ScrollView and List in detail

In this reading, you will learn how to create `Lists` and `ScrollViews`, two of the most important elements of SwiftUI, which let you display static data hard coded to the code and dynamic data from arrays or databases.

Similarities

As you have seen previously in this course, `ScrollViews` and `Lists` have very similar syntaxes.

Both elements, `ScrollViews` and `Lists` are scrollable, meaning they can hold any number of elements, but there is a caveat when dealing with hard coded elements. You will learn about this later in this reading.

Hard coding elements

`ScrollView` and `Lists` allow hard coding elements directly into them.

```
ScrollView {
    Image("littleLemonLogo")
    Image("littleLemonLogo")
    Image("littleLemonLogo")
    Image("littleLemonLogo")
}
```

```
List {
    Image("littleLemonLogo")
    Image("littleLemonLogo")
    Image("littleLemonLogo")
    Image("littleLemonLogo")
}
```

This approach will produce the following results:

You can note by observing the previous images that the `ScrollView` produces a cruder visual, with barely any formatting. On the other hand, the result produced by the `List`, in figure 2, is heavily formatted. You can see a grey background for the `List`, and each cell has a white background and line separations.

The problem with this approach is that the code is rigid and cannot be modified easily. Every change will require modifying the code and recompiling the code.

Another problem with using hard coded elements is that you will hit the SwiftUI limitation of 10 elements per view element.

The following code, for example, will not be compiled by Xcode, because there are eleven image elements inside the

`ScrollView` (the same is valid for `Lists` or any other SwiftUI view element):

```
struct ContentView: View {
    var body: some View {
        List {
            Image("littleLemonLogo")
            Image("littleLemonLogo")
            Image("littleLemonLogo")
            Image("littleLemonLogo")
            Image("littleLemonLogo")
            Image("littleLemonLogo")
            Image("littleLemonLogo")
            Image("littleLemonLogo")
            Image("littleLemonLogo")
            Image("littleLemonLogo")
            Image("littleLemonLogo")
        }
    }
}
```

To circumvent this limitation, you will have to group elements in packs of ten, like:

```
struct ContentView: View {
    var body: some View {
        List {
            Group{
                // add 10 images inside this group
            }
            Group{
                // add 10 images inside this group
            }
        }
    }
}

```

To solve these limitations, the best approach is to use dynamic `ScrollViews` or `Lists` over hard coded ones. By doing so, you can store the elements to be displayed inside a CSV/JSON file or a CoreData database and modify them on-the-fly, without having to recompile the code.

Dynamic elements

The most basic approach to creating a dynamic `ScrollView` or `List` is to store the elements inside an array. This is a bad approach for a large number of elements, especially those that are intended to change during the app's lifetime.

The first thing you have to do is to create an array, for example:

```
    let elements = ["Reservation", "Contacts", "Restaurant Locations"]
```

The next logical step would be using a syntax like the following one, to extract and print the elements from the array:

```
struct ContentView: View {
    let elements = ["Reservation", "Contacts", "Restaurant Locations"]
    var body: some View {
        List {
            ForEach(elements) {element in
                Text(element)
            }
        }
        .padding()
    }
}
```

However, if you use that, Xcode will show the following error:

`Referencing initializer init(_:content:) on ForEach requires that String conforms to Identifiable.`

This happens because the elements that `ForEach` is processing cannot be distinguished from each other. Every element must be unique, which means that SwiftUI must be able to discern one element from the other.

The error message says that `ForEach` requires that `String` conforms to `Identifiable`. `Identifiable` is a protocol that makes objects displayed in a `List` or `ScrollView` uniquely identifiable, by forcing them to have an identification field called `id`.

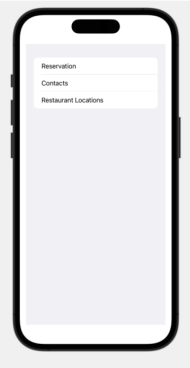
Solving the problem

To solve the problem, there are several approaches. The first one is to create an extension for `String`, like this:

```
extension String: Identifiable {
    public var id: String {
        self
    }
}
```

This extension makes `String` conform to Identifiable protocol and adds a variable called `id` to it. This variable will be used to uniquely identify every element on the list.

The final result is shown in the image below:



Another way to solve the problem is to add an `id` to each element on-the-fly, by adding `id: \.self` to the `List`:

```
struct ContentView: View {
    let elements = ["Reservation", "Contacts", "Restaurant Locations"]
    var body: some View {

        List {
            ForEach(elements, id: \.self) {element in
                Text(element)
            }
        }
        .padding()
    }
}
```

This code will produce the same result as before and also works for `ScrollViews` instead of `Lists`.

How List elements help you

You now know that `ScrollViews` and `Lists` are similar. You also know that `ScrollViews` produce a cruder visual with barely any formatting and that `Lists` produce a heavily formatted display of elements.

`Lists`, however, let you go way beyond `ScrollViews`.

Header and footers for Lists

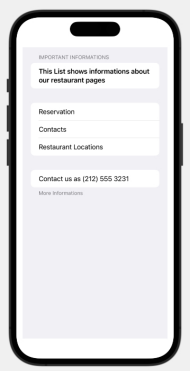
If you want to add headers and footers to a List, all you have to do is to add this code:

```
struct ContentView: View {
    let elements = ["Reservation", "Contacts", "Restaurant Locations"]
    var body: some View {
        List {
            Section(header: Text("Important Information")) {
                Text("This List shows information about our restaurant pages")
                    .font(.headline)
            }

            ForEach(elements, id: \.self) {element in
                Text(element)
            }

            Section(footer: Text("More Information")) {
                Text("Contact us as (212) 555 3231")
            }
        }
        .padding()
    }
}
```

This code will produce the following result:

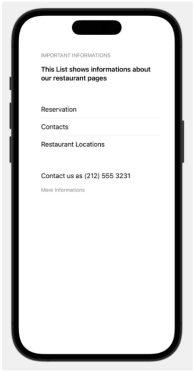


Getting rid of the List grey background

SwiftUI renders lists with a grey background. If you want to get rid of this background, add the following modifier to the `List`:

```
        .scrollContentBackground(.hidden)
```

And the final result is as follows:

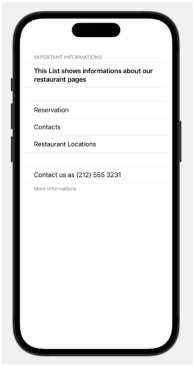


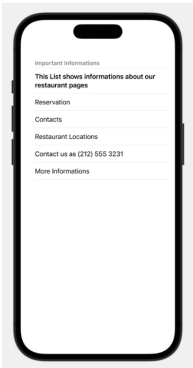
List styles

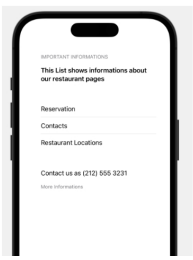
`Lists` can be displayed in many ways called styles. The ones available are: `.automatic`, `.grouped`, `.inset`, `.insetGrouped`, `.plain` and `.sidebar`. You use them by applying a modifier to the `List`, like this: `.listStyle(.grouped)`

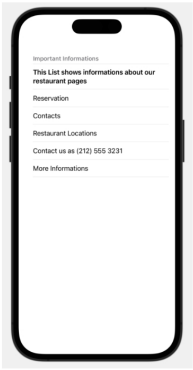
The result you see in the image below is the default one, equivalent to `.automatic`.

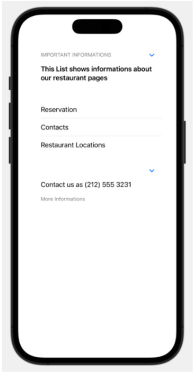
Other styles will produce the following results:











Tappable elements inside Lists

As mentioned earlier in this course, `NavigationLinks` are the elements you want to create tappable elements.

By adding the `NavigationLink` the previous code, you now have tappable elements:

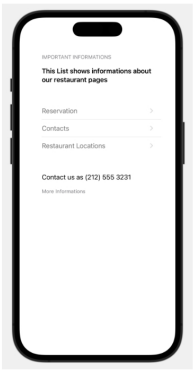
```
struct ContentView: View {
    let elements = ["Reservation", "Contacts", "Restaurant Locations"]
    var body: some View {
        List {
            Section(header: Text("Important Information")) {
                Text("This List shows information about our restaurant pages")
                    .font(.headline)
            }

            ForEach(elements, id: \.self) {element in
                NavigationLink(destination: DetailView()) {
                    Text(element)
                }
            }

            Section(footer: Text("More Information")) {
                Text("Contact us as (212) 555 3231")
            }
        }
        .scrollContentBackground(.hidden)
        .padding()
    }
}

```

This code will produce the following result:



Note the chevron at the right of every list element, indicating that these elements are tappable.

In this example, all elements, when tapped, will show the same view, `DetailView.` Added here as an illustrative example. In a real-life application, it would be better if they show their respective pages.

Conclusion

In this reading, you were presented with deeper details about `ScrollViews` and `Lists`.

In the following lessons, you will expand on these concepts and more.

Mark as completed

Like Dislike Report an issue