SwiftUI: From zero to hero

Hello brave humans!

Welcome to our SwiftUI workshop! We're happy to have you here!

Your task for today is very simple...

In the next 2 hours (or so) you will be building **Lamstagram** - a brand new social network for Llamas. The market is huge and business success is guaranteed.

Since Alpaca Inc. - our direct competitor - are planning to launch a similar product, we need to move fast. That is why we need to focus on getting a prototype out ASAP so that we can secure the first round of financing by the end of this week.

The app

Because of the rush, the scope is limited and we'll be building just a couple of screens.

Feed

Feed screen lists posts in the feed specific to the logged in user. In the case of our prototype we don't care about real data. We only care about cute images of llamas.

Tapping on image should bring up its detail. Tapping on heart should "like" the post and the likes should be persistent over all view contexts no matter how the user navigates to the feed.

We will ignore all other functionality for today.

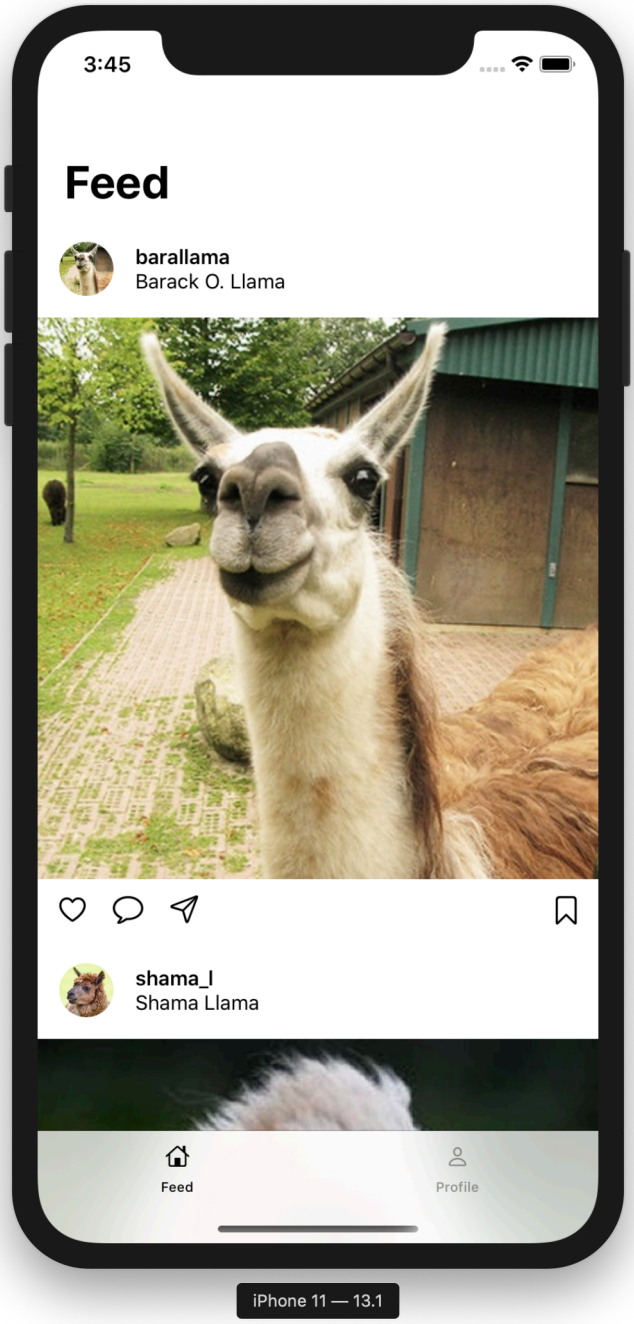


Photo detail

Photo detail screen is basically the photo presented modally. Simple, eh?

Profile

The profile screen shows information about currently logged user. Namely profile photo and information, horizontally scrolling list of friends and list of user's posts in a grid-like view.

Navigation

We're not going to build anything too complex here but we need to let the user navigate throughout the app.

General

Tabs

The tab bar at the bottom switches between Feed and Profile screens.

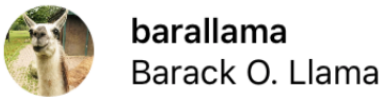
Navigation Views

Similar to `UINavigationController` each of the tabs contains a `NavigationView` to allow pushing and popping new views in the stack.

Feed

User info

Tapping on user's photo, name or nickname pushes that user's profile View into the navigation stack.



Photo

Tapping on the photo presents the photo detail modally.

Profile

Tapping on one of friends pushes their Profile view into the navigation stack.

Photo

Tapping on one of the photos pushes that user's Feed view into the navigation stack.

Let's start coding!

We're here to learn programming in SwiftUI so let's get started.

The project

I have prepared an Xcode project for you.

Please clone this repository: https://github.com

In this repository a lot of things are ready for you so that you can focus purely on building the UI. There are Models and data Stores prepared that you will use to display data in your UI. There are also some utilities but more about that later.

Now run the app and you should see a blank View with **Hello world!** message centered on the screen.

The only view ready for you is the root view, the `ContentView`. You can see it contains only one view `Text` with the message. In SwiftUI a View will be centered by default both horizontally and vertically unless specified differently.

Remember that in SwiftUI the views are composable. Simply initialize your view inside another View and it will be displayed within it. The views that accept views always have a function builder closure as the last parameter.

For example:

```
VStack {
  Text("Hello")
  Text("World")
}
```

Or more complex:

```
Button(action: { print("Hello world!") }) {
  Text("Hello world!")
}
```

To change appearance or behavior of views we use view modifiers that are simply functions called on the view we want to modify. View modifiers can be chained together.

For example:

```
Image("Photos/1")
  .resizable()
  .clipShape(Circle())
```

Enough of examples!

It's your turn to code now!

1. The tab bar

Let's create the TabBar with two tabs.

When you're done the UI should look like the screenshot below.

Start by creating a new Swift file named `MainView`. Copy the contents (ha ha) of `ContentView` into it and rename everything from `Content` to `Main`.

Now add the `TabView` into the `body` property of your shiny new View. This property contains the View's UI definition and is present in all SwiftUI Views.

For now let's just add a Text view with **Feed** and **Profile** labels respectively for each of the tabs.

Since we're going to need to use pushing and popping in these Views let's also wrap the Text views within the `TabView` in `NavigationView` so that we can do that later.

For the tab items, system images named `house` and `person` are a good choice. `Image` view has a nice `init` that you can use. By the way... These symbols and many more are provided by Apple for free!

The view hierarchy should look similar to this:

```
TabView
  NavigationView
    Text
  NavigationView
    Text
```

To add some bling let's apply `.accentColor(.black)` modifier to the whole `TabView` to have a nice black accent on the selected tab item.

Now in `ContentView` replace `Text("Hello world!")` by your new `MainView()`.

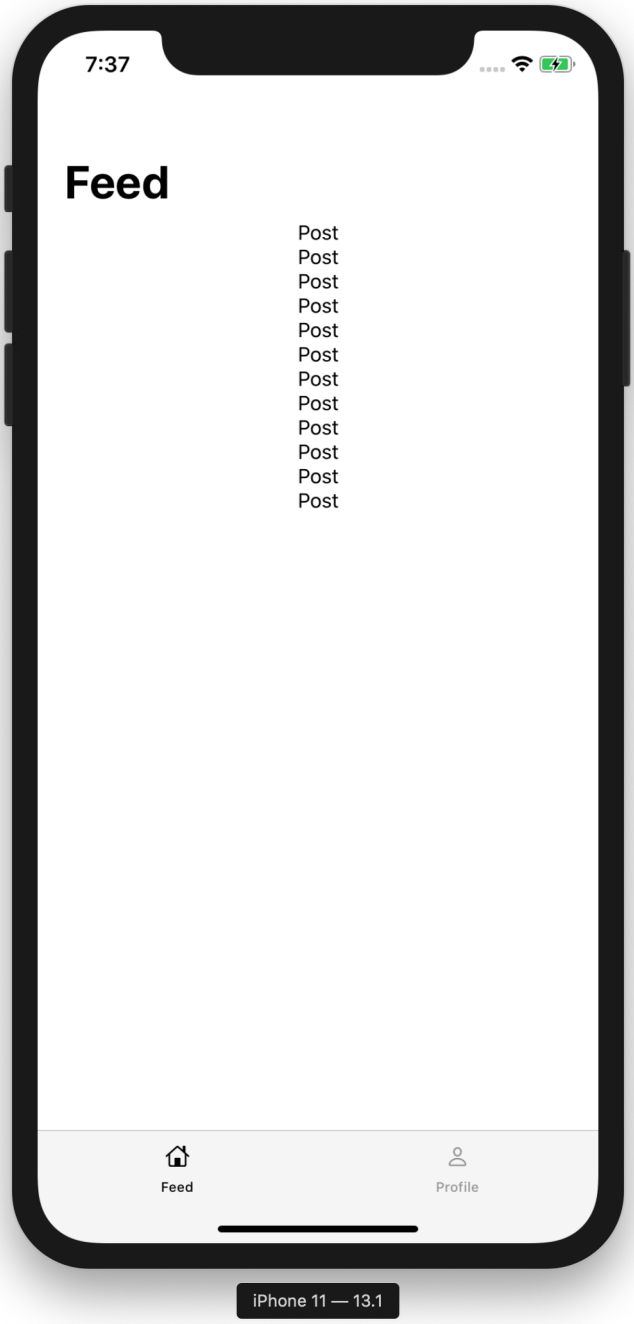
To make the text for the whole application nicely sized, apply a `.font` modifier on the `MainView` with the system font of size 15.

2. Feed basics

Now let's create the Feed view. The feed simply displays a list of posts that are provided by the `FeedStore` class that you can find in the `Stores` directory.

The store uses mocked data, because why not.

When you're done the UI should look like the screenshot below.



The data

To use data we need to provide it somehow to the `FeedView`. Fortunately this is very simple to do. We're not going to go into the details today but let's imagine a View can **observe** some object and update itself on its own when the data changed (e.g. when it's fetched from the API). To implement such behavior you will need the `@ObservedObject` property wrapper.

In your feed let's add a private variable annotated with `@ObservedObject` named `feedStore` of type `FeedStore`.

This property needs to be initialized somehow so let's also add an `init` that will take one parameter named `user` of type `User?`.

Within the initialized instantiate the `feedStore` property.

This will let us display either the full feed (when `user == nil`) or user-specific feed (when `user != nil`).

The UI

The UI here is quite simple...

Typically you would use `List` View (an equivalent of `UITableview`) for this but currently there is no way of removing the separators. Instead we will use a `ScrollView` while iterating over our content.

Let's add ScrollView as the top view and make it scroll vertically by providing it one unnamed parameter with value of `.vertical`.

(Yeah, the docs suck... This is unfortunately common with SwiftUI as of today.)

Now we need to display our posts in the `ScrollView`.

To do that let's add a ForEach. `ForEach` lets us enumerate data and create a set of Views based on it.

Note: Since we're displaying dynamic data the previews will not show the posts. This can be fixed but let's not do it today. To test your UI simply run the app in a simulator.

In this `ForEach` let's iterate over `feedStore`'s `posts` property and let's display the `PostView` for each one of them.

By the way... `ForEach`'s trailing closure takes one parameter which is the current element. This will be your `post`.

Your View hierarchy should now look similar to this:

```
ScrollView
  ForEach
    PostView
```

We're in the `NavigationView` context so we need to provide the title for this screen.

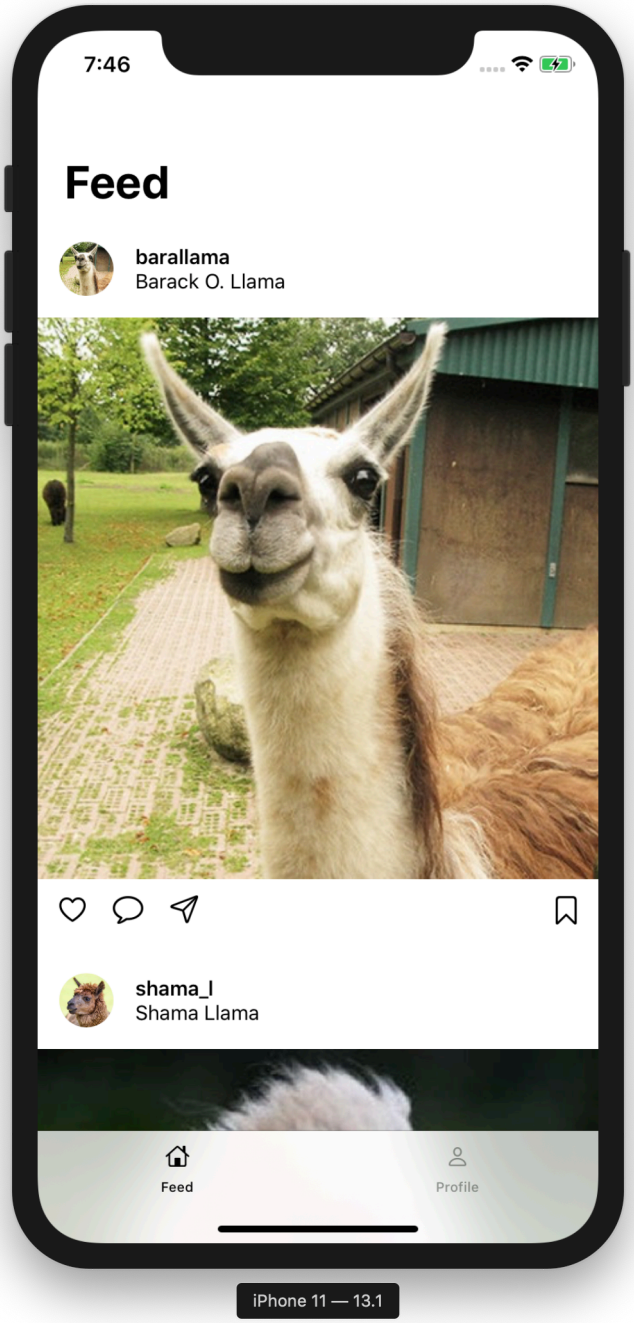
To do that let's apply `.navigationBarTitle()` modifier with a value of `Feed` on the topmost element of `FeedView` (e.g. the `ScrollView`).

Last but not least remember to add your `FeedView` into your `MainView` so that it gets displayed. Provide `nil` as the `user` parameter's value.

3. Displaying posts

Now is the time to make our `PostView` great again.

When you're done the UI should look like the screenshot below.



The data

Our `PostView` needs data and fortunately passing data between views is extremely simple. Just provide it in the initializer! And since all Views are `struct` and structs have synthetised initializers, if we don't want to do anything more complex, we just need to add the property.

Let's add a `let` property named `post` of type `Post` to our `PostView`.

Now the code will break but the only thing we need to do to fix it is to go back to `FeedView` and provide `post` as parameter of `PostView`.

That's it. We can now use the post data in our View.

The UI

`Postview`s UI is where things get fun!

Look at the screenshot above. The `PostView` consists of three different sections (from the top to bottom):

- Profile information
- Photo
- Toolbar

To construct this View we will need a `VStack` that lets us stack Views vertically.

Add a `VStack` with alignment of `.leading` and spacing of 0.

Profile information

The profile information will consist of a `HStack` with an `Image` and `VStack` (with two Text views). The View hierarchy should thus look like this:

```
HStack
  Image
  VStack
    Text
    Text
```

This is because the profile photo needs to be positioned at the leading edge and the two Text Views need to be positioned right next to it, one under the other.

First create the `HStack` with spacing of 16. `HStack` is similar to a `VStack`. The only difference is that it stack views horizontally.

In the `HStack` place an image of the user (`post.user.imageName` property).

To make the image look good we need to set some modifiers on it:

- `resizable()` to make the image be able to resize itself
- `clipShape()` with value of `Circle()` to make the image round
- `frame()` with both `width` and `height` of 40 to make the image correct size

Now add the `VStack` with `alignment` of `.leading`. The Text views should have values of `post.user.nickname` and `post.user.name`.

Finally the top Text view should have a `fontWeight` modifier applied to it with the value of `semibold`.

For added beauty, also apply `padding` modifier without any parameters to the `HStack` to get a default padding around the view.

Photo

The photo part is super-simple. Let's add an `Image` with a value of `post.imageName`.

To make it size properly, let's add two View modifiers to it:

- `resizable()` to make it be able to resize itself
- `aspectRatio()` with `contentMode` off `.fill` to make it fill its frame

Toolbar

The toolbar is basically just 4 icons next to each another. And by now you know how to stack things!

Let's add an `HStack` with `spacing` of 20.

In the `VStack` add 4 `Image`s with symbols **heart**, **message**, **paperplane** and **bookmark**.

To make the last image be pushed all the way to the trailing edge, add a `Spacer` view between the last two `Image`s.

The last thing to do is to apply a `font` view modifier with system font of size 20 to the `HStack` to make the icons nicely sized.

And again added beauty, also apply `padding` modifier without any parameters to the `HStack` to get a default padding around the view.

Wrapping up

This was a lot of code (well, at least as SwiftUI is concerned)! Our view is beautiful now though and so it was definitely worth it. Compare it to the code you'd have to write in UIKit!

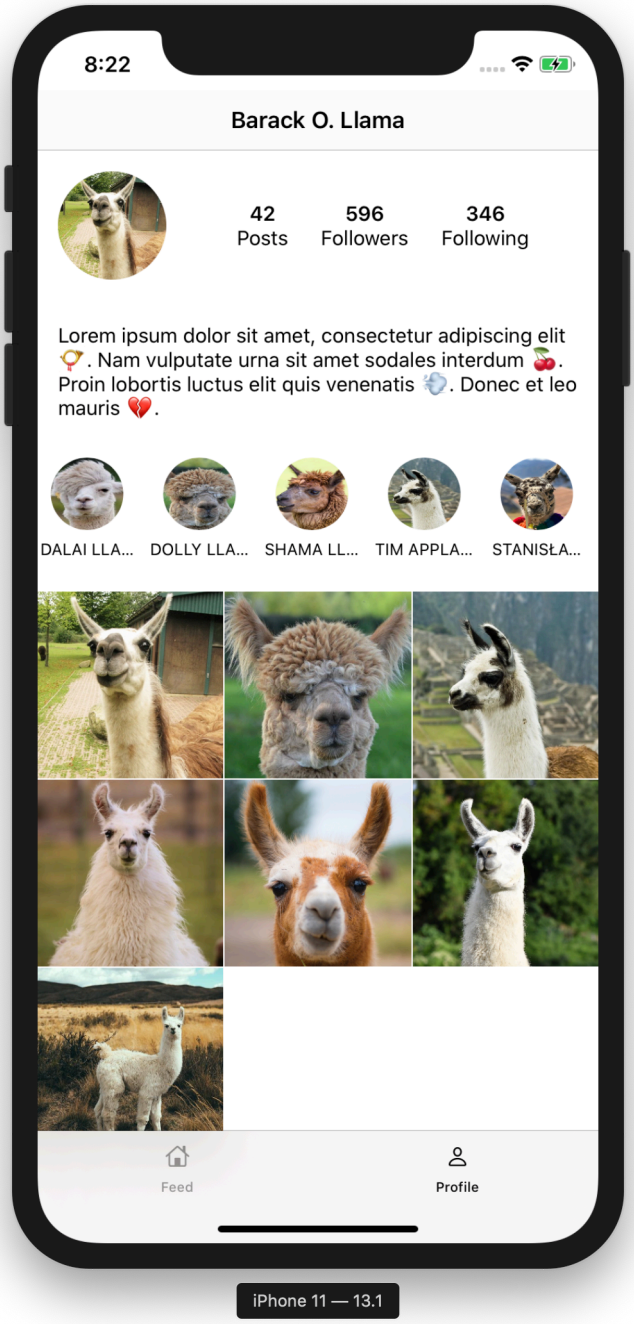
Your `PostView` view hierarchy should now look like this:

```
VStack
  HStack
    Image
    VStack
      Text
      Text
  Image
  HStack
    Image
    Image
    Image
    Spacer
    Image
```

4. Profile

We're done with the feed for now so let's work on the profile.

When you're finished the UI should look like the screenshot below.



This is **a lot** of UI so we'll need to split this view into multiple views so that we do not create a monster-view (remember and fear the Massive View Controller).

In fact we will need to create three views for each of the logical sections:

- **Header** with the photo and statistics
- **Friends list** with the list of user's friends
- **Photo grid** with the photos the user posted

Let's start simple by creating a new view and calling it `ProfileView`.

Remember to display this new view in our `MainView` instead of the `Text` placeholder.

The data

Our `ProfileView` shows user data and so we need to provide some `User` object to it.

We've done this before so let's just add a `user` property of type `User` to this view.

The tricky part here is to get the current user. To simulate authentication, I have provided a dummy `AuthenticatedUserStore` class. We do not need to care about the details for now.

Since the `ProfileView` now needs the user to be provided to it, we need to access the relevant data store in the `MainView`.

In SwiftUI you can pass data to Views in multiple ways...

- Passing them using `init`
- Using `@ObservedObject` property wrapper
- Using `@EnvironmentObbject` property wrapper

Environment objects are similar to Observed objects. The difference is where the stored are initialized.

When we implemented the `feedStore` using `@ObservedObject` in `FeedView` we had to initialize the store in the View. When using the `@EnvironmentObject` we rely on one of ancestor views to initialize the store instead.

For authenticated user this is nice as we want this user to be available throughout the app's view hierarchy.

In our case we initialize the store in the `SceneDelegate`. Feel free to look at how it's done using `environmentObject()` property wrapper. When you create an environment object like this, all children of this view will have access to the object using the `@EnvironmentObject` property wrapper.

Let's do it!

At the top of `MainView` add a `private var` property named `authenticatedUserStore` of type `AuthenticatedUserStore` and annotate it with `@EnvironmentObject` property wrapper.

Now simply pass the `authenticatedUserStore.user` property to the `ProfileView`.

Let's also force-unwrap it as we like to live dangerously and if the user is here unauthenticated, we should crash anyway.

Note: your previews will crash. To fix this issue add the environment object to your previews as well for whichever view you're using the `@EnvironmentObject` property in. For example:

```
struct MainView_Previews: PreviewProvider {
    static var previews: some View {
        MainView()
            .environmentObject(
                AuthenticatedUserStore(user: MockData.users.first!)
            )
    }
}
```

You will need to do this for all views that are ancestors of this view too (e.g. `MainView` and `ContentView`) to make the previews work. Simply add it when you see a crash.

The UI

Let's now create the skeleton of our `ProfileView`.

Since there may be many photos the view will have to scroll so let's add a vertically scrolling `ScrollView` as the topmost view.

Let's also create three empty views (add some placeholders to their bodies) named `ProfileHeaderView`, `ProfileFriendsView` and `ProfilePostsView`.

Apply `padding` of 16 to the `ProfileHeaderView` and bottom padding of 16 to the `ProfileFriendsView`. Hint: the first parameter of `padding` property wrapper can be the edge (or set of edges) to which you want to apply the padding.

Lastly apply `navigationBarTitle` modifier to the `ScrollView` with value of `user.name` and `displayMode` of `.inline`. This time you will need to wrap the value in `Text` to silence the compiler error.

At the end your view hierarchy should look like this:

```
ScrollView
  ProfileHeaderView
  ProfileFriendsView
  ProfilePostsView
```

5. Profile header

Let's build the profile header view now.

When you're finished the UI should look like the screenshot below.

The `ProfileHeaderView` needs to display user information and so we need to pass the user instance to it. You already know how to do this.

The UI is a bit complex but you already know everything from the previous exercises too!

I'm going to leave the coding to you but I'll give you a couple of hints:

- All the data you need is available on the `User` object.
- The spacing of `VStack` is 32.
- The spacing of `HStack` is 16.
- The width and height of the `Image` is 80.
- You need to apply two modifiers to the long text to size it properly:
  - `.fixedSize(horizontal: false, vertical: true)`
  - `.frame(maxWidth: .infinity, alignment: .leading)`

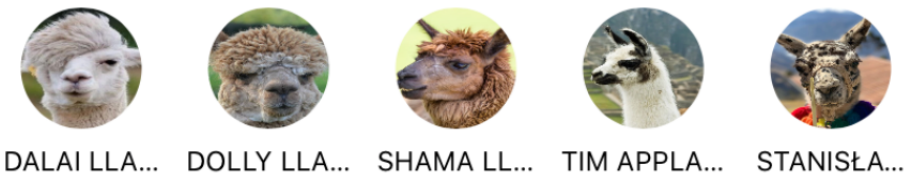
If you become lost here's a little hint of what your view hierarchy should look like:

```
VStack
  HStack
    Image
    Spacer
    HStack
      VStack
        Text
        Text
      VStack
        Text
        Text
      VStack
        Text
        Text
      Spacer
  Text
```

6. Profile friends

Now we need to build the friends view.

When you're finished the UI should look like the screenshot below.



The `ProfileFriendsView` needs to display a horizontally scrolling list of user's friends.

You know the drill! You can build it on your own.

I'm going to give you a few hints again:

- The list of friends is available on the `User` object.

- You will need to wrap the `ForEach` in a `HStack`.
- Both width and height of the `VStack` should be 75.
- You don't need to set any other frames.
- The font of friend's name is `.caption`.
- `ScrollView` has an additional parameter called `showsIndicators` that should be set to `false`.

If you become lost here's a little hint of what your view hierarchy should look like:

```
ScrollView
  HStack
    ForEach
      VStack
        Image
        Text
```

7. Profile photos

The last view we need to build for our profile to be complete is the post photo grid.

When you're finished the UI should look like the screenshot below.

This is where things get a bit complicated...

In UIKit you'd use `UICollectionView` to build a view like this. Unfortunately even though there's a `List` counterpart to `UITableView` there is no SwiftUI counterpart to `UICollectionView`.

In SwiftUI we will need to use an `VStack` of `HStacks` to build a grid like this.

The problem is sizing though. We need to size each of our `Image`s to 1/3 of the available horizontal space to lay them our properly. So far we haven't seen how to do this as views on their own do not expose their size to us.

To let them know what size our `Image`s should be we will need to use GeometryReader. The `GeometryReader` is a container view that let's us figure out what its dimensions are.

Let's use it!

In `ProfileView` wrap the `ScrollView` in `GeometryReader` so that your view hierarchy looks like this:

```
GeometryReader
  ScrollView
    ProfileHeaderView
    ProfileFriendsView
    ProfilePostsView
```

Note that the `GeometryView`s trailing closure accepts one parameter (let's name it `geometry`) that gives us the dimensions of the container.

To use it, in `ProfilePostsView` add a `width` property of type `CGFloat` and provide it to the view in `ProfileView` by using `geometry.size.width`.

The rest is easy...

In the `ProfilePostsView` create a grid structure like this:

```
VStack
  ForEach
    HStack
      ForEach
        Image
```

Let me give you a couple of hints again:

- Both stacks have `spacing` of 1.
- The first `ForEach` iterates over `user.posts.chunked(into: 3)` which is a convenience function I prepared for you to get the posts in arrays of 3.
- The second `ForEach` iterates over the `Chunk`'s items.
- The frame of the `Image` should be the width passed to the view divided by 3.
- The frame of the `HStack` should have `maxWidth` set to `.infinity` and `alignment` to `.leading` so that everything displays nicely even when there is less than 3 images.

And that's it for the friends view.

By now your `ProfileView` should be looking swell!

8. Navigation

Now it's time to create our "segues".

Feed post to user profile

As described at the top of this document, tapping on the profile information in the feed should transition us to author's profile.

In SwiftUI this is very simple to do!

Simply go to the `PostView` struct and wrap the `HStack` containing the profile information in a NavigationLink. A `NavigationLink` is a bbutton that when pressed transitions the user to its `destination` view.

In our example the `destination` should be `ProfileView(user: post.user)` and the trailing closure should contain the `HStack` we wrapped.

You will probably see that our `Image` has become black. This is due to the default button styling applied by SwiftUI. You can fix this bby applying a `buttonStyle` view modifier on the `NavigationLink` with value of `PlainButtonStyle()`.

User profile friend photo to user profile

Similarly - as described at the top of this document - tapping on the friend's photo in user profile should transition us to the author's profile.

Apply the same approach as above to the `VStack` containing the `Image` and `Text` in the `ProfileFriendsView`.

User profile post photo to feed

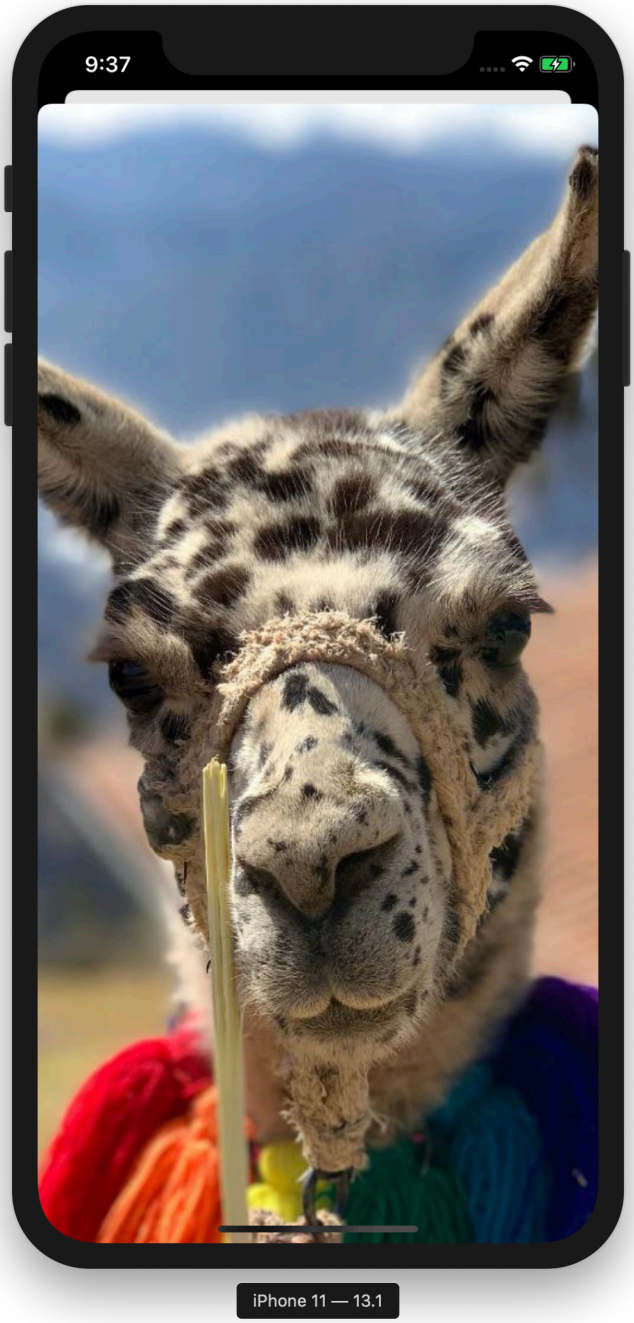
Lastly - again as described at the top of this document - tapping on the post photo in user profile should transition us to the author's feed.

Apply the same approach as above to the `Image` in the `ProfilePostsView`.

9. Displaying photo detail in feed

One last thing we need to do is to display a modal view with a detail of the photo when user taps it in the feed.

When you're finished the UI should look like the screenshot below.



By know you'd probably think that this will be very easy in SwiftUI. And you'd be right!

To implement this modal sheet we need to learn about one last piece of information about data storage in SwiftUI. That last piece of today's puzzle is the `@State` property wrapper.

Properties annotated with `@State` are used to store, well, a state of a SwiftUI view. SwiftUI monitors changes of `@State` properties and whenever such property changes SwiftUI invalidates the containing views's appearance and recomputes its `body`.

By the way... Thanks to the strongly typed nature of SwiftUI this is all done very efficiently and only views that need to change are actually re-rendered.

Sheets in SwiftUI are displayed using the `sheet` view modifier. This view modifier can be applied to any view and multiple sheets can be handled from one view.

This `sheet` view modifier accepts one `Bool` parameter named `isPresented` that decides whether the sheet should be displayed or not. The parameter value is not just a value itself though. It is something called a Binding which is basically a two-way connection between a view and its underlying model. This two-way connection is needed as not only - in our case - must a sheet know when to display itself based on the `@State` property value... It also needs to set the value to `false` when user dismissed the sheet by dragging.

To implement the sheet in our application we need to go to our `PostView` and a `@State` property named `modalIsPresented` with a default value of `false`. This will be the property that decides whether our sheet should be displayed.

Now wrap the `Image` in a `Button` view.

A button in `SwiftUI` takes an `action` closure that defines what should happen and a view builder closure that defines the contents of the button (similar to other views we used today).

Set the button's action to `self.photoModalIsPresented = true` and apply the `buttonStyle()` modifier to it so that the image is not black.

Lastly apply a `sheet()` view modifier to the button and set its `isPresented` parameter to `self.$photoModalIsPresented`. Notice the dollar sign that is used to access the two-way binding.

The trailing closure of our `sheet` should contain the same `Image` that the button contains. Additionally apply a `edgesIgnoringSafeArea` with value of `.bottom` to make it extra-beautiful.

Your code should now look something like this:

```
Button
  Image
.sheet
  Image
```

10. Likes

Now that our UI is done, let's play with data storage!

As mentioned at the top of this document, when user taps on the heart icon on a post in the feed the post should become liked. These likes should persist throughout the app and so if you like a post in the main feed it should also be marked as liked in that user's feed.

To implement this functionality let's use a naïve local storage that only lasts while the app is running. Let's use `@EnvironmentObject` to implement it in this example project. In real world you'd probably trigger a request to some API and this would not be the way to handle such scenario. Today we just want to keep it as simple as possible.

To handle the data layer I have prepared for you the `LikedPostsStore` that is very similar to other stores that we used today.

There are two functions on it that are of interest to us:

- `isLiked(post: Post) -> Bool` - to find whether a post is liked
- `toggle(post: Post)` - to like and unlike a post

The liking/unliking happens in our `PostView`. Let's first create the button.

First add the our `@EnvironmentObject` property. Call it for example `likedPostsStore`.

Now wrap the **heart** image in a button and set its action to call the `toggle` function on the store you just added.

Lastly we need to make the image react to the liked/unliked state. In the `Image` initializer add a condition and based on the `isLiked` function call result display either **heart** or **heart.fill** image.

Now run the app, tap the heart icon and see your app crash. Remember the `environmentObject` call in `SceneDelegate` we talked about earlier?

Add the environment object and your app will work like a charm!

Note: your previews will crash again. To fix this issue add the environment object to your previews as well for whichever view you're using the `@EnvironmentObject` property in. For example:

```
struct PostView_Previews: PreviewProvider {
    static var previews: some View {
        PostView(
            post: MockData.posts.first!
        )
        .environmentObject(LikedPostsStore())
    }
}
```

11. Bonus task #1

You may have noticed that the round profile photo is duplicated in both `ProfileHeaderView` and `PostView`.

Either create a custom view modifier or extract this piece of UI into its own view.

Google is your friend!

12 Bonus task #2

Are you done? You either played with SwiftUI before or you're very fast.

NICE!

Here's an ultimate bonus task.

Go to `FeedStore`, comment out the line setting posts and uncomment the piece of code marked as `Bonus task`.

Now run the app and see that there is a 5 second delay before the data is displayed.

Your task is to implement a loading indicator view that looks like the one on the video named `loading_indicator.mov` and integrate it to `FeedView`.

For now just implement it when `feedStore.posts.isEmpty`. You don't have to introduce any state management.

Here are some hints, in no particular order...

- You'll need `@State`.
- `Group` is your friend.
- You can use `if...else`.
- `withAnimation` is cool.
- `frame(maxWidth:maxHeight:)` helps
- The view both resizes and blurs.

The end

If you made it all the way here, you are brave!

Thank you for attending this workshop and keep coding :).