introduced in iOS 11.

Discover the new features for developers in iOS 11, such as ARKit, Core ML, Vision, drag & drop, document browsing, the new changes in Xcode 9 and Swift 4 — and much, much more.

Advanced Debugging and Reverse Engineering

<https://store.raywenderlich.com/products/advanced-apple-debugging-and-reverse-engineering>



In Advanced Apple Debugging and Reverse Engineering, you'll come to realize debugging is an enjoyable process to help you better understand software. Not only will you learn to find bugs faster, but you'll also learn how other developers have solved problems similar to yours.

You'll also learn how to create custom, powerful debugging scripts that will help you quickly find the secrets behind any bit of code that piques your interest.

After reading this book, you'll have the tools and knowledge to answer even the most obscure question about your code — or someone else's.

RxSwift: Reactive Programming with Swift

<https://store.raywenderlich.com/products/rxswift>

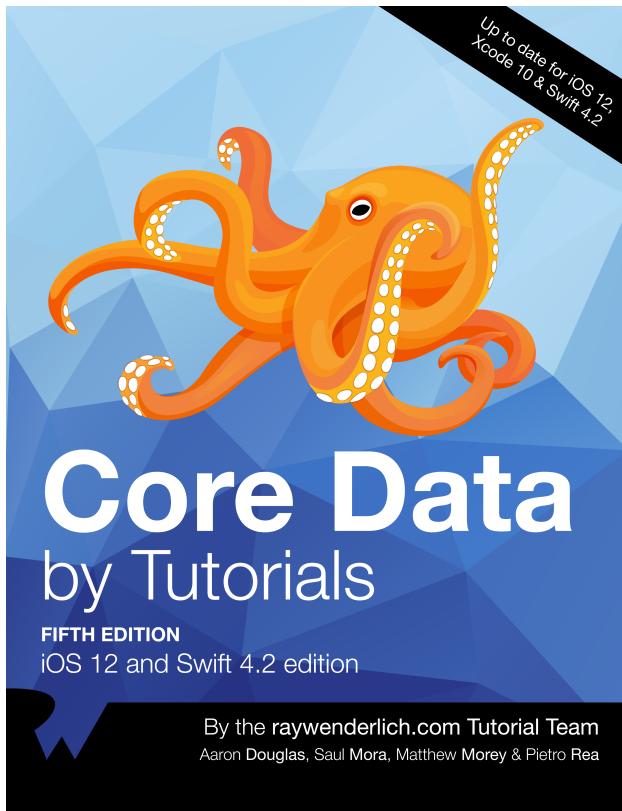


This book is for iOS developers who already feel comfortable with iOS and Swift, and want to dive deep into development with RxSwift.

Start with an introduction to the reactive programming paradigm; learn about observers and observables, filtering and transforming operators, and how to work with the UI, and finish off by building a fully-featured app in RxSwift.

Core Data by Tutorials

<https://store.raywenderlich.com/products/core-data-by-tutorials>

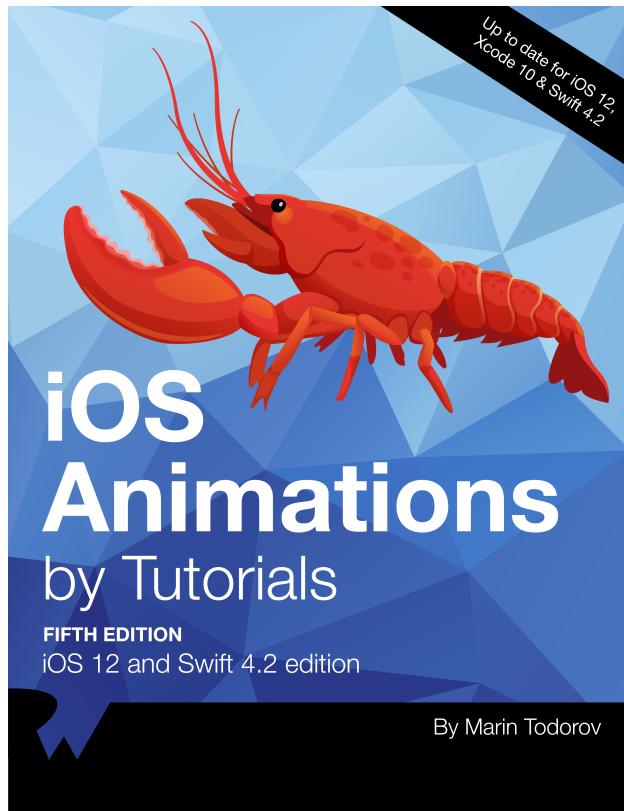


This book is for intermediate iOS developers who already know the basics of iOS and Swift 4 development but want to learn how to use Core Data to save data in their apps.

Start with the basics like setting up your own Core Data Stack all the way to advanced topics like migration, performance, multithreading, and more!

iOS Animations by Tutorials

<https://store.raywenderlich.com/products/ios-animations-by-tutorials>



This book is for iOS developers who already know the basics of iOS and Swift 4, and want to dive deep into animations.

Start with basic view animations and move all the way to layer animations, animating constraints, view controller transitions, and more!

ARKit by Tutorials

<https://store.raywenderlich.com/products/arkit-by-tutorials>



Learn how to use Apple's augmented reality framework, ARKit, to build five great-looking AR apps:

- Tabletop Poker Dice
- Immersive Sci-Fi Portal
- 3D Face Masking
- Location-Based Content
- Monster Truck Sim

watchOS by Tutorials

<https://store.raywenderlich.com/products/watchos-by-tutorials>



This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn how to make Apple Watch apps for watchOS 4.

tvOS Apprentice

<https://store.raywenderlich.com/products/tvos-apprentice>



This book is for complete beginners to tvOS development. No prior iOS or web development knowledge is necessary, however the book does assume at least a rudimentary knowledge of Swift.

This book teaches you how to make tvOS apps in two different ways: via the traditional method using UIKit, and via the new Client-Server method using TVML.

Metal by Tutorials

<https://store.raywenderlich.com/products/metal-by-tutorials>



This book will introduce you to graphics programming in Metal — Apple’s framework for programming on the GPU. You’ll build your own game engine in Metal where you can create 3D scenes and build your own 3D games.

Want to make games?

Learn how to make great-looking games that are deeply engaging and fun to play!

2D Apple Games by Tutorials

<https://store.raywenderlich.com/products/2d-apple-games-by-tutorials>



In this book, you will make 6 complete and polished mini-games, from an action game to a puzzle game to a classic platformer!

This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SpriteKit, you will learn a lot from this book!

3D Apple Games by Tutorials

<https://store.raywenderlich.com/products/3d-apple-games-by-tutorials>

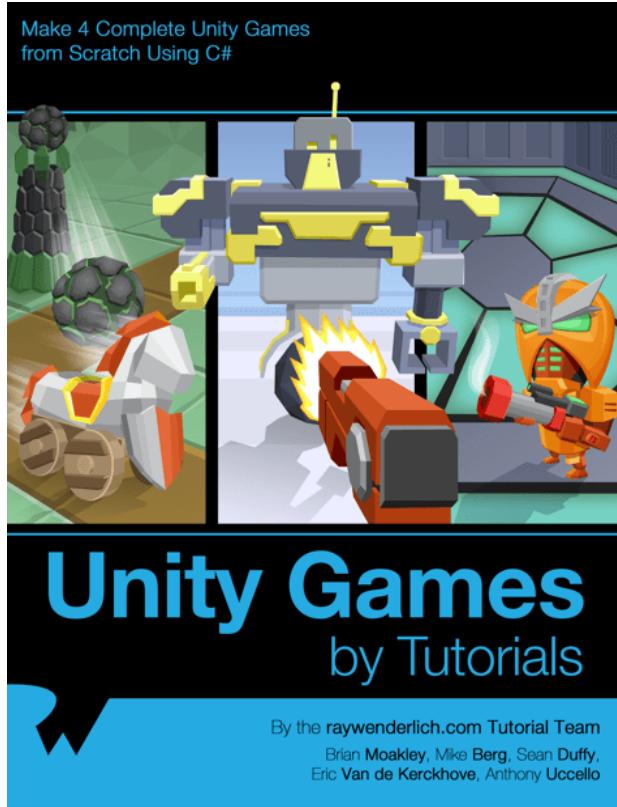


Through a series of mini-games and challenges, you will go from beginner to advanced and learn everything you need to make your own 3D game!

This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SceneKit, you will learn a lot from this book!

Unity Games by Tutorials

<https://store.raywenderlich.com/products/unity-games-by-tutorials>

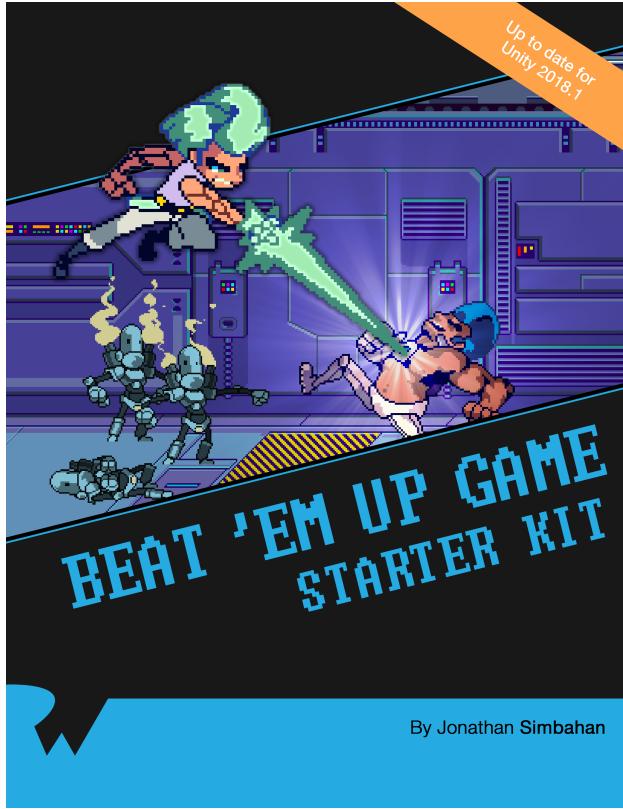


Through a series of mini-games and challenges, you will go from beginner to advanced and learn everything you need to make your own 3D game!

This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SceneKit, you will learn a lot from this book!

Beat 'Em Up Game Starter Kit - Unity

<https://store.raywenderlich.com/products/beat-em-up-game-starter-kit-unity>



The classic beat 'em up starter kit is back — for Unity!

Create your own side-scrolling beat 'em up game in the style of such arcade classics as Double Dragon, Teenage Mutant Ninja Turtles, Golden Axe and Streets of Rage.

This starter kit equips you with all tools, art and instructions you'll need to create your own addictive mobile game for Android and iOS.

Want to learn Android or Kotlin?

Get a head start on learning to develop great Android apps in Kotlin, the newest first-class language for building Android apps.

Android Apprentice

<https://store.raywenderlich.com/products/android-apprentice>



If you're completely new to Android or developing in Kotlin, this is the book for you!

The *Android Apprentice* takes you all the way from building your first app, to submitting your app for sale. By the end of this book, you'll be experienced enough to turn your vague ideas into real apps that you can release on the Google Play Store.

You'll build 4 complete apps from scratch — each app is a little more complicated than the previous one. Together, these apps will teach you how to work with the most common controls and APIs used by Android developers around the world.

Kotlin Apprentice

<https://store.raywenderlich.com/products/kotlin-apprentice>



This is a book for complete beginners to the new, modern Kotlin language.

Everything in the book takes place in a clean, modern development environment, which means you can focus on the core features of programming in the Kotlin language, without getting bogged down in the many details of building apps.

This is a sister book to the Android Apprentice the Android Apprentice focuses on making apps for Android, while the Kotlin Apprentice focuses on the Kotlin language fundamentals.