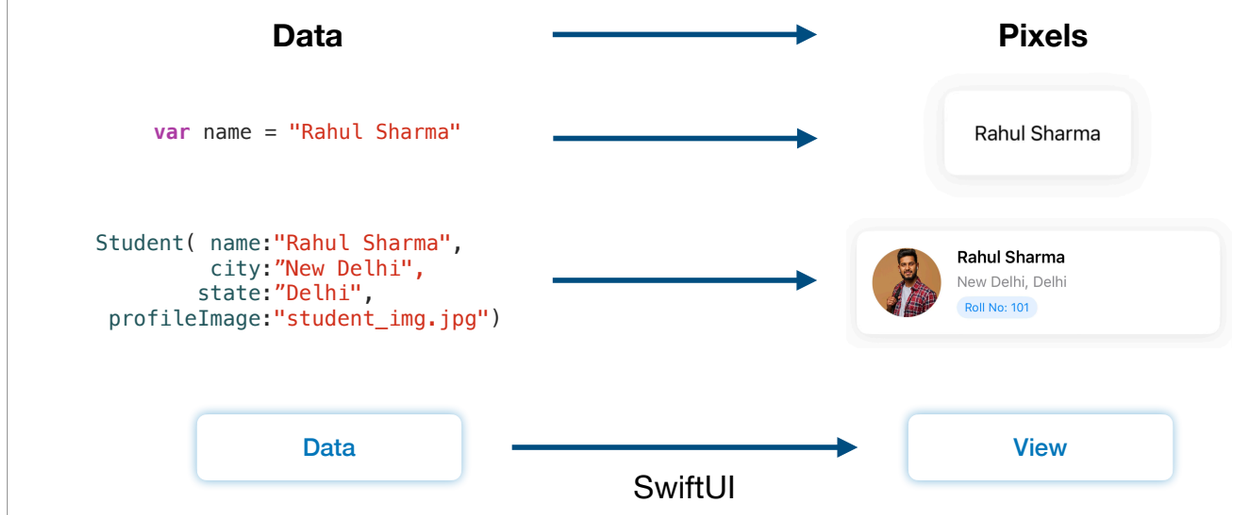
Introduction to SwiftUI

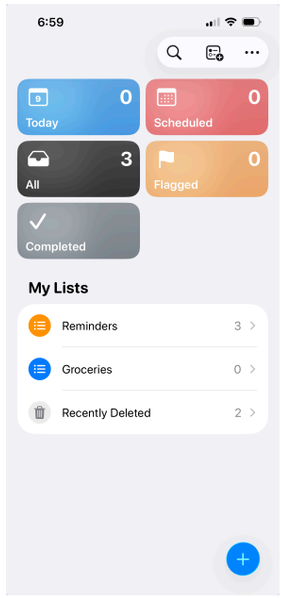
State Driven Views

View Basics


Data
Pixels
var name = "Rahul Sharma"
Rahul Sharma
Student(name:"Rahul Sharma",
city:"New Delhi",
state:"Delhi",
profileImage:"student_img.jpg")
Rahul Sharma
New Delhi, Delhi
Roll No: 101
Data
SwiftUI
View

View

- Views define pieces of your UI
- Everything you see on screen is a View
- Text, Image, Button, Stacks… all Views



Views are the building blocks of SwiftUI.
Every pixel on the screen can be traced back to a View — whether it’s a text label, an image, or even layout elements like HStack or VStack.
Just like UIKit has UIView, SwiftUI has View.
But SwiftUI expresses views very differently.

View

- Structs
- Lightweight
- Conforms to View protocol

```
protocol View {
    var body : View
}
```

Unlike UIKit classes that inherit from UIView, SwiftUI Views are value types.
In SwiftUI, every View is a **struct**.

Structs are simple types in Swift, and they don’t hold heavy logic.
They carry just enough information to describe what should appear onscreen.
They don’t store state internally.
They only describe and simply return what the UI should look like right now.

View

Example

```
struct SomeView: View {
    var body: some View {

    }
}
```

To be a SwiftUI View, a struct must conform to the View protocol
The View protocol has a single requirement: the body.
To conform to the View protocol, our struct must provide a body property.

The body describes the content and behavior of the view.
And every time your data changes, SwiftUI recalculates the body — giving you a fresh, updated UI automatically.

The body is just a description of the UI we want SwiftUI to display.
The body tells SwiftUI: this is what I want to show on the screen.

View

Built-in views

```
Text  TextField  SecureField  TextEditor  Label

Image  Shape  Color  Spacer  Divider

Button  EditButton  Link  NavigationLink

Toggle  Picker  DatePicker  Slider  Stepper
```

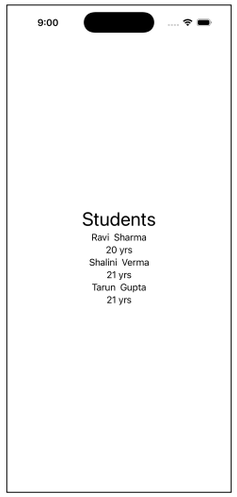
Code along

Project StudentHub

1. Open Xcode and click “Create a new Xcode project.”
2. Select **iOS → App** and press **Next**.
3. Give your project name **StudentHub**, make sure **Interface is set to SwiftUI** and **Language is Swift**, then press **Next**.
4. Choose where you want to save the project and click **Create**.
5. Xcode opens with the default SwiftUI template — you’ll see a `ContentView` file and a preview.
6. Now lets start building our **StudentHub**

Student Hub

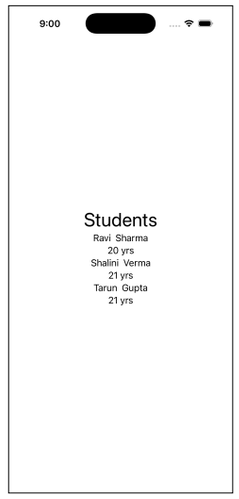
List of students



Student Hub

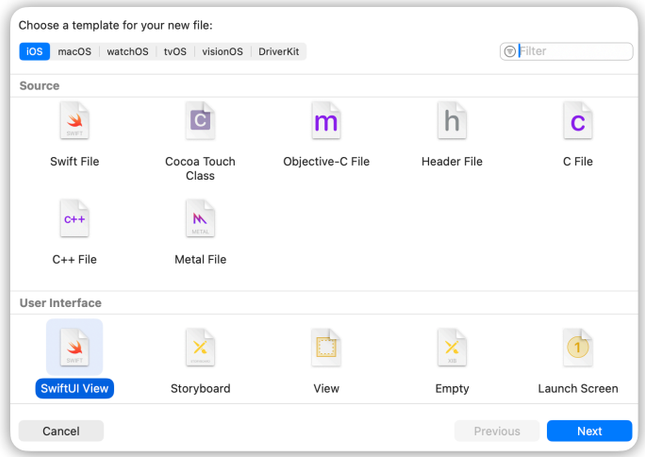
List of students

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Students")
                .font(.largeTitle)
            VStack {
                HStack {
                    Text("Ravi")
                    Text("Sharma")
                }
                Text("20 yrs")
            }
            VStack {
                HStack {
                    Text("Shalini")
                    Text("Verma")
                }
                Text("21 yrs")
            } ...
        }
    }
}
```



Creating new views

1. File → New → SwiftUI View. Give it a name as **StudentView**



Student Hub

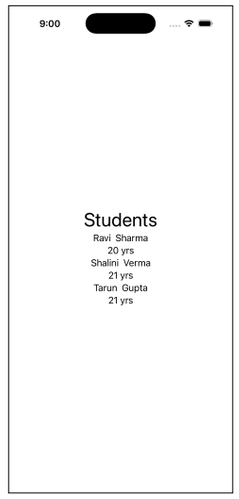
Student View

```
struct StudentView: View {
    var firstName:String = "Tarun"
    var lastName:String = "Verma"
    var age:Int = 0

    var body: some View {
        VStack {
            HStack {
                Text(firstName)
                Text(lastName)
            }
            Text("\(age) yrs")
        }
    }
}
```

Student Hub

List of students

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Students")
                .font(.largeTitle)
            VStack {
                HStack {
                    Text("Ravi")
                    Text("Sharma")
                }
                Text("20 yrs")
            }
            VStack {
                HStack {
                    Text("Shalini")
                    Text("Verma")
                }
                Text("21 yrs")
            } ...
        }
    }
}
```



Student Hub

List of students (Refactored through Composition)

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Students")
                .font(.largeTitle)
            StudentView(firstName: "Ravi", lastName: "Sharma", age: 20)
            StudentView(firstName: "Shalini", lastName: "Verma", age: 21)
            StudentView(firstName: "Tarun", lastName: "Gupta", age: 21)
        }
    }
}
```

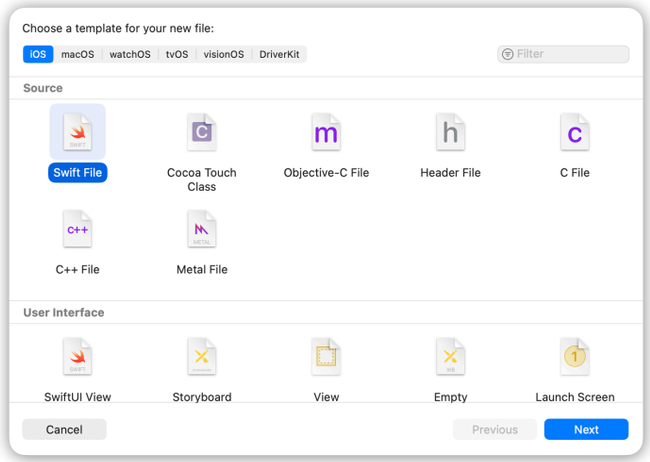
Student Hub

Student View

```
struct StudentView: View {
    var firstName:String = "Tarun"
    var lastName:String = "Verma"
    var age:Int = 0

    var body: some View {
        VStack {
            HStack {
                Text(firstName)
                Text(lastName)
            }
            Text("\(age) yrs")
        }
    }
}
```

Creating model

1. File → New → Swift File. Give it a name as **StudentModel**



Student Hub

Student Model

```
struct Student {
    var firstName:String = ""
    var lastName:String = ""
    var age:Int = 0
}
```

Student Hub

Student View

```
struct StudentView: View {
    var firstName:String = "Tarun"
    var lastName:String = "Verma"
    var age:Int = 0

    var body: some View {
        VStack {
            HStack {
                Text(firstName)
                Text(lastName)
            }
            Text("\(age) yrs")
        }
    }
}
```

Student Hub

Student View

```
struct StudentView: View {
    var student: Student = Student()

    var body: some View {
        VStack {
            HStack {
                Text(student.firstName)
                Text(student.lastName)
            }
            Text("\(student.age) yrs")
        }
    }
}
```

Student Hub

List of students (Refactored through Composition & Model)

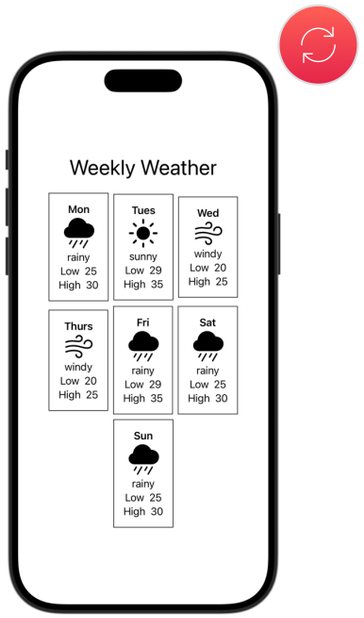
```
Text("Students")
     .font(.largeTitle)
StudentView(student: Student(firstName: "Ravi", lastName: "Sharma", age: 20))
StudentView(student: Student(firstName: "Shalini", lastName: "Verma", age:
21))
StudentView(student: Student(firstName: "Tarun", lastName: "Gupta", age: 21))
```

Activity

Week's Weather

Instructions

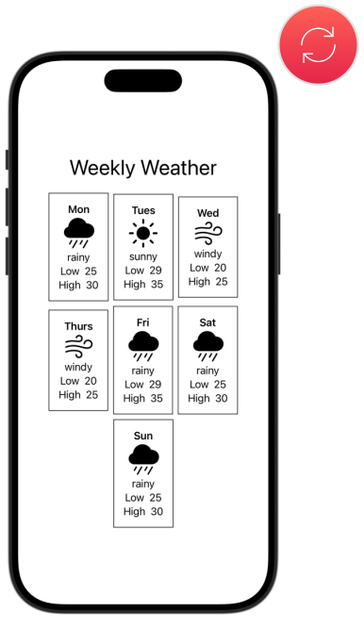
- Create a project called WeeklyWeather.
- The app shows the weather for the 7 days of the week.
- The weekly weather should be composed of views each of which displays the **high** and **low** temperatures of the day along with the day's overall weather prediction as 1 of the 4 - Sunny, cloudy, windy, or rainy.
- Design and implement a **data model** to represent the weather details for a single day.



Week’s Weather

Hints

- Use enums for the outlook - sunny, rainy, cloudy or windy.
- The enum can also contain a computed property that returns the name of the SF symbol for the respective outlook case
- .border() modifier on the VStack applies a border on the VStack
- .padding() gives a surrounding space around the view its applied to.



Code along

Counter

1. Open Xcode and click “Create a new Xcode project.”
2. Select **iOS → App** and press **Next**.
3. Give your project a name **“Counter”**, make sure **Interface is set to SwiftUI** and **Language is Swift**, then press **Next**.
4. Choose where you want to save the project and click **Create**.
5. Xcode opens with the default SwiftUI template.

Counter

Lets increment the counter

```
struct ContentView: View {
    var counter = 0
    var body: some View {
        Text("The counter is \(counter)")
        Button("Increment") {
            print("Counter incremented")
            counter += 1    Left side of mutating operator isn't mutable: 'self' is immutable
        }

    }
}
```

But… Structs Are Immutable

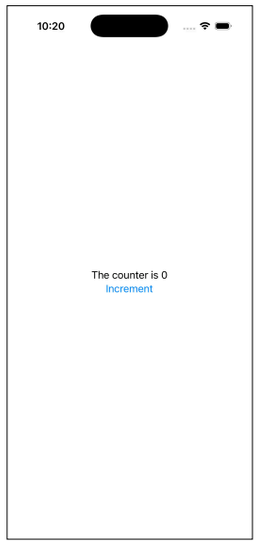
Because Views are structs, their properties normally can’t change.

And yet, we need interactive views, we need views that need changing data.— like toggles, counters, forms.
This is why SwiftUI gives us **@State.**

@State private

Counter incremented

```
struct ContentView: View {
    @State private var counter = 0
    var body: some View {
        Text("The counter is \(counter)")
        Button("Increment") {
            print("Counter incremented")
            counter += 1
        }

    }
}
```



Notice:
We never reload the view.
We don’t call refresh.
Changing counter triggers SwiftUI to regenerate the body

Counter lives in @State
Button updates it
Text reflects it

What is @State?

- A property wrapper
- Stores modifiable state **outside** the struct
- SwiftUI manages it
- Changing @State triggers UI update

@State turns an otherwise immutable struct into something that can react to changes.
It’s the single source of truth for a View’s data.

Not All Properties Need @State

Important

Only use @State for data that

✓ Belongs to the view

✓ Can trigger a UI update

For everything else, plain `let` or `var` is enough.

Activity

Light Switch
Part 1

- Create a `@State` variable that tracks whether the light is ON or OFF.
- Show a text label:
  - **“Light is ON 💡”** when the light is on
  - **“Light is OFF 🔌”** when the light is off
- Add a button that toggles the light state each time it is tapped.
- Update the UI immediately when the state changes.

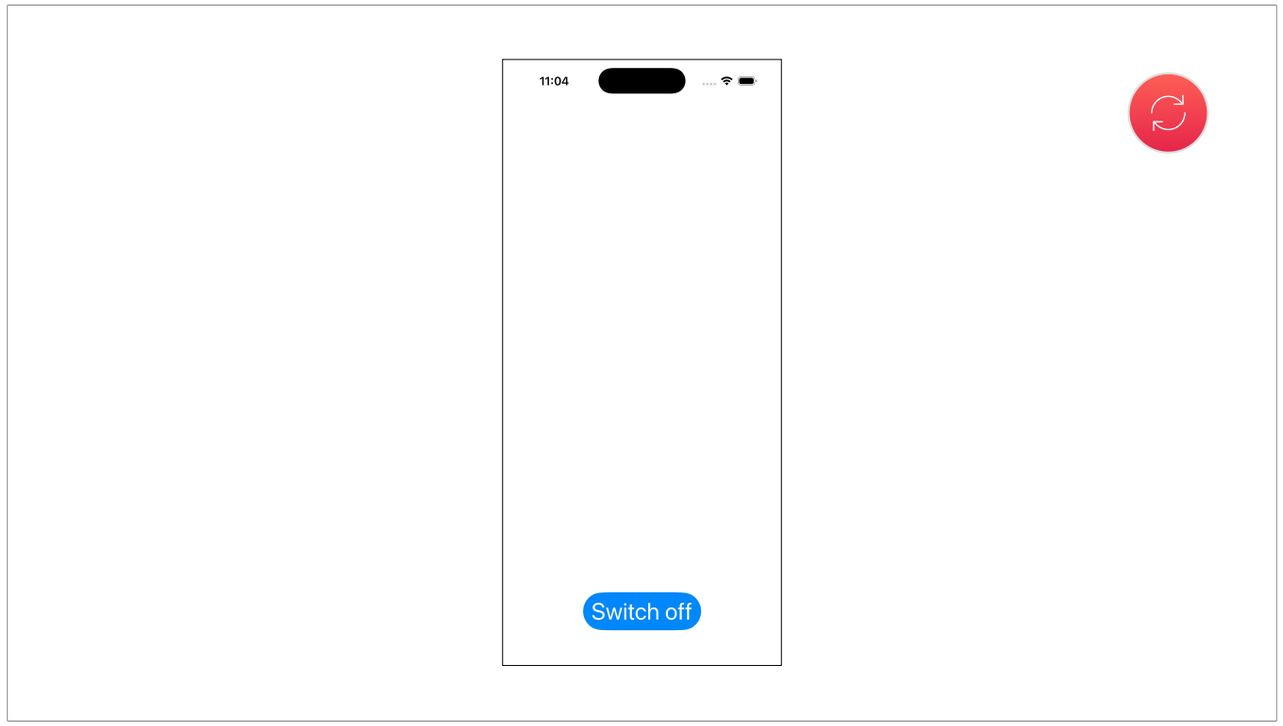
```
struct ContentView: View {
    @State private var isLightOn:Bool = true
    var body: some View {
        VStack {
            Text("Light is \(isLightOn ? "ON 💡" : "OFF 🔌")")
            Button("\(isLightOn ? "Switch off" : "Switch on" )"){
                isLightOn.toggle()
            }
            .buttonStyle(.borderedProminent)
        }
        .padding()
        .font(.largeTitle)
    }
}

#Preview {
    ContentView()
}
```

Hint - .buttonStyle(.borderedProminent)

Light Switch
Part 2

- Create a `@State` variable that tracks whether the light is ON or OFF.
- Show a white color view when the light is ON and a black color view when the light is OFF.
- Add a button whose label changes between **“Switch off”** and **“Switch on”**, and toggles the light state when tapped.
- The UI should update immediately when the state changes.

```
struct ContentView: View {
    @State private var isLightOn:Bool = true
    var body: some View {
        VStack {
            if isLightOn {
                Color.white
            } else {
                Color.black
            }
            Button("\(isLightOn ? "Switch off" : "Switch on" )"){
                isLightOn.toggle()
            }
            .buttonStyle(.borderedProminent)
        }
        .padding()
        .font(.largeTitle)
    }
}
```

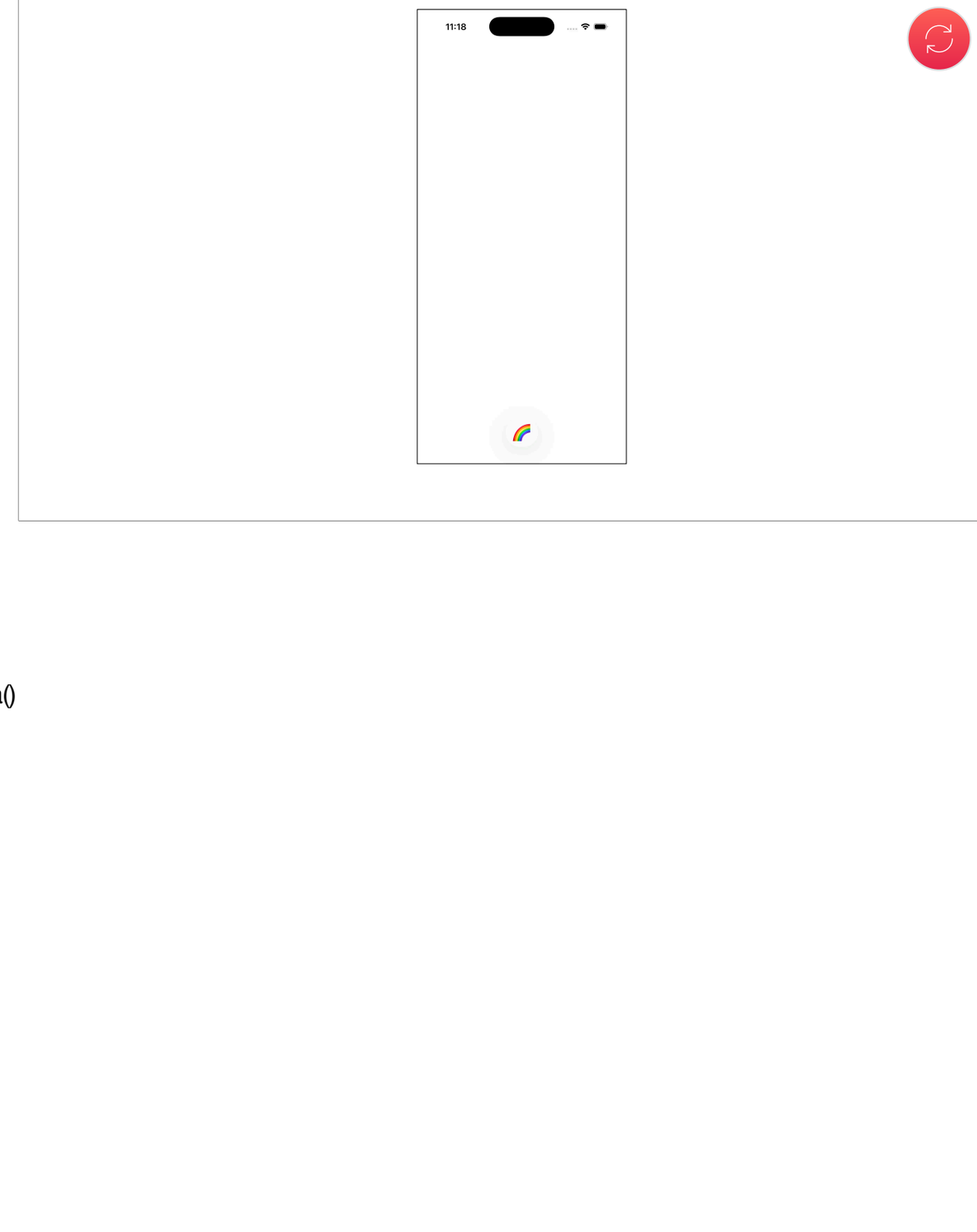


Hint - Color.white and Color.black are also views

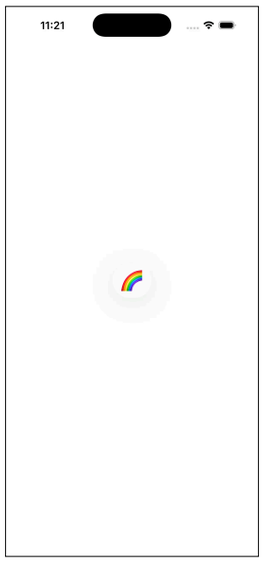
Rainbow Switch

- Create a `@State` variable to store the current color (e.g., `someColor`), starting with any initial color.
- Display this color as a large rectangular area on the screen.
- Add a button labeled **“Random Color”**.
- When the button is tapped, update the `@State` variable with a **new random color** by generating random red, green, and blue values.
- The color on the screen should change immediately after every tap.

```
struct ContentView: View {
    @State private var someColor:Color = .white
    var body: some View {
        someColor
        Button("🌈") {
            someColor = Color(red: .random(in: 0...1), green: .random(in: 0...1), blue: .random(in: 0...1))
        }
        .font(.largeTitle)
        .buttonStyle(.glass)
    }
}

#Preview {
    ContentView()
}
```



Hint - .buttonStyle(.glass)
Try using ZStack{} and .ignoreSafeArea()



```
struct ContentView: View {
    @State private var someColor:Color = .white
    var body: some View {
        ZStack {
            someColor
            Button("🌈") {
                someColor = Color(red: .random(in: 0...1), green: .random(in: 0...1), blue: .random(in: 0...1))
            }
            .font(.largeTitle)
            .buttonStyle(.glass)
        }
        .ignoresSafeArea()
    }
}
```