HACKING WITH SWIFT

HACKING WITH watchOS

SWIFTUI EDITION

Learn to make watchO
apps with real-wo
Swift projects

FREE SAMPLE

Paul Hudson

Project 1

NoteDictate

NoteDictate: Setting up

In this project you'll produce an application that lets users save notes directly to their watch, then browse through and read previous notes. In doing so you'll learn how to show lists of data for the user to choose from, how to move between different screens of information, and how to convert the user's input into text so it can be read.

Yes, I realize that sounds quite a lot, but don't worry: this project is deliberately designed to take it slow so everyone can learn. The pace picks up in later projects, but if you hit problems you can always refer back here as a reminder.

Let's get started: launch Xcode, and choose "Create a new project" from the welcome screen. You'll be asked to choose a template for the new project, so please choose watchOS > App.

For Product Name enter Project1, then make sure you have Watch-only App selected because we don't plan to use iOS at all here. You'll see a checkbox marked "Include Tests" – please uncheck that.

One of the fields you'll be asked for is "Organization Identifier", which is a unique identifier usually made up of your personal website domain name in reverse. For example, I would use **com.hackingwithswift** if I were making an app. You'll need to put something valid in there if you're deploying to devices, but otherwise you can just use **com.example**.

Now click Next again and you'll be asked where you want to save the project – your desktop is fine. Once that's done, you'll be presented with the example project that Xcode made for you.

The first thing we need to do is make sure you have everything set up correctly, and that means running the project as-is.

In the top-left corner of Xcode you'll see a play button and a stop button. Immediately to the right of that you'll see something like "Apple Watch Series 8 (45mm) via iPhone 14 Pro". There are several other sizes that exist, but this one is a common configuration.

Now go ahead and press the play button now to build and run your app. It will take a little

while at first because Xcode will launch the Watch Simulator, which has to boot up watchOS before your app can run.

Eventually you'll see our app, such as it is. Our "app" just shows an empty window with the words "Hello, world!" centered below a globe icon – it does nothing at all, at least not yet. You'll see the time in the top-right corner, which is automatically inserted by watchOS, cutting your available screen space even further.

You'll be starting and stopping projects a lot as you learn, so there are three basic tips you need to know:

- You can run your project by pressing Cmd+R. This is equivalent to clicking the play button.
- You can stop a running project by pressing Cmd+. when Xcode is selected.
- If you have made changes to a running project, just press Cmd+R again. Xcode will prompt you to stop the current run before starting another. Make sure you check the "Do not show this message again" box to avoid being bothered in the future.

Obviously this is a pretty dull app as it stands, so let's dive right into making something better…

Defining our data

The first step in many programs is to decide what your data should look like – what are we actually working with? In this introductory project our data is deliberately simple: we're storing notes from users, so that's just text.

However, one of the few hard and fast rules that SwiftUI gives us is that it wants to know how to identify each piece of application data uniquely. Think about it: if look on Twitter for people called Paul Hudson you'll find there are many of us out there, so Twitter needs a way to identify each of us uniquely. For Twitter that's a username – I'm @twostraws, for example – but behind the scenes they also store a unique identifying number, which for me is 55964332.

SwiftUI has the same problem as Twitter: if our user created three notes with the text "cook dinner", it needs a way to know which note is which. Without that, if we say "delete the note that says cook dinner," SwiftUI won't know which one should be removed.

This is solved using a Swift protocol called **Identifiable**, which has a single requirement: we must provide a property called **id** that is unique.

Let's try this out now. Go to the File menu and choose New > File, then select Swift File. Name it "Note.swift", then give it this code:

```
struct Note: Identifiable {
  let id: UUID
  let text: String
}
```

That creates a new **Note** struct conforming to the **Identifiable** protocol. As you can see, it stores a string called **text**, which is what contains the actual text for our user's note. However, there's another property in there: an **id** property that is a **UUID**. "UUID" stands for "universally unique identifier", and it's a long hexadecimal sequence of letters and numbers that is designed to be unique. Unless you create them by hand (please don't!), you could generate a million UUIDs every day for a million years and would still be unlikely to generate

the same one twice.

Note: The **Identifiable** protocol requires a property called **id** that identifies each instance of the struct uniquely. Using **UUID** is a good idea because they are naturally unique, but you can use anything you want as long as you're sure it will identify your data uniquely.

Working with views

Now that we have some data to work with, I want you to open ContentView.swift so we can start using it. This file is the default in all SwiftUI projects on all platforms, and provides a simple SwiftUI layout that produces the "Hello, World!" text you saw earlier.

You should see something like this:

```
struct ContentView: View {
  var body: some View {
    VStack {
      Image(systemName: "globe")
        .imageScale(.large)
        .foregroundColor(.accentColor)
      Text("Hello, world!")
    }
    .padding()
  }
}
```

There are lots of small but important things in there, so let me break it down quickly:

1. It defines a new struct called **ContentView**. All our SwiftUI layouts are structs because behind the scenes these are simpler and faster for the system to create and manager.
2. This struct conforms to the **View** protocol. In SwiftUI anything that renders content must conform to the **View** protocol so that it can tell SwiftUI what it draws.
3. The struct has a computed property called **body**, which is the only requirement of the **View** protocol. This is where you create your layouts.
4. As you can see, this has the return value **some View**, which means "this view will itself return some sort of views" – it might be images, text, sliders, buttons, and so on, but ultimately every view must return something to draw.
5. In this case our body contains two things: an image view loading the "globe" system icon from the SF Symbols set, and a **Text** view with the string "Hello, world!" We'll look at SF

Symbols more soon.

6. The text and image are wrapped inside a **VStack**, which means the image and text will be stacked vertically. Alongside **VStack**, SwiftUI provides other layout options including **HStack** to place views horizontally, and **ZStack** to place views so they overlap.
7. There are three method calls: **imageScale()** makes the icon larger than normal, **foregroundColor()** tints the icon with the apps accent color (blue by default), and **padding()** adds some spacing around a view so that other views don't butt up directly against it. In SwiftUI we call theses *modifiers*, because they modify the way the views look. It's common to have many modifiers stacked up for a single view.

Below that view struct is a second struct that conforms to the **PreviewProvider** protocol. This second struct is for debugging purposes only, and is designed to show you a live preview of your layouts while you work – that's the canvas on the right of your code. You can show or hide this canvas by going to the Editor menu and clicking Canvas.

Over time you'll learn more about how these work and what their subtleties are, but for now you know enough to continue.

We want this main **ContentView** struct to have an array of notes, so we can add, view, and remove them freely. This is as simple as adding a property, like this:

```
var notes = [Note]()
```

For now, let's start simple: a text view showing how many notes are in our array, plus a button that adds example notes every time it's pressed.

First, replace the existing **VStack** code with this:

```
VStack {
  Text("Notes: \(notes.count)")
}
```

That uses string interpolation to put the array count right into the text view.

A vertical stack containing only one view is no different from removing the **VStack** entirely, but I want you to keep it because we're going to add a second view below the text: a button to add test notes. SwiftUI gives us a number of ways of creating buttons, but the simplest one has a text string as its title and an action provided as a trailing closure. In our case, that means using "Add Note" as the title, and for the action we'll create a new note and append it to the **notes** array.

Put this below the text view:

```
Button("Add Note") {
  let note = Note(id: UUID(), text: "Example")
  notes.append(note)
}
```

That creates a new, random UUID for our note, plus the same example text each time.

If everything is going to plan, your code should look like this:

```
struct ContentView: View {
  var notes = [Note]()

  var body: some View {
    VStack {
      Text("Notes: \(notes.count)")

      Button("Add Note") {
        let note = Note(id: UUID(), text: "Example")
        notes.append(note)
      }
    }
  }
}
```

However, chances are Xcode is shouting at you – it doesn't like the code.

If you think back to the way structs work, you might remember the **mutating** keyword that must be used on any methods that change structs. This exists because Swift doesn't know if your struct was created as a variable or a constant, so we need to say ahead of time "this thing will change."

Well, here we have a similar problem: we're trying to change a property of our struct inside a computed property, and for all Swift knows that struct could be a constant. Sadly, there is no way to make mutating computed properties in Swift, which is why Xcode is complaining.

SwiftUI relies heavily on a solution called *property wrappers*. These are a Swift feature that let us add special behaviors to properties of a type – we might say "never let this number be negative," or "make this property load and save its value automatically," for example. In the case of SwiftUI, there is a special property wrapper called **@State**, which means "this property represents the active, changing state of my program, and I want SwiftUI to manage it."

So, by default we can't modify properties in our struct, but if we mark them with **@State** then we can – we tell SwiftUI to manage them for us, so even though the view struct is a constant, the values can change.

To bring in this property wrapper, change the **text** property to this:

```
@State var notes = [Note]()
```

That's enough to fix our compile error, which means our code will build now. However, before we try it out I want to make one further change:

```
@State private var notes = [Note]()
```

As you learn more about SwiftUI you'll see there are several property wrappers available to us, and the **@State** one is specifically used for simple values that are stored locally to a view – they aren't shared with other views. As a result, Apple encourages us to mark all **@State** properties as private, to reenforce that they aren't supposed to be shared.

Our code compiles now, so go and press Cmd+R to see it in action. You should see "Notes: 0", and every time you click the button that number should increment. Nice! Well, sort of – it's still hardly an app, but at least it's starting to work a little.

One thing to note here is how SwiftUI automatically updates the text string as **notes.count** changes. This is part of the behavior of **@State**: it will automatically reload your view's **body** property when its value changes.

Showing a list of notes

One of the most important view types in SwiftUI – not just on watchOS, but on all of Apple's platforms – is called **List**, and it shows a scrolling list of rows that is perfect for tabulated data.

You can create lists directly from data if you want: just pass the list some sort of array of data that conforms to **Identifiable**, along with a function to run that converts each item in the array into a new view. Put another way, you give **List** all the items you want it to render, and it will pass you back individual items and say "what do you want to do with this one?"

For example, we could make a really simple list in our UI by adding this directly after the button closure:

```
List(notes) { note in
  Text(note.text)
}
```

So, we provide **notes** as its input, along with a trailing closure that accepts a single **Note** instance. We then convert that note's data into a text view, which the list will then place inside a scrolling container.

If you run the app now you'll see our list in action – press the button a few times, and you'll find you can scroll around the table below. Nice!

Another way of creating lists is by looping over a range, and this is the form we'll be using in our app because it allows us to read the index of each item (note 0, note 1, note 2, etc) along with the note itself.

Here's how that looks:

```
List(0..<notes.count, id: \.self) { i in
  Text(notes[i].text)
}
```

You might wonder what that **id: \.self** part means, and with good reason – it's new!

Remember that rule I mentioned earlier, about how SwiftUI needs to know how it can identify each piece of application data uniquely? This applies to the views in a list too, which is why it's important that **Note** conform to **Identifiable**.

If you're looping over a range of numbers a fixed number of times – if you're always showing five buttons, for example – then using a range like **0..<5** is fine. But if your range will change over time, such as ours does because we're appending to the **notes** array, then we can't just use a simple range. Instead, we must provide a second parameter called **id**, which tells SwiftUI how it can identify values as they change over time.

When we were working with the **Note** struct, our identifying value was a UUID that was guaranteed to be unique for each note. Here, though, we have simple integers counting from 0 up to the number of notes we have, so we don't have an option to add anything else such as a UUID. Instead, we have each number itself, and nothing more.

In Swift, that means when we're asked for an identifier for each value, we can just write **\.self** to mean "the number itself will be unique." This means SwiftUI is happy that it can identify each value uniquely, and it's great for times like this when we have a variable array of data being shown.

Now that our list works properly, go ahead and remove the text view showing **notes.count** – we don't need that any more.

Reading user input

Obviously having "Example" repeated again and again doesn't make for a very useful app, so for our next step we're going to add a text input box that lets users add their own notes.

I mentioned the **@State** property wrapper earlier, saying that it stores the active state of our program. "Program state" is all the active values that represent how our app is being used right now – what screen the user is on, whether they have toggle a switch on or off, what they have typed in a text box, and more.

Our views are *not* state, but our program state will often dictate which views are showing. For example, typing a short password might show some text prompting the user to enter a longer password.

We already have one **@State** property to store the array of notes the user has created, but now we're going to add a second one: a string to store some user input. This will be an empty string by default, but we'll attach it to a text input view so as the user types their note it gets stashed away into this property.

Start by adding this new property to **ContentView**:

```
@State private var text = ""
```

We want to use that property inside a text input view, so the user can write their note. In SwiftUI this is called **TextField**. This is different a text view, which doesn't take any input – it just shows a value. So, whereas **Text** just reads a value, **TextField** reads *and writes* a value: it shows whatever is in its string, but if you change the text field then it updates the string as well.

Let's try this out now. Put this new view above the button:

```
TextField("Add new note", text: text)
```

That first parameter, "Add new note", is placeholder text that will be shown if the text field is

empty.

Don't worry if Xcode shows a big red error – that's intentional. You see, **TextField** doesn't just want to know where to read its text, it also wants to know where to *write* its text – where to store any changes the user made. In SwiftUI this is called a *binding*: when we bind a text field to a property, we mean "show its value here, but if we change the value on screen then update the property as well."

Using **@State** allows us to modify properties inside a struct freely, because it tells SwiftUI to manage the data automatically. However, it also does something else: it automatically creates a binding for our property that we can use anywhere we need to both read and write the value.

So, when we say **text: text** we means "read the value from the **text** property." But if we add a dollar sign – Swift's way of making a binding from an **@State** property – then we mean "read *and write* the value using the **text** property." This means the text field will start by showing whatever is in **text** when the view first appears, but when the user changes the text field will save the new value back into **text**.

Change your text field to this:

```
TextField("Add new note", text: $text)
```

And now the error is gone.

Let's go ahead and use **text** immediately by modifying the action closure for our button. Rather than just appending example notes again and again, we're instead going to:

1. Make sure the **text** property isn't empty, because we don't want the user to create empty notes by accident.
2. Create a new **Note** instance using the **text** property, appending it to our **notes** array.
3. Clear the **text** string after the note is added, so they can write another.

Replace your existing button with this:

```
Button("Add Note") {
  guard text.isEmpty == false else { return }

  let note = Note(id: UUID(), text: text)
  notes.append(note)
  text = ""
}
```

Go ahead and run the app now, then tap on the "Add New Note" placeholder. This will automatically bring up a new view showing the watchOS input options – voice dictation, emoji, and either the keyboard or a scribble space depending on your configuration.

Try entering "Hello" or whatever test data you like, then press Done to accept your input. You should return to the original content view, with your new text shown above the "Add Note" button. Even better, tapping that button will show your new note text in the list below, and clear the text field – brilliant!

However, now that our notes are meaningful you'll notice a problem with our app: half the screen is taken up with entering new notes, leaving only a small amount for actually scrolling through the list of what we have. Given that users are likely to be reading notes far more than they are writing notes, this isn't great so we need to address this.

A simple way to fix this is to put the text field and button on the same horizontal row, so they are positioned side by side rather than one above the other. This is done with a **HStack** in SwiftUI, and it's perfectly fine to put one stack inside another.

So, start by putting both the text field and button inside a **HStack**, like this:

```
VStack {
  HStack {
    TextField("Add new note", text: $text)

    Button("Add Note") {
```

```
            guard text.isEmpty == false else { return }

            let note = Note(id: UUID(), text: text)
            notes.append(note)
            text = ""
        }
    }

    List(0..<notes.count, id: \.self) { i in
        Text(notes[i].text)
    }
}
```

Immediately you'll see that's saved a lot of space, but it's caused a different problem: there isn't really enough room for both the button and the text view to sit on one line.

We can try to fix this by using something shorter than "Add Note", and helpfully watchOS comes with a massive library of Apple-created images called SF Symbols. These icons can be used anywhere in watchOS using an **Image** view, and there's even a dedicated SF Symbols app that lets you search through the library of images to find the one you want – if you search for "SF Symbols" online you'll find it as a free download from Apple.

Creating a button from an image looks a little different from using text, because rather than writing **Button("Some Text")** you instead provide it as a second trailing closure that describes the label of the button. This is a little more long-winded, but it does have the advantage that your button label can be anything you want – maybe you want a **HStack** with both an image *and* some text, for example.

Replace your existing button code with this:

```
Button {
    guard text.isEmpty == false else { return }
```

```
    let note = Note(id: UUID(), text: text)
    notes.append(note)
    text = ""
} label: {
    Image(systemName: "plus")
        .padding()
}
```

There are two new things in there:

1. The second trailing closure is called **label**, which defines the visual content of the button. You don't have to write this second trailing closure on the same line as the closing brace before it, but it's recommended for readability purposes.
2. The **Image** view has a **systemName:** initializer, which tells SwiftUI we want to load one of the SF Symbols icons – "plus", in this case.

If you run the app now you'll see the content of our button is smaller – just "+" rather than "Add Note" – but the button itself has stayed the same size. This happens because **HStack** will automatically split up its space so that each view inside it has an equal amount.

We don't want that behavior here, so instead we're going to tell the button to ignore whatever the parent suggests and instead fix its size at whatever makes sense for the button's label. As we have just a + sign for its label, this will make the button nice and small.

To fix a button's size, add the **fixedSize()** modifier at the end of the button, like this:

```
Button {
    guard text.isEmpty == false else { return }

    let note = Note(id: UUID(), text: text)
    notes.append(note)

    text = ""
```

```
} label: {
  Image(systemName: "plus")
    .padding()
}
.fixedSize()
```

And now it's looking much better – we have lots more space for our scrolling list, and our text field takes up as much space as it can.

To finish up, we're going to make one last small change: we're going to give the button a color, to help break up the otherwise monotonous grayscale we have right now.

Unlike iOS, buttons on watchOS have a very precise design by default, and there is limited scope to customize that beyond doing something completely unique. However, one thing we can do is tint the button using a color of our choosing – it will retain the default button shape and interactivity, but have a small amount of color.

This is all done using the **tint()** modifier, which can either use a custom color or one of the many built-in options. Add this below **fixedSize()**:

```
.tint(.blue)
```

It's a small change, but I think it helps bring the UI to life just a little.

Selecting and deleting notes

To make this app actually useful, we need to add at least two more things: the ability to tap on a note to see more information, and the ability to remove any notes we don't want any more.

First, let's create a new view that shows one note all by itself. Go to the File menu and choose New > File, or press Cmd+N to get the same result. From the list of options, please choose SwiftUI View, then name it DetailView.swift.

This new detail view is designed to show a single note, but we're also going to display the note's *number* – whether it's the first note, then 10th, and so on. We can't figure out the note's position in our array just from the note itself, so we're going to pass that index plus the note as properties for our new view.

So, start by adding these two as properties for **DetailView**:

```
let index: Int
let note: Note
```

You'll get an error because the preview provider in DetailView.swift tries to create a **DetailView** without passing any data for those two properties. Fortunately, that can be fixed by just passing in example values, like this:

```
struct DetailView_Previews: PreviewProvider {
  static var previews: some View {
    DetailView(index: 1, note: Note(id: UUID(), text: "Example
Note"))
  }
}
```

Inside the body of **DetailView** we don't want "Hello, world!", but instead we want to show our note's text, like this:

```
Text(note.text)
```

Like I said, we also want to show the note's number. We could put that in a second text view, but SwiftUI gives us a better solution with its **navigationTitle()** modifier. This places text of our choosing into the navigation space, which is where the clock is right now – it's a bonus bit of UI space that you should definitely make use of.

Because Swift's arrays count from 0, we need to add 1 to our note index to make the numbers make sense for users. So, use this for your final **body** property in **DetailView**:

```
Text(note.text)
  .navigationTitle("Note \(index + 1)")
```

Even though we added a navigation title to our view, you *won't* see it reflected in the preview. This is because we need to activate navigation for our view by wrapping it in a **NavigationStack**. Now, in a moment we're going to add a **NavigationStack** in **ContentView**, but that wouldn't help our preview – as far as our preview is concerned, **DetailView** is being shown in isolation, so even if we added a **NavigationStack** in **ContentView** nothing would change for our preview.

Fortunately, these previews are ours to customize – we can put any code we want into the **DetailView_Previews** struct and it will only affect the way our preview looks in Xcode, rather than actually changing the finished app that gets run.

So, we can tell SwiftUI to preview **DetailView** as it will actually look when the code runs, like this:

```
static var previews: some View {
  NavigationStack {
    DetailView(index: 1, note: Note(id: UUID(), text: "Example
Note"))
  }
}
```

It's not *exactly* the same as it will look at runtime, because when we place a navigation title in a detail view watchOS will automatically make the text smaller and part of a back button, but at least it means you can see the note title is up there.

That completes the detail view, so head back to ContentView.swift. When one of our list rows is tapped, we want to show **DetailView** with the correct index and note. SwiftUI comes with a built-in view for this purpose, called **NavigationLink**: give it a destination to show, plus some content that should be tappable, and it will do all the work of showing and hiding new views.

In this project, our destination view should be a new **DetailView** containing the correct index and note, and the label of the navigation link will be our existing **Text** view. So, adjust your current list to this:

```
List(0..<notes.count, id: \.self) { i in
  NavigationLink {
    DetailView(index: i, note: notes[i])
  } label: {
    Text(notes[i].text)
  }
}
```

As with button labels, the content of your navigation links can be whatever you want – SwiftUI will make it all interactive, so that tapping any part of the link will create and show a correctly configured detail view.

Remember, in order to make navigation actually work we need to wrap our current layout with a **NavigationStack**. Helpfully, we can actually replace the current **VStack** with **NavigationStack** to get exactly this result – SwiftUI will automatically place the button and list vertically inside the navigation.

Now that we have the ability to see a full note in our detail view, we can make a small change to the text view in our list rows. By default, text views automatically show their entire content, with long strings being wrapped over multiple lines.

Showing the full note will work great in our detail view, but it's likely to be rather odd in the list view because there are many notes. Fortunately, we can tell SwiftUI to limit the number of lines in our list text with the **lineLimit()** modifier:

```
Text(notes[i].text)
  .lineLimit(1)
```

That will make sure only one line is ever shown, but you could try 2 if you wanted. Line limits are a maximum, meaning that if the text is just a single line then it won't take up more than 1 line of space.

Before we're done with this project, I want to make two further small changes.

First, in the same way we used **navigationTitle()** in our detail view, we can place a title in the status bar for **ContentView**. Add this modifier at the end of the **List**:

```
.navigationTitle("NoteDictate")
.navigationBarTitleDisplayMode(.inline)
```

That needs to belong to a view *inside* the **NavigationStack** rather than the **NavigationStack** itself – remember, the currently active views will change over time, so watchOS automatically picks a title from whatever is currently showing.

You've seen the first of those modifiers previously, but the second is there to request a smaller font size for the title – without that it will occupy a significant amount of our screen space, which is a poor user experience.

Second, we should really let users delete the notes they create. This isn't *hard* to do, but it does require learning a slightly different way to create lists. You see, SwiftUI's lists are great at looping over arrays of identifiable content and turning each item into a view; that's what we are doing right now. But lists can also show fixed content: you might show a button, then show an array of content, then some text, then another array, and so on.

All this matters because lists don't let us delete their content, because they handle both fixed

and dynamic content. Instead, we need to rewrite our list so that it contains a loop inside it, using a new view type called **ForEach**. This works just the same as **List** does: we can provide it a range and an identifier, along with a closure to run for each item.

Change your current list code to this:

```
List {
  ForEach(0..<notes.count, id: \.self) { i in
    NavigationLink {
      DetailView(index: i, note: notes[i])
    } label: {
      Text(notes[i].text)
        .lineLimit(1)
    }
  }
}
```

That produces exactly the same code, but it does have one important difference: because it's really clear that part of our list is dynamic – is loaded from an array rather than static views, or a mix of static and dynamic – we can add a new modifier called **onDelete()**, which tells SwiftUI how to handle deletion of rows.

Before we add that modifier, we first need to solve a bigger problem: what should happen when we delete rows?

Well, Swift has a special kind of array called **IndexSet**. This is a bit like an array of integers, except no number can appear twice, and all numbers must be 0 or greater. This makes it great for referring to indexes inside an array, and it's what SwiftUI uses when deleting items from a list. Helpfully, it maps directly to a Swift array method called **remove(atOffsets:)**, which takes an **IndexSet** of items to remove.

So, we can delete items from the **notes** array by adding a single method to **ContentView**:

```
func delete(offsets: IndexSet) {
```