

HACKING WITH SWIFT



HACKING WITH macOS

COMPLETE TUTORIAL COURSE

Learn to make desktop
apps with real-world
Swift projects.

FREE SAMPLE

Paul Hudson

Project 1

Storm Viewer

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

Storm Viewer: 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!

Note: If you already learned iOS development with *Hacking with iOS*, this project builds the same app you made in project 1 of *Hacking with iOS*. If you've haven't read *Hacking with iOS*, don't worry – this project serves as a gentle introduction to macOS development. But if you *have* read *Hacking with iOS*, I hope the similarity of these two projects will help lower your learning curve a little.

Let's get started: launch Xcode, and choose "Create a new project" from the welcome screen. You'll be asked to choose a template for the new project, so please choose macOS > Cocoa Application, then click Next.



For Product Name enter Project1, then make sure you have Swift selected for language. This screen contains five different checkboxes that affect what kind of template you're given. None of the projects in this book use the bottom three, so they should always be unchecked. For this project, I'd like you to check "Use storyboards" and leave "Create Document-Based Application" unchecked.

One of the fields you'll be asked for is "Organization Identifier", which is a unique identifier usually made up of your personal website 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**.



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: look for the Play triangle button at the top-left of the Xcode window,

Project 1: Storm Viewer

and click it now. This will compile your code, which is the process of converting it to instructions that your computer can understand, then launch the app.

As you'll see when you interact with the app, our “app” just shows an empty window – it does nothing at all, at least not yet. You can resize it and move it around, minimize it, or even make it full screen, but it’s still just a big empty window.

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. You should have downloaded the project files for this book already, but if you haven't please get them from here: <https://github.com/twostraws/macOS>

If you look in the project1-files directory you'll see another folder called Content. I want you to drag that Content folder straight into your Xcode project, just under where it says "Info.plist".

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 warning 1: Do not choose "Create folder references" otherwise your project will not work.

Important warning 2: Make sure the box “Project1” next to “Add to targets” is checked.

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!



Splitting a window in two

Our app is going to show a list of pictures for the user to choose from, then show individual pictures zoomed large when one is chosen. This split approach is a very common layout on macOS: you see it in Finder, in iTunes, in Keynote, and even in Xcode, so it makes sense to tackle it first.

The project template Xcode made for you gives you two important files: `ViewController.swift`, and `Main.storyboard`. A “view controller” is Apple’s way of letting you control one piece of content on the screen. It might occupy all of a window or just part of it, and some types of view controller exist just to manage other view controllers.

In this app, we’ll be using three view controllers. The first will handle a table of data on the left of the window, showing a list of pictures for the user to choose from. The second will contain the image the user selected, nice and big so they can see the details. The third is one of those “management” view controllers: it will create both the other two with a divider between them, and ensure they both get space in the window.

All this layout is controlled in the file `Main.storyboard`. Storyboards are Xcode’s way of letting you plan out your app’s user interface in a visually logical way, and they also contain links back to your code so you can reference your user interface in Swift.

Open `Main.storyboard` now, and a new editing layout will be shown. This is called Interface Builder (IB for short), and it’s a graphical user interface design tool. On iOS IB is used extensively, but on macOS it has quite a few extra powers that don’t exist on iOS. So, while many iOS developers like to write their interfaces entirely in code, this is less common on macOS because you miss out on so much of the extra power IB has to offer for macOS developers.



Your storyboard will contain three “scenes”, which are independent parts of your user interface:

1. The first is called the application scene, and it contains the menu bar that runs across the top

- of the screen in macOS. It also contains a special object called “App Delegate”, which tells macOS where to send application messages.
2. The second is the window controller scene. Cunningly, it’s the one with “View Controller” written in the middle, even though it’s not a view controller. This scene is made of a window (the visual thing, with buttons and a title) and a window controller (code that determines how the window should behave).
 3. The third is the view controller scene. It has “View Controller” written in small letters at the top, but is otherwise blank. This scene is made of a view (the visual thing, which is currently empty) and a view controller (code that determines how the view should behave.)

There are three further things I’d like to point out before we start making changes.

First, notice the relationship between a window and a window controller is the same as between a view and a view controller: one is responsible for drawing things to the screen, and the other is responsible for decide how things should behave.

This split is an important part of application development on Apple’s platforms, and is commonly called the Model View Controller paradigm. What it means is that you keep your data (the model), your layout code (the views), and your business logic (the controllers) separated so your project doesn’t end up like spaghetti. In practice things aren’t always so clean cut, but it’s a goal you should aim for.

Second, notice the right-facing arrow pointing at the window controller scene. This marks the initial controller for the storyboard, which is the macOS way of deciding what should be shown when the program first runs.

Finally, notice the downwards-facing arrow that connects the window controller scene to the view controller scene – the one with a circle in the middle. This is a segue, which is IB’s way of showing relationships between two scenes. In this case, it means the view controller is used for the content of the window – it gets placed inside the window when the app runs.

OK, enough talking: time for some action. We don’t want a big gray view controller for our app, so we need to delete it. The easiest way to do this – and indeed most things in IB – is to

Project 1: Storm Viewer

use the document outline, which is a hierarchical representation of your layouts. If this isn't already showing, go to the Editor menu and choose Show Document Outline so that it's visible now.

You should see your three scenes in the document outline, so select View Controller Scene and press delete on your keyboard to zap it – we don't need any of it. Instead, we're going to drop a new kind of view controller in there called a Split View Controller.



IB offers a variety of UI objects that you can drag onto its canvas, and these are all found in the object library. This appears over the Xcode window when summoned by pressing Cmd+Shift+L, or by going to the View menu and choosing Libraries > Show Object Library. It starts with “Object”, then “View Controller”, but there are *many* things in there to work with – far more than you get on iOS.

Note: the object library has an icon view that shows everything using square pictures. This is hard enough to use on iOS, but frankly impossible on macOS where we have so many near-identical components to choose from. If you see a grid of icons, click the small “List View” icon at the bottom of the grid to switch to a more sensible layout.

Fortunately for all of us, there's a filter box at the top of the object library, which lets you type in a few letters to search for particular UI elements. If you type “split” in there the list should be filtered down to just four things: Vertical Split View Controller, Horizontal Split View Controller, Vertical Split View, and Horizontal Split View. They all sound similar, and they *are* similar, but I'd like you to choose the first one: Vertical Split View Controller.

I already said that windows and window controllers are different things, as are views and view controllers. What I *didn't* say is that one wraps the other: a window controller provides logic for a window, and it actually contains that window inside it. The same is true of a view controller: each view controller has a view that belongs to it, which it controls.

The difference between a vertical and horizontal split view is self-explanatory, but the difference between a split view and a split view *controller* is just like the difference between a view and a view controller: a split view is just a visual layout component that shows two things

side by side with a divider between them, but a split view *controller* has a split view inside it and adds the ability to write logic to control its behavior.

Don't worry if that's not clear just yet – give it time.

For now, I want you to click on Vertical Split View Controller in the object library, then drag it out into the storyboard's canvas to where the previous view controller was. As you do this, you'll see ghost-like shapes of three new view controllers appear: a split view controller on the left, plus two new view controllers on the right. When you let go of your mouse button, these new view controllers will be created.



Tip for iOS Developers: `NSSplitViewController` is similar to `UISplitViewController`, but far more configurable and far more commonly used on macOS.

When we deleted the original view controller, the window updated to say “No Content View Controller” – that’s IB’s way of saying the window doesn’t have anything to show. Now that we have a replacement with our split view controller, we can fix that.

Using the document outline, select “Window Controller” inside the window controller scene. Now hold down Ctrl, then click and drag a line from “Window Controller” to the split view controller on the canvas. As you drag, a blue line should follow your mouse cursor, and the whole split view controller should glow blue when you move your mouse over it. This whole line drawing thing is a peculiarity of IB, but it’s very common – in the future I’ll just be saying “Ctrl-drag”.



When that blue glow appears, let go of your mouse button and a menu should appear saying “Relationship segue” and “window content”. Please click on “window content” to establish a link between the window and the split view controller – it means our new split view controller will get shown when the window is created, which itself gets shown when the app is run.



Press `Cmd+R` now to build and run your app, and you’ll see it looks ever-so-slightly different.

Project 1: Storm Viewer

Yes, it's still mostly empty grayness, but there's now a thin line down the center, and you should be able to click on that line and drag it around. That's our split view in action – success!

Adding custom controllers

When we created the split view controller, it provided us with two new view controllers: the one on the top is used to fill the left pane of the split view, and the one of the bottom is used for the right pane. These view controllers are constructed using a class called **NSViewController**, which provides basic functionality such as notifying you when it's shown or hidden, adjusting its layout when the user resizes the window, and so on.

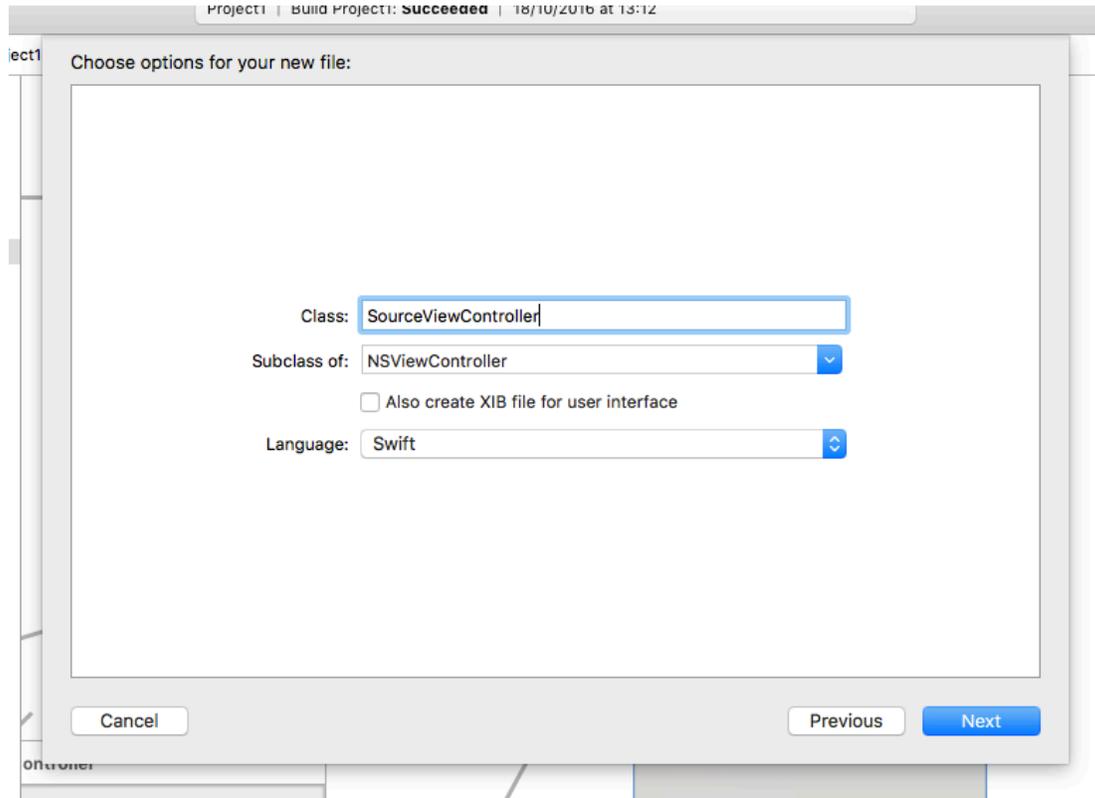
Tip for iOS Developers: **NSViewController** is similar to **UIViewController**, although much newer. A number of recent macOS releases have helped flesh out **NSViewController** to be more useful.

“Classes” are one of Swift’s ways of creating new data types, and they let you create complex functionality by attaching custom code to the data types you define.

What we’re going to do is create two custom versions of **NSViewController**, one for each side of our split view. This customization is called subclassing, or *inheritance*, because the new class you define inherits all the functionality of **NSViewController** and adds its own tweaks.

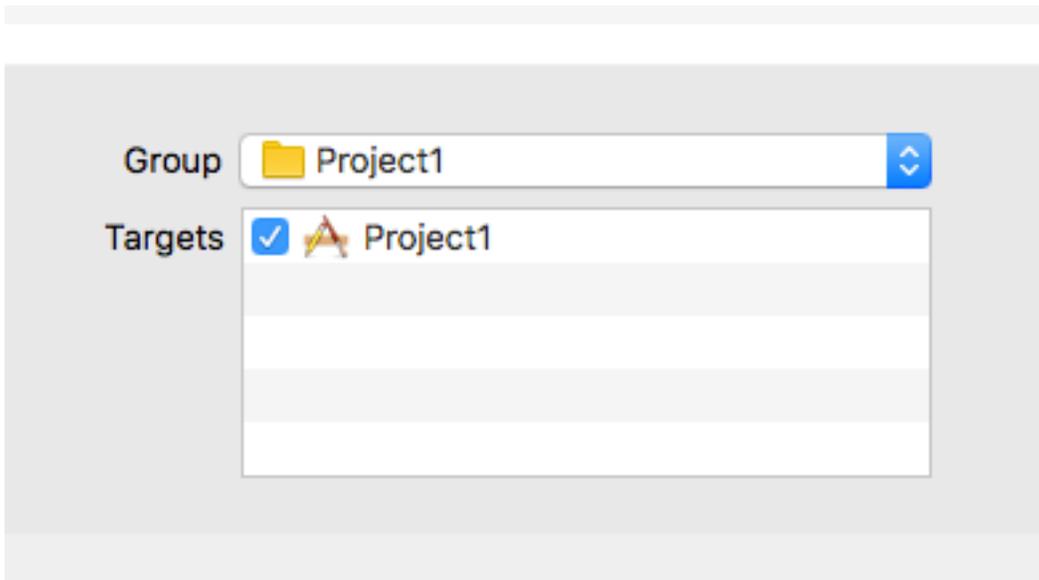
Go to the File menu in Xcode, and choose New > File. A window will appear prompting you to choose a template for the new file, and I’d like you to choose macOS > Cocoa Class, then click Next. You’ll be asked to provide a name for your custom class, so please enter “SourceViewController”. You’ll also be asked what it should be a subclass of, i.e. what it should inherit its behavior from. Please enter “NSViewController”. Make sure the “Also create XIB file for user interface” is not checked, then click Next.

Project 1: Storm Viewer



The next window asks you where you want to save the new file, and there's one thing you need to check carefully. In fact, it's so important I think I'd better write it in bold:

When creating new files like this, make sure you select "Project 1" with the yellow folder next to it for the Group. If it says "Project 1" with a blue project icon next to it, please change this. Always.



When you've done that, click Create to have Xcode create the new file and open it for editing. You should see something like this:

```
import Cocoa

class SourceViewController: NSViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do view setup here.
    }
}
```

Let's break down what that does:

- The **import Cocoa** line tells Swift we will be referencing Apple's Cocoa framework in our code. That gives us AppKit for user interfaces, Foundation for basic things like loading files and handling dates, and Core Data for managing object graphs.
- The line **class SourceViewController: NSViewController** means "we're creating a new class called SourceViewController, and we want it to build on NSViewController." That

Project 1: Storm Viewer

means it gets all the features of **NSViewController**, but can add its own customizations – after all, showing a large gray space is pretty dull!

- The **func viewDidLoad()** line marks the start of a function that gets called by macOS when the view being controlled by this view controller has finished loading. It has the **override** keyword before it because **NSViewController** already has a function called **viewDidLoad()** – we’re telling Swift we want to override the one we inherited and provide our own version instead.
- Just inside **viewDidLoad()** is the line **super.viewDidLoad()**, which means “call the **viewDidLoad()** method on my super class.” “Super class” is another name for the thing we inherited from, so what this effectively means is that our overridden version of **viewDidLoad()** starts by calling the version that’s built into **NSViewController**.
- The line **// Do view setup here.** starts with two slashes, which makes it a comment – this code is ignored when your code is built.

Note: Functions that exist inside classes, like our **viewDidLoad()**, are usually called “methods”. Swift uses the **func** keyword for both of them, but I’ll be referring to them as “methods” from now on.

All that code effectively does nothing, because even though we have a custom **viewDidLoad()** method, it just calls the standard **viewDidLoad()** that it got from **NSViewController**. Don’t worry, though: as the comment says, we now have somewhere we can add our own view set up.

We’ll come back to this code soon enough, but first I want you to create a *second* new view controller, this time called “DetailViewController”. See if you can do it from memory, but if not my instructions below - don’t worry, we’ll be repeating this process a *lot* in the rest of the book!

To create a new view controller, go to File > New > File, then choose macOS > Cocoa Class. Click Next, then name it “DetailViewController”. Make it a subclass of “NSViewController”, make sure “Also create XIB file for user interface” is unchecked, then click Next again. Finally, make certain that Group has a yellow folder icon next to it, then click Create.

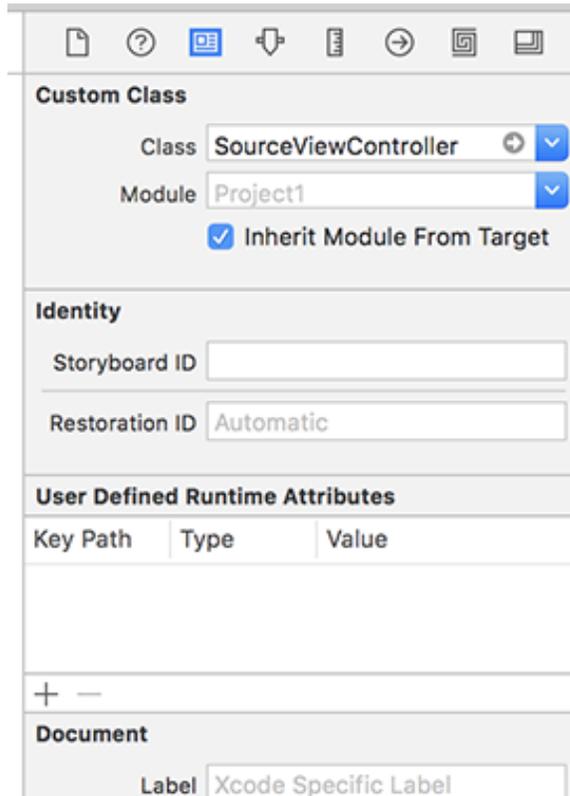
Xcode will create a second view controller that's identical to the first one, except you'll now see **class DetailViewController: NSViewController** – it has a different name.

We've just created two new view controllers in code, but Xcode doesn't understand that those are to be used inside our split view controller – there's no link between our storyboard and the code that Xcode just created for us.

To make that link, we need to return to Main.storyboard. Using the document outline, open the first View Controller Scene and select "View Controller". That's going to be our **SourceViewController**, and to make that link we need to use one of the inspectors inside IB – these are option panes that live on the right of IB, offering up a wealth of options to customize whatever you selected.

If you press Alt+Cmd+3, IB will show the identity inspector in the right-hand side of its window. The first option there is "Class", and you'll see "NSViewController" written in light gray text – that means it's the default value. Click in that box and start typing "SourceViewController"; all being well Xcode will autocomplete after you type "so" to save you the rest of the work.

Project 1: Storm Viewer



Note: If you see a default value of “NSView” rather than “NSViewController” it means you selected the view inside the view controller, rather than the view controller itself.

What we’ve just done is tell Xcode that when it creates the top view controller, it should attach it to the code inside `SourceViewController.swift` – we’ve created a link between our UI and our code.

Now to do the same for the second view controller. Again, use the document outline to select the second view controller scene, select the view controller inside it, then use the identity inspector to change its class. This time I’d like you to give it the class “`DetailViewController`”.

Designing our interface

All our work so far has been pretty dull, and – worse – it looks only fractionally different from the empty gray window we had straight out of the Xcode template.

However, things are about to get a bit more interesting: we’re going to start designing our user interface for real.

Let’s start with the source view controller first. This is going to contain a list of pictures for the user to choose from, and in macOS that’s handled by a component called **NSTableView**. If you search in the object library for “table” you’ll see it listed as “Table View”. Please click on that, then drag it out into the top view controller of the split view.



Tip for iOS Developers: **NSTableView** is analogous **UITableView**, although it’s much older and a lot cruftier. It does have some extra features, such as the ability to have columns of data and alternating grid lines.

You can place it wherever you want at first, but once it’s on the canvas I want you to move it so that its flush against the top-left corner of the view controller.

When the table view is selected, you’ll see eight resize handles on its corners and sides. Click and drag the bottom-right resize handle, then use it to stretch the table view so that it fills its view controller.

Now, take a look in the document outline, and you’ll see something that’s confusing at first: inside the view is a “Bordered Scroll View”, which contains a “Clip View”, which in turn contains our table view. This is how macOS handles scrolling inside the table view: the scroll view handles moving the table around, and the clip view ensures that the table doesn’t show outside the visible area of the scroll view.

There are a couple of other components that use a similar structure, and it’s an easy place for beginners to get confused because you can accidentally select the wrong thing. To avoid problems, it is almost always easier to select things using the document outline until you get

Project 1: Storm Viewer

the hang of it.

So, use the document outline to select the table view - we're going to make a few changes to the way it looks. With the table view selected, press `Alt+Cmd+4` to bring up a new inspector, called the attributes inspector. You should see lots of options appear in attributes inspector – so many that you almost certainly have to scroll to see them all.

Here's what I want you to change:

- Change Columns from 2 to 1. We don't need two columns, because this table is just going to show the name of each picture.
- Uncheck the "Headers" checkbox. These add clickable gray bars at the top of the table, just like in Finder's list view.
- Change Highlight from Regular to Source List. This automatically adds the "frosted glass" translucency effect that is common in macOS apps.



Before we're done with the table view, we're going to make two further changes. Select "Table View" in the document outline, then `Ctrl-drag` from there to "Source View Controller". When you release, choose "dataSource" from the menu that appears. Now do the same thing again, this time choosing "delegate" from the menu.

If you've used **UITableView** on iOS before, this will all be very familiar: you just told IB that the **SourceViewController** class is responsible for telling the table view what it contains (its data source), and should also be notified when the user performs any actions (its delegate). We'll be writing code for that soon, but first we have some more work in the storyboard.

So far we've created and configured an **NSTableView** for the left-hand pane of our split view. This will show a list of pictures for the user to choose from, but when they've made a choice we want to show that in the right-hand pane of the split view – the second view controller.

To make this work, we're going to use a different component from the object library. Use the filter box to search for "image", then drag an "Image View" into the bottom view controller – that's the one that will be shown on the right. As with the table view, I'd like you to position

the image view so that it is flush against the top-left edge of its container, then use the bottom-right resize handle to stretch it so that it occupies the full space available.

That's our complete user interface complete, but we're not done with IB just yet...

Introducing outlets

Earlier we created new classes for our two view controllers, and we then used the attributes inspector in IB to connect the view controller in the storyboard to the classes we created.

Well, we just created two new UI components: a table view (**NSTableView**), and an image (**UIImageView**). We need a way to be able to reference those things in code, and IB's solution for this is quite bizarre at first: you Ctrl-drag from IB directly into your code.

Now, before I show you how this is done, a repeated warning: when you work with table views, you actually have a table view inside a clip view inside a scroll view. When I tell you to Ctrl-drag from the table view, you need to make absolutely sure you have the table view selected in the document outline, and not the scroll view or clip view that surround it.

OK, let's give this a try. Select the first view controller – that's the top one, then press Alt+Cmd+Return. This activates the assistant editor, but you can also go to the View menu and choose Assistant Editor > Show Assistant Editor if you're more of a mouse person.

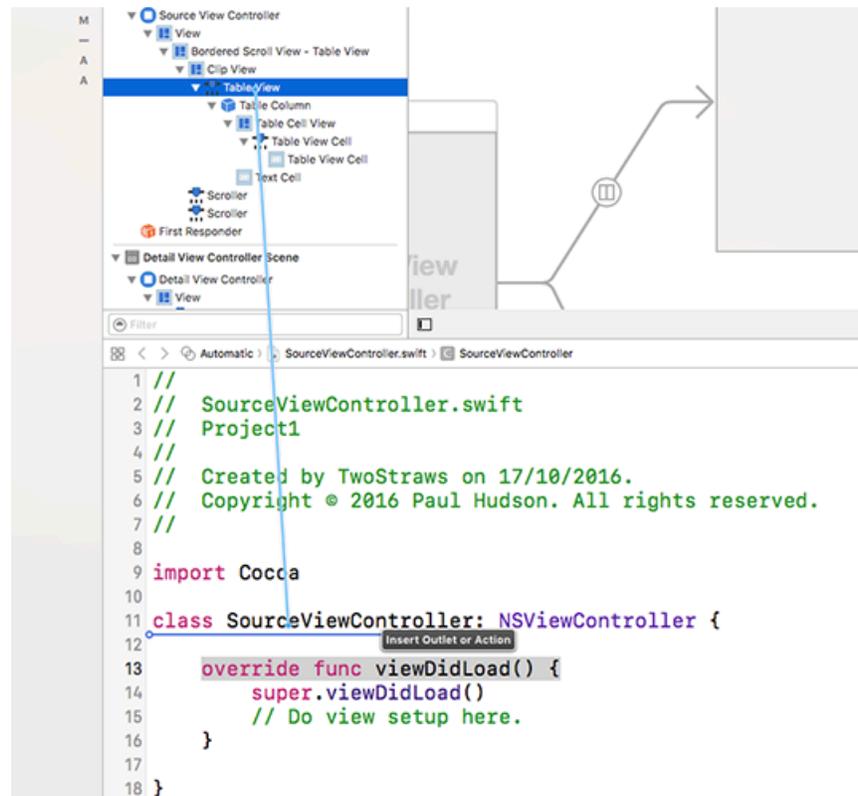
If everything is working well, you should see IB in the top half of Xcode, and the source code for `SourceViewController.swift` in the bottom half of Xcode. This is the assistant editor in action: it looks at what you have selected in the top pane, and shows the most relevant accompanying file in the bottom pane.

If the assistant editor got it wrong: look at the split between the top and bottom panes. You might see left and right chevrons on its right edge, and hopefully clicking one of those will show the correct file in the bottom pane. If that doesn't work either, look for a button on the left marked "Automatic" – click that, then select Manual > Project1 > Project1 > `SourceViewController.swift`. I find the assistant editor gets it right most of the time, but it's important to know what to do when things aren't quite right and you're dumped in `NSResponder.h`!

Anyway, you should be able to see IB and your source code at the same time. Now, using the document outline, I want you to Ctrl-drag from the table view into `SourceViewController.swift`, just above where it says **override func viewDidLoad() { Make**

sure you drag from the table view, not the scroll view.

When you do this correctly, a horizontal blue line will appear in your source code, next to the tooltip “Insert outlet or action.”



When you let go of your mouse button, a popup will appear offering various options:

- “Connection” can be either “Outlet” or “Action”. An outlet creates a property inside your class that you can reference in code. An action creates a method to be run, for example when a button is clicked. Make sure “Outlet” is selected.
- “Name” is the name you want to attach to the outlet or action. Please enter “tableView” in there.
- “Type” is the data type for this outlet, and it’s “NSTableView” by default, which is correct.
- “Storage” tells Swift whether you want to own the object (“strong”) or whether you want someone else to own it (“weak”). Please choose “Strong”.

Project 1: Storm Viewer

With all those fields set correctly, you should be able to click Connect. This will cause a new line of code to be written in `SourceViewController.swift`:

```
@IBOutlet var tableView: NSTableView!
```

That's four things in one, so let's break it down:

- The **@IBOutlet** attribute marks this property as being connectable inside IB. If you look to the left of the code you'll see a gray circle with a line around it – that's the signal that a connection exists.
- The value of the table view will change over time – specifically, when the view is created – so it's declared as a variable using **var**.
- We gave it the name “tableView”, so that appears next.
- Finally, its data type is **NSTableView!**.

That last part bears some further explanation, because it uses Swift's optionals. There are three ways you can create a table view:

- If you write **NSTableView**, it means the variable in question will always hold a table view and will never be nil.
- If you write **NSTableView?** it means the variable in question might hold a table view, but also might not – the only way to be sure is to check each time you want to use it.
- If you write **NSTableView!** it means the variable in question might hold a table but, but also might not. However, you don't want the bother of checking what it contains before using it.

If you try to access an **NSTableView!** that is empty – i.e., it hasn't been created yet, or it was created then destroyed – you'll get a crash. As a result, most Swift developers prefer to use **NSTableView?** where possible, because Swift will force you to check before trying to use the data, and thus will stop all those potential crashes.

Why, then, did Xcode just create this property as **NSTableView!** – does it want our app to

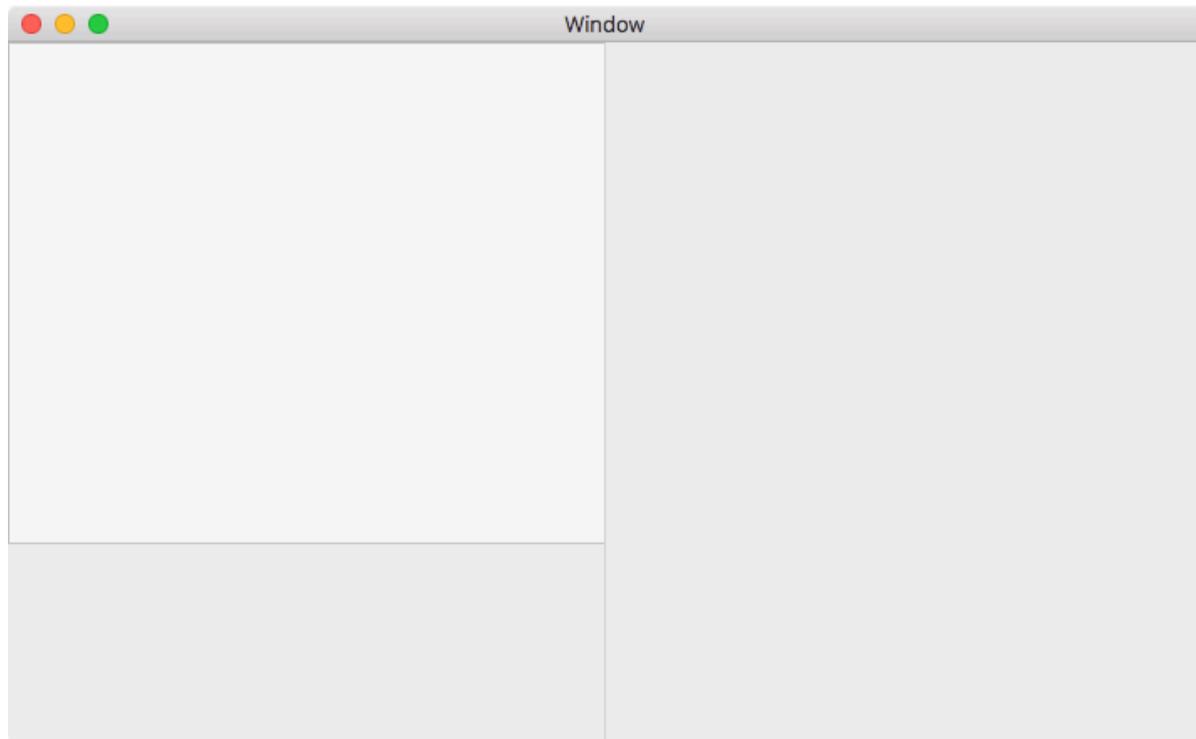
crash? Well, no. But Xcode also knows something important: when the view controller is first created the table view will indeed be nil, because it hasn't been loaded yet. But by the time **viewDidLoad()** is called – and indeed from then on – the property will never be empty, so it's safe to be accessed directly.

Now that we've created an outlet for the table view, we need to do the same for the image view in the detail view controller. If you click the image view and wait a moment, all being well the assistant editor should detect that your selection changed, and show `DetailViewController.swift` in the bottom pane. If it gets it wrong (it does happen!), use the manual approach instead.

Once you have `DetailViewController.swift` in the bottom pane, Ctrl-drag from the image view into the **DetailViewController** class, drawing a line to just above **viewDidLoad()**. Name it “imageView”, then click Connect.

That's all the connections complete, so try pressing Cmd+R and see what happens. Chances are you'll see something like the screenshot below – the table view occupies only part of the window, and if you make the window larger you'll see it doesn't resize along with the window.

Project 1: Storm Viewer



Worse, if you look in Xcode you should see some worrying error messages, such as “Illegal NSTableView data source”.

We’ll fix the error messages soon enough, but first we need to do the last piece of work in the storyboard: telling macOS how our app should behave when the window resizes.

Right now we’ve provided no instructions other than drawing things on the canvas in IB, so as far as Xcode is concerned we want those views to be sized exactly as they were placed. What we *really* want is for both the table view and image view to stick to the edges of their container, so that they resize intelligently along with the window.

To make that happen we’re going to use something called Auto Layout, which is a powerful, cross-platform toolkit for creating layout rules. You can tell IB how each view should be sized and placed when its situation changes, and Auto Layout will take care of the rest.

In this app, our layout is so simple Xcode can figure it out for us. Using the document outline,