Random Acts of Mac Development

By
Steve Barnett

Random Acts of Mac Development

By

Steve Barnett

Table of Contents

Figures

PREFACE

There are a great many books, videos and courses on the market that will teach you SWIFT and many more that will teach you iOS development. If you want to write an application for the Mac, however, the availability of information becomes somewhat rare, as does the quality of that information.

Dig around and you will find answers to most questions. Those answers will generally be in Objective-C and will almost always require re-work before you can do anything practical with them. What I want to achieve here is an accumulation of the bits and pieces of knowledge I have gained in pursuit of my ambition to write a Mac application using Xcode and Swift.

So, just exactly who am I?

Well, if you're hoping for someone with decades of Apple development, you're going to be disappointed. I've been writing code since the early 1980's so I guess that makes me an experienced programmer. My background has been in IBM mainframes and then on to PC development using a variety of languages.

When it comes to developing for the Apple world, I'm a newcomer. I've done a bunch of SWIFT courses and written a couple of iOS apps for my own amusement and education, but I have not written commercially. That doesn't make me an Apple expert but neither does it mean I have no ability to fathom out Apple development. A for loop is a for loop regardless of the language you use.

Why Read This Book?

So, why would you want to read this book? While I can't promise it'll be the best book you'll ever read on development or necessarily be the most accurate book on Mac development, I can offer you my take on how to do practical things on the Mac. I present an eclectic collection of development topics that cover the things that I needed to do when developing my applications. The things I present here are real-world and worked for me. That's always a good starting place.

What I can't promise is that I'll be getting in-depth with the patterns and practices of development. Honestly, while I accept that there are some great development methodologies out there, I confess to not always using them.

My plan here is to present software that I have used in my own programs. If it happens to match a recognised pattern, it'll be purely coincidence.

Chapter One

User Interaction

All programs interact with the user at one level or another. Whether it be menus, toolbars, input windows or messages. On the Mac, there are multiple mechanisms to make that interaction easier and more flexible. The problem with easy and flexible, as a new developer, is that it's sometimes hard to fathom out what class to use and how to work with it. It's only easy and flexible once you know how to do it.

This section is a random selection of user interaction mechanisms; the kind of thing Windows developers have been doing for years. As a Mac newbie, with a Windows background, I know what I want to do, just not how to do it. We'll cover a number of the day to day facilities that every Mac application is likely to need to use.

MESSAGE BOXES

NSAlert, the long way

iOS has a rather nice mechanism for displaying messages, It is, however, fairly complex to set-up and is async, so is built around closures and all sorts of messy stuff like that. When all you want to do is display a simple message, you need simpler code.

Luckily, OSX has that simple code for that simple message.

```
let msg = NSAlert()
msg.messageText = "Title Text"
msg.informativeText = "Informative Hello"
msg.runModal()
```

Can't get more simple than that and the results don't look too bad either.

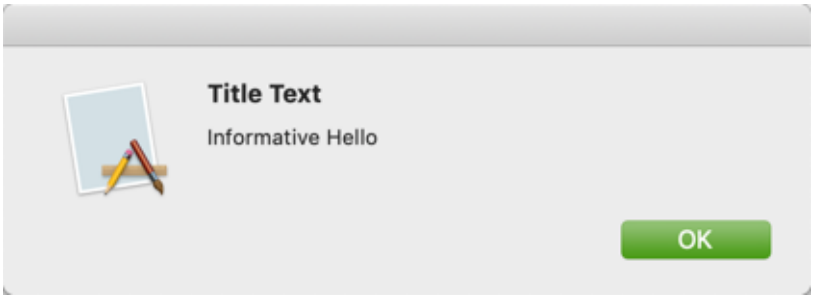


Figure 1: Standard message box.

Obviously, you can expand on this, maybe replacing the default icon or add some extra buttons. But for a quick result, this isn't too bad.

Changing the Image

So, lets change the image. Pretty straight forward, as it turns out.

```
let msg = NSAlert()
msg.messageText = "Title Text"
msg.informativeText = "Informative Hello"
msg.icon = NSImage(named: "MacOS")
msg.runModal()
```

I have an asset defined called MacOS which is a 48x48 px image. So I plug this into the NSAlert and the image gets rendered in the message.

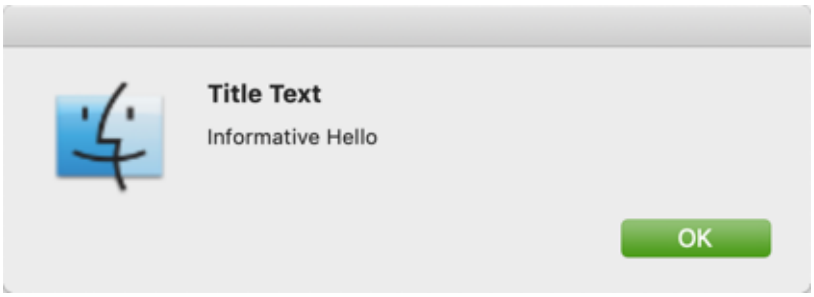


Figure 2: Message box with new image.

With buttons

Ok, that's nice, but what if you want to present your users with options. One button isn't going to be enough, so we expand our code to create a couple of buttons.

```
let msg = NSAlert()
msg.messageText = "Title Text"
msg.informativeText = "Informative Hello"
msg.icon = NSImage(named: "MacOS")
msg.addButton(withTitle: "Not Ok")
msg.addButton(withTitle: "Ok")
let result = msg.runModal()
```

The important changes here are that we have added two buttons and given them some text to be displayed. By adding these buttons, we also remove the default button.

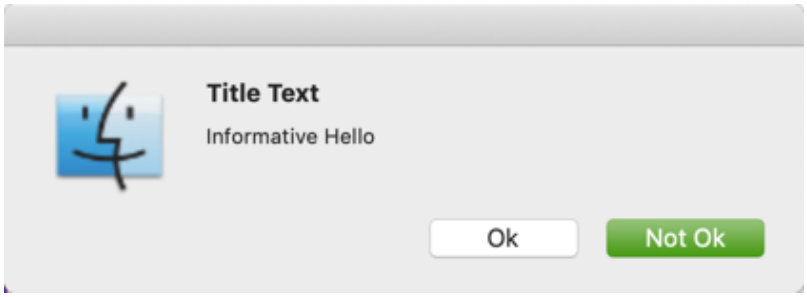


Figure 3: Message box with new image and buttons.

Of course, having the buttons is of little use if we can't do anything with them. So we also take the return value of the runModal call and assign it's value to the result variable. For buttons that we have added by title, the return value will be an integer starting at 1000 for the first button we added and incrementing by 1 for each subsequent button.

In this case, Not OK will return 1000 and OK will return 1001.

The first button added will become the default button.

NSAlert wrapped up

Using NSAlert to popup messages isn't a big deal. Well, for the first half dozen messages. After that it gets to be a bit repetitive and weldon't like repetitive code, do we (the answer is *No*, by the way). So it makes sense to wrap our NSAlert code into a helper class, which is what I present here.

So, where do we start? Lets start by defining the parts that go to make up a message box on the screen and which parts are optional.

Parts of a message

If we take a message that we displayed in the previous section, we can start to see the anatomy of a message:

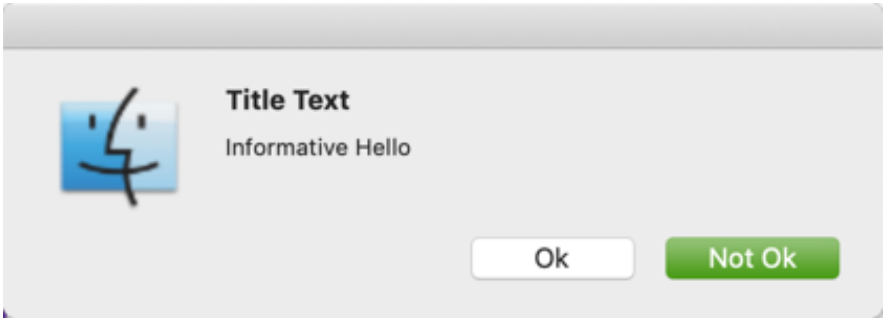


Figure 4: The components of a message box.

So, we have

- An icon on the left.
- Some buttons.
- Some title text.
- An information message.

There is probably a lot more that we can add to our message, but this configuration will match 99% of the messages we want to display.

Defining the icon.

Lets start with the icon. I know that NSAlert allows you to specify a custom icon and that's very nice, but we're trying to create a generic class here that can satisfy most of our requirements. If we need to support any old icon, we can add that some other time. Let's start by defining the icons we want to support… I make no bones about it, this is a hangover from my Windows days!

```
public enum MessageBoxIcon {
    case Stop
    case Information
    case Warning
    case Question
    case None
    case Default
}
```

The Mac has a default image for a message which we're supporting with the default case. I sincerely hope you will never use that default icon. It looks a tad unprofessional.

Defining the buttons

Like the icon, we want to restrict the buttons we want to be able to display. This makes our class more consistent and forces us to limit our *creativity* in favour of consistency. So, lets define the buttons we want to support:

```
public enum MessageBoxButton: String {
    case OK = "Ok"
    case Cancel = "Cancel"
    case Yes = "Yes"
    case No = "No"
}
```

Obviously, this is a pretty restrictive list of buttons but, as with the icons, it's going to provide 99% of the buttons we need, so lets code for the majority case and not the edge case.

We're going to want to allow multiple buttons so we have a choice to make; work out every combination of button (no, that would be crazy) or pass an array of buttons to our message method so we can add each button separately and so we can control the order that the buttons are added. Think I'll go with that solution.

Defining the text

The title text and the message content will be defined in the method we call and is free-format text, so needs no special consideration.

Which neatly brings us to the method that displays our message box.

```
public static func show(title: String, message: String,
                        icon: MessageBoxIcon = .Default,
                        buttons: [MessageBoxButton] = [.OK])
            -> MessageBoxButton  {

    let msg = NSAlert()
    msg.messageText = title
    msg.informativeText = message

    add(icon, toMessage: msg)
    add(buttons, toMessage: msg)

    let result = msg.runModal().rawValue - 1000
    return buttons[result]
}
```

That's a fairly simple method considering what it does. Let's go through it top to bottom. The method signature takes four parameters;

- The “title” of the message.
- The “message” itself.

These are required parameters, which makes sense as you can’t really have defaults for them. The next two paramagnets are option and have default values;

- The “icon” defines the type of the message box icon to display. This corresponds to the enumeration we defined for the available image types. It will default to .Default, which is the default icon from OSx. It’s probably the least useful default value too!
- The “buttons” parameter is an array of buttons you want to appear on the popup window. We default this to the OK button, which is probably the most useful default.

  When we want alternative buttons, we pass in an array of MessageBoxButton values from which we can construct the buttons.

Our return value is the button that was pressed.

Next we define the NSAlert instance that we are going to customise and inject the static text we want to be displayed. That’s all the basic set-up necessary to get a message out. However, we need to deal with icons and buttons too. That’s the purpose of the next two add calls. I’ll deal with those in a moment.

The final two lines are responsible for displaying the alert and returning the button pressed. Yes, I know, there is a magic number there, but I’ve yet to find a constant that defines with it. From experimentation I have found that custom buttons, which is what were adding, have IDs starting at 1000 and incrementing in value based on the order they are added. That works nicely for us as it gives us a nice simple index to use to work out which button was pressed and to return it to the caller.

Adding Buttons

Now, to deal with the lines I glossed over. The first call to add is intended to add the buttons.

The call is simple;

```
add(buttons, toMessage: msg)
```

The call is to the class level method called add, passing it the array of buttons that we were passed and the part filled out NSAlert message to add them to.

```
private static func add(_ buttons: [MessageBoxButton], toMessage msg: NSAlert) {
    for button in buttons {
        msg.addButton(withTitle: button.rawValue)
    }
}
```

Nice simple loop. Loop round the array of buttons we were passed in, creating a new button. The caption for the button is extracted from the enumeration. Simple.

Adding The Icon

Adding the icon is a tad more complex. The call is the same simple call to add that we used for buttons, but the implementation is more difficult

```
add(icon, toMessage: msg)
```

We will have been sent a single icon definition which we need to translate into a picture. That's easily achieved in a method with a simple switch.

```
private static func add(_ icon: MessageBoxIcon, toMessage msg: NSAlert) {

    let imageBundle = Bundle.init(for: MessageBox.self)

    switch icon {
    case .Information:
        msg.icon = imageBundle.image(forResource: "info.png")
        break
    case .Stop:
        msg.icon = imageBundle.image(forResource: "stop.png")
        break
    case .Warning:
        msg.icon = imageBundle.image(forResource: "warning.png")
        break
    case .Question:
        msg.icon = imageBundle.image(forResource: "question.png")
        break
    case .None:
        msg.icon = NSImage()
        break
    default:
        // Do nothing - displays the default icon
        break
    }
}
```

First thing to note is that we're using custom images that are embedded into the framework or project. I've created a folder and dragged in 48px x 48px images for my purposes;

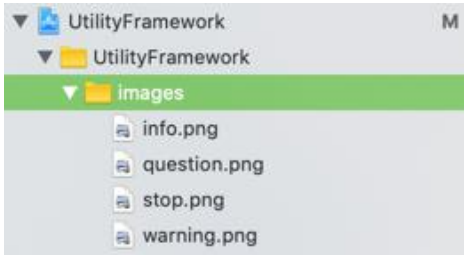


Figure 5: Message box images.

You will note there are four images and six icon definitions in the enum. That's fine, one of the enum's is for no image and one for the default, built-in, image, so we only need the four pictures for the remaining four icons.

First thing we need is a reference to the project bundle. Simplest way to get that is to get the bundle for the message box class:

```
let imageBundle = Bundle.init(for: MessageBox.self)
```

Now I have access to the bundle, I can retrieve the image depending on the icon selected.

It's probably not the most elegant way to do it, but it's very simple and effective.

FILE PROMPTS

This isn't a comprehensive analysis of file prompts. The subject is too big and the permutations of the options too large for a simple overview. The point of this section is to get you to a start point where you can ask the user for a file URL to open or to save to and to ask the user for a folder location. If you need more functionality, then you will have a sound starting point to work from.

File Open Prompt

Prompting the user for the name of a file to open is a relatively painless process. It does, however, involve creating a control on the fly and setting a lot of options. At the end of the process, we will either end up with the URL of the file to open or nil if the user cancels.

First thing to realise is that COCOA provides everything we need in the form of the NSOpenPanel. All we have to do is create it, display it and handle the results when the user presses a button.

To create the panel, you just need to instantiate the NSOpenPanel and set the options. To do this, I have a utility method.

```
private func createOpenPanel(ofType: [String], withTitle: String?) -> NSOpenPanel {

    let openPrompt = NSOpenPanel()

    if let titlePrompt = withTitle {
        openPrompt.message = titlePrompt
    }

    openPrompt.allowsMultipleSelection = false
    openPrompt.canChooseDirectories = false
    openPrompt.canChooseFiles = true
    openPrompt.resolvesAliases = true
    openPrompt.allowedFileTypes = ofType

    return openPrompt
}
```

In this case, I pass in a string array of the types of files I want to open. This is simply an array of file extensions. I also pass in a title for the window.

The title is irritating. There is a title property and, if you set it, nothing appears. There seems to be a split of opinions on whether this is right or not but many people on the interweb have

talked about raising a bug with Apple, but I doubt anything will happen. So, we use the work around that will work, and set the message property instead.

The resulting window has a title, of sorts, and lets you select a single file of a specific file type.

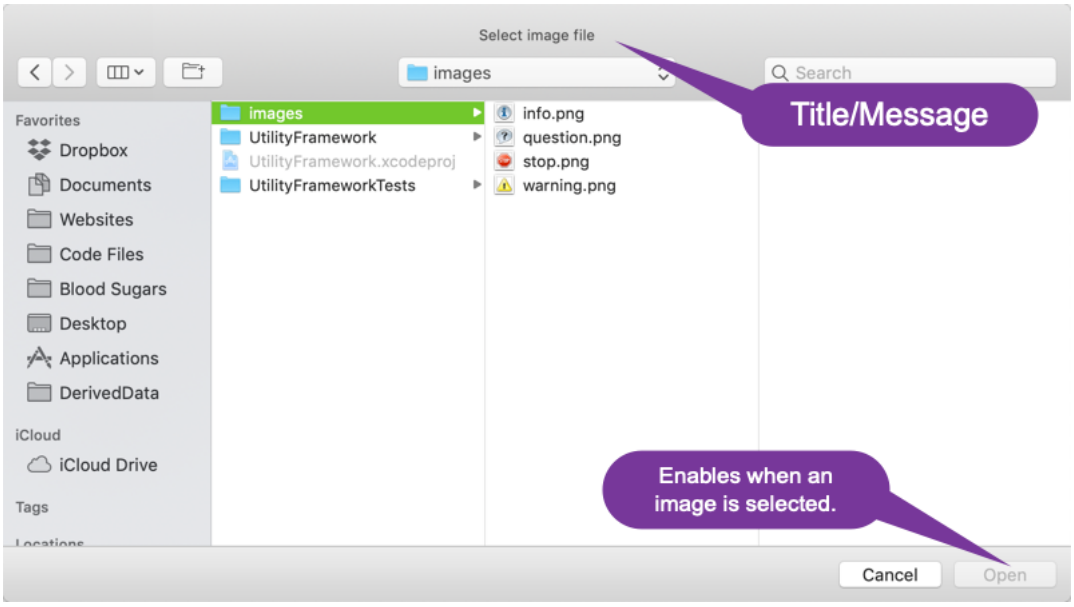


Figure 6: File Open selection window.

Now, creating the panel is only half the job. We need a utility method to create and display it. When I want to prompt for a file, the above method does the heavy lifting of creating the panel and the caller does the heavy lifting of handing the display and handling of the results.

```
public static func selectSingleInputFile(ofType fileTypes: [String], withTitle
windowTitle: String?) -> URL? {

    let openPrompt = FileHelpers().createOpenPanel(ofType: fileTypes, withTitle:
windowTitle)

    let result = openPrompt.runModal()

    if result == NSApplication.ModalResponse.OK {
        let fName = openPrompt.urls

        guard fName.count == 1 else { return nil }
        return fName[0].absoluteURL
    }

    return nil
}
```

So, having called the method to create the panel, we display it by calling the runModal() method. This will return Ok or Cancel depending on what the user decided to do. It can't respond with anything else, so that's an easy test.

Assuming the user pressed Ok to select a file, the url's property will return a list of selected file URL instances for you to work with. Even though our code to create NSOpenPanel

specifically stated that you could not select multiple files, you will get an array back, so we need to pick off the first entry in the list.

If they pressed Cancel, then we return nil.

Save file

Prompting the user for a file name to save to is very similar, just different enough to make it worth while doing a quick run through. This isn't going to actually save the file, just ask the user for the name of the file to save to. We're going to end up with a URL for a file of nil of the user cancels.

So, NSOpenPanel provides us with a file open window and NSSavePanel does the equivalent for saving files. It has way less options, so creating ns NSSavePanel is somewhat easier:

```
private func createSavePanel(ofType: [String], withTitle: String?) -> NSSavePanel {

    let openPrompt = NSSavePanel()

    if let titlePrompt = withTitle {
        openPrompt.message = titlePrompt
    }

    openPrompt.allowsOtherFileTypes = true
    openPrompt.canCreateDirectories = true
    openPrompt.prompt = "Save As..."
    openPrompt.allowedFileTypes = ofType
    openPrompt.nameFieldLabel = "Enter file name:"
    openPrompt.nameFieldStringValue = "file"

    return openPrompt
}
```

Ok, when I say way less options, I really meant different options. To be fair, you won't normally set all of these but I wanted to make sure the options were highlighted.

- allowsOtherFileTypes allows us to save a file with a diffrent file type to the ones we specified in the allowedFileTypes property. This would normally be left to default to false.
- prompt sets the text on the save button. It's sometimes useful to be able to change the caption to something more specific to the application.
- allowedFileTypes is as per the open panel. It lists the types of files you want the user to be able to create. The first item in this list becomes the default file extension in the save window.
- nameFieldLabel is the text that appears before the text box where you enter the file name. I would not normally expect to change this.
- nameFieldStringValue is the default file name that the user is expected to overtype. If you don't specify a name, then a default will be generated for you with a date and time in it.

As with NSOpenPanel, we need something to drive this code and that, as with NSOpenPanel is simple code:

```
public static func selectSaveFile(ofType fileTypes: [String], withTitle windowTitle:
String?) -> URL? {

    let openPrompt = FileHelpers().createSavePanel(ofType: fileTypes, withTitle:
windowTitle)

    let result = openPrompt.runModal()

    if result == NSApplication.ModalResponse.OK {
        let fName = openPrompt.url
        return fName
    }

    return nil
}
```

The main difference here is that we only get one file url returned.

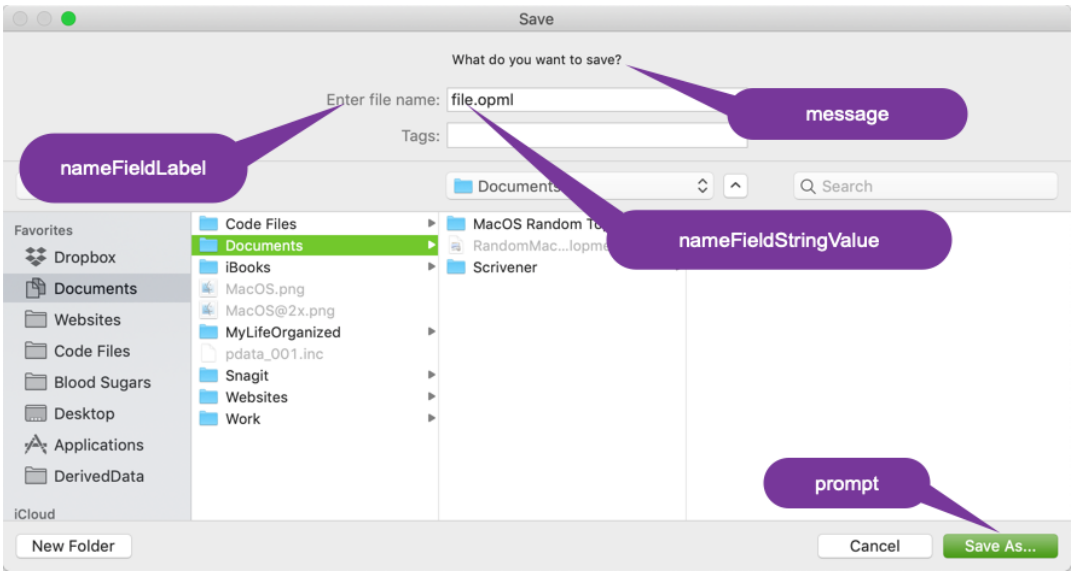


Figure 7: File Save selection window.

There are other things you can do with this panel and other options that you can set. However, this set of options gets you started and gets you a selection panel you can use with the minimum of fuss.

Sandboxing

Then life gets complicated and we run our code in XCode and get some totally obscure error reported in the console:

```
2019-03-17 10:40:38.713734+0000 MacOutliner[2031:55071] -[NSVBSavePanel init] caught
non-fatal NSInternalInconsistencyException 'bridge absent' with backtrace (
```

I'll save you the long trace that goes with it. As with so many messages you get during development, the cause is blindingly obvious from the error message...

Turns out, Google is your best chance of fixing this and you have to turn to sandboxing for the cause. In its default state, you can't write files, so it makes sense to not allow you to display a file save popup, so it fails on you. It's good about it, you don't get a horrible message thrown at you, just a subtle error in the console and an app that doesn't work.

The fix is straight forward, when you know where to go:

Figure 8: File Save Sandboxing Options.

Go to the target, select Capabilities and take a look at the App Sandbox. In there you will find a set of File Access settings. I change the User Selected File to read/write and the code starts working. Simple.

Of course, to change these settings you need to be in an app and not a framework project. If you're in a framework project, the save dialog will appear, but you won't be able to put a file name in. Be prepared for lost of frustrated beeping when you try to type something!

I also want to say that I have had some weirdness in this area. When changing this setting, I sometimes get an error with the build. If it talks about problems with the entitlements file turn the sandbox off and on again and the build problem goes away.

Selecting Folders

Sometimes we don't want a specific file, but want a folder. We may want to process all files in that folder or to locate a folder to wite several files to. Either way, a file open or file save prompt is of little use to us.

However, we can re-use the NSOpenPanel code we wrote earlier. Selecting a folder is very similar to selecting a file, so lets create a folder selection method"

```
public static func selectFolder() -> URL? {

    let openPrompt = FileHelpers().createOpenPanel(ofType: [""], withTitle: nil)

    openPrompt.canChooseDirectories = true
    openPrompt.canChooseFiles = false
    let result = openPrompt.runModal()

    if result == NSApplication.ModalResponse.OK {
        let fName = openPrompt.urls

        guard fName.count == 1 else { return nil }
        return fName[0].absoluteURL
    }

    return nil
}
```

We re-use the code to create the open panel. We don't need file extensions in this case as they are irrelevant. I'm not bothering with a title either.

The trick here is to enable the users ability to check directories with canChooseDirectories and disable their ability to pick files with canChooseFiles. This turns our file selection window into a folder selection window.

As with the file open panel, we will be returned an array of folders, so we just have to pick the first one out of the response.

TOOLBARS

Creating Toolbars

Toolbars, the staple of any application beyond the most trivial. On the Mac, unnecessarily complicated in my opinion, but doable once you get into a way of working. Lots of bits and pieces to deal with and some not entirely intuitive things to deal with.

So, where do we start. Well, the toolbar is tethered to the main Window, so we start by adding a toolbar control to the window. This gives us a toolbar with some pre-defined icons on it.

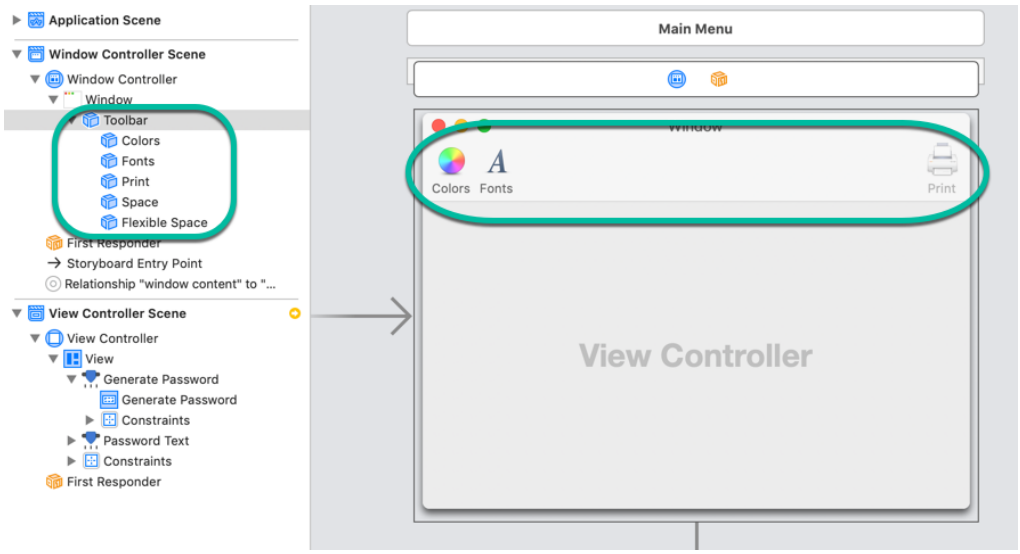


Figure 9: Window with default toolbar

If you click on one of the toolbar items, the toolbar editor window will pop-up. This will display the items that you have defined and that can be placed on the toolbar. The available items are at the top and the added items are on the representation of the toolbar below.

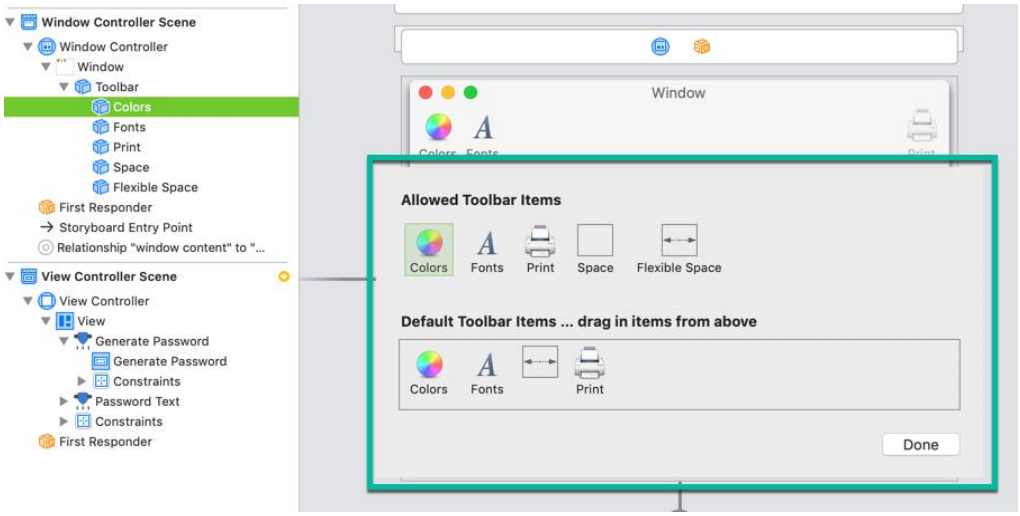


Figure 10: Window with default toolbar.

First job is usually to delete the items on the default toolbar. All except the space and the flexible space as these can be useful. Once we have a clean toolbar, we can start adding our own icons.

Open the object library and search for a Textured Rounded Button and drag it on to the toolbar in the outline. It's easier to drop onto the outline. What you will end up with is a button available for the toolbar. It's going to have a rubbish name and title, but we can start customising that. Most important to understand at this point is that, while the button exists in the outline, it is NOT on the toolbar!

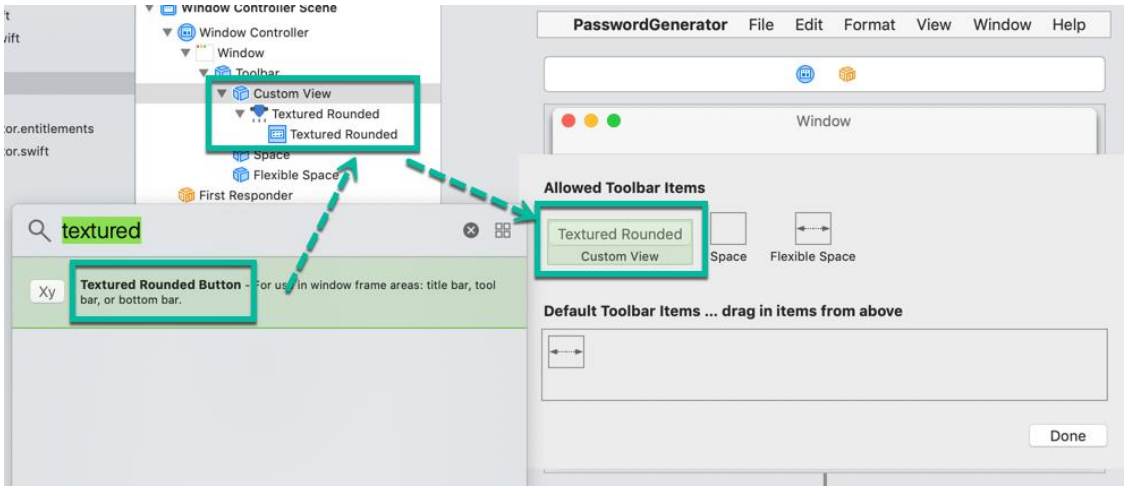


Figure 11: Window with default toolbar

Customising our button

So we have this `Textured Rounded Button` thing. Where do we start customising it?

I start by clicking on the 'Custom View' and renaming it to something that is more meaningful. In this example, it's been renamed to 'share' for no good reason at all. Then, it needs to be customised to something useful for the toolbar. So, click on the toolbar item and pop-over to the properties inspector and change the icon it will display and the caption.

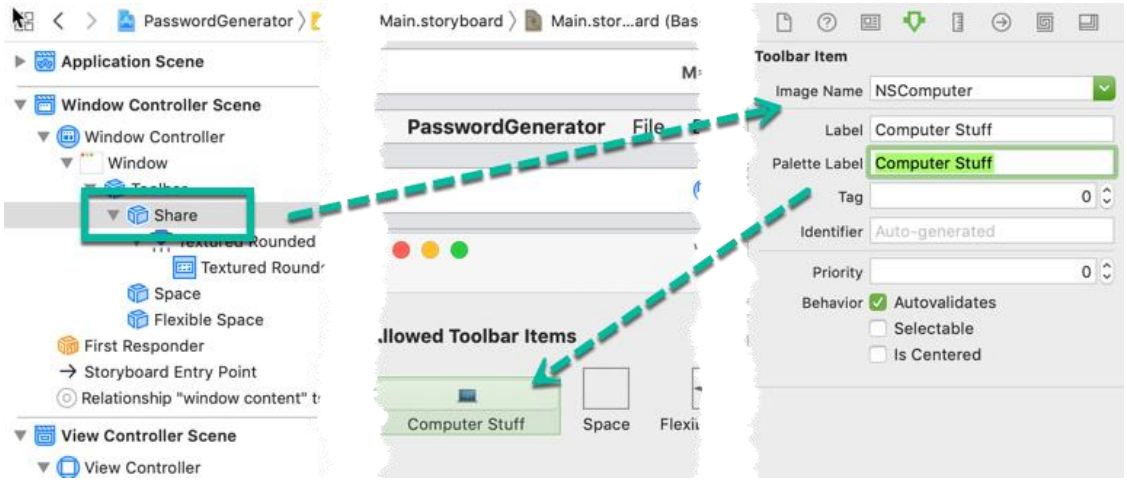


Figure 12: Renamed custom view

Depending on the options we select, the icon, the text or a combination of the icon and text will be displayed. That gets set at the `toolbar` level.

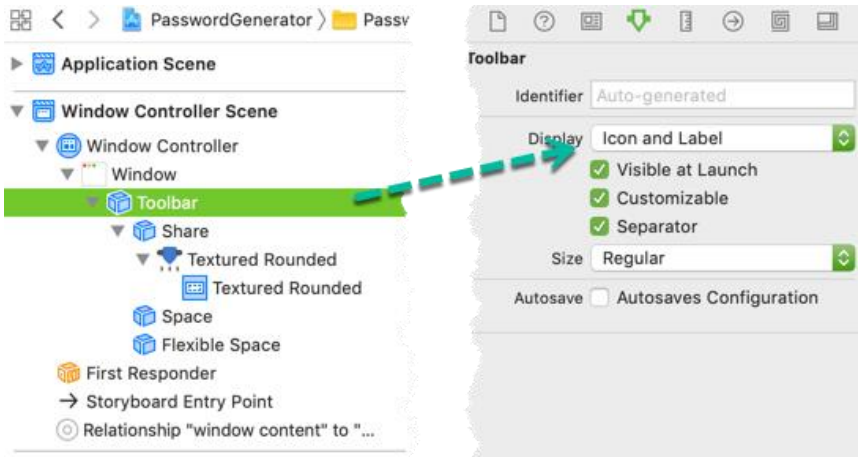


Figure 13: Setting the button display type

Final customisations to remove the title from the textured rounded button itself. If we don't do this, the title will be shown at runtime and that just looks plain ugly. So click on the button and delete the title text.

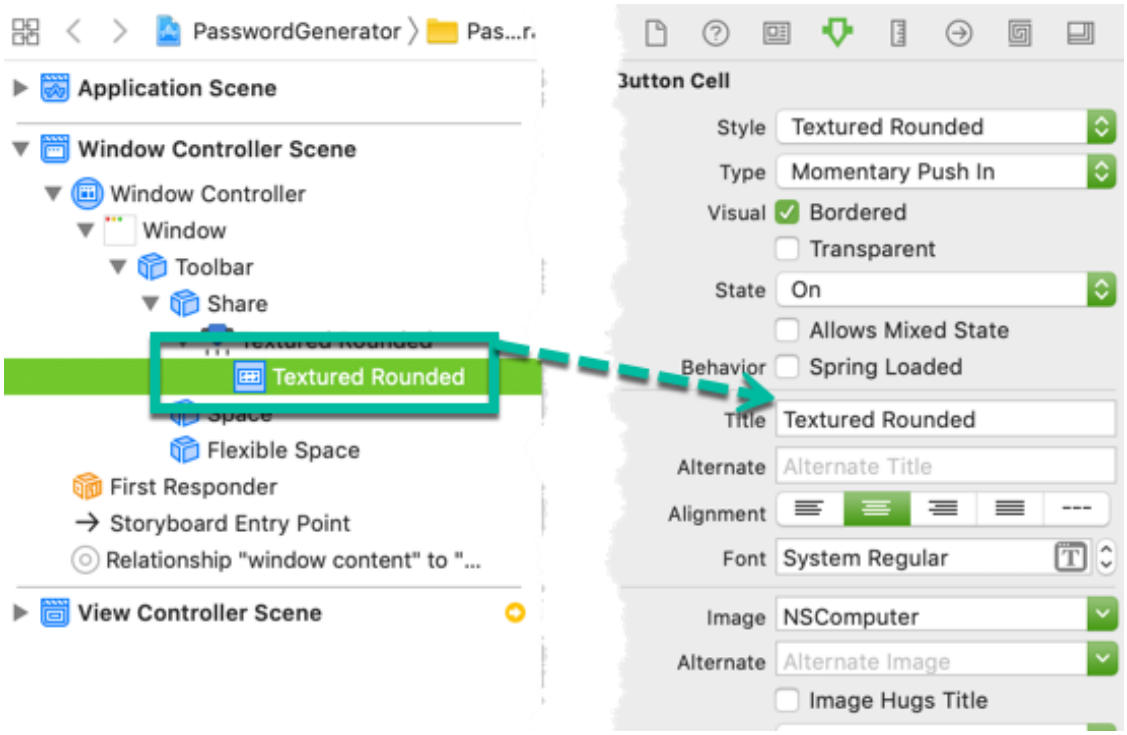


Figure 14: Clear the button title text

Adding to the toolbar

As I said earlier, we have a toolbar item, but it's not on the toolbar. To do that, just drag from the available items onto the toolbar. Don't expect to be able to drop it in the right place. Ain't gonna happen! Drop it on the toolbar and then drag it into the place you want it. It'll appear on the toolbar editor and on the toolbar in the window.

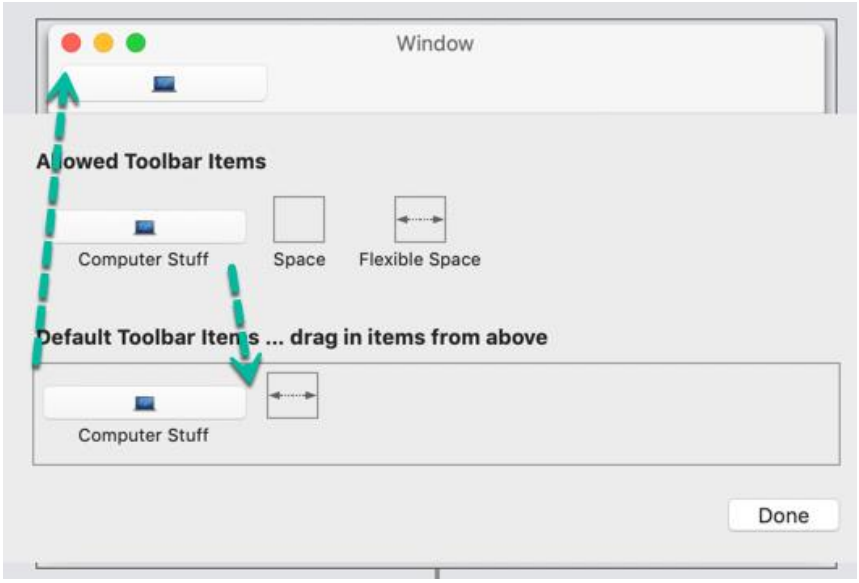


Figure 15: Drag the icon onto the toolbar

We can now run our app and see the results.

Figure 16: Viewing our toolbar

Job done? Not quite, our button doesn't do anything yet. This is where things get a bit unintuitive.

Connecting the icon

The toolbar lives within the window, but we want to process it in the view we created. All sorts of logical ways of doing this spring to mind, but it's not that simple. So, where do we start.

Firstly, we head off to the view where we want to handle the button. We're going to create an `@IBAction` in there to handle the click of the toolbar icon.

```
@IBAction func homePressed(_ sender: Any?) {
    print("In the action")
}
```

To connect the action, we have to tell the button the name of the ruction to call when it's pressed. That, we do in the structure browser by <control>Clicking and dragging to the `First Responder` item in the window.

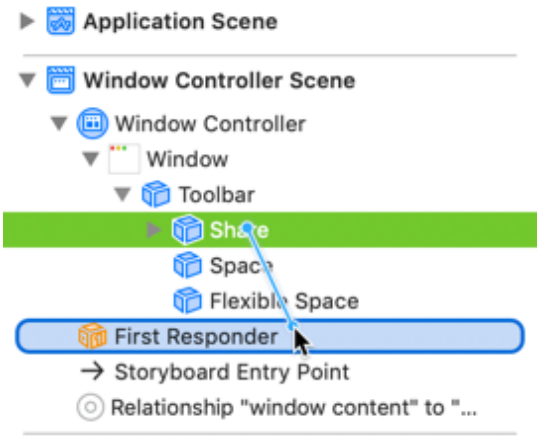


Figure 17: Setting the icon handler

When you do this, Xcode will search for any method that could service the request and will pop-up a list of all of them. We scroll to the `homePressed` function we defined and select it.

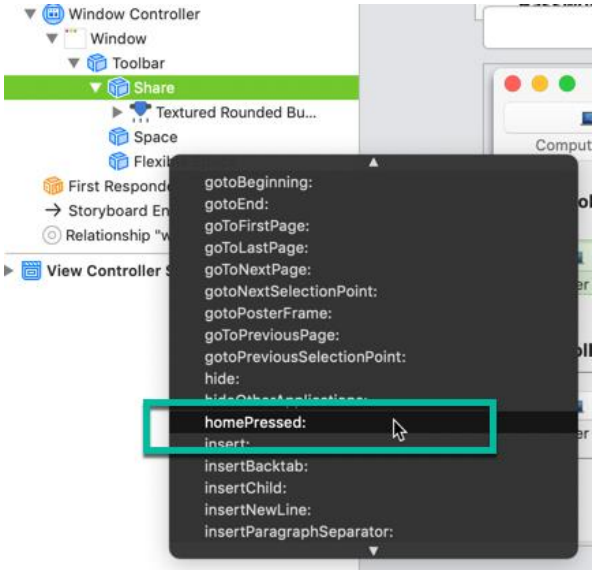


Figure 18: Selecting the handler method

That's it. When the toolbar icon is clicked, the `homePressed` function will be called.

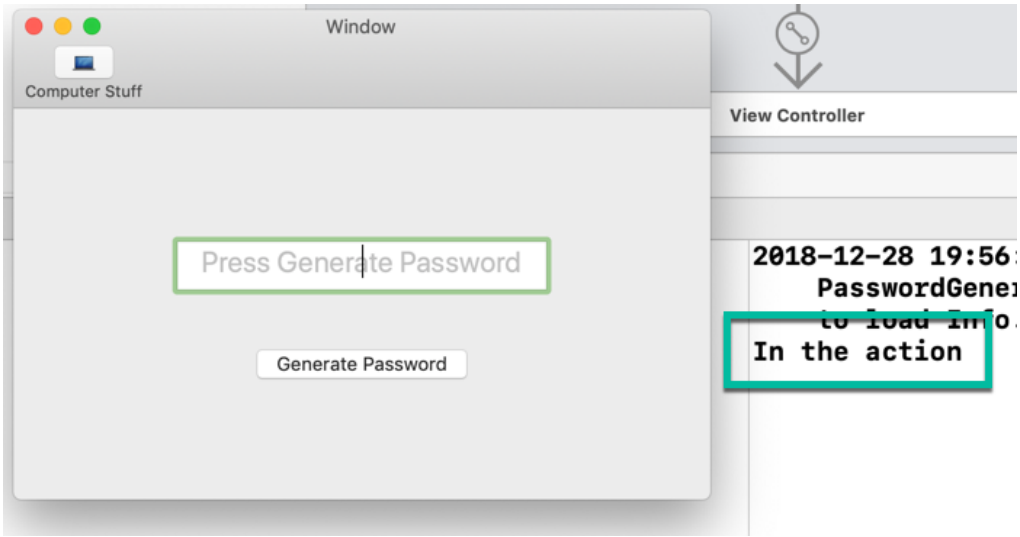


Figure 19: The end result

Where next

There are many potential issues with what we have done so far. The first is that the size of the icon is a bit variable if we don't display the text. To fix that, set the width and the maximum width of the button to, say, 47.

MENUS

Overview

When we create our application, Xcode will create a menu bar for us automatically and populate with a lot of entries. The nice side of this is that Xcode will also handle a number of these menu items for us so we don't need to write code. The down-side, is that there are a number of them that we must handle and the method of connecting them to code is not the easiest to deal with.

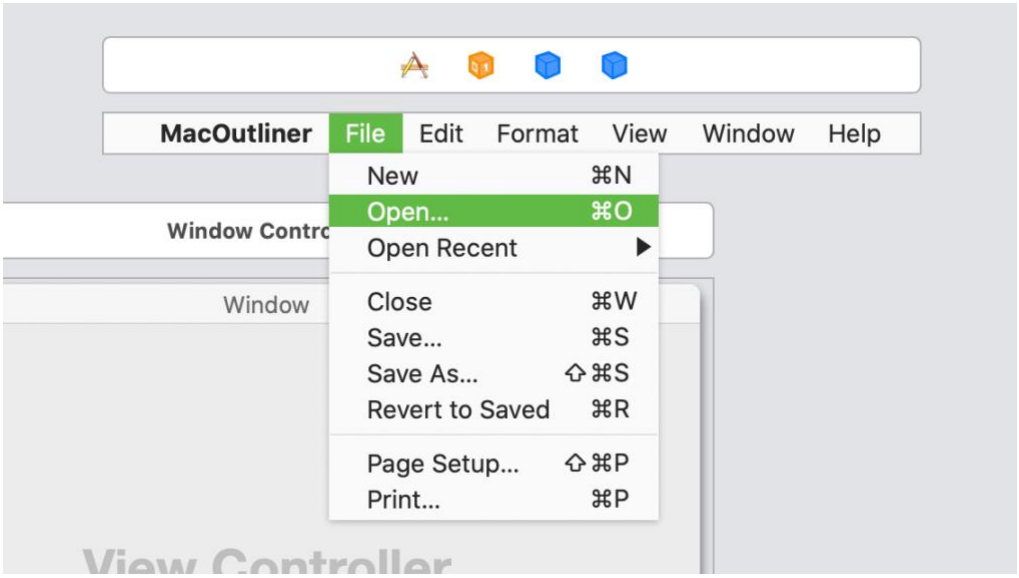


Figure 20: Default Menu for New Project.

There are multiple ways to connect your menu items to code. You can connect via the storyboard or you can connect in code. It really depends on your particular needs. Both will be explored below.

Connecting via Storyboard

When your needs are fairly straight forward, connecting via the storyboard makes sense. The simplest way is to find the menu item in the explorer and <control><click> then drag to the *FirstResponder* item.

However, before you connect a menu item, the method that you want to connect it to must pre-exist in your project. Unlike connecting views to view controllers, Xcode will not create the method stub for you. So, start the process by creating your menu handler:

```
@IBAction func openMenuClicked(_ sender: Any) {
    // Code to open a file
}
```

Once you have the method defined, find the menu item in the explorer, <control><click> then drag to the *FirstResponder* item.

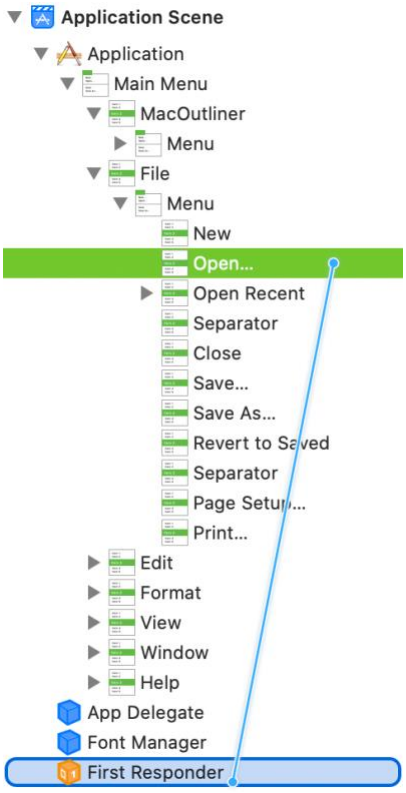


Figure 21: Connecting Menus.

When you release the mouse, Xcode will search your project for compatible method signatures that could satisfy a menu item click:

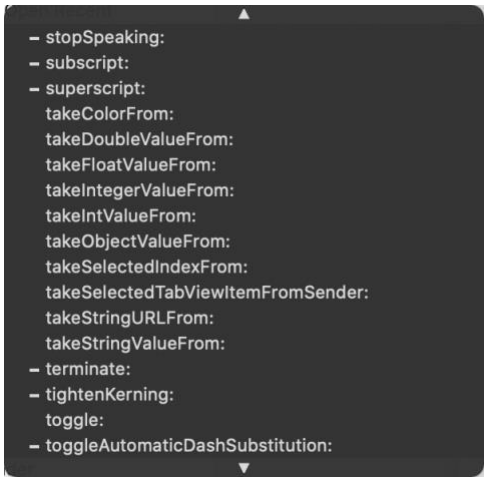


Figure 22: Connecting Menus.

You will need to search through the list until you find the method you created to handle the menu item and select it.

And that's it. When the menu is selected, your nominated method will be called. Where you put the method is entirely up to you; it does not have to be in the AppDelegate or a ViewController, just so long as the class exists at run time.

Connect Via Code

Connecting via the storyboard will probably be fine for most circumstances. However, there are times when it may be more convenient to connect in code. By way of example, our application has a File menu and the File menu has a number of sub-menus. We can connect all of these, one by one, to methods in our view controller or, as I describe here, we can create a file management class that handles all of the File menu options.

Menu Helpers

Given that we are likely to connect Manu menus in our code, a good place to start is with a helper class for dealing with the repetitive tasks. Key among those is finding the menu item you want to connect to or that you want to change the state of. So, let's start with a simple class called MenuHelpers.

```
import AppKit

class MenuHelpers {

}
```

The first method we're going to need is one to locate the top level menu. The structure of menu items is that there are top level menus (such as File, Edit etc) and each of these menus has a submenu containing more menu items. So, if I want the File->Open menu, I have to locate the File menu first:

```
static func getTopLevelMenu(withTitle title: String) -> NSMenuItem {

    guard let mainMenu = (NSApplication.shared).mainMenu else {
        fatalError("Failed to get access to the application main menu")
    }

    guard let topLevelMenuItem = mainMenu.item(withTitle: title) else {
        fatalError("Failed to get access to menu item \(title)")
    }

    return topLevelMenuItem
}
```

So, the menu for the application is attached to the application object. Our first guard statement gets us a reference to the mainMenu via the application mainMenu property. If we fail to get the menu, we kill the program, as it implies a critical issue with the code.

Once we have the mainMenu, it will have menu items representing the items at the top level. We want the File menu, so we search the mainMenu for a menu item with a title of "File".

Searching by title is only one of the options available to us. The framework also includes an option to search by the tag associated with the menu, or the item at a specific index. Probably the safest way to search is via tag as you get to set that in the storyboard and will not change. While the title of the menu should not change, it's possible that it will at run time (perhaps you change the menu for a different language).

However, for our purposes, we'll stick with the title. If you do go down the tag rabbit hole, make sure you create an enum to assign meaning to the tag values you give menu items; it will save you a lot of grief later on.

In our code, we're going to want to connect to a menu item that is in a sub-menu of the top level menu so we're only half way there so far. What we need is a way to get to the sub-menu. We achieve that with a second helper method:

```
static func getMenuItem(withTitle title: String,
                inTopLevelMenu topMenu: String) -> NSMenuItem {

    let topLevelMenu = MenuHelpers.getTopLevelMenu(withTitle: topMenu)

    guard let menuItem = topLevelMenu.submenu?.item(withTitle: title) else {
        fatalError("Failed to get access to menu item \(title) in top
                level menu \(topMenu)")
    }

    return menuItem
}
```

First task is to call the *getTopLevelMenu* method to retrieve the top level menu item that contains the menu item we're looking for. This will either work, or crash out with a fatal error. We can then retrieve the item we want from the *submenu* property. Again, for the purposes of demonstration, we're using the title of the menu item, but we could search by tag or take the item at a specific index.

This code also makes the assumption that, because we are doing this all programmatically in our code, we know that the menu item will exist. There should never be a circumstance

where it does not exist. That may not always be the case, of course, so you may want to beef up the error handling.

Adding A Handler

Now we have the means to locate a menu item, we want to attach a handler to it so we can respond when the item is selected.

This code can appear pretty much anywhere in your application. For the purposes of illustration, let's assume we're dealing with the File menu. There are several items on the File menu and I'll want to handle them all in one place, so I start out with a *FileManagement* class.

My class is going to need to initialize several menu items, so it makes sense to have a helper method that does the heavy lifting:

```
private func handleMenu(withTitle title: String, atLevel level: String, withHandler
action: Selector) {

    let identifier = (level + title)
                    .replacingOccurrences(of: "\u{2026}", with: "")
                    .replacingOccurrences(of: " ", with: "")

    let menuItem = MenuHelpers.getMenuItem(withTitle: title, inTopLevelMenu: level)
    menuItem.target = self
    menuItem.action = action
    menuItem.isEnabled = true
    menuItem.identifier = NSUserInterfaceItemIdentifier(rawValue: identifier)
    }
}
```

We need to know the title of the menu item we want to handle, the title of the parent menu it exists within and the method to call when the menu it selected. For the purposes of our code, I assume that the menu item should be enabled.

Every menu item needs a unique identifier. Since we have the top level menu title and the item title, we can generate the identifier ourselves. It would be very confusing to the user to have two menu items with the same title in the same parent menu, so we can assume that the identifier we come up with is unique. This will hold up even if we use tags for our search instead of titles.

Next we go fetch the menu item using our menu helper class. As shown previously, this will either return us the menu item or will crash the program.

We then set four properties:

- **target** defines the class that will handle the menu. You must override this as menu items will often be set to *FirstResponder* and, if you do not override it, your code will not be called.
- **action** defines the method to be called when the menu item is clicked on. This is the selector that is passed to our helper method.
- **isEnabled** is set to True to enable the menu item.
- **identifier** sets the menu item unique identifier.

This is the minimum necessary to connect to our menu. The remaining tasks are to code a method to handle the menu item and a method to make the connection.

Handling A Menu Item

The handler for a menu item is very simple.

```
@objc func fileNewMenuItem(_ sender: NSMenuItem) {
    // Handle the menu item
}
```

The method needs to be decorated @objc. Sad fact of life, is that objective-C is still with us and will be for a long time. The menu handler method is going to be called from Objective-C code in the operating system, so we need the decoration.

The menu handler will be passed a reference to the menu item that triggered it. This is the only parameter, do with it what you will.

Then we have the body of the code. That is obviously application specific.

Connecting the Menu Item

Making the connection between the menu item and the handler method is then a simple one liner:

```
handleMenu(withTitle: "New", atLevel: "File", withHandler:
#selector(fileNewMenuItem(_:)))
```

Given that we may have several of these, I recommend creating a configuration method:

```
private func configureMenus() {

    handleMenu(withTitle: "New", atLevel: "File", withHandler:
#selector(fileNewMenuItem(_:)))
    handleMenu(withTitle: "Open\u{2026}", atLevel: "File", withHandler:
#selector(fileOpenMenuItem(_:)))
    handleMenu(withTitle: "Save\u{2026}", atLevel: "File", withHandler:
#selector(fileSaveMenuItem(_:)))
    handleMenu(withTitle: "Save As\u{2026}", atLevel: "File", withHandler:
#selector(fileSaveAsMenuItem(_:)))
    handleMenu(withTitle: "Close", atLevel: "File", withHandler:
#selector(fileCloseMenuItem( :)))
}
```

What's With The \u{2026} ?

If you take a look at the File menu, you'll see a number of items with an ellipsis after them:

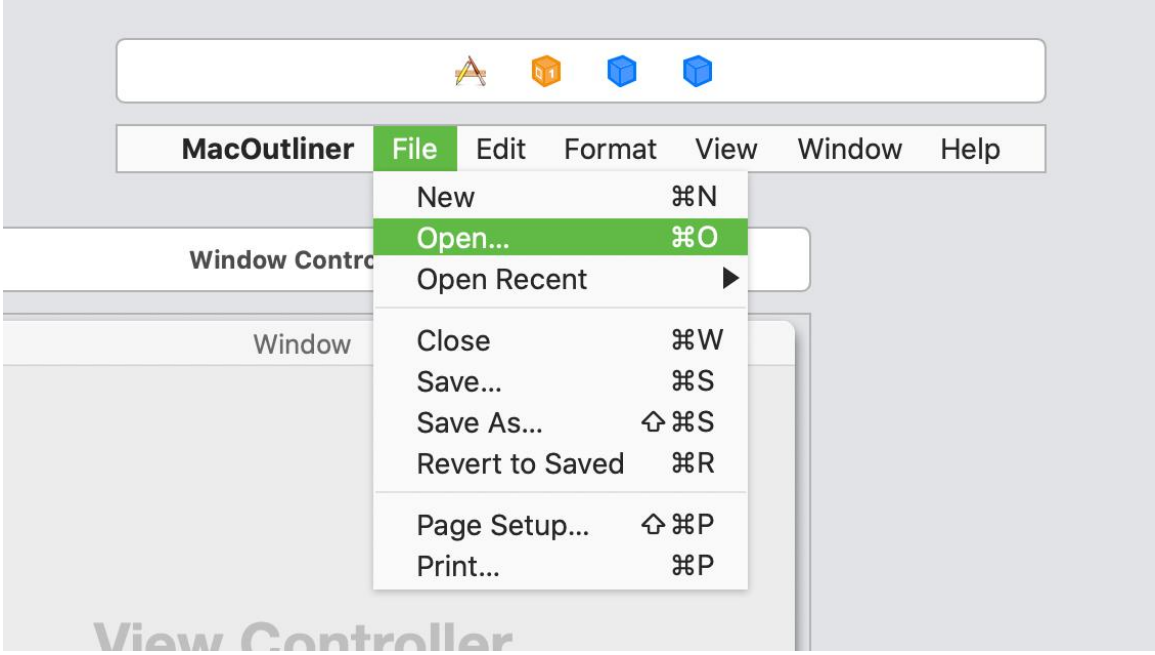


Figure 23 Menus With Ellipsis

It's tempting to assume that this is just three dots… Well, that's just not the case. The menu includes a specific character for the ellipsis and that is the unicode 2026 character. When we search for a menu item by title, we have to be careful to search for the ellipsis character and not three dots.

Similarly, when creating the identifier, I refer to remove the ellipsis from the identifier name. Not strictly necessary, but it makes the identifier easier to handle without a lump of Unicode in the middle.

Refactor for Safety

Having just gone through setting up the menu in code, we hit a couple of issues that we really should be aware of and that we absolutely must handle.

- Our menu contains that irritating Unicode sequence. Get that wrong and your code is going to crash.
- First customization is going to work, but what happens when the title of a menu item changes – your code is going to crash.
- Someone comes along and translates your menu, so your title searches are going to fail and your code is going to crash.

The list goes on. So, we need a more reliable way to deal with the menu that is not dependant on the title of the menu. And that is the menu item tag.

Menu Item Tags

When you click on a menu item in StoryBoard, the properties appear on the right:

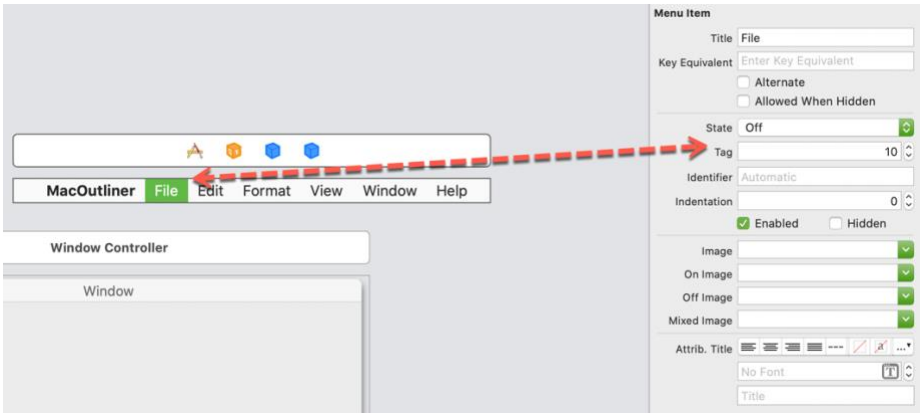


Figure 24 Menus With Tags

All menu items will have a tag and that tag is just a number. The number will not change as our application changes, so we deal with the issue of the title text changing or forgetting the Unicode sequence in one go. Also, if the menu gets translated, the tag remains the same, so it doesn't matter to us.

Your only concern is to make sure you give every menu item a tag and the rest takes care of itself. Well, after a bit of refactoring of our code, that is.

Menu Enum

We're going to be dealing with hard coded numbers in the StoryBoard and hard coding those numbers in our code would be a massive mistake. The chances of creating errors are extremely high, so we're going to add an enum to the code that we can use to map menu item tags to something we can use in code safely.

```
enum MenuTags: Int {

    // ****************************************File menu
    case file           = 10
    case fileNew        = 11
    case fileOpen       = 12
    case fileOpenRecent = 13
    case fileClose      = 14
    case fileSave       = 15
    case fileSaveAs     = 16
    case fileRevert     = 17

    case filePrintSetup = 18
    case filePrint      = 19

    // ****************************************Edit menu
    case edit           = 50

    // ****************************************Format menu
    case format         = 100
```

Our "File" menu will get a tag of 10 and the "New" menu item under File will get a tag of 11. And so on.

The pattern I use is to have the sub-menu items increment from the main menu item and to have the main menu items separated by a large increment. In this case, "File" is 10 and the next top-level menu

item, “Edit”, is 50 followed by “Format” which is 100. The first item under “File” is “New” so that gets a tag of 11.

Menu Helper

The menu helper class now needs minor refactoring to use tags instead of titles:

```
static func getTopLevelMenu(withTag tag: MenuTags) -> NSMenuItem {

    guard let mainMenu = (NSApplication.shared).mainMenu else {
        fatalError("Failed to get access to the application main menu")
    }

    guard let topLevelMenuItem = mainMenu.item(withTag: tag.rawValue) else {
        fatalError("Failed to get access to menu item \(tag)")
    }

    return topLevelMenuItem
}
```

We’re now passing in a MenuTag that corresponds to the top-level menu we want and the search takes the MenuTag rawValue, which will be an Int. We then need to refactor the sub-menu search:

```
static func getMenuItem(withTag tag: MenuTags, inTopLevelMenu topMenu: MenuTags) ->
NSMenuItem {

    let topLevelMenu = MenuHelpers.getTopLevelMenu(withTag: topMenu)

    guard let menuItem = topLevelMenu.submenu?.item(withTag: tag.rawValue) else {
        fatalError("Failed to get access to menu item \(tag) in top level menu
\(topMenu)")
    }

    return menuItem
}
```

Our error messages are a little less useful in that they will display the value of the enum now rather than the menu name, but that should be more than enough since the error should only ever happen in development if the code has been properly tested.

Usage Refactoring

The remaining task is to change the way we use the menu helpers. In the previous example, we have a file management class that handles the file menus. This includes a helper method to do the connection. That’s now going to take a couple of MenuTags enums:

```
private func handle(menu topMenu: MenuTags, item subMenu: MenuTags, withHandler
action: Selector) {

    let identifier = "mnu\(topMenu.rawValue)_\(subMenu.rawValue)"

    let menuItem = MenuHelpers.getMenuItem(withTag: subMenu, inTopLevelMenu: topMenu)
    menuItem.target = self
    menuItem.action = action
    menuItem.isEnabled = true
    menuItem.identifier = NSUserInterfaceItemIdentifier(rawValue: identifier)
}
```

The main difference here is how we determine the identifier. However, it's still fundamentally a combination of the menu and it's sub-menu. I also refactored the method signature to make it read a little easier.

That just leaves our calling routine, which now contains considerably fewer hard coded strings, yet retains its descriptive nature:

```
private func configureMenus() {

    handle(menu: .file, item: .fileNew, withHandler: #selector(fileNewMenuItem(_:)))
    handle(menu: .file, item: .fileOpen, withHandler: #selector(fileOpenMenuItem(_:)))
    handle(menu: .file, item: .fileSave, withHandler: #selector(fileSaveMenuItem(_:)))
    handle(menu: .file, item: .fileSaveAs, withHandler:
#selector(fileSaveAsMenuItem(_:)))
    handle(menu: .file, item: .fileClose, withHandler:
#selector(fileCloseMenuItem(_:)))
}
```

Enabling and Disabling Menu Items

There is one very important feature left to deal with and it comes with a massive gotcha if you don't know how to deal with it. To say it's far from intuitive is a major understatement!

All we want to do is enable and disable menu items. It's always going to be the case that menu items are not always appropriate, so they need to be disabled. File Save, for example, is not appropriate when you do not have a file open.

Adding methods to enable and disable menu items to our menu helper is very straight forward:

```
static func enableMenuItem(withTag tag: MenuTags, inParentMenu topMenu: MenuTags? =
nil) {

    guard let topMenu = topMenu else {
        // No top level menu specified, so assume this
        // is a top level menu item
        getMenuItem(withTag: tag).isEnabled = true
        return
    }

    getMenuItem(withTag: tag, inTopLevelMenu: topMenu).isEnabled = true
}
```

This method will enable a menu item by finding the sub-menu item in the parent menu and setting the *isEnabled* property to true. If no parent menu item is provided, it disabled all of the sub-menu items in the menu. Very straight forward.

It's personal preference whether you add a bool property to this method to set the value for *isEnabled*. Perdsonally, I refer to have a separate method for disabling to enabling as it makes the calling code more explicit about what it is doing. So, I also have a disable method:

```
static func disableMenuItem(withTag tag: MenuTags, inParentMenu topMenu: MenuTags? =
nil) {

    guard let topMenu = topMenu else {
        // No top level menu specified, so assume this
        // is a top level menu item
        getMenuItem(withTag: tag).isEnabled = false
        return
    }

    getMenuItem(withTag: tag, inTopLevelMenu: topMenu).isEnabled = false
}
```

It's a matter of readability and explicit methods just read better to me.

In our setup code, we can then just call the helper to disable the inappropriate menu items when we start up the program:

```
    MenuHelpers.disableMenuItem(withTag: .fileSaveAs, inParentMenu: .file)
    MenuHelpers.disableMenuItem(withTag: .fileSave, inParentMenu: .file)
    MenuHelpers.disableMenuItem(withTag: .fileClose, inParentMenu: .file)
```

Here we disable the File Save, File Save As and File Close menu items.

And therein lies the problem… when the program runs, these menu items remain enabled.

Enabling the Enable/Disable Functionality

By default, menu items will enable and disable themselves by following a hierarchy of rules, as defined in the Apple documentation, which generally end with the menu item enabled if certain handlers are not implemented. These rules override any setting we apply to *isEnabled*. Very irritating.

I spent several hours trying to diagnose this problem and ended up creating a simple test case for disabling a single menu item in preparation for a Stack Overflow question. This led to a solution without having to bother the Stack Overflow community.

In order to allow us to set the *isEnabled* property, we have to turn off a property of the menu called *autoenablesItems*. I started out setting this to false for the main menu; didn't help, my menu items remained enabeld. I then progressed to looking at the individual *NSMenuItem* menu items; didn't help, it doesn't have this property. While digging around, however, I found that the NSMenuItem has an optional *menu* property.

Dig around in the Apple documentation and you will see the stunning description:

THE MENU ITEM'S MENU.

Not sure how helpful this 'help topic' is, but I took a chance and set the *autoenablesItems* property to false, just to see whether it helped.

It did!

So, I incorporated this into the code that defines my menu items and the code to disable menu items started working:

```
private func handleMenu(_ subMenu: MenuTags, inParentMenu topMenu: MenuTags,
withHandler action: Selector) {

    let identifier = "mnu\(topMenu.rawValue)_\(subMenu.rawValue)"

    let menuItem = MenuHelpers.getMenuItem(withTag: subMenu, inTopLevelMenu: topMenu)
    menuItem.target = self
    menuItem.action = action
     menuItem.menu?.autoenablesItems = false
    menuItem.isEnabled = true
    menuItem.identifier = NSUserInterfaceItemIdentifier(rawValue: identifier)
}
```

Recently Used Files

I had no intention of making a section for this however, such a simple function, gave me so much trouble that I thought it best to make some notes.

I'm assuming that there is a Most Recently Used manager class in the application that can retrieve a list of recently used files. This is further assumed to be a list of file URL's. In StoryBoard, we have a menu item for Open Recent already, which has a number of pre-defined items under it:

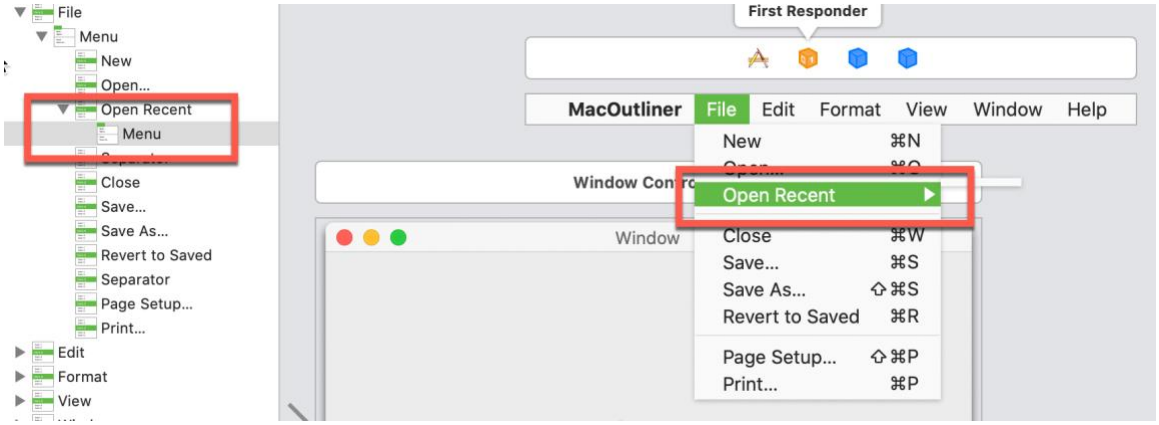


Figure 25 Open Recent Menu

In the designer, you'll probably have a child *NSMenu* and that will have a number of pre-defined items in a submenu. I had a lot of grief dealing with these default items and ended up deleting the *NSMenu* item under the *Open Recent* menu and adding a new, blank, *NSMenu* to the structure. That seemed to fix most of the issues I had been having, so I would suggest that's a good starting place.

Our *Open Recent* menu item has a tag associated with it, as shown in the previous enum section, so we can get easy access to it.

Minimal Initialisation

There is a minimal amount of initialization we need to do to our menu as a one-off task when the handler class is created.

Technically, we could do this every time we build the recent files sub-menu, but why repeat the effort. The initialization is called from the initializer of the file handling class and consists of:

```
func coinfigureMRUMenu() {
    let recentMenu = MenuHelpers.getMenuItem(withTag: .fileOpenRecent,
            inTopLevelMenu: .file)

    recentMenu.submenu?.removeAllItems()
    recentMenu.menu?.autoenablesItems = false
    recentMenu.target = self
    recentMenu.identifier = NSUserInterfaceItemIdentifier(rawValue: "ReuseMenu")
}
```

We retrieve the *Open Recent* menu item, clear its sub-menu of any design time items, turn off auto-enabling of the menu – we're going to have the items always enabled – set the target to our class and set an identifier.

Building the Recent Items menu

Building the menu then becomes a simple task. We're going to need a selector to handle the menu item if it is clicked, so we start with that:

```
@objc func reopenFileMenuItem(_ sender: NSMenuItem) {
    // Handle the menu item. The identifier gives us the
    // identifier of the specific menu item clicked.
}
```

As with other menu item handlers, this one has to be decorated *@objc* and will be passed the menu item that was clicked. We'll use the identifier of this item to determine what file was selected for open.

Building the menu is straight forward too:

```
func buildMRUMenu() {
    let files = mruManager.recentFiles
    let recentMenu = MenuHelpers.getMenuItem(withTag: .fileOpenRecent,
            inTopLevelMenu: .file)
    var itemIndex = 0

    recentMenu.submenu?.removeAllItems()

    for file in files {
        let identifier = "mnuRecent_\(itemIndex)"
        let caption = file.deletingPathExtension().lastPathComponent

        let menuItem = NSMenuItem()
        menuItem.title = caption
        menuItem.target = self
        menuItem.action = #selector(reopenFileMenuItem(_:))
        menuItem.isEnabled = true
        menuItem.identifier = NSUserInterfaceItemIdentifier(rawValue: identifier)

        recentMenu.submenu?.addItem(menuItem)
        itemIndex += 1
    }
}
```

For our purposes, the call to mruManager.recentFiles is assumed to return an array of the recently used files. I further assume that it will have limited the number of files to some pre-configured number, so I don't need to limit the number if the build code.

Next, we get access to the *Open Recent* menu and clear its existing sub-menu. We will be calling this build method many times from many different places, so we need to clear that sub-menu.

Now, we look through the files that the recent files manager passed us back.

So we can tie the selected menu item back to a file, I create an identifier with a prefix of "mnuRecent_" and the index of the file in the recent files array. This identifier will be padded to us as part of the menu handler.

The URL we have is just not suitable for display in a menu. For the menu display, we only want the name of the file, with the path removed and the file extension removed, so that's what we do next.

Then we build the menu item, setting the title to the name of the file, the target to ourselves, the action to our menu handler method and the identifier we generated to allow us to get the array index of the actual URL.

This menu item is then added to the sub-menu of the *Open Recent* menu and we loop round to the next file.

We're going to want to call this method when the file handler is first created and every time the user opens a new file, saves a file or saves a file with a new name.

The Main Problem

I said earlier I had problems. The main one was that, no matter how I went about building the submenu, it didn't appear in the program at run time. None of the code failed, but none of it worked either.

To this day, I have no idea why.

The eventual solution was to delete the menu item and re-add one. All very confusing and one of those things I tuck away to try on a fresh application next time I'm doing this.

Chapter Two

Dialogs

Writing for the Mac is really cool. When you create an application, it comes with a load of code already written for you that just works. Unfortunately, most of it is hidden and some of it, while useful, is inflexible.

For example, while it's great that an out-of-the-box application comes with an already hooked up and functional About popup, it's very basic and not customisable. So you quickly get to a place where you want to create your own popup windows.

This section is about a few of those pop-ups, such as an About box and a Preferences window.

PROJECT STRUCTURE

It's not a requirement, but I like the idea of starting my projects with a little structure. For dialogs, I like to have a dialogs folder where I can keep the files. For example:

Figure 26: Folder Structure

Reading from the top, we have the application root for the *MacWasher* application and under that we have the initial *MacWasher* folder. Under there I have a placeholder folder called *Dialogs* into which I will put all of the popup dialogs for my application. For each dialog, I create a sub-folder for that specific dialog; *AboutBox* in this case.

All of the following examples are going to follow this structure for storing the files related to that particular dialog.

About Box

About Box

Xcode templates are pretty cool. You create a new application and all sorts of stuff just works, right out of the box. One of those things is the *about application* menu which displays a default about popup window:

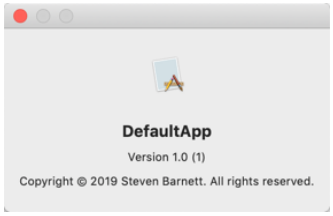


Figure 27: The default about box

Now, for a trivial app, this is fine. It gives you the basic information you want, has an icon and can be closed with the close button. Pretty much everything you want to do with an about box. Problem is, it has a few issues too.

The layout is fixed and the content is prescribed. You cannot add extra text and you cannot move anything around. It's a fixed size which is fairly small and, I intuitively look for a close button when a window is displayed and this doesn't have one.

None of these are disastrous issues, but they become important if you use libraries that require attribution - you just cannot add any to this window. The solution is, to say the least, painful to set-up when you're new to Mac programming, but very logical once you get your head round the process.

In this section, my aim is to end up with an about window like this:

Figure 28:Target About Box

The key aspects of this window, which is still pretty simple are;

1. I have a larger image which I have moved off to the left.
2. The name of the application is in more prominent text.
3. I have customised copyright text.
4. There is an attribution to the company that created the code code the app is constructed on top of.
5. I have a close button.
6. The title bar has gone.
7. You can move the window around by clicking and dragging anywhere in the window.

How much of this you might want in your about box is up to you. I tend to see this as a minimum which I would often extend to have links to web sites and a version number.

So, where do we start.

Creating The Storyboard

First thing I add to my folder is a new storyboard. I realise I have the main storyboard and I could easily add my new dialog to that, but a storyboard can get very complicated and very crowded very quickly so I take the stance that the about box is a stand-alone piece of code, has a single function, is self contained, so can go into a storyboard on it's own.

So, first job, add a storyboard. File -> New -> File and pick storyboard…

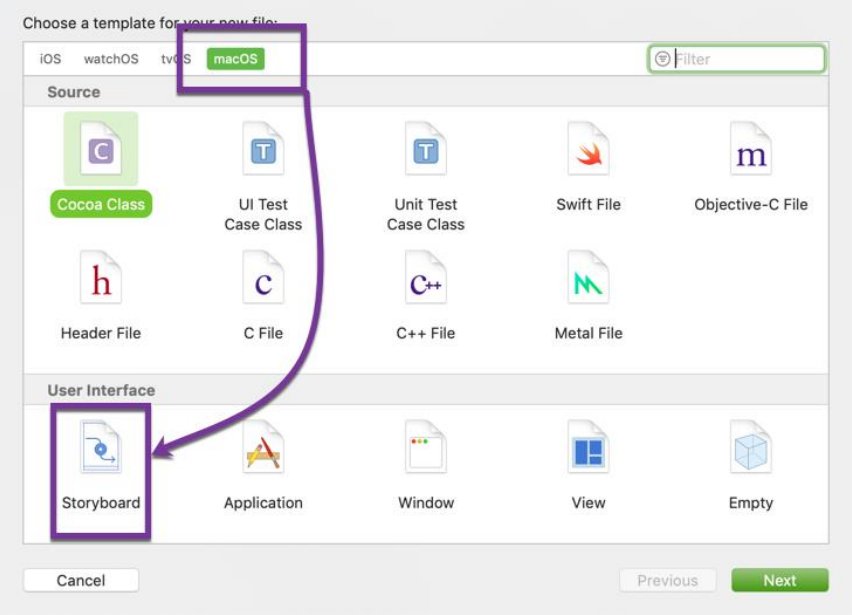


Figure 29: Create new storyboard

When prompted, set the name of the storyboard file to *AboutBox* so we end up with *AboutBox.Storyboard* in the AboutBox folder. The storyboard will be empty, of course, so the next step is to add a window. So, fire up the library and add a *Window Controller*.

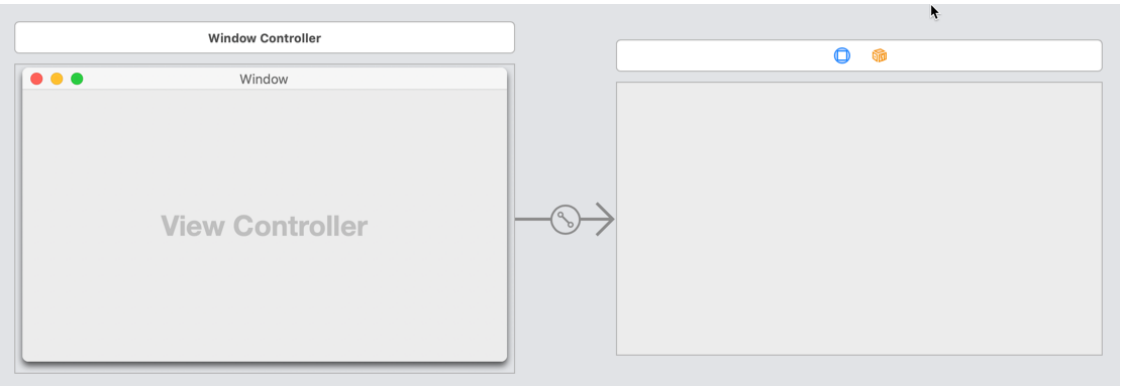


Figure 30: New Window Controller

We're going to need to do some work in code for the window and the view, so we might as well get the file creation over with now. So we need to add two new classes

- *AboutBoxWindowController* is a COCOA class, derived from NSWindowController. We need to create one of these and assign it to the window. Important is prompted to create an XIB file when creating the window controller, uncheck the box. We do not want an XIB file.
- *AboutBoxViewController* is. COCOA class, derived from NSViewController and assigned to the view that we will be dropping our layout onto.

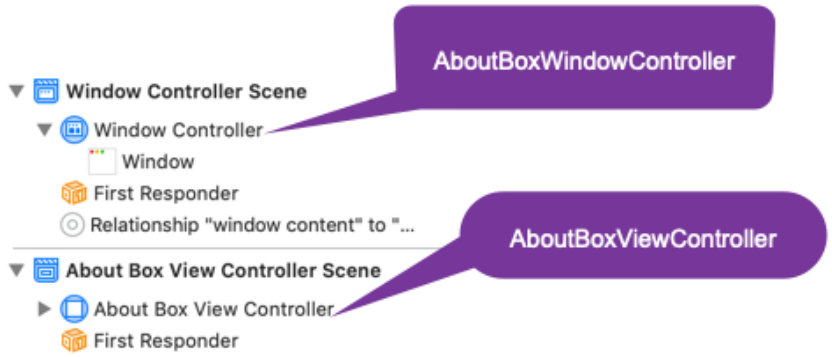


Figure 31: Controller assignments

While we're in there, we are going to want to refer to the window and the view in code, so we should give them names. So set the storyboard ID's for the window and the view.

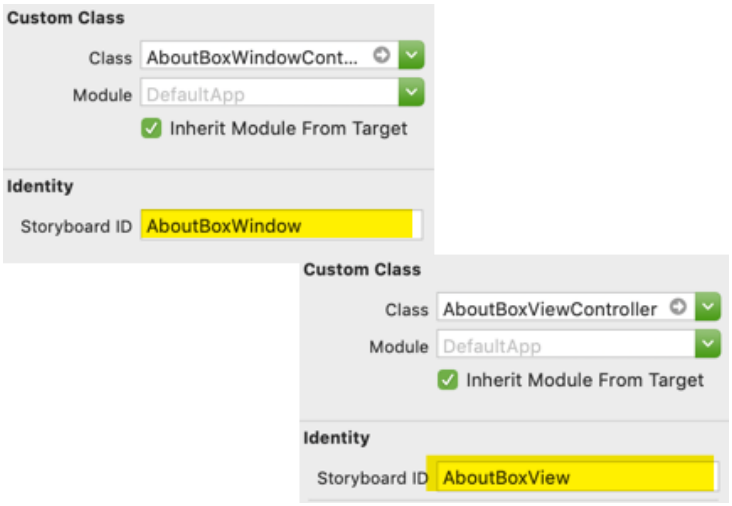


Figure 32: Story board ids for the window and view

Now, there is a lot of tidying up to be done and a few bits of ode to write yet, but we have a window and we have a view and it would be nice to see it on the screen. So lets just add a couple of components to the view so we have something to see, even if it does nothing. Add a label and add a button.

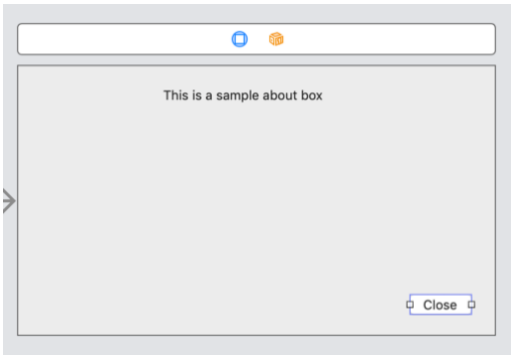


Figure 33: Dummy about box content

Now it momentarily gets complicated. We have our main storyboard, along with its menu system and, specifically, an *about* menu item. We also have a new storyboard with our dummy about box and neither knows about the other. Our task is to get them talking.

Connecting The Menu

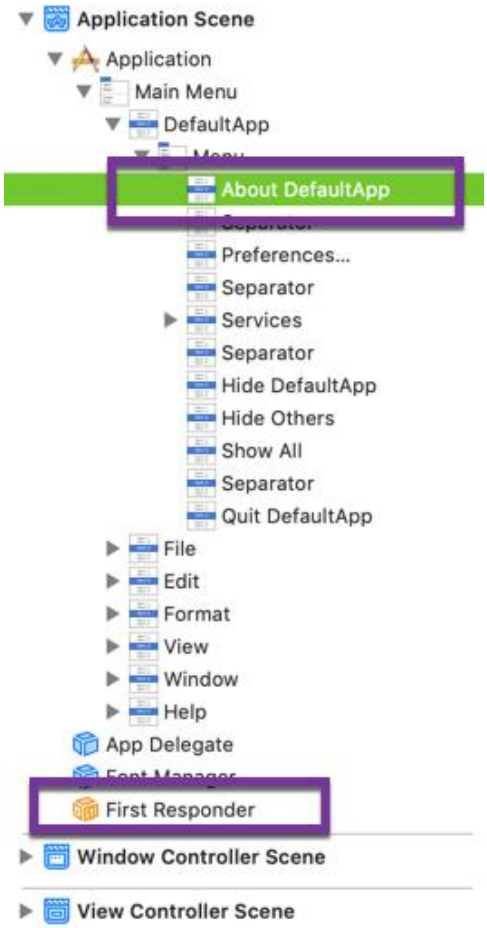
Our first task is to create something to be called when the About menu item is clicked. Since this can be called at any time regardless of the active window, I'm going to dump the code in the AppDelegate file:

```
@NSApplicationMain
class AppDelegate: NSObject, NSApplicationDelegate {

    @IBAction func showAbout(_ sender: Any) {
        print("Show about")
    }
}
```

Not that impressive yet, but we just want an end point to connect to. We are going to have to flesh it out before it does anything, but we have to hand-crank the method stub before we can connect to it from the menu. That signature, by the way, is very important. Get it right.

Now, we skip over to the main storyboard and find the About menu item in the structure.



To make the connection, <Control>Drag from the menu item to the item named *First Responder*. Why? Well we're connecting a menu in the application to a completely separate window. The application menu knows nothing of the about box window so you can't connect directly. Luckily, Xcode knows about the method we just created in the AppDelegate, so it is capable of making that connection.

When you <Control>Drag to First Responder, you are telling your program the name of the method you want to call when the menu it clicked and the app is responsible for finding someone, anyone, in your application that implements that method. When you release the mouse, all hell breaks lose, a window pops up and populates itself with every method that might possibly respond to the menu.

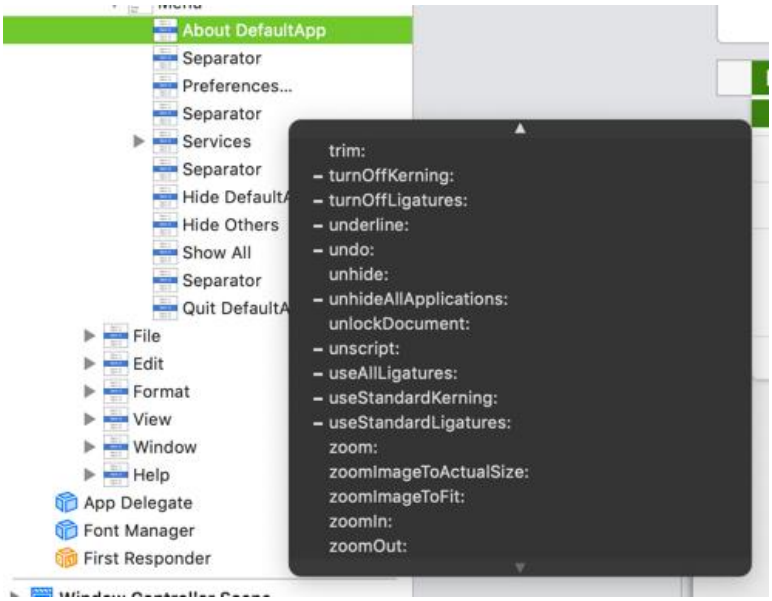


Figure 34: Potential first responders

All we have to do is scroll through that list and select the *showAbout* method we defined in the AppDelegate.

Ok, so run your application and select the About menu item. You should be rewarded with this stunning message in the debugging console:

> **Show about**

It may not be spectacular, but it shows we have the right linkage and right is always good.

Displaying The Dialog

Now we get to the detail of getting a window on screen. It's not particularly complicated, but several things have to happen at the same time. Lets's start at the start… finding the window.

When we defined our storyboard, we assigned storyboard id's to the window and the view. We're going to be using those to locate our window in the app delegate code. In this case, it's pretty simple code to locate our window…

```
@IBAction func showAbout(_ sender: Any) {
let storyboard = NSStoryboard(name: NSStoryboard.Name("AboutBox"), bundle: nil)

if let aboutController = storyboard.instantiateController(withIdentifier:
"AboutBoxWindow") as? AboutBoxWindowController {

        // We have the controller, so display the window
    }
}
```

So, what's going on here? The first let statement gets access to the storyboard we created for our about box:

```
let storyboard = NSStoryboard(name: NSStoryboard.Name("AboutBox"), bundle: nil)
```

The name is the file name that we gave the storyboard when we created it. This is our starting point into the windows and views. Next, we get a reference to the window that we created. There are two ways of doing this; one where we get the default start window of the storyboard and one where we get a named window. I prefer to be explicit about what I am doing, so I get a reference to the window with the storyboard ID I created in interface builder:

```
if let aboutController = storyboard.instantiateController(withIdentifier:
"AboutBoxWindow") as? AboutBoxWindowController {
```

I cast the result to ensure that I have access to the correct window type. If the coast fails, then nothing will happen. At this point, we have access to the AboutBox window and are in a position to be able to display it. More on that next.

Displaying the About Box Modally.

We want our about box to appear modally, so nothing else in the app is going to get focus until the about box is closed. This is going to involve two separate pieces of code, one to display the window and one to close it. **Important** you must have both pieces of code in place. If you put the display code in and not the close code, then your app will hang - there will be an app modal screen that you cannot close!

Inside the code block in the app delegate, we display our window with:

```
let app = NSApplication.shared
app.runModal(for: aboutController.window!)
```

We get a reference to the current application object and run the default window of the controller modally. The view associated with the window will be loaded and displayed. At this point, your application will wait for you to close the About Box.

Switch over to the about box you created earlier and create an IBAction for the close button. This requires two lines of code:

```
@IBAction func closeAboutBox(_ sender: Any) {
    NSApplication.shared.stopModal()
    self.view.window?.close()
}
```

The first line tells the application that we no longer want to be displayed modally. This frees up the application to respond to user input again. The second causes the About Box to close.

Run the code and we get our first about box displayed on the screen.

Figure 35: First about box displayed

When you press the close button, the about box will close. If you try to interact with the main window while the About box is displayed, you should get a comforting bleep.

Cleaning up

We now have a working about box and could stop here. Putting content onto the form is just the same as it is for any other form in our application, so its down to styling, placing icons and positioning text.

We do, however, have some issues we want to fix. If you look at the window, it has a title bar and icons for close, minimise and maximise. These are unhelpful. We want rid of them…

The minimise and restore buttons are going to destroy the layout of the view. This window is not designed to expand to full screen, nor is it supposed to be able to be minimised. The close button is positively dangerous - it will close our window without resetting the modal status of the app meaning the whole app will hang.

First things first, lets get rid of the title bar and the icons. Head over to the Window definition in the storyboard and go to the properties inspector. Change them as follows:

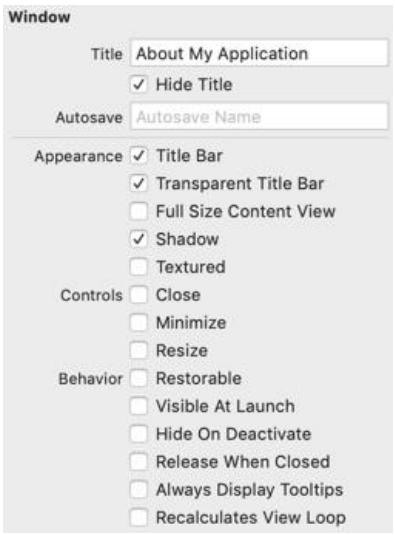


Figure 36: Modal Window Options

So, here we have;

- Hidden the title bar.
- Made the title bar transparent.
- Unticked the close, minimise and resize controls.
- Unticked the restorable option.

This gives us our slightly better looking modal window with none of those nasty buttons to break our design.

Figure 37: About box without a title

Looks better, but we have a new problem. Without a title bar, we cannot move the window. It appears in the middle of the screen and it stays there until you close it. What if you want to move it to the side to see something underneath it?

Making the window movable

Open up the code AboutBox Window Controller an update the WindowDidLoad as follows:

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

    if let window = window {
        window.isMovableByWindowBackground  = true
    }
}
```

This will modify the state of the about box window to make it dragable by clicking and dragging anywhere on the window. We no longer need the title bar to re-position our window.

The rest, as they say, is your problem. We have an about box and all we need now is to add content.

Alternate Ending

Ok, so you don't like windows without title bars and you really, desperately, want the close icon and a title. And I went and took all that away.

Well, there are several ways to display our about box, so we can achieve your needs with some refactoring.

Lets start with the About Box itself. We need the title bar back and the close icon. That's fairly straight forward, so go to the window and change the properties as follows:

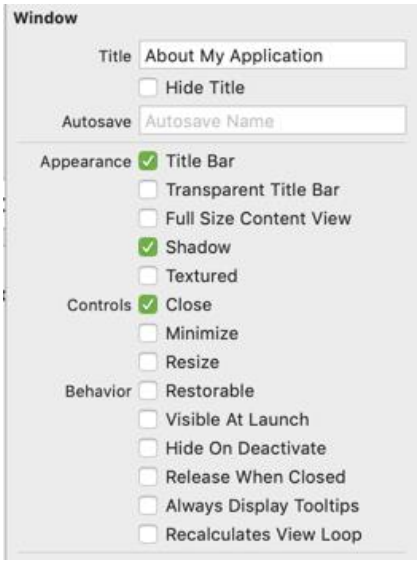


Figure 38: Alternate window options

This results in a new style of window with a title and a close icon.

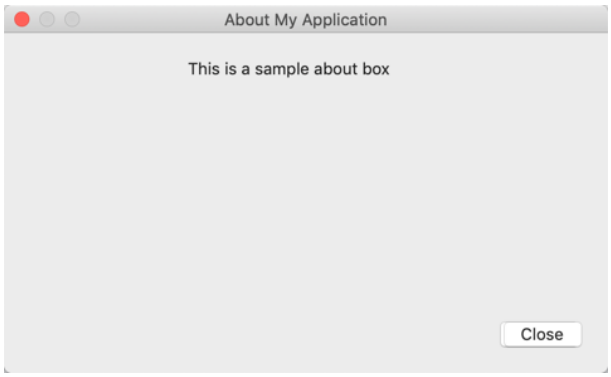


Figure 39: Alternative About Box

But we're not finished yet. We have that thorny problem of the close button and the application modal. We could handle the about box closing and undo the modal, I suppose, but lets try a different way of displaying the About Box in the first place.

Remember the close button code on the About Box... go change it to this:

```
@IBAction func closeAboutBox(_ sender: Any) {
    self.dismiss(self)
}
```

The modal and close code is gone, replaced with a simple dismiss call.

This means we need a new way of displaying our about box, so head over to the App Delegate and change the showAbout method as follows:

```
@IBAction func showAbout(_ sender: Any) {
    let storyboard = NSStoryboard(name: NSStoryboard.Name("AboutBox"), bundle: nil)

    if let aboutController = storyboard.instantiateInitialController() as?
NSWindowController {

        if let aboutView = aboutController.contentViewController as?
AboutBoxViewController {
            aboutView.presentAsModalWindow(aboutView)
        }
    }
}
```

Just for the fun of it and because we only have one window inthe storyboard, I've changed the code to get the controller to call storyboard.instantiateInitialController. This will find the initial controller (the only one in this case) regardless of the identifier. Once we have the window controller, we extract a reference to the view in the controller, which will be the about box view.

Once we get the view, we can present it as a modal window and OSX will handle the modality issues.

When we run this, nothing happens. Well, that's a bit of a bust, what's wrong?

Initial Controller

If you open up the About Box storyboard, everything looks fine:

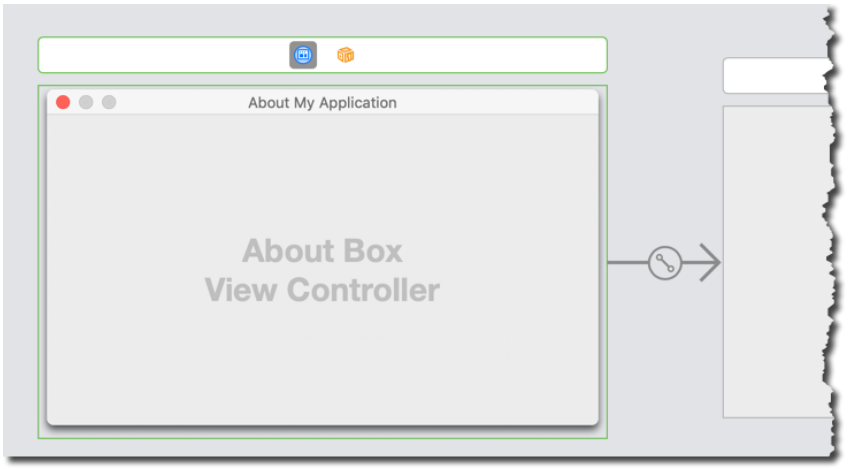


Figure 40: Failing storyboard

But, there is something missing from this picture. Every storyboard needs to identify the initial window and view and we have not done that. We got away with it before because we located the window by name. However, we're asking for the initial controller and we havenot set one.

We need to tell the storyboard which window to open by default.

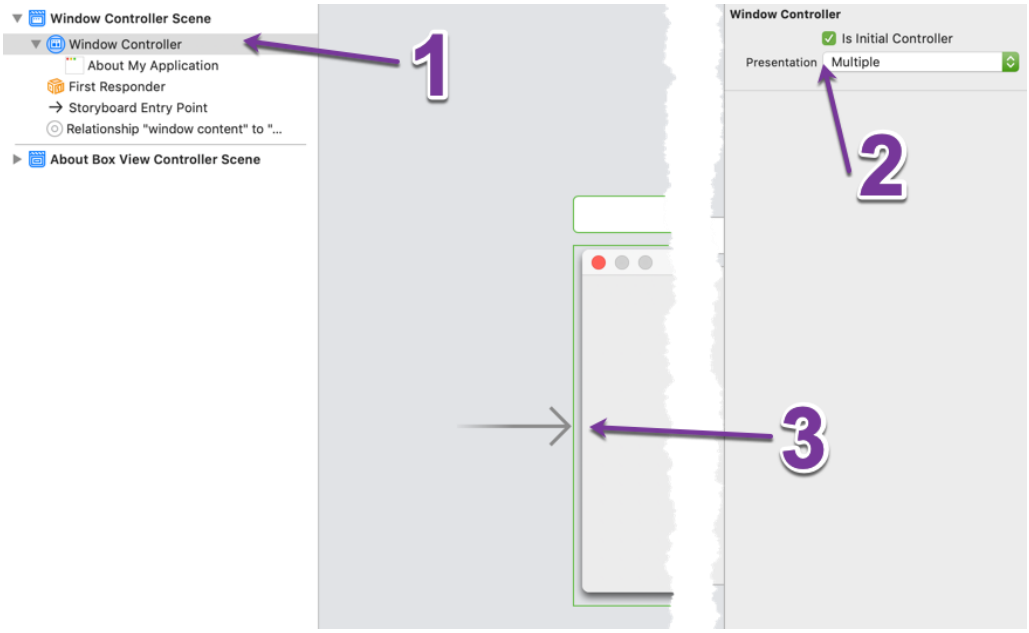


Figure 41: Setting the initial controller

So, select the window controller (1), tick the *is Initial Controller* tick box (2) and the initial controller indicator will appear. This is now the initial window for this storyboard.

While we're in there, we need to set the title for the window. That's set again the view:

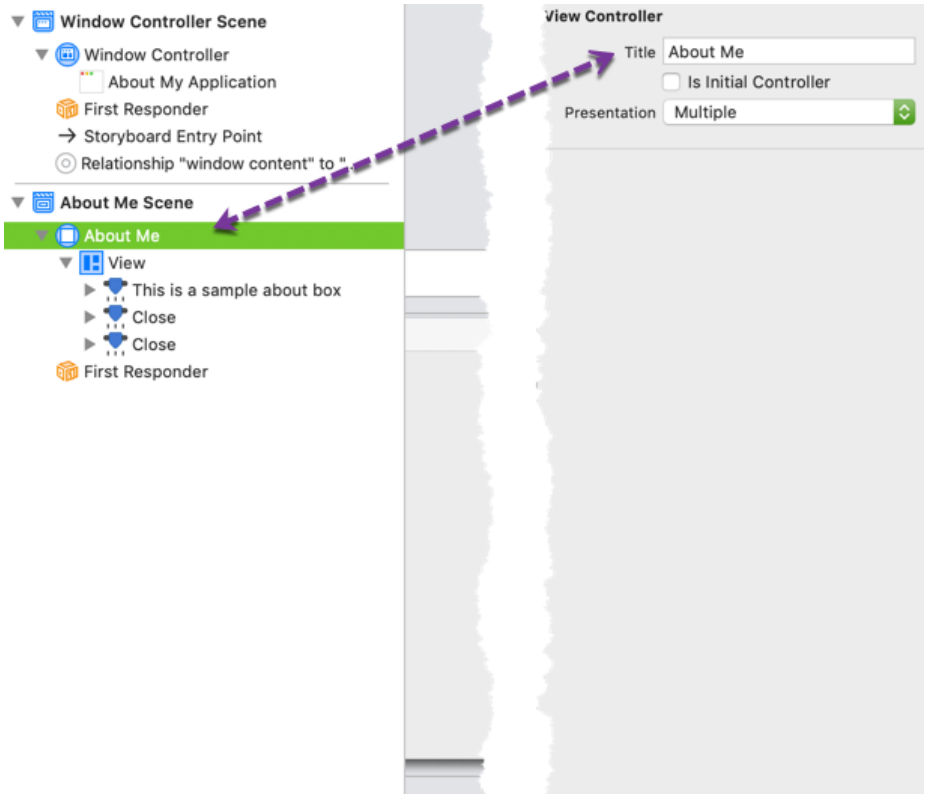


Figure 42: Setting the window title

This time, when we run the code, the about box will appear.

Just to pose one problem for you; you will see what appears to be a bug. The about box suddenly gets a maximise button. It's very irritating since we explicitly said we didn't want it and it does not appear in the designer. So far, I have not found a way to get rid of it.

PREFERENCES WINDOW

Preferences Windows

All but the most trivial applications allow users to customise them in some way. That's usually done with a preferences window.

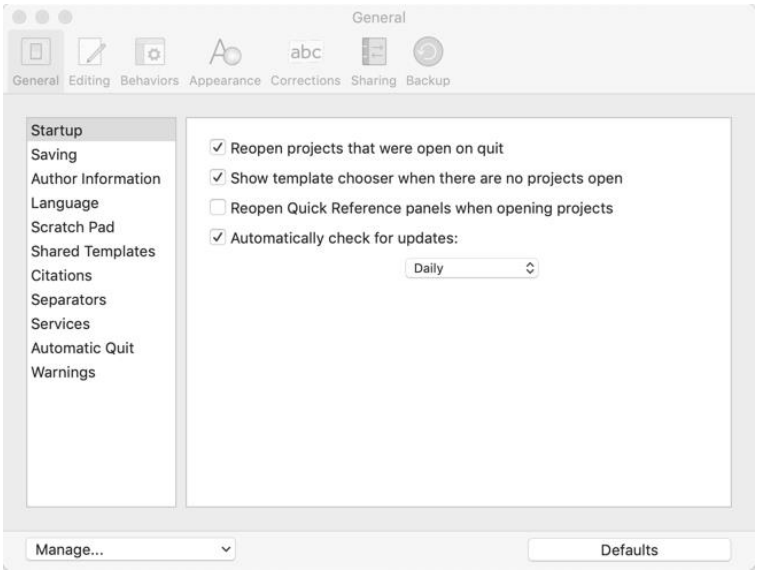


Figure 43: Sample preferences window.

Typically, we have a set of icons across the top of the window that categorise the preferences and a variable height section below that shows the category specific options. You might optionally have buttons along the bottom of the window. Options are typically applied as options are changed, though that does not always make sense. That decision is yours and yours alone.

Basic Setup

So, how do we get started?

Like the about box, my preference is to have my preferences window defined and maintained in it's own storyboard, so the first job is to create ourselves a folder and add a StoryBoard to it.

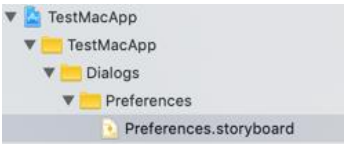


Figure 44: Preferences dialog folder

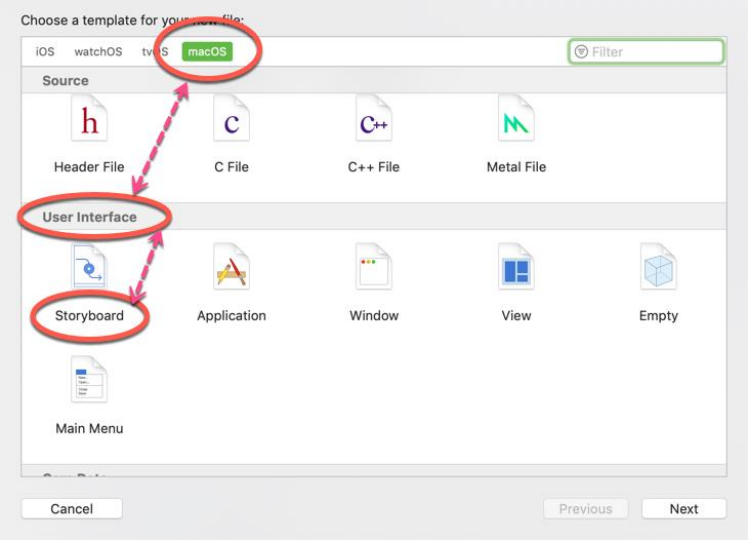


Figure 45: Preferences storyboard.

This will result in an empty storyboard. From the library, we add a Window Controller to put something on the canvas as a starting point. We also create a window controller class, which we will almost certainly use later. It's important to ensure that it's derived from NSWindowController and that we don't create an xib since e are creating a storyboard.

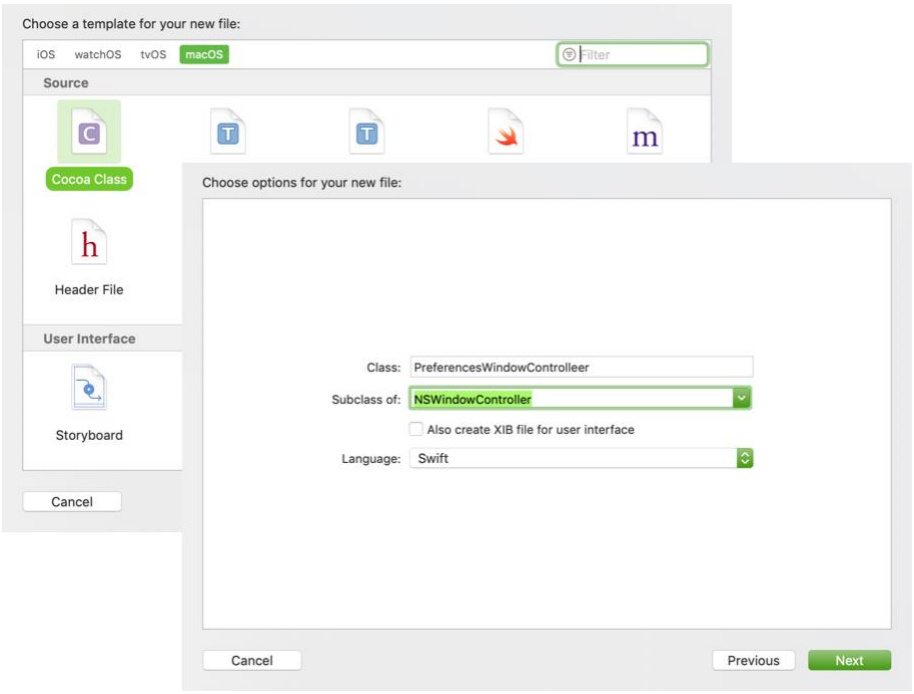


Figure 46: Preferences window controller.

Once we have the class, set to as the class of the window we just created. We're on our way!

The Tab View

When the window was created, it auto-generated a View Controller and view for you. These are good to have, normally, but not what we want. Click on the View Controller in the storyboard navigator and delete it. You'll end up with just the window.

Next, from the object library, add a Tab View Controller to the storyboard. You'll get a tab view and two default tab pages:

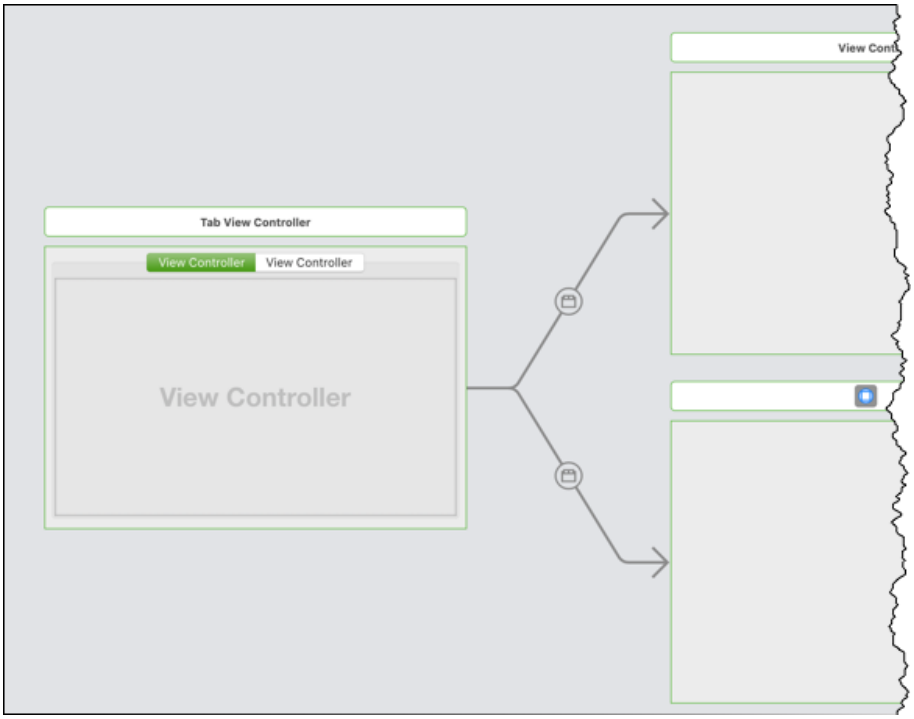


Figure 47: The default tab view

There are important things to note about this;

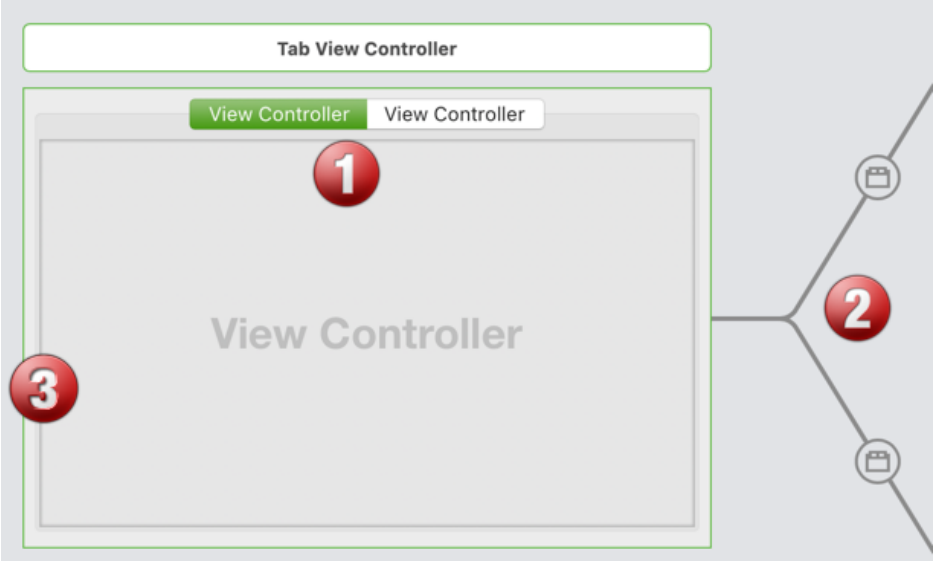


Figure 48: Special points to note

We have some issues we need to address with this new tab view.

1. By default a tab view comes with the tabs. We need to decide whether this is what we want, and it may work for only two or three options sets, or whether we want to display a toolbar of icons. For the purposes of this demo, we're going to change this to

a toolbar. You may decide not to, in which case, you're done with this part of the configuration.

2. We got two default tab pages generated for us. Brilliant if we only want two. We're going to be adding a third, just so we know what to do.
3. Third issue is more subtle and critical to the working of our app. When we deleted the view Xcode generated for us, it broke the linkage between the window and the initial view. Adding the tab view did not restore this for us so our new tab view will not get displayed if the window is displayed. In fact, nothing will get displayed.

Connecting our view

Lets deal with the easy issue first. Lets connect our tab view to the main window. That turns out to be very easy.

Click on the Window Controller and control-drag onto the tab view:

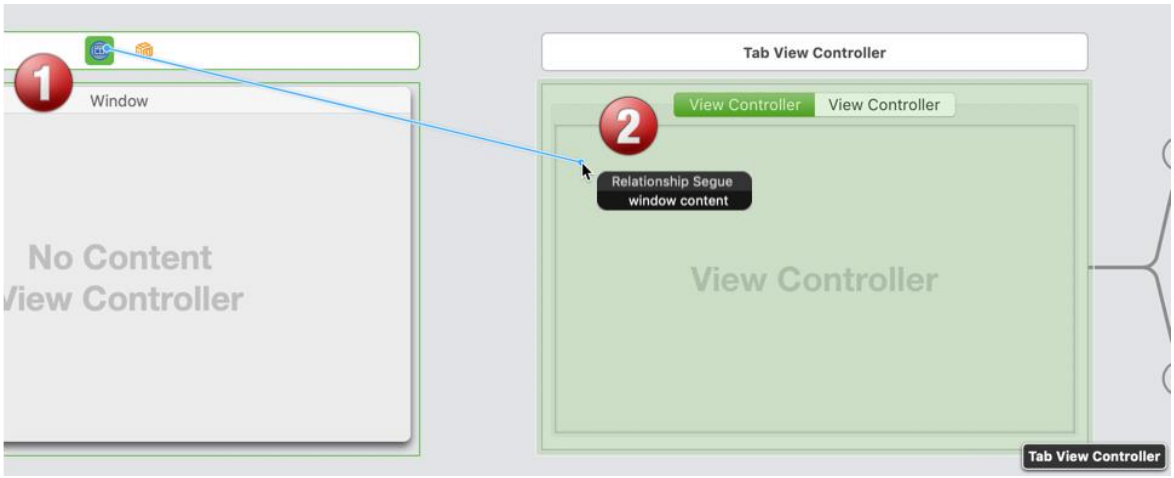


Figure 49: Connecting the view to the window

From the menu that appears when you release the mouse, select the *window content* option. That will make our tab view the initial view to appear in the window.

Trying it out

So, with no coding we have a preferences window. We can't do anything with it, but we have it. Next job is to prove to ourselves that what we have created is going to work and, to do that, we need to be able to display our dummy preferences window. To do that, were going to have to create a handler for the menu and connect the supplied menu to it.

Were going to be re-using the preferences window. It's a simple touch, but it means that, if the user opens the window a second time, they are positioned on the same tab. As with all of this, you may decide not to do this - it's a design decision. To do this, though, we're going to need to keep a reference to the window and we're going to hold it in the AppDelegate for ease:

```
@NSApplicationMain
class AppDelegate: NSObject, NSApplicationDelegate {

    var preferencesController: NSWindowController?
```

Our preferences is a class derived from NSWindowController, so we can define our variable as a window controller.

Next, we need a handler for the menu. We're going to keep this to the absolute minimum of code.

```
@IBAction func showPreferences(_ sender: Any) {

    if (preferencesController == nil) {
        let storyboard = NSStoryboard(name: NSStoryboard.Name("Preferences"),
bundle: nil)
        preferencesController = storyboard.instantiateInitialController()
as? NSWindowController
    }

    if (preferencesController != nil) {
        preferencesController!.showWindow(sender)
    }
}
```

So, we're defining an @IBAction. That's going to be needed because we're going to connect this function to a menu item and to make that connection, we need a pre-existing @IBAction.

First check is to see whether we have already initialised the preferencesController variable to an existing instance of the preferences window. If not, then e need to get a reference to the storyboard that we created (Called Preferences) and then we need to initialise the preferencesController variable to the controller that we connected as the initial controller for this window.

This leaves us with an initialised reference to the preferences window. We can then show it using the showWindow call.

The remaining task before we can see our masterpiece is to connect the preferences menu to our preferences @IBAction function.

So, open up the application scene and navigate down to the Preferences menu item. Once there, control-drag to the first responder and select our showPreferences method from the list. If you can't see it in the list, then they method signature is wrong; make sure it's an @IBAction and that it takes one parameter.

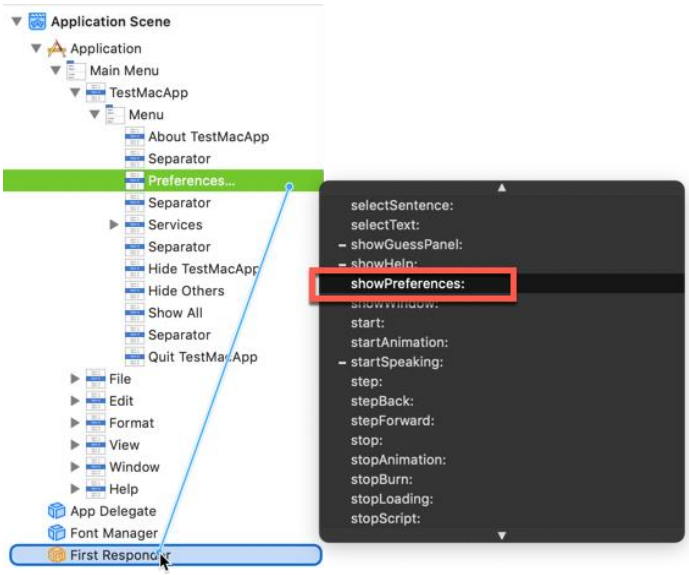


Figure 50: Connecting the menu item

At this point, you should be able to build and run the application. Selecting Preferences off the application menu will display our empty preferences dialog.

Changing the Tab Type

So, we have working code and have successfully created and displayed a preferences dialog. That still leaves us with the two problems of the tabs being displayed as tabs and not icons and being limed to two tabs. Lets deal with the tab type next and see if we can replace those tabs with a toolbar of icons.

Back to the Preferences storyboard and click on the Tab View Controller (not the scene, the controller). In the attributes inspector, you will see a number of options, one of which is the style:

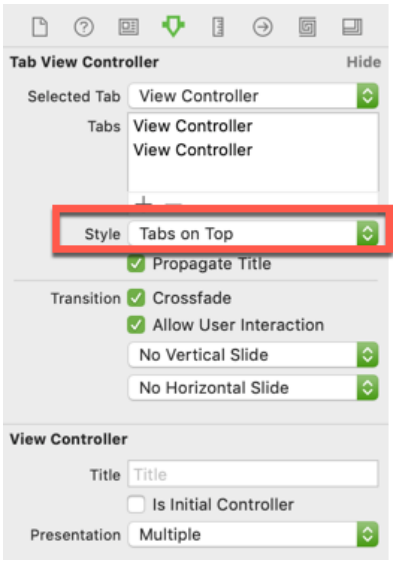


Figure 51: Tab styles

Go ahead and change this to *Toolbar*. Also, don't be surprised that your tabs will disappear and be replaced with nothing. That's expected (and very irritating)!

Click on the first Tab View Item under the Tab View Controller. Over in the attributes controller, set the Image property to one of the options in the drop down list. While you are there, change the label text to something specific to the preferences page. At the moment, it is defaulting to the name of the tab control, so change it to your custom text. Do the same for the second Tab View Item. You will not see any difference in the Tab View Controller - these icons will not be displayed at this time.

Run your application and display the preferences. The two tabs you had preciously will be replaced with a toolbar and two icons:

Figure 52: Toolbar buttons

Clicking on the buttons switches you between the two options set. If you want to verify that for yourself, add a label to the two View Controller's so you can see them swapping when you click the icons.

Adding Tabs

Our two tabs are good and, in many cases, all we will need. What do we do, though, when we need more options? How do we add more tabs?

Turns out to be very easy…

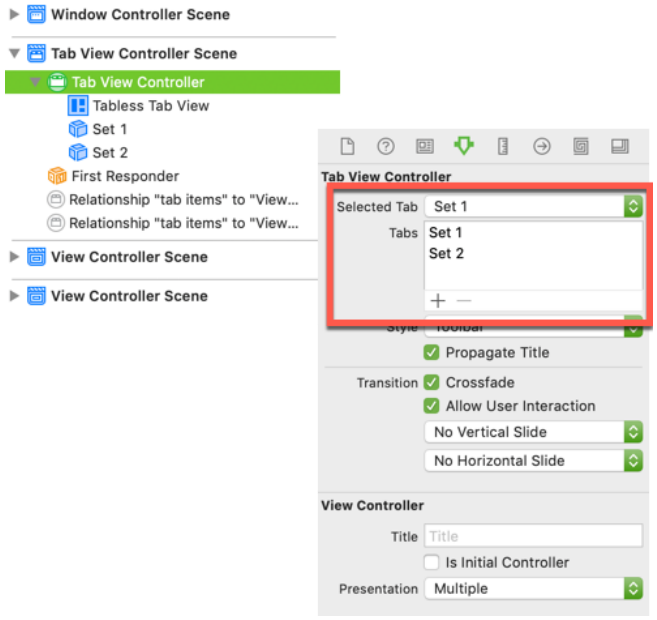


Figure 53: Adding a tab

Select the Tab View Controller and, in the attributes inspector, click the "+" to add a new tab. A new tab will be added to the storyboard and connected to our tab controller. You will need to go and set the label and icon and we're done; new tab added.

Getting a Little More Adventurous

Just when you thought you were nearly finished…

One of the features of Mac preferences windows is that they dynamically resize when you change tabs. They optimise themselves to match the content of the tab. This is designed in the storyboard, where we customise each tab to it's content:

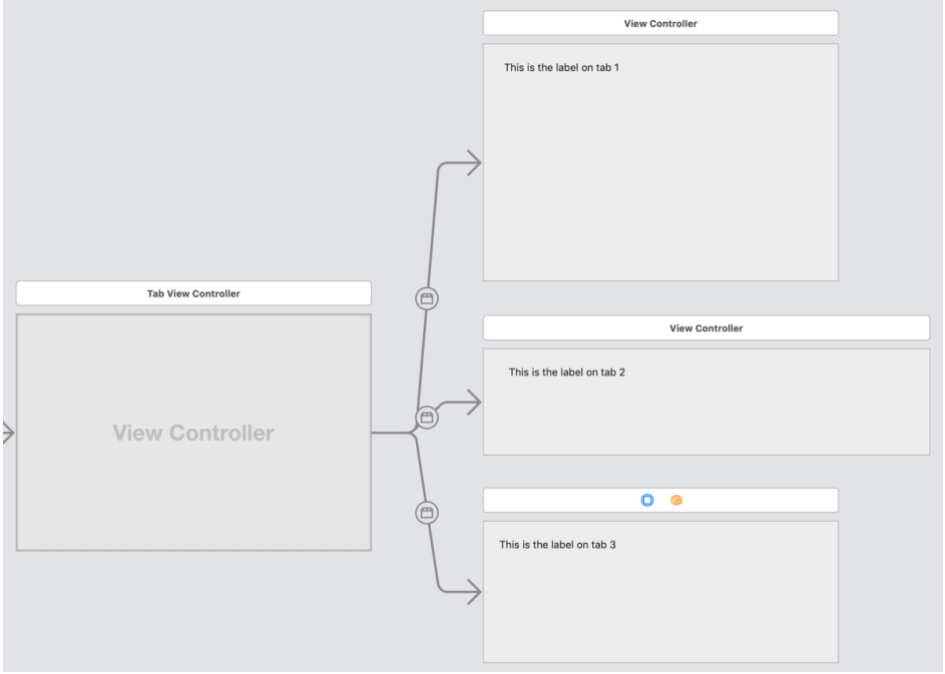


Figure 54: Tabs of different sizes

The problem we have is that this is not built-in functionality. If you run the app with these window sizes, you will find the are all re-scaled to the same size. Unhelpful. If we want to look professional, we need our settings window to dynamically resize when we change tabs.

To do that, we're going to need view controllers for each of our options tabs. So, start by creating a PreferencesViewController class with a base of NSViewController. We only need one of these because we're going to apply it to every tab. The code is common, so we can get away with it.

Once you have your class, select each of the tab scenes and set the class in the identity inspector.

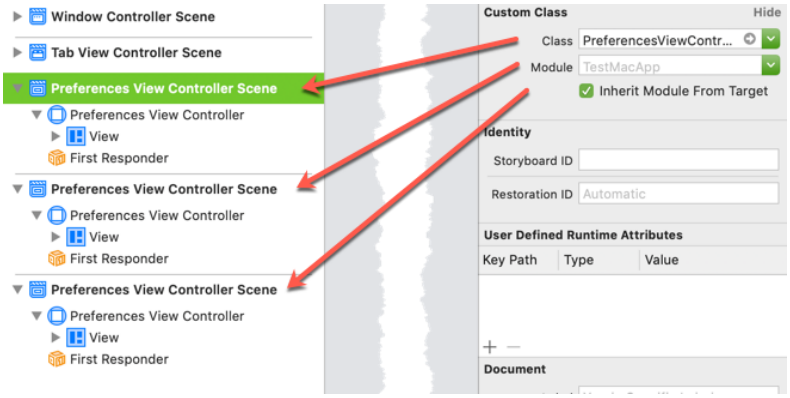


Figure 55: Attaching the view controller

This gets us in a good position to make our variable size dialog because the view controller will be initialised for each tab. All we need to do is add come code:

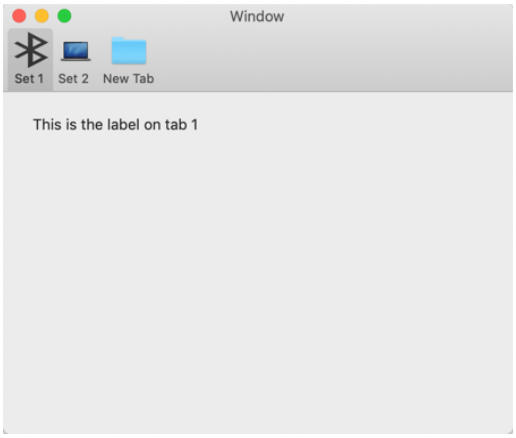
```
class PreferencesViewController: NSViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // Set the size of the window
        self.preferredContentSize = NSMakeSize(self.view.frame.size.width,
self.view.frame.size.height)
    }
}
```

The first change is in the viewDidLoad method. This is going to get called when the view is loaded and gives us an opportunity to set the preferred size for our view. If we set our preferred size to the size of the view at design time then, every time the view is displayed, the window will resize to match.

As we move between options sets, the window will resize to fit:



One additional touch here is that the title of the window is always fixed. It would be nicer if we could get the title of the window to change along with the tab. So, back to the PreferencesViewController and add a viewDidAppear method:

```
class PreferencesViewController: NSViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // Set the size of the window
        self.preferredContentSize = NSMakeSize(self.view.frame.size.width,
self.view.frame.size.height)
    }

    override func viewDidAppear() {
        super.viewDidAppear()

        // Set the window title to match the tab title
        self.parent?.view.window?.title = self.title!
    }
}
```

Each time a tab is activated, the title of the window will change to match the tab currently being displayed.

Transitioning between tabs

You now have a pretty nice preferences dialog and it looks pretty good when you are on a page. There is, however, a problem. As you click on each tab, there is a noticeable pause. The new tab appears, then it resizes. The result is find, but the transition is ugly.

We need to fix that.

So, back to the Storyboard and select the Tab View Controller in the structure. Over on the right, in the Attributes Inspector, you will see an option called Crossfade and it will be ticked. Untick it.

Crossfade is described as

> "A transition animation that fades the new view in and simultaneously fades the old view out. You can combine this animation option with any of the "slide" options in this enumeration."

And this is the problem. The old option set fades out, the new one fades in and then the window resizes. It makes for a jerky transition. With it turned off, the switch is much smoother.

Reusing our window

Back at the start, when we added code to the appdelegate, I mentioned re-using the window, which is why we are keeping a reference to it. That's pretty useless if the window is closed, so we need to keep the preferences window alive when the user closes it. We can achieve that with a simple call in the window controller:

```
class PreferencesWindowController: NSWindowController {

    func windowShouldClose(  sender: NSWindow) -> Bool {

        self.window?.orderOut(sender)
        return false
    }
}
```

windowShouldClose will be called when the preferences window is about to close. By returning false, we stop the window from closing. To ensure that the window doesn't stay on top of our application, we make a call to orderOut which sends the preferences window behind the main window in the window stack and removes it from the screen. The window will remain there, but not visible.

Making your tabs do something

We have come a long way and have ended up with a pretty good looking preferences window that we can populate with lots of options. And that is where our next problem come at us like an express train. We have options, but they can't do anything. If we try to connect them to our view controller, it fails - all the tabs are associated with a single view controller and it is of type NSViewController. The content of the tabs is an VSView. Not compatible.

To add functionality to our tabs, we need to create view controllers for their content views.

So, create a view and base it off NSView:

```
class OptionSet1ViewController: NSView {

    override func draw(_ dirtyRect: NSRect) {
        super.draw(dirtyRect)
    }
}
```

Click on the view for the first tab and set it's class to the class we just created:

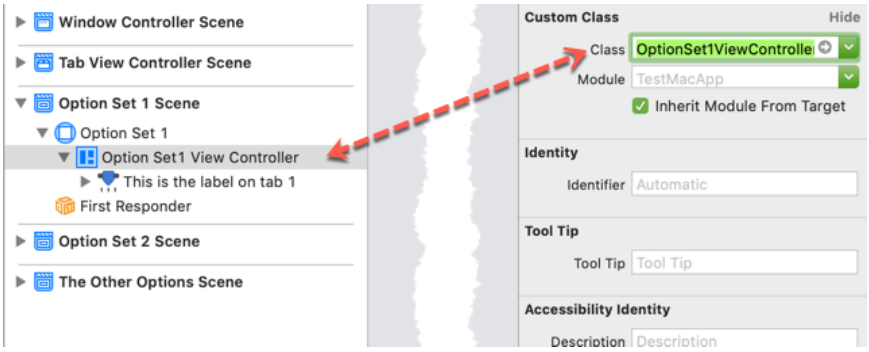


Figure 56: Tab view controller

Once we have a class backing our tab, we can add interaction as we would with any other view, creating @IBOutlet's and @IBAction's.

Wrapping Up

All you need to do now is write code to load and save the preferences.

The other thing you will need to deal with is informing the rest of your application that preferences have been changed. You could, of course, implement the new preferences when your application restarts, but it would be far better to apply them immediately. To do that, you need to inform every window indoor application of the change.

Trying to find every active window and class is going to be difficult and very unreliable. There is a solution, however, that lets your application decide what changes it is interested in. That's covered in the section on the Notification Centre.

Chapter Three

Internal Communications

There are many ways to communicate between your application and the outside world, be it the internet, the user or the file system. However, there are occasions when you need to communicate internally, between components of your own application. You might have a background task running that needs to inform the user when some data becomes available, or you might change a system setting in your preferences and need to propagate that change to the rest of your program.

You can do this manually with, for example, the observer pattern or you can be lazy, like me, and use the built in notifications mechanism. I suggest you go for lazy…

MASTER DETAIL VIEWS

Master Detail View Communications

Master/Detail views are all over Mac applications. In their simplest form, they have a master panel on the left and a detail panel on the right. For example;

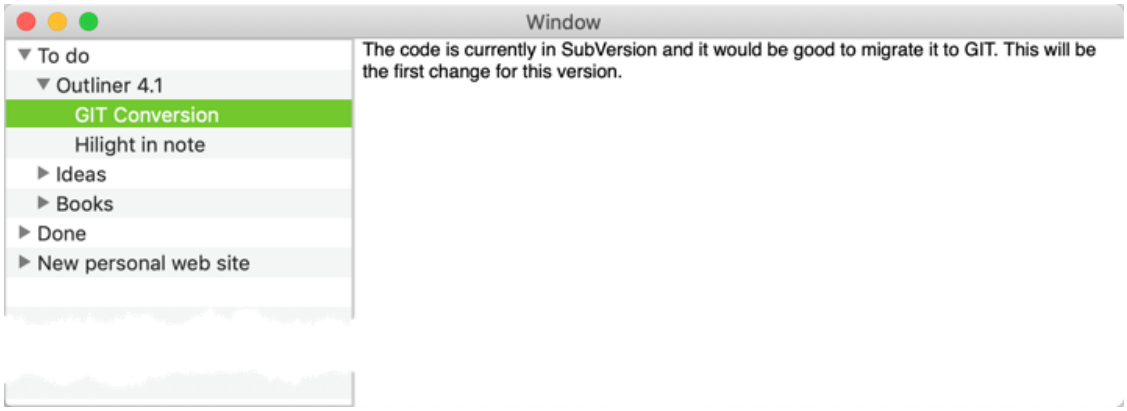


Figure 57: Master/Detail example application

On the left we have an outline control with a list of items in it. When I click on one of the items, the description text is populated on the right. Looking at this in the designer, we see that it is, in fact, three separate views:

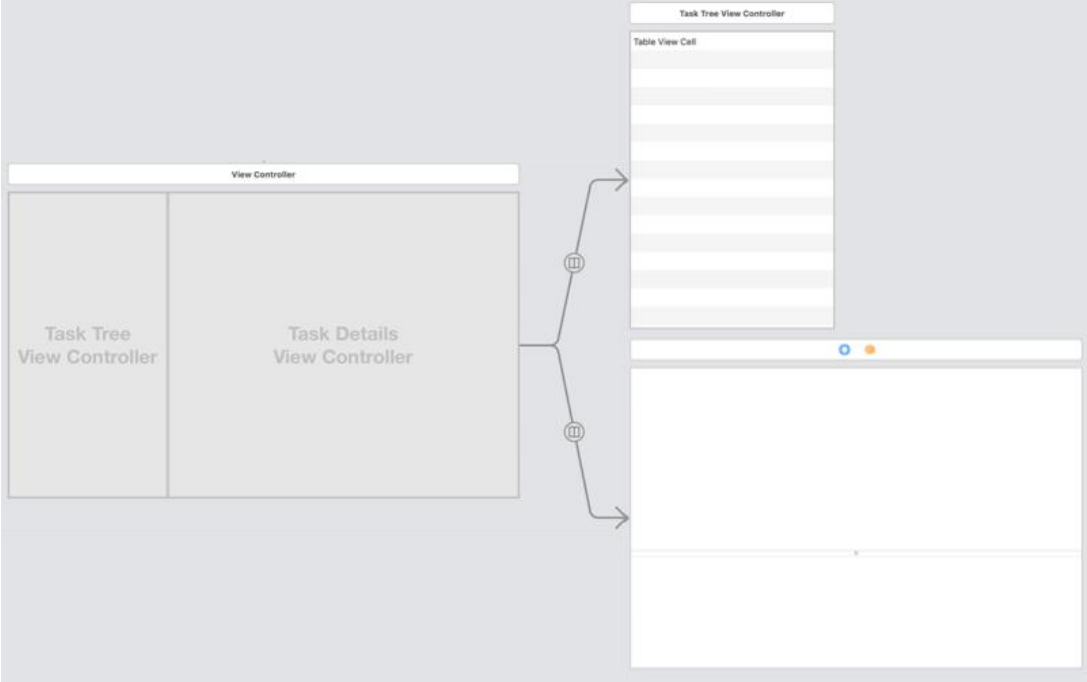


Figure 58: Master/Detail in the designer

The View Controller is a Vertical Split View Controller. This spawns two separate controllers, one for the "master" view and one for the "detail' view. In the image above, the master is the *Task Tree View Controller* and the detail is the *Task Details View Controller*.

The trick we need to pull off is for the master view to tell the detail view when it's selected item has changed so we can populate the details.

Getting the views to communicate

For the purposes of this example, the master view has a controller called TaskTreeViewController and the detail view a controller called TaskDetailsViewController. The master view is an outline control and the details view contains a text view.

When an item is clicked in the outline view, an event is raised to indicate that the selection has changed. This is where we need to be in order to start the notification process.

```
func outlineViewSelectionDidChange(_ notification: Notification) {

    guard let outline = notification.object as? NSOutlineView else {
        return
    }

    let selectedRow = outline.selectedRow
    if let outlineItem = outline.item(atRow: selectedRow)
                as? OutlineItem {
        let associatedNote = outlineItem.notes

        guard let splitVC = parent as? NSSplitViewController
                else { return }

        if let detail = splitVC.children[1]
                    as? TaskDetailsViewController {
            detail.notes = associatedNote
        }
    }
}
```

The first guard is there to ensure that the notification has come from an NSOutlineView. It's not a test that we should have to perform, but it's a case of better safe than sorry. It ensures that the code, if triggered somewhere else, will only respond to the outline view.

Next, we get the selected row from the outline view and cast it to an OutlineItem instance. OutlineItem is the class that I used to populate the rows in the outline view. It has a bunch of properties based on the content of the file I used to build the tree. The specifics of how this class is constructed is irrelevant for this discussion. Just take it that each row in the tree is an instance of an OutlineItem.

Having got the OutlineItem, the text to be displayed in the detail view is extracted to the associatedNode variable.

Now the magic happens,

The next guard statement gets a reference to the parent view of the master view. The parent is going to be the split view. The split view is going to contain an array of view controllers and, because we have precisely two child views, we can assume that view [0] is the master view and view[1] is the detail view.

Just to be sure, we cast view[1] to the TaskDetailsViewController type. Now we have the details view, we can set a property on that view to set the note text.

Over in the TaskDetailsViewController, there is a computed property for the notes:

```
public var notes: String {
    set {
        taskNotesUnformatted.string = newValue
    }

    get {
        return taskNotesUnformatted.string
    }
}
```

In this application, we just assign the text to the *taskNotesUnformatted* string property. taskNotesUnformatted is an NSTextView, to the note gets displayed on screen.

NOTIFICATIONS

Notifications

In modern development, connecting objects in order to allow them to communicate is frowned upon. Rightly so. Creating dependencies between objects in inevitable in any complex application. However, it should always be our aim to keep that kind of tight coupling to a minimum.

That said, there are going to be times when you need to inform one or more parts of your application that something has happened or something has changed or something needs to happen.

To use a trivial example, lets assume you have a day/light theme switching button. When the button is pressed, it switches the entire app between bark and light themes. We can tackle this is many ways including going round every component and changing it's style. I have no problem with that approach but what do you do if you have multiple components on the form that contain nested components. Or, If you have multiple windows open at the same time. How do you ensure that the theme change of communicated to them all?

That's where notifications can sort us out.

Other examples?

Well, you might code a preferences window. The user changes their preferences and saves the changes. At this point, you have two choices, make the user restart the application to get the new settings or broadcast a notification and let the application components re-customise themselves. A much better user experience.

Another thing you might want to consider - you have a process that checks the web for an update. You do this in a separate thread to ensure that the main app is not slowed down by an occasional check that will probably be quite slow. You might run this as a process that runs every 30 minutes. However, what do you do if you find an update? You re likely to want to present the user with the option of downloading the update. Problem is, you're running a separate process on a separate thread and have no idea what your main app is doing. The solution is to broadcast a notification and let the main thread receive the notification and handle it in the best way.

Basic Theory

The basic theory is that the piece of code that needs to let everyone know about a change (the notifier) sends a notification out into the system and assumes that anyone interested will deal with it. Anyone that is interested in that notification (the listener) will listen out for that type of notification and will know what to do with it if it is informed that it has been sent.

The notifier has no idea how many people are listening for its notifications and does not care whether there is anyone listening or not. It sends its notification and forgets about it.

The listener decides what notifications it wants to receive and provides a handler for it. As notifications arrive, they are handled and discarded.

Defining Notifications

Before we can issue or handle a notification, it needs to be defined. The notifier needs to know the name it should use when it sends and the listener needs to know what it is listening for.

We deal with that with a shared struct:

```
struct ThemeNotifications {
    static let ThemeChangeNotification: String = "themeChangeNotification"
}
```

There is only one notification in this struct. You are not, however, limited to one. If you need to raise several types of notification, just add more entries.The part that you need to make unique in your application is the string associated with the constant. In this case, our notification is identified as "themeChangeNotification". In code it will be referred to as

ThemeNotifications.ThemeChangeNotification

Listening for Notifications

Before we get into sending notifications, let's see what we have to do to listen for and handle a notification.

We have three things to do here;

1. Set-up the listener to listen for notifications.
2. Create a handler to deal with a notification when it arrives.
3. Add code to the class to remove the listener who the class is discarded.

Setting up the listener is very straight forward:

```
func initialiseColourTheme() {

 themeObserver = NotificationCenter.default.addObserver(
            forName: Notification.Name(ThemeNotifications.ThemeChangeNotification),
                    object: nil, queue: nil, using: { (_) in

            self.applyThemeChange()
        })
}
```

Yep, it's a one-liner. The NotificationCenter is a static class that exposes a default property that gives us access to the default notification system for our application. We add an observer to this, specifying the name of the notification that we want to listen out for. Ignore the rest of the parameters for now, as we want to keep this simple. Worth noting, you have to define the return from the addObserver call at the class level. We are going to need it later for clean-up purposes. It's defined as

```
private var themeObserver: Any?
```

When a notification comes through of the type we want, the closure will be called. In this case, we call to a handler called applyThemeChange to handle the event.

In it's simplest form, that's just about it.

Now, one last task that I mentioned above. When we add a listener, we really should remove it when the class goes away. That we do in the deinit method:

```
deinit {
    if let observerObject = themeObserver {
        NotificationCenter.default.removeObserver(observerObject)
    }
}
```

This releases the notification handler and ensures that we will not receive any more notifications.

Creating Notifications

Creating a new notification is just as simple as reading on. Simpler in fact;

```
let notification = Notification(name:
Notification.Name(ThemeNotifications.ThemeChangeNotification),
                                object: nil, userInfo: nil)

NotificationCenter.default.post(notification)
```

Create a Notification instance, specifying the name of the notification message and post it to the NotificationCenter. The notification is sent out to annoying listening for it.

Going Further - Passing simple data

Creating and consuming notifications is great in itself. However, how much better would it be if we could pass useful data too?

No problem…

Lets assume we want to pass the old theme name and the new theme name to anyone listening in. We can achieve that with a few lines of simple code:

```
@IBAction func buttonPress(_ sender: NSButton) {

    let oldTheme = "An Old Value"
    let appTheme = "The new theme"

    let userInfo = [
        "oldTheme": oldTheme,
        "newTheme": appTheme
    ]

    let notification = Notification(name:
Notification.Name(ThemeNotifications.ThemeChangeNotification),
                                    object: nil, userInfo: userInfo)

    NotificationCenter.default.post(notification)

}
```

For the purposes of illustration, suspend belief for a few moments and assume there is some code that gets the old and new theme names and assigns them to the oldTheme and newTheme variables. I've set them to specific strings for this exercise. I build these into a string dictionary.

When I create the notification message, I pass this dictionary in the userInfo constructor parameter. Job done!

The other end of the notification is the receiver. This one requires a little more work.

```
func initialiseColourTheme() {

    themeObserver = NotificationCenter.default.addObserver(
        forName: Notification.Name(ThemeNotifications.ThemeChangeNotification),
        object: nil, queue: nil, using: { (userData) in

            if let userInfo = userData.userInfo as NSDictionary? as! [String:String]?
            {
                let msg = NSAlert()
                msg.messageText = userInfo["oldTheme"] ?? "No old theme"
                msg.informativeText = userInfo["newTheme"] ?? "No new theme"
                msg.runModal()
            }

    })
}
```

First thing to note here is that I have changed the anonymous parameter to the closure to a named variable; userData. The userData we receive is of type Notification. It contains:

```
name = themeChangeNotification,
object = nil,
userInfo = Optional([AnyHashable("oldTheme"): "An Old Value",
        AnyHashable("newTheme"): "The new theme"])
```

The data we sent through as a string dictionary has been converted to an optional dictionary of type [AnyHashtable:Any]. Ok, that's no big deal, but we send a string dictionary, so we convert it back to a string dictionary:

```
if let userInfo = userData.userInfo as NSDictionary? as! [String:String]?
```

Then we can access our data as we would any dictionary.

Going Further - Passing more complex data

Your data more complex? Still not a problem.

Lets define a struct with some more complicated data:

```
struct sharedObject {
    var Property1 : String
    var Property2 : Int
    var Property3 : Decimal
}
```

We have three properties of various types collected together into a simple struct. To send this through, we extend our Notification object:

```
let mySharedObject = sharedObject(Property1: "My Object",
                                  Property2: 1,
                                  Property3: 2.3)

let notification = Notification(name:
Notification.Name(ThemeNotifications.ThemeChangeNotification),
                                object: mySharedObject,
                                userInfo: userInfo)

NotificationCenter.default.post(notification)
```

In our receiver object, the Notification we are sent now contains the object we sent too:

```
name = themeChangeNotification,
object = Optional(TestMacApp.sharedObject(Property1: "My Object",
        Property2: 1,
        Property3: 2.3)),
userInfo = Optional([AnyHashable("newTheme"): "The new theme",
        AnyHashable("oldTheme"): "An Old Value"])
```

As you would expect, this comes through as an optional, but we can unwrap it easily:

```
if let userDataObject = userData.object as? sharedObject {
    print(userDataObject)
}
```

Chapter Four

Controls

Houehouhowheoiweh owihj owij oi jwoije woiej woij oiej woiej wo. Houehouhowheoiweh owihj owij oi jwoije woiej woij oiej woiej wo. Houehouhowheoiweh owihj owij oi jwoije woiej woij oiej woiej wo. Houehouhowheoiweh owihj owij oi jwoije woiej woij oiej woiej wo.
Houehouhowheoiweh owihj owij oi jwoije woiej woij oiej woiej wo. Houehouhowheoiweh owihj owij oi jwoije woiej woij oiej woiej wo. Houehouhowheoiweh owihj owij oi jwoije woiej woij oiej woiej wo. Houehouhowheoiweh owihj owij oi jwoije woiej woij oiej woiej wo.
Houehouhowheoiweh owihj owij oi jwoije woiej woij oiej woiej wo. Houehouhowheoiweh owihj owij oi jwoije woiej woij oiej woiej wo.

NSOutline

NSOutline is very, very useful when trying to present a hierarchy of related data. Equally, the number of tutorials on how to use it can be counted on the fingers of one hand and they always seem to limit their hierarchies to a root node with one child node. Not very realistic.

What I present here is an outline view with any number of levels of the same kind of data. More like the file and folder layout you might see in ***Finder***. I will introduce an XML file format briefly as this is the file format that I used in my real-world app to populate the NSOutline control. But I will try not to dwell on it.

Basic Outlining

Chapter Five

Bits and Pieces

When you're new to Mac development, there are lots and lots of little things that you learn you need to do. You learn them by falling over little issues and constantly asking yourself "why isn't this the default behaviour?"

In this section, I hope to present a set of small but useful topics that address issues that are quick and easy to deal with, often with a very small amount of code.

TIPS

Closing your application

When you start an application, a window opens and a menu appears at the top of the screen. Behind the scenes, the menu has been built at the application level and the initial window is constructed and displayed for you.

In Windows I don't expect the application to close when the last window is closed. Oddly enough, I do expect it to on the Mac. Sadly, that takes a bit of coding to achieve. Not much, but a little.

In Windows, multi-document windows are held in a container, so it's obvious when there are no child windows open. On the Mac, the child windows are floating in amongst the rest of the open windows, so it is way less obvious when you close the last window. That tends to lead to me having loads of apps running but few open windows.

For my own apps, I want the app to terminate when you close the last window. Ok, there are going to one times when I don't but most times I will. This is especially important if I am not intending to support multiple concurrent open windows. In this environment, closing the window closely equates to closing the application, so that's what I want to happen.

To achieve it, I just need to include one function in the app.delegate. It goes by the subtle name of:

applicationShouldTerminateAfterLastWindowClosed

Do you think they get paid by the letter?

```
@NSApplicationMain
class AppDelegate: NSObject, NSApplicationDelegate {

    func applicationShouldTerminateAfterLastWindowClosed(_ sender: NSApplication) ->
Bool {
        return true
    }
}
```

Simple as that.

Dragging a Window

This tip came up when I was working on the About Box I presented elsewhere. I wanted a window without a title bar that I could still drag around the screen. It's not often useful but you could, for example, create a game where the title bar is a distraction. Ok, a but contrived, but it's still useful to know the technique.

Simply add code to the windowDidLoad and you're done:

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

    if let window = window {
        window.isMovableByWindowBackground  = true
    }
}
```

Holding the mouse down and dragging will cause the window to drag around the screen.

Window Position

I always find it a little irritating when I close and reopen an application to find that the main window opens in some default position with a default size and I am forced to manually resize it. Then, next time I open the application, it's reverted again.

Given how easy it is to do, all applications should save the window size and position.

To do that, just set the frame autosave name:

```
class MainWindowController: NSWindowController, NSWindowDelegate {

    override func windowDidLoad() {
        super.windowDidLoad()

        if let window = self.window {
            window.setFrameAutosaveName(
NSWindow.FrameAutosaveName("mainWindowSave"))
        }
    }
}
```

Just by setting the name in the window controller, the runtime will save the size and position of your window and will restore it the next time the window is opened. Simple as that and saves a great deal of irritation.

It's also worth noting that this isn't reserved for the main window only. If you have other windows you open, you can give them a unique name too and have them managed by the Mac OS. The window position will be saved to the application User Defaults.

Chapter Six

Files

Lots of what we do involves reading and writing files, parsing the content on read or formatting the output on write. JSON is a very popular file format that has built in support in SWIFT/MacOS. However, there are lots of other file formats and we will examine at least one of these in this section.

XML PARSING

Parsing an XML file

Mess around on the internet and you're going to come across JSON and XML files at some stage. Mess around with files from a Windows system and it's likely it will be XML. I will assume you have a vague idea of what an XML file looks like.

In this section, I would like to present an *Outliner* file, as used in a sample application that loads an XML file into an NSOutline control. The file format I am dealing with is very simple.

Basic File Structure

Our file is sectioned into two sections, one for options and one for a hierarchy of data elements.

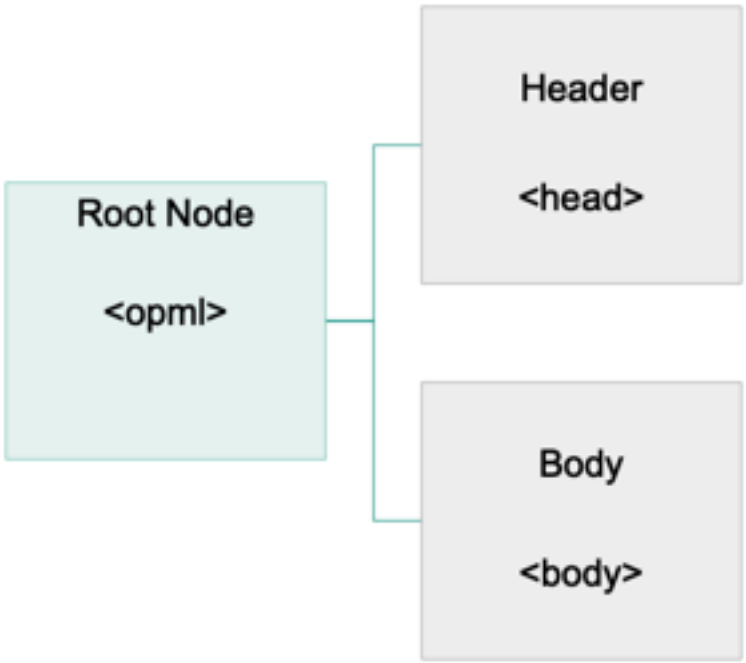


Figure 59: File header information

There can only be one top most root node. In out file structure, this will be the **<opml>** node. The opal file is split into two sections, the first being the **<head>** section where we put any file specific options and the second being the **<body>** section where we save the content of the file.

An example of the head structure is:

```
<head>
 <title>Project Title</title>
 <expansionState>0,1,5</expansionState>
</head>
```

So the head can have two child nodes, once called **<title>** and one called **<expansionState>**. The title is required and the expansion state is optional.

The body is more complex while being very simple. It contains one type of element called **<outline>**. It's a simple section in that there is only the one element type. It is complicated because an outline element has multiple attributes and can contain nested outline items. It is the nesting that gives us our structure:

```
 1.   <body>
 2.     <outline text='Credit Interface' _note='This is our note'
_status='indeterminate'>
 3.       <outline text='Code Transfer' _note='#Child item note'>
 4.         <outline text='Test 1'>
 5.            <outline text='Test 2' />
 6.            <outline text='Test 3' />
 7.         </outline>

 8.         <outline text='Test 1a'>
 9.            <outline text='Test 1a child' />
10.            <outline text='Test 1a child2' />
11.         </outline>
12.       </outline>
```

This is a small fragment of the file structure. All of our outline elements exist under the body element. The first item (Line 2) is the *Credit Interface* item. It has a child item below it (Line 3) called *Code Transfer* and Code Transfer has two child elements called *Test 1* and *Test 1a*. Test 1 has two child nodes called called *Test 2* and *Test 3*. Test 1a similarly has two child nodes called *Test 1a child* and *Test 1a child2*.

There is a lot more to the file, but is's all repetition of this simple structure.

An outline element has attributes to define its content;

1. text defines a short name for the item. It is mandatory.
2. _notes defines bulk text. The text to be displayed in our details view.
3. _status defines the status the node.

It's a very simple element.

Lets create some objects

Now we have a feel for the file format, it makes sense to create some helper objects. As a minimum, I would expect there to be

- an object to wrap the file itself,
- an object to hold the options in the head,
- an object to map the outline items into.

I'm going to throw a fourth class into the mix that I'm going to call *NodeHelpers*. A lot of what we do to the XML file will be repeated many times and it makes sense to group these together into a helper class.

So many classes, where to start?

NodeHelpers

It's not very TDD, but I'm going to start with the *NodeHelpers* class. Based on the file structure, I know for certain that I'm going to need at least two helper methods, so lets get them in place first. We'll start with an empty class:

```
class NodeHelpers {

}
```

From the file format, we can clearly see that a lot of our data is encoded as attributes of a node, so we'll add a class that extracts the attributes given an XML Element.

```
public static func loadAttributes(fromElement element: XMLElement) -> [String:String]
{

    var attributeList = [String:String]()

    if let attributes = element.attributes {
        for attribute in attributes {
            if let attributeName = attribute.name,
                let attributeValue = attribute.stringValue {

                attributeList[attributeName] = attributeValue
            }
        }
    }

    return attributeList
}
```

What we're going to do in this code is create a dictionary of key/value pairs where the attribute name is the key and the value is the string value from the XML Element we have been passed. If there are no attributes, we'll return an empty dictionary. It's up for debate whether the code should throw an error, but an empty dictionary just makes sense in this circumstance.

The other method I know I am going to need is some way of extracting the text value from an element. Our header options are all specified as elements with a text component, so we'll definitely use it there:

```
public static func getStringValue(fromNode: XMLElement, forName: String,
                usingDefault: String = "") -> String {

    let node = fromNode.elements(forName: forName)
    if node.count == 0 { return usingDefault }

    return node[0].stringValue ?? usingDefault
}
```

You'll note that the functions are static. That's just a convenience thing since we don't need to store state.

One last function I know I'll need… the file is split into two sections; one for the header and one for the body. I want a quick and easy way to extract there two sections for further processing, so I'll have a method for extracting them.

In my case, I know they're going to be direct descendants of the root node, so I could go with a simple process of checking the elements below the root for any with the name we are looking for. The search would only pull out an element that is a direct descendent, so we could be happy that we're not going to get anything below the first level.

```
public static func getFirstChildNode(fromRoot: XMLElement, forName: String) ->
XMLElement? {

    let elements = fromRoot.elements(forName: forName)
    return elements.count > 0 ? elements[0] : nil
}
```

That's ok for this code, but I want something a little more generic. What happens, for example, if the file structure changes or I need a way to get at an option directly or a new section is defined. I should not assume that what I have today will necessarily be there tomorrow and a utility method that is tied to the structure of a file is plain wrong. Utility methods should not be tied to file implementation details.

So, as an alternative, I have a nice little method that takes an xPath definition and returns the first matching element.

```
public static func getFirstNode(fromDocument: XMLDocument, forPath: String) ->
XMLElement? {

    do {
        let elements = try fromDocument.nodes(forXPath: forPath)
        if elements.count == 0 {
            return nil
        }

        return elements[0] as? XMLElement
    } catch {
        return nil
    }
}
```

It's only very slightly more complex than the previous version, but it's a lot more flexible. Because we have no idea where in the document we are intending to look, I pass in the loaded XML document. *forPath* is any xPath statement. For my limited purposes, this will be a simple path from the root. Yours may be more complicated.

Nodes.forXPath returns a list of XMLNodes rather than XMLElements. It's more generic, but we specifically want XMLElements, so I try to cast the result to an XMLElement. If it can't be cast, I will get nil back, which is Ok. Similarly, if the xPath is invalid, it will error and I will return nil again.

The return of nil should be an indictor to the caller of some kind of error. There is nothing that my utility class can do to fix problems, is it is pointless it trying. The caller can interpret the null return.

And that concludes the node helper and it's three helper methods;

- loadAttributes
- getStringValue
- getFirstNode

Header Class

My next class is, oddly enough, going to be another one that we can't use straight away. It's going to hold the data from the header section of the file. These are the options that we load before everything else and that determine the attributes of the file. In the sample, we have two options; both optional:

```
<head>
  <title>Project Title</title>
  <expansionState>0,1,5</expansionState>
</head>
```

As before, lets start with an empty class.

```
import Foundation

class OutlineHeader {
}
```

From what we know, there are going to be two options; the title of the file and the expansion state of elements. So we'll create two place holders for these values.

```
class OutlineHeader {
    public var title: String = ""
    public var expansionState: String = ""
}
```

At some stage that expansionState needs to be expanded out as we happen to know it's an array of integers. That's for another set of notes! So the remaining task is to get this object populated.

I'm a great fan of objects populating themselves, so I'm going to create a class initialisation function that is passed the XML document to extract whatever it needs. This lets the header be responsible for it's own state and makes it easier if I need to add more items later or if things get moved around.

```
class OutlineHeader {
    public var title: String = ""
    public var expansionState: String = ""

    init(fromDocument doc: XMLDocument) {
        guard let headerNode = NodeHelpers.getFirstNode(fromDocument: doc,
                forPath: "/opml/head") else { return }

        self.title = NodeHelpers.getStringValue(fromNode: headerNode, forName:
"title")
        self.expansionState = NodeHelpers.getStringValue(fromNode: headerNode,
forName: "expansionState")
    }
}
```

We're passed in the entire XML document and we use the *NodeHelprs* class to extract the head node and then again to extract data in the title and expansionState child elements.

And that's the entire header class.

Body Wrapper

The body of our document is a hierarchy of OutlineItem elements.

```
   <body>
     <outline text='Credit Interface' _note='This is our note' _status='indeterminate'>
       <outline text='Code Transfer' _note='#Child item note'>
         <outline text='Test 1'>
            <outline text='Test 2' />
            <outline text='Test 3' />
         </outline>

         <outline text='Test 1a'>
            <outline text='Test 1a child' />
            <outline text='Test 1a child2' />
         </outline>
       </outline>
```

Since we created. Class for the header, it seems sensible to have a class for the body too. It will be the wrapper around our outline structure and will contain the top level node(s). It will be responsible for finding the *body* node in the document and loading the first node that will be the container for all the child nodes.

The entire class consists of:

```
class OutlineBody {

    var outlineBody: OutlineItem?

    init(fromDocument doc: XMLDocument) {

        guard let bodyNode = NodeHelpers.getFirstNode(fromDocument: doc,
                forPath: "/opml/body") else { return }

        outlineBody = OutlineItem(fromOutlineNode: bodyNode, withParent: nil)
    }
}
```

Our input is the XML document we loaded previously. From that, we extract the *body* node. This is the top most node in the file and it's immediate children will be OutlineItems. So we

create our first OutlineItem and pass it the *body* node. This will start the cascade loading of the outline.

Outline Items

This is the class where the bulk our data is going to be loaded and where the bulk of the structure loading will take place. It's an important class!

From the previous discussion, you will recall that an outline item consists of an outline element and a bunch of attributes:

<outline text='Credit Interface' _note='This is our note' _status='indeterminate'>

So, the first thing we're going to need is a class to encapsulate the element data:

```
class OutlineItem: CustomStringConvertible {

    var text : String = ""
    var notes: String = ""
    var status: String = ""
    var completed: Bool = false
```

The *completed* field isn't strictly part of the data file, but is derived from the status, so I'm counting it in the base data.

Next, we need some supporting fields:

```
    var parent: OutlineItem?
    var children: [OutlineItem] = []
    var attributes: [String: String] = [:]
```

When we construct the outline item, we keep a reference to our parent element. It's not strictly necessary but it's a useful piece of information to have when we have an outline node and want to navigate up the hierarchy. It's an optional because the top most item in our hierarchy will not have a parent.

Any outline item may have one or more outline items below it in the hierarchy, so we have an array of child outline items. The relative position of the siblings is given by their position in the *children* array.

Finally, the input file is defined by attributes which are key/value pairs, where the value may be omitted. Since we know what attributes we want to process, we could just extract the ones we want and save the data. However, if new attributes are added at some later date, this would lose them. So the approach I'm taking here is to extract all the attributes, use the ones I need and store the rest for when I write the object out again. I lose nothing that way.

I now need a couple of helper computed properties. I defined my class as implementing CustomStringConvertible, so I am required to have a description property. I supplement that with a property to determine whether there are child nodes for this item:

```
    var hasChildren: Bool { get { return children.count != 0 }}

    var description: String { get { return text } }
```

Our next task is to get the outline item initialised. That'll be the constructors function:

```
init(fromOutlineNode: XMLElement, withParent: OutlineItem?) {
    self.parent = withParent

    populateFromAttributes(fromElement: fromOutlineNode)
    loadChildren(fromElement: fromOutlineNode)
}
```

Our constructor is passed the parent element from the XML file and the parent OutlineItem reference so we can save a pointer to our parent in the (you guessed it) *parent* variable.

At this point we have two tasks to perform;

- Load the current outline items' data.
- Populate the list of child items.

The method called populateFromAttributes is responsible for loading the attributes and extracting the data we want quick access to:

```
private func populateFromAttributes(fromElement: XMLElement) {

    self.attributes = NodeHelpers.loadAttributes(fromElement: fromElement)

    if let text = attributes["text"] {
        self.text = text
    }

    if let notes = attributes["_note"] {
        self.notes = notes
    }

    if let checked = attributes["_status"] {
        self.status = checked
        self.completed = checked == "checked"
    }
}
```

From an XML point of view, the important call here is the loadAttributes call we make to our NodeHelpers class. As you will recall this extracts the attributes from the element.attributes property and converts them to a string dictionary. We will get all attributes whether we want them or not so we can save them all later.

The following three blocks are just there to extract the three fields that we want.

Once we have populated our own outline item, the next job is to load the child elements, if there are any. This turns out to be very simple:

```
private func loadChildren(fromElement parentNode: XMLElement) {

    let childNodes = parentNode.elements(forName: "outline")
    if childNodes.count == 0 { return }

    // Load the child nodes if there are any.
    for node in childNodes {
        let childNode = OutlineItem(fromOutlineNode: node, withParent: self)
        children.append(childNode)
    }
}
```

First job is to interrogate the current XMLElement and see if it has any child elements. If there are none, we return without any further work. The array of children will have been initialised with no child nodes, so we are in a fit state to return.

If we have child elements, then we drop into loop to construct a child OutlineItem class for each child element. We add these to the array of child elements, establishing the order that they should appear in to be the order they are in the file. As part of the construction, we pass a reference to the current element, so each child can set their parent.

Neat and simple and we end up with a hierarchy of OutlineItem objects that matches the content of the XML file.

Wrapping It All Up

So, we now have classes that

- NodeHelpers - Provides for some re-use when parsing our XML file. There are several things we need to do when parsing any XML file and concentrating them in the NodeHelpers class.
- OutlineHeader - Our XML file is in two parts, one of which is a set of options. The header class parses out those helpers.
- OutlineBody - The second part of our file consists of a hierarchy of items. Unfortunately, there can be many 'top level' items below the body node, so we create an interim element to encapsulate the body node. This is the root of our hierarchy.
- OutlineItem - defines the hierarchy of items. Each item is responsible for extracting the attributes it needs from the element in the file and for populating it's children. As a helper for using the structure, we added a reference to the owning item in the parent property.

The only thing we are missing now is a mechanism to load the file and populate the header and body nodes.

The basis of our class is:

```
class OpmlFile {

    private var fileUrl: URL

    public var outline: OutlineBody?
    public var header: OutlineHeader?
```

This will give us variable to store the URL of the file we are loading and two placeholders for the header and body nodes. Our class can then be initialised using:

```
init(fromUrl: URL) {
    fileUrl = fromUrl

    guard let doc = loadDocument(url: fromUrl) else {
        // TODO: Log the error.
        return
    }

    self.header = OutlineHeader(fromDocument: doc)
    self.outline = OutlineBody(fromDocument: doc)
}
```

So, we save the URL of the file and then load the XML. All sorts of things can go wrong with loading a file, so we wrap it in a method and guard that we get a result. If we don't get an XMLDocument, then we drop out uninitialised.

After that, it's just a case of creating the header and body nodes. We delegated their initialisation to the nodes themselves, so we don't need to do anything else here.

The only method we need to cover here is the loadDocument method.

```
private func loadDocument(url: URL) -> XMLDocument? {

    let options = XMLNode.Options()
    return try? XMLDocument(contentsOf: url, options: options)
}
```

It doesn't get much simpler than that. Ok, to be honest, I should have lots of error trapping around the creation of the XMLDocument and I should have lots of logging and I should be making sure the user is informed of a load error. But this is a quick run through of parsing an XML file and all that stuff isn't.

Code recap

So, to recap the code…

```
class OutlineHeader {
    public var title: String = ""
    public var expansionState: String = ""

    init(fromDocument doc: XMLDocument) {
        guard let headerNode = NodeHelpers.getFirstNode(fromDocument: doc, forPath:
"/opml/head") else { return }

        self.title = NodeHelpers.getStringValue(fromNode: headerNode, forName:
"title")
        self.expansionState = NodeHelpers.getStringValue(fromNode: headerNode,
forName: "expansionState")
    }
}
```

The header class to contain the options

```
class OutlineBody {

    var outlineBody: OutlineItem?

    init(fromDocument doc: XMLDocument) {

        guard let bodyNode = NodeHelpers.getFirstNode(fromDocument: doc,
                forPath: "/opml/body") else { return }

        outlineBody = OutlineItem(fromOutlineNode: bodyNode, withParent: nil)
    }
}
```

The body class encapsulating the item structure

```
class OutlineItem: CustomStringConvertible {

    var text : String = ""
    var notes: String = ""
    var status: String = ""
    var completed: Bool = false

    var parent: OutlineItem?
    var children: [OutlineItem] = []
    var attributes: [String: String] = [:]

    var hasChildren: Bool { get { return children.count != 0 }}

    var description: String { get { return text } }

    init(fromOutlineNode: XMLElement, withParent: OutlineItem?) {
        self.parent = withParent

        populateFromAttributes(fromElement: fromOutlineNode)
        loadChildren(fromElement: fromOutlineNode)
    }

    private func loadChildren(fromElement parentNode: XMLElement) {

        let childNodes = parentNode.elements(forName: "outline")
        if childNodes.count == 0 { return }

        // Load the child nodes if there are any.
        for node in childNodes {
            let childNode = OutlineItem(fromOutlineNode: node, withParent: self)
            children.append(childNode)
        }
    }

    private func populateFromAttributes(fromElement: XMLElement) {

        self.attributes = NodeHelpers.loadAttributes(fromElement: fromElement)

        if let text = attributes["text"] {
            self.text = text
        }

        if let notes = attributes["_note"] {
            self.notes = notes
        }

        if let checked = attributes["_status"] {
            self.status = checked
            self.completed = checked == "checked"
        }
    }
}
```

Our most complex class encapsulating our outline data

```
class OpmlFile {

    private var fileUrl: URL

    public var outline: OutlineBody?
    public var header: OutlineHeader?

    init(fromUrl: URL) {
        fileUrl = fromUrl

        guard let doc = loadDocument(url: fromUrl) else {
            // TODO: Log the error.
            return
        }

        self.header = OutlineHeader(fromDocument: doc)
        self.outline = OutlineBody(fromDocument: doc)
    }

    private func loadDocument(url: URL) -> XMLDocument? {

        let options = XMLNode.Options()
        return try? XMLDocument(contentsOf: url, options: options)
    }
}
```

The wrapper round the entire file and out main interface

These classes represent everything we need to load our file. We've kept things simple and tried to make each class responsible for maintaining it's own data. This has allowed us to create a simple class for sharing the load of handling the XML functions:

```
class NodeHelpers {

    public static func loadAttributes(fromElement element: XMLElement) ->
[String:String] {

        var attributeList = [String:String]()

        if let attributes = element.attributes {
            for attribute in attributes {
                if let attributeName = attribute.name,
                    let attributeValue = attribute.stringValue {

                    attributeList[attributeName] = attributeValue
                }
            }
        }

        return attributeList
    }

    public static func getStringValue(fromNode: XMLElement, forName: String,
            usingDefault: String = "") -> String {

        let node = fromNode.elements(forName: forName)
        if node.count == 0 { return usingDefault }

        return node[0].stringValue ?? usingDefault
    }

    public static func getFirstNode(fromDocument: XMLDocument, forPath: String)
            -> XMLElement? {

        do {
            let elements = try fromDocument.nodes(forXPath: forPath)
            if elements.count == 0 {
                return nil
            }

            return elements[0] as? XMLElement
        } catch {
            print(error)
            return nil
        }
    }
}
```

Our XML helper class for extracting data

Chapter Seven

Tooling

I'm a fan of tooling where it adds benefit or just plain makes my life easier. As a result, I use a small amount of tooling but it's always worthwhile.

This section is about providing some pointers to that tooling and how to use it.

TOOLING

Checking code syntax - Swift Lint

I'm not a great fan of lint programs. Most of the ones I have had experience of have been painful to use. They tend to over-analyse your code, to pick out every trivial style issue whether it has a significant impact on the code or not. Technically, that is what they are for. Just most of them seem to go so over the top that you end up missing the issues in a sea of trivia.

Swift has its own lint program. And I thought I should give it a go. Glad I did.

Install

There is a GitHub repo with the details, so that's a good place to start. You'll find that at Swiftlint on GitHub. Installing is a one-off task and consists of running:

```
brew install swiftlint
```

That gets it on to your machine. Now, you can run it as a command line and that's what I've seen the most. It's fine to do that, but it gives you a get out of jail free card by letting you ignore it or ignore the issues it raises. A better approach is to get it to run every time you build your project. Its a great incentive to fix the issues rather than pretend they're not there.

Integrate into Xcode

Integrating it into your project is as simple as adding a new script action:

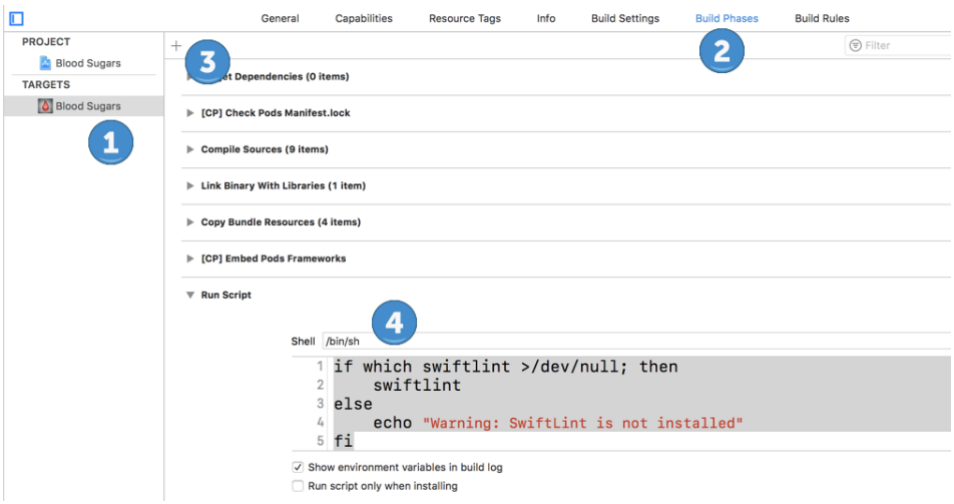


Figure 60: Adding the SwiftLint script

1. Select the target.
2. Select the Build Phases tab.
3. Click the + and add a run script phase.
4. Add the script.

The code of the script is

```
if which swiftlint >/dev/null; then
    swiftlint
else
    echo "Warning: SwiftLint is not installed"
fi
```

Now, every time you build, it will run Swiftlint against that project. If you have multiple projects within the workspace, then you'll need to add the script to each of the ones you want checked. This is pretty useful when you have a load of dependencies introduced with the likes of Cocoapods, where third party code is added to your workspace.

Configuring

Ok, so not every test is a problem and not every setting will meet your requirements. Swiftlint is configurable to allow you to disable some tests, introduce some disabled ones or adjust the parameters of enabled tests. Config is stored in a file called .swiftlint.yml in the same folder as your project. Watch out! That file name starts with a full-stop.

I prefer not to change too many of the rules, so my default yml only has two overrides:

```
line_length: 160
vertical_parameter_alignment: disabled
```

The default for line length is stupidly short, so I give myself some leeway. The rule for vertical parameter alignment is just an irritation to me. I like to line up continuation lines my own way, so I have to turn this off.

Automated cleanup

There are some rules that feel trivial and that you propagate without thinking of them too much. Luckily, some of them are fixable with Swiftlint itself.

To do that, open the terminal at the folder where the project is in stalled and type

```
swiftlint autocorrect
```

It'll go through and fix a load of simple issues for you automatically. A very useful thing to do when you first set-up swiftlint in the project.

Logging

I come from a world where there are a large number of tools to help you with debugging your programs. When I got to Xcode and Swift, I was presented with LLDB (for which I still know very few commands) and using print statements. Not ideal.

To be fair, nothing much has changed from those early debugging days, except I now have a much snazzier form of printing to the console in the form of SwiftyBeaver.

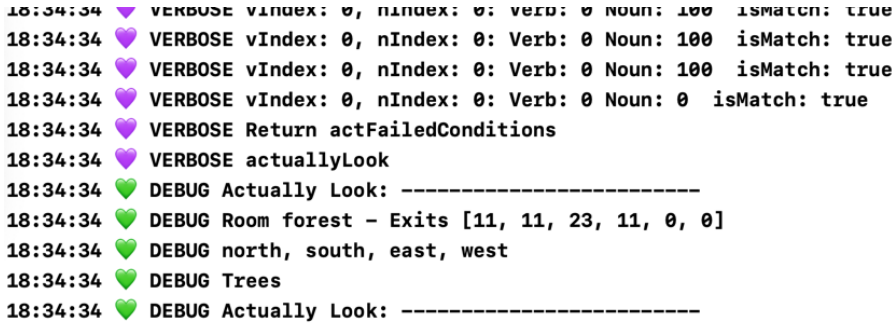


Figure 61: Logging output

You can get install instructions over at github for SwiftyBeaver. For my part., I prefer using Carthage for my install, but you can pick some other way if you want.

Set-up

There are two phases to using SwiftyBeaver; the set-up of the environment and the logging to the console. To get the set-up done, you need to go over to the AppDelegate.swift file. For the purposes of our logging, we're going to log to the console. You can log to all sorts of pother locations, but the console is the simplest place for debugging purposes.

So, first thing we need to do is reference the library and create a global logging variable:

```
import SwiftyBeaver
let log = SwiftyBeaver.self
```

The log variable is a convenience, but a very convenient one, so it's worth the ridicule of having a global variable!

The next bit of set-up, I usually split into a separate method:

```
fileprivate func initialiseConsoleLogging() {
    let console = ConsoleDestination()  // log to Xcode Console

    // Set a custom format to get date/time and add some colour
    // indicators to the log to make spotting message level easier.
    console.format = "$DHH:mm:ss$d $L $M"
    console.levelString.verbose = "💜 VERBOSE"
    console.levelString.debug = "💙 DEBUG"
    console.levelString.info = "💚 INFO"
    console.levelString.warning = "💛 WARNING"
    console.levelString.error = "❤ ERROR"

    // Setting the level here determines how much output we get
    console.minLevel = .verbose

    // add the destinations to SwiftyBeaver
    log.addDestination(console)
    log.info("App started and log initialised.")
}
```

What we are doing here is defining the console as a destination and configuring the output. The format determines how the output will look while the levelString is used to add some colour to the console which makes it easier to spot messages.

The minlevel tells SwiftyBeaver what level to report at. While intensively debugging, this is usually set to .verbose as this gives me the maximum number of messages. Once the app settles down, it drops to .debug.

For production, you might as well set it to .error, since you cannot get the console output anyway. For production, you might want to look at the paid options!

Logging

Logging is the second part of the process. That is simply a case of lobbing calls to the log object throughout your code.

```
let item = items.filter() {
    $0.ItemName.uppercased() == noun && $0.Location == location
}

log.debug("Auto get for \(noun) returned \(item.count) results.")
```

log provides methods for outputting a string at various levels. Here I have used the debug level which will show provided the minLevel is .debug or higher (.verbose). For the harder to debug issues, I might use verbose level debugging:

```
if tmp != 0 {
    vIndex = VERB_GO
    nIndex = tmp
    log.verbose("Go verb found. Direction is \(verb) which translates to \(tmp)")
}
```

The level at which you log is entirely up to you. I tend to find I can stabilise my programs at the debug and verbose levels during development. For production running, it might make more sense to log to a file at a higher level.