SWIFT ALPS 2019

# SwiftUI Workshop

Tasty Slopes

Lea Marolt Sonnenschein
@hellosunschein
lea.marolt@gmail.com

# Contents

**0. Introduction** **4**
Getting Started 6
**1. Piste Picker** **7**
Task 2: Image 12
Task 3: VStack 16
Task 4: Reusable 19
Task 5: ScrollView 22
Task 6: Handling Events 27
**2. Piste Log** **29**
Task 7: Cells 31
Task 8: Lists 35
Task 9: @State 38
**3: Tabbed Navigation** **41**
Task 10: Tabbed Navigation 43
Task 11: Navigation Title 46
Task 12: Profile View 48
**4: Managing Data** **50**
Task 13: App State 51
Task 14: Forms and Sections 53
Task 15: @ObservedObject 58
Task 16: Navigation Buttons 60
Task 17: Refactor with AppState 62
Task 18: Refactor with AppState #2 64
Task 19: Showing the Goods 66
**5: Extra Credit** **68**
Task 20: Alert 69
Task 21: Modals and Pushes 71
**6: Resources** **73**

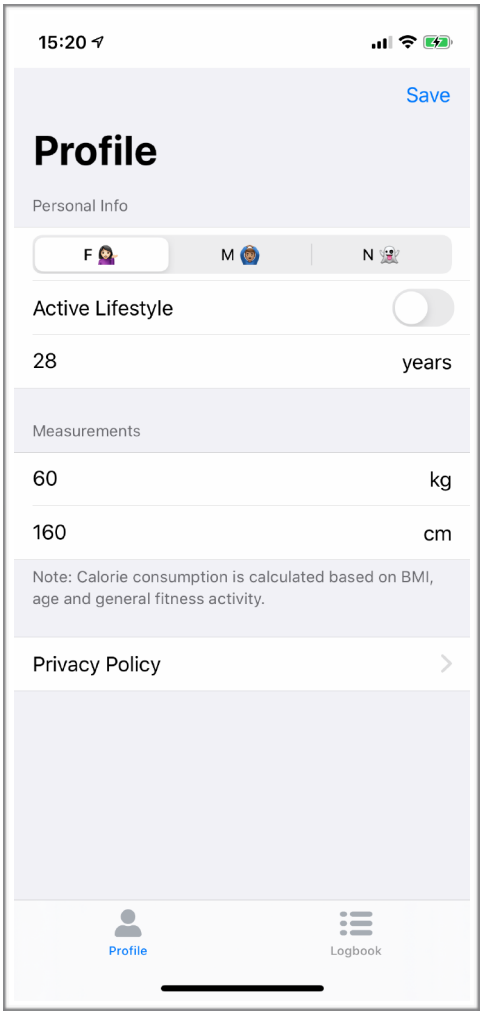
| | |
|---|---|
| Free | 73 |
| Paid | 73 |

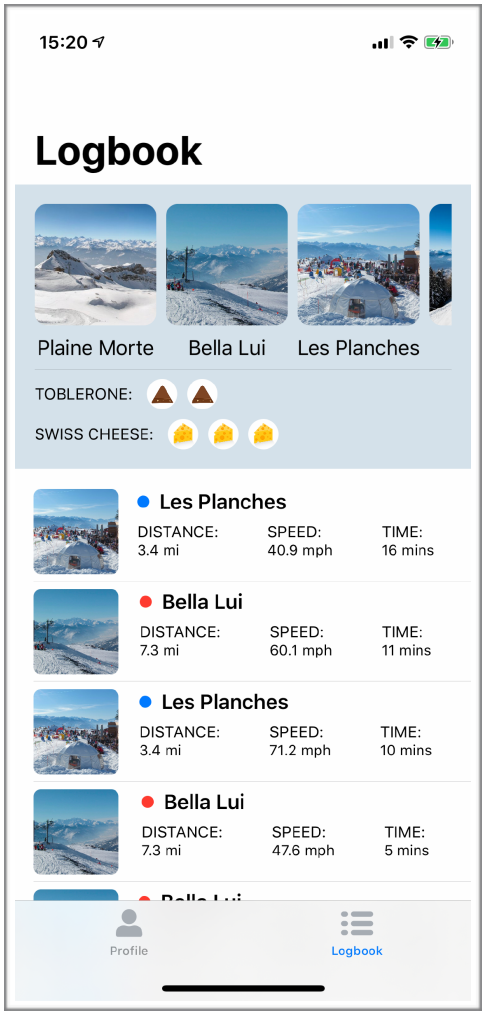
# 0. Introduction

Welcome to the SwiftUI Workshop!

The goal of this workshop is to get you familiar with the fundamental principles and building blocks of SwiftUI by building a simple app from scratch in the next 2 hours.

During these 2 hours you'll pair up with another participant and work through challenges to build Tasty Slopes.



Tasty Slopes is an app that lets you log your daily ski activities and shows you how many tasty treats (i.e Toblerone and Swiss Cheese) you deserve to eat!

I've split up the 2 hours in 21 tasks for you to accomplish. Instead of bogging you down with the theory up front, you'll be introduced to bits and pieces as you work through the challenges, so that you can apply your knowledge immediately, which is the best way to learn (:

To get the most out of this workshop you should:

1. Work in pairs, and follow the principles of pair programming, where one person drives for about 20 minutes and then switch.
2. Ask questions to your fellow participants, me or Google (in that order!).
3. If you're still stuck... you can take a peek at the 5 chapter folders which have the final versions of the app at the end of each chapter.

**Note:** This workshop is **most likely** too long for 2 hours if you haven't already dabbled a little with SwiftUI. Do not worry! It's designed to be that way, to make sure that it's engaging for people at different levels.

**So if you only make it through half of it, that'd perfectly fine!** You'll have all the resources needed to finish it later at your own pace (:

# Getting Started

So that you don't have to start from scratch, I've created a starter project in **Projects > 0-Starter** that includes the following:

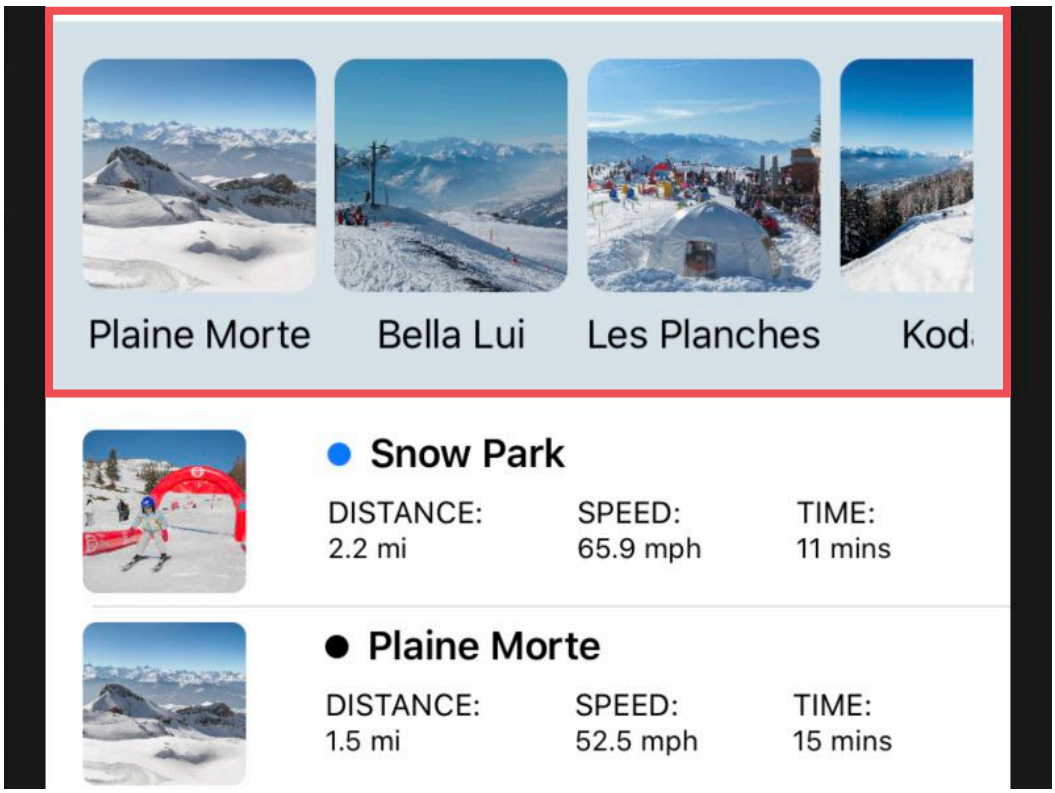
1. All the image and colour assets you'll need
2. A file called `slopesData.json` that contains all the data to render the slopes in a JSON format
3. An extension on `UserDefaults+Extensions.swift` that allows you to store and retrieve user data in **TastySlopes > Helpers**
4. An extension on `TimeInterval+Extensions.swift` and `Int` to nicely render minute strings in **TastySlopes > Helpers**
5. Object models which contain simple structs you can use for easier data management in **TastySlopes > Models:**
    a. `User.swift,`
    b. `Piste.swift,`
    c. `Run.swift`
    d. `Log.swift`
    e. `Treats.swift`
6. A few files for you to fill in as we go along, because you'd learn nothing from going through the motions of making them:
    1. `AppState.swift`
    2. `LogbookView.swift`
    3. **TastySlopes > Views >** `PrivacyPolicyView.swift`
    4. **TastySlopes > View Models >** `ProfileVM.swift`

Let's get going :D!

# 1. Piste Picker

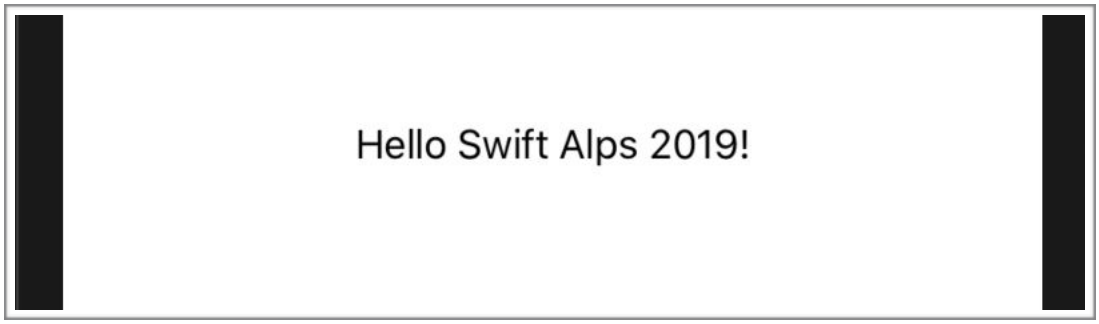
**Goal:** You'll first tackle creating the piste picker. This is where you scroll through the 5 top pistes. When you tap on a piste you add a run to your log.

**Learning:** In this section, you'll learn the basic principles of creating, modifying and grouping views, creating reusable views and responding to events.




Let's take this step by step. Step number one, build and run the app!

When you first build and run, you should see this lovely greeting:




It's rendered in **LogbookView.swift**:

```
struct LogbookView: View {

  var body: some View {
    Text("Hello Swift Alps 2019!")
  }
}
```

And called from the **SceneDelegate.swift** as the root view. In SwiftUI, the **SceneDelegate** is responsible for managing life-cycle events, rather than the **AppDelegate**.

There are a two other things to unpack here, related to Swift 5.1.

**1. Opaque return types** [reference]**:**

```
some View
```

`some` is an **opaque return type** that specifies what protocols the return value has to conform to. In our case it's the `View` protocol.

**2. Implicit returns** [reference]**:**

```
Text("Hello Swift Alps 2019!")
```

You'll notice that there's no `return` statement before `Text`. That's because, Swift 5.1 lets us omit the `return` keyword when there's only one expression.

# Task 1: Data

## Show the name of a `Piste`.


Since we're in the Swiss Alps, there's nothing better to do than ski! Therefore, the app and the data you're working with is all about hitting the slopes. **`slopesData.json`** defines 5 of the most popular pistes to ski on in Crans Montana (not actually true...).

Instead of the current greeting, show the name of a piste.

You can access all the pistes through the static `pistes` variable in **`Piste.swift`**.

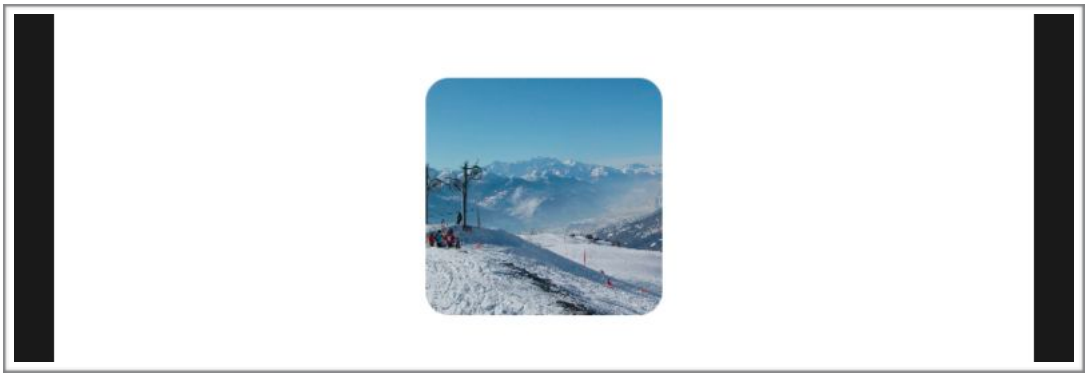
```
static var pistes: [Piste]
```

**To Do:** Show the name of an actual piste and your app should look similar to:


**Tip:** Instead of building and running the simulator each time, you can use the **Canvas** view next to your Editor to preview your UI changes. Make sure it's turned on.

# Task 2: Image

Show and style the image of a `Piste`.



Seeing the name of the piste is nice, but an image is worth a thousand words. To render images in SwiftUI, you use the `Image` struct. There's many ways to create an image, but you'll most commonly create them from your asset folder or system icons.

Asset folder:

```
var body: some View {
  Image("example")
}
```

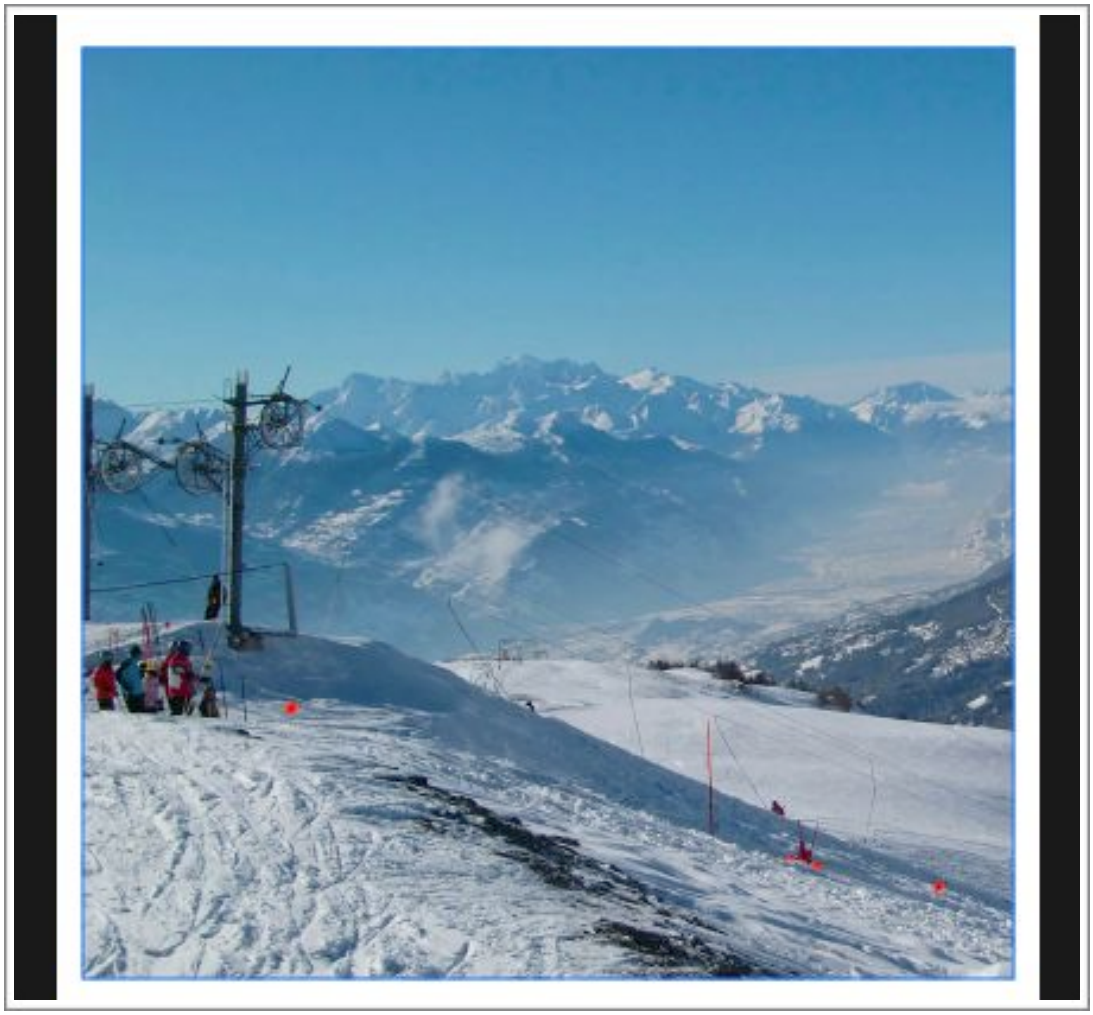
System icons in SF Symbols:

```
var body: some View {
  Image(systemName: "camera.rotate")
}
```

Each `piste` has an `imgName` property that corresponds to an image in the asset folder.

```
var imgName: String
```

Use the property to render your image and you should see something like this:



That's a bit too big, so let's give it some style.

In SwfitUI you style your views using **View Modifiers**. The name is slightly deceptive, because a modifier returns a new view all-together, rather than changing the original.

To make an image resizable, you apply `resizable()` which returns a new `Image`:

```
var body: some View {
    Image("example")
      .resizable()
}
```

To give it a specific frame, you apply `frame()` which returns a `View`:

```
var body: some View {
  Image(“example")
    .frame(width: 50, height: 50)
}
```

And to round the edges, you apply `cornerRadius()` which returns a `View`:

```
var body: some View {
  Image(“example")
    .cornerRadius(10)
}
```

You can apply multiple modifiers in sequence, and each one applies to the `View` that was returned by the previous one.

```
var body: some View {
  Text("Swift Alps")
    .padding(3)
    .background(Color.purple)
    .cornerRadius(5)
    .padding(5)
    .background(Color.blue)
}
```

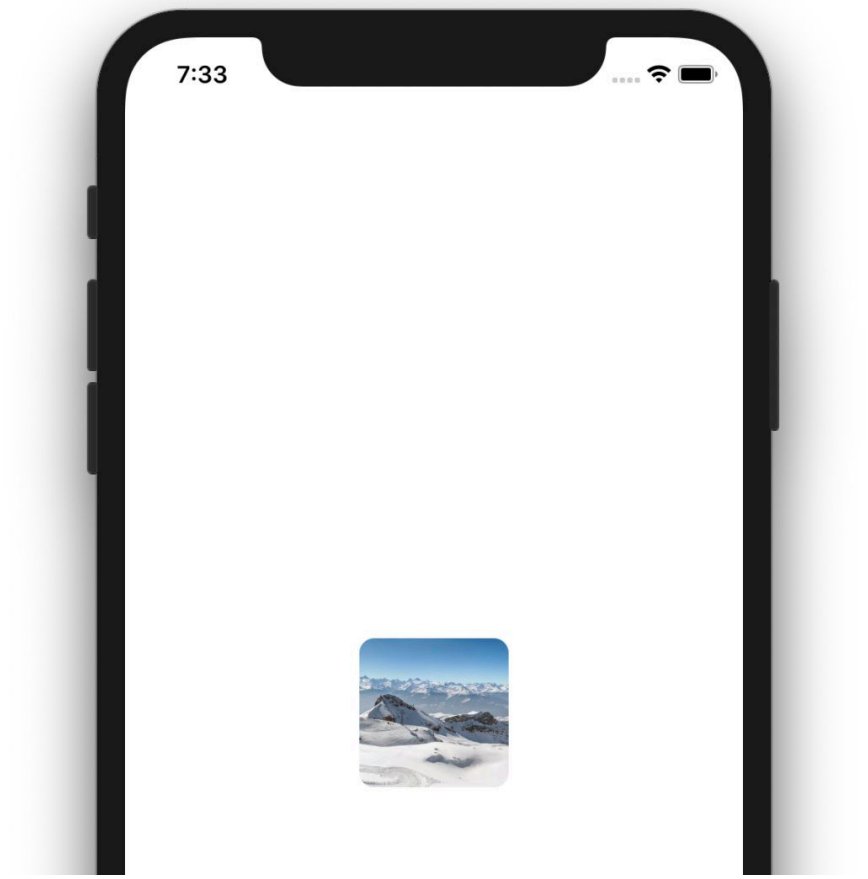
For example, if we use multiple `padding()` and `background()` modifiers in sequence, we can achieve this bordered effect:


Not all views can be modified in the same way. For example, you can’t apply `resizable()` to `Text` nor to a `View`. So trying to first give an `Image` a frame and then making it resizable wouldn’t work, because `frame()` returns a `View`, and `resizable()` only works on an `Image`.

```
var body: some View {
    Image("example")
        .frame(width: 50, height: 50)
        .resizable()
}
```

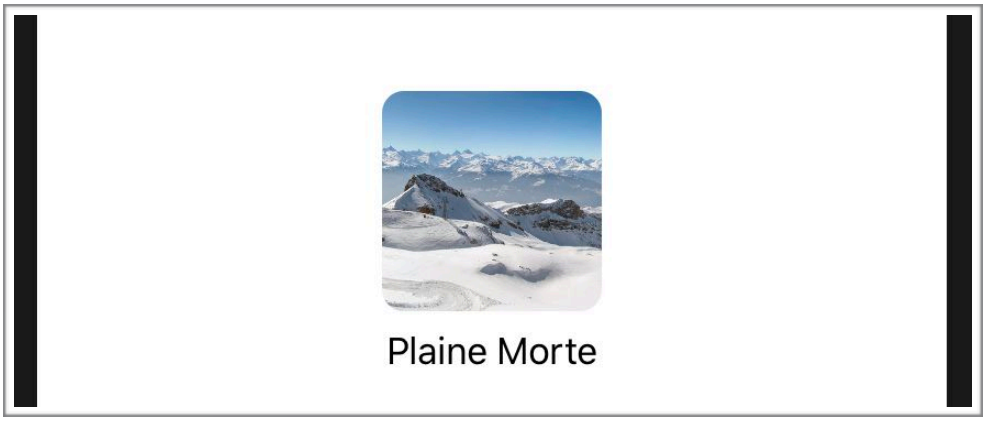
Value of type 'some View' has no member 'resizable'

**To Do:** Show the image of the piste and your app should look similar to:



# Task 3: VStack

Combine an `Image` and `Text` in a `VStack`.




When you want to show multiple views you can't just add them together like this:

```
var body: some View {
  Text("Swift")
  Text("Alps!")
}
```

The above statement makes two individual views, but `body` expects only one. So, SwiftUI expects us to group them together.

Similar to `UIStackViews` SwiftUI provides a handy way to handle putting multiple views in vertical, horizontal and overlayed groups.

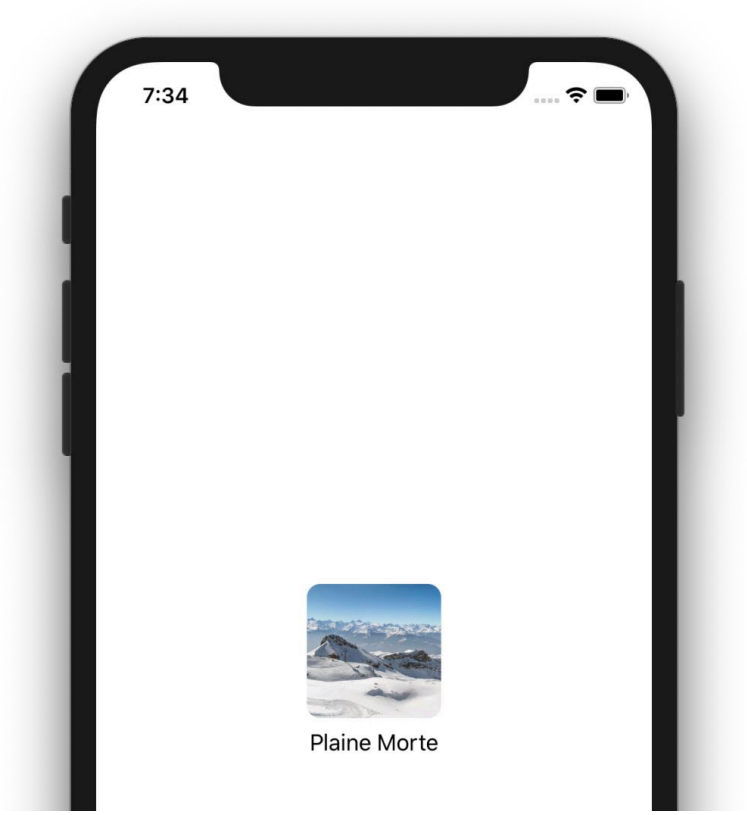
A `VStack` stacks views vertically, an `HStack` stacks them horizontally and a `ZStack` puts them on top of each other.

```
var body: some View {
  VStack {
    Text("Greetings")
      .background(Color.red)
    Text("Swift")
      .background(Color.blue)
    Text("Alps!")
      .background(Color.yellow)
  }
}
```


By default, the views inside a `VStack` will be aligned to the center with 0 spacing between them. But that's easy enough to change by specifying the alignment and spacing:

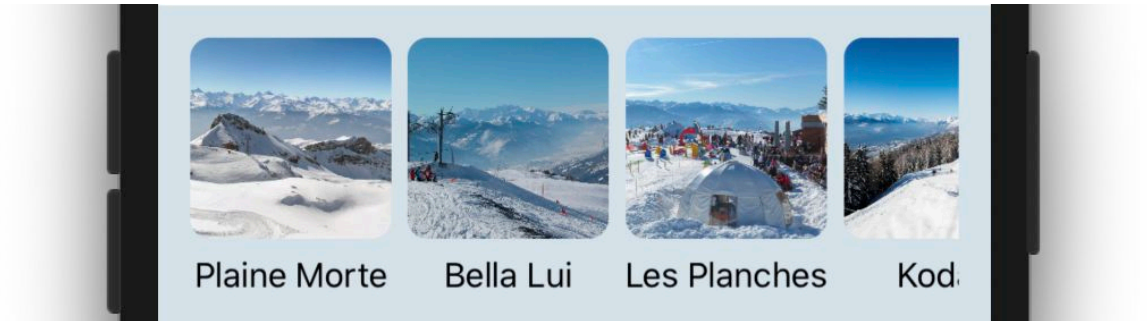
```
var body: some View {
  VStack(alignment: .leading, spacing: 5) {
    Text("Greetings")
      .background(Color.red)
    Text("Swift")
      .background(Color.blue)
    Text("Alps!")
      .background(Color.yellow)
  }
}
```

**To Do:** Combine the image and the name of the piste to make a view like this:



# Task 4: Reusable

## Create a new file for a reusable version of PisteView and display all 5 in a row.




Since the point of this view is to be used to display all the pistes, we really shouldn't have to write it every single time.

At the same time, writing SwiftUI tends to get verbose quite quickly and ends up looking a bit like spaghetti code. So it's important that you start chunking your code up early on.

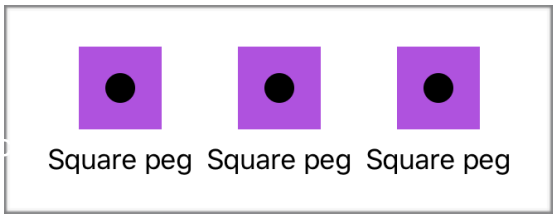
There are a few ways to make a SwiftUI View reusable.

You can extract it to a computed variable returning `some View`:

```
var squarePeg: some View {
  VStack {
    Circle()
      .padding()
      .background(Color.purple)
      .frame(width: 50, height: 50)

    Text("Square peg")
  }
}
```

And stack them together with `HStack`:

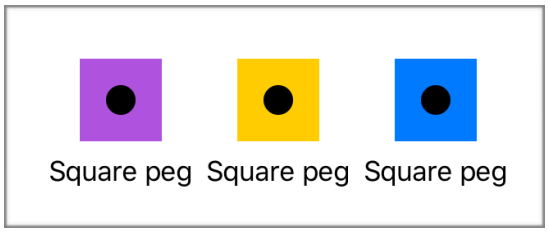
```
var body: some View {
  HStack {
    squarePeg
    squarePeg
    squarePeg
  }
}
```




Or abstract it to a function to change some parameters, like the color:

```
func squarePeg(color: Color) -> some View {
  VStack {
    Circle()
      .padding()
        .background(color)
      .frame(width: 50, height: 50)

    Text("Square peg")
  }
}
```

```
var body: some View {
  HStack {
    squarePeg(color: .purple)
    squarePeg(color: .yellow)
    squarePeg(color: .blue)
  }
}
```

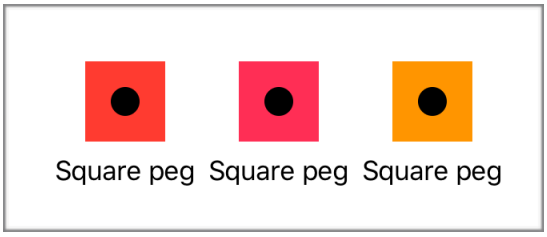



Or you could create a new SwiftUI file all-together by going through **File > New > File > User Interface > SwiftUI > SquarePeg**

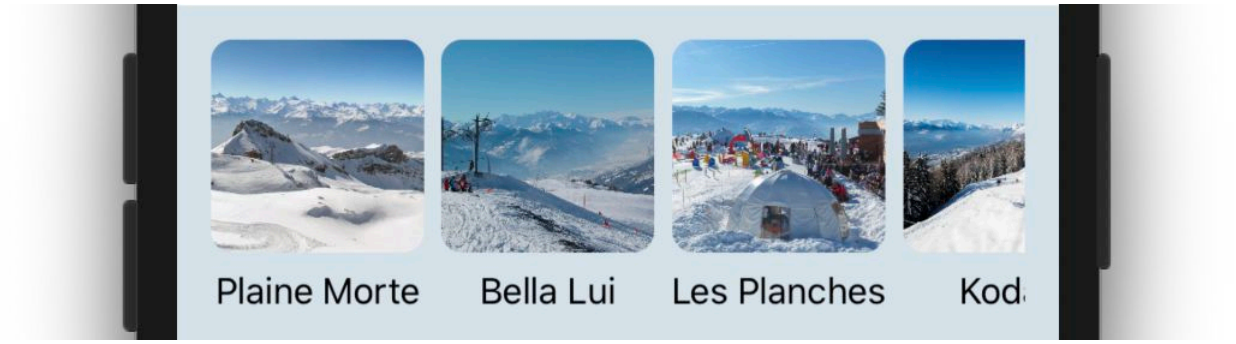
```
struct SquarePeg: View {
  var color: Color

  var body: some View {
```

```
    VStack {
      Circle()
        .padding()
        .background(color)
        .frame(width: 50, height: 50)

      Text("Square peg")
    }
  }
}
```

```
var body: some View {
  HStack {
    SquarePeg(color: .red)
    SquarePeg(color: .pink)
    SquarePeg(color: .orange)
  }
}
```

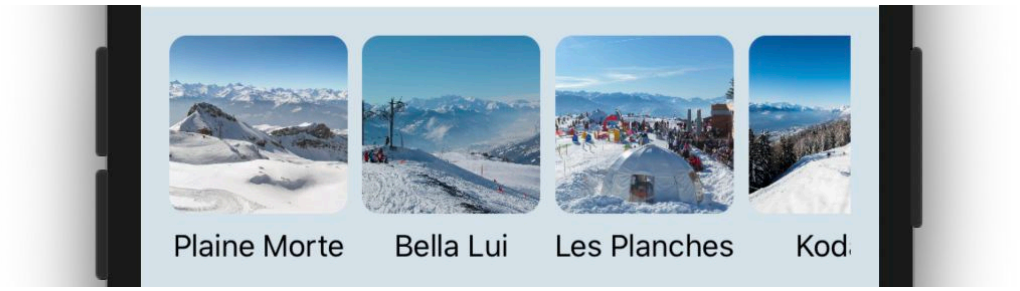



**To Do:** Pick an option to abstract the piste view and render all 5 in a row.



# Task 5: ScrollView

Put the reusable views in a `ScrollView`.

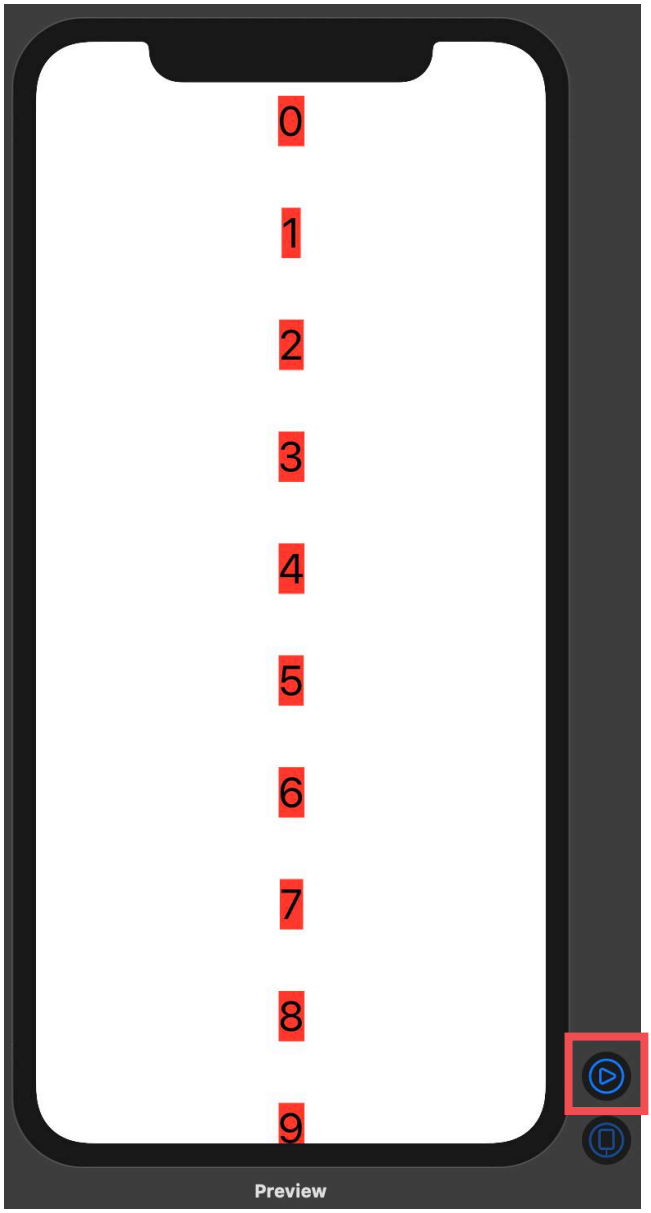



Compared to UIKit, making scrollable content is a breeze with SwiftUI because it automatically sizes itself around the scrollable content.

To make a scrollable list of 10 numbers we can use a `ScrollView` Like this:

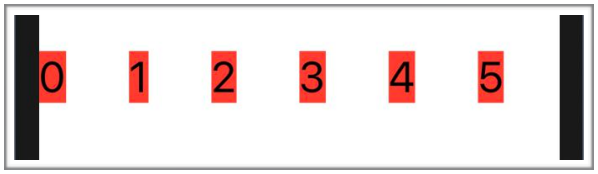
```
var body: some View {
  ScrollView {
    VStack(spacing: 50) {
      ForEach(0..<10) {
        Text("\($0)")
          .font(.largeTitle)
            .background(Color.red)
      }
    }
  }
}
```

And to make sure the scrolling actually works you can press the play button in your Canvas to interact with it!



ScrollViews are vertical by default, but you can change that simply by making the axis `.horizontal` and using `HStack` instead of `VStack` to group the items.

```
var body: some View {
  ScrollView(.horizontal, showsIndicators: false) {
    HStack(spacing: 50) {
      ForEach(0..<10, id: \.self) { number in
        Text("\(number)")
          .font(.largeTitle)
          .background(Color.red)
      }
    }
  }
}
```




`ForEach` is used when you want to loop over a sequence to create your views. `ForEach` is itself a view, so you can return it on its own if you'd like, but it's most commonly used inside `ScrollViews` and `Lists`.

For `ForEach` to work, it has to uniquely identify every item in the sequence you're displaying so that it can correctly handle adding or removing items.

For primitive types like `Int` and `String`, that's done implicitly using `self`, so instead of writing this:

```
ForEach(0..<10, id: \.self) { number in
  Text("\(number)")
}
```

You can shorten it to:

```
  ForEach(0..<10) {
    Text("\($0)")
  }
```

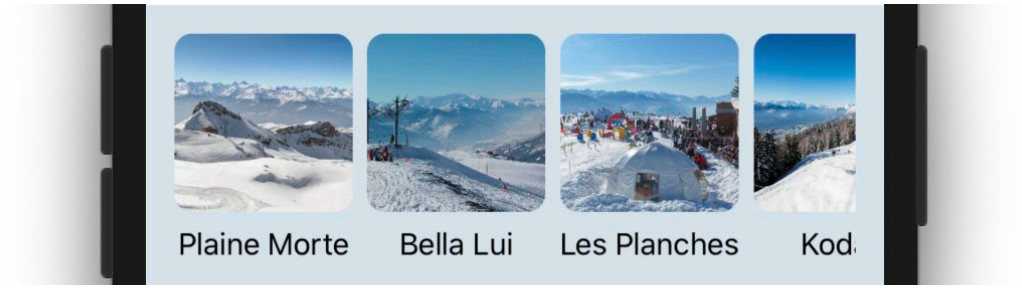
The `ForEach` allows you to give it a range or a sequence of data. To loop over a sequence of custom objects we need to conform to the `Identifiable` protocol and give our objects a unique `id` property.

For example:

```
struct Flower: Identifiable {
  var id: Int
  var name: String
}

struct FlowerView: View {
  var flowers: [Flower]

  var body: some View {
    ForEach(flowers, id: \.id) { flower in
      Text(flower.name)
    }
  }
}
```

Luckily the `Piste` struct already conforms to `Identifiable` and gets its `id` from **`slopesData.json`**.

**To Do:** Create a horizontally scrolling list of all the pistes like this:



**Tip:** Add some padding to give your new view some breathing room. And if you want to use the same background color as the example, use:

```
Color(“lightBG")
```

# Task 6: Handling Events

Print the piste name on a tap event.


Many things can happen to a view. It can appear, disappear, get tapped on or deleted. SwiftUI lets views respond to events through view modifiers - basically, everything that happens to a view happens through a modifier.

Adding an `onTapGesture()` modifier allows the view to receive tap events and respond to them in a closure:

```
var body: some View {
  Text("Tap Me!")
    .onTapGesture {
      print("Tapped")
  }
}
```

You can also specify how many times to tap on it before the closure gets executed:

```
var body: some View {
  Text("Tap Me 5 times!")
    .onTapGesture(count: 5) {
      print("Hooray!")
  }
}
```

Events you can automatically attach to your views will generally start with `on`.

Now print the name of a piste when you tap on it to the debugger.

```
Bella Lui
Les Planches
```

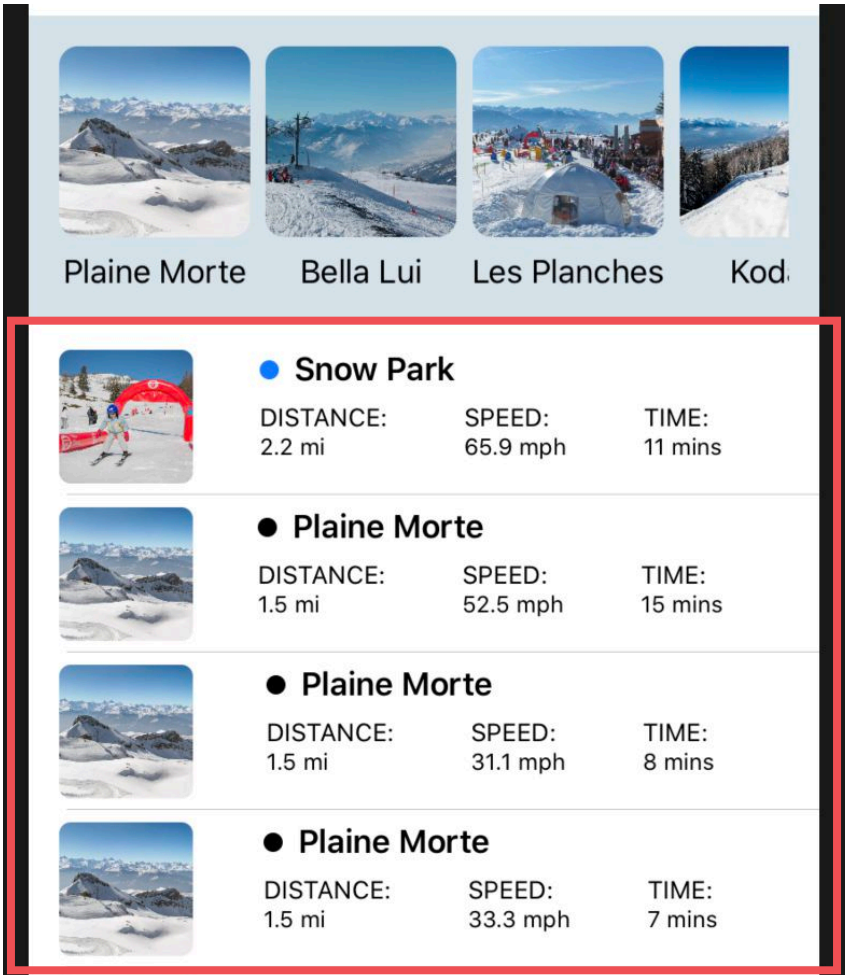
**Tip:** You'll have to actually run the app, because you can't communicate with the debugger through the Canvas preview.

# 2. Piste Log

Hooray! Great job on tackling that scrolling piste picker. Take a moment to appreciate how short the code to accomplish that actually is and think how much longer it would take to achieve that with UIKit (:

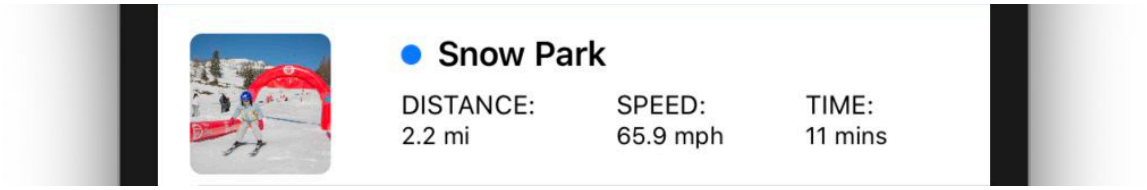
**Goal:** Next up, we have the piste log. In this section you'll use the piste picker control you just made to add individual runs to your list.

**Learning:** In this section, you'll how practice creating complex views, learn how to use Lists and how to update views using `@State`.



# Task 7: Cells

Create a LogView, the view for a single log.




Before we can make a list of logs, we first have to make a visual representation of a single log. Think of this as the `UITableViewCell` equivalent to the `UITableView`, except without all the horrors like typed identifiers, registering and delegation 😅.

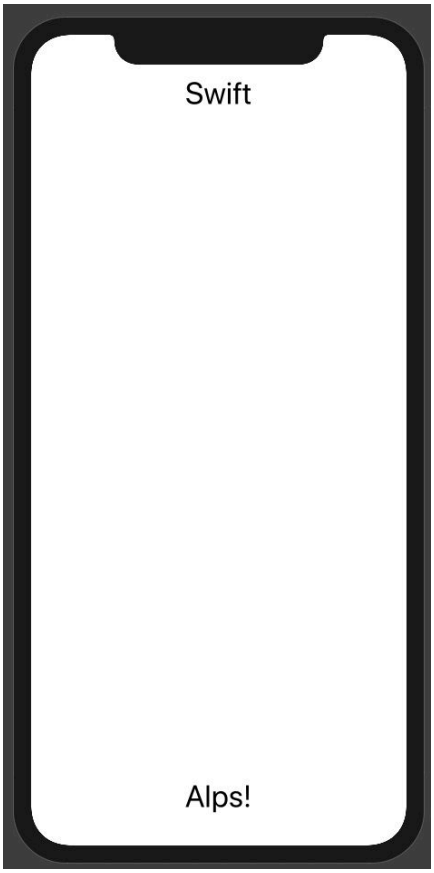
At this point, you know almost everything you need to know to render the above view, except for one important bit of layout magic; the `Spacer()`.

The spacer is a view that does exactly what it says. It adds space. In fact, it adds **as much space as it can** between views on the specified axis.

If it's contained in a `VStack`, it adds vertical space, and if it's contained in an `HStack`, it adds horizontal space.

This puts two text views on the top and bottom of the `VStack`:

```
var body: some View {
  VStack {
    Text(“Swift")

    Spacer()

    Text("Alps!")
  }
}
```




This puts two text views in the middle of the `VStack`:

```
var body: some View {
  VStack {
    Spacer()

    Text("Swift")
    Text(“Alps!")

    Spacer()
  }
}
```

To get all the necessary data to render the cell view, use the Log object located in `Log.swift` that also has a static property `random` for you to generate a random log:

```
  static var random: Log {
    let piste = Piste.random
    let run = Run.random
    return Log(piste: piste, run: run)
  }
```

**To Do:** Create the cell view:


**Tip:** I recommend you create a new SwiftUI file for this view.

**Tip:** In order to initialise a View struct with data, you declare the properties it needs at the top of the file.

```
struct SquarePeg: View {
  var color: Color

  var body: some View {
    // use color inside
  }
}

struct ContentView: View {

  var body: some View {
    SquarePeg(color: .red)
  }
}
```

**Tip:** To create a circle, you can use the `Circle()` `Shape`.

**Tip:** To nicely render the time in minutes you can use `timeFromSeconds` on the **`Run.swift`**'s `time` property.

**Tip:** Use this to format Strings to specific decimal points:

```
String(format: "%.2f mph", 34.23244)
```

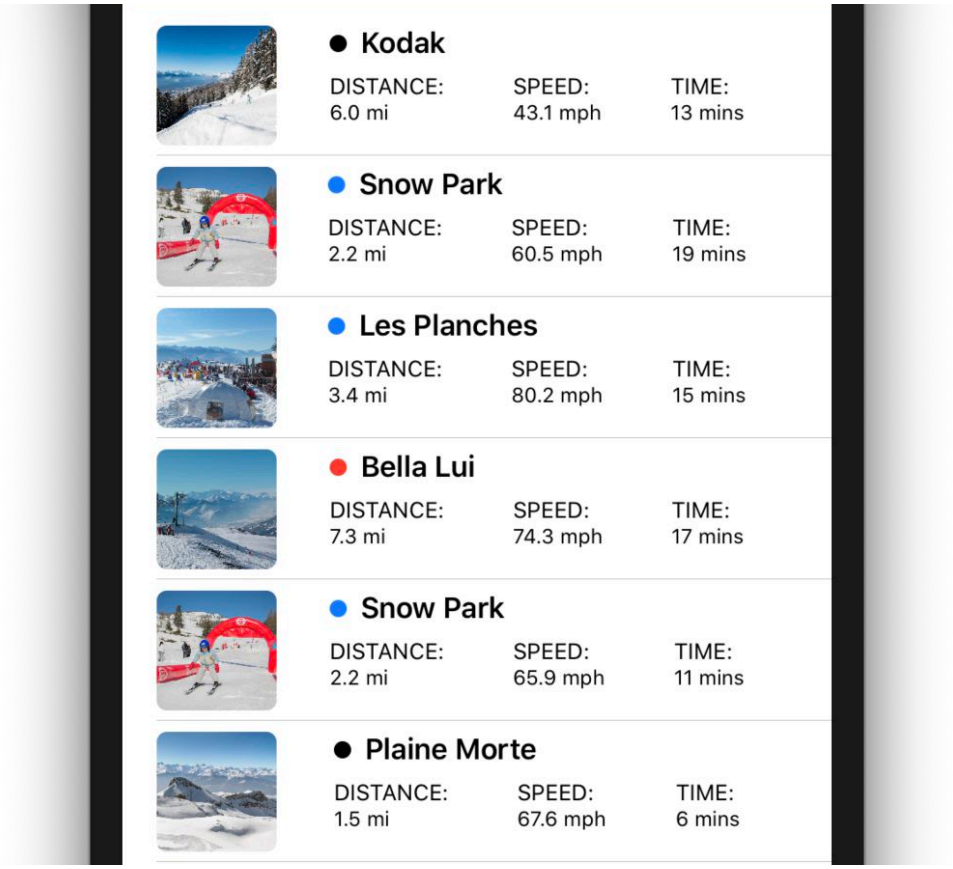
Output: `34.23`

**Tip:** To change the font sizes, you can use Apple's predefined values with the `font()` modifier:

```
.font(.headline)
.font(.footnote)
```

# Task 8: Lists

Create a list of logs.




Creating `UITableViews` and `UITableViewCells` is the bread and butter of any iOS developer. But it's often a pain to setup. Luckily, SwiftUI makes the process extremely painless!

You're already familiar with `ScrollView`. The principles and syntax of creating a `List` are very similar.

Creating a `ScrollView`:

```
var body: some View {
  ScrollView {
    VStack(spacing: 50) {
      ForEach(0..<10) {
        Text("\($0)")
      }
    }
  }
}
```

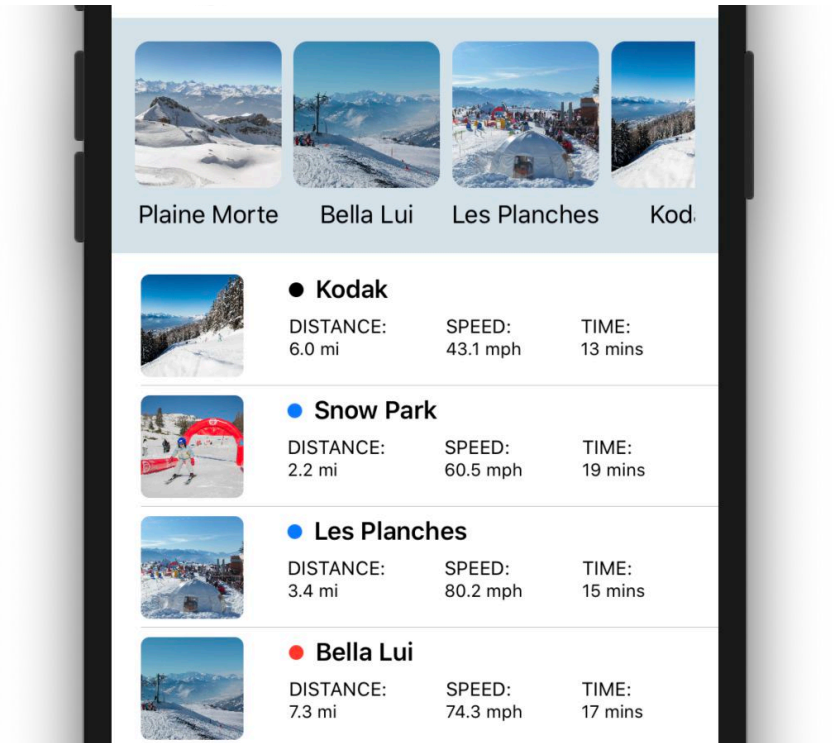
Creating a `List`:

```
var body: some View {
  List {
    ForEach(0..<10) {
      Text("\($0)")
    }
  }
}
```

But there are a few key differences to mention:

1. A `List` can only show vertical content, which is why we had to use the `ScrollView` to create a horizontally scrolling view.
2. List uses reusable `UITableViewCells` and `UITableViews` for its underlying structure, and the `ScrollView` uses the stacks themselves. This makes using `List` much more performant for large data sets.
3. On top of that, `List` comes built in with all of the affordances (annoyances?) that a `UITableView` does, like automatically showing separators.

**To Do:** Create a list of logs and vertically stack it after the piste picker:




**Tip:** To get a list of random logs to display use the `randomLogs` property in `Log.swift`:

```
static var randomLogs: [Log] {
  return (0..<10).map { _ in Log.random }
}
```

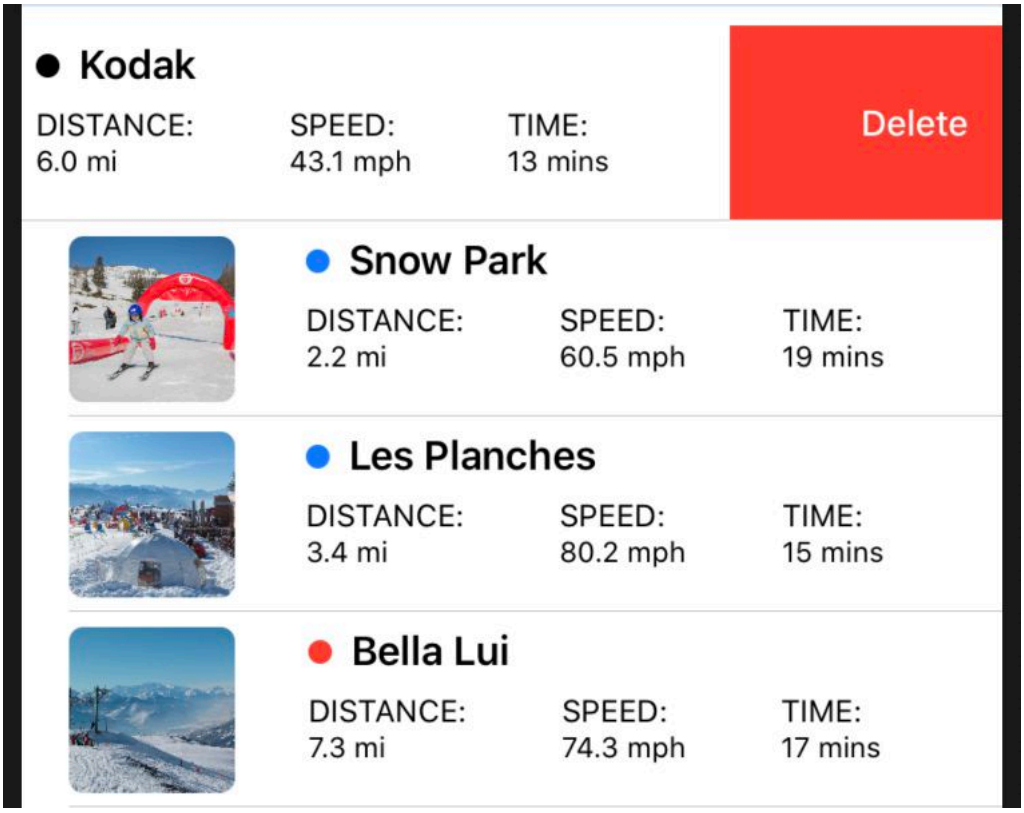
You can use it before the `body` in `LogbookView.swift`:

```
var logs: [Log] = Log.randomLogs
```

**Tip:** Using `ForEach` to loop over a sequence of objects requires those objects to conform to a unique id. It's already implemented for the `Log` struct, you just have to use it.

# Task 9: @State

Allow the user to add or remove logs by managing view updates through @State.




SwiftUI is a **declarative** UI framework. That means that **the view is a function of state**, not a sequence of events. So instead of directly manipulating individual views on events, the views respond to changes of state. Every time the state changes, it triggers a re-render of the entire view.

The state is generally described with the properties that we initialise `Views` with. But, since `Views` are structs, we can't directly manipulate any of their properties.

For example, trying to change the `random` property inside the body throws an error:

```
struct ContentView: View {
  var random = 0

  var body: some View {
    Text("Random number: \(random)")
      .onTapGesture {
        self.random = Int.random(in: 0...10)
    }
  }
}
```

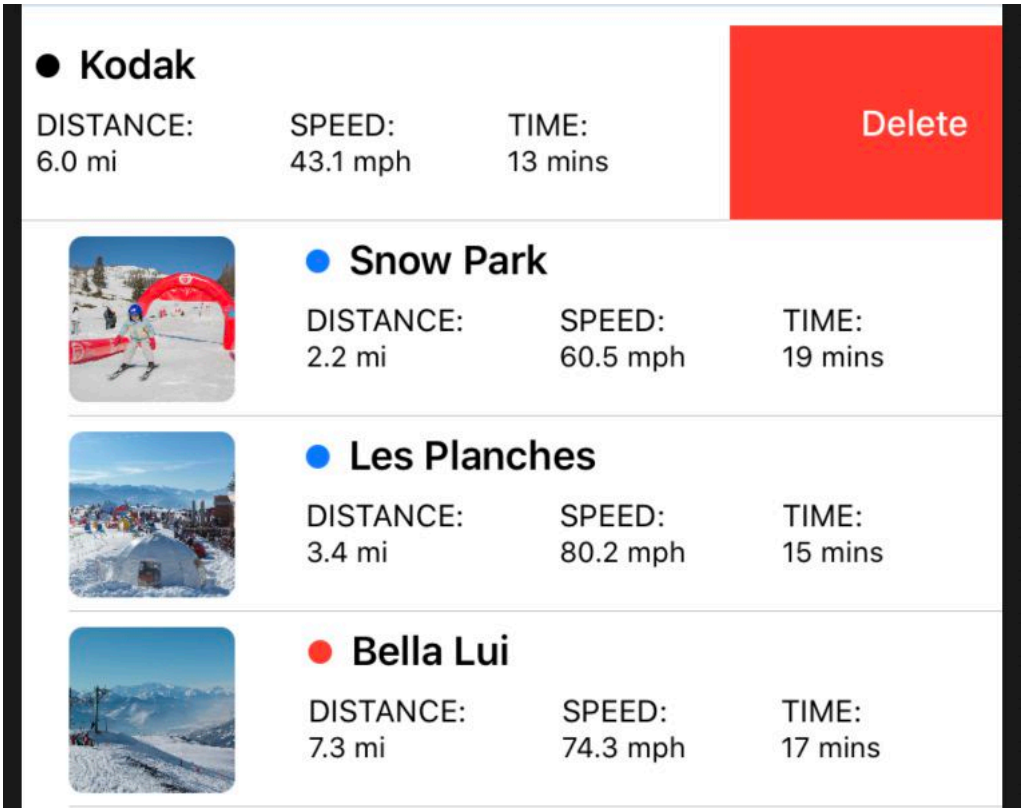

The way to propagate changes to the view is by adding a property wrapper `@State` in front of the property:

```
struct LogbookView: View {
  @State var random = 0

  var body: some View {
    Text("Random number: \(random)")
      .onTapGesture {
        self.random = Int.random(in: 0...10)
    }
  }
}
```

If an object marked with `@State` is used to render the UI, any changes to it will trigger a re-render of the entire view.

So, in order to allow the user to add and remove logs we need to manage the state of `LogbookView`. Specifically the state of the array of data holding the logs.

**To Do:** Use `@State` to allow the user to add logs by tapping a specific piste in the piste picker, and remove them by swiping a log cell away.




**Tip:** To delete a cell from the List, use the `onDelete(perform:)` view modifier on `ForEach`.

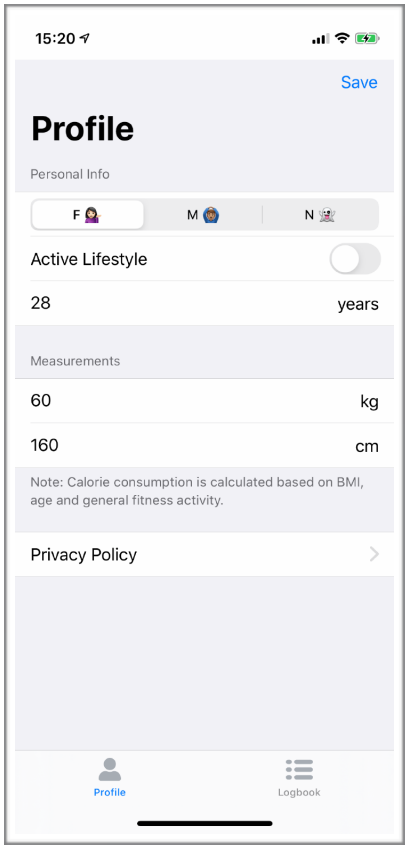
**Tip:** Rather than inserting a new log at the end of the list, insert it at the beginning so that you can see the difference better.

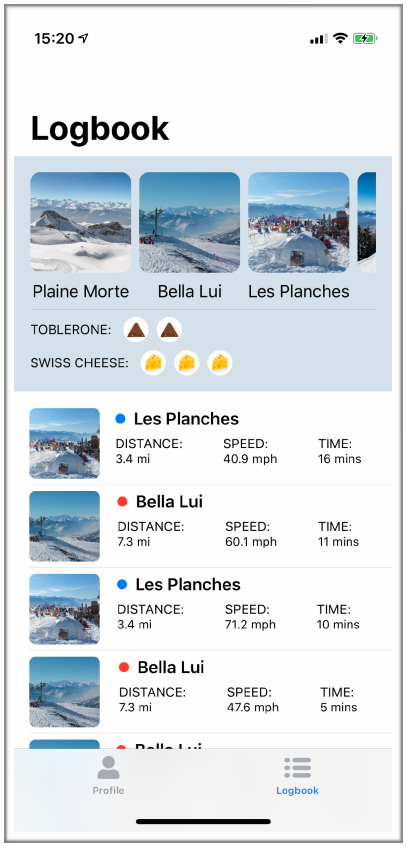
# 3: Tabbed Navigation

Great job on finishing the list!

**Goal:** In this section, we'll add a profile view embedded in a tabbed navigation so that the user can enter their details. We'll then save and use those details to show how many Toblerones and Swiss Cheese they can eat for that day to encourage them to ski more!

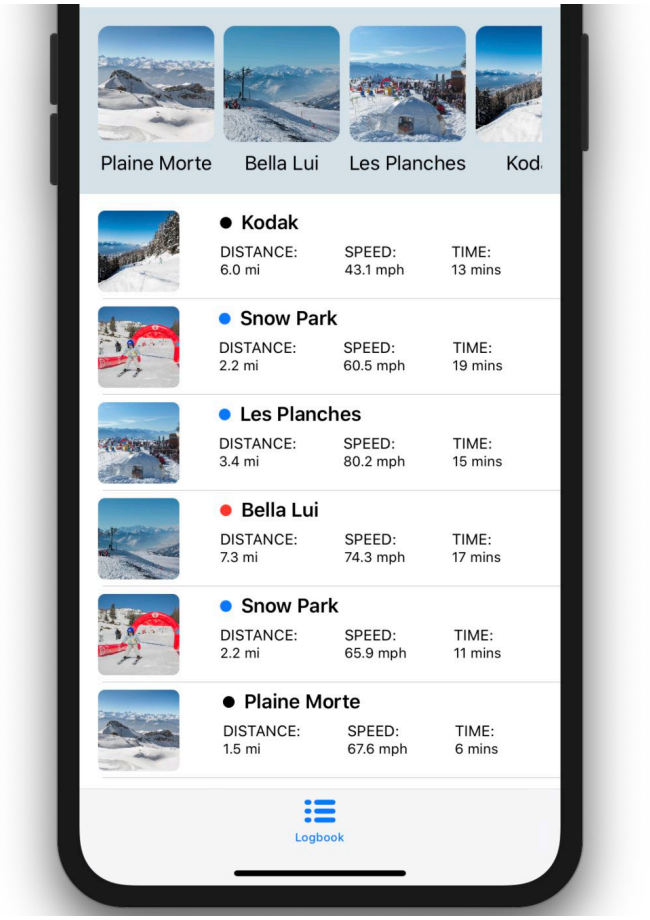
**Learning:** In this section, you'll learn how to create implement a tabbed navigation and use navigation views.



# Task 10: Tabbed Navigation

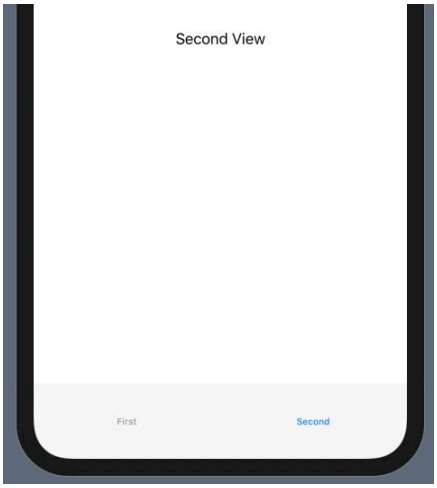
Embed LogbookView in a Tabbed Navigation.




To implement a tabbed navigation, you use the `TabView`.

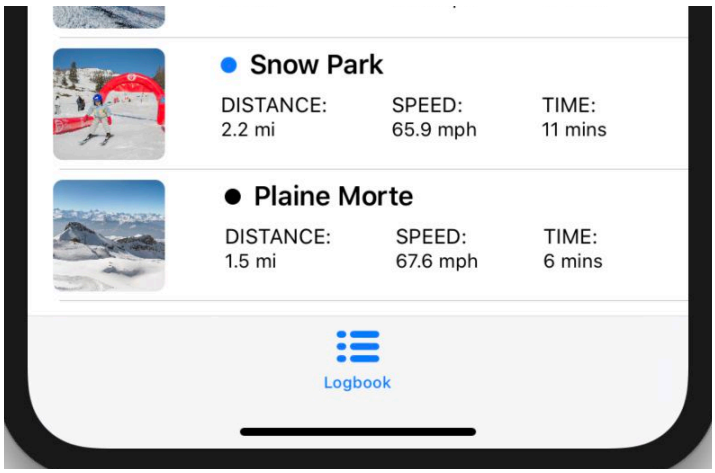
We can initialize a `TabView` by giving it a list of items we want to show for each tab and applying a `tabItem` modifier which controls what gets shown on the tab bar itself:

```
struct ContentView: View {
  var body: some View {
    TabView {
      Text("First View")
        .tabItem {
          Text("First")
      }
      Text("Second View")
        .tabItem {
          Text("Second")
      }
    }
  }
}
```




The `.tabItem()` modifier allows you to define how a tab bar item looks with one or two views. These can only ever be `Image` and `Text`, regardless of the order. In the example above, we're only using `Text`.

**To Do:** Embed the `LogbookView` inside a `TabView` so that the final result looks like this:




**Tip:** Create a new file for the Tab View, so that it's easier to manage.

**Tip:** Update the root view to render the new file as the root in the `SceneDelegate.swift`.

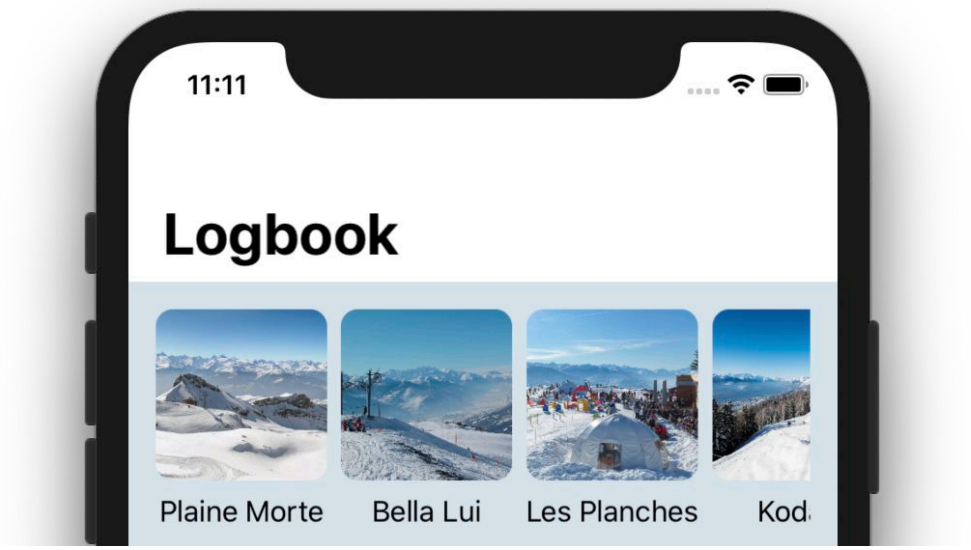
**Tip:** There's an image called "list" in the asset folder for you to use.

**Tip:** Don't worry if the Image doesn't turn blue. It's a bug. You can fix it by changing the `Image` property to a `Button` one:

```
Button(action: {
  // Nothing
}) {
  Image("list")
}
```

# Task 11: Navigation Title

Add a Navigation Title to the LogbookView.




Implementing a clear navigation pattern can be a bit confusing with SwiftUI. You can add navigation titles and properties to views, but they'll only render once that particular view is wrapped inside a `NavigationView`.

Adding a navigation title to a `View` is simple:

```
.navigationBarTitle(Text("Title"))
```

If the content of the view is scrollable and extends beyond the bounds of the view, the title will shrink automatically.

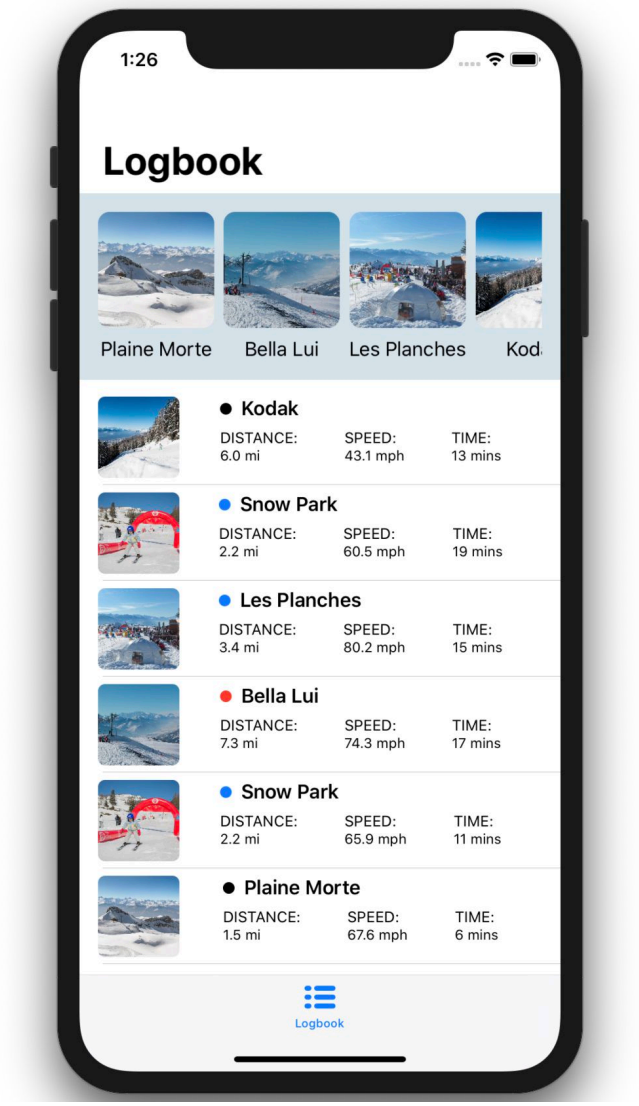
Wrapping a `View` inside a `NavigationView` is equally simple:

```
NavigationView {
  Text("My View")
}
```

And here's how you could put them together:

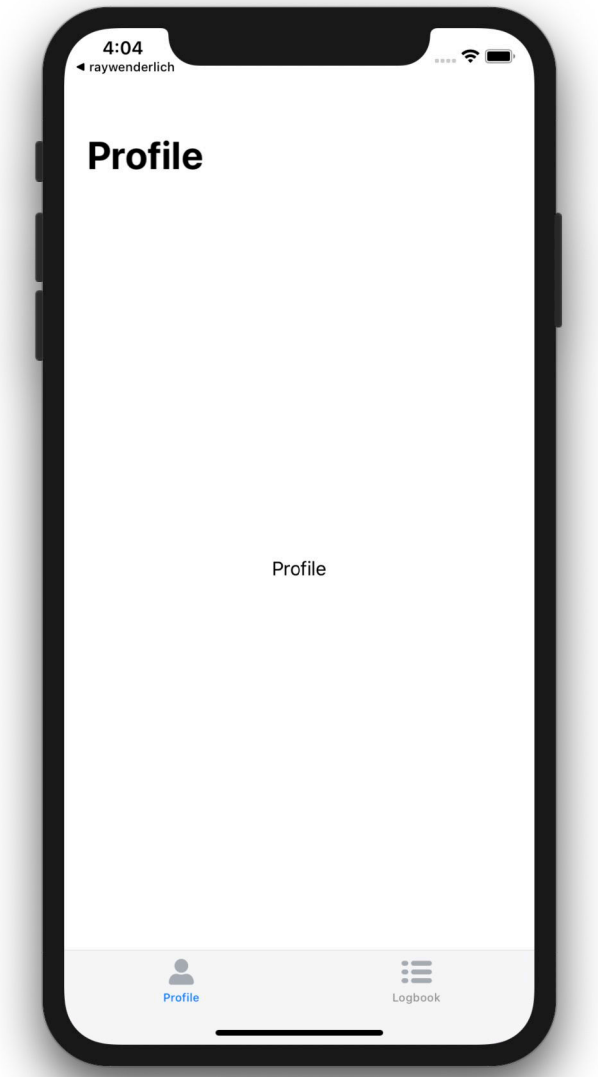
```
NavigationView {
  Text("First")
  .navigationBarTitle("First Title")
}
```

**To Do:** add a navigation title to `LogbookView` and add it to the `TabView` wrapped in a `NavigationView`. Your final result should look like this:



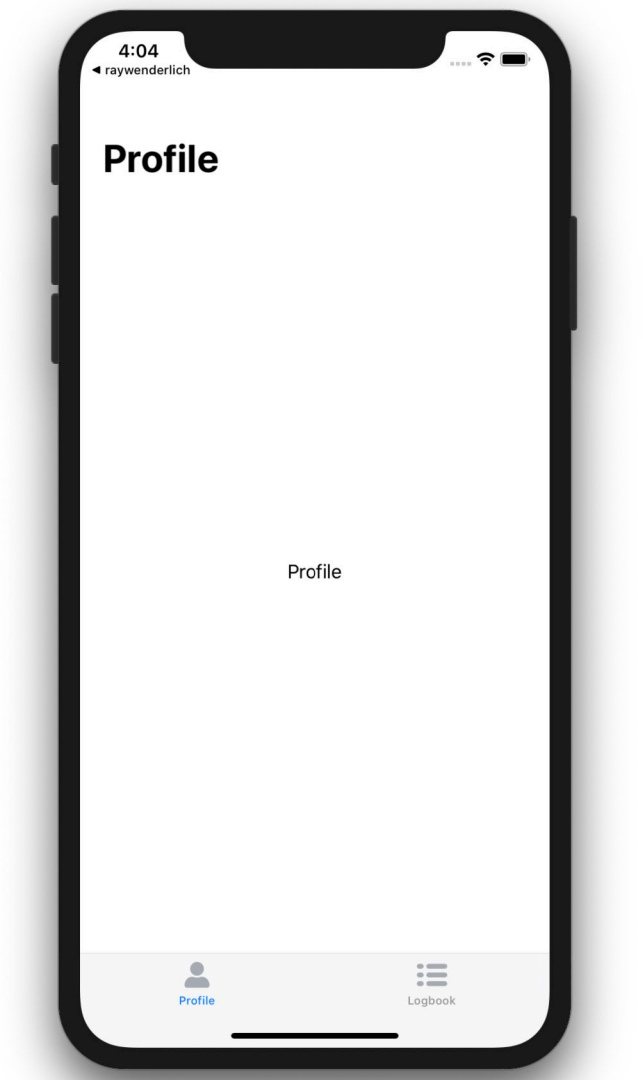
# Task 12: Profile View

Add a ProfileView to the tab bar.




The point of having a tabbed navigation is to have more than one option to choose from.

**To Do:** Add a skeleton profile view to the tab bar with a navigation title and make it look like this:




**Tip:** Add a new file for the profile view, so that you can work with it later.

**Tip:** Use the "profile" icon for the image.

# 4: Managing Data

One of the key differences between UIKit and SwiftUI development is the data flow. Data is king in SwiftUI and being able to properly manage data or state is essential in keeping you sane.

In SwiftUI all the Views are structs. This means that views are value types, rather than reference types. Therefore, none of the “normal” properties a View has are mutable.

But obviously, we **have** to change the views at some points. Which means we have to change the state and we can do with SwiftUIs new property wrappers `@State`, `@ObservedObject` and `@EnvironmentObject`.

**Goal:** In this section, you’ll implement the Profile View, store the user’s details in UserDefaults and finally display all the treats you can eat!

**Learning:** You’ll learn how to work with `@State`, `@ObservedObject` and `@EnvironmentObject,` how to logically separate the UI layer from the data layer in a SwiftUI-y way.

# Task 13: App State

## Convert AppState to an `ObservableObject` and add it as an `@EnvironmentObject` to the root view to share the `user` property between multiple views.

One of the ways you can share data between views is to use environment objects. They're a bit like singletons. You create them once and then share them across multiple views.

You can make an environment object out of any class by:

1. Conforming to the `ObservableObject` protocol
2. Marking properties with `@Published` to broadcast changes

For example, we can take this class:

```
class State {
  var user: User
}
```

And turn it into an environment object like this:

```
class State: ObservableObject {
  @Published var user: User
}
```

Now, whenever the `user` property changes, all the views that are using the property will be notified and get re-rendered.

To add an environment object to a view you pass it through when you initialise it:

```
let state = State()
```

```
let contentView = ContentView().environmentObject(State)
```

And to make it accessible in your view you add:

```
struct ContentView: View {
  @EnvironmentObject private var state: State

  var body: some View {
  }
}
```

Generally the Top Level View is the one that should be initialised with the environment object(s), because it's accessible to that Parent View and all of its Children.

Tasty Slopes contains a class called **`AppState.swift`** that has a `user` property and a few helper methods to save and update the user object when they fill out their profile or log ski runs.

We want to access the `user` property from both the profile and the Logbook View, so it makes sense to add it to their Parent View, the Tab Bar View.

**To Do:** Convert **`AppState.swift`** so that it can be used as an environment object and pass it to your Tab Bar View.

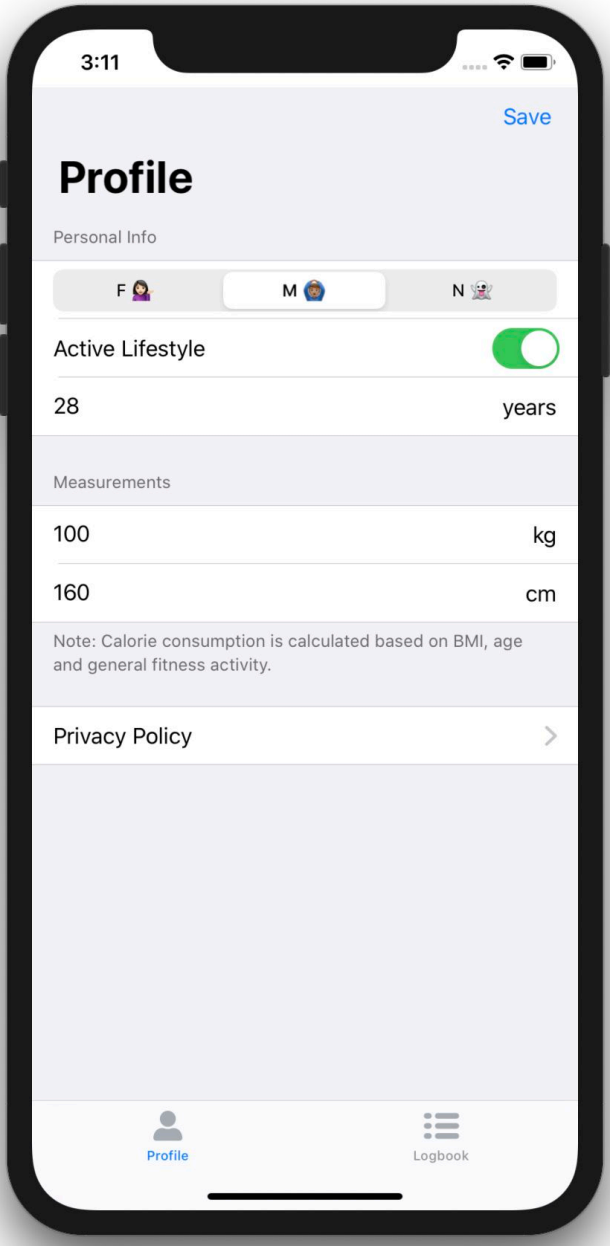
**Tip:** Remember that you're initialising your Tab View in the **`SceneDelegate.swift`**

**Tip:** In order to conform to `ObservableObject` the class has to have two imports:

```
import SwiftUI
import Combine
```

# Task 14: Forms and Sections

Create a Profile View the user can fill out using Form and Section Views.

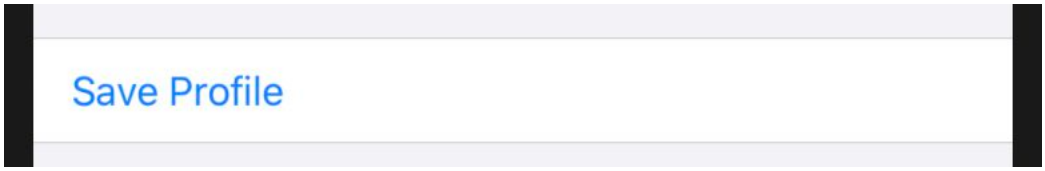


Static table views used to give me nightmares. But creating a profile page with SwiftUI is pretty straightforward by using the `Form` and `Section` Views.

Similar to `VStack`, `Form` is a grouping container, so you can pass multiple Views to it. But a `Form` is special, because SwiftUI automatically renders it for the platform. So in iOS a `Form` appears as a grouped table view.

Do this to create a form with a button in a cell:

```
var body: some View {
  Form {
    Button(action: {
      // Save profile
    }) {
      Text("Save Profile")
    }
  }
}
```




Forms are mainly used to let users enter data in digestible chunks by using controls like Pickers, Textfields and Toggles.

To create a `Toggle` to turn notifications on or off you can write something like this:

```
struct SettingsView: View {
  @State private var notificationsOn = false

  var body: some View {
    Form {
      Toggle(isOn: $notificationsOn) {
        Text("Notifications on")
      }
    }
  }
}
```

The core nature of the `Toggle` control is that it changes its state back and forth between on and off. In order to get the view to change, we store the state in a `@State` property wrapper.

You're probably wondering what the `$` is all about. You've used `@State` before to trigger a view re-render, but it didn't need a `$`:

```
struct ContentView: View {
  @State var random = 0

  var body: some View {
    Text("Random number: \(random)")
      .onTapGesture {
        self.random = Int.random(in: 0...10)
    }
  }
}
```

That's because `Text` only needs to read the value of `random`. `Toggle` on the other hand needs to both read and write the value.

In order to do that we have to mark the property with `$` to create a two-way binding.

Most Views that you'll end up using when building Forms use a two-way binding.

For example, to create a Segmented Control that lets you choose between two Swiss treats you create a `Picker` like this where `selectedTreat` is used as a two-way binding:

```
struct ContentView: View {
  @State private var selectedTreat = 0
  private var treats = ["Toblerone", "Cheese"]

  var body: some View {
    Form {
      Picker(selection: $selectedTreat,
             label: Text("Select a treat")) {
        ForEach(0 ..< treats.count) {
          Text(self.treats[$0])
        }
      }
      .pickerStyle(SegmentedPickerStyle())
    }
  }
}
```

You create a `Textfield` in a similar way and bind the `name` property:

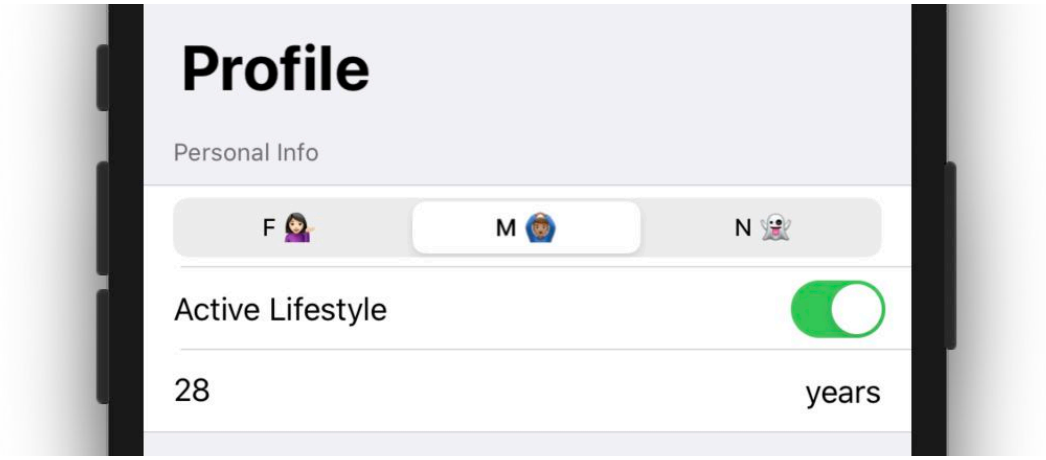
```
  @State private var name: String = ""

  var body: some View {
    Form {
      TextField("Name", text: $name)
   }
  }
```

The first parameter for the `TextField` is the placeholder.

You can use `Section` to visually group Form elements together and give them a Header or Footer:

```
struct ProfileView: View {
  @State private var name: String = ""
  @State private var age: String = ""

  var body: some View {
    Form {
      Section(header: Text("Personal Info")) {
        TextField("Name", text: $name)
        TextField("Age", text: $name)
      }
    }
  }
}
```

**To Do:** Start the profile Form by creating the Personal Info Section:




**Footer Text:** "Note: Calorie consumption is calculated based on BMI, age and general fitness activity."

# Task 15: @ObservedObject

## Create a new class to manage the ProfileView's data by using `@ObservedObject`.

Task 15 was probably a little tricky. You can now see how quickly SwiftUI Views can get a bit unruly with managing only three different data points.

So far you've seen `@State` and `@EnvironmentObject` in action, but in times when you want to observe multiple properties or custom objects you should use `@ObservedObject`. I like to think of them as view models.

You can transform any class into an `@ObservedObject` and the setup is the same as for the `@EnvironmentObject:`

1. Conform to the `ObservableObject` protocol
2. Mark properties with `@Published` to broadcast changes

```
class State: ObservableObject {
  @Published var user: User
}
```

The only difference is in how we use it in the View itself and how we initialise Views with it:

Declaring an `@ObservedObject:`

```
@ObservedObject var viewModel: ViewModel
```

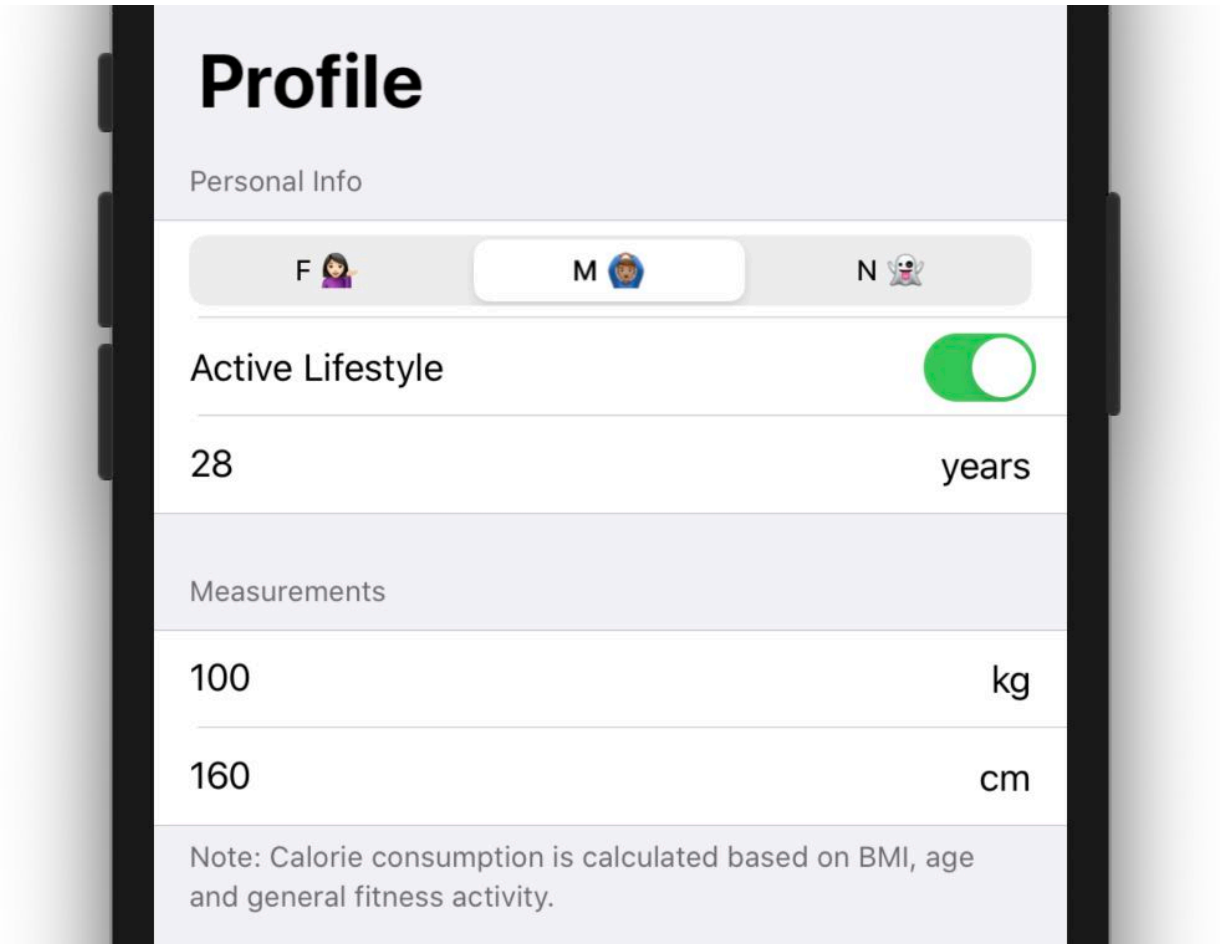
Initialising a view is the same as with any other property:

```
ContentView(viewModel: ViewModel())
```

There's already a class called `ProfileVM.swift` located in **TastySlopes > View Models** that's setup to be used with the Profile view.

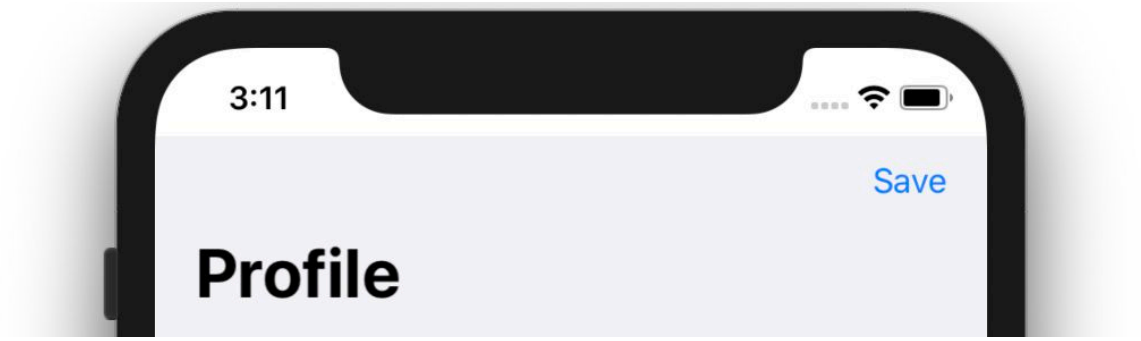
**To Do:**

1. Convert the ProfileVM to an Observable Object
2. Add it as a property on Profile View
3. Refactor the current Profile View setup with the `@Published` properties on ProfileVM
4. Implement the rest of this form



# Task 16: Navigation Buttons

Add a Save Button to the Navigation Bar to save the user's changes.




Once a user has filled out the profile, we need to give them a way to save it. In order to avoid keyboard issues covering up bits on pieces of the UI, the best way to put a Button is in the Navigation Bar.

This is how you create a `Button`:

```
Button(action: {
  self.updateProfile()
}) {
  Text("Save")
}
```

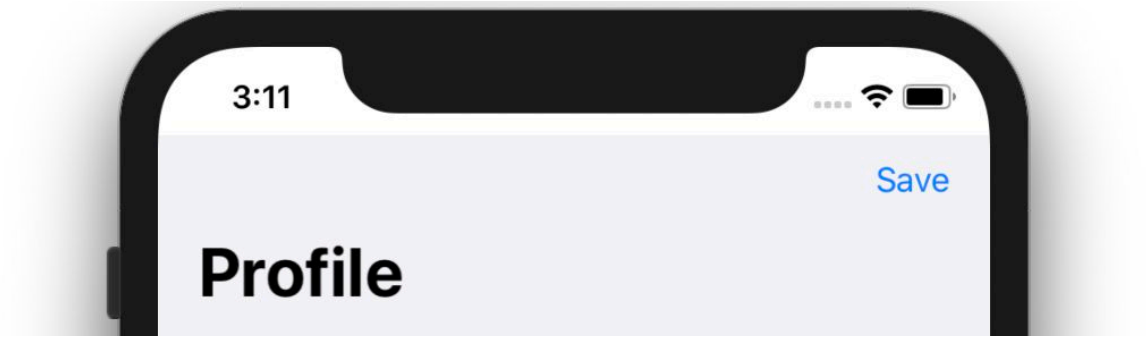
You give it an action in the action closure, and describe how it should look like underneath.

To add a Navigation Item to a View, you modify it with the `navigationBarItems()` modifier:

```
var body: some View {
  Text("Settings")
  .navigationBarItems(trailing:
    Button(action: {
      // Do cool things
    }) {
      Text("Save")
    }
  )
}
```

Note that the Navigation Button won't actually show unless you wrap the Parent View in a `NavigationView`.

**To Do:** Add a Save Navigation Bar Button to the Profile's Nav Bar to make it look like this:




**Tip:** The navigation bar items will only render when the parent view is embedded in a navigation view.

**Tip:** You can leave the Save function empty for now.

# Task 17: Refactor with AppState

## Refactor ProfileView with AppState to save user details.

In order to actually save the user details you'll have to manipulate the user property on the `AppState`. That means you'll have to access appState from the Profile View.

If a the parent View gets initialised with an `@EnvironmentObject` that means all its children Views get access to it. All we need to do is expose it on the child class itself.

```
let state = State()
let parentView = Parent().environmentObject(state)

struct ParentView: View {
  @EnvironmentObject var state: State

  var body: some View { }
}

struct ChildView: View {
  @EnvironmentObject var state: State

  var body: some View { }
}
```

**To Do:** Add `AppState` to Profile View and update the Save function on to actually store the user details.

**Tip:** Use this handy function to update the profile on the `AppState`.

```
private func updateProfile() {

UIApplication.shared.sendAction(#selector(UIResponder.resignFi
rstResponder), to: nil, from: nil, for: nil)

  guard let user = profileVM.updatedUser() else {
    return
  }

  appState.updateProfile(age: user.age,
                         weight: user.weight,
                         height: user.height,
                         gender: user.gender,
                         active: user.activeLifestyle)
}
```

**Note:** You HAVE to populate all the numeric value in order to save the user, otherwise we're not able to calculate the calories.

**Note:** Your `appState` or `profileVM` properties might be called differently.

# Task 18: Refactor with AppState #2

Refactor LogbookView with AppState to save user logs.

```
private func updateProfile() {

UIApplication.shared.sendAction(#selector(UIResponder.resignFirstResponder), to: nil, from: nil, for: nil)

  guard let user = profileVM.updatedUser() else {
    return
  }

  appState.updateProfile(age: user.age,
                         weight: user.weight,
                         height: user.height,
                         gender: user.gender,
                         active: user.activeLifestyle)
}
```

The first bit of the `updateProfile()` function makes sure that we remove any keyboards from the View.

But the important part of the function is updating the user profile on the `appState`. `AppState` is an `ObservableObject` with one `@Published` property.

```
class AppState: ObservableObject {
  @Published var user: User?
}
```

The `updateProfile()` function changes the `user` property on `appState`. This means that every time we press the save button, every View that's listening to the `AppState` will get updated!

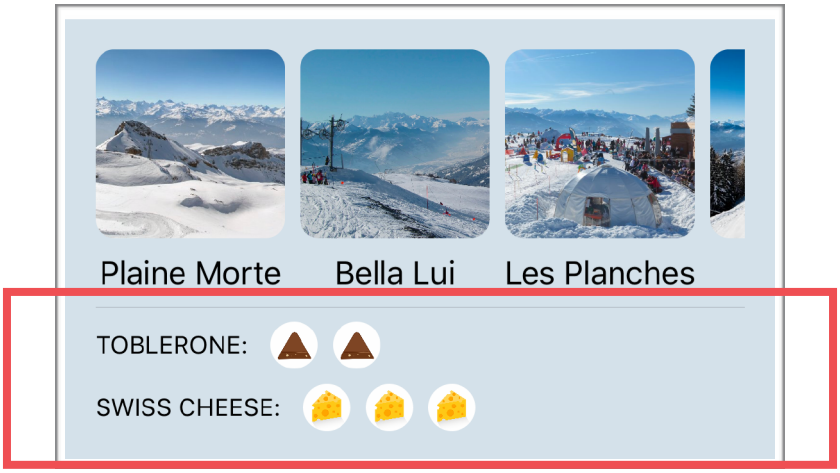
**To Do:** Figure out how to refactor Logbook View in order to add or remove the user's logs on `AppState`.

**Tip:** `AppState` has function a function `log(piste:)` to log a piste that generates a random piste and adds it to the user's log:

**Tip:** `AppState` has function `removeLog(at:)` to remove a log at a particular index.

# Task 19: Showing the Goods

Show how many Toblerones and Swiss cheese the user can eat based on their BMI and how much they've skied.



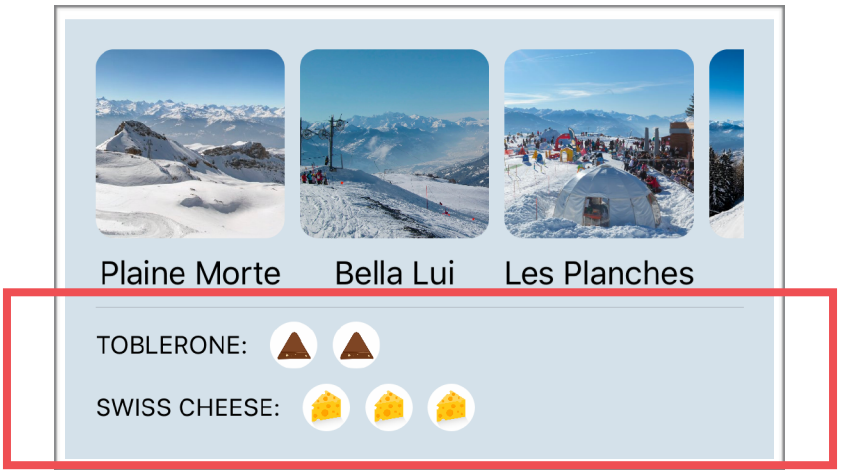

Time to wrap up!

At this point, you possess all the skills to generate this view, except how to create that thin grey line you see separating the Piste Picker and the Treats View.

To create that line you use a divider:

```
Divider()
```

**To Do:** Create the Treats View on Profile View and make sure the view updates when the user's profile information is saved and when they add new logs.



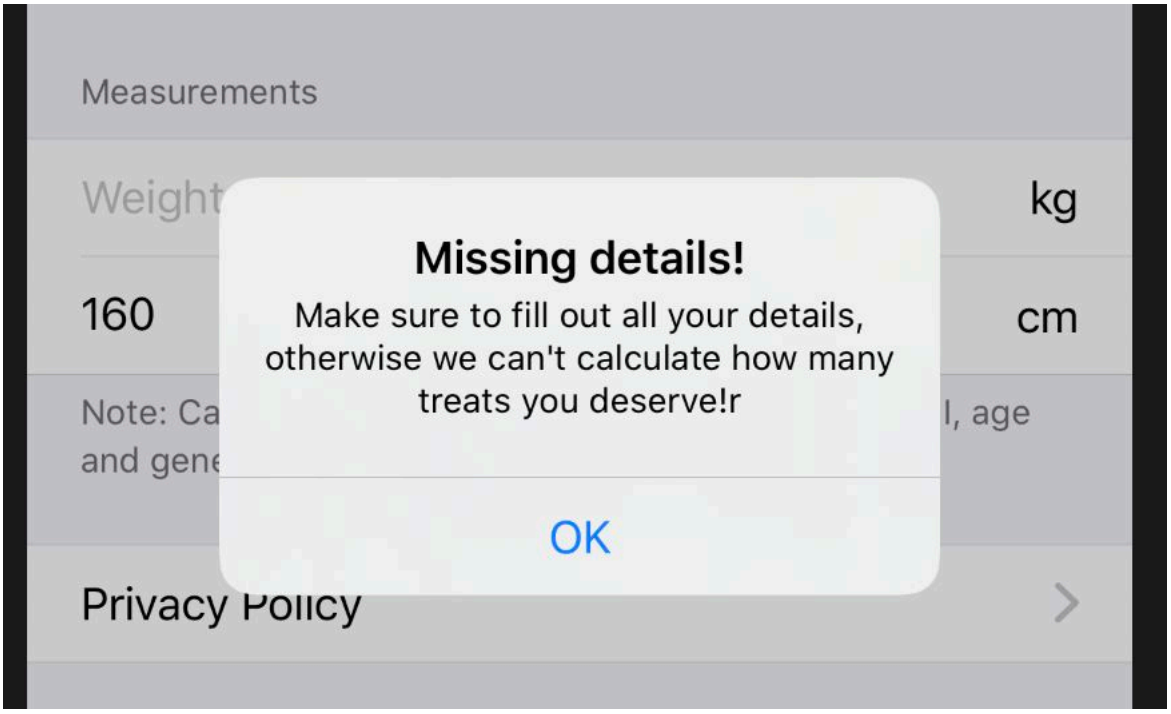

**Tip:** You can get a treat's information like calories, image and name in the `Treats.swift` file.

**Tip:** The total number of calories a user spends that day (including the ski runs) is the computed `caloriesSpentToday` property on the `User` object.

# 5: Extra Credit

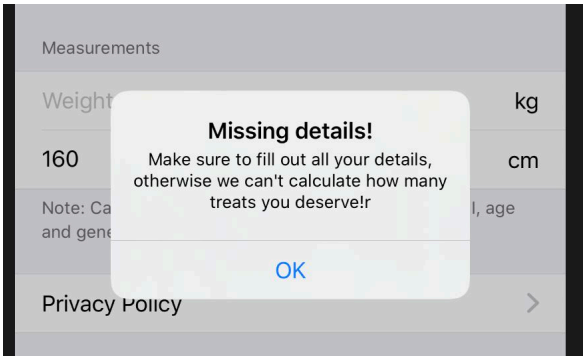
**Goal:** Display an alert if some of the details are missing from the user profile and add a privacy policy view.

**Learning:** You'll learn how to present action sheets, modal sheets and views on the navigation stack.



# Task 20: Alert

Display an alert if the user hasn't filled out all of their details.




Displaying an alert is a temporary change of a View's local state.

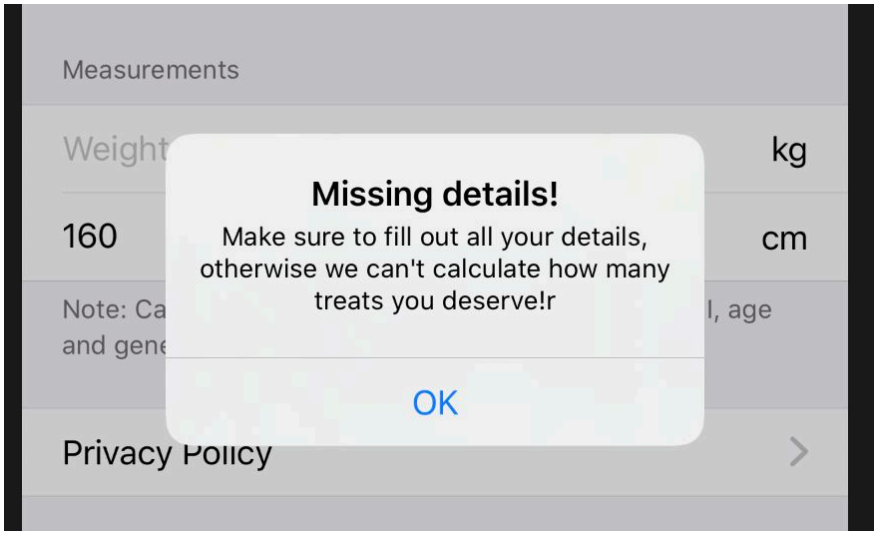
To display an alert the view has to:

1. Keep a `@State` bool property to tell the view whether to display the alert or not
2. Add a view modifier to the view to tell it how it should be displayed

```
struct ContentView: View {
  @State var isShowingAlert = false

  var body: some View {
    Button(action: {
      self.isShowingAlert = true
    }) {
      Text("Alert me!")
    }
    .alert(isPresented: $isShowingAlert) {
      Alert(title: Text("Alert!"),
            message: Text("Abort, can not compute."),
            dismissButton: .default(Text("OK"), action: {
              self.isShowingAlert.toggle()
            })
      )
    }
  }
}
```

1. You create a button showing "Alert Me!" that sets the `isShowingAlert` property to true.
2. You add the `.alert()` modifier that you initialise with a two-way binding `isShowingAlert` .
3. You create an `Alert` object with a title, message and appropriate actions.
4. The dismiss button of the `Alert`, toggles the `isShowingAlert`.

**To Do:** If the user hasn't filled out all of the details, show an alert, because otherwise we can't properly calculate their calorie expenditure.




**Description text:** "Make sure to fill out all your details, otherwise we can't calculate how many treats you deserve!"

# Task 21: Modals and Pushes

## Push and present a privacy policy view.

Being able to create a navigation architecture and hierarchy is one of the most important aspects of iOS developers. What it boils down to is pushing and showing view controllers.

Showing a “view controller” modally works similarly to showing an alert view:

1. Keep a `@State bool` property to tell the view whether to display the modal or not
2. Add a `.sheet()` view modifier to the view to tell it how it should be displayed

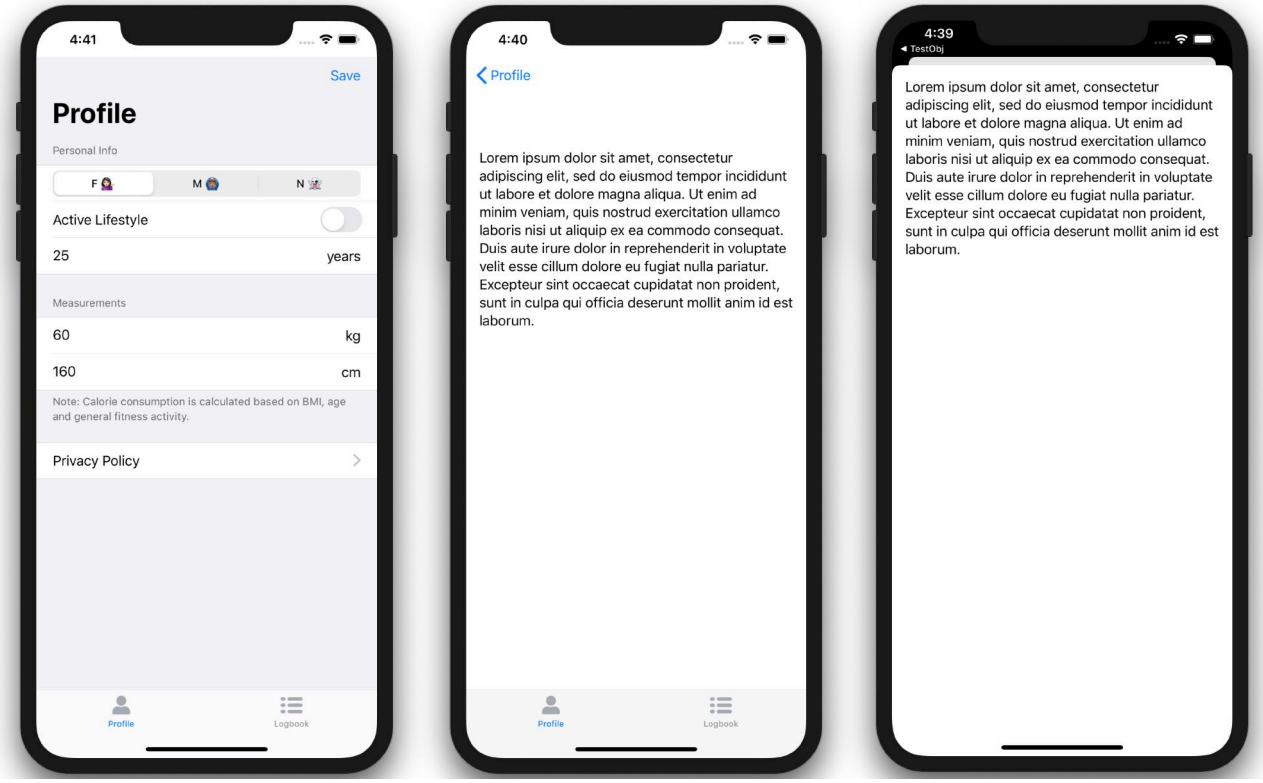
```
struct ContentView: View {
  @State var isShowingModal = false

  var body: some View {
    Button(action: {
      self.isShowingModal = true
    }) {
      Text("Tap me!")
    }
    .sheet(isPresented: $isShowingModal) {
      Text("I’m modally presented!")
    }
  }
}
```

Pushing a new “view controller” on the navigation stack works differently though:

1. The Parent View has to be a `NavigationView`.
2. The Child View has to be wrapped in a `NavigationLink`
3. SwiftUI handles the rest.

```
struct ContentView: View {
  var body: some View {
    NavigationView {
      NavigationLink(destination:
        Text("I've been pushed!"))
      {
        Text("Tap me!")
      }
    }
  }
}
```

**To Do:** There’s a `PrivacyPolicyView` View already in the application. Add a new section to the `ProfileView` and first show the `PrivacyPolicyView` as a `NavigationLink`, then show it as a modal.



# 6: Resources

## Free

1. Hacking with Swift; fantastic starting point and a broad overview
2. SwiftUI Lab; more in-depth explorations for specific use-cases
3. Maijid's Blog; in-depth explanations of core concepts
4. MovieSwiftUI; an open-source "real-world" app + a write-up
5. raywenderlich.com; a collection of free SwiftUI articles
6. SwiftUI Bugs; open-source repo of SwiftUI bugs so you don't waste your time trying to figure out if it's you or the framework
7. goshdarnswiftui.com: A cheat sheet showing the conversions between UIKit and SwiftUI.

## Paid

1. Hacking with SwiftUI Book; Paul Hudson
2. raywenderlich.com SwiftUI Screencasts, Video Courses and Books
3. objc.io SwiftUI collection