HACKING WITH SWIFT

Practical iOS 10

COMPLETE TUTORIAL COURSE

Learn to develop apps for iOS 10 by building real-world projects

FREE SAMPLE

Paul Hudson

Chapter 1
Happy Days

Setting up

In this project we're going to build Happy Days: an app that stores photos the user has selected, along with voice recordings recounting what's in the picture.

To add some spice to the project, we'll be using the new iOS 10 Speech framework to transcribe the user's voice recordings, then store that transcription in Core Spotlight. iOS 10 also gives us the ability to search our Spotlight index inside the app using the new `CSSearchQuery` class, so we'll be drawing on that too.

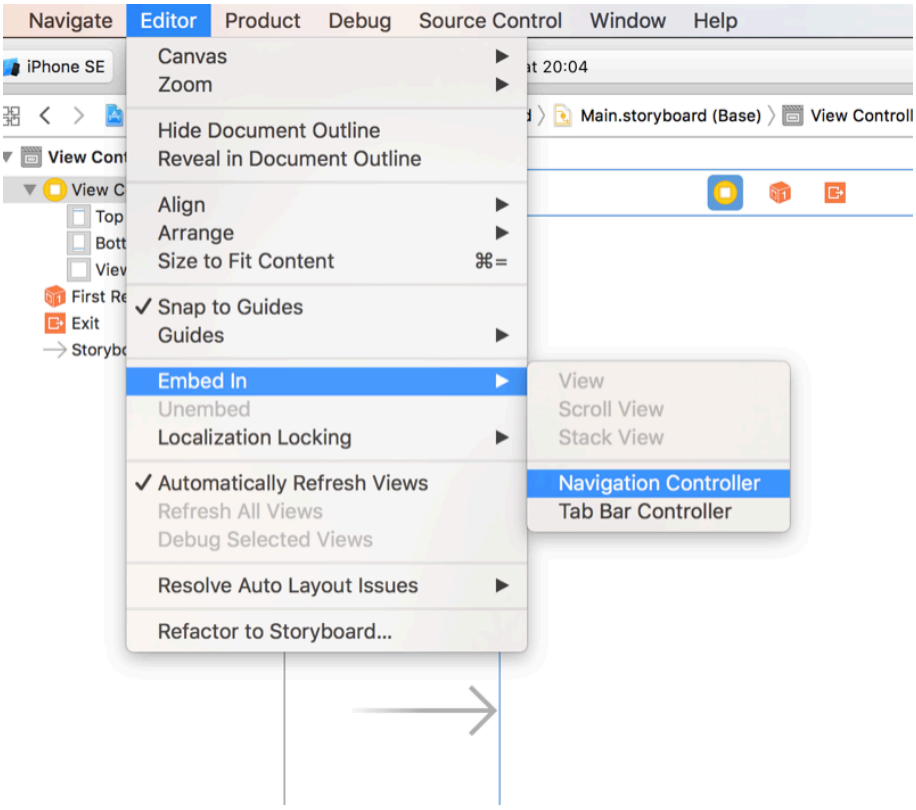
Launch Xcode 8, and create a new project using the Single View Application template. Give it the name HappyDays, then set the device to Universal. All set for some Swift 3? Let's do it!

Building the user interface

Happy Days is going to display the user's photos inside a collection view controller, which will itself be inside a navigation controller. It will also need a second view controller for asking permissions when the app first runs: we need to access the user's photo library and microphone, and also request permission to transcribe their speech – Apple are predictably cautious about apps transcribing speech without user permission!

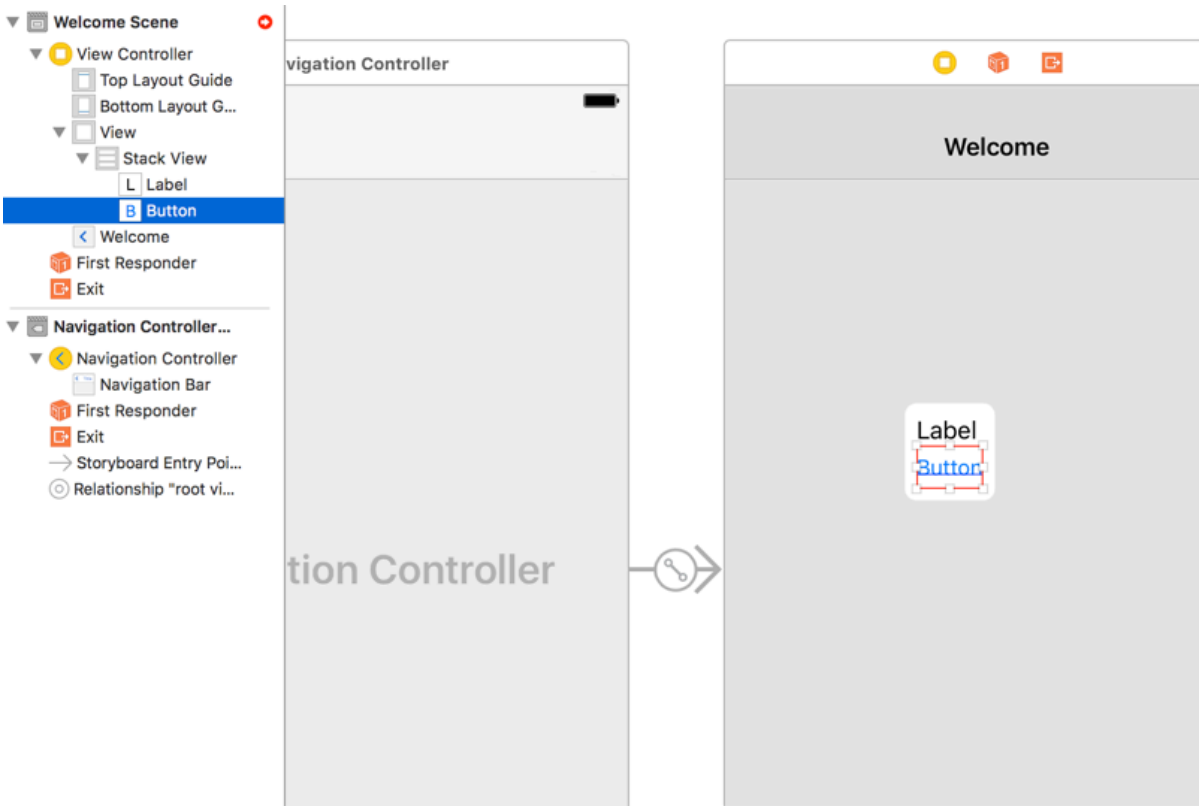
The Single View Application template gave us a regular view controller, so we can use that for our permissions view controller. You could, if you wanted, do without this and just request permissions on demand, but because we're asking for three at once I found it was a more pleasant experience to ask for them when the app first launches rather than confuse the user with requests later on.

Let's start with the permissions first run screen, because it's easy enough. Select the current view controller, and embed it inside a navigation controller by going to Editor > Embed In > Navigation Controller. You'll see a simulated navigation bar appear in the view controller – double-click in the center of that to edit the title, and write "Welcome".

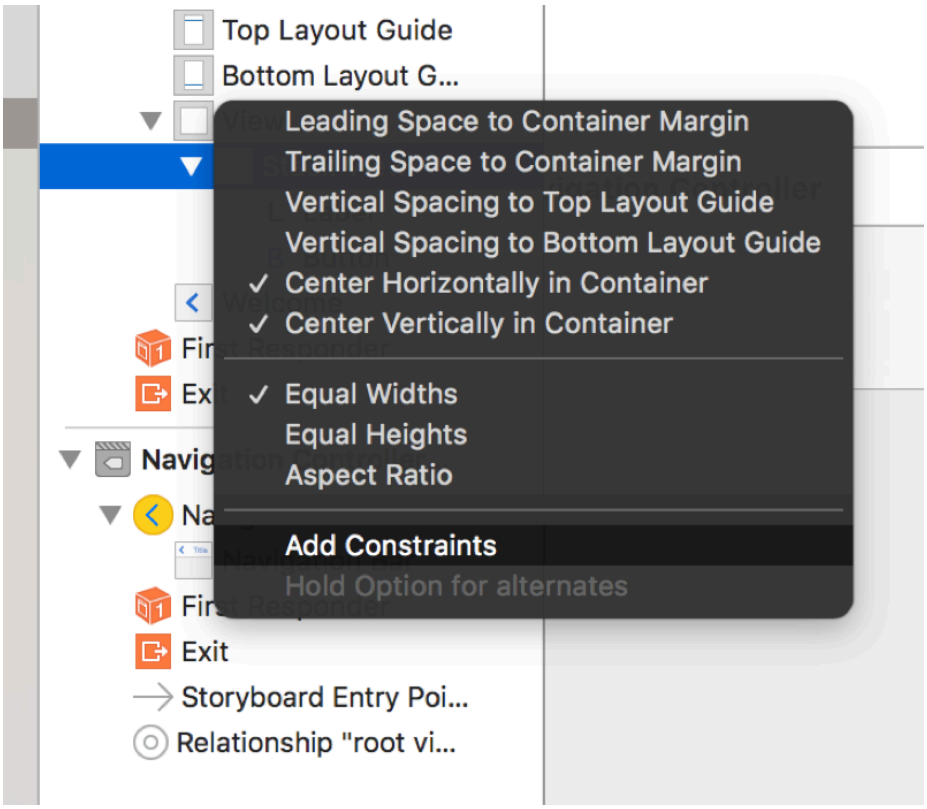


Next, drop a Vertical Stack View object into the view controller anywhere inside – we'll be using this to host a label describing what's going on, as well as a button let users proceed. Having these two inside a stack view allows us to center the stack inside the parent view, even when the text resizes.

Drag a new label into the stack view, and you'll see the stack view shrink so that it wraps itself around the label. Please also drag a button into the stack view, below the label – again you'll see the stack view resize so that it fits them both neatly.



We want to position the stack view so that it fits neatly onto the screen. To do that, Ctrl-drag from the stack view to the parent view to create constraints, and select Center Horizontally in Container, Center Vertically in Container. Finally, then Equal Widths. Having the stack view fill the view horizontally doesn't look great, though, so we're going to make a tiny change to that last constraint to pull it in from the edges.



Make sure the stack view is selected, then go to the size inspector (Alt+Cmd+5) so you can see the constraints we just created. Look for the one marked "Equal width to: Superview" and click the Edit button next to it. In the popup that appears, enter -40 for the Constant value so that the stack view is 40 points slimmer than its parent.

That's all our layout done, but before we update the frames we need to make a couple more changes. First, select the label:

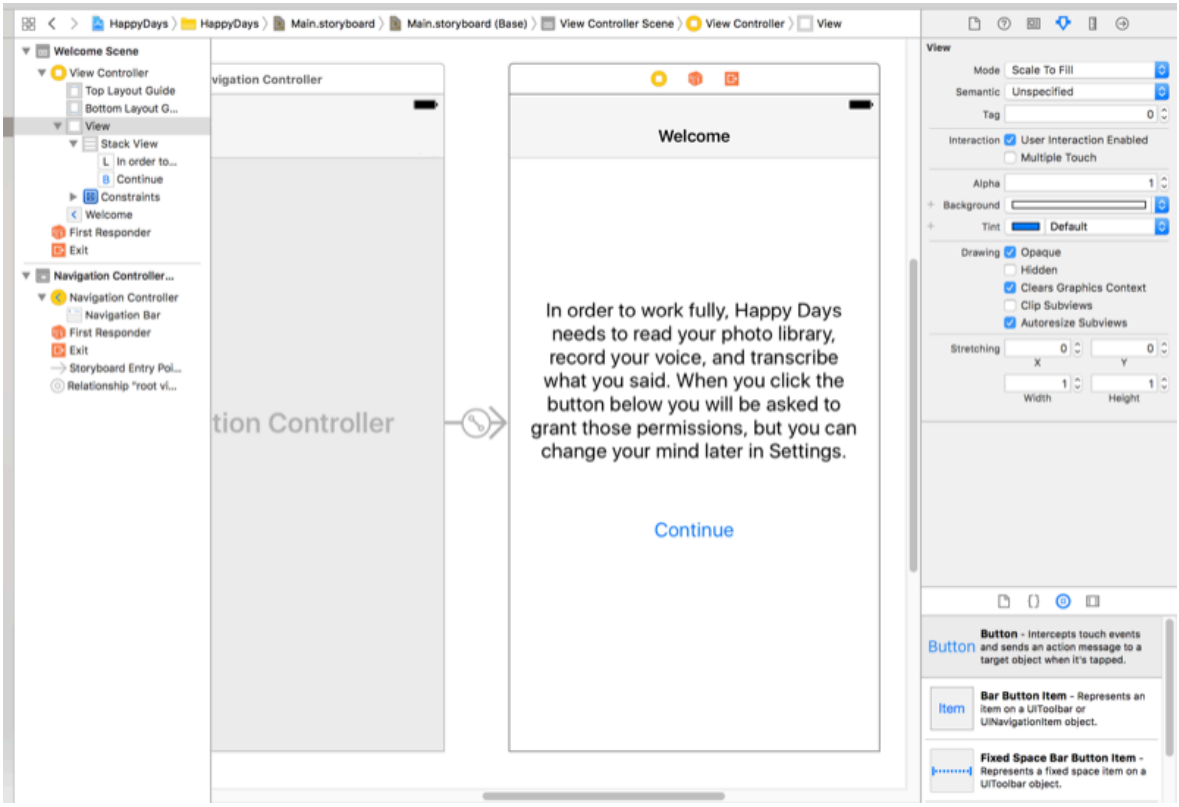
- Set its font size to 20.
- Set its text to be center aligned.
- Set its Lines value to 0.

Now give it this text: "In order to work fully, Happy Days needs to read your photo library, record your voice, and transcribe what you said. When you click the button below you will be asked to grant those permissions, but you can change your mind later in Settings."

Second: select the button, then set its font size to 25 and its text to "Continue".

By default stack views place their arranged views directly adjacent to each other, which doesn't look that great. To fix that, change Spacing to be 50 and you'll see the label and button jump apart.

That's all our layout done, so click in the whitespace above the label to select the main view. Now go to the Editor menu and choose Resolve Auto Layout Issues > Update Frames – make sure you choose the one under "All Views in View Controller".

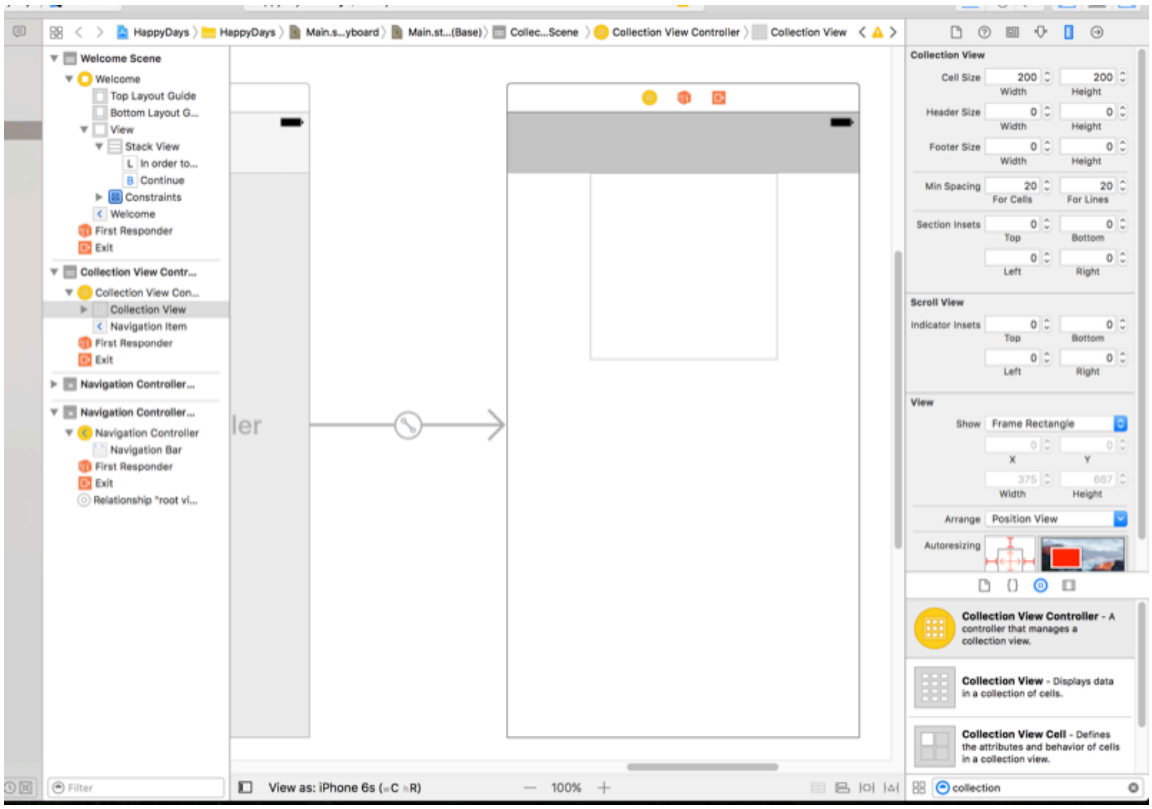


Creating the main view controller

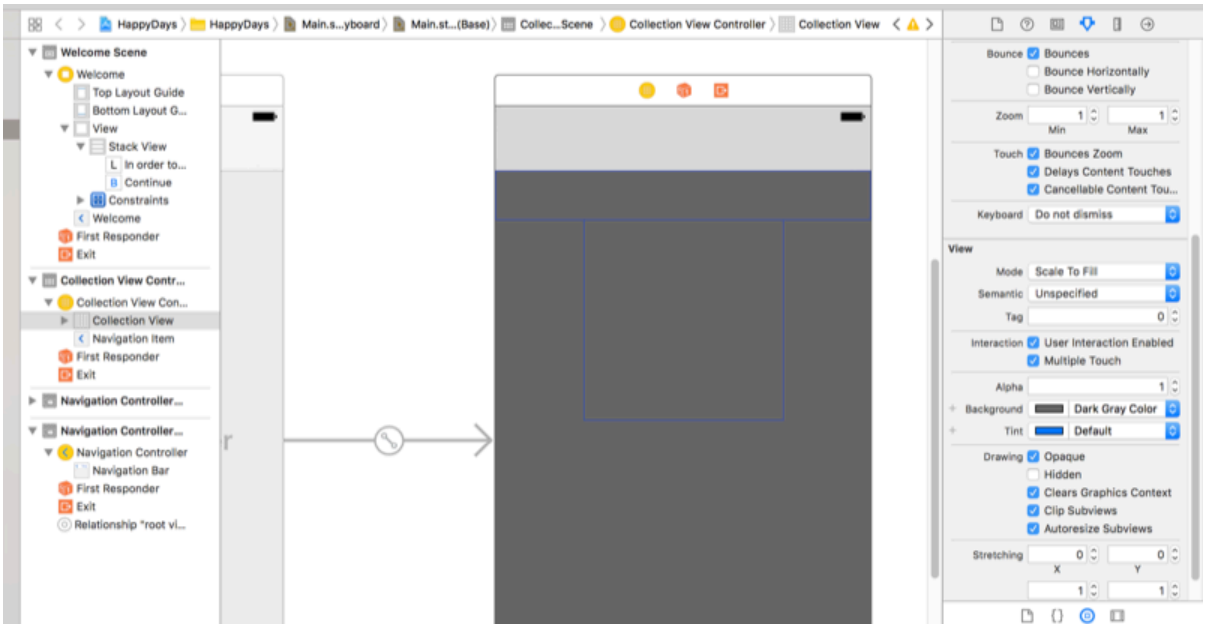
We're done with the first run screen for the time being – we'll come back to it soon enough to hook it up to code. Now let's turn to the main view controller: the part that lets the user interact with their memories.

Start by dragging a new collection view controller onto your storyboard – that's the one in a yellow circle, not the plain collection view. Again, please embed it in a navigation controller: click the yellow circle directly above the new view, then choose Editor > Embed In > Navigation Controller.

We get one collection view cell prototype by default, but it's pretty small. To make it larger, click the large white space in the view to select the collection view, then go to the size inspector and change Cell Size from 50x50 to 200x200. While you're there, change Min Spacing to 20 and 20 for the "For Cells" and "For Lines" fields. We'll be using that to show photos.



Go back to the attributes inspector for the collection view, then check the box marked Section Header – we'll be using *that* to hold our search box. But first: give the collection view a dark gray background color, so that the images we load are prominent.

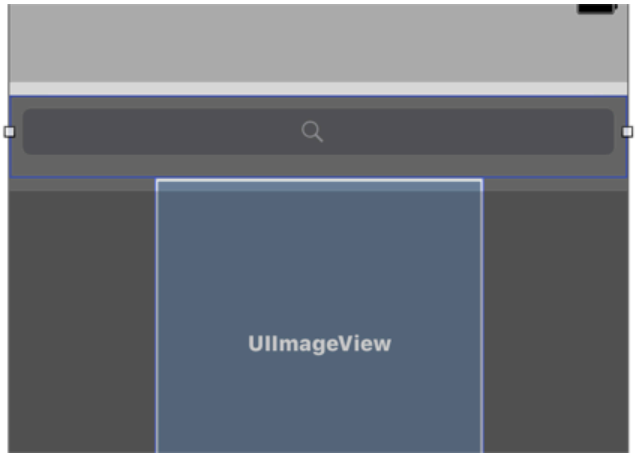


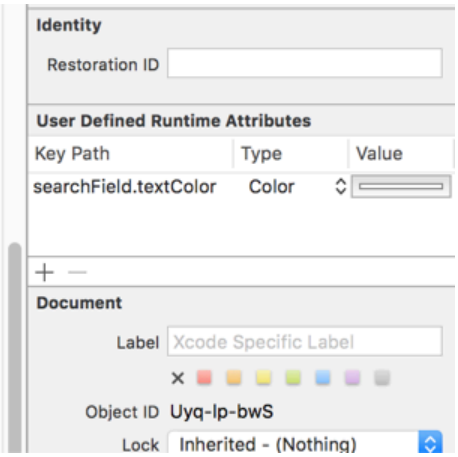
Right now we have one 200x200 collection view cell prototype taking up most of the screen. This is going to be used to hold the photos in our collection view, so please add an image view to it now. Xcode will make the image view a completely different size and shape to the cell – the easiest way to fix that is by going to the attributes inspector and hand-typing the values 0 for X and Y, and 200 for Width and Height. Now select the new image view in the document outline, then go to Editor > Resolve Auto Layout Issues > Add Missing Constraints. Technically the image view was already selected, but Xcode occasionally gets confused when you've been typing into one of the inspectors!

As for the collection view's section header, please drag a search bar object into there. These things are a fixed size, but you might need to ensure it has its X and Y position set to 0. I found that changing the Search Style to Minimal gave the best look, but you're welcome to experiment. If you always want to go with the minimal style, you should know that it defaults to black text, which looks pretty poor on a dark gray background. There's no color picker to change that in Xcode, so instead we're going to modify a key path using the identity inspector – something you don't really have to do very much, thankfully.

Make sure the search bar is selected, then press Alt+Cmd+3 to activate the identity inspector. Under "User Defined Runtime Attributes" click the + button, then double-click where it says "keyPath" in the newly inserted row. Replace "keyPath" with "searchField.textColor", then change Type from "Boolean" to "Color". Doing that will change the Value field from a

checkbox to a color well, so click that now and change it to white. It might seem like a bit of a hack, but that's the only way to do it without poking around in code!





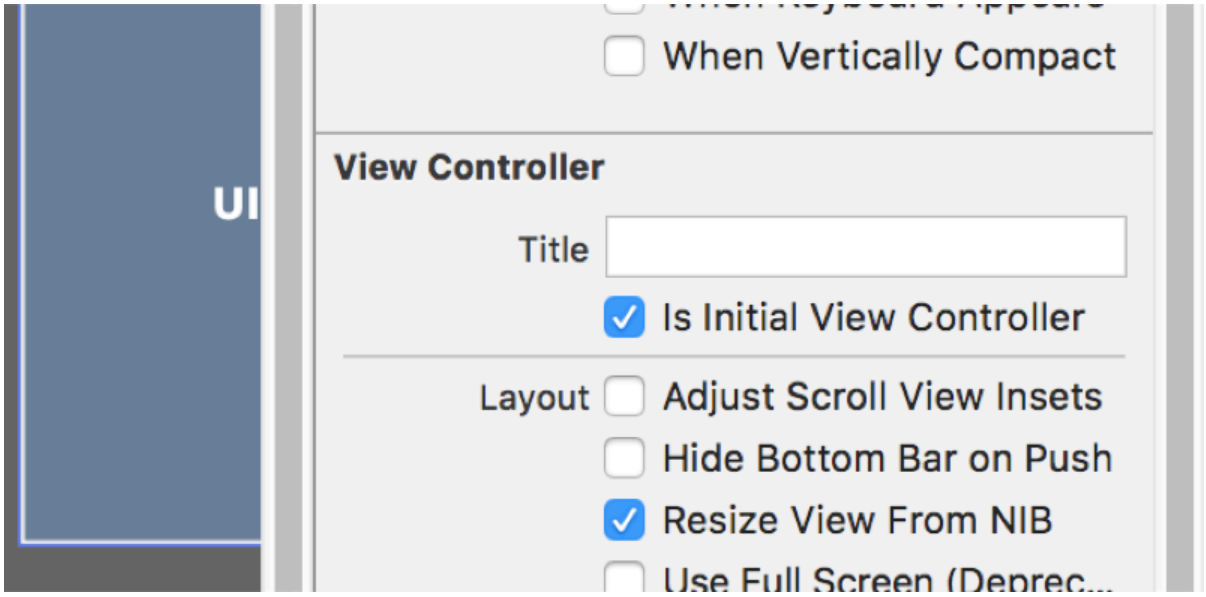
Finally, double-click the simulated navigation bar and give it the title Happy Days.

Connecting the interface to code

Although the drag-and-drop parts of our user interface are done, but we still need to make a few changes so that everything sits together neatly:

1. Ensure the collection view's navigation controller is the initial view controller.
2. Set a delegate for the search bar so we can act on the user's typing.
3. Configure re-use identifiers for the collection view cell and section header.
4. Give the first run screen a storyboard identifier so we can create it on demand.
5. Create a class for the collection view controller.
6. Create a class for the collection view cell that will hold images so that we can reference the image view in code.
7. Set up outlets and actions.

Let's start with number 1: making the collection view's navigation controller the initial view controller for the app. That's trivial: click the navigation controller, select the attributes inspector, then check the box marked Is Initial View Controller. One down!



Number 2: set a delegate for the search bar so that we can act on the user's typing. To make this happen, Ctrl-drag from the search bar to Collection View Controller in the document outline, then select Delegate from the popup that appears. Boom: two down. Who said coding was hard?

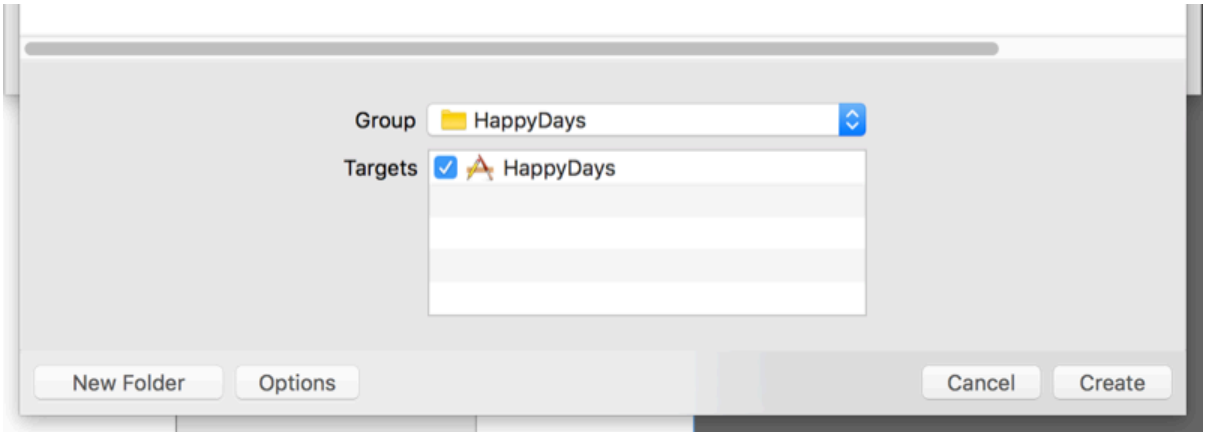
Number 3: configure re-use identifiers for the collection view cell and section header. Because both our cell and header have child views that occupy their full space, you should use the document to select each item. So, select Collection Reusable View then enter "Header" for the Identifier value in the attributes inspector. Then select Collection View Cell and give it the identifier "Memory".

Number 4: give the first run screen a storyboard identifier so we can create it on demand. To do this, select the navigation controller that wraps the first run screen, then choose the identity inspector. Look for the Storyboard ID field in there, and give it the value FirstRun.

Number 5: create a class for the collection view controller. This is the first non-trivial task, but even then we we can re-use the existing ViewController.swift file given to us by the template.

To create a class for the collection view controller, choose File > New > File, then select iOS > Source > Cocoa Touch Class and click Next. Make the new class inherit from **`UIViewController`**, give it the name MemoriesViewController, then click Next again. In the final screen, make sure Group is set to Happy Days with the yellow folder icon next to it,

then click Create.



Note: if you're paying attention, you should have read that last paragraph and thought "why are we subclassing **`MemoriesViewController`** from **`UIViewController`** when it's clearly a **`UICollectionViewController`**?" The answer is simple: Xcode's default code for **`UICollectionViewController`** subclasses is suboptimal, and we can do better.

Now let's connect that new class to our user interface. Because we created a regular **`UIViewController`** subclass, we need to make one tiny tweak in MemoriesViewController.swift before we can go into IB. Open that file, and change this line:

```
class MemoriesViewController: UIViewController {
```

To this:

```
class MemoriesViewController: UICollectionViewController {
```

Now open Main.storyboard in Interface Builder, and select the Collection View Controller. We need to tell IB that this view controller is an instance of the **`MemoriesViewController`** we just created, so go to the identity inspector and change Class to "MemoriesViewController". That's task number 5 done!

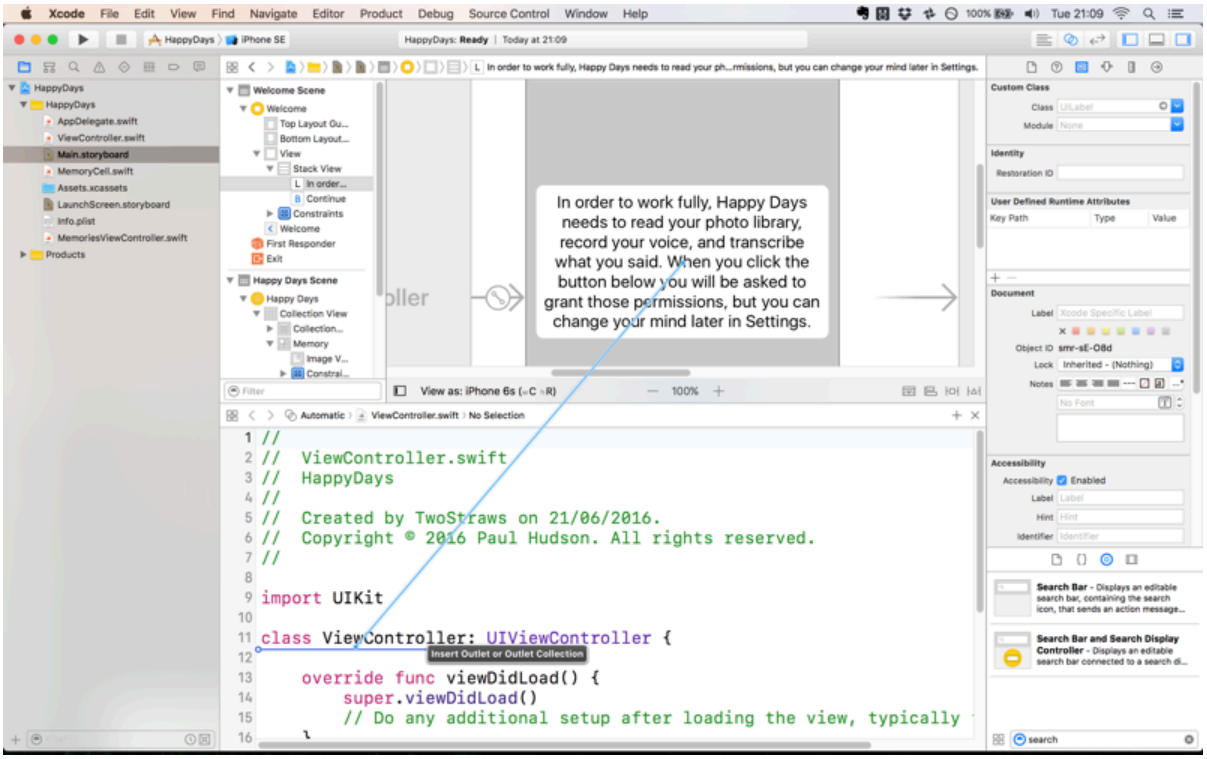
On to task number 6: create a class for the collection view cell that will hold images so that we can reference the image view in code. This isn't strictly needed because the cell is so trivial we could just use a tag, but later on you'll want to expand this so adding a full class is a smart move.

Go to File > New > File, then choose iOS > Source > Cocoa Touch Class and press Next. Make it a subclass of **`UICollectionViewCell`** then name it “MemoryCell” and click Next. As before, make sure you use Happy Days with the yellow folder icon for your group, then click Create.

That creates the class, so now it’s just a matter of connecting the cell prototype to the class in IB. To do that, select “Memory” in the document outline, then use the identity inspector to change the Class value to “MemoryCell”.

And now the final task, number 7: set up outlets and actions. We’ll do the first run screen first, so select the first run controller in the document outline, then change to the assistant editor so we can see its code while working in IB. This works sometimes and other times either selects the wrong thing or selects nothing at all - in theory you should have IB in the top pane, and ViewController.swift in the bottom pane. If you have something else (or nothing!) in the bottom pane, click where it says Automatic and choose Manual > Happy Days > Happy Days > ViewController.swift.

I’d like you to create an outlet for the label called **`helpLabel`**, and an action for the button called **`requestPermissions`**.



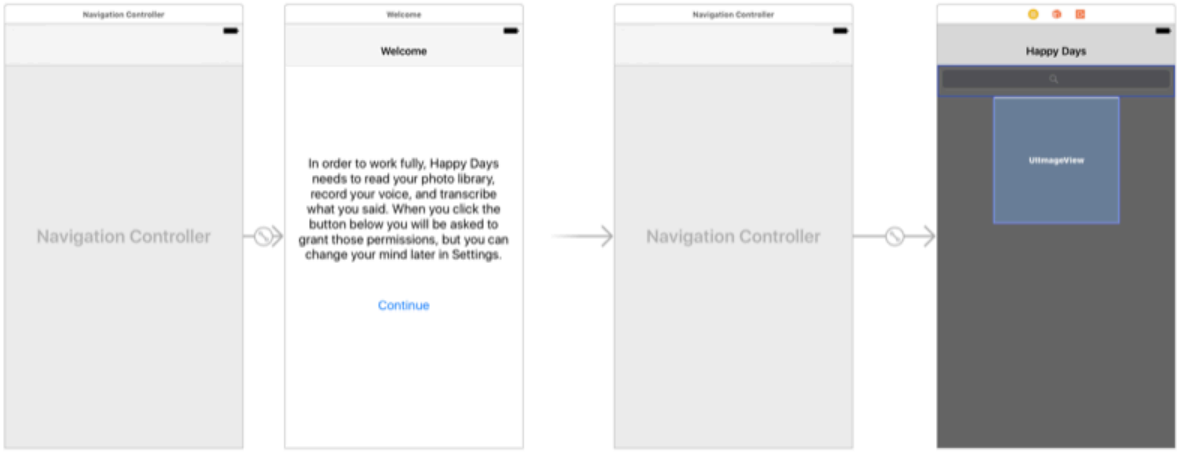
While running through this project a couple of times for testing, Xcode would occasionally refuse to create an outlet because it somehow couldn't find **`ViewController`** – if this happens to you, select your project icon in the project navigator, then choose HappyDays under the Targets list. Select the Build Phases tab, then open Compile Sources – you should see ViewController.swift in there. Select it, then click - to stop it from being built, then click + and re-add it. For whatever reason, that was the only thing I found that would clear the glitch, but hopefully it doesn't happen to you!

Anyway, back to Main.storyboard: you should have an outlet for the label, and an action for the button like this:

```
@IBAction func requestPermissions(_ sender: AnyObject) {
}
```

The other outlet we're going to create is for the image view inside **`MemoryCell`**. Find the "Memory" collection view cell prototype in your IB document outline, then use manual selection in the bottom pane of the assistant editor to open MemoryCell.swift. Now Ctrl-drag from the image view to MemoryCell.swift to create a property called **`imageView`**.

Finally, we're done with Interface Builder, so switch to back to the standard editor view and open ViewController.swift for editing.



Permissions! Permissions everywhere!

We need three different kinds of permissions to make this app work. I'm going to code the app so that it requires all three to work, but you're welcome to make it more fine-grained if you want. If any of the permissions fail, I'm going to rewrite the contents of **`helpLabel`** with a meaningful message so that the user can go to Settings and grant the permissions by hand if they change their mind.

The permissions system in iOS is all asynchronous, which means that you make a request then receive a callback when the user has made a choice. These callbacks can happen on any thread, so you should always push work to the main thread before taking any action – particularly if you're manipulating the user interface. We're going to use these callbacks to create a smooth chain of permission requests: rather than bombard the user with three requests at once, we're going to request one, wait for a callback, request another, wait for a callback, then request the last one.

Each of these permission requests are done on a different framework: AVFoundation handles the microphone, Photos handles accessing user photos, and Speech handles transcription. So, please add these three imports near the top of ViewController.swift:

```
import AVFoundation
import Photos
import Speech
```

With that done, we can start the authorization process – Photos authorization is as good a place as any, so we'll start there.

To get access to their user's photo library, we need to call **`requestAuthorization()`** on the **`PHPhotoLibrary`** class. You need to give it a closure that will be called when the user has either granted or denied permission to access the photo library, and you'll be given a parameter to work with that represents the user's permission status. If that's set to **`.authorized`** it means we're good to continue, otherwise we'll show a message in the help label and stop.

Add this method to ViewController.swift:

```
func requestPhotosPermissions() {
    PHPhotoLibrary.requestAuthorization { [unowned self]
authStatus in
        DispatchQueue.main.async {
            if authStatus == .authorized {
                self.requestRecordPermissions()
            } else {
                self.helpLabel.text = "Photos permission was
declined; please enable it in settings then tap Continue
again."
            }
        }
    }
}
```

You'll get an error because we haven't written **`self.requestRecordPermissions()`** just yet - we'll do that in just a moment.

First, though: notice that I'm pushing all the work to the main thread using **`DispatchQueue.main.async`**. This might be your first exposure to Grand Central Dispatch in Swift 3, but I hope you can appreciate how marvelously simple it is compared to GCD in Swift 2.2!

Next: the **`requestRecordPermissions()`** method. This is almost identical to requesting access to the photo library, except now we call **`requestRecordPermission()`** on the shared **`AVAudioSession`** instance. This will again call a closure, but the parameter is simpler this time: we'll get a boolean that's true if we've been given access to the microphone.

As before, we immediately push our work to the main thread to avoid problems, then either call another method to continue the permissions requests, or set the help label to something meaningful. Here's the code:

```
func requestRecordPermissions() {
    AVAudioSession.sharedInstance().requestRecordPermission()
{ [unowned self] allowed in
```

```
            DispatchQueue.main.async {
                if allowed {
                    self.requestTranscribePermissions()
                } else {
                    self.helpLabel.text = "Recording permission was
declined; please enable it in settings then tap Continue
again."
                }
            }
        }
    }
```

The last permissions method we need to write is **`requestTranscribePermissions()`**, which is almost identical to requesting permissions to Photos – the only difference is that you call the method on the new **`SFSpeechRecognizer`** class rather than **`PHPhotoLibrary`**. Here's the code:

```
func requestTranscribePermissions() {
    SFSpeechRecognizer.requestAuthorization { [unowned self]
authStatus in
        DispatchQueue.main.async {
            if authStatus == .authorized {
                self.authorizationComplete()
            } else {
                self.helpLabel.text = "Transcription permission was
declined; please enable it in settings then tap Continue
again."
            }
        }
    }
}
```

That's the last of the permissions work. We need to make two further additions, though. First, add this method to handle dismissing the first run controller when permissions are fully

granted:

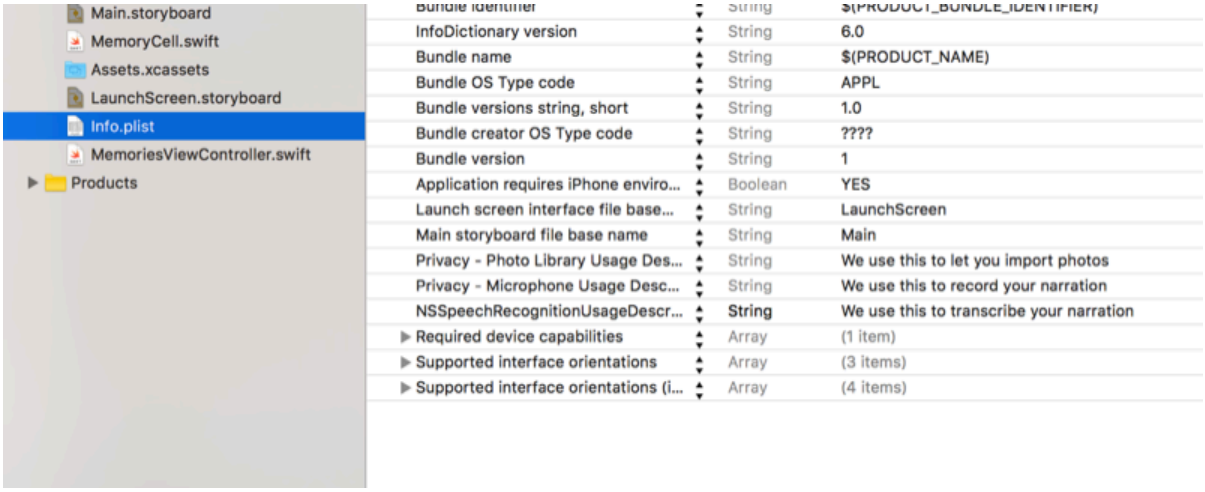
```
func authorizationComplete() {
    dismiss(animated: true)
}
```

Second, we need to kick off the whole permissions flow by adding a call to **`requestPhotosPermissions()`** inside the **`requestPermissions()`** method we created earlier, like this:

```
@IBAction func requestPermissions(_ sender: AnyObject) {
    requestPhotosPermissions()
}
```

That's all the code for permissions: we request photo library access, then request microphone access, then request transcription access. However, even though the code is complete and correct, it won't work quite yet because we need to tell the user *why* we're requesting the permissions.

This is done in the Info.plist file: we need to add three keys there to describe why we're asking for each of the permissions. Please add the keys **`NSPhotoLibraryUsageDescription`**, **`NSMicrophoneUsageDescription`**, and **`NSSpeechRecognitionUsageDescription`**. Give each of them some meaningful text - something like "We use this to let you import photos", "We use this to record your narration", and "We use this to transcribe your narration" respectively.



We're done with ViewController.swift – it's sole purpose is to guide the user through all the permissions requests, so all that's left to do is call it at the right time. We know when that "right time" is inside the **`viewDidAppear()`** method of **`MemoriesViewController`**: we can check what permissions we have, and show **`ViewController`** if any are lacking.

Open MemoriesViewController.swift, and add these three just above the existing **`import UIKit`** line:

```
import AVFoundation
import Photos
import Speech
```

I'm going to encapsulate the permissions check in its own method, **`checkPermissions()`**, because it's possible you'll want to check this in other places later on. All it's going to do is go through **`PHPhotoLibrary`**, **`AVAudioSession`**, and **`SFSpeechRecognizer`** to make sure each of them have permissions granted, then combine them all in to a single boolean, **`authorized`**, which can be checked. If that's false, it means we're missing at least one permission, so we'll show the "FirstRun" view controller, which is the navigation controller that contains the permissions view.

Here's the **`checkPermissions()`** method – please put this inside MemoriesViewController.swift:

```
func checkPermissions() {
    // check status for all three permissions
```

```
    let photosAuthorized = PHPhotoLibrary.authorizationStatus()
== .authorized
    let recordingAuthorized =
AVAudioSession.sharedInstance().recordPermission() == .granted
    let transcibeAuthorized =
SFSpeechRecognizer.authorizationStatus() == .authorized

    // make a single boolean out of all three
    let authorized = photosAuthorized && recordingAuthorized &&
transcibeAuthorized

    // if we're missing one, show the first run screen
    if authorized == false {
        if let vc =
storyboard?.instantiateViewController(withIdentifier:
"FirstRun") {
            navigationController?.present(vc, animated: true)
        }
    }
}
```

With that method written, all we need to do now is call it inside **`viewDidAppear()`**. Add this now:

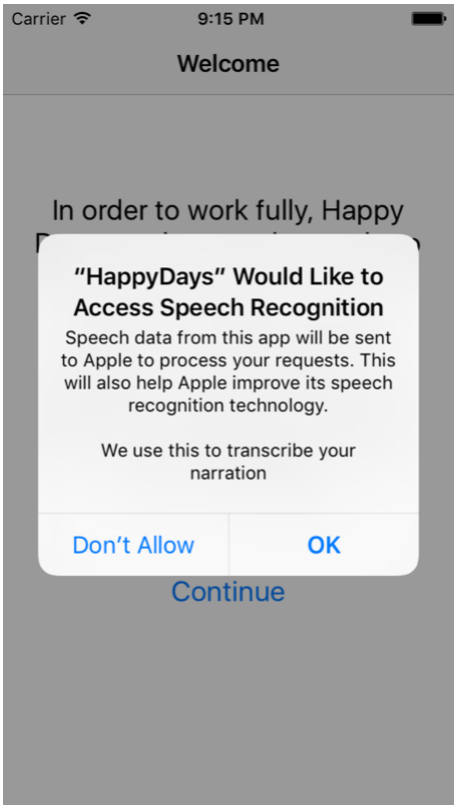
```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    checkPermissions()
}
```

Done!

At this point we finally have some code that can be run, so press Cmd+R to build and launch in the simulator. All being well you should see the first run screen appear and ask for various

permissions – you may not see the microphone permission appear in the simulator, but it will appear on real devices.

When the final permission is granted, the welcome screen will disappear and you'll see the empty collection view controller. Let's tackle that now…



Importing into the collection view

The first big task we're going to tackle is to fill our collection view with pictures the user has selected. The final version of the app will have pictures, audio, and text for each memory, so we need some sort of way to tie this together.

We *could* use a custom class for this task: by making it conform to `NSCoding` we could read and write whole arrays of data using something like `NSKeyedArchiver`. However, I think that would prove inefficient: picture data can easily be 10MB depending on the device you have, and there's audio data too – we wouldn't want to have to write all that data out when the user changed one small part.

So instead we're going to go for something much simpler: every memory will have a unique name, and we'll just hang different file extensions off that unique name to provide the various components. For example, if a memory had the name abc123, we would have the following files:

- abc123.jpg: the full-size picture the user imported.
- abc123.thumb: the thumbnail-sized picture
- abc123.m4a: the audio narration they recorded
- abc123.txt: the plain text transcription of what they said.

Of those, only the first two can be guaranteed to exist - the latter two won't exist until the user has added narration and we've transcribed it.

Working with files used to be quite easy in Swift because you could read and write files using path strings. However, in the name of safety, Apple has slowly been forcing us to migrate to URLs for files, which in turn means we need to litter optionals and try/catch code throughout our code. It's safer in the long term, but it might frustrate you until you get the hang of it!

Loading memories

Let's start with the easiest part: loading memories into an array. We're going to create a property that contains the full path to the root name of a memory – that's the memory name without the ".jpg" or ".m4a". To keep things uniform, this will be an array of URLs, so please add this property to `MemoriesViewController`:

```
var memories = [URL]()
```

We'll fill that array in a dedicated method, **`loadMemories()`**, so that we can call it later on – for example, when we add a new memory from the photo library.

This method needs to do several things:

1. Remove any existing memories. We'll be calling it multiple times, so we don't want duplicates.
2. Pull out a list of all the files stored in our app's documents directory. This needs to be work with URLs, rather than path strings.
3. Loop over every file that was found, and, if it was a thumbnail, add it to our memories array. We only count thumbnails to avoid duplicating items - we don't want to add the .jpg and .thumb for each memory.

That last point is actually quite complicated thanks to the rather graceless way Swift handles URLs, but first the easy bit – finding our app's documents directory can be done using this helper function:

```
func getDocumentsDirectory() -> URL {
    let paths =
FileManager.default.urls(for: .documentDirectory,
in: .userDomainMask)
    let documentsDirectory = paths[0]
    return documentsDirectory
}
```

Now for the **`loadMemories()`** method itself. I've added comments directly inline with the code, because it's harder than you might think. Add this method to **`MemoriesViewController`**:

```
func loadMemories() {
    memories.removeAll()
```