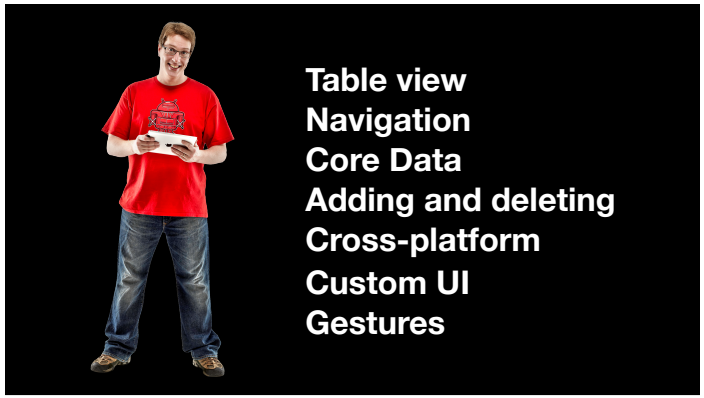
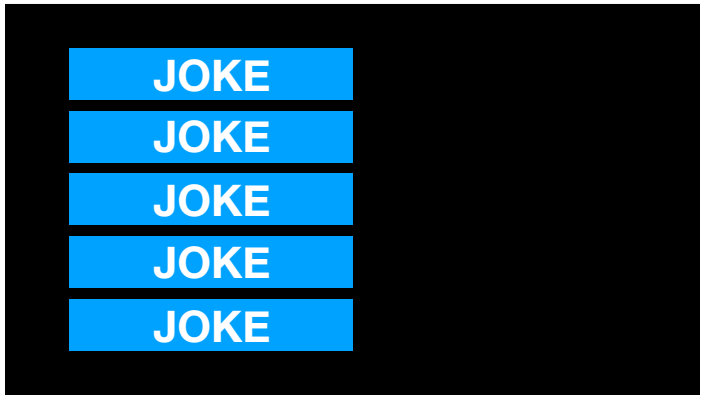
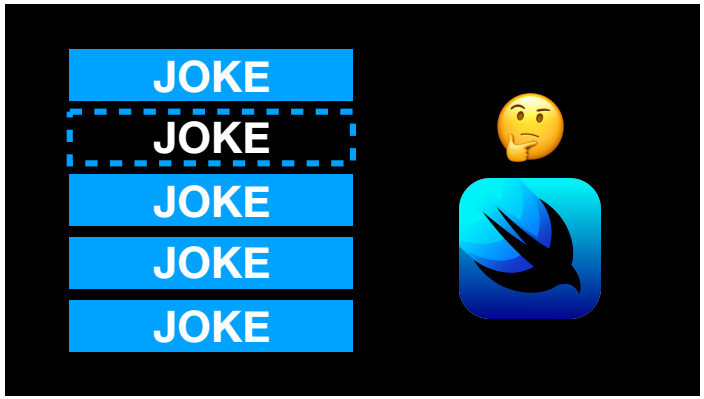


In this project we're going to build an app with all these features.



If we have an array of jokes being listed in a SwiftUI view…



…when we delete one, SwiftUI needs to know which one was removed so it can update its layout correctly.

```
struct Joke {
    var setup: String
    var punchline: String
    var rating: String
}
```

Here's how our Joke struct looks right now.

```
struct Joke: Identifiable {
    var setup: String
    var punchline: String
    var rating: String
}
```

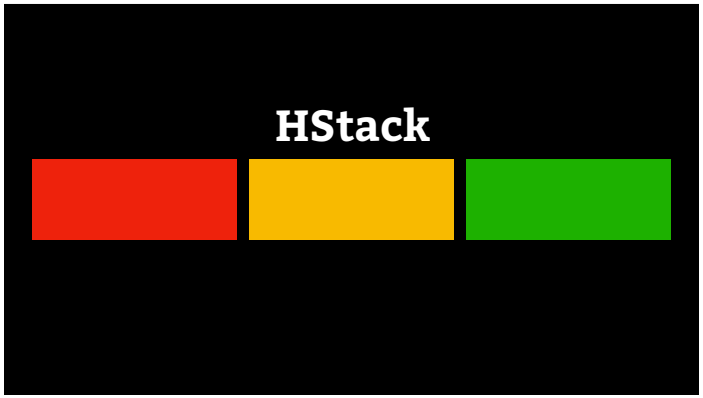
Swift 5.1 introduces a new Identifiable protocol that can be used to identify objects uniquely in a collection.

```
struct Joke: Identifiable {
    var id = UUID()
    var setup: String
    var punchline: String
    var rating: String
}
```

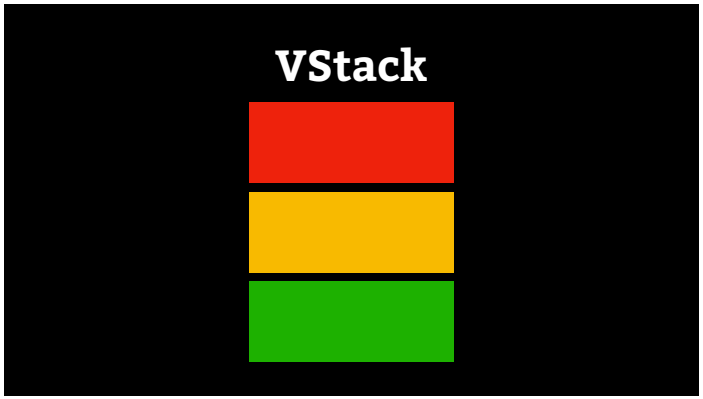
All it requires is that we provide a unique "id" property. In this case, giving each joke a universally unique identifier (UUID) would work.

```
struct Joke {
    var setup: String
    var punchline: String
    var rating: String
}
```

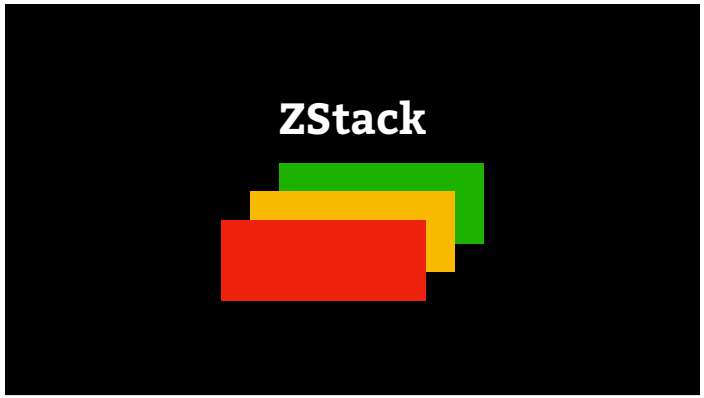
But actually our Joke struct already has something unique: its "setup" property. These aren't repeated in our app, so we can use that instead.



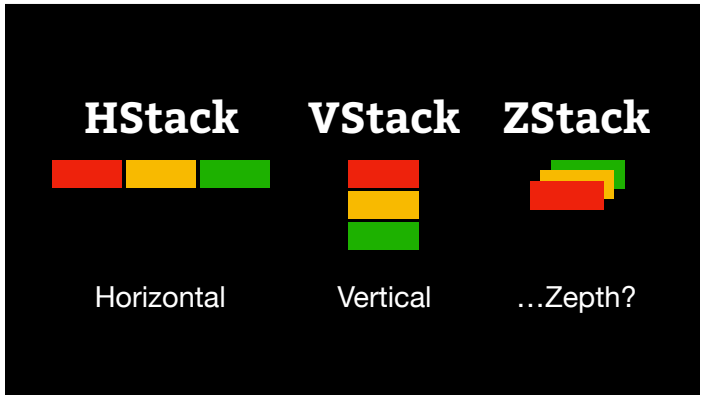
HStack orders views horizontally.



VStack works identically, except it orders views vertically.



And ZStack orders them by depth, so they overlap.



These three basic stack types let us arrange views however we need.

And they are amazing for accessibility – the screen reader system automatically knows the order our views are in.

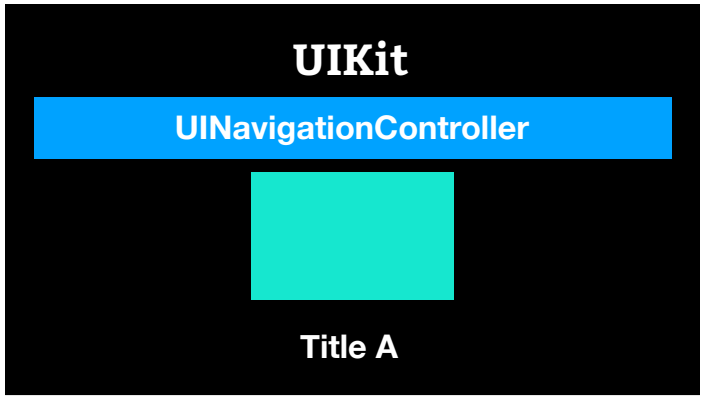
That completes our table view; we can look at navigation now.

```
NavigationView {
    List {
        …
    }
    .navigationBarTitle("🤔")
}
```

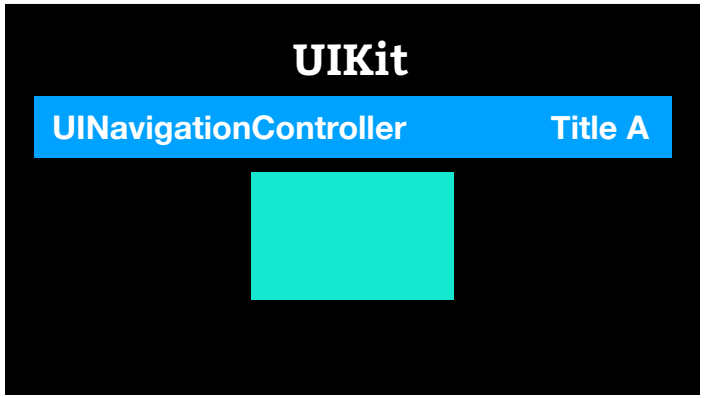
Some people are confused why the navigationBarTitle() modifier is attached to a list, rather than the navigation view around the list.

```
NavigationView {
    List {
        …
    }
}
.navigationBarTitle("🤔")
```

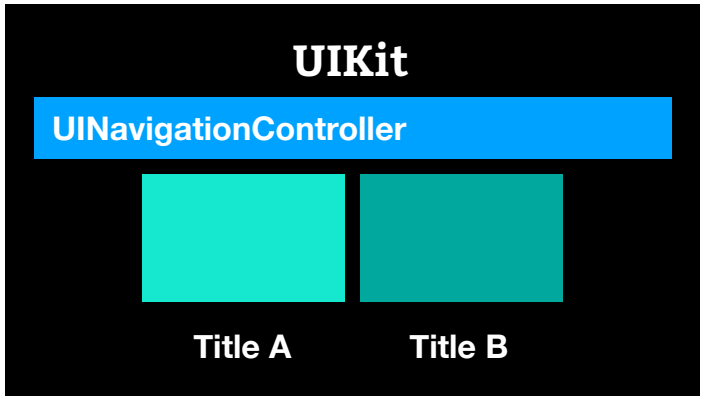
They feel that it should go here instead.



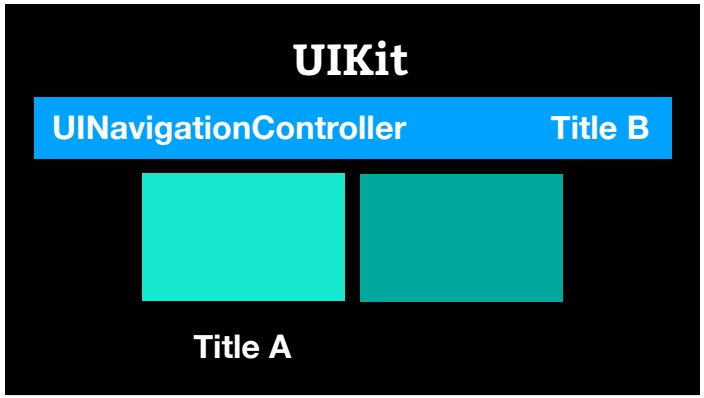
But think about the way UIKit worked. If we have a view controller with Title A inside a navigation controller…



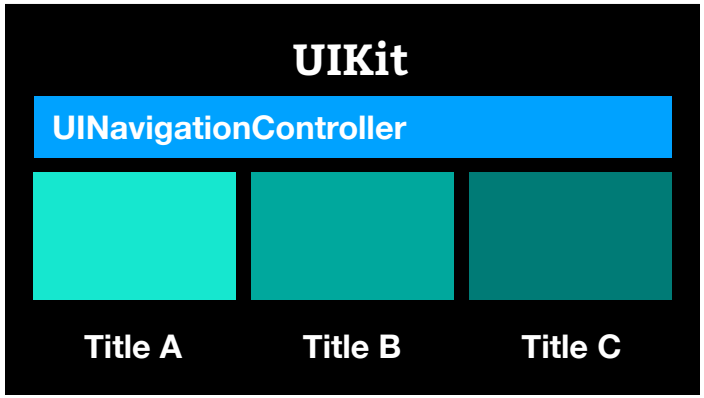
…that title would automatically bubble upwards.



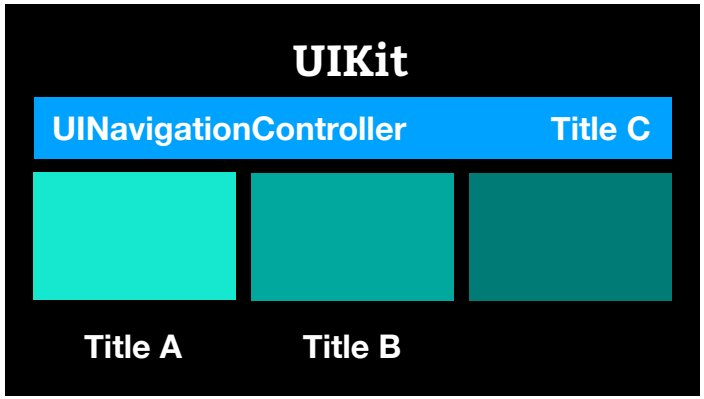
So when we showed view controller B with a different title…



…that would also bubble up and replace Title A.



And when a third view controller was pushed onto the stack…



…that would also be displayed in the navigation bar.

UIKit

- UINavigationController
- UIViewController

So we attach titles to the things inside navigation controllers, not navigation controllers themselves. This lets UIKit animate changing titles and buttons.

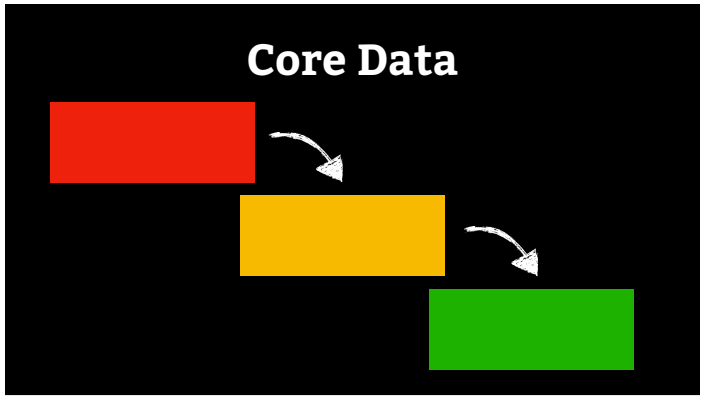
SwiftUI

- NavigationView
- YourCustomView

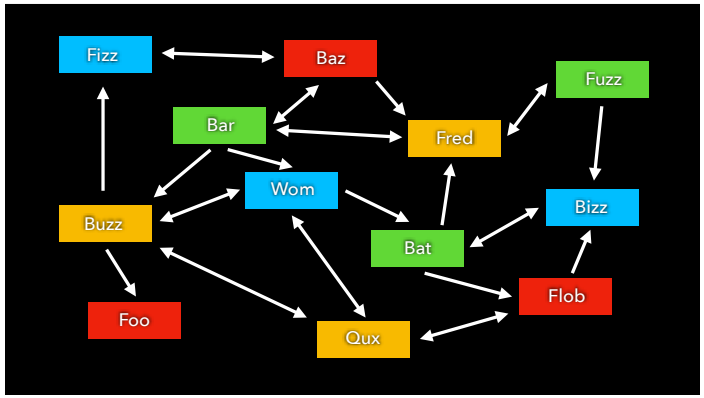
The same is true in SwiftUI: we attach our navigation title to the view inside the navigation view, not the navigation view itself.

~~Table view~~
~~Navigation~~
Core Data
Adding and deleting
Cross-platform
Custom UI
Gestures

So that's navigation. Next is Core Data.

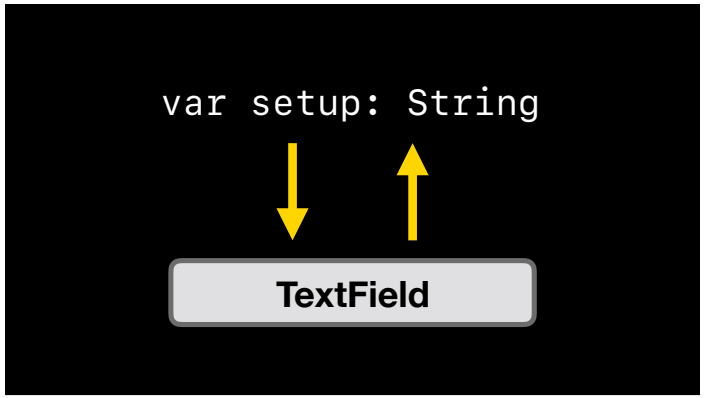


In UIKit we're used to the idea of creating a Core Data instance somewhere then passing it around the app.

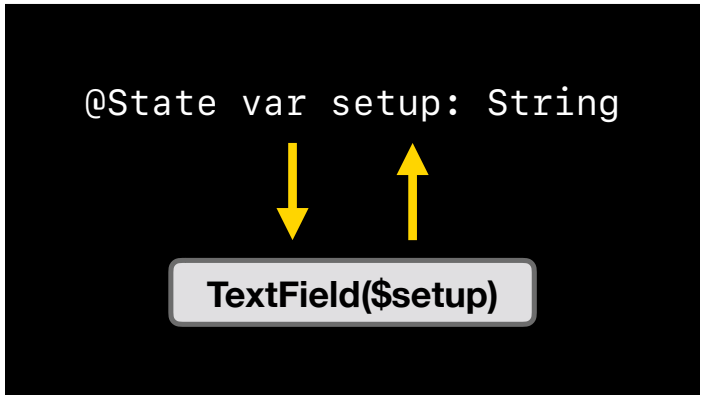


This can be confusing when we have lots of view controllers; we need to keep passing objects around.

SwiftUI gives us a different approach, inspired by the Reader monad in Haskell: we create an object and place it into a shared pool called the Environment, which other views can read.



When working with controls such as TextField we want properties to be both read (shown inside the text field) and written (updated when the text field changes.)



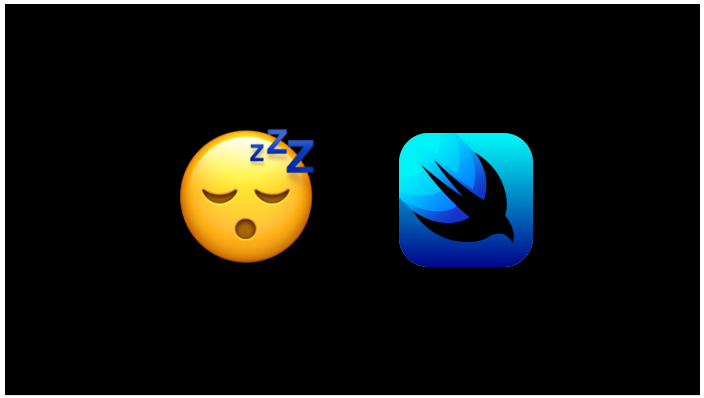
In SwiftUI that means marking the property with @State so it can change, then using $ before the property name in the text field, so it creates a two-binding to the underlying value.

@State automatically transfers control of a property to SwiftUI, which allows us to do something that would otherwise be impossible.

So that's Core Data complete.

And that's Adding and Deleting complete too.

Er… and that's Cross Platform done too. Sorry, this part of the slides doesn't really make sense without the video!



So far everything we've made has been plain vanilla SwiftUI stuff: our next stop is making a custom UI by doing away with our list entirely.

This part of the video was great – honest!

And so was this. Again, you should really watch the video; there weren't any slides here.

You should follow me on Twitter 🙂