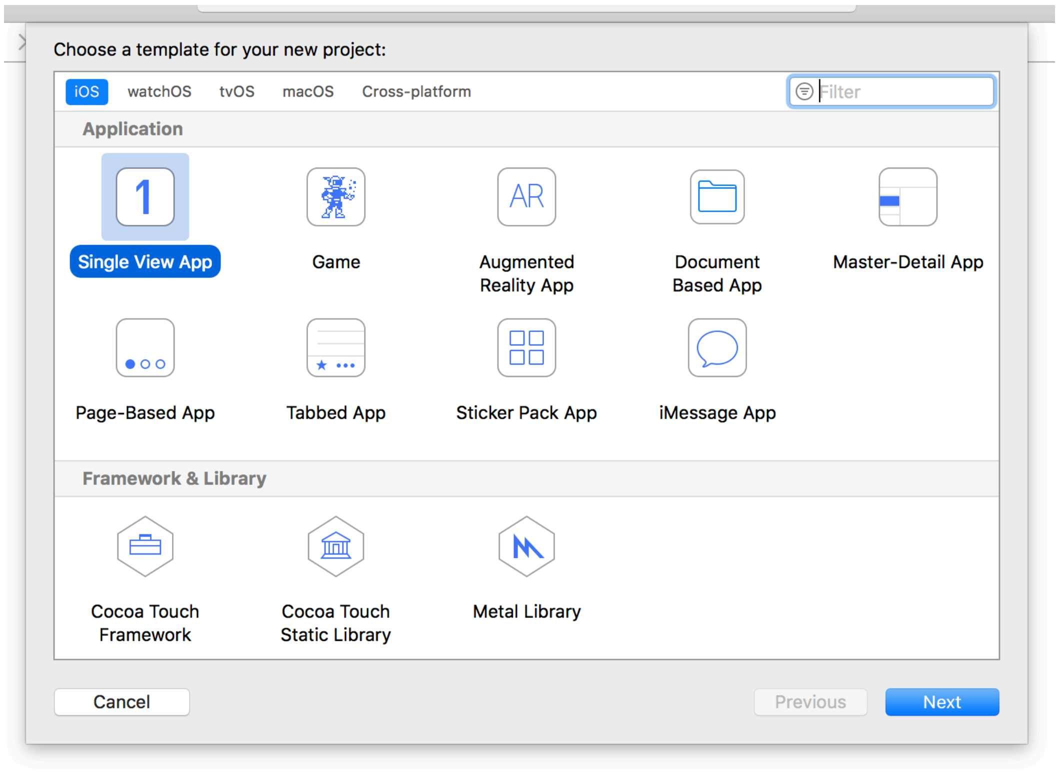
Project 1
Storm Viewer

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

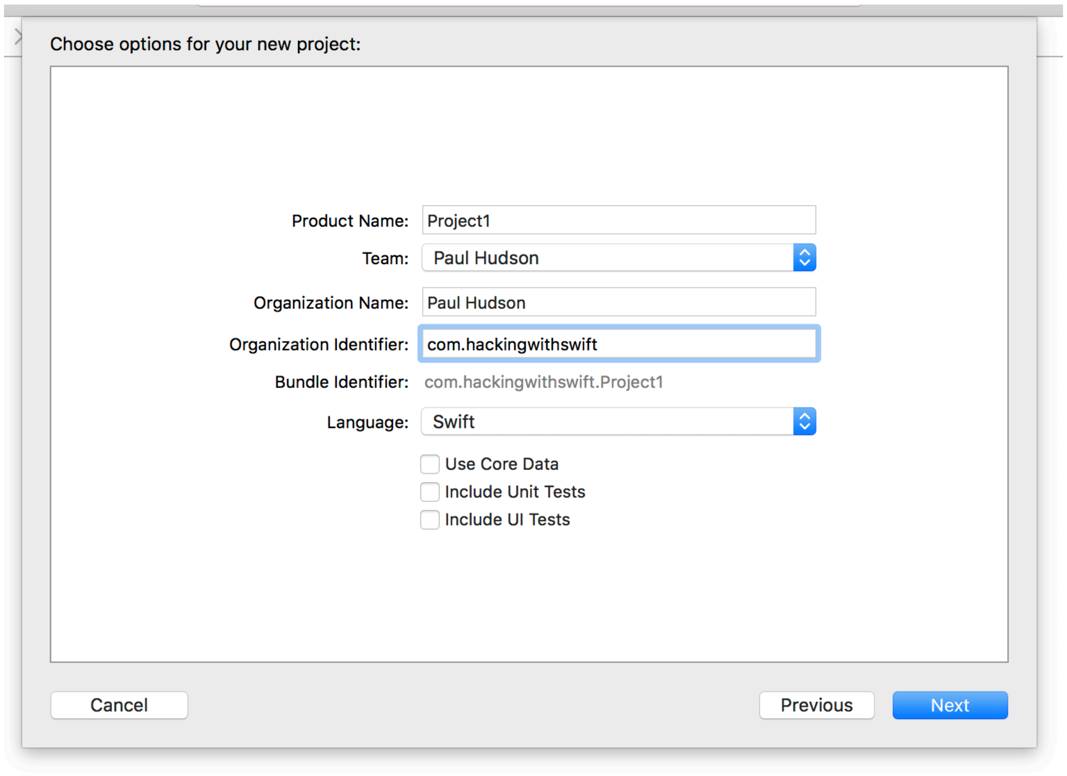
Setting up

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 Xcode project" from the welcome screen. Choose Single View App 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**.


Choose options for your new project:

Product Name: Project1
Team: Paul Hudson
Organization Name: Paul Hudson
Organization Identifier: com.hackingwithswift
Bundle Identifier: com.hackingwithswift.Project1
Language: Swift

[ ] Use Core Data
[ ] Include Unit Tests
[ ] Include UI Tests

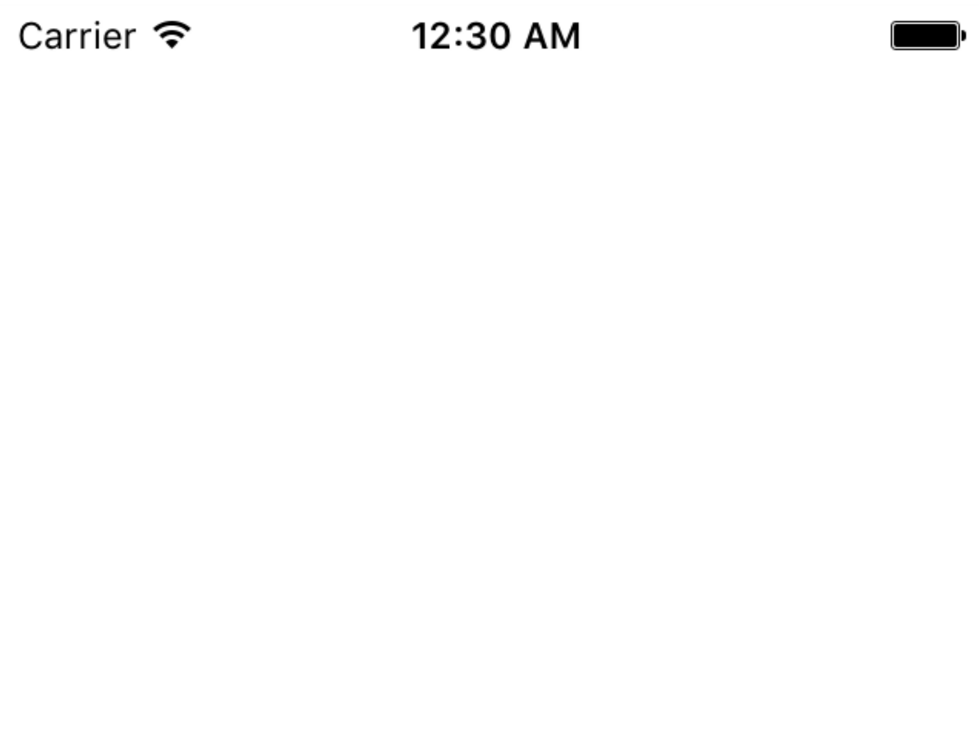
Important note: some of Xcode's project templates have checkboxes saying "Use Core Data", "Include Unit Tests" and "Include UI Tests". Please ensure these boxes are unchecked for this project and indeed almost all projects in this series – there's only one project where this isn't the case, and it's made pretty clear there!

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 8, 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 8, 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.



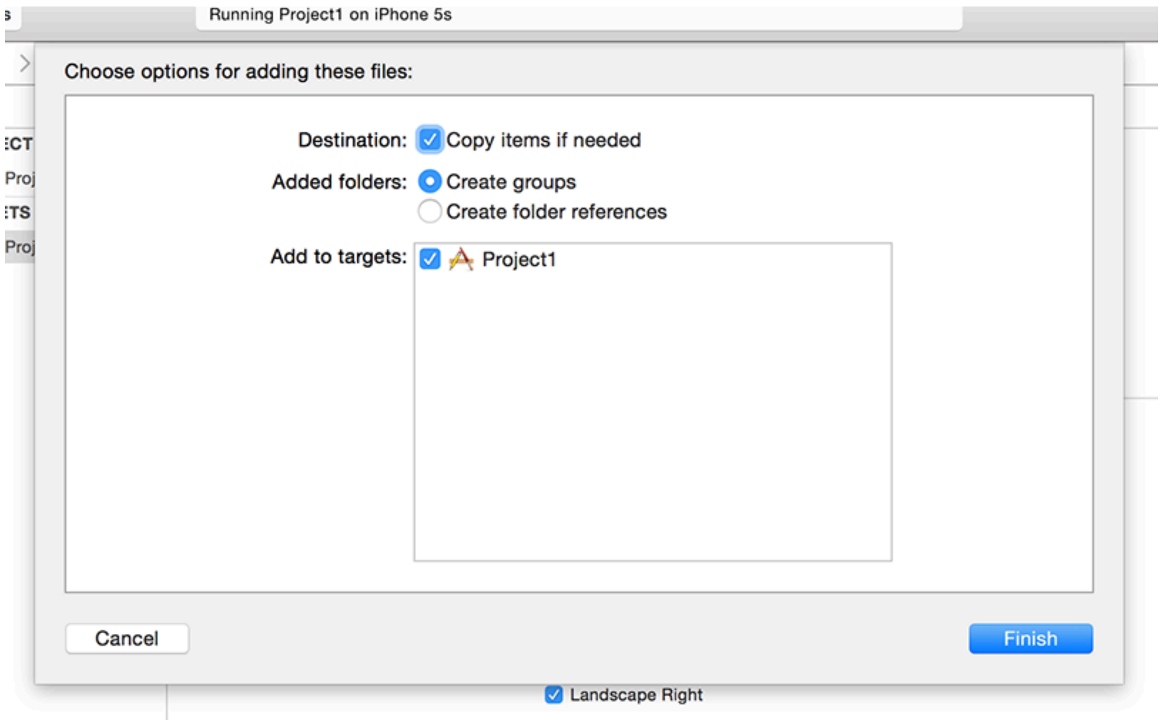
You'll be starting and stopping projects a lot as you learn, so there are three basic tips you need to know:

- 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-files" folder. You'll see another folder in there called Content, and I'd like 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.**


Running Project1 on iPhone 5s

Choose options for adding these files:

Destination: [✓] Copy items if needed

Added folders: [● Create groups] ○ Create folder references

Add to targets: [✓] Project1

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!

REALLY IMPORTANT WARNING: Several versions of Xcode managed to ship with a critical bug that affects files being added to projects, and it may already have affected you. For some reason, Xcode appears to add files to the project, but *doesn't* add them when the project gets built, so they might as well not exist.

To find out if this has affected you, select one of the images you just added inside the project navigator, e.g. “nssl0033.jpg”. Now press Alt+Cmd+1 to activate the file inspector on the right of the Xcode window, and look for the checkbox beneath “Target Membership” – if you see an unchecked checkbox next to Project1 it means you’ve been affected by the bug.

If this bug affects you: fortunately the fix is really easy. After you add any files to Xcode, select them in the project navigator, go to the file inspector, then check the box under Target Membership. **You will need to keep doing this for all future files you add in all the projects in this book** – I’m sorry for all the hassle, but it’s an Xcode bug and I can’t magic it away.

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 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:

```swift
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()`**:

```swift
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 contents 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 extra functionality for free as you'll see in a moment.

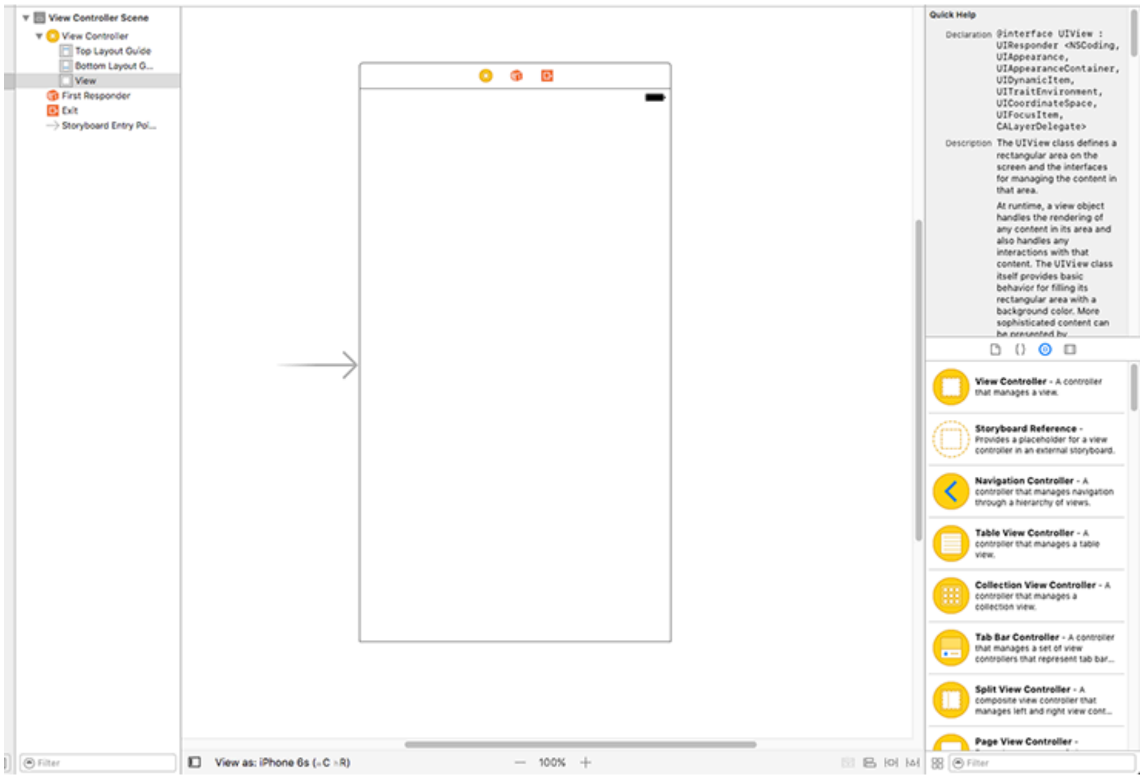
Behind the scenes, **`UITableViewController`** still builds on top of

`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:



Declaration
```
@interface UIView :
UIResponder <NSCoding,
UIAppearance,
UIAppearanceContainer,
UIDynamicItem,
UITraitEnvironment,
UICoordinateSpace,
UIFocusItem,
CALayerDelegate>
```

Description The UIView class defines a rectangular area on the screen and the interfaces for managing the content in that area.

At runtime, a view object handles the rendering of any content in its area and also handles any interactions with that content. The UIView class itself provides basic behavior for filling its rectangular area with a background color. More sophisticated content can be presented by

View Controller - A controller that manages a view.

Storyboard Reference - Provides a placeholder for a view controller in an external storyboard.

Navigation Controller - A controller that manages navigation through a hierarchy of views.

Table View Controller - A controller that manages a table view.

Collection View Controller - A controller that manages a collection view.

Tab Bar Controller - A controller that manages a set of view controllers that represent tab bar...

Split View Controller - A composite view controller that manages left and right view cont...

Page View Controller -

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.

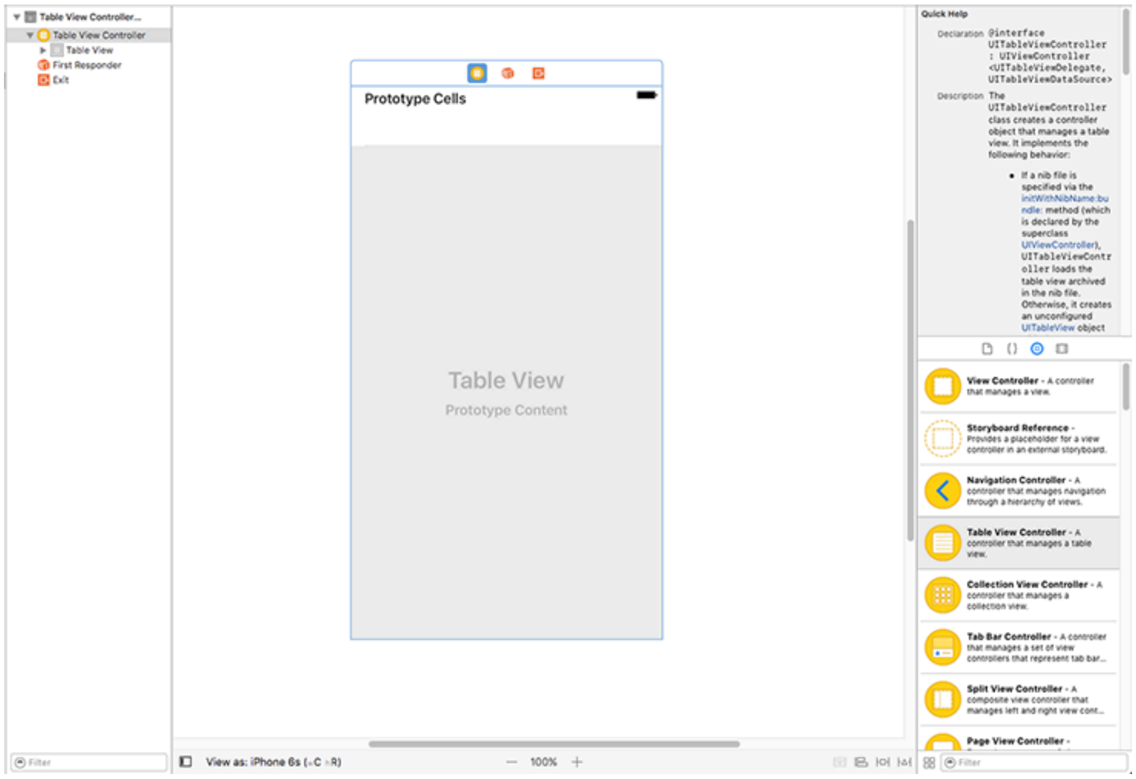
The document outline shows you all the components in all the screens in your storyboard. You should see "View Controller Scene" already in there, so please select it, then press Backspace on your keyboard to remove it.

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:



Declaration `@interface UITableViewController : UIViewController <UITableViewDelegate, UITableViewDataSource>`

Description The `UITableViewController` class creates a controller object that manages a table view. It implements the following behavior:

- If a nib file is specified via the initWithNibName:bundle: method (which is declared by the superclass UIViewController), `UITableViewController` loads the table view archived in the nib file. Otherwise, it creates an unconfigured UITableView object

View Controller - A controller that manages a view.

Storyboard Reference - Provides a placeholder for a view controller in an external storyboard.

Navigation Controller - A controller that manages navigation through a hierarchy of views.

Table View Controller - A controller that manages a table view.

Collection View Controller - A controller that manages a collection view.

Tab Bar Controller - A controller that manages a set of view controllers that represent tab bar...

Split View Controller - A composite view controller that manages left and right view cont...

Page View Controller -

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()`** – if you start typing "numberof" then you can use Xcode's code completion to do most of the work for you:

```
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 it's 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 gives 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…

Building a detail screen

At this point in our app, we have a list of pictures to choose from, but although we can tap on them nothing happens. Our next goal is to design a new screen that will be shown when the user taps any row. We're going to make it show their selected picture full screen, and it will slide in automatically when a picture is tapped.

This task can be split into two smaller tasks. First, we need to create some new code that will host this detail screen. Second, we need to draw the user interface for this screen inside Interface Builder.

Let's start with the easy bit: create new code to host the detail screen. From the menu bar, go to the File menu and choose New > File, and a window full of options will appear. From that list, choose iOS > Cocoa Touch Class, then click Next.

You'll be asked to name the new screen, and also tell iOS what it should build on. Please enter "DetailViewController" for the name, and "UIViewController" for "Subclass Of". Make sure "Also create XIB file" is deselected, then click Next and Create to add the new file.

That's the first job done – we have a new file that will contain code for the detail screen.

The second task takes a little more thinking. Go back to Main.storyboard, and you'll see our existing two view controllers there: that's the navigation view controller on the left, and the table view controller on the right. We're going to add a new view controller – a new screen – now, which will be our detail screen.

First, look in the bottom-right corner of the Xcode window for the object library, and find "View Controller" in there. Drag it out into the space to the right of your existing view controller. You could place it anywhere, really, but it's nice to arrange your screens so they flow logically from left to right.

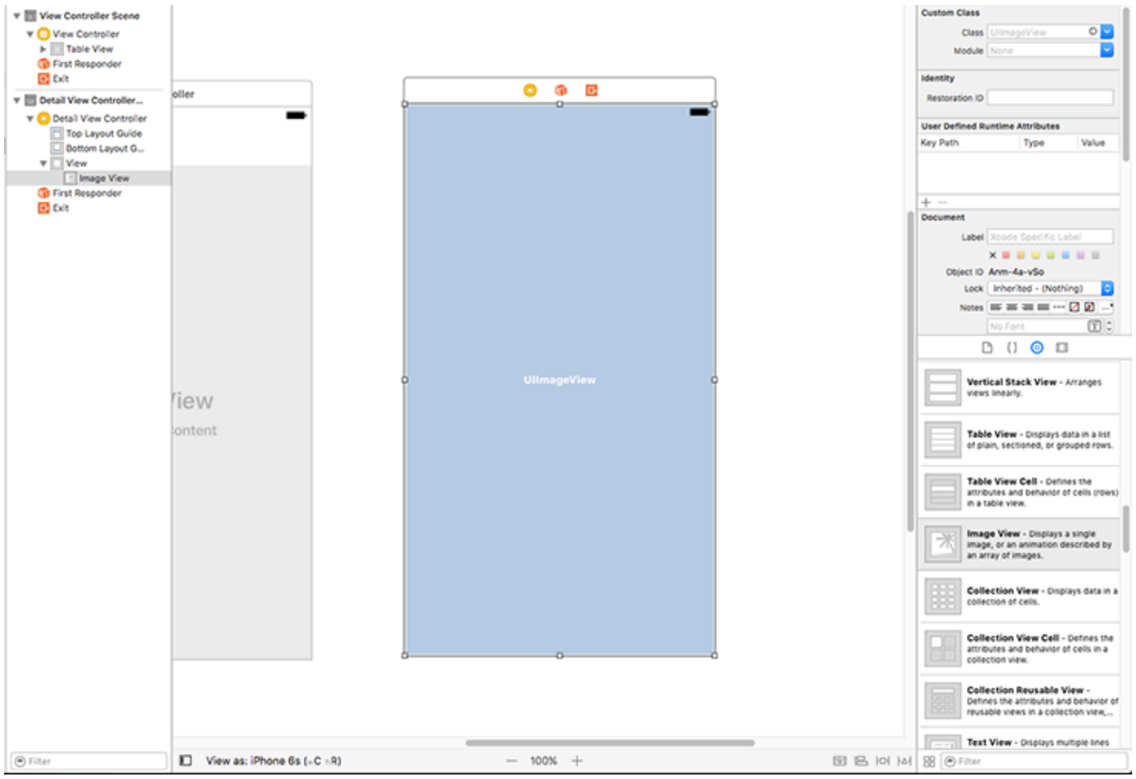
Now, if you look in the document outline you'll see a second "View Controller scene" has appeared: one for the table view, and one for the detail view. If you're not sure which is which, just click in the new screen – in the big white empty space that just got created – and it should select the correct scene in the document outline.

When we created our table view cell previously, we gave it an identifier so that we could load it in code. We need to do the same thing for this new screen. When you selected it a moment ago, it should have highlighted “View” in the document outline. Above that will be “View Controller” with a yellow icon next to it – please click on that to select the whole view controller now.

To give this view controller a name, go to the identity inspector by pressing Cmd+Alt+3 or by using the menu. Now enter “Detail” where it says “Storyboard ID”. That’s it: we can now refer to this view controller as “Detail” in code. While you’re there, please click the arrow next to the Class box and select “DetailViewController” so that our user interface is connected to the new code we made earlier.

Now for the interesting part: we want this screen to display the user’s selected image nice and big, so we need to use a new user interface component called `UIImageView`. As you should be able to tell from the name, this is a part of UIKit (hence the “UI”), and is responsible for viewing images – perfect!

Look in the object library to find Image View; you might find it easiest to use the filter box again. Click and drag the image view from the object library onto the detail view controller, then let go. Now, drag its edges so that it fills the entire view controller.



This image view has no content right now, so it's filled with a pale blue background and the word `UIImageView`. We won't be assigning any content to it right now, though – that's something we'll do when the program runs. Instead, we need to tell the image view how to size itself for our screen, whether that's iPhone or iPad.

This might seem strange at first, after all you just placed it to fill the view controller, and it has the same size as the view controller, so that should be it, right? Well, not quite. Think about it: there are lots of iOS devices your app might run on, all with different sizes. So, how should the image view respond when it's being shown on a 6 Plus or perhaps even an iPad?

iOS has an answer for this. And it's a brilliant answer that in many ways works like magic to do what you want. It's called Auto Layout: it lets you define rules for how your views should be laid out, and it automatically makes sure those rules are followed.

But – and this is a big but! – it has two rules of its own, both of which must be followed by you:

- Your layout rules must be complete. That is, you can't specify only an X position for something, you must also specify a Y position. If it's been a while since you were at school, "X" is position from the left of the screen, and "Y" is position from the top of the screen.

- Your layout rules must not conflict. That is, you can't specify that a view must be 10 points away from the left edge, 10 points away from the right edge, and 1000 points wide. An iPhone 5 screen is only 320 points wide, so your layout is mathematically impossible. Auto Layout will try to recover from these problems by breaking rules until it finds a solution, but the end result is never what you want.

You can create Auto Layout rules – known as *constraints* – entirely inside Interface Builder, and it will warn you if you aren't following the two rules. It will even help you correct any mistakes you make by suggesting fixes. Note: the fixes it suggests *might* be correct, but they might not be – tread carefully!

We're going to create four constraints now: one each for the top, bottom, left and right of the image view so that it expands to fill the detail view controller regardless of its size. There are lots of ways of adding Auto Layout constraints, but the easiest way right now is to select the image view then go to the Editor menu and choose > Resolve Auto Layout Issues > Reset To Suggested Constraints.

You'll see that option listed twice in the menu because there are two subtly different options, but in this instance it doesn't matter which one you choose. If you prefer keyboard shortcuts, press Shift+Alt+Cmd+= to accomplish the same thing.

Visually, your layout will look pretty much identical once you've added the constraints, but there are two subtle differences. First, there's a thin blue line surrounding the **`UIImageView`** on the detail view controller, which is Interface Builder's way of showing you that the image view has a correct Auto Layout definition.

Second, in the document outline pane you'll see a new entry for "Constraints" beneath the image view. All four constraints that were added are hidden under that Constraints item, and you can expand it to view them individually if you're curious.

With the constraints added, there's one more thing to do here before we're finished with Interface Builder, and that's to connect our new image view to some code. You see, having the image view inside the layout isn't enough – if we actually want to *use* the image view inside code, we need to create a property for it that's attached to the layout.

This property is like the `pictures` array we made previously, but it has a little bit more “interesting” Swift syntax we need to cover. Even more cunningly, it’s created using a really bizarre piece of user interface design that will send your brain for a loop if you’ve used other graphical IDEs.

Let’s dive in, and I’ll explain on the way. Xcode has a special display layout called the Assistant Editor, which splits your Xcode editor in two: the view you had before on top, and a related view at the bottom. In this case, it's going to show us Interface Builder on top, and the code for the detail view controller below.

Xcode decides what code to show based on what item is selected in Interface Builder, so make sure the image view is still selected and choose View > Assistant Editor > Show Assistant Editor from the menu. You can also use the keyboard shortcut Alt+Cmd+Return if you prefer.

Xcode can display the assistant editor as two vertical panes rather than two horizontal panes. I find the horizontal panes easiest – i.e., one above the other. You can switch between them by going to View > Assistant Editor and choosing either Assistant Editors On Right or Assistant Editors on Bottom.

Regardless of which you prefer, you should now see the detail view controller in Interface Builder in one pane, and in the other pane the source code for DetailViewController.swift. Xcode knows to load DetailViewController.swift because you changed the class for this screen to be “DetailViewController” just after you changed its storyboard ID.

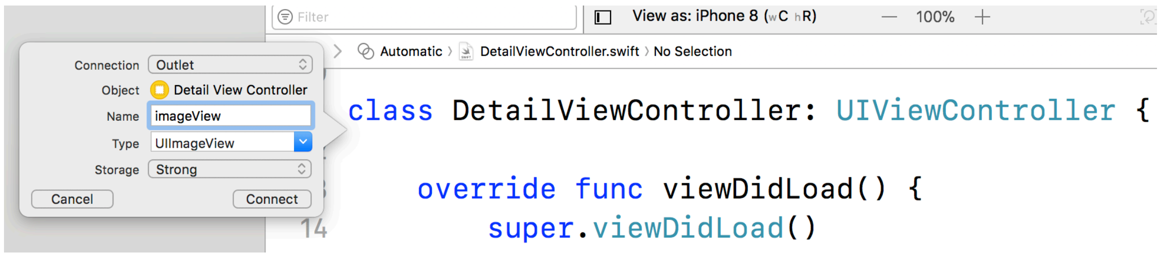
Now for the bizarre piece of UI. What I want you to do is this:

1. Make sure the image view is selected.
2. Hold down the Ctrl key on your keyboard.
3. Press your mouse button down on the image view, but hold it down – don’t release it.
4. While continuing to hold down Ctrl and your mouse button, drag from the image view into your code – into the other assistant editor pane.
5. As you move your mouse cursor, you should see a blue line stretch out from the image

view into your code. Stretch that line so that it points between **`class DetailViewController: UIViewController {`** and **`override func viewDidLoad() {`**.

6. When you're between those two, a horizontal blue line should appear, along with a tooltip saying Insert Outlet Or Outlet Connection. When you see that, let go of both Ctrl and your mouse button. (It doesn't matter which one you release first.)

If you follow those steps, a balloon should appear with five fields: Connection, Object, Name, Type, and Storage.



Connection: Outlet
Object: Detail View Controller
Name: imageView
Type: UIImageView
Storage: Strong

```
class DetailViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
```

By default the options should be "Outlet" for connection, "Detail View Controller" for Object, nothing for name, "UIImageView" for type, and "Strong" for storage.

Leave all of them alone except for Name – I'd like you to enter "imageView" in there. When you've done that click the Connect button, and Xcode will insert a line of code into DetailViewController.swift. You should see this:

```
class DetailViewController: UIViewController {
    @IBOutlet var imageView: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()
```

To the left of the new line of code, in the gutter next to the line number, is a gray circle with a line around it. If you move your mouse cursor over that you'll see the image view flash – that little circle is Xcode's way of telling you the line of code is connected to the image view in your storyboard.

So, we Ctrl-dragged from Interface Builder straight into our Swift file, and Xcode wrote a line of code for us as a result. Some bits of that code are new, so let's break down the whole line:

- `@IBOutlet`: This attribute is used to tell Xcode that there's a connection between this line of code and Interface Builder.
- `var`: This declares a new variable or variable property.
- `imageView`: This was the property name assigned to the `UIImageView`. Note the way capital letters are used: variables and constants should start with a lowercase letter, then use a capital letter at the start of any subsequent words. For example, `myAwesomeVariable`. This is sometimes called camel case.
- `UIImageView!`: This declares the property to be of type `UIImageView`, and again we see the implicitly unwrapped optional symbol: `!`. This means that that `UIImageView` may be there or it may not be there, but we're certain it definitely will be there by the time we want to use it.

If you were struggling to understand implicitly unwrapped optionals (don't worry; they are complicated!), this code might make it a bit clearer. You see, when the detail view controller is being created, its view hasn't been loaded yet – it's just some code running on the CPU.

When the basic stuff has been done (allocating enough memory to hold it all, for example), iOS goes ahead and loads the layout from the storyboard, then connects all the outlets from the storyboard to the code.

So, when the detail controller is first made, the `UIImageView` doesn't exist because it hasn't been created yet – but we still need to have some space for it in memory. At this point, the property is `nil`, or just some empty memory. But when the view gets loaded and the outlet gets connected, the `UIImageView` will point to a real `UIImageView`, not to `nil`, so we can start using it.

In short: it starts life as `nil`, then gets set to a value before we use it, so we're certain it won't ever be `nil` by the time we want to use it – a textbook case of implicitly unwrapped optionals. If you still don't understand implicitly unwrapped optionals, that's perfectly fine – keep on going and they'll become clear over time.

That’s our detail screen complete – we’re done with Interface Builder for now, and can return to code. This also means we’re done with the assistant editor, so you can return to the fullscreen editor by going to View > Standard Editor > Show Standard Editor.

Loading images with UIImage

At this point we have our original table view controller full of pictures to select, plus a detail view controller in our storyboard. The next goal is to show the detail screen when any table row is tapped, and have it show the selected image.

To make this work we need to add another specially named method to **`ViewController`**. This one is called **`tableView(_, didSelectRowAt:)`**, which takes an **`IndexPath`** value just like **`cellForRowAt`** that tells us what row we're working with. This time we need to do a bit more work:

1. We need to create a property in **`DetailViewController`** that will hold the name of the image to load.
2. We'll implement the **`didSelectRowAt`** method so that it loads a **`DetailViewController`** from the storyboard.
3. Finally, we'll fill in **`viewDidLoad()`** inside **`DetailViewController`** so that it loads an image into its image view based on the name we set earlier.

Let's solve each of those in order, starting with the first one: creating a property in **`DetailViewController`** that will hold the name of the image to load.

This property will be a string – the name of the image to load – but it needs to be an *optional* string because when the view controller is first created it won't exist. We'll be setting it straight away, but it still starts off life empty.

So, add this property to **`DetailViewController`** now, just below the existing **`@IBOutlet`** line:

```
var selectedImage: String?
```

That's the first task done, so onto the second: implement **`didSelectRowAt`** so that it loads a **`DetailViewController`** from the storyboard.

When we created the detail view controller, you gave it the storyboard ID "Detail", which allows us to load it from the storyboard using a method called **`instantiateViewController(withIdentifier:)`**. Every view controller has a property called `storyboard` that is either the storyboard it was loaded from or nil. In the case of `ViewController` it will be Main.storyboard, which is the same storyboard that contains the detail view controller, so we'll be loading from there.

We can break this task down into three smaller tasks, two of which are new:

1. Load the detail view controller layout from our storyboard.
2. Set its `selectedImage` property to be the correct item from the `pictures` array.
3. Show the new view controller.

The first of those is done using by calling `instantiateViewController`, but it has two small complexities. First, we call it on the `storyboard` property that we get from Apple's `UIViewController` type, but it's optional because Swift doesn't know we came from a storyboard. So, we need to use `?` just like when we were setting the text label of our cell: "try doing this, but do nothing if there was a problem."

Second, even though `instantiateViewController()` will send us back a `DetailViewController` if everything worked correctly, Swift *thinks* it will return back a `UIViewController` because it can't see inside the storyboard to know what's what.

This will seem confusing if you're new to programming, so let me try to explain using an analogy. Let's say you want to go out on a date tonight, so you ask me to arrange a couple of tickets to an event. I go off, find tickets, then hand them to you in an envelope. I fulfilled my part of the deal: you asked for tickets, I got you tickets. But what tickets are they – tickets for a sporting event? Tickets for an opera? Train tickets? The only way for you to find out is to open the envelope and look.

Swift has the same problem: `instantiateViewController()` has the return type `UIViewController`, so as far as Swift is concerned any view controller created with it is actually a `UIViewController`. This causes a problem for us because we want to adjust the property we just made in `DetailViewController`. The solution: we need to tell Swift that what it has is not what it thinks it is.

The technical term for this is "typecasting": asking Swift to treat a value as a different type. Swift has several ways of doing this, but we're going to use the safest version: it effectively means, "please try to treat this as a DetailViewController, but if it fails then do nothing and move on."

Once we have a detail view controller on our hands, we can set its **`selectedImage`** property to be equal to **`pictures[indexPath.row]`** just like we were doing in **`cellForRowAt`** – that's the easy bit.

The third mini-step is to make the new screen show itself. You already saw that view controllers have an optional **`storyboard`** property that either contains the storyboard they were loaded from or nil. Well, they also have an optional **`navigationController`** property that contains the navigation controller they are inside if it exists, or nil otherwise.

This is perfect for us, because navigation controllers are responsible for showing screens. Sure, they provide that nice gray bar across the top that you see in lots of apps, but they are also responsible for maintaining a big stack of screens that users navigate through.

By default they contain the first view controller you created for them in the storyboard, but when new screens are created you can push them onto the navigation controller's stack to have them slide in smoothly just like you see in Settings. As more screens are pushed on, they just keep sliding in. When users go back a screen – i.e. by tapping Back or by swiping from left to right – the navigation controller will automatically destroy the old view controller and free up its memory.

Those three mini-steps complete the new method, so it's time for the code. Please add this method to ViewController.swift – I've added comments to make it easier to understand:

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {
   // 1: try loading the "Detail" view controller and
typecasting it to be DetailViewController
   if let vc =
storyboard?.instantiateViewController(withIdentifier: "Detail")
as? DetailViewController {
      // 2: success! Set its selectedImage property
      vc.selectedImage = pictures[indexPath.row]
```

```
        // 3: now push it onto the navigation controller
        navigationController?.pushViewController(vc, animated:
true)
    }
}
```

Let’s look at the `if let` line a bit more closely for a moment. There are three parts of it that might fail: the `storyboard` property might be nil (in which case the `?` will stop the rest of the line from executing), the `instantiateViewController()` call might fail if we had requested “Fzzzzz” or some other invalid storyboard ID, and the typecast – the `as?` part – also might fail, because we might have received back a view controller of a different type.

So, three things in that one line have the potential to fail. If you’ve followed all my steps correctly they *won’t* fail, but they have the *potential* to fail. That’s where `if let` is clever: if any of those things return nil (i.e., they fail), then the code inside the `if let` braces won’t execute. This guarantees your program is in a safe state before any action is taken.

There’s only one small thing left to do before you can take a look at the results: we need to make the image actually load into the image view in `DetailViewController`.

This new code will draw on a new data type, called `UIImage`. This doesn't have "View" in its name like `UIImageView` does, so it's not something you can view – it's not something that's actually visible to users. Instead, `UIImage` is the data type you'll use to load image data, such as PNG or JPEGs.

When you create a `UIImage`, it takes a parameter called `named` that lets you specify the name of the image to load. `UIImage` then looks for this filename in your app's bundle, and loads it. By passing in the `selectedImage` property here, which was sent from `ViewController`, this will load the image that was selected by the user.

However, we can’t use `selectedImage` directly. If you remember, we created it like this:

```
var selectedImage: String?
```

That `?` means it might have a value or it might not, and Swift doesn’t let you use these “maybes” without checking them first. This is another opportunity for **`if let`**: we can check that **`selectedImage`** has a value, and if so pull it out for usage; otherwise, do nothing.

Add this code to **`viewDidLoad()`** inside **`DetailViewController`**, *after* the call to **`super.viewDidLoad()`**:

```
if let imageToLoad = selectedImage {
   imageView.image  = UIImage(named: imageToLoad)
}
```

The first line is what checks and unwraps the option in **`selectedImage`**. If for some reason **`selectedImage`** is nil (which it should never be, in theory) then the **`imageView.image`** line will never be executed. If it has a value, it will be placed into **`imageToLoad`**, then passed to **`UIImage`** and loaded.

OK, that’s it: press play or Cmd+R now to run the app and try it out! You should be able to select any of the pictures to have them slide in and displayed full screen.

Notice that we get a Back button in the navigation bar that lets us return back to **`ViewController`**. If you click and drag carefully, you’ll find you can create a swipe gesture too – click at the very left edge of the screen, then drag to the right, just as you would do with your thumb on a phone.

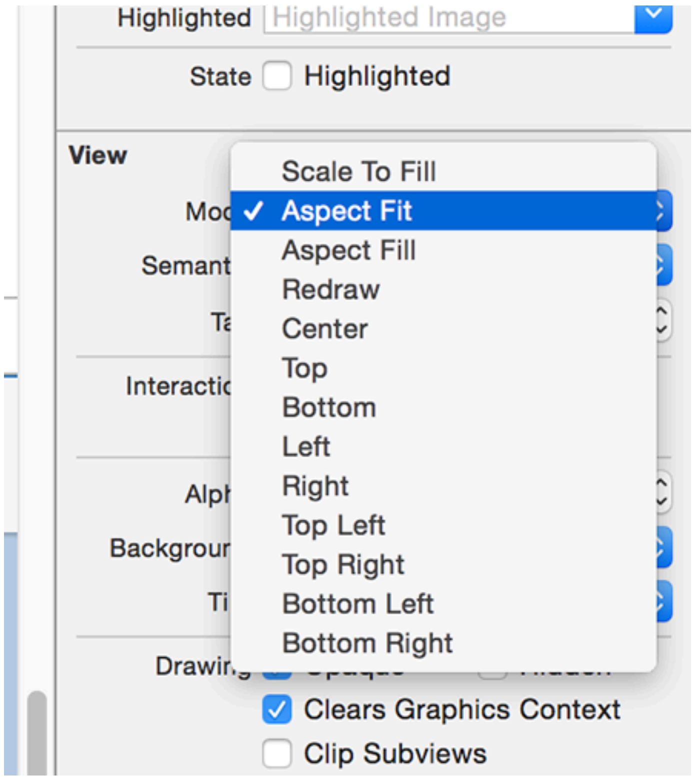
Final tweaks: hidesBarsOnTap, safe area margins

At this point you have a working project: you can press Cmd+R to run it, flick through the images in the table, then tap one to view it. But before this project is complete, there are several other small changes we're going to make that makes the end result a little more polished.

First, you might have noticed that all the images are being stretched to fill the screen. This isn't an accident – it's the default setting of `UIImageView`.

This takes just a few clicks to fix: choose Main.storyboard, select the image view in the detail view controller, then choose the attributes inspector. This is in the right-hand pane, near the top, and is the fourth of six inspectors, just to the left of the ruler icon.

If you don't fancy hunting around for it, just press Cmd+Alt+4 to bring it up. The stretching is caused by the view mode, which is a dropdown button that defaults to "Scale to Fill." Change that to be "Aspect Fit," and this first problem is solved.



If you were wondering, Aspect Fit sizes the image so that it's all visible. There's also Aspect

Fill, which sizes the image so that there's no space left blank – this usually means cropping either the width or the height. If you use Aspect Fill, the image effectively hangs outside its view area, so you should make sure you enable Clip To Bounds to avoid the image overspilling.

The second change we're going to make is to allow users to view the images fullscreen, with no navigation bar getting in their way. There's a really easy way to make this happen, and it's a property on **`UINavigationController`** called **`hidesBarsOnTap`**. When this is set to true, the user can tap anywhere on the current view controller to hide the navigation bar, then tap again to show it.

Be warned: you need to set it carefully when working with iPhones. If we had it set on all the time then it would affect taps in the table view, which would cause havoc when the user tried to select things. So, we need to enable it when showing the detail view controller, then disable it when hiding.

You already met the method **`viewDidLoad()`**, which is called when the view controller's layout has been loaded. There are several others that get called when the view is about to be shown, when it has been shown, when it's about to go away, and when it has gone away. These are called, respectively, **`viewWillAppear()`**, **`viewDidAppear()`**, **`viewWillDisappear()`** and **`viewDidDisappear()`**. We're going to use **`viewWillAppear()`** and **`viewWillDisappear()`** to modify the **`hidesBarsOnTap`** property so that it's set to true only when the detail view controller is showing.

Open DetailViewController.swift, then add these two new methods directly below the end of the **`viewDidLoad()`** method:

```swift
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    navigationController?.hidesBarsOnTap = true
}

override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)
    navigationController?.hidesBarsOnTap = false
}
```

There are some important things to note in there:

- We're using override for each of these methods, because they already have defaults defined in `UIViewController` and we're asking it to use ours instead. Don't worry if you aren't sure when to use override and when not, because if you don't use it and it's required Xcode will tell you.
- Both methods have a single parameter: whether the action is animated or not. We don't really care in this instance, so we ignore it.
- Both methods use the `super` prefix again: `super.viewWillAppear()` and `super.viewWillDisappear()`. This means "tell my parent data type that these methods were called." In this instance, it means that it passes the method on to `UIViewController`, which may do its own processing.
- We're using the `navigationController` property again, which will work fine because we were pushed onto the navigation controller stack from `ViewController`. We're accessing the property using `?`, so if somehow we *weren't* inside a navigation controller the `hidesBarsOnTap` lines will do nothing.

If you run the app now, you'll see that you can tap to see a picture full size, and it will no longer be stretched. While you're viewing a picture you can tap to hide the navigation bar at the top, then tap to show it again.

The third change is a small but important one. If you look at other apps that use table views and navigation controllers to display screens (again, Settings is great for this), you might notice gray arrows at the right of the table view cells. This is called a disclosure indicator, and it's a subtle user interface hint that tapping this row will show more information.

It only takes a few clicks in Interface Builder to get this disclosure arrow in our table view. Open Main.storyboard, then click on the table view cell – that's the one that says "Title", directly below "Prototype Cells". The table view contains a cell, the cell contains a content view, and the content view contains a label called "Title" so it's easy to select the wrong thing. As a result, you're likely to find it easiest to use the document outline to select exactly the right thing – you want to select the thing marked "Picture", which is the reuse identifier we attached to our table view cell.

When that's selected, you should be able go to the attributes inspector and see "Style: Basic", "Identifier: Picture", and so on. You will also see "Accessory: None" – please change that to "Disclosure Indicator", which will cause the gray arrow to show.

The fourth is small but important: we're going to place some text in the gray bar at the top. You've already seen that view controllers have **`storyboard`** and **`navigationController`** properties that we get from **`UIViewController`**. Well, they also have a **`title`** property that automatically gets read by navigation controller: if you provide this title, it will be displayed in the gray navigation bar at the top.

In **`ViewController`**, add this code to **`viewDidLoad()`** after the call to **`super.viewDidLoad()`**:

```
title = "Storm Viewer"
```

This title is also automatically used for the "Back" button, so that users know what they are going back to.

In **`DetailViewController`** we *could* add something like this to **`viewDidLoad()`**:

```
title = "View Picture"
```

That would work fine, but instead we're going to use some dynamic text: we're going to display the name of the selected picture instead.

Add this to **`viewDidLoad()`** in **`DetailViewController`**:

```
title = selectedImage
```

We don't need to unwrap **`selectedImage`** here because both **`selectedImage`** and **`title`** are optional strings – we're assigning one optional string to another. **`title`** is optional because it's nil by default: view controllers have no title, thus showing no text in the navigation bar.

Large titles in iOS 11

This is an entirely optional change, but I wanted to introduce it to you nice and early so you can try it for yourself and see what you think.

iOS 11 revamped Apple's design guidelines in many places, but the most noticeable is the use of *large titles* – the text that appears in the gray bar at the top of apps. The default style is small text, which is what we've had so far, but with a couple of lines of code we can adopt the new design.

First, add this to **`viewDidLoad()`** in ViewController.swift:

```
navigationController?.navigationBar.prefersLargeTitles = true
```

That enables large titles across our app, and you'll see an immediate difference: "Storm Viewer" becomes much bigger, and in the detail view controller all the image titles are also big. You'll notice the title is no longer static either – if you pull down gently you'll see it stretches ever so slightly, and if you try scrolling up in our table view you'll see the titles shrinks away.

Apple recommends you use large titles only when it makes sense, and that usually means only on the first screen of your app. As you've seen, the default behavior when enabled is to have large titles everywhere, but that's because each new view controller that pushed onto the navigation controller stack inherits the style of its predecessor.

In this app we want "Storm Viewer" to appear big, but the detail screen to look normal. To make that happen we need to add a line of code to **`viewDidLoad()`** in DetailViewController.swift:

```
navigationItem.largeTitleDisplayMode = .never
```

That's all it takes – the large titles should behave properly now.

And what about iPhone X?

iPhone X introduces the first iPhone screen that isn't rectangular, which creates some interesting problems. In particular, the rounded corners of the screen mean that content can easily be clipped if it isn't positioned correctly, so the system tries to adapt by making *safe areas* – parts of the screen that are automatically avoided so that clipping doesn't happen.

In this app, our table view controller will automatically run edge to edge because Apple has done all the hard work for us, but the image in our detail view controller won't – it will have a white gap at the bottom to leave space for the home indicator.

That is sometimes what you'll want, but here it looks pretty poor. Fortunately, it's trivial to fix: open Main.storyboard, select the view inside Detail View Controller, then uncheck the Safe Area Layout Guide box in the size inspector. That will ask the view (and all its subviews) to run edge to edge, which looks better.

While we're talking about the iPhone X, let's extend our use of **`hidesBarsOnTap`** so that the home indicator gets hidden too. This is controlled by overriding a new method called **`prefersHomeIndicatorAutoHidden()`** – if that returns true, the home indicator will disappear after a couple of seconds, only to automatically reappear when the user touches the screen.

We're going to write this method so that it returns the value of our navigation controller's **`hidesBarsOnTap`** property, meaning that the bars and the home indicator should disappear or reappear together.

We should in theory always have a navigation controller so we could probably force unwrap it and read its property. However a better idea is to use it optionally and provide **`false`** as the default value if the navigation controller isn't present. Add this method to DetailViewController.swift now:

```
override func prefersHomeIndicatorAutoHidden() -> Bool {
    return navigationController?.hidesBarsOnTap ?? false
}
```

Note: The **`??`** operator is called the *nil coalescing operator*, and in this case it means "if the navigation controller doesn't exist, send back **`false`** rather than trying to read its **`hidesBarsOnTap`** property."

That method gets checked when the view controller is first shown, but will also automatically get called whenever the bars get toggled using `hidesBarsOnTap`. So, that's our work done: the home indicator should automatically disappear a few seconds after the bars disappear.

That's the last of the changes – we're done! Go ahead and run the project now and admire your handiwork.

Wrap up

This has been a very simple project in terms of what it can do, but you've also learned a huge amount about Swift, Xcode and storyboards. I know it's not easy, but trust me: you've made it this far, so you're through the hardest part.

To give you an idea of how far you've come, here are just some of the things we've covered: constants and variables, method overrides, table views and image views, app bundles, `FileManager`, typecasting, arrays, loops, optionals, view controllers, storyboards, outlets, Auto Layout, `UIImage` and more.

Yes, that's a *huge* amount, and to be brutally honest chances are you'll forget half of it. But that's OK, because we all learn through repetition, and if you continue to follow the rest of this series you'll be using all these and more again and again until you know them like the back of your hand.