First Edition — Xcode 13 · Swift 5.5 · watchOS 8

watchOS With SwiftUI by Tutorials

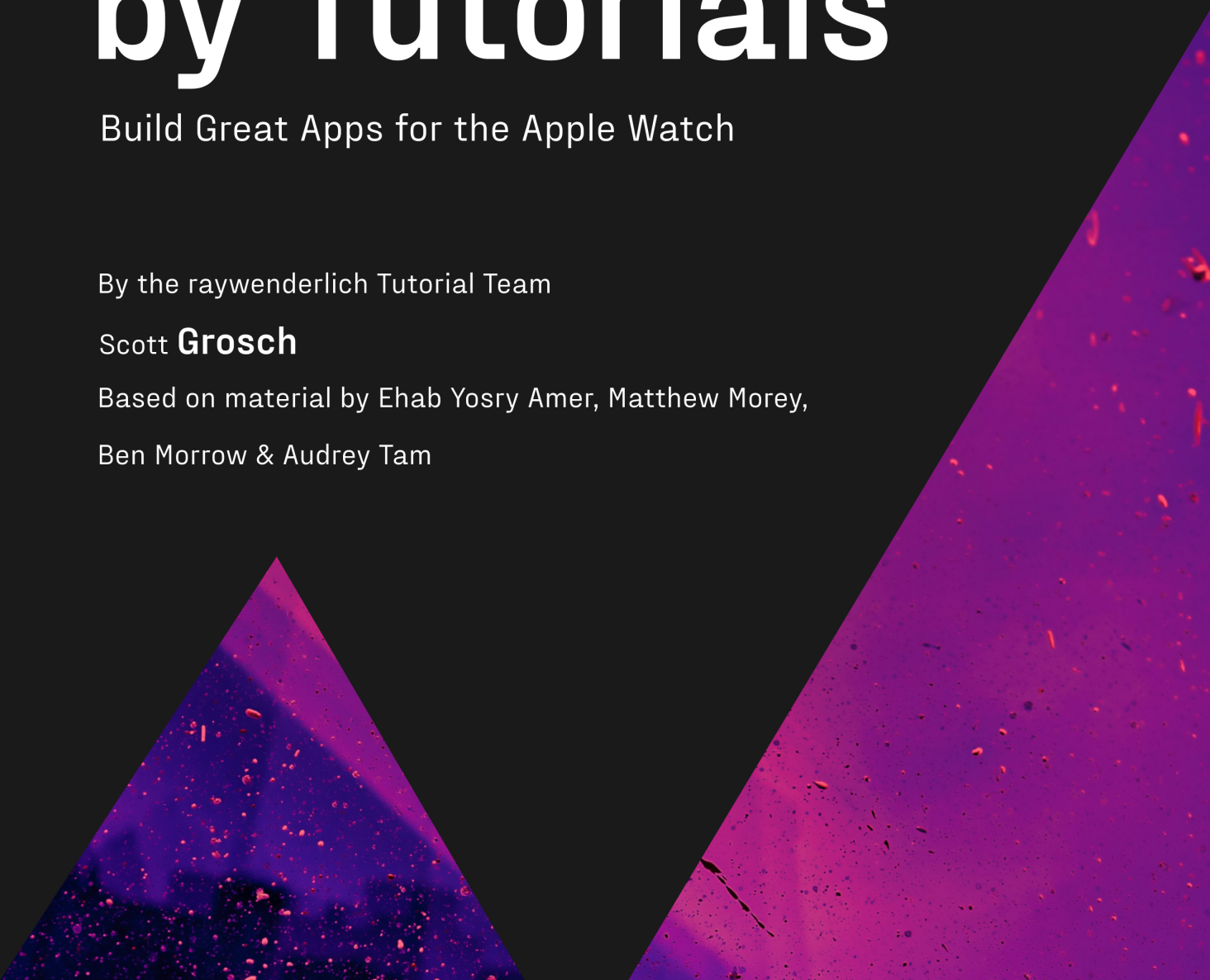
Build Great Apps for the Apple Watch

By the raywenderlich Tutorial Team

Scott **Grosch**

Based on material by Ehab Yosry Amer, Matthew Morey,

Ben Morrow & Audrey Tam



watchOS With SwiftUI by Tutorials

Scott Grosch

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Book License 9

Before You Begin 10

What You Need 11

Book Source Code & Forums 12

Introduction 16

Section I 18

Chapter 1: Hello, Apple Watch 19

Chapter 2: Project Structure 26

Chapter 3: Digital Crown 35

Chapter 4: Watch Connectivity 44

Chapter 5: Snapshots 73

Chapter 6: Notifications 96

Chapter 7: Lifecycle 115

Chapter 8: Introduction to Complications 132

Chapter 9: Complications 151

Chapter 10: Keeping Complications Updated 169

Chapter 11: Tinted Complications 186

Chapter 12: SwiftUI Complications 197

Chapter 13: Face Sharing 212

Chapter 14: Sign in With Apple 228

Chapter 15: HealthKit 238

Conclusion 265

Table of Contents: Extended

Book License 9

Before You Begin 10

What You Need 11

Book Source Code & Forums 12

About the Author 14

About the Editors 14

Introduction 16

How to read this book 17

Section I 18

Chapter 1: Hello, Apple Watch 19

Hello, World! 20

It's just SwiftUI 21

Sprucing things up 22

But... watchOS code? 25

Key points 25

Chapter 2: Project Structure 26

Naming your app 31

App vs. extension folders 33

Adding Apple Watch support later 34

Key points 34

Chapter 3: Digital Crown 35

Pong 40

Key points 43

Chapter 4: Watch Connectivity 44

Device-to-device communication 45

Getting started 48
Setting up watch connectivity 51
Key points 72
Where to go from here? 72

Chapter 5: Snapshots 73
Getting started 74
The Dock 74
Snapshot API 75
Working with snapshots 78
Key points 95
Where to go from here? 95

Chapter 6: Notifications 96
Where did it go? 97
Short looks 98
Long looks 98
Local notifications 99
Remote push notifications 106
Key points 114
Where to go from here? 114

Chapter 7: Lifecycle 115
Common state transitions 116
Always on state 121
State change sample 121
Extended runtime sessions 122
Key points 131
Where to go from here? 131

Chapter 8: Introduction to Complications 132
Why complications? 133
Complication families 135
Complication identifiers 135

Complication templates 135
Tinted complications 149
Key points 150
Where to go from here? 150

Chapter 9: Complications 151
Exploring the sample 151
Complication data source 152
Updating the complication's data 158
Supporting multiple families 160
Freshness 167
Key points 168
Where to go from here? 168

Chapter 10: Keeping Complications Updated 169
Scheduled background tasks 170
Background URL downloads 174
Push notifications 181
Key points 185

Chapter 11: Tinted Complications 186
Full-color 186
Desaturation 189
Layered images 189
SwiftUI complications 191
Key points 196
Where to go from here? 196

Chapter 12: SwiftUI Complications 197
Showing an appointment 198
The event complication 205
Tinting 210
Key points 211
Where to go from here? 211

Chapter 13: Face Sharing. 212
Sharing faces 213
Key points 227
Where to go from here? 227
Chapter 14: Sign in With Apple. 228
To authenticate or not 229
Authenticate via username and password 229
Handling sign in. 230
Signing in with Apple 232
Key points 237
Where to go from here? 237
Chapter 15: HealthKit 238
Adding HealthKit 239
Tracking brushing 243
Tracking water. 247
Reading single day data 254
Reading multiple days of data 260
Key points 264
Conclusion. 265

Book License

By purchasing *watchOS With SwiftUI by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *watchOS With SwiftUI 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 *watchOS With SwiftUI by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *watchOS With SwiftUI by Tutorials*, available at www.raywenderlich.com".
- The source code included in *watchOS With SwiftUI by Tutorials* is for your personal use only. You are NOT allowed to distribute or sell the source code in *watchOS With SwiftUI 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.

Before You Begin

This section tells you a few things you need to know before you get started, such as what you'll need for hardware and software, where to find the project files for this book, and more.

What You Need

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

- A Mac running macOS Monterey (12.0) or later.
- It is strongly recommended to have an Apple Watch with watchOS 8.0 or higher
- Xcode 13.1 or later. Xcode is the main development tool for iOS. You'll need Xcode 13.1 or later to make use of SwiftUI and the latest features explained throughout the book. You can download the latest version of Xcode from Apple's developer site here: apple.co/2asi58y — If you have an Apple Developer account, you can also download any Xcode version here: apple.co/3GWcz96.

Note: The code covered in this book was developed and tested with Swift 5.5, macOS Monterey and Xcode 13.1 — so even though you can work with slightly earlier versions of them, we encourage you to update to those versions to follow along the book without unexpected errors.

Book Source Code & Forums

Where to download the materials for this book

The materials for this book can be cloned or downloaded from the GitHub book materials repository:

- https://github.com/raywenderlich/wos-materials/tree/editions/1.0

Forums

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

"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

About the Author

Scott Grosch 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

Pablo Mateo is the final pass editor for this book. He is Delivery Manager at at one of the biggest Banks in the world, and was also founder and CTO of a Technology Development company in Madrid. His expertise is focused on web and mobile app development, although he first started as a Creative Art Director. He has been for many years the Main Professor of the iOS and Android Mobile Development Masters Degree at a well-known technology school in Madrid (CICE). He has a masters degree in Artificial Intelligence & Machine-Learning and is currently learning Quantum Computing at MIT.

Nick Rogness is a tech editor for this book. He has been writing code professionally since 2004 and an iOS developer since iOS 9. He looks for any reason to get outdoors, running, biking, hiking, camping, whatever it takes.

Piotr Fulmanski is a tech editor for this book. Piotr has been working at the Faculty of Mathematics and Computer Science of the University of Lodz since 2001 where he teaches various computer science subjects. When he is not preparing materials for his students, he runs in the forest or makes toys for his children. He spends his vacation in the mountains with his family.

April Rames is the english language editor for this book. April is a former high school English and theatre teacher and director. When not volunteering at her daughters' schools, she usually spends her time being asked to pretend to be a unicorn, zombie princess or super hero. In her spare time, she enjoys reading, making pasta and exploring the Gulf Coast with her family.

Introduction

Since the Apple Watch came out in 2015, step by step it has been making a dominan position in the market, becoming an indispensable complement for many users. It has proven to be a tremendously useful device with which you can see all your notifications without having to take out the phone, or check valuable information from your favorite apps at a glance. It has even become the perfect companion for those who like practicing all sort of sports.

This book will teach you how to make the most of it. Implementing all the functionalities and capabilities this small device you wear on your wrist provides. From showing notifications to how to use snapshots to provide updated information. Or why complications are so important. Even how to integrate HealthKit into your apps!

As well as many other concepts that we encourage you to discover by reading this book. All this using **SwiftUI**, the declarative framework to build applications designed by Apple. You will see how easy it is to build incredible applications for your Watch.

We hope you enjoy this book almost as much as we have enjoyed creating it. We can't wait to see what you're going to build after reading it!

How to read this book

This book can be read from cover to cover, but you can also pick and choose the chapters that interest you the most, or the chapters you need immediately for your current projects. Some chapters go together, like the Complications ones, but most of them can be read independently.

We also recommend you to read through *SwiftUI Apprentice* or *SwiftUI by Tutorials*, if you don't have a strong knowledge on the SwiftUI framework.

Finally, for all readers, raywenderlich.com is committed to providing quality, up-to-date learning materials. We'd love to have your feedback. What parts of the book gave you one of those aha learning moments? Was some topic confusing? Did you spot a typo or an error? Let us know at forums.raywenderlich.com and look for the particular forum category for this book. We'll make an effort to take your comments into account in the next update of the book.

Section I

Chapter 1: Hello, Apple Watch

By Scott Grosch

In 2015, Apple promised the world it was going to release a new, revolutionary device. On April 24, 2015, it released the first generation Apple Watch to the public. Some people felt the device was awesome and had huge potential, while others were firmly in the *"meh"* camp.

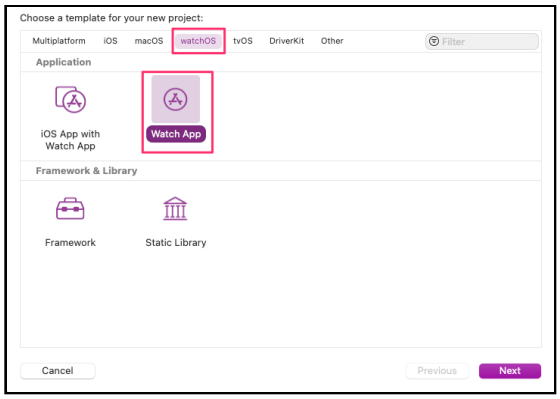
Full disclosure, I was initially one of the latter people.

However, with the addition of features like the Wallet and Apple Pay, as well as the option to purchase cellular data plans for the watch, it's hard to dismiss the device as a fad. Whatever your feelings regarding the Apple Watch, it's a part of the Apple ecosystem you'll definitely want to support to ensure maximum exposure for your apps.

During this chapter, you'll create your first watchOS app and see how much of your existing iOS development knowledge directly transfers to the Apple Watch.

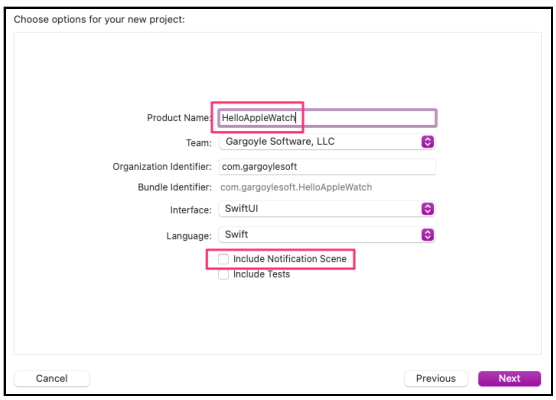
Hello, World!

Open Xcode and create a new project by selecting **File ▸ New ▸ Project...** or pressing **Shift-Command-N**. In the dialog window where you choose your project type, select the **watchOS** tab and then choose the **Watch App** template:



When developing projects that support the Apple Watch, you can choose whether you'll provide a companion iOS app. Since this book aims to teach items specific to watchOS development, most of the chapters focus exclusively on the watchOS project. Only features that require a companion iOS app will include an iOS project.

Next, you'll see the standard options dialog. Enter a **Product Name**, such as **HelloAppleWatch**, and then uncheck the box at the bottom titled **Include Notification Scene**:



In a future chapter, you'll learn how to send push notifications to the Apple Watch, so there's no need to include the relevant code in the created project.

Select **Product ▸ Run** or press **Command-R** to build and run the app. After a moment, the Apple Watch simulator will appear and show a spinner:

It usually takes a few minutes for the simulator to load the watchOS app, so don't worry if the app doesn't start immediately. After a while, you'll see the obligatory "Hello, World!" example that all good programming books start with:

It's just SwiftUI

Take a look at **ContentView.swift**. You'll see something surprising.

It's just SwiftUI code! If you've written apps for watchOS in the past, you're probably incredibly excited right now. You no longer have to use the watchOS specific classes, such as `WKInterfaceController` and `WKInterfaceLabel`, or any other UI-type class. Now, you simply write SwiftUI code, just like you do for iOS. :]

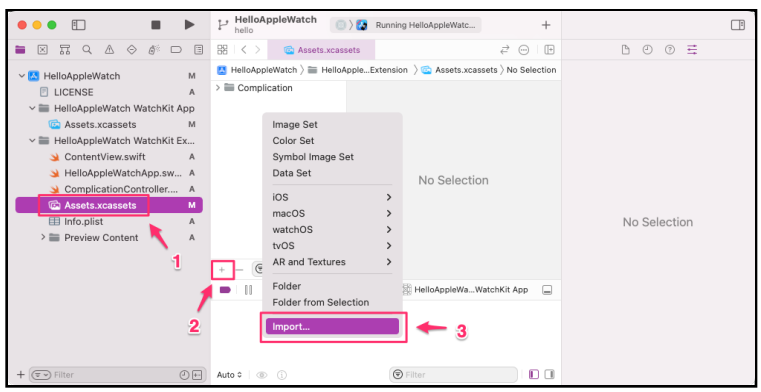
The **WatchKit** specific classes are still available and supported, but there's no reason to use them for new apps unless you need to target devices that aren't running watchOS 6.0 or newer.

Sprucing things up

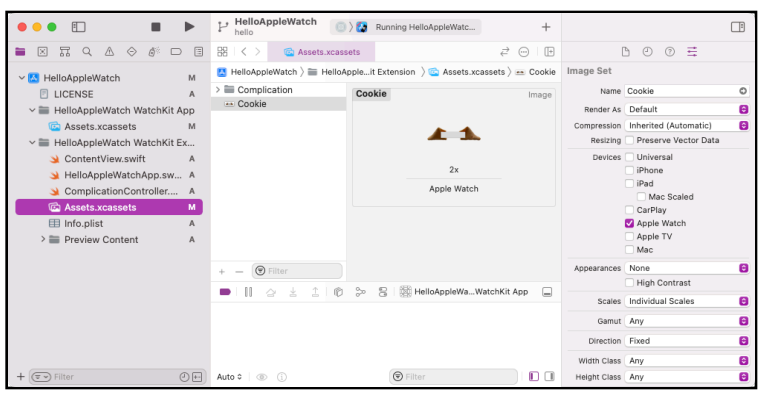
You won't win any awards if you deploy the app as it looks now. The materials for this project include a fortune cookie image. Import that into your **HelloAppleWatch WatchKit Extension** group's assets.

Adding an image

1. Click **Assets.xcassets**.
2. Near the bottom of Xcode, click the + button to bring up a context menu.
3. Select **Import...**.
4. Navigate to this chapter's materials and select **Cookie.imageset**.



Now you have the fortune cookie image available for your app:



Open **ContentView.swift** again and replace the contents of `body` with:

```
VStack {
  Image("Cookie")
    .resizable()
    .scaledToFit()
}
```

Xcode's preview now shows two pieces of a fortune cookie with a blank sheet of paper between them:

Now you need some fortunes!

Adding fortunes

The materials for this chapter also include a file named **EmojiSentence.swift**. Add that to your project by dragging and dropping it into **HelloAppleWatch WatchKit Extension**.

Feel free to look at the code. It provides some example sentences in both English and Emoji, such as:

```
(text: "Not my cup of tea", emoji: "🤦‍♀️ ☕️")
```

Back in **ContentView.swift**, add a new property to the `View`:

```
@StateObject private var sentence = EmojiSentence()
```

The `@StateObject` property wrapper ensures `sentence` is only created a single time, not each time the view's body recalculates. Add the following code just after the `.scaledToFit()` lines:

```
// 1
.overlay(
  // 2
  Text(sentence.emoji)
    // 3
    .font(.title3)
    .padding(.top, 10)
    .buttonStyle(.plain)
)
```

Here's what the preceding code does:

1. You use the `.overlay` call when you wish to layer one view on top of another.
2. You add the emoji symbols as text on top of the fortune cookie image.
3. Then, you use a few modifiers to change the font size and position to make everything fit nicely.

Build and run again. You'll see your emoji fortune:

Unfortunately, not everyone is fluent in Emoji. After the full `Image` definition, just before closing the `VStack`, add:

```
Text(sentence.text)
  .font(.caption)
  .padding(.top, 20)
```

Build and run again — or look in the Xcode preview — and you'll see the English version of the emoji text:

Getting new fortunes

There's one last improvement to add to your app. Just after closing the `VStack`, before the end of `body`, add a tap gesture recognizer:

```
.onTapGesture { sentence.next() }
```

Build and run one final time. Tap the screen, and the fortune changes. Each time you tap, you'll receive the next fortune in the list.

But… watchOS code?

At this point, you may be thinking that there's nothing watchOS specific here, other than the different folder structure and using the watch simulator instead of the phone simulator.

That's the whole point of this chapter! By finally supporting SwiftUI on the Apple Watch, Apple has made it much easier for you to delve into supporting watchOS. You no longer need a very specific skill set to create an Apple Watch app.

The rest of the chapters in this book will delve into development specific to the Apple Watch.

Key points

- SwiftUI removes the need for custom frameworks to develop watchOS apps.
- Storyboards are no longer required.
- SwiftUI has been supported since watchOS 6.0.

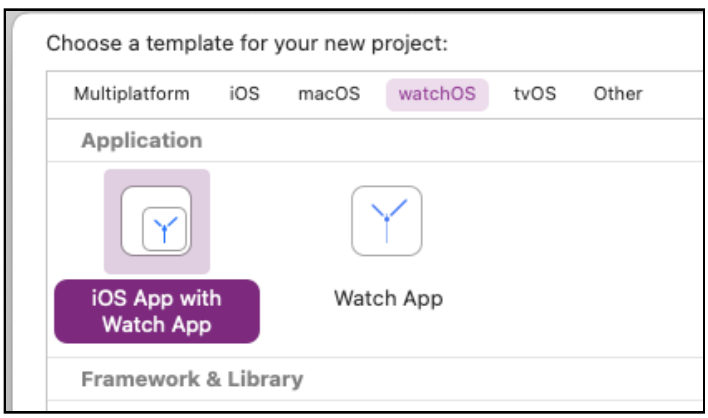
Chapter 2: Project Structure

By Scott Grosch

You'll use Xcode when creating an app to run on the **Apple Watch**, just like when you develop for iOS. The structure is a little different, though. In iOS, you usually have a single target for the app, but you'll always have **three** when creating a watchOS app.

Start Xcode and create a new project. You'll notice some tabs at the top of the template chooser dialog. The **Multiplatform** tab is the default selection.

While you *could* start from the **App** template, you shouldn't. Instead, select the **watchOS** tab. You'll see two templates you can choose from:



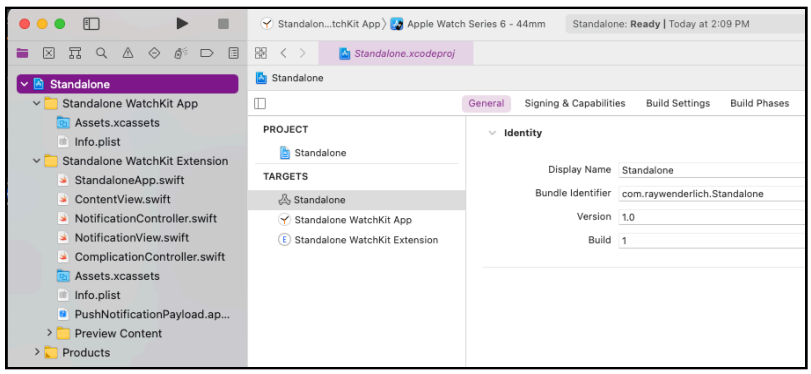
If you're reading this book, you probably need a watch app. However, what type of app are you going to create?

1. Does the watch app stand alone, with no corresponding iOS app?
2. Does the watch app require a companion iOS app to function properly?
3. Does the watch app function on its own, but there's also an iOS app?

Standalone app

Most of this book's chapters pretend that you've chosen the first option. The watch app will be fully functional and won't need a companion iOS app. As the Apple Watch is the focus of this book, I don't want to confuse the examples with a ton of code for an iOS app that doesn't add value. One of the later chapters will provide instructions for tying your Watch App to an iOS app if required.

Select the **Watch App** template and pick a name for your project. I chose **Standalone**.



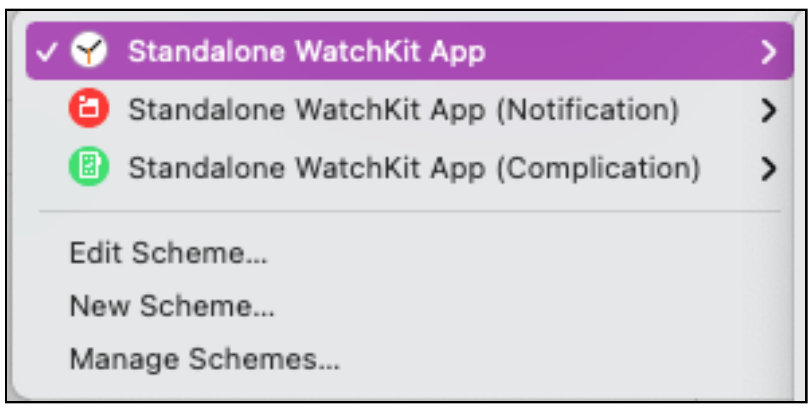
Take a look at the targets, and you'll see the template created three for you:

1. **Standalone** is the general target for your project. It acts as a wrapper for your project so that you can submit it to the App Store.
2. **Standalone WatchKit App** is the target for building the watchOS app. An app bundle contains your watchOS app's storyboard and any assets used by the storyboard.
3. **Standalone WatchKit Extension** is the target for building the watchOS extension. An extension contains your watchOS app's code.

You'll seldom interact with the first two targets. When you think of an Apple Watch *app* you're talking about the *extension*.

Standalone schemes

Look at the scheme selector. You'll see three default schemes:



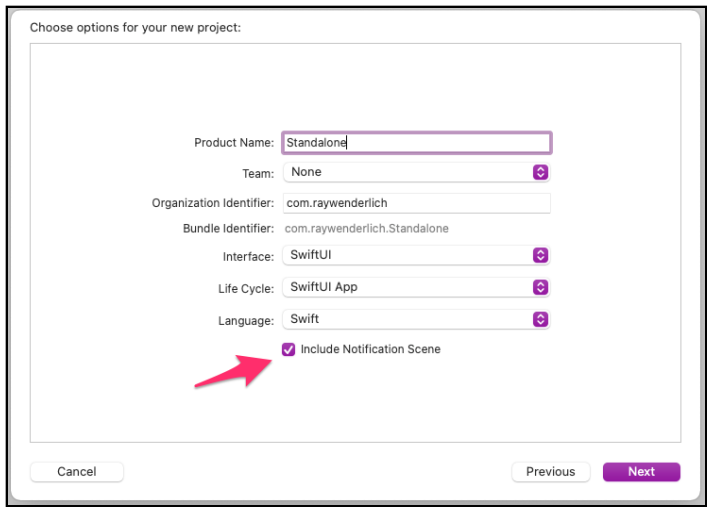
The schemes don't directly map to the three default targets. Instead, they control what type of execution you wish to perform when running in the simulator or on a physical device.

Standalone WatchKit App

The first scheme is the one you'll use most often. You can think of this scheme as the standard "run my app". When you build and run using the **Standalone WatchKit App** scheme, the simulator or physical device will execute what the end-user would see when starting your app.

Standalone WatchKit App (Notification)

You created the second scheme when you left the **Include Notification Scene** checkbox selected while creating the project. More recent versions of watchOS let you send notifications directly to the device.



If you select the **Standalone WatchKit App (Notification)** scheme, then build and run the app, you won't see the standard user interface you'd expect. Instead, the app will jump directly to the notification view which you've defined. You'll only use this scheme when you're testing your local or remote push notifications.

Notifications are a complex topic that you'll address in Chapter 6, "Notifications."

Standalone WatchKit App (Complication)

Due to the complexity of writing a watchOS app, Apple created a final scheme specifically to address that difficulty. Kidding!

Complications, which will be the topic of multiple later chapters, are the small pieces of information you see on the watch's face. Timers, small gauges and the activity ring are all examples of complications.

When debugging your watch app's *optional* complication, you'll use the **Standalone WatchKit App (Complication)** scheme.

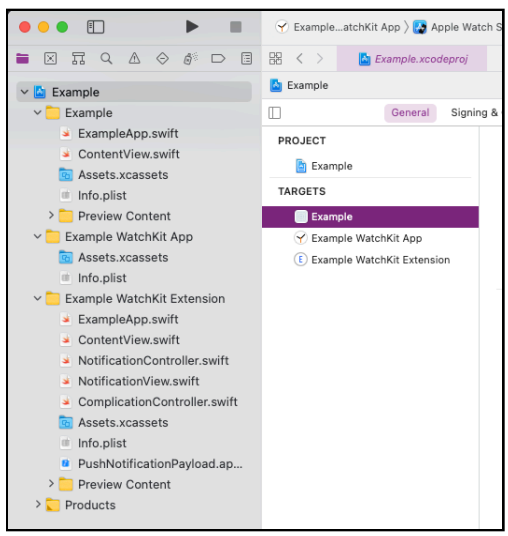
> **Note**: Make sure you select the proper scheme when debugging different parts of your app.

Including an iOS app

If you're going to have both an iOS and a watchOS app, select the **iOS App with Watch App** template when creating a project. You can choose any name you like — I chose **Example** — and leave the defaults for everything else. Click **Next** and choose a location. You'll get Xcode's main window.

Take a look at the targets. You'll see three:

1. **Example** is the target for building the iOS app.
2. **Example WatchKit App** is the target for building the watchOS app.
3. **Example WatchKit Extension** is the target for building the watchOS extension.

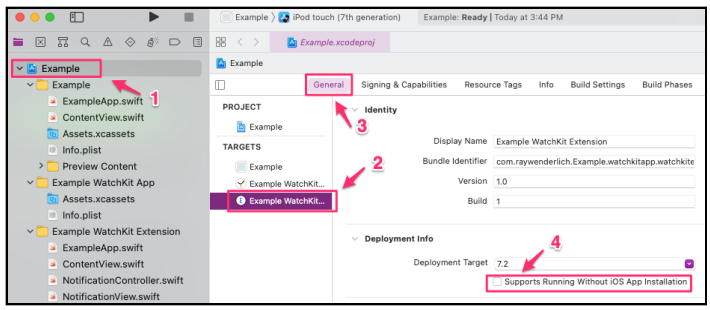


Requiring a companion app

While many watchOS apps require a companion iOS project, it's also possible for a watch app to fully function without a companion iOS app. By default, Xcode assumes your watch app can run without an installed companion iOS app.

If your watch app can't run without a corresponding iOS app, then you'll need to make a small configuration change:

1. Click the project in the Project Navigator, or press **Command-1**.
2. Select the extension target.
3. Select the **General** tab.
4. In the **Deployment Info** section, uncheck the **Supports Running Without iOS App Installation** checkbox.



Naming your app

Run the standalone project in the simulator using the **Standalone WatchKit App scheme**. Once the simulator appears, open the **Settings** app. There are four ways to get there:

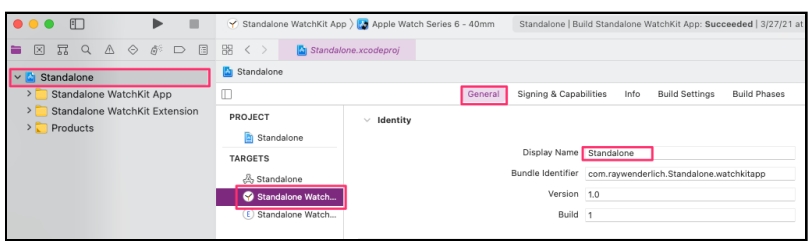
1. Click the digital crown on the simulator.
2. Click the picture of the home icon above the watch image.
3. Choose **Device ▸ Home** from the menubar.
4. Press the **Shift-Command-H** keyboard shortcut.

After tapping the **Settings** app, scroll down to **App View**. Then select **List View**:

Now go back to the home screen and scroll down to the bottom of the list of apps. You'll see your app displayed with a horrible name:

By default, Xcode will append **WatchKit App** to the name you entered when creating the app. The fix is pretty simple if you know where to look.

Bring up the Project Navigator by pressing **Command-1** and select the **Standalone WatchKit App** target. Then switch to the **General** tab, if not already there:



Once there, it's a simple matter of entering the text you want to display by editing the **Display Name**.

Build and rerun the app. Go to the home screen and scroll to your app name in the list. You'll see the shorter name that you likely expected to see in the first place:

App vs. extension folders

Look at the list of files and directories that Xcode created. You'll see three primary directories below the project:

- Standalone WatchKit App
- Standalone WatchKit Extension
- Products

Like any normal Xcode project, you won't do anything with the Products folder.

When assigning the images for your app icon, you'll want to open the directory ending with **App**, for example, **Standalone WatchKit App/Assets.xcassets**. Don't do anything else inside of that folder structure.

For all other assets, as well as all the coding you'll perform, you'll work in the **Extensions** directory, for example, **Standalone WatchKit Extension**.

> **Note**: Previous versions of watchOS worked differently. Don't put your *non-app icon* images in the app directory's **Assets.xcassets** area any longer!

Adding Apple Watch support later

It's OK if you forget to include support for a watch app. Maybe you weren't ready to work on the watch app yet, or perhaps you're adding watch support to a legacy app. Maybe you just plain forgot.

With your existing iOS project open, simply add a new target via **File ▸ New ▸ Target...** and then choose the **Watch App** template.

> **Note**: You won't be able to name the watchOS target the same as the iOS target because Xcode would try to create conflicting names.

Having to pick a different name might be a hard stop for your team. If that's the case, be sure to start with the watch template instead of the iOS template.

Key points

- Watch apps can be standalone, without an accompanying iOS app, *require* a companion iOS app or work independently of the iOS app. There's also an iOS app you may use but don't have to.
- You can add watch support to an iOS app at a later date.
- All your work, except the app icon, should be in the extension folder.
- Remember to change the **Display Name** of the app target.

3 Chapter 3: Digital Crown

By Scott Grosch

In the starter materials for this chapter, please open the **NumberScroller** project. **Build and run** the project. You'll see a lonely number:

Scrolling the **Digital Crown** makes it easy for your users to edit a number's value, such as the one shown.

If you're debugging with a physical device, rotate the crown. If you're debugging on the simulator, scrolling your mouse simulates turning the digital crown. Give it a spin, and watch the number change.

Binding a value

Did nothing happen? That shouldn't come as much of a surprise since you didn't tell watchOS what to do when you rotated the crown.

To let the Digital Crown interact with a SwiftUI control, you use `digitalCrownRotation`. It takes a ridiculous number of parameters in its full form, but you'll start with just one.

Open **ContentView.swift**. Modify the `Text` field as follows:

```
Text("\(number, specifier: "%.1f")")
  .focusable()
  .digitalCrownRotation($number)
```

By default, a `Text` field won't accept focus as it's not an interactive element. When using the `digitalCrownRotation` modifier, typically you'll also immediately precede the call with `.focusable()`, so watchOS lets you interact with the view you're modifying.

`digitalCrownRotation` always takes a binding as the first parameter to anything which implements the `BinaryFloatingPoint` protocol, such as a `Double`. **Build and run** again. This time, when you scroll the Digital Crown, you'll see the number change.

Limiting the scroll range

Generally, having the number scroll infinitely in either direction isn't what you'll want to accomplish. You can limit the range of values by adding two more parameters.

Update your method call like so:

```
.digitalCrownRotation($number, from: 0.0, through: 12.0)
```

You specified a lower limit of `0.0` and an upper limit of `12.0`. **Build and run** the project. You'll see some surprising results.

Scrolling all the way in either direction displays a lower value of `-0.1` and an upper value of `12.1`. When using ranges, you must add one more parameter to tell the Digital Crown how much to change the value.

Now, update the method call again:

```
.digitalCrownRotation($number, from: 0.0, through: 12.0, by:
0.1)
```

Build and run again. This time, you'll get the results you were expecting.

> **Note**: Always include the `by:` parameter when specifying `from:` and `through:`. That way there is no doubt of what the expected behavior is.

The joys of floating-point

Remember that a floating-point value is seldom *exactly* what you expect it to be. The OS tracks floating-point to many more decimal places than a human does. To see what the Digital Crown is doing, add this right before `.focusable()`:

```
.onChange(of: number) { print($0) }
```

onChange lets you display a property's new value any time it changes. Activate Xcode's console by pressing **Shift-Command-C**. Then **build and run** yet again.

As you scroll the Digital Crown, you'll see output like this in the console:

```
9.74101986498215
9.735646908273624
9.730977724331446
9.726920129971415
9.700000000000001
```

There are two main takeaways from that output:

1. The bound property updates with many more values than you'd assume.
2. The bound property is usually, but not *always*, limited by the `by:` parameter when rotation stops.

The first point doesn't matter in an app like the one you're currently working on since the display formats the value as you'd expect. The user will never see those intermediate values.

What *is* important to keep in mind is the second point. When the Digital Crown stops moving, you might have a value of `9.700000000000001` instead of the expected `9.7`. Depending on what you've bound the Digital Crown's value to, that may or may not make a difference.

If you need to perform an equality check against a floating point number, use the `rounded` method, like this:

```
if (value * 10).rounded(.towardZero) / 10 == 9.7 {
  ...
}
```

`rounded(.towardZero)` rounds the value provided to an integral value by always rounding toward zero. You multiply and divide by ten to the power of **X**, where **X** is the number of decimal places you're comparing against.

In the example, `9.7` has a single decimal place, so you compare against 10. If you were comparing against `9.78`, then it's two decimal places, thus `10 ^ 2`, or 100.

Sensitivity

When scrolling the Digital Crown, you can specify how sensitive the update should be. Think of the sensitivity as how much you need to scroll the crown for a change to take effect. Add another parameter:

```
.digitalCrownRotation(
  $number,
  from: 0.0,
  through: 12.0,
  by: 0.1,
  sensitivity: .high
)
```

Build and run, paying attention to how much you have to turn the Digital Crown to scroll through the full range. Next, change the value from `.high` to `.medium` and try again. Finally, change `.medium` to `.low` and try a final time.

When the value is `.low`, you'll notice you have to turn the Digital Crown quite a bit more than when you set the value to `.high`. The default value is `.high`.

Wrapping around

Until now, when you reached the `from` or `through` values, the number stopped changing. You could continue turning the Digital Crown, but it wouldn't make any updates.

For a number range, stopping makes sense. In other scenarios, where the actual value isn't visible to the end-user, wrapping the value around to the starting value might make sense.

In other words, you can scroll up to 12, as normal, but if you keep scrolling the value goes to 0 and starts climbing again. Apple calls this **continuous scrolling**. The default is `false`, but you can enable it by adding another parameter:

```
.digitalCrownRotation(
  $number,
  from: 0.0,
  through: 12.0,
  by: 0.1,
  sensitivity: .high,
  isContinuous: true
)
```

Build and run again. Continuously scroll the Digital Crown, and you'll see the numbers wrap around repeatedly.

Haptic feedback

By default, scrolling the Digital Crown provides a small amount of haptic feedback to the user. If that doesn't make sense for your app, you can turn it off by using the final parameter to the `digitalCrownRotation` call:

```
.digitalCrownRotation(
  $number,
  from: 0.0,
  through: 12.0,
  by: 0.1,
  sensitivity: .high,
  isContinuous: true,
  isHapticFeedbackEnabled: false
)
```

> **Note**: The simulator never provides haptic feedback. :]

Pong

While scrolling numbers is super fun, you won't always show the actual numeric value to your app's user.

In 1972, Atari released **Pong**, the first commercially successful video game. At the time, it was groundbreaking. Show it to your kids today…I dare you. :]

Each player controlled a paddle that only moved up and down to block an incoming ball. Seems like a perfect use case for the Digital Crown to me!

Open **Pong.xcodeproj** from this project's starter materials. Then **build and run**, preparing yourself for a visual feast:

While clearly a gorgeous app, you currently have no way to move the paddle, so the player on the left scores continuously. Time to fix that.

Hooking up the Digital Crown

Open **ContentView.swift**. You'll see that Pong is implemented as a **SpriteKit** game. The `SpriteView(scene:)` initializer makes adding a SpriteKit scene to your SwiftUI app easy. To provide the scene's size to SpriteKit, the `SpriteView` is wrapped in a `GeometryReader`.

You need to link the `crownPosition` to the value of the Digital Crown. Add these two modifiers right after the `SpriteView` line:

```
.focusable()
.digitalCrownRotation($crownPosition)
```

In this case, you don't need any of the numerous options available on the `.digitalCrownRotation` call. All that matters is you constantly update the value of `crownPosition`. The SpriteKit scene will handle the bounds checking. Your `ContentView` shouldn't know or care about the scene's logic.

Now that you've linked the Digital Crown to the scene, it's time to move those paddles.

Paddle movement

Open **PongScene.swift**, which implements all of the SpriteKit scene logic. Scroll down to the overridden `update(_:)`. SpriteKit will call `update(_:)` once per frame before it simulates any actions or physics.

Your goal is to have the defender's paddle move in reaction to the user rotating the Digital Crown. Movement means an action taking place, so `update(_:)` is the proper location to respond to the Digital Crown's value changing.

First, you need to determine whether the user rotated the Digital Crown since the last frame update. At the end of the method, add:

```
let newPosition = crownPosition

// 1
defer { previousCrownPosition = newPosition }

// 2
guard newPosition != previousCrownPosition else { return }
```

Here's what the code does:

1. Regardless of why you exit the method, the code ensures the current crown position reflects as `previousCrownPosition` on the next update.
2. If there's no change, you exit the method right away to keep performance as fast as possible.

Once you know the paddle needs to move, assign a new position by adding this code to the end of the method:

```
// 1
let offset = newPosition - previousCrownPosition
let y = paddleBeingMoved.position.y + offset

// 2
guard minPaddleY ... maxPaddleY ~= y else { return }

// 3
paddleBeingMoved.position.y = y
```

In the preceding code, you:

1. Calculate the amount by which the user scrolled the Digital Crown, storing the paddle's new location.
2. Perform bounds checking on the paddle's location. The scene knows the minimum and maximum y values, so if you continue to turn the Digital Crown, the paddle doesn't leave the game zone.
3. Everything is copacetic, so you tell the paddle about its new location.

The operators used in step two may be unfamiliar to you. The `...` range operator works similarly to the more familiar `..<` operator. The difference is the former includes the final value, whereas the latter doesn't. Using a range operator with the contains operator (~=) lets you determine if a value is in the range.

Step two is a more concise syntax for this equivalent code:

```
guard y >= minPaddleY && y <= maxPaddleY else {
  return
}
```

Build and run again.You should now have a playable Pong video game in your wrist!

> **Note**: This was my first ever SpriteKit project. If you're a professional game developer, try not to cringe too much. :]

Key points

- Remember to add the `focusable` method modifier when using the Digital Crown.
- If you need to compare two floating-point values for equality, use the `rounded(.towardZero)` method.

Chapter 4: Watch Connectivity

By Scott Grosch

The magic of the Apple Watch experience comes from seamless interactions between the watch and your iOS apps.

> **Note**: This is the first chapter that requires the app to run on **both** the Apple Watch and iPhone at the same time. While this setup is possible by starting both simulators from Xcode, the connectivity mechanisms you'll use in this chapter rarely work between them. You may need to run the example projects on real devices to see them in action. Even if you don't have the hardware, it's still a good read.

Watch Connectivity, an Apple-provided framework, lets an iOS app and its counterpart watchOS app transfer data and files back and forth. If both apps are active, communication occurs *mostly* in real time. Otherwise, communication happens in the background, so data is available as soon as the receiving app launches.

The OS takes many factors into account when determining exactly how quickly to pass data packets between devices. While the transfers frequently happen within a matter of moments, sometimes you'll see a significant lag.

Be aware, to transfer data between devices, multiple system resources, such as Bluetooth, must be on. This can result in significant energy use.

> **Note**: Bundle messages together whenever possible to limit battery consumption.

In this chapter, after learning about the different types of messaging options, you'll implement data transfers between the iPhone and Apple Watch versions of **CinemaTime**, an app for patrons of a fictional theater. It lets customers view movie showtimes and buy tickets right from their iPhones and Apple Watches.

Device-to-device communication

The Watch Connectivity framework provides five different methods for transferring data between devices. Four of those methods send arbitrary data, while the fifth sends files between devices. All of the methods are part of `WCSession`.

> **Note**: Although most data transfer methods accept a dictionary of type `[String: Any]`, this doesn't mean you can send just anything. The dictionary can only accept primitive types. See the Property List Programming Guide (http://apple.co/1PZEPXD), for a complete list of supported types.

Those five methods are further subdivided into two categories: interactive messaging and background transfers.

Interactive messaging

Interactive messaging works best in situations where you need to transfer information immediately. For example, if a watchOS app needs to trigger the iOS app to check the user's current location, the interactive messaging API can transfer the request from the Apple Watch to the iPhone.

However, remember there's no guarantee that interactive messages will be delivered. They're sent as soon as possible and delivered asynchronously in **first-in, first-out**, or **FIFO**, order.

If you send an interactive message from your watchOS app, the corresponding iOS app will wake in the background and become reachable.When you send an interactive message from your iOS app, and the watch app isn't active, the watchOS app will **not** wake up.

If you have a dictionary of data, keyed by string, use `sendMessage(_:replyHandler:errorHandler:)`. If, instead, you have a `Data` object, then use `sendMessageData(_:replyHandler:errorHandler:)`.

Reply handlers

When sending an interactive message, you probably expect a reply from the peer device. You may pass a closure of type `([String: Any]) -> Void` as the `replyHandler`, which will receive the message that the peer device sends back.

If you ask the iPhone to generate some type of data, the message will return to the `replyHandler`.

Error handlers

When you wish to know when something goes wrong during a message transfer, you can use the `errorHandler` and pass a `(Error) -> Void`. For example, you'd call the handler if the network fails.

Background transfers

If only one of the apps is active, it can still send data to its counterpart app using background transfer methods.

Background transfers let iOS and watchOS choose a good time to transfer data between apps, based on characteristics such as battery use and how much other data is waiting to transfer.

There are three types of background transfers:

- Guaranteed user information
- Application context
- Files

Take a look at guaranteed user information first.

Guaranteed user information

`transferUserInfo(_:)` makes the first type of background transfer. When calling this method, you specify the data is critical and must be delivered as soon as possible.

The device will continue to attempt to send the data until the peer device receives it. Once the data transfer beings, the operation will continue even if the app suspends.

`transferUserInfo(_:)` delivers every data packet you send in a FIFO manner.

Next, you'll take a look at a similar type of background transfer, application context.

Application context, aka high priority user information

High priority messages, delivered via `updateApplicationContext(_:)`, are similar to guaranteed user information with two important differences:

1. The OS sends the data when it feels it's appropriate to send it.
2. It only sends the most recent message.

If you have data that updates frequently, and you only need the most recent data, you should use `updateApplicationContext(_:)`.

Here are a few examples of when application context makes sense:

- Sending the most recent score in a game. The paired device only needs the most up-to-date score.
- Updating the Apple Watch's dock image.
- A gas-finding app might send updates on prices. It doesn't matter what the gas *used* to cost. You only need the current price.

While guaranteed user information and high priority message work for some situations, you need a different approach with files.

Files

Sometimes you need to send actual files between devices, as opposed to just data. For example, the iPhone might download an image from the network and then send that image to the Apple Watch.

You send a file via `transferFile(_:metadata:)`. You can send any type of dictionary data, keyed by `String`, via the **metadata** parameter. Use metadata to provide information like the file's name, size and when it was created.

Now you're ready to get started.

Getting started

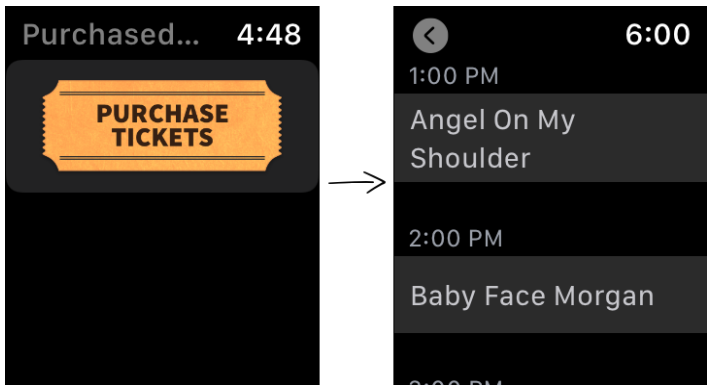
Open the **CinemaTime** starter project in Xcode. Then build and run the **CinemaTime** scheme. The simulator for the iPhone will appear.

Now build and run the **CinemaTime Watchkit App** scheme to start the Apple Watch simulator.

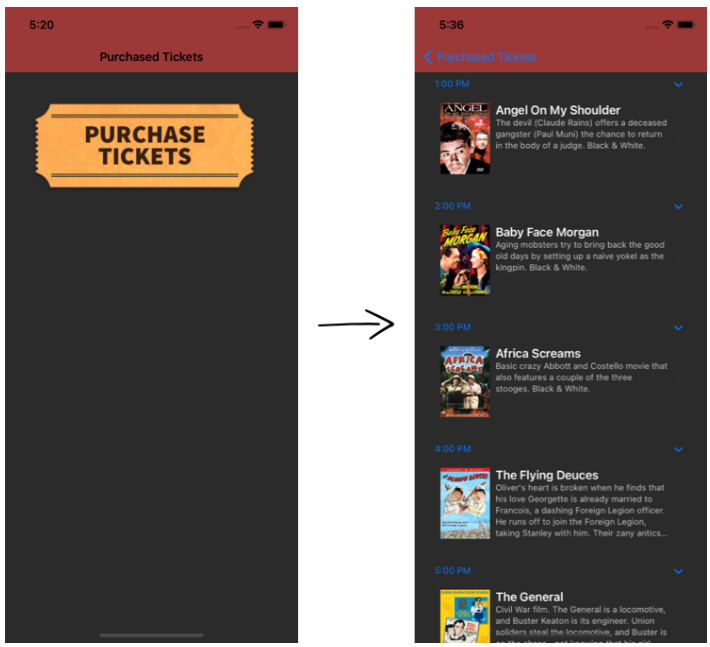
Take a moment to compare the two apps.

Compare and contrast

In the Apple Watch simulator, tap **Purchase Tickets**. Explore the app to see what you have to work with:



Now switch to the iPhone simulator and do the same thing:



While the initial screen looks pretty similar, there are a few noticeable differences on the second screen:

- The iPhone version's grouped list items are collapsible, whereas the watchOS version is not.
- The iPhone includes the movie's poster. There's no room on the Apple Watch to display a poster.
- The Apple Watch version doesn't include details about the movie.

While you *could* include a poster on the Apple Watch version, the image would be so small your customers wouldn't gain any benefit from having it. More importantly, including an image would mean the movie title would have to be even smaller to still fit in a decent amount of space.

The phone has plenty of space to include a short synopsis of the movie, but the Apple Watch doesn't. If you were to include the synopsis, then the person running your app would have to scroll quite a bit more than they likely want to.

Explore the app further by placing an order.

Place your order

Buy a movie ticket in either app. Then view the list of purchased movie tickets in the other app. Do you see the issue?

Movie tickets purchased on one device don't show as purchased on the other device! The apps aren't transferring ticket purchase data between them. Imagine if a customer bought a movie ticket in the iPhone app, then tried to use the Apple Watch app to get into the theater. They'd be turned away, as the Apple Watch wouldn't have a ticket!

Customers have a reasonable expectation that data should be accessible from both versions of an app regardless of which app created the data.

In the rest of this chapter, you'll use the **Watch Connectivity** framework to sync the customer's purchased movie tickets between the iOS and watchOS versions of the app.

First, you'll set up watch connectivity.

Setting up watch connectivity

You should handle all connectivity between your devices from a single location in your code. Hopefully, the term **singleton** comes to mind!

The code you write will be nearly the same for both iOS and watchOS. So, under the **Shared** folder, create a new file called **Connectivity.swift** with the following contents:

```
import Foundation
// 1
import WatchConnectivity

final class Connectivity {
  // 2
  static let shared = Connectivity()

  // 3
  private init() {
    // 4
    #if !os(watchOS)
    guard WCSession.isSupported() else {
      return
    }
    #endif

    // 5
    WCSession.default.activate()
  }
}
```

Here's a code breakdown:

1. You must include the `WatchConnectivity` framework.
2. Use `static` to access singletons.
3. Ensure the initializer is `private` so the class is only accessible via `shared`.
4. You should only start a session if it's supported. An Apple Watch will always support a session. An iOS device will only support a session if there's a paired Apple Watch.
5. When you initialize `Connectivity`, you tell the device to activate the session, which lets you talk to a paired device.

> **Note**: You don't have to wrap the `isSupported` call in an OS check, but I like to make it clear when something is only necessary on a certain type of device.

Be sure to add the file to the target membership of both the iOS app and the extension in the file inspector. Press **Alt-Command-1** to bring up the **File Inspector**.

If you were to run the above code right now, you'd receive an error log message. The documentation explicitly says that activating a session without a delegate is a no-no. So, you'll fix that now!

Preparing for WCSessionDelegate

The `WCSessionDelegate` protocol extends `NSObjectProtocol`. That means for `Connectivity` to be the delegate, it must inherit from `NSObject`.

Change the class declaration to this:

```
final class Connectivity: NSObject {
```

As soon as you subclass from `NSObject`, you'll receive a compiler error on the initializer. Since you've subclassed, you have to override the initializer and call the parent's initializer. Change the init you already have to start like this:

```
override private init() {
  super.init()
```

Next, you'll implement the delegate.

Implementing WCSessionDelegate

You need to make `Connectivity` conform to `WCSessionDelegate`. At the bottom of the file, add:

```
// MARK: - WCSessionDelegate
extension Connectivity: WCSessionDelegate {
  func session(
      _ session: WCSession,
      activationDidCompleteWith activationState:
WCSessionActivationState,
      error: Error?
  ) {
  }
```

```
  func sessionDidBecomeInactive(_ session: WCSession) {
  }

  func sessionDidDeactivate(_ session: WCSession) {
  }
}
```

Almost immediately, you'll receive compiler errors on the last two methods. Those methods are part of the delegate on iOS, but they're *not* present in watchOS.

Since the methods are only valid for iOS, wrap them in a compiler check:

```
#if os(iOS)
func sessionDidBecomeInactive(_ session: WCSession) {
}

func sessionDidDeactivate(_ session: WCSession) {
}
#endif
```

While it may boggle the mind, some people have more than one Apple Watch! If the user swaps watches, the session will deactivate. Apple suggests when the session deactivates, simply restart it. That covers the case of swapping watches.

Add these lines to `sessionDidDeactivate(_:)`:

```
// If the person has more than one watch, and they switch,
// reactivate their session on the new device.
WCSession.default.activate()
```

Now jump back up to `init` and mark the class as the delegate on the line right before calling `WCSession.default.activate()`:

```
WCSession.default.delegate = self
```

Now that you've established a connection, you need some way to send a message to the paired device.

Sending messages

In the class, add:

```
public func send(movieIds: [Int]) {
  guard WCSession.default.activationState == .activated else {
    return
  }
}
```

Whenever you purchase a ticket to a new movie or delete a ticket, you'll want to tell the companion app which movies are now valid. Whenever sending a message, the first thing you must do is ensure the session is active.

Remember, the session state could change for any number of reasons. For example, one device's battery might die, or the user might be in the middle of swapping watches.

Notice how there are no errors when the session isn't active. There's nothing to do, and you don't want to display a confusing message to your end-user.

In Chapter 2, "Project Structure", you learned that a watchOS app doesn't always need a companion iOS app, and vice versa. That means, before sending any message, you need to verify the peer device is there.

Add the following code to your `send(movieIds:)`:

```
// 1
#if os(watchOS)
// 2
guard WCSession.default.isCompanionAppInstalled else {
  return
}
#else
// 3
guard WCSession.default.isWatchAppInstalled else {
  return
}
#endif
```

There are three key points to note in the above code:

1. You use a compiler check to ensure you call the proper method.
2. The Apple Watch checks if the app is on the phone. If it's not, there's nothing else to do.
3. The iOS device checks if the app is on the Apple Watch. If it's not, there's nothing else to do.

Unfortunately, due to legacy watchOS code, the two operating systems have separately named methods.

Now that you've found an active session with your app installed on both devices, it's time to send the data. Add these lines to the end of the method you're working on:

```
// 1
let userInfo: [String: [Int]] = [
```

```
    ConnectivityUserInfoKey.purchased.rawValue: movieIds
  ]

  // 2
  WCSession.default.transferUserInfo(userInfo)
```

Here's a code breakdown:

1. When sending data to a paired device, you must use a dictionary keyed by a string. The sample project includes an `enum` to specify which keys are valid for transfers, so you don't hardcode strings.
2. Once the data is ready, you can transfer it to the paired device.

Refer back to the start of this chapter, and you'll likely conclude that `transferUserInfo(_:)` is the wrong method to use. You're correct!

Before the end of the chapter, you'll use each type of connectivity and build out a mostly generic connectivity class you can use as a base in all your projects.

Receiving messages

After transferring the user information, you need some way to receive it on the other device. You'll receive the data in the `WCSessionDelegate` of `session(_:didReceiveUserInfo:)`. Using the Combine framework is a great way to let your app know about the updates.

Modify your class to implement `ObservableObject`, and then add a publisher:

```
final class Connectivity: NSObject, ObservableObject {
  @Published var purchasedIds: [Int] = []
```

> **Note**: You can learn more about `@Published` in our RW book: *Combine: Asynchronous Programming with Swift (*https://bit.ly/3y8Ot5o*)*.

Using `@Published`, you enable other parts of your program to observe changes to `purchasedIds`.

Add the following method to the delegate extension:

```
// 1
func session(
  _ session: WCSession,
  didReceiveUserInfo userInfo: [String: Any] = [:]
```

```
  ) {
    // 2
    let key = ConnectivityUserInfoKey.purchased.rawValue
    guard let ids = userInfo[key] as? [Int] else {
      return
    }

    // 3
    self.purchasedIds = ids
  }
```

Here's a code breakdown:

1. When transferring data via `transferUserInfo(_:)`, you receive it via the aptly named `session(_:didReceiveUserInfo:)`.
2. Check to see if the provided user information contains a list of purchased movie keys. If not, quietly do nothing.
3. Assigning to `purchasedIds` automatically notifies all observers about the change.

Build your project to ensure you aren't missing anything at this point. You won't get any errors or warnings, but you can't run yet as there's still nothing visible to see. Almost there!

The ticket office

When you purchase or delete a ticket, you need to let companion devices know. **Shared/TicketOffice.swift** handles purchasing and deleting tickets, so it seems like a good place to handle connectivity! Open that file.

Look through it. You'll see both `purchase(_:)` and `delete(at:)`. These methods need to send a message to the companion app.

Since you would never consider duplicating code, add a private method to the bottom of the file:

```
  private func updateCompanion() {
    // 1
    let ids = purchased.map { $0.id }

    // 2
    Connectivity.shared.send(movieIds: ids)
  }
```

In the preceding code:

1. There's no need to transfer entire `Movie` objects between devices. Instead, you grab the ID of each ticket you've purchased.
2. Using your newly created `Connectivity`, you send the data to the other device.

Add a call to `updateCompanion()` as the last step in both `delete(at:)` and `purchase(_:)`.

You're now sending and receiving ticket purchase updates. However, if you build and run the app at this point, the tickets still won't update. Why not?

Look at **PurchasedTicketsListView.swift** in either the iOS or watchOS target. You'll see the list of tickets comes from the `purchased` property of `TicketOffice`. Update the initializer to include the following code at the end of the method:

```
// 1
Connectivity.shared.$purchasedIds
  // 2
  .dropFirst()
  // 3
  .map { ids in
    movies.filter { ids.contains($0.id) }
  }
  // 4
  .receive(on: DispatchQueue.main)
  // 5
  .assign(to: \.purchased, on: self)
  //6
  .store(in: &cancellable)
```

That's quite the chain of calls! If you're not familiar with the Combine framework, that probably looks pretty scary. Here's a line-by-line breakdown:

1. By prefacing the property with `$`, you tell Swift to look at the publishing item rather than just the value.
2. You declared `purchasedIds` with an initial value of `[]`, meaning an empty array. By dropping the first item, you essentially skip the empty assignment.
3. Next, you retrieve `Movie` objects identified by the IDs sent to the device. Performing the inner filter ensures an ID sent for a movie that doesn't exist doesn't throw an error.
4. Because you'll use the list of purchased movies to update the user interface, you switch over to the main thread. Always make UI updates on the main thread!

5. Now that you've created a list of `Movie` objects, assign that to `purchased`.
6. Finally, include the standard boilerplate for Combine, which stores the chain in `Set<AnyCancellable>`.

It's the moment of truth! Build and run the updated app on both devices. Then purchase a ticket on the phone.

After a few moments, the Apple Watch will update.

> **Note**: Remember, updates can take some time. While writing this book, I saw the iPhone update the Apple Watch almost immediately, whereas the watch sometimes updated the phone three minutes or so later.

Application context

While functional, using `transferUserInfo(:_)` isn't the best choice for **CinemaTime**. If you purchase a movie ticket on one device, you don't need to see it immediately on the other device. Unless you're standing right outside the theater, it's OK if the transfer happens later. Even if you *are* outside the theater, you'd still use the device you purchased the ticket on.

In this case, the best choice is to use `updateApplicationContext(_:)`. You need the message to arrive, so none of the `sendMessage` variants would make sense.

Add a new file named **Shared/Delivery.swift** with the following contents:

```
enum Delivery {
  /// Deliver immediately. No retries on failure.
  case failable

  /// Deliver as soon as possible. Automatically retries on
failure.
  /// All instances of the data will be transferred
sequentially.
  case guaranteed

  /// High priority data like app settings. Only the most recent
value is
  /// used. Any transfers of this type not yet delivered will be
replaced
  /// with the new one.
  case highPriority
}
```

> **Note**: Remember to update the target membership to include both the iOS app **and** the watchOS extension.

You'll use that `enum` to specify the type of delivery the caller wants to use. I chose the labels I did because they make more sense to me. `guaranteed` and `highPriority` make sense to me when I glance at code later, instead of trying to remember whether it was a user information transfer or an updated application context guaranteed to arrive quickly.

Now jump back to **Shared/Connectivity.swift**. Add a couple of extra parameters to the send method:

```
public func send(
  movieIds: [Int],
  delivery: Delivery,
  errorHandler: ((Error) -> Void)? = nil
) {
```

Now you can specify both the type of delivery you'd like to use as well as an optional error handler.

Then, at the bottom of that method, replace `transferUserInfo(_:)` with:

```
switch delivery {
case .failable:
  break

case .guaranteed:
  WCSession.default.transferUserInfo(userInfo)

case .highPriority:
  do {
    try WCSession.default.updateApplicationContext(userInfo)
  } catch {
    errorHandler?(error)
  }
}
```

You'll handle the `.failable` case in a moment. In the case of a high-priority delivery, there's a new wrinkle to handle.

Updating the application context might throw an exception, which is why the send method now accepts an optional error handler.

> **Note**: Remember, you call the session delegate methods on a background thread, so be kind to your consumers and dispatch back to the main queue for them.

In the delegate, you need to receive the application context like you received user information. Since the code is going to be the same, do a quick bit of refactoring.

Extract the contents of `session(_:didReceiveUserInfo:)` into a private method:

```
private func update(from dictionary: [String: Any]) {
  let key = ConnectivityUserInfoKey.purchased.rawValue
  guard let ids = dictionary[key] as? [Int] else {
    return
  }

  self.purchasedIds = ids
}
```

Then call it from the delegate method:

```
func session(
  _ session: WCSession,
  didReceiveUserInfo userInfo: [String: Any] = [:]
) {
  update(from: userInfo)
}
```

Also call it from a new delegate method for receiving the application context:

```
func session(
  _ session: WCSession,
  didReceiveApplicationContext applicationContext: [String: Any]
) {
  update(from: applicationContext)
}
```

Switch back to **Shared/TicketOffice.swift** and replace the connectivity call in `updateCompanion()` with:

```
Connectivity.shared
    .send(movieIds: ids, delivery: .highPriority, errorHandler:
{
        print($0.localizedDescription)
    })
```

Now, when a user purchases a ticket, you'll use the application context transference method instead. A production app would, however, likely want to do something better than just printing to the console in case of an error.

Optional messages

Remember, interactive messages might fail to send. While that makes them inappropriate for the CinemaTime app, you'll make the appropriate updates to `Connectivity` to support them so you can reuse the code later.

It's a bit trickier to deal with interactive messages. They take an optional reply handler and an optional error handler. If you can't send the message or ask for a reply and can't receive it, the error handler is called.

The most common reason for an error is when the paired device isn't reachable, but other errors are possible.

> **Note**: Remember, if you're not expecting a reply from the peer device, you *must* pass `nil` as the reply handler when calling `sendMessage(_:replyHandler:errorHandler:)`. If you pass a closure to the parameter, you tell the OS you expect a reply, and it should generate an error if one isn't received.

Also, don't directly pass `replyHandler` and `errorHandler` from your custom send method, as the handlers would then run on a background thread, not the main thread.

How do you handle both situations? With the addition of a small helper function to **Shared/Connectivity.swift**. It's quite clean.

Add this to the end of `Connectivity`:

```
// 1
typealias OptionalHandler<T> = ((T) -> Void)?

// 2
private func optionalMainQueueDispatch<T>(
  handler: OptionalHandler<T>
) -> OptionalHandler<T> {
  // 3
  guard let handler = handler else {
    return nil
  }
```

```
    // 4
    return { item in
      // 5
      DispatchQueue.main.async {
        handler(item)
      }
    }
  }
```

There are a ton of powerful features here in a few lines of code. Here's a breakdown:

1. While not strictly necessary, using `typealias` makes the rest of the code easier to read. You create an alias for an optional method that takes a single generic parameter `T` and has no return value.
2. You declare a method of that same `T` type and take an optional handler. Remember that `OptionalHandler<T>` is already optional, so you don't add `?` to the end of the type.
3. If no handler was provided, then return `nil`.
4. The return type is an `OptionalHandler<T>`, meaning you need to return a closure to represent the call. The closure will take a single item of type `T`, as expected by the definition of the type.
5. You dispatch the provided handler to the main thread using the delivered data.

It's quite a bit to wrap your head around if you've never returned a closure from another method. So, don't be afraid to work through the code a few times to be sure you understand it.

Non-binary data

Optional messages might or might not expect a reply from the peer device. So, add a new `replyHandler` to your send method in **Shared/Connectivity.swift** so it looks like this:

```
  public func send(
    movieIds: [Int],
    delivery: Delivery,
    replyHandler: (([String: Any]) -> Void)? = nil,
    errorHandler: ((Error) -> Void)? = nil
  ) {
```

Now it's time to enjoy the fruits of your labor. Replace the `break` statement in the `.failable` case with the following:

```
WCSession.default.sendMessage(
  userInfo,
  replyHandler: optionalMainQueueDispatch(handler:
replyHandler),
  errorHandler: optionalMainQueueDispatch(handler: errorHandler)
)
```

By implementing `optionalMainQueueDispatch(_:)`, you keep a clean `case` that handles all the complexities of passing a handler vs. `nil`. You also ensure that both handlers, if provided, are called on the main thread.

Technically, you'd be OK always providing an error handler. But why make the OS perform the extra work of configuring and dispatching an error if you're going to ignore it anyway?

To handle receiving messages, add two separate delegate methods:

```
// This method is called when a message is sent with failable
priority
// *and* a reply was requested.
func session(
  _ session: WCSession,
  didReceiveMessage message: [String: Any],
  replyHandler: @escaping ([String: Any]) -> Void
) {
  update(from: message)

  let key = ConnectivityUserInfoKey.verified.rawValue
  replyHandler([key: true])
}

// This method is called when a message is sent with failable
priority
// and a reply was *not* requested.
func session(
  _ session: WCSession,
  didReceiveMessage message: [String: Any]
) {
  update(from: message)
}
```

It's unfortunate, but Apple implemented two completely separate delegate methods instead of having one with an optional reply handler. In both instances, you'll call `update(from:)`, just like in the other delegate methods you've added so far.

The only difference is that in the delegate method, which expects to send a reply back to the other device, you have to invoke the provided handler. The data you send back is arbitrary. For this example, you send back a `true` response.

Binary data

Optional messages can also transfer binary data. It's unclear why only optional messages provide a binary option.

You'll need a separate sending method in `Connectivity` to handle the `Data` type. As you coded it, you'd quickly see that both methods need all the same `guard` clauses. So, refactor them into a method of their own:

```
private func canSendToPeer() -> Bool {
  guard WCSession.default.activationState == .activated else {
    return false
  }

  #if os(watchOS)
  guard WCSession.default.isCompanionAppInstalled else {
    return false
  }
  #else
  guard WCSession.default.isWatchAppInstalled else {
    return false
  }
  #endif

  return true
}
```

Those are the same checks you performed previously. Now, you move them to a method that returns `true` when you can send.

Remove those guard clauses from `send(movieIds:delivery:replyHandler:errorHandler:)` and replace them with a call to the new method:

```
guard canSendToPeer() else { return }
```

Now you can implement the method to handle binary data:

```
public func send(
  data: Data,
  replyHandler: ((Data) -> Void)? = nil,
  errorHandler: ((Error) -> Void)? = nil
) {
  guard canSendToPeer() else { return }
```

```
    WCSession.default.sendMessageData(
      data,
      replyHandler: optionalMainQueueDispatch(handler:
replyHandler),
      errorHandler: optionalMainQueueDispatch(handler:
errorHandler)
    )
  }
```

Receiving binary data, of course, results in two more delegate methods:

```
  func session(
    _ session: WCSession,
    didReceiveMessageData messageData: Data
  ) {
  }

  func session(
    _ session: WCSession,
    didReceiveMessageData messageData: Data,
    replyHandler: @escaping (Data) -> Void
  ) {
  }
```

I haven't provided a sample implementation as that would be specific to the type of binary data your app is transferring.

Transferring files

If you run the app on your iOS device and purchase a ticket, you'll notice that the movie details include a QR code. Ostensibly, that QR code is what the theater would scan to grant entry. Purchasing a ticket on the Apple Watch, however, does not display a QR code.

You'll see a file called **CinemaTime/QRCode.swift** that generates the QR code using the **CoreImage** library. Unfortunately, CoreImage does not exist in watchOS.

An image like this is an excellent example wherein you might opt to use file transfers. When you purchase a ticket on the Apple Watch, the iOS device gets a message with the new movie list. Wouldn't that be a great time to ask the iOS device to generate a QR code and send it back?

> **Note**: When debugging file transfer issues, consider using the `.failable` delivery type, so the transfer is attempted immediately.

QR Codes

Move **CinemaTime/QRCode.swift** into **Shared**. Then add the watch extension to the target membership. To fix the error that immediately appears, wrap both the import of CoreImage and `generate(movie:size:)` in a compiler check:

```
import SwiftUI
#if canImport(CoreImage)
import CoreImage.CIFilterBuiltins
#endif

struct QRCode {
  #if canImport(CoreImage)
  static func generate(movie: Movie, size: CGSize) -> UIImage? {
    // Code removed for brevity. [...]
  }
  #endif
}
```

When the Apple Watch displays the details of a purchased ticket, it needs to know where to look for the QR code's image. Add a helper method to the `struct` but *outside* the `canImport` check:

```
#if os(watchOS)
static func url(for movieId: Int) -> URL {
  let documents = FileManager.default.urls(
    for: .documentDirectory,
    in: .userDomainMask
  )[0]

  return documents.appendingPathComponent("\(movieId).png")
}
#endif
```

iOS doesn't need to look at a file URL, but watchOS does. The preceding code gets the path to the app's documents directory and then appends the movie's ID to the path.

> **Note**: If you don't name the file with a `.png` suffix, then the OS will refuse to turn that file into an image.

Open **Shared/Connectivity.swift** and edit `send(movieIds:delivery:replyHandler:errorHandler:)` to add `wantedQrCodes`:

```
public func send(
  movieIds: [Int],
```

```
    delivery: Delivery,
    wantedQrCodes: [Int]? = nil,
    replyHandler: (([String: Any]) -> Void)? = nil,
    errorHandler: ((Error) -> Void)? = nil
  ) {
```

Then, change `userInfo` from a `let` to a `var` and assign the wanted QR codes:

```
  var userInfo: [String: [Int]] = [
    ConnectivityUserInfoKey.purchased.rawValue: movieIds
  ]

  if let wantedQrCodes = wantedQrCodes {
    let key = ConnectivityUserInfoKey.qrCodes.rawValue
    userInfo[key] = wantedQrCodes
  }
```

That provides a way for the Apple Watch to request a QR code for a list of movies from the iOS device. Why a list instead of just one ID? The previous request to deliver an image might have failed for any number of possible reasons.

Now you can implement the method that will run on the iOS device to generate and send the QR code images. Add this code to the end of `Connectivity`:

```
  #if os(iOS)
  public func sendQrCodes(_ data: [String: Any]) {
    // 1
    let key = ConnectivityUserInfoKey.qrCodes.rawValue
    guard let ids = data[key] as? [Int], !ids.isEmpty else
  { return }

    let tempDir = FileManager.default.temporaryDirectory

    // 2
    TicketOffice.shared
      .movies
      .filter { ids.contains($0.id) }
      .forEach { movie in
        // 3
        let image = QRCode.generate(
          movie: movie,
          size: .init(width: 100, height: 100)
        )

        // 4
        guard let data = image?.pngData() else { return }

        // 5
        let url =
  tempDir.appendingPathComponent(UUID().uuidString)
```

```
        guard let _ = try? data.write(to: url) else {
          return
        }

        // 6
        WCSession.default.transferFile(url, metadata: [key:
movie.id])
    }
}
#endif
```

Here's a code breakdown:

1. If the data passed to the method doesn't contain a list of movie IDs that require a QR code, then exit the method.
2. Grab all of the movies which exist with the requested IDs. Silently ignore bad IDs.
3. Now that you have an actual `Movie` object, generate a QR code for it at a size that will work well on the Apple Watch.
4. If either the QR code failed to generate an image or the image couldn't generate PNG data, there's nothing more to do here. So, move on to the next one.
5. Generate a temporary file with a unique name and write the PNG data to that file. If it fails, quietly return.
6. Finally, initiate a file transfer to the peer device, meaning the Apple Watch.

Notice how `metadata` contains the ID of the movie whose QR code you just generated.

Call that method first in `update(from:)`:

```
#if os(iOS)
sendQrCodes(dictionary)
#endif
```

It's important to do so *before* the rest of the code because you might have been asked for a QR code again and had no new purchases.

Of course, it wouldn't be connectivity code if you didn't have to implement another delegate! Add the following method to the delegate section:

```
// 1
#if os(watchOS)
func session(_ session: WCSession, didReceive file:
WCSessionFile) {
```

```
    // 2
    let key = ConnectivityUserInfoKey.qrCodes.rawValue
    guard let id = file.metadata?[key] as? Int else {
      return
    }

    // 3
    let destination = QRCode.url(for: id)

    // 4
    try? FileManager.default.removeItem(at: destination)
    try? FileManager.default.moveItem(at: file.fileURL, to:
destination)
}
#endif
```

In the preceding code, you:

1. Only receive a file transfer on watchOS, so you wrap it in a compiler check.
2. Pull the movie's ID from the metadata. If no ID exists, quietly exit the method.
3. Determine where you should write the indicated movie's QR code.
4. Remove the QR code if there's already one there, and then move the received file to the proper location.

> **Note**: watchOS will delete the received file if it still exists when the method ends. You must *synchronously* move the file to a new location if you wish to keep it.

Now that you have a way to request and receive QR codes, you just need to modify the `TicketOffice` code. Open **Shared/TicketOffice.swift** and edit the `delete` method. If you delete a ticket from the Apple Watch, you might as well remove the image, too. At the start of the method, add:

```
#if os(watchOS)
offsets
  .map { purchased[$0].id }
  .forEach { id in
    let url = QRCode.url(for: id)
    try? FileManager.default.removeItem(at: url)
  }
#endif
```

Here, you map each row you delete to the corresponding movie's ID. Once you have a list of IDs, you determine where to store the QR code and then remove it.

Now edit `updateCompanion` to figure out which movies still need a QR code. Replace the existing connectivity call with:

```
// 1
var wantedQrCodes: [Int] = []

// 2
#if os(watchOS)
wantedQrCodes = ids.filter { id in
  let url = QRCode.url(for: id)
  return !FileManager.default.fileExists(atPath: url.path)
}
#endif

// 3
Connectivity.shared.send(
  movieIds: ids,
  delivery: .highPriority,
  wantedQrCodes: wantedQrCodes
)
```

You add a few minor updates here:

1. You store a list of all QR codes you need to request, defaulting to none.
2. If running on the Apple Watch, you identify all purchased movies that don't have a stored QR code yet.
3. Finally, you include `wantedQrCodes` in the connectivity call.

Next, you'll add some code to the end of the struct to retrieve the locally stored QR code. Open **Shared/Movie.swift** and add the following:

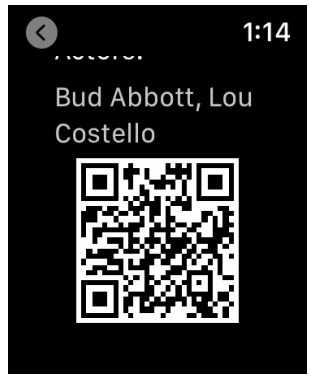
```
#if os(watchOS)
func qrCodeImage() -> Image? {
  let path = QRCode.url(for: id).absoluteString
  if let image = UIImage(contentsOfFile: path) {
    return Image(uiImage: image)
  } else {
    return Image(systemName: "xmark.circle")
  }
}
#endif
```

If the QR code exists where it's supposed to, you return it as a SwiftUI `Image`. If the image doesn't exist, then you return an appropriate default image instead.

Finally, you need to check for the movie purchase. Open **CinemaTime WatchKit Extension/MovieDetailsView** and add `else` like this:

```
if !TicketOffice.shared.isPurchased(movie) {
  PurchaseTicketView(movie: movie)
} else {
  movie.qrCodeImage()
}
```

Build and rerun your app. Purchase a movie from the Apple Watch. After some time, you'll see the QR code at the bottom of the details screen if you navigate away from and back to the details view.



Key points

- There are many methods available for sending data. Be sure you choose one based on how quickly you need the data to arrive.
- If you send an interactive message from your watchOS app, the corresponding iOS app will wake up in the background and become reachable.
- If you send an interactive message from your iOS app while the watchOS app is *not* in the foreground, the message will fail.
- Bundle messages together whenever possible to limit battery consumption.
- Do not supply a reply handler if you aren't going to reply.

Where to go from here?

In this chapter, you set up the Watch Connectivity framework, learned about the different ways to transfer data between counterpart iOS and watchOS apps, and successfully implemented the application context transfer method.

Keeping your iOS and watchOS apps in sync through data transfer is important. That way, users can use both devices indistinctly. In the next chapter, you will learn how to provide users a quick view of the current state of your app. By using Snapshots, you produce apps that feel responsive and up to date.

5 Chapter 5: Snapshots

By Scott Grosch

The Dock on the Apple Watch lets the wearer see either the recently run apps or their favorite apps. Since watchOS already displays your app at the appropriate time, why should you care about the Dock?

While watchOS will insert an image of your app into the Dock, it'll only display whatever the current screen contains, which may not be the best user experience.

In this chapter, you'll build **UHL**, the official app for the Underwater Hockey League. You may not have heard of this sport before, and you're not alone. Underwater hockey has a cult following. Two teams maneuver a puck across the bottom of a swimming pool.

You're probably feeling the urge to dive in, so time to get to it. CANNONBALL!

Getting started

Open **UHL.xcodeproj** from the starter folder. Build and run the **UHL WatchKit App** scheme.

With the UHL app, you can track the best team in the league: Octopi. Take a moment to explore the app. You'll see Octopi's record as well as the details of what matches are coming up next.

The Dock

The Apple Watch has only two physical buttons. While the Digital Crown is the most visible, there's also a Side button right below the crown. Most users interact with the Side button by quickly pressing it twice to bring up Apple Pay.

If you press the button a single time, the Dock will launch. The Dock displays one of two sets of apps:

1. Recently run
2. Favorites

By default, recently run apps will display in the Dock:

> **Note**: If you choose favorites, then you'll *only* see apps that you've identified. The most recently run app no longer displays in the Dock. To choose what appears, you have to open the Watch app on your iPhone, tap **My Watch**, then tap **Dock** and select either **Recents** or **Favorites**.

The Dock provides multiple benefits:

- Tapping an image launches the app *immediately*.
- Quickly switching between apps.
- Showing current app status at a glance.
- App organization.

Apps currently in the Dock have an almost immediate launch time, as they're kept in memory. In this chapter, your focus is the third bullet. Apps in the Dock show a screenshot of the app's current state called a **snapshot**.

To see an example of snapshots in action, grab your Apple Watch and start a timer. You'll need a **physical device** as the simulator doesn't provide timers.

Once the timer is running, press the **Side button** and scroll to the Timer. It *appears* to be counting down in the Dock, but in reality, that's not what's happening.

The Dock is nothing more than a *static image* of the current state of the app. There are no interactive controls in the Dock. If you tap the snapshot, the app will launch. The Timer app *appears* to be updating as the app has configured itself to take a new snapshot every second.

Snapshot API

By default, when the Dock appears, the user sees what each app looked like before moving to the background. For most apps, nothing more is necessary. Sometimes, such as in the case of the Timer app, a single snapshot is not enough.

You, as the developer, are responsible for telling watchOS if it needs to perform extra snapshots once the app moves to the background. Before diving into the code, you should keep some rules of thumb in mind.

Snapshot tips

Next, we are going to learn some **tips and tricks** you should take into account if you want to optimize your app snapshots.

Optimizing for miniaturization

The snapshot is a scaled-down image of your app's full-size dimensions. Be sure to carefully look at all the screens which you capture in a snapshot. Is the text still readable? Do images make sense at a smaller size?

Remember:

- At smaller sizes, bolder fonts are easier to read. You may wish to use bolded fonts or larger font sizes for important information.
- You may also need to consider removing less important elements from the screen.

Customizing the interface

Recall that the snapshot is just an image of the app's current display. With that in mind, you could make a custom `View` for when watchOS takes a snapshot and *completely redesign the UI*. **Don't do that**.

Having a custom view could make sense in some situations, especially if you need to remove some elements from the snapshot. If you make a custom view, you want to ensure that the snapshot doesn't look radically different from the normal display. People expect the snapshot to represent your app. If the snapshot looks too different, it becomes hard to recognize and find the app they're looking for.

If you determine that you do need a different visual, please keep these points in mind:

1. Focus on important information.
2. Hide objects that aren't as relevant when viewed in the Dock.
3. Exaggerate the size of certain objects for legibility.
4. Don't make the interface look radically different.

Progress and status

Progress screens, timers and general status updates are great use cases for snapshots. You may think that complications cover all of those cases. Hopefully, you created said complications! However, you need to remember that your customers may not want to use your complication.

During the COVID-19 pandemic, food delivery services became a major business. Online ordering apps are a perfect example of custom snapshots.

When you order the food, you see one view. Once the restaurant receives your order, the screen could change to show how long until the food is ready for pickup. When the driver gets the food, the screen could show how long until delivery.

During state changes, be sure that you're not snapshotting errors or confirmation dialogs. Such issues are better handled via a local notification.

Changing screens

Keep the currently active view the same as when the user last interacted with your app whenever possible. If the app's state changes in unpredictable ways, it can confuse and disorient the user. You want the interaction between the Dock and your app to be so seamless that your customers don't understand anything special is happening. If you need to leave the app in a different state, ensure that the end-user can quickly determine what happened.

> **Note**: In the interests of keeping the sample app less complicated, UHL blatantly violates this tip. :]

Anticipating a timeline

The inverse to the previous tip is that you should anticipate what the user would want to see when they look in the Dock.

Similar to the food delivery example, consider a sporting event. Depending on the time of the event, you might want to have something like this:

- Shortly before the game, users want to see the time and location.
- During the game, users want to see the current score.
- After the game, users want to see the final score and your team's record.
- Other times, users likely want to see the season's schedule.

User preferences

Not every user is going to want to see the same thing. Can your app offer the possibility of customizing views even further?

A fair-weather fan who only cares about a single team will need a different experience than a sports fanatic who wants to follow the entire league. The fair-weather fan would probably expect the last game's score to stick around in a snapshot much longer than the fanatic who constantly keeps up-to-date.

When snapshots happen

watchOS automatically schedules snapshots to update on your behalf in many different scenarios:

- When the Apple Watch boots up.
- When a complication update occurs.
- When the app transitions from the foreground to the background.
- When the user views a long look notification.
- One hour after the user last interacted with the app.
- At least once an hour for apps in the Dock. The Dock takes one snapshot every six minutes rotating through each app in sequence. If the user has fewer than ten apps in the Dock, each app will receive more frequent snapshot tasks than one per hour.
- At optional, scheduled times, with the Background Refresh API.

Working with snapshots

Now that you better understand how to use snapshots, it's time to implement what you've learned in the UHL app!

> **Note**: The simulator frequently gets confused while working with snapshots and the Dock. If that happens, quit and restart the simulator or preferably use a physical device instead.

Snapshots handler

First, you need to implement the method watchOS called when it's time to take snapshot. Open **ExtensionDelegate.swift** and add the following method:

```
func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
  // 1
  backgroundTasks.forEach { task in
    // 2
    guard let snapshot = task as?
WKSnapshotRefreshBackgroundTask else {
      task.setTaskCompletedWithSnapshot(false)
      return
    }

    // 3
    print("Taking a snapshot")

    // 4
    task.setTaskCompletedWithSnapshot(true)
  }
}
```

Whenever watchOS wakes up your app from the background, it calls `handle(_:)`. Here's what the preceding code accomplishes:

1. First, it schedules one or more tasks for you to handle. Loop through each one.
2. You only care about snapshots, so mark the task completed if it's not a snapshot. Passing `false` tells the system not to auto-schedule another snapshot.
3. I'll bet this statement confuses you! :]
4. Finally, you mark the snapshot task as complete and specify `true` so that watchOS automatically schedules another snapshot for an hour from now.

When watchOS calls `handle(_:)`, you have a limited amount of time, on the order of seconds, to finish the task. If you neglect to mark the task as having completed, the system continues to run it in the background. Your task will then run until all available time has been consumed, which wastes battery power.

In step two, where you pass `false` to `setTaskCompletedWithSnapshot`, you might be wondering why you don't specify `true` instead. Since you took no action on the task, there's no reason to force an immediate snapshot to happen.

Bring up Xcode's Debug area by pressing **Shift-Command-C**. Then build and run your app.

Once the simulator is running your app, switch to the Home screen. Be patient — after an indeterminate, but likely short, amount of time, you'll see the message that your app took a snapshot.

Forcing a snapshot

When you switched to the Home screen, the snapshot task didn't immediately run because watchOS was busy performing other tasks. That might be OK for a normal production deployment, but when building the app, you'll want to be able to tell the simulator to take a snapshot *right now*.

Remember that snapshots will only occur if the app is *not* in the foreground. The simulator — or physical device — must be showing any other app or the clock face. Of course, the app must be running via Xcode for you to see the fruits of your labor.

Once your app is running in the background, in *Xcode*'s menu bar, choose **Debug ▸ Simulate UI Snapshot**.

You might think you'd use the simulator to force a snapshot in it. Unfortunately, you'd be wrong. It's easy to get confused and wonder why you can't find the **Simulate UI Snapshot** menu bar item.

You'll see the message in the Debug area again, letting you know watchOS took a snapshot. Notice how nothing else on the watch face changes. Snapshots are a background task, meaning there's no reason for watchOS to bring your app to the user's attention.

Viewing a snapshot

You can see what your snapshot looks like by visiting the Dock. In the simulator, you get to the Dock in three separate ways:

1. Click the Side button in the simulated Apple Watch display.
2. Click **Device ▸ Side Button** in the simulator's menu bar.
3. Press **Shift-Command-B** on your keyboard. Scholars have pondered for ages why it's not **Shift-Command-D**. :]

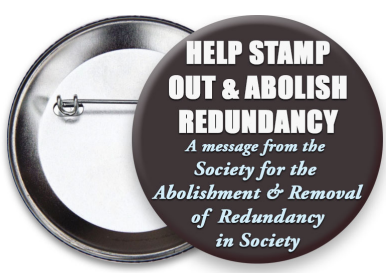
After bringing up the dock, you'll see that your app is the first on the list:

That doesn't give you a great view of your snapshot, though. Run any other app, and then jump back to the dock. Now, you'll have a better view of your snapshot:

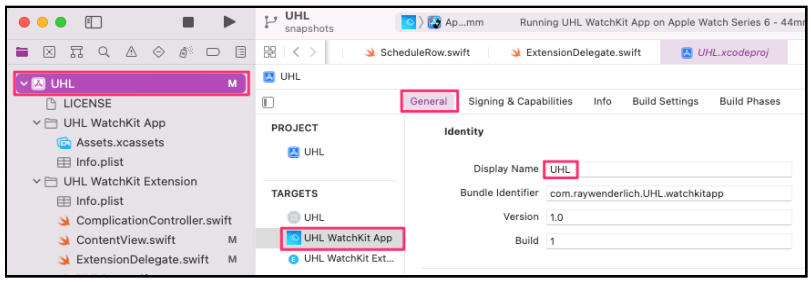
Customizing the app name

Most of the time, the display name of your WatchKit app doesn't matter: Icons on the Home screen don't display a label like they do in iOS.

The Dock, however, does display the name. As you can see in the previous images, Xcode defaults the name of the app to "**[app name] WatchKit App**". When you see a name like that in the Dock, it just feels a bit redundant.



In the project navigator, which you can access by pressing **Command-1**, select the **UHL WatchKit App** target. Then click the **General** tab, and change the **Display Name** to **UHL**:



Build and run the app. Then switch back to the Dock. Now you see a much better name displayed:

Customizing the snapshot

As previously discussed, your app's snapshot defaults to the last screen which was visible. That's not helpful for Octopi fans.

If the last match played occurred yesterday or today, you should show them the score of that match. If, instead, the next match will occur today or tomorrow, show that match's schedule. Otherwise, take a new snapshot two days later.

Switch to **ExtensionDelegate.swift** and add the following method:

```
private func nextSnapshotDate() -> Date {
  // 1
  guard let nextMatch = Season.shared.nextMatch else {
    return .distantFuture
  }

  // 2
```

```
    let twoDaysLater = Calendar.current.date(
      byAdding: .day,
      value: 2,
      to: nextMatch.date
    )!

    // 3
    return Calendar.current.startOfDay(for: twoDaysLater)
  }
```

The preceding code accomplishes these three tasks:

1. If there's no upcoming match, you return `.distantFuture` for the date. Using `.distantFuture` essentially tells watchOS that there are no more scheduled snapshots.
2. Using calendrical calculations, you determine what the date will be in two days. This calculation can't fail, so it's safe to force unwrap the return value.
3. Finally, you determine the start of the day for when the snapshot will happen.

Snapshot user info

The Snapshot API is based around a user info type dictionary. While you *could* use a dictionary, there's a better way to handle the data. You don't want to have to hardcode strings for the keys or define global `let` type constants. Instead, create a new Swift file named **SnapshotUserInfo.swift**.

When a snapshot happens, you have to switch the app to the appropriate screen depending on the rules previously defined. The sample app already has much of the code implemented to handle that task for you. The goal here is to learn about snapshots, not the craziness that is controlling which SwiftUI view gets automatically pushed onto the navigation stack.

Start with:

```
import Foundation

struct SnapshotUserInfo {
  // 1
  let handler: () -> Void
  // 2
  let destination: ContentView.Destination
  // 3
  let matchId: Match.ID?
}
```

`SnapshotUserInfo` implemented this so far:

1. You have to tell the snapshot when it's completed. More on that in a moment.
2. `ContentView.Destination` is an `enum` identifying which view will be pushed onto the navigation stack.
3. You need to identify a match to snapshot.

Look in **Match.swift** located in **Model** group. You'll see that the `Identifiable` protocol is implemented, and the type of the identifier is a `UUID`. While you could use a `UUID` type throughout the app, it's a better idea to reference `Match.ID` instead. Why? If at a later date you realize you need the identifier to be an `Int` instead, there's only a single location you need to update.

Next, implement the initializer for `SnapshotUserInfo`:

```
init(
  handler: @escaping () -> Void,
  destination: ContentView.Destination,
  matchId: Match.ID? = nil
) {
  self.handler = handler
  self.destination = destination
  self.matchId = matchId
}
```

While not technically required, implementing the initializer lets you initialize the structure instance without having to explicitly specify `nil` for the match.

Now you need a way to generate the dictionary that the Snapshot API wants. Add:

```
// 1
private enum Keys: String {
  case handler, destination, matchId
}

// 2
func encode() -> [AnyHashable: Any] {
  return [
    Keys.handler.rawValue: handler,
    Keys.destination.rawValue: destination,
    // 3
    Keys.matchId.rawValue: matchId as Any
  ]
}
```

Here's a code breakdown:

1. You define an `enum` to store the keys to the dictionary. You could also use `let` strings, but the `enum` is cleaner for this use case.
2. You create a method that encodes the `struct` into the type of dictionary the Snapshot API requires.
3. The match identifier could be `nil`, so you must explicitly add the `as Any` cast to make the compiler happy.

You'll also need a way to convert from the API's dictionary into the object you defined. Add a custom error type right outside the `struct`:

```
enum SnapshotError: Error {
  case noHandler, badDestination, badMatchId, noUserInfo
}
```

Then inside the `struct`, implement the method to convert from the API's dictionary information:

```
// 1
static func from(notification: Notification) throws -> Self {
  // 2
  guard let userInfo = notification.userInfo else {
    throw SnapshotError.noHandler
  }

  guard let handler = userInfo[Keys.handler.rawValue] as? () ->
Void else {
    throw SnapshotError.noHandler
  }

  guard
    let destination = userInfo[Keys.destination.rawValue] as?
ContentView.Destination
  else {
    throw SnapshotError.badDestination
  }

  // 3
  return .init(
    handler: handler,
    destination: destination,
    // 4
    matchId: userInfo[Keys.matchId.rawValue] as? Match.ID
  )
}
```

Here's what's happening:

1. Using a `static` method allows for a factory pattern type of creation.
2. You pull the individual pieces out of the posted `Notification`, throwing an appropriate error if anything goes wrong.
3. Then, you create and return a properly initialized object.
4. Similarly to the other method, using an `as? Match.ID` cast handles the case when no match identifier is provided.

Viewing the last game's score

Sometimes, even the best of fans will miss a game. If the most recent match was yesterday or today, wouldn't it be great if you showed the score?

In **Season.swift**, find the following line of code:

```
  bySettingHour: Int.random(in: 18 ... 20),
```

Update the values to be something earlier in the day than the current hour. You want to make it appear like a match already happened today.

Add the following method to the **ExtensionDelegate** class:

```
  private func lastMatchPlayedRecently() -> Bool {
    // 1
    guard let last = Season.shared.pastMatches().last?.date else {
       print("No last date")
      return false
    }

    print("The date is \(last.formatted()) and now is \
(Date.now.formatted())")

    // 2
    return Calendar.current.isDateInYesterday(last) ||
           Calendar.current.isDateInToday(last)
  }
```

Here's what's happening:

1. If there's a previously played match, grab the date. If not, simply return `false`.
2. Determine if the last match occurred yesterday or today.

> **Note**: Remember to always use the calendrical calendar methods provided by Apple. Adding 3,600 seconds to the date is always the wrong thing to do!

In the `handle(_:)` method, delete the final line which completes the snapshot and replace it with:

```
let nextSnapshotDate = nextSnapshotDate()

let handler = {
  snapshot.setTaskCompleted(
    restoredDefaultState: false,
    estimatedSnapshotExpiration: nextSnapshotDate,
    userInfo: nil
  )
}
```

`setTaskCompletedWithSnapshot(_:)` is a convenience method for the longer `setTaskCompleted(restoredDefaultState:estimatedSnapshotExpiration:userInfo)`. Try saying that one three times fast!

- When watchOS takes the snapshot, you tell it to complete the task. You pass `false` to the first parameter because you left the app in a different state than it was originally in.
- The second parameter, `estimatedSnapshotExpiration`, tells watchOS when the current snapshot is no longer valid. Essentially, the date you provide is when it needs to take a new snapshot.
- Finally, for the `userInfo` parameter, simply pass `nil` as you don't need the next snapshot to know anything about the app's current state. If you did, `userInfo` is where you could pass something along.

Notice how you've assigned that completion call to a `handler` variable. You don't want to *complete* the task until you're ready for watchOS to take the snapshot, which means you first have to get the views pushed onto the navigation stack. That's why your `SnapshotUserInfo` has a `handler` property.

Continue adding to `handle(_:)`:

```
// 1
var snapshotUserInfo: SnapshotUserInfo?

// 2
if lastMatchPlayedRecently() {
  snapshotUserInfo = SnapshotUserInfo(
```

```
      handler: handler,
      destination: .record
    )
  }

  // 3
  if let snapshotUserInfo = snapshotUserInfo {
    NotificationCenter.default.post(
      name: .pushViewForSnapshot,
      object: nil,
      userInfo: snapshotUserInfo.encode()
    )
  } else {
    // 4
    handler()
  }
```

Lots going on with that code:

1. You create a variable of type `SnapshotUserInfo`, which is currently not set.
2. If there's been a recent match, you populate that variable with the `handler` you just defined and specify that the record view is what should display in the snapshot.
3. If there's data for the `snapshotUserInfo`, you post a notification with that object. Notice how the `userInfo` of the notification is the encoded object.
4. If not, then you simply call the handler to complete the task.

The sample project includes a file called **Notification.Name+Extension** which defines the notification type for you.

Putting the details into a `struct` instead of just populating the `userInfo` dictionary directly makes the code much cleaner. By abstracting away the data's details, it also becomes much easier to test.

Phew! That's quite a bit of code. Press **Command-B** to do a quick build of your project to make sure you haven't missed anything. It'll compile cleanly with neither warnings nor errors at this point.

When the system wants to take a snapshot, and there's a recently played match, you now post a notification with the details. When the notification happens, the app needs to navigate to `RecordView`.

Edit **RecordView.swift** located in **Record** and add a new property to the `RecordView` structure:

```
let snapshotHandler: (() -> Void)?
```

Then update the `PreviewProvider` appropriately:

```
RecordView(snapshotHandler: nil)
```

Remember that the handler you created in `ExtensionDelegate` needs to be called when the snapshot is ready to be taken.

When `RecordView` is the active view, then, and only then, can you complete the background task. watchOS 8 added a wonderful new `View` method called `task`, which is perfect for your needs. The code in the `task` block will run one time when the view appears and will cancel when the view goes away.

Add a call to the snapshot handler to the `List` view:

```
var body: some View {
  List(season.pastMatches().reversed()) {
    ...
  }
  ...
  .task {
    snapshotHandler?()
  }
}
```

Once the view appears, if `snapshotHandler` has a value, the method will be called, appropriately signifying to the snapshot task that it is complete.

Finally, edit **ContentView.swift** and add two new properties:

```
// 1
@State private var snapshotHandler: (() -> Void)?

// 2
private let pushViewForSnapshotPublisher = NotificationCenter
  .default
  .publisher(for: .pushViewForSnapshot)
```

For those two properties:

1. Since `ContentView` is the top of your navigation stack, the property you store the snapshot handler in has to be `@State` as this is the view that will assign the value based on the notification.
2. Taking advantage of Combine makes responding to notifications quite simple in SwiftUI.

Then, pass the handler to `RecordView` in the `NavigationLink`, replacing:

```
NavigationLink(
  destination: RecordView(),
```

With:

```
NavigationLink(
  destination: RecordView(snapshotHandler: snapshotHandler),
```

Next, add a new method to handle when the notification is called:

```
private func pushViewForSnapshot(_ notification: Notification) {
  // 1
  guard
    let info = try? SnapshotUserInfo.from(notification:
notification)
  else {
    return
  }

  // 2
  snapshotHandler = info.handler
  selectedMatchId = info.matchId

  // 3
  activeDestination = info.destination
}
```

For `pushViewForSnapshot(_:)`:

1. You pull the `SnapshotUserInfo` details from the notification you posted in `ExtensionDelegate`.
2. Then, you extract the handler and the specified match into local variables for ease of use.
3. You specify the `activeDestination` based on what you set in the notification.

Controlling a navigation stack is still ridiculously complex in SwiftUI. As mentioned earlier, that's outside the scope of this book. But if you take a look at `NavigationLink`'s construction and the provided `isDestinationActive(_:)` method, you'll get a good idea of how to make it work.

All that's left to do is "catch" the notification. After the second `NavigationLink`, before closing the `VStack`, call your helper method when a notification appears:

```
.onReceive(pushViewForSnapshotPublisher) {
  pushViewForSnapshot($0)
}
```

> **Note**: There are multiple ways to catch and handle a notification in SwiftUI. Use whatever method feels most natural to you.

At this point, build and run the app. Perform the following steps to test all your hard work:

1. Ensure you've navigated to the first screen of the app.
2. Change to the Home screen.
3. In Xcode's menu bar, click **Debug ▸ Simulate UI Snapshot**.
4. In the simulator, bring up the Dock.

You'll now see that your app did move to the proper view before taking a snapshot.

Wow, that was a ton of code! Take a quick break if you need to, and then it'll be time to handle upcoming matches.

Viewing upcoming matches

If there's no recent match, there might be a pending one. Edit **ExtensionDelegate.swift** and add another method:

```
private func idForPendingMatch() -> Match.ID? {
  guard let match = Season.shared.nextMatch else {
    return nil
  }

  let date = match.date
  let calendar = Calendar.current

  if calendar.isDateInTomorrow(date) ||
calendar.isDateInToday(date) {
    return match.id
  } else {
    return nil
  }
}
```

The code is pretty self-explanatory. If there's a match today or tomorrow, return the match's identifier. Otherwise, return `nil`.

Find `if`, where you check for a recently played match:

```
if lastMatchPlayedRecently() {
  snapshotUserInfo = SnapshotUserInfo(
    handler: handler,
    destination: .record
  )
}
```

And add `else` branch with print statements, so it'll take the following form:

```
if lastMatchPlayedRecently() {
  print("Going to record")
  snapshotUserInfo = SnapshotUserInfo(
    handler: handler,
    destination: .record
  )
} else if let id = idForPendingMatch() {
  print("Going to schedule")
  snapshotUserInfo = SnapshotUserInfo(
    handler: handler,
    destination: .schedule,
    matchId: id
  )
}
```

When the notification happens, the app needs to navigate to `ScheduleView` and then `ScheduleDetailView` for the specified `matchId`. That's a bit complicated in SwiftUI, but the code is already there for you. You just need to add in the snapshot specific details now.

Working backward, to help avoid compiler errors while you're coding, first edit **ScheduleDetailView.swift**. You'll perform similar steps to what you did for the record view. Add a new property:

```
let snapshotHandler: (() -> Void)?
```

Then calls the snapshot handler in a `.task`:

```
VStack {
  ...
  VStack {
    ...
  }
}
.task {
  snapshotHandler?()
}
```

And update the `PreviewProvider`:

```
ScheduleDetailView(
  match: Season.shared.nextMatch!,
  snapshotHandler: nil
)
```

In your best pirate voice, think, "Here be compiler errors."

Now that you've added a new property to `ScheduleDetailView`, you'll need to edit **ScheduleView.swift** to pass in the handler. Add the same property to the top of `ScheduleView`:

```
let snapshotHandler: (() -> Void)?
```

Then add it as a parameter to the initializer for `ScheduleDetailView`, replacing:

```
destination: ScheduleDetailView(
  match: match
),
```

With:

```
destination: ScheduleDetailView(
```

```
    match: match,
    snapshotHandler: snapshotHandler
  ),
```

And, of course, update the `PreviewProvider`:

```
ScheduleView(selectedMatchId: .constant(nil), snapshotHandler:
nil)
```

Finally, back in **ContentView.swift**, add the new parameter to the call to `ScheduleView`:

```
destination: ScheduleView(
  selectedMatchId: $selectedMatchId,
  snapshotHandler: snapshotHandler
),
```

Build and run the app. This time, tap the first button to navigate to the upcoming matches view. Swipe left on a row, and you'll see a button to add a match as well as delete the current match:

Tap the + button to create a random match that occurs today. Also, the most recent match which occurred earlier today or yesterday is removed to ensure that the record view doesn't appear.

Force another snapshot as you did for the record view:

1. Ensure you've navigated to the first screen of the app.
2. Change to the Home screen.
3. In Xcode's menu bar, click **Debug ▸ Simulate UI Snapshot**.
4. In the simulator, bring up the Dock.

You'll see that you now display the schedule view in the Dock:

Key points

- Make sure you always mark background tasks as completed as soon as possible. If you don't, you'll waste the user's battery.
- Snapshots are smaller than your app. Consider bolding text, removing images or making other minor changes to increase the information displayed.

Where to go from here?

The chapter's sample code includes a **ModifiedRecordView.swift**, which shows an example of how you might detect that a snapshot is about to happen so that you can present a different view entirely if that makes sense for your app.

Now you're a Dock connoisseur, and you've gained a deep understanding of how watchOS uses snapshots. You also learned how to modify snapshots based on contextual relevance.

If you'd like to take this further, try tweaking the size or color of objects in the snapshot. Remember that the snapshot is miniaturized, so you might want to make the text more readable.

Chapter 6: Notifications

By Scott Grosch

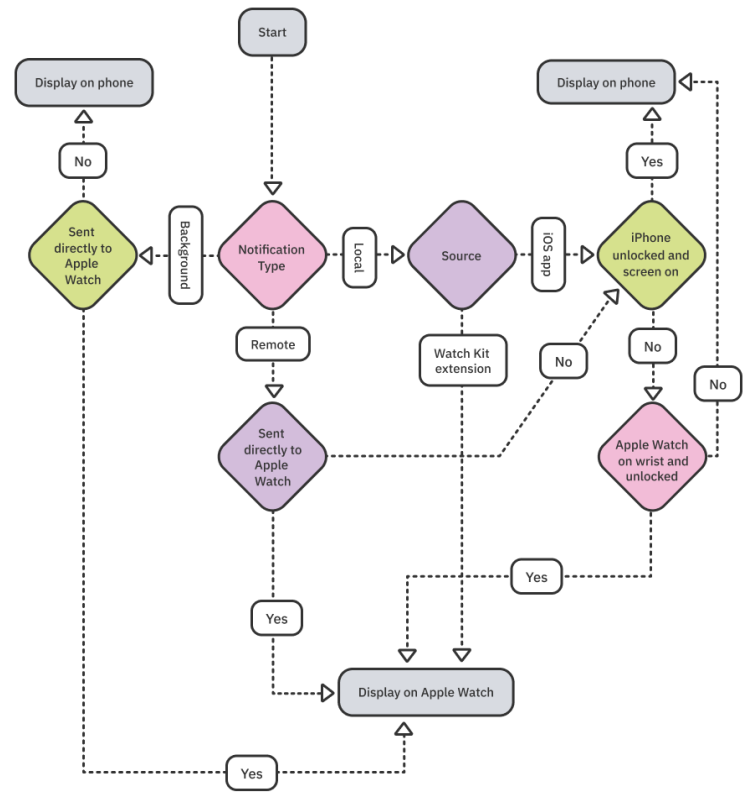
Local and remote notifications are a great way to inform your users about new or relevant information. There may be new content available, it might be their turn in the game or they may have just won the lottery. If your app has an accompanying iPhone app that supports notifications, by default, your Apple Watch will display the notification when appropriate. However, you can do better!

Notification is a massive topic that could fill a book all by itself. This chapter will focus solely on some of the differences you must be aware of when working with watchOS. You can learn everything you need to know about push notifications in our book, *Push Notifications by Tutorials (*https://bit.ly/3soGsr8*)*, which is available as part of the professional subscription.

Where did it go?

Apple tries to determine the best target device to receive a notification. If you only have an Apple Watch, it'll go there. However, if you use a watch and another device, the destination depends not only on the type of notification but also on its source.

The diagram below will help you understand how Apple chooses which device should display the notification. As you can see, the notification type, whether it is Local, Remote or a Background notification, will define where it should go. For the first two options, local and background, it will prioritize the Apple Watch. Local Notifications on the other hand will prioritize depending on the source. Check the following image to see the different paths:



You'll notice two locations in the diagram where it asks if Apple sent the notification directly to the watch. In watchOS 6 and later, the Apple Watch is a valid target for remote and background notifications. The Apple Watch extension receives a unique device token when registering for remote notifications, just like in iOS.

Short looks

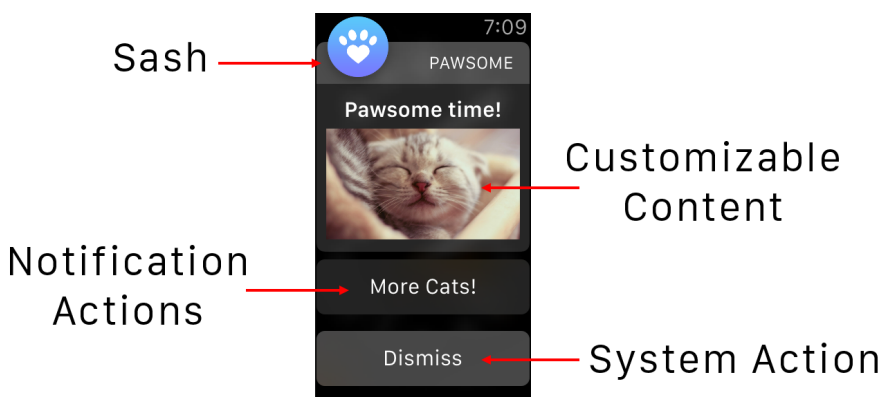
When the Apple Watch receives a notification, it notifies the user via a subtle vibration. If the user views the notification by raising their wrist, the Apple Watch shows an abbreviated version called a **short look**. If the user views the notification for more than a split second, the Apple Watch will offer a more detailed version, or **long look**.

The short look notification is a quick summary for the user. Short looks show the app's icon and name, as well as the optional notification title, in a predefined layout:

The optional notification title is a short blurb about the notification, such as "New Bill", "Reminder" or "Score Alert", and is added to the `alert` key's value. This lets the user decide whether to stick around for the long look interface.

Long looks

The long look is a scrolling interface you can customize, with a default static interface or an optional dynamically-created interface. Unlike the short look interface, the long look offers significant customization.



The **sash** is the horizontal bar at the top. It's translucent by default, but you can set it to any color and opacity value.

You can customize the content area by implementing a SwiftUI `View`, which you'll learn about later.

While you can implement several `UNNotificationAction` items, remember that more than a few will require quite a bit of scrolling on the user's part, leading to a poor user experience.

The system-provided Dismiss button is always present at the bottom of the interface. Tapping Dismiss hides the notification without informing the Apple Watch extension.

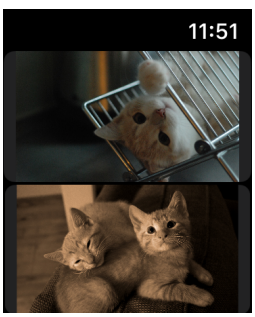
Now that you know about the short and long look notifications, it's time to put the theory into practice.

Local notifications

Pawsome is for all cat lovers who procrastinate during the day by looking at cute cat pictures. The Pawsome app will make this easier by interrupting you throughout the day with cute cat pictures that are certain to trigger a smile… unless you're a dog person!

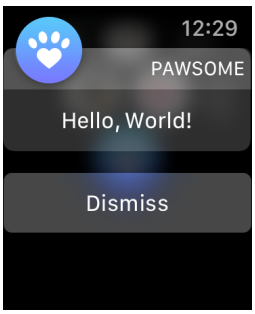
Getting started

Open the Pawsome starter project in Xcode. Then build and run the **Pawsome WatchKit App** scheme. You'll see a collection of cute kitty cats that you can easily browse:



Testing notifications with the simulator

Switch to the **Pawsome WatchKit App (Notification)** scheme and rerun the project. This time, instead of seeing those adorable cats, you'll see a pretty boring notification:



It's not the *purrrtiest*, but you'll soon fix that.

Take a look at **LocalNotifications.swift**, and you'll see the code that creates and schedules your local notifications. There's nothing specific to watchOS, which is why I provided that file for you. At the top of the class, you'll find `categoryIdentifier`. When a notification triggers, that's the identifier you'll use.

Open **PawsomeApp.swift**. You'll see the following line, which Xcode adds if you say you want to include a notification when you create your project:

```
WKNotificationScene(
  controller: NotificationController.self,
  category: "myCategory"
)
```

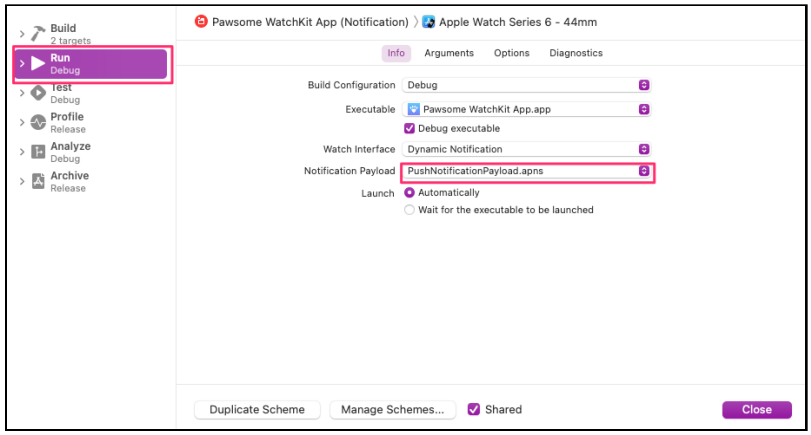
Calling `WKNotificationScene` is how you tell the Apple Watch what view to display for each category identified in your payload. Replace the category with the one from `LocalNotifications`:

```
WKNotificationScene(
  controller: NotificationController.self,
  category: LocalNotifications.categoryIdentifier
)
```

Build and run the app. You'll see this notification:

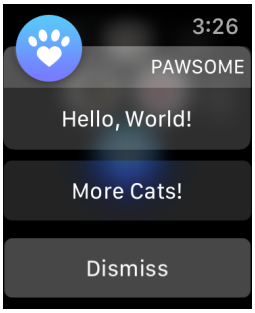
What happened? The `NotificationView` wasn't updated, so why the change?

If you look at the **Run** step of the **Pawsome WatchKit App (Notification)** scheme, you'll see the notification payload is set to a file in your project, called **PushNotificationPayload.apns**:



Take a look at **PushNotificationPayload.apns**. If you've worked with push notifications at all, this should look familiar to you. The **category** sent with the notification is set to **myCategory**. However, you updated **PawsomeApp.swift** to respond to a different category name. When the category sent to the app doesn't match something you've registered for, it configures a default display.

Change **myCategory** to **Pawsome**, which is the value of `LocalNotifications.categoryIdentifier`.
Run the app again. This time you'll see:



Where did the "More Cats!" button come from? **PawsomeApp.swift** creates an instance of `LocalNotifications`, which creates a default action button in its initializer.

Since the category in the JSON matches what you specified in `WKNotificationScene(controller:category:)`, watchOS created an instance of `NotificationController` and used that to display the notification. Look closely, and you'll notice the title and body are missing from the displayed view. Time to fix that!

Custom long look notification

Edit **NotificationController.swift**, and you'll see `body` returns an instance of `NotificationView`. The *controller* is where you receive and parse the notification. The *view* is then where you use the data gathered by the controller.

Switch over to **NotificationView.swift** to make the notification appear the way you want. Replace the entire contents of the default file with:

```
import SwiftUI

struct NotificationView: View {
  // 1
  let message: String
  let image: Image

  // 2
  var body: some View {
    ScrollView {
      Text(message)
        .font(.headline)

      image
        .resizable()
        .scaledToFit()
    }
  }
}

struct NotificationView_Previews: PreviewProvider {
  static var previews: some View {
    // 3
    NotificationView(
      message: "Awww",
      image: Image("cat\(Int.random(in: 1...20))")
    )
  }
}
```

The code is pretty straight-forward:

1. You need to provide a message and an image for the view to display.
2. The body simply displays those two properties in a scrolling list.
3. A random image is chosen from the asset catalog for the previews.

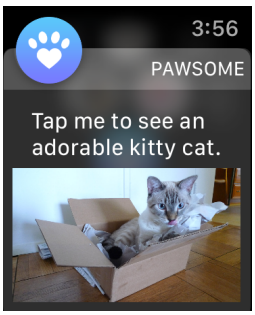
Now that you've created a view to display when a notification arrives, it's time to use it. Head back to **NotificationController.swift** and replace the contents of the class with:

```
// 1
var image: Image!
var message: String!

// 2
override var body: NotificationView {
  return NotificationView(message: message, image: image)
}

// 3
override func didReceive(_ notification: UNNotification) {
  let content = notification.request.content
  message = content.body

  let num = Int.random(in: 1...20)
  image = Image("cat\(num)")
}
```

Like all good code, the controller is short and sweet. It only has a few steps:

1. You store the title and image so you can send them to the view.
2. Then you call the initializer for the view you're going to display, passing in the appropriate parameters.
3. You pull out the details from the payload body, which you then store in the class' properties.

Build and run again, trying to limit the immense feelings of joy caused by the display:



You probably want to display a specific cat, not a random one. Replace the last two lines of `didReceive(_:)` with:

```
let validRange = 1...20

if
  let imageNumber = content.userInfo["imageNumber"] as? Int,
  validRange ~= imageNumber {
  image = Image("cat\(imageNumber)")
} else {
  let num = Int.random(in: validRange)
  image = Image("cat\(num)")
}
```

The asset catalog provided with the starter project has images numbered from one to twenty. If the payload includes an image number in that range, you display the specified cat photo. If not, you provide a random image.

> **Note**: Since the view will always be generated, you need to ensure that valid data is always available, even if it means presenting a default set of data.

Add the following line right before the final closing } character in **PushNotificationPayload.apns**:

```
, "imageNumber": 5
```

Build and run again. You'll see **cat5** from the asset catalog.

Remote push notifications

Most apps use push notifications, not local notifications. You're probably wondering why I spent all that time on something that's used less frequently. Well, the answer is that Apple made push notifications much easier on watchOS than they are on iOS.

In iOS, you have to create an extension to modify the incoming payload and yet another extension if you want a custom interface. Worse yet, you can't perform the custom interface with SwiftUI as of iOS 15. In watchOS, push notifications work exactly like local notifications!

Everything you learned about using `WKUserNotificationHostingController` to parse the payload and return a custom SwiftUI `View` works the same when you're developing push notifications.

Token creation

This chapter assumes you already created your push notification token, a file you download from the Apple developer portal that ends with a **p8** extension. If you need help generating your token, please see our *Push Notifications by Tutorials (*https://bit.ly/3soGsr8*)* book.

Don't worry if you can't create a token file. For example, you might not have permission to create a token in your team's portal. This chapter will use the simulator and a JSON file to simulate receiving a remote push notification.

Create a WKExtensionDelegate

In iOS, you register for push notifications using `AppDelegate`. That class doesn't exist on watchOS. Instead, you use `WKExtensionDelegate`. Create a new file called **ExtensionDelegate.swift** and paste:

```
import WatchKit
import UserNotifications

// 1
final class ExtensionDelegate: NSObject, WKExtensionDelegate {
  // 2
  func didRegisterForRemoteNotifications(withDeviceToken
deviceToken: Data) {
    print(deviceToken.reduce("") { $0 + String(format: "%02x",
$1) })
  }
```

```
  // 3
  func applicationDidFinishLaunching() {
    async {
      do {
        let success = try await UNUserNotificationCenter
          .current()
          .requestAuthorization(options:
[.badge, .sound, .alert])

        guard success else { return }

        // 4
        await MainActor.run {
          WKExtension.shared().registerForRemoteNotifications()
        }
      } catch {
        print(error.localizedDescription)
      }
    }
  }
}
```

Here's a code breakdown:

1. You declare a class that implements `WKExtensionDelegate`. Since that protocol is based on `NSObjectProtocol` you also need to derive from `NSObject`.
2. Just like in iOS, you grab the `deviceToken` whenever registration happens. A production app would, of course, store and use the token, not just print it.
3. Surprisingly not named `extensionDidFinishLaunching`. You do the standard dance in this method to request permission to use push notifications.
4. If permissions are granted, you use the `WKExtension` singleton to register for push notifications, which call `didRegisterForRemoteNotifications(withDeviceToken:)` if successful.

To tell watchOS it should use your `ExtensionDelegate`, add the following two lines to the top of the `struct` in **PawsomeApp.swift**:

```
@WKExtensionDelegateAdaptor(ExtensionDelegate.self)
private var extensionDelegate
```

The MVC of push notifications

Instead of making you copy and paste a ton of code, I've provided a **Remote Notifications** group in the starter project, which contains the relevant files for a push notification.

Generally, you'll want to use some type of model to represent the data that will pass between your `WKUserNotificationHostingController` and `View`. While you could use individual properties, as in the preceding examples for local notifications, it's better to use a real model, such as the one provided in **RemoteNotificationModel.swift**.

Look at **RemoteNotificationView.swift**, and you'll see it's just a simple setup that shows a small number of details by default. If you tap the toggle, it displays more details. Remember, unlike the iPhone, the Apple Watch has limited display space. You'll need to think differently about how you present notification data to the user.

Next, open **RemoteNotificationController.swift**. Even though you're working with a remote push notification, you'll see everything works the same as when you implemented local notifications. Pay special attention to the `guard` statement.

In an ideal world, the payload provided to your app would always be 100% perfect. However, we don't live in an ideal world, so it's important to always validate the input. If anything goes wrong, you *still* have to provide a model for the notification to display. Don't let your app crash because of bad data!

Add the capability

Xcode will perform magic if you add the **Push Notifications** capability. In the Project navigator, **Command-1**, select the project name, **Pawsome**. Then on the right, in the project editor, select the extension target. Make sure you choose the extension, not the app. Generally, that'll be the last target listed.

On the **Signing & Capabilities** tab, add the **Push Notifications** capability. If you try to add a capability and nothing seems to show, that generally means you haven't chosen the extension.

> **Note**: Xcode does not generate a valid identifier for watchOS push notifications!

Years ago, Apple finally automated almost everything around certificate and identifier generation. Unfortunately, for watchOS push notifications, it broke things.

When sending a push notification directly to the Apple Watch, you need to use the app's bundle identifier, not the extension's bundle identifier. In the case of Pawsome, that means you send your push to **com.raywenderlich.Pawsome.watchkitapp** and not **com.raywenderlich.Pawsome.watchkitapp.watchkitextension**. Unfortunately, because you have to add the capability to the extension, the generated profile thinks you're using the latter identifier, not the former.

The solution, while annoying to have to deal with, is quite simple. Go into your team's Certificates, Identifiers & Profiles (https://apple.co/31pwmKV) page on the Apple Developer Portal and manually create an identifier for your watchOS app. Use the app identifier **com.raywenderlich.Pawsome.watchkitapp** and add Push Notification support.

> **Note**: Replace **raywenderlich** in the app identifier with your team's name.

Add a scheme

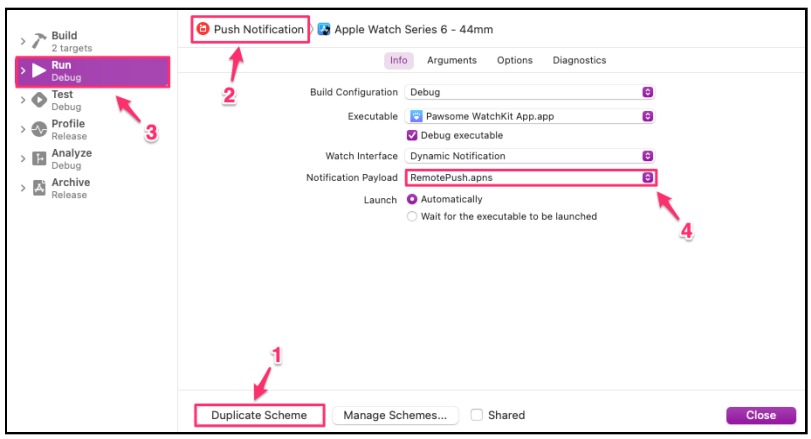
Edit the current scheme via your preferred method. I like to press the **Command+<** keyboard shortcut.

The default project Xcode created includes a notification scheme you used to test the local notifications. You'll need another scheme to test the remote notification because the payload is a different file. You could, of course, just change the file on the existing scheme if you wanted to.

> **Note**: If your app supports more than one notification, you can add multiple APNS files and multiple schemes to make it easy to test each one.

Follow these steps to configure a new scheme:

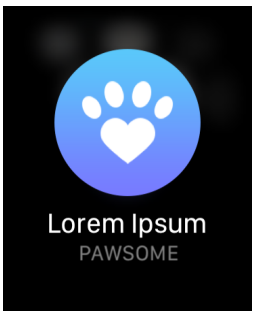
1. With the notification scheme still chosen, press **Duplicate Scheme** at the bottom of the dialog window.
2. Choose a name for the new scheme, such as **Push Notification**.
3. Click **Run** on the left side of the dialog window.
4. Select **RemotePush.apns** as the notification payload to use.

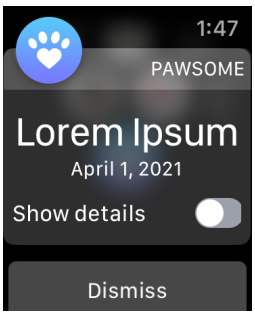


You'll find **RemotePush.apns** in **Remote Notifications**. It contains a simple example remote push notification payload:

```
{
  "aps": {
    "alert": {
      "body": "Lorem ipsum dolor sit amet, consectetur...",
      "title": "Lorem Ipsum",
    },
    "category": "lorem"
  },

  "date": "2021-04-01T12:00:00Z"
}
```

Build and run. You probably expected to see the following short and long look notifications:





Take a moment to try and figure out why you see something different.

Remember that watchOS has no way of associating a payload to a controller if you don't link them. Go back into **PawsomeApp.swift** and add the following statement:

```
WKNotificationScene(
  controller: RemoteNotificationController.self,
  category: RemoteNotificationController.categoryIdentifier
)
```

Build and run again. You'll now see the proper notification screen.

Interactive notifications

Tap **Show details**. Did something unexpected happen? An average user would expect to see details. Instead, you were taken into the app and shown a cat picture. Surprisingly, that's by design.

By default, push notifications are not interactive. As far as the Apple Watch knows, anything that's not one of the action buttons is just an image on the screen.

Adding an action button wouldn't make sense to show more details. Instead, add the following line to **RemoteNotificationController**:

```
override class var isInteractive: Bool { true }
```

Build and run again. This time, when you tap the toggle, the details appear and disappear as you'd expect them to:

The `isInteractive` type property of `WKUserNotificationHostingController<T>` specifies whether the notification should accept user input. The default value is `false`, meaning you can only interact with buttons. By changing the value to `true`, you tell watchOS the notification should accept user input.

You solved one problem but might have introduced another. Tapping no longer opens the app. If the user taps the app icon or anywhere on the sash, the app will still open.

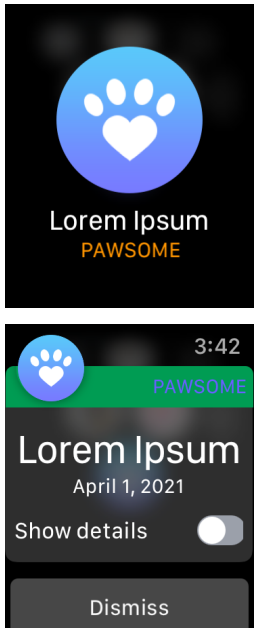
Styling

If the default colors don't work well with your app's theme, you can override a few properties to perform minimal customization.

Edit **RemoteNotificationController.swift** and add the following lines at the end of the class:

```
override class var sashColor: Color? {
  Color(red: 0, green: 156 / 255, blue: 83 / 255)
}
override class var titleColor: Color? { Color.purple }
override class var subtitleColor: Color? { Color.orange }
override class var wantsSashBlur: Bool { true }
```

Build and run again. You'll notice the colors have changed to something hideous:



The only short look customization you make is changing the subtitle to orange, as specified by `subtitleColor`.

For the long look, you made three changes. You set the sash color to the official Razeware green via `sashColor` and modified the title text inside the sash to purple via `titleColor`.

`wantsSashBlur`, which defaults to `false`, determines whether the sash includes a blur over the background.

Great work! Now you know how to add custom notification interfaces to your Watch apps.

Key points

- How Local & Remote Notifications work with Apple Watch.
- Short & Long looks and how to customize them.
- Testing Push Notifications on Apple Watch

Where to go from here?

In this chapter, you tested Watch notifications, learned about short look and long look interfaces and how they differ. Most impressively, you built a custom, dynamically updating, long look local notification for the Apple Watch.

Now you know the basics of showing custom notifications on watchOS, but there's a lot more you can do from here, including handling actions selected by users from your notifications. Please see Apple's User Notifications (https://apple.co/3hFtwKR) documentation as well as our *Push Notifications by Tutorials (*https://bit.ly/3soGsr8*)* book, which is available as part of the professional subscription.

For more details on creating schemes and JSON payloads, as well as testing directly on the watch, please see Testing Custom Notifications (https://apple.co/3tTDE5F) in Apple's developer documentation.

Chapter 7: Lifecycle

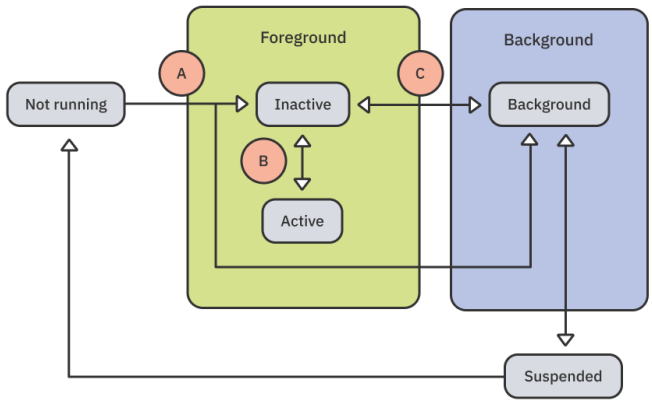
By Scott Grosch

The lifecycle of a watchOS app is a bit more complex than that of iOS apps. There are five possible states that a watchOS app may find itself in:

- Not running
- Inactive
- Active
- Background
- Suspended

Common state transitions

The five possible states that a watchOS app finds itself in are represented by the grey boxes in the following image:



As a developer, you'll only interact with three of the states. The *not running* and *suspended* states are only active, pun intended :], when your app is not running.

Launching to the active state

If the user has yet to run the app or the system purged the app from memory, the app begins in the *not running* state. Once the app launches, it follows path *A* and transitions to the *inactive* state.

While in the inactive state, the app still runs in the foreground but doesn't receive any actions from the user. However, the app may still be executing code. Don't let the name trick you into thinking the app can't work while in this state.

Almost immediately, the app will follow path B and transition to the *active* state, which is the normal mode for apps running on the screen. When in the active state, the app can receive actions from the physical controls on the Apple Watch and user gestures.

When the user starts your app, and it's not already running, watchOS will perform these actions:

- Call `applicationDidFinishLaunching()` and set the `scenePhase` to `.inactive`
- Create the app's initial scene and the root view.
- Call `applicationWillEnterForeground()`.

- Call `applicationDidBecomeActive()` and set the `scenePhase` to `.active`
- The app will appear on the screen, and then watchOS will call the root view's `onAppear(perform:)`.

> **Note**: As of watchOS 7 and later, the `scenePhase` environment variable is preferred over the `WKExtensionDelegate` methods when using SwiftUI.

> **Note**: watchOS will not call any `WKExtensionDelegate` lifecycle methods other than `applicationDidFinishLaunching()` for events unrelated to the main interface. Things like complications and notifications will not trigger lifecycle events.

Transitioning to an inactive state

As soon as the user lowers their arm, they're no longer actively using the app. At that point, watchOS will change to the *inactive* state. As previously mentioned, the app is still running and executing your code. Having your app still running, even though it's in an inactive state, is a point of confusion for many new watchOS developers.

The inactive state is a great time to reduce your app's impact on the Apple Watch's battery. You should pause any battery-intensive actions which don't need to continue. For example, you may be able to disable any active animations that are currently running.

You might also consider whether there's any active state you need to save. Does it make sense to save your **Core Data** stack? Should you write anything to `UserDefaults`?

While your immediate thought is likely yes, consider the usage model of your app. As soon as the user lifts their arm again, the app will become active. Therefore, making too many saves might have more of an impact on the battery's life.

When your app transitions to the inactive state, watchOS will set `scenePhase` to `.inactive` and then call `WKExtensionDelegate`'s `applicationWillResignActive()`.

Transitioning to the background

Two minutes after transitioning to the inactive state, or when the user switches to another app, your app will transition to the *background* state. By following the lower part of path *A*, you can also launch the app directly to the background mode via the system. Background sessions and background tasks will both launch an app directly to the background state.

The OS gives background apps a small but non-deterministic amount of time before the app suspends.

If your app has transitioned to the background state by following path *C*, you'll want to quickly perform whatever necessary actions to recreate the current app state.

You may use the SwiftUI `ScenePhase` or `WKExtensionDelegate`'s `applicationDidEnterBackground` to determine when your app has transitioned to the background.

When transitioning from the inactive state to the background state, watchOS will set `scenePhase` to `.background` and then call `applicationDidEnterBackground()`. Should it require more resources, watchOS will eventually suspend the app.

Returning to clock

Before watchOS 7, you could ask for eight minutes before your app transitioned to the background. Your app can no longer make that change because developers kept forgetting to set it back to two minutes.

Now the user can control the timeout by going to **Settings ▸ General ▸ Return to Clock** on the Apple Watch. There are three options that they can choose:

1. Always.
2. After 2 minutes.
3. After 1 hour.

The default is two minutes. The user may also set the time at an individual app level. As part of your documentation, you may wish to tell the user how to change the setting to one hour for your app if the expectation is that they'll frequently interact with your app outside of the two-minute timeout.

Additional background execution time

If the amount of work you need to perform when transitioning to the background takes more time than watchOS provides your app, you need to rethink what you're doing. For example, this is not the time to make a network call. If you've performed all the optimizations you can and still need a bit more processing time, you can call the `performExpiringActivity(withReason:using:)` method of the `ProcessInfo` class.

If called while your app is in the foreground, you'll get 30 seconds. If called when in the background, you'll receive ten seconds.

The system will asynchronously try to perform the block of code you provide to the `using` parameter. It will then return a boolean value, letting you know whether or not the process is about to suspend.

If you receive a value of `false`, then you may perform your activities as quickly as possible. On the other hand, if you receive a `true` value, the system won't provide you extra time and you need to stop immediately.

Note that just because the system lets you start your extra work, that doesn't mean it will give you enough time to complete it. If your block of code is still running, and the OS needs to suspend your app, then your block of code will be called a second time with the `true` parameter. Your code should be able to handle this cancellation request.

For example, you could check before each action whether watchOS has told you to stop your work. Assuming you had a boolean instance property `cancel`, you might do something like this:

```
processInfo.performExpiringActivity(
  withReason: "I'm really slow"
) { suspending in
  // 1
  guard !suspending else {
    cancel = true
    return
  }

  // 2
  guard !cancel else { return }
  try? managedObjectContext.save()

  // 3
  guard !cancel else { return }
  userDefaults.set(someData(), forKey: "criticalData")
}
```

In the preceding code, you:

1. Immediately check if you're allowed to run. If the system tells you to suspend, then you set your `cancel` property to true.
2. Before trying to save your Core Data model, ensure that `cancel` is not set. Another thread might have called this same method with a request to suspend.
3. Before saving to `UserDefaults`, check if the OS told you to stop.

It may look a bit odd to check for cancellation each time, but doing so ensures you honor the OS's directions. In the given example, you simply stop when told. In a production app, you may need to do something else *quickly* to flag that you couldn't complete the desired action.

Transitioning back to the active state

If the user interacts with the app while it's in the background state, watchOS will transition it back to active via this process:

- Restart the app in the `.background` state.
- Call `applicationWillEnterForeground()`.
- Set the `scenePhase` to `.active`.
- Call `applicationDidBecomeActive()`.

If you've been paying attention, you're likely confused by how the user could interact with the app while it's in the background state. The answer, of course, is that they tap the complication that you provided!

Transitioning to the suspended state

When your app finally transitions to the *suspended* state, all code execution stops. Your app is still in memory but is not processing events. The system will transition your app to the suspended state when your app is in the background and doesn't have any pending tasks to complete.

Once your app has moved to a suspended state, it's eligible for purging. If the OS needs more memory, it may, without notice, purge any apps that are in a suspended state from memory. Just because your app has moved to a suspended state doesn't mean it *will* be purged, only that it's *eligible* for purging.

The system will do its best not to purge the most recently executed app, any apps in the Dock and any apps that have a complication on the currently active watch face. If the system must purge one of the aforementioned apps, it'll relaunch the app when memory becomes available.

Always on state

Until watchOS 6, the Apple Watch would appear to go to sleep when the user had not recently interacted with it. **Always On** state changed that so that the watch continued to display the time. However, watchOS would blur the currently running app and show the time over your app's display.

Now, by default, your app's user interface displays instead of the time. watchOS won't blur it as long as it's either the frontmost app or running a background session. When in the Always On state, the watch will dim, and the UI will update slower to preserve battery life.

If the user interacts with your app, the system will return to its active state. One noticeable advantage of Always On relates to dates and times. If your app displays a timer, an offset or a relative date, the UI will continue to update with the correct value.

If you wish to disable Always On for your app, simply set the `WKSupportsAlwaysOnDisplay` key to false in the WatchKit Extension's **Info.plist**.

> **Note**: Users can disable *Always On* for your app, or the entire device, by going to **Settings ▸ Display & Brightness ▸ Always On**.

State change sample

The sample materials for this chapter contain a **Lifecycle** project which you can run against a physical device to observe state changes. When you raise and lower your wrist, you'll see the state changing between active and inactive. If you leave the app inactive for two minutes, you'll notice it switching to background mode.

Extended runtime sessions

It's possible to keep your app running, sometimes even while in the background, for four specific types of use cases.

Self care

Apps focused on the user's emotional well-being or health will run in the foreground, even when the watch screen isn't on. watchOS will give your app a *10 minute* session that will continue until the user switches to another app or you invalidate the session.

Mindfulness

Silent meditation has become a popular practice in recent years. Like self-care, mindfulness apps will stay in the foreground. Meditation is a time-consuming process, though, so watchOS will give your app a *1 hour* session.

If you wish to play audio during the meditation session, you shouldn't use an extended runtime session. Instead, enable background audio and use an `AVAudioSession` as that will keep your app alive. However, it's probably not the best time to play your favorite thrash metal band! :]

Physical therapy

Stretching, strengthening and range-of-motion exercises are perfect for a **physical therapy** session. Unlike the last two session types, physical therapy sessions run in the *background*. A background session will run until the time limit expires or the app invalidates the session, even if the user launches another app.

Physical therapy sessions can run for up to *1 hour*. PTs are expensive, and Apple knows you can't afford more than an hour-long session.

Smart alarm

Smart alarms are a great option when you need to schedule a time to check the user's heart rate and motion. You'll get a *30 minute* session for your watch to run in the *background*.

Unlike the other three session types, you must schedule smart alarms to start in the future. You need to start the session within the next 36 hours and schedule it while your app is in the `WKApplicationState.active` state. Your app will likely suspend or terminate, but the session will continue.

When it's time to handle the session, watchOS will call your `WKExtensionDelegate`'s `handle(_:)`.

> **Note**: You must set the session's `delegate` before the handler exits, or your session will terminate.

Once the session is running, you must trigger an alarm by calling the session's `notifyUser(hapticType:repeatHandler:)`. If you forget, watchOS will display a warning and offer to disable future sessions.

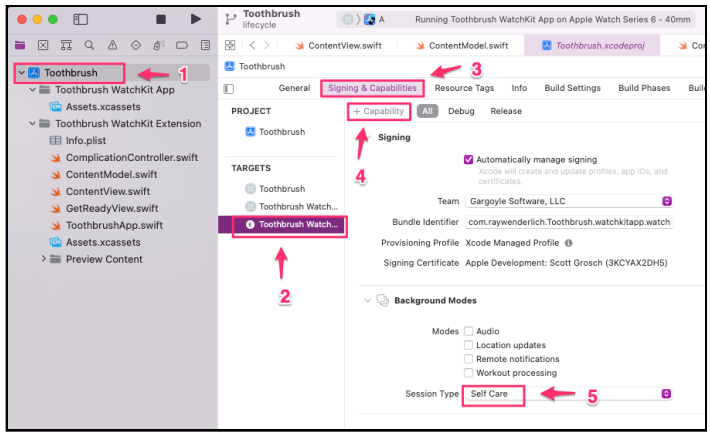
Brush your teeth

If you have children, you know what a chore it can be to get them to brush their teeth. Not only do you have to convince them to *start* brushing, but then they have to brush for the full two minutes recommended by most dentists. Seems like a great job for an Apple Watch app!

Open **Toothbrush.xcodeproj** from this chapter's starter materials.

Assigning a session type

Brushing teeth falls into the self-care session type, but Xcode doesn't know that unless you tell it. Following the steps in the image below, add a new capability to your extension target. First, select the **Toothbrush** app from the Project Navigator menu. Then, select the **Toothbrush WatchKit Extension** Target. Finally choose **Signing & Capabilities** from the main view and press the **+ Capability** option. Select **Background Modes** from the list of capabilities when prompted.



The content model

In the extension target, create a new file named **ContentModel.swift**. Add:

```
import SwiftUI

// 1
final class ContentModel: NSObject, ObservableObject {
  // 2
  @Published var roundsLeft = 0
  @Published var endOfRound: Date?
  @Published var endOfBrushing: Date?

  // 3
  private var timer: Timer!
  private var session: WKExtendedRuntimeSession!
}
```

In the preceding code:

1. You need to conform to `ObservableObject` so the model can update the `ContentView`. You also need to subclass `NSObject` because it's a requirement for conforming to `WKExtendedRuntimeSessionDelegate`, which you'll add in just a bit.
2. The first three properties are `@Published` so `ContentView` responds to updates. You'll use them to track how long you still have to brush.
3. Finally, you need a way to know when time is up, and control the session.

Once the user starts brushing their teeth, you'll need to create the session and update the text displayed on the watch face's button. Add this to `ContentModel`:

```
func startBrushing() {
  session = WKExtendedRuntimeSession()
  session.delegate = self
  session.start()
}
```

You know how picky Xcode is about protocol conformance. So, add the following code to the end of the file to resolve the error that assigning the delegate caused:

```
extension ContentModel: WKExtendedRuntimeSessionDelegate {
  // 1
  func extendedRuntimeSessionDidStart(
    _ extendedRuntimeSession: WKExtendedRuntimeSession
  ) {
  }

  // 2
  func extendedRuntimeSessionWillExpire(
    _ extendedRuntimeSession: WKExtendedRuntimeSession
  ) {
  }

  // 3
  func extendedRuntimeSession(
    _ extendedRuntimeSession: WKExtendedRuntimeSession,
    didInvalidateWith reason:
WKExtendedRuntimeSessionInvalidationReason,
    error: Error?
  ) {
  }
}
```

Conforming to the protocol is pretty simple:

1. The system calls `extendedRuntimeSessionDidStart(_:)` once the session starts running.
2. watchOS will call `extendedRuntimeSessionWillExpire(_:)` just before forcibly expiring your session if your app is about to exceed the session's time limit.
3. Finally, watchOS calls `extendedRuntimeSession(_:didInvalidateWith:error:)` when the session completes for whatever reason.

Notice that the timer wasn't initialized in `startBrushing()`. Until the session is running, you shouldn't start counting down the time. Instead, you'll want to perform startup actions in the delegate method.

Inside `extendedRuntimeSessionDidStart(_:)`, add:

```
let secondsPerRound = 30.0
let now = Date.now

// 1
endOfRound = now.addingTimeInterval(secondsPerRound)
endOfBrushing = now.addingTimeInterval(secondsPerRound * 4)

roundsLeft = 4

// 2
let device = WKInterfaceDevice.current()
device.play(.start)
```

Startup code is mostly straightforward:

1. This is one of the few instances where it's valid to add several seconds instead of performing calendrical operations. You don't care about the actual date or time: you just want a specific number of seconds.
2. When the session starts, it's nice to have the watch perform a quick vibration.

Also, note that both dates are based on the same `now` moment. While unlikely, it's possible that the second of the two date assignments changes, resulting in incorrect calculations.

Now that you know how long each round of brushing will take, it's time to set up a `Timer`. Add the code below to finish out the method:

```
// 1
timer = Timer(
  fire: endOfRound!,
  interval: secondsPerRound,
  repeats: true
) { _ in
  self.roundsLeft -= 1

  // 2
  guard self.roundsLeft == 0 else {
    self.endOfRound =
Date.now.addingTimeInterval(secondsPerRound)
    device.play(.success)
    return
  }

  // 3
  device.play(.success)
  device.play(.success)
}

// 4
RunLoop.main.add(timer, forMode: .common)
```

Here's what the code does:

1. You generate a timer that starts at the end of the current round and repeats every `secondsPerRound` seconds.
2. If there are still rounds of brushing left to perform, you update the time that the round ends so the view's display updates. Having the watch vibrate lets the user know it's time to switch to a new section of their mouth.
3. If the final round is complete, you perform two vibrations to let your kid know they can finally stop their onerous chore.
4. Finally, you schedule the timer into the main run loop.

Did you spot the issue with the preceding code? While you signaled the user via the vibrations that they finished brushing, you didn't let watchOS know you were done. In 30 seconds, they'll get tapped again.

When the last round is complete, you need to perform cleanup:

- Disable the timer.
- Stop the extended runtime session.
- Update the UI.

The `extendedRuntimeSession(_:didInvalidateWith:error:)` seems like the obvious location for cleanup. Right before step three in the code above, cancel the session:

```
extendedRuntimeSession.invalidate()
```

When you invalidate the session, watchOS will call that delegate method. Add the following code to `extendedRuntimeSession(_:didInvalidateWith:error:)`:

```
timer.invalidate()
timer = nil

endOfRound = nil
endOfBrushing = nil
roundsLeft = 0
```

Build your project to make sure you've entered everything properly so far. Now it's time to handle the UI.

The content view

Edit **ContentView.swift**, and you'll see that it's already configured to print the `ScenePhase` for you during phase updates. The first task required is, of course, to use the model you just created. So, add the following line to the view:

```
@ObservedObject private var model = ContentModel()
```

Using an `@ObservedObject`, SwiftUI will update the `body` any time one of the published variables from your model updates. Next, replace the default text in the `body` with:

```
// 1
VStack {
  // 2
  Button {
    model.startBrushing()
  } label: {
    Text("Start brushing")
  }
```

```
    .disabled(model.roundsLeft != 0)
    .padding()

    // 3
    if let endOfBrushing = model.endOfBrushing,
       let endOfRound = model.endOfRound {
      Text("Rounds Left: \(model.roundsLeft - 1)")
      Text("Total time left: \(endOfBrushing, style: .timer)")
      Text("This round time left: \(endOfRound, style: .timer)")
    }
  }
```

In the preceding code:

1. Using a `VStack` lets you place multiple views in a vertical layout.
2. When the user taps **Start brushing**, you'll begin the process by calling into your model. Be sure to disable the button if they're already brushing!
3. If there are dates set in the model, that means a session is active, and you'll therefore want to let the user know how much time is left. Take note of the `style: .timer` modifier in the string interpolation, which may be new syntax to you. Instead of displaying a date, SwiftUI will automatically update a countdown timer.

Build and run the app on a *physical device*. While you could run in the simulator, the scene phase will never switch to inactive or background if you do.

Once you tap the button, you'll see the countdown begin:

Active the console by pressing **Shift**-**Command**-**C**. Then clear out all unhelpful debugging information Xcode emits by default by pressing **Control**-**K**.

Lift and lower your arm a few times, and you'll see messages in the console that the app has switched to the active or inactive states. However, the app never moves to the background, even though you'll eventually pass the two-minute mark. The session has kept the app alive.

Ready, set, go

While functional, you can do better! Have you ever used the **Workout** app on your watch? When you start a workout, you get a few seconds to get ready before it begins. That seems useful for your app as well. The starter sample contains **GetReadyView.swift** that simulates that display.

In **ContentModel.swift**, add another property:

```
@Published var showGettingReady = false
```

By default, the getting ready timer shouldn't display. You'll also want to set it to false once brushing has started. So, add this as the first line of `startBrushing()`:

```
showGettingReady = false
```

Then, back in **ContentView.swift**, replace the call to `model.startBrushing()` with:

```
model.showGettingReady = true
```

To make something happen when you tell the timer to display, add an overlay modifier to the `VStask` immediately before the `.onChange(of:)` call:

```
// 1
.overlay(
  // 2
  VStack {
    if model.showGettingReady {
      // 3
      GetReadyView {
        model.startBrushing()
      }
      .frame(width: 125, height: 125)
      .padding()
    } else {
      // 4
      EmptyView()
    }
  }
)
```

Cool stuff happening:

1. An `overlay` layers a secondary view in front of another view.
2. By using a `VStack`, even though you're only going to show a single view, you gain the ability to display a timer conditionally.
3. If you asked to show it, then you provide the `GetReadyView`. The closure is called when the countdown is complete, at which point it's time to start brushing.
4. By using the `EmptyView`, you don't display anything, but SwiftUI is still happy about the conditional check.

Build and rerun the app. This time, you'll get a couple of seconds to get ready:

Key points

- An inactive phase on watchOS doesn't mean the app isn't running.
- Prefer SwiftUI's `.scenePhase` environment variable over extension delegate methods.
- For specific types of apps, extended runtimes let your app keep running even when in the background.

Where to go from here?

Check out Apple's documentation for WatchKit Life Cycles (https://apple.co/2WTQgyR) and Extended Runtime Sessions (https://apple.co/2WPvCA3).

Chapter 8: Introduction to Complications

By Scott Grosch

When you design an iOS app, it's normal to expect users will engage with your app for some time. On the other hand, when designing a watchOS app, you'll likely find users engage with the app for mere seconds. Therefore, you should always strive to add **complications** to your app.

Wait, what? Not *that* kind of complication — a *watch* complication. According to Wikipedia:

> A **complication** is any feature of a mechanical timepiece beyond the display of hours, minutes and seconds.

By that definition, the Apple Watch is *full* of complications! :]

On the Apple Watch, complications have been slightly redefined as elements on the watch face that display small, immediately relevant bits of information. They are by far one of the most compelling and useful features of the Apple Watch. They lie *right* on the watch face, making accessing information as fast as raising your wrist.

This chapter will speak to a high-level overview of what complications are and why you should use them. The following chapters will dive into the technical details of actually implementing them.

Unless there's a compelling reason not to include one, every watchOS app should include *at least* one complication.

Why complications?

As previously mentioned, interacting with a watchOS app is generally on the order of a second or two, if even that long. For example, when using a mapping app, you probably just want to know how far away your next turn is and which direction you'll be heading.

Even if you don't feel that your app contains any data which would make sense to display on the watch face, providing a complication gives your user a way to launch your app with a simple touch of the screen. On top of that, watchOS provides several benefits when your user includes your complication on their watch face:

- watchOS keeps your app in memory and gives it extra update time, making app launches almost instantaneous.
- You complication can receive up to 50 pushes per day containing updated information.
- Your app can perform additional background refresh tasks.
- Your complication can be featured in the Apple Watch Face Gallery on the phone.

Imagine the types of questions you can instantly answer by having a complication available on the watch face:

- What's the score?
- How hot is it outside?
- Is the air safe to breathe?
- What is your name?
- What is your quest?
- What is the average airspeed velocity of an unladen swallow?

> **Note**: If you're confused by the last three questions, they're from a famous scene in the 1975 movie *Monty Python and the Holy Grail*. The internet seems to think a swallow would average around 20 miles per hour or about 9 meters per second.

Recent versions of iOS provide a similar concept via **Widgets**. However, unlike the Apple Watch, to see the data on a widget, you'll need to pull the phone out of your pocket or purse, unlock the screen, possibly swipe to another screen and then see your data. Complications are quicker.

The following chapters will dive into the technical details of using the **ClockKit** framework to implement complications and show how to keep them up-to-date with current information.

In other words, by providing complications, you'll *uncomplicate* your app. :]

Complication families

If you've spent any time looking at the Apple Watch's available watch faces, you've seen that there are quite a few. Each new release of watchOS tends to include more faces than the previous version.

Beyond the layout of the watch face, each separate face provides different types of complications. Apple groups the types of complications displayed into families, represented by the `CLKComplicationFamily` enumeration type.

There are currently 12 different families which you should implement. Each family contains a distinct set of templates that you can apply to complications of that specific family. Support *all* of the complication families to provide the best user experience. That snazzy Utilitarian watch face you're using might be your favorite, but your customer might prefer a completely different watch face.

Complication identifiers

Most watch faces include the ability to specify multiple complications. While most apps only provide a single complication, ideally supporting all families, you can support more than one. If you've created a weather app, you might provide one complication which shows the temperature and a separate complication that displays the current air quality index.

All complications you support are created via the `CLKComplicationDescriptor(identifier:displayName:supportedFamilies:)` initializer. Including a unique identifier lets you provide multiple versions of the same complication family.

Complication templates

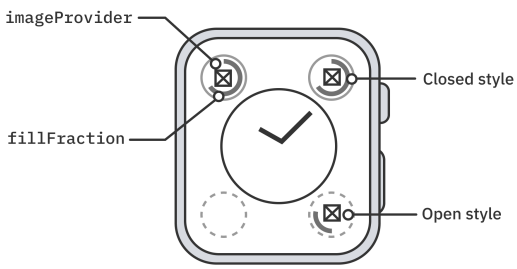
Within each family, Apple provides multiple templates which describe the layout of the complication. Most of the templates consist of simple text and images, though some provide partial SwiftUI `View` support.

Circular small

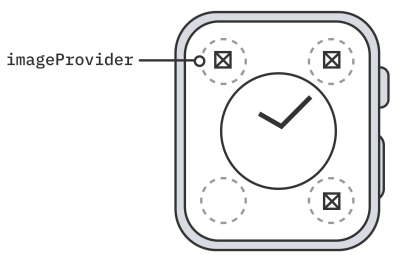
The Circular Small template provides a small circle in the corners of the **Color** watch face that can display a few characters of text, an image or a circular progress ring. It includes the following templates:

Image templates

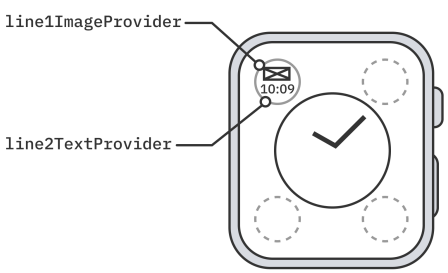
- `CLKComplicationTemplateCircularSmallRingImage`: A single image with a progress ring.



- `CLKComplicationTemplateCircularSmallSimpleImage`: A single image with no text.

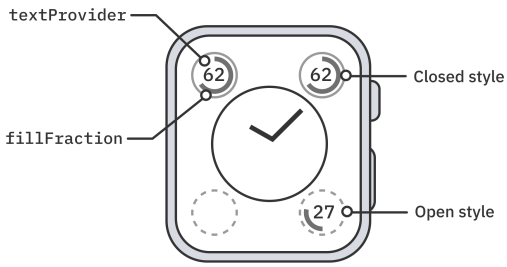


- `CLKComplicationTemplateCircularSmallStackImage`: An image with a line of text below it.

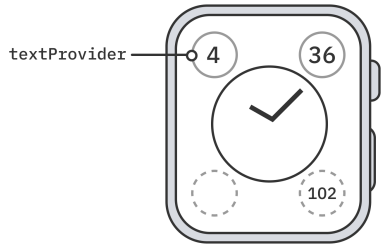


Text templates

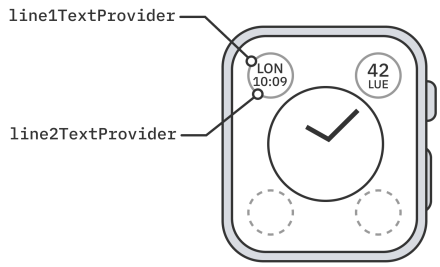
- `CLKComplicationTemplateCircularSmallRingText`: A short text string inside of a progress ring.



- `CLKComplicationTemplateCircularSmallSimpleText`: A short text string with no image.



- `CLKComplicationTemplateCircularSmallStackText` : Two short text strings, one atop the other.

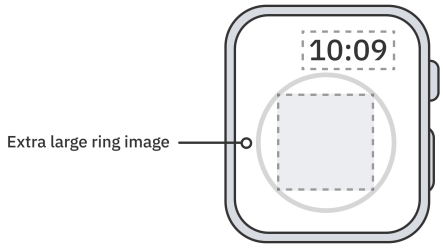


Extra-large

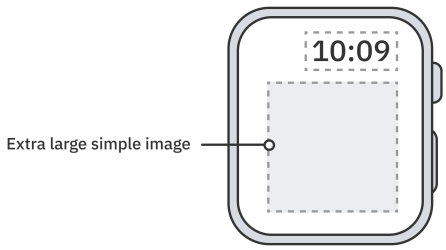
The extra-large templates are similar to the Circular Small templates but larger and with a 2x2 grid template. They're designed for the **X-Large** watch face.

Image templates

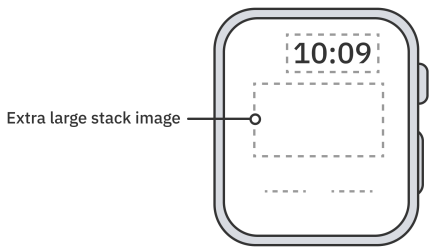
- `CLKComplicationTemplateExtraLargeRingImage`: A single image with a progress ring.



- `CLKComplicationTemplateExtraLargeSimpleImage`: A single image with no text.

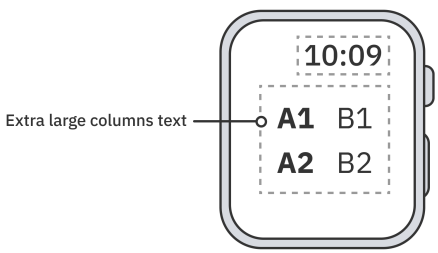


- `CLKComplicationTemplateExtraLargeStackImage`: An image with a line of text below it.

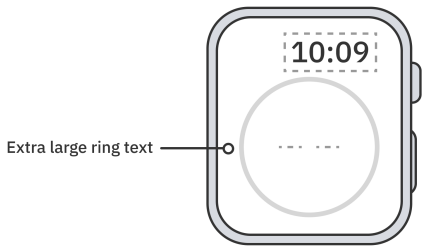


Text templates

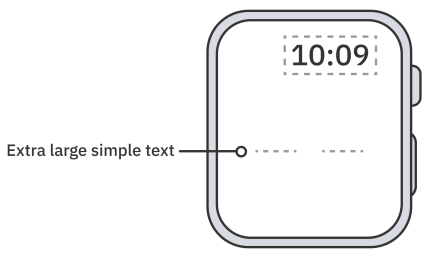
- `CLKComplicationTemplateExtraLargeColumnsText`: A 2x2 grid of text.



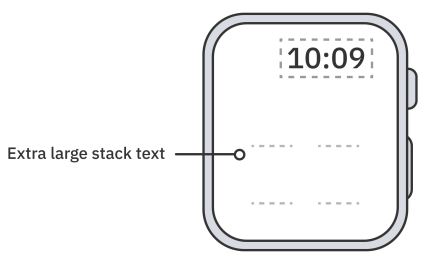
- `CLKComplicationTemplateExtraLargeRingText`: A short text string inside of a progress ring.



- `CLKComplicationTemplateExtraLargeSimpleText`: A short text string with no image.



- `CLKComplicationTemplateExtraLargeStackText`: Two short text strings, one atop the other.

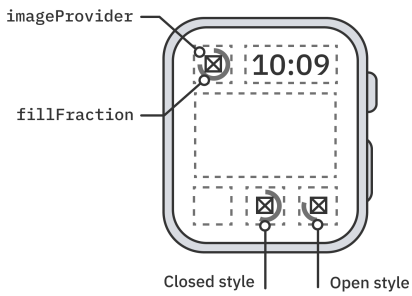


Modular small

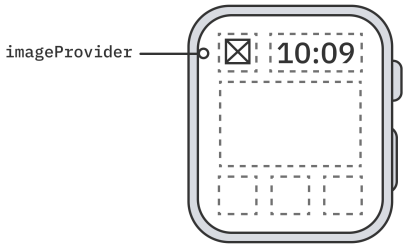
When using the **Modular** watch face, you can provide content to the smaller spaces via these templates.

Image templates

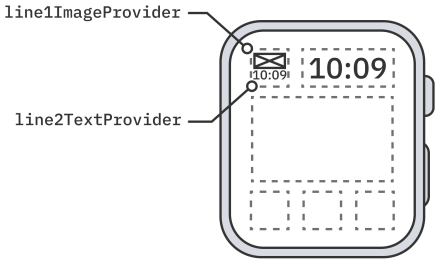
- `CLKComplicationTemplateModularSmallRingImage`: A single image with a progress ring.



- `CLKComplicationTemplateModularSmallSimpleImage`: A single image.

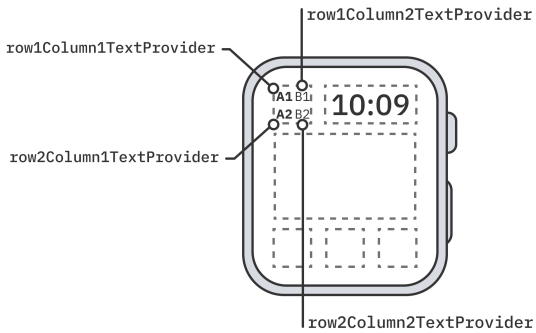


- `CLKComplicationTemplateModularSmallStackImage`: An image with a line of text below it.

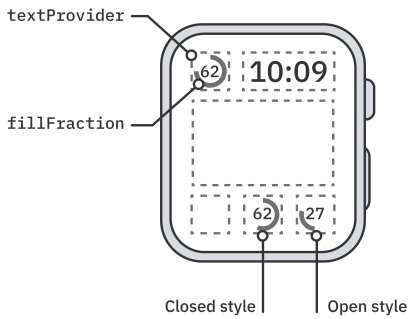


Text templates

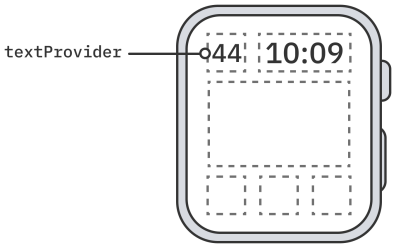
- `CLKComplicationTemplateModularSmallColumnsText`: A 2x2 grid of text.



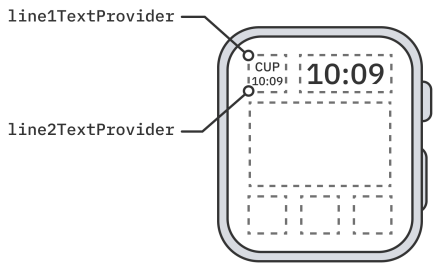
- `CLKComplicationTemplateModularSmallRingText` : A short text string inside of a progress ring.



- `CLKComplicationTemplateModularSmallSimpleText`: A short text string with no image.



- `CLKComplicationTemplateModularSmallStackText`: Two short text strings, one atop the other.

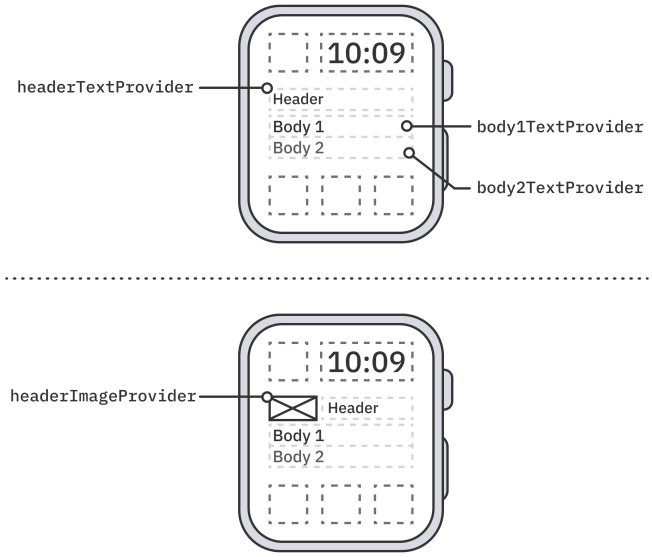


Modular large

The modular large templates target the large canvas on the **Modular** watch face, providing up to three lines of content.

Body templates

- `CLKComplicationTemplateModularLargeStandardBody`: Displays a header row and two lines of text.



- `CLKComplicationTemplateModularLargeTallBody`: Displays a header row and a single tall line of text.

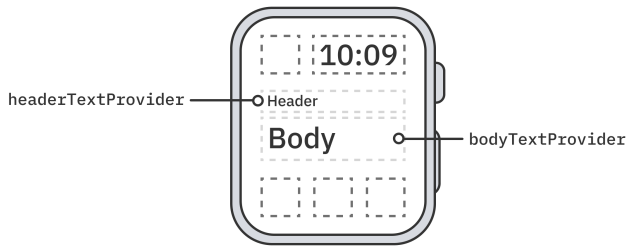
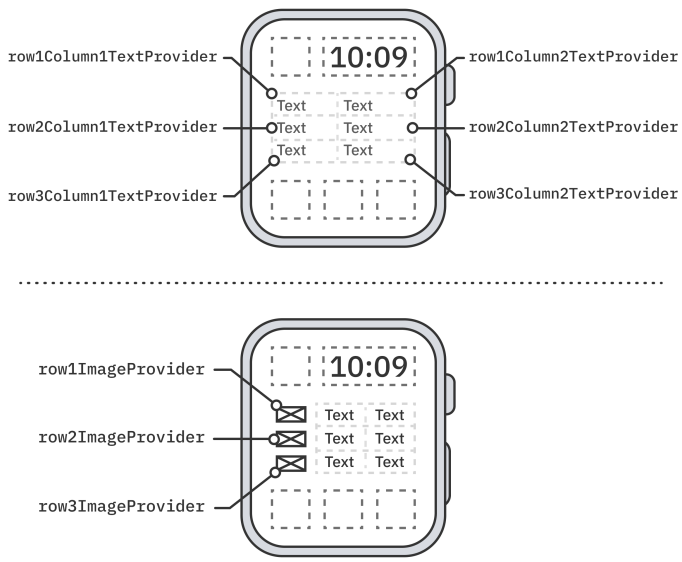
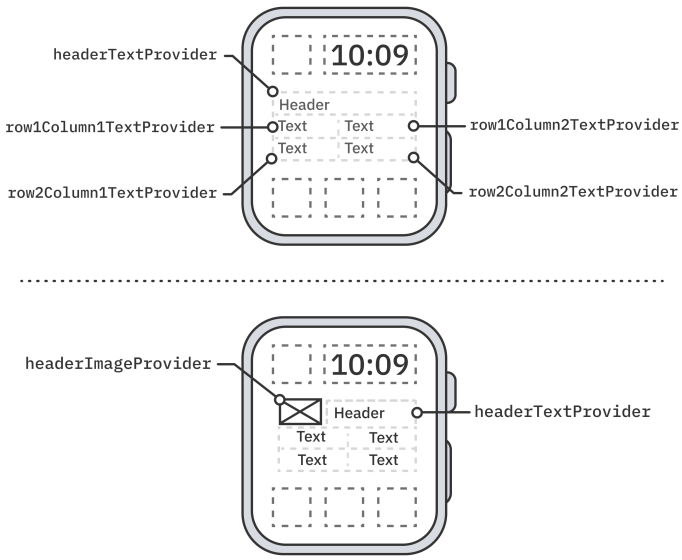


Table templates

- `CLKComplicationTemplateModularLargeColumns`: Displays either a 3x2 grid of text or a 3x3 grid where the first column is a small image.



- `CLKComplicationTemplateModularLargeTable`: Displays a header row with an optional small image, and then a 2x2 grid of text.

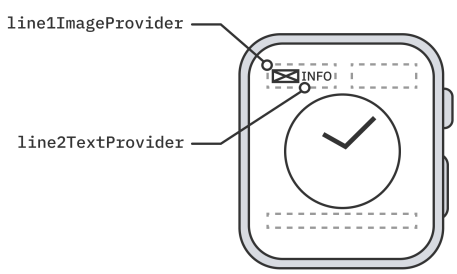


Utilitarian

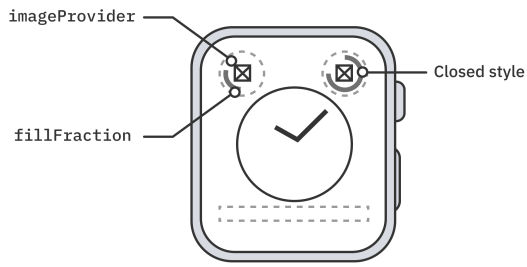
The utilitarian templates provide multiple ways to display content on numerous watch faces, including **Utility**, **Chronograph**, **Simple** and the character watch faces.

Utilitarian small

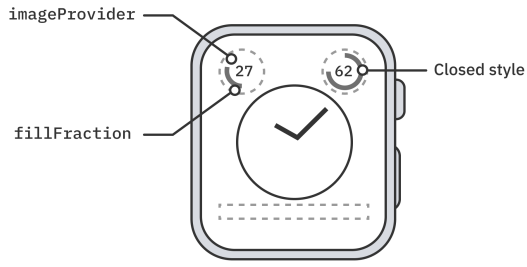
- `CLKComplicationTemplateUtilitarianSmallFlat`: Displays an image followed by short text on a single line.



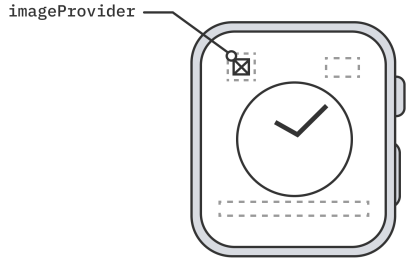
- `CLKComplicationTemplateUtilitarianSmallRingImage`: Displays a small image inside of a circular progress ring.



- `CLKComplicationTemplateUtilitarianSmallRingText`: Displays a short text string inside of a circular progress ring.

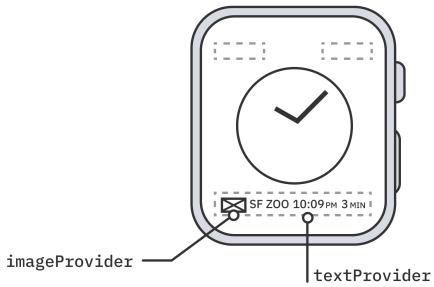


- `CLKComplicationTemplateUtilitarianSmallSquare`: Displays a small square image.



Utilitarian large

- `CLKComplicationTemplateUtilitarianLargeFlat`: Displays an image followed by a long text string on a single line.



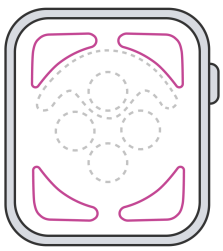
Graphic

The graphic templates display visually rich content on several different watch faces, including the **Infograph**, **Infograph Modular** and **Solari Dial** faces.

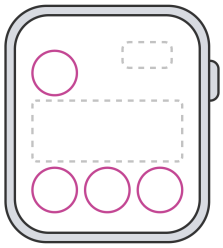
Unlike the other templates, the graphic templates also let you use SwiftUI views in place of rigid layouts. There are almost 30 different templates in the graphic family, which is too many to enumerate here. The following chapters will speak directly to using SwiftUI in your complications.

The graphic templates fall into five families:

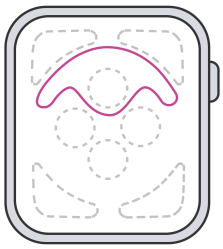
- `CLKComplicationFamily.graphicCorner`: The curved areas that fill the corners of the Infograph watch face.



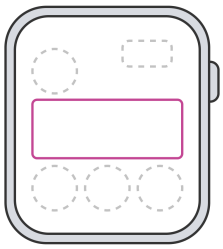
- `CLKComplicationFamily.graphicCircular`: The circular areas on the Infograph and Infograph Modular watch faces.



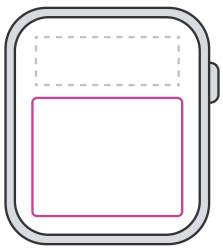
- `CLKComplicationFamily.graphicBezel`: A circular area with optional curved text along the bezel of the Infograph watch face.



- `CLKComplicationFamily.graphicRectangular`: The large rectangular area in the center of the Infograph Modular watch face.



- `CLKComplicationFamily.graphicExtraLarge`: A large square area on the X-Large watch face.



Tinted complications

While you can include full-color images in many complication templates, you need to be careful not to make your complication *depend* on those colors being visible.

One reason, of course, is that many people are color blind. The Colour Blind Awareness (https://www.colourblindawareness.org/colour-blindness/) organization estimates that approximately one in twelve men and one in 200 women globally are color blind.

From a purely technical standpoint, your users can disable the colors! Watch faces that support graphic complications can be tinted, displaying a low-color version of the complication. If the watch face is tinted, the following changes apply to the template:

- Gauges which previously used a color gradient now display a solid color based on the selected tint.
- Text color is based on the user's watch face color, while multicolor text providers display a single color.
- Images are desaturated by default though you can provide a custom tinted version for the image.

Key points

- *Always* provide a complication, even if just to provide a quick way to launch your app.
- Support *every* complication family, not just one or two.
- Consider the different pieces of data someone might want to regularly see on the watch face and create a complication for each one.

Where to go from here?

- Apple's documentation on **ClockKit (**https://apple.co/3xRz4X8**)** shows images of exactly where each type of complication displays on the watch face, as well as examples of each template type.

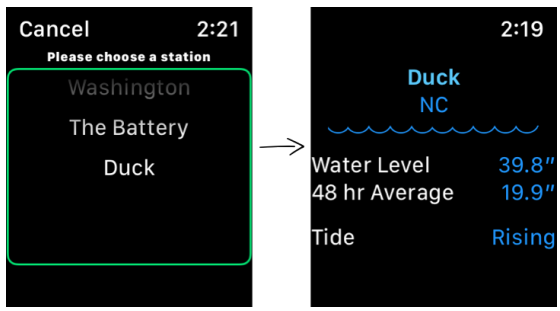
Chapter 9: Complications

By Scott Grosch

Exploring the sample

Please build and run the **TideWatch** app from this chapter's starter materials. After a moment, you'll see the current tide conditions at the Point Reyes tide station in California.

Tap the station name to pick a new location:



Even though the app is amazingly useful, as designed, your customers have to open the app to find out what the current water level is. Wouldn't it be great if they could see the information right on their watch face?

Complication data source

When you create a watchOS project, Xcode will generate **ComplicationController.swift**. For the sample project, I've moved that file into the **Complications** folder. Also, I removed everything except the one method required by `CLKComplicationDataSource`. Most of the boilerplate code is unnecessary.

The current timeline entry

When watchOS wants to update the data displayed for your complication, it calls `currentTimelineEntry(for:)`. You're expected to return either the data to display right now or `nil` if you can't provide any data.

If you can't provide a data point for the current time, then watchOS will look in your extension's **Assets.xcassets** bundle. You've likely noticed that there's a **Complication** folder inside the asset bundle, which you haven't used before. When `currentTimelineEntry(for:)` returns `nil`, watchOS will use the appropriately named image from the asset bundle if it exists.

The simulator's default watch face is **Meridian**, which uses the `.graphicCircular` complication family for most configurable complications. For your first foray into supporting complications in your app, replace the method body with:

```
// 1
guard complication.family == .circularSmall else {
  return nil
```

```
  }

  // 2
  let template = CLKComplicationTemplateGraphicCircularStackText(
    line1TextProvider: .init(format: "Surf's"),
    line2TextProvider: .init(format: "Up!")
  )

  // 3
  return .init(date: Date(), complicationTemplate: template)
```

That's quite a bit of code to tell somebody to surf! Here's what's happening:

1. If the Apple Watch is showing a complication family type you don't support, then you return `nil`.
2. Then, you create a complication template of the appropriate type and configure the text to display.
3. You return a `CLKComplicationTimelineEntry` that specifies the time of the data point and the template to display. The date specified should never be in the future, but it may be in the past.

In step two, notice the template takes two text providers. This is because each template uses different textual or graphical elements. Consult the documentation for the various templates to determine which is appropriate for your needs.

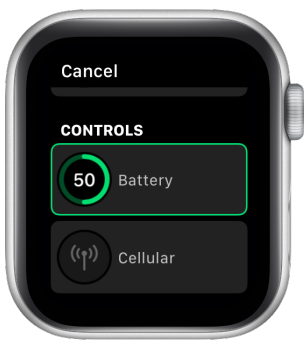
Switch the active scheme to **TideWatch –> WatchKit –> App –> (Complication)** and then **build and run** again. Using the complication scheme ensures that your supported families are used without caching. The scheme will also launch the simulator directly to the watch face and give your app a small amount of background processing time.

Tap and hold on the watch face, and the editor will appear:

> **Note**: The simulator sometimes has issues bringing up the editor. If the edit button doesn't appear, quit and restart the simulator, or use a physical device instead.

Tap **Edit**, then swipe left two times so you can pick the complication you want to replace:

Tap whichever circular complication you wish to replace, *other than the top image of the Earth*, to see the list of complications you may choose from:



Scroll until you see your app listed so you can choose the shiny new complication you just created. You don't see your app listed? Oh no! What went wrong?

`CLKComplicationDataSource` has an *optional* method named `complicationDescriptors()`. Unfortunately, it's not optional. If you don't provide the method, you won't see your complication listed.

> **Note**: The previous version of watchOS looked in **Info.plist** for the supported complications. That's why the method is optional. Don't use the **Info.plist** anymore, per Apple's recommendation.

Add the following method:

```
func complicationDescriptors() async ->
[CLKComplicationDescriptor] {
  return [
    // 1
    .init(
      // 2
      identifier: "com.raywenderlich.TideWatch",
      // 3
      displayName: "Tide Conditions",
      // 4
      supportedFamilies: [.graphicCircular]
    )
  ]
}
```

There's quite a bit happening in the code:

1. You provide an array of `CLKComplicationDescriptor` items as the return value for this method. Each descriptor appears in the list of complications to choose from.
2. Each complication you support should have a unique name. Ensure that the names are deterministic and don't change between app launches.
3. The `displayName` is what the user sees when choosing a complication from the list that your app supports.
4. Complications provide an array of the families they support.

Build and run again. This time, when you scroll, you'll see your complication listed as an option to pick.

> **Note**: The app name displayed in the list is based on the Display Name set on your **TideWatch WatchKit App** target.

You're making progress, but what's up with the -- in the circle? Why isn't it showing the message you specified in the timeline?

Sample data

The current timeline entry is neither displayed in this list nor the Watch app on your iPhone. When asking for the current timeline data, your app may have to perform an expensive operation or run something asynchronously.

Instead, you use a sample set of data to make the display happen immediately and avoid potential side effects in your app. The `CLKComplicationDataSource` provides another *optional* — yah, not really optional IMHO — method called `localizableSampleTemplate(for:)` that watchOS calls when the Apple Watch needs to display the complication selector in the list.

Implement the method as shown below:

```
func localizableSampleTemplate(
  for complication: CLKComplication
) async -> CLKComplicationTemplate? {
  // 1
  guard
    complication.family == .graphicCircular,
    let image = UIImage(named: "tide_rising")
  else {
    return nil
```

```
    }

    // 2
    let tide = Tide(entity: Tide.entity(), insertInto: nil)
    tide.date = Date()
    tide.height = 24
    tide.type = .high

    // 3
    return CLKComplicationTemplateGraphicCircularStackImage(
      line1ImageProvider: .init(fullColorImage: image),
      line2TextProvider: .init(format: tide.heightString())
    )
  }
```

Here's a breakdown of the method's three key elements:

1. It ensures the family is one that you support. Also, it makes sure you can load the default image to display in the complication preview.
2. Then, it generates sample tide data. The app includes the `Tide` Core Data model you're using. By inserting into `nil`, you prevent actual Core Data updates from occurring.
3. For the template, you display the image you loaded in step one on top of the tide height. **Core Data/Tide+Extension.swift** provides a helper method, `heightString(unitStyle:)`, to properly format the height in the user's locale.

Build and run again. This time, when you try to select the complication, you'll see a much better display:

The app uses meters to store all heights. Using a `MeasurementFormatter` and `Measurement<UnitLength>`, you ensure a localized display is available on the watch face. You want to surf in a 78.7 -foot high wave, right?

Tap the row to select the complication and then go back to the Apple Watch's home screen. You'll see your complication displays, but still with no data:

Updating the complication's data

When people first learn about complications, the missing "Ah-ha!" moment is that the Apple Watch will only attempt to update the complication on the watch face when you specify that new data is available. Imagine the battery drain if watchOS had to query your complication every second to see if a new data point was available?

Telling watchOS there's new data

Open **CoOpsApi.swift**, and you'll see `getLowWaterHeights(for:)`, the method the app calls when it needs to download new tide data. Using the **Combine** framework allows for a very clean data download pipeline.

At the top of the file, add an import for **ClockKit**:

```
import ClockKit
```

Back down in `getLowWaterHeights(for:)`, scroll to `add(predictions:to:in:)`, which the app calls when new data has been successfully downloaded and decoded from the network. Right after `add(predictions:to:in:)`, add:

```
DispatchQueue.main.async {
  let server = CLKComplicationServer.sharedInstance()
  server.activeComplications?.forEach {
    server.reloadTimeline(for: $0)
  }
}
```

Once your Core Data model updates, you tell watchOS that it needs to reload the whole timeline for any complication currently on the watch face. Depending on your app and its data model, reloading the entire timeline might not be the most efficient option.

If the existing data in your complication's timeline is still valid, and you're simply adding new data, you should instead call `extendTimeline(for:)`.

> **Note**: If you've already exceeded your app's budgeted execution time, then calls to either method won't perform any action.

Providing data to the complication

Switch back to **Complications/ComplicationController.swift** and replace the body of `currentTimelineEntry(for:)` with:

```
// 1
guard
  complication.family == .graphicCircular,
  let tide = Tide.getCurrent()
else {
  return nil
}

// 2
let template = CLKComplicationTemplateGraphicCircularStackImage(
  line1ImageProvider: .init(fullColorImage: tide.image()),
  line2TextProvider: .init(format: tide.heightString())
)

// 3
return .init(date: tide.date, complicationTemplate: template)
```

Here's what's happening with the new method:

1. If watchOS asks for a family you don't support, or there's no current data to display, then the method returns `nil`.
2. In the second step, you create the appropriate graphic and text template, just like you did for the sample data.
3. For the timeline, specify the date of the data you're providing, as well as the template.

The sample project provides helper methods on the `Tide` object since the focus of this chapter isn't how to handle Core Data but rather how to update your complication properly. Feel free to investigate **Core Data/Tide+Extension.swift** at your leisure.

Build and run again. Give it a moment to download data from the network and then switch back to the watch face. You'll see real data displayed now:

Supporting multiple families

While you now have a fully functional app with complication support, it's pretty limited. For your customers to use your complication, they must use one of the watch faces that supports `.graphicCircular`. Whenever you're designing complications for the Apple Watch, you should strive to support every type of family you can.

Recall that you specified a single family to the `supportedFamilies` parameter when you generated `CLKComplicationDescriptor` in `complicationDescriptors()`. While it's just a few keystrokes for you to add the rest of the types, or even simply specify `CLKComplicationFamily.allCases`, you still have to handle each distinct template type.

Most of the resources you'll see online tell you to simply create a `switch` statement, against the family, in each method to determine what actions to take. While you could do that, please don't. The controller will become massively bloated and incredibly hard to maintain.

Factory Method design pattern

There's a common design pattern, called **Factory Method**, which you can implement to great effect. Create a new file in **Complications** called **ComplicationTemplateFactory.swift**. Consider the code you've written so far, and you can likely see some common patterns that you'll need to replicate across each family.

You needed to take the following steps, just for a single supported family:

- Generate the current timeline entry.
- Generate the sample template.
- Generate the text which displays the tide height.
- Generate the image which represents the tide type.

A `protocol` is a great way to represent those actions.

Current timeline entry

Start by adding the following code to your new file:

```
import ClockKit

protocol ComplicationTemplateFactory {
  func template(for waterLevel: Tide) -> CLKComplicationTemplate
}
```

Regardless of type, any family you support needs a way to convert from your `Tide` data model to a `CLKComplicationTemplate`. The code for each type of family will be distinct, so you can't provide a default implementation.

Samples

Samples, however, can use a default implementation. Add the following line to your protocol:

```
func templateForSample() -> CLKComplicationTemplate
```

Consider what `localizableSampleTemplate(for:)` currently does. It generates a fake `Tide` entry, creates the template for the correct family and then returns that template. You've just specified that any class, which conforms to `ComplicationTemplateFactory`, knows how to generate a template based on real data. Why not pass that method some fake data instead?

Add the follow to the end of the file, after the `protocol` definition:

```
extension ComplicationTemplateFactory {
  func templateForSample() -> CLKComplicationTemplate {
    let tide = Tide(entity: Tide.entity(), insertInto: nil)
    tide.date = Date()
    tide.height = 24
    tide.type = .falling

    return template(for: tide)
  }
}
```

Recall that protocols allow for default implementations of their methods by providing the method in an `extension` for the protocol. By implementing `templateForSample()` as a default implementation, you've ensured that every single complication family you support will already know how to generate a localizable sample.

Tide height text

Text providers support both a *short* and *long* version of the text. Right now, your code simply shows the height, but you can do better than that.

Add another method to the `protocol`:

```
func textProvider(for waterLevel: Tide, unitStyle:
Formatter.UnitStyle) -> CLKSimpleTextProvider
```

Then provide a default implementation in the extension:

```
// 1
func textProvider(
  for waterLevel: Tide,
  unitStyle: Formatter.UnitStyle = .short
) -> CLKSimpleTextProvider {
  // 2
  let shortText = waterLevel.heightString(unitStyle: unitStyle)

  // 3
  let longText = "\(waterLevel.type.rawValue.capitalized), \
```

```
  (shortText)"

    // 4
    return .init(text: longText, shortText: shortText)
  }
```

Here's a code breakdown:

1. The text provider will generate the appropriate verbiage based on a provided `Tide`. Depending on the type of family, you might want to display the height in different lengths. You'll default to `.short`.
2. The short text will simply be the tide's height.
3. The long text will be the tide's type, followed by its height.
4. You pass both types of text to `CLKSimpleTextProvider`.

When you use the two-parameter version of `CLKSimpleTextProvider`, the Apple Watch will choose which text to display based on the configuration of the family. If the longer text fits, that will display. If the complication in use is too narrow, then the shorter version will show.

Why the `unitStyle` parameter? While you don't know exactly how much text will fit in any given complication, you do know the type you're working with. If, for example, you're displaying against `.graphicCircle`, you wouldn't want a height string that spells out the distance. However, you might want to use a longer formant when using the `.extraLarge` family.

Tide image

Images are as easy to support as text. Add two more protocol methods:

```
  func fullColorImageProvider(for waterLevel: Tide) ->
  CLKFullColorImageProvider
  func plainImageProvider(for waterLevel: Tide) ->
  CLKImageProvider
```

While the sample app doesn't differentiate between full color and plain images, the factory you're designing here will be reusable across all your apps.

The default implementations are quite simple. Add these methods to the extension:

```
  func fullColorImageProvider(for waterLevel: Tide) ->
  CLKFullColorImageProvider {
    .init(fullColorImage: waterLevel.image())
  }
```

```
func plainImageProvider(for waterLevel: Tide) ->
CLKImageProvider {
  .init(onePieceImage: waterLevel.image())
}
```

You would likely provide two separate image generation methods on the `Tide` object in your production app.

Templates by family

Please create a **Templates** folder group inside **Complications**. Inside **Templates**, you'll create a file per family that you support. Start by creating **GraphicCircular.swift** and filling it with:

```
import ClockKit

struct GraphicCircular: ComplicationTemplateFactory {
  func template(for waterLevel: Tide) -> CLKComplicationTemplate
{
    return CLKComplicationTemplateGraphicCircularStackImage(
      line1ImageProvider: fullColorImageProvider(for:
waterLevel),
      line2TextProvider: textProvider(for: waterLevel)
    )
  }
}
```

Since `GraphicCircular` conforms to the `ComplicationTemplateFactory` which you just implemented, you already have a ton of functionality. The only piece you need to handle is the creation of the actual template.

All `template(for:)` has to do is return the appropriate `CLKComplicationTemplate`. In the code above, you can see that you simply copied the `CLKComplicationTemplateGraphicCircularStackImage` you were already using.

Next, you'll need to implement a method to determine which template family `struct` to use. Create **ComplicationTemplates.swift** in **Complications** with:

```
import ClockKit

// 1
enum ComplicationTemplates {
  // 2
  static func generate(
    for complication: CLKComplication
  ) -> ComplicationTemplateFactory? {
    // 3
```

```
      switch complication.family {
      case .graphicCircular: return GraphicCircular()

      // 4
      default:
        return nil
      }
    }
  }
```

Here's what the code does:

1. When an object's implementation only contains `static` methods, you use an `enum` to prevent accidental instantiation.
2. Given a `CLKComplication`, you return the appropriate factory method.
3. By looking at the `family`, you return the appropriate `struct` which implements `ComplicationTemplateFactory`.
4. If the given complication family isn't supported, you return `nil`.

Updating the complication controller

Now that you've implemented the factory pattern, it's time to put it to use. Edit **ComplicationController.swift** again to take advantage of your hard work.

First, replace the body of `currentTimelineEntry(for:)` with:

```
  guard
    // 1
    let factory = ComplicationTemplates.generate(for:
  complication),
    // 2
    let tide = Tide.getCurrent()
  else {
    return nil
  }

  // 3
  let template = factory.template(for: tide)
  return .init(date: tide.date, complicationTemplate: template)
```

Here's a step-by-step explanation:

1. By calling the factory generation method, you determine whether the provided complication is supported. No more looking at family types in the complication controller.
2. If there's not a current data point to display, then there's nothing else to do.
3. Using the `factory` generated in step one, you create the appropriate template for the given `tide`.

The `localizableSampleTemplate(for:)` becomes incredibly compact now. Replace the method's body with this statement:

```
return ComplicationTemplates.generate(for:
complication)?.templateForSample()
```

Because you had `generate(for:)` return `nil` when a family isn't supported, you can use a null chain operation. If the family isn't supported, `template` will set to `nil`. If it is, then you'll assign the actual template sample.

But...why?

If it's not clear why you added the extra level of indirection, imagine your manager tells you that now you must support the `.graphicBezel` complication family. How much effort will that take? Not much!

There are only three quick steps required. First, add `.graphicBezel` to `supportedFamilies` in `complicationDescriptors()` of **ComplicationController.swift**:

```
supportedFamilies: [.graphicCircular, .graphicBezel]
```

Next, add a new entry to `switch` in **ComplicationTemplates.swift**:

```
case .graphicBezel: return GraphicBezel()
```

Ignore the compiler error telling you that `GraphicBezel()` doesn't exist. Finally, create **GraphicBezel.swift** in **Templates**:

```
import ClockKit

struct GraphicBezel: ComplicationTemplateFactory {
  func template(for waterLevel: Tide) -> CLKComplicationTemplate
{
```

```
    let circularTemplate =
CLKComplicationTemplateGraphicCircularImage(
      imageProvider: fullColorImageProvider(for: waterLevel)
    )

    return CLKComplicationTemplateGraphicBezelCircularText(
      circularTemplate: circularTemplate,
      textProvider: textProvider(for: waterLevel,
unitStyle: .long)
    )
  }
}
```

When generating a complication, the `CLKComplicationTemplate` subclass you wish to use will drive how `template(for:)` is implemented.

`CLKComplicationTemplateGraphicBezelCircularText` wants both a `CLKComplicationTemplateGraphicCircularImage` as well as a `CLKTextProvider`. While you've already coded the method to generate the text provider, you need a circular image. Looking at the documentation for `CLKComplicationTemplateGraphicCircularImage` shows you that it requires a `CLKFullColorImageProvider`. You've got a helper method for that!

At this point, you can see how simple it becomes to add new complication families to your app. Beyond that, maintenance is contained in a single file, named after the complication.

If you decide to switch the `.graphicCircular` family from a `CLKComplicationTemplateGraphicCircularStackImage` to a `CLKComplicationTemplateGraphicCircularImage`, the update is quick and simple. You know that **GraphicCircular.swift** is the only file you'll need to edit.

Freshness

Great work! You implemented your first complication. Have you noticed the issue with the data? Your complication is only going to be correct if your customer's run the app hourly.

In the next chapter, you'll learn how to use the future data you've downloaded, as well as keep the data up-to-date if the user doesn't run the app some time.

Key points

- The complication controller's methods are all asynchronous.
- Using a factory pattern makes adding newly supported complication families incredibly simple.
- Support as many complication families as possible to provide the best user experience.Even sim

Where to go from here?

The sample project in **final** shows implementations of almost all the supported complication types. You'll learn about the SwiftUI-specific complications in a later chapter.

Apple's Human Interface Guidelines (https://apple.co/3pusUcN) for watchOS contains a wealth of useful material related to complications. For example, you'll find image size and composition guidance, descriptions of each family type and example images of how the complication family appears on the watch face.

If you'd like to dive deeper into **Design Patterns**, like the Factory Method design pattern that you implemented in this chapter, please check out our book, Design Patterns by Tutorials (https://bit.ly/3y9yPH7).

Chapter 10: Keeping Complications Updated

By Scott Grosch

Now that your complications are available to place on the watch face, you need to address one last consideration. How do you ensure that the displayed data is up to date?

You learned how to reload the timeline when new data becomes available. Now it's time to learn different ways to retrieve that data. There are four options available to you:

- Update based on changes while the app is active.
- Schedule background tasks to make changes.
- Schedule background `URLSession` downloads.
- Send notifications via `PushKit`.

Since you've already learned how to reload a timeline, this chapter will address the latter three techniques.

Scheduled background tasks

There will be times when you know an update should take place in the future, but the watch likely won't be running your app during that time. Calling `scheduleBackgroundRefresh(withPreferredDate:userInfo:scheduledCompletion:)` from `WKExtension` lets you specify a future date when watchOS should wake your app up in the background and perform work.

When watchOS starts the background task, your app gets four seconds of CPU time and 15 seconds of total time to complete the task. While you're allowed to schedule up to four background tasks per hour, you may only have one scheduled at any given time. If you schedule a second task while one is already scheduled, the previous task will cancel automatically.

Open **ExtensionDelegate.swift** from this chapter's starter materials. When watchOS launches your app to perform a background task, it calls the `handle(_:)` method from `WKExtensionDelegate`.

Add the following method to `ExtensionDelegate`:

```
func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
  // 1
  backgroundTasks.forEach { task in
    // 2
    switch task {
    default:
      // 3
      task.setTaskCompletedWithSnapshot(false)
    }
  }
}
```

Here's what's happening:

1. watchOS provides you with one *or more* tasks, so you must iterate through each task.
2. `WKRefreshBackgroundTask` is a base class, which you'll need to examine to determine the specific subclass. You'll implement that check in a moment.
3. If the task type provided isn't one you care about, mark the task as completed. Pass `false` so that a new snapshot isn't scheduled since you haven't performed any changes.

Refer back to Chapter 5, "Snapshots", if you need a refresher.

There are four steps involved when a background task launches:

1. Perform the necessary work to complete the task.
2. Update your complications *if* something has changed based on the task.
3. Schedule the next background task, if required.
4. Mark the task as completed.

> **Note**: Pay special attention to the fact that you need to schedule the next background task *before* marking the current task as complete. `watchOS` will stop providing cycles to your app once you specify the task is done.

The background worker

Create a new file named **BackgroundWorker.swift** and add:

```
import Foundation
import WatchKit

final class BackgroundWorker {
  // 1
  public func schedule(firstTime: Bool = false) {
    let minutes = firstTime ? 1 : 15

    // 2
    let when = Calendar.current.date(
      byAdding: .minute,
      value: minutes,
      to: Date.now
    )!

    // 3
    WKExtension
      .shared()
      .scheduleBackgroundRefresh(
        withPreferredDate: when,
        userInfo: nil
      ) { error in
        if let error = error {
          print("Unable to schedule: \
(error.localizedDescription)")
        }
      }
  }
}
```

Your worker needs to be able to schedule jobs:

1. If the app is just starting, you might need to schedule a first background job immediately. If it's the first run, then start a minute from now, otherwise start 15 minutes later. Remember, you only get four updates an hour. So you need to wait at least 15 minutes for subsequent calls if you plan to spread the calls over the hour.
2. Calendrical calculations should be familiar by now. You're simply adding the number of `minutes` to the current time.
3. Schedule the job to run at the desired time.

If you need data to be available to the job when watchOS launches it, use the `userInfo` parameter.

Add one more method to complete your background worker class:

```
public func perform(_ completion: (Bool) -> Void) {
  // Do your background work here
  completion(true)
}
```

When `ExtensionDelegate` is ready to run your scheduled job, it'll call the `perform(_:)` method. Handle all the required work, then call the completion handler with `true` if the active complications should update with new values, otherwise `false`.

ExtensionDelegate background task

Switch back to **ExtensionDelegate.swift** and add a new property to `ExtensionDelegate`:

```
private let backgroundWorker = BackgroundWorker()
```

Then add the following code to the `switch` statement in the `handle(_:)` method:

```
// 1
case let task as WKApplicationRefreshBackgroundTask:
  // 2
  backgroundWorker.perform { updateComplications in
    // 3
    if updateComplications {
      Self.updateActiveComplications()
    }
```

```
        // 4
        backgroundWorker.schedule()
        task.setTaskCompletedWithSnapshot(false)
    }
```

Here, you:

1. Check if the current `task` is of type `WKApplicationRefreshBackgroundTask`.
2. Call the `perform` method and supply the completion handler to call when the work finishes.
3. If you passed `true` to the completion handler, you tell the complications to update themselves.
4. Finally, you schedule the next background task and mark the task as completed.

Notice how you mark the task as completed *inside* of the completion handler. `handle(_:)` will complete before your job finishes. If you mistakenly mark the task as complete *outside* of the completion handler, your job will never fully run because watchOS will terminate it.

> **Note**: If your complications are updated, watchOS will schedule a snapshot automatically. Therefore, always pass `false` to `task.setTaskCompletedWithSnapshot`.

Depending on your app's requirements, it may not make sense to automatically schedule the next task. The pattern above assumes that you call `schedule(true)` from somewhere like `applicationDidFinishLaunching` and need to repeat on a known time cycle.

While most frameworks are available to your app during a background task, the notable exception is URL downloads. If you try to perform a URL download from a background task, watchOS will hand you an error.

Background URL downloads

Downloading data from the network follows the same general pattern as background tasks. However, they're a bit trickier as network downloads require delegates, and there could be more than one running at a time.

Like background tasks, you can run up to four downloads per hour. Unlike background tasks, you can run all four at once if you wish. While the specific details are unclear, Apple warns that the actual number depends on factors such as Wi-Fi availability, cellular signal strength and battery life.

URLSession setup and configuration

Create a file named **UrlDownloader.swift** and add:

```
import Foundation

// 1
final class UrlDownloader: NSObject {
  // 2
  let identifier: String

  init(identifier: String) {
    self.identifier = identifier
  }

  // 3
  private lazy var backgroundUrlSession: URLSession = {
    // 4
    let config = URLSessionConfiguration.background(
      withIdentifier: identifier
    )

    // 5
    config.isDiscretionary = false

    // 6
    config.sessionSendsLaunchEvents = true

    // 7
    return .init(
      configuration: config,
      delegate: self,
      delegateQueue: nil
    )
  }()
}
```

```
// 8
extension UrlDownloader: URLSessionDownloadDelegate {
  func urlSession(
    _ session: URLSession,
    downloadTask: URLSessionDownloadTask,
    didFinishDownloadingTo location: URL
  ) {
  }
}
```

Lots of code, but nothing too confusing:

1. You declare a class to handle URL downloads. It needs to implement `URLSessionDownloadDelegate`, so it has to subclass from `NSObject`.
2. Background `URLSession` tasks are assigned identifiers, so you provide callers with a way to specify which identifier to use. Subclassing `NSObject` means you'll need to provide an explicit initializer.
3. `URLSession` should only be created once, on-demand.
4. Background downloads must use the special `URLSessionConfiguration` and be provided with your desired `identifier`.
5. By setting `isDiscretionary` to `false`, which is the default, you tell watchOS that it should try to run your download as soon as you ask it to, instead of letting it determine the best time.
6. Setting `sessionSendsLaunchEvents` to `true`, the default, tells watchOS to automatically wake up or launch your app in the background when required.
7. Finally, you create the `URLSession` object with your specified configuration. watchOS will create a serial operation queue to handle the delegate callbacks if you specify `nil` for the `delegateQueue` parameter.
8. You'll learn more about the delegate in just a bit, but you need to implement the protocol to keep Xcode from displaying errors.

Scheduling a network download

To schedule the download, add a new property to `UrlDownloader`:

```
private var backgroundTask: URLSessionDownloadTask?
```

Then implement the scheduling method:

```
// 1
func schedule(firstTime: Bool = false) {
  let minutes = firstTime ? 1 : 15

  let when = Calendar.current.date(
    byAdding: .minute,
    value: minutes,
    to: Date.now
  )!

  // 2
  let url = URL(
    string: "https://api.weather.gov/gridpoints/TOP/31,80/
forecast"
  )!
  let task = backgroundUrlSession.downloadTask(with: url)

  // 3
  task.earliestBeginDate = when

  // 4
  task.countOfBytesClientExpectsToSend = 100
  task.countOfBytesClientExpectsToReceive = 12_000

  // 5
  task.resume()

  // 6
  backgroundTask = task
}
```

In the preceding code:

1. The initial setup, including date calculation, is the same as for background tasks.
2. You generate a download task using the `backgroundSession` you just configured.
3. By setting `earliestBeginDate`, you let watchOS know that it shouldn't start the network download before the indicated date. If the API you call uses caching headers, use those to help define the earliest beginning date.
4. Telling watchOS exactly how many bytes you expect to send, including header count, and receive helps it optimize when to perform the download. Pay close attention to the property names! There are multiple properties with almost the same name, and it's easy to get confused.

5. If you forget to call `resume`, the network download won't start.
6. Finally, you store the task in your class' property to use in delegate methods.

> **Note**: Use the delegate pattern for background URL downloads. You may not use the newer `async` methods.

URLSessionDownloadDelegate

When the `backgroundTask` finishes downloading, watchOS will call the `urlSession(_:downloadTask:didFinishDownloadingTo:)` defined by `URLSessionDownloadDelegate`. Add the following code to that method:

```
// 1
let decoder = JSONDecoder()

guard
  // 2
  location.isFileURL,
  // 3
  let data = try? Data(contentsOf: location),
  // 4
  let decoded = try? decoder.decode(Weather.self, from: data),
  // 5
  let temperature =
decoded.properties.periods.first?.temperature
else {
  return
}

// 6
UserDefaults.standard.set(temperature, forKey: "temperature")
```

In the preceding code:

1. The data provided by the API call is JSON, so you need a way to decode it.
2. The location you provide should be a file URL. The check is probably not entirely necessary, but better safe than sorry. :]
3. You read the contents of the file that watchOS wrote the data to.
4. Then, you decode the data based on the `Weather` structure provided with the sample project.

5. You grab the first temperature provided.
6. Then, store the downloaded temperature to `UserDefaults` so that you can access it in the complication. The provided `ComplicationController` class displays the temperature stored in this location.

Keep in mind, when this delegate method ends, watchOS will automatically delete the file at `location`. If you download an image or movie, copy the file somewhere appropriate. If you downloaded JSON data, such as in this example, store the data as needed in something like Core Data, `@AppStorage` or `UserDefaults`.

> **Note**: In this example, you grab whatever temperature is *first*, which isn't really the *current* temperature. Book…sample…you get it.

Once the session has fully completed, watchOS will call the `urlSession(_:task:didCompleteWithError:)` delegate method. Add that method to your delegate implementation:

```
func urlSession(
  _ session: URLSession,
  task: URLSessionTask,
  didCompleteWithError error: Error?
) {
  backgroundTask = nil

  DispatchQueue.main.async {
    self.completionHandler?(error == nil)
    self.completionHandler = nil
  }
}
```

Recall that the delegate methods run on a serial dispatch queue. When you call the completion handler, you need to dispatch that back to the main queue.

Xcode isn't so happy right now about that whole completion handler thing. Fix that up now.

Preparing for download

There's one piece left to your `UrlDownloader` class. In `UrlDownloader`, add another property:

```
private var completionHandler: ((Bool) -> Void)?
```

Then implement the `perform` method:

```
public func perform(_ completionHandler: @escaping (Bool) ->
Void) {
  self.completionHandler = completionHandler
  _ = backgroundUrlSession
}
```

Err…uh…that looks pointless! You've stumbled upon the major confusion of background URL downloads.

Recall that your app may go in and out of background mode while the download occurs. When watchOS re-attaches your app, you have to let it know that it should reuse the previous session, so it calls the delegate methods properly.

By recreating a session with the *exact same identifier* that you originally used, watchOS ties everything together for you. Assigning to the _ variable discards the result because you don't need to hold onto it but has the side effect of creating the session again if it doesn't already exist.

ExtensionDelegate network download

Switch back to **ExtensionDelegate.swift** and add a property to track network downloads:

```
private var downloads: [String: UrlDownloader] = [:]
```

Then add a helper method to manage the dictionary:

```
// 1
private func downloader(for identifier: String) -> UrlDownloader
{
  // 2
  guard let download = downloads[identifier] else {
    let downloader = UrlDownloader(identifier: identifier)
    downloads[identifier] = downloader
    return downloader
  }

  // 3
  return download
}
```

Here's what the code is doing:

1. You declare a method that returns a `UrlDownloader` for a given `identifier`.
2. If the `UrlDownloader` for the given identifier doesn't already exist, create a new one.
3. If it does exist, directly return it.

Finally, add another `case` to the `switch` statement:

```
// 1
case let task as WKURLSessionRefreshBackgroundTask:
  // 2
  let downloader = downloader(for: task.sessionIdentifier)

  // 3
  downloader.perform { updateComplications in
    if updateComplications {
      Self.updateActiveComplications()
    }

    downloader.schedule()
    task.setTaskCompletedWithSnapshot(false)
  }
```

The code is quite similar to the `WKApplicationRefreshBackgroundTask` code:

1. You verify that you received a `WKURLSessionRefreshBackgroundTask`.
2. Using your helper method, you grab the appropriate `UrlDownloader` instance for the task's `sessionIdentifier`.
3. You call the `perform` method and pass in a completion handler to call when the download completes. Like before, you update your complications, schedule the next download and mark the task as complete.

> **Note**: If your complications are updated, watchOS will automatically schedule a snapshot. Therefore, always pass `false` to `task.setTaskCompletedWithSnapshot`.

If your network download requires extra delegate events, such as authentication challenges, you'll need to call the completion handler from the `urlSessionDidFinishEvents(forBackgroundURLSession:)` delegate method as well. However, you must *not* schedule a new download at that point because the download itself hasn't happened yet.

Updating ContentView

Add a new property to **ContentView.swift**:

```
@State private var downloader = UrlDownloader(identifier:
"ContentView")
```

Remember, you can use any name for the `identifier` parameter. It just has to stay consistent for the same type of downloads.

Then replace the entire `body` with:

```
Button {
  downloader.schedule(firstTime: true)
} label: {
  Text("Download")
}
```

Build and run the app. Once it launches, return to the home screen and add the Updates complication to your watch face. Tap the complication to launch the app, and then tap **Download**.

Press the Digital Crown to go back to the home screen and then drop your wrist. Around a minute from now, watchOS will perform the background download and update the complication on your watch face, showing a temperature.

Push notifications

> **Note**: At the time of writing, watchOS has a bug — verified by Apple — that sometimes prevents your app from registering for `PushKit` notifications. Apple told me it believes it has determined the root cause. However, there's no ETA on when the fix will be available.

While background tasks and background URL downloads work well, they're not always the best solution. watchOS may kill your app, or it may crash. I know, *your* apps are 100% bug-free, but that stuff your colleague writes…

If you control the server your app pulls data from, it may make more sense to implement complication updates via push notifications. Using **PushKit**, you can send up to 50 updates per day to your Apple Watch.

PushKit registration

Complication push notifications are a bit different from standard remote push notifications. Create a new file named **PushNotificationProvider.swift** and add:

```
import Foundation
import PushKit

// 1
final class PushNotificationProvider: NSObject {
  // 2
  let registry = PKPushRegistry(queue: .main)

  override init() {
    super.init()

    // 3
    registry.delegate = self

    // 4
    registry.desiredPushTypes = [.complication]
  }
}
```

PushKit setup is pretty straightforward:

1. You'll conform to a delegate protocol in a moment, so you must subclass `NSObject`.
2. Initialize PushKit and specify that the delegate methods should be called on the main UI thread.
3. Assign this class as the delegate.
4. Specifying the `desiredPushTypes` lets PushKit know you're sending complication updates.

Now start implementing the delegate. Add the following code to the end of the file, outside of the `PushNotificationProvider` class:

```
// 1
extension PushNotificationProvider: PKPushRegistryDelegate {
  // 2
  func pushRegistry(
    _ registry: PKPushRegistry,
    didUpdate pushCredentials: PKPushCredentials,
    for type: PKPushType
  ) {
    // 3
```

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

      // Send token to your server.
    }
  }
```

Implementing the delegate starts with these steps:

1. Your class needs to conform to `PKPushRegistryDelegate`.
2. watchOS calls `pushRegistry(_:didUpdate:for:)` when your app successfully registers with PushKit.
3. Convert the unusable `Data` token to a string that can be sent to your server.

PushKit suffers from the same usability annoyance as standard push notifications: watchOS gives you a `Data` token instead of a usable string.

Once you decode the token, you'll send it to your server. Be sure you somehow specify, on your server, that the token is for PushKit, not a normal push notification.

Receiving a push notification

To handle receiving a PushKit notification, implement the following delegate method:

```
// 1
func pushRegistry(
  _ registry: PKPushRegistry,
  didReceiveIncomingPushWith payload: PKPushPayload,
  for type: PKPushType
) async {
  // 2
  print(payload.dictionaryPayload)

  // 3
  await ExtensionDelegate.updateActiveComplications()
}
```

In the preceding code:

1. Notice that the method signature includes the `async` keyword. Having this method asynchronous makes life easier as the updates you perform are likely asynchronous.
2. The payload sent with the push notification is available in the payload's `dictionaryPayload` property.
3. Once you take appropriate action based on the incoming payload, tell your complications to update.

apns-topic

There's one special consideration when sending a PushKit notification. The **apns-topic** header should be the name of your *extension's* bundle identifier with `.complication` appended to it. For example, when sending a notification to the sample app, you would set the apns-topic to `com.raywenderlich.Updates.watchkitapp.watchkitextension.complication`.

Testing

Other than the extra text you need to add to the apns-topic, there's nothing special about testing push notification. You can use the same server-side tools that you use for the rest of your app's push notifications.

If you're able to register your app for `PushKit` notifications, you can test it the same way you would do with an iPhone. Make sure you have your app notifications activated on the Apple Watch. It is recommended to use a real device for testing it. There are some interesting frameworks which allow sending notifications to the simulator. However, it is out of the scope of this book explaining how to set them up.

Key points

- Always schedule the next task before marking the current task as complete.
- Call your completion handler from `urlSessionDidFinishEvents(forBackgroundURLSession:)` if using authentication, but do *not* schedule the next task at that point.
- The WWDC 2020 session, _Keep your complications up to date (https://apple.co/3vGlmqx)_, says to create an app identifier with a *.complication* suffix. However, that guidance is no longer valid, as per Apple.
- Append `.complication` to the extension's bundle identifier for the apns-topic header.

Chapter 11: Tinted Complications

By Scott Grosch

The default setting for the Apple Watch shows complications in full-color. When selecting and populating a watch face, the user may instead choose to select a tint color. If they select a tint, all elements on the watch face will change to honor the selected tint color.

Full-color

Open **Happy.xcodeproj** from this project's starter materials. Then press **Control-0** (that's a zero) to select the **Happy WatchKit App (Complication)** scheme. This chapter focuses solely on the complication, not the app itself, so switching to the complication scheme makes sense instead of running the app.

Build and run. Once launched, long-press the watch face to bring up the face editor:

Then swipe left to get to the add new face screen:

Tap **Add**, then scroll to the bottom of the list to select the **X-Large** face:

Once you've tapped **Add**, the color selection screen will appear. Scroll to the **MULTICOLOR** choice if it's not already selected:

Swipe left again to see the empty complications screen.

After tapping the empty square, scroll down to the **Happy** app:

Tap to select it, then press the home button twice.

Looks amazing, right? :]

What happens when the watch face is tinted?

Desaturation

By default, if the user selects a tint color for the watch face, watchOS will **desaturate** the full-color image. Desaturation is the process of making colors more muted. By adding more black or white to the image, the colors become less vibrant. The more you desaturate an image, the more color you remove.

To see an example of what that means, select a tint color by following steps similar to those you performed when adding the watch face:

1. Long-press the watch face.
2. Tap **Edit**.
3. Swipe right to the color screen.
4. Scroll through the colors.
5. Return to the Home screen by pressing the Digital Crown twice.

Scrolling through the colors lets you see what the image would look like for each tint color. Depending on the image you're using, desaturation might work just fine. However, if your colors are too similar to each other in hue, the desaturation might not work as the image blends into itself too much.

Layered images

When you require more control over how the image looks while tinted, you can split it into two separate images. For example, consider the smiley-face image you're using to have two separate parts. The circular part of the face, in green, is the background. The eyes and mouth are then the foreground.

Open **Assets.xcassets**, and you'll see the **Full** image currently in use. There are also two other images. Click **Background**, and you'll see it's simply the green circle from the face. Next, click **eyesAndMouth**, and you'll see...wait for it...the eyes and mouth. :]

Open **ComplicationController.swift** and replace everything in `currentTimelineEntry(for:)`, except for the `return` statement, with:

```
// 1
guard
  let full = UIImage(named: "Full"),
  let background = UIImage(named: "Background"),
  let eyesAndMouth = UIImage(named: "eyesAndMouth")
else {
  fatalError("Images are missing from the asset catalog.")
}

let template =
CLKComplicationTemplateGraphicExtraLargeCircularImage(
  imageProvider: CLKFullColorImageProvider(
    // 2
    fullColorImage: full,
    // 3
    tintedImageProvider: .init(
      // 4
      onePieceImage: full,
      // 5
      twoPieceImageBackground: background,
      // 6
      twoPieceImageForeground: eyesAndMouth
    )
  )
)
```

Here's what happening:

1. First, you create the images you'll use. The `fatalError` makes sense as you want to catch typos before shipping the app.
2. Just like before, you specify the full-color image to display.
3. Using the two-parameter initializer to `CLKFullColorImageProvider`, you can now specify exactly what to do when tinting.
4. Specify the same full-color image again.
5. Then specify that the colored circle is the background layer.
6. The eyes and mouth will be the foreground layer.

watchOS will apply tinting based on the two separate layers you provide. Sometimes it will tint the foreground layer. Other times it will tint the background layer. Which tint receives the user's preferred color is dependent on the type of complication family you select.

Build and run again. After the app starts and the simulator updates, you'll see the effect of your changes:

As you can see, when using a `CLKComplicationTemplateGraphicExtraLargeCircularImage` template, watchOS will apply the tint color to the background. watchOS chooses the foreground color.

When using layered images, watchOS only uses the opacity. It completely ignores the color.

SwiftUI complications

SwiftUI graphic complication views also support tinting but with a different syntax.

Create a new SwiftUI View file called **HappyComplication.swift** and replace the contents of the body with the full-color image:

```
Image("Full")
  .resizable()
  .aspectRatio(contentMode: .fit)
```

> **Note**: If the Canvas isn't showing, press **Alt-Command-Enter** to bring it up.

Desaturation

By default, SwiftUI complications will be desaturated, like the non-SwiftUI versions. Add **ClockKit** to the top of the file:

```
import ClockKit
```

Then replace the `previews` view with code to show as a complication:

```
CLKComplicationTemplateGraphicExtraLargeCircularView(
  HappyComplication()
)
  .previewContext()
```

Looks great, but it's not tinted. The `previewContext` will take a `faceColor` parameter to support tinting. The parameter isn't a `Color`, though. It's a `CLKComplicationTemplate.PreviewFaceColor` enum.

Why does that matter? You can use a loop to see multiple versions at once. Replace your preview code again with this:

```
// 1
Group {
  // 2
  ForEach(CLKComplicationTemplate.PreviewFaceColor.allColors) {
    CLKComplicationTemplateGraphicExtraLargeCircularView(
      HappyComplication()
    )
      // 3
      .previewContext(faceColor: $0)
  }
}
```

The previous code performs these actions:

1. Using a `Group` lets you display multiple watch faces at once.
2. Using the `.allColors` enumeration value, you loop through each predefined color.
3. You pass the face color to the preview context to tint to that color.

A couple of years later, once the Canvas has finally refreshed, you'll see your complication first in full-color and then tinted to the seven pre-defined preview colors. Using the `enum` value is an excellent way to ensure your complication looks good across multiple colors.

Layered

To use layered tinting, change the image name from **Full** to **eyesAndMouth**. Next, add another modifier to the `Image` like so:

```
.complicationForeground()
```

By adding that modifier, you let watchOS know that it should consider the image as the foreground layer of the tinting. At this point, your preview will show all the mouths, except the first, in white.

What about the rest of the face? You have to create the background layer as well. Wrap the `Image` with:

```
ZStack {
  Circle()

  // The Image here
}
```

When the Canvas refreshes the preview, you'll once again have a "body" for your face, with the proper background color, even though you didn't specify a color explicitly.

As a quick test, replace `CLKComplicationTemplateGraphicExtraLargeCircularView` with `CLKComplicationTemplateGraphicRectangularFullView` in the preview. Notice how the colors are reversed in the full rectangular face. Remember that watchOS decides whether to tint the foreground or the background, depending on the family.

Before continuing through the chapter, please switch back to the `CLKComplicationTemplateGraphicExtraLargeCircularView`.

Rendering modes

There will be times when you still want a bit more control, depending on whether the user tints the watch face. SwiftUI has you covered!

Add the following property to the top of your view:

```
@Environment(\.complicationRenderingMode) var renderingMode
```

watchOS will set that property to `.fullColor` if the watch face displayed is multicolor or `.tinted` if the user has a tinted face. Replace the body with:

```
ZStack {
  // 1
  if renderingMode == .fullColor {
    // 2
    Image("Full")
      .resizable()
      .aspectRatio(contentMode: .fit)
      .complicationForeground()
  } else {
    // 3
    Circle()

    Image("eyesAndMouth")
      .resizable()
      .aspectRatio(contentMode: .fit)
      .complicationForeground()
  }
}
```

In the preceding code:

1. *Inside* of the `Stack`, you check to determine if the user wants a full-color image.
2. If they do, then you show the normal full-color image you created.
3. If the user wants tinting, you show the tinted versions.

Now, when you look at the Canvas, the first image is still the full-color version. By checking the rendering mode, you now have greater control.

You may want completely different images, depending on whether the complication is full-color or not. To see this in action, add the following code right after the `Circle()` line:

```
.fill(LinearGradient(
  gradient: Gradient(
    colors: [.red.opacity(0.3), .blue.opacity(1.0)]
  ),
  startPoint: .top,
  endPoint: .bottom))
```

You filled the circle with a linear gradient, from top to bottom. That's standard SwiftUI code. However, notice the specified colors. The first is red, and the second is blue. I purposely chose colors so that you could see the actual color is ignored. When using a tinted complication, remember that watchOS *only* uses the opacity.

It makes for a bit of a creepy smiley face:

Update the complication controller

So far, you've performed all of your previewing of the complication from Xcode's Canvas. Remember, to let the user pick your updated complication you must import SwiftUI in **ComplicationController.swift**:

```
import SwiftUI
```

And then modify `currentTimelineEntry(for:)` to use the SwiftUI version:

```
let template =
CLKComplicationTemplateGraphicExtraLargeCircularView(
  HappyComplication()
)

return .init(date: Date.now, complicationTemplate: template)
```

Now build and run. You'll see the creepy smile with the gradient you just added.

Key points

- Be sure to look at your complications on tinted watch faces.
- Split your `Image` or `Shape` items into foreground and background layers.
- When using tinted complications, watchOS ignores the color and only honors the opacity.

Where to go from here?

In the following chapter, you'll continue to use complications. But this time, you will focus on SwiftUI complications to round up your knowledge on this useful and sometimes underestimated Apple Watch feature. If you want to get a hint of what to expect:

- raywenderlich.com Tutorial _Complications for watchOS with SwiftUI (https://bit.ly/3aLYWL4)_.

Chapter 12: SwiftUI Complications

By Scott Grosch

For most complications, you'll use the templates you learned about in Chapter 8: "Complications Introductory". Sometimes, however, you'll want greater control over the design. watchOS lets you design *some* complications with SwiftUI.

Graphs are, of course, a great use case for SwiftUI-based complications. That's no fun, though, so you're going to show the next calendar entry for today.

Showing an appointment

Open **CalendarComplication.xcodeproj** from this chapter's starter materials. **EventStore.swift** reads the local calendar via **EventKit**, as well as **EventView.swift**, which is the view you'll modify.

Build and run the project on a physical device. Your watch will ask you to grant calendar permissions, which of course, you must agree to for the project to work.

> **Note**: You must use a physical device because the Simulator doesn't include a calendar. Also, **EventKit** will only display calendar items from your local calendar, not calendars in the cloud.

Event display

Edit **EventView.swift**, the view you'll see in the complication. I already performed the pieces related to EventKit to save you some time. Your goal is to create a display similar to the one Apple provides for their Calendar app.

Find the "Hello, World!" `Text` line and replace it with an `HStack` that contains a vertical line:

```
HStack {
  // 1
  RoundedRectangle(cornerRadius: 3)
    // 2
    .frame(width: 5)
    // 3
    .foregroundColor(Color(event.calendar.cgColor))
}
```

In the preceding code:

1. You draw a rectangle with a small corner radius, so the edges are slightly rounded.
2. By making it five pixels wide, you essentially create a vertical line.
3. EventKit provides the calendar color as a `CGColor`, which you then convert to a SwiftUI `Color`, using that for the line's color.

Calendar appointments usually show a time range instead of just the start time. To format the dates, add a new property to the top of the `struct`:

```
// 1
private let formatter: DateIntervalFormatter = {
  let formatter = DateIntervalFormatter()
  // 2
  formatter.dateStyle = .none
  formatter.timeStyle = .short
  return formatter
// 3
}()
```

That pattern might look a bit strange to you if you're new to Swift. Here's what's happening:

1. You let the compiler know that you're creating a property of type `DateIntervalFormatter`, which you then configure.
2. The formatter won't show dates and will use a short time style, like 2:00 pm.
3. The `}()` closes the configuration, runs it and then assigns the result to the formatter. This type of structure lets you perform the necessary configuration for a property right with the initialization.

Excellent. Now you're ready to show the dates. Inside the `HStack`, after the rectangle, show the details of the calendar item:

```
// 1
VStack(alignment: .leading) {
  // 2
  Text(formatter.string(from: event.startDate, to:
event.endDate))
    .font(.subheadline)
  // 3
  Text(event.title)
    .font(.headline)
  // 4
  if let location = event.location {
```

```
      Text(location)
        .font(.subheadline)
    }
  }
```

Pretty standard SwiftUI there:

1. Creating a `VStack` with the `.leading` alignment ensures the start of the text items will line up. By default, the items would be centered.
2. You show the date range for the appointment in a `.subheadline` font for a slightly smaller size.
3. Using a `.headline` font makes the title slightly larger than the date range.
4. Locations are optional, so you only show the location if one is set.

If you have the Canvas displayed, you'll notice that you don't see the appointment due to calendar permissions. That's not ideal.

Build and run the app. As long as you've created an appointment on your *local* calendar, the one labeled **ON MY IPHONE**, you'll see the event:

Event refactoring

Right now, the code has multiple issues. Not only are you unable to preview the complication, but also everything is tightly tied to **EventKit**. What happens when you work with **CalDAV** or any of the other calendaring platforms?

Creating an Event type

Create a new file named **Event.swift** and paste in:

```
import SwiftUI
import EventKit

// 1
struct Event {
  let color: Color
  let startDate: Date
  let endDate: Date
  let title: String
  let location: String?

  // 2
  init(ekEvent: EKEvent) {
    color = Color(ekEvent.calendar.cgColor)
    startDate = ekEvent.startDate
    endDate = ekEvent.endDate
    title = ekEvent.title
    location = ekEvent.location
  }

  // 3
  init(
    color: Color,
    startDate: Date,
    endDate: Date,
    title: String,
    location: String?
  ) {
    self.color = color
    self.startDate = startDate
    self.endDate = endDate
    self.title = title
    self.location = location
  }
}
```

There's nothing magical in the preceding code:

1. You've created a custom type to hold your calendar events.
2. `EKEvent` is a common use case, so providing a constructor that takes that type simplifies the rest of the codebase.
3. Sometimes, especially for previews, you'll want to specify the different values manually.

If your app expands in the future to include other calendar types, you would simply add a new initializer to this file.

Notice how you're expecting callers to pass you a `Color`, not a `CGColor`. Your use case is currently for events, which use `CGColor`, but you may be passing events from a web download, for example.

Refactoring the view

Sometimes you find that you've named a view poorly and need to fix it. In this case, `EventView` should really be `EventComplicationView` because you need an `EventView` for previewing the event.

Perform the following steps:

1. Open **EventView.swift**.
2. Right-click the `EventView` name, then choose **Refactor ▸ Rename...**
3. Place the mouse just before the **V** and type **Complication** so that the view is named **EventComplicationView**, then press **Enter**. With that step, you rename all instances of the view across the entire project as well as the filename.
4. You'll need to manually rename `EventView_Previews` to `EventComplicationView_Previews`.
5. Select the entire `HStack` and press **Control-X** to cut the code and add it to the clipboard.
6. Create a new SwiftUI View named **EventView.swift** and paste the `HStack` as the contents of the body, replacing `Text("Hello, World!")`.

At this point, Xcode isn't happy because it doesn't know what `event` means. Add the following property:

```
let event: Event
```

Then, replace `Color(event.calendar.cgColor)` with `event.color`.

Next, move the `DateIntervalFormatter` from **EventComplicationView.swift** to **EventView.swift**.

Your refactored view is now complete, with the exception of fixing the `EventView_Previews`. Add a `static` property to represent an `Event`:

```
static var event = Event(
  color: .blue,
  startDate: .now,
  endDate: .now.addingTimeInterval(3600),
  title: "Gnomes rule!",
  location: "Everywhere"
)
```

Since this is only for a development preview, it's OK to skip the standard calendrical calculation methods and directly add one hour to the current time.

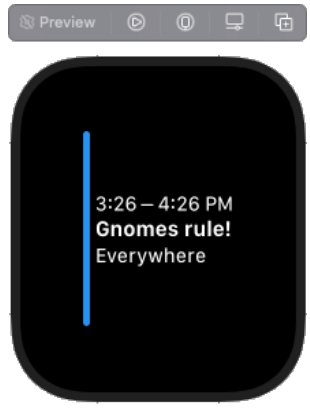
You want to see the view as it will appear on a complication. So, import **ClockKit** at the top of the file:

```
import ClockKit
```

Then replace the `previews` body with:

```
Group {
  EventView(event: event)

  CLKComplicationTemplateGraphicRectangularFullView(
    EventView(event: event)
  )
    .previewContext()
}
```

If it's not already showing, bring up the canvas by pressing **Option-Command-Enter**. The canvas will now show what the view looks like by itself and how it appears on the watch face:



Now that you know the display looks the way you wanted, switch back to **EventComplicationView.swift**.

Add a new property to the view:

```
let event: Event?
```

Find the line where the `Stack` used to be, plus the `if` check that went with it:

```
} else if let event = eventStore.nextEvent {
```

Replace with these lines:

```
} else if let event = event {
  EventView(event: event)
```

You're simply passing the event-specific details to the view that shows them properly. In `EventComplicationView_Previews`, pass `nil` to make the compiler happy:

```
EventComplicationView(event: nil)
```

Now, build the app to ensure you don't have any compiler errors from missed steps.

Whoops! You didn't fix **CalendarComplicationApp.swift** to use the new complication view as the entry point instead of `EventView`. It's definitely not because I forgot to tell you to do so. :]

OK, replace:

```
EventComplicationView()
```

With:

```
EventComplicationView(event: nil)
```

Now the app builds without errors. If you run the app right now, you'd always see a message saying there were no more events for today. Why? You just told `CalendarComplicationApp` to pass `nil` for the event. Remember, this app is all about the complication, so you don't really care about what displays if you *run* the app.

The event complication

It's finally time to use your SwiftUI view in a complication.

This example uses the `.graphicRectangular` family since it works well for a calendar display. To keep the example focused, you won't implement any other complication families.

Event to timeline entry

All the complication methods need to be able to create a `CLKComplicationTimelineEntry` from an `EKEvent`. So, add the following method to **ComplicationController.swift**:

```
private func timelineEntry(for ekEvent: EKEvent?) ->
CLKComplicationTimelineEntry {
  // 1
  let event: Event?
  if let ekEvent = ekEvent {
    event = Event(ekEvent: ekEvent)
  } else {
    event = nil
  }

  // 2
  let template =
CLKComplicationTemplateGraphicRectangularFullView(
    EventComplicationView(event: event)
  )

  // 3
  return .init(
    date: event?.startDate ?? .now,
    complicationTemplate: template
  )
}
```

The code creates your timeline entry:

1. First, you convert the `EKEvent` you have to an `Event` since that's what your views expect. Remember, if you pass `nil` to the `event` parameter of the `EventComplicationView` initializer, it will display a message saying there are no more events.
2. `CLKComplicationTemplateGraphicRectangularFullView` is one of the template types which expects you to give it a SwiftUI view to use as a template.
3. You generate a `CLKComplicationTimelineEntry` based on the event's start date and the SwiftUI template. If there isn't an event, then use the current date.

Since you're using `EKEvent`, you'll need to import `EventKit` at the top of the file:

```
import EventKit
```

Localizable sample

It's important to have a sample complication for users to see when they're choosing complications. Provide one with the following delegate method, still in **ComplicationController.swift**:

```
func localizableSampleTemplate(
  for complication: CLKComplication
) async -> CLKComplicationTemplate? {
  // 1
  let start = Calendar.current.date(
    bySettingHour: 10, minute: 0, second: 0, of: .now
  )!

  // 2
  let end = Calendar.current.date(
    byAdding: .hour, value: 1, to: start
  )!

  // 3
  return CLKComplicationTemplateGraphicRectangularFullView(
    EventView(event: .init(
      color: .blue,
      startDate: start,
      endDate: end,
      title: "Gnomes rule!",
      location: "Everywhere"
    ))
  )
}
```

Here you:

1. Create a `Date` entry for the current day at 10:00 am. Recall that your date display doesn't include the day, so using today is OK.
2. Add one hour to the start time to indicate when the event ends.
3. Create a `CLKComplicationTemplateGraphicRectangularFullView` with some fake data to display in the sample template.

The current appointment

Just like when using non-SwiftUI templates, you have to provide the current timeline entry if one exists. Replace the body of `currentTimelineEntry(for:)` with:

```
return timelineEntry(for: EventStore.shared.nextEvent)
```

Notice how even if there's not an event, you don't return `nil`. If you return `nil`, you don't get an actual display for the complication, which isn't what you want. If there's no event, you still want the "No more events" message.

Future appointments

If your work calendar is anything like mine, you have way more than one event every day. You'll want to provide future events like you did in previous chapters.

Paste the following method into your code:

```
func timelineEntries(
  for complication: CLKComplication,
  after date: Date,
  limit: Int
) async -> [CLKComplicationTimelineEntry]? {
  // 1
  guard let events = EventStore.shared.eventsForToday() else {
    return [timelineEntry(for: nil)]
  }

  let wanted = events
    // 2
    .filter {
      date.compare($0.startDate) == .orderedAscending
    }
    // 3
    .prefix(limit)
    // 4
    .map { timelineEntry(for: $0) }

  // 5
  return wanted.count > 0 ? wanted : [timelineEntry(for: nil)]
}
```

A lot is going on in that code:

1. If there are no more events for today, return the “No more events” placeholder event.
2. If there are events left, ensure that they’re after the date specified in the `date` method parameter.
3. Honor the `limit` method parameter by only creating that many timeline entries.
4. Convert each `EKEvent` to a `CLKComplicationTimelineEntry`.
5. Finally, if there were events, return them.

Notice how, in step five, you might have zero entries after filtering. Be sure you don’t return `nil`. Instead, return the “No more entries” item.

Build and run the app. Once the app starts on your Apple Watch, add a new watch face using the Modular Compact design. That face includes the graphic rectangular complication. Select your calendar app for the complication and then return to the home screen.

Tinting

In Chapter 11: "Tinted Complications", you learned about tinting. SwiftUI views let you specify the foreground via the `complicationForeground` modifier.

In **EventView.swift**, just after the `.foregroundColor(event.color)` line of `RoundedRectangle`, add:

```
.complicationForeground()
```

Then do the same for the event title and add it right after:

```
Text(event.title)
  .font(.headline)
```

Now, in the `previews` body, change the `previewContext` to:

```
.previewContext(faceColor: .green)
```

In the canvas, you'll see that both the rectangle and the title now receive the tint color:

Ideally, specifying the foreground will be all you need to make your complication look perfect. Depending on whether the watch face is tinted, you might need to do something more drastic at times.

ClockKit provides `ComplicationRenderingMode`, an `enum` with two values:

- `.tinted`: For when the watch face uses tinting.
- `.fullColor`: For when the watch face doesn't use tinting.

The rendering mode is available as an environment variable that you can add to your view with:

```
@Environment(\.complicationRenderingMode) var renderingMode
```

You may wish, for example, to use some type of gradient when the watch face is tinted. By examining the value of `renderingMode` in your code, you can take appropriate action.

Key points

- ClockKit provides multiple graphic complication types that use a SwiftUI `View`.
- SwiftUI views easily support tinting via the `complicationForeground()` modifier.
- For complete tinting control, use the `ComplicationRenderingMode` environment property.

Where to go from here?

For more information, check out these resources:

- Apple's documentation on Building Complications with SwiftUI (https://apple.co/3A8rNDB).
- Apple's WWDC2020 Build complication in SwiftUI (https://apple.co/3or9es5) video.

Chapter 13: Face Sharing

By Scott Grosch

While the ability to create and use **complications** has existed for years, being able to share the perfect watch face with your friends and customers is a new feature. Most apps you develop will only have a single complication. Sometimes, however, you might provide multiple complications.

Consider the **TideWatch** app you built in previous chapters. Currently, there's a single complication that displays the tide. You might wish to extend the app to include complications that provide the current water temperature and wind conditions. When it makes sense to group multiple complications from your app *or other apps*, you can now offer a way for your customers to receive a preconfigured watch face easily.

> **Note**: You'll need a physical device to send a watch face from the iPhone to the Apple Watch.

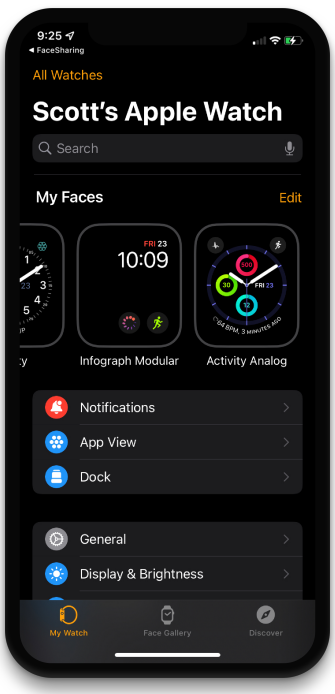
Sharing faces

There are four ways you can share a watch face:

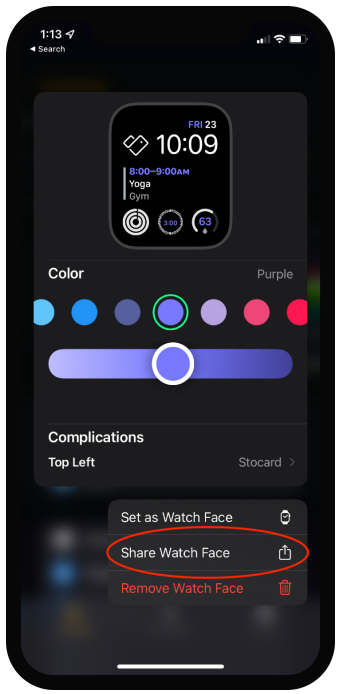
1. Using the Watch app on your iOS device.
2. Directly from your watch.
3. Downloading a **watchface** file from the web.
4. Adding it via your custom iOS app.

From the iPhone

One of the simplest ways to share a watch face is by running the **Watch** app on your iPhone. When the app opens, you'll see all of the watch faces you have currently configured:



Long-press any of the watch faces. You'll see a sheet appear which will let you share the face:



From the Apple Watch

It's even easier to share from the Apple Watch. From the home screen, long-press the watch face and choose the share icon. Then select to whom you'd like to send the watch face:

From the web

If you want to share multiple watch faces, you may wish to host them on your app's website. Point your favorite browser to Apple's Human Interface Guide (HIG) (https://apple.co/3s33S5P). Scroll down to the **Technologies** section, and you'll see a download available for **Add Apple Watch Face**. Download the file, which is a DMG, and then double-click it. You'll see a window containing the developer license as well as an **Assets** folder.

The Assets folder contains multiple images you can place on your webpage to let customers know they can download a watch face. Each style of button contains images that work well with both light and dark mode websites. Apple has also provided the images in **PDF**, **PNG** and **SVG** formats.

Once you create a watch face on your physical device that looks the way you want with the appropriately placed complications, share the face with yourself by choosing the **Mail** app. Unlike the other sharing types, if you choose Mail, you'll get both the watch face file and a graphic showing how the face looks.

> **Note**: The preview is only included in the email if you share the face from the Watch app on the iPhone, not directly from the Apple Watch.

Now, you simply create a standard webpage that showcases your awesome watch face with a few simple steps:

1. Choose the **SVG** graphic you wish to use for the download button.
2. Save both files that you mailed to yourself when sharing the face.
3. Create a page that displays the watch face and has accompanying download buttons.

You can quickly create a page like this:

Using your favorite text editor, create a file called **index.html** somewhere on your Mac. It shouldn't be part of the Xcode project.

```
<html>
<head>
<link rel="stylesheet" href="style.css">
</head>
<body>
<h1>My Cool App</h1>
<p>We really think you'll love this watch face!</p>

<hr />
<div>
  <img src="preview.png">
  <br/>
  <br/>

  <a href="FaceToShare.watchface"
     type="application/vnd.apple.watchface">
    <img src="download.svg">
  </a>
</div>

<div>
  <p>
    You can scan the below QR code to install the watch face
from your
    paired iOS device.
  </p>
  <img alt="QR Code" class="qrcode" src="qrcode.png">
</div>
</body>
</html>
```

After saving the file, double-click on it in **Finder** to launch your web browser.

Obviously, that HTML shows how quickly you can make your watch faces available to your customers. A production website would, of course, look much nicer and include all appropriate HTML tags.

Notice, on the anchor tag, the `<a>` element, the referenced location is the watchface file. Then inside the anchor, the image points to the SVG graphic you chose from the HIG. That's one example of making a clickable "button" to download the watch face. Also, notice that the anchor tag's `type` attribute is `application/vnd.apple.watchface`.

While the download button is great, you can't assume that your customer has arrived on your website while browsing on their phone. If they're surfing the web on their laptop, and decide they want your watch face, then providing a QR code that points to the downloadable watch face makes their life easier. They just pull out their phone, point at the screen and download your watch face.

Take a look at Watchfacely (https://www.watchfacely.com) for great examples of webpages providing shared watch faces. https://bit.ly/3s1zWXY is a specific example showing a shared health management face.

From the iOS app

You have one final option for providing a preconfigured watch face. You can embed the face right into your iOS app. The advantage of this approach is that the person using your app doesn't have to transition to a website. You simply provide them some mechanism to send the face directly to the paired Apple Watch via one of the HIG buttons you downloaded.

Open **FaceSharing.xcodeproj** from this chapter's starter materials. You'll notice the project already includes a few items for you:

- The **FaceToShare.watchface** file.
- The appropriate HIG button to share the face is in **Assets.xcassets**.
- The face's preview image is in **Assets.xcassets**.
- **Session.swift** with the standard boilerplate.

> **Note**: Be sure you include any **.watchface** file you add to Xcode in the **FaceSharing** target membership. If you use Xcode to add the file, that's automatic. If you drag the file in from Finder, then it isn't.

Unlike other chapters in this book, there's no watchOS target. iOS handles sending a watch face and only requires an active and paired Apple Watch.

Setting up the session

In the Watch Connectivity chapter, you learned about `WCSession` and `WCSessionDelegate`. To know whether an Apple Watch is paired to the iOS device, you'll need the same type of session running. If there's not a paired Apple Watch, then you wouldn't want to provide the option to install a custom watch face.

Open **Session.swift**. Add a property to `Session` to track whether or not you should show a share button, as well as the method which will update the value:

```
// 1
@MainActor
@Published
var showFaceSharing = false

private func updateFaceSharing(_ session: WCSession) {
  // 2
  let activated = session.activationState == .activated
  let paired = session.isPaired

  // 3
  DispatchQueue.main.async {
    self.showFaceSharing = activated && paired
  }
}
```

In the preceding code:

1. You use `@Published` to notify SwiftUI when changes to the property occur. Since the property intends to determine if you should display a UI element, you'll also want to use `@MainActor` so that Xcode helps ensure you only update the property from the main thread.
2. Using the supplied `WCSession`, you determine whether the session is active on a paired Apple Watch. No need to check if the app is installed.
3. Finally, you set `showFaceSharing`, ensuring you first dispatch to the main queue since you're triggering UI updates.

Next, in `sessionDidBecomeInactive(_:)` and `session(_:activationDidCompleteWith:error:)`, you call the method you just wrote. So add this line to both methods:

```
updateFaceSharing(session)
```

Add one more delegate method:

```
func sessionWatchStateDidChange(_ session: WCSession) {
  updateFaceSharing(session)
}
```

This way, you've ensured `showFaceSharing` always knows whether or not there's an active and paired Apple Watch.

Showing the share button

Now open **ContentView.swift**. Currently, you only show an image of the face which you'll send to the watch. You need to include a download button, but only if there's a paired Apple Watch.

Start by adding the following code, right after the `Image` line:

```
// 1
if session.showFaceSharing {
} else {
  // 2
  Text("Unable to share watch face")
    .font(.title)
  // 3
  Text("Please pair an Apple Watch first")
    .font(.title3)
}
```

Here's a code breakdown:

1. By looking at the published property you just configured in **Session.swift**, SwiftUI can determine what to display.
2. If face sharing isn't available, you tell the user they can't share the face in a `.title` font.
3. It's also a good idea to identify *why* they can't share the face by telling them there's no paired Apple Watch in a slightly smaller `.title3` font.

Remember that SwiftUI reevaluates the `body` whenever a property marked as `@Published` changes. If face sharing *is* available, you need to call a method that transfers the face.

Add the following method:

```
// 1
func sendWatchFace() async {
  // 2
  guard let url = Bundle.main.url(
    forResource: "FaceToShare",
    withExtension: "watchface"
  ) else {
    // 3
    fatalError("You didn't include the watchface file in the
bundle.")
  }
}
```

Here's what's happening in that code:

1. You defined a method to transfer the face. Because it might take a bit of time, you mark the method as `async`.
2. You load the watchface file that you included in the iOS target.
3. While a `fatalError` call is usually bad, it's appropriate in this situation as you want to catch the missing bundle file during development rather than once you've published the app.

The library you'll use to actually perform the transfer comes from **ClockKit**. So, add the appropriate import to the top of the file:

```
import ClockKit
```

Finally, you'll use the `CLKWatchFaceLibrary` to perform the transfer. Add the following to the end of `sendWatchFace`:

```
// 1
let library = CLKWatchFaceLibrary()

// 2
do {
  // 3
  try await library.addWatchFace(at: url)
} catch {
  // 4
  print(error.localizedDescription)
}
```

In the preceding code:

1. As mentioned, you create an object of type `CLKWatchFaceLibrary`.
2. The transfer might fail, so you need to wrap the call in a `do/catch` block.
3. You `try` to perform `addWatchFace(at:)` by passing the URL of the face you loaded from the `Bundle`. As the method is asynchronous, you add the `await` keyword after `try`.
4. If the face can't be transferred, you'll need to handle the error. In a production app, that would likely mean displaying some type of alert.

> **Note**: Remember, it's always `try await`, never `await try`. You try to wait for the call to complete. You don't wait for a method to try and run.

According to Apple's documentation on `addWatchFace(at:)`:

> *All of the complications on the watch face must come from apps with a valid App Store ID, such as an app from the App Store or a TestFlight build. If you try to use complications from a development build, the system won't recognize the development ID as a valid App Store ID.*

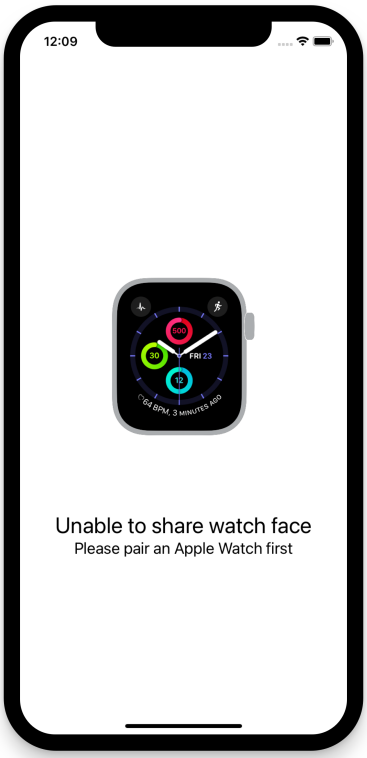
Now it's time to use your new method. Go back to the `body` definition and inside of the `if` condition, add:

```
Button {
  Task { await sendWatchFace() }
} label: {
  Image("ShareButton")
}
```

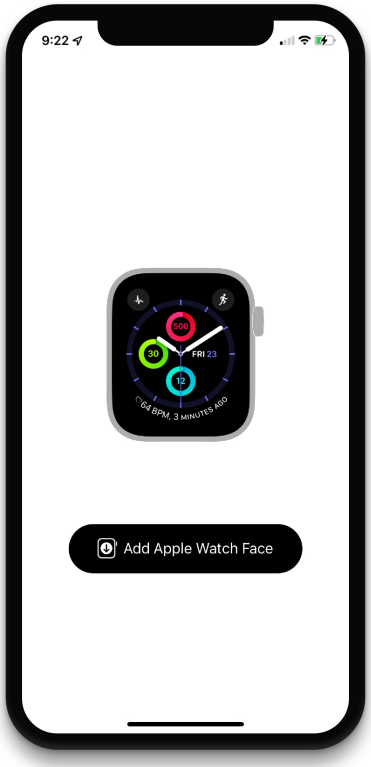
That's pretty standard SwiftUI for adding a button with one exception. When the user clicks the button, you create a new top-level `Task`. The method `sendWatchFace` is asynchronous, but the button click's action isn't aware of the asynchronous method.

By wrapping the call in the `Task` block, you allow the method call to still properly run asynchronously. This is a common pattern you'll use when calling async methods from SwiftUI bodies.

At this point, you're ready to test things out. Build and run the app in the Simulator, not on your physical device. You'll see the message saying you can't share the watch face. The Simulator isn't configured with a paired watch face.



Build and run the app again, this time targeting your iPhone. This time you can share the watch face. Of course, you'll have to have a paired Apple Watch handy! :]

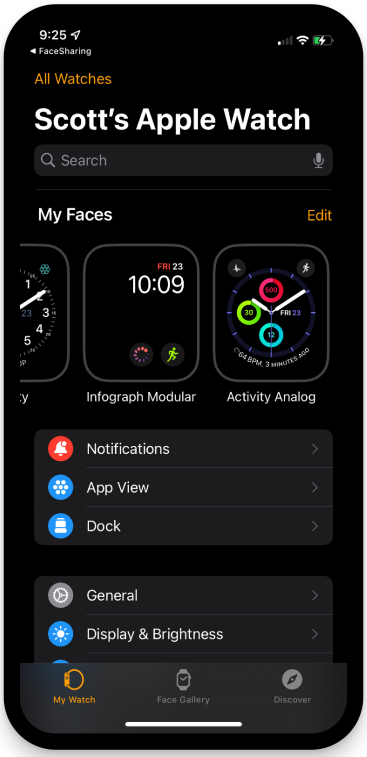


Tap the **add button**, and you'll be asked to add the face to your Apple Watch:

While it seems redundant since you just tapped the download button, Apple wants to make sure faces don't magically appear on your devices without you being aware.

> **Note**: Never install a face to the watch without the user explicitly requesting it via a download button.

After you tap **Add to My Faces**, go to the Watch app on your iPhone, and the new watch face should now be visible:



Finally, for the *pièce de résistance*. Set the newly installed watch face and then glance at your Apple Watch. You'll see your shiny new watch face!

Handling shared complications

When the Apple Watch receives a shared complication, ClockKit will call `handleSharedComplicationDescriptors(_:)` on your watchOS extension's `CLKComplicationDataSource`. Depending on how your app structured its complications, you may or may not need to implement that delegate method.

The method receives an array of `CLKComplicationDescriptor` objects for each complication in the shared watch face. If your complication includes specific user data via the descriptor's `userInfo` object, you'll need to handle that data.

Consider an automobile app where you have different complications available per truck type. You would then want to implement the following method:

```
func handleSharedComplicationDescriptors(
  _ complicationDescriptors: [CLKComplicationDescriptor]
) {
  for descriptor in complicationDescriptors {
    guard
      let userInfo = descriptor.userInfo,
      let type = userInfo[truckTypeKey] as? String
    else {
      return
    }

    // Add the new truck type
  }
}
```

If the app already contains the relevant details for the truck type, then there's nothing to do. If, however, that truck type doesn't yet exist in the app, you may have to perform setup to support that new type, such as downloading details from the network.

Key points

- Only install a watch face if the user requests it.
- Don't offer to install a watch face if there is no paired device.
- Anyone can create shareable watch faces, but only you can offer them from inside the app.
- Remember to handle complication-specific data.

Where to go from here?

- The sample materials for this chapter contain a **web** folder that includes the full source to the sample webpage.
- Check out the amazing Watchfacely (https://www.watchfacely.com) site for great examples of sharing faces via the web.
- Many sites on the web will generate QR codes for free, such as QR Code Generator (https://www.qr-code-generator.com).

Chapter 14: Sign in With Apple

By Scott Grosch

Sign in with Apple, or **SIWA**, has been around for a few years. Your customers will greatly appreciate the simplicity of registration and authentication when using a device with a screen as small as the Apple Watch.

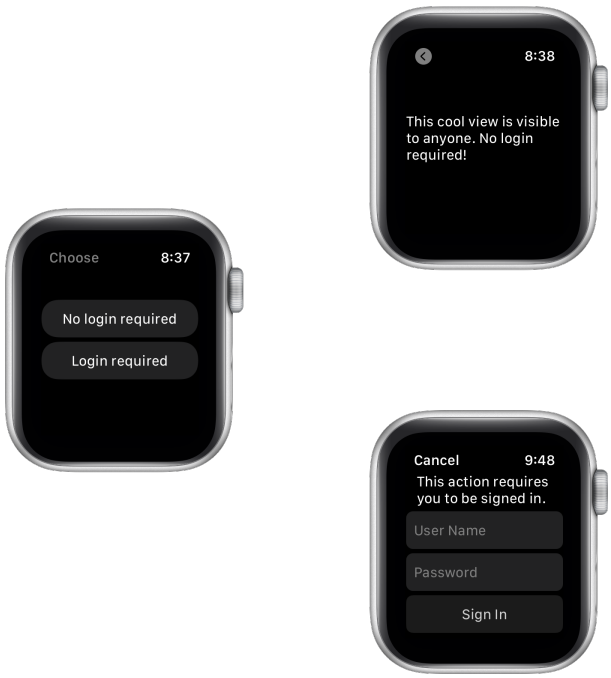
You should consider this a bonus chapter, as there's nothing Apple Watch-specific related to SIWA. If you're already familiar with how to implement SIWA, feel free to skip this chapter. Due to the extra importance of a streamlined experience on the Apple Watch, I felt it made sense to include this content in the book.

> **Note**: To complete this chapter, you'll need a paid Apple developer account to set up the required identifiers and profiles.
>
> This chapter only deals with the watchOS parts of SIWA. Embedded Sign in with Apple JS, web server and developer portal configurations are outside the scope of this book.

Some vendors won't implement SIWA due to the anonymization of email addresses. When a customer emails the vendor with a support ticket, it's harder to locate the customer's account. However, you should keep in mind that your *user's* experience is what you should prioritize, not yours.

To authenticate or not

Build and run the **SignIn** app from this chapter's starter materials. You'll see a simple screen that lets you choose which of two views to display. Tap the top button, and you're taken to the part of the app which doesn't require authentication. However, tap the second button, and you're asked to authenticate:



Authenticate via username and password

The authentication screen you currently see is what most apps present: a simple username and password entry view.

Take a look at **Authenticating/PasswordView.swift**. For the Apple Watch, it's especially important to specify the text content type:

```
TextField("User Name", text: $userName)
  .textContentType(.username)

SecureField("Password", text: $password)
  .textContentType(.password)
```

Specifying the content type gives watchOS clues about how to present and handle the data. A `.username` or `.password` field can be appropriately linked to the keychain, for example.

Next, notice `isButtonDisabled` computed property:

```
private var isButtonDisabled: Bool {
  return userName.trimmingCharacters(in: .whitespaces).isEmpty
    || password.isEmpty
}
```

As well as its use on the button:

```
Button("Sign In", action: signInTapped)
  .disabled(isButtonDisabled)
```

There's no point in letting the user tap the **Sign In** button if they haven't properly supplied all required fields.

> **Note**: Instead of disabling the button, you could also present an alert sheet when the user provides invalid data.

Finally, notice how self-contained the view is. It performs a single task, which is to gather the input. When your user taps the Sign In button, *something* happens, but this view doesn't care what. It simply delegates the work elsewhere by passing the details to the `completionHandler`.

Handling sign in

Now switch to **Authenticating/SignInView.swift**. The `SignInView` is where you handle the *how* of your app's login. The first piece you'll notice is the use of app storage:

```
@AppStorage("userName") private var storedUserName = ""
```

Tracking the username isn't a detail of `PasswordView` because it shouldn't care where the username comes from. While app storage will be a common location, you might just as easily store the user's information in Core Data.

The completion handler for the `PasswordView` performs your actual authentication.

```
private func userPasswordCompletion(userName: String, password:
String) {
  // 1
  storedUserName = userName
  // 2
  var request = URLRequest(
    url: URL(string: "https://your.site.com/login")!
  )
  request.httpMethod = "POST"
  // request.httpBody = ...

  DispatchQueue.main.async {
    // 3
    onSignedIn("some token here")
    // 4
    dismiss()
  }
}
```

Here, the code:

1. Updates the username you're storing with whatever value the user supplied. If they change usernames, they likely want to use the same one next time.
2. Generates your `URLRequest` as appropriate to send the username and password to your web service.
3. Once the authentication successfully returns, it calls the completion handler supplied to `SignInView`. The example above assumes your web service returns some type of token to use for future network calls.
4. Normally, this view is pushed onto a stack or displayed in a modal dialog. If authentication was successful, it dismisses the view.

Your app may send a `Decodable` object back, it might supply values in a header or any other combination of responses. You'll need to update the code as appropriate. Keep three key details in mind.

- Call the completion handler with the appropriate response data.
- Dismiss the view if authentication was successful.
- Ensure you're working against the main thread when calling your completion handler, dismissing the view and presenting any errors.

Signing in with Apple

At this point, your login screen is fully functional. Functional, but not friendly. I don't know about you, but I always struggle to draw letters exactly as the Apple Watch wants them. Wouldn't it be better to have a Sign in with Apple button available?

Time to finally do some coding!

Adding the SIWA button

While still editing **SignInView.swift**, add an import to the top of the file:

```
import AuthenticationServices
```

`AuthenticationServices` provides the code necessary for SIWA.

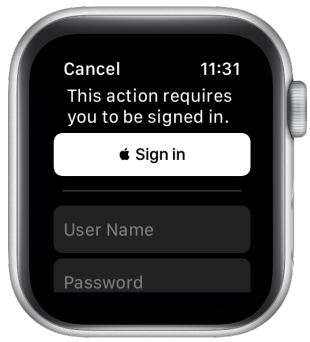
Now add two stub methods to the `SignInView` in this file:

```
private func onRequest(request: ASAuthorizationAppleIDRequest) {
}

private func siwaCompletion(result: Result<ASAuthorization,
Error>) {
}
```

You'll populate both methods in a moment, but they're required for the button to be used inside `body`. Now place the following code between the `Text` and `PasswordView` elements in the `body`:

```
SignInWithAppleButton(
  onRequest: onRequest,
  onCompletion: siwaCompletion
)
  .signInWithAppleButtonStyle(.white)

Divider()
  .padding()
```

Take a look in the Canvas, inside of Xcode, and you'll now see your shiny new SIWA button:



That's all it takes to place the button. Now that SIWA has full SwiftUI support, you no longer need to implement a `UIViewRepresentable` yourself. Nice.

Configuring the request

You'll configure the request in the `onRequest(request:)` stub you generated. Add the following code there:

```
request.requestedScopes = [.fullName]
request.state = "some state string"
```

You told Apple that you need to know the **full name** of the person who is authenticating. `requestedScopes` allows for a `.email` option as well, but you should *only* request it if you need it. Keep in mind that most users will hide their email from you and use Apple's relay service. Apple will provide you a unique identifier to represent the user in your database, so don't ask for the email just so you have a unique "key" to represent the user.

`state` returns to you unmodified in the credential Apple supplies. You can provide whatever user-defined state you wish as it's simply echoed back to you in Apple's response.

Optionally, you may use `nonce` to mitigate replay attacks. A replay attack occurs when a malicious user copies and resends a data packet to you. By using a unique `nonce` value in every request, you ensure that Apple sent the packet you received. At the top of `SignInView`, add a private property:

```
private let nonce = UUID().uuidString
```

Then, inside `onRequest(request:)` set the request's nonce:

```
request.nonce = nonce
```

> **Note**: Decoding and validating the nonce is outside the scope of this chapter.

Handling the reply

When you get a reply back from Apple, there are three possible outcomes:

1. The authentication failed.
2. This is the first time successfully authenticating against your app.
3. You successfully authenticated, and it's not the first time.

To handle the first case, add the following code to `siwaCompletion(result:)`:

```
guard
  // 1
  case .success(let authorization) = result,
  // 2
  let credential = authorization.credential as?
ASAuthorizationAppleIDCredential
else {
  // 3
  if case .failure(let error) = result {
    print("Failed to authenticate: \
(error.localizedDescription)")
  }

  return
}
```

In the preceding code:

1. First, you store the `authorization` returned from Apple on success.
2. Next, you retrieve the `ASAuthorizationAppleIDCredential`.
3. If either of the first two steps fails and an error returns, display it. Your production code should, of course, display a message to the user as opposed to just printing to the console.

If a failure didn't occur, then you handle either login or registration. If `fullName`, which you requested in `onRequest(request:)` via `requestedScopes`, contains a value, then this is the first time you've authenticated this user.

If `fullName` is `nil`, then you've authenticated an existing user. It's unfortunate, but Apple will only return the user's full name and email address the first time. It's your responsibility to save that data in an app-appropriate way.

In your method, you'll simply check the property's value. Add the following code after the `guard` block you just added:

```
if credential.fullName == nil {
  // You've logged in an existing user account.
} else {
  // The user does not exist, so register a new account.
}
```

Depending on your app's configuration, you'll then need to take appropriate action. Usually, that means making a call to your web server to generate a new account or sign in to an existing account. In either case, you'll likely want to pass the `credential.user`, `credential.identityToken` and `credential.authorizationCode` to your server.

> **Note**: Remember to call `onSignedIn()` after contacting your web server.

There's a critical component of new user registration to keep in mind. Apple will only provide the name and email address the first time you authenticate. If the registration call you make to your web server fails for any reason, you have to have a way to try again. Consider a scenario where your web server crashes or the Apple Watch loses network connectivity.

You should use some form of secure storage to store the data you need for registration until your server successfully registers the user. You can use the KeyChain, Core Data or any other secure storage mechanism that doesn't require the network.

Storing the user credentials

The `user` property of the `credential` contains the unique identifier that Apple assigned to your user. You'll likely want to pass that identifier to all the calls you send to your web server. A simple way to handle that requirement is to add another `@AppStorage` to the top of `SignInView`:

```
@AppStorage("userCredential") private var userCredential = ""
```

And then, at the end of `siwaCompletion(result:)`, assign the value:

```
userCredential = credential.user
```

Storing their name properly

If you've requested the user's full name, keep in mind that the value provided is a `PersonNameComponents`, not a `String`. To display the name, use the appropriate formatter, like so:

```
PersonNameComponentsFormatter.localizedString(
  from: name,
  style: .default
)
```

Different cultures display names differently, so you should never assume it's safe to send something like this to your server:

```
let me = credential.fullName
let badNameExample = "\(me.givenName) \(me.familyName)"
```

You may be tempted to use the formatter and then send *that* value to the web server. Don't do that either! Users from one country may see names of users from another country, so you want to ensure each person sees the name formatted correctly for their locale.

`PersonNameComponents` is `Codable`, meaning it's incredibly simple to send to your web server:

```
let encoder = JSONEncoder()
guard let data = try? encoder.encode(credential.fullName) else {
  // handle error appropriately
  return
}

let encoded = data.base64EncodedString()
```

Send the `encoded` value to your web server for storage. When you retrieve the name from your web server, perform the operations in reverse:

```
let decoder = JSONDecoder()

guard
  let data = Data(base64Encoded: encoded),
  let components = try? decoder.decode(
    PersonNameComponents.self,
    from: data
  )
else {
  // handle error appropriately
  return
}

let name = PersonNameComponentsFormatter.localizedString(
  from: components,
  style: .short
)
```

Key points

- Adding SIWA is relatively easy, so use it wherever possible.
- Ensure that you store new user registration details local to the device until your app's web server performs a successful registration.
- Only request the user's name and email address if you truly need them.
- Keep in mind that most email addresses you receive will be an Apple relay address.
- Always store the full `PersonNameComponentsFormatter` details on your web server.

Where to go from here?

For full details on SIWA, please see Apple's document, *Sign in with Apple (*https://apple.co/3uWndWk*)*.

Vapor (https://vapor.codes) is a great server-side option when using Sign in with Apple. See our book, *Server-Side Swift with Vapor (*https://bit.ly/3pL3doK*)*, for complete implementation details.

Chapter 15: HealthKit

By Scott Grosch

Apple Watch is an incredible device for tracking health and fitness. The sheer number of apps related to workout tracking is staggering. Therefore, in this chapter, you'll build a workout tracking…

No! Why not do something a bit different? In **Chapter 7, "Lifecycle"**, you built an app to help kids know how long to brush their teeth. Did you know that Apple Health has a section for tracking that?

1. Open the Apple Health app *on your iPhone*.
2. Tap **Browse** in the toolbar.
3. Tap **Other Data** from the **Health Categories** list.
4. You'll see **Toothbrushing** in the **No Data Available** section.

Who knew? :] Might as well log the fact you brushed your teeth, right?

> **Note**: There's nothing different about using HealthKit on the Apple Watch. However, it's such a pervasive use case that I wanted to include a chapter on it.

> **Note**: While the simulator can read and write to Apple Health, you can't launch the Apple Health app yourself. You'll need a physical device to truly complete this chapter.

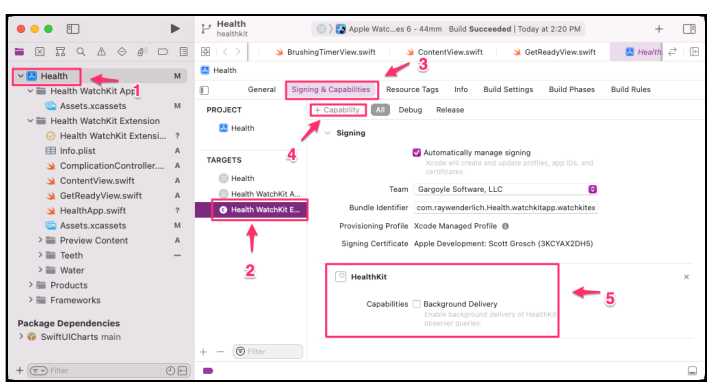
Adding HealthKit

The starter materials for this chapter contain a slightly modified version of the Toothbrush app you built previously. **HealthKit** is one of the frameworks that requires permission from the user before you use it since it contains personal information.

Signing & Capabilities

First, you need to add the HealthKit capability to Xcode:

1. Open **Health.xcodeproj** from the starter materials.
2. In the Project Navigator (**Command**-**1**) select the **extension** target.
3. Select **Signing & Capabilities**.
4. Click **+ Capability**.
5. In the dialog that appears, choose **HealthKit**.



Info.plist descriptions

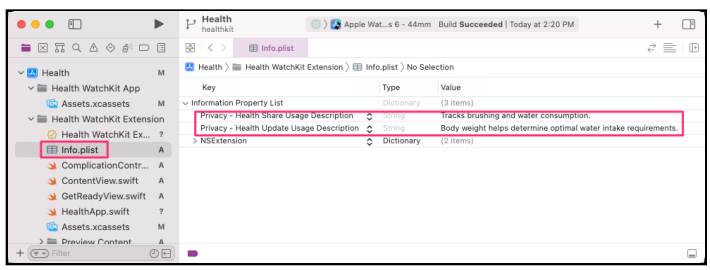
Once you've done that, you'll then need to open **Info.plist** from the **Health WatchKit Extension** and add two keys:

- Privacy - Health Share Usage Description
- Privacy - Health Update Usage Description

For the first key, set the value to the reason you're asking the user to let you write data to HealthKit. For example, you might put: **Tracks brushing and water consumption.**

> **Note**: Spoiler alert, you'll also add water tracking to the app.

For the second key, set the value to the reason why you need to read data from HealthKit. For example, you might put: **Bodyweight helps determine optimal water intake requirements.**



Creating the store

Select the **Health WatchKit Extension** folder in the Project Navigator.Create a new swift file and name it **HealthStore.swift** and add to it:

```
// 1
import Foundation
import HealthKit

final class HealthStore {
  // 2
  static let shared = HealthStore()

  // 3
  private var healthStore: HKHealthStore?
```

```
    // 4
    private init() {
      // 5
      guard HKHealthStore.isHealthDataAvailable() else {
        return
      }

      healthStore = HKHealthStore()
    }
  }
```

What's that code doing? Here's a breakdown:

1. HealthKit is a framework, so you need to import the package.
2. The `HealthStore` is a singleton.
3. You need a variable to hold the `HKHealthStore`.
4. A private initializer ensures that callers must use the `shared` singleton instance.
5. If the device can't use Apple Health, you quietly exit. If it can, you initialize the `HKHealthStore`.

Your wrapper class implements the singleton pattern because Apple specifies that you should only create a single instance of `HKHealthStore` for your app. If the device can't use HealthKit, then you shouldn't try to initialize the store.

To initialize `HealthStore`, add the following code to **HealthApp.swift** just after the `NavigationView`'s closing squiggly bracket:

```
.task {
  _ = HealthStore.shared
}
```

`.task` runs a single time when the app starts. Even though you threw away the return of the call to `HealthStore.shared`, the initializer ran.

> **Note**: There are multiple camps of thought on a class like `HealthStore`. Some people like the singleton pattern, while others feel the class should be an `ObservableObject` so you can place it into the environment. In the case of `HealthStore`, I chose the singleton pattern so it would be available outside of a SwiftUI `View`, if necessary.

Now that you have HealthKit configured, it's time to track some data.

Saving data

Saving data to Apple Health is an asynchronous operation. All data types convert to an `HKSample` before saving. To let the app continue to use the `async / await` pattern, add the following method to **HealthStore.swift**:

```
private func save(_ sample: HKSample) async throws {
  // 1
  guard let healthStore = healthStore else {
    throw HKError(.errorHealthDataUnavailable)
  }

  // 2
  let _: Bool = try await withCheckedThrowingContinuation {
    continuation in

    // 3
    healthStore.save(sample) { _, error in
      if let error = error {
        // 4
        continuation.resume(throwing: error)
        return
      }

      // 5
      continuation.resume(returning: true)
    }
  }
}
```

Wrapping a callback asynchronous method into the newer `await`/`async` syntax might be new to you. Here's what's happening:

1. If `healthStore` wasn't set, you shouldn't have called this method to begin with, so you throw an appropriate error.
2. `withCheckedThrowingContinuation` takes a body of code and pauses until the provided `CheckedContinuation<T, Error>` is called.
3. Then, you save the sample to Apple Health.
4. If the asynchronous save fails, you pass the error thrown to `resume(throwing:)`.
5. If the call succeeds, you have to return something. In this case, it's just a boolean true value.

The signature for step two is a bit ugly. `withCheckedThrowingContinuation` is a generic method. If you don't specify the return type as a `Bool`, Xcode can't determine the type of the generic, which results in an error. You don't care about the return value, so just assign to the _ placeholder.

Tracking brushing

Apple Health stores activities differently depending on the type of data. For example, when brushing your teeth, the Apple Health app tracks the start and end times.

Brushing HealthKit configuration

Add the following property to your `HealthStore` class so that Apple Health knows you're going to log data related to brushing your teeth:

```
private let brushingCategoryType = HKCategoryType.categoryType(
  forIdentifier: .toothbrushingEvent
)!
```

As this book targets watchOS 8, it's safe to force unwrap the category type. You'll use `brushingCategoryType` in multiple places, and there's no reason to have to unwrap the optional constantly.

You would only get a failure if you tried to create an identifier on an OS version where it didn't yet exist. The `toothbrushingEvent` has existed since watchOS 6. If you're targeting older releases, which don't support all of your identifiers, you'll need to make them optional and perform the appropriate safety checks in all relevant locations.

Now it's time to ask the user for permission to read and write data related to how often they brush their teeth. Add the following code to the end of the initializer:

```
Task {
  try await healthStore!.requestAuthorization(
    toShare: [brushingCategoryType],
    read: [brushingCategoryType]
  )
}
```

When your app calls `requestAuthorization(toShare:read:)`, the user will see a request to authorize your app to read and write the specified data. As long as you don't ask for types that you haven't previously, watchOS will only display the request a single time.

Wrapping the asynchronous call in a `Task` lets you use the call in a method not marked `async`.

> **Note**: At the time of writing, HealthKit doesn't include `async` methods.

Sharp-eyed readers will notice that `requestAuthorization(toShare:read:)` is a throwing method, yet you've not protected against the method throwing. A peculiar side effect of `Task` is that *it eats all exceptions*.

Build and run. Once it launches, watchOS will present screens to you requesting permission to read and write to Apple Health. Once you've approved both read and write access, relaunch the app. This time, there's no prompt as the call to `requestAuthorization(toShare:read:)` doesn't include any new types.

Open **HealthStore.swift** and add:

```
// 1
func logBrushing(startDate: Date) async throws {
  // 2
  let status = healthStore?.authorizationStatus(
    for: brushingCategoryType
  )

  guard status == .sharingAuthorized else {
    return
  }

  // 3
  let sample = HKCategorySample(
    type: brushingCategoryType,
    value: HKCategoryValue.notApplicable.rawValue,
    start: startDate,
    end: Date.now
  )

  // 4
  try await save(sample)
}
```

Here's what's happening:

1. You define a method that takes the `Date` when the user started brushing their teeth and note that it's both asynchronous and can throw.
2. At any point, your user can enable or disable access to Apple Health. Therefore, you need a way to know whether they're letting you write data for brushing their teeth.
3. You create a sample, which is what HealthKit stores, of type `brushingCategoryType`. Brushing doesn't have a *value*, so you pass the `HKCategoryValue.notApplicable.rawValue` enum value. Then you log the start and end time of the brushing.
4. Save the sample using your just written `save(_:)`.

> **Note**: You can't check for read access, only write. If you don't have read access, you just get no results back.

Based on your app's configuration, you need to decide whether to silently `return` in the `guard` clause of step two or throw some type of exception. In this app, it makes sense to let them brush their teeth and get the 30-second timer without writing to Apple Health.

Hopefully, you agree that by wrapping the default save with an `async / await` pattern, you've made the rest of your code cleaner.

All that's left is to update the session end to call `logBrushing(startDate:)`.

Open **BrushingModel.swift** inside the **Teeth** folder. You'll recognize this file from Chapter 7: "Lifecycle". Near the bottom of `extendedRuntimeSessionDidStart(_:)`, you'll find the line of code that invalidates the session:

```
self.session.invalidate()
```

When the session completes successfully, you need to write to Apple Health. Replace that line of code with:

```
// 1
Task {
  // 2
  try? await HealthStore.shared.logBrushing(
    startDate: self.started
  )
```

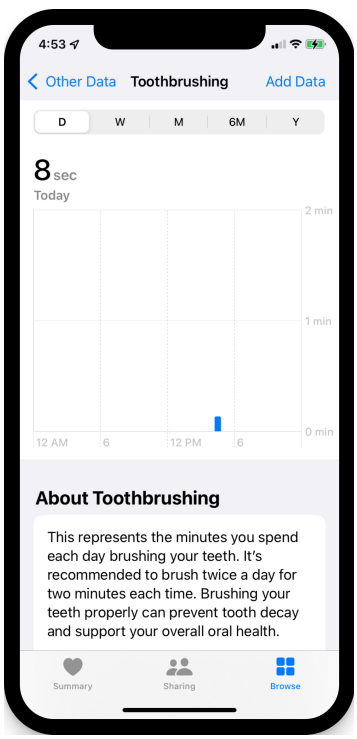
```
    // 3
    self.session.invalidate()
  }
```

The pattern is probably starting to look familiar:

1. You're in a method that is *not* `async`, but you want to call an `async` method, so you pass the call to a `Task`.
2. Log the fact that the user finished brushing their teeth. If it fails, you probably want to ignore it as there's nothing for the user to do.
3. Mark the session as complete once the update to Apple Health finishes.

It's time to build and run. Before doing so, you may wish to change the value of `secondsPerRound` in **Teeth/BrushingModel.swift** to something smaller, like 2.0, so that you don't have to wait two minutes to see results.

Once you've built and run the app, tap the brush button and watch the magic happen. When the timer runs out, switch over to the Apple Health app. Now repeat the steps from the start of this chapter to look at the toothbrushing category, and you'll see an entry:



Tracking water

Like brushing their teeth, many kids have a hard time remembering to drink water. Seems like a great addition to your app!

While brushing teeth logged time rather than a value, water intake is the opposite. You're likely less concerned with *when* they drank water and more interested in *how much* water they drank each day.

There's quite a bit of work to do to support water intake. This may be a good time to take a quick break and grab a snack!

Updating permissions

Recommended water intake is based on body mass. So, you'll have to ask for two more types of data from HealthKit. In **HealthStore.swift**, add:

```
private let waterQuantityType = HKQuantityType.quantityType(
  forIdentifier: .dietaryWater
)!

private let bodyMassType = HKQuantityType.quantityType(
  forIdentifier: .bodyMass
)!
```

Notice that you're now using quantity types, not category types.

Next, update the call to `requestAuthorization(toShare:read:)` so that it includes the new identifiers:

```
Task {
  try await healthStore!.requestAuthorization(
    toShare: [brushingCategoryType, waterQuantityType],
    read: [brushingCategoryType, waterQuantityType,
bodyMassType]
  )
}
```

Pay attention to the fact that you would like to read the person's weight, or body mass, but you have no reason to write their weight. **Don't request permissions that you don't need**.

Build and run now, and you'll notice that Apple Health again asks for permissions. The dialog appeared again because you added new items to the list of authorizations you're requesting. You'll also notice that nothing related to brushing teeth is shown in the request, as you've already answered those questions.

Water HealthKit configuration

Time to update the `HealthStore` to handle water. Add the following to **HealthStore.swift**:

```
// 1
var isWaterEnabled: Bool {
  let status = healthStore?.authorizationStatus(
    for: waterQuantityType
  )

  return status == .sharingAuthorized
}

// 2
func logWater(quantity: HKQuantity) async throws {
  guard isWaterEnabled else {
    return
  }

  // 3
  let sample = HKQuantitySample(
    type: waterQuantityType,
    quantity: quantity,
    start: Date.now,
    end: Date.now
  )

  // 4
  try await save(sample)
}
```

In the preceding code, you:

1. Need to know whether the user is *currently* allowing you to write water-related data.
2. Implement an `async` method to store water consumption as you did for brushing your teeth.
3. Generate a sample to save. Notice how you specify an actual *quantity* this time, and the start and end dates are the current time.
4. Save the data, if you can.

Log water button

To keep the app simple, you'll provide two buttons to let the user enter an amount of water. Inside the **Water** folder, you'll find a file named **LogWaterButton.swift**. I've cheated a bit in the interest of simplicity and hardcoded two water sizes based on whether you're using the metric system.

When the user taps the button, you need to generate the appropriately sized `HKQuantity` to tell Apple Health how much water the person drank. Edit `tapped()` and paste:

```
// 1
let unit: HKUnit
let value: Double

if Locale.current.usesMetricSystem {
  // 2
  unit = .literUnit(with: .milli)
  value = size == .small ? 250 : 500
} else {
  // 3
  unit = .fluidOunceUS()
  value = size == .small ? 8 : 16
}

// 4
let quantity = HKQuantity(unit: unit, doubleValue: value)

// 5
onTap(quantity)
```

Here's what the code is doing:

1. HealthKit uses an `HKUnit` to identify the unit type for the value you'll store.
2. If the user's device uses the metric system, you create a value in milliliters.
3. If the user's device isn't using metric, they're clearly from the United States and thus want fluid ounces. Yes, that's sarcasm. I told you we were cheating here.
4. Using `unit` and `value`, you create an `HKQuantity`, which you can later convert to an `HKSample` for saving.
5. You pass that quantity to the completion handler.

Water view

Now create another SwiftUI view called **WaterView.swift** to act as the UI when taking a drink. Be sure to import HealthKit at the top of the file:

```
import HealthKit
```

In the `body`, replace the default "Hello, World!" text with the following code. Don't worry about the closure argument list error you see – you'll fix it in a moment:

```
// 1
ScrollView {
  VStack {
    // 2
    if HealthStore.shared.isWaterEnabled {
      Text("Add water")
        .font(.headline)

      HStack {
        LogWaterButton(size: .small) {  }
        LogWaterButton(size: .large) {  }
      }
      .padding(.bottom)
    } else {
      // 3
      Text("Please enable water tracking in Apple Health.")
    }
  }
}
```

The view right now is pretty simple:

1. You have a vertically scrolling view to contain everything. You don't need the `ScrollView` right now, but you will soon.
2. If the user lets you write to Apple Health, display some text and the two buttons to add the amounts of water consumed.
3. If they don't grant permissions, display a message.

While you shouldn't even present this view to the user if they haven't granted write access, it's always a good idea to protect as many areas of the app as possible. You may refactor other views and inadvertently remove a critical permission check.

When the user taps a button, you need to take action. So, implement:

```
private func logWater(quantity: HKQuantity) {
  Task {
    try await HealthStore.shared.logWater(quantity: quantity)
  }
}
```

The helper wraps the asynchronous call in `Task` since the SwiftUI button doesn't know how to call an `async`. In a production app, you'll likely want to set a `@State` that would result in the user seeing an alert on failure. Now that you have the helper method, call it on button taps:

```
LogWaterButton(size: .small) { logWater(quantity: $0) }
LogWaterButton(size: .large) { logWater(quantity: $0) }
```

Updating ContentView

Go back to the main extension folder. In **ContentView.swift**, import HealthKit at the top of the file:

```
import HealthKit
```

Create another state property:

```
@State private var wantsToDrink = false
```

And then, right after the existing `Button`, add another one:

```
if HealthStore.shared.isWaterEnabled {
  Button {
    wantsToDrink.toggle()
  } label: {
    Image(systemName: "drop.fill")
      .foregroundColor(.blue)
  }
}
```

If, and only if, the user allows your app to write water consumption to Apple Health, you'll show a button that toggles the state variable.

Right after the sheet which displays `BrushingTimerView()`, add one to display the `WaterView()`:

```
.sheet(isPresented: $wantsToDrink) {
  WaterView()
}
```

Build and run. HealthKit will ask you for permission to read and write water consumption data. Once you approve, the dialog goes away and… nothing looks different!

Where's your water button? Take a moment to think through what's happening to see if you can figure it out.

Getting the view to update

Remember that in SwiftUI, the body only updates if something being observed, like a `@State` property, changes. SwiftUI doesn't know when the value for `HealthStore.shared.isWaterEnabled` changes. You need to explicitly tell the view that a change has happened.

Start by adding another `let` in **Notification.Name+Extension.swift**:

```
static let healthStoreLoaded = Notification.Name(
  rawValue: UUID().uuidString
)
```

Then, in **HealthStore.swift**, at the end of the `Task` in the initializer, post that notification:

```
await MainActor.run {
  NotificationCenter.default.post(
    name: .healthStoreLoaded,
    object: nil
  )
}
```

Once HealthKit finishes asking for any required permissions, you let the rest of the app know via the notification. Remember that you're no longer on the main thread because you're running inside of a `Task`. Since you intend for the notification you're posting to trigger a UI update, you dispatch the post to the `MainActor`, which means the thread running the UI.

Change back to **ContentView.swift** and add two new properties:

```
// 1
@State private var waitingForHealthKit = true

// 2
private let healthStoreLoaded =
NotificationCenter.default.publisher(
  for: .healthStoreLoaded
)
```

Here's what those properties are for:

1. Using a `@State` variable to track whether HealthKit has completed checking permissions means the body will see the update.
2. A publisher is the SwiftUI way of identifying a notification that you'll listen for.

Next, wrap the code inside the body's `VStack` with a check:

```
if waitingForHealthKit {
  Text("Waiting for HealthKit prompt.")
} else {
  // Existing code from VStack
}
```

The user should never see the message as HealthKit will pop up the permission check too quickly. However, having the body look at `@State` means the view will refresh when it changes.

Finally, right after the existing `.onReceive(stopBrusing)`, add one more:

```
.onReceive(healthStoreLoaded) { _ in
  self.waitingForHealthKit = false
}
```

Once the notification is received, you update the property, which causes the view to refresh. Reset the simulator by going to its menu and selecting **Device ▸ Erase All Content and Settings...** so that HealthKit has to ask for permissions again.

Build and run, this time tapping the water icon and then one of the two buttons. Once you've done so, switch to the Apple Health app on your iPhone and check out the water category in the **Nutrition** section. Congratulations, you drank some water! :]

Writing the data is great, but what about viewing it? You need to present the data to your users, especially on the Apple Watch, where the Apple Health app is mysteriously missing.

Reading single day data

Common practice bases the amount of water you should drink on how much you weigh. It would be great to tell the user how much water they still need to drink today.

Querying HealthKit

In **HealthStore.swift**, add a method to determine the user's current body mass:

```
// 1
private func currentBodyMass() async throws -> Double? {
  // 2
  guard let healthStore = healthStore else {
    throw HKError(.errorHealthDataUnavailable)
  }

  // 3
  let sort = NSSortDescriptor(
    key: HKSampleSortIdentifierStartDate,
    ascending: false
  )

  // 4
  return try await withCheckedThrowingContinuation
{ continuation in
    // 5
    let query = HKSampleQuery(
      sampleType: bodyMassType,
      predicate: nil,
      limit: 1,
      sortDescriptors: [sort]
    ) { _, samples, _ in
      // 6
      guard let latest = samples?.first as? HKQuantitySample
else {
        continuation.resume(returning: nil)
        return
      }

      // 7
      let pounds = latest.quantity.doubleValue(for: .pound())
      continuation.resume(returning: pounds)
```

```
      }

      // 8
      healthStore.execute(query)
    }
  }
```

There's a lot going on in that code. Here's a breakdown:

1. You create an `async` wrapper to get the user's body mass.
2. If the store isn't assigned, then you throw an error.
3. You only want the current weight, so you sort the results by the start date in descending order.
4. As before, you use `withCheckedThrowingContinuation` to wrap a completion handler method as an `async` type method.
5. When reading data, you use `HKSampleQuery`. You ask for a single entry with no date limit. Use whatever they recorded as their most recent weight.
6. If there's not a sample to read, or you don't have read access, a `nil` value returns.
7. Otherwise, you determine how many pounds the person weighs and return that value.
8. Finally, you ask HealthKit to execute the query.

Hardcoding pounds here seems wrong, doesn't it? What about kilograms or stones? In the United States, the general theory is you divide your body weight in half and then drink that many ounces of water a day. Countries using the metric system would multiply their weight in kilograms by 0.033 to accomplish the same thing.

Coding every type of calculation would be a difficult task. Instead, take advantage of Apple providing the tools you need to convert between different measurements for you. By *internally* using pounds and ounces, you can perform a single known calculation and then display it to the user in their preferred measurement system.

Now that you potentially know how much they weigh, it's time to determine how much they've already drunk today. In **HealthStore.swift**, add:

```
private func drankToday() async throws -> (
  ounces: Double,
  amount: Measurement<UnitVolume>
) {

  guard let healthStore = healthStore else {
    throw HKError(.errorHealthDataUnavailable)
  }

  // 1
  let start = Calendar.current.startOfDay(for: Date.now)

  let predicate = HKQuery.predicateForSamples(
    withStart: start,
    end: Date.now,
    options: .strictStartDate
  )

  // 2
  return await withCheckedContinuation { continuation in
    // 3
    let query = HKStatisticsQuery(
      quantityType: waterQuantityType,
      quantitySamplePredicate: predicate,
      options: .cumulativeSum
    ) { _, statistics, _ in
      // 4
      guard let quantity = statistics?.sumQuantity() else {
        continuation.resume(
          returning: (0, .init(value: 0, unit: .liters))
        )
        return
      }

      // 5
      let ounces = quantity.doubleValue(for: .fluidOunceUS())
      let liters = quantity.doubleValue(for: .liter())

      // 6
      continuation.resume(
        returning: (ounces, .init(value: liters, unit: .liters))
      )
    }

    // 7
    healthStore.execute(query)
  }
}
```

That method is a bit more complicated:

1. You want to determine all water consumed between the start of the day and now.
2. Once again, you wrap a completion handler method to use it `async` instead. Notice this time, nothing inside the block will throw an error, so you use `withCheckedContinuation` instead of `withCheckedThrowingContinuation`.
3. Using an `HKStatisticsQuery` lets you ask HealthKit to add all the values via the `.cumulativeSum`, so you get a single result.
4. If you don't have read permission or data, you state that the user drank 0 liters.
5. Determine both the number of US fluid ounces and the number of liters the user drank.
6. Return both the number of ounces the user drank as well as the liters.
7. Execute the query.

You provide the numerical value of ounces so you can perform math on it later. Using `Measurement<UnitVolume>` makes it easier for consumers to display the data in whatever format they need. Liters tend to be the "standard" value to use by default.

Finally, write the public method to determine what to display to the user:

```
func currentWaterStatus() async throws -> (
  Measurement<UnitVolume>, Double?
) {
  // 1
  let (ounces, measurement) = try await drankToday()

  // 2
  guard let mass = try? await currentBodyMass() else {
    return (measurement, nil)
  }

  // 3
  let goal = mass / 2.0
  let percentComplete = ounces / goal

  // 4
  return (measurement, percentComplete)
}
```

In the preceding code, you:

1. First, determine how much water the person drank today.
2. If you can't determine how much they weigh, you send back the water measurement with no percentage.
3. Then, perform simple math to determine how far they are towards the daily goal.
4. Return the water measurement and the percentage complete.

Notice that a single calculation is performed internally against US conversions, but what goes outside to the user is a `Measurement<UnitVolume>` using appropriate types.

Updating WaterView

Add two properties to **WaterView.swift**:

```
@State private var consumed = ""
@State private var percent = ""
```

Then write the method which will populate those properties:

```
private func updateStatus() async {
  // 1
  guard
    let (measurement, percent)
      = try? await HealthStore.shared.currentWaterStatus()
  else {
    consumed = "0"
    percent = "Unknown"
    return
  }

  // 2
  consumed = consumedFormat.string(from: measurement)

  // 3
  self.percent = percent?
    .formatted(.percent.precision(.fractionLength(0))) ??
"Unknown"
}
```

Apologies for the formatting required to keep the lines from wrapping. Here's what's happening:

1. If you can't read the current water status, you set a couple of default values.
2. If you can read the data, you format the amount of water consumed via a `MeasurementFormatter` that you'll create in a moment.
3. Taking advantage of Foundation's new number formatters, you either calculate the daily percentage consumed or specify that it's `"Unknown"` if you couldn't determine the person's weight.

To resolve the compiler error, add the following formatter at the bottom of the file, outside of the `struct`:

```
private let consumedFormat: MeasurementFormatter = {
  var fmt = MeasurementFormatter()
  fmt.unitOptions = .naturalScale
  return fmt
}()
```

Next, update the `body` to show the data by adding this code right after the existing `Stack`:

```
HStack {
  Text("Today:")
    .font(.headline)
  Text(consumed)
    .font(.body)
}

HStack {
  Text("Goal:")
    .font(.headline)
  Text(percent)
    .font(.body)
}
```

Finally, update the status when the view appears by adding a `task` after the `ScrollView`:

```
.task {
  await updateStatus()
}
```

Now, to update the display if you record new data, add the update line just before the end of the `Task` block in `logWater(quantity:)`:

```
await updateStatus()
```

Build and run again. This time you'll see that you have logged water previously:

Reading multiple days of data

Finally, to make the interface a bit nicer, why not show the amount of water the user consumed over the last week? Reading multiple days of data is a bit more complicated than reading just a single day. Back in **HealthStore.swift**, add a new property to the top of the class:

```
private var preferredWaterUnit = HKUnit.fluidOunceUS()
```

And then set the value at the end of the `Task` block in the initializer:

```
guard let types = try? await healthStore!.preferredUnits(
  for: [waterQuantityType]
) else {
  return
}

preferredWaterUnit = types[waterQuantityType]!
```

As you can see, it's possible to determine what type of units your user prefers to see their measurements. You initialized the property to `HKUnit.fluidOuncesUS()` because there must be a value before the initializer ends.

Now add a new method to query the week's water consumption:

```
func waterConsumptionGraphData(
  completion: @escaping ([WaterGraphData]?) -> Void
) throws {
  guard let healthStore = healthStore else {
    throw HKError(.errorHealthDataUnavailable)
  }

  // 1
  var start = Calendar.current.date(
    byAdding: .day, value: -6, to: Date.now
  )!
  start = Calendar.current.startOfDay(for: start)

  let predicate = HKQuery.predicateForSamples(
    withStart: start,
    end: nil,
    options: .strictStartDate
  )

  // 2
  let query = HKStatisticsCollectionQuery(
    quantityType: waterQuantityType,
    quantitySamplePredicate: predicate,
    options: .cumulativeSum,
    anchorDate: start,
    intervalComponents: .init(day: 1)
  )

  // 3
  query.initialResultsHandler = { _, results, _ in
  }

  // 4
  query.statisticsUpdateHandler = { _, _, results, _ in
  }

  healthStore.execute(query)
}
```

Here's a code breakdown:

1. After verifying you can use HealthKit, you determine the start of the day six days ago and then set up a predicate to query all data from that time forward.
2. Then, you construct a query for water consumption, using the `predicate`, and ask HealthKit to sum up each day for you. You specify that it should perform the summation across day boundaries.

3. You call `initialResultsHandler` the first time the query completes.
4. Then you call `statisticsUpdateHandler` any time the user adds new data. By not specifying an end date on the predicate, you ensure that updates are captured.

Since both of the handlers perform the same actions, create a helper method that they'll both call:

```
func updateGraph(
  start: Date,
  results: HKStatisticsCollection?,
  completion: @escaping ([WaterGraphData]?) -> Void
) {
  // 1
  guard let results = results else {
    return
  }

  // 2
  var statsForDay: [Date: WaterGraphData] = [:]

  for i in 0 ... 6 {
    let day = Calendar.current.date(
      byAdding: .day, value: i, to: start
    )!
    statsForDay[day] = WaterGraphData(for: day)
  }

  // 3
  results.enumerateStatistics(from: start, to: Date.now) {
    statistic, _ in

    var value = 0.0

    // 4
    if let sum = statistic.sumQuantity() {
      value = sum
        .doubleValue(for: self.preferredWaterUnit)
        .rounded(.up)
    }

    // 5
    statsForDay[statistic.startDate]?.value = value
  }

  // 6
  let statistics = statsForDay
    .sorted { $0.key < $1.key }
    .map { $0.value }
```

```
    // 7
    completion(statistics)
  }
```

Can you see why you don't want to duplicate that code twice? Here's what the code does:

1. If no results return, then you exit early.
2. You create a dictionary keyed by the last seven days that contains the data to graph.
3. Then, you loop through the computed results.
4. If there's data for the day, you sum the data counts, convert the value to their preferred unit of measurement and round up to the nearest whole number.
5. Then, you record the amount of water consumed for the day.
6. You take the values for each day in ascending order.
7. Finally, you pass the data back through the completion handler.

If there's no data for a given day, you won't receive a value for that date. That's why you store the values in a dictionary, so you have an easy way to generate an array with seven elements in the proper order, even if there's no data for a given day.

Finally, call this method from the handlers:

```
  query.initialResultsHandler = { _, results, _ in
    self.updateGraph(
      start: start, results: results, completion: completion
    )
  }

  query.statisticsUpdateHandler = { _, _, results, _ in
    self.updateGraph(
      start: start, results: results, completion: completion
    )
  }
```

Back in **WaterView.swift**, add a new property:

```
  @State private var graphData: [WaterGraphData]?
```

Add the `BarChart` to the `body` just after the final `HStack`:

```
BarChart(data: graphData)
  .padding()
```

Then call the query method from the `.task` at the end of the `body`:

```
.task {
  await updateStatus()
  try? HealthStore.shared.waterConsumptionGraphData() {
    self.graphData = $0
  }
}
```

Build and run one last time. You have a nice bar chart that keeps itself updated as you record entries:

Key points

- Make sure you don't try to track a type of data that isn't available on the OS versions you support. If you find yourself in that situation, ensure that you define the identifiers as optionals.
- Always use the user's preferred types when displaying or converting data. Never hardcode a unit type to display to the user.

16 Conclusion

As you've seen, the Apple Watch is a small device with endless possibilities if you know how to make the most of its potential.We hope this book has given you all the knowledge you need to create amazing apps using SwiftUI on your Apple Watch.

Remember that you can always go back to the chapters to review some specific concepts. We've tried to make it useful not only for beginners starting to develop for this device but also for more advanced programmers to have it as a reference book.

No matter what your case is, we hope you've really enjoyed this book. Remember, the best thing you can do from now on is try to implement everything you've learned. When you develop a new application, update an existing one or create one just for fun, don't forget that you can add great value to it if you accompany it with an Apple Watch app.

If you have any questions or comments as you work through this book, please stop by our forums at https://forums.raywenderlich.com and look for the particular forum category for this book.

Thank you again for purchasing this book. Your continued support makes the books, tutorials, videos and other things we do at raywenderlich.com possible. We truly appreciate it!

– The *watchOS With SwiftUI by Tutorials* team