A Mac Development Project Template

By
Steve Barnett

A Mac Development Template Project

By
Steve Barnett

Contents

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

Introduction

One of the really nice things about Xcode development is that you can, with a couple of clicks, create a complete working app. Being able to get the basics in place so quickly is great. Unfortunately the apps that get generated are so basic that they are practically useless. Worse, the template for the Mac app is missing several key items to make the app work properly.

The purpose of this document is to run through a simple set of customisations that need to be done to a project to make it work. We then expand this out to define a series of changes that build on these changes to give us a template project with a good deal of functionality on tap ready to use.

YAGNI

One point I should make early on. There is a concept in development called YAGNI. It stands for You Ain't Going to Need It. The basic principle is that you don't write a piece of code until you actually need to use it. It's a way of ensuring that your app only contains code that you need and you don't end up with unused code polluting the apps code.

I think this principle is great and I largely agree that it has merit.

As I have developed applications over the years I have also come to the conclusion that I re-write the same basic chunks of code time and time again. Pretty much every app I have created, whether iOS, MacOS, Windows or anything else, has settings and needs a way to manage them. They all also have an "about" screen of some sort giving the user access to program version information and support links.

I could code these every single time from scratch. I prefer to have a template with the boiler-plate code already in place. It kind of breaks YAGNI but I prefer to think that it's allowed because I know I'm going to need it!

The Default Project

As a starting point, fire up Xcode and create a MacOS app. It'll give us a chance to see what problems we get from the default and how easily they are fixed.

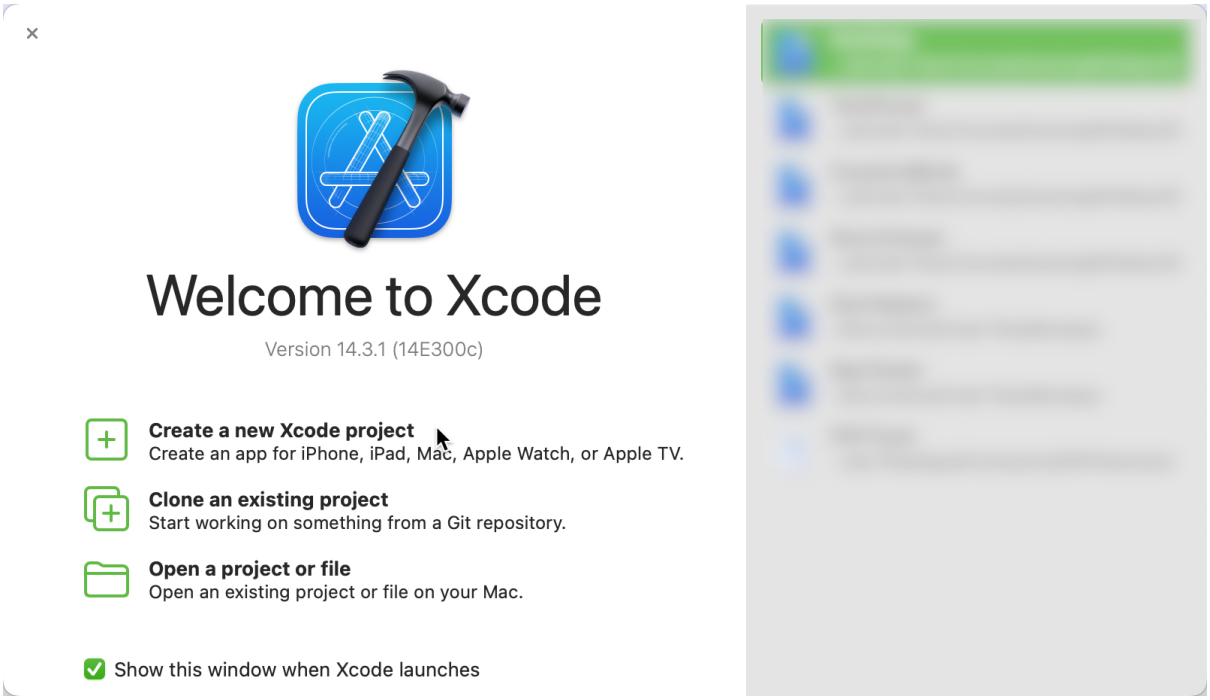


Figure 1: Xcode Welcome Window

Select the option to Create a new Xcode project.

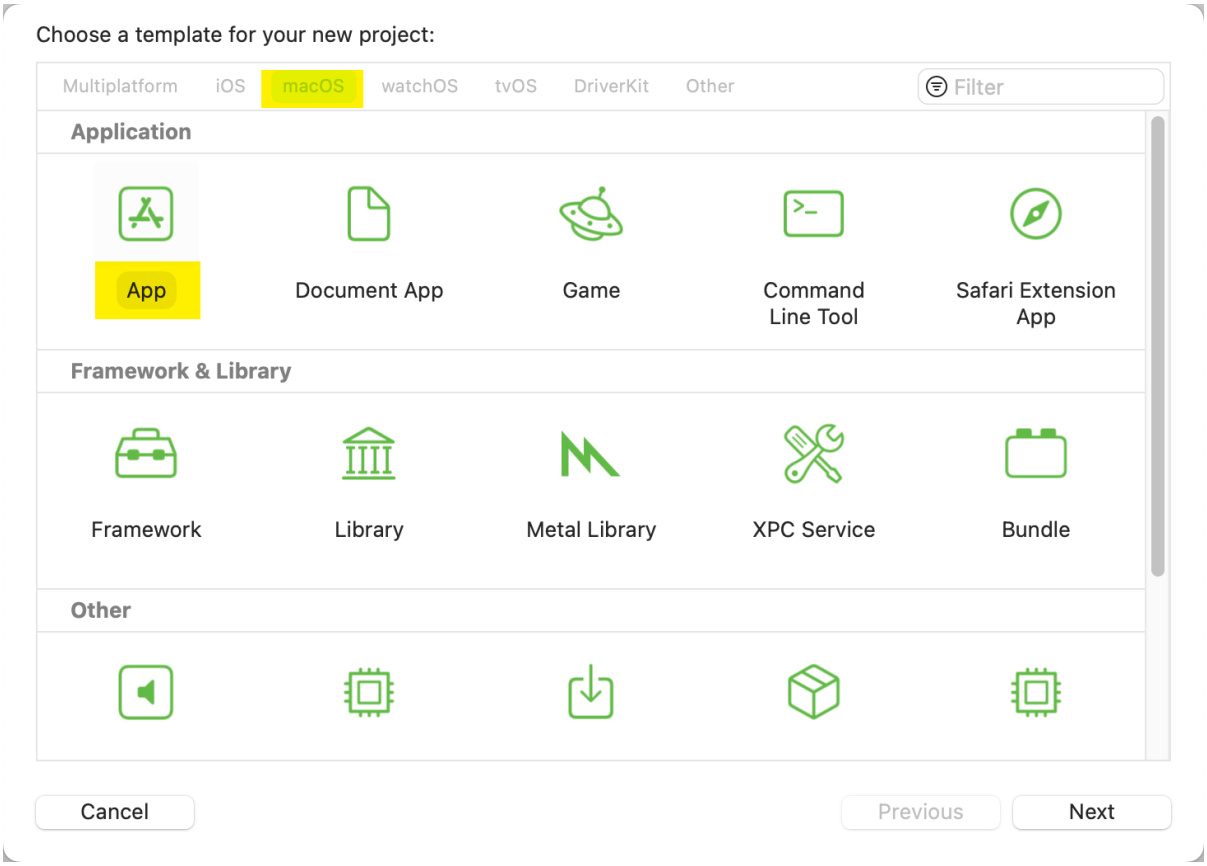


Figure 2: Select an Xcode template

From the template selection window, select the *macOS* tab and select the *App* icon, then press *Next*. Xcode will prompt you to set your project properties. These are going to identify your project and define a bunch of internal names such as the bundle name.

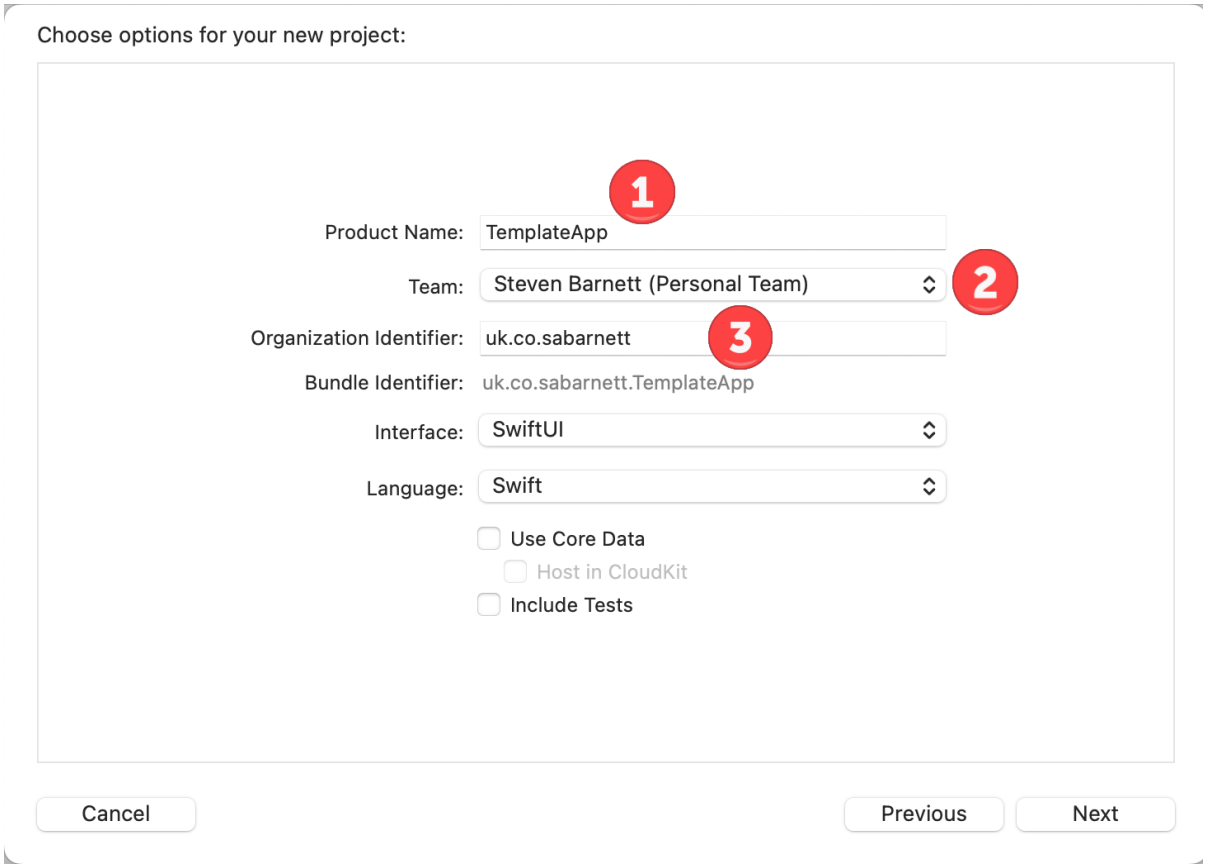


Figure 3: Xcode project properties

You're going to give your project a product name (1), select your team name (2) to allow the project to compile cleanly and specify the organisation identifier (3) which is typically your domain name reversed. The ID needs to be unique if your app is going to compile and run properly, so the combination of product name and organisation identifier need to result in a unique name.

Press *Next* and Xcode will ask you to save your project. Select a location and press Save. Xcode will grab its template files and generate you a working project! Excellent:

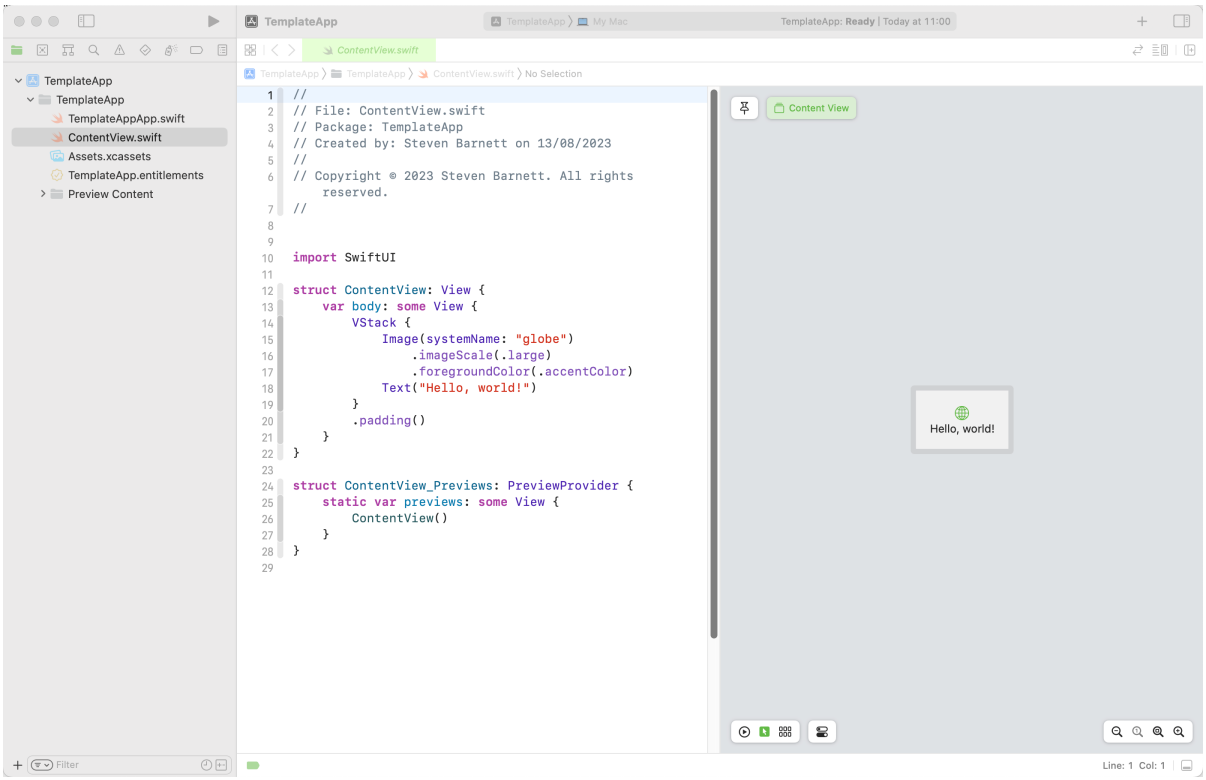


Figure 4: Xcode created project

There are the bare minimum of files listed in the project explorer on the left. You'll be interested in the App and ContentView files to begin with. However, before we get into what we need to do, let's run the app and see what we get.

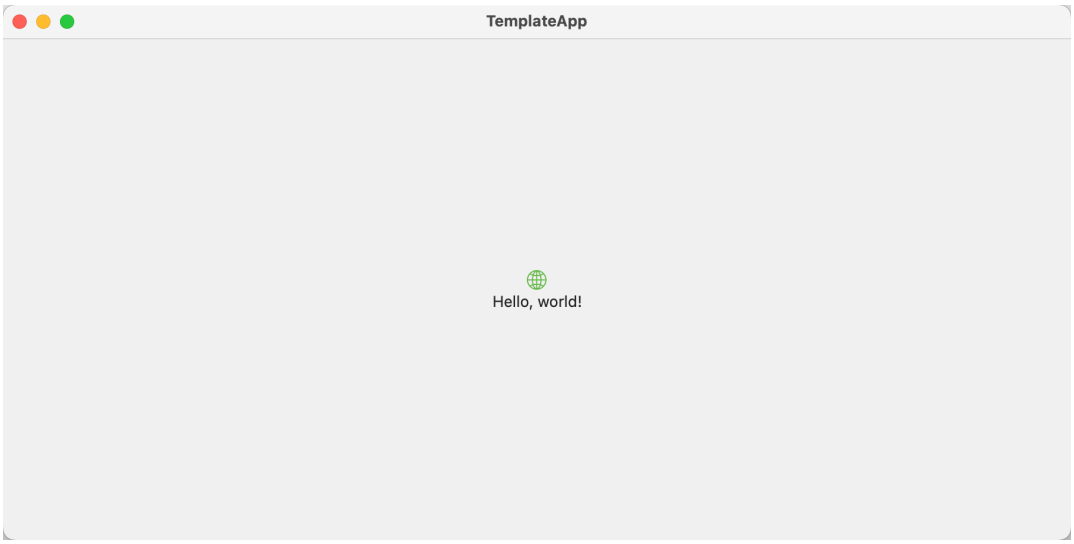


Figure 5: The default app

Firstly, we get a rather large window with our tiny amount of content sat in the middle of it. We also get a standard set of menu items attached to the app.

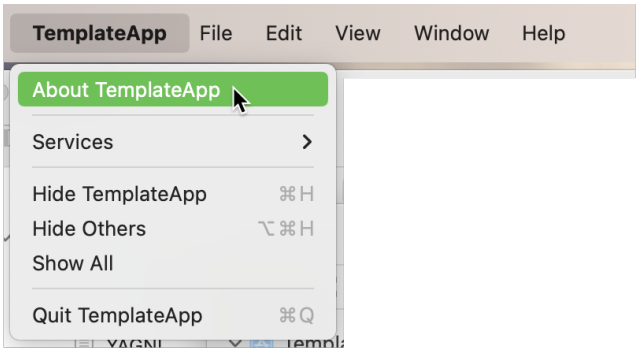


Figure 6: The default app menu

This has a few issues. As you can see, there is no settings menu. If you scroll along the menus, you'll also find a bunch of other menu items, Manu of which may not be appropriate to your app.

Finally, click the close icon on the main window. It will close, as you expect. However, if you look at the menu, you will see that the app is still running. It doesn't close when you close the main window. Now, that may be something you want but, for most apps I have seen, this is not what I would expect to happen and is not helpful.

We're going to address these issues in the up-coming sections.

First Fix Jobs

Ok, so we need a few fixes to get us past these initial issues. For a first fix, we're going to keep the changes small. The aim is just to get some basic functionality created that makes the app more usable.

Window Size

Let's start with the easy one though; the window size.

Our aim here is to have the window initially sized to fit the contents we define in our initial view. The template code comes with a label and a small image. That gets lost in the full sized window, so we need to tell Xcode to resize the window to fit around the actual contents of our view:

```
@main
struct TemplateAppApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        .windowResizability(.contentSize)
    }
}
```

The *windowResizability* modifier informs the system that the window should be resized automatically to fit the content of the view. This results in an unappealing little window, but one that is fitted to our content:

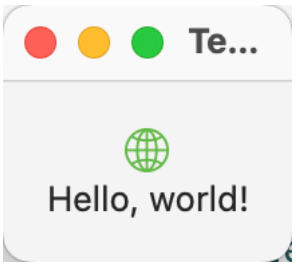


Figure 7: The auto-resized window

However, this also presents us with our next challenge. The window that results cannot be resized. It is always going to fit the content. Normaly, this isn't going to be a problem as the content of the window itself will be resizable (we'll address this later), so the window will be resizable along with its content. In this case, however, the content size if fixed and cannot scale, so the window can never be resized.

So, how do we fix this? If we pop over to the ContentView, we can add a frame definition to the VStack and give it a frame with a minimum width and height:

```
struct ContentView: View {
```

```
    var body: some View {
        VStack {
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundColor(.accentColor)
            Text("Hello, world!")
        }
        .padding()
        .frame(minWidth: 160, minHeight: 160)
    }
}
```

Well, that helped a little by making the window a little larger. However, it still sticks to the size we defined and still isn't resizable. We need to revisit the App definition.

```
@main
struct TemplateAppApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        .windowResizability(.contentMinSize)
    }
}
```

When we set-up the *windowResizability*, we told Xcode to resize the window to the content size. It does this and strictly applies the rule. That's going to be great of your main window is intended to be of a fixed size and you really do not want it resizable. In our template, this isn't the case. We want the user to be able to resize our window, so we set the *windowResizability* to *.contentMinSize*. This will ensure that the minimum window size is determined by the frame in our ContentView but that we can allow it to be resized.

As a result of this change, we are able to resize the window, though we cannot make it smaller than the minimum size we defined.

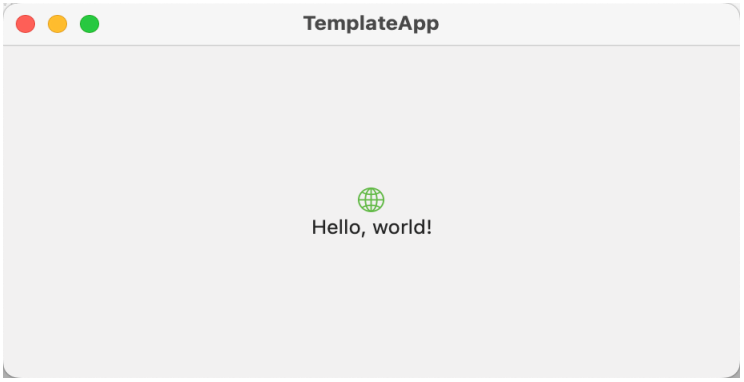


Figure 8: The auto-resized window

Just as importantly, MacOS will remember the size and position of the window so it is restored each time the app is opened again.

We're not finished yet!

What we have above works perfectly well and is good for the main window. However, it suffers one fatal flaw. If you File->New to create a new window, the window will be created using a default size determined by MacOS. That's not great.

So, while the previous method is fine if you're only going to have one window, we need a better method if we intend to allow multiple windows to be opened. Fortunately, that's pretty easy too. Change the TemplateApp code to this:

```
var body: some Scene {
    WindowGroup {
        MainView()
    }
    .defaultSize(width: 800, height: 600)
    .windowResizability(.contentMinSize)
```

We've added a default size, which will be used to set the size of any new windows created.

So, we now have the best of all worlds. Any new windows we create will be sized to the default size we specified (800x500) with a constraint that the window cannot be shrunk below the minimum content size.

Window Title

While we have our window properly resizable, it also has a default window title. A particularly ugly one at that, since it was generated when the project was created and matches the project name we chose. To change it, we need to add a *navigationTitle* to our ContentView:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundColor(.accentColor)
            Text("Hello, world!")
        }
        .padding()
        .frame(minWidth: 160, minHeight: 160)
        .navigationTitle("A Mac Template Window")
    }
}
```

The resulting window now have a much more useful title.

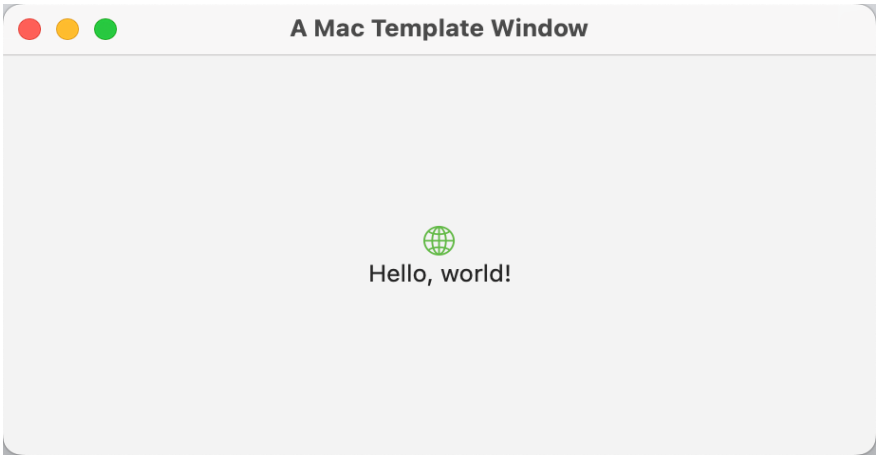


Figure 9: A new window title

You can optionally include a sub-title in your window. It's going to appear below the title in a smaller, lighter, font. There are applications where this might be appropriate, but we're not going to bother for the template. It's a little bit too specialised for a generic template file. If you want to try it, just add:

```
        .navigationSubtitle("This is a sub-title.")
```

below the .navigationTitle line.

Closing Our App

When you close the initial window, you will notice that the app continues to run. It still has a menu bar but there is no window to see. For some apps, that's a perfectly acceptable thing to happen. You might, for example, have File -> New or File -> Open menu items to open a file. In that case, you would legitimately leave the app running when there is no open window.

For many apps, however, when you close the last open window it is more likely that you want to close the app at the same time.

We can achieve this in one of two ways, depending on the style of your app. To understand the difference, you need to understand the difference in behaviour between a Window and a WindowGroup in the app definition.

We currently have an app definition that consists of:

```
@main
struct TemplateAppApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        .windowResizability(.contentMinSize)
           .defaultSize(width: 800, height: 600)
    }
}
```

Our scene contains a WindowGroup and the WindowGroup contains a single ContentView. When our app starts, the scene is created and the WindowGroup opened. This displays the ContentView. Because we have used WindowGroup, the app assumes that we may have other windows, so it forces the app to remain active even when the last window is closed.

Auto-close behaviour - Method one

If we know that closing the main window is the end of our app, when we can change the WindowGroup to a Window.

```
@main
struct TemplateAppApp: App {
    var body: some Scene {
        Window("Test App", id: "ContentView") {
            ContentView()
        }
        .windowResizability(.contentMinSize)
    }
}
```

By switching to a Window, we ensure that the app will only ever create one top level window and tell the app that it is finished when this window closes. Because we are creating a window, we are obliged to give the window a name and to give it an identifier. The identifier must be unique within our app.

With this small change, you will find that the app closes when the top level window closes. That's the simple scenario dealt with.

It does, however, have the down side that we lose the window management that WindowGroup gives us. We are limited to a single top level window and the automatically generated *File->New Window* menu item will not be created.

Auto-close behaviour - Method two

Using a Window is fine if you only have one main window and do not want to support multiple copies of your app running at the same time. Good practice, however, suggests that you should be using WindowGroup. In order to support automatic closure of the app when the last window closes when using WindowGroup, we have to do a little more work.

Firstly, we need to delve into a little AppKit code. Our app, at the AppKit level, exposes various methods that we can override once we become a delegate for the app. This means we're going to need access to an AppDelegate class.

So, create a new file and define the AppDelegate:

```
import Foundation
import AppKit

class AppDelegate: NSObject, NSApplicationDelegate {
    func applicationShouldTerminateAfterLastWindowClosed(_ sender: NSApplication)
-> Bool {
        return true
    }
}
```

An AppDelegate is based on an NSObject and implements the NSApplicationDelegate protocol. We are not required to implement anything in the delegate. However, for our needs we need to override a function that will be called to determine what happens when the last window closes; *applicationShouldTerminateAfterLastWindowClosed*. By returning *true* we tell our app to terminate itself.

This is only half of the job. We now have access to the AppDelegate and all of the functions it exposes. However, the app itself will not use it until we modify the App source.

Nip back to the app definition and modify it to declare the app delegate:

```
@main
struct TemplateAppApp: App {

    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        .windowResizability(.contentMinSize)
           .defaultSize(width: 800, height: 600)
```

```
    }
}
```

The *@NSApplicationDelegateAdapter* will make the connection between the app and the app delegate class we defined. With this in place, run the app and close the main window. The app will terminate when you close the main window.

As a better test, run the app and use File->New Window to create some more instances of the main window. As you close them, you will see that the app remains active until the last one is closed.

Clean up

With the minimal amount of code we have added so far, the overall structure of the app hasn't changed much. Now is a good time to make some changes to the structure; before we end up with a lot of files and a lot more work to do.

Structure Changes

At present, the structure looks like this:

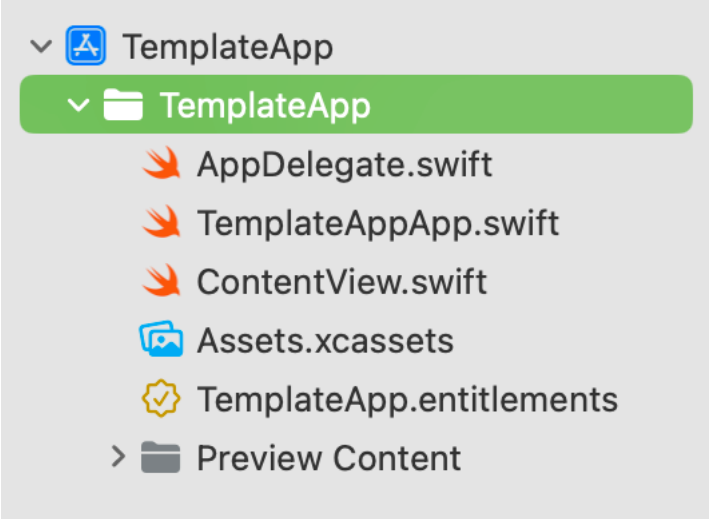


Figure 10: Our project structure

The structure of a project is a very personal thing. It revolves around how your brain works and how you organise things mentally in a logical way. I could offer all sorts of advice and it may work for you. It may not. So I offer a personal preference. It should be said, I do not dive in and create a lot of folders up-front. I prefer to evolve my folder structure as I work. At this stage, we have a small number of files, so few only need a small number of folders to get us started.

My restructured project looks like this:

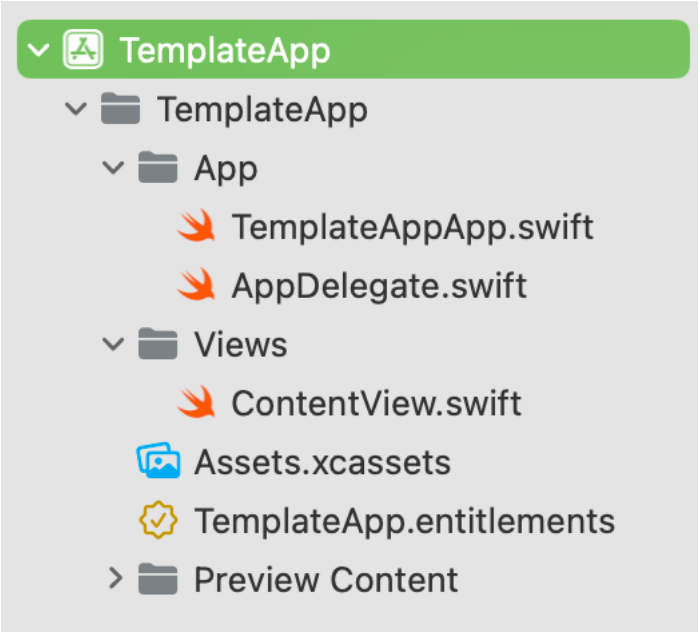


Figure 11: Our new project structure

The change is simple, I have moved the app specific file to a folder called *App* and our initial view to a folder called *Views*. I expect this structure to change as the app evolves, but this is a good place to start.

File Changes

Something I have been guilty of many times is leaving my main view called ContentView. It's not particularly descriptive and, while you could find it by convention, it's better to rename it to something more meaningful. Since it's our main root view and our app doesn't actually have a purpose yet, let's rename it to *MainView.swift*.

Open the ContentView.swift file, right click on ContentView in the structure and select Refactor->Rename:

Figure 12: Rename ContentView

Xcode will scan the project and highlight all instances of ContentView. Just type the new view name and press enter:

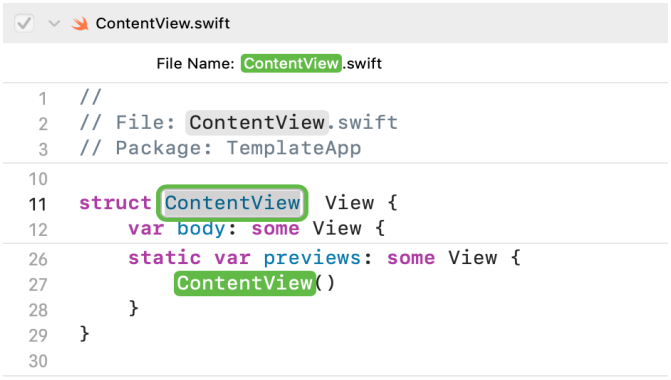


Figure 13: Entering ContentView new name

Sadly, it's not 100% perfect, which is why we're going this up-front. It misses the preview!

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        MainView()
    }
}
```

Figure 14: Rename the preview

Constants

Our app is cleaner than when we started. However, we have also written in a few constants. Constants have their place, but I try to gather together any constants that may need to change over time and, since this is a template, we want to make customisation as easy as possible. So, let's create a new structure in the App folder called *Constants.swift* to hold these constants.

```
import SwiftUI

struct Constants {

}
```

Figure 15: App level constants

Scanning through our app, we have a small number of things we want to make constants. For this initial pass, we're going to define:

```
import SwiftUI

struct Constants {
    // Main window
    static let mainWindowWidth: CGFloat = 800
    static let mainWindowHeight: CGFloat = 500
    static let mainWindowMinWidth: CGFloat = 160
    static let mainWindowMinHeight: CGFloat = 160
    static let mainWindowTitle: String = "A Mac Template Window"
}
```

There will be corresponding changes to the MainView to use these constants:

```
struct MainView: View {
    var body: some View {
        VStack {
```

```
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundColor(.accentColor)
            Text("Hello, world!")
        }
        .padding()
        .frame(minWidth: Constants.mainWindowMinWidth,
               minHeight: Constants.mainWindowMinHeight)
        .navigationTitle(Constants.mainWindowTitle)
    }
}
```

This will make customising our project easier. We also need to update the App file to use the default window sizes:

```
    var body: some Scene {
        WindowGroup {
            MainView()
        }
        .windowResizability(.contentMinSize)
        .defaultSize(width: Constants.mainWindowWidth, height:
Constants.mainWindowHeight)
```

Conclusion

We have made minimal changes to the out of the box template that Xcode provided us with. However, the app now has a simple way of controlling the size of a new window and behaves in a more consistent way when closing windows.

Chapter Two

Dialogs

Almost every app you look at will have at least two dialogs; An *About* box and a *Settings* screen. Your default app template will have created an about box for you automatically.

The Settings is all your own problem, though there is infrastructure in place to help.

About Box

The Default About Box

When the app was created, Xcode created an about box for you. You can't see the code for this, but it is there. When you run you app, you'll have a menu option:

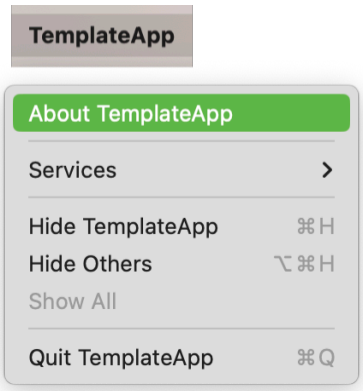


Figure 16: About App Menu

If you select this, a popup window will appear with the system generated about box content:

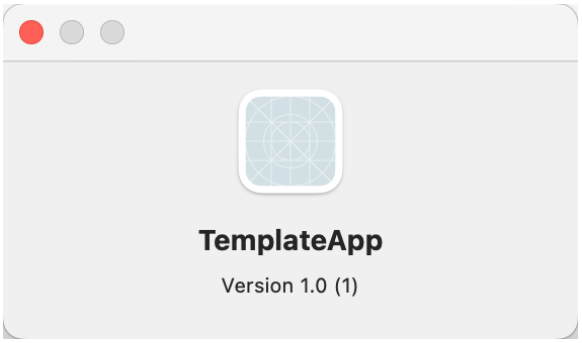


Figure 17: About App Window

For something you get for free, it's not too bad. The image is your App Icon, the title is your target name and the version is the app version from the bundle.

Renaming your app

Calling our app "TemplateApp" is fine while we're in development, but makes the menu item and the about box title look rather poor. We can get a quick fix here by renaming the target name. Go over to the project settings and click on the target to set a new name:

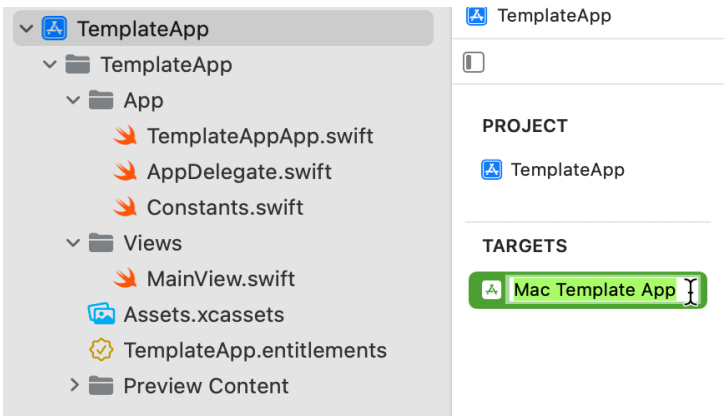
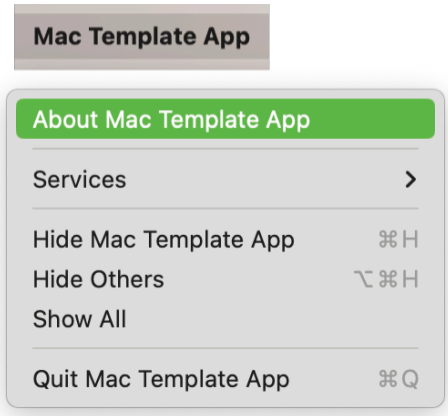
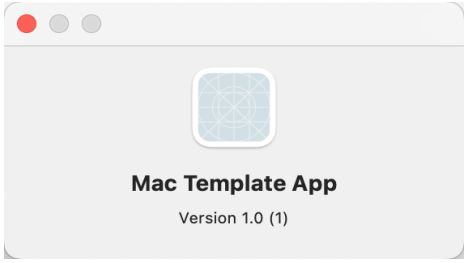


Figure 18: Rename the target

This one simple change makes the menu and the about box a little cleaner:



The title of the app has changed, the menu item has changed and the about box has changed.



Enhancing the About Box

For all its functionality, it is a little basic. There is no copyright, no author name or any contact details.

As so often happens, there are multiple ways to address this shortfall and which route you go down depends on how complex you want to make your About box.

Extending the default about box

This is the easiest option. It will allow you to add formatted text at the bottom of the existing about box.

To achieve this, add a file called *Credits.rtf* to the project; I've added mine to the App folder:

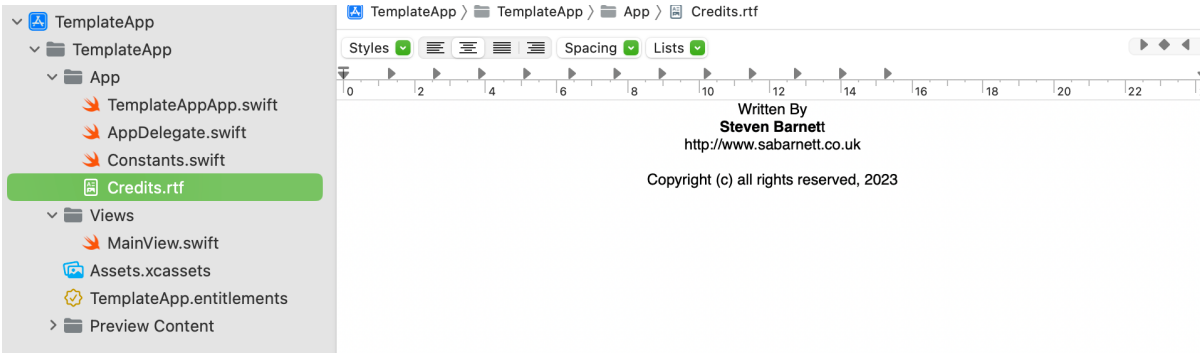


Figure 19: Adding Credits

You can edit the content to include any kind of text you want. For this example, I've added my name, web site address and a copyright notice. Basic identification details. Because this file exists, when the about box is built by the system, it will include these details on the dialog that gets shown:

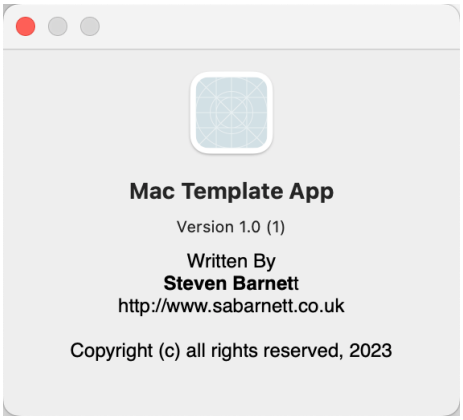


Figure 20: About box with credits

Not a bad result for so little work. It does, however, have one issue that we can't fix using the Credits.rtf file; the link to my web site isn't clickable. That may not be a big deal, but it would be nice to fix it.

Replacing the default about box

When the default about box isn't sufficient, we need to revert to Plan-B and replace it entirely. This is going to be slightly complicated and will introduce more AppKit like code. We're sticking with SwiftUI, so bear with me.

The first job is to replace the About menu item so we can control what happens when it's clicked. To do that, we go back to our app struct and add the following:

```
@main
struct TemplateAppApp: App {

    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            MainView()
                .frame(minWidth: Constants.mainWindowMinWidth,
                       minHeight: Constants.mainWindowMinHeight)
```

```
        }
        .windowResizability(.contentMinSize)
           .defaultSize(width: Constants.mainWindowWidth, height:
Constants.mainWindowHeight)

        .commands {
            // Replace the About menu item.
            CommandGroup(replacing: CommandGroupPlacement.appInfo) {
                Button("About \(Bundle.main.appName)") {
                    appDelegate.showAboutWnd()
                }
            }
        }
    }
}
```

The *.commands* modifier gives us access to the menu bar, allowing us to replace existing menu items, remove them completely or add new ones. In this specific case, we're looking to replace what happens when the About menu item is clicked. This particular menu item is referenced using the *CommandGroupPlacement.appInfo* enum.Here we have elected to replace it with a button that will call the *showAboutWnd* method of the *appDelegate*.

Luckily, we defined an appDelegate earlier, so we can add our method to that:

```
class AppDelegate: NSObject, NSApplicationDelegate {

    private var aboutBoxWindowController: NSWindowController?

    func showAboutWnd() {
        if aboutBoxWindowController == nil {
            let styleMask: NSWindow.StyleMask =
[.closable, .miniaturizable, .titled]
            let window = NSWindow()
            window.styleMask = styleMask
            window.title = "About \(Bundle.main.appName)"
            window.contentView = NSHostingView(rootView: AboutView())
            window.center()
            aboutBoxWindowController = NSWindowController(window: window)
        }

        aboutBoxWindowController?.showWindow(aboutBoxWindowController?.window)
    }

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

We define a variable called *aboutBoxWindowController* to keep track of whether the about box has been shown before. We don't want to recreate it every time we display it. If it doesn't currently exist, we create an NSWindow, which is an AppKit style window, and assign our SwiftUI view as its *contentView*. This way we have an NSWindow with SwiftUI content. Finally, we tell the controller to display the window.

You are also going to have to import SwiftUI in order to use NSHostingView, so make sure you add tat to the top of your file.

If you compile at this stage, it's not going to work! For the window title, we are assigning the appName. This isn't an actual property of the bundle, so that property isn't going to be found. We

need to add that property by extending the Bundle to add a few properties that we might find useful here and in the about box.

For the sake of organisation, create a new folder and call it Extensions. We're going to group our extension code in this one place so we always know where to go to find any extensions we code. Once you have the folder, create a new Swift file and call it *Bundle+Extensions.swift* to contain the extension code. Then create the code:

```
import Foundation

extension Bundle {
    public var appName: String { getInfo("CFBundleName")  }
    public var copyright: String {getInfo("NSHumanReadableCopyright")
        .replacing("\\\\n", with: "\n") }

    public var appBuild: String { getInfo("CFBundleVersion") }
    public var appVersionLong: String { getInfo("CFBundleShortVersionString") }

    fileprivate func getInfo(_ str: String) -> String {
        infoDictionary?[str] as? String ?? "⚠️"
    }
}
```

We're almost there. We now have everything we need to replace the about box with our custom one, so now we just need to code a new About box.

Under the *Views* folder, add a new folder called *Dialogs* and add a new SwiftUI View called *AboutView*. This is what we are going to create:

```
struct AboutView: View {
    var body: some View {
        VStack(spacing: 10) {
            Image(nsImage: NSImage(named: "AppIcon")!)

            Text("\(Bundle.main.appName)")
                .font(.system(size: 20, weight: .bold))
                .textSelection(.enabled)

            Link(Constants.homeAddress,
                 destination: Constants.homeUrl )

            Text("Ver: \(Bundle.main.appVersionLong) (\(Bundle.main.appBuild)) ")
                .textSelection(.enabled)

            Text(Bundle.main.copyright)
                .font(.system(size: 10, weight: .thin))
                .multilineTextAlignment(.center)
        }
        .padding(20)
        .frame(minWidth: 350, minHeight: 300)
    }
}
```

We will have an image with the name of the app below it. Under that, we will have a link to our support page. Following that we will have the app version and a copyright notice.

As before, if you compile this code now, it will fail to compile. I've added a couple of new constants; one for the support page message and one for the support page URL. You'll need to add these to the *Constants* swift file.

```
//About box
static let homeUrl: URL = URL(string: "http://www.sabarnett.co.uk")!
static let homeAddress: String = "App support page"
```

This will give us code that compiles, but doesn't run. It will crash when we display the about box because we have referenced the AppIcon image and we have not added an AppIcon yet. So, create yourself an AppIcon before you run this code. Once you have an AppIcon, run your code and select the About menu item.

If you're struggling to create an AppIcon, take a look at a package on the Mac App Store called Bakery. It's really useful for creating icons.

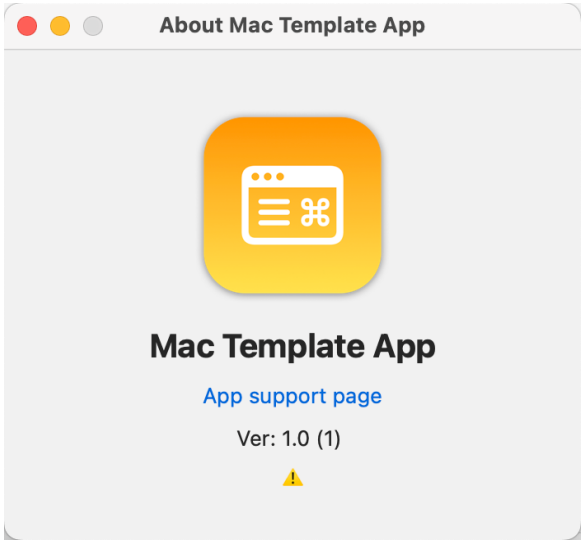


Figure 21: New About Box

Oops, we have a problem. The copyright isn't being shown.

By default, there isn't a human readable copyright generated for our App, so we need to set the text to be displayed ourselves. We need to add this to the target. It's easier to find, if you search!

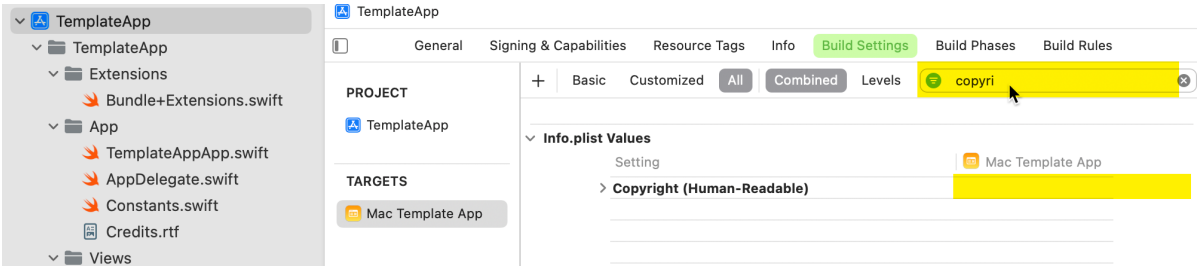


Figure 22: Set the copyright

Enter the copyright text you want to see against the Human Readable Copyright option. Once that's entered, re-run your app and check that the copyright is being picked up correctly.

Figure 23: The Fixed About Box

The content is entirely up to you. What we have done here is replace the baked-in about box with one that we can customise to fit our needs. While the process seems a little complicated it isn't really. There are a few steps to follow and we gain a large degrees of flexibility.

Also worth noting, if you click no the *App support page* text, your web browser will open up on the page you defined.

Settings

Very few of the apps I have ever used didn't have some kind of user settings. Little tweaks that make the app more customisable by the user. They are generally small options that can have a big impact on the way the app works.

Our template does not have any settings nor does it have a settings menu option. Crafting useful user settings can be difficult but adding a settings menu and dialog are relatively straight forward.

The Settings Dialog

Before we can add our menu item, we need a settings dialog to show. This can be as simple as a window with a single setting or as complex as a tabbed dialog with hundreds of settings. Here, we will plan for a basic template settings dialog, based around tabs and a couple of example settings.

In the View->Dialogs folder we create a new SwiftUI struct called *SettingsView*. This is what we're going to show when the user selects the settings menu.

```
struct SettingsView: View {
    var body: some View {
        TabView {
            Text("General")
                .tabItem {
                    Label("General", systemImage: "gearshape")
                }

            Text("Set 1")
                .tabItem {
                    Label("One", systemImage: "square.and.arrow.up")
                }

            Text("Set 3")
                .tabItem {
                    Label("Advanced", systemImage: "gearshape.2")
                }
        }
        .frame(minWidth: 460)
    }
}
```

This is a simple placeholder view for now. It's a TabView with three tabs, each of which consists of a Text view so we can see something on the screen. We can expand on this later.

It is important to set the frame width to some sensible value. While the window is going to resize itself, we need to endure that there is always enough space to define all of the icons.

Connecting the Settings Dialog up

Switch back to the App definition. We're going to add a link to our settings dialog here. After the .commands, we add Settings() and set the content to be displayed to our settings view.

```
@main
struct TemplateAppApp: App {

    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            MainView()
        }
        .windowResizability(.contentMinSize)
           .defaultSize(width: Constants.mainWindowWidth, height:
Constants.mainWindowHeight)

        .commands {
            // Replace the About menu item.
            CommandGroup(replacing: CommandGroupPlacement.appInfo) {
                Button("About \(Bundle.main.appName)") {
                    appDelegate.showAboutWnd()
                }
            }
        }

        Settings {
            SettingsView()
        }
    }
}
```

It's that simple. Run the app and open the app menu. You'll see that there is a new menu item that was generated by Xcode with the standard settings shortcut:

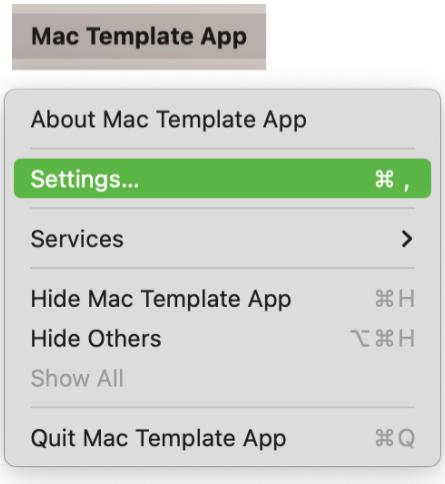


Figure 24: The Settings Menu

If you select this menu item, you'll see a rather uninspiring settings dialog appear:

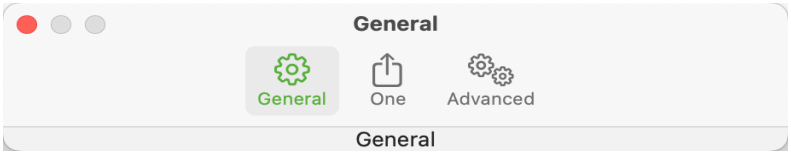


Figure 25: The Settings Dialog

While uninspiring, it comes with some basic functionality. As you click on the tabs, the title of the window will change and the content of the view will change:

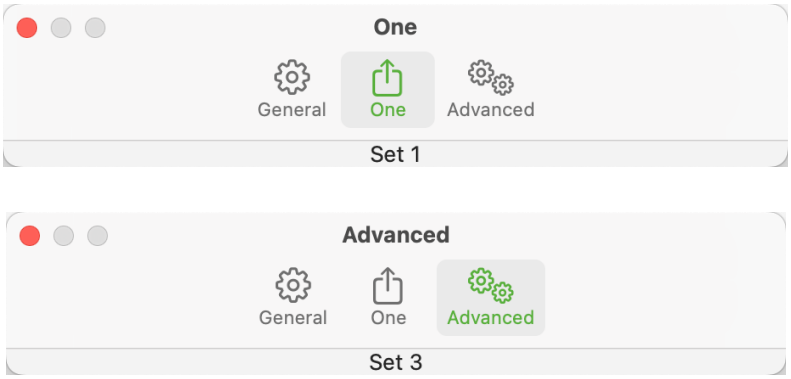


Figure 25: The Settings Tabs

The window title changes to the text of the label we used to define the tab item. The content is whatever we defined as the content of the tab item, the Text() in this case.

Options Model

In a real app, we're going to have several options and coding all of these into the settings view is quickly going to make it very large, so we need to have a View Model handle the options. In a simple app this may seem over-kill, but simple apps have a habit of becoming more complex over time, so it's best to prepare for the future and it's good practice to split the view from it's data. So, win-win.

Over in *Views->Dialogs* create a file called *SettingsViewModel* and populate it with this basic content:

```
import SwiftUI

class SettingsViewModel: ObservableObject {

}
```

It's a basic Observable Object into which we will place our settings.

Back in the SettingsView, add a reference to the view model:

```
struct SettingsView: View {

    @StateObject private var settings = SettingsViewModel()

    var body: some View {
        TabView {
```

This gets our settings available to our view. All we need now are some settings and some code to let us set them.

Basic Template Settings

Sample Settings Content

We now have a settings screen which we can display through the menu. It has three tabs defined and we have sample, if useless, content one each tab. For the template project to be useful, we need to define some example settings that we can customise at a later date when the template gets used.

Tabs

Each tab is currently represented with a Text() view. We can replace these with a VStack with our content or, in order to not over complicate our view code, we can separate the settings into separate views. Which method you use will depend entirely n the number and complexity of the settings you want to support. Being a template, we'll go for a half-way solution.

Head over to the *SettingsView* and make these changes to the body:

```
    var body: some View {
        TabView {
            generalSettings()
                .tabItem {
                    Label("General", systemImage: "gearshape")
                }

            set1Settings()
                .tabItem {
                    Label("One", systemImage: "square.and.arrow.up")
                }

            advancedSettings()
                .tabItem {
                    Label("Advanced", systemImage: "gearshape.2")
                }
        }
        .frame(minWidth: 460)
    }
```

The individual Text() views have been replaced with function calls that will return the actual content. To achieve this, add three functions to the same file.

```
private func generalSettings() -> some View {
        VStack {
            Text("These are the general settings")
        }.padding(15)
    }

    private func set1Settings() -> some View {
        VStack {
            Text("These are the set 1 settings")
```

```
        }.padding(15)
    }

    private func advancedSettings() -> some View {
        VStack {
            Text("These are the advanced settings")
        }.padding(15)
    }
```

Each function corresponds to each tab and will define the settings that we want on that tab. It's also worth noting that I have added padding around the individual VStack's to give us some space within the window.

Doing it this way gives us a degree of separation in the view definition and the content of each tab. It also leaves us set-up if we decide to move the content of the tab to a separate file at some later date.

General Settings

On the general settings tab, we're going to have a few basic settings, just for illustration. Fir the general tab, let's start with some toggle settings. First off, create the settings in the View Model:

```
class SettingsViewModel: ObservableObject {

    // General tab options
    @AppStorage("setting1") var toggle1: Bool = false
    @AppStorage("setting2") var toggle2: Bool = false
    @AppStorage("setting3") var toggle3: Bool = false

}
```

This goes us something to bind to. Back in the Settings view, modify the *generalSettings* function to implement these settings:

```
private func generalSettings() -> some View {
    VStack {
        Text("These are the general settings and these are some instructions to the
user.")

        Toggle(isOn: $settings.toggle1, label: { Text("This is setting 1")} )
        Toggle(isOn: $settings.toggle2, label: { Text("This is setting 2")} )
        Divider()
        Toggle(isOn: $settings.toggle3, label: { Text("This is setting 3")} )
    }.padding(15)
}
```

We have some instructional text for the user and three Toggle()'s to allow the settings to be changed. When we run this, we get a slightly more impressing General tab.

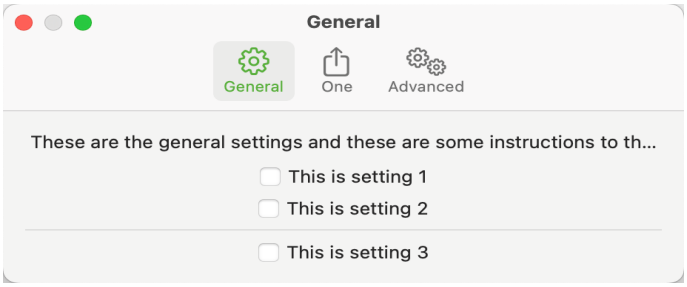


Figure 26: The General Setting Tab

All the content is entered, which is a side effect of the VStack and we have three check boxes; one for each setting. It's also worth noting that the window has expanded to fit the content. If you select another setting tab, the window will shrink back to fir the content on that specific tab.

Set 1 Settings

We've seen how toggles work. For the Set 1 settings, we'll add some other settings. Modify the Set 1 settings as follows:

```
    private func set1Settings() -> some View {
        VStack {
            HStack {
                Text("Support email:")
                TextField("Your email", text: $settings.supportEmail)
            }

            Picker("Experience Level", selection: $settings.experienceLevel) {
                ForEach(UserExperience.allCases, id: \.self) { level in
                    Text(level.rawValue.capitalized)
                }
            }

            HStack {
                Slider(value: $settings.yearsExperience,
                       in: 1...11,
                       label: { Text("Years Experience")},
                       minimumValueLabel: {Text("1")},
                       maximumValueLabel: {Text(">10")})
                Text(String(format: "%.0f", settings.yearsExperience))
                    .font(.caption)
            }

            Button("Reset Settings") { print("Reset settings") }
        }.padding(15)
    }
```

We've added four input types here;

- A plain text field
- A picker control attached to an enum
- A slider control - using an HStack to include the current value
- A button.

To make this work, we're going to have to add some settings to the view model:

```
class SettingsViewModel: ObservableObject {

    // General tab options
    @AppStorage("setting1") var toggle1: Bool = false
    @AppStorage("setting2") var toggle2: Bool = false
    @AppStorage("setting3") var toggle3: Bool = false

    // Set 1 settings
    @AppStorage("SupportEmail") var supportEmail: String = ""
```

```
    @AppStorage("ExperienceLevel") var experienceLevel: UserExperience = .beginner
    @AppStorage("yearsExperience") var yearsExperience: Double = 1.0
}
```

Our experienceLevel setting is based on the UserExperience enum, so we also need to create that:

```
enum UserExperience: String, CaseIterable {
    case beginner
    case novice
    case junior
    case senior
    case lead
}
```

We've defined it as a String base type so we can use the raw value in the picker. We also define it as CaseIterable so we can iterate over all cases when we build the picker.

The resulting settings tab looks like this:

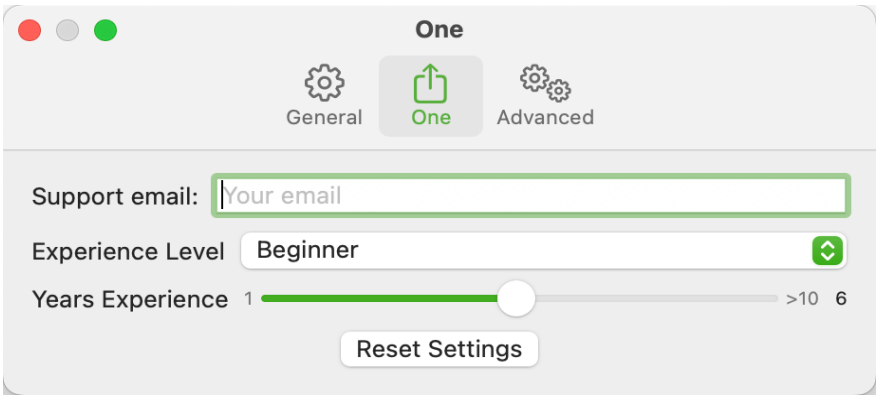


Figure 27: The Set 1 Setting Tab

Not wildly inspiring, but it illustrates some of the potential input types.

Advanced Settings

Advanced Settings

For the Advanced Settings tab, we're going to take a different route. If you open up the Xcode settings, you'll see something like this:

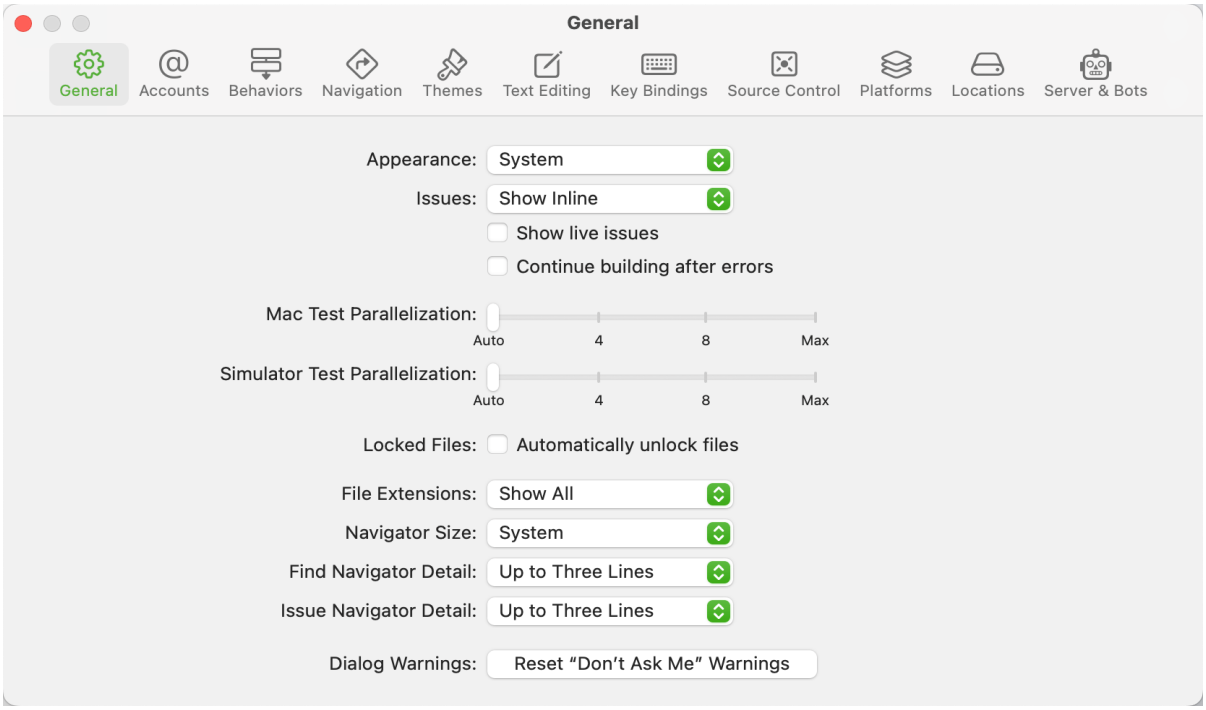


Figure 28: Xcode settings

The overall style is that there is a label on the left(optional for check boxes) and the input fields are vertically aligned on the left. It makes is easier and quicker for the user to scan down the settings and looks a lot better than raggedly aligned options. We want to achieve this in out advanced settings.

Be warned, there are lots of ways to achieve this kind of layout and none of them are particularly simple to achieve. For reference purposes, this is the advanced settings that we are going to create for our template.

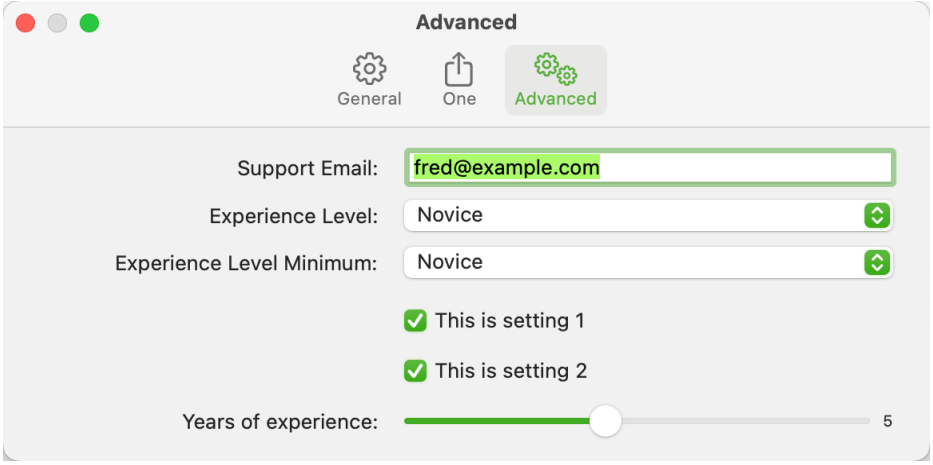


Figure 29: Advanced settings

Not an earth shattering window, but it serves to illustrate a few of the issues we're going to have along the way.

Housekeeping

Firstly, we need to do a little housekeeping. We need two new constants in our constants file; one for the width of the settings window and one for the amount of space to allow for the setting label:

```
struct Constants {
    // Main window
    static let mainWindowWidth: CGFloat = 800
    static let mainWindowHeight: CGFloat = 500
    static let mainWindowMinWidth: CGFloat = 160
    static let mainWindowMinHeight: CGFloat = 160
    static let mainWindowTitle: String = "A Mac Template Window"

    //About box
    static let homeUrl: URL = URL(string: "http://www.sabarnett.co.uk")!
    static let homeAddress: String = "App support page"

    // Settings window width
    static let settingsWindowWidth: CGFloat = 550
    static let settingsWindowLabelWidth: CGFloat = 220
}
```

Because we're going to be cramming a lot onto one line, the width of the settings window really needs to be increased, so I've made it 550. That'll give us enough room to get the label in and to ensure that there is enough space for the options.

We then need to change the corresponding frame the SettingsView:

```
TabView {
    generalSettings()
        .tabItem {
            Label("General", systemImage: "gearshape")
        }

    set1Settings()
        .tabItem {
            Label("One", systemImage: "square.and.arrow.up")
        }

    advancedSettings()
```

```
        .tabItem {
            Label("Advanced", systemImage: "gearshape.2")
        }
}
.frame(width: Constants.settingsWindowWidth)
```

Settings Line Helper

We're going to need a far amount of code to define the content of a setting. There's going to be the left side content, which is a right aligned label and there is going to be the right side content, which is the left aligned input control. We need to ensure that the label side is of a fixed width and we need to ensure that the controls on the right side all line up.

To make some of this work easier, we will define a new view that encompasses a single settings line:

```
struct SettingsLine<Content: View>: View {
    var content: () -> Content
    var label: String

    init(label: String, @ViewBuilder content: @escaping () -> Content) {
        self.label = label
        self.content = content
    }

    var body: some View {
        HStack {
            HStack(alignment: .top) {
                Spacer()
                Text(label)
            }
            .frame(width: Constants.settingsWindowLabelWidth)

            HStack {
                content()
                Spacer()
            }
        }
        .frame(width: Constants.settingsWindowWidth)
    }
}
```

Since the type of input control will vary, we're defining this as dynamically created content that is passed to us by the caller of the view. By using @ViewBuilder, we leave the responsibility for the content with the caller.

The body of our view consists of an HStack, that we constrain to the width we specified in our Constants, that contains two further HStack's. The first defines the label, with a Spacer to force it to right aligned, and constrains its width. The second defines the user supplied input control followed by a Spacer to ensure it's left aligned.

With most views, I would usually create them in separate files to make them easier to find in the project structure. In this case, since this view will only ever be used in the SettingsView, I have added it to SettingsView.swift. It removes a little of the potential clutter in the project.

Building The Advanced Settings

Lets start simple and add an email collection input:

```
    private func advancedSettings() -> some View {
        VStack {
            SettingsLine(label: "Support Email:", content:
                         {TextField("Your email", text: $settings.supportEmail)
            })
        }.padding(15)
    }
```

We have defined a single SettingsLine using our helper view with a left side label of “Support Email:” and a TextField to collect the data. The encompassing VStack has some padding to make sure it doesn’t but up against the window edges. When we run this, we get:

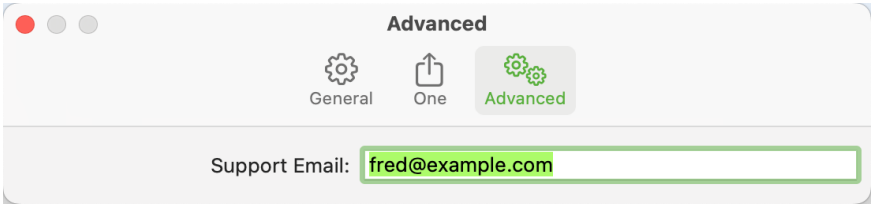


Figure 30: Our first advanced setting

Looks good and seems to be working as expected. So, let’s add some more settings to see how it looks with a little more input.

```
    private func advancedSettings() -> some View {
        VStack {
            SettingsLine(label: "Support Email:", content:
                         {TextField("Your email", text: $settings.supportEmail)
            })

            SettingsLine(label: "Experience Level:", content: {
                Picker("", selection: $settings.experienceLevel) {
                    ForEach(UserExperience.allCases, id: \.self) { level in
                        Text(level.rawValue.capitalized)
                    }
                }
            })

            SettingsLine(label: "Experience Level Minimum:", content: {
                Picker("", selection: $settings.experienceLevelMin) {
                    ForEach(UserExperience.allCases, id: \.self) { level in
                        Text(level.rawValue.capitalized)
                    }
                }
            })
        }.padding(15)
    }
```

For our next attempt, we’ve added two drop-down pickers. Again, using the SettingsLine helper view, it’s straight forward to deal with the layout requirements. When we run this, we get:

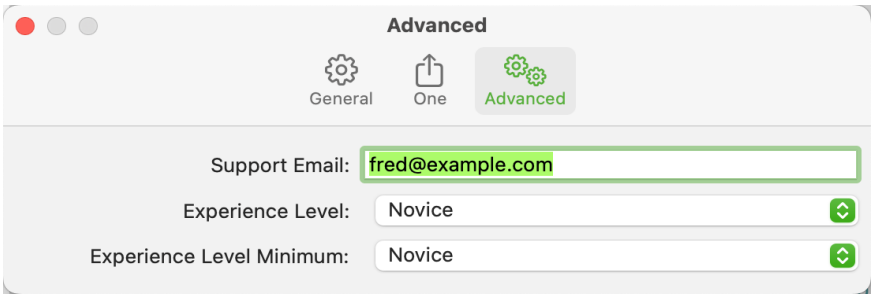


Figure 31: Our second advanced setting

Ok, looks good, but I'm seeing an issue here. The padding around controls isn't consistent, so we may end up with an alignment issue. Lets continue to add some more settings and see where we get to:

```
    private func advancedSettings() -> some View {
        VStack {
            SettingsLine(label: "Support Email:", content:
                            {TextField("Your email", text: $settings.supportEmail)
            })

            SettingsLine(label: "Experience Level:", content: {
                Picker("", selection: $settings.experienceLevel) {
                    ForEach(UserExperience.allCases, id: \.self) { level in
                        Text(level.rawValue.capitalized)
                    }
                }
            })

            SettingsLine(label: "Experience Level Minimum:", content: {
                Picker("", selection: $settings.experienceLevelMin) {
                    ForEach(UserExperience.allCases, id: \.self) { level in
                        Text(level.rawValue.capitalized)
                    }
                }
            })

            SettingsLine(label: "", content: {
                Toggle(isOn: $settings.toggle1, label: { Text("This is setting
1")} )
            })

            SettingsLine(label: "", content: {
                Toggle(isOn: $settings.toggle2, label: { Text("This is setting
2")} )
            })

            SettingsLine(label: "Years of experience:", content: {
                HStack {
                    Slider(value: $settings.yearsExperience,
                           in: 1...11,
                           label: { Text("")})
                    if settings.yearsExperience == 11 {
                        Text(">10")
                            .font(.caption)
                    } else {
                        Text(String(format: "%.0f", settings.yearsExperience))
                            .font(.caption)
                    }
                }
            })
        }.padding(15)
    }
```

I've added a couple of check boxes ad our slider to expand the view out. When we run this, we get a better view of the alignment issue we're having:

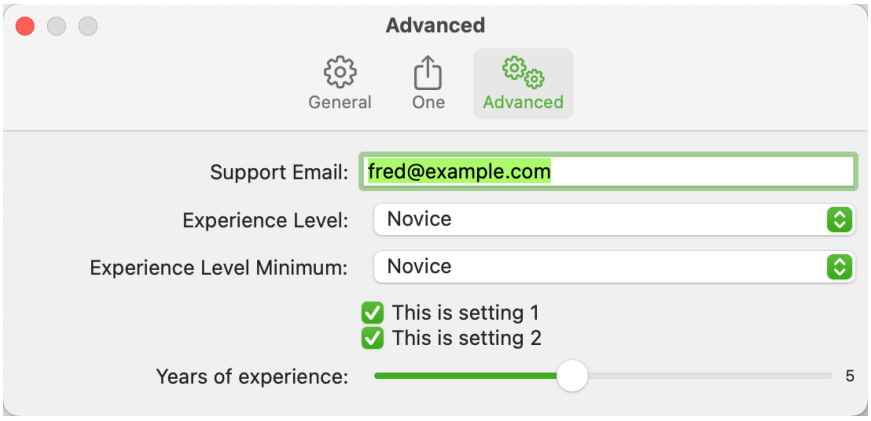


Figure 32: Our third advanced setting

Now we get a clear view of the problem. While the labels are properly aligned, the default padding around the various inputs means that nothing on the right lines up and the spacing between settings is not very consistent. Basically, it works but looks ugly.

There is no scientific way to fix this. We're going to need to change the padding around each of the inputs until we arrive at better alignment. It's hit and mss but, thankfully, consistent. What works for one Toggle will work for them all. After some experimentation, our advanced settings gets modified as follows:

```
    private func advancedSettings() -> some View {
        VStack {
            SettingsLine(label: "Support Email:", content:
                            {TextField("Your email", text: $settings.supportEmail)
                    .padding(.leading, 10)
            })

            SettingsLine(label: "Experience Level:", content: {
                Picker("", selection: $settings.experienceLevel) {
                    ForEach(UserExperience.allCases, id: \.self) { level in
                        Text(level.rawValue.capitalized)
                    }
                }
            })

            SettingsLine(label: "Experience Level Minimum:", content: {
                Picker("", selection: $settings.experienceLevelMin) {
                    ForEach(UserExperience.allCases, id: \.self) { level in
                        Text(level.rawValue.capitalized)
                    }
                }
            })

            SettingsLine(label: "", content: {
                Toggle(isOn: $settings.toggle1, label: { Text("This is setting
1")} )
                    .padding(8)
            })

            SettingsLine(label: "", content: {
                Toggle(isOn: $settings.toggle2, label: { Text("This is setting
2")} )
                    .padding(8)
            })
```

```
            SettingsLine(label: "Years of experience:", content: {
                HStack {
                    Slider(value: $settings.yearsExperience,
                           in: 1...11,
                           label: { Text("")})
                    if settings.yearsExperience == 11 {
                        Text(">10")
                            .font(.caption)
                    } else {
                        Text(String(format: "%.0f", settings.yearsExperience))
                            .font(.caption)
                    }
                }
            })
        }.padding(15)
    }
```

To improve the alignment, we've added padding to the SupportEmail setting and to the two Toggles. Nothing has been added to the pickers or the Slider as they already have in-built padding that we are trying to align to. When we run this, we get to:

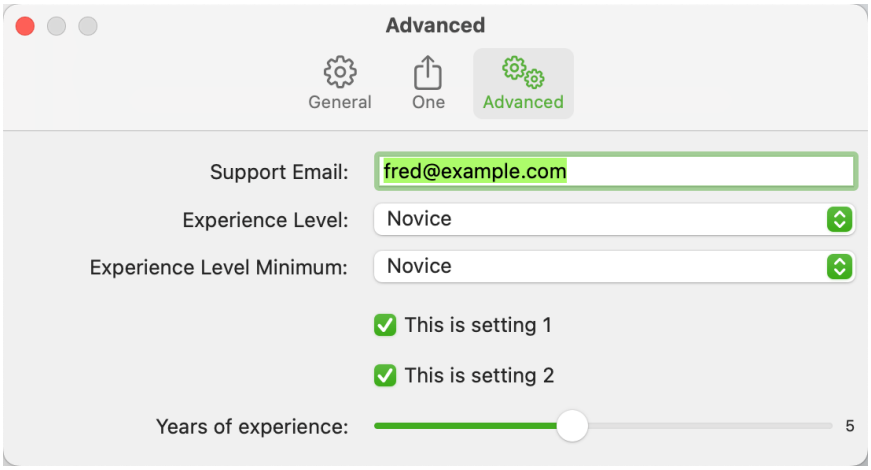


Figure 33: Our final advanced setting

This looks a lot better. Everything aligns and the spacing of the Toggle's has improved.

With that, we can declare our settings example code done!

Dark Mode

Dark Mode Toggle

To finish off the setting discussion, let's add a 'useful' setting to our template; one that will be useful in a real application.

Users are polarised to using applications in Light Mode or Dark Mode. Back in the day, there was only light mode and you had to live with it. These days, users expect to be able to toggle between seeing their app in light or dark mode or, just to complicate things, to have the application switch automatically between light and dark mode.

So, any application we build from this template is likely to need to support the setting of the light/dark state.

DisplayMode Enum

The first thing we need is an enum that describes the available modes. While enums are generally quite simple, they can be quite complex too. In our case, we're going to need to implement a few protocols to make it usable for our purposes. As it's an enum for a specific purpose that will only be used in a small number of places throughout the app, I've created a folder called *enumerations* and have added a file to it called *DisplayMode*, the contents of which are:

```
enum DisplayMode: String, Identifiable, CaseIterable, Equatable {
    case light
    case dark
    case auto

    var id: String {
        return self.description
    }

    var description: String {
        switch self {
        case .light:
            return "Light"
        case .dark:
            return "Dark"
        case .auto:
            return "Auto"
        }
    }
}
```

Why so complicated or an enum with only three values? Well, we have to define the enum with a type that can be stored in AppStorage and that's quite limited, so we define our enum as a String

type. We're going to use the enum in a loop in the settings screen, so it has to be Identifiable; hence the id property too. Because we want the loop to display all possible values, we make the enum CaseIterable which will give us an *allCases* property to iterate over every possible value. At some stage, we're going to need to know when the value of the display mode changes, so we need to make it Equatable.

To give us some control over what appears in the setting screen, we define a property called *description* to get us a displayable value.

The Settings Model

We're going to need to edit the setting and we are going to need to persist it to AppStorage so e are going to need to add a value to our SettingsViewModel. We're going to add our setting to the General tab, so we add it to the General properties:

```
class SettingsViewModel: ObservableObject {

    // General tab options
    @AppStorage("setting1") var toggle1: Bool = false
    @AppStorage("setting2") var toggle2: Bool = false
    @AppStorage("setting3") var toggle3: Bool = false
    @AppStorage("displayMode") var displayMode: DisplayMode = .auto
```

We default the value to .auto as that is the default setting for any app.

The Settings View

The final part of editing our appearance is to add the code to the settings view to allow the user to change the value. Since we're adding this to the general settings, we edit the *generalSettings* function to add our new code:

```
    private func generalSettings() -> some View {
        VStack {
            Text("These are the general settings and these are some instructions to
the user.")

            Toggle(isOn: $settings.toggle1, label: { Text("This is setting 1")} )
            Toggle(isOn: $settings.toggle2, label: { Text("This is setting 2")} )
            Divider()
            Toggle(isOn: $settings.toggle3, label: { Text("This is setting 3")} )

            Divider()
            Picker(selection: $settings.displayMode, content: {
                ForEach(DisplayMode.allCases) { mode in
                    Text(mode.description).tag(mode)
                }
            }, label: {
                Text("Display mode")
            })
        }.padding(15)
    }
```

The picker is bound to the value we defined in the view model and the content of the picker is created by iterating over the values in the DisplayMode enum. It is very important to set the tag on each item

in the list. This tag is what the picker uses to select the current item in the list and to save the selected value to the selection.

If we run this, we will see our option in the general tab and can select one of our three options:

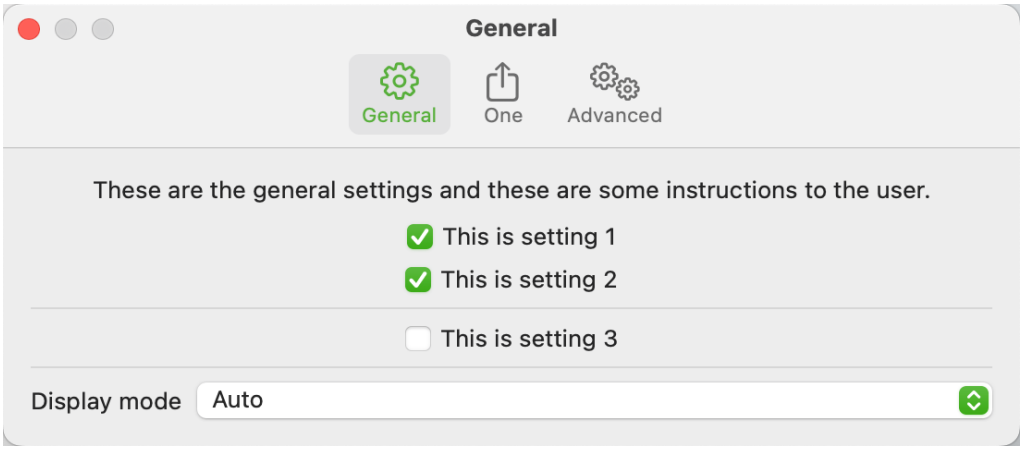


Figure 34: Display Mode Selection

Changing the display mode

Selecting a display mode from the list will, of course, have no effect on our application. We need to add a little mode code to the template app definition to enact the change of mode.

```
struct TemplateAppApp: App {

    @AppStorage("displayMode") var displayMode: DisplayMode = .auto
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            MainView()
        }
        .windowResizability(.contentMinSize)
        .defaultSize(width: Constants.mainWindowWidth, height:
Constants.mainWindowHeight)

        .onChange(of: displayMode, perform: { newVal in
            switch displayMode {
            case .light:
                NSApp.appearance = NSAppearance(named: .aqua)
            case .dark:
                NSApp.appearance = NSAppearance(named: .darkAqua)
            case .auto:
                NSApp.appearance = nil
            }
        })
```

By adding an @AppStorage variable to the app, we get access to the last stored value that the user set. If they have not set anything, then we default to .auto.

On the face of it, this works. If you go into the settings and change the appearance to Dark, the whole app will change to dark mode. There is, however, a small problem. If you close and re-open the app, it

will come back in "auto" mode. When you check the setting, it will still reflect the Dark setting, but the app will not show dark.

Initial Appearance

We can work round this quite simply. Firstly, refactor the code within the onChange to a separate method:

```
    fileprivate func setDisplayMode() {
        switch displayMode {
        case .light:
            NSApp.appearance = NSAppearance(named: .aqua)
        case .dark:
            NSApp.appearance = NSAppearance(named: .darkAqua)
        case .auto:
            NSApp.appearance = nil
        }
    }
```

We're going to need this code in two places, so it makes sense to isolate it. Next, refactor the code in the body:

```
    var body: some Scene {
        WindowGroup {
            MainView()
                .onAppear {
                    setDisplayMode()
                }
        }
        .windowResizability(.contentMinSize)
        .defaultSize(width: Constants.mainWindowWidth, height:
Constants.mainWindowHeight)

        .onChange(of: displayMode, perform: { newVal in
            setDisplayMode()
        })
```

We've refactored the .onChange modifier to call our new function. We have also added an onAppear modifier to the MainView. This means that, the first time the app is run, it will call the function to set the display appearance. So, when we close and re-open our application, the last saved value will be re-instated.

Light Mode

In Light Model, we have light windows:

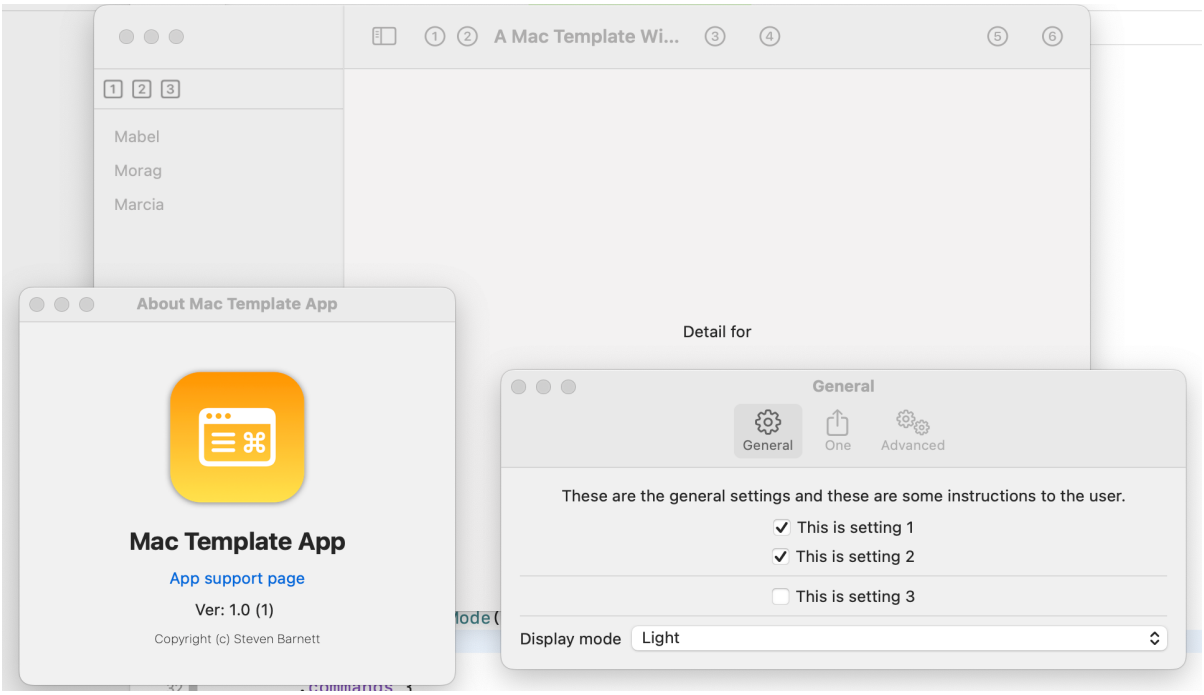


Figure 35: Light Mode Windows

Dark Mode

In dark Mode we get dark windows:

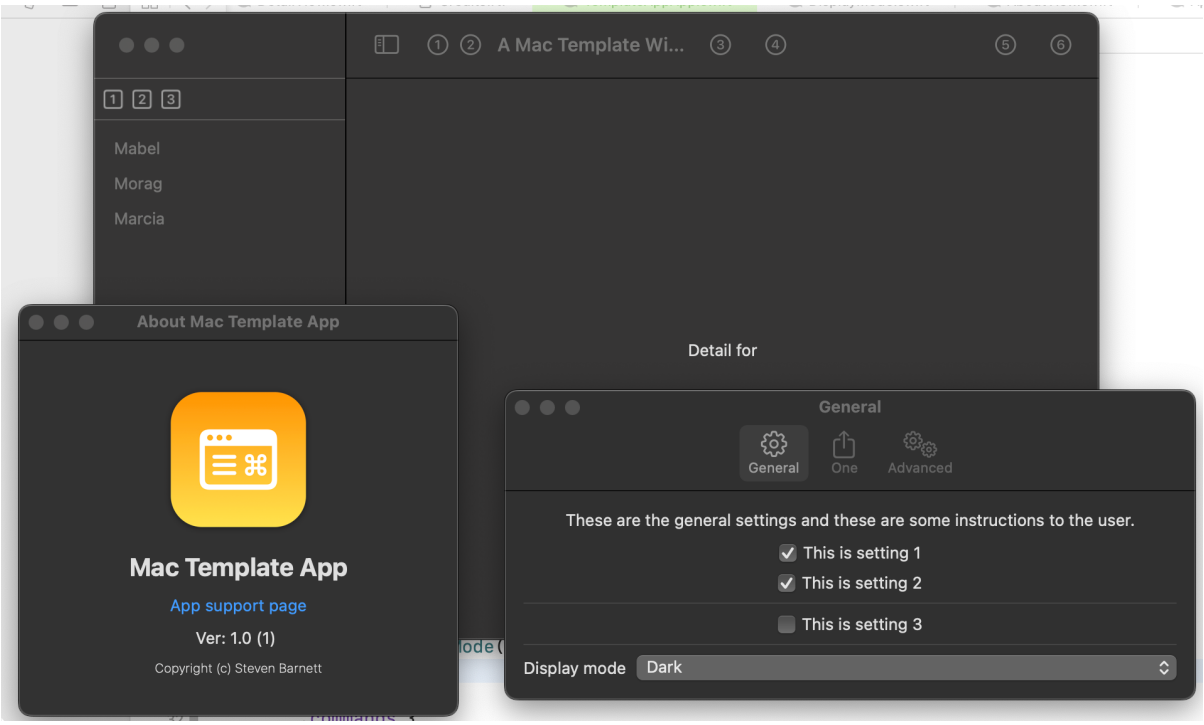


Figure 36: Dark Mode Windows

Chapter Three

The Main Window

Our main window is a little uninspiring. It is the default content that Xcode generated for us and it does nothing. Since this is a template app, we can't determine what the future needs of our app will be, but we can make some assumptions that will apply for the majority of future Mac Apps.

There is a typical layout for apps that consists of a sidebar, a main window area and an optional toolbar over the main window. We can achieve this layout relatively easily, which is the subject of this section.

The existing code

When we run our app, we get a simple main window.

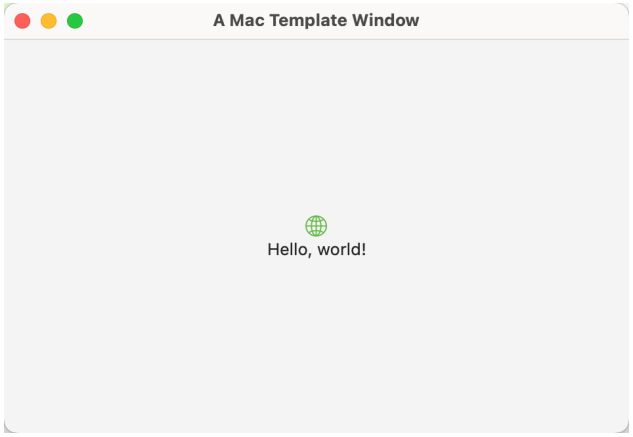


Figure 37: The Main Window

It comes with some basic functionality in that we have maximise and minimise buttons and the window is resizable. When we close the window, MacOS will save its size and position for us and will restore it the next time the window is opened. We modified the default code to give the window a default size and to set the title to appear on the title bar.

```
struct MainView: View {
    var body: some View {
        VStack {
```

```
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundColor(.accentColor)
            Text("Hello, world!")
        }
        .padding()
        .frame(minWidth: Constants.mainWindowMinWidth,
               minHeight: Constants.mainWindowMinHeight)
        .navigationTitle(Constants.mainWindowTitle)
    }
}
```

It's possible to work with this layout for many simple apps. However, our goal is to define a default layout that is more applicable to a Mac app.

Basic Layout Changes

Basic Layout Changes

With a minimum of changes, we can achieve a new look.

```
struct MainView: View {
    var body: some View {
        NavigationSplitView(sidebar: {
            Text("Sidebar")
        }, detail: {
            Text("Main Window")
        })
        .frame(minWidth: Constants.mainWindowMinWidth,
               minHeight: Constants.mainWindowMinHeight)
        .navigationTitle(Constants.mainWindowTitle)
    }
}
```

The previous VStack has been replaced with a NavigationSplitView defining the sidebar content and the main window content. We also removed the padding… it looks very odd if you leave the padding there! The resulting window looks a lot more like the result we are after:

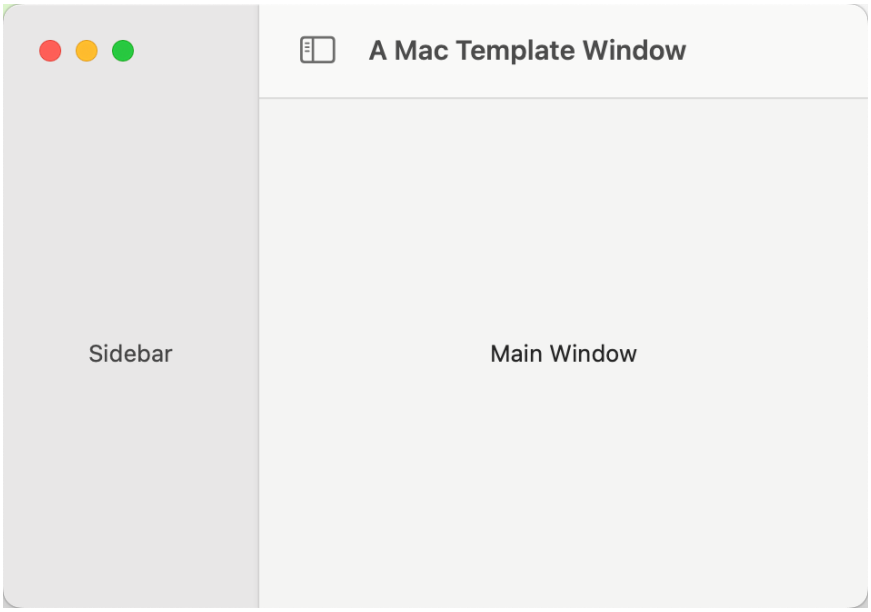


Figure 38: The Revised (basic) Main Window

We have retained the useful parts of our previous window with it's sizing abilities but we now have a sidebar and main window area and, on the window title line, we have an auto-generated icon that will hide/show the sidebar.

The Sidebar Width

The sidebar will be given a default width by the NavigationSplitView. This might be appropriate for your app, but it's quite possible that there may not be enough space to accommodate your content. Luckily, changing the size of the sidebar is just a matter of adding a frame:

```
struct MainView: View {
    var body: some View {
        NavigationSplitView(sidebar: {
            Text("Sidebar")
                .frame(minWidth: 200)
        }, detail: {
            Text("Main Window")
        })
        .frame(minWidth: Constants.mainWindowMinWidth,
               minHeight: Constants.mainWindowMinHeight)
        .navigationTitle(Constants.mainWindowTitle)
    }
}
```

With the frame added, the sidebar will be resized accordingly.

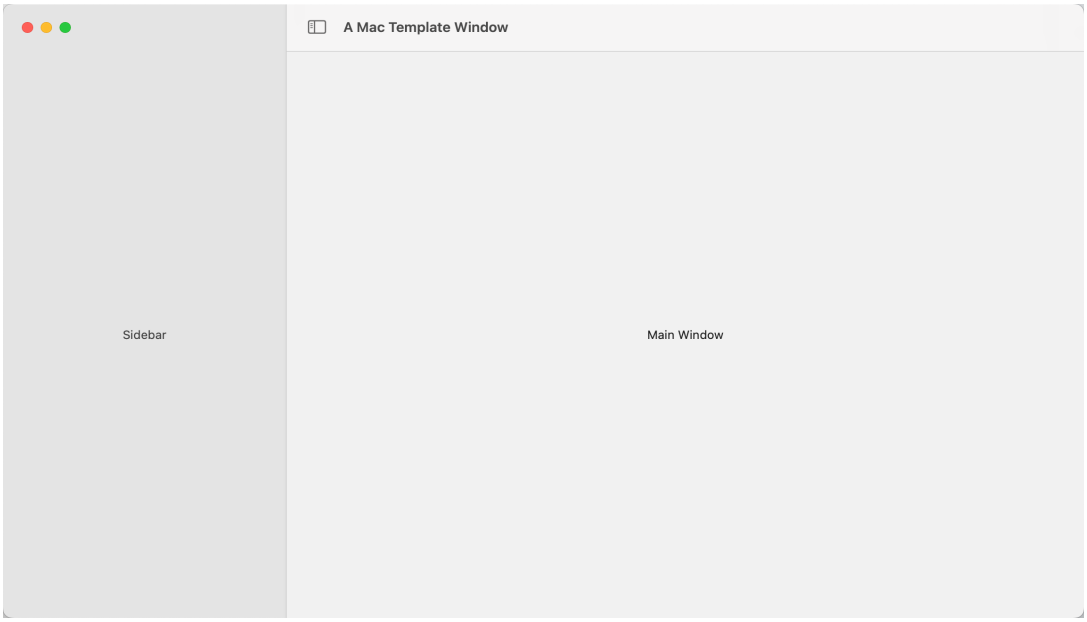


Figure 39: Wider Sidebar

This gives us the added advantage that, should the user shrink the size of the window, the sidebar will remain visible:

Figure 40: Sidebar in a shrunken window

The user can still drag the sidebar to make it wider, but they cannot make it smaller than the width you require. It's also worth noting that the sidebar size will be saved when the window is closed, so the user can adjust it to meet their needs while retaining your minimum size requirement and the operating system will remember their preference.

To remain in keeping with what we have done so far, lets replace that hard coded number with a value in our Constants file:

```
struct Constants {
    // Main window
    static let mainWindowWidth: CGFloat = 800
    static let mainWindowHeight: CGFloat = 500
    static let mainWindowMinWidth: CGFloat = 160
    static let mainWindowMinHeight: CGFloat = 160
    static let mainWindowTitle: String = "A Mac Template Window"
    static let mainWindowSidebarMinWidth: CGFloat = 200
```

We make the corresponding change to the MainView:

```
struct MainView: View {
    var body: some View {
        NavigationSplitView(sidebar: {
            Text("Sidebar")
                .frame(minWidth: Constants.mainWindowSidebarMinWidth)
        }, detail: {
            Text("Main Window")
        })
```

This just makes the code cleaner and helps remove those nasty 'magic numbers' from the code.

Window Components

Before we go too much further, we need to deal with the NavigationStackView components. It has a sidebar and it has a detail view. At the moment, we're displaying simple Text boxes:

```
struct MainView: View {
    var body: some View {
        NavigationSplitView(sidebar: {
            Text("Sidebar")
                .frame(minWidth: Constants.mainWindowSidebarMinWidth)
        }, detail: {
            Text("Main Window")
        })
        .frame(minWidth: Constants.mainWindowMinWidth,
```

```
                minHeight: Constants.mainWindowMinHeight)
        .navigationTitle(Constants.mainWindowTitle)
    }
}
```

That's fine for a very short amount of time. However, it will quickly become an issue to us as the content grows. So, now is a good time to split this up.

Firstly, in the *Views* folder, we create two new views called *SidebarView* and *DetailView* with some very basic content:

```
struct SidebarView: View {
    var body: some View {
        Text("Sidebar")
    }
}
```

and

```
struct DetailView: View {
    var body: some View {
        Text("Detail")
    }
}
```

We then modify the MainView to use these views in place of the Text views:

```
struct MainView: View {
    var body: some View {
        NavigationSplitView(sidebar: {
            SidebarView()
                    .frame(minWidth: Constants.mainWindowSidebarMinWidth)
        }, detail: {
            DetailView()
        })
        .frame(minWidth: Constants.mainWindowMinWidth,
               minHeight: Constants.mainWindowMinHeight)
        .navigationTitle(Constants.mainWindowTitle)
    }
}
```

With that simple change, we have reduced the complexity of our main window.

Toolbars

Adding A Tool Bar

In a 'real-world' app, we're unlikely to just have a simple main window. It's highly likely we're going to want to provide the user with quick ways to do things and providing that functionality generally falls to either menus or toolbars. Menus are a fairly large topic but we can provide the basics of toolbars pretty easily.

There are three areas where we can put our toolbar icons.

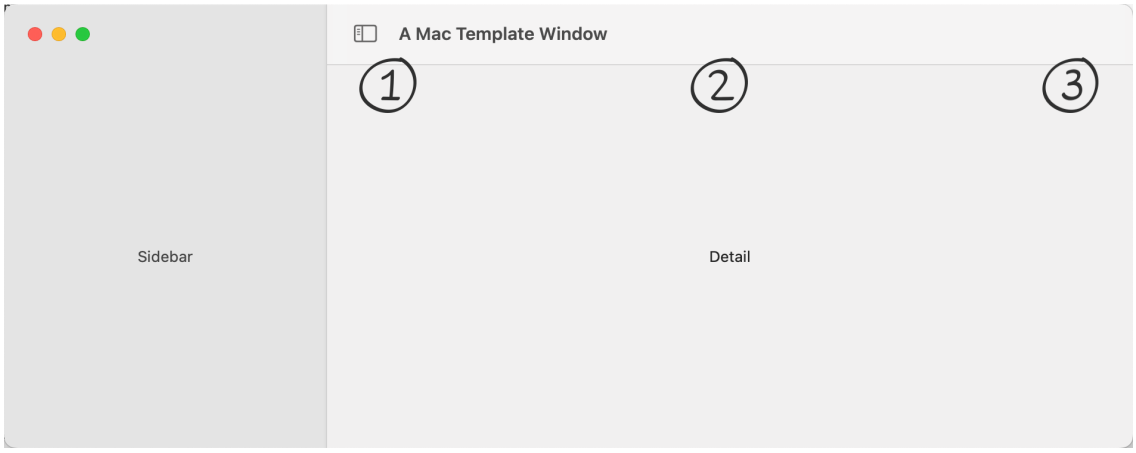


Figure 41: Toolbar icon positions

1. On the leading edge of the window.

2. In the centre of the window.

3. On the trailing edge of the window.

These are referred to as the "navigation", "principal" and "primary" areas of the toolbar. Because this is a template and we have no idea what will be required in any final app, we're going to add icons to all three locations. That's going to be done in the *DetailView* view. We're going to replace the existing Text view with a VStack and the toolbar definition.

```
struct DetailView: View {
    var body: some View {
        VStack {
            Text("Detail")
        }
        .toolbar(content: {
            ToolbarItemGroup(placement: .navigation) {
                Button(action: { print("Nav_1") },
                       label: { Image(systemName: "1.circle") })
```

```
                Button(action: { print("Nav_2") },
                       label: { Image(systemName: "2.circle") })
            }
        })
    }
}
```

Conventional wisdom has it that we put multiple toolbar buttons inside a *ToolbarItemGroup*. This results in two buttons on the toolbar in the navigation area:

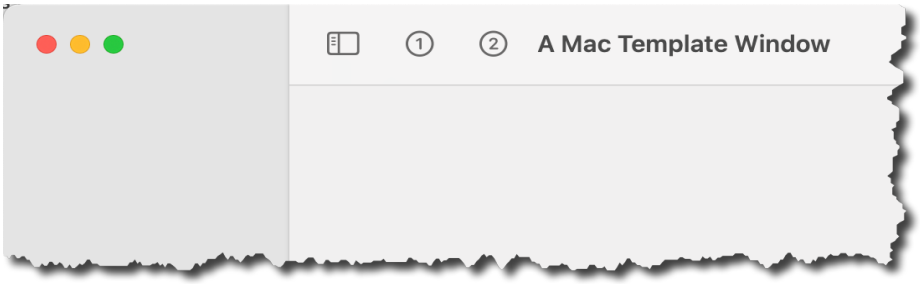


Figure 42: Navigation Area toolbar icons

That's fine and does what we asked. Personally, I dislike this result because the toolbar icons are too separated. It won't take long to fill up the toolbar with that level of spacing. Sadly, *ToolbarItemGroup* doesn't have any options for spacing, so we're stuck with it.

Except… we can work around it with a few more layout options. Let's replace the *ToolbarItemGroup* with a standard *ToolbarItem* and embed our icons in an HStack without spacing:

```
struct DetailView: View {
    var body: some View {
        VStack {
            Text("Detail")
        }
        .toolbar(content: {
            ToolbarItem(placement: .navigation) {
                HStack(spacing: 0) {
                    Button(action: { print("Nav_1") },
                           label: { Image(systemName: "1.circle") })
                    Button(action: { print("Nav_2") },
                           label: { Image(systemName: "2.circle") })
                }
            }
        })
    }
}
```

Run that and you'll see no difference. As. Last resort, we need to limit the size of the buttons to try to eliminate the spacing:

```
        .toolbar(content: {
            ToolbarItem(placement: .navigation) {
                HStack(spacing: 0) {
                    Button(action: { print("Nav_1") },
                           label: { Image(systemName: "1.circle") })
                    .frame(width: 26, height: 32)
                    Button(action: { print("Nav_2") },
                           label: { Image(systemName: "2.circle") })
                    .frame(width: 26, height: 32)
                }
            }
        })
```

And, with that change, we reduce the spacing.

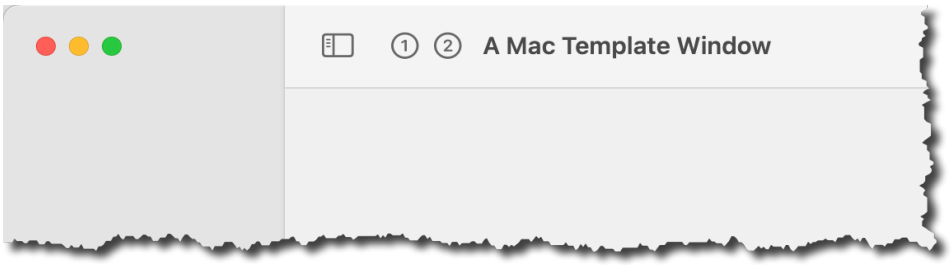


Figure 43: Navigation Area reduced spacing

It's debatable whether this is better or worse. It's non-standard and whether you use this hack will depend on how many icons you end up with. It's good to know the option exists.

Next, let's expand the toolbar to include icons in the principal (centre) area and the primary (trailing) area of the window.

```
        .toolbar(content: {
            ToolbarItem(placement: .navigation) {
                HStack(spacing: 0) {
                    Button(action: { print("Nav_1") },
                           label: { Image(systemName: "1.circle") })
                    .frame(width: 26, height: 32)
                    Button(action: { print("Nav_2") },
                           label: { Image(systemName: "2.circle") })
                    .frame(width: 26, height: 32)
                }
            }

            ToolbarItemGroup(placement: .principal) {
                Button(action: { print("Prn_1") },
                       label: { Image(systemName: "3.circle") })
                Button(action: { print("Prn_2") },
                       label: { Image(systemName: "4.circle") })
            }

            ToolbarItemGroup(placement: .primaryAction) {
                Spacer()
                Button(action: { print("Pri_1") },
                       label: { Image(systemName: "5.circle") })
                Button(action: { print("Pri_2") },
                       label: { Image(systemName: "6.circle") })
            }
        })
```

The spacing here is less of an issue, so I'm going with the defaults. When this get's run, we can see the correctly positioned toolbar icons.

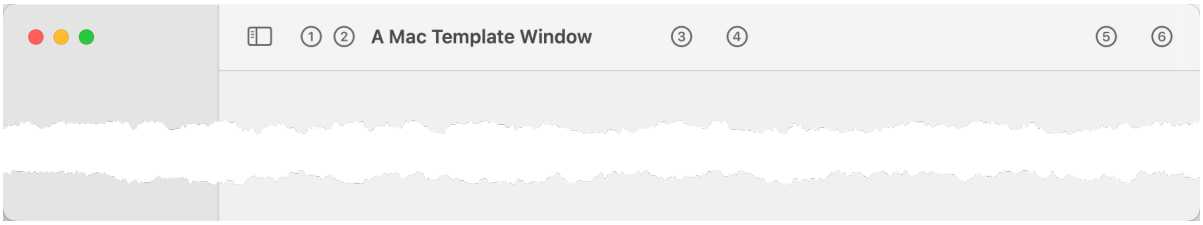


Figure 44: Principal and Primary icons

Too many icons is going to make this all look very untidy. However, we're creating a template, so the goal is to provide examples of what's possible and this code does that.

Recommendation

One recommendation if you're going to have more than a few icons on the toolbar - keep the window title short. The more text you put in the title, the more cramped the toolbar will become. If we start to run out of space, MacOS will shrink your title, adding ellipses to the end, but it's not a great look.

Better yet, have a think about whether you need the title at all!

Sidebar Toolbar

At this point you might find yourself wanting some toolbar icons in the sidebar. It's something that Xcode does and it often names sense to have sidebar specific icons. I would advise against spending a lot of time trying to code *.toolbar* modifiers in.a vain attempt to get a toolbar. I tried many, many combinations and every one of them ended with the icons added over the detail view.

The method I have chosen produces a nice toolbar like effect but without the *.toolbar* modifier. There is an argument to be had that this method could be used to create the whole app toolbar, releasing the title bar to contain just the window title. That's a better for you and your design. If you wanted a single toolbar across the window, you would just need to add this code to the main view above the N*avigationSplitView*.

Sidebar changes

Lets start with a crude change to the SidebarView code

```
struct SidebarView: View {
    var body: some View {
        VStack {
            Divider()
            HStack {
                Button(action: {
                    print("action 1")
                }, label: {
                    Image(systemName: "1.square").scaleEffect(1.3)
                }).buttonStyle(.plain)

                Button(action: {
                    print("action 2")
                }, label: {
                    Image(systemName: "2.square").scaleEffect(1.3)
                }).buttonStyle(.plain)

                Button(action: {
                    print("action 3")
                }, label: {
                    Image(systemName: "3.square").scaleEffect(1.3)
                }).buttonStyle(.plain)

                Spacer()
            }
            .padding(.vertical, 1)
            .padding(.horizontal, 8)
            .frame(maxWidth: .infinity)
            .foregroundColor(.gray)

            Divider()
```

```
            Text("Sidebar")
            Spacer()
        }
    }
}
```

The first and last *Divider()* are there to provide some separation between the title bar and the sidebar content. I've also added a *Spacer()* below the *Text* to push it to the top of the window. The toolbar is defined in the *HStack* and consists of a series of buttons. To give the HStack more of a toolbar look, I've added some padding, extended the frame to the full width of the sidebar and made the icons grey.

The result doesn't look too bad:

Figure 45: Sidebar Toolbar

The down-side is that defining even a single icon is a lot of repetitive code. We need to simplify this.

Simplified Sidebar Button

We need a new view. We could add this to the views folder but, since it will probably only ever be used in the sidebar, we can define it there. So, add this view:

```
struct ToolButton: View {

    let systemName: String
    let action: (() -> Void)

    var body: some View {
        Button(action: {
            action()
        }, label: {
            Image(systemName: systemName)
                .scaleEffect(1.3)
        }).buttonStyle(.plain)
    }
}
```

All we are doing is encapsulating the definition of a toolbar button. Importantly, it gives us a single place for the definition so we only have to make changes once if we decide to refactor our buttons. We can then change the SidebarView to redefine our buttons to use this new view.

```
            HStack {
                ToolButton(systemName: "1.square") {
                    print("1 square clicked")
                }
                ToolButton(systemName: "2.square") {
                    print("2 square clicked")
                }
```

```
                ToolButton(systemName: "3.square") {
                    print("3 square clicked")
                }

                Spacer()
            }
```

It's a little more readable and provides all the functionality we need.

Conclusion

There are a lot of other ways to achieve this effect, but this one will work for our template.

Tidy Up

We can do better!

I like to keep the content of the body as concise as possible when possible. What we have now feels wrong and will only get worse as we add more items. It takes up just too much space.

There are two solutions; we can refactor this out into another view, which will give us issues with how to communicate which icon was pressed, or we can refactor it out of the way into a function. In this case, I think a function is our way to clean up the body.

Lets refactor the code to separate out the toolbar:

```
struct SidebarView: View {
    var body: some View {
        VStack {
            toolbarIcons()

            Text("Sidebar")
            Spacer()
        }
    }

    func toolbarIcons() -> some View {
        VStack {
            Divider()
            HStack {
                ToolButton(systemName: "1.square") {
                    print("1 square clicked")
                }
                ToolButton(systemName: "2.square") {
                    print("2 square clicked")
                }
                ToolButton(systemName: "3.square") {
                    print("3 square clicked")
                }

                Spacer()
            }
            .padding(.vertical, 1)
            .padding(.horizontal, 8)
            .frame(maxWidth: .infinity)
            .opacity(0.6)

            Divider()
        }
    }
}
```

This makes our body definition so much cleaner and separates out the toolbar to a single place. While I was at it, I replaced the hard coded foregroundColor (grey) with opacity. We want the icon to be slightly faded out but don't want to dictate the colour.

Tips

One of the nice features of working on a Mac app is that we can help the user by displaying tips with our icons. We may want to use this facility or we may not. However, it's nice to have the option, so lets build that into the *ToolButton* view:

```
struct ToolButton: View {

    let systemName: String
    var helpText: String?
    let action: (() -> Void)

    var body: some View {
        Button(action: {
            action()
        }, label: {
            Image(systemName: systemName)
                .scaleEffect(1.3)
        })
        .buttonStyle(.plain)
        .help(helpText ?? "")
    }
}
```

It's a simple change. We add the helpText property and we add the *.help* modifier to display it if it's available. We can then add tips to the toolbar definition.

```
HStack {
    ToolButton(systemName: "1.square",
               helpText: "Click for option 1") {
        print("1 square clicked")
    }
    ToolButton(systemName: "2.square",
               helpText: "Click for option 2") {
        print("2 square clicked")
    }
    ToolButton(systemName: "3.square",
               helpText: "Click for option 3") {
        print("3 square clicked")
    }

    Spacer()
}
```

When the user now hovers over an icon, they will get a tip displayed that lets them know what the icon will do.

Figure 46: Toolbar Tips

Much more user friendly.

View Model

The View Model

We're almost done with the changes for our template. However, there is still one issue to address and that's how the sidebar and the detail are going to communicate. When I select something in the sidebar, how do I communicate the selection to the detail view.

Since this is a template, I have no firm data structure to deal with. So our functionality can only be generic and will definitely need to be changed when we create a 'proper' application.

To get ourselves started, let's create a new Swift file called *MainViewModel*. Now, there is a lot of debate about where this file should exist; should it be in the same folder as the view it relates to, or should it exist in a ViewModels folder, or any other number of places. The answer is that it depends; depends on the complexity of the application or the number of component views. For the sake of our template, I create a folder called *ViewModels* and create the file in there.

The content is very simple:

```
import Foundation

class MainViewModel: ObservableObject {

}
```

It's going to be a class and it's going to contain published properties, so we inherit from ObservableObject.

View Changes

We're going to need three view changes to make use of this class.

In our mainView, we define a StateObject that creates the view model:

```
struct MainView: View {

    @StateObject var vm: MainViewModel = MainViewModel()

    var body: some View {
        NavigationSplitView(sidebar: {
            SidebarView(vm: vm)
                .frame(minWidth: Constants.mainWindowSidebarMinWidth)
        }, detail: {
            DetailView(vm: vm)
        })
        .frame(minWidth: Constants.mainWindowMinWidth,
               minHeight: Constants.mainWindowMinHeight)
```

```
        .navigationTitle(Constants.mainWindowTitle)

    }
}
```

We then pass this view model to the sidebar and the detail views.

The sidebar view gets a reference to the view model:

```
struct SidebarView: View {

    @ObservedObject var vm: MainViewModel

    var body: some View {
        VStack {
            toolbarIcons()

            Text("Sidebar")
            Spacer()
        }
    }
```

Because we're being passed a StateObject instance, we code it in the child view as an @ObservedObject. We make the same change to the detail view:

```
struct DetailView: View {

    @ObservedObject var vm: MainViewModel

    var body: some View {
        VStack {
            Text("Detail")
        }
```

We can now create and set properties that we can change in the sidebar view which will trigger an update in the detail view.

Testing The View Model

While we can be sure that the view model is going to work, it's helpful to see something changing in the template that illustrates the mechanism working. We cannot predict the data type we will be dealing with, but we can produce a working example with the minimum of code.

Lets enhance our view model to include some sample data:

```
class MainViewModel: ObservableObject {

    @Published var items: [String] = ["Mabel", "Morag", "Marcia"]
    @Published var selectedItem: String = ""

}
```

We're going to have a list of names in the sidebar, as defined by the *items* array. When an item is selected, the *selectedItem* value will be changed.

```
    @ObservedObject var vm: MainViewModel

    var body: some View {
        VStack {
```

```
            toolbarIcons()

            List(vm.items, id: \.self) { item in
                Text(item)
                    .onTapGesture {
                        vm.selectedItem = item
                    }
            }
            Spacer()
        }
    }
```

The list in the sidebar is a simple list of strings and an onTapGesture that sets the selectedItem in the view model when a name is tapped.

The detail view is then updated to use the *selectedItem* property from the view model:

```
    var body: some View {
        VStack {
            Text("Detail for")
            Text(vm.selectedItem)
        }
```

Every time we tap a name in the sidebar, the detail is updated to reflect the new selection:

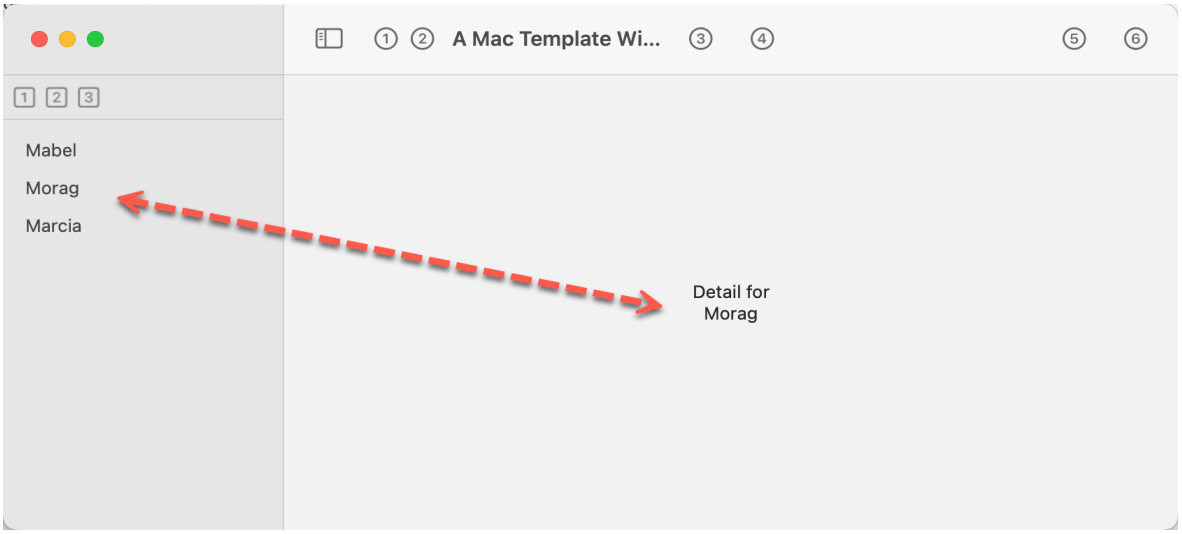


Figure 47: Testing the View Model

Chapter Four

Cleanup

Structure Cleanup

If you've followed along this far, you will see that we have introduced a couple of structure inconsistencies. They're not major, but we're at a checkpoint and it makes sense to clean them up now. It's inevitable that, as a project grows, decisions you make over structure may need to change. There is no ideal structure.

Views

The existing views folder currently contains

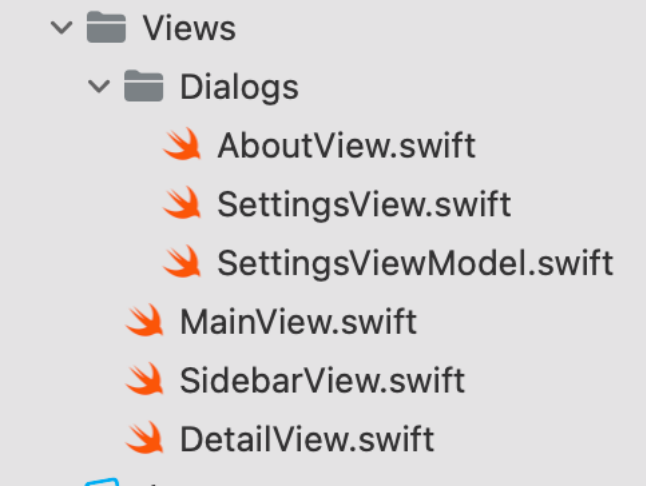


Figure 48: The Existing Views Folder

We have a sub-folder for the AboutView and the SettingsView. In the Dialogs folder, we have the View Model that corresponds to the SettingsView. My reference is to keep the view model in the same folder as the view that uses it, so this is a sensible approach for the settings.

However, we also have a view model that is associated with the MainView and we put that into a ViewModels folder. That's our first inconsistency.

So, My first fix it to group MainView, SidebarView and DetailView into a folder of their own and to move the MainViewModel in to that folder. This gives us the new structure:

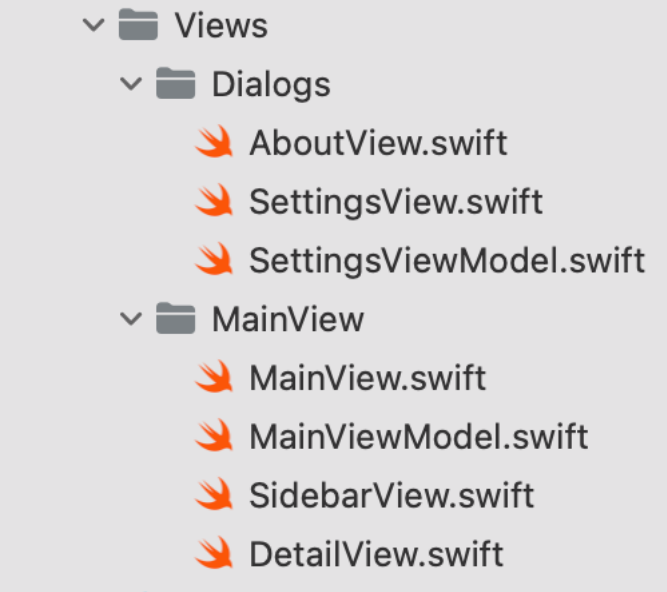


Figure 49: The Refactored Views Folder

Once the MainViewModel has been moved, the old ViewModels folder can be deleted.

Enumerations

We also created a folder for Enumerations, which currently contains a single enum:

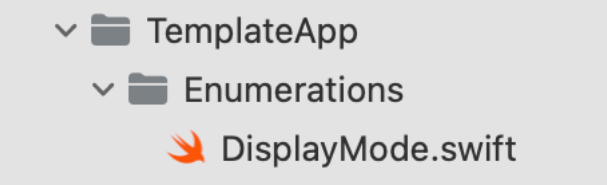


Figure 50: The Existing Enumerations Folder

That's fine, except we also defined an enum in the SettingsViewModel:

```
enum UserExperience: String, CaseIterable {
    case beginner
    case novice
    case junior
    case senior
    case lead
}
```

This needs to be refactored out into it's own file in the Enumerations folder:

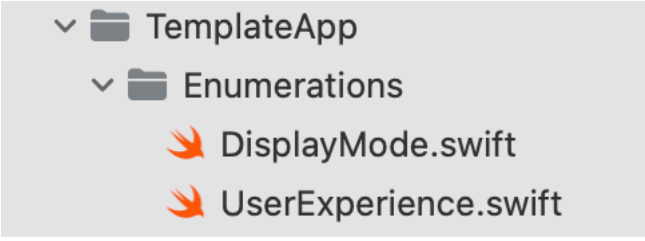


Figure 51: The Refactored Enumerations Folder

It is quite possible that this particular enum will be removed in any final app, so it makes sense to have it in a consistent place to allow us to delete the file.

Unused File

Back when we started this journey, we extended the default About window using a Credits.rtf file in the App folder:

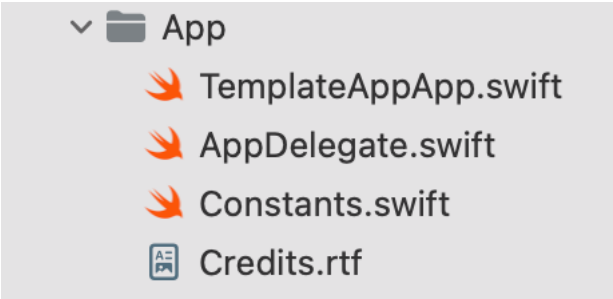


Figure 52: Redundant Credits.rtf file

Since we defined our own about dialog, this file isn't required any more, so we can delete it.

SwiftLint

SwiftLint

Linters are a matter of taste. They're not 100% necessary and they can be really frustrating at times but they also help ensure that our code remains consistent and some of the common issues that we introduce into our code can be mitigated against. So far, in our template, we have written a lot of code and, being honest, that code has been thrown in at times and refactored to arrive at an end result. It's not bad code, but it has some issues with layout.

Some of this can be addresses by SwiftLint, which you can get from https://github.com/realm/SwiftLint

How you install is entirely up to you. I chose to use the pkg file because I have had so much trouble using home-brew on an Apple Silicon machine. You may have fixed this for yourself… I hope so!

First Pass

Running SwiftLint from the command line in the root folder gives us a few issues to deal with:

Figure 53: First SwiftLint output

65 violations isn't too bad given the amount of code we've written at this point. If you checkout the issues, a lot of they are style issues to do with vertical and trailing space. It would be irritating to have to go through all of these and fix them manually, so we will let SwiftLint fix them for us.

AutoCorrect

There are a number of style fixed that SwiftLint will fix for us. All we need to do is run:

```
SwiftLint —fix
```

The result of this is:

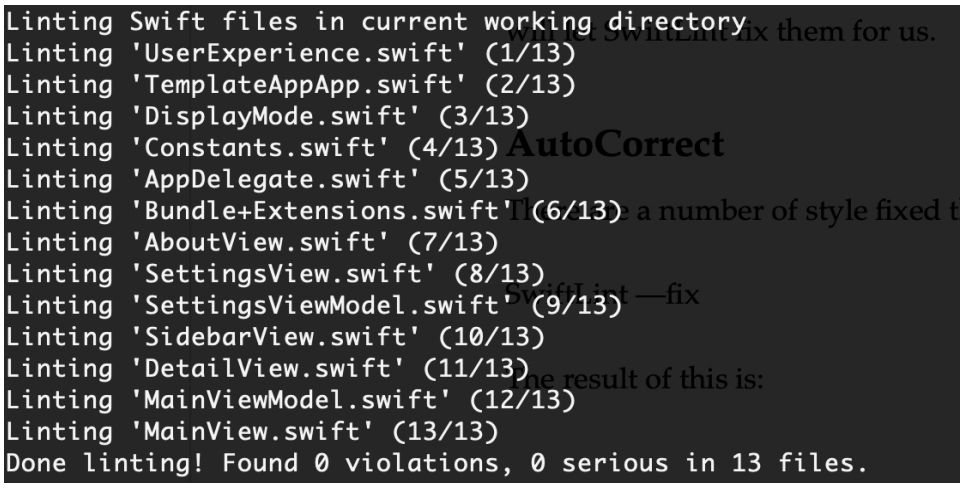


Figure 54: Fixed SwiftLint output

Well, that is a result! Turns out our code was not that bad after all.

Configuration

To be fair, we got this result with a little configuration tweaking. For example, SwiftLint imposes a minimum identifier length of 3 characters while we regularly use “vm” for the view model name. This generates a linter error. Personally, I like using “vm”, so I tweaked the configuration file to allow it.

Configuration is in the “.swiftlint.yaml” file that is in the root folder. This is mine:

```
disabled_rules: # rule identifiers to exclude from running
#  - colon
#  - comma
#  - control_statement
#  - file_length
  - force_cast
#  - force_try
#  - function_body_length
#  - leading_whitespace
#  - line_length
  - nesting
  - large_tuple
  - unused_closure_parameter
#  - opening_brace
#  - operator_whitespace
#  - return_arrow_whitespace
#  - statement_position
#  - todo
#  - trailing_newline
#  - trailing_semicolon
  - trailing_whitespace
#  - type_body_length
```

```
  - type_name
  - xctfail_message
#  - variable_name_max_length
#  - variable_name_min_length
#  - variable_name
#included: # paths to include during linting. `--path` is ignored if present. takes
precendence over `excluded`.

excluded: # paths to ignore during linting. overridden by `included`.
  - Carthage
  - Pods
  - Modules

line_length: 130

identifier_name:
  excluded: # excluded via string array
    - vm
    - id
    - URL
    - GlobalAPIKey
```

Most of the rules are commented out, but I leave them there as I can never remember the syntax! To deal with our use of "vm" I have two options; allow 2 character identifiers or exclude "vm" from the validated identifiers. Since we don't want to encourage ourselves to break the rules, I have chosen to exclude "vm" from the identifier rule.

Integrating with Xcode

Dropping to the terminal to run SwiftLint is going o get tedious very quickly. Luckily, we can get listing run on every build.

Select the project and switch to the Build Phases tab. Click the + to add a "New Run Script Phase"

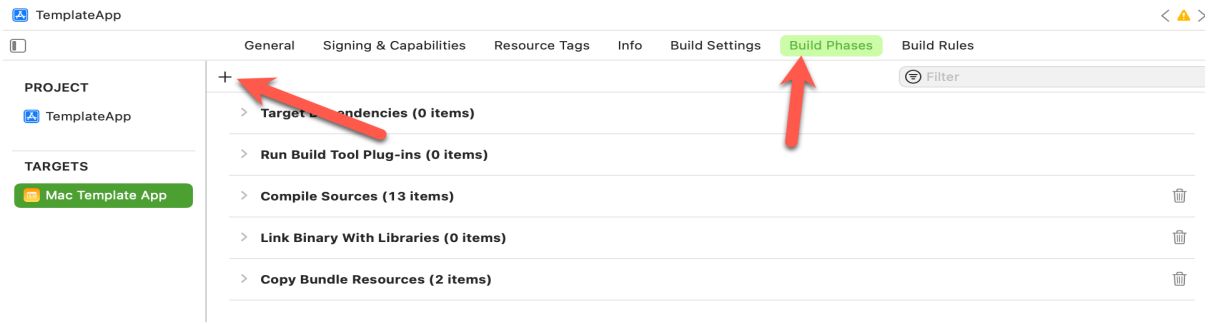


Figure 55: New run script phase to the build

When you add the new script phase, it will be added to the build process. You'll need to configure is as follows:

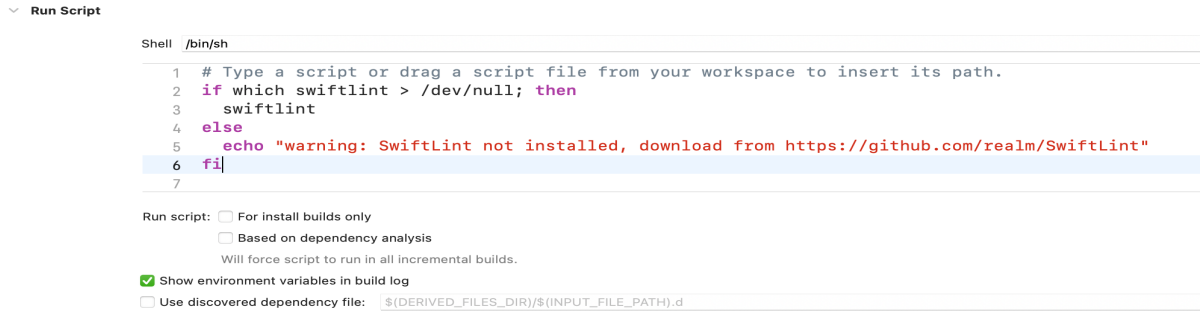


Figure 56: SwiftLint script

Now, when you build your project, SwiftLint will be run and any issues found will be flagged up right there in the source code.

Chapter Five

Menus

Introduction

Mac desktop apps have menus. They're often overlooked and rarely documented in the courses I have looked at for Mac and SwiftUI, so we need to ensure that our template has enough information to show us how to:

- Remove unused menu items.
- Rename an existing item.
- Add 'standard' menu items defined by the Mac.
- Add new menu items.
- Enable and Disable menu items depending on the app state.
- Create menu items dynamically - such as a recent files list.

This section will attempt to address these things and build examples into our template.

We already have a clue in out code because we created code to override the About box display:

```
        .commands {
            // Replace the About menu item.
            CommandGroup(replacing: CommandGroupPlacement.appInfo) {
                Button("About \(Bundle.main.appName)") {
                    appDelegate.showAboutWnd()
                }
            }
        }
```

By default an About box is created for the app which is pretty basic. We wanted full control over the information displayed so we created our own About box, coded in SwiftUI. The above code finds the supplied About menu item and replaces the text of the menu item and the handler if it is clicked.

Menu items supplied as default have an id defined in the *CommandGroupPlacement* enumeration. The specific item for the about box is the *appInfo* enum.

Menu customisation is performed in the *.commands* modifier and individual commands are part of a *CommandGroup*.

That's the basic's. More detail will follow.

Initial Setup

Before we get into the detail of menus, we need to do some initial setup. The definition of menus can get out of hand quickly and we don't want all that in out app definition file, so we're going to move the menus into a separate file of their own. We can further modularise the menus as we go along and it makes it clear where the menus are defined.

So, create a file called *Menus.swift* in the App folder. I've put it in the app folder because menus tend to be over all windows so are more appropriate at the app level.

The content of the menus file will be based on the existing About box menu item. It gives us a starting point:

```
import SwiftUI

struct Menus: Commands {

    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Commands {
        // Replace the About menu item.
        CommandGroup(replacing: CommandGroupPlacement.appInfo) {
            Button("About \(Bundle.main.appName)") {
                appDelegate.showAboutWnd()
            }
        }
    }
}
```

The menus are treated in a similar way to views, except that the base is *Commands* instead of *Views*. As with Views, we have to conform to the Commands protocol by creating a body property which returns *some Commands*. In that body, we return the code we previously had in the app.

Since we're going to need access to the appDelegate, we move the definition into our menus file.

The app definition is then simplified to:

```
struct TemplateAppApp: App {

    @AppStorage("displayMode") var displayMode: DisplayMode = .auto

    var body: some Scene {
        WindowGroup {
            MainView()
                .onAppear {
                    setDisplayMode()
                }
```

```
        }
        .windowResizability(.contentMinSize)
        .defaultSize(width: Constants.mainWindowWidth, height:
Constants.mainWindowHeight)

        .onChange(of: displayMode, perform: { _ in
            setDisplayMode()
        })

        .commands {
            Menus()
        }

        Settings {
            SettingsView()
        }
    }
```

We're removed the appDelegate definition and have replaced the menu with Menus() to load our new file.

Replacing Menu Items

Replacing Menu Items

Since we have already done it, let's get replacing existing menu items out of the way. Taking a look at the *Menus.swift* file, we see:

```
import SwiftUI

struct Menus: Commands {

    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Commands {
        // Replace the About menu item.
        CommandGroup(replacing: CommandGroupPlacement.appInfo) {
            Button("About \(Bundle.main.appName)") {
                appDelegate.showAboutWnd()
            }
        }
    }
}
```

The *CommandGroup* statement is what adds or replaces a menu item. In this incarnation, we have initialised the *CommandGroup* with the *replacing* argument. This specifies which of the existing menu items we want to replace.

You'll need to check the documentation for *CommandGroupPlacement* to see what can be overridden in this way.

The content of the closure defines the content of the menu. Here we have chosen t display a button and make the action of the button display the about box. You can, however, display other controls, such as a Toggle or a Picker.

Adding Menu Items

Standard Menus

There are a number of standard menus that provide us with additional menu items for free. These are coded in to the system for us to add if we need the specific functionality. There are five groups to pick from:

- SidebarCommands()
- ToolbarCommands()
- TextEditingCommands()
- TextFormattingCommands()
- EmptyCommands()

SidebarCommands adds a menu item to toggle the sidebar. We already have a toolbar icon for that and the menu item replicates that functionality for those of us who like menus.

ToolbarCommands adds menu items to toggle the toolbar on/off and to customise the toolbar if you have enabled that functionality. I have not covered that in this document, but you can google it.

TextEditingCommands adds menu items relating to text editing, such as find and spell check (there are many more).

TextFormattingCommands adds commands dealing with text formatting, such as fonts.

EmptyCommands is a special case that does absolutely nothing.

To use the commands, just add them to the menu definition:

```
struct Menus: Commands {

    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Commands {
        ToolbarCommands()
        SidebarCommands()

        // Replace the About menu item.
        CommandGroup(replacing: CommandGroupPlacement.appInfo) {
            Button("About \(Bundle.main.appName)") {
                appDelegate.showAboutWnd()
            }
        }
    }
}
```

When we run our app, we get the new functionality added to the menus.

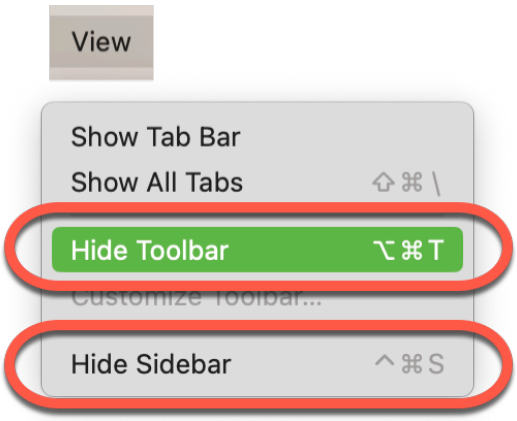


Figure 57: Standard Menus

With the sidebar and toolbar toggled off, the menu automatically changes the text of the menu items to a more appropriate value:

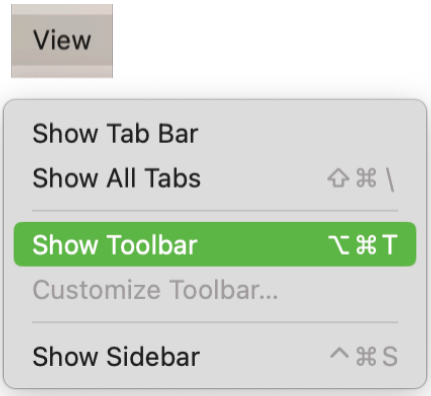


Figure 58: Toolbar and Sidebar toggle

The resulting window now does not have a sidebar or a toolbar:

Figure 59: Window with Toolbar and Sidebar Toggled Off

At this point, you can see the usefulness of the sidebar toggle menu… with the toolbar dismissed there is no other way to toggle the sidebar.

Add a New Menu

Let's start big. Let's add a completely new menu across the top of the menu bar. Out existing menu consists of:

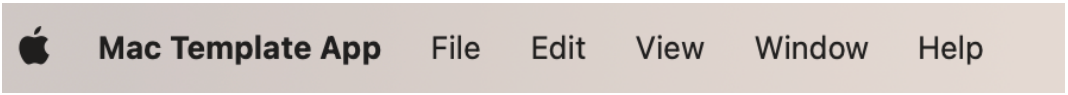


Figure 60: Our initial menu

There is a. Lot of functionality tied up in this menu and most of it is handled automatically for us. When our app evolves, though, we're going to want to add new menus with custom items. The new menu may have one or a dozen different items and sub-items, so we need a quick and easy way to create it. That's where *CommandMenu* comes in to play.

In the *Menus.swift* file, add the following right after our About Box CommandGroup:

```
        CommandMenu("Display") {
            Button("Item 1") {
                print("Item 1 selected")
            }
            Button("Item 2") {
                print("Item 2 selected")
            }
            Divider()
            Button("Item 3") {
                print("Item 3 selected")
            }
        }
```

When we run the app, we'll get a new menu called *Display* at the top with three sub-menu items:

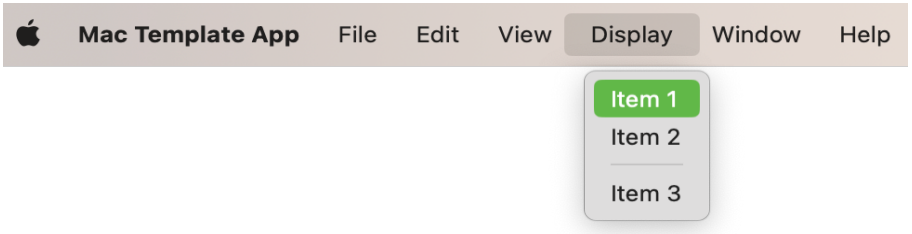


Figure 61: Our Display Menu

This code illustrates adding simple buttons and placing a divider between options. It's the simplest form of new menu.

Adding a Toggle

Now we have a new menu, let's move things up a bit and add a toggle switch. Back when we created the Settings form, we created some toggle settings. These were defined in the *SettingsViewModel*.

```
class SettingsViewModel: ObservableObject {

    // General tab options
    @AppStorage("setting1") var toggle1: Bool = false
    @AppStorage("setting2") var toggle2: Bool = false
    @AppStorage("setting3") var toggle3: Bool = false

}
```

Lets add a toggle to our new menu for "setting 1". First thing we need is an @AppStorage for the item we want to change, so add this to the top of the Menus file:

```
struct Menus: Commands {

    @AppStorage("setting1") var toggle1: Bool = false
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate
```

If the menu toggles this value, all references will also change, so it's instant feedback. Add this to the new menu definition:

```
        CommandMenu("Display") {
            Button("Item 1") {
                print("Item 1 selected")
            }
            Button("Item 2") {
                print("Item 2 selected")
            }
            Divider()
            Button("Item 3") {
                print("Item 3 selected")
            }

            Divider()
            Toggle(isOn: $toggle1) { Text("Setting 1") }
        }
```

When you run the app, you'll get the Setting 1 menu item:

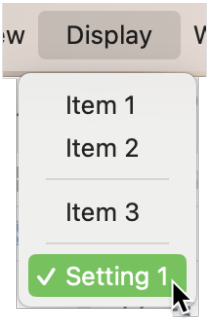


Figure 62: Setting 1 Toggle Menu Item

If you click the item, the check mark will disappear. It works exactly as you would expect; it toggles the setting on and off. You can validate this by looking at the settings dialog as the menu is connected to Setting 1 on the general tab. Whatever state you set on the menu will be reflected in the settings. Similarly, when you set the state of Setting 1 in the Settings dialog, its state will be reflected in the menu.

Picker

The final option we want to add to our sample menu is a picker.

We have a setting that allows us to toggle the display mode between light and dark mode. It presents a drop down selector for Light, Dark or Auto mode. What we're going to replicate here is the same ability to change display mode but via a menu.

As with the settings, we need a connection to AppStorage, so add the following to the menu:

```
struct Menus: Commands {

    @AppStorage("displayMode") var displayMode: DisplayMode = .auto
    @AppStorage("setting1") var toggle1: Bool = false
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate
```

The DisplayMode enum is as we defined it previously. We then extend the CommandMenu t include a picker control:

```
        CommandMenu("Display") {
            Button("Item 1") {
                print("Item 1 selected")
            }
            Button("Item 2") {
                print("Item 2 selected")
            }
            Divider()
            Button("Item 3") {
                print("Item 3 selected")
            }

            Divider()
            Toggle(isOn: $toggle1) { Text("Setting 1") }

            Divider()
            Picker(selection: $displayMode, content: {
                ForEach(DisplayMode.allCases) { mode in
                    Text(mode.description).tag(mode)
                }
            }, label: {
                Text("Display mode")
            })
        }
```

This is exactly the same picker that we used in the settings to display the three options. It results in a sub-menu for setting the display mode:

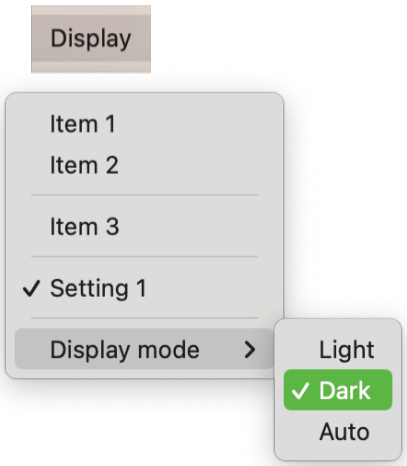


Figure 63: Display Mode menu

Whichever menu item you select the display will instantly change to reflect the option you pick. This is happening because of the *onChange* that was coded in the App which is triggered when you change the menu.

Conclusion

Overall, for a minimal amount of code, we have obtained a useful new menu.

Adding To Existing Menus

We've already seen how to replace a menu item in Replacing Menu Items. That used the *CommandGroup* control with the *replacing* initialiser. CommandGroup has two other initialisers that allow you to add a new menu item before or after an existing menu item.

- CommandGroup(after:, addition:)
- CommandGroup(before:, addition:)

Let's add a new item called *New from template* that new would use to create a new file from a template file. Adding this menu item is pretty trivial (well, the menu item is):

```
    var body: some Commands {
        ToolbarCommands()
        SidebarCommands()

        // Replace the About menu item.
        CommandGroup(replacing: CommandGroupPlacement.appInfo) {
            Button("About \(Bundle.main.appName)") {
                appDelegate.showAboutWnd()
            }
        }

        CommandGroup(after: CommandGroupPlacement.newItem) {
            Button("New from template") {
                print("New from template")
            }
        }
```

When this code is run, we get a new item on the menu after the File->New menu:

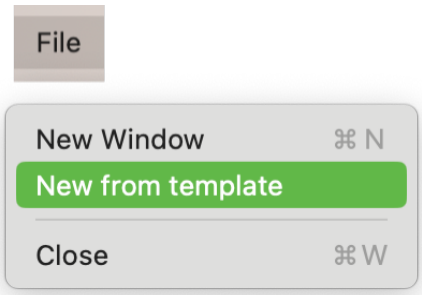


Figure 64: Add Menu Item After

In a similar way, we can add an option after an existing menu item:

```
        // Replace the About menu item.
        CommandGroup(replacing: CommandGroupPlacement.appInfo) {
            Button("About \(Bundle.main.appName)") {
                appDelegate.showAboutWnd()
            }
        }

        CommandGroup(after: CommandGroupPlacement.newItem) {
            Button("New from template") {
                print("New from template")
            }
        }

        CommandGroup(before: CommandGroupPlacement.appTermination) {
            Button("Discard Changes") {
                print("Discard Changes")
            }
        }
```

In this case, we have added a menu called *Discard Changes* to appear before the app close menu:

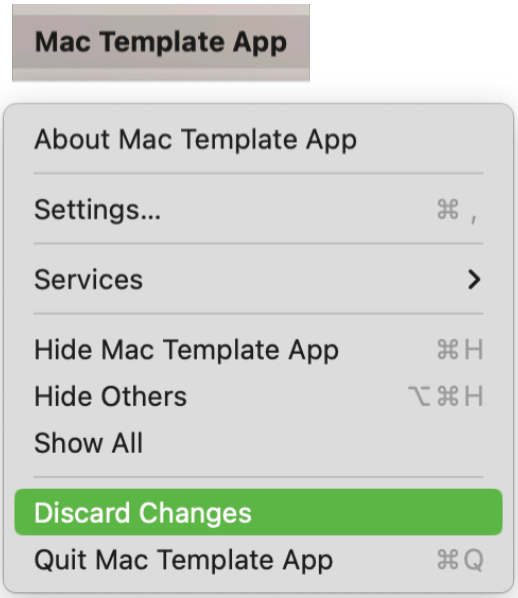


Figure 65: Add Menu Item Before

Web link

One useful example we can add to our template that will be useful for more than just place holding is a link to our product web site from the help menu. We already have a link to the hone web page that we use for the About Box defined in our *Constants* file:

```
    // About box
    static let homeUrl: URL = URL(string: "http://www.sabarnett.co.uk")!
    static let homeAddress: String = "App support page"
```

So, we can re-use this from our menu.

```
        CommandGroup(replacing: CommandGroupPlacement.appInfo) {
            Button("About \(Bundle.main.appName)") {
                appDelegate.showAboutWnd()
            }
        }

        CommandGroup(before: CommandGroupPlacement.help) {
            Link(Constants.homeAddress,
                 destination: Constants.homeUrl)
            Divider()
        }

        CommandGroup(after: CommandGroupPlacement.newItem) {
            Button("New from template") {
                print("New from template")
            }
        }
```

Adding this new command group adds a new option to the help menu that, when clicked, takes the user to your web site:

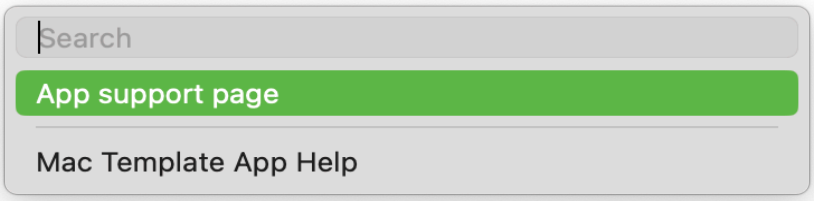


Figure 66: Link to a web site

Removing Standard Menu Items

Remove A Menu Item

It's good that we're provided with so many menu items for free. But what if there are menu items that we just do not want. How can we remove them?

This is where the EmptyCommands() built in group we met in standard menus comes to our rescue. Coupled with a CommandGroup replacing, we can remove a menu item.

On the Edit menu, you can see two options for Undo/Redo.

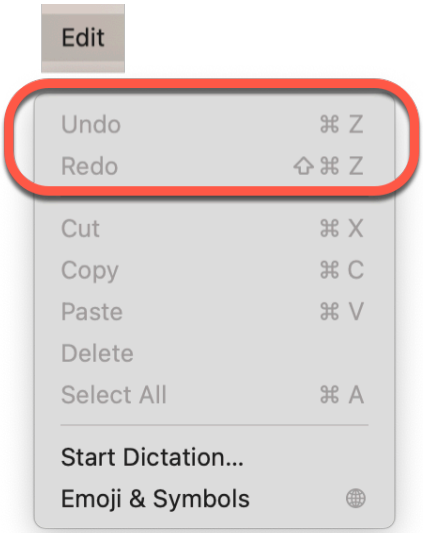


Figure 67: Remove a Menu Item

Let's suppose our app doesn't support undo/redo, then these menu items should not appear. We need to remove them:

```
        CommandGroup(replacing: CommandGroupPlacement.undoRedo) {
            EmptyView()
        }
```

There is a standard group that defines the undo/redo menu items called *.undoRedo*. To remove the menu, we just replace this group with an empty view. Because there is no content, the menu will not appear:

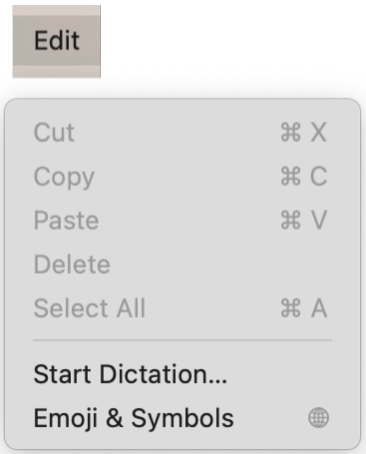


Figure 68: Undo/Redo removed

A Little Refactoring

Refactoring The Menus.

Menu structures, like views, have a limit to the number of items you can put at the top most level. We've been adding a lot of menus items and we're in danger of hitting that limit quite quickly. So, we're going to have an interlude here to split our menu into smaller pieces to avoid any future issues.

Lets start by refactoring the *Display* menu into a separate structure:

```
public struct DisplayCommands: Commands {

    @AppStorage("displayMode") var displayMode: DisplayMode = .auto
    @AppStorage("setting1") var toggle1: Bool = false

    public var body: some Commands {
        CommandMenu("Display") {
            Button("Item 1") {
                print("Item 1 selected")
            }

            Button("Item 2") {
                print("Item 2 selected")
            }

            Divider()
            Button("Item 3") {
                print("Item 3 selected")
            }

            Divider()
            Toggle(isOn: $toggle1) { Text("Setting 1") }

            Divider()
            Picker(selection: $displayMode, content: {
                ForEach(DisplayMode.allCases) { mode in
                    Text(mode.description).tag(mode)
                }
            }, label: {
                Text("Display mode")
            })
        }
    }
}
```

Now we have this in a separate struct, we can replace the code in the menu with a call to DisplayCommands:

```
        CommandGroup(replacing: CommandGroupPlacement.undoRedo) {
            EmptyView()
        }
```

```
        DisplayCommands()
```

The entire Display menu snow in its own structure allowing us to work on it in isolation. Ok, since we're using a CommandMenu, we're actually replacing a single *CommandMenu* with a single *DisplayCommands*, but the code is structured so much better.

In a later section, we're also going to extend the File menu and they are going to be separate menu items, so now is also a good time to refactor the File menu out too.

```
public struct FileCommands: Commands {

    public var body: some Commands {

        CommandGroup(after: CommandGroupPlacement.newItem) {
            Button("New from template") {
                print("New from template")
            }
        }

    }
}
```

This is then referenced in the menus in the same way as the DisplayCommands.

```
        CommandGroup(before: CommandGroupPlacement.help) {
            Link(Constants.homeAddress,
                 destination: Constants.homeUrl)
            Divider()
        }

        FileCommands()

        CommandGroup(before: CommandGroupPlacement.appTermination) {
            Button("Discard Changes") {
                print("Discard Changes")
            }
        }
```

These changes, while they appear minor at this stage, will put us in a better position for the next couple of sections.

Connecting a Menu to your View

Connecting Menus to Views

Everything so far has either acted at a global level or has been hooked up to an AppStorage setting. For the most part, this is sufficient for simple apps. However, there will inevitably come a time when you need to initiate an action in a view when you select a menu item.

There is a mechanism built in that will do some of this for you based around *@FocusedBinding* and *.focusedValue*. The mechanism allows you to connect a value maintained in the app/menu to a bound value in the currently focused view. That works great for binding a value (though I have had some issues with it properly determining which window is focused). It does not help us if what we want to do is run a function in the View Model.

For binding to a View Model, we will need a slightly different mechanism based around *@FocusedObject* and *.focusedSceneObject*.

We want to achieve two things from this code:

- A way to tell a view model when a menu item is clicked.
- A way for the view model to tell the menu when items should be enabled or disabled.

We added three sample menu items earlier to a menu called *Display* and we will be adding code that allows the currently active view model to work with these three menu items:

```
        CommandMenu("Display") {
            Button("Item 1") {
                print("Item 1 selected")
            }

            Button("Item 2") {
                print("Item 2 selected")
            }

            Divider()
            Button("Item 3") {
                print("Item 3 selected")
            }
        }
```

Menu Handler Protocol

We're going to be accessing our view model in the menus. That's fine, but we want some delineation between the view model and the menu specific functions. We are forced to use the full view model in the menus, but I want to clearly show what functions are specific to menus, even if it's more of a convenience and documentation thing. So, we're going to start our menu handler with a protocol:

Create a group called *Protocols* and add a file called *MenuHandlerProtocol* with this content:

```
import SwiftUI

protocol MenuHandlerProtocol {
    func item1MenuClick()
    func item2MenuClick()
    func item3MenuClick()
}
```

In our custom menu we defined three menu items that we want to handle in the currently active view. What we have defined here are the names of the handlers that will respond when we click one of those menu items.

Updating our View Model

Before we can connect the view to the menus, we need to add to our view model. As stated previously, we could just code these handlers, but we chose to define a protocol to ensure that we always code everything that we need. So we need to add an extension to the view model to implement the menu protocol:

```
import Foundation

class MainViewModel: ObservableObject {

    @Published var items: [String] = ["Mabel", "Morag", "Marcia"]
    @Published var selectedItem: String = ""

}

// Handlers for the menus we added.
extension MainViewModel: MenuHandlerProtocol {

    func item1MenuClick() {
        print("Item 1 click handler called")
    }

    func item2MenuClick() {
        print("Item 2 click handler called")
    }

    func item3MenuClick() {
        print("Item 3 click handler called")
    }
}
```

I've chosen to use an extension here as it concentrates the menu handlers into a single place. We conform to the *MenuHandlerProtocol* which means we must define three functions to handle the three menu items we defined in the protocol. Because the code is being called in the view model, we have

access to any functionality that the view model contains and can set any published properties to cause the view to update.

Making the Connection

There are two aspects of making the connection. The first is to define the "menu" end of the connection. We're gong to do this using *@FocusedObject*.

Add this to the top of the DisplayCommands struct:

```
public struct DisplayCommands: Commands {

    @FocusedObject private var menuHandlers: MainViewModel?

    @AppStorage("displayMode") var displayMode: DisplayMode = .auto
    @AppStorage("setting1") var toggle1: Bool = false

    public var body: some Commands {
```

The @FocusedObject variable is our bridge between the system level menus and the view specific view model. When the focus switches between windows, the *menuHandlers* variable will be updated to reference the view model of the currently active window. It needs to be defined as an optional because it is possible that the focused window will not have a view model of the correct type or there may not be a focused view.

We can now update the menu handlers to call the functions we defined in the view model:

```
        CommandMenu("Display") {
            Button("Item 1") {
                menuHandlers?.item1MenuClick()
            }

            Button("Item 2") {
                menuHandlers?.item2MenuClick()
            }

            Divider()
            Button("Item 3") {
                menuHandlers?.item3MenuClick()
            }
```

If we run this now, nothing is going to happen, as we only have half of the equation in place. Next, we need to define the connection at the View end.

The view exposes its view model using the *.focusedSceneObject* modifier. This is a simple change to the view:

```
    var body: some View {
        NavigationSplitView(sidebar: {
            SidebarView(vm: vm)
                .frame(minWidth: Constants.mainWindowSidebarMinWidth)
        }, detail: {
            DetailView(vm: vm)
        })
        .frame(minWidth: Constants.mainWindowMinWidth,
               minHeight: Constants.mainWindowMinHeight)
        .navigationTitle(Constants.mainWindowTitle)
        .focusedSceneObject(vm)
```

focusedScemeObject identifies the object that we want to share, so exposes our view model.

Once we have this connection, the DisplayCommands menu will be connected to whichever view is being displayed that exposes its view model of the correct type.

Now, when we click on the Item 1/2/3 menus, the corresponding function will be called in the view model and the message will be displayed on the console.

Enabling/Disabling Menu Items

Enabling and Disabling Menu Items

At this stage, we have almost all of the basic functionality we need to deal with menus. We have learnt how to create and remove menus and how to provide a custom handler for a menu in our view model. This is all great, but there is one issue left that we need to address in our template.

Menu items will not always be appropriate and allowing the user to click them when they have no context to work with is not a good user experience. We need a way to control whether a particular item is enabled or not.

We could take the same approach as we defined for handling menu items and create a function in the protocol for each menu item, but that's probably going to introduce a lot of functions. That can quickly get out of hand. While a separate click handler makes sense, we need something simpler for the enable/disable code.

Identifying Menu Items

We need to start with a way to identify menu items when we try to determine if they are enabled or disabled. We'll do that with a simple enum. So, create a file called *CustomMenuItems* in the enums folder:

```
import Foundation

enum CustomMenuItems {
    case menuItem1
    case menuItem2
    case menuItem3
}
```

We have one entry for each of the menu items that we want to handle clicks for and that we need to be able to enable/disable.

Extending The Menu Protocol

In order to connect the menu to the view model, we need to extend the protocol:

```
import SwiftUI

protocol MenuHandlerProtocol {
    func item1MenuClick()
    func item2MenuClick()
    func item3MenuClick()

    func isMenuDisabled(_ menuItem: CustomMenuItems) -> Bool
}
```

The SwiftUI modifier for enabling/disabling menu items is *.disabled*, so we have added a function that returns whether a particular item is disabled or not. If you compile this, it will fail because we need to add the implementation of the method to our MainView extension:

```
// Handlers for the menus we added.
extension MainViewModel: MenuHandlerProtocol {

    func item1MenuClick() {
        print("Item 1 click handler called")
    }

    func item2MenuClick() {
        print("Item 2 click handler called")
    }

    func item3MenuClick() {
        print("Item 3 click handler called")
    }

    func isMenuDisabled(_ menuItem: CustomMenuItems) -> Bool {
        // TODO: Logic to decide whether menu items are disabled
        switch menuItem {
        case .menuItem1:
            return true
        case .menuItem2:
            return false
        case .menuItem3:
            return false
        }
    }
}
```

At this point, we can't possibly know what the logic is for disabling a menu item, so we have hard coded the return values. We have gone with the first item being enabled and the remaining two being disabled.

Disabling Menu Items

The final task is to modify the *Menus* file to include the logic for disabling menu items that should be disabled. That's a straight forward change:

```
        CommandMenu("Display") {
            Button("Item 1") {
                menuHandlers?.item1MenuClick()
            }.disabled(menuHandlers?.isMenuDisabled(.menuItem1) ?? true)

            Button("Item 2") {
                menuHandlers?.item2MenuClick()
            }.disabled(menuHandlers?.isMenuDisabled(.menuItem2) ?? true)
```

```
            Divider()
            Button("Item 3") {
                menuHandlers?.item3MenuClick()
            }.disabled(menuHandlers?.isMenuDisabled(.menuItem3) ?? true)
```

If we have an active view model, we will call it to enable or disable the menu items using the enum we created earlier.

Cleanup

Cleanup

We've written a lot of code and have achieved a lot of functionality. There are still things we could do to extend menus and menu handling, but they're not as generic as the functionality we have just added, so can be left to development in a 'real' app. At least we have the infrastructure to be able to extend the functionality as needed.

One thing remains that does bother me though. There are several files involved in defining a new custom menu item and remembering what has to change to add a new item could be problematic in a month or two when we next use the template. So, I feel the need to add some instructions to our template to help 'future me'.

Commenting

Commenting is a decisive subject. There are those at one extreme who firmly believe that the code is (should be) self documenting and there is no need to add additional comments to the code. Then, there are those that comment almost every line in the code in an effort to ensure that another developer will know their intent.

Personally, I prefer a happy medium between the two; the code should be self documenting by the use of properly crafted naming of structures, classes, variables and functions. Comments should be reserved for explaining complex logic or for documenting framework libraries and API's.

I also have one additional exception; templates. As much of the template should be self documenting as possible. However, where there are interdependencies between files that might trip up the template user, there needs to be documentation built in to the template to guide the developer.

New Menu Item Documentation

Adding a menu and its menu items is not complex, but involves updating several files. We need to ensure that all files are updated in order to avoid compile errors. Luckily, if you miss a file the code should not compile, so we have a cross check. However, a little guidance won't go amiss.

For our purposes, I have added the following comments to the menus file:

```
    public var body: some Commands {
        /* Adding Menu Items

         Requirements for a new menu item:
```

```
         1. Add a button to the menus file.
         2. Update the MenuHandlerProtocol file to add a click handler if you
            expect the view model to handle the click.
         3. Update the MainViewModel extension to add a click handler.
         4. Update the CustomMenuItems enum file to add an id for the new menu
            item if you expect to view model to be able to eable/disable the menu
item.
         5. Update the MainViewModel extension to handle enabling/disabling the
menu item.

         Files to update:
         * Menus.swift
         * MenuHandlerProtocol.swift
         * MainViewModel.swift

         */
        CommandMenu("Display") {
            Button("Item 1") {
                menuHandlers?.item1MenuClick()
            }.disabled(menuHandlers?.isMenuDisabled(.menuItem1) ?? true)
```

It's not a lot of commentary, but serves as a check-list of files to update.

Chapter Six

File Handling

There is a lot available to us in SwiftUI. If, however, you want to deal with files and folders, you're going to have to resort to AppKit as your API of choice. Not all Mac applications are going to need to interact with files selected by the user, so it may not be a problem. However, if you want to open a file, save (as) a file or select a folder, you're out of luck with SwiftUI.

This section presents a file class for displaying open/save dialogs and for presenting a folder selection dialog. Basic functionality wrapped up in a class that you can quickly delete if you don't need it.

For testing purposes, we'll also add a new menu for File-> Open, one for File-> SaveAs and one for File->Select Folder. They aren't going to be connected to a view model and they're not going to do any more than display the system popup's, as more than that os application specific.

Testing Menu Items

In order to add our testing menu items, we need to add three new items focused around the existing "new from template" menu option. We created a FileCommands structure earlier, so our additions go in there:

```
public struct FileCommands: Commands {

    public var body: some Commands {

        CommandGroup(before: CommandGroupPlacement.newItem) {
            Button("Open") {
            }
            Divider()
        }

        CommandGroup(after: CommandGroupPlacement.newItem) {
            Button("New from template") {
                print("New from template")
            }
        }

        CommandGroup(after: CommandGroupPlacement.newItem) {
```

```
            Divider()
            Button("Save-As") {
            }
        }

        CommandGroup(after: CommandGroupPlacement.newItem) {
            Divider()
            Button("Select Folder") {
            }
        }
    }
}
```

Nothing revolutionary here, but it gives us a way to test the file functionality we're about to add.

With this in place, run the code and take a look at the file menu:

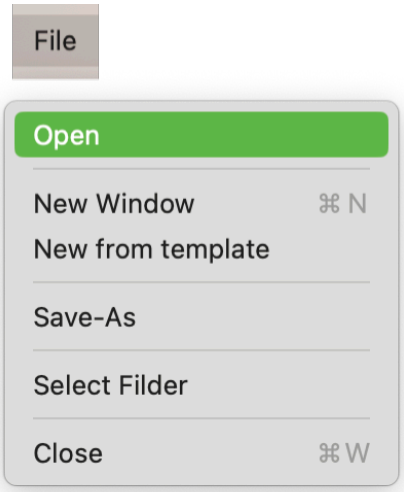


Figure 69: The new File menu items

Starter File

We're going to build our file functionality into a helper file. So, start by creating a new top level folder called *Utilities* and create a swift file in there called *FileHelpers*. In there, we can define the starter struct:

```
import Foundation
import AppKit
import UniformTypeIdentifiers

struct FileHelpers {

}
```

We're going to need to import AppKit because we will need access to the NSApplication object for our app. UniformTypeIdentifiers came in after macOS 12.0. Previously, we used an array of strings to define the types of files we wanted to open. In more modern times, we'er using an array of UTType definitions and these are defined in the UniformTypeIdentifiers import.

File Open

Opening A File

The AppKit way of prompting for a file is to create and display an NSOpenPanel. We check the response from the NSOpenPanel to see whether the user selected a file (or multiple files) or they cancelled. What we expect to get back, if the user selects one or more files, is the URL of the selected file(s).

Lets start by adding some code to the helper struct to create an NSOpenPanel:

```
struct FileHelpers {

    // MARK: - Private helpers

    private func createOpenPanel(ofType: [UTType],
                                 withTitle: String?,
                                 allowsMultiple: Bool = false) -> NSOpenPanel {

        let openPrompt = NSOpenPanel()

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

        openPrompt.allowsMultipleSelection = allowsMultiple
        openPrompt.canChooseDirectories = false
        openPrompt.canChooseFiles = true
        openPrompt.resolvesAliases = true
        openPrompt.allowedContentTypes = ofType

        return openPrompt
    }
}
```

The title of the window is optional, so we'll only set that if the caller passed us one. The main purpose of this function is to create the NSOpenPanel and set some default options specific to selecting a file. Nothing else.

Select a Single File

We now have a way to open the NSOpenPanel popup, so next we need a way to prompt the user. For that, we add a new function called *selectSingleInputFile*.

```
    public static func selectSingleInputFile(ofType fileTypes: [UTType], 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
    }
```

This helper function calls our function to create the NSOpenPanel, passing it a list of the types of files we want to be able to select.

The NSOpenPanel is displayed when we call its *runModal* function, which will return a response when the user either selects a file or presses cancel.

Assuming the user selects file, the *urls* property will be set with an array of URL's for the selected files. Since this function is supposed to return a single file, we return the first element of the list.

If the user selects Cancel, we return nil.

Selecting an Image File

We have enough to be able to select files. However, we want to centralise things a little; we don't really want to be coding UTType lists all over our code. So, lets add another helper function to out FileHelpers that specifically select image files:

```
    public static func selectSingleImageFile(withTitle windowTitle: String?) ->
URL? {
        let imageTypes: [UTType] = [UTType.image]

        return selectSingleInputFile(ofType: imageTypes, withTitle: windowTitle)
    }
```

This function allows our caller to select an image file. All the caller needs to do is pass us a title for the window and we'll figure out the file types for images. What we will return is the URL of the file, if one is selected, or nil of the user cancels.

Sample File Select Code

Earlier, we defined a menu item that we wanted to use to test our file code. So, lets update the File->Open menu item to call our new functions:

```
public struct FileCommands: Commands {

    public var body: some Commands {
```

```
        CommandGroup(before: CommandGroupPlacement.newItem) {
            Button("Open") {
                if let selectedFile = FileHelpers.selectSingleImageFile(withTitle:
"Select an image") {
                    print("You selected: \(selectedFile)")
                } else {
                    print("You cancelled the open request.")
                }
            }
            Divider()
        }
```

When the user selects File->Open from the menus, a file prompt will be displayed:

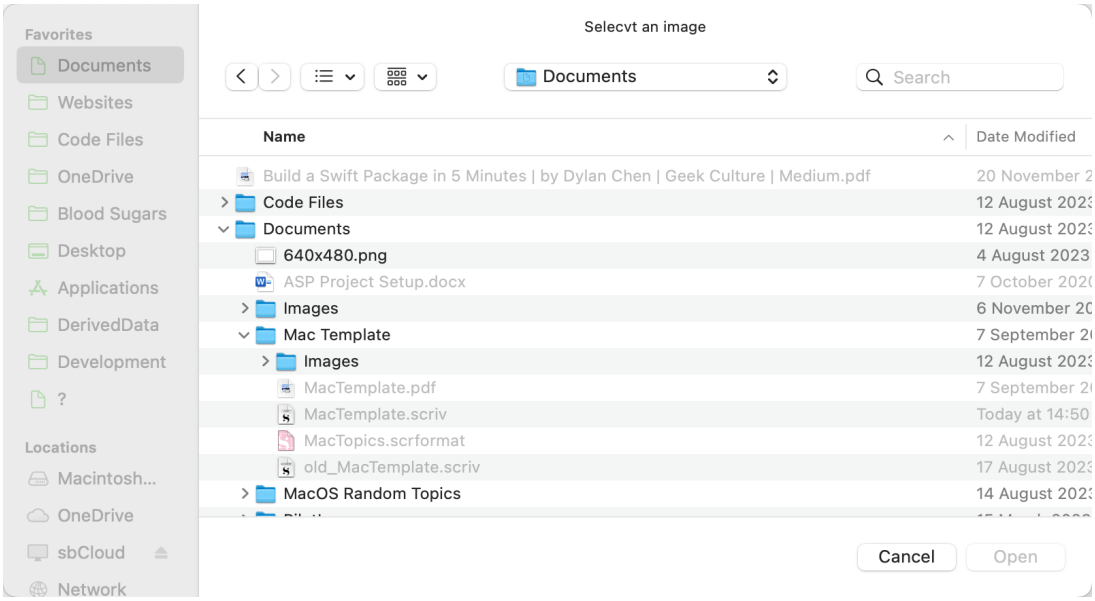


Figure 70: Select a File Prompt

The window title we passed through is displayed at the top of the popup and only image files are highlighted as being selectable.

When you select a file and click *open*, the URL of the file is output to the debug console:

```
Warning: Window SwiftUI.AppKitWindow 0x11d71c8a0 ordered front from a non-active
application and may order beneath the active application's windows.
You selected: file:///Users/stevenbarnett/Documents/Documents/640x480.png
CGSWindowShmemCreateWithPort failed on port 0
```

Figure 71: The resulting file selection

Custom File Types

Life becomes a little more complicated when we want to define a non-standard file type. It's complicated, so we need to step through it slowly!

Our File Extension

For the purposes of the template, we are going to add a file type for files with the extension "xyz". We're then going to extend our file selection code to request files with an extension of "xyz".

Our starting point if the project target and the *info.plist* to which we need to adda couple of definitions. I would not recommend editing the info.plist directly; there is a perfectly good and helpful designer built into Xcode that we're going to use. So, select the project file, select the target and click on the Info tab.

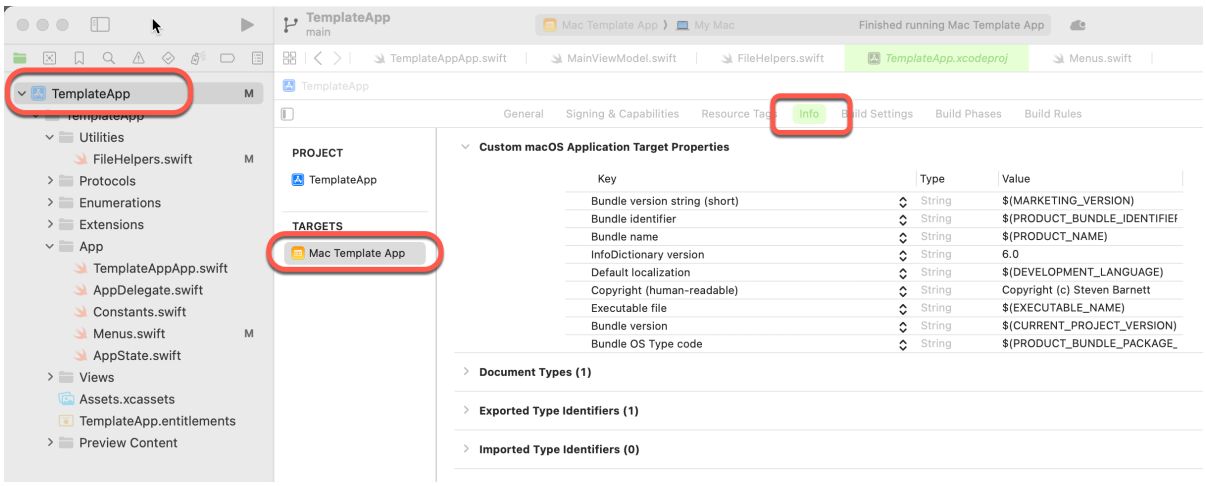


Figure 72: Where to define file types

In the list of sections, there is a section for *Document Types*. Expand this and click the + to add a new document type:

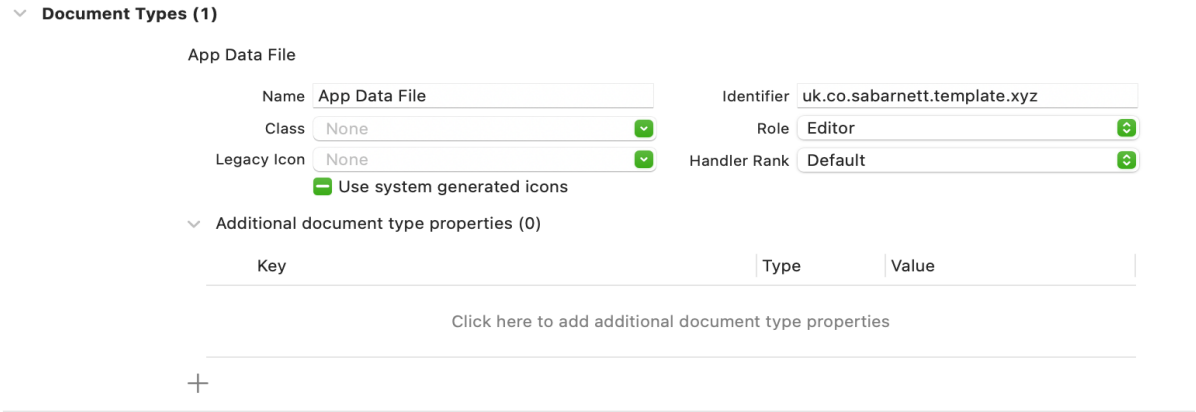


Figure 73: New Document Type

You need to provide

- The descriptive name of the data file.
- An identifier (this is important) which takes a similar form as your bundle name.

This gets you half way there. Next, expand the Exported Type Identifiers and create a new exported type:

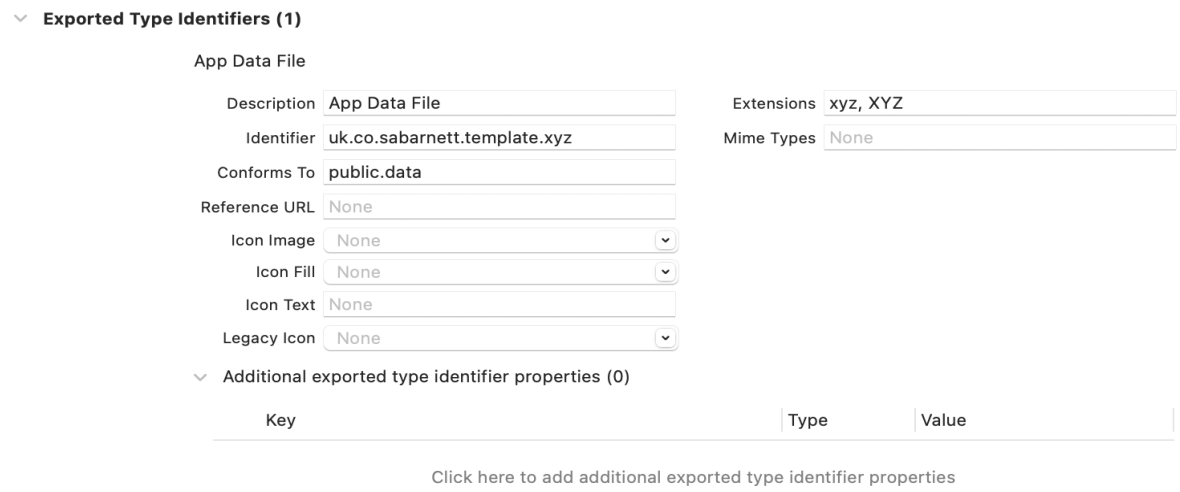


Figure 74: Exported Document Type

You need to provide:

- The type description, which you should generally make the same as the document type descriptive name.
- The identifier, which must be the same as the document type identifier.
- Conforms To should be set to public.data.
- Define the Extensions that you want to support for your document type. In this case, new are using “xyz” in both lower and upper case.

We’re almost ready to go!

Extending UTType

To use our type in a file selection window, we will need to be able to tell the window what UTType to open. The simplest way to achieve this is to extend UTType to include a definition for our new type.

At the top of the FileHelpers file, add this code:

```
import Foundation
import AppKit
import UniformTypeIdentifiers

extension UTType {
    static var dataFileType: UTType {
        UTType("uk.co.sabarnett.template.xyz")!
    }
}

struct FileHelpers { …
```

The name of the variable we define is the name that we are going to refer to in our code for this document type. The name that we use to initialise the UTType is the identifier we defined earlier.

Select Files of Type

Now we have defined our app's file type and have extended UTType to define the name of the identifier, we can add a function to the *FileHelpers* that we can call to select a file of this type:

```
struct FileHelpers {

    public static func selectSingleDataFile(withTitle windowTitle: String?) -> URL?
{
        return selectSingleInputFile(ofType: [.dataFileType], withTitle:
windowTitle)
    }
```

If we modify our File->Open to call this new function, we will get a prompt to open files with an extension of ".xyz":

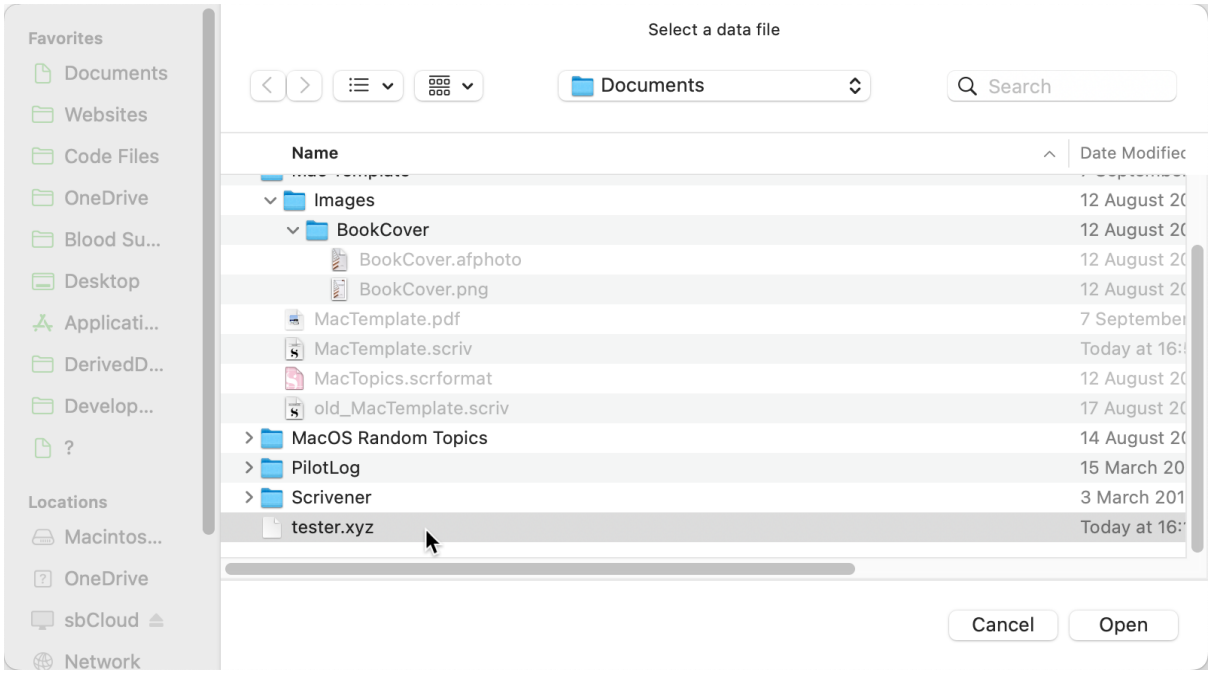


Figure 75: Select Files with our Document Type

Cleanup

Like most things we have done so far, this is complete but needs some minor cleanup. We need two things here;

- Move the extension to the extensions folder.
- Document the settings changes as we are very unlikely to use .xyz as a file extension in any real application.

First job is to create a file called *DTType+Extension* in the *Extensions* folder and move the extension code in to it:

```
import Foundation
import UniformTypeIdentifiers

extension UTType {
    static var dataFileType: UTType {
        UTType("uk.co.sabarnett.template.xyz")!
```

```
    }
}
```

The second part is probably best handled by adding a comment to this code:

```
/*
 Handles adding a custom file type to our app. To properly
 coordinate this change, you need to review the following items:

 Project Settings -> Target -> info tab.

 Review the Document Types for the App Data File.
 Review the Exported Type Identifiers.

 Ensure that you change the Identifier and Extensions to match
 your needs.

 Then, update the extension below for the updated identifier.

 */
extension UTType {
    static var dataFileType: UTType {
        UTType("uk.co.sabarnett.template.xyz")!
    }
}
```

Given that the code is fully written, these notes should be sufficient to remind us of what we need to change.

File Save-As

File Save-As

As with selecting a file, selecting a file to save to requires dropping to AppKit and using the NSSavePanel.

So, our starting point is to add a function to the FileHelpers struct to create the NSSavePanel:

```
    // MARK: - Private helpers

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

        let openPrompt = NSSavePanel()

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

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

        return openPrompt
    }
```

As with the open prompt, we create an NSSavePanel and set some basic options before returning the NSSavePanel to the caller.

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 different 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.

Select A File To Save To

We can now display the save panel, so we need to provide a function to call that will display it.

```
    // MARK: - Save Functions

    public static func selectSaveFile(ofType fileTypes: [UTType], 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
    }
```

This provides us with a simple wrapper around our save panel code that pops up the prompt. As with the file selection code, the save window appears when the *runModal* function is called. If a file is selected, we will get back the URL of the file and if the window is cancelled, we get back nil.

Commenting it all up

While we can call this code directly, we have the same issue as we had with the file selection prompt; we don't want to scatter generic calls all over the code. The solution is also the same; create a type specific function in the helpers struct. To save an image file, we would have:

```
    public static func selectImageFileToSave(withTitle windowTitle: String?) ->
URL? {
        let imageTypes: [UTType] = [.image]
        return selectSaveFile(ofType: imageTypes, withTitle: windowTitle)
    }
```

We can then add this to our save-as menu code:

But it doesn't work

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

> **CLIENT ERROR: remote view delegate NSSavePanel lacks method which can react to the details of Error Domain=com.apple.ViewBridge Code=14**

UserInfo={com.apple.ViewBridge.error.hint=<private>,

com.apple.ViewBridge.error.description=NSViewBridgeErrorServiceBootstrap}

Unable to display save panel: your app has the User Selected File Read

entitlement but it needs User Selected File Read/Write to display save panels.

Please ensure that your app's target capabilities include the proper entitlements.

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...

Sandboxing

You have to turn to sandboxing for the cause. In it's 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. Luckily, the error messages have improved over the years and Apple provide you the clues you need to fix it.

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

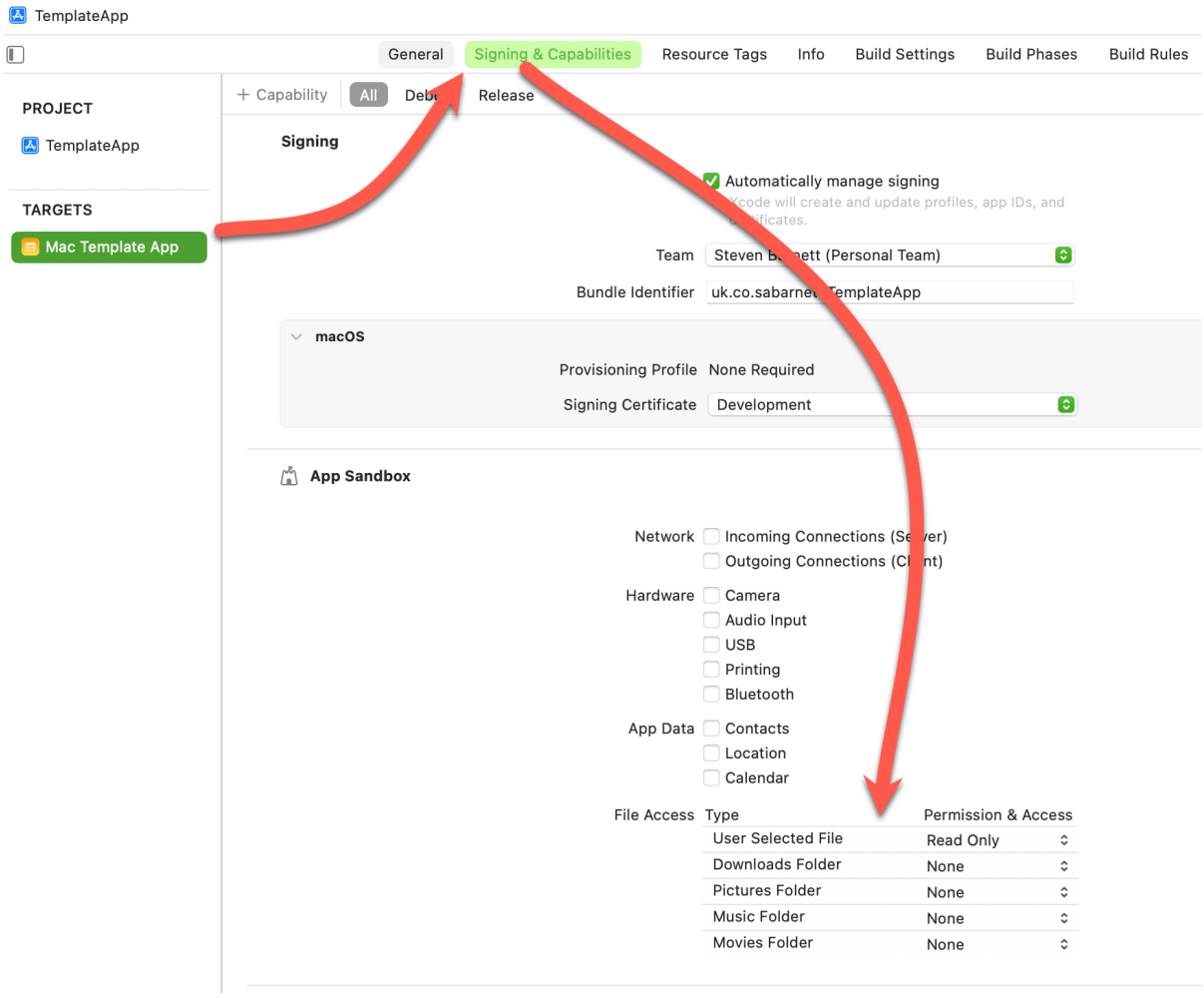


Figure 76: Sandboxing Capabilities

If you set the *User Selected File* option to *Read/Write*, and try again, you will see the save prompt:

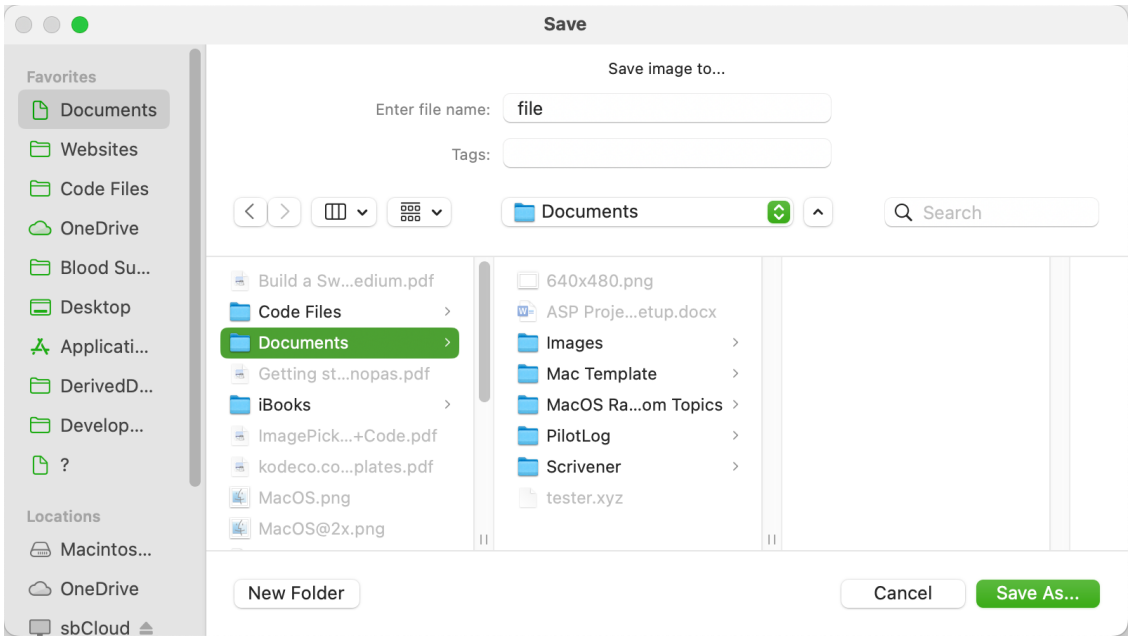


Figure 77: Save-as Popup Window

In a production application, you probably want a better default file name (perhaps date/time based).

When you get the URL back, it's worth noting that it will not have a file extension unless the user types one:

Save the file to: file:///Users/stevenbarnett/Documents/Documents/fileToSave

You will need to check for the extension and add it if necessary.

Selecting Folders

Selecting Folders

The final functionality we want to create is the ability to select a folder. You might want to prompt for a folder when generating output or when inputting multiple files. So, having the ability to select just a folder is useful to us.

There is, sadly, no specific functionality to be able to select a folder. So, we have to re-use the file selection code and change a few of the options. We can do this by adding the following function to our FileHelpers:

```
    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 can test this code in our menus

```
        CommandGroup(after: CommandGroupPlacement.newItem) {
            Divider()
            Button("Select Folder") {
                if let selectedFolder = FileHelpers.selectFolder() {
                    print("Selected folder: \(selectedFolder)")
                } else {
                    print("You cancelled the folder selection prompt")
                }
            }
        }
```

When you run this you will see a folder prompt:

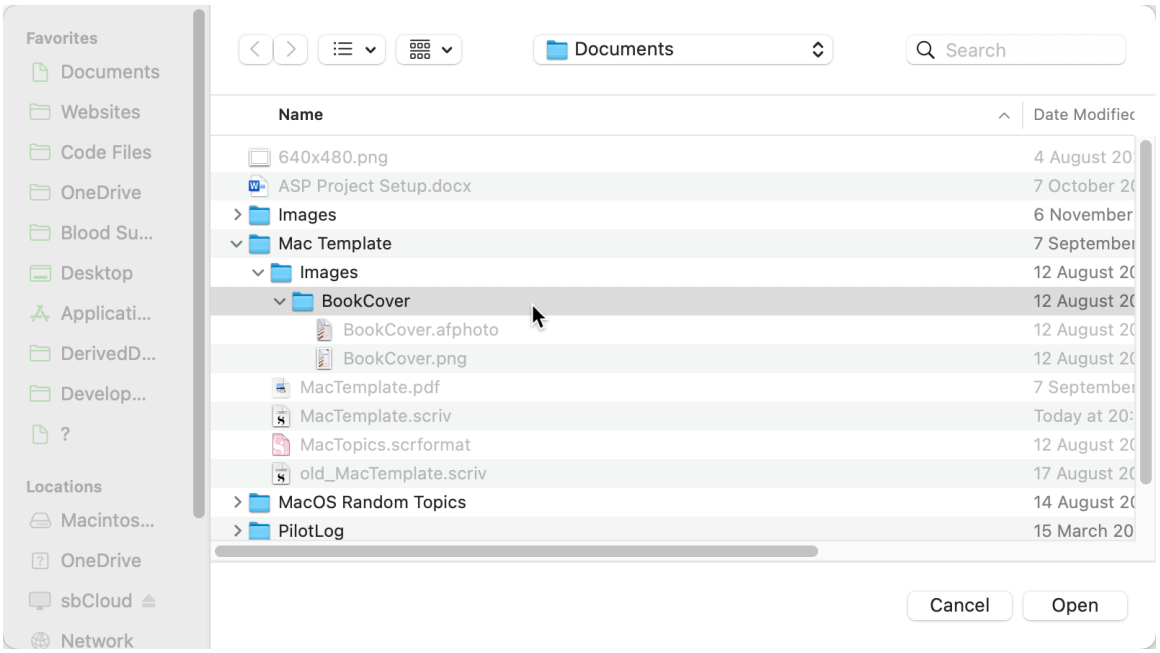


Figure 78: Folder Selection Prompt

When you select a folder and click *Open*, the URL of the selected folder is returned:

Selected folder: file:///Users/stevenbarnett/Documents/Documents/

Mac%20Template/Images/BookCover/

As you can see from the output, the URL is encoded to handle embedded spaces in the path. As with the other prompts, if you cancel, nil is returned.

Chapter Seven

Notifications

Introduction

Introduction

This section is entirely dependent on the application and its specific needs, so should be considered optional. I suspect most applications will never need to do internal communications between windows/view models. However, the mechanism can be useful in some obscure circumstances.

One contrived example I can come up with is a menu item to do a 'refresh all' that needs to inform every open window that it needs to refresh. You cannot use the view model connection we have in FocusedObject as that is a one to one connection. So we need a tool that can broadcast a message throughout the application. Alternatively, you might have a background task running that is monitoring a remote resource and you need to inform all open windows when the resource changes.

You can do this manually with, for example, the observer pattern or you can use the built in notifications mechanism. This section is about notifications.

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 it's notifications and does not care whether there is anyone listening or not. It sends it's 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

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 by creating a file called *Notifications* in the *Utilities* folder containing:

```
struct AppNotifications {
    static let RefreshAllNotification: String = “refreshAllNotification"
}
```

While there is only one notification in this struct, you are not 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 “refreshAllNotification”. In code it will be referred to as

```
AppNotifications.RefreshAllNotification
```

Creating Notifications

Creating a new notification is as simple as posting the notification to the NotificationCenter.

Create a *Notification* instance, specifying the name of the notification message and post it to the *NotificationCenter*. The notification is sent out to all that are listening for it. By way of example, lets expand our *Display* menu with a new button:

```
Divider()
Button("Refresh All") {
    let notificationName =
Notification.Name(AppNotifications.RefreshAllNotification)
    let notification = Notification(name: notificationName,
                                    object: nil,
                                    userInfo: nil)
    NotificationCenter.default.post(notification)
}
```

Listening For Notifications

Listening for Notifications

Before we get in to sending notifications, lets 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.

We're going to set-up our listener in the MainViewModel since that's where we want to handle refresh all notification messages for our open windows. Setting up the listener is very straight forward:

```
extension MainViewModel {

    func initialiseRefreshListener() {

        refreshObserver = NotificationCenter.default.addObserver(
            forName: Notification.Name(AppNotifications.RefreshAllNotification),
            object: nil,
            queue: nil,
            using: { (_) in
                 DispatchQueue.main.async {
                    self.refreshAll()
                 }
            })
    }

    func refreshAll() {
        print("Refresh all notification received")
    }
}
```

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 value from the addObserver call at the class level. We are going to need it later for clean-up purposes. It's defined as

```
class MainViewModel: ObservableObject {
```

```
    @Published var items: [String] = ["Mabel", "Morag", "Marcia"]
    @Published var selectedItem: String = ""

    private var refreshObserver: 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 *refreshAll* to handle the event and we wrap this is a DispatchQueue.main.async to ensure it gets run on the main UI thread..

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

Now, one last task that I mentioned above. We need to initialise our listener when the view model is created and we need to remove it when the view model is removed. We can add the listener in the class init().

```
    init() {
        initialiseRefreshListener()
    }
```

Run the code and open a couple of windows. Then select Display -> Refresh All. You should see one print message for every open window:

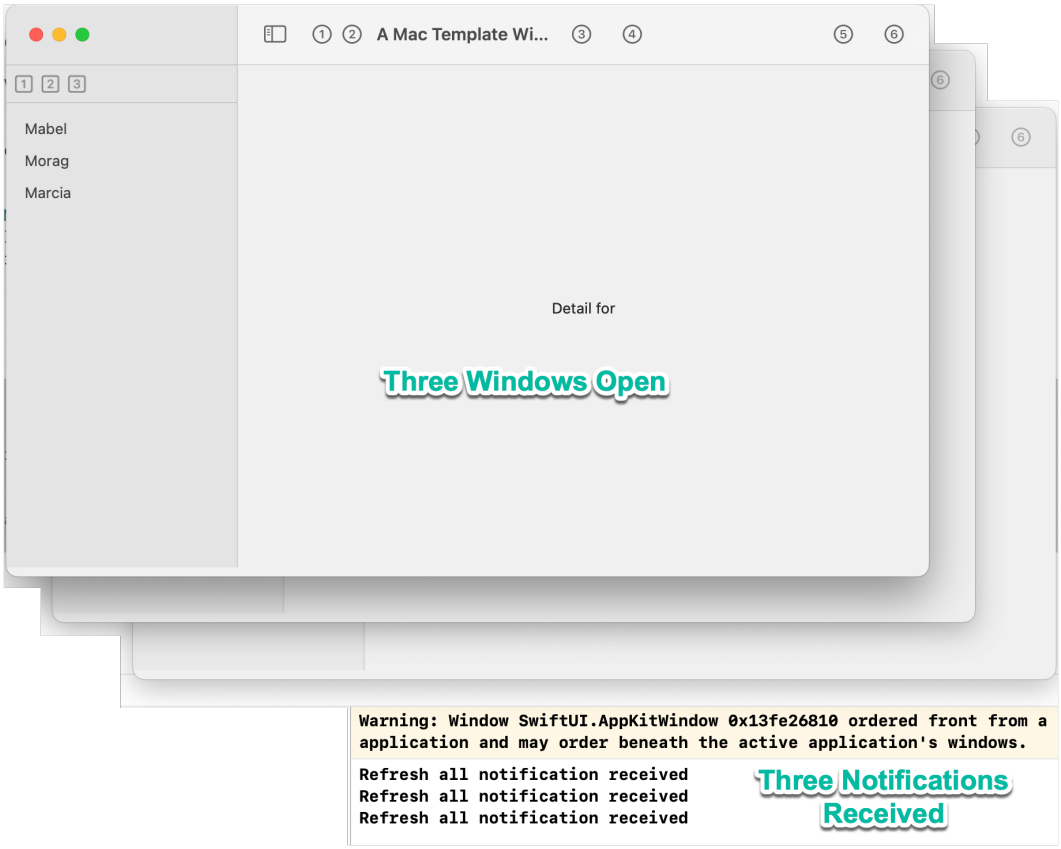


Figure 79: Notifications Received

Problem is, after we close the window, we continue to receive notifications!

Resetting Our Observer

If we leave our observer in place, the window isn't going to be closed properly. As long as the observer is active, the window and its view model will hang around in memory. In fact, the observer will continue to be triggered even after we think we have closed the window.

Removing the observer is a little more tricky. The class *deinit* will not be called while there are active notification observers, so the view and view model will not be cleaned app properly leading to a memory leak. In this case, we have a reference to the observer, so our deinit is not being called. We have to jump through some hoops to deal with this one. In the view model, we code a *reset()* function to deal with the removal of the listener:

```
    init() {
        initialiseRefreshListener()
    }

    func reset() {
        if let observerObject = refreshObserver {
            NotificationCenter.default.removeObserver(observerObject)
        }
    }
```

We check whether there is a listener before removing it. When the listener has been removed, the view model can be removed when the window closes. This leaves us with the issue of how to call the reset() function. We deal with that in the main view. We're going to need to get hold of an identifier for our window so we know when it gets closed. Each window has such a unique identifier in the form of the windowNumber. To get it, we first need to get hold of the widow hosting our view.

This is achieved using a helper view which I have called *HostingWindowFinder*. Create this file in View->Helpers and enter this content:

```
import SwiftUI

struct HostingWindowFinder: NSViewRepresentable {
    var callback: (NSWindow?) -> Void

    func makeNSView(context: Self.Context) -> NSView {
        let view = NSView()
        DispatchQueue.main.async { [weak view] in
            self.callback(view?.window)
        }
        return view
    }
    func updateNSView(_ nsView: NSView, context: Context) {}
}
```

The code works by creating a new NSView which exposes its window property. We pass this back to the SwiftUI view via a closure.

```
        HostingWindowFinder { window in
          if let window = window {
              windowNumber = window.windowNumber
          }
        }.frame(height: 0)
```

The windowNumber is saved at the view level because it won't change.

To know when our window is closed, we add an observer for the NSWindow.willCloseNotification message that gets sent every time a window is closed. The problem is, and the reason for needing the

window number, is that we will get a notification for every window that closes, not just our own. So, we need to receive the notification and check that it was for our window:

```
        .onReceive(NotificationCenter.default.publisher(for:
NSWindow.willCloseNotification)) { newValue in
            guard let win = newValue.object as? NSWindow,
                  win.windowNumber == windowNumber
            else { return }

            vm.reset()
        }

        HostingWindowFinder { window in
          if let window = window {
              windowNumber = window.windowNumber
          }
        }.frame(height: 0)
```

Assuming the notification is for our window, we call the view model *reset* function which resets the observer allowing the view and view model to be reinitialised correctly.

You can add and remove the listener at any time so, if it makes more sense, you can initialise or remove the listener whenever it is convenient. The only point to bear in mind is that you must remove the listener before disposing of the view model. Not doing so may introduce all sorts of subtle and difficult to track bugs into your code as the notification center may try to run code that is no longer there.

Passing Data

Passing Data In A Notification

In our refresh all notification, we are passing nil as the userInfo:

```
Button("Refresh All") {
    let notificationName =
Notification.Name(AppNotifications.RefreshAllNotification)
    let notification = Notification(name: notificationName,
                                    object: nil,
                                    userInfo: nil)
    NotificationCenter.default.post(notification)
}
```

For the majority of notifications, this is fine as we do not need to pass any additional data to the receiver. However, there may be a need to pass some additional context to our observer, so we need a way to package that data up and pass it on.

For the purposes of illustration, let's assume we want to tell our observers three additional pieces of information; a sender id, a target id and a filter. It's a contrived example, go with me here. At the point where we send the notification, we build a dictionary of parameters to send:

```
Button("Refresh All") {
    let userData: [AnyHashable: Any] = [
        "sender": "Sender id",
        "target": "Target id",
        "filter": ""
    ]

    let notificationName =
Notification.Name(AppNotifications.RefreshAllNotification)
    let notification = Notification(name: notificationName,
                                    object: nil,
                                    userInfo: userData)
    NotificationCenter.default.post(notification)
}
```

And that's all we need to do to send additional information.

In the observer, we need to decode this data, if it was passed:

```
    func initialiseRefreshListener() {

        refreshObserver = NotificationCenter.default.addObserver(
            forName: Notification.Name(AppNotifications.RefreshAllNotification),
            object: nil,
            queue: nil,
            using: { (userData) in
```

```
                if let userInfo = userData.userInfo as NSDictionary? as! [String:
String]? {
                    print("Sender: \(userInfo["sender"]!)")
                    print("Target: \(userInfo["target"]!)")
                    print("Filter: \(userInfo["filter"]!)")
                }

                DispatchQueue.main.async {
                    self.refreshAll()
                }
            })
    }
```

The closure is passed the userInfo which may be nil, so we first unwrap it. Once unwrapped we will have a dictionary from which we can extract the values we were passed.

```
Sender: Sender id
Target: Target id
Filter:
Refresh all notification received
```

Figure 80: Notification Data Received