HACKING WITH SWIFT

PROJECTS 1-39

Learn to make iOS ap
with real projects

FREE SAMPLE

Paul Hudson

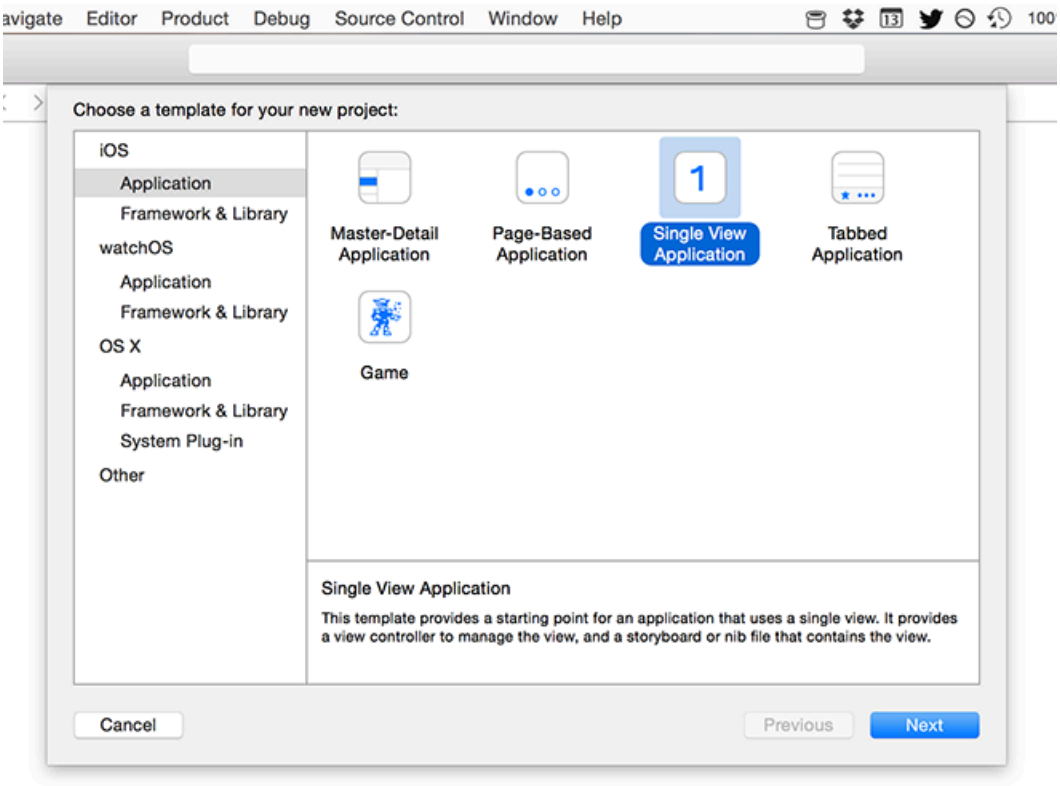
Project 1
Storm Viewer

Get started coding in Swift by making an image viewer app and learning key concepts.

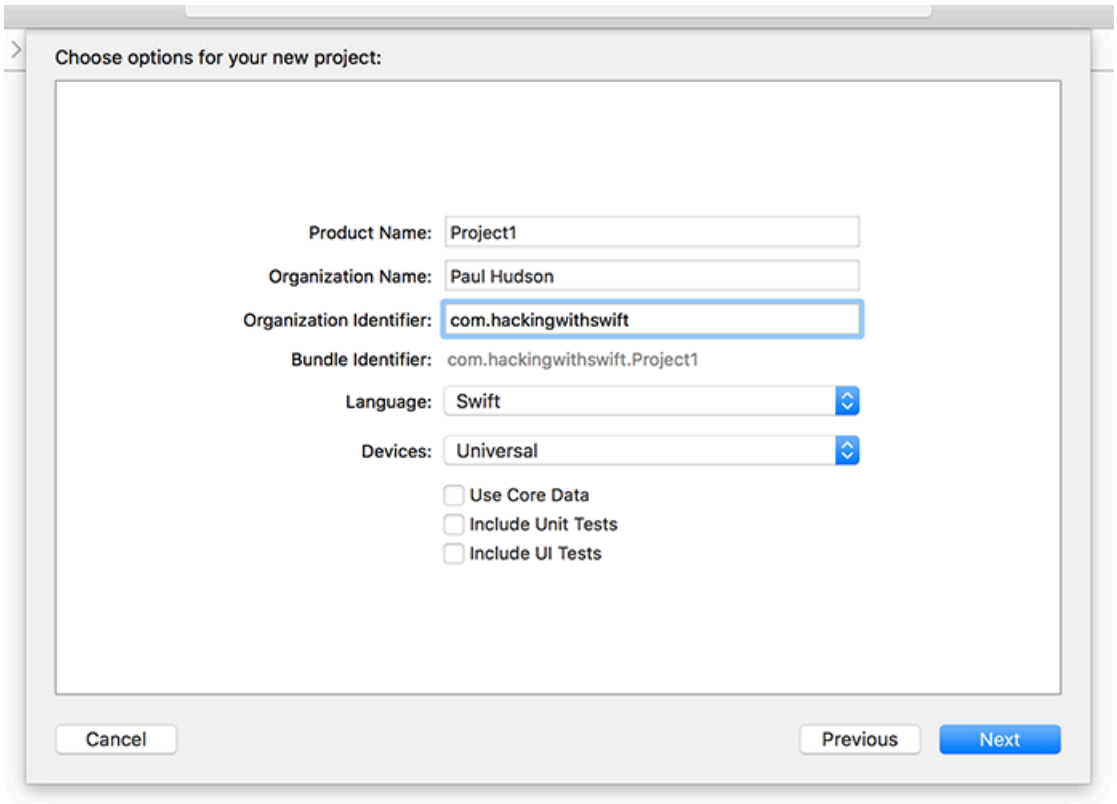
Setting up

In this project you'll produce an application that lets users scroll through a list of images, then select one to view. It's deliberately simple, because there are many other things you'll need to learn along the way, so strap yourself in – this is going to be long!

Launch Xcode, and choose "Create a new project" from the welcome screen. Choose Single View Application from the list and click Next. For Product Name enter Project1, then make sure you have Swift selected for language and Universal for devices.



One of the fields you'll be asked for is "Organization Identifier", which is a unique identifier usually made up of your personal web site domain name in reverse. For example, I would use **com.hackingwithswift** if I were making an app. You'll need to put something valid in there if you're deploying to devices, but otherwise you can just use **com.example**.



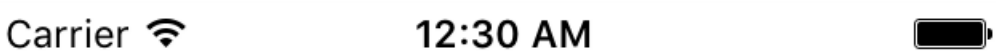
Important note: some of Xcode's project templates have checkboxes saying "Use Core Data", "Include Unit Tests" and "Include UI Tests". Please ensure these boxes are unchecked for this project and indeed all projects in this series.

Now click Next again and you'll be asked where you want to save the project – your desktop is fine. Once that's done, you'll be presented with the example project that Xcode made for you. The first thing we need to do is make sure you have everything set up correctly, and that means running the project as-is.

When you run a project, you get to choose what kind of device the iOS Simulator should pretend to be, or you can also select a physical device if you have one plugged in. These options are listed under the Product > Destination menu, and you should see iPad Air, iPhone 7, and so on.

There's also a shortcut for this menu: at the top-left of Xcode's window is the play and stop button, but to the right of that it should say Project1 then a device name. You can click on that device name to select a different device.

For now, please choose iPhone 6, and click the Play triangle button in the top-left corner. This will compile your code, which is the process of converting it to instructions that iPhones can understand, then launch the simulator and run the app. As you'll see when you interact with the app, our “app” just shows a large white screen – it does nothing at all, at least not yet.



You'll be starting and stopping projects a lot as you learn, so there are three basic tips you need to know:

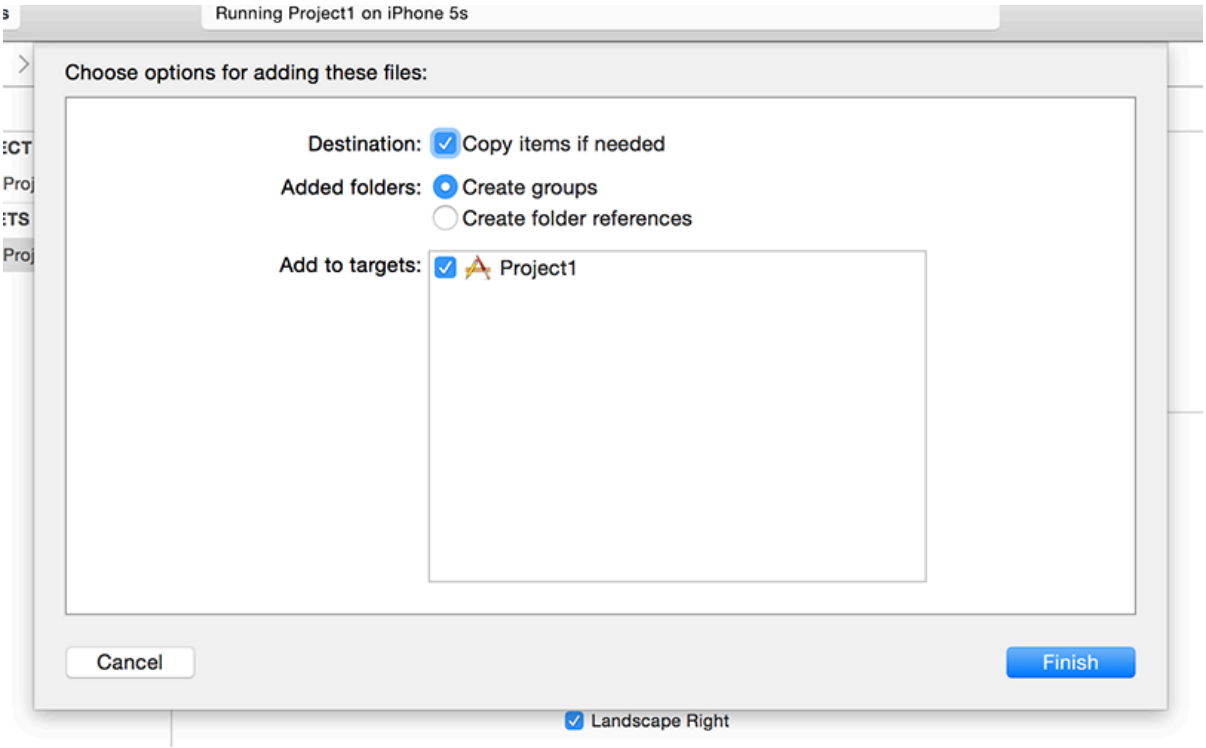
- You can run your project by pressing Cmd+R. This is equivalent to clicking the play button.
- You can stop a running project by pressing Cmd+. when Xcode is selected.
- If you have made changes to a running project, just press Cmd+R again. Xcode will prompt you to stop the current run before starting another. Make sure you check the "Do not show this message again" box to avoid being bothered in the future.

This project is all about letting users select images to view, so you're going to need to import some pictures. Download the files for this project from GitHub, and look in the Project1 folder.

You'll see another folder in there called Project1, and inside that a folder called Content. I want you to drag that Content folder straight into your Xcode project, just under where it says "Info.plist".

Warning: some very confused people have ignored the word “download” above and tried to drag files straight from GitHub. *That will not work*. You need to download the files as a zip file, extract them, then drag them from Finder into Xcode.

A window will appear asking how you want to add the files: make sure "Copy items if needed" is checked, and "Create groups" is selected. **Important: do not choose "Create folder references" otherwise your project will not work.**



Click Finish and you'll see a yellow Content folder appear in Xcode. If you see a blue one, you didn't select "Create groups", and you'll have problems following this tutorial!

Listing images with FileManager

The images I've provided you with come from the National Oceanic and Atmospheric Administration (NOAA), which is a US government agency and thus produces public domain content that we can freely reuse. Once they are copied into your project, Xcode will automatically build them into your finished app so that you can access them.

Behind the scenes, an iOS (and macOS) app is actually a directory containing lots of files: the binary itself (that's the compiled version of your code, ready to run), all the media assets your app uses, any visual layout files you have, plus a variety of other things such as metadata and security entitlements.

These app directories are called bundles, and they have the file extension .app. Because our media files are loose inside the folder, we can ask the system to tell us all the files that are in there then pull out the ones we want. You may have noticed that all the images start with the name "nssl" (short for National Severe Storms Laboratory), so our task is simple: list all the files in our app's directory, and pull out the ones that start with "nssl".

For now, we'll load that list and just print it to Xcode's built in log viewer, but soon we'll make them appear in our app.

So, step 1: open ViewController.swift. A view controller is best thought of as being one screen of information, and for us that's just one big blank screen. ViewController.swift is responsible for showing that blank screen, and right now it won't contain much code. You should see something like this:

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view,
typically from a nib.
    }

    override func didReceiveMemoryWarning() {
```

```
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

That contains six interesting things I want to discuss before moving on.

1. The file starts with **`import UIKit`**, which means "this file will reference the iOS user interface toolkit."
2. The **`class ViewController: UIViewController`** line means "I want to create a new screen of data called ViewController, based on UIViewController." When you see a data type that starts with "UI", it means it comes from UIKit. **`UIViewController`** is Apple's default screen type, which is empty and white until we change it.
3. The line **`override func viewDidLoad()`** starts a method (a block of code), which is a piece of code inside our **`ViewController`** screen. The **`override`** keyword is needed because it means "we want to change Apple's default behavior from **`UIViewController`**." **`viewDidLoad()`** is called when the screen has loaded, and is ready for you to customize.
4. The line **`override func didReceiveMemoryWarning()`** starts another method, and again overrides Apple's default behavior from **`UIViewController`**. This method is called when the system is running low on resources, and you're expected to release any RAM you don't need any more.
5. There are lots of **`{`** and **`}`** characters. These symbols, known as *braces* (or sometimes *curly brackets*) are used to mark chunks of code, and it's convention to indent lines inside braces so that it's easy to identify where code blocks start and end. The outermost braces contain the entire **`ViewController`** data type, and the two sets of inner braces mark the start and end of the **`viewDidLoad()`** and **`didReceiveMemoryWarning()`** methods.
6. The **`viewDidLoad()`** method contains one line of code saying **`super.viewDidLoad()`** and one line of comment (that's the line starting with **`//`**); **`didReceiveMemoryWarning()`** contains a call to **`super.didReceiveMemoryWarning()`** and another comment line. These **`super`** calls mean "tell Apple's **`UIViewController`** to run its own code before I

run mine," and you'll see this used a lot.

We'll come back to this code a *lot* in future projects; don't worry if it's all a bit hazy right now.

No line numbers? While you're reading code, it's frequently helpful to have line numbers enabled so you can refer to specific code more easily. If your Xcode isn't showing line numbers by default, I suggest you turn them on now: go to the Xcode menu and choose Preferences, then choose the Text Editing tab and make sure "Line numbers" is checked.

As I said before, the **`viewDidLoad()`** method is called when the screen has loaded and is ready for you to customize. Everything between **`func viewDidLoad() {`** and the **`}`** that follows a few lines later is part of that method, and will get called when you can start customizing the screen.

We're going to put some more code into that method to load the NSSL images. Add this beneath the line that says **`super.viewDidLoad()`**:

```
let fm = FileManager.default
let path = Bundle.main.resourcePath!
let items = try! fm.contentsOfDirectory(atPath: path)

for item in items {
    if item.hasPrefix("nssl") {
        // this is a picture to load!
    }
}
```

That's a big chunk of code, all of which is new. Let's walk through what it does line by line:

- The line **`let fm = FileManager.default`** declares a constant called **`fm`** and assigns it the value returned by **`FileManager.default`**. This is a data type that lets us work with the filesystem, and in our case we'll be using it to look for files.
- The line **`let path = Bundle.main.resourcePath!`** declares a constant called **`path`** that is set to the resource path of our app's bundle. Remember, a bundle is

a directory containing our compiled program and all our assets. So, this line says, "tell me where I can find all those images I added to my app."

- The line `let items = try! fm.contentsOfDirectory(atPath: path)` declares a third constant called `items` that is set to the contents of the directory at a path. Which path? Well, the one that was returned by the line before. As you can see, Apple's long method names really does make their code quite self-descriptive! The `items` constant is an array – a collection – of the names of all the files that were found in the resource directory for our app.
- The line `for item in items {` starts a *loop*. Loops are a block of code that execute multiple times. In this case, the loop executes once for every item we found in the app bundle. Note that the line has an opening brace at the end, signaling the start of a new block of code, and there's a matching closing brace four lines beneath. Everything inside those braces will be executed each time the loop goes around. We could translate this line as "treat items as a series of text strings, then pull out each one of those text strings, give it the name `item`, then run the following code…"
- The line `if item.hasPrefix("nssl") {` is the first line inside our loop. By this point, we'll have the first filename ready to work with, and it'll be called `item`. To decide whether it's one we care about or not, we use the `hasPrefix()` method: it takes one parameter (the prefix to search for) and returns either true or false. That "if" at the start means this line is a conditional statement: if the item has the prefix "nssl", then… that's right, another opening brace to mark another new code block. This time, the code will be executed only if `hasPrefix()` returned true.
- Finally, the line `// this is a picture to load!` is a comment – if we reach here, `item` contains the name of a picture to load from our bundle, so we need to store it somewhere.

In just those few lines of code, there's quite a lot to take in, so before continuing let's recap:

- We use `let` to declare constants. Constants are pieces of data that we want to reference, but that we know won't have a changing value. For example, your birthday is a constant, but your age is not – your age is a variable, because it varies.
- Swift coders really like to use constants in places most other developers use variables. This is because when you're actually coding you start to realize that most of the data you store doesn't actually change very much, so you might as well make it constant.

Doing so allows the system to make your code run faster, and also adds some extra safety because if you try to change a constant Xcode will refuse to build your app.

- Text in Swift is represented using the `String` data type. Swift strings are extremely powerful and guaranteed to work with any language you can think of – English, Chinese, Klingon and more.
- Collections of values are called arrays, and are usually restricted to holding one data type at a time. An array of strings is written as `[String]` and can hold only strings. If you try to put numbers in there, Xcode won't build your app.
- The `try!` keyword means "the following code has the potential to go wrong, but I'm absolutely certain it won't." If the code *does* fail – for example if the directory we asked for doesn't exist – our app will crash. At the same time, if this code fails it means our app can't read its own data, so something must be seriously wrong!
- You can use `for someItem in someArray` to loop through every item in an array. Swift pulls out each item and runs the code inside your loop once for each item.

If you're extremely observant you might have noticed one tiny, tiny little thing that is also one of the most complicated parts of Swift, so I'm going to keep it as simple as possible for now, then expand more over time: it's the exclamation mark at the end of `Bundle.main.resourcePath!` No, that wasn't a typo from me. If you take away the exclamation mark the code will no longer work, so clearly Xcode thinks it's important – and indeed it is. Swift has three ways of working with data:

1. A variable or constant that holds the data. For example, `foo: String` is a string of letters called `foo`.
2. A variable or constant that might hold the data, but we're not sure. This is called an optional type, and looks like this: `foo: String?` You can't use these directly, instead you need to ask Swift to check they have a value first.
3. A variable or constant that might hold the data or might not, but we're 100% certain it does – at least once it has first been set. This is called an implicitly unwrapped optional, and looks like this: `foo: String!` You *can* use these directly.

When I explain this to people, they nearly always get confused, so please don't worry if the above made no sense to you – we'll be going over optionals again and again in coming

projects, so just give yourself time.

We'll look at optionals in more depth later, but for now what matters is that `Bundle.main.resourcePath` may or may not return a string, so what it returns is a `String?` – that is, an optional string. By adding the exclamation mark to the end we are force unwrapping the optional string, which means we're saying, "I'm sure this will return a real string, it will never be `nil`, so please just give it to me as a regular string."

Important warning: if you ever try to use a constant or variable that has a `nil` value, your app will crash. As a result, some people have named `!` the "crash" operator because it's easy to get wrong. The same is true of `try!`, which is also easy to get wrong. Don't worry if this all sounds hard for now – you'll be using it more later, and it will make more sense over time.

Right now our code loads the list of files that are inside our app bundle, then loops over them all to find the ones with a name that begins with “nssl”. However, it doesn’t actually do anything with those files, so our next step is to create an array of all the “nssl” pictures so we can refer to them later rather than having to re-read the resources directory again and again.

The three constants we already created – `fm`, `path`, and `items` – live inside the `viewDidLoad()` method, and will be destroyed as soon as that method finishes. What we want is a way to attach data to the whole `ViewController` type so that it will exist for as long as our screen exists. In Swift this is done using a “property”: we can give `ViewController` as many of these properties as we want, then read and write them as often as needed while the screen exists.

To create a property, you need to declare it *outside* of methods. We’ve been creating constants using `let` so far, but this array is going to be changed inside our loop so we need to make it variable. We also need to tell Swift exactly what kind of data it will hold – in our case that’s an array of strings, where each item will be the name of an “nssl” picture.

Add this line of code *before* `viewDidLoad()`:

```
var pictures = [String]()
```

If you’ve placed it correctly, your code should look like this:

```
class ViewController: UIViewController {
    var pictures = [String]()

    override func viewDidLoad() {
        super.viewDidLoad()

        let fm = FileManager.default
```

The **var** keyword is used to create variables, in the same way that **let** is used to create constants. Where things get a bit crazy is in the second half of the line: **[String]()**. That's really two things in one: **[String]** means "an array of strings", and **()** means "create one now." The parentheses here are just like those in the **viewDidLoad()** method – it signals the name of some other code that should be run, in this case the code to create a new array of strings.

That **pictures** array will be created when the **ViewController** screen is created, and exist for as long as the screen exists. It will be empty, because we haven't actually filled it with anything, but at least it's there ready for us to fill.

What we *really* want is to add to the **pictures** array all the files we match inside our loop. To do that, we need to replace the existing **// this is a picture to load!** comment with code to add each picture to the **pictures** array.

Helpfully, Swift's arrays have a built-in method called **append** that we can use to add any items we want. So, replace the **// this is a picture to load!** comment with this:

```
pictures.append(item)
```

That's it! Annoyingly, after all that work our app won't appear to do anything when you press play – you'll see the same white screen as before. Did it work, or did things just silently fail?

To find out, add this line of code at the end of **viewDidLoad()**, just before the closing brace:

```
print(pictures)
```

That tells Swift to print the contains of `pictures` to the Xcode debug console. When you run the program now, you should see this text appear at the bottom of your Xcode window: “["nssl0033.jpg", "nssl0034.jpg", "nssl0041.jpg", "nssl0042.jpg", "nssl0043.jpg", "nssl0045.jpg", "nssl0046.jpg", "nssl0049.jpg", "nssl0051.jpg", "nssl0091.jpg”]”

Note: iOS likes to print lots of uninteresting debug messages in the Xcode debug console. Don’t fret if you see lots of other text in there that you don’t recognize – just scroll around until you see the text above, and if you see that then you’re good to go.

Designing our interface

Our app loads all the storm images correctly, but it doesn't do anything interesting with them – printing to the Xcode console is helpful for debugging, but I can promise you it doesn't make for a best-selling app!

To fix this, our next goal is to create a user interface that lists the images so users can select one. UIKit – the iOS user interface framework – has a lot of built-in user interface tools that we can draw on to build powerful apps that look and work the way users expect.

For this app, our main user interface component is called **`UITableViewController`**. It's based on **`UIViewController`** – Apple's most basic type of screen – but adds the ability to show rows of data that can be scrolled and selected. You can see **`UITableViewController`** in the Settings app, in Mail, in Notes, in Health, and many more – it's powerful, flexible, and extremely fast, so it's no surprise it gets used in so many apps.

Our existing **`ViewController`** screen is based on **`UIViewController`**, but what we want is to have it based on **`UITableViewController`** instead. This doesn't take much to do, but you're going to meet a new part of Xcode called Interface Builder.

We'll get on to Interface Builder in a moment. First, though, we need to make a tiny change in ViewController.swift. Find this line:

```
class ViewController: UIViewController {
```

That's the line that says "create a new screen called **`ViewController`** and have it build on Apple's own **`UIViewController`** screen." I want you to change it to this:

```
class ViewController: UITableViewController {
```

It's only a small difference, but it's an important one: it means **`ViewController`** now inherits its functionality from **`UITableViewController`** instead of **`UIViewController`**, which gives us a huge amount of functionality for free as you'll see in a moment.

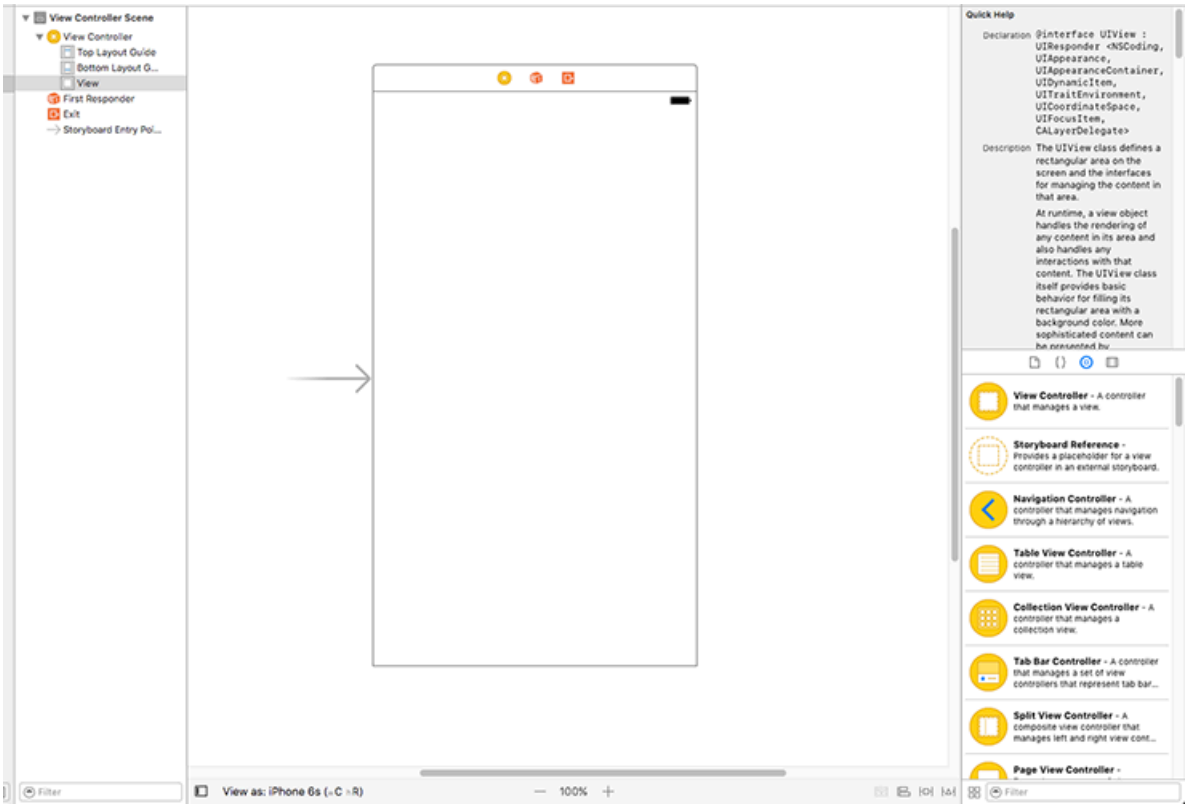
Behind the scenes, **`UITableViewController`** still builds on top of

`UIViewController` – this is called a "class hierarchy", and is a common way to build up functionality quickly.

We've changed the code for **`ViewController`** so that it builds on **`UITableViewController`**, but we also need to change the user interface to match. User interfaces can be written entirely in code if you want – and many developers do just that – but more commonly they are created using a graphical editor called Interface Builder. We need to tell Interface Builder (usually just called "IB") that **`ViewController`** is a table view controller, so that it matches the change we just made in our code.

Up to this point we've been working entirely in the file ViewController.swift, but now I'd like you to use the project navigator (the pane on the left) to select the file Main.storyboard. Storyboards contain the user interface for your app, and let you visualize some or all of it on a single screen.

When you select Main.storyboard, you'll switch to the Interface Builder visual editor, and you should see something like the picture below:



That big white space is what produces the big white space when the app runs. If you drop new components into that space, they would be visible when the app runs. However, we don't want to do that – in fact, we don't want that big white space at all, so we're going to delete it.

The best way to view, select, edit, and delete items in Interface Builder is to use the document outline, but there's a good chance it will be hidden for you so the first thing to do is show it. Go to the Editor menu and choose Show Document Outline – it's probably the third option from the top. If you see Hide Document Outline instead, it means the document outline is already visible.

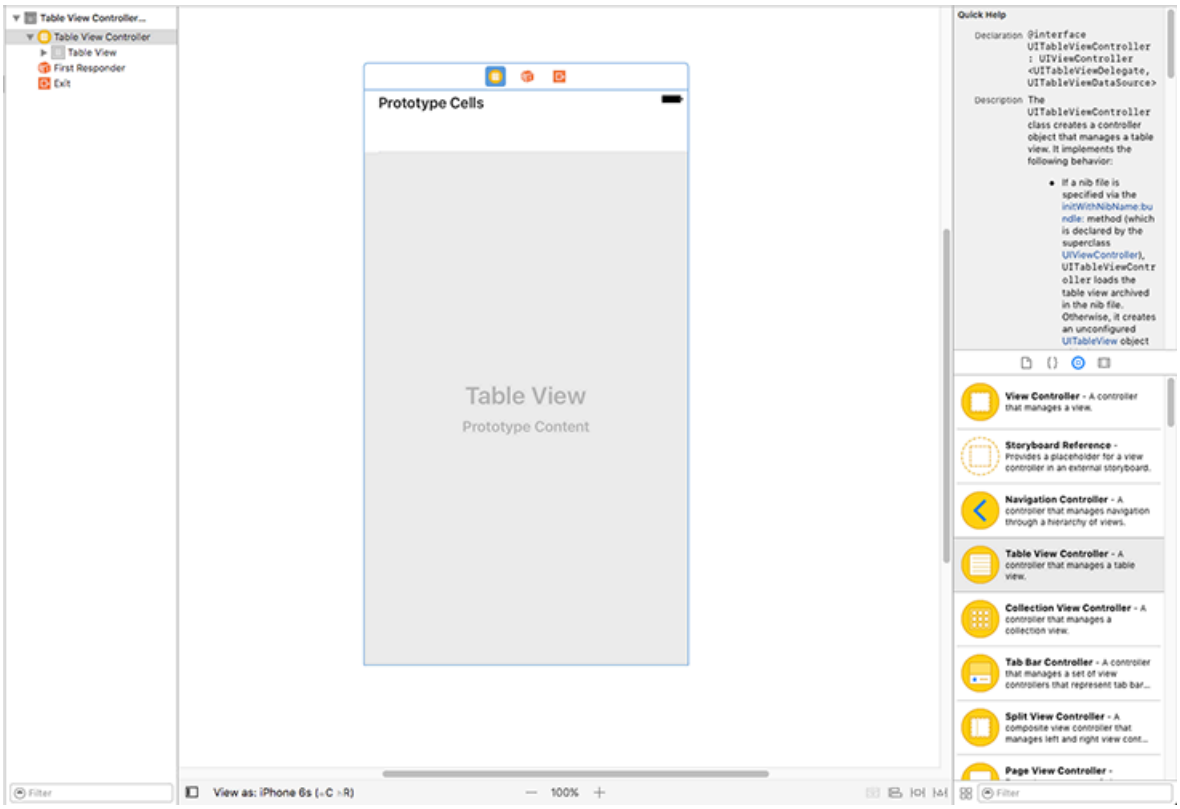
The document outline shows you all the components in all the screens in your storyboard. You should see "View Controller Scene" already in there, so please select it, then press Backspace on your keyboard to remove it.

Instead of a boring old **`UIViewController`**, we want a fancy new **`UITableViewController`** to match the change we made in our code. To create one, press Ctrl+Alt+Cmd+3 to show the object library. Alternatively, if you dislike keyboard shortcuts you can go to the View menu and choose Utilities > Show Object Library instead.

The object library sits in the bottom-right corner of the Xcode window, and contains a selection of graphical components that you can drag out and re-arrange to your heart's content. It contains quite a lot of components, so you might find it useful to enter a few letters into the "Filter" box to slim down the selection.

Right now, the component we want is called Table View Controller. If you type "table" into the Filter box you'll see Table View Controller, Table View, and Table View Cell. They are all different things, so please make sure you choose the Table View Controller – it has a yellow background in its icon.

Click on the Table View Controller component, then drag it out into the large open space that exists where the previous view controller was. When you let go to drop the table view controller onto the storyboard canvas, it will transform into a screen that looks like the below:



Finishing touches for the user interface

Before we're done here, we need to make a few small changes.

First, we need to tell Xcode that this storyboard table view controller is the same one we have in code inside ViewController.swift. To do that, press Alt+Cmd+3 to activate the identity inspector (or go to View > Utilities > Show Identity Inspector), then look at the very top for a box named "Class". It will have "UITableViewController" written in there in light gray text, but if you click the arrow on its right side you should see a dropdown menu that contains "ViewController" – please select that now.

Second, we need to tell Xcode that this new table view controller is what should be shown when the app first runs. To do that, press Alt+Cmd+4 to activate the attributes inspector (or go to View > Utilities > Show Attributes Inspector), then look for the checkbox named "Is Initial View Controller" and make sure it's checked.

Third, I want you to use the document outline to look inside the new table view controller. Inside you should see it contains a "Table View", which in turn contains "Cell". A table view

cell is responsible for displaying one row of data in a table, and we're going to display one picture name in each cell. Please select "Cell" then, in the attributes inspector, enter the text "Picture" into the text field marked Identifier. While you're there, change the Style option at the top of the attributes inspector – it should be Custom right now, but please change it to Basic.

Finally, we're going to place this whole table view controller inside something else. It's something we don't need to configure or worry about, but it's an extremely common user interface element on iOS and I think you'll recognize it immediately. It's called a navigation controller, and you see it in action in apps like Settings and Mail – it provides the thin gray bar at the top of the screen, and is responsible for that right-to-left sliding animation that happens when you move between screens on iOS.

To place our table view controller into a navigation controller, all you need to do is go to the Editor menu and choose Embed In > Navigation Controller. Interface Builder will move your existing view controller to the right and add a navigation controller around it – you should see a simulated gray bar above your table view now. It will also move the "Is Initial View Controller" property to the navigation controller.

At this point you've done enough to take a look at the results of your work: press Xcode's play button now, or press Cmd+R if you want to feel a bit elite. Once your code runs, you'll now see the plain white box replaced with a large empty table view. If you click and drag your mouse around, you'll see it scrolls and bounces as you would expect, although obviously there's no data in there yet. You should also see a gray navigation bar at the top; that will be important later on.

Showing lots of rows

The next step is to make the table view show some data. Specifically, we want it to show the list of "nssl" pictures, one per row. Apple's **`UITableViewController`** data type provides default behaviors for a lot of things, but by default it says there are zero rows.

Our **`ViewController`** screen builds on **`UITableViewController`** and gets to override the default behavior of Apple's table view to provide customization where needed. You only need to override the bits you want; the default values are all sensible.

To make the table show our rows, we need to override two behaviors: how many rows should be shown, and what each row should contain. This is done by writing two specially named methods, but when you're new to Swift they might look a little strange at first. To make sure everyone can follow along, I'm going to take this slowly – this is the very first project, after all!

Let's start with the method that sets how many rows should appear in the table. Add this code just after the *end* of **`viewDidLoad()`**:

```
override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
    return pictures.count
}
```

Note: that needs to be *after* the *end* of **`viewDidLoad()`**, which means after its closing brace.

That method contains the word "table view" three times, which is deeply confusing at first, so let's break down what it means.

- The **`override`** keyword means the method has been defined already, and we want to override the existing behavior with this new behavior. If you didn't override it, then the previously defined method would execute, and in this instance it would say there are no rows.
- The **`func`** keyword starts a new function or a new method; Swift uses the same keyword for both. Technically speaking a method is a function that appears inside a class, just like our **`ViewController`**, but otherwise there's no difference.
- The method's name comes next: **`tableView`**. That doesn't sound very useful, but the way Apple defines methods is to ensure that the information that gets passed into them – the parameters – are named usefully, and in this case the very first thing that gets passed in is the table view that triggered the code. A table view, as you might have gathered, is the scrolling thing that will contain all our image names, and is a core component in iOS.
- As promised, the next thing to come is **`tableView: UITableView`**, which is the table view that triggered the code. But this contains two pieces of information at once:

`tableView` is the name that we can use to reference the table view inside the method, and `UITableView` is the data type – the bit that describes what it is.

- The most important part of the method comes next: `numberOfRowsInSection section: Int`. This describes what the method actually does. We know it involves a table view because that's the name of the method, but the `numberOfRowsInSection` part is the actual action: this code will be triggered when iOS wants to know how many rows are in the table view. The `section` part is there because table views can be split into sections, like the way the Contacts app separates names by first letter. We only have one section, so we can ignore this number. The `Int` part means "this will be an integer," which means a whole number like 3, 30, or 35678 number."
- Finally, `-> Int` means "this method must return an integer", which ought to be the number of rows to show in the table.

There was one more thing I missed out, and I missed it out for a reason: it's a bit confusing at this point in your Swift career. Did you notice that `_` in there? That's an underscore. It changes the way the method is called. To illustrate this, here's a very simple function:

```
func doStuff(thing: String) {
    // do stuff with "thing"
}
```

It's empty, because its contents don't matter. Instead, let's focus on how it's called. Right now, it's called like this:

```
doStuff(thing: "Hello")
```

You need to write the name of the `thing` parameter when you call the `doStuff()` function. This is a feature of Swift, and helps make your code easier to read. Sometimes, though, it doesn't really make sense to have a name for the first parameter, usually because it's built into the method name.

When that happens, you use the underscore character like this:

```
func doStuff(_ thing: String) {
```

```
    // do stuff with "thing"
}
```

That means “when I call this function I don’t want to write **thing**, but inside the function I want to use **thing** to refer to the value that was passed in.

This is what’s happening with our table view method. The method is called **tableView()** because its first parameter is the table view that you’re working with. It wouldn’t make much sense to write **tableView(tableView: someTableView)**, so using the underscore means you would write **tableView(someTableView)** instead.

I'm not going to pretend it's easy to understand how Swift methods look and work, but the best thing to do is not worry too much if you don't understand right now because after a few hours of coding they will be second nature.

At the very least you do need to know that these methods are referred to using their name (**tableView**) and any named parameters. Parameters without names are just referenced as underscores: **_**. So, to give it its full name, the method you just wrote is referred to as **tableView(_:numberOfRowsInSection:)** – clumsy, I know, which is why most people usually just talk about the important bit, for example, "in the **numberOfRowsInSection** method."

We wrote only one line of code in the method, which was **return pictures.count**. That means “send back the number of pictures in our array,” so we’re asking that there be as many table rows as there are pictures.

Dequeuing cells

That’s the first of two methods we need to write to complete this stage of the app. The second is to specify what each row should look like, and it follows a similar naming convention to the previous method. Add this code now:

```
override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier:
```

```
"Picture", for: indexPath)
    cell.textLabel?.text = pictures[indexPath.row]
    return cell
}
```

Let's break it down into parts again, so you can see exactly how it works.

First, **`override func tableView(_ tableView: UITableView`** is identical to the previous method: the method name is just **`tableView()`**, and it will pass in a table view as its first parameter. The **`_`** means it doesn't need to have a name sent externally, because its the same as the method name.

Second, **`cellForRowAt indexPath: IndexPath`** is the important part of the method name. The method is called **`cellForRowAt`**, and will be called when you need to provide a row. The row to show is specified in the parameter: **`indexPath`**, which is of type **`IndexPath`**. This is a data type that contains both a section number and a row number. We only have one section, so we can ignore that and just use the row number.

Third, **`-> UITableViewCell`** means this method must return a table view cell. If you remember, we created one inside Interface Builder and gave it the identifier "Picture", so we want to use that.

Here's where a little bit of iOS magic comes in: if you look at the Settings app, you'll see it can fit only about 12 rows on the screen at any given time, depending on the size of your phone. To save CPU time and RAM, iOS only creates as many rows as it needs to work. When one rows moves off the top of the screen, iOS will take it away and put it into a reuse queue ready to be recycled into a new row that comes in from the bottom. This means you can scroll through hundreds of rows a second, and iOS can behave lazily and avoid creating any new table view cells – it just recycles the existing ones.

This functionality is baked right into iOS, and it's exactly what our code does on this line:

```
let cell = tableView.dequeueReusableCell(withIdentifier:
"Picture", for: indexPath)
```

That creates a new constant called **cell** by dequeuing a recycled cell from the table. We have to give it the identifier of the cell type we want to recycle, so we enter the same name we gave Interface Builder: "Picture". We also pass along the index path that was requested; this gets used internally by the table view.

That will return to us a table view cell we can work with to display information. You can create your own custom table view cell designs if you want to (more on that much later!), but we're using the built-in Basic style that has a text label. That's where line two comes in: it give the text label of the cell the same text as a picture in our array. Here's the code again:

```
cell.textLabel?.text = pictures[indexPath.row]
```

The **cell** has a property called **textLabel**, but it's optional: there might be a text label, or there might not be – if you had designed your own, for example. Rather than write checks to see if there is a text label or not, Swift lets us use a question mark – **textLabel?** – to mean "do this only if there is an actual text label there, or do nothing otherwise."

We want to set the label text to be the name of the correct picture from our **pictures** array, and that's exactly what the code does. **indexPath.row** will contain the row number we're being asked to load, so we're going to use that to read the corresponding picture from **pictures**, and place it into the cell's text label.

The last line in the method is **return cell**. Remember, this method expects a table view cell to be returned, so we need to send back the one we created – that's what the **return cell** does.

With those two pretty small methods in place, you can run your code again now and see how it looks. All being well you should now see 10 table view cells, each one with a different picture name inside. If you click on one of them it will turn gray, but nothing else will happen. Let's fix that now…