

Contents

[Xamarin.Mac](#)

[Get Started](#)

[Installation](#)

[Hello, Mac](#)

[Related Documentation](#)

[Application Fundamentals](#)

[Accessibility](#)

[Common Patterns and Idioms](#)

[Console apps](#)

[macOS APIs](#)

[.xib Files](#)

[.storyboard/.xib-less User Interface Design](#)

[Images](#)

[Data Binding and Key-Value Coding](#)

[Databases](#)

[Copy and Paste](#)

[Sandboxing a Xamarin.Mac App](#)

[Playing Sound with AVAudioPlayer](#)

[User Interface](#)

[Windows](#)

[Dialogs](#)

[Alerts](#)

[Menus](#)

[Standard Controls](#)

[Toolbars](#)

[Table Views](#)

[Outline Views](#)

[Source Lists](#)

[Collection Views](#)

[Creating Custom Controls](#)

[Platform Features](#)

[Introduction to macOS Mojave](#)

[Getting Started](#)

[Introduction to macOS High Sierra](#)

[Introduction to macOS Sierra](#)

[Building Modern macOS Apps](#)

[Additional macOS Sierra Framework Changes](#)

[Troubleshooting](#)

[Binding Mac libraries](#)

[Introduction to OpenTK](#)

[Introduction to Storyboards](#)

[Storyboards Quick Start](#)

[Working with Storyboards](#)

[Xamarin.Mac Extension Support](#)

[Target Framework](#)

[Deployment and Testing](#)

[Application Icon](#)

[Debugging a native crash](#)

[Linker](#)

[Performance](#)

[Publishing to the App Store](#)

[Certificates and Identifiers](#)

[Provisioning Profiles](#)

[Mac App Configuration](#)

[Sign with Developer ID](#)

[Bundle for Mac App Store](#)

[Upload to Mac App Store](#)

[Apple Platform](#)

[Apple Account Management](#)

[Unified API](#)

[Overview](#)

- [Update Existing Apps](#)
- [Update Existing iOS Apps](#)
- [Update Existing Mac Apps](#)
- [Update Existing Xamarin.Forms Apps](#)
- [Migrating a Binding to the Unified API](#)
- [Tips for Updating Code to the Unified API](#)
- [Binding Objective-C](#)
 - [Overview](#)
 - [Binding Objective-C Libraries](#)
 - [Binding Types Reference](#)
 - [Objective Sharpie](#)
 - [Getting Started](#)
 - [Tools & Commands](#)
 - [Features](#)
 - [ApiDefinitions & StructsAndEnums](#)
 - [Binding Native Frameworks](#)
 - [Verify Attributes](#)
 - [Examples](#)
 - [Xcode](#)
 - [CocoaPod](#)
 - [Advanced \(manual\)](#)
 - [Release History](#)
 - [Troubleshooting](#)
- [Native References](#)
- [Native Types](#)
 - [32 and 64 bit Platform Considerations](#)
 - [Updating Xamarin.Mac Unified Apps to 64-bit](#)
 - [Working with Native Types in Cross-Platform Apps](#)
 - [HttpClient and SSL/TLS Implementation Selector](#)
 - [Build Optimizations](#)
 - [Advanced Concepts and Internals](#)
 - [AOT](#)

[Available Assemblies](#)

[Mac Architecture](#)

[How Xamarin.Mac works](#)

[Frameworks](#)

[Registrar](#)

[Troubleshooting](#)

[Troubleshooting Tips](#)

[Error messages \(mmp\)](#)

[Xamarin.Mac on Q&A](#)

[Release Notes](#)

[Samples](#)

Getting Started with Xamarin.Mac

10/28/2019 • 2 minutes to read • [Edit Online](#)

Installation

This article covers installation of Xamarin.Mac, including the requirements, pre-requisite software, and activation.

Hello, Mac

This guide walks through creating a simple, first Xamarin.Mac app, and in the process introduces the development toolchain, including Visual Studio for Mac, Xcode and Interface Builder. It also introduces Outlets and Actions, which expose UI controls to code, and finally, it illustrates how to build, run and test a Xamarin.Mac application.

Related Documentation

Links to additional useful documentation.

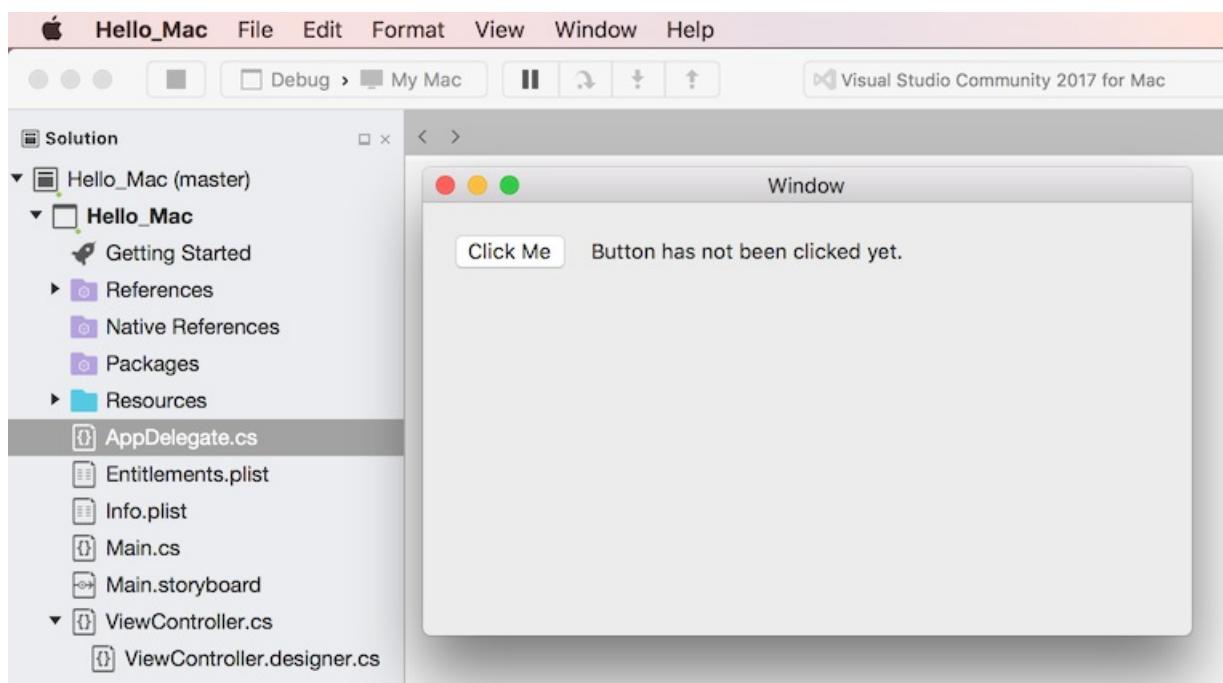
Hello, Mac – Walkthrough

11/2/2020 • 24 minutes to read • [Edit Online](#)

Xamarin.Mac allows for the development of fully native Mac apps in C# and .NET using the same macOS APIs that are used when developing in *Objective-C* or *Swift*. Because Xamarin.Mac integrates directly with Xcode, the developer can use Xcode's *Interface Builder* to create an app's user interfaces (or optionally create them directly in C# code).

Additionally, since Xamarin.Mac applications are written in C# and .NET, code can be shared with Xamarin.iOS and Xamarin.Android mobile apps; all while delivering a native experience on each platform.

This article will introduce the key concepts needed to create a Mac app using Xamarin.Mac, Visual Studio for Mac and Xcode's Interface Builder by walking through the process of building a simple **Hello, Mac** app that counts the number of times a button has been clicked:



The following concepts will be covered:

- **Visual Studio for Mac** – Introduction to the Visual Studio for Mac and how to create Xamarin.Mac applications with it.
- **Anatomy of a Xamarin.Mac Application** – What a Xamarin.Mac application consists of.
- **Xcode's Interface Builder** – How to use Xcode's Interface Builder to define an app's user interface.
- **Outlets and Actions** – How to use Outlets and Actions to wire up controls in the user interface.
- **Deployment/Testing** – How to run and test a Xamarin.Mac app.

Requirements

Xamarin.Mac application development requires:

- A Mac computer running macOS High Sierra (10.13) or higher.
- [Xcode 10 or higher](#).
- The latest version of [Xamarin.Mac and Visual Studio for Mac](#).

To run an application built with Xamarin.Mac, you will need:

- A Mac computer running macOS 10.7 or greater.

WARNING

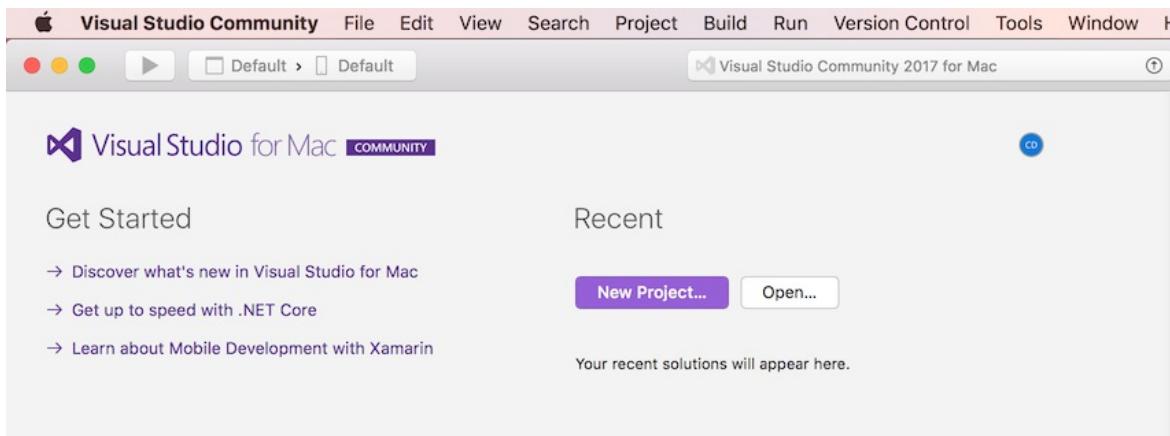
The upcoming Xamarin.Mac 4.8 release will only support macOS 10.9 or higher. Previous versions of Xamarin.Mac supported macOS 10.7 or higher, but these older macOS versions lack sufficient TLS infrastructure to support TLS 1.2. To target macOS 10.7 or macOS 10.8, use Xamarin.Mac 4.6 or earlier.

Starting a new Xamarin.Mac App in Visual Studio for Mac

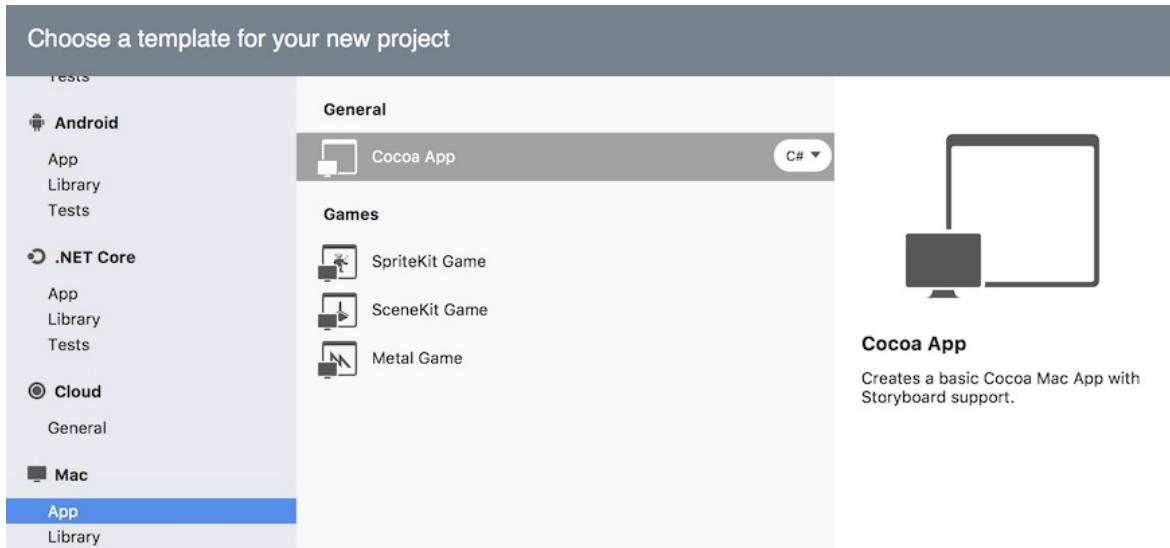
As stated above, this guide will walk through the steps to create a Mac app called `Hello_Mac` that adds a single button and label to the main window. When the button is clicked, the label will display the number of times it has been clicked.

To get started, do the following steps:

1. Start Visual Studio for Mac:



2. Click on the **New Project...** button to open the **New Project** dialog box, then select **Mac > App > Cocoa App** and click the **Next** button:



3. Enter `Hello_Mac` for the **App Name**, and keep everything else as default. Click **Next**:

Configure your Mac app

App Name: Hello_Mac

Organization Identifier: com.companyname ?

Bundle Identifier: com.companyname.Hello-Mac

4. Confirm the location of the new project on your computer:

Configure your new Cocoa App

Project Name: Hello_Mac

Solution Name: Hello_Mac

Location: /Users/me/Projects Browse...

Create a project directory within the solution directory.

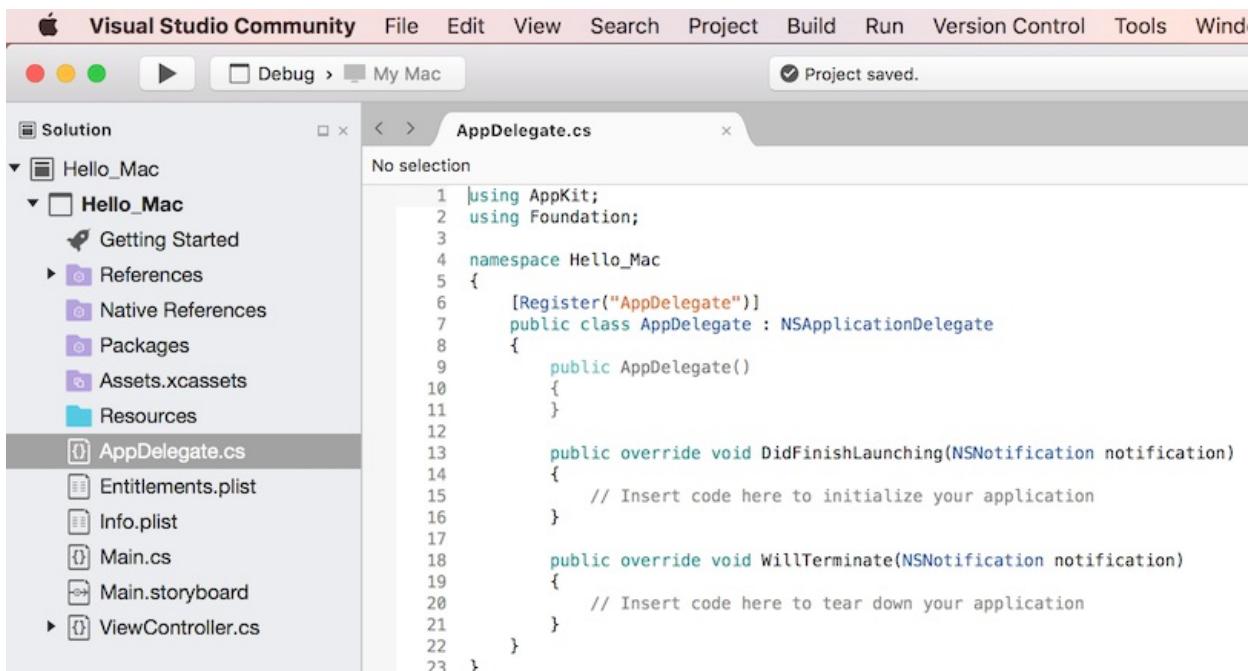
Version Control: Use git for version control.
 Create a .gitignore file to ignore inessential files.

PREVIEW

- /Users/me/Projects
- Hello_Mac
- Hello_Mac.sln
- Hello_Mac
- Hello_Mac.csproj

5. Click the **Create** button.

Visual Studio for Mac will create the new Xamarin.Mac app and display the default files that get added to the app's solution:



Visual Studio for Mac uses the same **Solution** and **Project** structure as Visual Studio 2019. A solution is a container that can hold one or more projects; projects can include applications, supporting libraries, test applications, etc. The **File > New Project** template creates a solution and an application project automatically.

Anatomy of a Xamarin.Mac Application

Xamarin.Mac application programming is very similar to working with Xamarin.iOS. iOS uses the CocoaTouch framework, which is a slimmed-down version of Cocoa, used by Mac.

Take a look at the files in the project:

- **Main.cs** contains the main entry point of the app. When the app is launched, the `Main` class contains the very first method that is run.
- **AppDelegate.cs** contains the `AppDelegate` class that is responsible for listening to events from the operating system.
- **Info.plist** contains app properties such as the application name, icons, etc.
- **Entitlements.plist** contains the entitlements for the app and allows access to things such as Sandboxing and iCloud support.
- **Main.storyboard** defines the user interface (Windows and Menus) for an app and lays out the interconnections between Windows via Segues. Storyboards are XML files that contain the definition of views (user interface elements). This file can be created and maintained by Interface Builder inside of Xcode.
- **ViewController.cs** is the controller for the main window. Controllers will be covered in detail in another article, but for now, a controller can be thought of the main engine of any particular view.
- **ViewController.designer.cs** contains plumbing code that helps integrate with the main screen's user interface.

The following sections, will take a quick look through some of these files. Later, they will be explored in more detail, but it's a good idea to understand their basics now.

Main.cs

The **Main.cs** file is very simple. It contains a static `Main` method which creates a new Xamarin.Mac app instance and passes the name of the class that will handle OS events, which in this case is the `AppDelegate` class:

```
using System;
using System.Drawing;
using Foundation;
using AppKit;
using ObjCRuntime;

namespace Hello_Mac
{
    class MainClass
    {
        static void Main (string[] args)
        {
            NSApplication.Init ();
            NSApplication.Main (args);
        }
    }
}
```

AppDelegate.cs

The `AppDelegate.cs` file contains an `AppDelegate` class, which is responsible for creating windows and listening to OS events:

```

using AppKit;
using Foundation;

namespace Hello_Mac
{
    [Register ("AppDelegate")]
    public class AppDelegate : NSApplicationDelegate
    {
        public AppDelegate ()
        {

        }

        public override void DidFinishLaunching (NSNotification notification)
        {
            // Insert code here to initialize your application
        }

        public override void WillTerminate (NSNotification notification)
        {
            // Insert code here to tear down your application
        }
    }
}

```

This code is probably unfamiliar unless the developer has built an iOS app before, but it's fairly simple.

The `DidFinishLaunching` method runs after the app has been instantiated, and it's responsible for actually creating the app's window and beginning the process of displaying the view in it.

The `WillTerminate` method will be called when the user or the system has instantiated a shutdown of the app. The developer should use this method to finalize the app before it quits (such as saving user preferences or window size and location).

ViewController.cs

Cocoa (and by derivation, CocoaTouch) uses what's known as the *Model View Controller* (MVC) pattern. The `ViewController` declaration represents the object that controls the actual app window. Generally, for every window created (and for many other things within windows), there is a controller, which is responsible for the window's lifecycle, such as showing it, adding new views (controls) to it, etc.

The `viewController` class is the main window's controller. The controller is responsible for the life cycle of the main window. This will be examined in detail later, for now take a quick look at it:

```

using System;

using AppKit;
using Foundation;

namespace Hello_Mac
{
    public partial class ViewController : NSViewController
    {
        public ViewController (IntPtr handle) : base (handle)
        {
        }

        public override void ViewDidLoad ()
        {
            base.ViewDidLoad ();

            // Do any additional setup after loading the view.
        }

        public override NSObject RepresentedObject {
            get {
                return base.RepresentedObject;
            }
            set {
                base.RepresentedObject = value;
                // Update the view, if already loaded.
            }
        }
    }
}

```

ViewController.Designer.cs

The designer file for the Main Window class is initially empty, but it will be automatically populated by Visual Studio for Mac as the user interface is created with Xcode Interface Builder:

```

// WARNING
//
// This file has been generated automatically by Visual Studio for Mac to store outlets and
// actions made in the UI designer. If it is removed, they will be lost.
// Manual changes to this file may not be handled correctly.
//
using Foundation;

namespace Hello_Mac
{
    [Register ("ViewController")]
    partial class ViewController
    {
        void ReleaseDesignerOutlets ()
        {
        }
    }
}

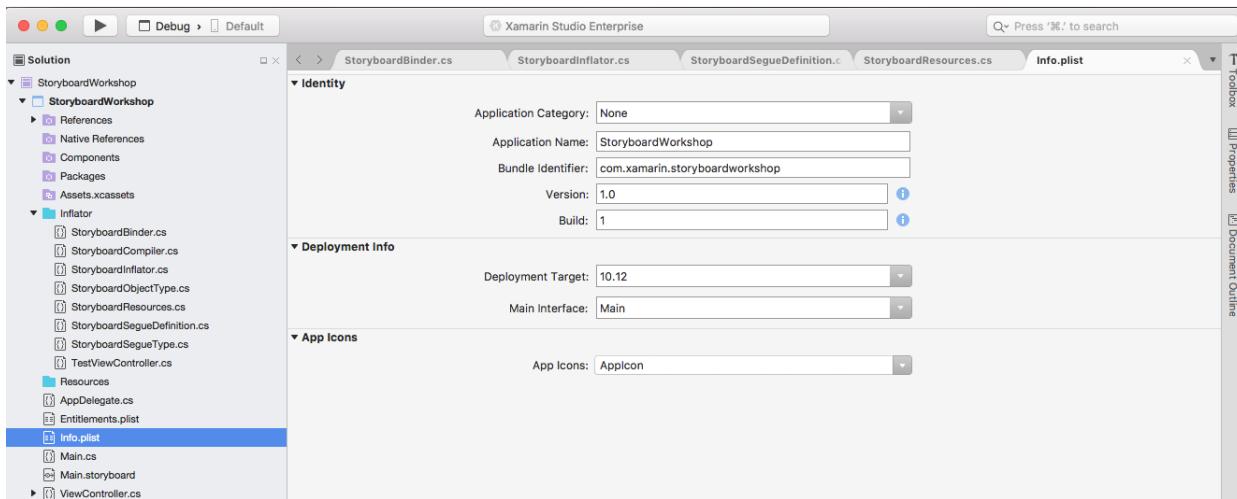
```

Designer files should not be edited directly, as they're automatically managed by Visual Studio for Mac to provide the plumbing code that allows access to controls that have been added to any window or view in the app.

With the Xamarin.Mac app project created and a basic understanding of its components, switch to Xcode to create the user interface using Interface Builder.

Info.plist

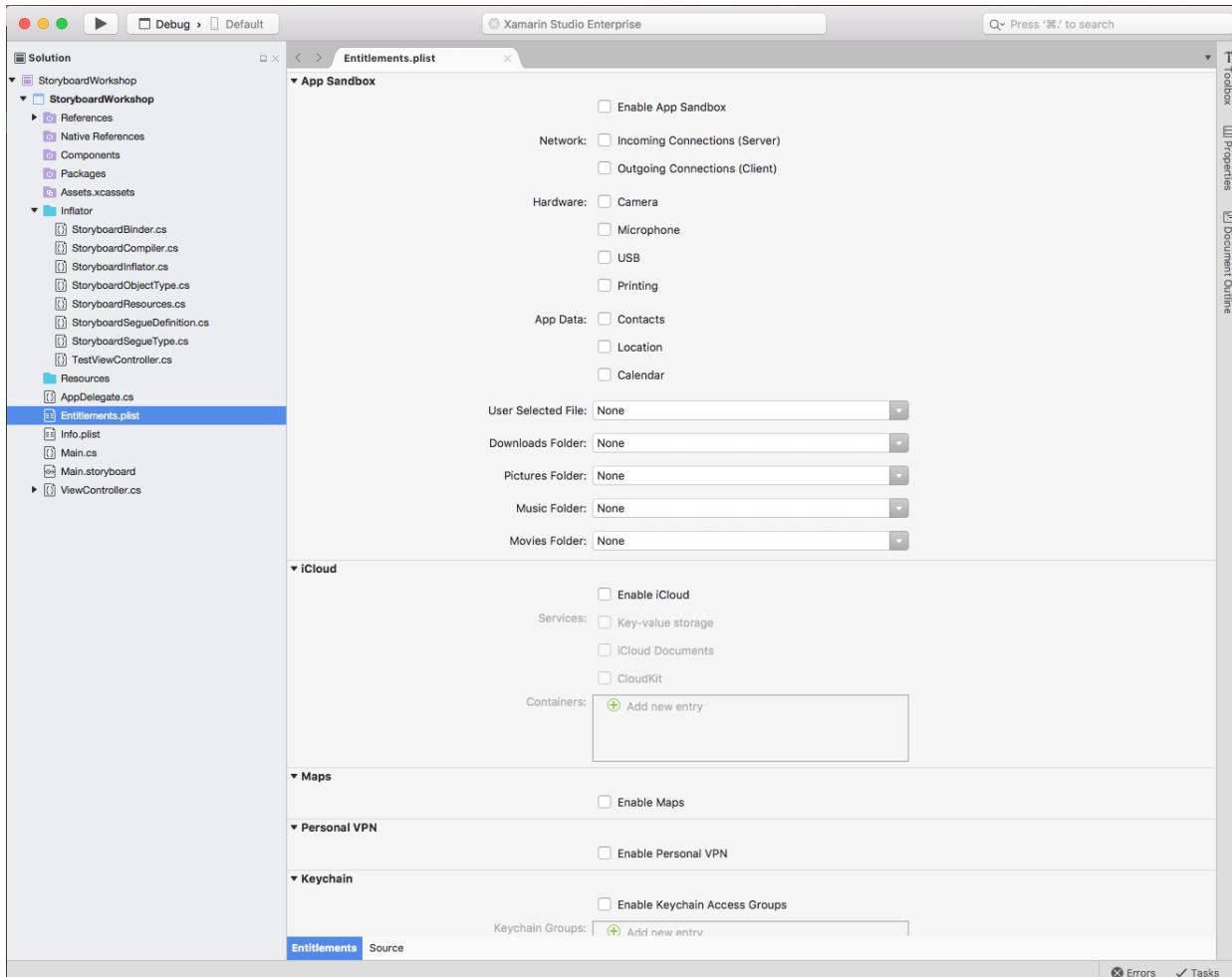
The `Info.plist` file contains information about the Xamarin.Mac app such as its **Name** and **Bundle Identifier**:



It also defines the *Storyboard* that will be used to display the user interface for the Xamarin.Mac app under the **Main Interface** dropdown. In example above, `Main` in the dropdown relates to the `Main.storyboard` in the project's source tree in the **Solution Explorer**. It also defines the app's icons by specifying the *Asset Catalog* that contains them (`AppIcon` in this case).

Entitlements.plist

The app's `Entitlements.plist` file controls entitlements that the Xamarin.Mac app has such as **Sandboxing** and **iCloud**:

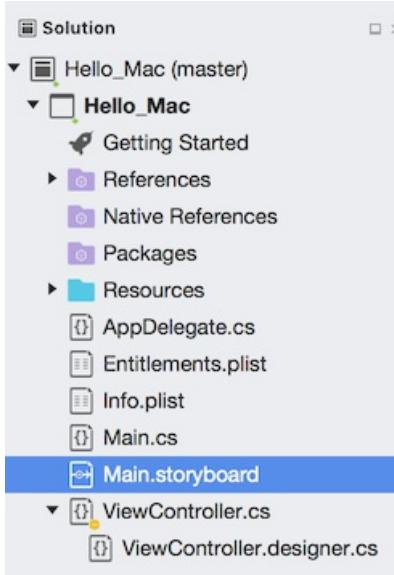


For the Hello World example, no entitlements will be required. The next section shows how to use Xcode's Interface Builder to edit the `Main.storyboard` file and define the Xamarin.Mac app's UI.

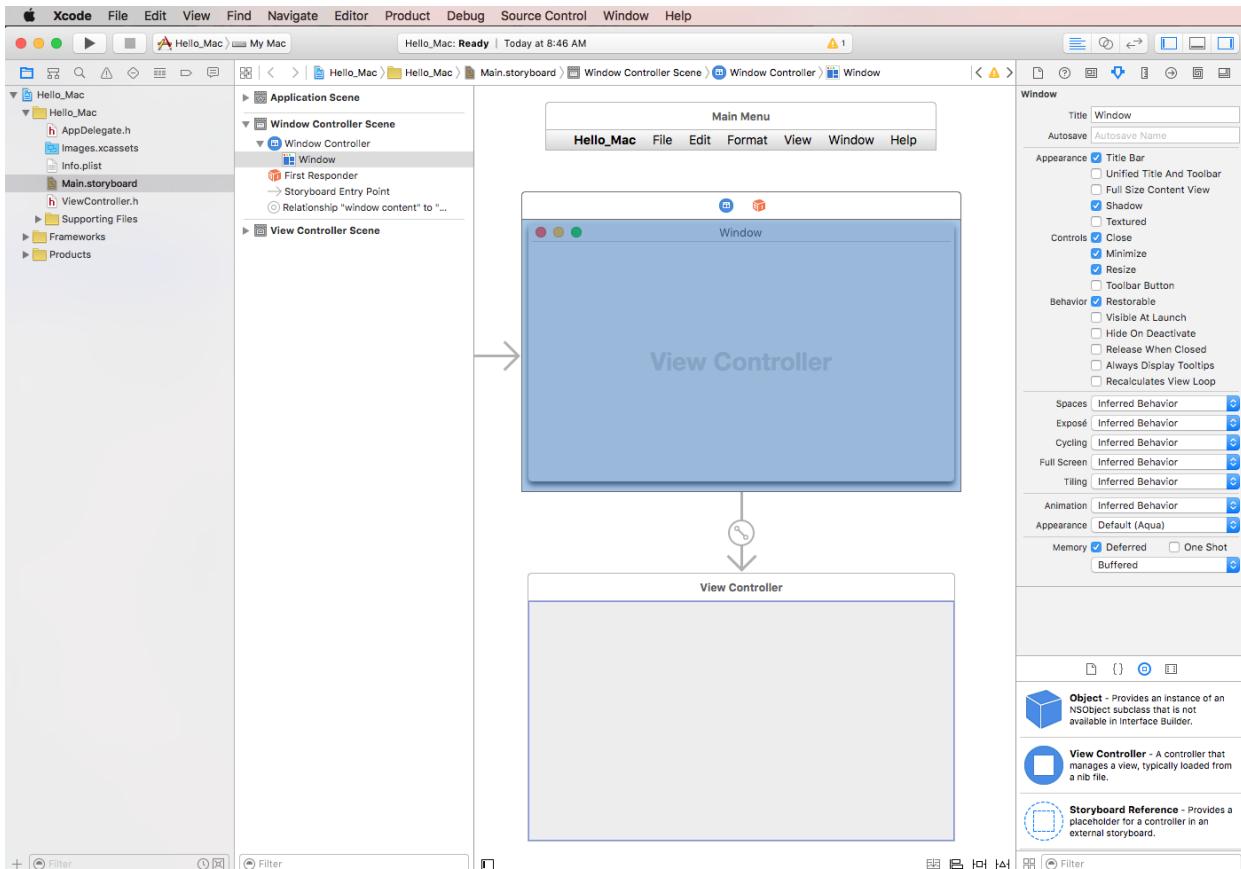
Introduction to Xcode and Interface Builder

As part of Xcode, Apple has created a tool called Interface Builder, which allows a developer to create a user interface visually in a designer. Xamarin.Mac integrates fluently with Interface Builder, allowing UI to be created with the same tools as Objective-C users.

To get started, double-click the `Main.storyboard` file in the **Solution Explorer** to open it for editing in Xcode and Interface Builder:



This should launch Xcode and look like this screenshot:



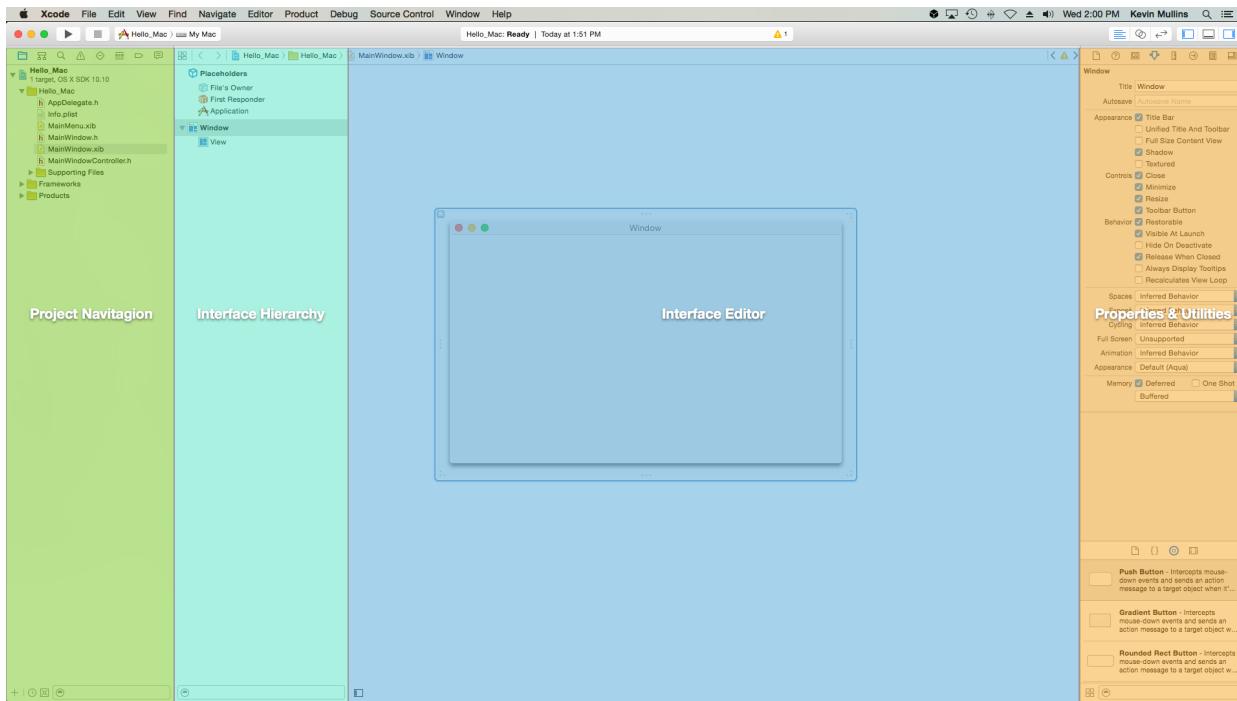
Before starting to design the interface, take a quick overview of Xcode to orient with the main features that will be used.

NOTE

The developer doesn't have to use Xcode and Interface Builder to create the user interface for a Xamarin.Mac app, the UI can be created directly from C# code but that is beyond the scope of this article. For the sake of simplicity, it will be using Interface Builder to create the user interface throughout the rest of this tutorial.

Components of Xcode

When opening a **.storyboard** file in Xcode from Visual Studio for Mac, it opens with a **Project Navigator** on the left, the **Interface Hierarchy** and **Interface Editor** in the middle, and a **Properties & Utilities** section on the right:



The following sections take a look at what each of these Xcode features do and how to use them to create the interface for a Xamarin.Mac app.

Project Navigation

When opening a **.storyboard** file for editing in Xcode, Visual Studio for Mac creates a **Xcode Project File** in the background to communicate changes between itself and Xcode. Later, when the developer switches back to Visual Studio for Mac from Xcode, any changes made to this project are synchronized with the Xamarin.Mac project by Visual Studio for Mac.

The **Project Navigation** section allows the developer to navigate between all of the files that make up this *shim* Xcode project. Typically, they will only be interested in the **.storyboard** files in this list such as `Main.storyboard`.

Interface Hierarchy

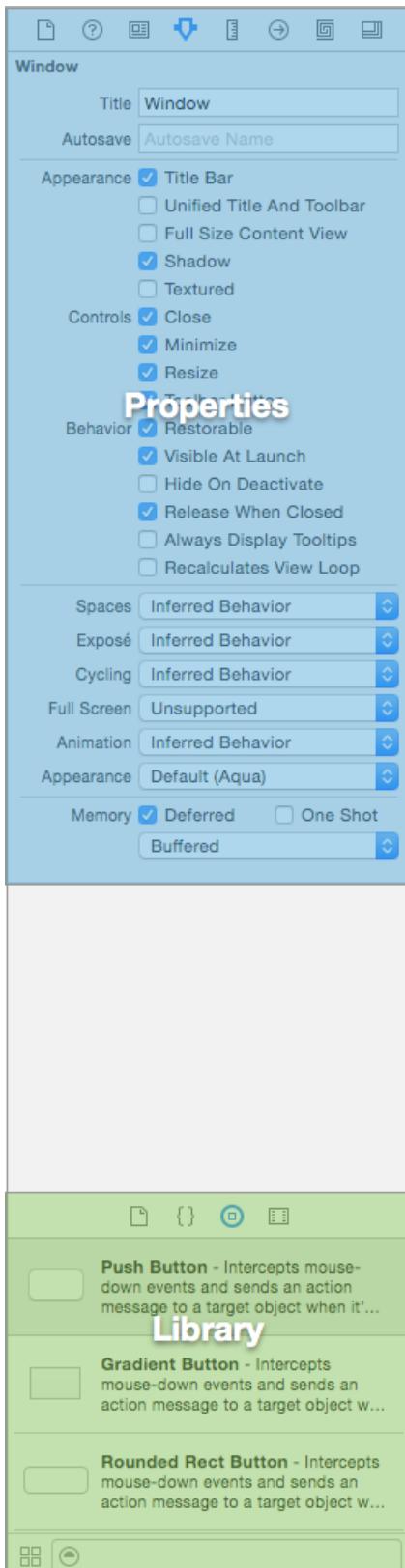
The **Interface Hierarchy** section allows the developer to easily access several key properties of the user interface such as its **Placeholders** and main **Window**. This section can be used to access the individual elements (views) that make up the user interface and to adjust the way they are nested by dragging them around within the hierarchy.

Interface Editor

The **Interface Editor** section provides the surface on which the user interface is graphically laid out. Drag elements from the **Library** section of the **Properties & Utilities** section to create the design. As user interface elements (views) are added to the design surface, they will be added to the **Interface Hierarchy** section in the order that they appear in the **Interface Editor**.

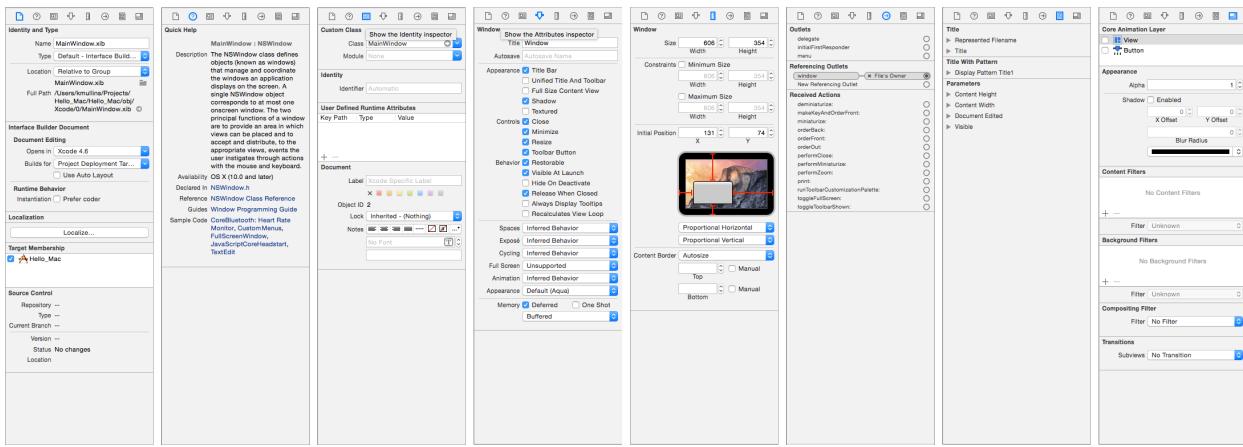
Properties & Utilities

The Properties & Utilities section is divided into two main sections, **Properties** (also called Inspectors) and the **Library**:



Initially this section is almost empty, however if the developer selects an element in the **Interface Editor** or **Interface Hierarchy**, the **Properties** section will be populated with information about the given element and properties that they can adjust.

Within the **Properties** section, there are eight different *Inspector Tabs*, as shown in the following illustration:

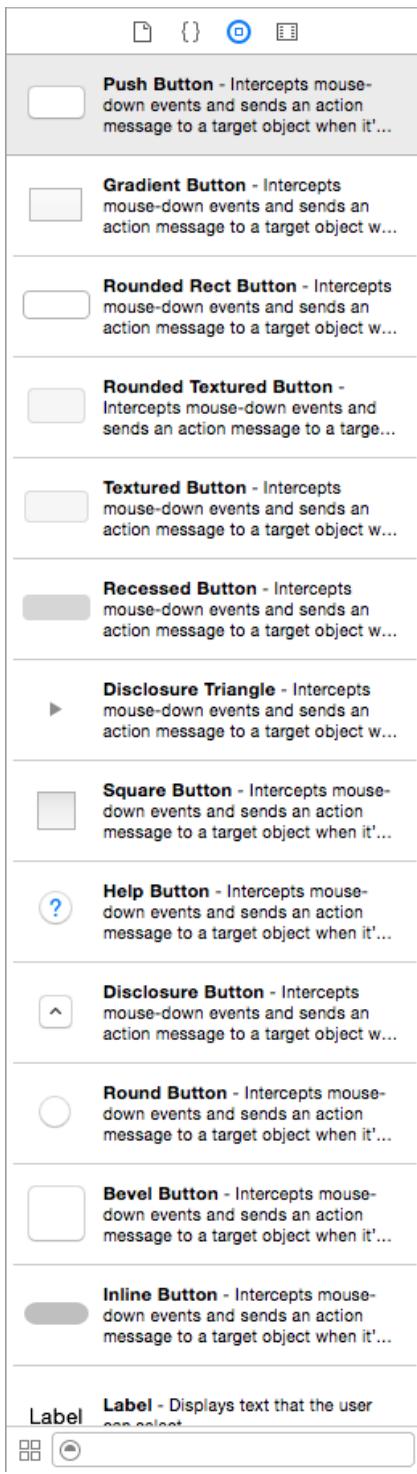


Properties & Utility Types

From left-to-right, these tabs are:

- **File Inspector** – The File Inspector shows file information, such as the file name and location of the Xib file that is being edited.
- **Quick Help** – The Quick Help tab provides contextual help based on what is selected in Xcode.
- **Identity Inspector** – The Identity Inspector provides information about the selected control/view.
- **Attributes Inspector** – The Attributes Inspector allows the developer to customize various attributes of the selected control/view.
- **Size Inspector** – The Size Inspector allows the developer to control the size and resizing behavior of the selected control/view.
- **Connections Inspector** – The Connections Inspector shows the **Outlet** and **Action** connections of the selected controls. Outlets and Actions will be discussed in detail below.
- **Bindings Inspector** – The Bindings Inspector allows the developer to configure controls so that their values are automatically bound to data models.
- **View Effects Inspector** – The View Effects Inspector allows the developer to specify effects on the controls, such as animations.

Use the **Library** section to find controls and objects to place into the designer to graphically build the user interface:

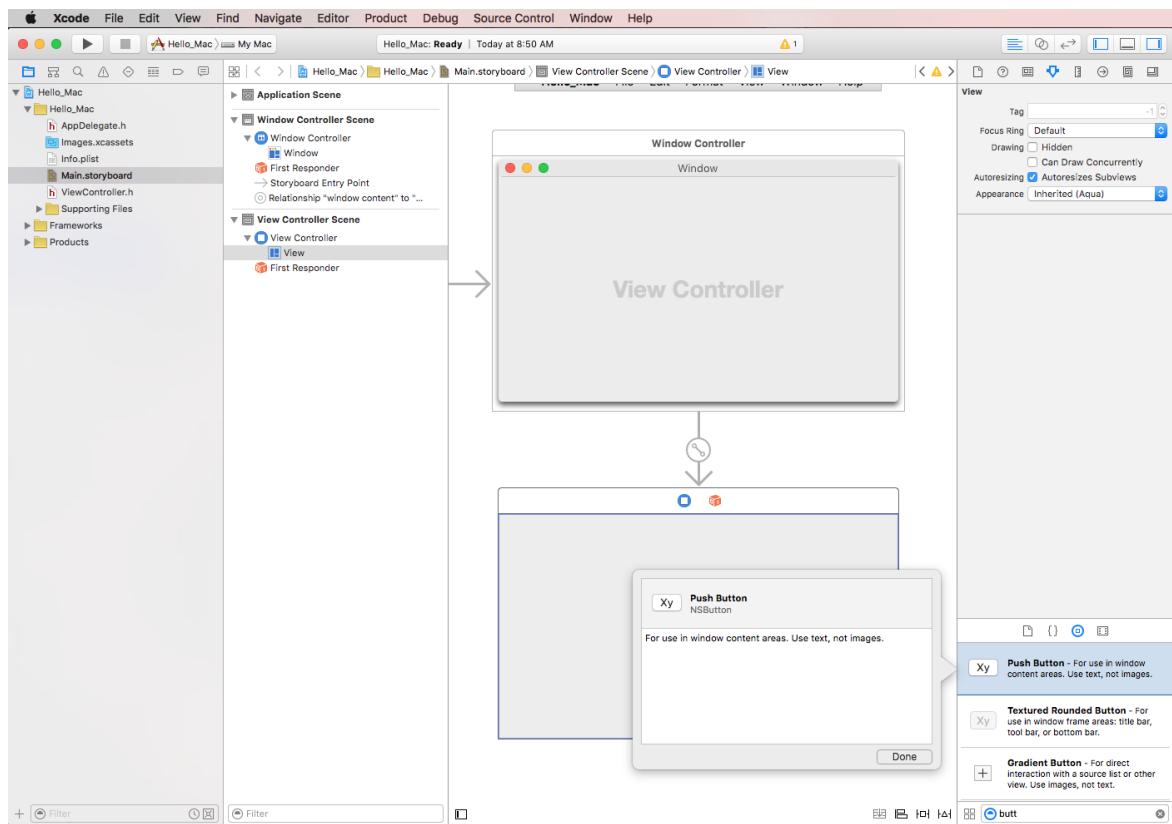


Creating the Interface

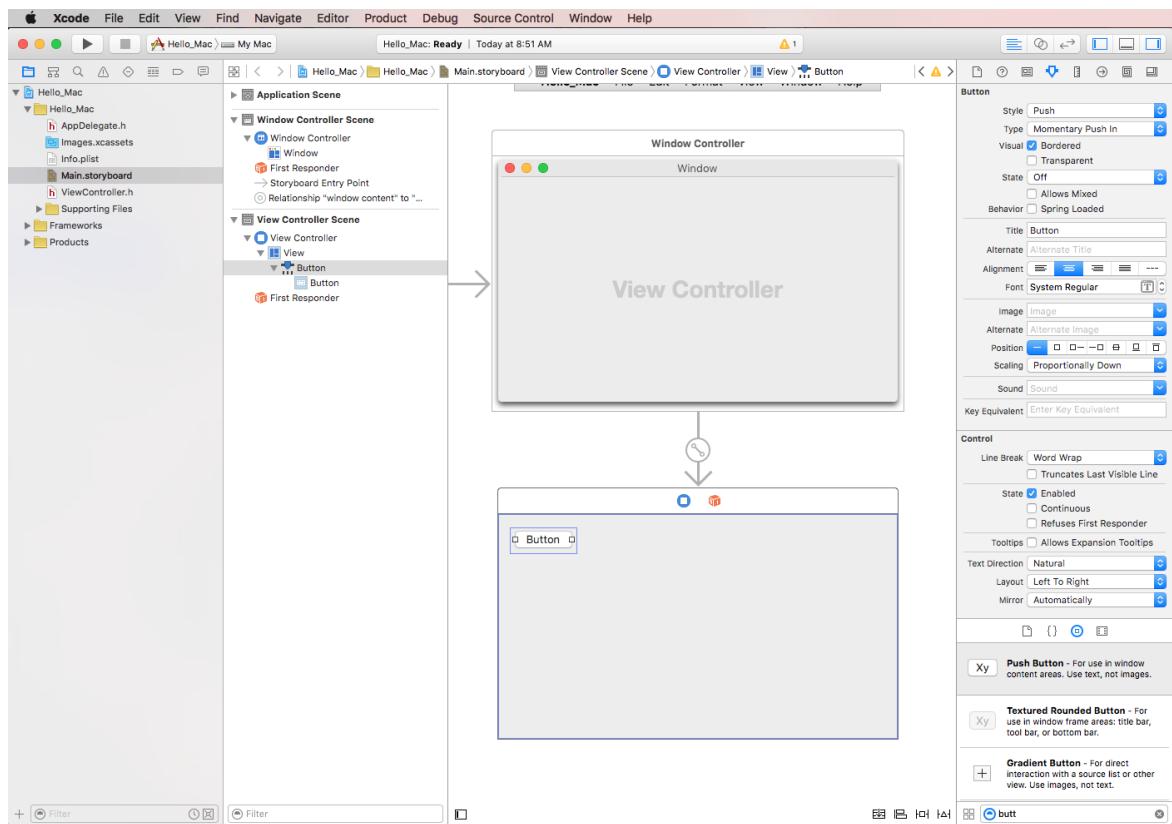
With the basics of the Xcode IDE and Interface Builder covered, the developer can create the user interface for the main view.

Follow these steps to use Interface Builder:

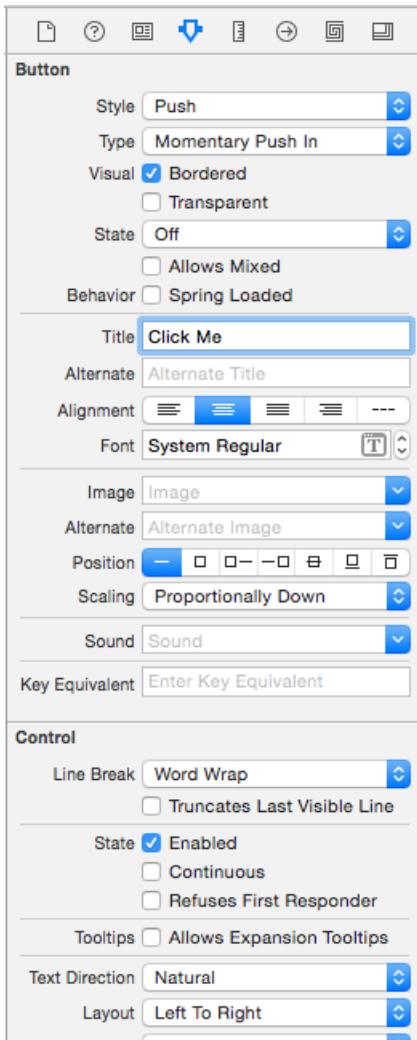
1. In Xcode, drag a **Push Button** from the Library Section:



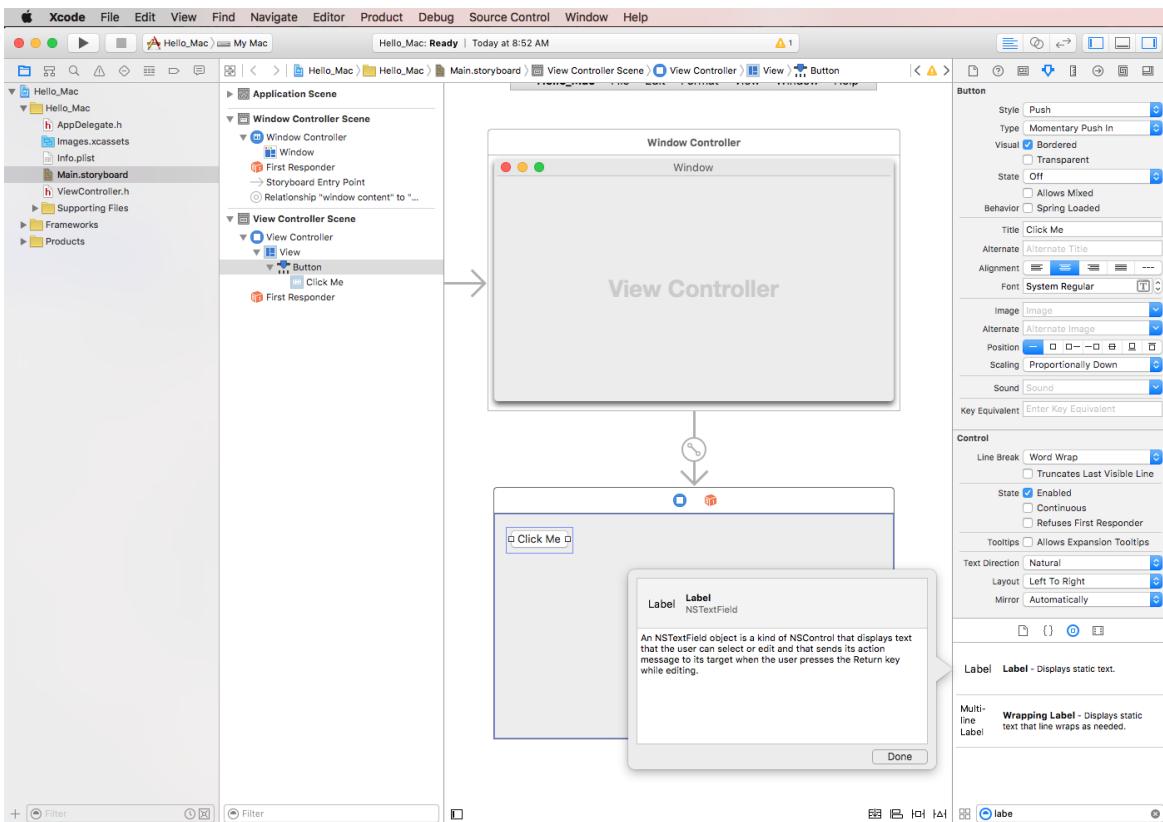
2. Drop the button onto the View (under the Window Controller) in the Interface Editor:



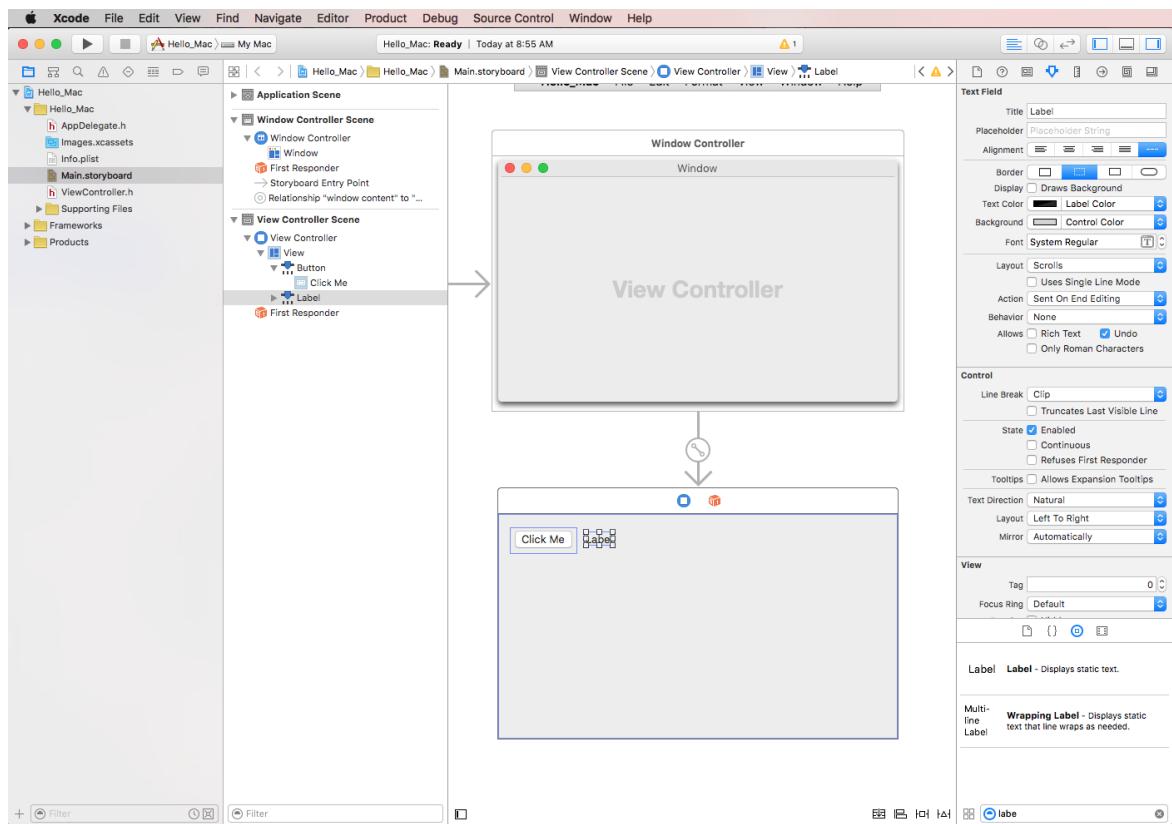
3. Click on the Title property in the Attribute Inspector and change the button's title to Click Me:



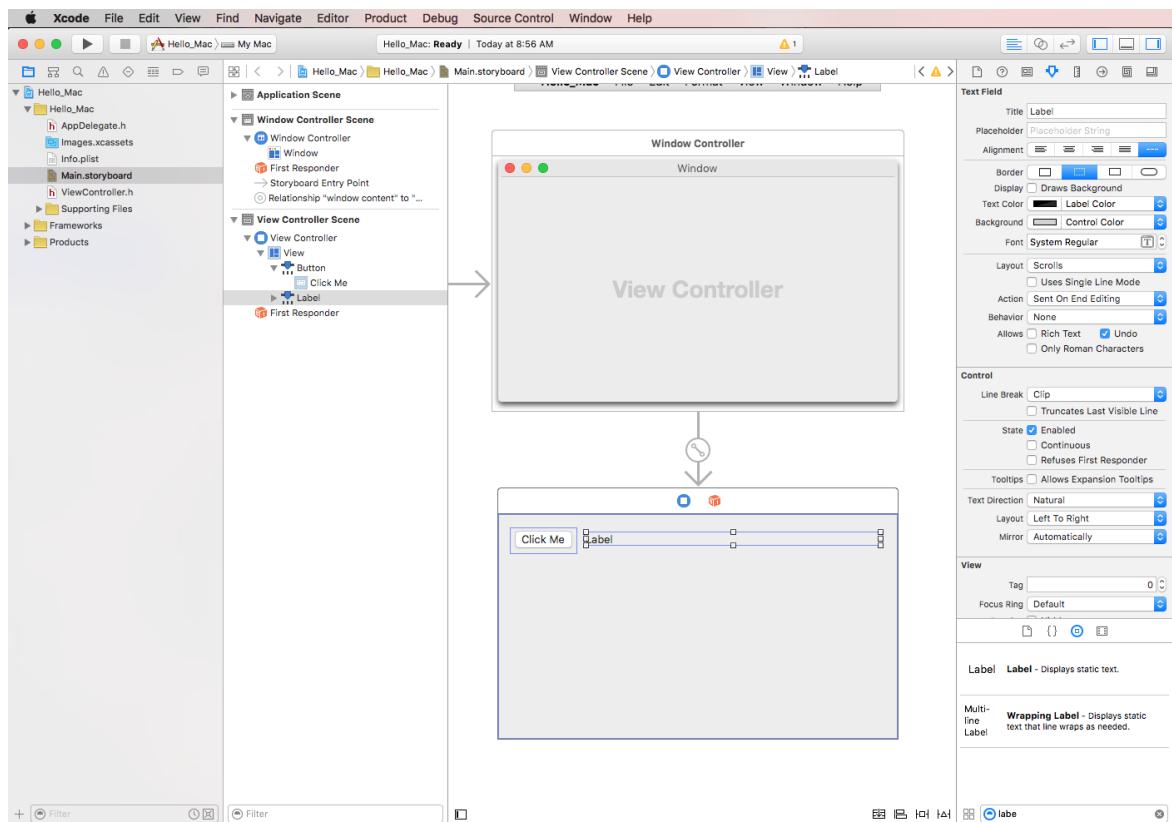
4. Drag a Label from the Library Section:



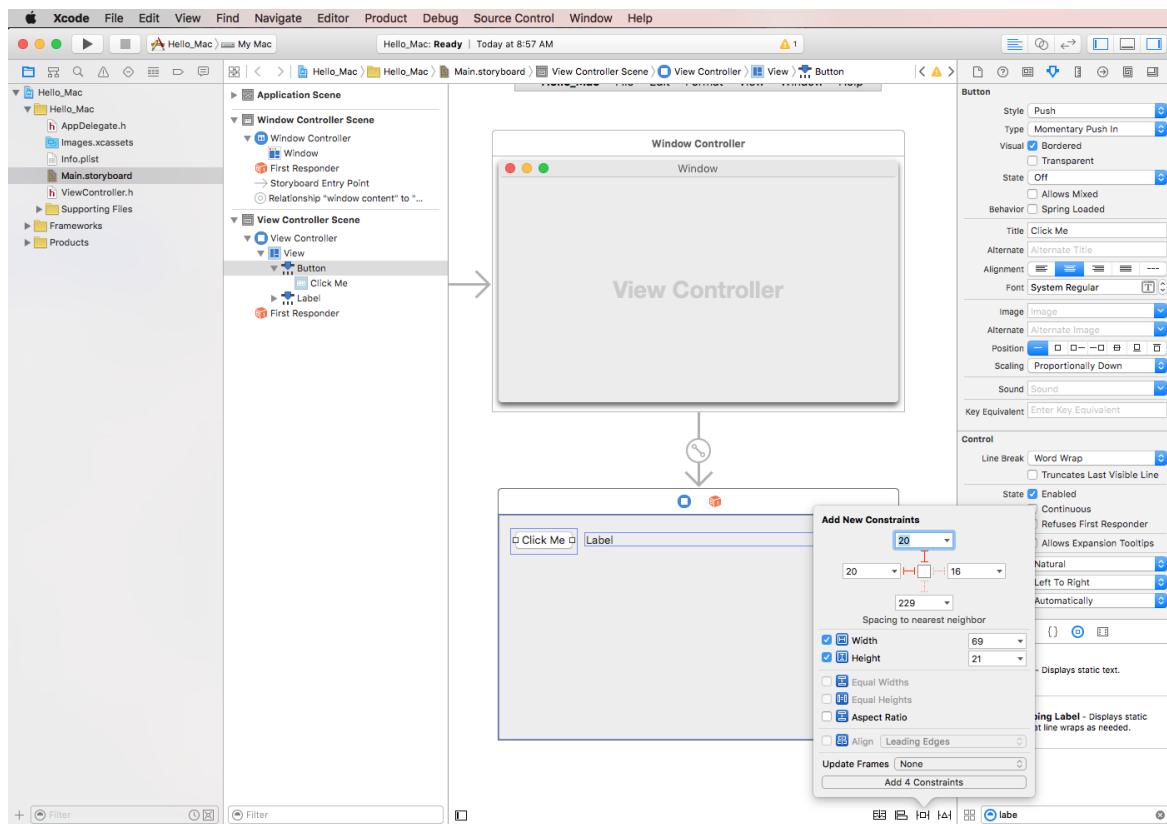
5. Drop the label onto the Window beside the button in the Interface Editor:



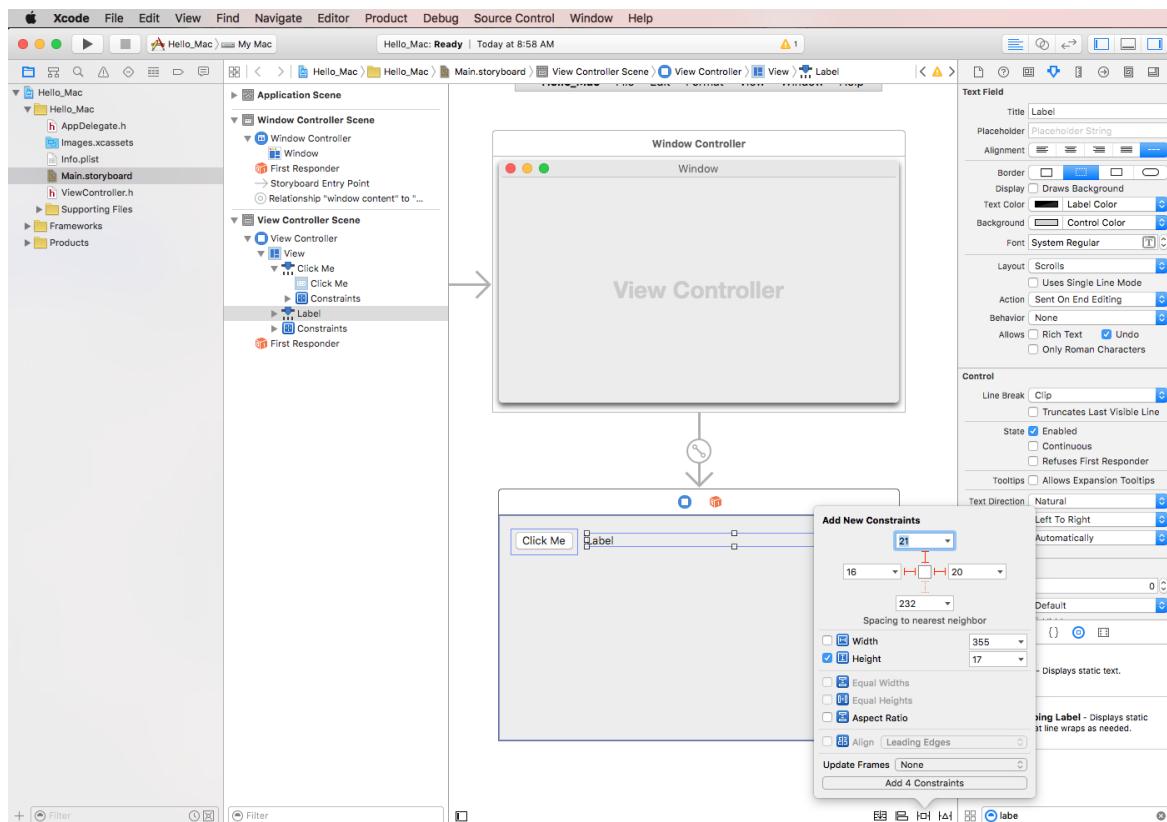
6. Grab the right handle on the label and drag it until it is near the edge of the window:



7. Select the Button just added in the **Interface Editor**, and click the **Constraints Editor** icon at the bottom of the window:



8. At the top of the editor, click the **Red I-Beams** at the top and left. As the window is resized, this will keep the button in the same location at the top left corner of the screen.
9. Next, check the **Height** and **Width** boxes and use the default sizes. This keeps the button at the same size when the window resizes.
10. Click the **Add 4 Constraints** button to add the constraints and close the editor.
11. Select the label and click the **Constraints Editor** icon again:



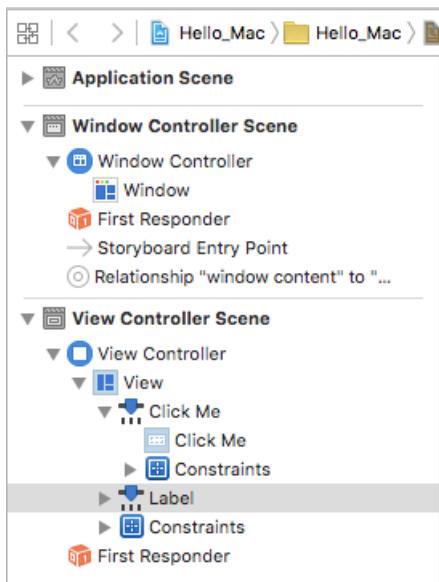
12. By clicking **Red I-Beams** at the top, right and left of the **Constraints Editor**, tells the label to be stuck to

its given X and Y locations and to grow and shrink as the window is resized in the running application.

13. Again, check the **Height** box and use the default size, then click the **Add 4 Constraints** button to add the constraints and close the editor.
14. Save the changes to the user interface.

While resizing and moving controls around, notice that Interface Builder gives helpful snap hints that are based on [macOS Human Interface Guidelines](#). These guidelines will help the developer to create high quality apps that will have a familiar look and feel for Mac users.

Look in the **Interface Hierarchy** section to see how the layout and hierarchy of the elements that make up the user interface are shown:



From here the developer can select items to edit or drag to reorder UI elements if needed. For example, if a UI element was being covered by another element, they could drag it to the bottom of the list to make it the top-most item on the window.

With the user interface created, the developer will need to expose the UI items so that Xamarin.Mac can access and interact with them in C# code. The next section, **Outlets and Actions**, shows how to do this.

Outlets and Actions

So what are **Outlets** and **Actions**? In traditional .NET user interface programming, a control in the user interface is automatically exposed as a property when it's added. Things work differently in Mac, simply adding a control to a view doesn't make it accessible to code. The developer must explicitly expose the UI element to code. In order to do this, Apple provides two options:

- **Outlets** – Outlets are analogous to properties. If the developer wires up a control to an Outlet, it's exposed to the code via a property, so they can do things like attach event handlers, call methods on it, etc.
- **Actions** – Actions are analogous to the command pattern in WPF. For example, when an Action is performed on a control, say a button click, the control will automatically call a method in the code. Actions are powerful and convenient because the developer can wire up many controls to the same Action.

In Xcode, **Outlets** and **Actions** are added directly in code via *Control-dragging*. More specifically, this means that to create an **Outlet** or **Action**, the developer will choose a control element to add an **Outlet** or **Action** to, hold down the **Control** key on the keyboard, and drag that control directly into the code.

For Xamarin.Mac developers, this means that the developer will drag into the Objective-C stub files that correspond to the C# file where they want to create the **Outlet** or **Action**. Visual Studio for Mac created a file called `ViewController.h` as part of the shim Xcode Project it generated to use Interface Builder:

The screenshot shows the Xcode interface with the 'Hello_Mac' project selected. The left sidebar displays the project structure, including 'AppDelegate.h', 'Info.plist', 'Main.storyboard', and 'ViewController.h'. The 'ViewController.h' file is currently selected and its code is visible in the main editor area:

```

// WARNING
// This file has been generated automatically by Xamarin Studio to
// mirror C# types. Changes in this file made by drag-connecting
// from the UI designer will be synchronized back to C#, but
// more complex manual changes may not transfer correctly.

#import <Foundation/Foundation.h>
#import <AppKit/AppKit.h>

@interface ViewController : NSViewController {
}

@end

```

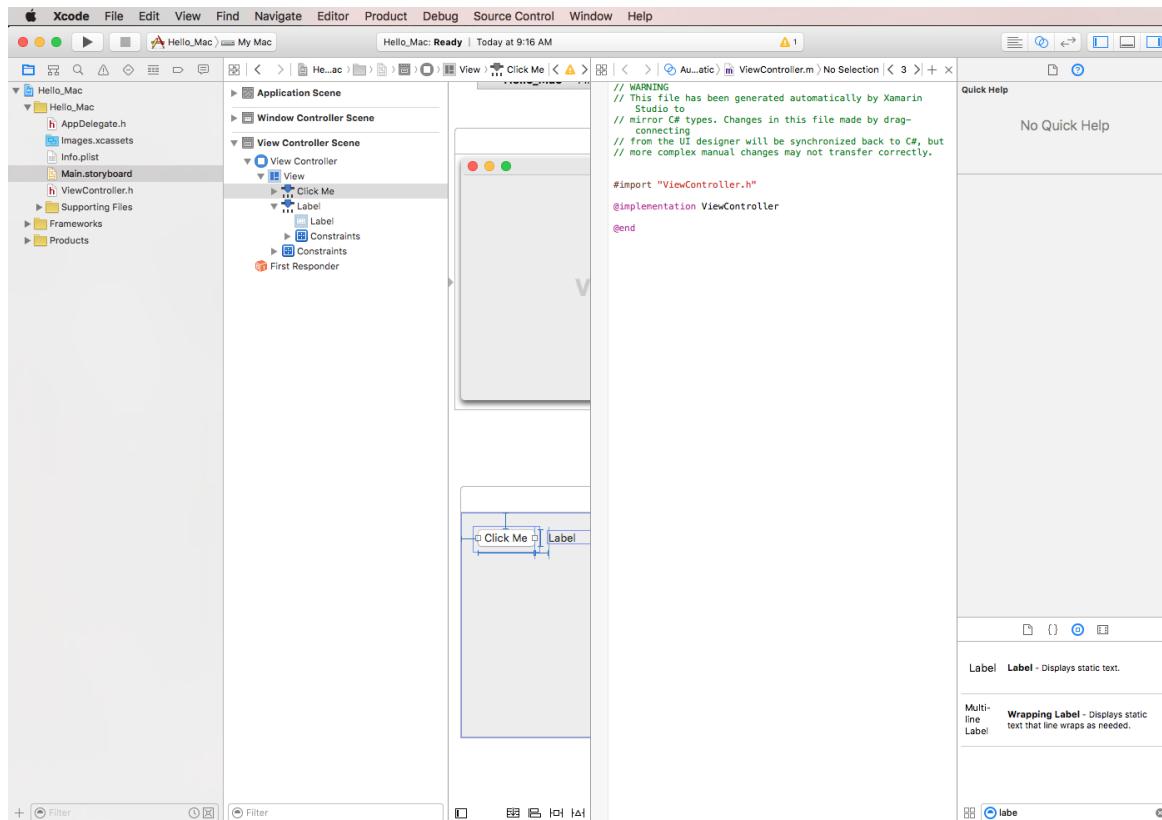
This stub `.h` file mirrors the `ViewController.designer.cs` that is automatically added to a Xamarin.Mac project when a new `NSWindow` is created. This file will be used to synchronize the changes made by Interface Builder and is where the **Outlets** and **Actions** are created so that UI elements are exposed to C# code.

Adding an Outlet

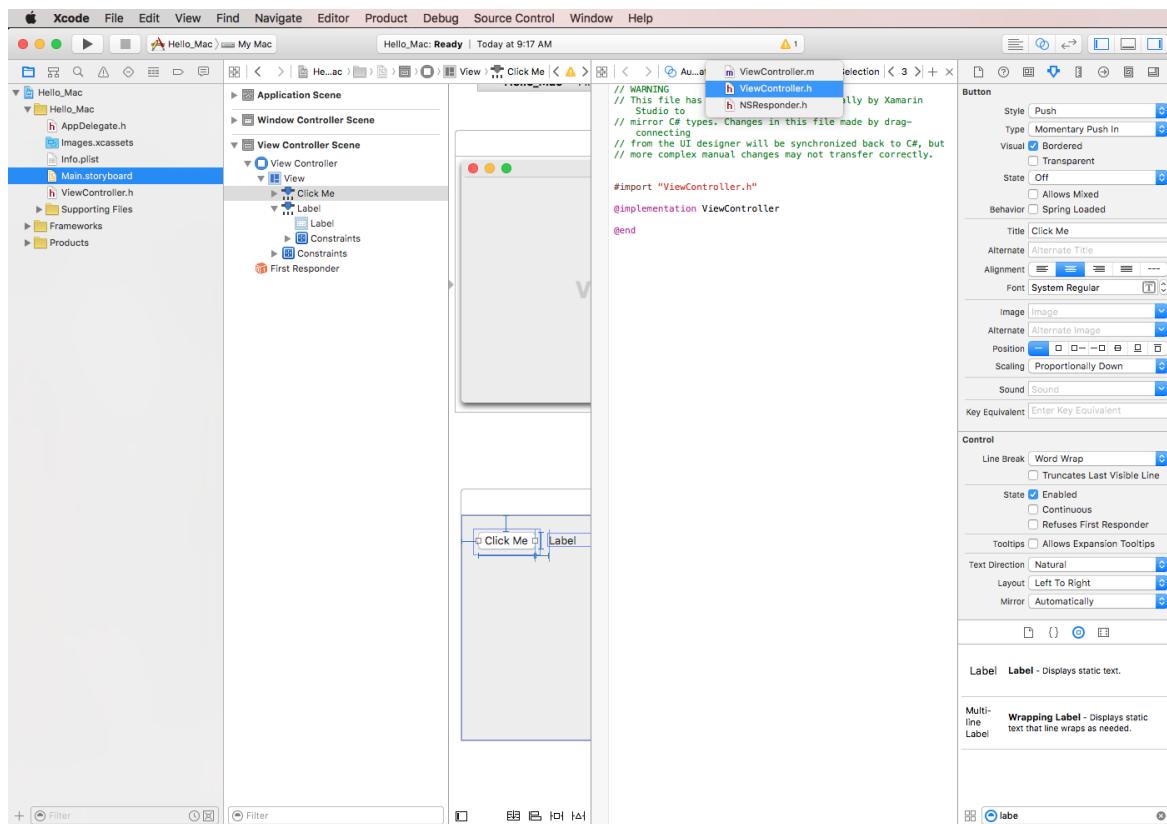
With a basic understanding of what **Outlets** and **Actions** are, create an **Outlet** to expose the Label created to our C# code.

Do the following:

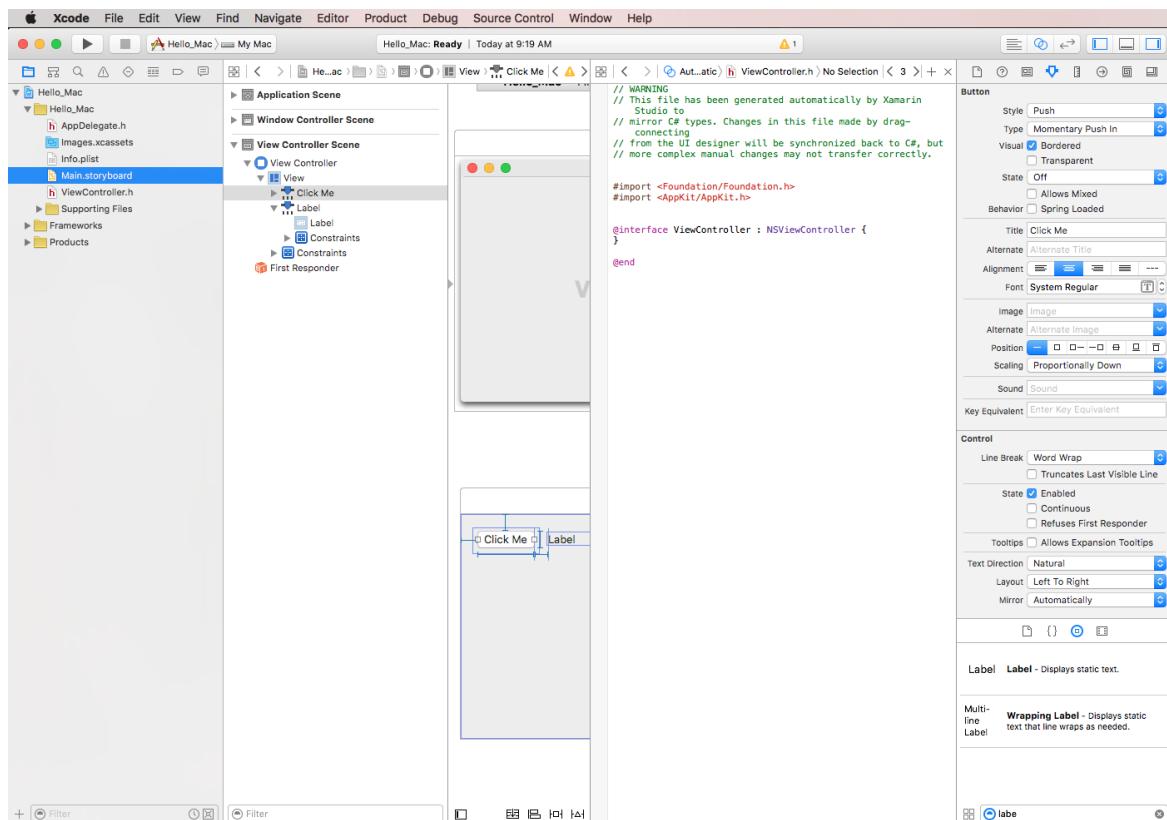
1. In Xcode at the far right top-hand corner of the screen, click the **Double Circle** button to open the **Assistant Editor**:



2. The Xcode will switch to a split-view mode with the **Interface Editor** on one side and a **Code Editor** on the other.
3. Notice that Xcode has automatically picked the `ViewController.m` file in the **Code Editor**, which is incorrect. From the discussion on what **Outlets** and **Actions** are above, the developer will need to have the `ViewController.h` selected.
4. At the top of the **Code Editor** click on the **Automatic Link** and select the `ViewController.h` file:

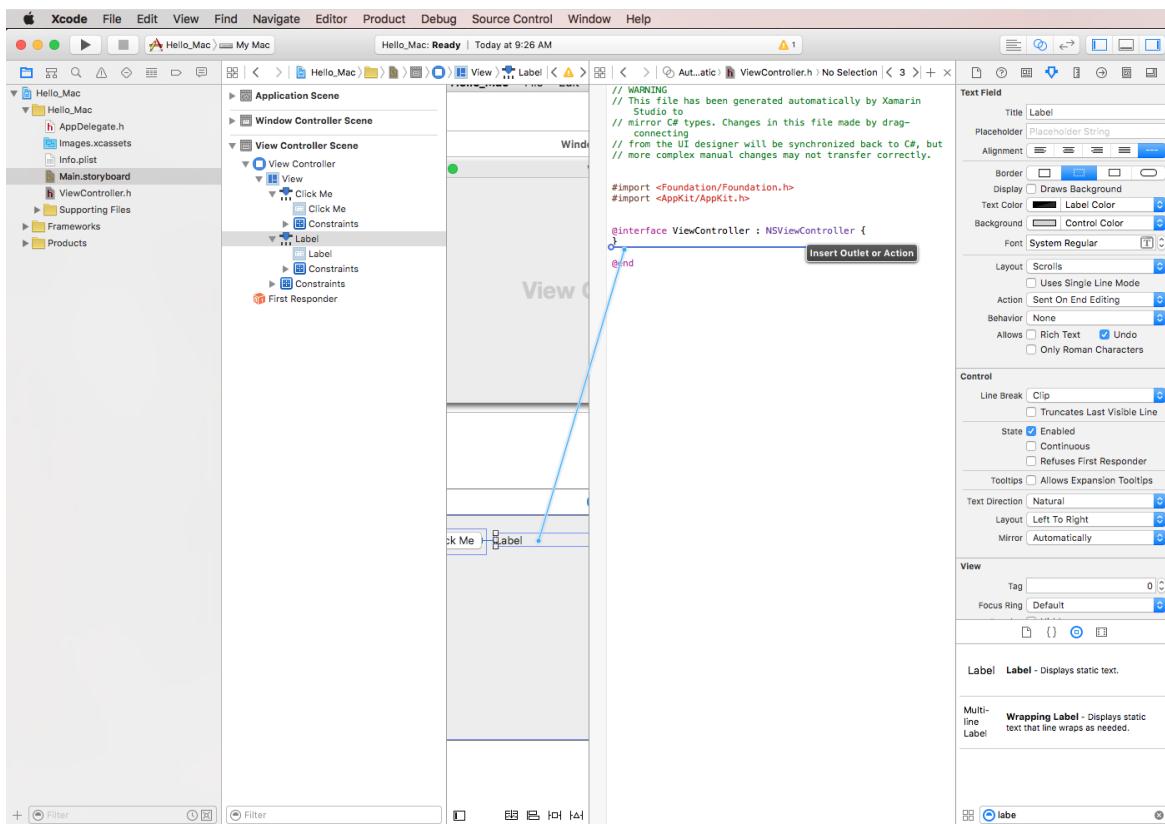


5. Xcode should now have the correct file selected:

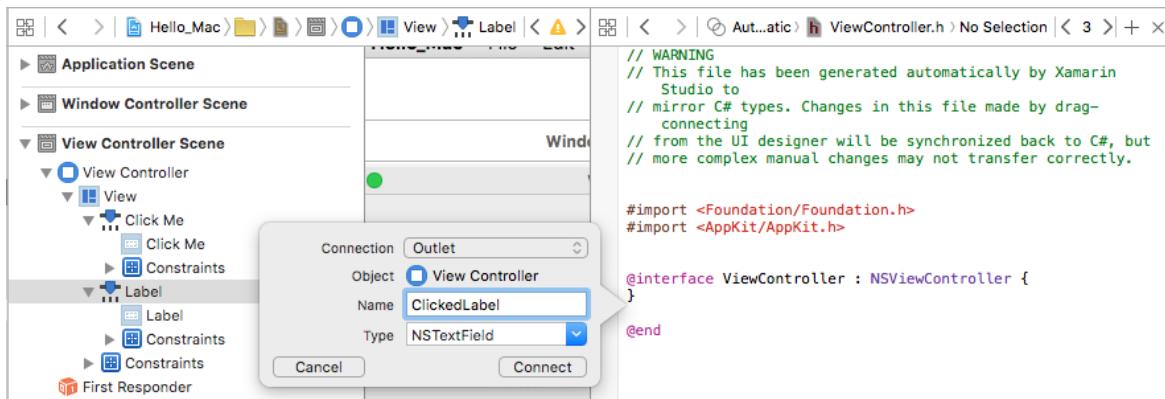


6. **The last step was very important!**: if you didn't have the correct file selected, you won't be able to create Outlets and Actions, or they will be exposed to the wrong class in C#!

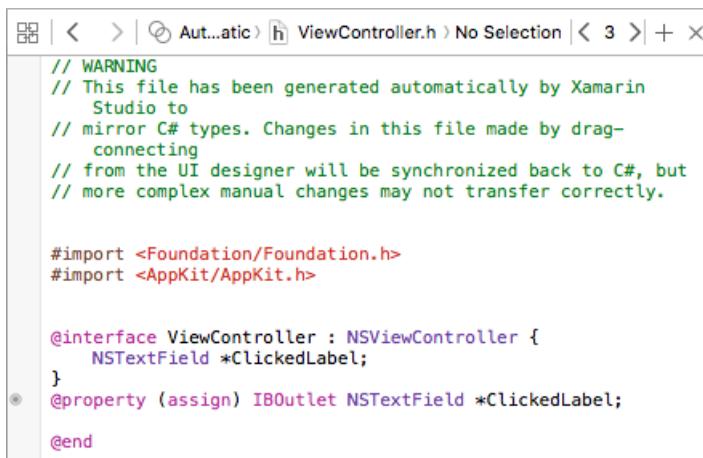
7. In the **Interface Editor**, hold down the **Control** key on the keyboard and click-drag the label created above onto the code editor just below the `@interface ViewController : NSViewController {}` code:



8. A dialog box will be displayed. Leave the Connection set to **Outlet** and enter `clickedLabel` for the Name:



9. Click the **Connect** button to create the Outlet:



10. Save the changes to the file.

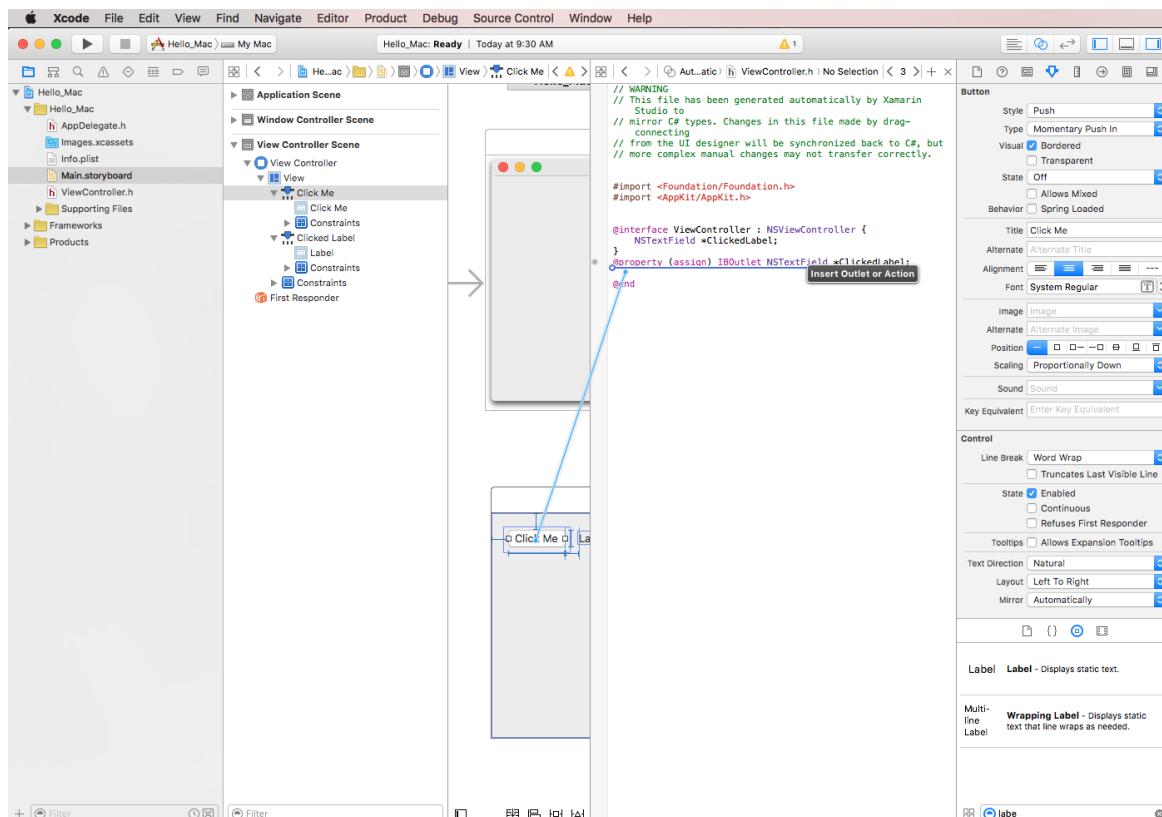
Adding an Action

Next, expose the button to C# code. Just like the Label above, the developer could wire the button up to an

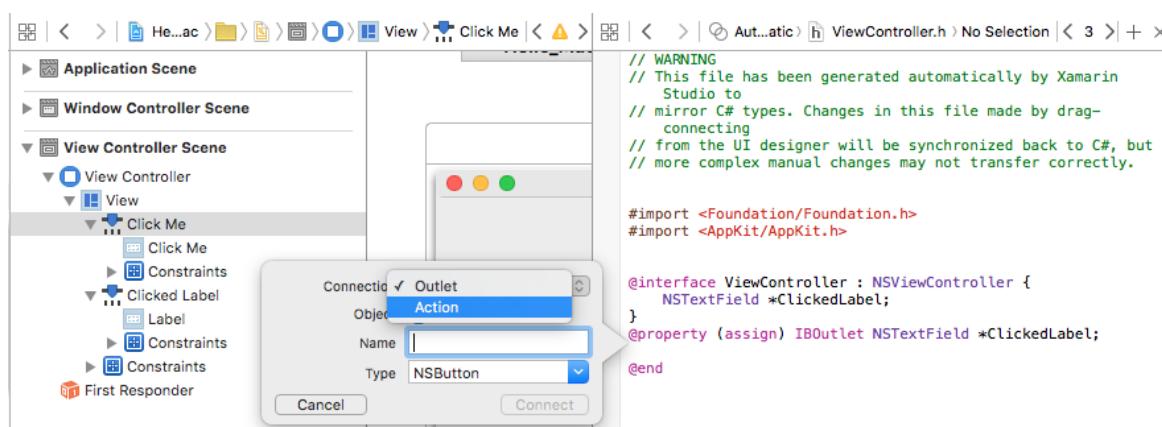
Outlet. Since we only want to respond to the button being clicked, use an **Action** instead.

Do the following:

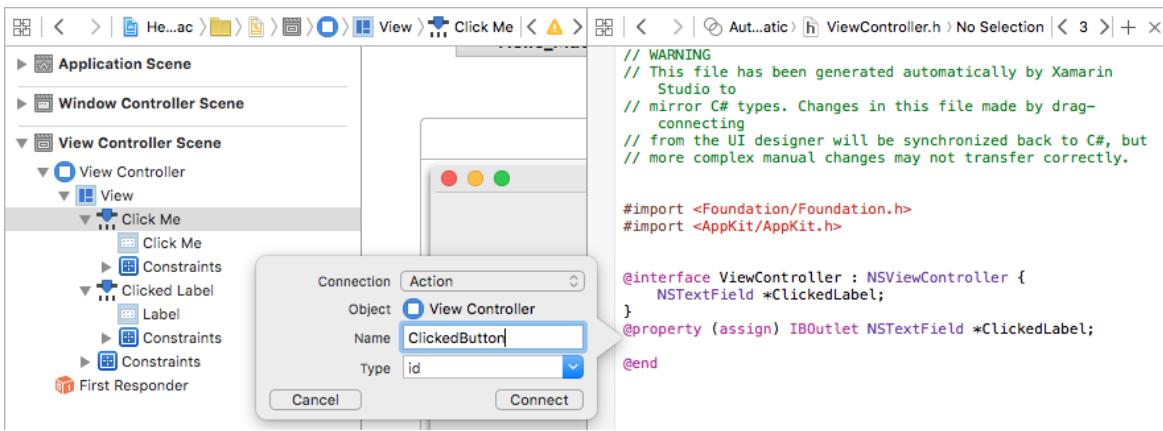
1. Ensure that Xcode is still in the **Assistant Editor** and the **ViewController.h** file is visible in the **Code Editor**.
2. In the **Interface Editor**, hold down the **Control** key on the keyboard and click-drag the button created above onto the code editor just below the `@property (assign) IBOutlet NSTextField *ClickedLabel;` code:



3. Change the Connection type to **Action**:



4. Enter `ClickedButton` as the **Name**:



5. Click the **Connect** button to create Action:

```
// WARNING
// This file has been generated automatically by Xamarin
// Studio to
// mirror C# types. Changes in this file made by drag-
// connecting
// from the UI designer will be synchronized back to C#, but
// more complex manual changes may not transfer correctly.

#import <Foundation/Foundation.h>
#import <AppKit/AppKit.h>

@interface ViewController : NSViewController {
    NSTextField *ClickedLabel;
}
@property (assign) IBOutlet NSTextField *ClickedLabel;
- (IBAction)ClickedButton:(id)sender;
@end
```

6. Save the changes to the file.

With the user interface wired-up and exposed to C# code, switch back to Visual Studio for Mac and let it synchronize the changes made in Xcode and Interface Builder.

NOTE

It probably took a long time to create the user interface and **Outlets** and **Actions** for this first app, and it may seem like a lot of work, but a lot of new concepts were introduced and a lot of time was spent covering new ground. After practicing for a while and working with Interface Builder, this interface and all its **Outlets** and **Actions** can be created in just a minute or two.

Synchronizing Changes with Xcode

When the developer switches back to Visual Studio for Mac from Xcode, any changes that they have made in Xcode will automatically be synchronized with the `Xamarin.Mac` project.

Select the `ViewController.designer.cs` in the **Solution Explorer** to see how the **Outlet** and **Action** have been wired up in the C# code:

The screenshot shows the Visual Studio Community interface. In the Solution Explorer on the left, there is a project named "Hello_Mac" with several files listed: AppDelegate.cs, Entitlements.plist, Info.plist, Main.cs, Main.storyboard, ViewController.cs, and ViewController.designer.cs. The ViewController.designer.cs file is currently selected. In the main code editor window, the code for ViewController.cs is displayed:

```
1 // WARNING ...
2 using Foundation;
3 using System.CodeDom.Compiler;
4
5 namespace Hello_Mac
6 {
7     [Register ("ViewController")]
8     partial class ViewController
9     {
10         [Outlet]
11         AppKit.NSTextField ClickedLabel { get; set; }
12
13         [Action ("ClickedButton:")]
14         partial void ClickedButton (Foundation.NSObject sender);
15
16         void ReleaseDesignerOutlets ()
17         {
18             if (ClickedLabel != null)
19                 ClickedLabel.Dispose ();
20             ClickedLabel = null;
21         }
22     }
23 }
24 }
```

Notice how the two definitions in the `ViewController.designer.cs` file:

```
[Outlet]
AppKit.NSTextField ClickedLabel { get; set; }

[Action ("ClickedButton:")]
partial void ClickedButton (Foundation.NSObject sender);
```

Line up with the definitions in the `ViewController.h` file in Xcode:

```
@property (assign) IBOutlet NSTextField *ClickedLabel;
- (IBAction)ClickedButton:(id)sender;
```

Visual Studio for Mac listens for changes to the `.h` file, and then automatically synchronizes those changes in the respective `.designer.cs` file to expose them to the app. Notice that `ViewController.designer.cs` is a partial class, so that Visual Studio for Mac doesn't have to modify `ViewController.cs` which would overwrite any changes that the developer has made to the class.

Normally, the developer will never need to open the `ViewController.designer.cs`, it was presented here for educational purposes only.

NOTE

In most situations, Visual Studio for Mac will automatically see any changes made in Xcode and sync them to the Xamarin.Mac project. In the off occurrence that synchronization doesn't automatically happen, switch back to Xcode and then back to Visual Studio for Mac again. This will normally kick off a synchronization cycle.

Writing the Code

With the user interface created and its UI elements exposed to code via **Outlets** and **Actions**, we are finally ready to write the code to bring the program to life.

For this sample app, every time the first button is clicked, the label will be updated to show how many times the button has been clicked. To accomplish this, open the `ViewController.cs` file for editing by double-clicking it in the **Solution Explorer**:

The screenshot shows the Visual Studio Community interface on a Mac. The title bar says "Visual Studio Community". The menu bar includes File, Edit, View, Search, Project, Build, Run, Version Control, Tools, Window, and Help. The toolbar has icons for Debug, Run, Stop, and others. The status bar says "Visual Studio Community 2017 for Mac". The Solution Explorer on the left shows a project named "Hello_Mac" with files like AppDelegate.cs, Entitlements.plist, Info.plist, Main.cs, Main.storyboard, and ViewController.cs. The "ViewController.cs" file is selected. The main editor window shows the following C# code:

```
1  using System;
2
3  using AppKit;
4  using Foundation;
5
6  namespace Hello_Mac
7  {
8      public partial class ViewController : NSViewController
9      {
10          public override NSObject RepresentedObject {
11              get {
12                  return base.RepresentedObject;
13              }
14              set {
15                  base.RepresentedObject = value;
16                  // Update the view, if already loaded.
17              }
18          }
19
20          public ViewController (IntPtr handle) : base (handle)
21          {
22          }
23
24          public override void ViewDidLoad ()
25          {
26              base.ViewDidLoad ();
27
28              // Do any additional setup after loading the view.
29          }
30      }
31  }
```

First, create a class-level variable in the `ViewController` class to track the number of clicks that have happened. Edit the class definition and make it look like the following:

```
namespace Hello_Mac
{
    public partial class ViewController : NSViewController
    {
        private int numberOfTimesClicked = 0;
        ...
    }
}
```

Next, in the same class (`ViewController`), override the `ViewDidLoad` method and add some code to set the initial message for the label:

```
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();

    // Set the initial value for the label
    ClickedLabel.StringValue = "Button has not been clicked yet.";
}
```

Use `ViewDidLoad`, instead of another method such as `Initialize`, because `ViewDidLoad` is called *after* the OS has loaded and instantiated the user interface from the `.storyboard` file. If the developer tried to access the label control before the `.storyboard` file has been fully loaded and instantiated, they would get a `NullReferenceException` error because the label control would not exist yet.

Next, add the code to respond to the user clicking the button. Add the following partial method to the `ViewController` class:

```
partial void ClickedButton (Foundation.NSObject sender) {
    // Update counter and label
    ClickedLabel.StringValue = string.Format("The button has been clicked {0}"
time{1}.",++numberOfTimesClicked, (numberOfTimesClicked < 2) ? "" : "s");
}
```

This code attaches to the **Action** created in Xcode and Interface Builder and will be called any time the user clicks the button.

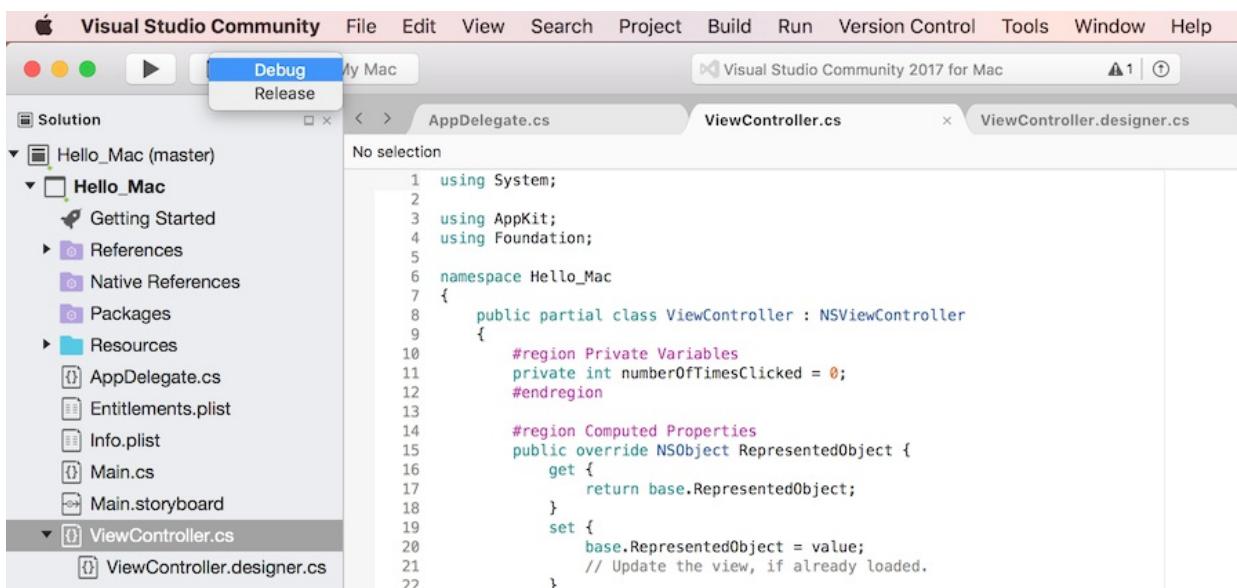
Testing the Application

It's time to build and run the app to make sure it runs as expected. The developer can build and run all in one step, or they can build it without running it.

Whenever an app is built, the developer can choose what kind of build they want:

- **Debug** – A debug build is compiled into an .app (application) file with a bunch of extra metadata that allows the developer to debug what's happening while the app is running.
- **Release** – A release build also creates an .app file, but it doesn't include debug information, so it's smaller and executes faster.

The developer can select the type of build from the **Configuration Selector** at the upper left-hand corner of the Visual Studio for Mac screen:



Building the Application

In the case of this example, we just want a debug build, so ensure that **Debug** is selected. Build the app first by either pressing **⌘B**, or from the **Build** menu, choose **Build All**.

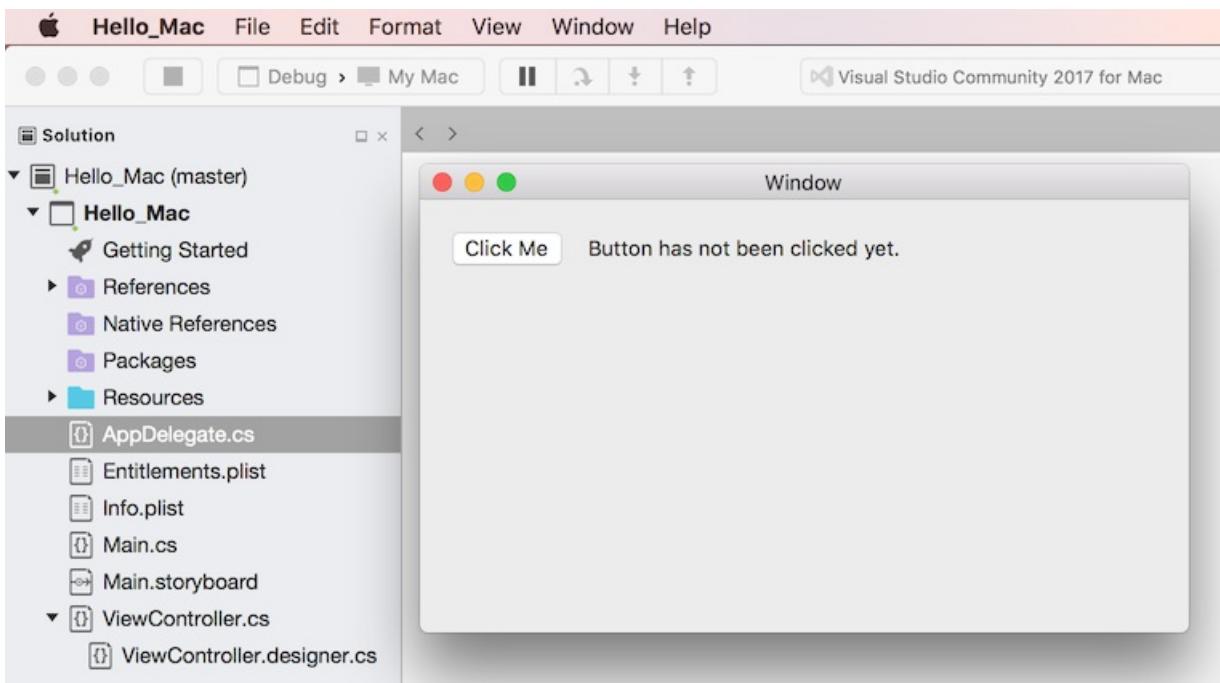
If there weren't any errors, a **Build Succeeded** message will be displayed in Visual Studio for Mac's status bar. If there were errors, review the project and make sure that the steps above have been followed correctly. Start by confirming that the code (both in Xcode and in Visual Studio for Mac) matches the code in the tutorial.

Running the Application

There are three ways to run the app:

- Press **⌘+Enter**.
- From the **Run** menu, choose **Debug**.
- Click the **Play** button in the Visual Studio for Mac toolbar (just above the **Solution Explorer**).

The app will build (if it hasn't been built already), start in debug mode and display its main interface window:



If the button is clicked a few times, the label should be updated with the count:



Where to Next

With the basics of working with a Xamarin.Mac application down, take a look at the following documents to get a deeper understanding:

- [Introduction to Storyboards](#) - This article provides an introduction to working with Storyboards in a Xamarin.Mac app. It covers creating and maintaining the app's UI using storyboards and Xcode's Interface Builder.
- [Windows](#) - This article covers working with Windows and Panels in a Xamarin.Mac application. It covers creating and maintaining Windows and Panels in Xcode and Interface builder, loading Windows and Panels from .xib files, using Windows and responding to Windows in C# code.
- [Dialogs](#) - This article covers working with Dialogs and Modal Windows in a Xamarin.Mac application. It covers creating and maintaining Modal Windows in Xcode and Interface builder, working with standard dialogs, displaying and responding to Windows in C# code.
- [Alerts](#) - This article covers working with Alerts in a Xamarin.Mac application. It covers creating and displaying Alerts from C# code and responding to Alerts.
- [Menus](#) - Menus are used in various parts of a Mac application's user interface; from the application's main menu at the top of the screen to pop up and contextual menus that can appear anywhere in a window. Menus are an integral part of a Mac application's user experience. This article covers working with Cocoa Menus in a Xamarin.Mac application.

- [Toolbars](#) - This article covers working with Toolbars in a Xamarin.Mac application. It covers creating and maintaining Toolbars in Xcode and Interface builder, how to expose the Toolbar Items to code using Outlets and Actions, enabling and disabling Toolbar Items and finally responding to Toolbar Items in C# code.
- [Table Views](#) - This article covers working with Table Views in a Xamarin.Mac application. It covers creating and maintaining Table Views in Xcode and Interface builder, how to expose the Table View Items to code using Outlets and Actions, populating Table Items and finally responding to Table View Items in C# code.
- [Outline Views](#) - This article covers working with Outline Views in a Xamarin.Mac application. It covers creating and maintaining Outline Views in Xcode and Interface builder, how to expose the Outline View Items to code using Outlets and Actions, populating Outline Items and finally responding to Outline View Items in C# code.
- [Source Lists](#) - This article covers working with Source Lists in a Xamarin.Mac application. It covers creating and maintaining Source Lists in Xcode and Interface builder, how to expose the Source Lists Items to code using Outlets and Actions, populating Source List Items and finally responding to Source List Items in C# code.
- [Collection Views](#) - This article covers working with Collection Views in a Xamarin.Mac application. It covers creating and maintaining Collection Views in Xcode and Interface builder, how to expose the Collection View elements to code using Outlets and Actions, populating Collection Views and finally responding to Collection Views in C# code.
- [Working with Images](#) - This article covers working with Images and Icons in a Xamarin.Mac application. It covers creating and maintaining the images needed to create an app's Icon and using Images in both C# code and Xcode's Interface Builder.

The [Mac Samples Gallery](#) contains ready-to-use code examples to help learn Xamarin.Mac.

One complete Xamarin.Mac app that includes many of the features a user would expect to find in a typical Mac application is the [SourceWriter Sample App](#). SourceWriter is a simple source code editor that provides support for code completion and simple syntax highlighting.

The SourceWriter code has been fully commented and, where available, links have been provided from key technologies or methods to relevant information in the Xamarin.Mac documentation.

Summary

This article covered the basics of a standard Xamarin.Mac app. It covered creating a new app in Visual Studio for Mac, designing the user interface in Xcode and Interface Builder, exposing UI elements to C# code using **Outlets** and **Actions**, adding code to work with the UI elements and finally, building and testing a Xamarin.Mac app.

Related Links

- [Hello, Mac \(sample\)](#)
- [macOS Human Interface Guidelines](#)

Xamarin.Mac – Related Documentation

10/28/2019 • 2 minutes to read • [Edit Online](#)

In addition to the Mac section of [Microsoft Docs](#) there are three great sources of documentation that can also be of assistance with Xamarin.Mac questions:

- **Xamarin.iOS documentation** - For many APIs (primarily outside of AppKit/UIKit) there are only small differences between the iOS and macOS versions. In some cases where a given iOS API has a name of `UIFoo`, a similar API named `NSFoo` can be found on macOS. These examples will be generally in C# already.
- **Apple's Mac Dev Center** - Many times an example of what APIs to call in Objective-C can be converted to C# in a straightforward manner. See [Understanding Mac APIs](#) for details on how to do this.
- **Stack Overflow** - A great resource for simple one off questions such as "[How do I auto expand all nodes in an NSOutlineView](#)". These examples will often be in Objective-C and need to be converted to C#, but there is a subset of answers in C#.

User Interface

When working with C# and .NET in a Xamarin.Mac application, the Developer has access to the same User Interface controls that a developer working in *Objective-C* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, the developer can use Xcode's *Interface Builder* to create and maintain an app's User Interfaces (or optionally create them directly in C# code).

The guides listed below give detailed information about working with macOS elements in a Xamarin.Mac application:

- [Windows](#)
- [Dialogs](#)
- [Alerts](#)
- [Menus](#)
- [Toolbars](#)
- [Table Views](#)
- [Outline Views](#)
- [Source Lists](#)

Xamarin.Mac application fundamentals

10/28/2019 • 2 minutes to read • [Edit Online](#)

Common patterns and idioms

Throughout the Apple APIs exposed via C#, certain idioms and patterns come up over and over again. If you have experience with programming with Xamarin.iOS, these may look familiar. Documentation will often refer to these patterns and idioms repeatedly, so having a solid understanding of them will help you make sense of the documentation you find.

Understanding Mac APIs

For much of your time developing with Xamarin.Mac, you can think, read, and write in C# without much concern with the underlying Objective-C APIs. However, sometimes you'll need to read the API documentation from Apple, translate an answer from Stack Overflow to a solution for your problem, or compare to an existing sample.

Console apps

You can also build "headless" console apps that access native macOS APIs using Xamarin.Mac.

Working with .xib files

This article covers working with .xib files created in Xcode's Interface Builder to create and maintain user interfaces for a Xamarin.Mac application.

.storyboard/.xib less user interface design

This article covers creating a Xamarin.Mac application's user interface directly from C# code without using Xcode's Interface Builder with .storyboard or .xib files.

Working with images

This article covers working with images and icons in a Xamarin.Mac application. It covers creating and maintaining the images needed to create your application's icon and using images in both C# code and Xcode's Interface Builder.

Data binding and key-value coding

This article covers using key-value coding and key-value observing to allow for data binding to UI elements in Xcode's Interface Builder. Using this technique, you greatly reduce the amount of C# code that needs to be written for your Xamarin.Mac application.

Working with databases

This article covers using key-value coding and key-value observing to allow for data binding with direct access to SQLite databases to UI elements in Xcode's Interface Builder. It also covers using the SQLite.NET ORM to provide access to SQLite data.

Working with copy and paste

This article covers working with the pasteboard to provide copy and paste in a Xamarin.Mac application. It shows how to work with standard data types that can be shared between multiple apps and how to support custom data within a give app.

Sandboxing a Xamarin.Mac app

This article covers sandboxing a Xamarin.Mac application for release on the App Store. It covers all of the elements that go into sandboxing: container directories, entitlements, user-determined permissions, privilege separation, and kernel enforcement.

Playing sound with AVAudioPlayer

This article shows how to use a helper class to control the playback of sound using an AVAudioPlayer.

Reporting bugs

Sometimes we all get stuck while working on a project, either on the inability to get an API to work the way we want or in trying to work around a bug. Our goal at Xamarin is for you to be successful in writing your mobile and desktop applications, and we've provided some resources to help.

Accessibility on macOS

12/12/2019 • 2 minutes to read • [Edit Online](#)

This page describes how to use the macOS Accessibility APIs to build apps according to the [accessibility checklist](#). Refer to the [Android accessibility](#) and [iOS accessibility](#) pages for other platform APIs.

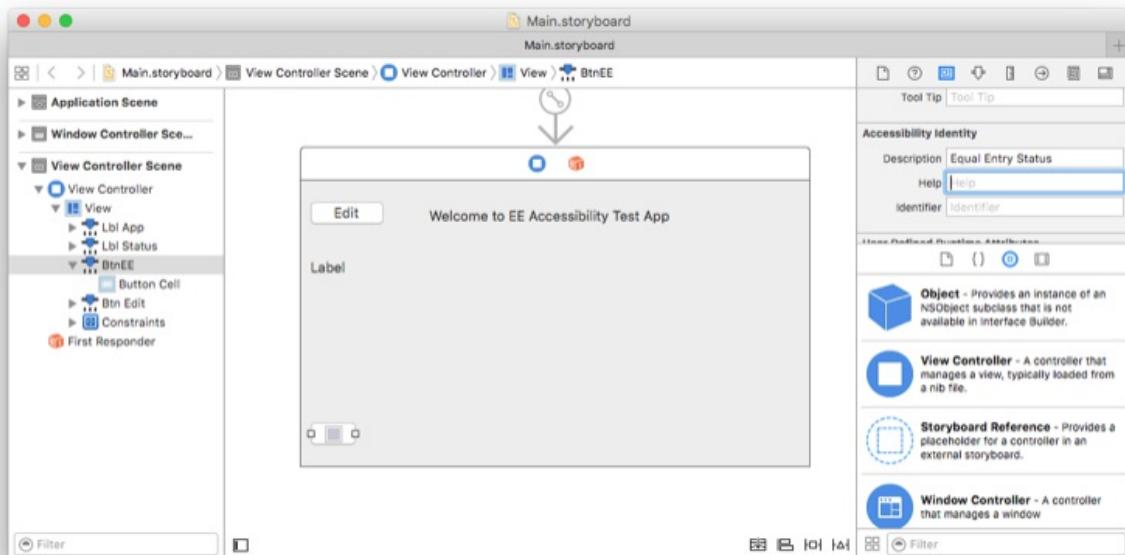
To understand how the accessibility APIs work in macOS (formerly called OS X), first review the [OS X accessibility model](#).

Describing UI elements

AppKit uses the `NSAccessibility` protocol to expose APIs that help make the user interface accessible. This includes a default behavior that attempts to set meaningful values for accessibility properties, such as setting a button's `AccessibilityLabel`. The label is typically a single word or short phrase describing the control or view.

Storyboard Files

Xamarin.Mac uses the Xcode Interface Builder to edit storyboard files. Accessibility information can be edited in the **Identity inspector** when a control is selected on the design surface (as shown in the screenshot below):



Code

Xamarin.Mac does not currently expose as `AccessibilityLabel` setter. Add the following helper method to set the accessibility label:

```

public static class AccessibilityHelper
{
    [System.Runtime.InteropServices.DllImport (ObjCRuntime.Constants.ObjectiveCLibrary)]
    extern static void objc_msgSend (IntPtr handle, IntPtr selector, IntPtr label);

    static public void SetAccessibilityLabel (this NSView view, string value)
    {
        objc_msgSend (view.Handle, new ObjCRuntime.Selector ("setAccessibilityLabel:").Handle, new NSString (value).Handle);
    }
}

```

This method can then be used in code as shown:

```
AccessibilityHelper.SetAccessibilityLabel (someButton, "New Accessible Description");
```

The `AccessibilityHelp` property is for an explanation of what the control or view does, and should only be added when the label may not provide sufficient information. The help text should still be kept as short as possible, for example "Deletes the document".

Some user interface elements are not relevant for accessible access (such as a label next to an input that has its own accessibility label and help). In these cases, set `AccessibilityElement = false` so that these controls or views will be skipped by screen readers or other accessibility tools.

Apple provides [accessibility guidelines](#) that explains the best practices for accessibility labels and help text.

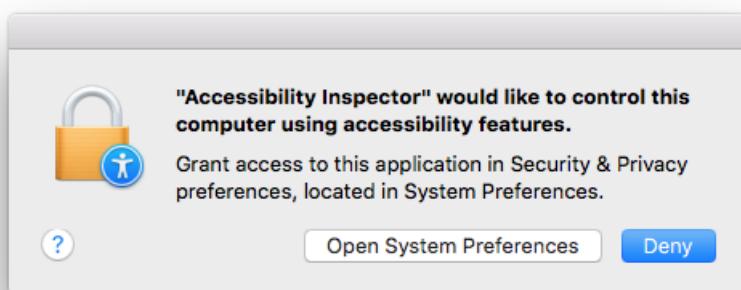
Custom controls

Refer to Apple's [guidelines for accessible custom controls](#) for details on the additional steps required.

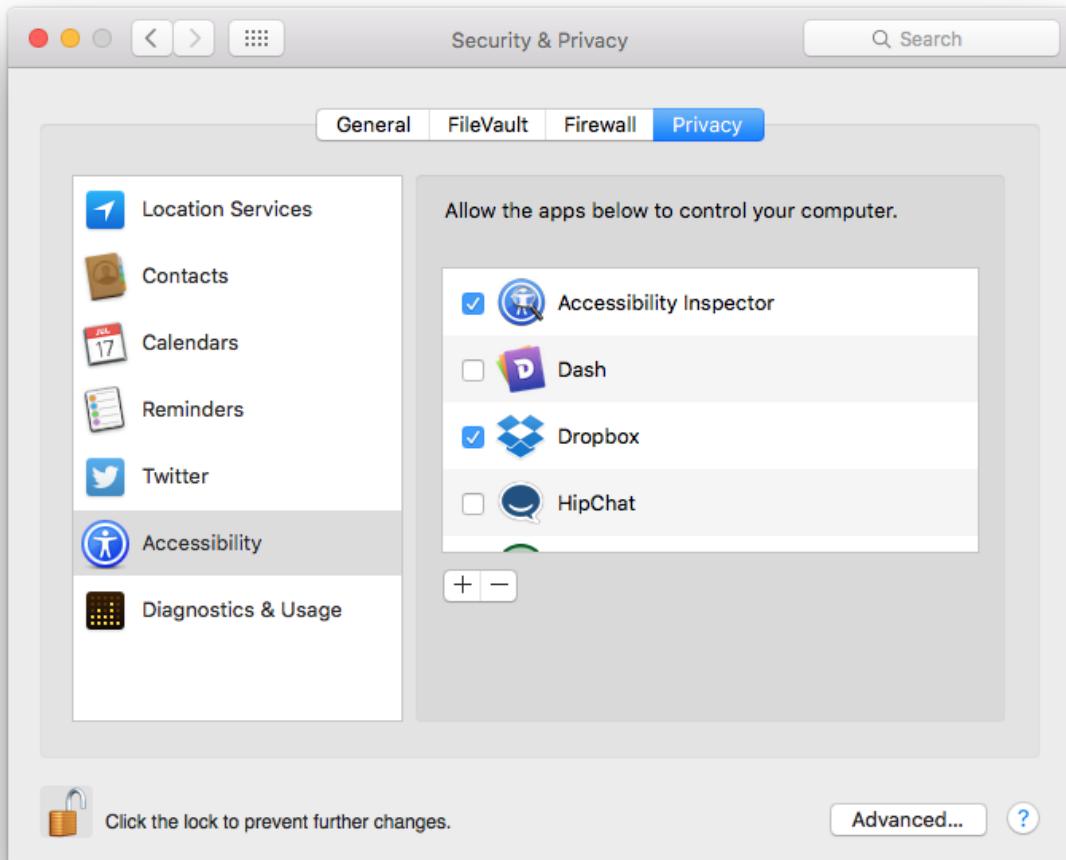
Testing accessibility

macOS provides an **Accessibility Inspector** that helps test accessibility functionality. The inspector is included with Xcode.

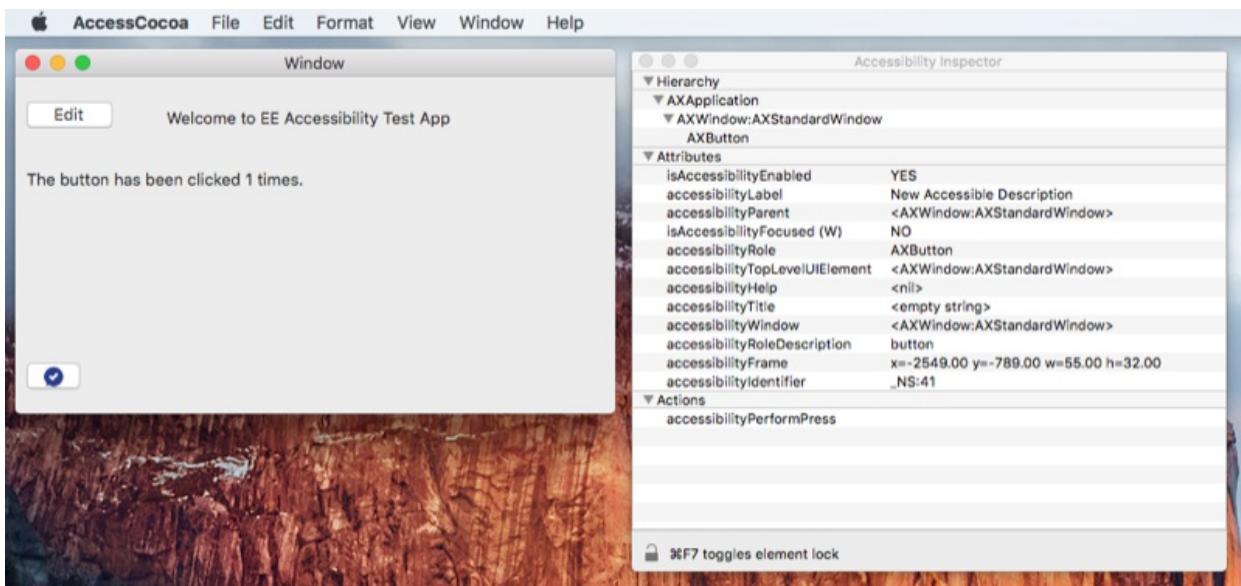
The first time it is launched, the **Accessibility Inspector** will require permission to control the computer via accessibility:



Unlock the settings screen (if required, on the lower-left) and tick the **Accessibility Inspector**:



Once enabled, the inspector appears as a floating window that can be moved around the screen. The screenshot below shows the inspector running next to a sample Mac app. As the cursor is moved over the window, the inspector displays all the accessible properties of each control:



For more information, read the [testing accessibility for OS X guide](#).

Related Links

- [Cross-platform accessibility](#)
- [Mac Accessibility](#)

Common patterns and idioms in Xamarin.Mac

10/28/2019 • 3 minutes to read • [Edit Online](#)

Throughout the Apple APIs exposed via C#, certain idioms and patterns come up over and over again. If you have experience with programming with Xamarin.iOS, these may look familiar. Documentation will often refer to these patterns and idioms repeatedly, so having a solid understanding of them will help you make sense of the documentation you find.

MVC - Model View Controller

Model View Controller, or MVC for short, is a very common pattern found throughout Cocoa. A detailed discussion is beyond the scope of this document, but in brief it is a way of structuring your application into components:

- **Model** objects represent the underlying data being viewed and manipulated (Like addresses in an address book)
- **View** objects handle the drawing of a given object on screen and handling user interaction (A text field showing the address on screen)
- **Controller** objects handle the interaction between the Model and View. They push Model changes "up" to update the View and push "down" changes from the View when users make changes in the UI.

If you are familiar with MVVM (Model View ViewModel) from other libraries such as WPF, the Controller acts similar to the ViewModel but is often more closely bound to the specific UI elements.

More details can be found here:

- [Learning MVC on Apple's website](#)
- [Model View Controller in Objective-C](#)
- [Working with Windows](#)

Data source / delegate / subclassing

Another very common pattern in Cocoa deals with providing data to UI elements and reacting to user interactions. Using `NSTableView` as an example, you need to somehow provide the data for each row, some set of UI elements that represent that row, some set of behaviors to react to user interactions, and possibly some amount of customization. The data source and delegate patterns let you handle most cases without having to resort to subclassing `NSTableView` yourself.

- The `DataSource` property is assigned an instance of a custom subclass of `NSTableViewDataSource` which is called to populate the table with data (via `GetRowCount` and `GetObjectValue`).
- The `Delegate` property is assigned an instance of a custom subclass of `NSTableViewDelegate` which provides the view for a given model object (via `GetViewForItem`) and handles UI interactions (via `DidClickTableColumn` , `MouseDownInHeaderOf TableColumn` , etc).

In some cases, you'll want to customize a control in a way beyond the hooks given in the delegate or data source and you can subclass the view directly. Be careful however, in many cases overriding default behavior will then require you to handle all of that behavior yourself (customizing selection behavior may require you to implement all of the selection behaviors yourself).

In Xamarin.iOS, some APIs, such as `UITableView` have been extended with a property that implements both the delegate and the data source (`UITableViewSource`). This is to work around the common limitation that a single C# class can only have one base class, and our surfacing of protocols is done via base classes.

For more information on working with table Views in a Xamarin.Mac application, please see our [Table View](#) documentation.

Protocols

Protocols in Objective-C can be compared to interfaces in C#, and in many cases are used in similar situations.

For example the `NSTableView` example above, both the delegate and the data source are actually protocols.

Xamarin.Mac exposes these as base classes with virtual methods you can override. The primary difference between C# interfaces and Objective-C protocols is that some methods in a protocol may be optional to implement. You'll have to look at the documentation and/or definition of an API to determine what is optional.

More information please see our [Delegates, Protocols and Events](#) documentation.

Related Links

- [Table views](#)
- [Working with windows](#)
- [Delegates, protocols, and events](#)
- [Model-View-Controller](#)

Xamarin.Mac bindings in console apps

9/6/2019 • 3 minutes to read • [Edit Online](#)

There are some scenarios where you want to use some of Apple native APIs in C# to build a headless application – one that does not have a user interface – using C#.

The project templates for Mac applications include a call to `NSApplication.Init()` followed by a call to `NSApplication.Main(args)`, it usually looks like this:

```
static class MainClass {
    static void Main (string [] args)
    {
        NSApplication.Init ();
        NSApplication.Main (args);
    }
}
```

The call to `Init` prepares the Xamarin.Mac runtime, the call to `Main(args)` starts the Cocoa application main loop, which prepares the application to receive keyboard and mouse events and show the main window of your application. The call to `Main` will also attempt to locate Cocoa resources, prepare a toplevel window and expects the program to be part of an application bundle (programs distributed in a directory with the `.app` extension and a very specific layout).

Headless applications do not need a user interface, and do not need to run as part of an application bundle.

Creating the console app

So it is better to start with a regular .NET Console project type.

You need to do a few things:

- Create an empty project.
- Reference the Xamarin.Mac.dll library.
- Bring the unmanaged dependency to your project.

These steps are explained in more detail below:

Create an empty Console Project

Create a new .NET Console Project, make sure that it is .NET and not .NET Core, as Xamarin.Mac.dll does not run under the .NET Core runtime, it only runs with the Mono runtime.

Reference the Xamarin.Mac library

To compile your code, you will want to reference the `Xamarin.Mac.dll` assembly from this directory:

```
/Library/Frameworks/Xamarin.Mac.framework/Versions/Current//lib/x86_64/full
```

To do this, go to the project references, select the `.NET Assembly` tab, and click the `Browse` button to locate the file on the file system. Navigate to the path above, and then select the `Xamarin.Mac.dll` from that directory.

This will give you access to the Cocoa APIs at compile time. At this point, you can add `using AppKit` to the top of your file, and call the `NSApplication.Init()` method. There is only one more step before you can run your application.

Bring the unmanaged support library into your project

Before your application will run, you need to bring the `xamarin.Mac` support library into your project. To do this, add a new file to your project (in project options, select **Add**, and then **Add Existing File**) and navigate to this directory:

```
/Library/Frameworks/Xamarin.Mac.framework/Versions/Current/lib
```

Here, select the file `libxammac.dylib`. You will be offered a choice of copying, linking or moving. I personally like linking, but copying works as well. Then you need to select the file, and in the property pad (select **View>Pads>Properties** if the property pad is not visible), go to the **Build** section and set the **Copy to Output Directory** setting to **Copy if newer**.

You can now run your Xamarin.Mac application.

The result in your bin directory will look like this:

```
Xamarin.Mac.dll  
Xamarin.Mac.pdb  
consoleapp.exe  
consoleapp.pdb  
libxammac.dylib
```

To run this app, you will need all those files in the same directory.

Building a standalone application for distribution

You might want to distribute a single executable to your users. To do this, you can use the `mkbundle` tool to turn the various files into a self-contained executable.

First, make sure that your application compiles and runs. Once you are satisfied with the results, you can run the following command from the command line:

```
$ mkbundle --simple -o /tmp/consoleapp consoleapp.exe --library libxammac.dylib --config  
/Library/Frameworks/Mono.framework/Versions/Current/etc/mono/config --machine-config  
/Library/Frameworks/Mono.framework/Versions/Current//etc/mono/4.5/machine.config  
[Output from the bundling tool]  
$ _
```

In the above command line invocation, the option `-o` is used to specify the generated output, in this case, we passed `/tmp/consoleapp`. This is now a standalone application that you can distribute and has no external dependencies on Mono or Xamarin.Mac, it is a fully self contained executable.

The command line manually specified the `machine.config` file to use, and a system-wide library mapping configuration file. They are not necessary for all applications, but it is convenient to bundle them, as they are used when you use more capabilities of .NET

Project-less builds

You do not need a full project to create a self-contained Xamarin.Mac application, you can also use simple Unix makefiles to get the job done. The following example shows how you can setup a makefile for a simple command line application:

```
XAMMAC_PATH=/Library/Frameworks/Xamarin.Mac.framework/Versions/Current//lib/x86_64/full/
DYLD=/Library/Frameworks/Xamarin.Mac.framework/Versions/Current//lib
MONODIR=/Library/Frameworks/Mono.framework/Versions/Current/etc/mono

all: consoleapp.exe

consoleapp.exe: consoleapp.cs Makefile
    mcs -g -r:$(_XAMMAC_PATH)/Xamarin.Mac.dll consoleapp.cs

run: consoleapp.exe
    MONO_PATH=$(_XAMMAC_PATH) DYLD_LIBRARY_PATH=$(DYLD) mono --debug consoleapp.exe $(COMMAND)

bundle: consoleapp.exe
    mkbundle --simple consoleapp.exe -o ncsharp -L $(XAMMAC_PATH) --library $(DYLD)/libxammac.dylib --config
    $(MONODIR)/config --machine-config $(MONODIR)/4.5/machine.config
```

The above `Makefile` provides three targets:

- `make` will build the program
- `make run` will build and run the program in the current directory
- `make bundle` will create a self-contained executable

macOS APIs for Xamarin.Mac Developers

11/2/2020 • 2 minutes to read • [Edit Online](#)

Overview

For much of your time developing with Xamarin.Mac, you can think, read, and write in C# without much concern with the underlying Objective-C APIs. However, sometimes you'll need to read the API documentation from Apple, translate an answer from Stack Overflow to a solution for your problem, or compare to an existing sample.

Reading enough Objective-C to be dangerous

Sometimes it will be necessary to read an Objective-C definition or method call and translate that to the equivalent C# method. Let's take a look at an Objective-C function definition and break down the pieces. This method (a *selector* in Objective-C) can be found on `NSTableView`:

```
- (BOOL)canDragRowsWithIndexes:(NSIndexSet *)rowIndexes atPoint:(NSPoint)mouseDownPoint
```

The declaration can be read left to right:

- The `-` prefix means it is an instance (non-static) method. `+` means it is a class (static) method
- `(BOOL)` is the return type (bool in C#)
- `canDragRowsWithIndexes` is the first part of the name.
- `(NSIndexSet *)rowIndexes` is the first param and with its type. The first parameter is in the format:
`(Type) paramName`
- `atPoint:(NSPoint)mouseDownPoint` is the second param and its type. Every parameter after the first is in the format: `selectorPart:(Type) paramName`
- The full name of this message selector is: `canDragRowsWithIndexes:atPoint:`. Note the `:` at the end - it is important.
- The actual Xamarin.Mac C# binding is: `bool CanDragRows (NSIndexSet rowIndexes, PointF mouseDownPoint)`

This selector invocation can be read the same way:

```
[v canDragRowsWithIndexes:set atPoint:point];
```

- The instance `v` is having its `canDragRowsWithIndexes:atPoint` selector called with two parameters, `set` and `point`, passed in.
- In C#, The method invocation looks like this: `v.CanDragRows (set, point);`

Finding the C# member for a given selector

Now that you've found the Objective-C selector you need to invoke, the next step is mapping that to the equivalent C# member. There are four approaches you can try (continuing with the `NSTableView CanDragRows` example):

1. Use the auto completion list to quickly scan for something of the same name. Since we know it is an instance of `NSTableView` you can type:

- `NSTableView x;`
- `x.` [ctrl+space if the list does not appear).
- `CanDrag` [enter]
- Right-click the method, go to declaration to open the Assembly Browser where you can compare the `Export` attribute to the selector in question

2. Search the entire class binding. Since we know it is an instance of `NSTableView` you can type:

- `NSTableView x;`
- Right-click `NSTableView`, go to declaration to Assembly Browser
- Search for the selector in question

3. You can use the [Xamarin.Mac API online documentation](#).

4. Miguel provides a "Rosetta Stone" view of the Xamarin.Mac APIs [here](#) that you can search through for a given API. If your API is not AppKit or macOS specific, you may find it there.

.xib files in Xamarin.Mac

3/5/2021 • 20 minutes to read • [Edit Online](#)

This article covers working with .xib files created in Xcode's Interface Builder to create and maintain user interfaces for a Xamarin.Mac application.

NOTE

The preferred way to create a user interface for a Xamarin.Mac app is with storyboards. This documentation has been left in place for historical reasons and for working with older Xamarin.Mac projects. For more information, please see our [Introduction to Storyboards](#) documentation.

Overview

When working with C# and .NET in a Xamarin.Mac application, you have access to the same user interface elements and tools that a developer working in *Objective-C* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, you can use Xcode's *Interface Builder* to create and maintain your user interfaces (or optionally create them directly in C# code).

A .xib file is used by macOS to define elements of your application's user interface (such as Menus, Windows, Views, Labels, Text Fields) that are created and maintained graphically in Xcode's Interface Builder.



In this article, we'll cover the basics of working with .xib files in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, as it covers key concepts and techniques that we'll be using in this article.

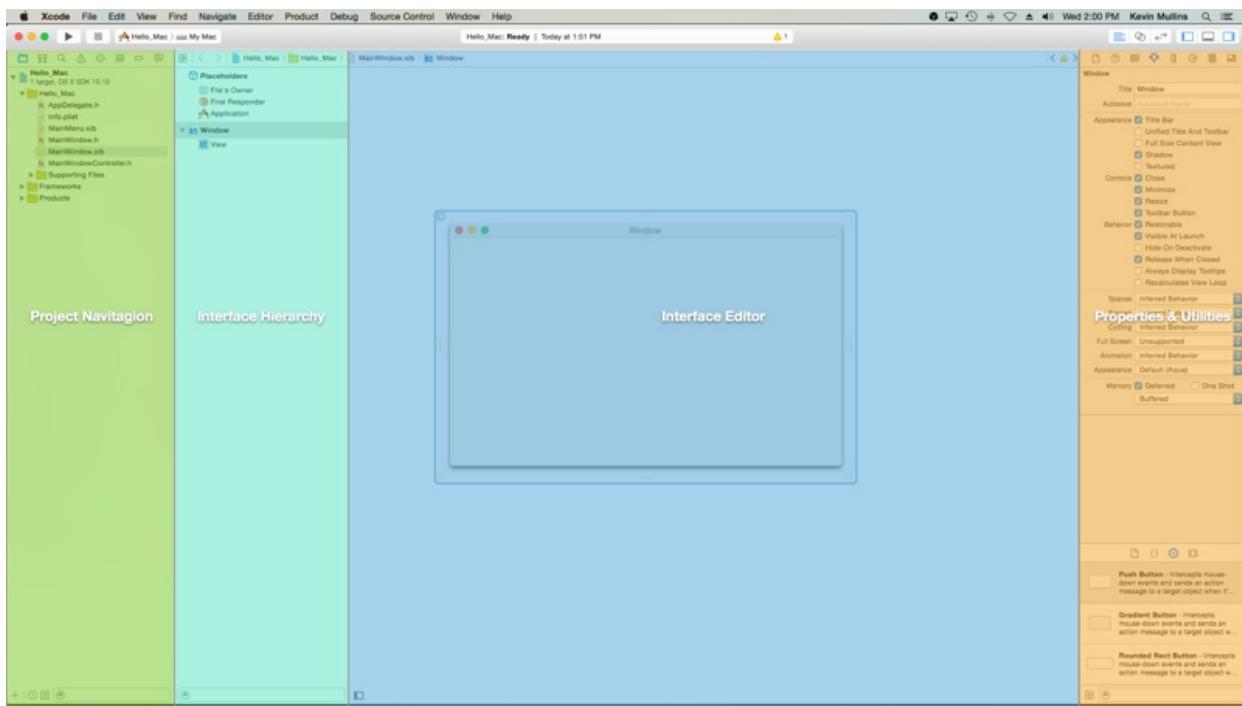
You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` attributes used to wire up your C# classes to Objective-C objects and UI elements.

Introduction to Xcode and Interface Builder

As part of Xcode, Apple has created a tool called Interface Builder, which allows you to create your User Interface visually in a designer. Xamarin.Mac integrates fluently with Interface Builder, allowing you to create your UI with the same tools that Objective-C users do.

Components of Xcode

When you open a .xib file in Xcode from Visual Studio for Mac, it opens with a **Project Navigator** on the left, the **Interface Hierarchy** and **Interface Editor** in the middle, and a **Properties & Utilities** section on the right:



Let's take a look at what each of these Xcode sections does and how you will use them to create the interface for your Xamarin.Mac application.

Project navigation

When you open a .xib file for editing in Xcode, Visual Studio for Mac creates an Xcode project file in the background to communicate changes between itself and Xcode. Later, when you switch back to Visual Studio for Mac from Xcode, any changes made to this project are synchronized with your Xamarin.Mac project by Visual Studio for Mac.

The **Project Navigation** section allows you to navigate between all of the files that make up this *shim* Xcode project. Typically, you will only be interested in the .xib files in this list such as **MainMenu.xib** and **MainWindow.xib**.

Interface hierarchy

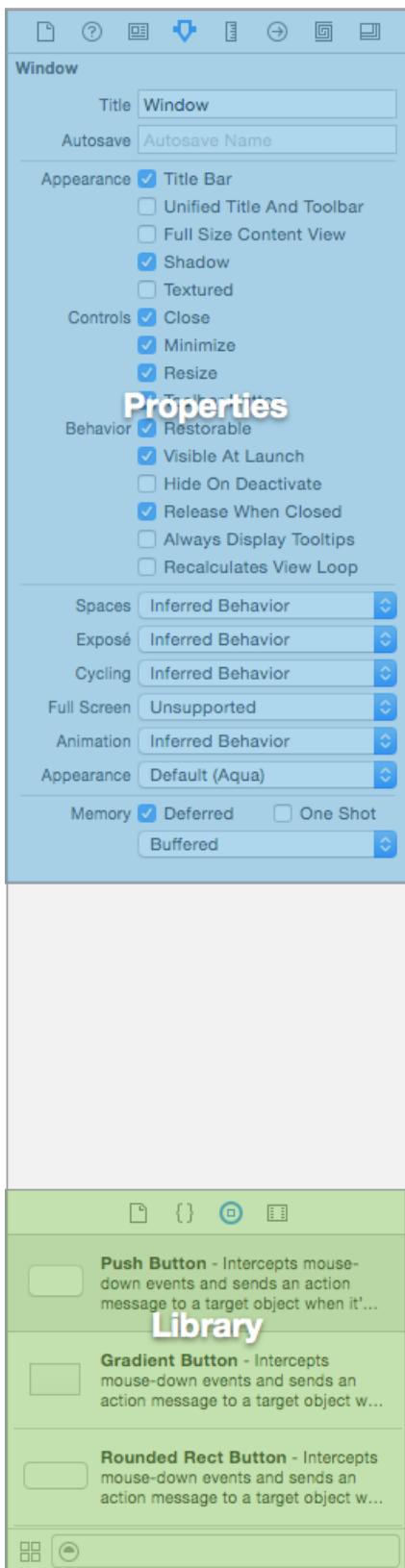
The **Interface Hierarchy** section allows you to easily access several key properties of the User Interface such as its **Placeholders** and main **Window**. You can also use this section to access the individual elements (views) that make up your user interface and the adjust the way that they are nested by dragging them around within the hierarchy.

Interface editor

The **Interface Editor** section provides the surface on which you graphically layout your User Interface. You'll drag elements from the **Library** section of the **Properties & Utilities** section to create your design. As you add user interface elements (views) to the design surface, they will be added to the **Interface Hierarchy** section in the order that they appear in the **Interface Editor**.

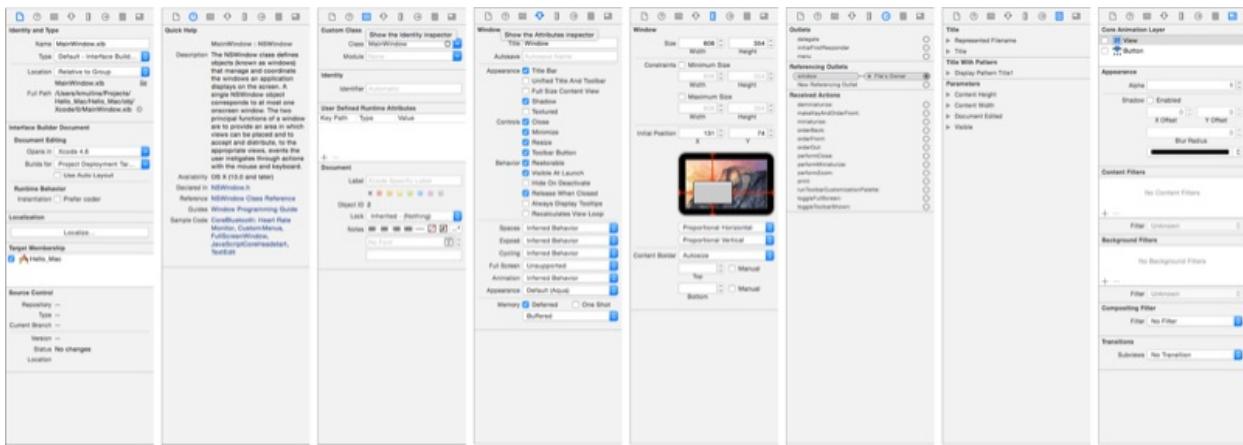
Properties & utilities

The **Properties & Utilities** section is divided into two main sections that we will be working with, **Properties** (also called Inspectors) and the **Library**:



Initially this section is almost empty, however if you select an element in the **Interface Editor** or **Interface Hierarchy**, the **Properties** section will be populated with information about the given element and properties that you can adjust.

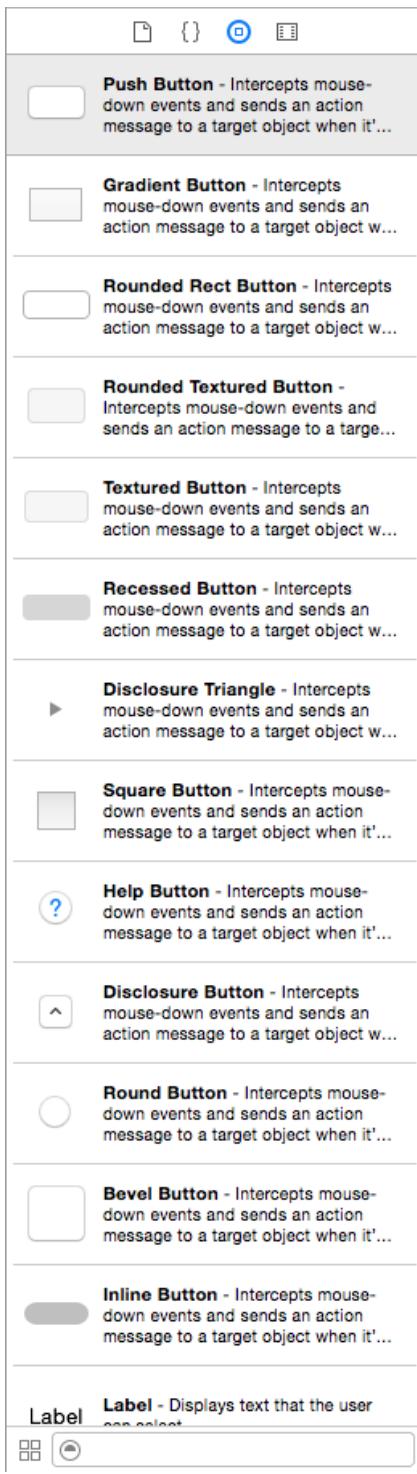
Within the **Properties** section, there are 8 different *Inspector Tabs*, as shown in the following illustration:



From left-to-right, these tabs are:

- **File Inspector** – The File Inspector shows file information, such as the file name and location of the Xib file that is being edited.
- **Quick Help** – The Quick Help tab provides contextual help based on what is selected in Xcode.
- **Identity Inspector** – The Identity Inspector provides information about the selected control/view.
- **Attributes Inspector** – The Attributes Inspector allows you to customize various attributes of the selected control/view.
- **Size Inspector** – The Size Inspector allows you to control the size and resizing behavior of the selected control/view.
- **Connections Inspector** – The Connections Inspector shows the outlet and action connections of the selected controls. We'll examine Outlets and Actions in just a moment.
- **Bindings Inspector** – The Bindings Inspector allows you to configure controls so that their values are automatically bound to data models.
- **View Effects Inspector** – The View Effects Inspector allows you to specify effects on the controls, such as animations.

In the **Library** section, you can find controls and objects to place into the designer to graphically build your user interface:



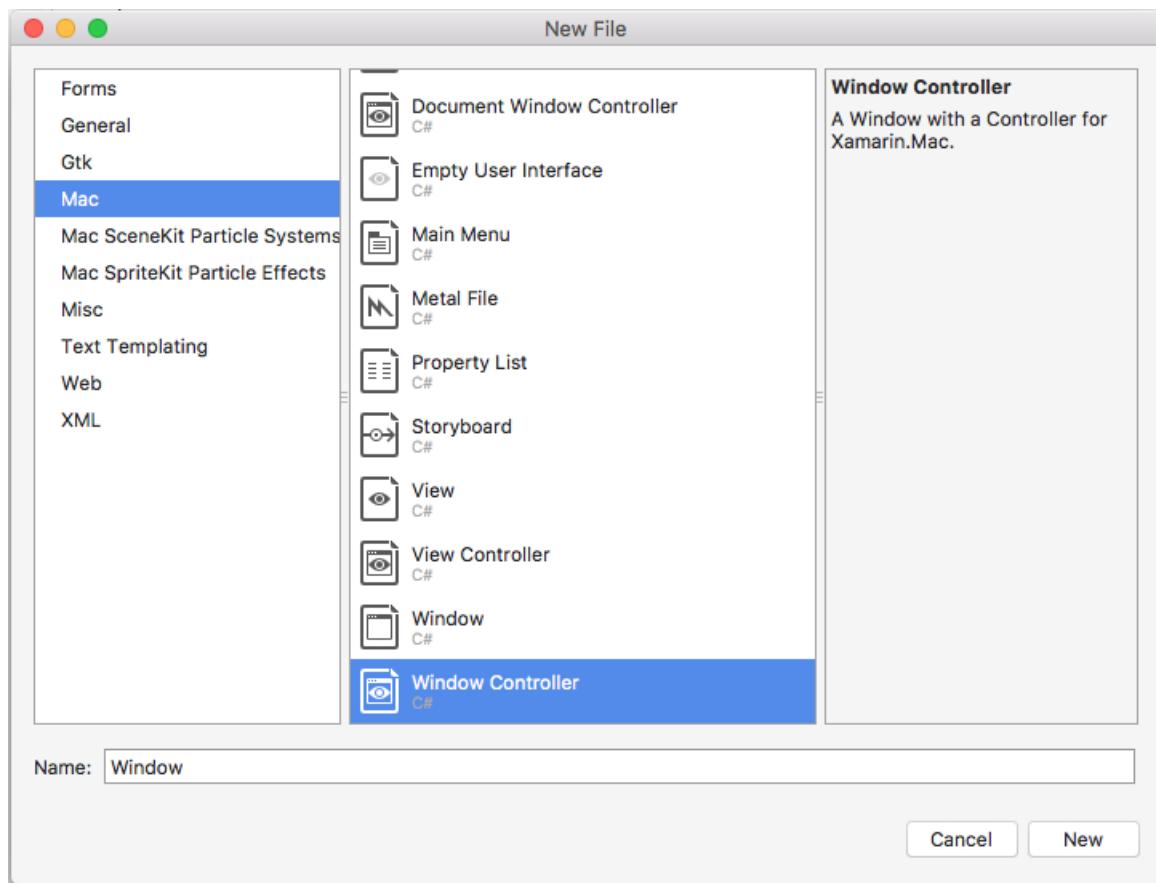
Now that you are familiar with the Xcode IDE and Interface Builder, let's look at using it to create a user interface.

Creating and maintaining windows in Xcode

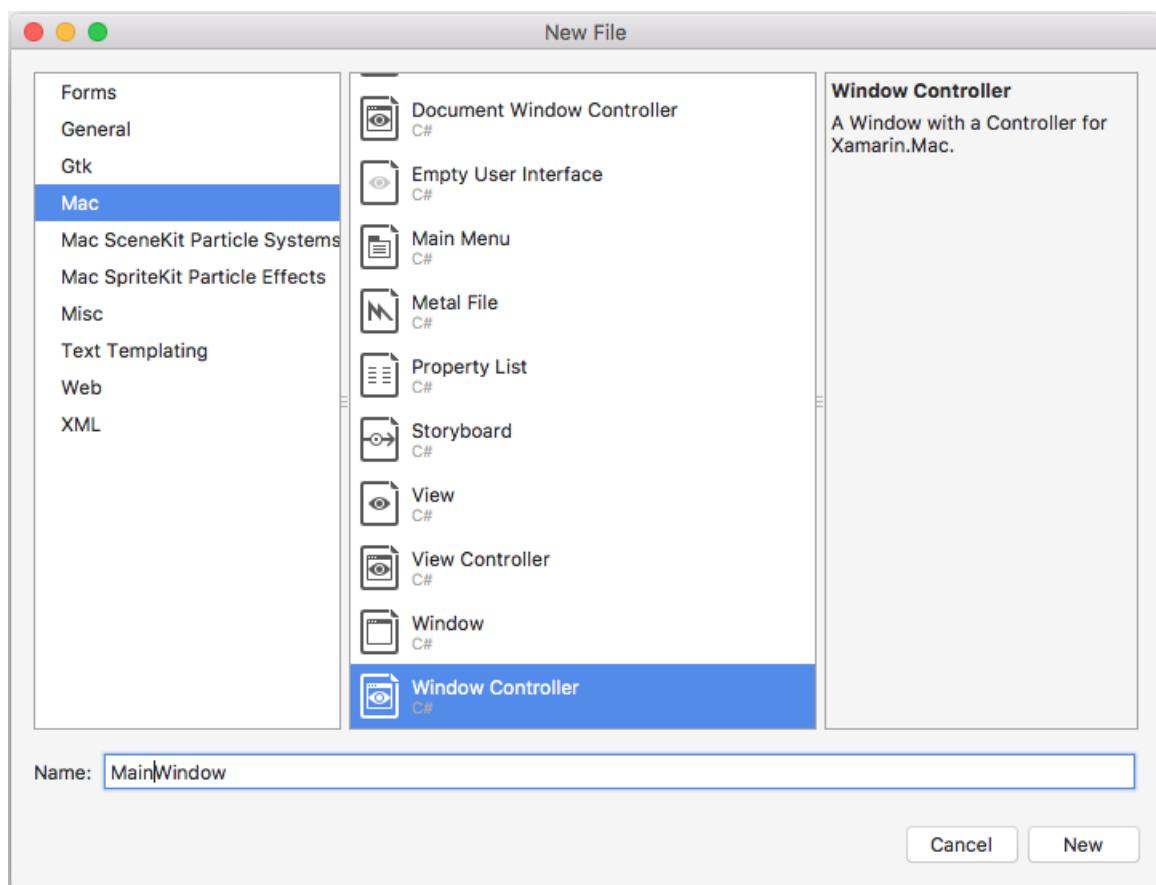
The preferred method for creating a Xamarin.Mac app's User Interface is with Storyboards (please see our [Introduction to Storyboards](#) documentation for more information) and, as a result, any new project started in Xamarin.Mac will use Storyboards by default.

To switch to using a .xib based UI, do the following:

1. Open Visual Studio for Mac and start a new Xamarin.Mac project.
2. In the **Solution Pad**, right-click on the project and select **Add > New File...**
3. Select **Mac > Windows Controller**:

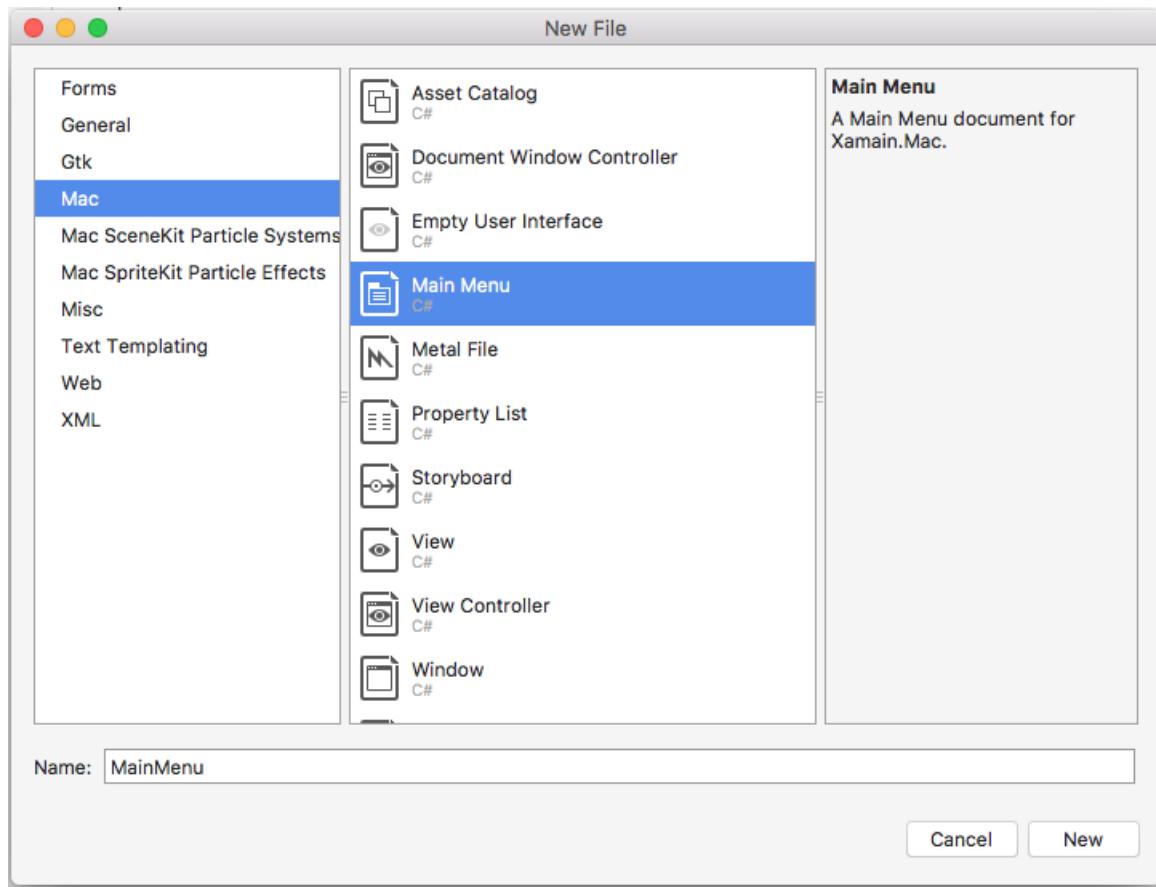


4. Enter `MainWindow` for the name and click the **New** button:

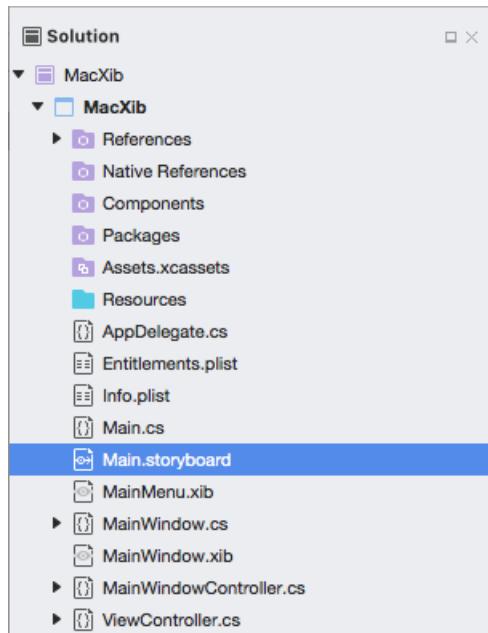


5. Right-click on the project again and select Add > New File...

6. Select Mac > Main Menu:



7. Leave the name as `MainMenu` and click the **New** button.
8. In the **Solution Pad** select the `Main.storyboard` file, right-click and select **Remove**:

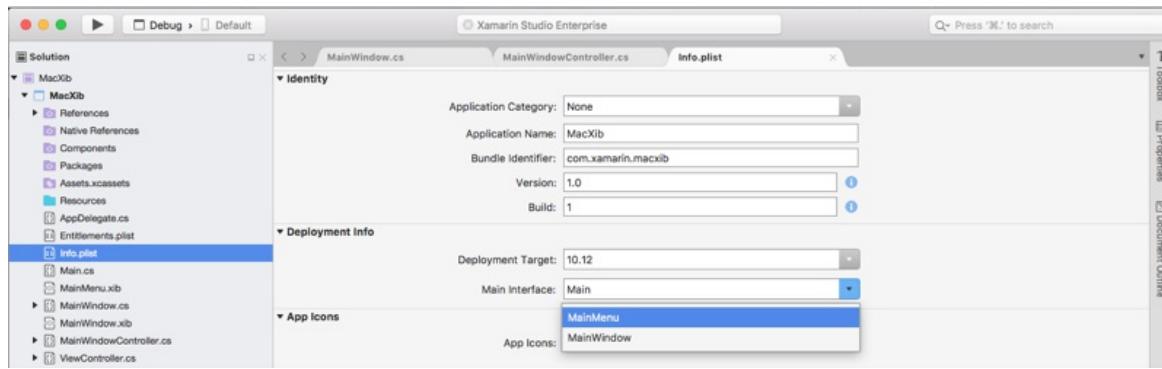


9. In the Remove Dialog Box, click the **Delete** button:



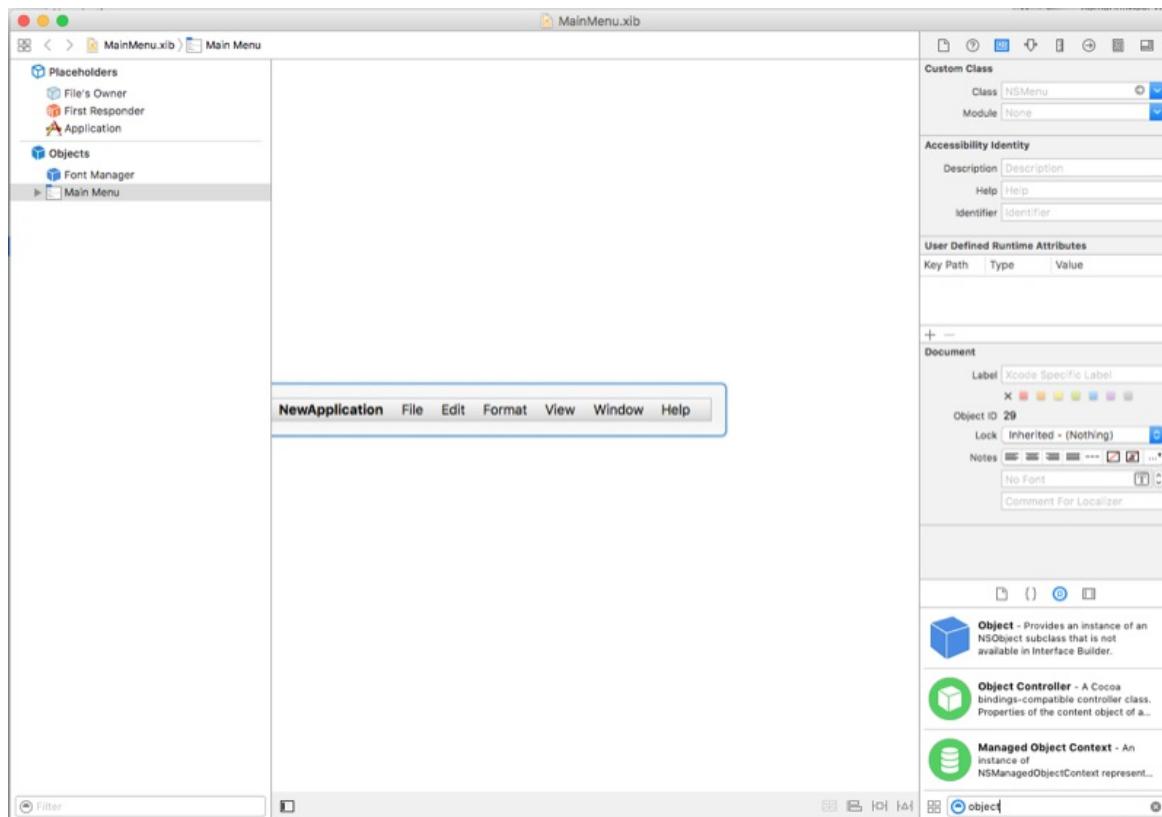
10. In the **Solution Pad**, double-click the `Info.plist` file to open it for editing.

11. Select `MainMenu` from the **Main Interface** dropdown:

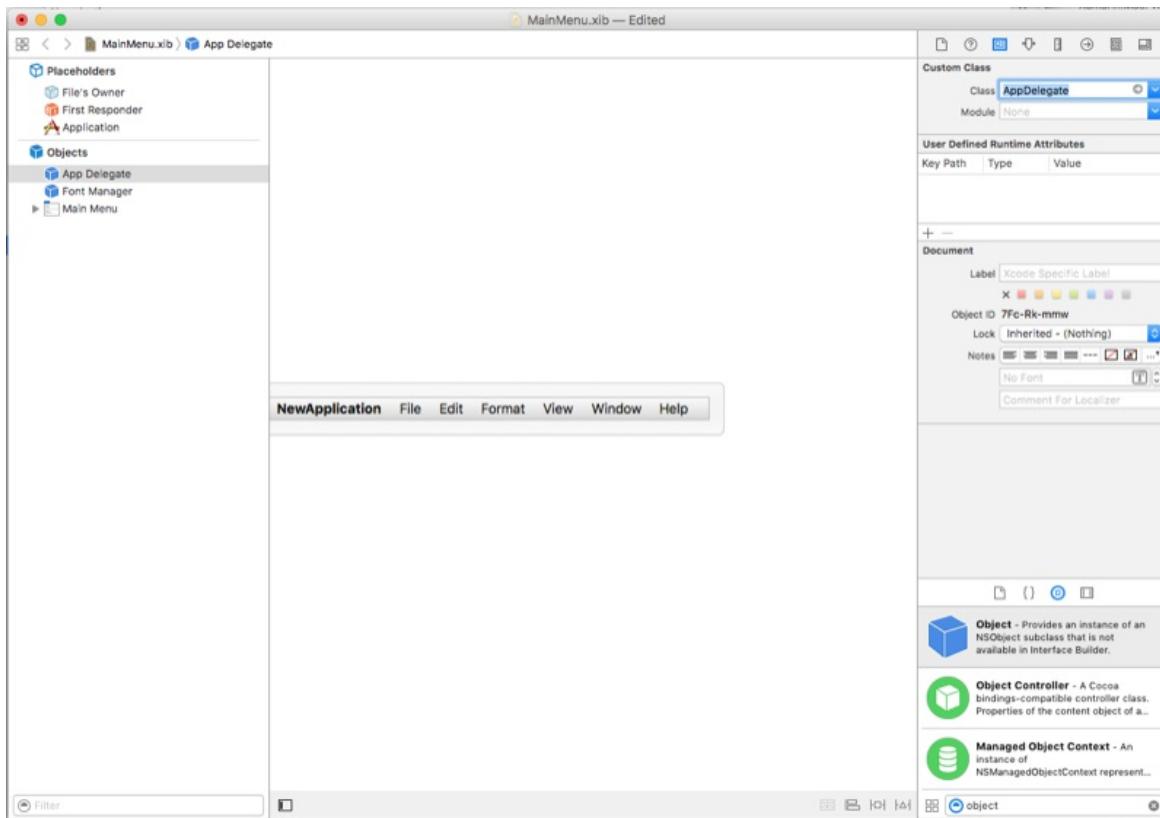


12. In the **Solution Pad**, double-click the `MainMenu.xib` file to open it for editing in Xcode's Interface Builder.

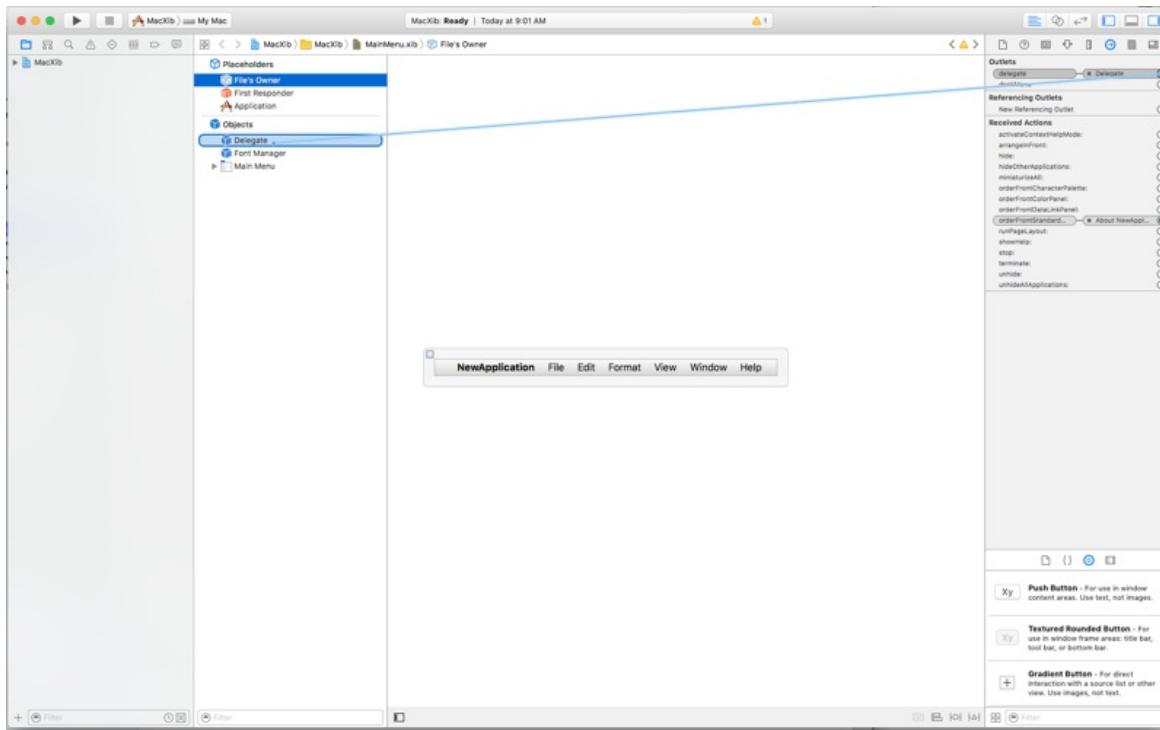
13. In the **Library Inspector**, type `object` in the search field then drag a new **Object** onto the design surface:



14. In the **Identity Inspector**, enter `AppDelegate` for the **Class**:



15. Select **File's Owner** from the **Interface Hierarchy**, switch to the **Connection Inspector** and drag a line from the delegate to the **AppDelegate** Object just added to the project:



16. Save the changes and return to Visual Studio for Mac.

With all these changes in place, edit the **AppDelegate.cs** file and make it look like the following:

```

using AppKit;
using Foundation;

namespace MacXib
{
    [Register ("AppDelegate")]
    public class AppDelegate : NSApplicationDelegate
    {
        public MainWindowController mainWindowController { get; set; }

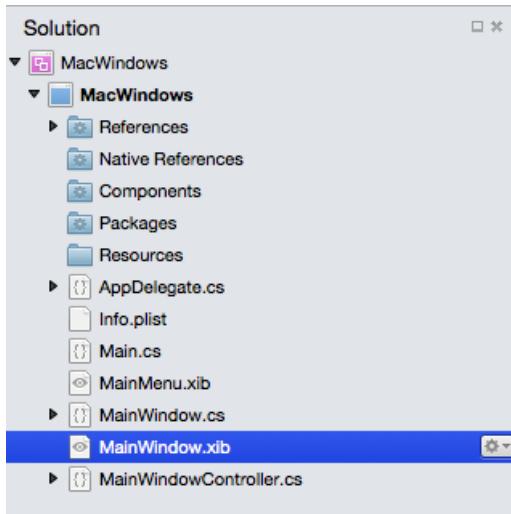
        public AppDelegate ()
        {
        }

        public override void DidFinishLaunching (NSNotification notification)
        {
            // Insert code here to initialize your application
            mainWindowController = new MainWindowController ();
            mainWindowController.Window.MakeKeyAndOrderFront (this);
        }

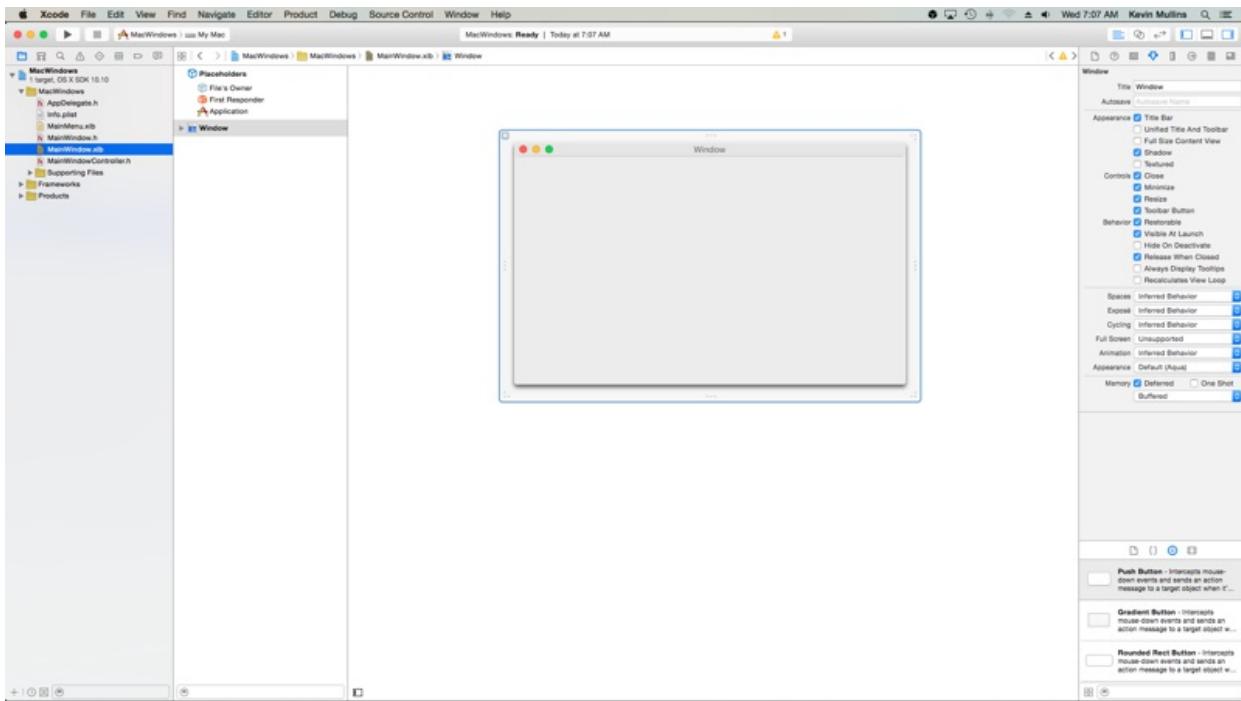
        public override void WillTerminate (NSNotification notification)
        {
            // Insert code here to tear down your application
        }
    }
}

```

Now the app's Main Window is defined in a **.xib** file automatically included in the project when adding a Window Controller. To edit your windows design, in the **Solution Pad**, double click the **MainWindow.xib** file:



This will open the window design in Xcode's Interface Builder:



Standard window workflow

For any window that you create and work with in your Xamarin.Mac application, the process is basically the same:

1. For new windows that are not the default added automatically to your project, add a new window definition to the project.
2. Double-click the .xib file to open the window design for editing in Xcode's Interface Builder.
3. Set any required window properties in the **Attribute Inspector** and the **Size Inspector**.
4. Drag in the controls required to build your interface and configure them in the **Attribute Inspector**.
5. Use the **Size Inspector** to handle the resizing for your UI elements.
6. Expose the window's UI elements to C# code via outlets and actions.
7. Save your changes and switch back to Visual Studio for Mac to sync with Xcode.

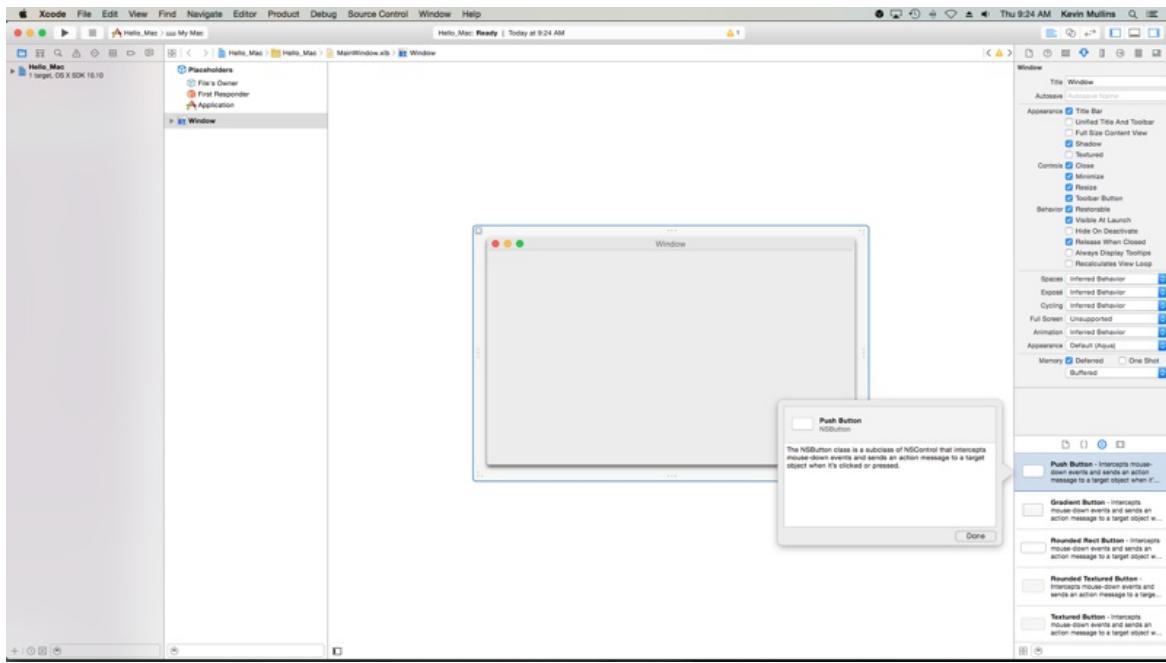
Designing a window layout

The process for laying out a User Interface in Interface builder is basically the same for every element that you add:

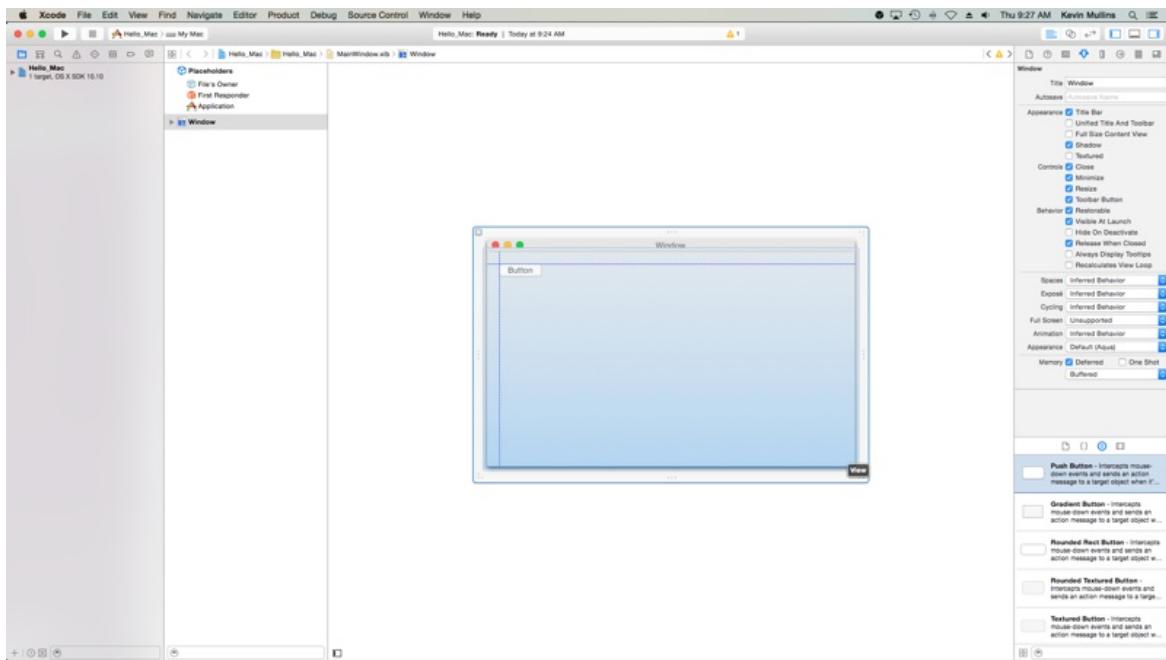
1. Find the desired control in the **Library Inspector** and drag it into the **Interface Editor** and position it.
2. Set any required window properties in the **Attribute Inspector**.
3. Use the **Size Inspector** to handle the resizing for your UI elements.
4. If you are using a custom class, set it in the **Identity Inspector**.
5. Expose the UI elements to C# code via outlets and actions.
6. Save your changes and switch back to Visual Studio for Mac to sync with Xcode.

For Example:

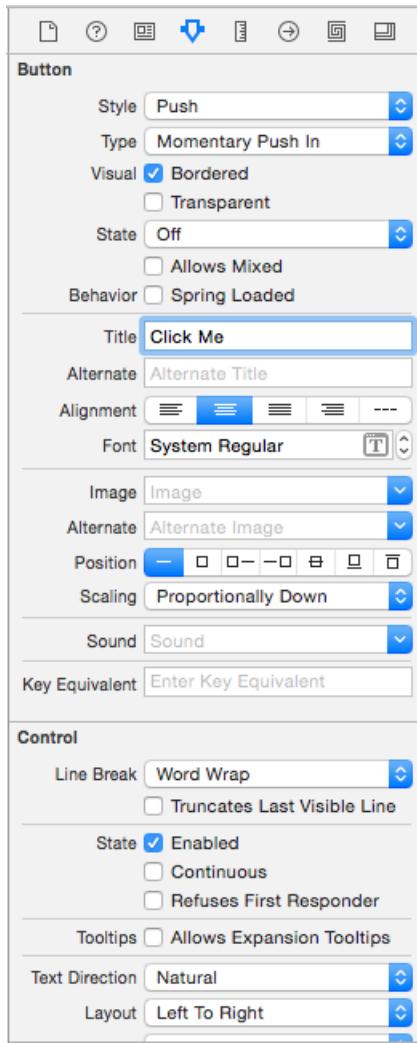
1. In Xcode, drag a **Push Button** from the **Library Section**:



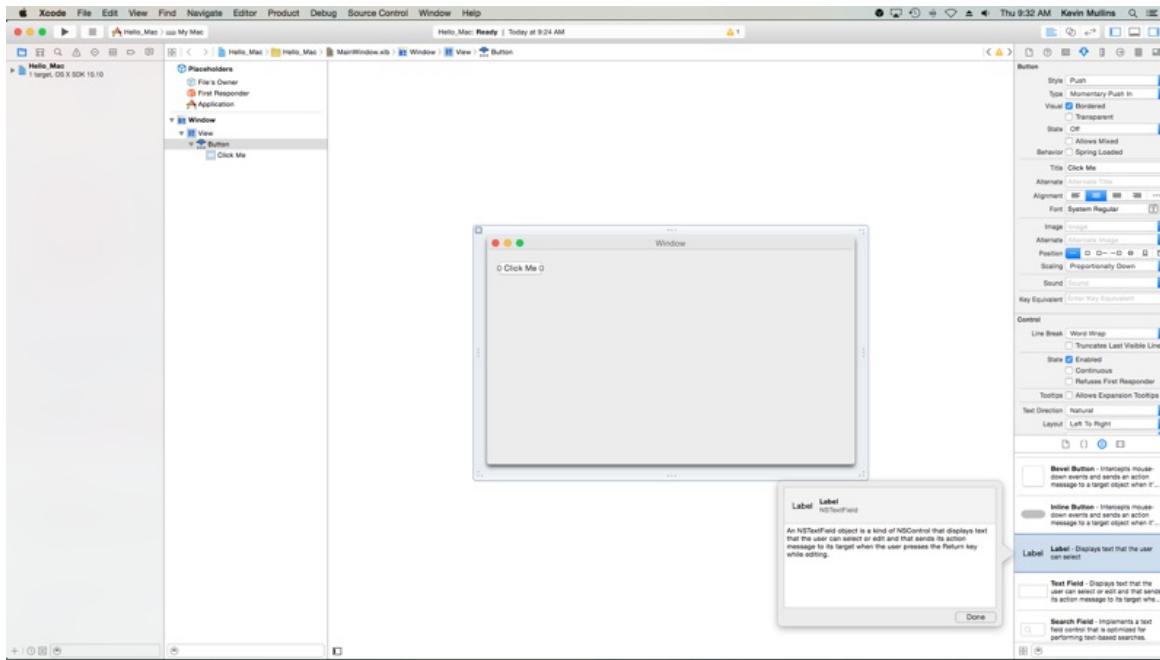
2. Drop the button onto the **Window** in the **Interface Editor**:



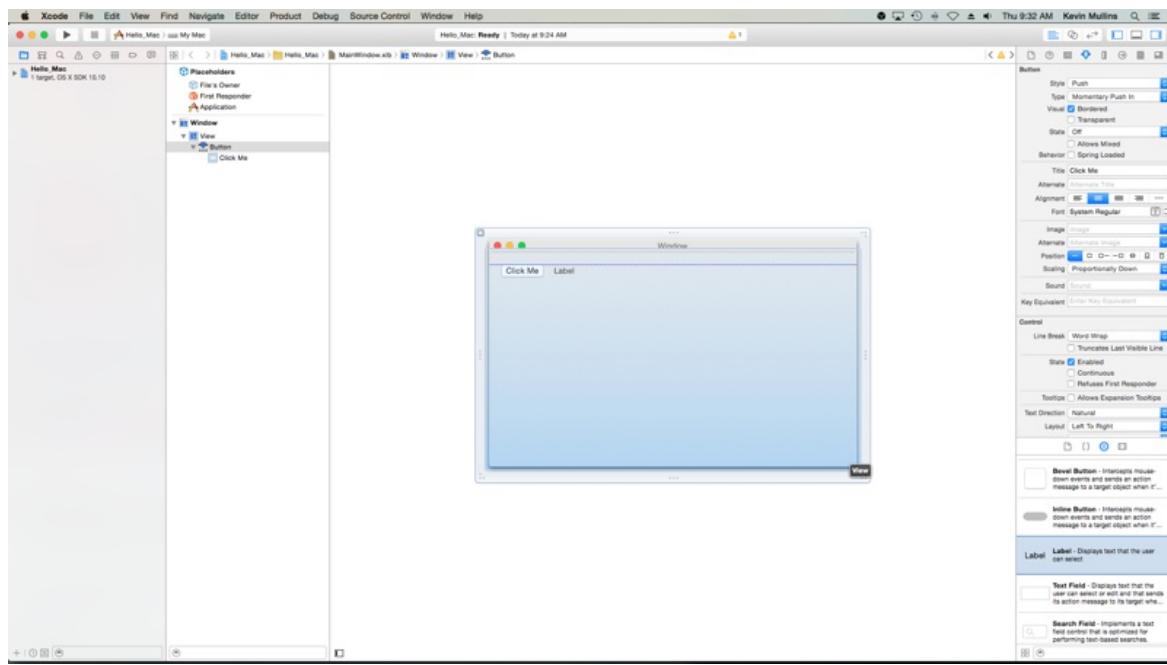
3. Click on the **Title** property in the **Attribute Inspector** and change the button's title to **Click Me**:



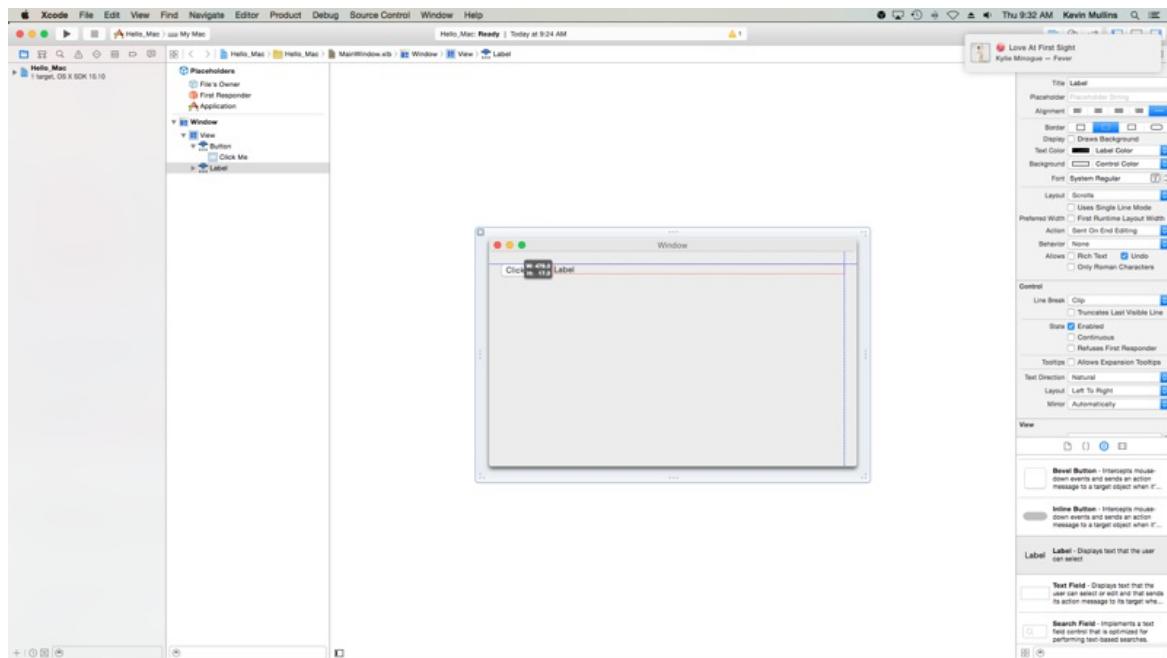
4. Drag a Label from the Library Section:



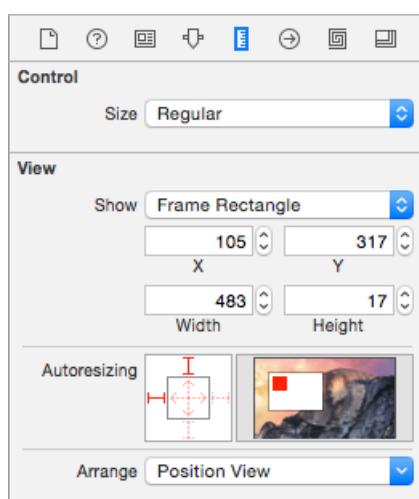
5. Drop the label onto the Window beside the button in the Interface Editor:



6. Grab the right handle on the label and drag it until it is near the edge of the window:

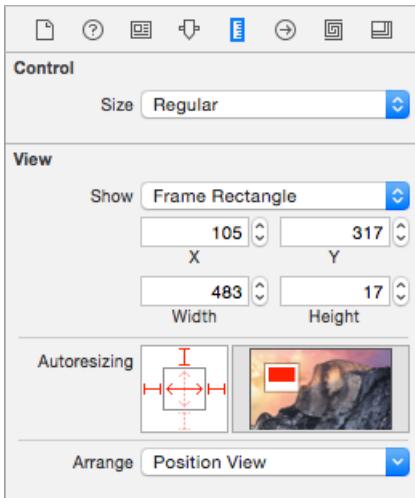


7. With the label still selected in the Interface Editor, switch to the Size Inspector:



8. In the Autosizing Box click the Dim Red Bracket at the right and the Dim Red Horizontal Arrow in

the center:

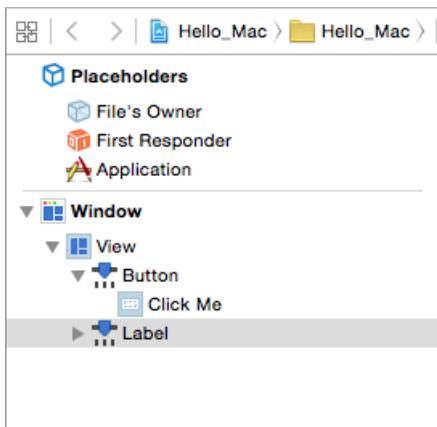


9. This ensures that the label will stretch to grow and shrink as the window is resized in the running application. The **Red Brackets** and the top and left of the **Autosizing Box** tell the label to be stuck to its given X and Y locations.

10. Save your changes to the User Interface

As you were resizing and moving controls around, you should have noticed that Interface Builder gives you helpful snap hints that are based on [OS X Human Interface Guidelines](#). These guidelines will help you create high quality applications that will have a familiar look and feel for Mac users.

If you look in the **Interface Hierarchy** section, notice how the layout and hierarchy of the elements that make up our user interface are shown:



From here you can select items to edit or drag to reorder UI elements if needed. For example, if a UI element was being covered by another element, you could drag it to the bottom of the list to make it the top-most item on the window.

For more information on working with Windows in a Xamarin.Mac application, please see our [Windows](#) documentation.

Exposing UI elements to C# code

Once you have finished laying out the look and feel of your user interface in Interface Builder, you'll need to expose elements of the UI so that they can be accessed from C# code. To do this, you'll be using actions and outlets.

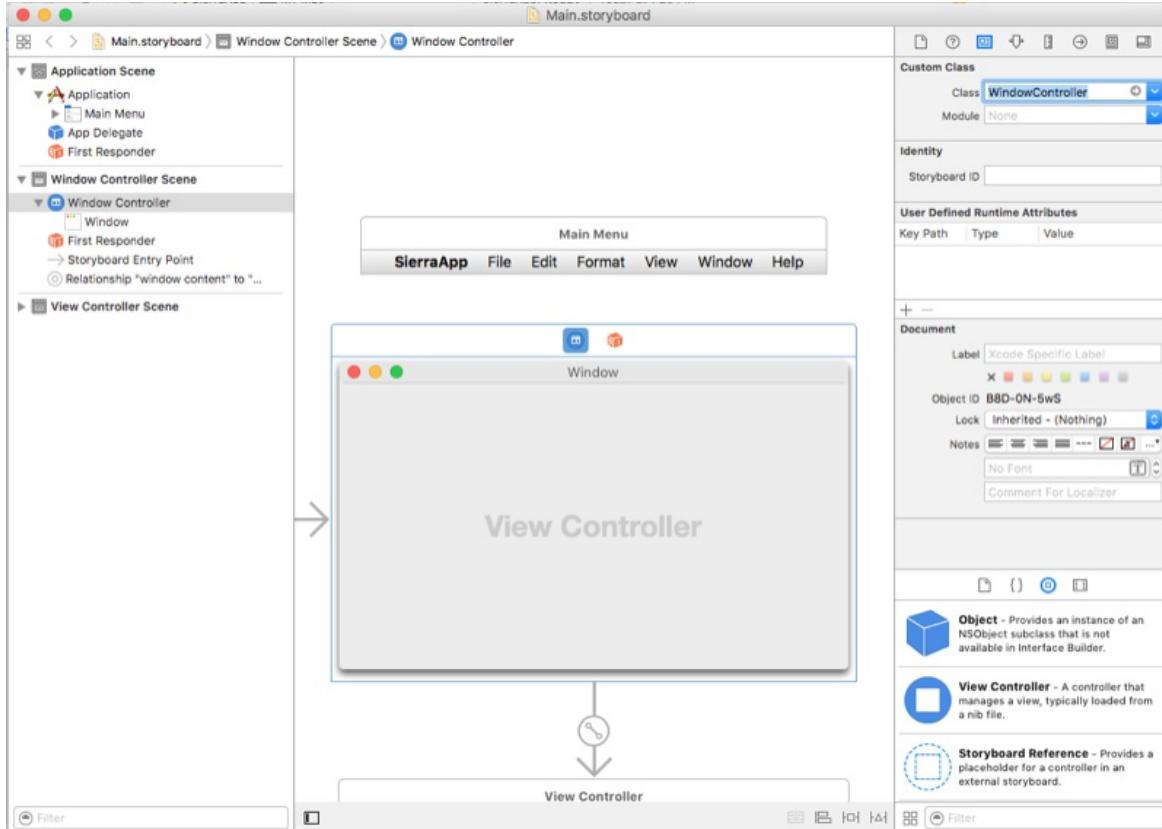
Setting a custom main window controller

To be able to create Outlets and Actions to expose UI elements to C# code, the Xamarin.Mac app will need to be

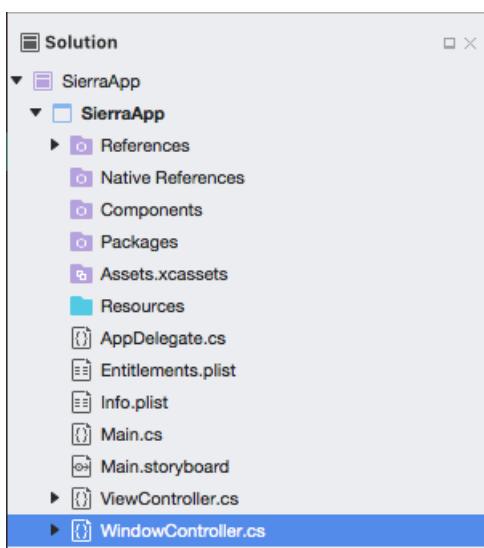
using a Custom Window Controller.

Do the following:

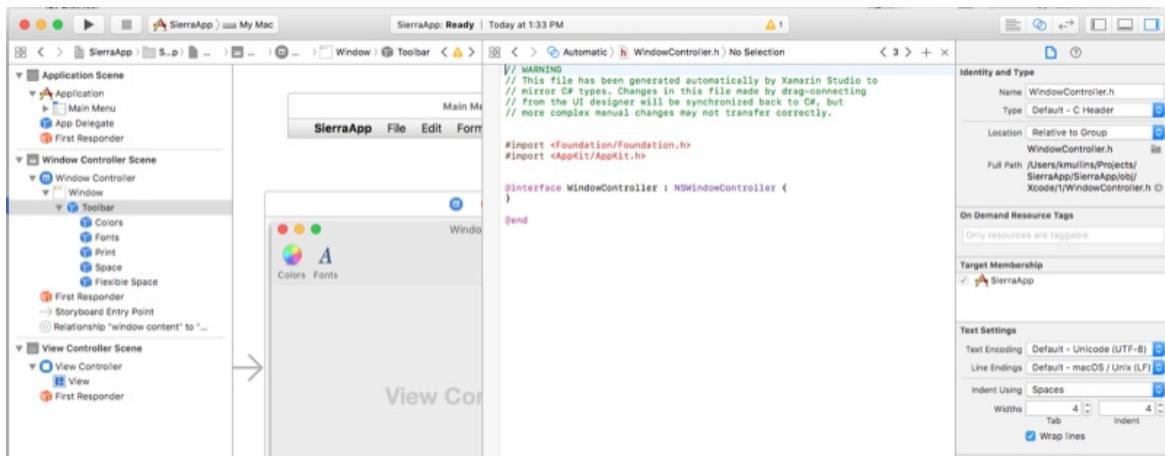
1. Open the app's Storyboard in Xcode's Interface Builder.
2. Select the `NSWindowController` in the Design Surface.
3. Switch to the **Identity Inspector** view and enter `WindowController` as the **Class Name**:



4. Save your changes and return to Visual Studio for Mac to sync.
5. A `WindowController.cs` file will be added to your project in the **Solution Pad** in Visual Studio for Mac:



6. Reopen the Storyboard in Xcode's Interface Builder.
7. The `WindowController.h` file will be available for use:



Outlets and actions

So what are outlets and actions? In traditional .NET User Interface programming, a control in the User Interface is automatically exposed as a property when it's added. Things work differently in Mac, simply adding a control to a view doesn't make it accessible to code. The developer must explicitly expose the UI element to code. In order to do this, Apple gives us two options:

- **Outlets** – Outlets are analogous to properties. If you wire up a control to an Outlet, it's exposed to your code via a property, so you can do things like attach event handlers, call methods on it, etc.
- **Actions** – Actions are analogous to the command pattern in WPF. For example, when an Action is performed on a control, say a button click, the control will automatically call a method in your code. Actions are powerful and convenient because you can wire up many controls to the same Action.

In Xcode, outlets and actions are added directly in code via *Control-dragging*. More specifically, this means that to create an outlet or action, you choose which control element you'd like to add an outlet or action, hold down the **Control** button on the keyboard, and drag that control directly into your code.

For Xamarin.Mac developers, this means that you drag into the Objective-C stub files that correspond to the C# file where you want to create the outlet or action. Visual Studio for Mac created a file called **MainWindow.h** as part of the shim Xcode project it generated to use the Interface Builder:



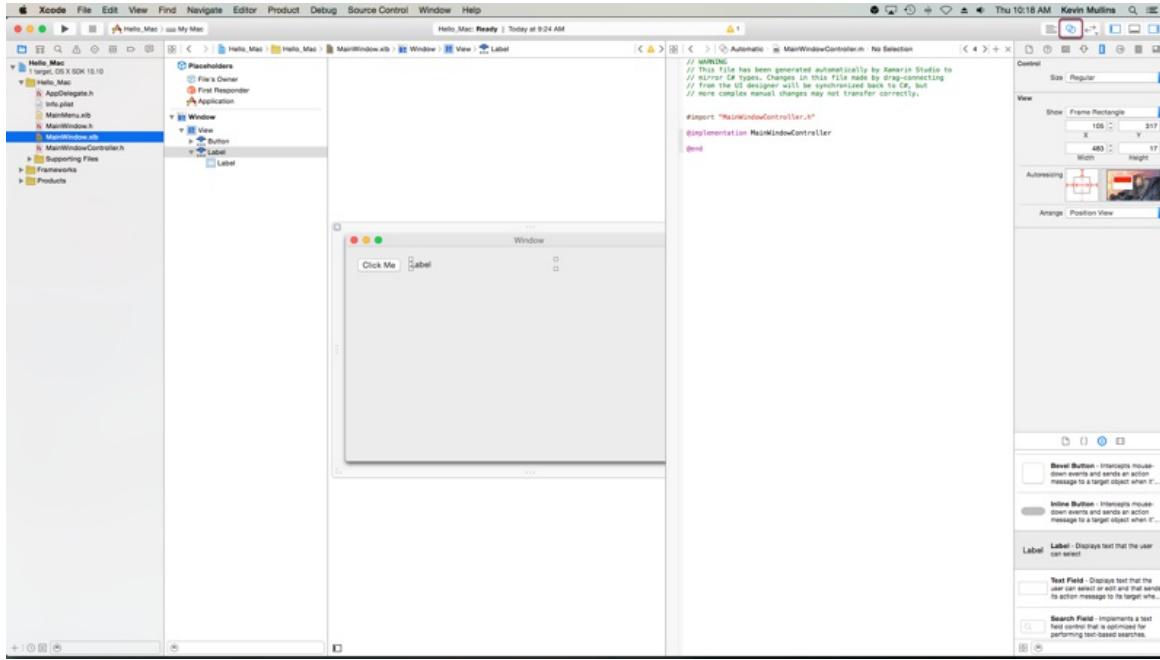
This stub .h file mirrors the **MainWindow.designer.cs** that is automatically added to a Xamarin.Mac project when a new **NSWindow** is created. This file will be used to synchronize the changes made by Interface Builder and is where we will create your outlets and actions so that UI elements are exposed to C# code.

Adding an outlet

With a basic understanding of what outlets and actions are, let's look at creating an outlet to expose a UI element to your C# code.

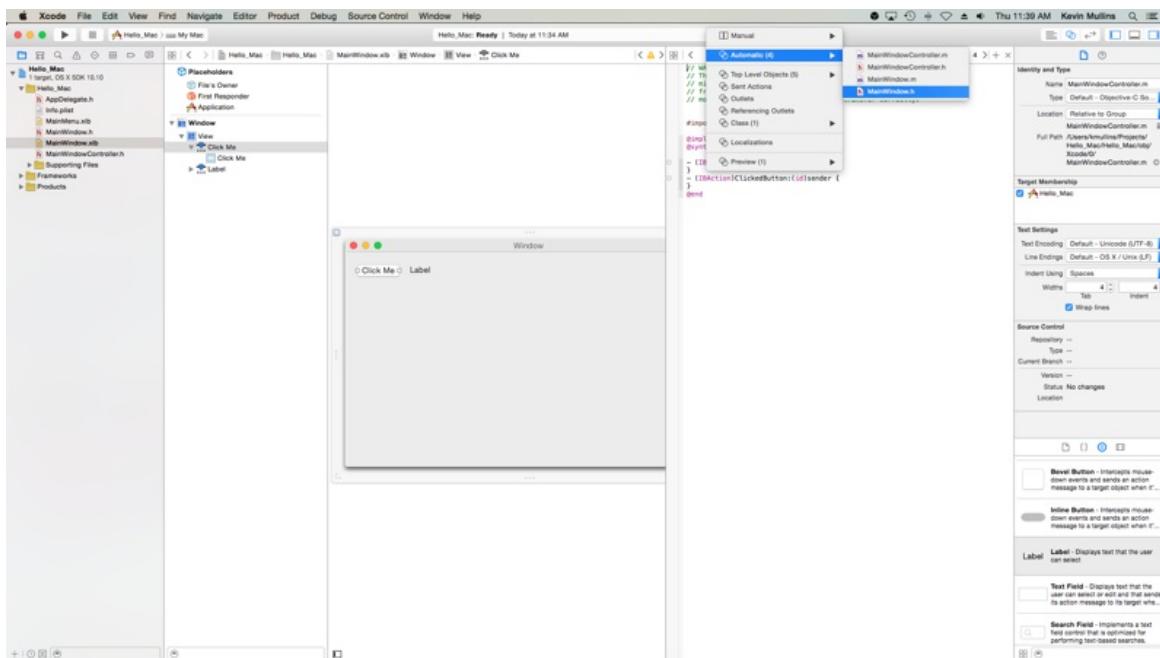
Do the following:

1. In Xcode at the far right top-hand corner of the screen, click the **Double Circle** button to open the **Assistant Editor**:

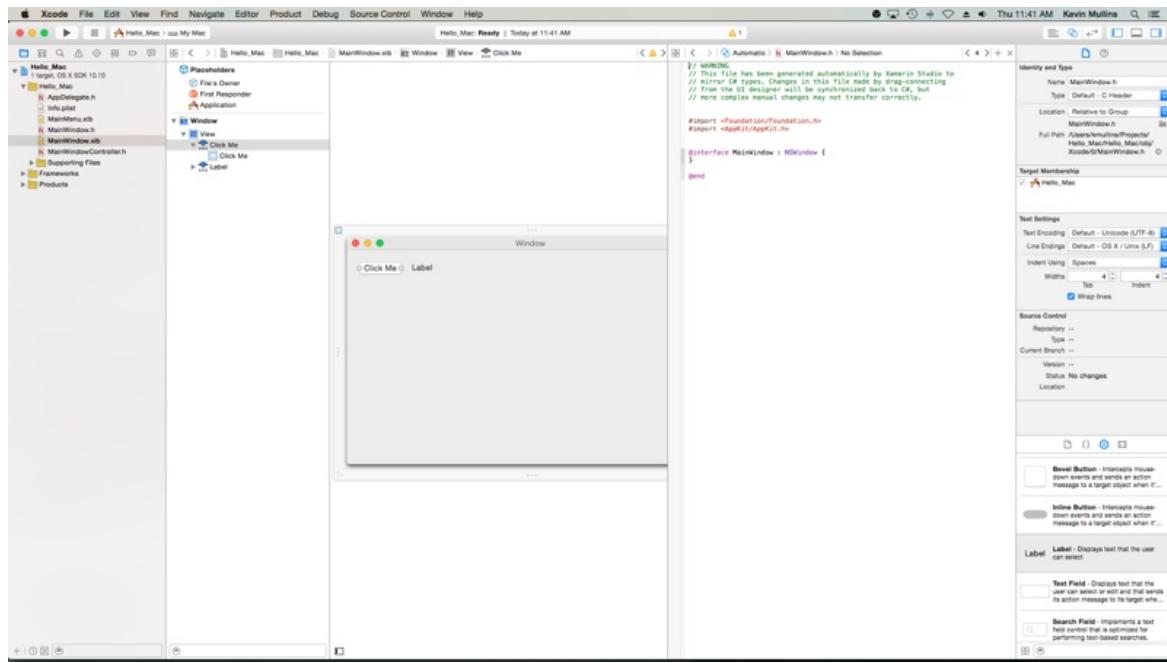


2. The Xcode will switch to a split-view mode with the **Interface Editor** on one side and a **Code Editor** on the other.
3. Notice that Xcode has automatically picked the **MainWindowController.m** file in the **Code Editor**, which is incorrect. If you remember from our discussion on what outlets and actions are above, we need to have the **MainWindow.h** selected.

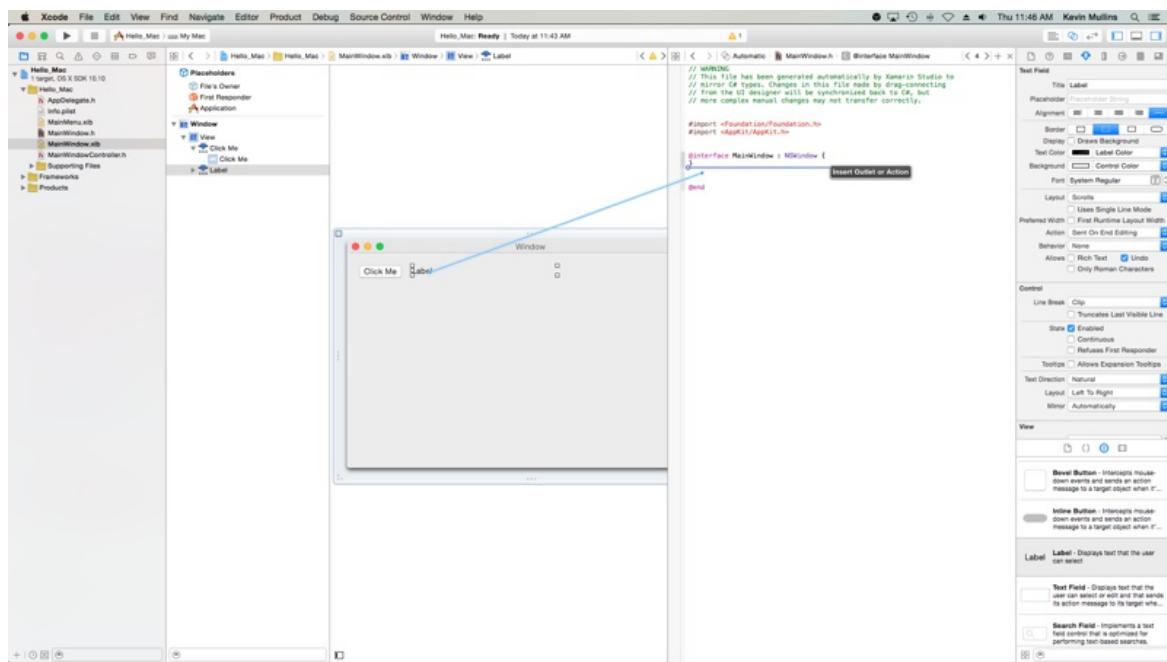
4. At the top of the **Code Editor** click on the **Automatic Link** and select the **MainWindow.h** file:



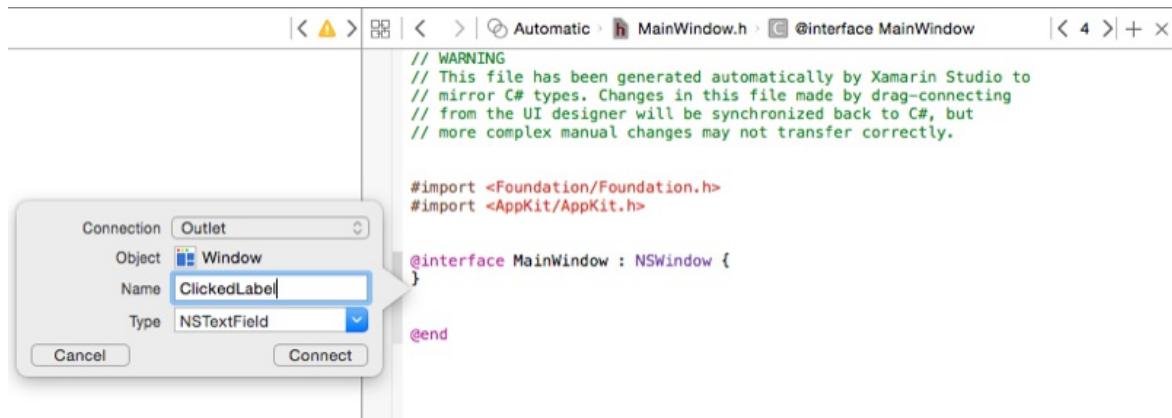
5. Xcode should now have the correct file selected:



6. **The last step was very important!** If you don't have the correct file selected, you won't be able to create outlets and actions or they will be exposed to the wrong class in C#!
7. In the **Interface Editor**, hold down the **Control** key on the keyboard and click-drag the label we created above onto the code editor just below the `@interface MainWindow : NSWindow { }` code:



8. A dialog box will be displayed. Leave the **Connection** set to **outlet** and enter `clickedLabel` for the **Name**:



9. Click the **Connect** button to create the outlet:

```
// WARNING
// This file has been generated automatically by Xamarin Studio to
// mirror C# types. Changes in this file made by drag-connecting
// from the UI designer will be synchronized back to C#, but
// more complex manual changes may not transfer correctly.

#import <Foundation/Foundation.h>
#import <AppKit/AppKit.h>

@interface MainWindow : NSWindow {
    NSTextField *ClickedLabel;
}
@property (assign) IBOutlet NSTextField *ClickedLabel;

@end
```

10. Save the changes to the file.

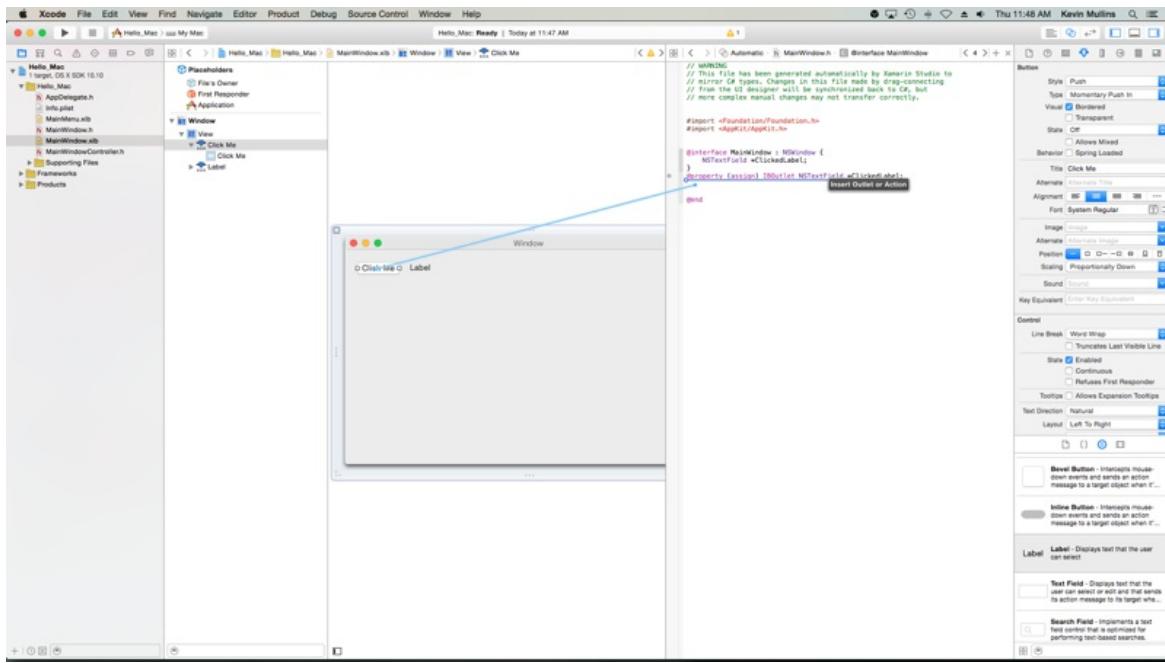
Adding an action

Next, let's look at creating an action to expose a user interaction with UI element to your C# code.

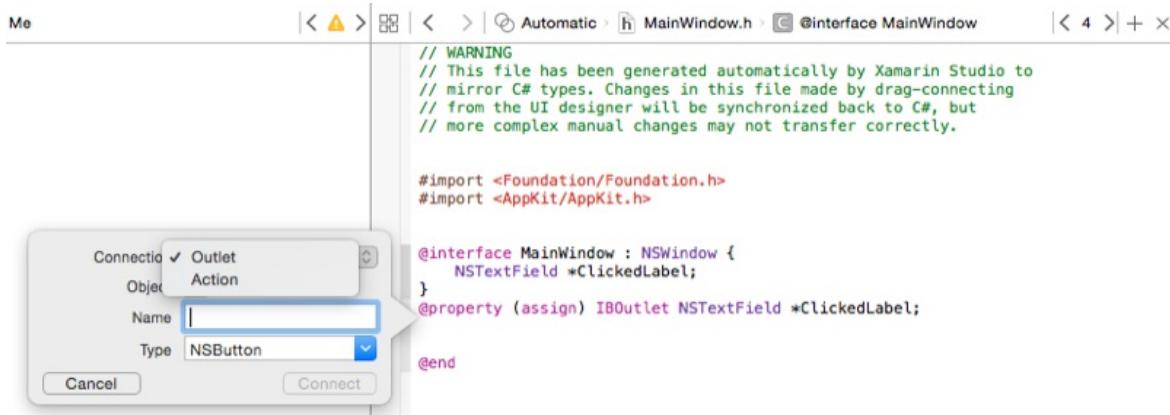
Do the following:

1. Make sure we are still in the **Assistant Editor** and the **MainWindow.h** file is visible in the **Code Editor**.
2. In the **Interface Editor**, hold down the **Control** key on the keyboard and click-drag the button we created above onto the code editor just below the

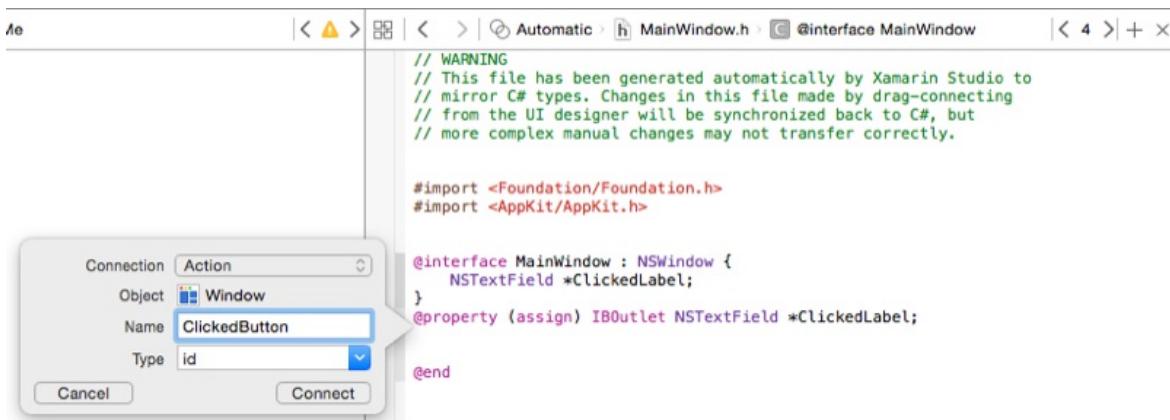
`@property (assign) IBOutlet NSTextField *ClickedLabel;`



3. Change the Connection type to action:



4. Enter `ClickedButton` as the Name:



5. Click the Connect button to create action:

```

// WARNING
// This file has been generated automatically by Xamarin Studio to
// mirror C# types. Changes in this file made by drag-connecting
// from the UI designer will be synchronized back to C#, but
// more complex manual changes may not transfer correctly.

#import <Foundation/Foundation.h>
#import <AppKit/AppKit.h>

@interface MainWindow : NSWindow {
    NSTextField *ClickedLabel;
}
@property (assign) IBOutlet NSTextField *ClickedLabel;
- (IBAction)ClickedButton:(id)sender;

@end

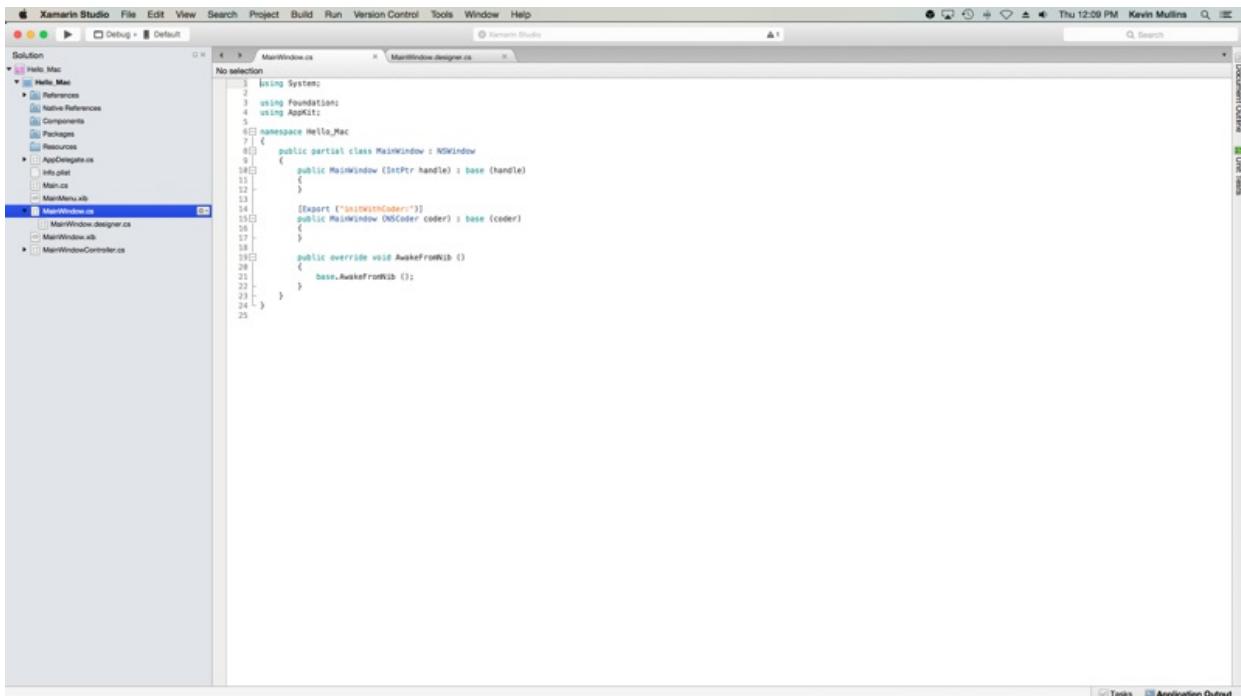
```

6. Save the changes to the file.

With your User Interface wired-up and exposed to C# code, switch back to Visual Studio for Mac and let it synchronize the changes from Xcode and Interface Builder.

Writing the code

With your User Interface created and its UI elements exposed to code via outlets and actions, you are ready to write the code to bring your program to life. For example, open the `MainWindow.cs` file for editing by double-clicking it in the Solution Pad:



And add the following code to the `MainWindow` class to work with the sample outlet that you created above:

```

private int numberoftimesClicked = 0;
...
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

    // Set the initial value for the label
    ClickedLabel.StringValue = "Button has not been clicked yet.";
}

```

Note that the `NSLabel` is accessed in C# by the direct name that you assigned it in Xcode when you created its outlet in Xcode, in this case, it's called `ClickedLabel`. You can access any method or property of the exposed object the same way you would any normal C# class.

IMPORTANT

You need to use `AwakeFromNib`, instead of another method such as `Initialize`, because `AwakeFromNib` is called *after* the OS has loaded and instantiated the User Interface from the .xib file. If you tried to access the label control before the .xib file has been fully loaded and instantiated, you'd get a `NullReferenceException` error because the label control would not be created yet.

Next, add the following partial class to the `MainWindow` class:

```
partial void ClickedButton (Foundation.NSObject sender) {

    // Update counter and label
    ClickedLabel.StringValue = string.Format("The button has been clicked {0}
time{1}.", ++numberOfTimesClicked, (numberOfTimesClicked < 2) ? "" : "s");
}
```

This code attaches to the action that you created in Xcode and Interface Builder and will be called any time the user clicks the button.

Some UI elements automatically have built in actions, for example, items in the default Menu Bar such as the **Open...** menu item (`openDocument:`). In the **Solution Pad**, double-click the `AppDelegate.cs` file to open it for editing and add the following code below the `DidFinishLaunching` method:

```
[Export ("openDocument")]
void OpenDialog (NSObject sender)
{
    var dlg = NSOpenPanel.OpenPanel;
    dlg.CanChooseFiles = false;
    dlg.CanChooseDirectories = true;

    if (dlg.RunModal () == 1) {
        var alert = new NSAlert ();
        AlertStyle = NSAlertStyle.Informational,
        InformativeText = "At this point we should do something with the folder that the user just
selected in the Open File Dialog box...",
        MessageText = "Folder Selected"
    };
    alert.RunModal ();
}
}
```

The key line here is `[Export ("openDocument")]`, it tells `NSMenu` that the `AppDelegate` has a method `void OpenDialog (NSObject sender)` that responds to the `openDocument:` action.

For more information on working with Menus, please see our [Menus](#) documentation.

Synchronizing changes with Xcode

When you switch back to Visual Studio for Mac from Xcode, any changes that you have made in Xcode will automatically be synchronized with your Xamarin.Mac project.

If you select the `MainWindow.designer.cs` in the **Solution Pad** you'll be able to see how our outlet and action have been wired up in our C# code:



Notice how the two definitions in the **MainWindow.designer.cs** file:

```
[Outlet]
AppKit.NSTextField ClickedLabel { get; set; }

[Action ("ClickedButton:")]
partial void ClickedButton (Foundation.NSObject sender);
```

Line up with the definitions in the **MainWindow.h** file in Xcode:

```
@property (assign) IBOutlet NSTextField *ClickedLabel;
- (IBAction)ClickedButton:(id)sender;
```

As you can see, Visual Studio for Mac listens for changes to the .h file, and then automatically synchronizes those changes in the respective .designer.cs file to expose them to your application. You may also notice that **MainWindow.designer.cs** is a partial class, so that Visual Studio for Mac doesn't have to modify **MainWindow.cs** which would overwrite any changes that we have made to the class.

You normally will never need to open the **MainWindow.designer.cs** yourself, it was presented here for educational purposes only.

IMPORTANT

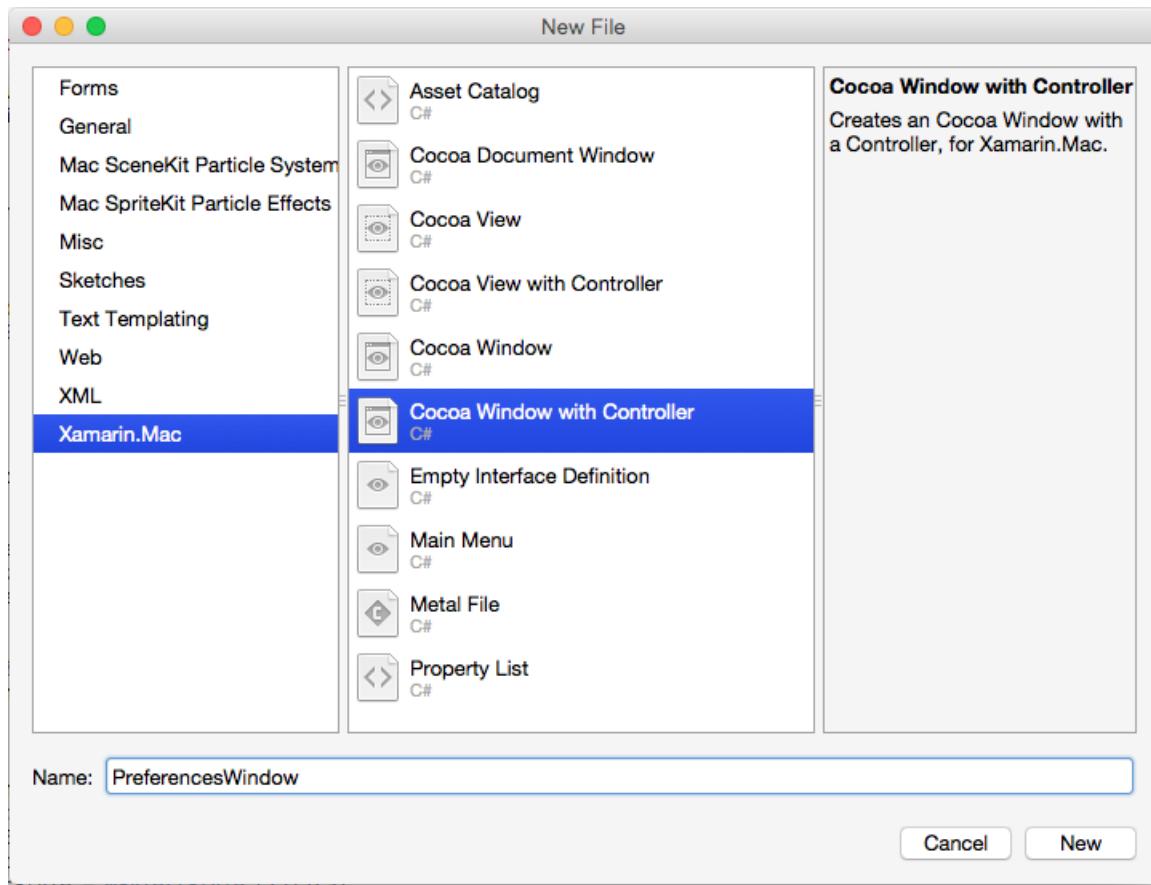
In most situations, Visual Studio for Mac will automatically see any changes made in Xcode and sync them to your Xamarin.Mac project. In the off occurrence that synchronization doesn't automatically happen, switch back to Xcode and them back to Visual Studio for Mac again. This will normally kick off a synchronization cycle.

Adding a new window to a project

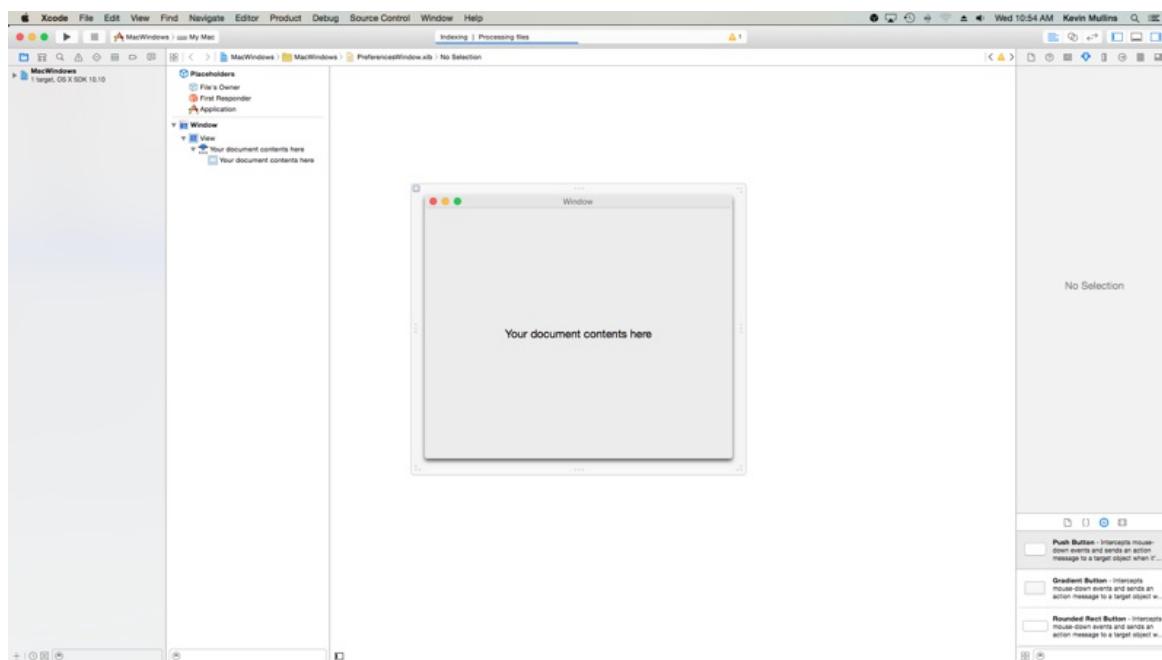
Aside from the main document window, a Xamarin.Mac application might need to display other types of windows to the user, such as Preferences or Inspector Panels. When adding a new Window to your project you should always use the **Cocoa Window with Controller** option, as this makes the process of loading the Window from the .xib file easier.

To add a new window, do the following:

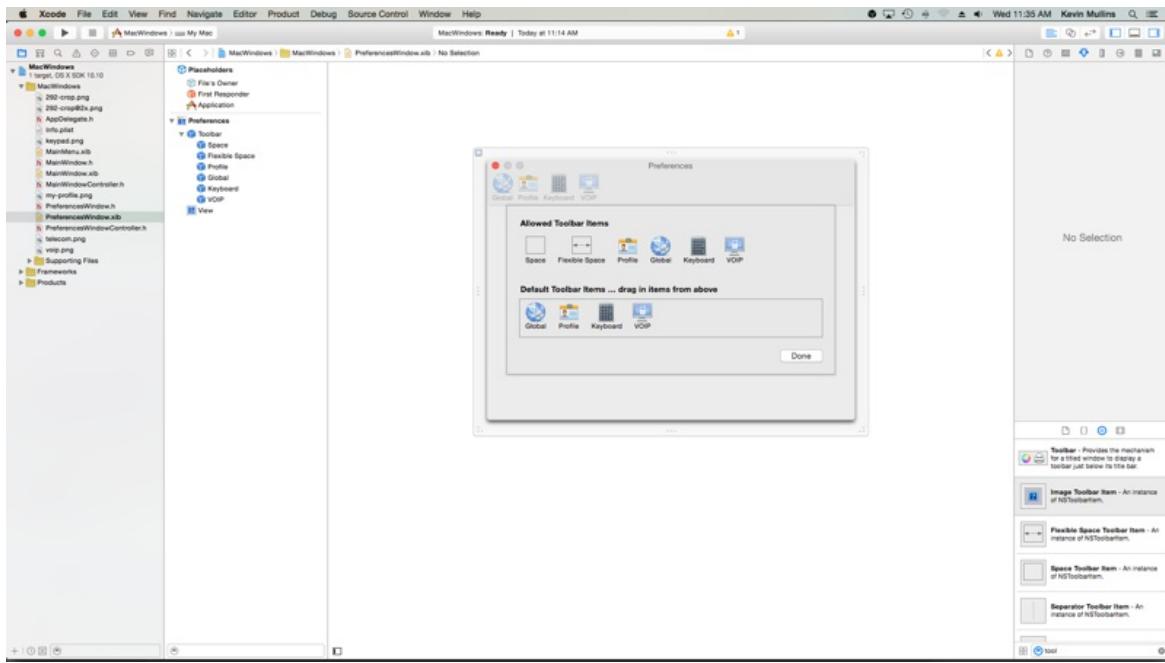
1. In the Solution Pad, right-click on the project and select Add > New File...
2. In the New File dialog box, select Xamarin.Mac > Cocoa Window with Controller:



3. Enter **PreferencesWindow** for the Name and click the New button.
4. Double-click the **PreferencesWindow.xib** file to open it for editing in Interface Builder:



5. Design your interface:



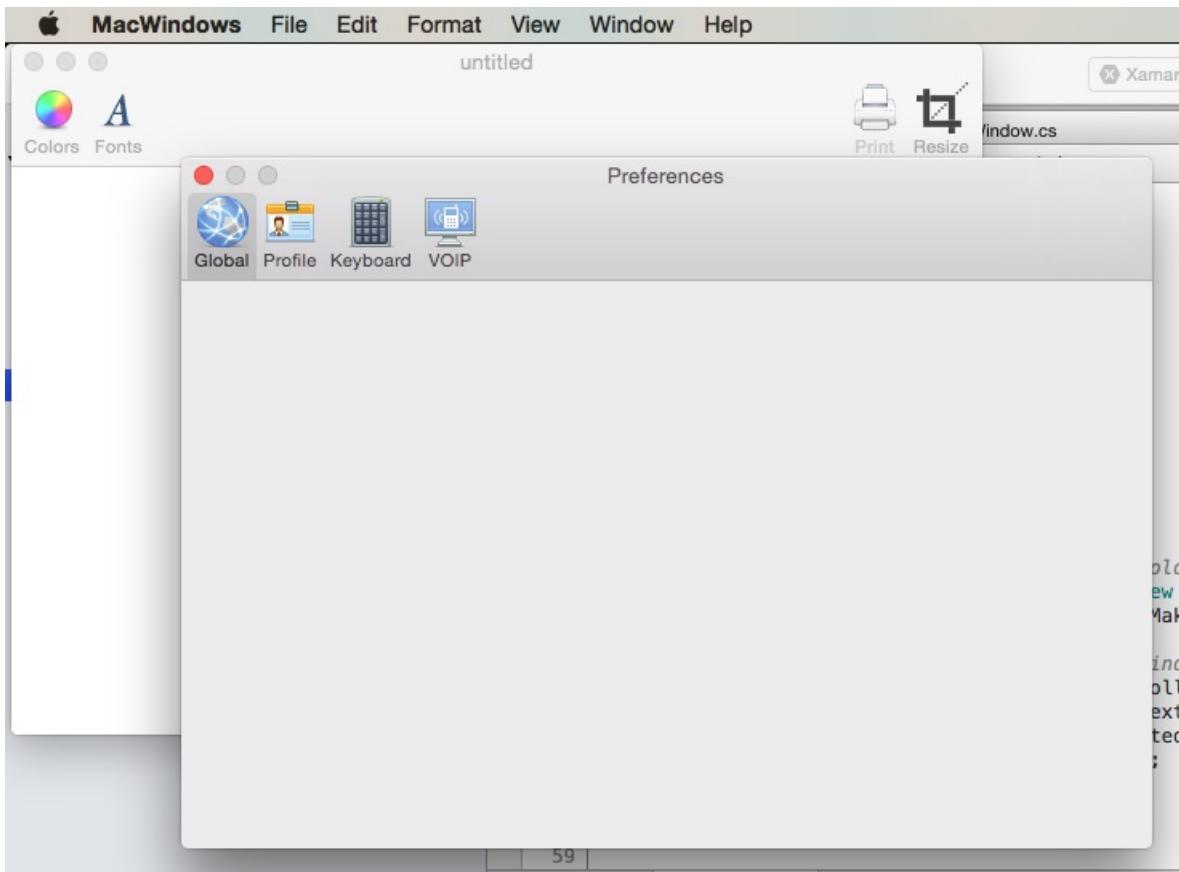
6. Save your changes and return to Visual Studio for Mac to sync with Xcode.

Add the following code to **AppDelegate.cs** to display your new window:

```
[Export("applicationPreferences")]
void ShowPreferences (NSObject sender)
{
    var preferences = new PreferencesWindowController ();
    preferences.Window.MakeKeyAndOrderFront (this);
}
```

The `var preferences = new PreferencesWindowController ();` line creates a new instance of the Window Controller that loads the Window from the .xib file and inflates it. The `preferences.Window.MakeKeyAndOrderFront (this);` line displays the new Window to the user.

If you run the code and select the **Preferences...** from the **Application Menu**, the window will be displayed:



For more information on working with Windows in a Xamarin.Mac application, please see our [Windows](#) documentation.

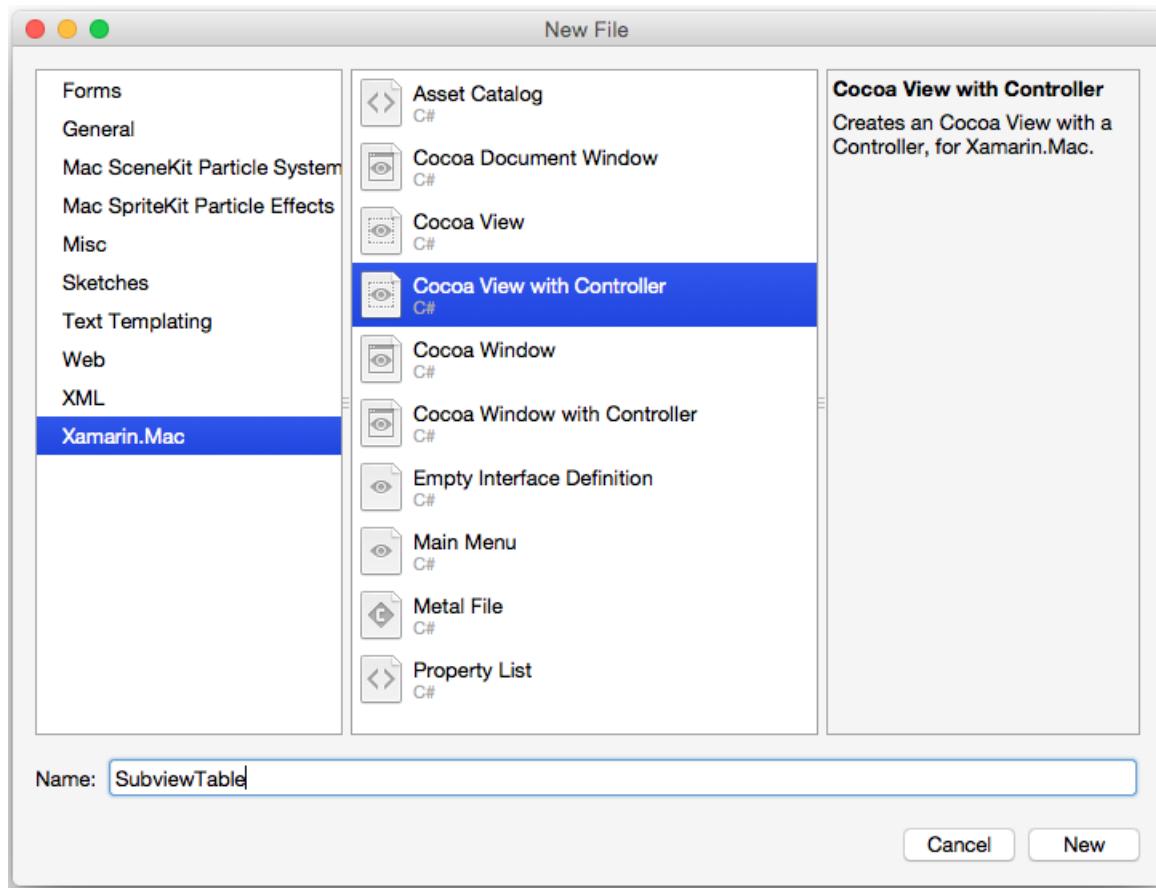
Adding a new view to a project

There are times when it is easier to break your Window's design down into several, more manageable .xib files. For example, like switching out the contents of the main Window when selecting a Toolbar item in a **Preferences Window** or swapping out content in response to a **Source List** selection.

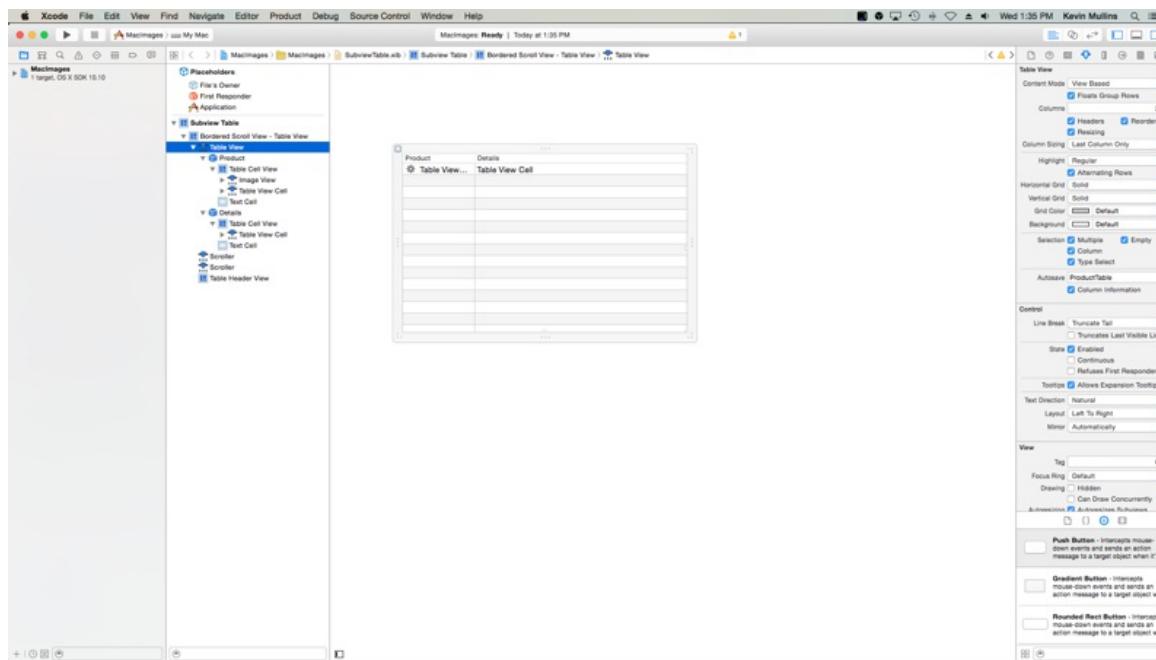
When adding a new View to your project you should always use the **Cocoa View with Controller** option, as this makes the process of loading the View from the .xib file easier.

To add a new view, do the following:

1. In the **Solution Pad**, right-click on the project and select **Add > New File...**
2. In the New File dialog box, select **Xamarin.Mac > Cocoa View with Controller**:



3. Enter **SubviewTable** for the Name and click the New button.
4. Double-click the **SubviewTable.xib** file to open it for editing in Interface Builder and Design the User Interface:



5. Wire up any required actions and outlets.
6. Save your changes and return to Visual Studio for Mac to sync with Xcode.

Next edit the **SubviewTable.cs** and add the following code to the **AwakeFromNib** file to populate the new View when it is loaded:

```

public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

    // Create the Product Table Data Source and populate it
    var DataSource = new ProductTableDataSource ();
    DataSource.Products.Add (new Product ("Xamarin.iOS", "Allows you to develop native iOS Applications in C#"));
    DataSource.Products.Add (new Product ("Xamarin.Android", "Allows you to develop native Android Applications in C#"));
    DataSource.Products.Add (new Product ("Xamarin.Mac", "Allows you to develop Mac native Applications in C#"));
    DataSource.Sort ("Title", true);

    // Populate the Product Table
    ProductTable.DataSource = DataSource;
    ProductTable.Delegate = new ProductTableDelegate (DataSource);

    // Auto select the first row
    ProductTable.SelectRow (0, false);
}

```

Add an enum to the project to track which view is currently being display. For example, **SubviewType.cs**:

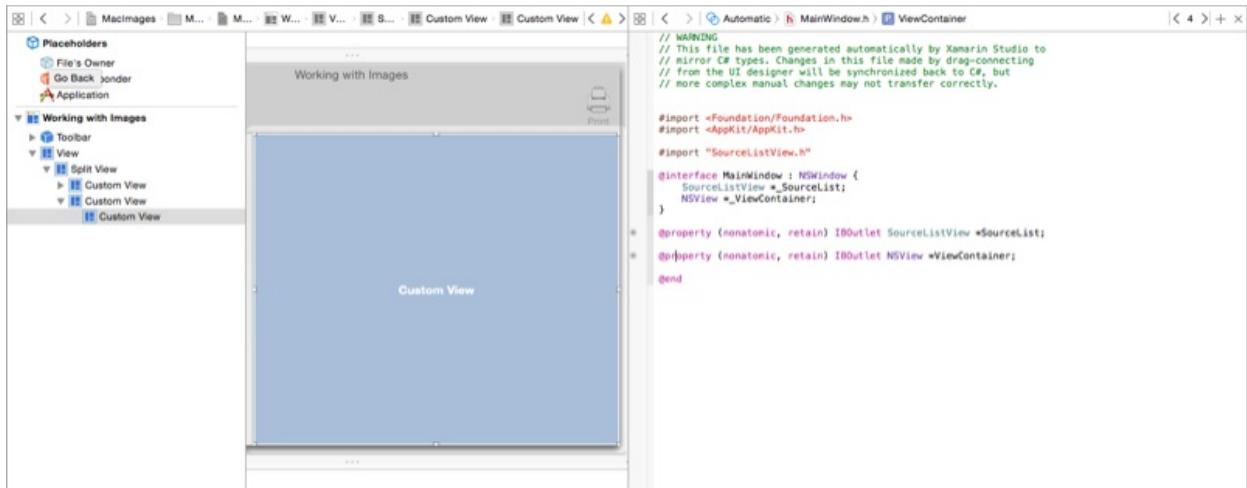
```

public enum SubviewType
{
    None,
    TableView,
    OutlineView,
    ImageView
}

```

Edit the .xib file of the window that will be consuming the View and displaying it. Add a **Custom View** that will act as the container for the View once it is loaded into memory by C# code and expose it to an outlet called

ViewContainer :



Save your changes and return to Visual Studio for Mac to sync with Xcode.

Next, edit the .cs file of the Window that will be displaying the new view (for example, **MainWindow.cs**) and add the following code:

```

private SubviewType viewType = SubviewType.None;
private NSViewController SubviewController = null;
private NSView Subview = null;
...
private void DisplaySubview(NSViewController controller, SubviewType type) {

    // Is this view already displayed?
    if (ViewType == type) return;

    // Is there a view already being displayed?
    if (Subview != null) {
        // Yes, remove it from the view
        Subview.RemoveFromSuperview ();

        // Release memory
        Subview = null;
        SubviewController = null;
    }

    // Save values
    viewType = type;
    SubviewController = controller;
    Subview = controller.View;

    // Define frame and display
    Subview.Frame = new CGRect (0, 0, ViewContainer.Frame.Width, ViewContainer.Frame.Height);
    ViewContainer.AddSubview (Subview);
}

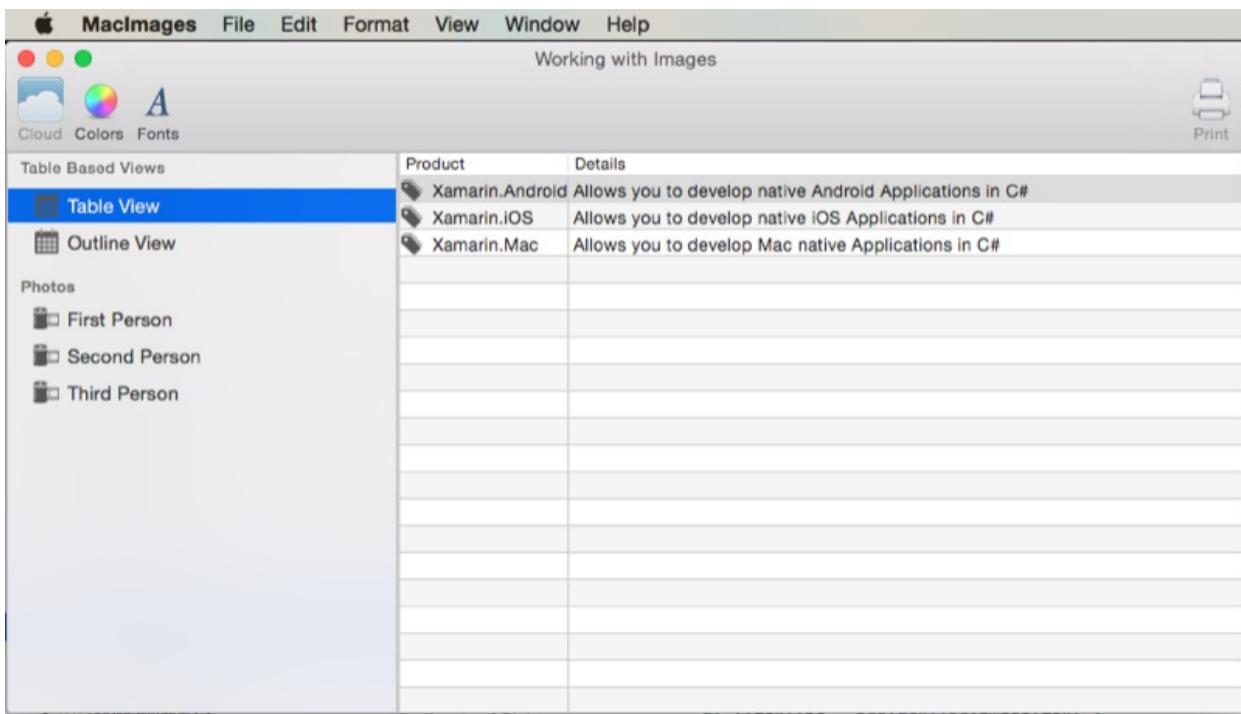
```

When we need to show a new View loaded from a .xib file in the Window's Container (the **Custom View** added above), this code handles removing any existing view and swapping it out for the new one. It looks to see if you already have a view displayed, if so it removes it from the screen. Next it takes the view that has been passed in (as loaded from a View Controller) resizes it to fit in the Content Area and adds it to the content for display.

To display a new view, use the following code:

```
DisplaySubview(new SubviewTableController(), SubviewType.TableView);
```

This creates a new instance of the View Controller for the new view to be displayed, sets its type (as specified by the enum added to the project) and uses the `DisplaySubview` method added to the Window's class to actually display the view. For example:



For more information on working with Windows in a Xamarin.Mac application, please see our [Windows](#) and [Dialogs](#) documentation.

Summary

This article has taken a detailed look at working with .xib files in a Xamarin.Mac application. We saw the different types and uses of .xib files to create your application's User Interface, how to create and maintain .xib files in Xcode's Interface Builder and how to work with .xib files in C# code.

Related Links

- [MacImages \(sample\)](#)
- [Hello, Mac](#)
- [Windows](#)
- [Menus](#)
- [Dialogs](#)
- [Working with images](#)
- [macOS Human Interface Guidelines](#)

.storyboard/.xib-less user interface design in Xamarin.Mac

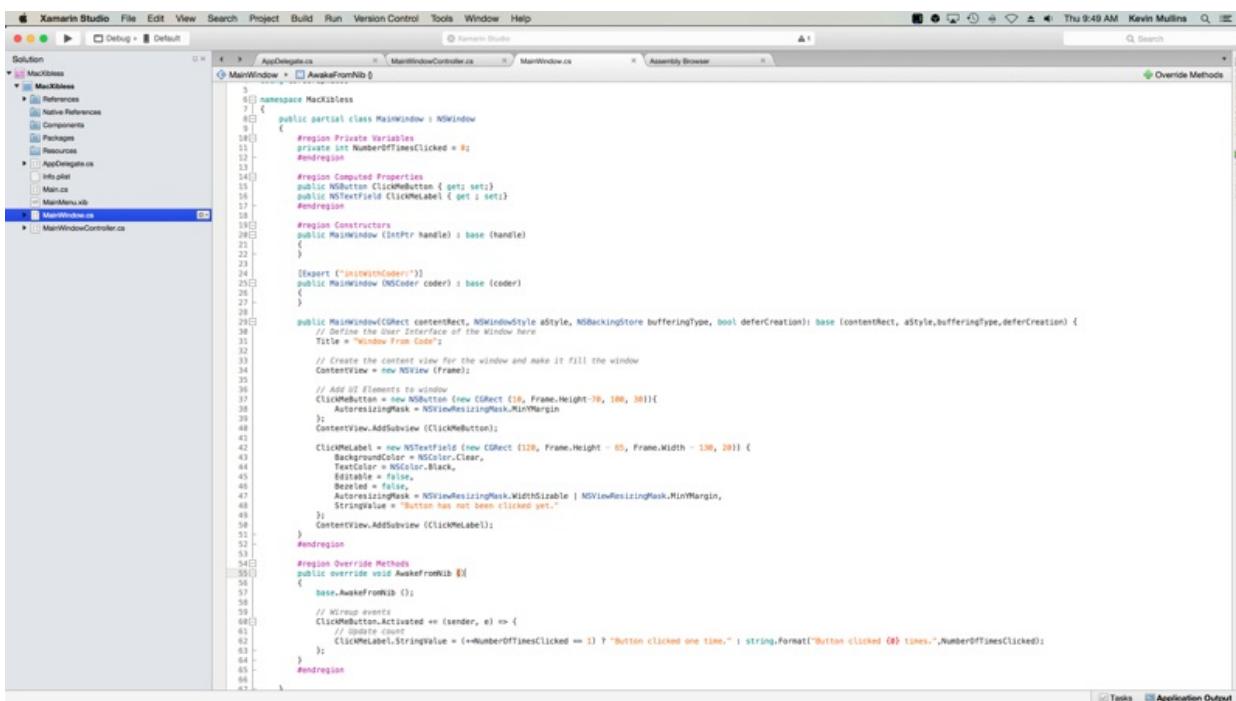
11/2/2020 • 10 minutes to read • [Edit Online](#)

This article covers creating a Xamarin.Mac application's user interface directly from C# code, without .storyboard files, .xib files, or Interface Builder.

Overview

When working with C# and .NET in a Xamarin.Mac application, you have access to the same user interface elements and tools that a developer working in *Objective-C* and *Xcode* does. Typically, when creating a Xamarin.Mac application, you'll use Xcode's Interface Builder with .storyboard or .xib files to create and maintain your application's user interface.

You also have the option of creating some or all of your Xamarin.Mac application's UI directly in C# code. In this article, we'll cover the basics of creating user interfaces and UI elements in C# code.



The screenshot shows the Xamarin Studio IDE. The solution tree on the left shows a project named "MacXbless" with files like AppDelegate.cs, Info.plist, Main.cs, and MainMenu.xib. The main editor window displays the code for MainWindow.cs. The code defines a MainWindow class that inherits from NSWindow. It includes properties for ClickMeButton and ClickMeLabel, and a constructor that initializes these elements and sets up event handlers. The code uses standard C# syntax and includes comments explaining the Objective-C-like interface definition.

```
using System;
using Foundation;
using AppKit;

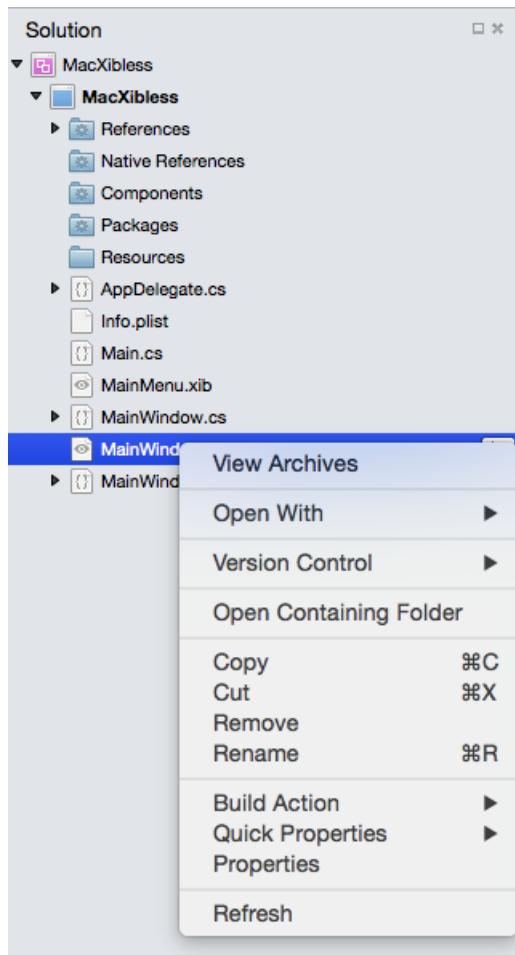
namespace MacXbless
{
    public partial class MainWindow : NSWindow
    {
        private int NumberOfTimesClicked = 0;
        #region Computed Properties
        public NSButton ClickMeButton { get; set; }
        public NSTextField ClickMeLabel { get; set; }
        #endregion
        #region Constructors
        public MainWindow (IntPtr handle) : base (handle)
        {
        }
        [Export ("initWithCoder:")]
        public MainWindow (NSCoder coder) : base (coder)
        {
        }
        public MainWindow (CGRect contentRect, NSWindowStyle aStyle, NSBackingStore bufferingType, bool deferCreation) : base (contentRect, aStyle, bufferingType, deferCreation)
        {
            // Define the User Interface of the window here
            Title = "Window From Code";
            // Create the content view for the window and make it fill the window
            ContentView = new NSView (Frame);
            // Add UI Elements to window
            ClickMeButton = new NSButton (new CGRect (10, Frame.Height - 70, 100, 30));
            ClickMeButton.Autore resizingMask = NSViewResizingMask.MinMargin;
            ContentView.AddSubview (ClickMeButton);
            ClickMeLabel = new NSTextField (new CGRect (120, Frame.Height - 65, Frame.Width - 130, 20));
            ClickMeLabel.BackgroundColor = NSColor.Clear;
            ClickMeLabel.TextColor = NSColor.Black;
            ClickMeLabel.setEditable = false;
            ClickMeLabel.Autore sizingMask = NSViewResizingMask.WidthSizable | NSViewResizingMask.MinMargin;
            ClickMeLabel.StringValue = "Button has not been clicked yet.";
            ContentView.AddSubview (ClickMeLabel);
        }
        #endregion
        #region Override Methods
        public override void AwakeFromNib ()
        {
            base.AwakeFromNib ();
            // Wire up events
            ClickMeButton.Activated += (sender, e) => {
                // Update count
                ClickMeLabel.StringValue = (++NumberOfTimesClicked == 1) ? "Button clicked one time." : string.Format ("Button clicked {0} times.", NumberOfTimesClicked);
            };
        }
        #endregion
    }
}
```

Switching a window to use code

When you create a new Xamarin.Mac Cocoa application, you get a standard blank, window by default. This window is defined in a **Main.storyboard** (or traditionally a **MainWindow.xib**) file automatically included in the project. This also includes a **ViewController.cs** file that manages the app's main view (or again traditionally a **MainWindow.cs** and a **MainWindowController.cs** file).

To switch to a Xibless window for an application, do the following:

1. Open the application that you want to stop using **.storyboard** or .xib files to define the user interface in Visual Studio for Mac.
2. In the **Solution Pad**, right-click on the **Main.storyboard** or **MainWindow.xib** file and select **Remove**:



3. From the **Remove Dialog**, click the **Delete** button to remove the .storyboard or .xib completely from the project:



Now we'll need to modify the `MainWindow.cs` file to define the window's layout and modify the `ViewController.cs` or `MainWindowController.cs` file to create an instance of our `MainWindow` class since we are no longer using the .storyboard or .xib file.

Modern Xamarin.Mac apps that use Storyboards for their user interface may not automatically include the `MainWindow.cs`, `ViewController.cs` or `MainWindowController.cs` files. As required, simply add a new Empty C# Class to the project (Add > New File... > General > Empty Class) and name it the same as the missing file.

Defining the window in code

Next, edit the `MainWindow.cs` file and make it look like the following:

```
using System;
using Foundation;
using AppKit;
using CoreGraphics;

namespace MacXibless
{
    public partial class MainWindow : NSWindow
```

```

{
    #region Private Variables
    private int NumberOfTimesClicked = 0;
    #endregion

    #region Computed Properties
    public NSButton ClickMeButton { get; set; }
    public NSTextField ClickMeLabel { get ; set; }
    #endregion

    #region Constructors
    public MainWindow (IntPtr handle) : base (handle)
    {
    }

    [Export ("initWithCoder:")]
    public MainWindow (NSCoder coder) : base (coder)
    {
    }

    public MainWindow(CGRect contentRect, NSWindowStyle aStyle, NSBackingStore bufferingType, bool deferCreation): base (contentRect, aStyle,bufferingType,deferCreation) {
        // Define the user interface of the window here
        Title = "Window From Code";

        // Create the content view for the window and make it fill the window
        ContentView = new NSView (Frame);

        // Add UI elements to window
        ClickMeButton = new NSButton (new CGRect (10, Frame.Height-70, 100, 30)){
            AutoresizingMask = NSViewResizingMask.MinYMargin
        };
        ContentView.AddSubview (ClickMeButton);

        ClickMeLabel = new NSTextField (new CGRect (120, Frame.Height - 65, Frame.Width - 130, 20)) {
            BackgroundColor = NSColor.Clear,
            TextColor = NSColor.Black,
            Editable = false,
            Bezeled = false,
            AutoresizingMask = NSViewResizingMask.WidthSizable | NSViewResizingMask.MinYMargin,
            StringValue = "Button has not been clicked yet."
        };
        ContentView.AddSubview (ClickMeLabel);
    }
    #endregion

    #region Override Methods
    public override void AwakeFromNib ()
    {
        base.AwakeFromNib ();

        // Wireup events
        ClickMeButton.Activated += (sender, e) => {
            // Update count
            ClickMeLabel.StringValue = (++NumberOfTimesClicked == 1) ? "Button clicked one time." :
string.Format("Button clicked {0} times.",NumberOfTimesClicked);
        };
    }
    #endregion

}
}

```

Let's discuss a few of the key elements.

First, we added a few *Computed Properties* that will act like outlets (as if the window was created in a .storyboard or .xib file):

```
public NSButton ClickMeButton { get; set; }
public NSTextField ClickMeLabel { get; set; }
```

These will give us access to the UI elements that we are going to display on the window. Since the window isn't being inflated from a .storyboard or .xib file, we need a way to instantiate it (as we'll see later in the `MainWindowController` class). That's what this new constructor method does:

```
public MainWindow(CGRect contentRect, NSWindowStyle aStyle, NSBackingStore bufferingType, bool deferCreation): base (contentRect, aStyle, bufferingType, deferCreation) {
    ...
}
```

This is where we will design the layout of the window and place any UI elements needed to create the required user interface. Before we can add any UI elements to a window, it needs a *Content View* to contain the elements:

```
ContentView = new NSView (Frame);
```

This creates a Content View that will fill the window. Now we add our first UI element, an `NSButton`, to the window:

```
ClickMeButton = new NSButton (new CGRect (10, Frame.Height-70, 100, 30)) {
    AutoresizingMask = NSViewResizingMask.MinYMargin
};
ContentView.AddSubview (ClickMeButton);
```

The first thing to note here is that, unlike iOS, macOS uses mathematical notation to define its window coordinate system. So the origin point is in the lower left hand corner of the window, with values increasing right and towards the upper right hand corner of the window. When we create the new `NSButton`, we take this into account as we define its position and size on screen.

The `AutoresizingMask = NSViewResizingMask.MinYMargin` property tells the button that we want it to stay in the same location from the top of the window when the window is resized vertically. Again, this is required because (0,0) is at the bottom left of the window.

Finally, the `ContentView.AddSubview (ClickMeButton)` method adds the `NSButton` to the Content View so that it will be displayed on screen when the application is run and the window displayed.

Next a label is added to the window that will display the number of times that the `NSButton` has been clicked:

```
ClickMeLabel = new NSTextField (new CGRect (120, Frame.Height - 65, Frame.Width - 130, 20)) {
    BackgroundColor = NSColor.Clear,
    TextColor = NSColor.Black,
    Editable = false,
    Bezeled = false,
    AutoresizingMask = NSViewResizingMask.WidthSizable | NSViewResizingMask.MinYMargin,
    StringValue = "Button has not been clicked yet."
};
ContentView.AddSubview (ClickMeLabel);
```

Since macOS doesn't have a specific *Label*/UI element, we've added a specially styled, non-editable `NSTextField` to act as a Label. Just like the button before, the size and location takes into account that (0,0) is at the bottom left of the window. The `AutoresizingMask = NSViewResizingMask.WidthSizable | NSViewResizingMask.MinYMargin` property is using the `or` operator to combine two `NSViewResizingMask` features. This will make the label stay in the same location from the top of the window when the window is resized vertically and shrink and grow in

width as the window is resized horizontally.

Again, the `ContentView.AddSubview (ClickMeLabel)` method adds the `NSTextField` to the Content View so that it will be displayed on screen when the application is run and the window opened.

Adjusting the window controller

Since the design of the `MainWindow` is no longer being loaded from a .storyboard or .xib file, we'll need to make some adjustments to the window controller. Edit the `MainWindowController.cs` file and make it look like the following:

```
using System;
using Foundation;
using AppKit;
using CoreGraphics;

namespace MacXibless
{
    public partial class MainWindowController : NSWindowController
    {
        public MainWindowController (IntPtr handle) : base (handle)
        {
        }

        [Export ("initWithCoder:")]
        public MainWindowController (NSCoder coder) : base (coder)
        {
        }

        public MainWindowController () : base ("MainWindow")
        {
            // Construct the window from code here
            CGRect contentRect = new CGRect (0, 0, 1000, 500);
            base.Window = new MainWindow(contentRect, (NSWindowStyle.Titled | NSWindowStyle.Closable |
NSWindowStyle.Miniaturizable | NSWindowStyle.Resizable), NSBackingStore.Buffered, false);

            // Simulate Awaking from Nib
            Window.AwakeFromNib ();
        }

        public override void AwakeFromNib ()
        {
            base.AwakeFromNib ();
        }

        public new MainWindow Window {
            get { return (MainWindow)base.Window; }
        }
    }
}
```

Let discuss the key elements of this modification.

First, we define a new instance of the `MainWindow` class and assign it to the base window controller's `Window` property:

```
CGRect contentRect = new CGRect (0, 0, 1000, 500);
base.Window = new MainWindow(contentRect, (NSWindowStyle.Titled | NSWindowStyle.Closable |
NSWindowStyle.Miniaturizable | NSWindowStyle.Resizable), NSBackingStore.Buffered, false);
```

We define the location of the window of screen with a `CGRect`. Just like the coordinate system of the window,

the screen defines (0,0) as the lower left hand corner. Next, we define the style of the window by using the `Or` operator to combine two or more `NSWindowStyle` features:

```
... (NSWindowStyle.Titled | NSWindowStyle.Closable | NSWindowStyle.Miniaturizable | NSWindowStyle.Resizable)  
...
```

The following `NSWindowStyle` features are available:

- **Borderless** - The window will have no border.
- **Titled** - The window will have a title bar.
- **Closable** - The window has a Close Button and can be closed.
- **Miniaturizable** - The window has a Miniaturize Button and can be minimized.
- **Resizable** - The window will have a Resize Button and be resizable.
- **Utility** - The window is a Utility style window (panel).
- **DocModal** - If the window is a Panel, it will be Document Modal instead of System Modal.
- **NonactivatingPanel** - If the window is a Panel, it will not be made the main window.
- **TexturedBackground** - The window will have a textured background.
- **Unscaled** - The window will not be scaled.
- **UnifiedTitleAndToolbar** - The window's title and toolbar areas will be joined.
- **Hud** - The window will be displayed as a heads-up display Panel.
- **FullScreenWindow** - The window can enter full screen mode.
- **FullSizeContentView** - The window's content view is behind the title and toolbar Area.

The last two properties define the *Buffering Type* for the window and if drawing of the window will be deferred. For more information on `NSWindows`, please see Apple's [Introduction to Windows](#) documentation.

Finally, since the window isn't being inflated from a .storyboard or .xib file, we need to simulate it in our `MainWindowController.cs` by calling the windows `Window.AwakeFromNib` method:

```
Window.AwakeFromNib();
```

This will allow you to code against the window just like a standard window loaded from a .storyboard or .xib file.

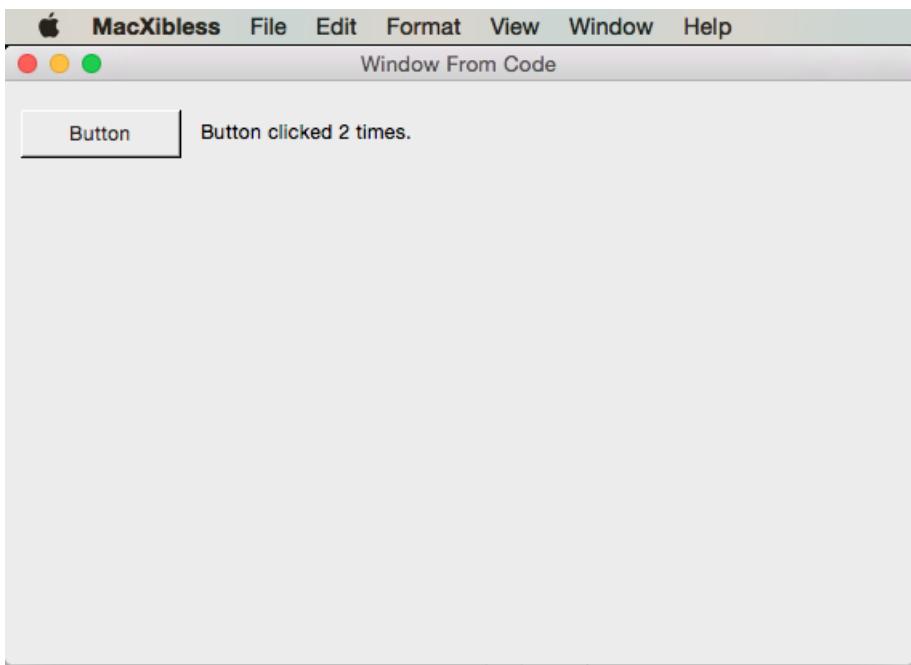
Displaying the window

With the .storyboard or .xib file removed and the `MainWindow.cs` and `MainWindowController.cs` files modified, you'll be using the window just as you would any normal window that had been created in Xcode's Interface Builder with a .xib file.

The following will create a new instance of the window and its controller and display the window on screen:

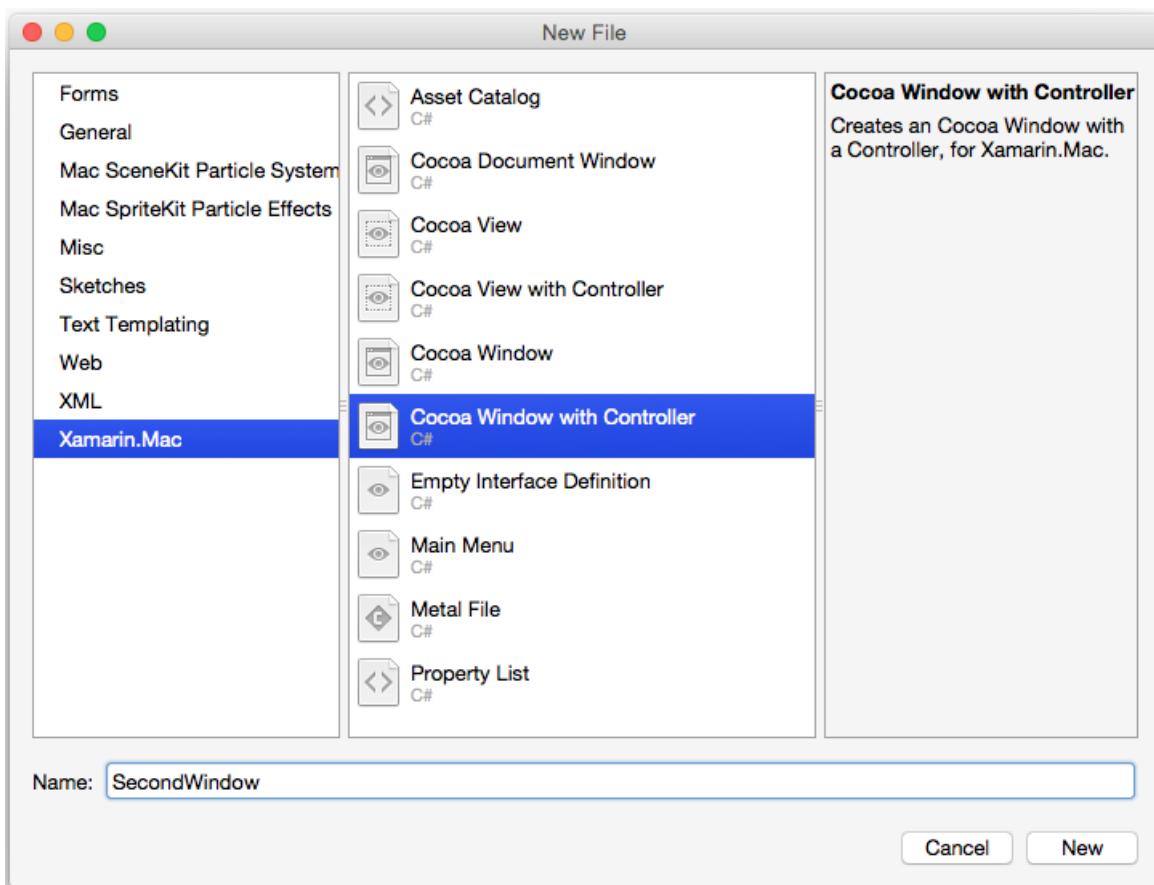
```
private MainWindowController mainWindowController;  
...  
  
mainWindowController = new MainWindowController();  
mainWindowController.Window.MakeKeyAndOrderFront (this);
```

At this point, if the application is run and the button clicked a couple of times, the following will be displayed:



Adding a code only window

If we want to add a code only, xibless window to an existing Xamarin.Mac application, right-click on the project in the Solution Pad and select Add > New File... In the New File dialog choose **Xamarin.Mac > Cocoa Window with Controller**, as illustrated below:



Just like before, we'll delete the default .storyboard or .xib file from the project (in this case **SecondWindow.xib**) and follow the steps in the [Switching a Window to use Code](#) section above to cover the window's definition to code.

Adding a UI element to a window in code

Whether a window was created in code or loaded from a .storyboard or .xib file, there might be times where we want to add a UI element to a window from code. For example:

```
var ClickMeButton = new NSButton (new CGRect (10, 10, 100, 30)){
    AutoresizingMask = NSViewResizingMask.MinYMargin
};
MyWindow.ContentView.AddSubview (ClickMeButton);
```

The above code creates a new `NSButton` and adds it to the `MyWindow` window instance for display. Basically any UI element that can be defined in Xcode's Interface Builder in a .storyboard or .xib file can be created in code and displayed in a window.

Defining the menu bar in code

Because of current limitations in Xamarin.Mac, it is not suggested that you create your Xamarin.Mac application's Menu Bar—`NSMenuBar`—in code but continue to use the `Main.storyboard` or `MainMenu.xib` file to define it. That said, you can add and remove Menus and Menu Items in C# code.

For example, edit the `AppDelegate.cs` file and make the `DidFinishLaunching` method look like the following:

```
public override void DidFinishLaunching (NSNotification notification)
{
    mainWindowController = new MainWindowController ();
    mainWindowController.Window.MakeKeyAndOrderFront (this);

    // Create a Status Bar Menu
    NSSStatusBar statusBar = NSSStatusBar.SystemStatusBar;

    var item = statusBar.CreateStatusItem (NSSStatusItemLength.Variable);
    item.Title = "Phrases";
    item.HighlightMode = true;
    item.Menu = new NSMenu ("Phrases");

    var address = new NSMenuItem ("Address");
    address.Activated += (sender, e) => {
        Console.WriteLine("Address Selected");
    };
    item.Menu.AddItem (address);

    var date = new NSMenuItem ("Date");
    date.Activated += (sender, e) => {
        Console.WriteLine("Date Selected");
    };
    item.Menu.AddItem (date);

    var greeting = new NSMenuItem ("Greeting");
    greeting.Activated += (sender, e) => {
        Console.WriteLine("Greetings Selected");
    };
    item.Menu.AddItem (greeting);

    var signature = new NSMenuItem ("Signature");
    signature.Activated += (sender, e) => {
        Console.WriteLine("Signature Selected");
    };
    item.Menu.AddItem (signature);
}
```

The above creates a Status Bar menu from code and displays it when the application is launched. For more information on working with Menus, please see our [Menus](#) documentation.

Summary

This article has taken a detailed look at creating a Xamarin.Mac application's user interface in C# code as opposed to using Xcode's Interface Builder with .storyboard or .xib files.

Related Links

- [MacXibless \(sample\)](#)
- [Windows](#)
- [Menus](#)
- [macOS Human Interface Guidelines](#)
- [Introduction to Windows](#)

Images in Xamarin.Mac

11/2/2020 • 14 minutes to read • [Edit Online](#)

This article covers working with images and icons in a Xamarin.Mac application. It describes creating and maintaining the images needed to create your application's icon and using images in both C# code and Xcode's Interface Builder.

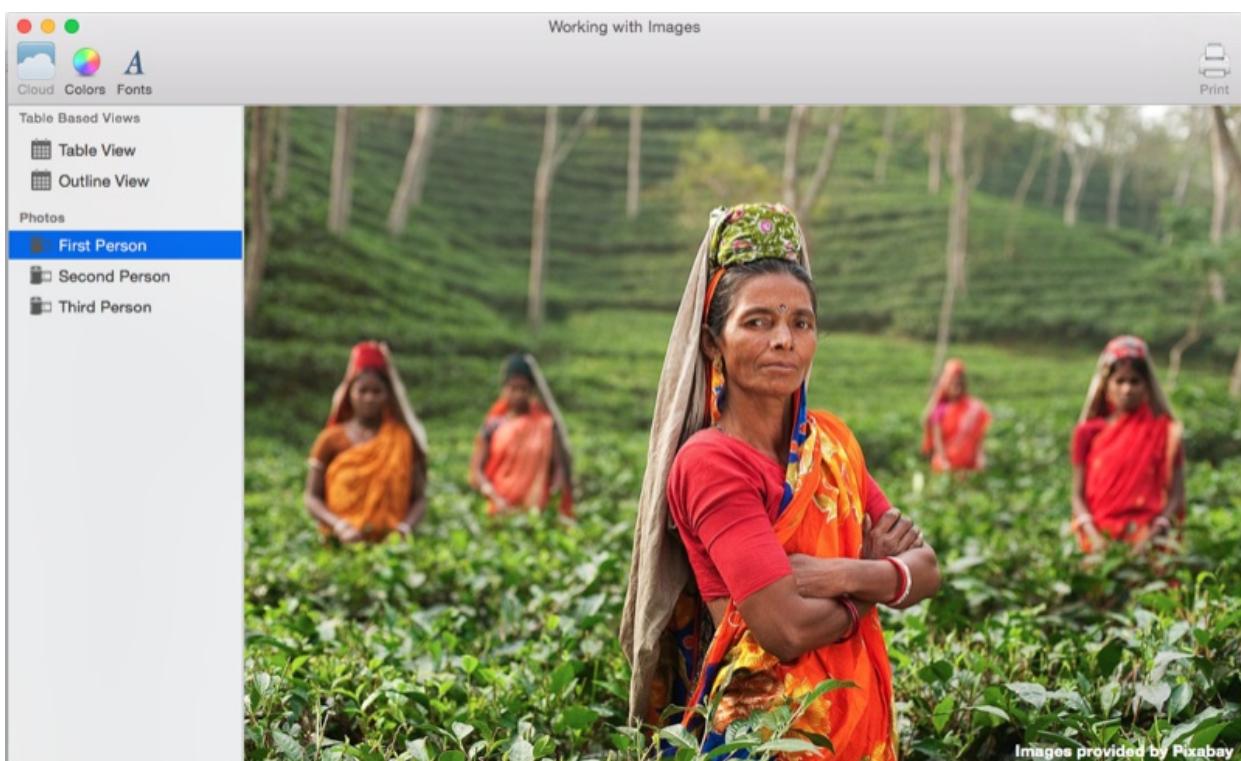
Overview

When working with C# and .NET in a Xamarin.Mac application, you have access to the same Image and Icon tools that a developer working in *Objective-C* and *Xcode* does.

There are several ways that image assets are used inside a macOS (formerly known as Mac OS X) application. From simply displaying an image as part of your application's UI to, assigning it to a UI control such as a Tool Bar or Source List Item, to providing Icons, Xamarin.Mac makes it easy to add great artwork to your macOS applications in the following ways:

- **UI elements** - Images can be displayed as backgrounds or as part of your application in a Image View (`NSImageView`).
- **Button** - Images can be displayed in buttons (`NSButton`).
- **Image Cell** - As part of a table based control (`NSTableView` or `NSOutlineView`), images can be used in a Image Cell (`NSImageCell`).
- **Toolbar Item** - Images can be added to a Toolbar (`NSToolbar`) as a Image Toolbar Item (`NSToolBarItem`).
- **Source List Icon** - As part of a Source List (a specially formatted `NSOutlineView`).
- **App Icon** - A series of images can be grouped together into a `.icns` set and used as your application's icon. See our [Application Icon](#) documentation for more information.

Additionally, macOS provides a set of predefined images that can be used throughout your application.



In this article, we'll cover the basics of working with Images and Icons in a Xamarin.Mac application. It is highly

suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

Adding images to a Xamarin.Mac project

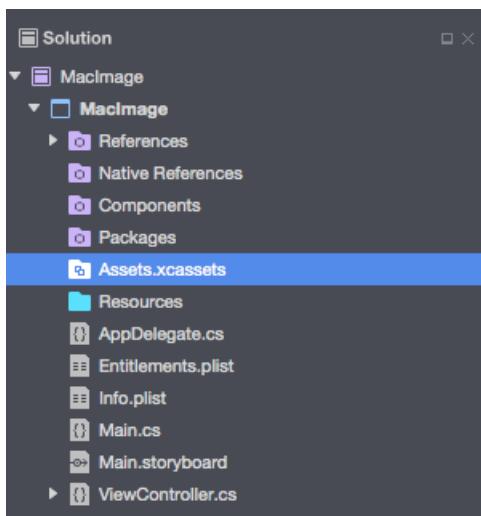
When adding an image for use in a Xamarin.Mac application, there are several places and ways that the developer can include image file to the project's source:

- **Main project tree [deprecated]** - Images can be added directly to the projects tree. When calling images stored in the main project tree from code, no folder location is specified. For example:
`NSImage image = NSImage.ImageNamed("tags.png"); .`
- **Resources folder [deprecated]** - The special **Resources** folder is for any file that will become part of the Application's Bundle such as Icon, Launch Screen or general Images (or any other image or file the developer wishes to add). When calling images stored in the **Resources** folder from code, just like images stored in the main project tree, no folder location is specified. For example: `NSImage.ImageNamed("tags.png") .`
- **Custom Folder or Subfolder [deprecated]** - The developer can add a custom folder to the projects source tree and store the images there. The location where the file is added can be nested in a subfolder to further help organize the project. For example, if the Developer added a `Card` folder to the project and a sub folder of `Hearts` to that folder, then store an image `Jack.png` in the `Hearts` folder,
`NSImage.ImageNamed("Card/Hearts/Jack.png")` would load the image at runtime.
- **Asset Catalog Image Sets [preferred]** - Added in OS X El Capitan, **Asset Catalogs Image Sets** contain all the versions or representations of an image that are necessary to support various devices and scale factors for your application. Instead of relying on the image assets filename (@1x, @2x).

Adding images to an asset catalog image set

As stated above, an **Asset Catalogs Image Sets** contain all the versions or representations of an image that are necessary to support various devices and scale factors for your application. Instead of relying on the image assets filename (see the Resolution Independent Images and Image Nomenclature above), **Image Sets** use the Asset Editor to specify which image belongs to which device and/or resolution.

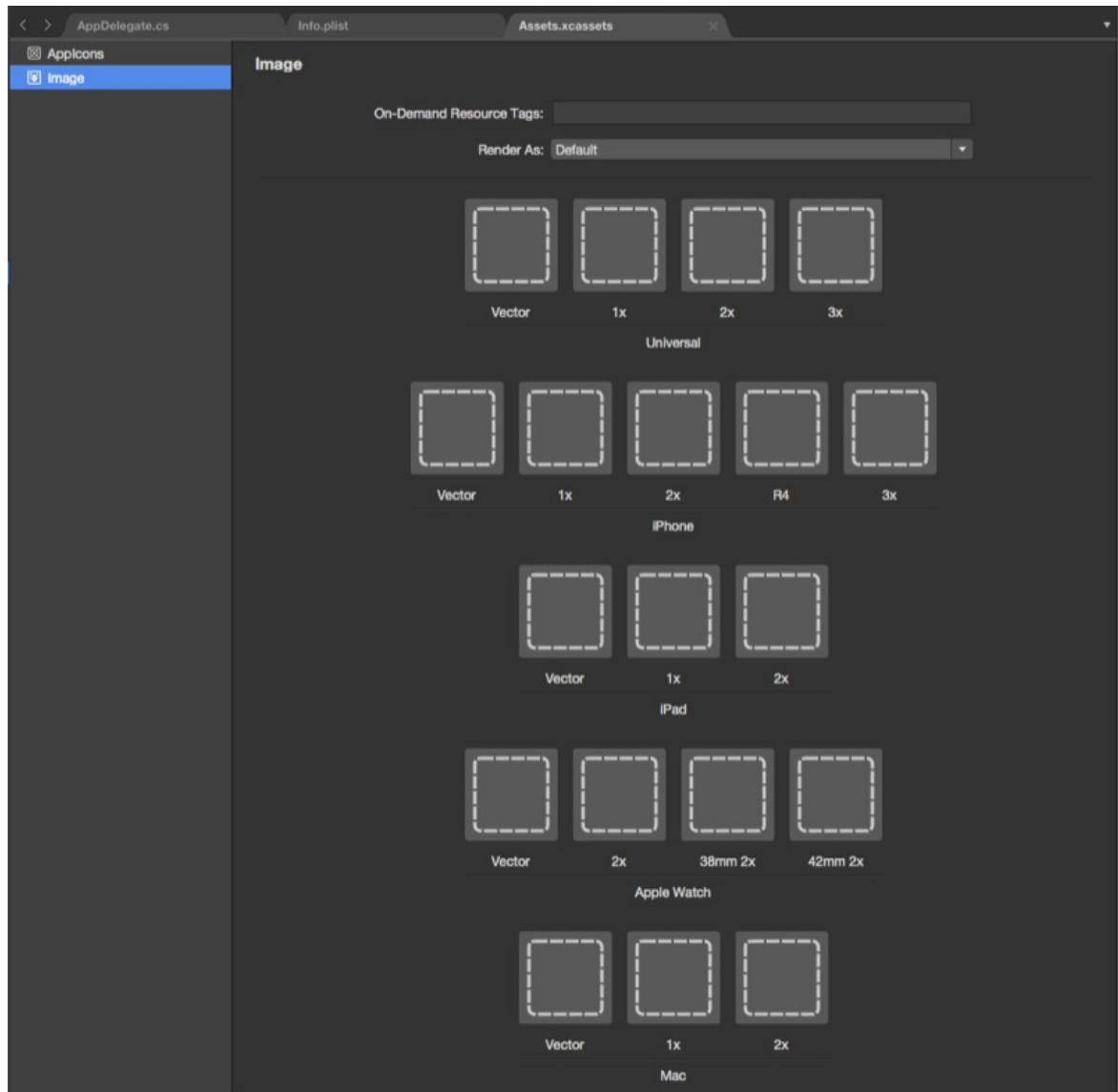
1. In the **Solution Pad**, double-click the `Assets.xcassets` file to open it for editing:



2. Right-click on the **Assets List** and select **New Image Set**:

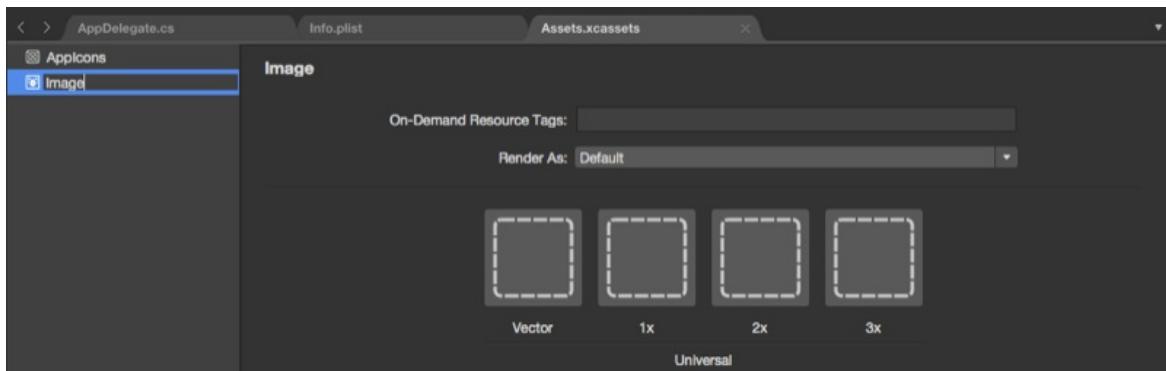


3. Select the new image set and the editor will be displayed:



4. From here we can drag in images for each of the different devices and resolutions required.

5. Double-click the new image set's Name in the Assets List to edit it:



A special **Vector** class has been added to **Image Sets** that allows us to include a *PDF* formatted vector image in the catalog instead of including individual bitmap files at the different resolutions. Using this method, you supply a single vector file for the @1x resolution (formatted as a vector PDF file) and the @2x and @3x versions of the file will be generated at compile time and included in the application's bundle.



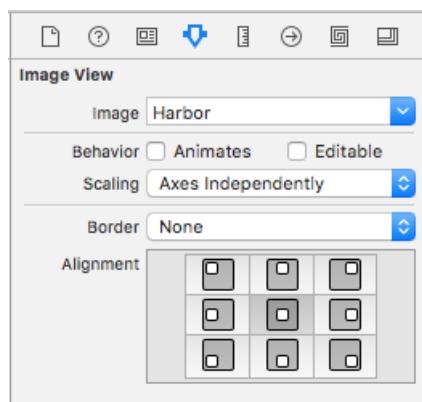
For example, if you include a `MonkeyIcon.pdf` file as the vector of an Asset Catalog with a resolution of 150px x 150px, the following bitmap assets would be included in the final app bundle when it was compiled:

1. `MonkeyIcon@1x.png` - 150px x 150px resolution.
2. `MonkeyIcon@2x.png` - 300px x 300px resolution.
3. `MonkeyIcon@3x.png` - 450px x 450px resolution.

The following should be taken into consideration when using PDF vector images in Asset Catalogs:

- This is not full vector support as the PDF will be rasterized to a bitmap at compile time and the bitmaps shipped in the final application.
- You cannot adjust the size of the image once it has been set in the Asset Catalog. If you attempt to resize the image (either in code or by using Auto Layout and Size Classes) the image will be distorted just like any other bitmap.

When using an **Image Set** in Xcode's Interface Builder, you can simply select the set's name from the dropdown list in the **Attribute Inspector**:

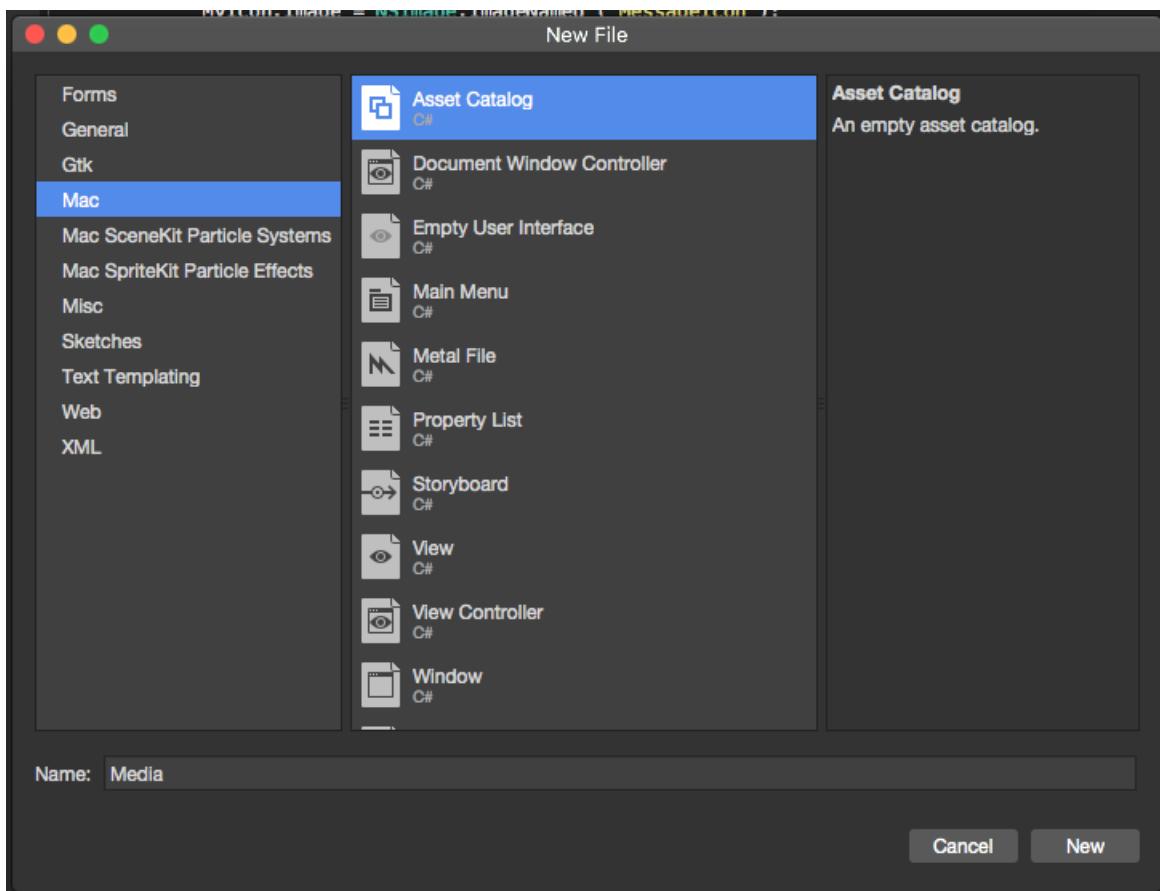


Adding new assets collections

When working with images in Assets Catalogs there might be times when you want to create a new collection, instead of adding all of your images to the **Assets.xcassets** collection. For example, when designing on-demand resources.

To add a new Assets Catalog to your project:

1. Right-click on the project in the **Solution Pad** and select **Add > New File...**
2. Select **Mac > Asset Catalog**, enter a **Name** for the collection and click the **New** button:



From here you can work with the collection in the same way as the default **Assets.xcassets** collection automatically included in the project.

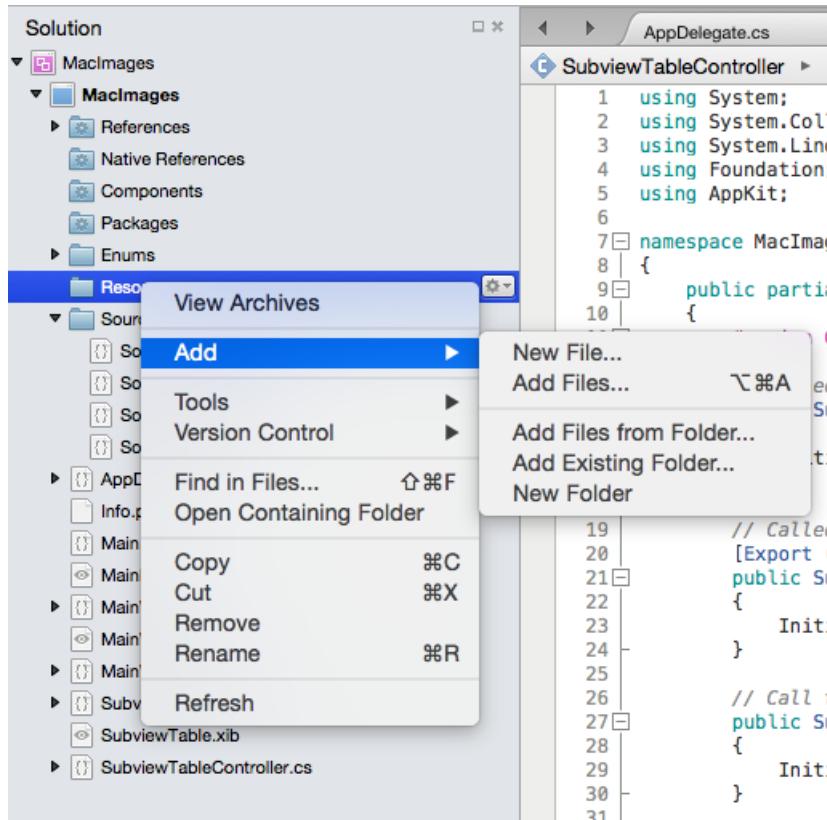
Adding images to resources

IMPORTANT

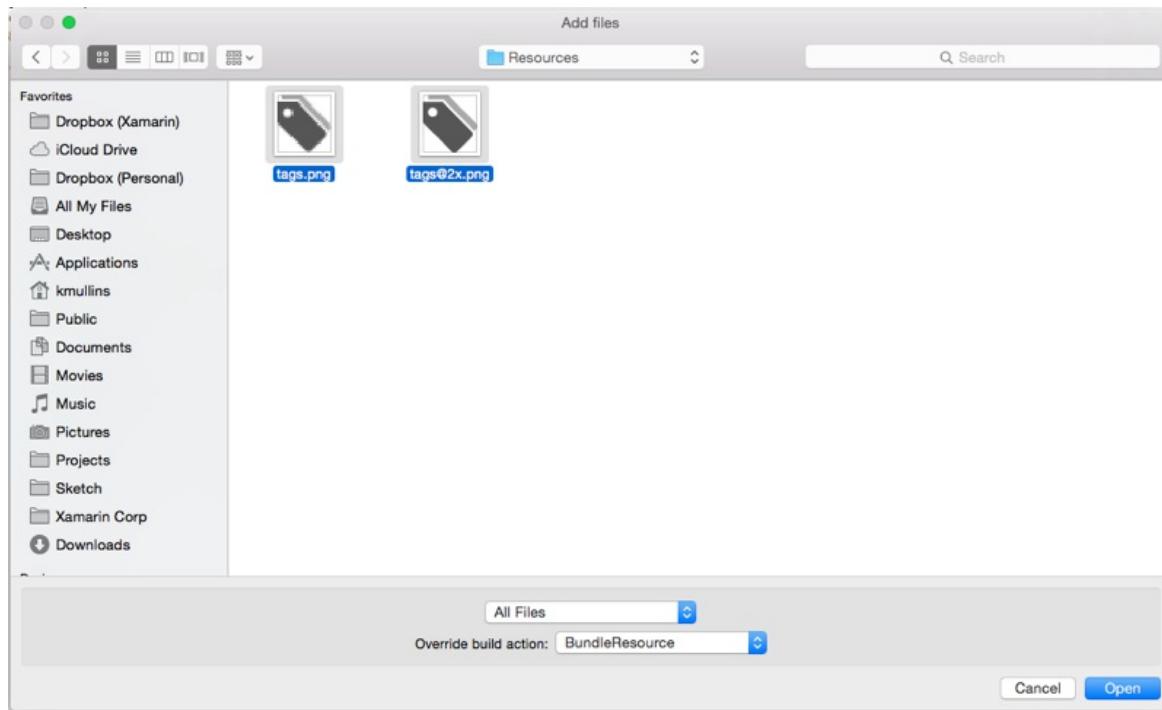
This method of working with images in a macOS app has been deprecated by Apple. You should use [Asset Catalog Image Sets](#) to manager your app's images instead.

Before you can use an Image file in your Xamarin.Mac application (either in C# code or from Interface Builder) it needs to be included in the project's **Resources** folder as a **Bundle Resource**. To add a file to a project, do the following:

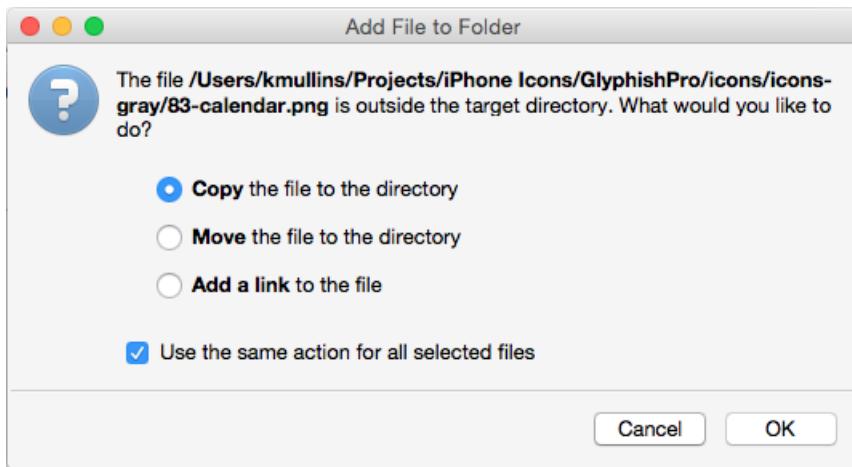
1. Right-click on the **Resources** folder in your project in the **Solution Pad** and select **Add > Add Files...**:



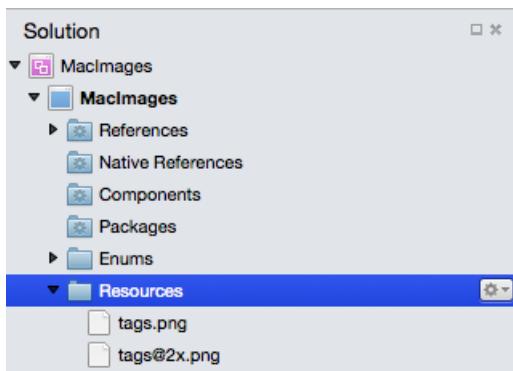
- From the Add Files dialog box, select the images files to add to the project, select `BundleResource` for the **Override build action** and click the **Open** button:



3. If the files are not already in the **Resources** folder, you'll be asked if you want to **Copy**, **Move** or **Link** the files. Pick which every suits your needs, typically that will be **Copy**:



4. The new files will be included in the project and ready for use:



5. Repeat the process for any image files required.

You can use any png, jpg, or pdf file as a source image in your Xamarin.Mac application. In the next section, we'll look at adding High Resolution versions of our Images and Icons to support Retina based Macs.

IMPORTANT

If you are adding Images to the **Resources** folder, you can leave the **Override build action** set to **Default**. The default Build Action for this folder is **BundleResource**.

Provide high-resolution versions of all app graphics resources

Any graphic asset that you add to a Xamarin.Mac application (icons, custom controls, custom cursors, custom artwork, etc.) need to have high-resolution versions in addition to their standard-resolution versions. This is required so that your application will look its best when run on a Retina Display equipped Mac computer.

Adopt the @2x naming convention

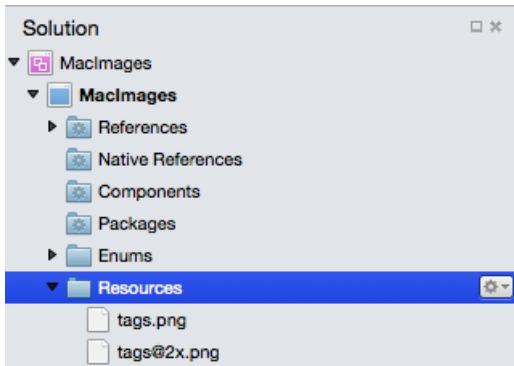
IMPORTANT

This method of working with images in a macOS app has been deprecated by Apple. You should use [Asset Catalog Image Sets](#) to manage your app's images instead.

When you create the standard and high-resolution versions of an image, follow this naming convention for the image pair when including them in your Xamarin.Mac project:

- Standard-Resolution - **ImageName.filename-extension** (Example: **tags.png**)
- High-Resolution - **ImageName@2x.filename-extension** (Example: **tags@2x.png**)

When added to a project, they would appear as follows:



When an image is assigned to a UI element in Interface Builder you'll simply pick the file in the *ImageName.filename-extension* format (Example: **tags.png**). The same for using an image in C# code, you'll pick the file in the *ImageName.filename-extension* format.

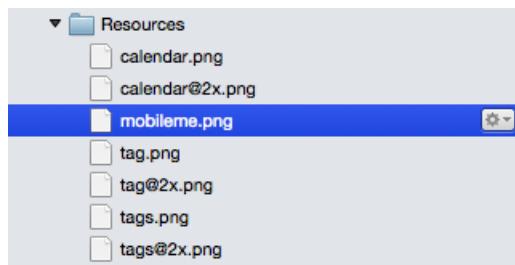
When you Xamarin.Mac application is run on a Mac, the *ImageName.filename-extension* format image will be used on Standard Resolution Displays, the **ImageName@2x.filename-extension** image will automatically be picked on Retina Display bases Macs.

Using images in Interface Builder

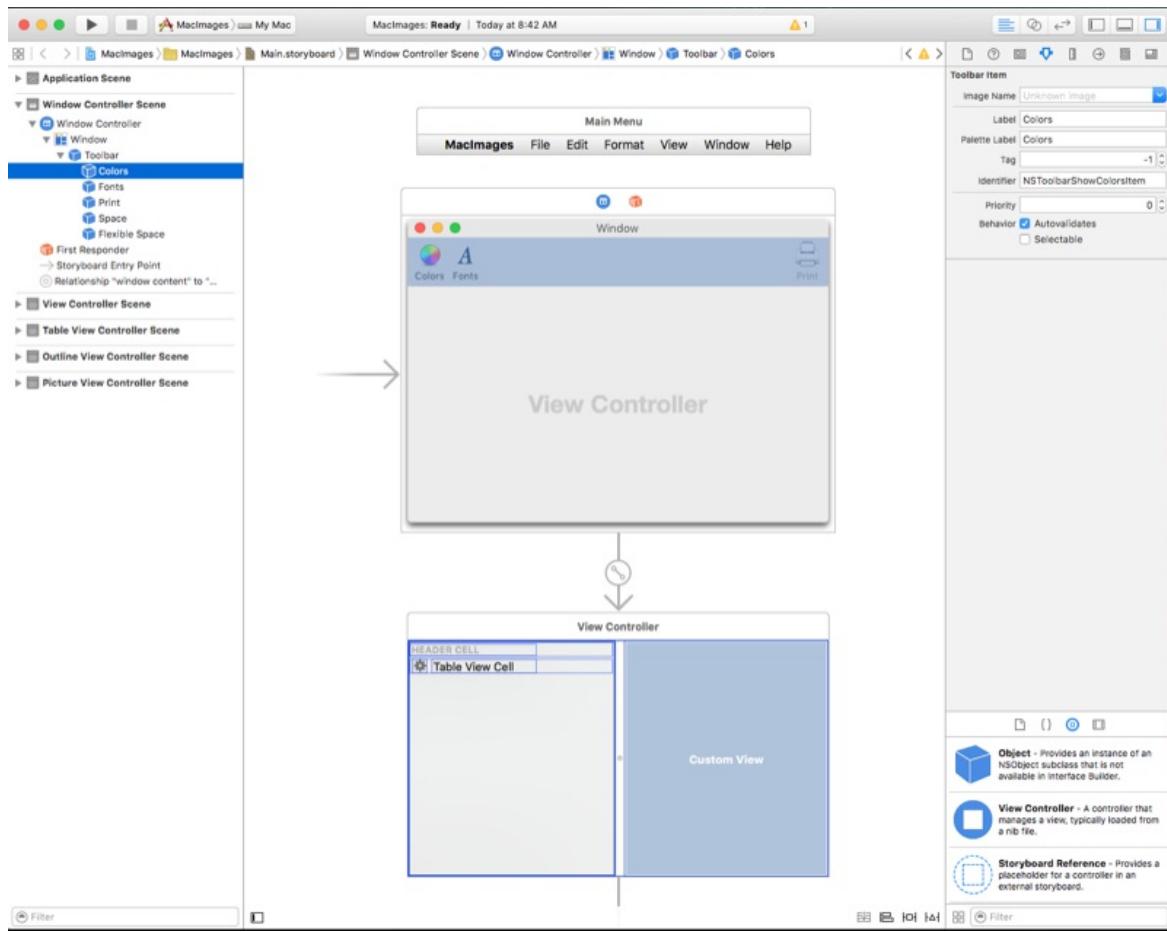
Any image resource the you have added to the **Resources** folder in your Xamarin.Mac project and have set the build action to **BundleResource** will automatically show up in Interface Builder and can be selected as part of a UI element (if it handles images).

To use an image in interface builder, do the following:

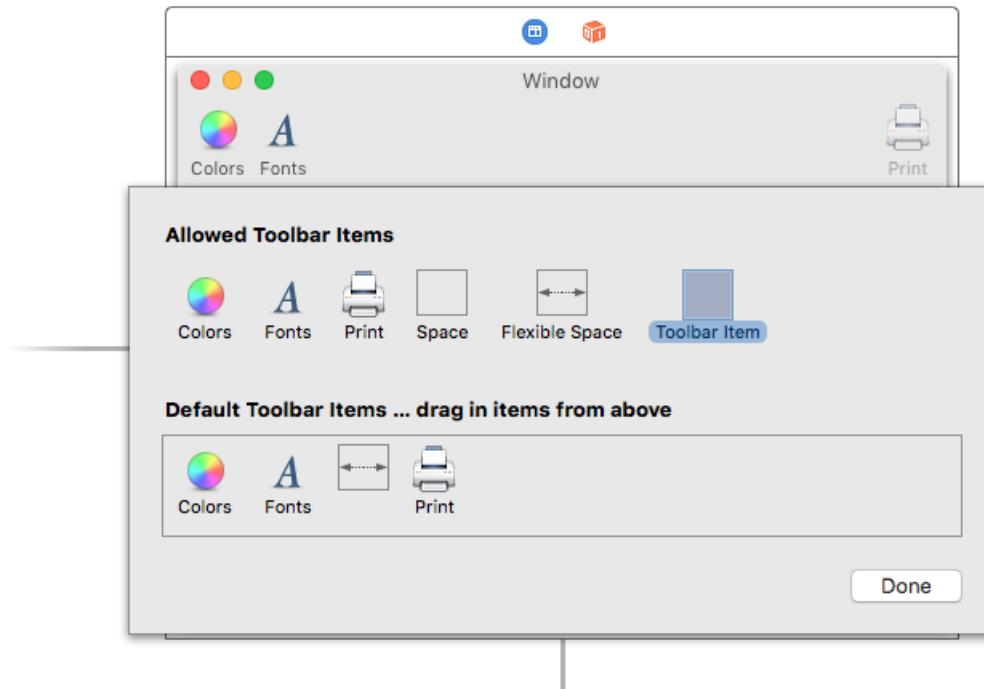
1. Add an image to the **Resources** folder with a **Build Action** of **BundleResource**:



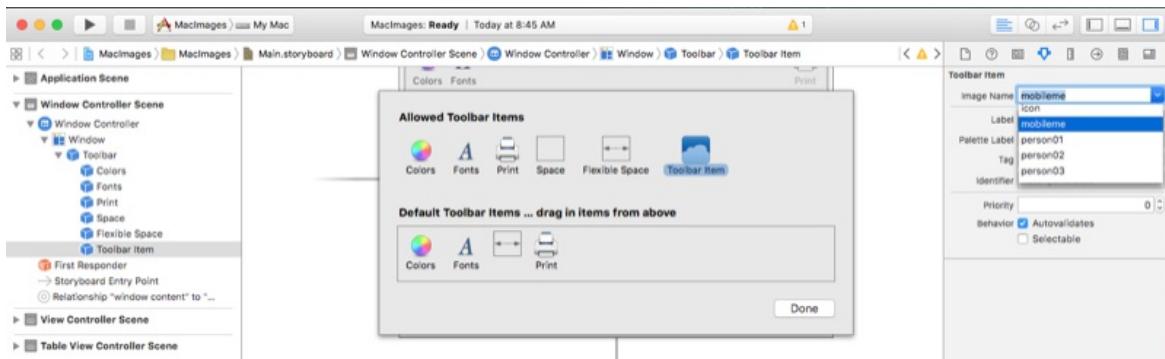
2. Double-click the **Main.storyboard** file to open it for editing in Interface Builder:



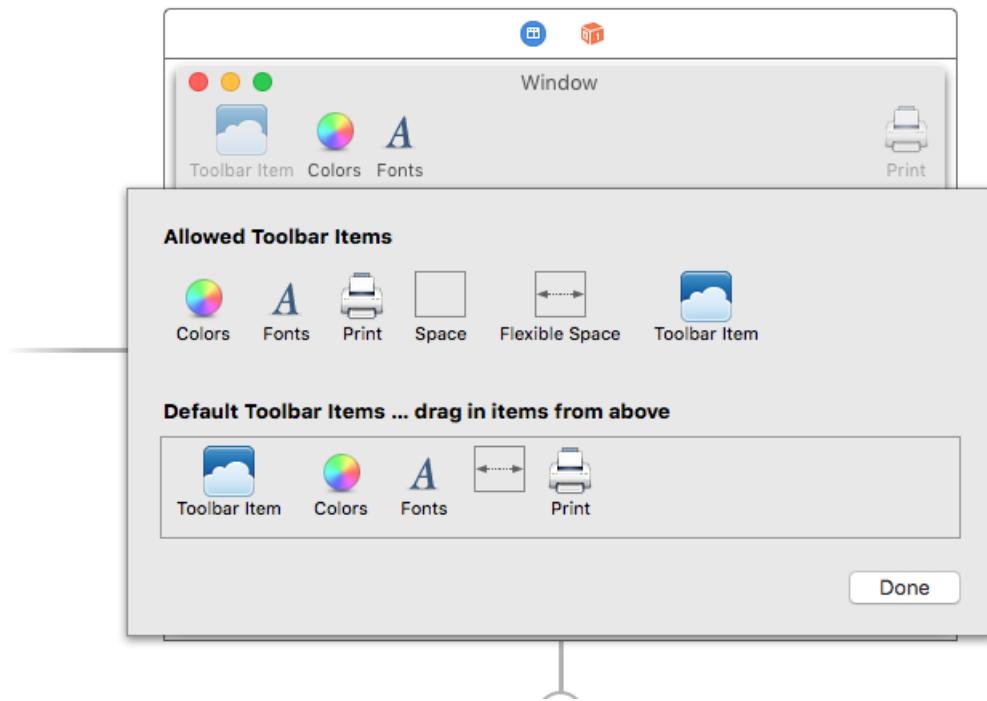
- Drag a UI element that takes images onto the design surface (for example, a **Image Toolbar Item**):



- Select the Image that you added to the Resources folder in the **Image Name** dropdown:



5. The selected image will be displayed in the design surface:



6. Save your changes and return to Visual Studio for Mac to sync with Xcode.

The above steps work for any UI element that allows their image property to be set in the **Attribute Inspector**. Again, if you have included a @2x version of your image file, it will automatically be used on Retina Display based Macs.

IMPORTANT

If the Image isn't available in the **Image Name** dropdown, close your .storyboard project in Xcode and reopen it from Visual Studio for Mac. If the image still isn't available, ensure that its **Build Action** is `BundleResource` and that the image has been added to the **Resources** folder.

Using images in C# code

When loading an image into memory using C# code in your Xamarin.Mac application, the image will be stored in a `NSImage` object. If the image file has been included in the Xamarin.Mac application bundle (included in resources), use the following code to load the image:

```
NSImage image = NSImage.ImageNamed("tags.png");
```

The above code uses the static `ImageNamed("...")` method of the `NSImage` class to load the given image into

memory from the **Resources** folder, if the image cannot be found, `null` will be returned. Like Images assigned in Interface Builder, if you have included a @2x version of your image file, it will automatically be used on Retina Display based Macs.

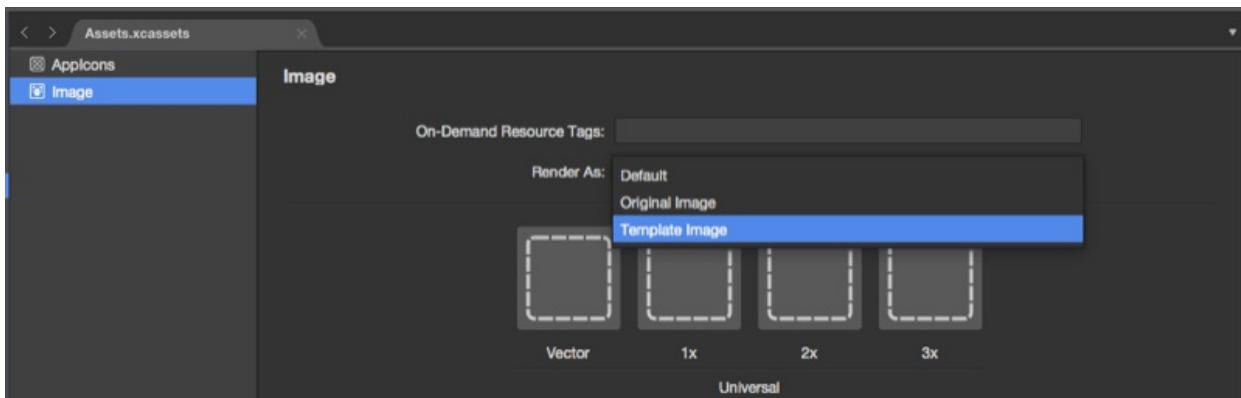
To load images outside of the application's bundle (from the Mac file system), use the following code:

```
NSImage image = new NSImage("/Users/KMullins/Documents/photo.jpg")
```

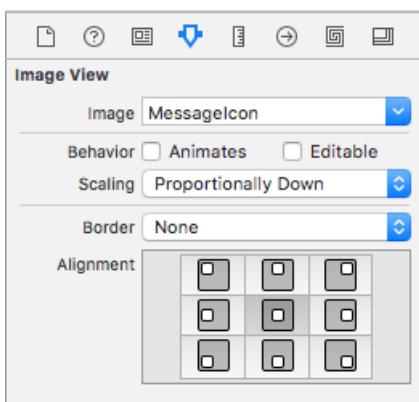
Working with template images

Based on the design of your macOS app, there might be times when you need to customize an icon or image inside of the User Interface to match a change in color scheme (for example, based on user preferences).

To achieve this effect, switch the *Render Mode* of your Image Asset to **Template Image**:



From the Xcode's Interface Builder, assign the Image Asset to a UI control:



Or optionally set the image source in code:

```
MyIcon.Image = NSImage.ImageNamed ("MessageIcon");
```

Add the following public function to your View Controller:

```
public NSImage ImageTintedWithColor(NSImage sourceImage, NSColor tintColor)
=> NSImage.ImageWithSize(sourceImage.Size, false, rect => {
    // Draw the original source image
    sourceImage.DrawInRect(rect, CGRect.Empty, NSCompositingOperation.SourceOver, 1f);

    // Apply tint
    tintColor.Set();
    NSGraphics.RectFill(rect, NSCompositingOperation.SourceAtop);

    return true;
});
```

IMPORTANT

Particularly with the advent of Dark Mode in macOS Mojave, it is important to avoid the `LockFocus` API when reating custom-rendered `NSImage` objects. Such images become static and will not be automatically updated to account for appearance or display density changes.

By employing the handler-based mechanism above, re-rendering for dynamic conditions will happen automatically when the `NSImage` is hosted, for example, in an `NSImageView`.

Finally, to tint a Template Image, call this function against the image to colorize:

```
MyIcon.Image = ImageTintedWithColor (MyIcon.Image, NSColor.Red);
```

Using images with table views

To include an image as part of the cell in a `NSTableView`, you'll need to change how the data is returned by the Table View's `NSTableViewDelegate`'s `GetViewForItem` method to use a `NTableViewCell` instead of the typical `NSTextField`. For example:

```

public override NSView GetViewForItem (NSTableView tableView, NSTableColumn tableColumn, nint row)
{

    // This pattern allows you reuse existing views when they are no-longer in use.
    // If the returned view is null, you instance up a new view
    // If a non-null view is returned, you modify it enough to reflect the new data
    NSTableCellView view = (NSTableCellView) tableView.MakeView (tableColumn.Title, this);
    if (view == null) {
        view = new NSTableCellView ();
        if (tableColumn.Title == "Product") {
            view.ImageView = new NSImageView (new CGRect (0, 0, 16, 16));
            view.AddSubview (view.ImageView);
            view.TextField = new NSTextField (new CGRect (20, 0, 400, 16));
        } else {
            view.TextField = new NSTextField (new CGRect (0, 0, 400, 16));
        }
        view.TextField.AutoresizingMask = NSViewResizingMask.WidthSizable;
        view.AddSubview (view.TextField);
        view.Identifier = tableColumn.Title;
        view.TextField.BackgroundColor = NSColor.Clear;
        view.TextField.Bordered = false;
        view.TextField.Selectable = false;
        view.TextField.Editable = true;

        view.TextField.EditingEnded += (sender, e) => {

            // Take action based on type
            switch (view.Identifier) {
                case "Product":
                    DataSource.Products [(int)view.TextField.Tag].Title = view.TextField.StringValue;
                    break;
                case "Details":
                    DataSource.Products [(int)view.TextField.Tag].Description = view.TextField.StringValue;
                    break;
            };
        };

        // Tag view
        view.TextField.Tag = row;
    }

    // Setup view based on the column selected
    switch (tableColumn.Title) {
        case "Product":
            view.ImageView.Image = NSImage.ImageNamed ("tags.png");
            view.TextField.StringValue = DataSource.Products [(int)row].Title;
            break;
        case "Details":
            view.TextField.StringValue = DataSource.Products [(int)row].Description;
            break;
    }

    return view;
}

```

There are a few lines of interest here. First, for columns that we want to include an image, we create a new `NSImageView` of the required size and location, we also create a new `NSTextField` and place its default position based on whether or not we are using an image:

```

if (tableColumn.Title == "Product") {
    view.ImageView = new NSImageView (new CGRect (0, 0, 16, 16));
    view.AddSubview (view.ImageView);
    view.TextField = new NSTextField (new CGRect (20, 0, 400, 16));
} else {
    view.TextField = new NSTextField (new CGRect (0, 0, 400, 16));
}

```

Secondly, we need to include the new Image View and Text Field in the parent `NSTableCellView`:

```

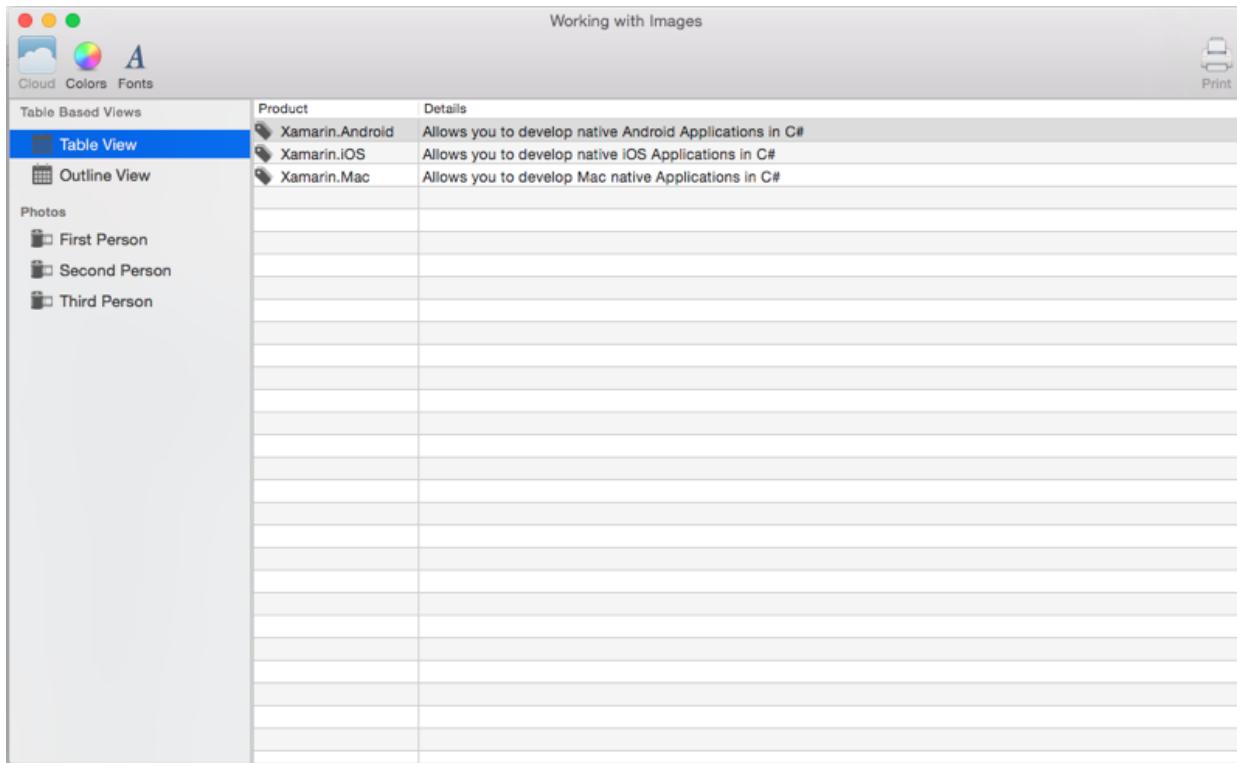
view.AddSubview (view.ImageView);
...
view.AddSubview (view.TextField);
...

```

Lastly, we need to tell the Text Field that it can shrink and grow with the Table View Cell:

```
view.TextField.AutoresizingMask = NSViewResizingMask.WidthSizable;
```

Example output:



For more information on working with Table Views, please see our [Table Views](#) documentation.

Using images with outline views

To include an image as part of the cell in a `NSOutlineView`, you'll need to change how the data is returned by the Outline View's `NSTableViewDelegate`'s `GetView` method to use a `NSTableCellView` instead of the typical `NSTextField`. For example:

```

public override NSView GetView (NSOutlineView outlineView, NSTableColumn tableColumn, NSObject item) {
    // Cast item
    var product = item as Product;

    // This pattern allows you reuse existing views when they are no-longer in use.
    // If the returned view is null, you instance up a new view
    // If a non-null view is returned, you modify it enough to reflect the new data
    NSTableCellView view = (NSTableCellView)outlineView.MakeView (tableColumn.Title, this);
    if (view == null) {
        view = new NSTableCellView ();
        if (tableColumn.Title == "Product") {
            view.ImageView = new NSImageView (new CGRect (0, 0, 16, 16));
            view.AddSubview (view.ImageView);
            view.TextField = new NSTextField (new CGRect (20, 0, 400, 16));
        } else {
            view.TextField = new NSTextField (new CGRect (0, 0, 400, 16));
        }
        view.TextField.AutoresizingMask = NSViewResizingMask.WidthSizable;
        view.AddSubview (view.TextField);
        view.Identifier = tableColumn.Title;
        view.TextField.BackgroundColor = NSColor.Clear;
        view.TextField.Bordered = false;
        view.TextField.Selectable = false;
        view.TextField.Editable = !product.IsProductGroup;
    }

    // Tag view
    view.TextField.Tag = outlineView.RowForItem (item);

    // Allow for edit
    view.TextField.Ending += (sender, e) => {

        // Grab product
        var prod = outlineView.ItemAtRow(view.Tag) as Product;

        // Take action based on type
        switch(view.Identifier) {
            case "Product":
                prod.Title = view.TextField.StringValue;
                break;
            case "Details":
                prod.Description = view.TextField.StringValue;
                break;
        }
    };

    // Setup view based on the column selected
    switch (tableColumn.Title) {
        case "Product":
            view.ImageView.Image = NSImage.ImageNamed (product.IsProductGroup ? "tags.png" : "tag.png");
            view.TextField.StringValue = product.Title;
            break;
        case "Details":
            view.TextField.StringValue = product.Description;
            break;
    }

    return view;
}

```

There are a few lines of interest here. First, for columns that we want to include an image, we create a new `NSImageView` of the required size and location, we also create a new `NSTextField` and place its default position based on whether or not we are using an image:

```

if (tableColumn.Title == "Product") {
    view.ImageView = new NSImageView (new CGRect (0, 0, 16, 16));
    view.AddSubview (view.ImageView);
    view.TextField = new NSTextField (new CGRect (20, 0, 400, 16));
} else {
    view.TextField = new NSTextField (new CGRect (0, 0, 400, 16));
}

```

Secondly, we need to include the new Image View and Text Field in the parent `NSTableCellView`:

```

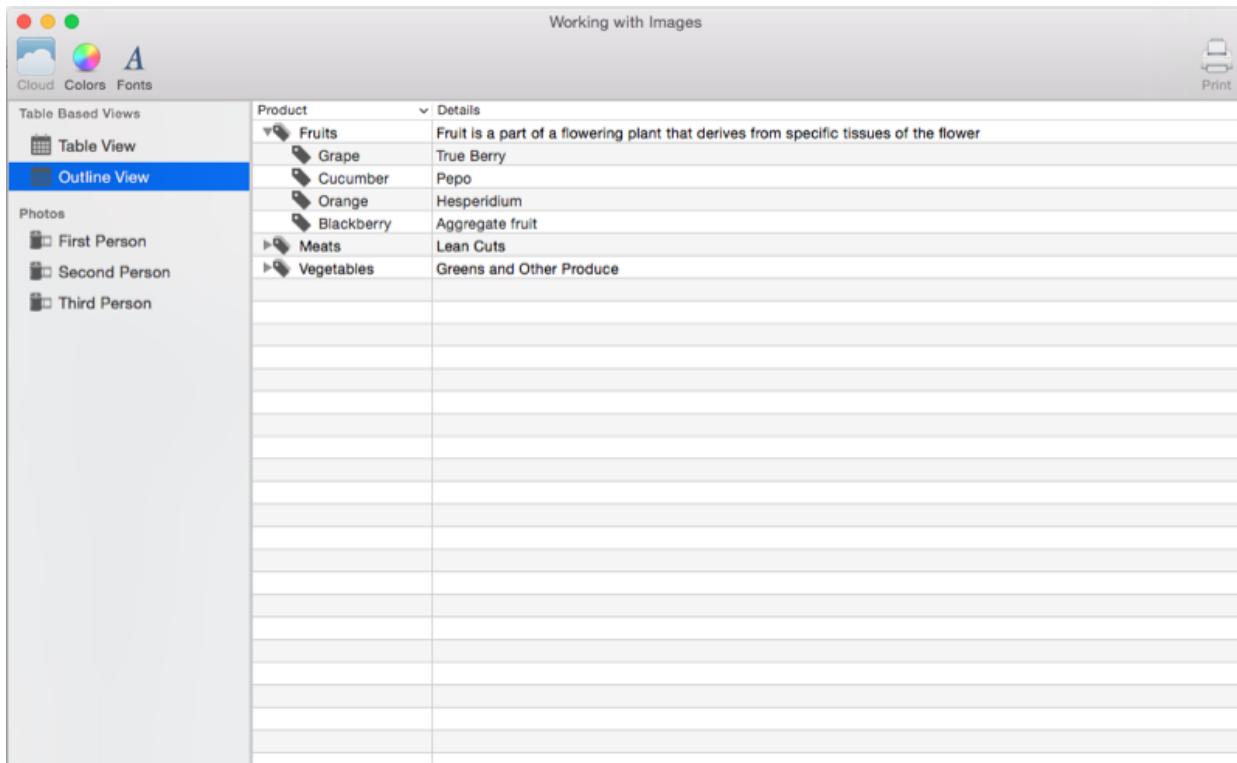
view.AddSubview (view.ImageView);
...
view.AddSubview (view.TextField);
...

```

Lastly, we need to tell the Text Field that it can shrink and grow with the Table View Cell:

```
view.TextField.AutoresizingMask = NSViewResizingMask.WidthSizable;
```

Example output:



For more information on working with Outline Views, please see our [Outline Views](#) documentation.

Summary

This article has taken a detailed look at working with Images and Icons in a Xamarin.Mac application. We saw the different types and uses of Images, how to use Images and Icons in Xcode's Interface Builder and how to work with Images and Icons in C# code.

Related Links

- [MacImages \(sample\)](#)

- [Hello, Mac](#)
- [Table views](#)
- [Outline views](#)
- [macOS X Human Interface Guidelines](#)
- [About High Resolution for OS X](#)

Data binding and key-value coding in Xamarin.Mac

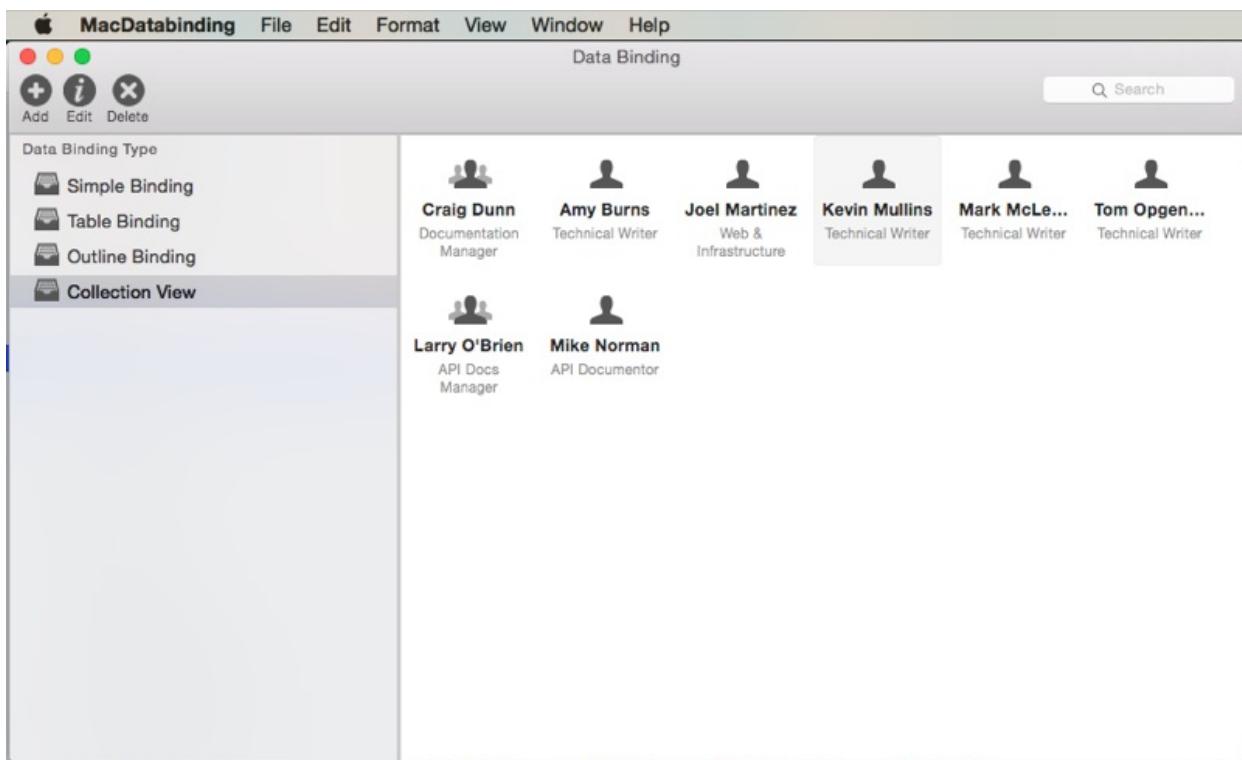
3/5/2021 • 21 minutes to read • [Edit Online](#)

This article covers using key-value coding and key-value observing to allow for data binding to UI elements in Xcode's Interface Builder.

Overview

When working with C# and .NET in a Xamarin.Mac application, you have access to the same key-value coding and data binding techniques that a developer working in *Objective-C* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, you can use Xcode's *Interface Builder* to Data Bind with UI elements instead of writing code.

By using key-value coding and data binding techniques in your Xamarin.Mac application, you can greatly decrease the amount of code that you have to write and maintain to populate and work with UI elements. You also have the benefit of further decoupling your backing data (*Data Model*) from your front end User Interface (*Model-View-Controller*), leading to easier to maintain, more flexible application design.



In this article, we'll cover the basics of working with key-value coding and data binding in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` attributes used to wire up your C# classes to Objective-C objects and UI elements.

What is key-value coding

Key-value coding (KVC) is a mechanism for accessing an object's properties indirectly, using keys (specially

formatted strings) to identify properties instead of accessing them through instance variables or accessor methods (`get/set`). By implementing key-value coding compliant accessors in your Xamarin.Mac application, you gain access to other macOS (formerly known as OS X) features such as key-value observing (KVO), data binding, Core Data, Cocoa bindings, and scriptability.

By using key-value coding and data binding techniques in your Xamarin.Mac application, you can greatly decrease the amount of code that you have to write and maintain to populate and work with UI elements. You also have the benefit of further decoupling your backing data (*Data Model*) from your front end User Interface (*Model-View-Controller*), leading to easier to maintain, more flexible application design.

For example, let's look at the following class definition of a KVC compliant object:

```
using System;
using Foundation;

namespace MacDatabinding
{
    [Register("PersonModel")]
    public class PersonModel : NSObject
    {
        private string _name = "";

        [Export("Name")]
        public string Name {
            get { return _name; }
            set {
                WillChangeValue ("Name");
                _name = value;
                DidChangeValue ("Name");
            }
        }

        public PersonModel ()
        {
        }
    }
}
```

First, the `[Register("PersonModel")]` attribute registers the class and exposes it to Objective-C. Then, the class needs to inherit from `NSObject` (or a subclass that inherits from `NSObject`), this adds several base method that allow to the class to be KVC compliant. Next, the `[Export("Name")]` attribute exposes the `Name` property and defines the Key value that will later be used to access the property through KVC and KVO techniques.

Finally, to be able to be Key-Value Observed changes to the property's value, the accessor must wrap changes to its value in `WillChangeValue` and `DidChangeValue` method calls (specifying the same Key as the `Export` attribute). For example:

```
set {
    WillChangeValue ("Name");
    _name = value;
    DidChangeValue ("Name");
}
```

This step is *very* important for data binding in Xcode's Interface Builder (as we will see later in this article).

For more information, please see Apple's [Key-Value Coding Programming Guide](#).

Keys and key paths

A *Key* is a string that identifies a specific property of an object. Typically, a key corresponds to the name of an accessor method in a key-value coding compliant object. Keys must use ASCII encoding, usually begin with a

lowercase letter, and may not contain whitespace. So given the example above, `Name` would be a Key Value of `Name` property of the `PersonModel` class. The Key and the name of the property that they expose don't have to be the same, however in most cases they are.

A *Key Path* is a string of dot separated Keys used to specify a hierarchy of object properties to traverse. The property of the first key in the sequence is relative to the receiver, and each subsequent key is evaluated relative to the value of the previous property. In the same way you use dot notation to traverse an object and its properties in a C# class.

For example, if you expanded the `PersonModel` class and added `Child` property:

```
using System;
using Foundation;

namespace MacDatabinding
{
    [Register("PersonModel")]
    public class PersonModel : NSObject
    {
        private string _name = "";
        private PersonModel _child = new PersonModel();

        [Export("Name")]
        public string Name {
            get { return _name; }
            set {
                WillChangeValue ("Name");
                _name = value;
                DidChangeValue ("Name");
            }
        }

        [Export("Child")]
        public PersonModel Child {
            get { return _child; }
            set {
                WillChangeValue ("Child");
                _child = value;
                DidChangeValue ("Child");
            }
        }

        public PersonModel ()
        {
        }
    }
}
```

The Key Path to the child's name would be `self.Child.Name` or simply `child.Name` (based on how the Key Value was being used).

Getting values using key-value coding

The `ValueForKey` method returns the value for the specified Key (as a `NSString`), relative to the instance of the KVC class receiving the request. For example, if `Person` is an instance of the `PersonModel` class defined above:

```
// Read value
var name = Person.ValueForKey (new NSString("Name"));
```

This would return the value of the `Name` property for that instance of `PersonModel`.

Setting values using key-value coding

Similarly, the `SetValueForKey` set the value for the specified Key (as a `NSString`), relative to the instance of the KVC class receiving the request. So again, using an instance of the `PersonModel` class, as shown below:

```
// Write value
Person.SetValueForKey(new NSString("Jane Doe"), new NSString("Name"));
```

Would change the value of the `Name` property to `Jane Doe`.

Observing value changes

Using key-value observing (KVO), you can attach an observer to a specific Key of a KVC compliant class and be notified any time the value for that Key is modified (either using KVC techniques or directly accessing the given property in C# code). For example:

```
// Watch for the name value changing
Person.AddObserver ("Name", NSKeyValueObservingOptions.New, (sender) => {
    // Inform caller of selection change
    Console.WriteLine("New Name: {0}", Person.Name)
});
```

Now, any time the `Name` property of the `Person` instance of the `PersonModel` class is modified, the new value is written out to the console.

For more information, please see Apple's [Introduction to Key-Value Observing Programming Guide](#).

Data binding

The following sections will show how you can use a key-value coding and key-value observing compliant class to bind data to UI elements in Xcode's Interface Builder, instead of reading and writing values using C# code. In this way you separate your *Data Model* from the views that are used to display them, making the Xamarin.Mac application more flexible and easier to maintain. You also greatly decrease the amount of code that has to be written.

Defining your data model

Before you can Data Bind a UI element in Interface Builder, you must have a KVC/KVO compliant class defined in your Xamarin.Mac application to act as the *Data Model* for the binding. The Data Model provides all of the data that will be displayed in the User Interface and receives any modifications to the data that the user makes in the UI while running the application.

For example, if you were writing an application that managed a group of employees, you could use the following class to define the Data Model:

```
using System;
using Foundation;
using AppKit;

namespace MacDatabinding
{
    [Register("PersonModel")]
    public class PersonModel : NSObject
    {
        #region Private Variables
        private string _name = "";
        private string _occupation = "";
        private bool _isManager = false;
        private NSMutableArray _people = new NSMutableArray();
        #endregion
    }
}
```

```

#region Computed Properties
[Export("Name")]
public string Name {
    get { return _name; }
    set {
        WillChangeValue ("Name");
        _name = value;
        DidChangeValue ("Name");
    }
}

[Export("Occupation")]
public string Occupation {
    get { return _occupation; }
    set {
        WillChangeValue ("Occupation");
        _occupation = value;
        DidChangeValue ("Occupation");
    }
}

[Export("isManager")]
public bool isManager {
    get { return _isManager; }
    set {
        WillChangeValue ("isManager");
        WillChangeValue ("Icon");
        _isManager = value;
        DidChangeValue ("isManager");
        DidChangeValue ("Icon");
    }
}

[Export("isEmployee")]
public bool isEmployee {
    get { return (NumberOfEmployees == 0); }
}

[Export("Icon")]
public NSImage Icon {
    get {
        if (isManager) {
            return NSImage.ImageNamed ("group.png");
        } else {
            return NSImage.ImageNamed ("user.png");
        }
    }
}

[Export("personModelArray")]
public NSArray People {
    get { return _people; }
}

[Export("NumberOfEmployees")]
public nint NumberOfEmployees {
    get { return (nint)_people.Count; }
}
#endregion

#region Constructors
public PersonModel ()
{
}

public PersonModel (string name, string occupation)
{
    // Initialize
}

```

```

        this.Name = name;
        this.Occupation = occupation;
    }

    public PersonModel (string name, string occupation, bool manager)
    {
        // Initialize
        this.Name = name;
        this.Occupation = occupation;
        this.isManager = manager;
    }
}

#endregion

#region Array Controller Methods
[Export("addObject:")]
public void AddPerson(PersonModel person) {
    WillChangeValue ("personModelArray");
    isManager = true;
    _people.Add (person);
    DidChangeValue ("personModelArray");
}

[Export("insertObject:inPersonModelAtIndex:")]
public void InsertPerson(PersonModel person, nint index) {
    WillChangeValue ("personModelArray");
    _people.Insert (person, index);
    DidChangeValue ("personModelArray");
}

[Export("removeObjectFromPersonModelAtIndex:")]
public void RemovePerson(nint index) {
    WillChangeValue ("personModelArray");
    _people.RemoveObject (index);
    DidChangeValue ("personModelArray");
}

[Export("setPersonModelArray:")]
public void SetPeople(NSMutableArray array) {
    WillChangeValue ("personModelArray");
    _people = array;
    DidChangeValue ("personModelArray");
}
#endregion
}
}

```

Most of the features of this class were covered in the [What is key-value coding](#) section above. However, let's look at a few specific elements and some additions that were made to allow this class to act as a Data Model for **Array Controllers** and **Tree Controllers** (which we'll be using later to Data bind **Tree Views**, **Outline Views** and **Collection Views**).

First, because an employee might be a manager, we've used a `NSArray` (specifically a `NSMutableArray` so the values can be modified) to allow the employees that they managed to be attached to them:

```

private NSMutableArray _people = new NSMutableArray();
...

[Export("personModelArray")]
public NSArray People {
    get { return _people; }
}

```

Two things to note here:

1. We used a `NSMutableArray` instead of a standard C# array or collection since this is a requirement to Data Bind to AppKit controls such as **Table Views**, **Outline Views** and **Collections**.
2. We exposed the array of employees by casting it to a `NSArray` for data binding purposes and changed its C# formatted name, `People`, to one that data binding expects, `personModelArray` in the form `{class_name}Array` (note that the first character has been made lower case).

Next, we need to add some specially named public methods to support **Array Controllers** and **Tree Controllers**:

```
[Export("addObject")]
public void AddPerson(PersonModel person) {
    WillChangeValue ("personModelArray");
    isManager = true;
    _people.Add (person);
    DidChangeValue ("personModelArray");
}

[Export("insertObject:inPersonModelArrayAtIndex")]
public void InsertPerson(PersonModel person, nint index) {
    WillChangeValue ("personModelArray");
    _people.Insert (person, index);
    DidChangeValue ("personModelArray");
}

[Export("removeObjectFromPersonModelArrayAtIndex")]
public void RemovePerson(nint index) {
    WillChangeValue ("personModelArray");
    _people.RemoveObject (index);
    DidChangeValue ("personModelArray");
}

[Export("setPersonModelArray")]
public void SetPeople(NSMutableArray array) {
    WillChangeValue ("personModelArray");
    _people = array;
    DidChangeValue ("personModelArray");
}
```

These allow the controllers to request and modify the data that they display. Like the exposed `NSArray` above, these have a very specific naming convention (that differs from the typical C# naming conventions):

- `addObject:` - Adds an object to the array.
- `insertObject:in{class_name}ArrayAtIndex:` - Where `{class_name}` is the name of your class. This method inserts an object into the array at a given index.
- `removeObjectFrom{class_name}ArrayAtIndex:` - Where `{class_name}` is the name of your class. This method removes the object in the array at a given index.
- `set{class_name}Array:` - Where `{class_name}` is the name of your class. This method allows you to replace the existing array with a new one.

Inside of these methods, we've wrapped changes to the array in `WillChangeValue` and `DidChangeValue` messages for KVO compliance.

Finally, since the `Icon` property relies on the value of the `isManager` property, changes to the `isManager` property might not be reflected in the `Icon` for Data Bound UI elements (during KVO):

```
[Export("Icon")]
public NSImage Icon {
    get {
        if (isManager) {
            return NSImage.ImageNamed ("group.png");
        } else {
            return NSImage.ImageNamed ("user.png");
        }
    }
}
```

To correct that, we use the following code:

```
[Export("isManager")]
public bool isManager {
    get { return _isManager; }
    set {
        WillChangeValue ("isManager");
        WillChangeValue ("Icon");
        _isManager = value;
        DidChangeValue ("isManager");
        DidChangeValue ("Icon");
    }
}
```

Note that in addition to its own Key, the `isManager` accessor is also sending the `WillChangeValue` and `DidChangeValue` messages for the `Icon` Key so it will see the change as well.

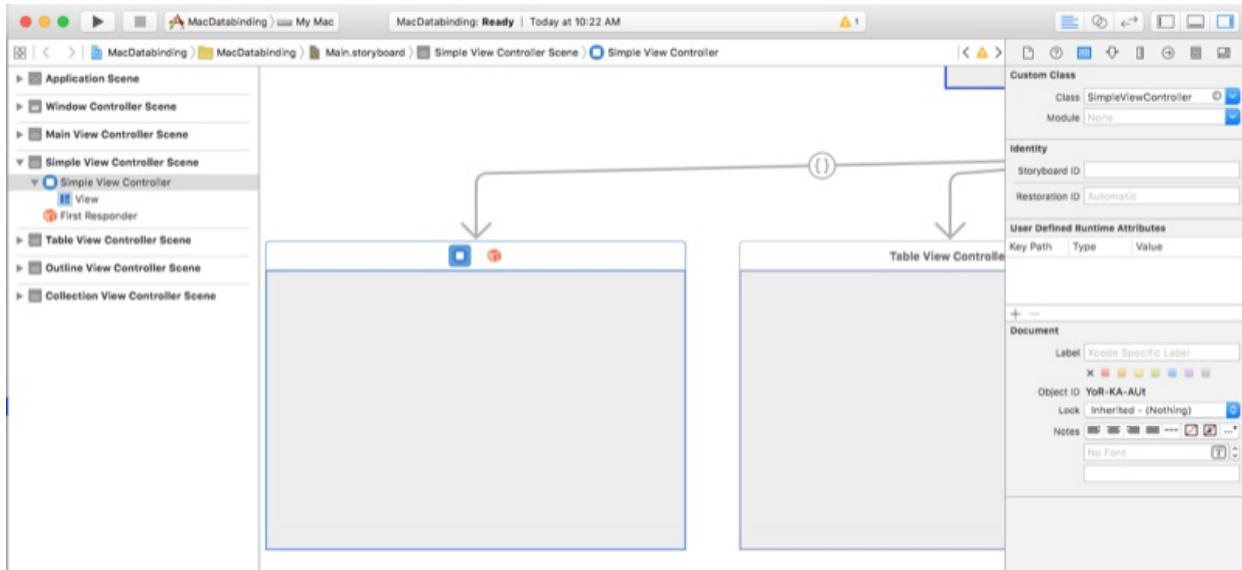
We'll be using the `PersonModel` Data Model throughout the rest of this article.

Simple data binding

With our Data Model defined, let's look at a simple example of data binding in Xcode's Interface Builder. For example, let's add a form to our Xamarin.Mac application that can be used to edit the `PersonModel` that we defined above. We'll add a few Text Fields and a Check Box to display and edit properties of our model.

First, let's add a new **View Controller** to our `Main.storyboard` file in Interface Builder and name its class

`SimpleViewController`:



Next, return to Visual Studio for Mac, edit the `SimpleViewController.cs` file (that was automatically added to our project) and expose an instance of the `PersonModel` that we will be data binding our form to. Add the following code:

```

private PersonModel _person = new PersonModel();
...

[Export("Person")]
public PersonModel Person {
    get {return _person; }
    set {
        WillChangeValue ("Person");
        _person = value;
        DidChangeValue ("Person");
    }
}

```

Next when the View is loaded, let's create an instance of our `PersonModel` and populate it with this code:

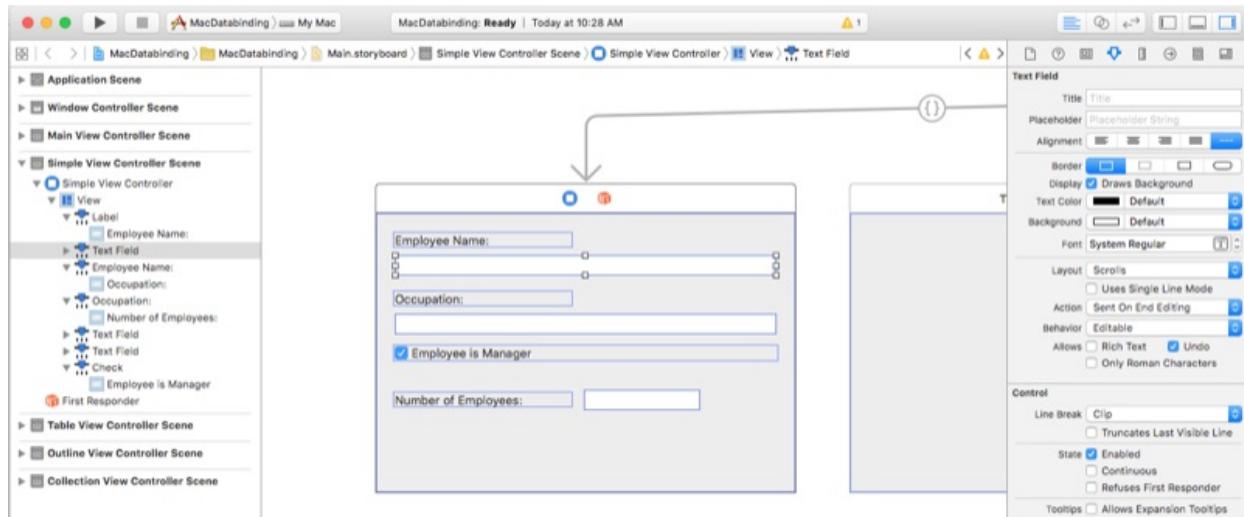
```

public override void ViewDidLoad ()
{
    base.AwakeFromNib ();

    // Set a default person
    var Craig = new PersonModel ("Craig Dunn", "Documentation Manager");
    Craig.AddPerson (new PersonModel ("Amy Burns", "Technical Writer"));
    Craig.AddPerson (new PersonModel ("Joel Martinez", "Web & Infrastructure"));
    Craig.AddPerson (new PersonModel ("Kevin Mullins", "Technical Writer"));
    Craig.AddPerson (new PersonModel ("Mark McLemore", "Technical Writer"));
    Craig.AddPerson (new PersonModel ("Tom Opgenorth", "Technical Writer"));
    Person = Craig;
}

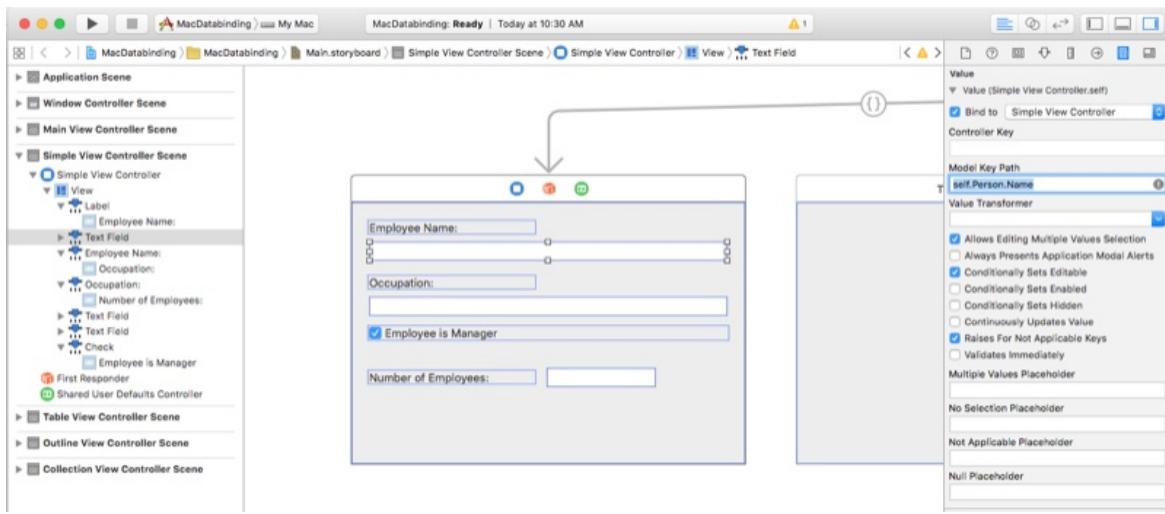
```

Now we need to create our form, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Layout the form to look something like the following:

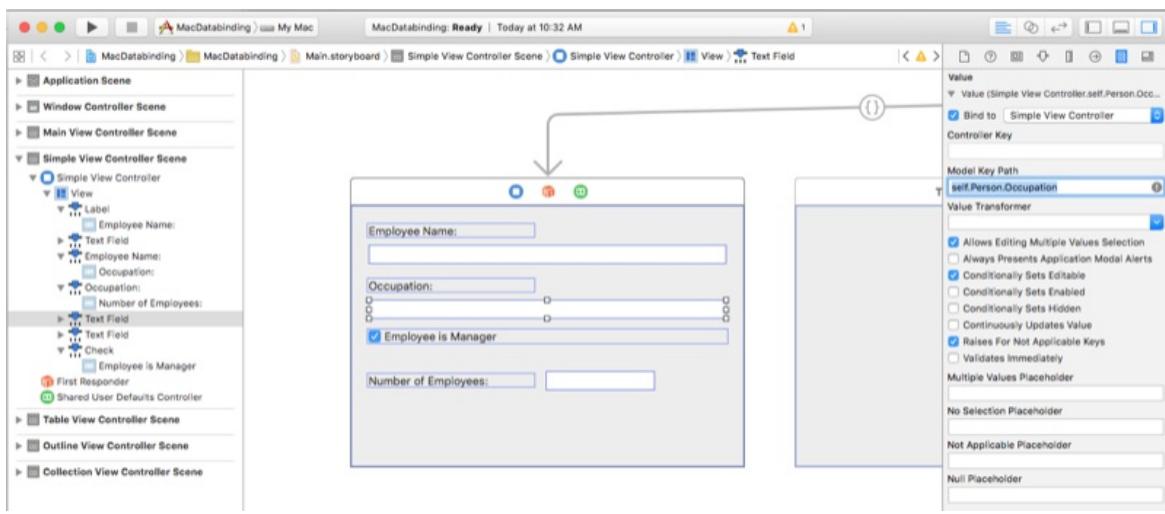


To Data Bind the form to the `PersonModel` that we exposed via the `Person` Key, do the following:

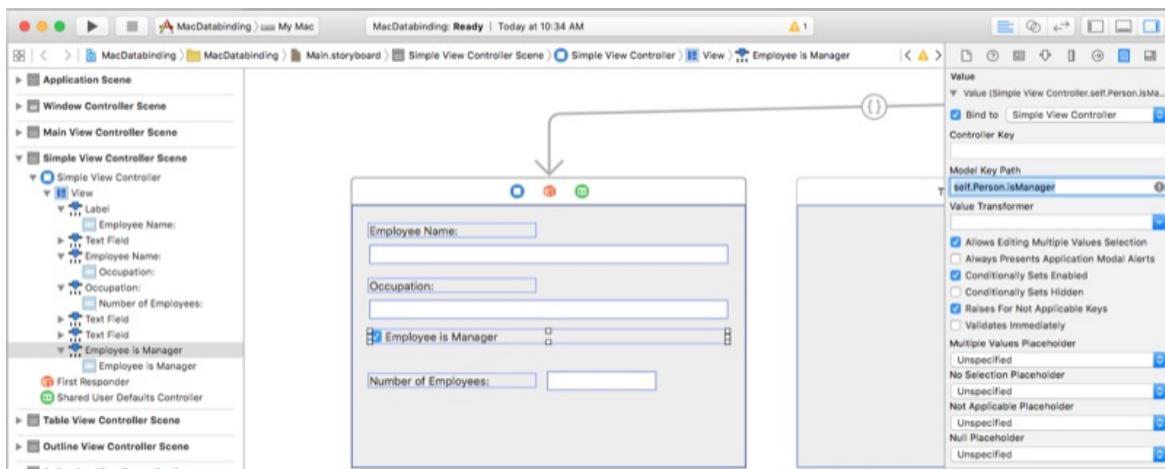
1. Select the **Employee Name** Text Field and switch to the **Bindings Inspector**.
2. Check the **Bind to** box and select **Simple View Controller** from the dropdown. Next enter `self.Person.Name` for the **Key Path**:



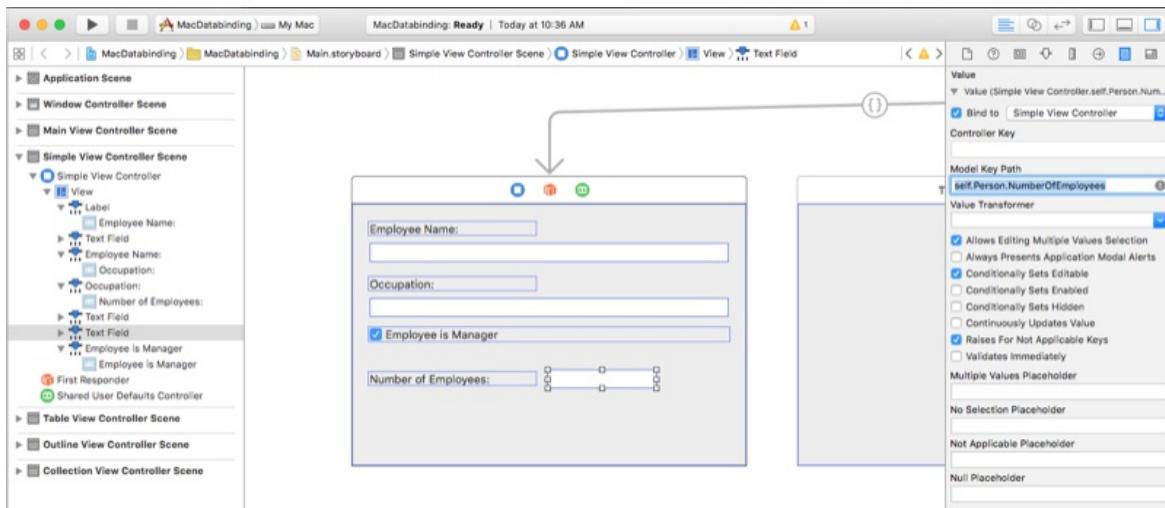
3. Select the Occupation Text Field and check the Bind to box and select Simple View Controller from the dropdown. Next enter `self.Person.Occupation` for the Key Path:



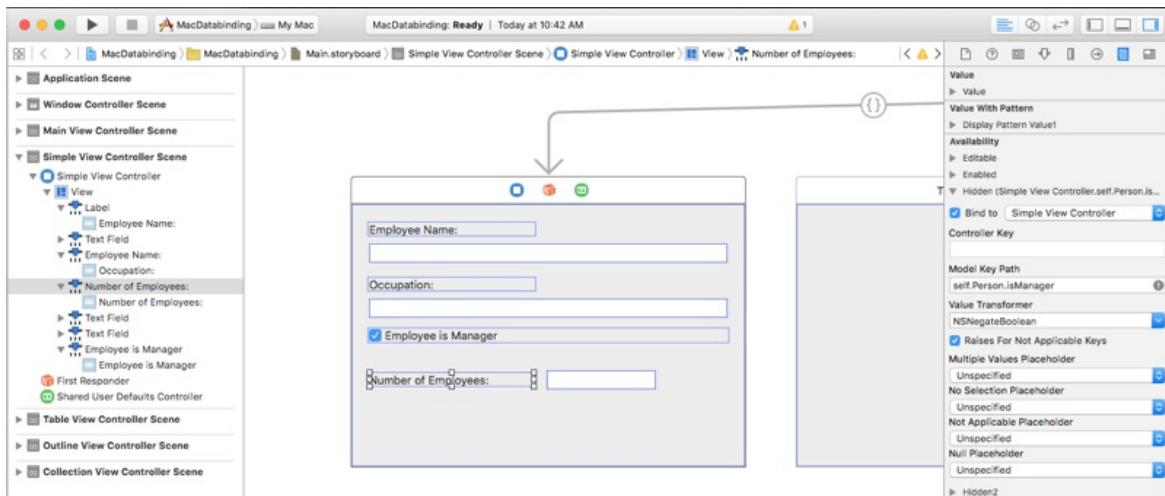
4. Select the Employee is a Manager Checkbox and check the Bind to box and select Simple View Controller from the dropdown. Next enter `self.Person.isManager` for the Key Path:



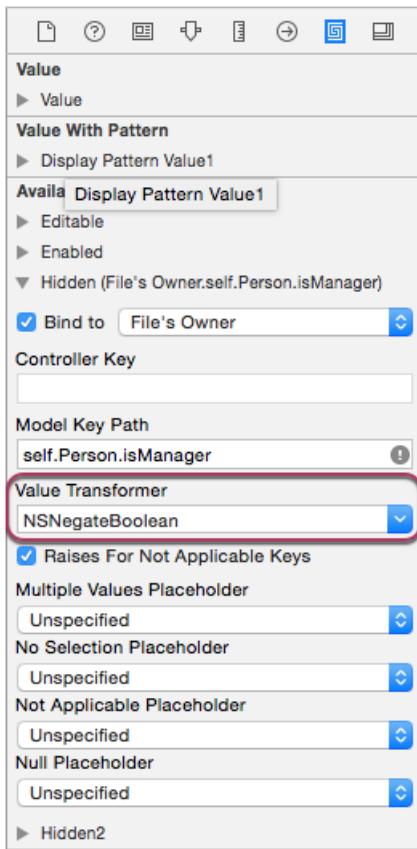
5. Select the Number of Employees Managed Text Field and check the Bind to box and select Simple View Controller from the dropdown. Next enter `self.Person.NumberOfEmployees` for the Key Path:



6. If the employee is not a manager, we want to hide the Number of Employees Managed Label and Text Field.
7. Select the **Number of Employees Managed** Label, expand the **Hidden** turndown and check the **Bind to** box and select **Simple View Controller** from the dropdown. Next enter `self.Person.isManager` for the **Key Path**:



8. Select `NSNegateBoolean` from the **Value Transformer** dropdown:

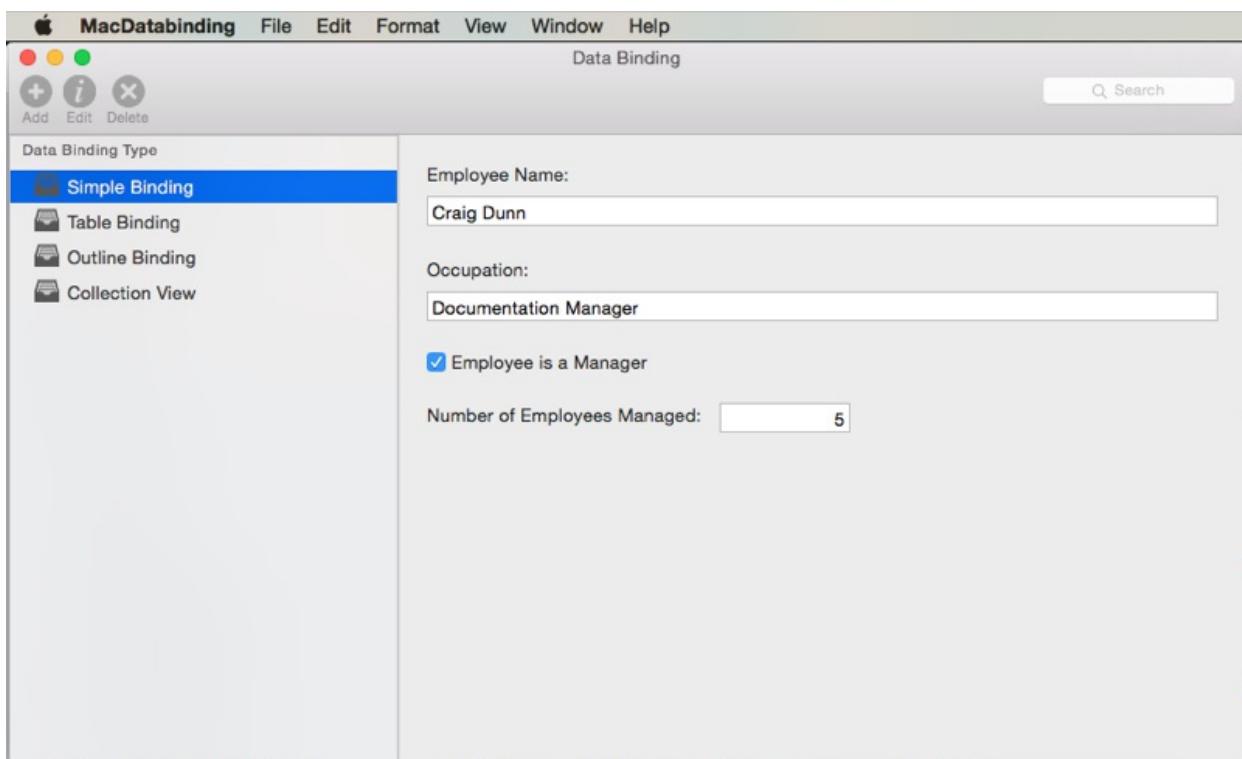


9. This tells data binding that the label will be hidden if the value of the `isManager` property is `false`.

10. Repeat steps 7 and 8 for the **Number of Employees Managed** Text Field.

11. Save your changes and return to Visual Studio for Mac to sync with Xcode.

If you run the application, the values from the `Person` property will automatically populate our form:



Any changes that the users makes to the form will be written back to the `Person` property in the View Controller. For example, unselecting **Employee is a Manager** updates the `Person` instance of our `PersonModel` and the **Number of Employees Managed** Label and Text Field are hidden automatically (via data binding):

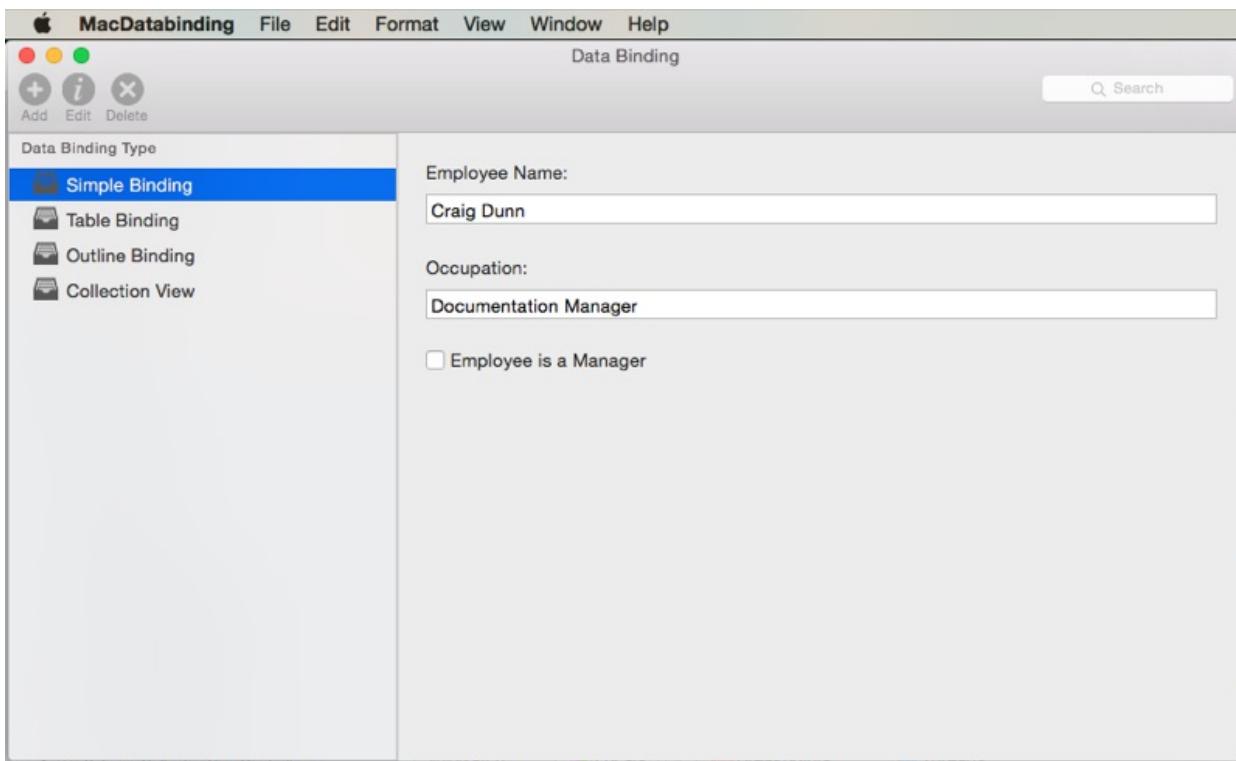
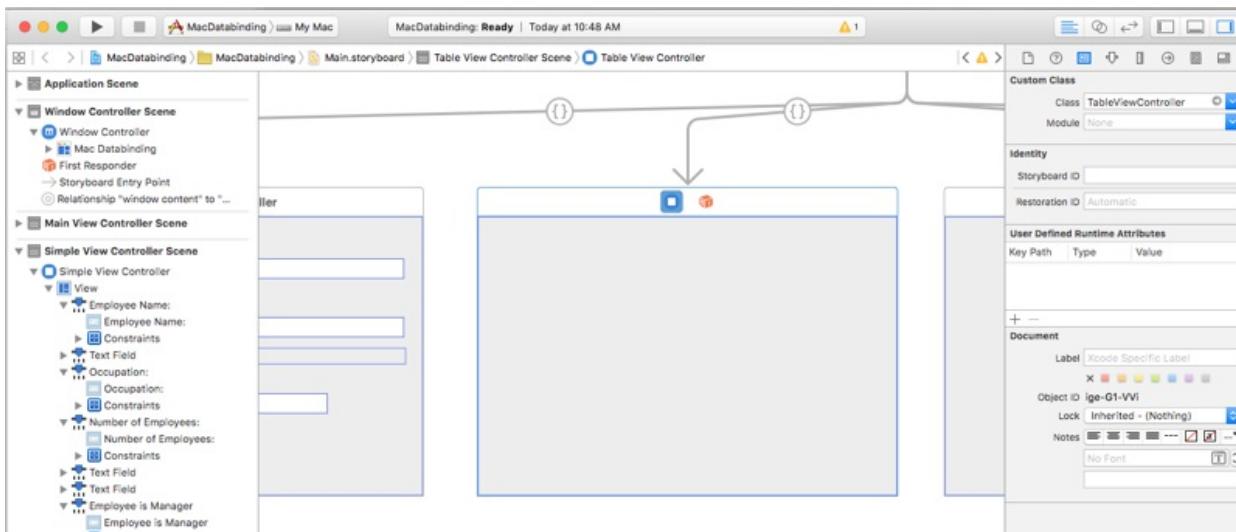


Table view data binding

Now that we have the basics of data binding out of the way, let's look at a more complex data binding task by using an *Array Controller* and data binding to a Table View. For more information on working with Table Views, please see our [Table Views](#) documentation.

First, let's add a new **View Controller** to our **Main.storyboard** file in Interface Builder and name its class

`TableViewController`:



Next, let's edit the **TableViewController.cs** file (that was automatically added to our project) and expose an array (`NSArray`) of `PersonModel` classes that we will be data binding our form to. Add the following code:

```

private NSMutableArray _people = new NSMutableArray();
...
[Export("personModelArray")]
public NSArray People {
    get { return _people; }
}
...
[Export("addObject:")]
public void AddPerson(PersonModel person) {
    WillChangeValue ("personModelArray");
    _people.Add (person);
    DidChangeValue ("personModelArray");
}

[Export("insertObject:inPersonModelArrayAtIndex:")]
public void InsertPerson(PersonModel person, nint index) {
    WillChangeValue ("personModelArray");
    _people.Insert (person, index);
    DidChangeValue ("personModelArray");
}

[Export("removeObjectFromPersonModelArrayAtIndex:")]
public void RemovePerson(nint index) {
    WillChangeValue ("personModelArray");
    _people.RemoveObject (index);
    DidChangeValue ("personModelArray");
}

[Export("setPersonModelArray:")]
public void SetPeople(NSMutableArray array) {
    WillChangeValue ("personModelArray");
    _people = array;
    DidChangeValue ("personModelArray");
}

```

Just like we did on the `PersonModel` class above in the [Defining your Data Model](#) section, we've exposed four specially named public methods so that the Array Controller and read and write data from our collection of `PersonModels`.

Next when the View is loaded, we need to populate our array with this code:

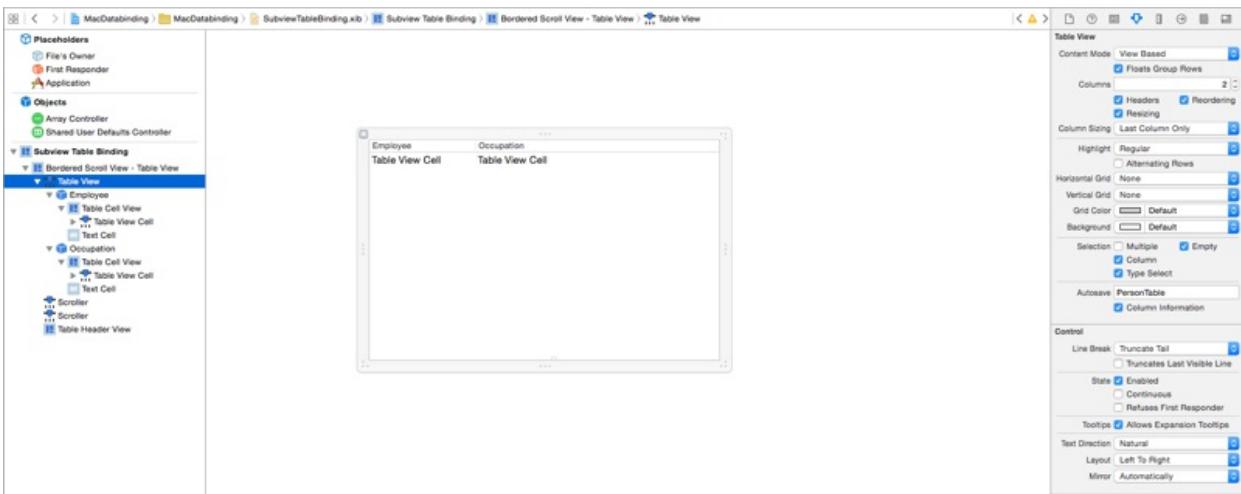
```

public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

    // Build list of employees
    AddPerson (new PersonModel ("Craig Dunn", "Documentation Manager", true));
    AddPerson (new PersonModel ("Amy Burns", "Technical Writer"));
    AddPerson (new PersonModel ("Joel Martinez", "Web & Infrastructure"));
    AddPerson (new PersonModel ("Kevin Mullins", "Technical Writer"));
    AddPerson (new PersonModel ("Mark McLemore", "Technical Writer"));
    AddPerson (new PersonModel ("Tom Opgenorth", "Technical Writer"));
    AddPerson (new PersonModel ("Larry O'Brien", "API Documentation Manager", true));
    AddPerson (new PersonModel ("Mike Norman", "API Documenter"));
}

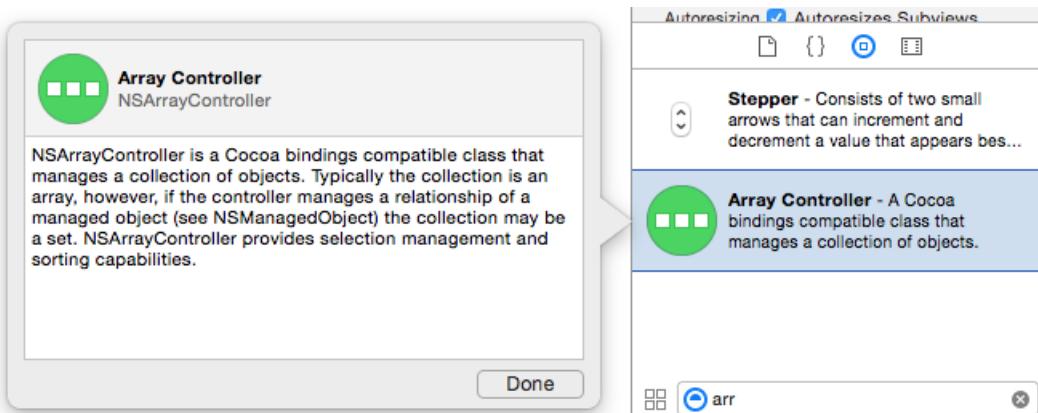
```

Now we need to create our Table View, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Layout the table to look something like the following:

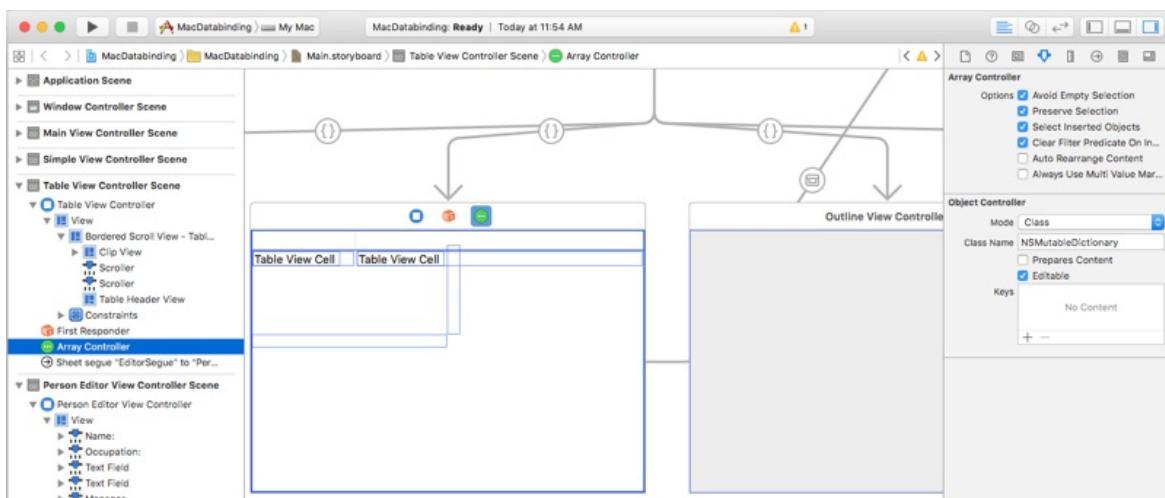


We need to add an **Array Controller** to provide bound data to our table, do the following:

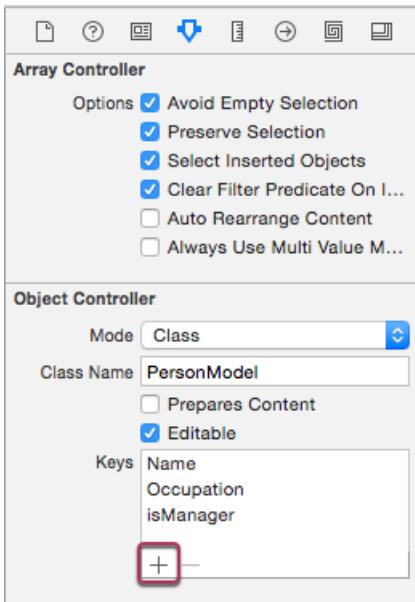
1. Drag an **Array Controller** from the Library Inspector onto the Interface Editor:



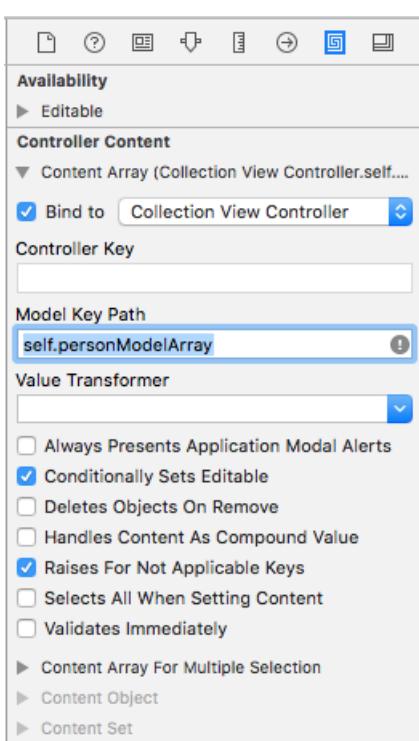
2. Select **Array Controller** in the Interface Hierarchy and switch to the Attribute Inspector:



3. Enter `PersonModel` for the **Class Name**, click the Plus button and add three Keys. Name them `Name`, `Occupation` and `isManager`:



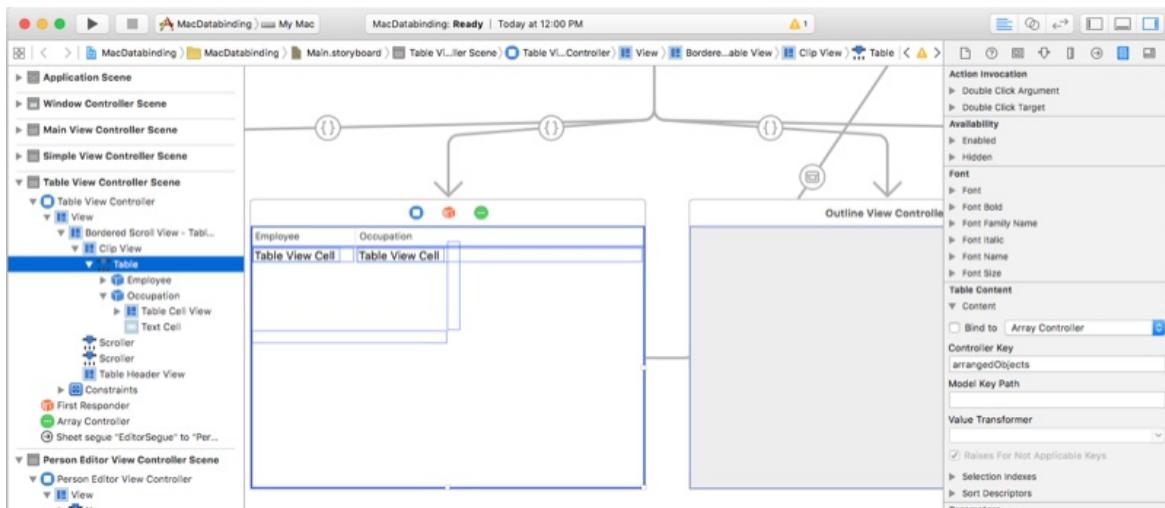
4. This tells the Array Controller what it is managing an array of, and which properties it should expose (via Keys).
5. Switch to the **Bindings Inspector** and under **Content Array** select **Bind to** and **Table View Controller**. Enter a Model Key Path of `self.personModelArray`:



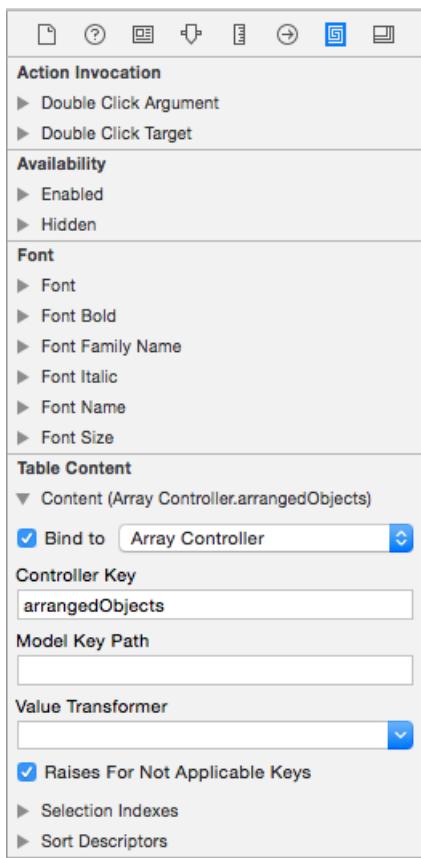
6. This ties the Array Controller to the array of `PersonModels` that we exposed on our View Controller.

Now we need to bind our Table View to the Array Controller, do the following:

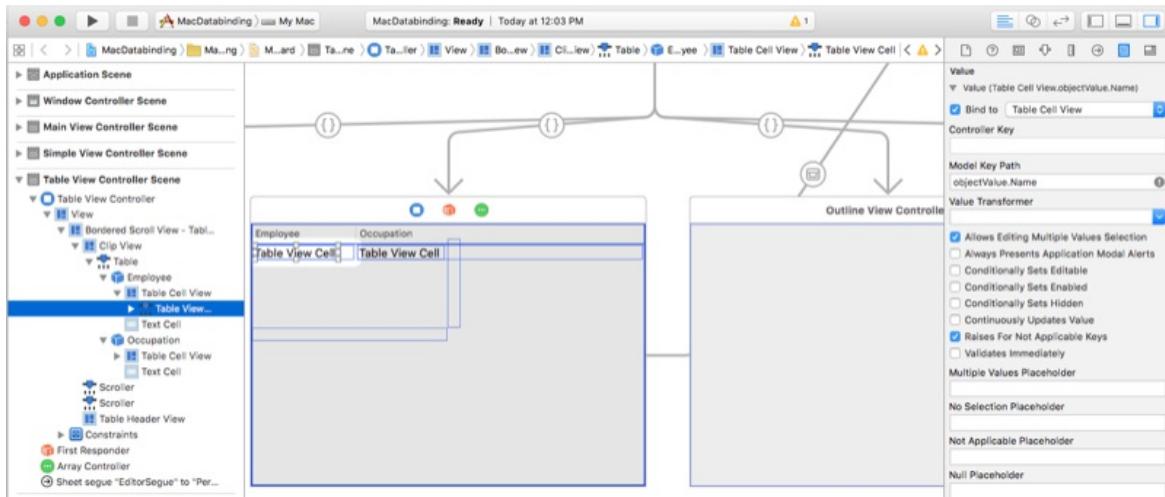
1. Select the Table View and the **Binding Inspector**:



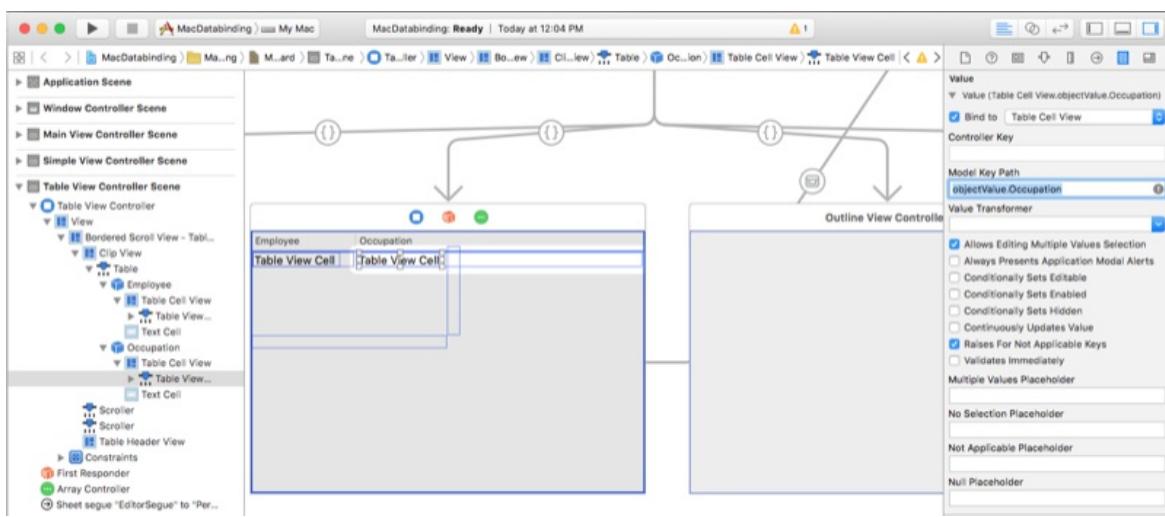
2. Under the **Table Contents** turndown, select **Bind to** and **Array Controller**. Enter `arrangedObjects` for the **Controller Key** field:



3. Select the **Table View Cell** under the **Employee** column. In the **Bindings Inspector** under the **Value** turndown, select **Bind to** and **Table Cell View**. Enter `objectValue.Name` for the **Model Key Path**:



4. `objectValue` is the current `PersonModel` in the array being managed by the Array Controller.
5. Select the **Table View Cell** under the **Occupation** column. In the **Bindings Inspector** under the **Value** dropdown, select **Bind to** and **Table Cell View**. Enter `objectValue.Occupation` for the **Model Key Path**:



6. Save your changes and return to Visual Studio for Mac to sync with Xcode.

If we run the application, the table will be populated with our array of `PersonModels`:

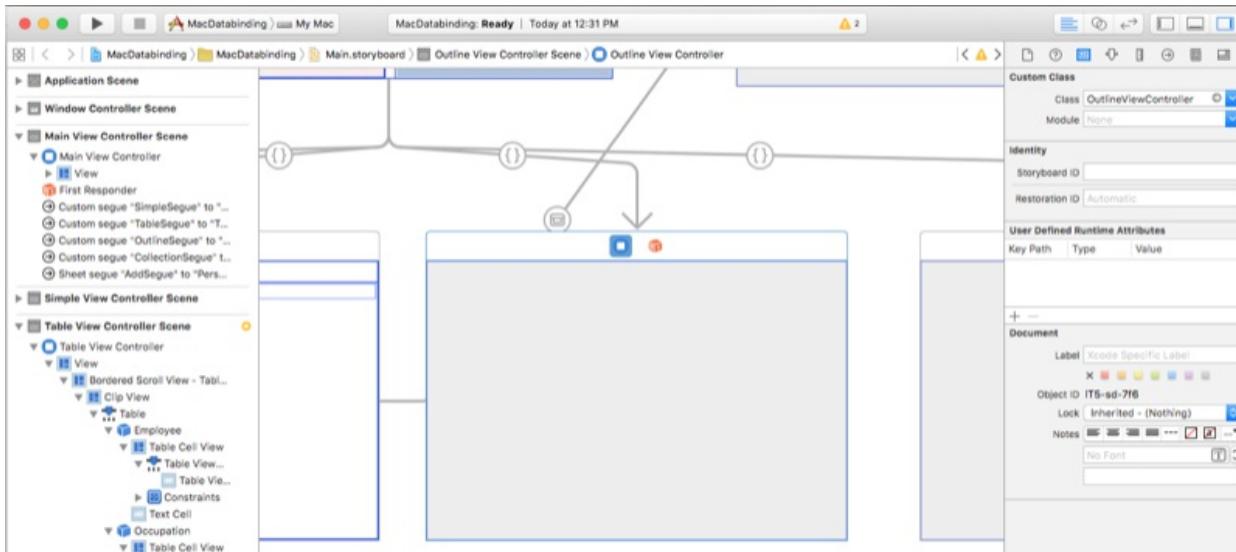
Employee	Occupation
Craig Dunn	Documentation Manager
Amy Burns	Technical Writer
Joel Martinez	Web & Infrastructure
Kevin Mullins	Technical Writer
Mark McLemore	Technical Writer
Tom Opgenorth	Technical Writer
Larry O'Brien	API Documentation Manager
Mike Norman	API Documentor

Outline view data binding

Data binding against an Outline View is very similar to binding against a Table View. The key difference is that we'll be using a **Tree Controller** instead of an **Array Controller** to provide the bound data to the Outline View. For more information on working with Outline Views, please see our [Outline Views](#) documentation.

First, let's add a new **View Controller** to our **Main.storyboard** file in Interface Builder and name its class

`OutlineViewController`:



Next, let's edit the `OutlineViewController.cs` file (that was automatically added to our project) and expose an array (`NSArray`) of `PersonModel` classes that we will be data binding our form to. Add the following code:

```

private NSMutableArray _people = new NSMutableArray();
...
[Export("personModelArray")]
public NSArray People {
    get { return _people; }
}
...
[Export("addObject:")]
public void AddPerson(PersonModel person) {
    WillChangeValue ("personModelArray");
    _people.Add (person);
    DidChangeValue ("personModelArray");
}

[Export("insertObject:inPersonModelArrayAtIndex:")]
public void InsertPerson(PersonModel person, nint index) {
    WillChangeValue ("personModelArray");
    _people.Insert (person, index);
    DidChangeValue ("personModelArray");
}

[Export("removeObjectFromPersonModelArrayAtIndex:")]
public void RemovePerson(nint index) {
    WillChangeValue ("personModelArray");
    _people.RemoveObject (index);
    DidChangeValue ("personModelArray");
}

[Export("setPersonModelArray:")]
public void SetPeople(NSMutableArray array) {
    WillChangeValue ("personModelArray");
    _people = array;
    DidChangeValue ("personModelArray");
}

```

Just like we did on the `PersonModel` class above in the [Defining your Data Model](#) section, we've exposed four specially named public methods so that the Tree Controller and read and write data from our collection of `PersonModels`.

Next when the View is loaded, we need to populate our array with this code:

```

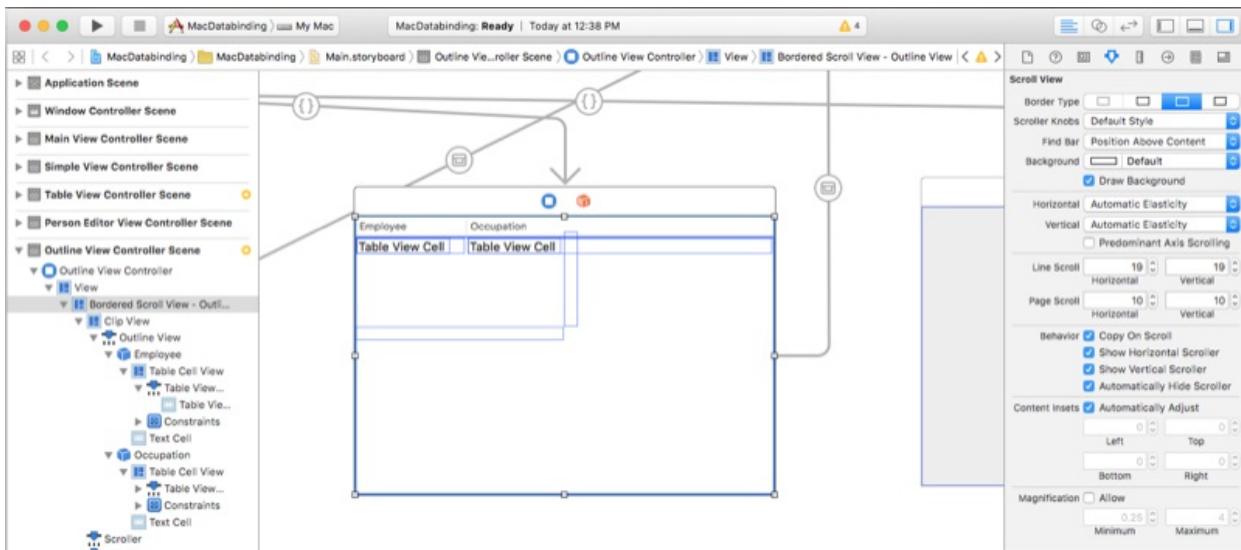
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

    // Build list of employees
    var Craig = new PersonModel ("Craig Dunn", "Documentation Manager");
    Craig.AddPerson (new PersonModel ("Amy Burns", "Technical Writer"));
    Craig.AddPerson (new PersonModel ("Joel Martinez", "Web & Infrastructure"));
    Craig.AddPerson (new PersonModel ("Kevin Mullins", "Technical Writer"));
    Craig.AddPerson (new PersonModel ("Mark McLemore", "Technical Writer"));
    Craig.AddPerson (new PersonModel ("Tom Opgenorth", "Technical Writer"));
    AddPerson (Craig);

    var Larry = new PersonModel ("Larry O'Brien", "API Documentation Manager");
    Larry.AddPerson (new PersonModel ("Mike Norman", "API Documenter"));
    AddPerson (Larry);
}

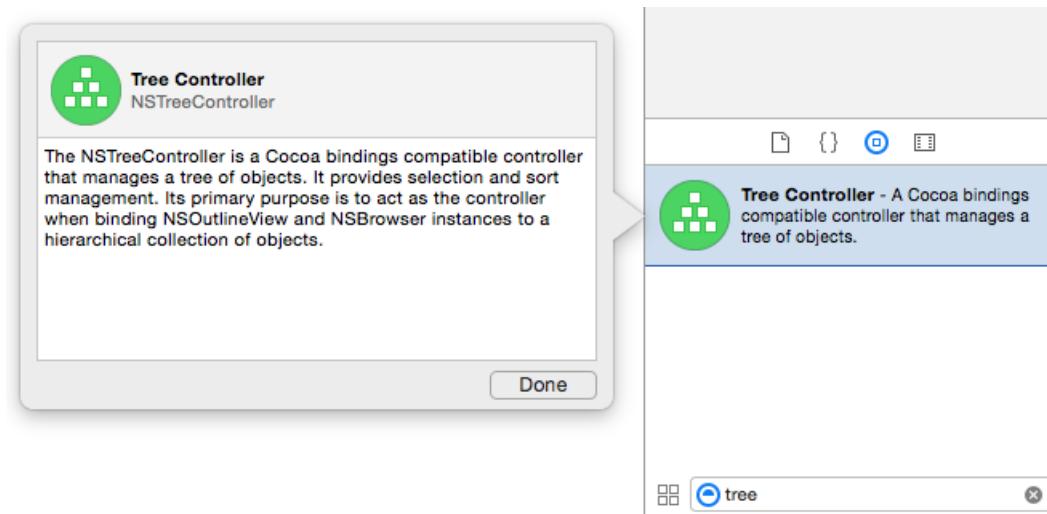
```

Now we need to create our Outline View, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Layout the table to look something like the following:

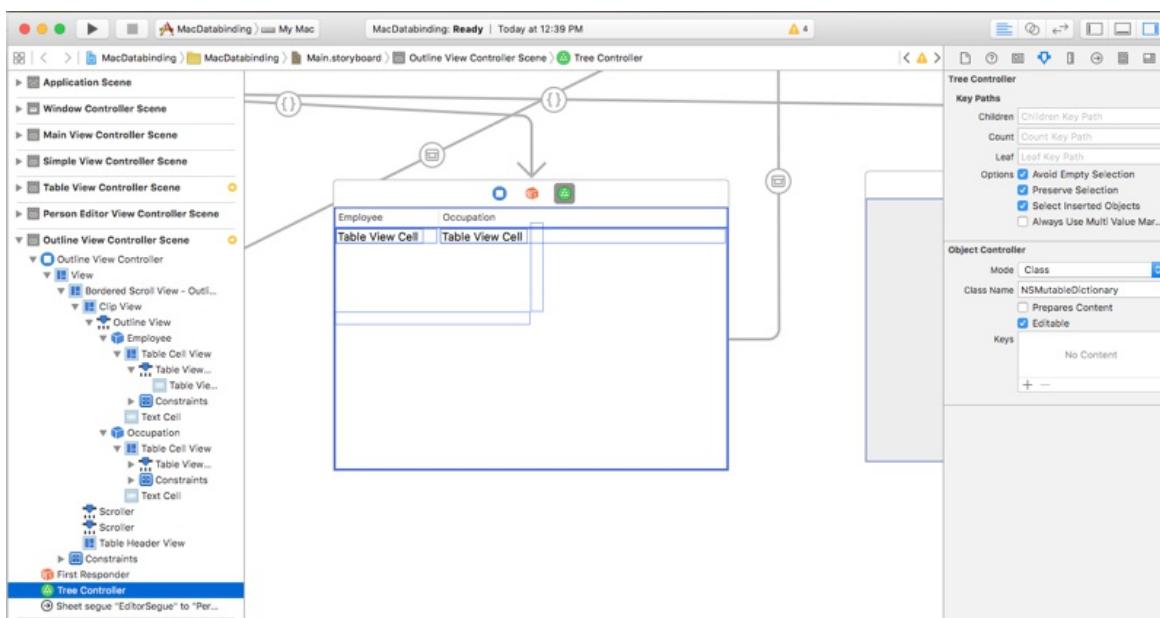


We need to add an **Tree Controller** to provide bound data to our outline, do the following:

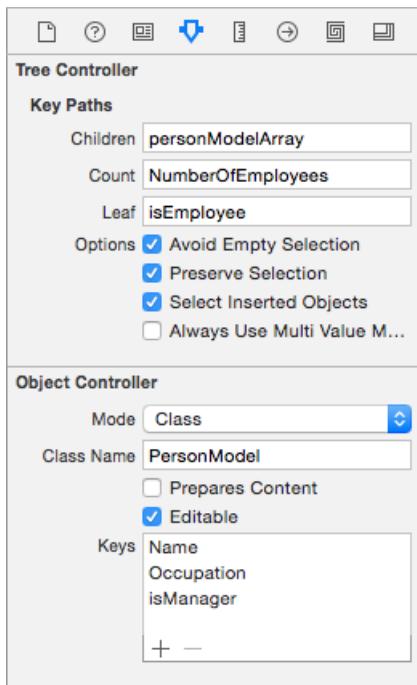
1. Drag an **Tree Controller** from the Library Inspector onto the Interface Editor:



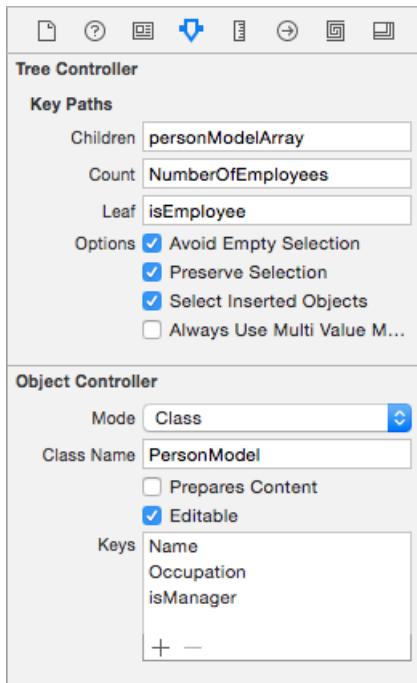
2. Select **Tree Controller** in the Interface Hierarchy and switch to the Attribute Inspector:



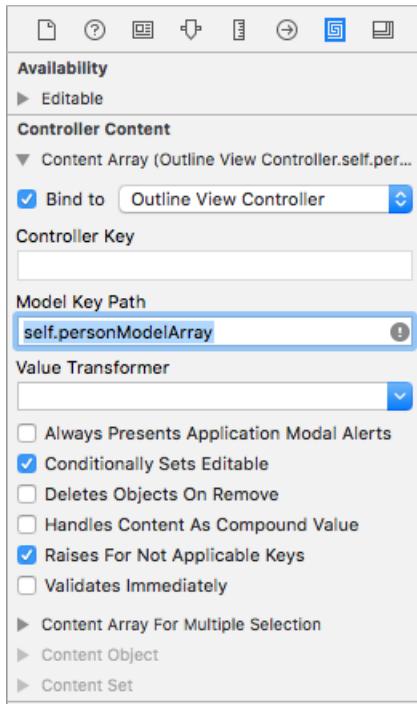
3. Enter **PersonModel** for the **Class Name**, click the Plus button and add three Keys. Name them **Name**, **Occupation** and **isManager**:



4. This tells the Tree Controller what it is managing an array of, and which properties it should expose (via Keys).
5. Under the **Tree Controller** section, enter `personModelArray` for **Children**, enter `NumberOfEmployees` under the **Count** and enter `isEmployee` under **Leaf**:



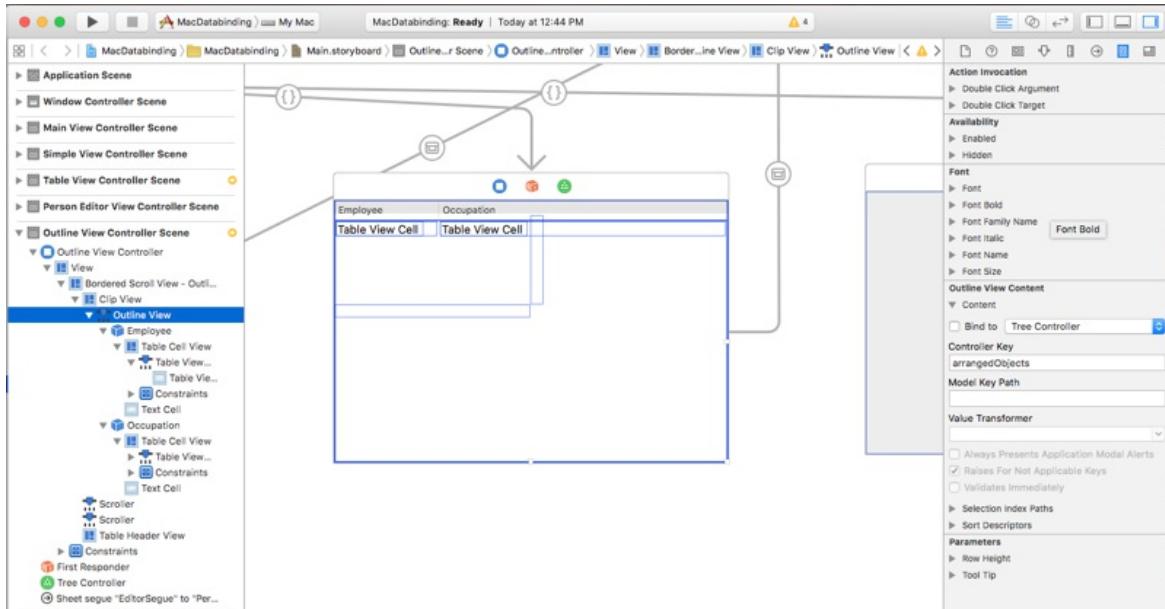
6. This tells the Tree Controller where to find any child nodes, how many child nodes there are and if the current node has child nodes.
7. Switch to the **Bindings Inspector** and under **Content Array** select **Bind to** and **File's Owner**. Enter a **Model Key Path** of `self.personModelArray` :



8. This ties the Tree Controller to the array of `PersonModels` that we exposed on our View Controller.

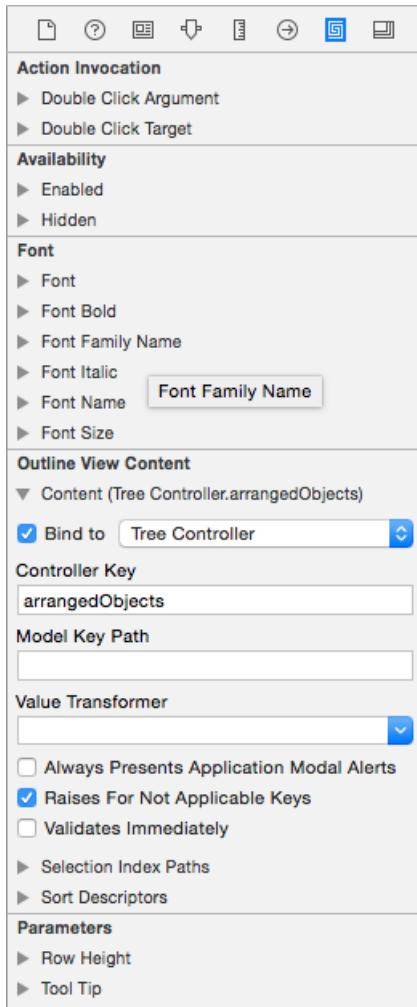
Now we need to bind our Outline View to the Tree Controller, do the following:

1. Select the Outline View and in the Binding Inspector select :

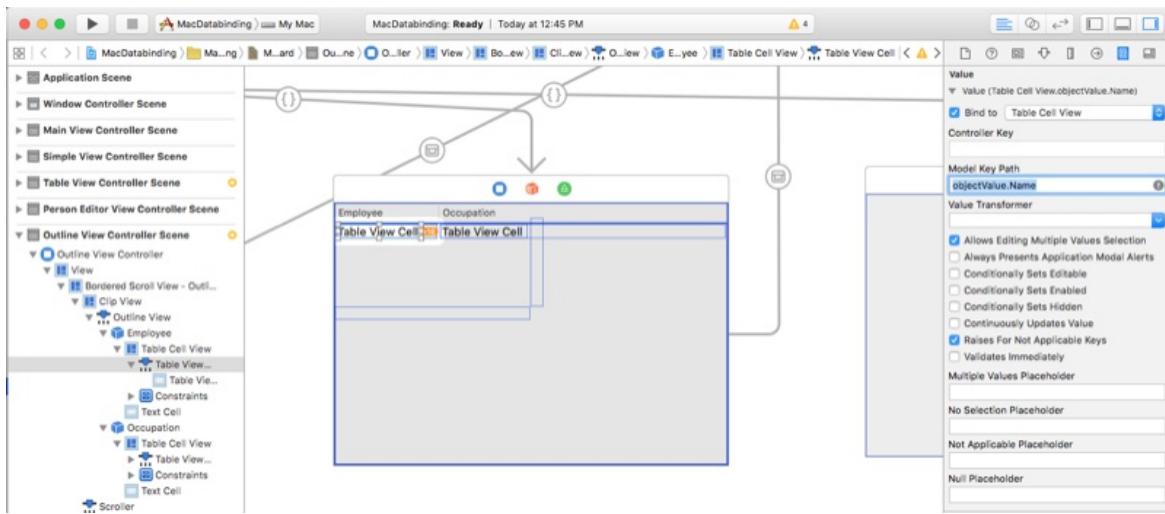


2. Under the Outline View Contents turndown, select Bind to and Tree Controller. Enter

`arrangedObjects` for the Controller Key field:

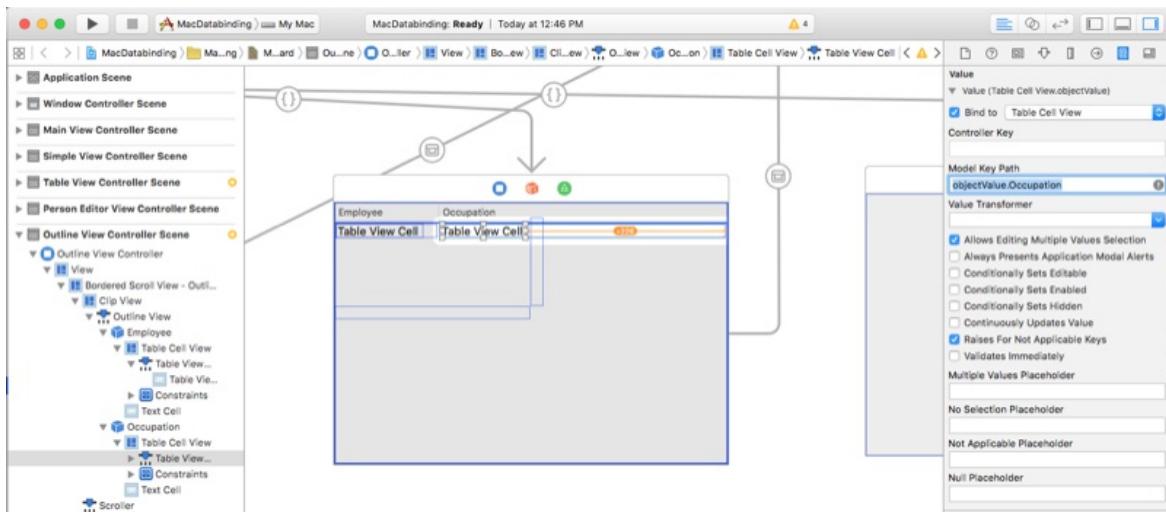


3. Select the **Table View Cell** under the **Employee** column. In the **Bindings Inspector** under the **Value** turndown, select **Bind to** and **Table Cell View**. Enter `objectValue.Name` for the **Model Key Path**:



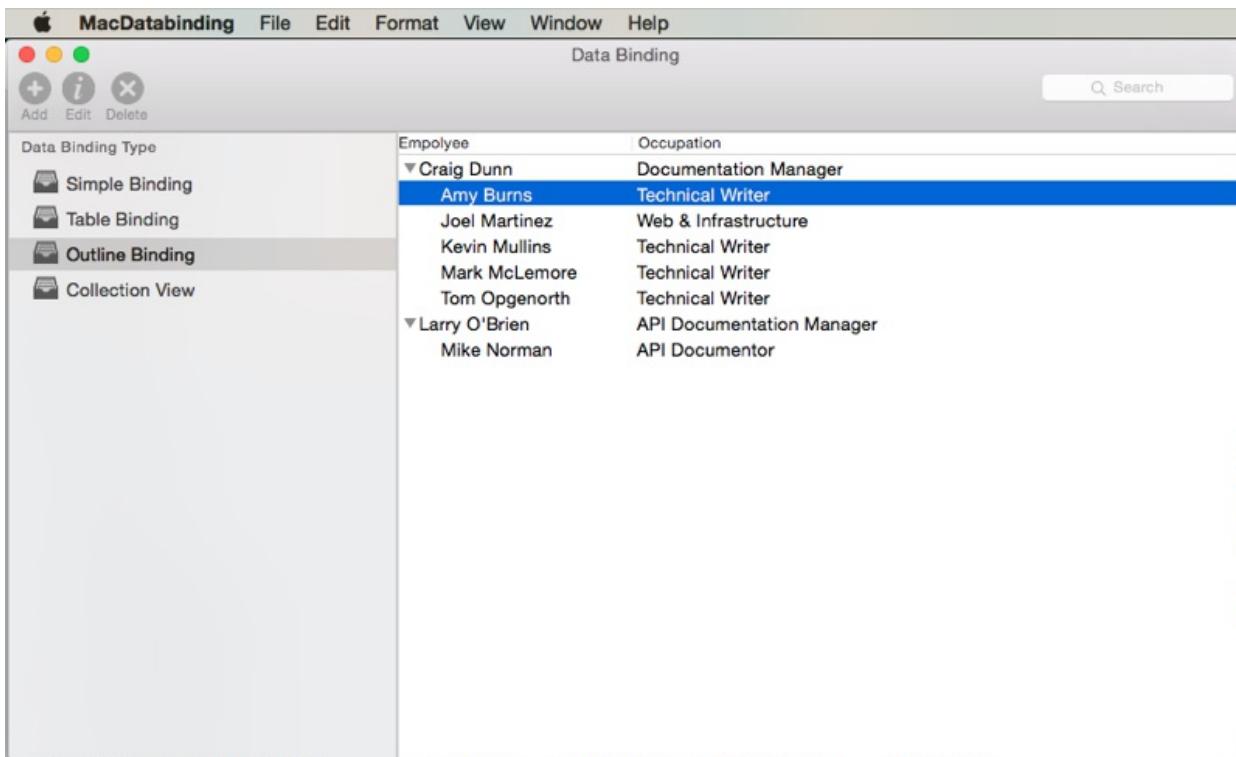
4. `objectValue` is the current `PersonModel1` in the array being managed by the Tree Controller.

5. Select the **Table View Cell** under the **Occupation** column. In the **Bindings Inspector** under the **Value** turndown, select **Bind to** and **Table Cell View**. Enter `objectValue.Occupation` for the **Model Key Path**:



- Save your changes and return to Visual Studio for Mac to sync with Xcode.

If we run the application, the outline will be populated with our array of `PersonModels`:



Collection view data binding

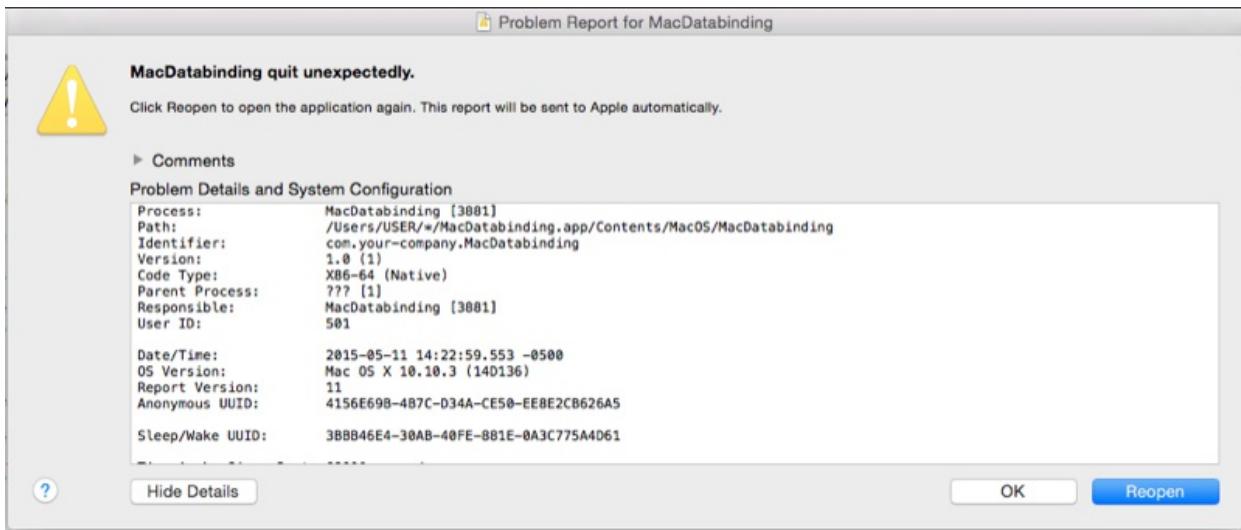
Data binding with a Collection View is very much like binding with a Table View, as an Array Controller is used to provide data for the collection. Since the collection view doesn't have a preset display format, more work is required to provide user interaction feedback and to track user selection.

IMPORTANT

Due to an issue in Xcode 7 and macOS 10.11 (and greater), Collection Views are unable to be used inside of a Storyboard (.storyboard) files. As a result, you will need to continue to use .xib files to define your Collection Views for your Xamarin.Mac apps. Please see our [Collection Views](#) documentation for more information.

Debugging native crashes

Making a mistake in your data bindings can result in a *Native Crash* in unmanaged code and cause your Xamarin.Mac application to fail completely with a `SIGABRT` error:



There are typically four major causes for native crashes during data binding:

1. Your Data Model does not inherit from `NSObject` or a subclass of `NSObject`.
2. You did not expose your property to Objective-C using the `[Export("key-name")]` attribute.
3. You did not wrap changes to the accessor's value in `WillChangeValue` and `DidChangeValue` method calls (specifying the same Key as the `Export` attribute).
4. You have a wrong or mistyped Key in the **Binding Inspector** in Interface Builder.

Decoding a crash

Let's cause a native crash in our data binding so we can show how to locate and fix it. In Interface Builder, let's change our binding of first Label in the Collection View example from `Name` to `Title`:



Let's save the change, switch back to Visual Studio for Mac to sync with Xcode and run our application. When the Collection View is displayed, the application will momentarily crash with a `SIGABRT` error (as shown in the **Application Output** in Visual Studio for Mac) since the `PersonModel` does not expose a property with the Key `Title`:

The screenshot shows the Kamarin Studio interface with the following details:

- Solution Explorer:** Shows a project named "MacDataBinding" with files like MainWindows.cs, SubviewCollectionView.cs, SubviewCollectionView.cs, SubviewCollectionViewController.cs, and SubviewCollectionView.designer.cs.
- Assembly Browser:** Displays the assembly structure for MacDataBinding.
- Application Output:** Shows a large amount of log output from the application, including:
 - Method entries and exits.
 - Exception handling information, such as stack traces for `ObjectDisposedException`.
 - Runtime events like `MarshalValue`, `MarshalReturn`, and `MarshalReturnValues`.
 - Native API calls and their parameters.
- Search Bar:** Located at the top right of the interface.
- Toolbar:** Includes icons for file operations, search, and other common functions.
- Help:** A link to the documentation.

If we scroll to the very top of the error in the **Application Output** we can see the key to solving the issue:

```
[Application Output]
Loaded assembly: /Users/kmullins/Projects/MacDatabinding/MacDatabinding/bin/Debug/MacDatabinding.app/Contents/MonoBundle/MacDatabinding.exe
Loaded assembly: /Users/kmullins/Projects/MacDatabinding/MacDatabinding/bin/Debug/MacDatabinding.app/Contents/MonoBundle/Xamarin.Mac.dll [External]
Loaded assembly: /Users/kmullins/Projects/MacDatabinding/MacDatabinding/bin/Debug/MacDatabinding.app/Contents/MonoBundle/System.dll [External]
Loaded assembly: /Users/kmullins/Projects/MacDatabinding/MacDatabinding/bin/Debug/MacDatabinding.app/Contents/MonoBundle/System.Net.Http.dll [External]
WARNING: The runtime version supported by this application is unavailable.
Using default runtime: v4.0.30319
2015-05-11 14:22:41.041 MacDatabinding[3881:297650] An uncaught exception was raised
2015-05-11 14:22:41.043 MacDatabinding[3881:297650] [-[PersonModel 0x61000000fd40 valueForUndefinedKey:]: this class is not key value coding-compliant
2015-05-11 14:22:41.044 MacDatabinding[3881:297650] (
    0 CoreFoundation                      0x00007fff8f001a03c __exceptionPreprocess + 172
    1 libobjc.A.dylib                     0x00007fff8fbd476e objc_exception_throw + 43
    2 CoreFoundation                      0x00007fff8f08fa0bd9 -[NSException raise] + 9
```

This line is telling us that the key `Title` doesn't exist on the object that we are binding to. If we change the binding back to `Name` in Interface Builder, save, sync, rebuild and run, the application will run as expected without issue.

Summary

This article has taken a detailed look at working with data binding and key-value coding in a Xamarin.Mac application. First, it looked at exposing a C# class to Objective-C by using key-value coding (KVC) and key-value observing (KVO). Next, it showed how to use a KVO compliant class and Data Bind it to UI elements in Xcode's Interface Builder. Finally, it showed complex data binding using [Array Controllers](#) and [Tree Controllers](#).

Related Links

- MacDatabinding Storyboard (sample)
 - MacDatabinding XIBs (sample)
 - Hello, Mac
 - Standard controls
 - Table views
 - Outline views
 - Collection views
 - Key-Value Coding Programming Guide
 - Introduction to Key-Value Observing Programming Guide
 - Introduction to Cocoa Bindings Programming Topics
 - Introduction to Cocoa Bindings Reference
 - NSCollectionView

- macOS Human Interface Guidelines

Databases in Xamarin.Mac

11/2/2020 • 42 minutes to read • [Edit Online](#)

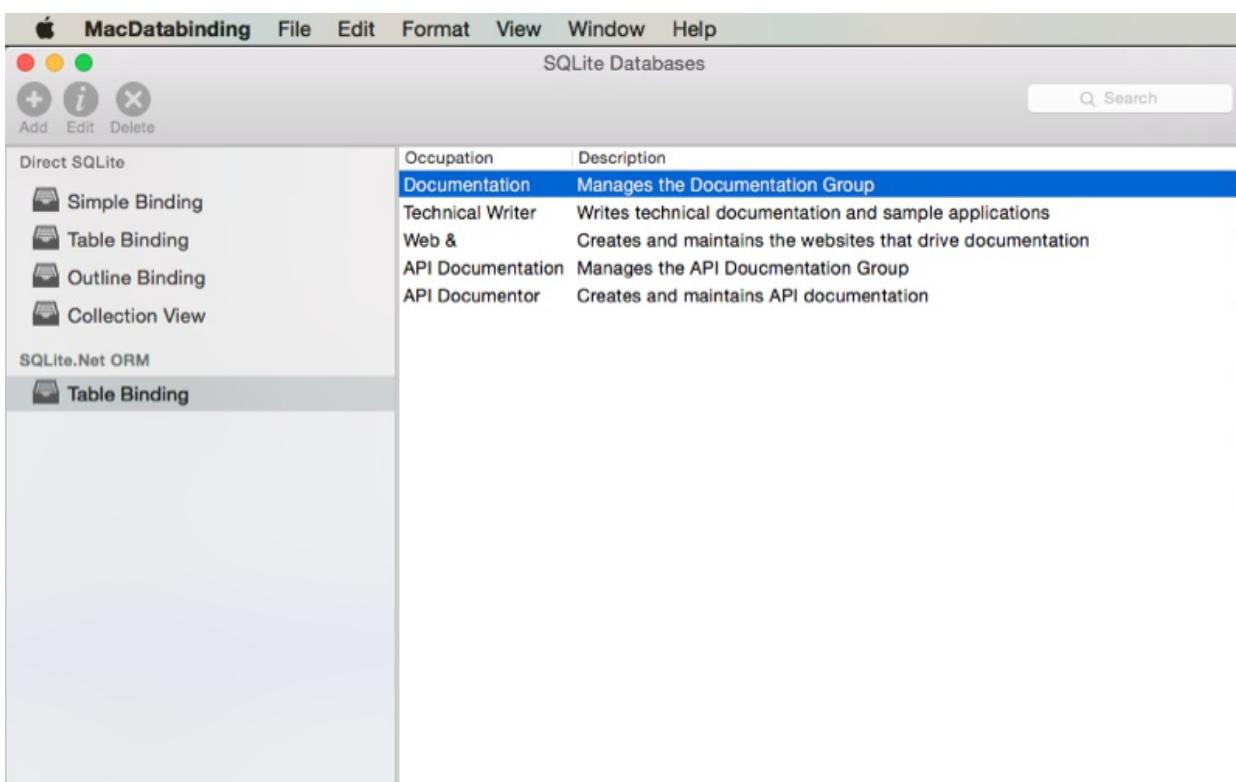
This article covers using key-value coding and key-value observing to allow for data binding between SQLite databases and UI elements in Xcode's Interface Builder. It also covers using the SQLite.NET ORM to provide access to SQLite data.

Overview

When working with C# and .NET in a Xamarin.Mac application, you have access to the same SQLite databases that a Xamarin.iOS or Xamarin.Android application can access.

In this article we will be covering two ways to access SQLite data:

1. **Direct Access** - By directly accessing a SQLite Database, we can use data from the database for key-value coding and data binding with UI elements created in Xcode's Interface Builder. By using key-value coding and data binding techniques in your Xamarin.Mac application, you can greatly decrease the amount of code that you have to write and maintain to populate and work with UI elements. You also have the benefit of further decoupling your backing data (*Data Model*) from your front end User Interface (*Model-View-Controller*), leading to easier to maintain, more flexible application design.
2. **SQLite.NET ORM** - By using the open source [SQLite.NET](#) Object Relationship Manager (ORM) we can greatly reduce the amount of code required to read and write data from a SQLite database. This data can then be used to populate a user interface item such as a Table View.



In this article, we'll cover the basics of working with key-value coding and data binding with SQLite Databases in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

Since we will be using key-value coding and data binding, please work through the [Data binding and key-value](#)

coding first, as core techniques and concepts will be covered that will be used in this documentation and its sample application.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` attributes used to wire up your C# classes to Objective-C objects and UI elements.

Direct SQLite access

For SQLite data that is going to be bound to UI elements in Xcode's Interface Builder, it is highly suggested that you access the SQLite database directly (as opposed to using a technique such as an ORM), since you have total control over the way the data is written and read from the database.

As we have seen in the [Data Binding and Key-Value Coding](#) documentation, by using key-value coding and data binding techniques in your Xamarin.Mac application, you can greatly decrease the amount of code that you have to write and maintain to populate and work with UI elements. When combined with direct access to a SQLite database, it can also greatly reduce the amount of code required to read and write data to that database.

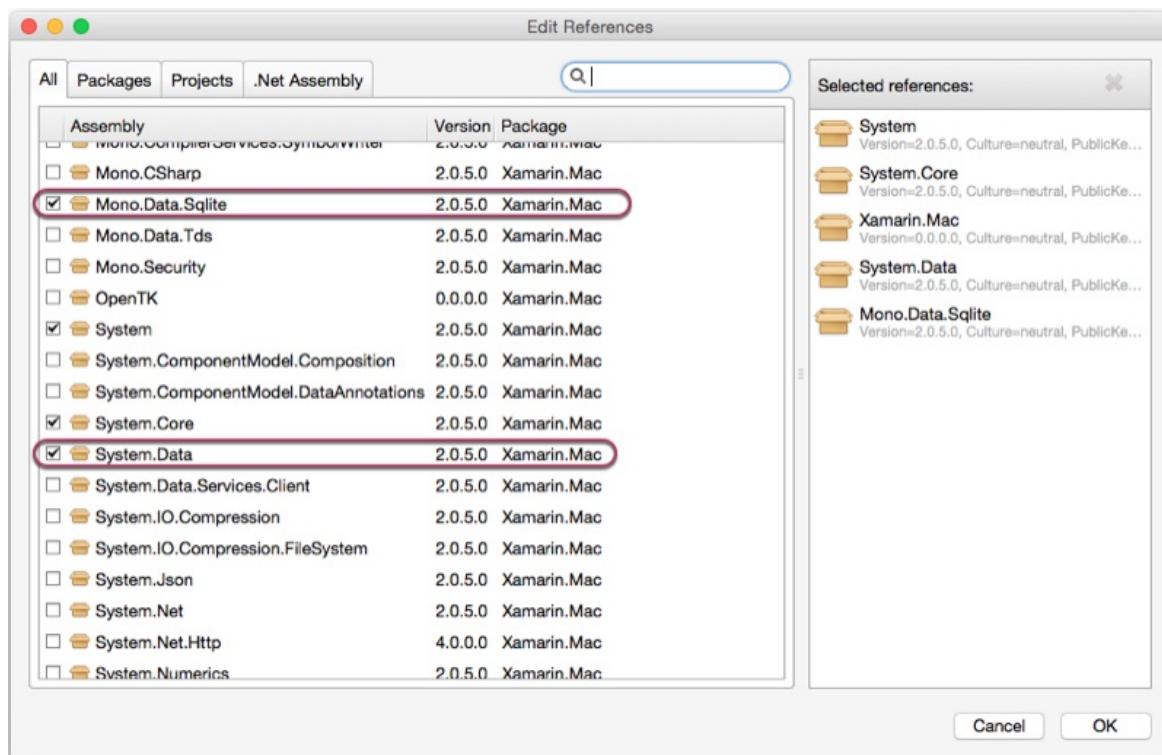
In this article, we will be modifying the sample app from the data binding and key-value coding document to use a SQLite Database as the backing source for the binding.

Including SQLite database support

Before we can continue, we need to add SQLite database support to our application by including References to a couple of .DLLs files.

Do the following:

1. In the **Solution Pad**, right-click on the **References** folder and select **Edit References**.
2. Select both the **Mono.Data.Sqlite** and **System.Data** assemblies:



3. Click the **OK** button to save your changes and add the references.

Modifying the data model

Now that we have added support for directly accessing a SQLite database to our application, we need to modify our Data Model Object to read and write data from the database (as well as provide key-value coding and data

binding). In the case of our sample application, we'll edit the `PersonModel.cs` class and make it look like the following:

```
using System;
using System.Data;
using System.IO;
using Mono.Data.Sqlite;
using Foundation;
using AppKit;

namespace MacDatabase
{
    [Register("PersonModel")]
    public class PersonModel : NSObject
    {
        #region Private Variables
        private string _ID = "";
        private string _managerID = "";
        private string _name = "";
        private string _occupation = "";
        private bool _isManager = false;
        private NSMutableArray _people = new NSMutableArray();
        private SqliteConnection _conn = null;
        #endregion

        #region Computed Properties
        public SqliteConnection Conn {
            get { return _conn; }
            set { _conn = value; }
        }

        [Export("ID")]
        public string ID {
            get { return _ID; }
            set {
                WillChangeValue ("ID");
                _ID = value;
                DidChangeValue ("ID");
            }
        }

        [Export("ManagerID")]
        public string ManagerID {
            get { return _managerID; }
            set {
                WillChangeValue ("ManagerID");
                _managerID = value;
                DidChangeValue ("ManagerID");
            }
        }

        [Export("Name")]
        public string Name {
            get { return _name; }
            set {
                WillChangeValue ("Name");
                _name = value;
                DidChangeValue ("Name");

                // Save changes to database?
                if (_conn != null) Update (_conn);
            }
        }

        [Export("Occupation")]
        public string Occupation {
            get { return _occupation; }
            set {
                WillChangeValue ("Occupation");
                _occupation = value;
                DidChangeValue ("Occupation");
            }
        }
    }
}
```

```

        set {
            WillChangeValue ("Occupation");
            _occupation = value;
            DidChangeValue ("Occupation");

            // Save changes to database?
            if (_conn != null) Update (_conn);
        }
    }

    [Export("isManager")]
    public bool isManager {
        get { return _isManager; }
        set {
            WillChangeValue ("isManager");
            WillChangeValue ("Icon");
            _isManager = value;
            DidChangeValue ("isManager");
            DidChangeValue ("Icon");

            // Save changes to database?
            if (_conn != null) Update (_conn);
        }
    }

    [Export("isEmployee")]
    public bool isEmployee {
        get { return (NumberOfEmployees == 0); }
    }

    [Export("Icon")]
    public NSImage Icon {
        get {
            if (isManager) {
                return NSImage.ImageNamed ("group.png");
            } else {
                return NSImage.ImageNamed ("user.png");
            }
        }
    }

    [Export("personModelArray")]
    public NSArray People {
        get { return _people; }
    }

    [Export("NumberOfEmployees")]
    public nint NumberOfEmployees {
        get { return (nint)_people.Count; }
    }
    #endregion

    #region Constructors
    public PersonModel ()
    {

    }

    public PersonModel (string name, string occupation)
    {
        // Initialize
        this.Name = name;
        this.Occupation = occupation;
    }

    public PersonModel (string name, string occupation, bool manager)
    {
        // Initialize
        this.Name = name;
        this.Occupation = occupation;
        ...
    }
}

```

```

        this.isManager = manager;
    }

    public PersonModel (string id, string name, string occupation)
    {
        // Initialize
        this.ID = id;
        this.Name = name;
        this.Occupation = occupation;
    }

    public PersonModel (SqliteConnection conn, string id)
    {
        // Load from database
        Load (conn, id);
    }
}

#endregion

#region Array Controller Methods
[Export("addObject:")]
public void AddPerson(PersonModel person) {
    WillChangeValue ("personModelArray");
    isManager = true;
    _people.Add (person);
    DidChangeValue ("personModelArray");
}

[Export("insertObject:inPersonModelArrayAtIndex:")]
public void InsertPerson(PersonModel person, nint index) {
    WillChangeValue ("personModelArray");
    _people.Insert (person, index);
    DidChangeValue ("personModelArray");
}

[Export("removeObjectFromPersonModelArrayAtIndex:")]
public void RemovePerson(nint index) {
    WillChangeValue ("personModelArray");
    _people.RemoveObject (index);
    DidChangeValue ("personModelArray");
}

[Export("setPersonModelArray:")]
public void SetPeople(NSMutableArray array) {
    WillChangeValue ("personModelArray");
    _people = array;
    DidChangeValue ("personModelArray");
}
#endregion

#region SQLite Routines
public void Create(SqliteConnection conn) {

    // Clear last connection to prevent circular call to update
    _conn = null;

    // Create new record ID?
    if (ID == "") {
        ID = Guid.NewGuid ().ToString();
    }

    // Execute query
    conn.Open ();
    using (var command = conn.CreateCommand ()) {
        // Create new command
        command.CommandText = "INSERT INTO [People] (ID, Name, Occupation, isManager, ManagerID) VALUES (@COL1, @COL2, @COL3, @COL4, @COL5)";

        // Populate with data from the record
        command.Parameters.AddWithValue ("@COL1", ID);

```

```

        command.Parameters.AddWithValue ("@COL2", Name);
        command.Parameters.AddWithValue ("@COL3", Occupation);
        command.Parameters.AddWithValue ("@COL4", isManager);
        command.Parameters.AddWithValue ("@COL5", ManagerID);

        // Write to database
        command.ExecuteNonQuery ();
    }
    conn.Close ();

    // Save children to database as well
    for (nuint n = 0; n < People.Count; ++n) {
        // Grab person
        var Person = People.GetItem<PersonModel>(n);

        // Save manager ID and create the sub record
        Person.ManagerID = ID;
        Person.Create (conn);
    }

    // Save last connection
    _conn = conn;
}

public void Update(SqliteConnection conn) {

    // Clear last connection to prevent circular call to update
    _conn = null;

    // Execute query
    conn.Open ();
    using (var command = conn.CreateCommand ()) {
        // Create new command
        command.CommandText = "UPDATE [People] SET Name = @COL2, Occupation = @COL3, isManager = @COL4, ManagerID = @COL5 WHERE ID = @COL1";

        // Populate with data from the record
        command.Parameters.AddWithValue ("@COL1", ID);
        command.Parameters.AddWithValue ("@COL2", Name);
        command.Parameters.AddWithValue ("@COL3", Occupation);
        command.Parameters.AddWithValue ("@COL4", isManager);
        command.Parameters.AddWithValue ("@COL5", ManagerID);

        // Write to database
        command.ExecuteNonQuery ();
    }
    conn.Close ();

    // Save children to database as well
    for (nuint n = 0; n < People.Count; ++n) {
        // Grab person
        var Person = People.GetItem<PersonModel>(n);

        // Update sub record
        Person.Update (conn);
    }

    // Save last connection
    _conn = conn;
}

public void Load(SqliteConnection conn, string id) {
    bool shouldClose = false;

    // Clear last connection to prevent circular call to update
    _conn = null;

    // Is the database already open?
    if (conn.State != ConnectionState.Open) {

```

```

        shouldClose = true;
        conn.Open ();
    }

    // Execute query
    using (var command = conn.CreateCommand ()) {
        // Create new command
        command.CommandText = "SELECT * FROM [People] WHERE ID = @COL1";

        // Populate with data from the record
        command.Parameters.AddWithValue ("@COL1", id);

        using (var reader = command.ExecuteReader ()) {
            while (reader.Read ()) {
                // Pull values back into class
                ID = (string)reader [0];
                Name = (string)reader [1];
                Occupation = (string)reader [2];
                isManager = (bool)reader [3];
                ManagerID = (string)reader [4];
            }
        }
    }

    // Is this a manager?
    if (isManager) {
        // Yes, load children
        using (var command = conn.CreateCommand ()) {
            // Create new command
            command.CommandText = "SELECT ID FROM [People] WHERE ManagerID = @COL1";

            // Populate with data from the record
            command.Parameters.AddWithValue ("@COL1", id);

            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Load child and add to collection
                    var childID = (string)reader [0];
                    var person = new PersonModel (conn, childID);
                    _people.Add (person);
                }
            }
        }
    }

    // Should we close the connection to the database
    if (shouldClose) {
        conn.Close ();
    }

    // Save last connection
    _conn = conn;
}

public void Delete(SqliteConnection conn) {

    // Clear last connection to prevent circular call to update
    _conn = null;

    // Execute query
    conn.Open ();
    using (var command = conn.CreateCommand ()) {
        // Create new command
        command.CommandText = "DELETE FROM [People] WHERE (ID = @COL1 OR ManagerID = @COL1)";

        // Populate with data from the record
        command.Parameters.AddWithValue ("@COL1", ID);

        // Write to database
    }
}

```

```

        command.ExecuteNonQuery ();
    }
    conn.Close ();

    // Empty class
    ID = "";
    ManagerID = "";
    Name = "";
    Occupation = "";
    isManager = false;
    _people = new NSMutableArray();

    // Save last connection
    _conn = conn;
}
#endregion
}
}

```

Let's take a look at the modifications in detail below.

First, we've added several using statements that are required to use SQLite and we've added a variable to save our last connection to the SQLite database:

```

using System.Data;
using System.IO;
using Mono.Data.Sqlite;
...

private SqliteConnection _conn = null;

```

We'll use this saved connection to automatically save any changes to the record to the database when the user modifies the contents in the UI via data binding:

```

[Export("Name")]
public string Name {
    get { return _name; }
    set {
        WillChangeValue ("Name");
        _name = value;
        DidChangeValue ("Name");

        // Save changes to database?
        if (_conn != null) Update (_conn);
    }
}

[Export("Occupation")]
public string Occupation {
    get { return _occupation; }
    set {
        WillChangeValue ("Occupation");
        _occupation = value;
        DidChangeValue ("Occupation");

        // Save changes to database?
        if (_conn != null) Update (_conn);
    }
}

[Export("isManager")]
public bool isManager {
    get { return _isManager; }
    set {
        WillChangeValue ("isManager");
        WillChangeValue ("Icon");
        _isManager = value;
        DidChangeValue ("isManager");
        DidChangeValue ("Icon");

        // Save changes to database?
        if (_conn != null) Update (_conn);
    }
}

```

Any changes made to the **Name**, **Occupation** or **isManager** properties will be sent to the database if the data has been saved there before (e.g. if the `_conn` variable is not `null`). Next, let's look at the methods that we've added to **Create**, **Update**, **Load** and **Delete** people from the database.

Create a new record

The following code was added to create a new record in the SQLite database:

```

public void Create(SqliteConnection conn) {

    // Clear last connection to prevent circular call to update
    _conn = null;

    // Create new record ID?
    if (ID == "") {
        ID = Guid.NewGuid ().ToString();
    }

    // Execute query
    conn.Open ();
    using (var command = conn.CreateCommand ()) {
        // Create new command
        command.CommandText = "INSERT INTO [People] (ID, Name, Occupation, isManager, ManagerID) VALUES (@COL1, @COL2, @COL3, @COL4, @COL5)";

        // Populate with data from the record
        command.Parameters.AddWithValue ("@COL1", ID);
        command.Parameters.AddWithValue ("@COL2", Name);
        command.Parameters.AddWithValue ("@COL3", Occupation);
        command.Parameters.AddWithValue ("@COL4", isManager);
        command.Parameters.AddWithValue ("@COL5", ManagerID);

        // Write to database
        command.ExecuteNonQuery ();
    }
    conn.Close ();

    // Save children to database as well
    for (uint n = 0; n < People.Count; ++n) {
        // Grab person
        var Person = People.GetItem<PersonModel>(n);

        // Save manager ID and create the sub record
        Person.ManagerID = ID;
        Person.Create (conn);
    }

    // Save last connection
    _conn = conn;
}

```

We are using a `SQLiteCommand` to create the new record in the database. We get a new command from the `SQLiteConnection` (`conn`) that we passed into the method by calling `CreateCommand`. Next, we set the SQL instruction to actually write the new record, providing parameters for the actual values:

```

command.CommandText = "INSERT INTO [People] (ID, Name, Occupation, isManager, ManagerID) VALUES (@COL1, @COL2, @COL3, @COL4, @COL5)";

```

Later we set the values for the parameters using the `Parameters.AddWithValue` method on the `SQLiteCommand`. By using parameters, we ensure that values (such as a single quote) get properly encoded before being sent to SQLite. Example:

```

command.Parameters.AddWithValue ("@COL1", ID);

```

Finally, since a person can be a manager and have a collection of employees under them, we are recursively calling the `Create` method on those people to save them to the database as well:

```

// Save children to database as well
for (uint n = 0; n < People.Count; ++n) {
    // Grab person
    var Person = People.GetItem<PersonModel>(n);

    // Save manager ID and create the sub record
    Person.ManagerID = ID;
    Person.Create (conn);
}

```

Updating a record

The following code was added to update an existing record in the SQLite database:

```

public void Update(SqliteConnection conn) {

    // Clear last connection to prevent circular call to update
    _conn = null;

    // Execute query
    conn.Open ();
    using (var command = conn.CreateCommand ()) {
        // Create new command
        command.CommandText = "UPDATE [People] SET Name = @COL2, Occupation = @COL3, isManager = @COL4,
ManagerID = @COL5 WHERE ID = @COL1";

        // Populate with data from the record
        command.Parameters.AddWithValue ("@COL1", ID);
        command.Parameters.AddWithValue ("@COL2", Name);
        command.Parameters.AddWithValue ("@COL3", Occupation);
        command.Parameters.AddWithValue ("@COL4", isManager);
        command.Parameters.AddWithValue ("@COL5", ManagerID);

        // Write to database
        command.ExecuteNonQuery ();
    }
    conn.Close ();

    // Save children to database as well
    for (uint n = 0; n < People.Count; ++n) {
        // Grab person
        var Person = People.GetItem<PersonModel>(n);

        // Update sub record
        Person.Update (conn);
    }

    // Save last connection
    _conn = conn;
}

```

Like **Create** above, we get a `SQLiteCommand` from the passed in `SqliteConnection`, and set our SQL to update our record (providing parameters):

```

command.CommandText = "UPDATE [People] SET Name = @COL2, Occupation = @COL3, isManager = @COL4, ManagerID =
@COL5 WHERE ID = @COL1";

```

We fill in the parameter values (example: `command.Parameters.AddWithValue ("@COL1", ID);`) and again, recursively call update on any child records:

```
// Save children to database as well
for (nuint n = 0; n < People.Count; ++n) {
    // Grab person
    var Person = People.GetItem<PersonModel>(n);

    // Update sub record
    Person.Update (conn);
}
```

Loading a record

The following code was added to load an existing record from the SQLite database:

```

public void Load(SqliteConnection conn, string id) {
    bool shouldClose = false;

    // Clear last connection to prevent circular call to update
    _conn = null;

    // Is the database already open?
    if (conn.State != ConnectionState.Open) {
        shouldClose = true;
        conn.Open ();
    }

    // Execute query
    using (var command = conn.CreateCommand ()) {
        // Create new command
        command.CommandText = "SELECT * FROM [People] WHERE ID = @COL1";

        // Populate with data from the record
        command.Parameters.AddWithValue ("@COL1", id);

        using (var reader = command.ExecuteReader ()) {
            while (reader.Read ()) {
                // Pull values back into class
                ID = (string)reader [0];
                Name = (string)reader [1];
                Occupation = (string)reader [2];
                isManager = (bool)reader [3];
                ManagerID = (string)reader [4];
            }
        }
    }

    // Is this a manager?
    if (isManager) {
        // Yes, load children
        using (var command = conn.CreateCommand ()) {
            // Create new command
            command.CommandText = "SELECT ID FROM [People] WHERE ManagerID = @COL1";

            // Populate with data from the record
            command.Parameters.AddWithValue ("@COL1", id);

            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Load child and add to collection
                    var childID = (string)reader [0];
                    var person = new PersonModel (conn, childID);
                    _people.Add (person);
                }
            }
        }
    }

    // Should we close the connection to the database
    if (shouldClose) {
        conn.Close ();
    }

    // Save last connection
    _conn = conn;
}

```

Because the routine can be called recursively from a parent object (such as a manager object loading their employees object), special code was added to handle opening and closing the connection to the database:

```

bool shouldClose = false;
...
// Is the database already open?
if (conn.State != ConnectionState.Open) {
    shouldClose = true;
    conn.Open ();
}
...
// Should we close the connection to the database
if (shouldClose) {
    conn.Close ();
}

```

As always, we set our SQL to retrieve the record and use parameters:

```

// Create new command
command.CommandText = "SELECT ID FROM [People] WHERE ManagerID = @COL1";

// Populate with data from the record
command.Parameters.AddWithValue ("@COL1", id);

```

Finally, we use a Data Reader to execute the query and return the record fields (which we copy into the instance of the `PersonModel` class):

```

using (var reader = command.ExecuteReader ()) {
    while (reader.Read ()) {
        // Pull values back into class
        ID = (string)reader [0];
        Name = (string)reader [1];
        Occupation = (string)reader [2];
        isManager = (bool)reader [3];
        ManagerID = (string)reader [4];
    }
}

```

If this person is a manager, we need to also load all of their employees (again, by recursively calling their `Load` method):

```

// Is this a manager?
if (isManager) {
    // Yes, load children
    using (var command = conn.CreateCommand ()) {
        // Create new command
        command.CommandText = "SELECT ID FROM [People] WHERE ManagerID = @COL1";

        // Populate with data from the record
        command.Parameters.AddWithValue ("@COL1", id);

        using (var reader = command.ExecuteReader ()) {
            while (reader.Read ()) {
                // Load child and add to collection
                var childID = (string)reader [0];
                var person = new PersonModel (conn, childID);
                _people.Add (person);
            }
        }
    }
}

```

Deleting a record

The following code was added to delete an existing record from the SQLite database:

```

public void Delete(SqliteConnection conn) {

    // Clear last connection to prevent circular call to update
    _conn = null;

    // Execute query
    conn.Open ();
    using (var command = conn.CreateCommand ()) {
        // Create new command
        command.CommandText = "DELETE FROM [People] WHERE (ID = @COL1 OR ManagerID = @COL1)";

        // Populate with data from the record
        command.Parameters.AddWithValue ("@COL1", ID);

        // Write to database
        command.ExecuteNonQuery ();
    }
    conn.Close ();

    // Empty class
    ID = "";
    ManagerID = "";
    Name = "";
    Occupation = "";
    isManager = false;
    _people = new NSMutableArray();

    // Save last connection
    _conn = conn;
}

```

Here we provide the SQL to delete both the managers record and the records of any employees under that manager (using parameters):

```
// Create new command
command.CommandText = "DELETE FROM [People] WHERE (ID = @COL1 OR ManagerID = @COL1)";

// Populate with data from the record
command.Parameters.AddWithValue ("@COL1", ID);
```

After the record has been removed, we clear out the current instance of the `PersonModel` class:

```
// Empty class
ID = "";
ManagerID = "";
Name = "";
Occupation = "";
isManager = false;
_people = new NSMutableArray();
```

Initializing the database

With the changes to our Data Model in place to support reading and writing to the database, we need to open a connection to the database and initialize it on the first run. Let's add the following code to our `MainWindow.cs` file:

```

using System.Data;
using System.IO;
using Mono.Data.Sqlite;
...

private SqliteConnection DatabaseConnection = null;
...

private SqliteConnection GetDatabaseConnection() {
    var documents = Environment.GetFolderPath (Environment.SpecialFolder.Desktop);
    string db = Path.Combine (documents, "People.db3");

    // Create the database if it doesn't already exist
    bool exists = File.Exists (db);
    if (!exists)
        SqliteConnection.CreateFile (db);

    // Create connection to the database
    var conn = new SqliteConnection("Data Source=" + db);

    // Set the structure of the database
    if (!exists) {
        var commands = new[] {
            "CREATE TABLE People (ID TEXT, Name TEXT, Occupation TEXT, isManager BOOLEAN, ManagerID TEXT)"
        };
        conn.Open ();
        foreach (var cmd in commands) {
            using (var c = conn.CreateCommand()) {
                c.CommandText = cmd;
                c.CommandType = CommandType.Text;
                c.ExecuteNonQuery ();
            }
        }
        conn.Close ();
    }

    // Build list of employees
    var Craig = new PersonModel ("0","Craig Dunn", "Documentation Manager");
    Craig.AddPerson (new PersonModel ("Amy Burns", "Technical Writer"));
    Craig.AddPerson (new PersonModel ("Joel Martinez", "Web & Infrastructure"));
    Craig.AddPerson (new PersonModel ("Kevin Mullins", "Technical Writer"));
    Craig.AddPerson (new PersonModel ("Mark McLemore", "Technical Writer"));
    Craig.AddPerson (new PersonModel ("Tom Opgenorth", "Technical Writer"));
    Craig.Create (conn);

    var Larry = new PersonModel ("1","Larry O'Brien", "API Documentation Manager");
    Larry.AddPerson (new PersonModel ("Mike Norman", "API Documentor"));
    Larry.Create (conn);
}

// Return new connection
return conn;
}

```

Let's take a closer look at the code above. First, we pick a location for the new database (in this example, the user's Desktop), look to see if the database exists, and if it doesn't, create it:

```

var documents = Environment.GetFolderPath (Environment.SpecialFolder.Desktop);
string db = Path.Combine (documents, "People.db3");

// Create the database if it doesn't already exist
bool exists = File.Exists (db);
if (!exists)
    SqliteConnection.CreateFile (db);

```

Next, we establish the connect to the database using the path we created above:

```
var conn = new SqliteConnection("Data Source=" + db);
```

Then we create all the SQL tables in the database that we require:

```
var commands = new[] {
    "CREATE TABLE People (ID TEXT, Name TEXT, Occupation TEXT, isManager BOOLEAN, ManagerID TEXT)"
};

conn.Open ();
foreach (var cmd in commands) {
    using (var c = conn.CreateCommand ()) {
        c.CommandText = cmd;
        c.CommandType = CommandType.Text;
        c.ExecuteNonQuery ();
    }
}
conn.Close ();
```

Finally, we use our Data Model (`PersonModel`) to create a default set of records for the database the first time the application is run or if the database is missing:

```
// Build list of employees
var Craig = new PersonModel ("0", "Craig Dunn", "Documentation Manager");
Craig.AddPerson (new PersonModel ("Amy Burns", "Technical Writer"));
Craig.AddPerson (new PersonModel ("Joel Martinez", "Web & Infrastructure"));
Craig.AddPerson (new PersonModel ("Kevin Mullins", "Technical Writer"));
Craig.AddPerson (new PersonModel ("Mark McLemore", "Technical Writer"));
Craig.AddPerson (new PersonModel ("Tom Opgenorth", "Technical Writer"));
Craig.Create (conn);

var Larry = new PersonModel ("1", "Larry O'Brien", "API Documentation Manager");
Larry.AddPerson (new PersonModel ("Mike Norman", "API Documentor"));
Larry.Create (conn);
```

When the application starts and opens the Main Window, we make a connection to the database using the code we added above:

```
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

    // Get access to database
    DatabaseConnection = GetDatabaseConnection ();
}
```

Loading bound data

With all the components for directly accessing bound data from a SQLite database in place, we can load the data in the different views that our application provides and it will automatically be displayed in our UI.

Loading a single record

To load a single record where the ID is known, we can use the following code:

```
Person = new PersonModel (Conn, "0");
```

Loading all records

To load all people, regardless if they are a manager or not, use the following code:

```

// Load all employees
_conn.Open ();
using (var command = _conn.CreateCommand ()) {
    // Create new command
    command.CommandText = "SELECT ID FROM [People]";

    using (var reader = command.ExecuteReader ()) {
        while (reader.Read ()) {
            // Load child and add to collection
            var childID = (string)reader [0];
            var person = new PersonModel (_conn, childID);
            AddPerson (person);
        }
    }
}
_conn.Close ();

```

Here, we are using an overload of the constructor for the `PersonModel` class to load the person into memory:

```
var person = new PersonModel (_conn, childID);
```

We are also calling the Data Bound class to add the person to our collection of people `AddPerson (person)`, this ensures that our UI recognizes the change and displays it:

```

[Export("addObject")]
public void AddPerson(PersonModel person) {
    WillChangeValue ("personModelArray");
    isManager = true;
    _people.Add (person);
    DidChangeValue ("personModelArray");
}

```

Loading top level records only

To load only managers (for example, to display data in an Outline View), we use the following code:

```

// Load only managers employees
_conn.Open ();
using (var command = _conn.CreateCommand ()) {
    // Create new command
    command.CommandText = "SELECT ID FROM [People] WHERE isManager = 1";

    using (var reader = command.ExecuteReader ()) {
        while (reader.Read ()) {
            // Load child and add to collection
            var childID = (string)reader [0];
            var person = new PersonModel (_conn, childID);
            AddPerson (person);
        }
    }
}
_conn.Close ();

```

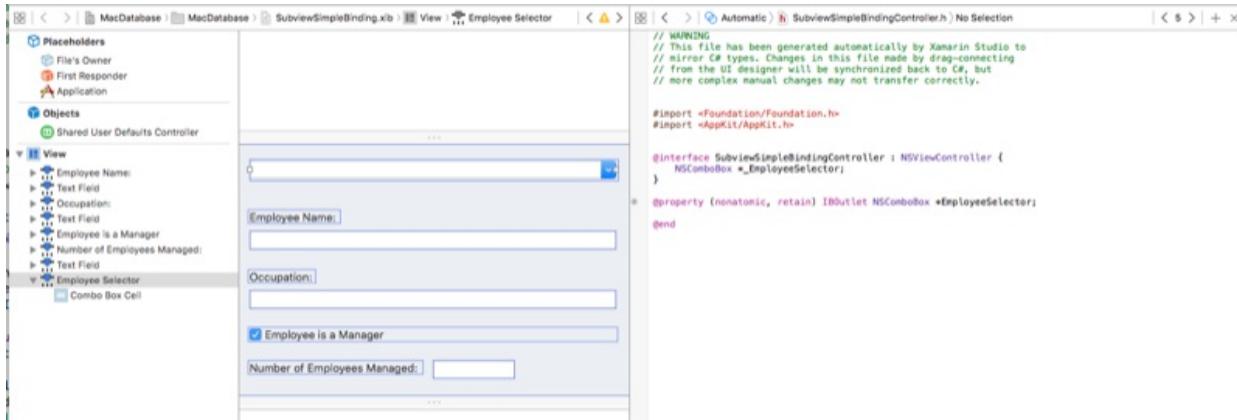
The only real difference in the in SQL statement (which loads only managers

`command.CommandText = "SELECT ID FROM [People] WHERE isManager = 1"`) but works the same as the section above otherwise.

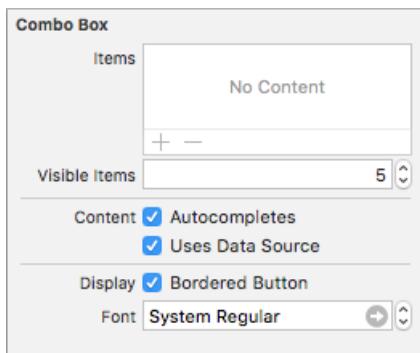
Databases and comboboxes

The Menu Controls available to macOS (such as the Combo Box) can be set to populate the dropdown list either from an internal list (that can be pre-defined in Interface Builder or populated via code) or by providing your own custom, external data source. See [Providing Menu Control Data](#) for more details.

As an example, edit the Simple Binding example above in Interface Builder, add a Combo Box and expose it using an outlet named `EmployeeSelector`:



In the **Attributes Inspector**, check the **Autocompletes** and **Uses Data Source** properties:



Save your changes and return to Visual Studio for Mac to sync.

Providing combobox data

Next, add a new class to the project called `ComboBoxDataSource` and make it look like the following:

```
using System;
using System.Data;
using System.IO;
using Mono.Data.Sqlite;
using Foundation;
using AppKit;

namespace MacDatabase
{
    public class ComboBoxDataSource : NSComboBoxDataSource
    {
        #region Private Variables
        private SqliteConnection _conn = null;
        private string _tableName = "";
        private string _IDField = "ID";
        private string _displayField = "";
        private nint _recordCount = 0;
        #endregion

        #region Computed Properties
        public SqliteConnection Conn {
            get { return _conn; }
            set { _conn = value; }
        }
    }
}
```

```

public string TableName {
    get { return _tableName; }
    set {
        _tableName = value;
        _recordCount = GetRecordCount ();
    }
}

public string IDField {
    get { return _IDField; }
    set {
        _IDField = value;
        _recordCount = GetRecordCount ();
    }
}

public string DisplayField {
    get { return _displayField; }
    set {
        _displayField = value;
        _recordCount = GetRecordCount ();
    }
}

public int RecordCount {
    get { return _recordCount; }
}
#endregion

#region Constructors
public ComboBoxDataSource (SqliteConnection conn, string tableName, string displayField)
{
    // Initialize
    this.Conn = conn;
    this.TableName = tableName;
    this.DisplayField = displayField;
}

public ComboBoxDataSource (SqliteConnection conn, string tableName, string idField, string
displayField)
{
    // Initialize
    this.Conn = conn;
    this.TableName = tableName;
    this.IDField = idField;
    this.DisplayField = displayField;
}
#endregion

#region Private Methods
private int GetRecordCount ()
{
    bool shouldClose = false;
    int count = 0;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDField != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT count({IDField}) FROM [{TableName}]";

            // Get the results from the database

```

```

        using (var reader = command.ExecuteReader ()) {
            while (reader.Read ()) {
                // Read count from query
                var result = (long)reader [0];
                count = (nint)result;
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return the number of records
    return count;
}
#endregion

#region Public Methods
public string IDForIndex (nint index)
{
    NSString value = new NSString ("");
    bool shouldClose = false;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDFField != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT {IDField} FROM [{TableName}] ORDER BY {DisplayField} ASC
LIMIT 1 OFFSET {index}";

            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Read the display field from the query
                    value = new NSString ((string)reader [0]);
                }
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return results
    return value;
}

public string ValueForIndex (nint index)
{
    NSString value = new NSString ("");
    bool shouldClose = false;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDFField != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
        }
    }
}

```

```

        shouldClose = true;
        Conn.Open ();
    }

    // Execute query
    using (var command = Conn.CreateCommand ()) {
        // Create new command
        command.CommandText = $"SELECT {DisplayField} FROM [{TableName}] ORDER BY {DisplayField}
ASC LIMIT 1 OFFSET {index}";

        // Get the results from the database
        using (var reader = command.ExecuteReader ()) {
            while (reader.Read ()) {
                // Read the display field from the query
                value = new NSString ((string)reader [0]);
            }
        }
    }

    // Should we close the connection to the database
    if (shouldClose) {
        Conn.Close ();
    }
}

// Return results
return value;
}

public string IDForValue (string value)
{
    NSString result = new NSString ("");
    bool shouldClose = false;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDField != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT {IDField} FROM [{TableName}] WHERE {DisplayField} =
@VAL";

            // Populate parameters
            command.Parameters.AddWithValue ("@VAL", value);

            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Read the display field from the query
                    result = new NSString ((string)reader [0]);
                }
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return results
    return result;
}
#endif

```

```

#endregion

#region Override Methods
public override nint ItemCount (NSComboBox comboBox)
{
    return RecordCount;
}

public override NSObject ObjectValueForItem (NSComboBox comboBox, nint index)
{
    NSString value = new NSString ("");
    bool shouldClose = false;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDFielD != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT {DisplayField} FROM [{TableName}] ORDER BY {DisplayField}
ASC LIMIT 1 OFFSET {index}";

            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Read the display field from the query
                    value = new NSString((string)reader [0]);
                }
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return results
    return value;
}

public override nint IndexOfItem (NSComboBox comboBox, string value)
{
    bool shouldClose = false;
    bool found = false;
    string field = "";
    nint index = NSRange.NotFound;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDFielD != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT {DisplayField} FROM [{TableName}] ORDER BY {DisplayField}
ASC";
            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {

```

```

        while (reader.Read () && !found) {
            // Read the display field from the query
            field = (string)reader [0];
            ++index;

            // Is this the value we are searching for?
            if (value == field) {
                // Yes, exit loop
                found = true;
            }
        }
    }

    // Should we close the connection to the database
    if (shouldClose) {
        Conn.Close ();
    }
}

// Return results
return index;
}

public override string CompletedString (NSComboBox comboBox, string uncompletedString)
{
    bool shouldClose = false;
    bool found = false;
    string field = "";

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDFiel != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Escape search string
        uncompletedString = uncompletedString.Replace ("'", "");

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT {DisplayField} FROM [{TableName}] WHERE {DisplayField}
LIKE @VAL";

            // Populate parameters
            command.Parameters.AddWithValue ("@VAL", uncompletedString + "%");

            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Read the display field from the query
                    field = (string)reader [0];
                }
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return results
    return field;
}
#endifregion

```

```
    }  
}
```

In this example, we are creating a new `NSComboBoxDataSource` that can present Combo Box Items from any SQLite Data Source. First, we define the following properties:

- `Conn` - Gets or sets a connection to the SQLite database.
- `TableName` - Gets or sets the table name.
- `IDField` - Gets or sets the field that provides the unique ID for the given Table. The default value is `ID`.
- `DisplayField` - Gets or sets the field that is displayed in the dropdown list.
- `RecordCount` - Gets the number of records in the given Table.

When we create a new instance of the object, we pass in the connection, table name, optionally the ID field and the display field:

```
public ComboBoxDataSource (SqliteConnection conn, string tableName, string displayField)  
{  
    // Initialize  
    this.Conn = conn;  
    this.TableName = tableName;  
    this.DisplayField = displayField;  
}
```

The `GetRecordCount` method returns the number of records in the given Table:

```

private nint GetRecordCount ()
{
    bool shouldClose = false;
    nint count = 0;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDField != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT count({IDField}) FROM [{TableName}]";

            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Read count from query
                    var result = (long)reader [0];
                    count = (nint)result;
                }
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return the number of records
    return count;
}

```

It is called any time the `TableName`, `IDField` or `DisplayField` properties value is changed.

The `IDForIndex` method returns the unique ID (`IDField`) for the record at the given dropdown list item index:

```

public string IDForIndex (nint index)
{
    NSString value = new NSString ("");
    bool shouldClose = false;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDFiel !="" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT {IDFiel} FROM [{TableName}] ORDER BY {DisplayField} ASC LIMIT 1
OFFSET {index}";

            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Read the display field from the query
                    value = new NSString ((string)reader [0]);
                }
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return results
    return value;
}

```

The `ValueForIndex` method returns the value (`DisplayField`) for the item at the given dropdown list index:

```

public string ValueForIndex (nint index)
{
    NSString value = new NSString ("");
    bool shouldClose = false;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDField != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT {DisplayField} FROM [{TableName}] ORDER BY {DisplayField} ASC
LIMIT 1 OFFSET {index}";

            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Read the display field from the query
                    value = new NSString ((string)reader [0]);
                }
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return results
    return value;
}

```

The `IDForValue` method returns the unique ID (`IDField`) for the given value (`DisplayField`):

```

public string IDForValue (string value)
{
    NSString result = new NSString ("");
    bool shouldClose = false;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDFiel !="" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT {IDFiel} FROM [{TableName}] WHERE {DisplayField} = @VAL";

            // Populate parameters
            command.Parameters.AddWithValue ("@VAL", value);

            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Read the display field from the query
                    result = new NSString ((string)reader [0]);
                }
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return results
    return result;
}

```

The `ItemCount` returns the precomputed number of items in the list as calculated when the `TableName`, `IDFiel` or `DisplayField` properties are changed:

```

public override nint ItemCount (NSComboBox comboBox)
{
    return RecordCount;
}

```

The `objectValueForItem` method provides the value (`DisplayField`) for the given dropdown list item index:

```

public override NSObject ObjectValueForItem (NSComboBox comboBox, nint index)
{
    NSString value = new NSString ("");
    bool shouldClose = false;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDFielD != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT {DisplayField} FROM [{TableName}] ORDER BY {DisplayField} ASC
LIMIT 1 OFFSET {index}";

            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Read the display field from the query
                    value = new NSString((string)reader [0]);
                }
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return results
    return value;
}

```

Notice that we are using the `LIMIT` and `OFFSET` statements in our SQLite command to limit to the one record that we are needed.

The `IndexOfItem` method returns dropdown item index of the value (`DisplayField`) given:

```

public override nint IndexOfItem (NSComboBox comboBox, string value)
{
    bool shouldClose = false;
    bool found = false;
    string field = "";
    nint index = NSRange.NotFound;

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDFfield != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT {DisplayField} FROM [{TableName}] ORDER BY {DisplayField} ASC";

            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {
                while (reader.Read () && !found) {
                    // Read the display field from the query
                    field = (string)reader [0];
                    ++index;

                    // Is this the value we are searching for?
                    if (value == field) {
                        // Yes, exit loop
                        found = true;
                    }
                }
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return results
    return index;
}

```

If the value cannot be found, the `NSRange.NotFound` value is returned and all items are deselected in the dropdown list.

The `CompletedString` method returns the first matching value (`DisplayField`) for a partially typed entry:

```

public override string CompletedString (NSComboBox comboBox, string uncompletedString)
{
    bool shouldClose = false;
    bool found = false;
    string field = "";

    // Has a Table, ID and display field been specified?
    if (TableName != "" && IDField != "" && DisplayField != "") {
        // Is the database already open?
        if (Conn.State != ConnectionState.Open) {
            shouldClose = true;
            Conn.Open ();
        }

        // Escape search string
        uncompletedString = uncompletedString.Replace ("'", "");

        // Execute query
        using (var command = Conn.CreateCommand ()) {
            // Create new command
            command.CommandText = $"SELECT {DisplayField} FROM [{TableName}] WHERE {DisplayField} LIKE
@VAL";

            // Populate parameters
            command.Parameters.AddWithValue ("@VAL", uncompletedString + "%");

            // Get the results from the database
            using (var reader = command.ExecuteReader ()) {
                while (reader.Read ()) {
                    // Read the display field from the query
                    field = (string)reader [0];
                }
            }
        }

        // Should we close the connection to the database
        if (shouldClose) {
            Conn.Close ();
        }
    }

    // Return results
    return field;
}

```

Displaying data and responding to events

To bring all of the pieces together, edit the `SubviewSimpleBindingController` and make it look like the following:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Data;
using System.IO;
using Mono.Data.Sqlite;
using Foundation;
using AppKit;

namespace MacDatabase
{
    public partial class SubviewSimpleBindingController : AppKit.NSViewController
    {
        #region Private Variables
        private PersonModel _person = new PersonModel();
        private SqliteConnection Conn;
        #endregion

```

```

#region Computed Properties
//strongly typed view accessor
public newSubviewSimpleBinding View {
    get {
        return (SubviewSimpleBinding)base.View;
    }
}

[Export("Person")]
public PersonModel Person {
    get {return _person; }
    set {
        WillChangeValue ("Person");
        _person = value;
        DidChangeValue ("Person");
    }
}

public ComboBoxDataSource DataSource {
    get { return EmployeeSelector.DataSource as ComboBoxDataSource; }
}
#endregion

#region Constructors
// Called when created from unmanaged code
public SubviewSimpleBindingController (IntPtr handle) : base (handle)
{
    Initialize ();
}

// Called when created directly from a XIB file
[Export ("initWithCoder:")]
public SubviewSimpleBindingController (NSCoder coder) : base (coder)
{
    Initialize ();
}

// Call to load from the XIB/NIB file
public SubviewSimpleBindingController (SqliteConnection conn) : base ("SubviewSimpleBinding",
NSBundle.MainBundle)
{
    // Initialize
    this.Conn = conn;
    Initialize ();
}

// Shared initialization code
void Initialize ()
{
}
#endregion

#region Private Methods
private void LoadSelectedPerson (string id)
{
    // Found?
    if (id != "") {
        // Yes, load requested record
        Person = new PersonModel (Conn, id);
    }
}
#endregion

#region Override Methods
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();
}

```

```

// Configure Employee selector dropdown
EmployeeSelector.DataSource = new ComboBoxDataSource (Conn, "People", "Name");

// Wireup events
EmployeeSelector.Changed += (sender, e) => {
    // Get ID
    var id = DataSource.IDForValue (EmployeeSelector.StringValue);
    LoadSelectedPerson (id);
};

EmployeeSelector.SelectionChanged += (sender, e) => {
    // Get ID
    var id = DataSource.IDForIndex (EmployeeSelector.SelectedIndex);
    LoadSelectedPerson (id);
};

// Auto select the first person
EmployeeSelector.StringValue = DataSource.ValueForIndex (0);
Person = new PersonModel (Conn, DataSource.IDForIndex(0));

}
#endregion
}
}

```

The `DataSource` property provides a shortcut to the `ComboBoxDataSource` (created above) attached to the Combo Box.

The `LoadSelectedPerson` method loads the person from the database for the given Unique ID:

```

private void LoadSelectedPerson (string id)
{
    // Found?
    if (id != "") {
        // Yes, load requested record
        Person = new PersonModel (Conn, id);
    }
}

```

In the `AwakeFromNib` method override, first we attach an instance of our custom Combo Box Data Source:

```
EmployeeSelector.DataSource = new ComboBoxDataSource (Conn, "People", "Name");
```

Next, we respond to the user editing the text value of the Combo Box by finding the associated unique ID (`IDField`) of the data presenting and loading the given person if found:

```

EmployeeSelector.Changed += (sender, e) => {
    // Get ID
    var id = DataSource.IDForValue (EmployeeSelector.StringValue);
    LoadSelectedPerson (id);
};

```

We also load a new person if the user selects a new item from the dropdown list:

```

EmployeeSelector.SelectionChanged += (sender, e) => {
    // Get ID
    var id = DataSource.IDForIndex (EmployeeSelector.SelectedIndex);
    LoadSelectedPerson (id);
};

```

Finally, we auto-populate the Combo Box and displayed person with the first item in the list:

```

// Auto select the first person
EmployeeSelector.StringValue = DataSource.ValueForIndex (0);
Person = new PersonModel (Conn, DataSource.IDForIndex(0));

```

SQLite.NET ORM

As stated above, by using the open source [SQLite.NET](#) Object Relationship Manager (ORM) we can greatly reduce the amount of code required to read and write data from a SQLite database. This may not be the best route to take when binding data because of several of the requirements that key-value coding and data binding place on an object.

According to the SQLite.Net website, *"SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed database engine in the world. The source code for SQLite is in the public domain."*

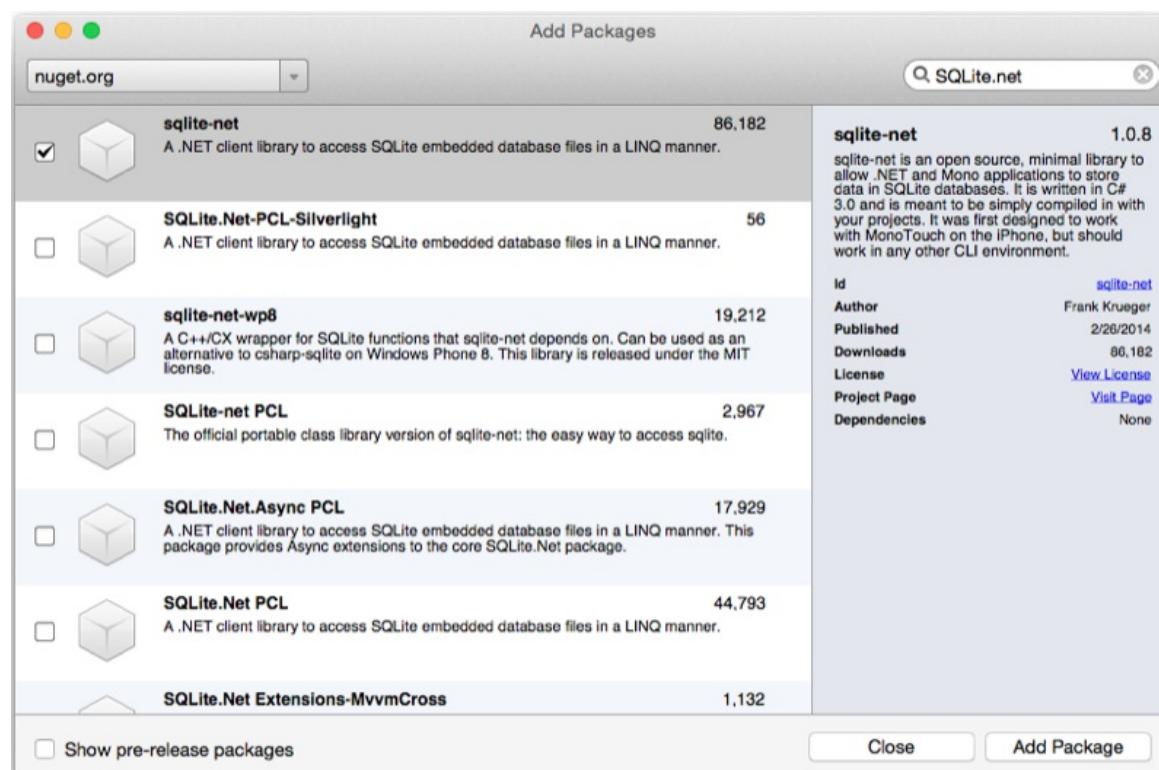
In the following sections, we'll show how to use SQLite.Net to provide data for a Table View.

Including the SQLite.net NuGet

SQLite.NET is presented as a NuGet Package that you include in your application. Before we can add database support using SQLite.NET, we need to include this package.

Do the following to add the package:

1. In the **Solution Pad**, right-click the **Packages** folder and select **Add Packages...**
2. Enter **SQLite.net** in the **Search Box** and select the **sqlite-net** entry:



3. Click the Add Package button to finish.

Creating the data model

Let's add a new class to the project and call it `OccupationModel`. Next, let's edit the `OccupationModel.cs` file and make it look like the following:

```
using System;
using SQLite;

namespace MacDatabase
{
    public class OccupationModel
    {
        #region Computed Properties
        [PrimaryKey, AutoIncrement]
        public int ID { get; set; }

        public string Name { get; set; }
        public string Description { get; set; }
        #endregion

        #region Constructors
        public OccupationModel ()
        {
        }

        public OccupationModel (string name, string description)
        {

            // Initialize
            this.Name = name;
            this.Description = description;

        }
        #endregion
    }
}
```

First, we include SQLite.NET (`using SQLite`), then we expose several Properties, each of which will be written to the database when this record is saved. The first property we make as the primary key and set it to auto increment as follows:

```
[PrimaryKey, AutoIncrement]
public int ID { get; set; }
```

Initializing the database

With the changes to our Data Model in place to support reading and writing to the database, we need to open a connection to the database and initialize it on the first run. Let's add the following code:

```

using SQLite;
...

public SQLiteConnection Conn { get; set; }
...

private SQLiteConnection GetDatabaseConnection() {
    var documents = Environment.GetFolderPath (Environment.SpecialFolder.Desktop);
    string db = Path.Combine (documents, "Occupation.db3");
    OccupationModel Occupation;

    // Create the database if it doesn't already exist
    bool exists = File.Exists (db);

    // Create connection to database
    var conn = new SQLiteConnection (db);

    // Initially populate table?
    if (!exists) {
        // Yes, build table
        conn.CreateTable<OccupationModel> ();

        // Add occupations
        Occupation = new OccupationModel ("Documentation Manager", "Manages the Documentation Group");
        conn.Insert (Occupation);

        Occupation = new OccupationModel ("Technical Writer", "Writes technical documentation and sample
applications");
        conn.Insert (Occupation);

        Occupation = new OccupationModel ("Web & Infrastructure", "Creates and maintains the websites that
drive documentation");
        conn.Insert (Occupation);

        Occupation = new OccupationModel ("API Documentation Manager", "Manages the API Documentation
Group");
        conn.Insert (Occupation);

        Occupation = new OccupationModel ("API Documenter", "Creates and maintains API documentation");
        conn.Insert (Occupation);
    }

    return conn;
}

```

First, we get a path to the database (the User's Desktop in this case) and see if the database already exists:

```

var documents = Environment.GetFolderPath (Environment.SpecialFolder.Desktop);
string db = Path.Combine (documents, "Occupation.db3");
OccupationModel Occupation;

// Create the database if it doesn't already exist
bool exists = File.Exists (db);

```

Next, we establish a connection to the database at the path we created above:

```

var conn = new SQLiteConnection (db);

```

Finally, we create the table and add some default records:

```

// Yes, build table
conn.CreateTable<OccupationModel> ();

// Add occupations
Occupation = new OccupationModel ("Documentation Manager", "Manages the Documentation Group");
conn.Insert (Occupation);

Occupation = new OccupationModel ("Technical Writer", "Writes technical documentation and sample
applications");
conn.Insert (Occupation);

Occupation = new OccupationModel ("Web & Infrastructure", "Creates and maintains the websites that drive
documentation");
conn.Insert (Occupation);

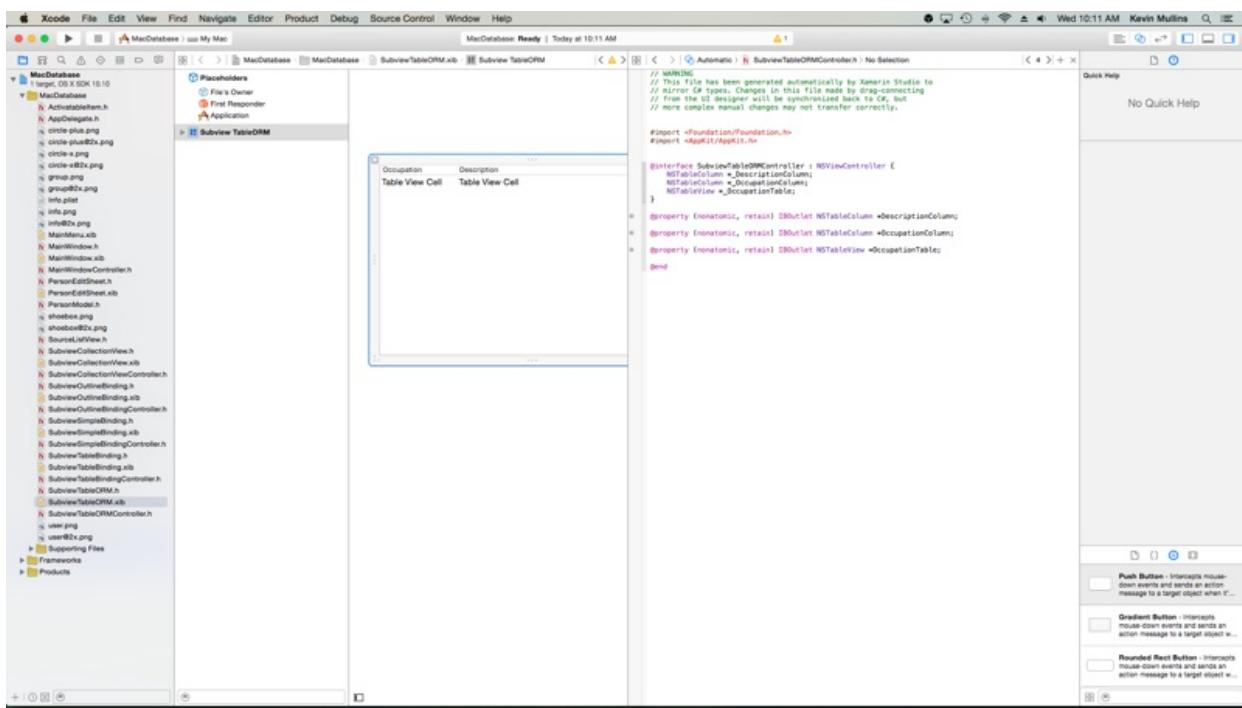
Occupation = new OccupationModel ("API Documentation Manager", "Manages the API Documentation Group");
conn.Insert (Occupation);

Occupation = new OccupationModel ("API Documenter", "Creates and maintains API documentation");
conn.Insert (Occupation);

```

Adding a table view

As an example usage, we'll add a Table View to our UI in Xcode's Interface builder. We'll expose this Table View via an outlet (`occupationTable`) so we can access it via C# code:



Next, we'll add the custom classes to populate this table with data from the SQLite.NET database.

Creating the table data source

Let's create a custom Data Source to provide data for our table. First, add a new class called `TableORMDatasource` and make it look like the following:

```

using System;
using AppKit;
using CoreGraphics;
using Foundation;
using System.Collections;
using System.Collections.Generic;
using SQLite;

namespace MacDatabase
{
    public class TableORMDatasource : NTableViewDataSource
    {
        #region Computed Properties
        public List<OccupationModel> Occupations { get; set; } = new List<OccupationModel>();
        public SQLiteConnection Conn { get; set; }
        #endregion

        #region Constructors
        public TableORMDatasource (SQLiteConnection conn)
        {
            // Initialize
            this.Conn = conn;
            LoadOccupations ();
        }
        #endregion

        #region Public Methods
        public void LoadOccupations() {

            // Get occupations from database
            var query = Conn.Table<OccupationModel> ();

            // Copy into table collection
            Occupations.Clear ();
            foreach (OccupationModel occupation in query) {
                Occupations.Add (occupation);
            }

        }
        #endregion

        #region Override Methods
        public override nint GetRowCount (NSTableView tableView)
        {
            return Occupations.Count;
        }
        #endregion
    }
}

```

When we create an instance of this class later, we'll pass in our open SQLite.NET database connection. The `LoadOccupations` method queries the database and copies the found records into memory (using our `OccupationModel` data model).

Creating the table delegate

The final class we need is a custom Table Delegate to display the information that we have loaded from the SQLite.NET database. Let's add a new `TableORMDelegate` to our project and make it look like the following:

```

using System;
using AppKit;
using CoreGraphics;
using Foundation;
using System.Collections;
using System.Collections.Generic;
using SQLite;

namespace MacDatabase
{
    public class TableORMDelegate : NSTableViewDelegate
    {
        #region Constants
        private const string CellIdentifier = "OccCell";
        #endregion

        #region Private Variables
        private TableORMDatasource DataSource;
        #endregion

        #region Constructors
        public TableORMDelegate (TableORMDatasource dataSource)
        {
            // Initialize
            this.DataSource = dataSource;
        }
        #endregion

        #region Override Methods
        public override NSView GetViewForItem (NSTableView tableView, NSTableColumn TableColumn, nint row)
        {
            // This pattern allows you reuse existing views when they are no-longer in use.
            // If the returned view is null, you instance up a new view
            // If a non-null view is returned, you modify it enough to reflect the new data
            NSTextField view = (NSTextField)tableView.MakeView (CellIdentifier, this);
            if (view == null) {
                view = new NSTextField ();
                view.Identifier = CellIdentifier;
                view.BackgroundColor = NSColor.Clear;
                view.Bordered = false;
                view.Selectable = false;
                viewEditable = false;
            }

            // Setup view based on the column selected
            switch (TableColumn.Title) {
                case "Occupation":
                    view.StringValue = DataSource.Occupations [(int)row].Name;
                    break;
                case "Description":
                    view.StringValue = DataSource.Occupations [(int)row].Description;
                    break;
            }

            return view;
        }
        #endregion
    }
}

```

Here we use the Data Source's `Occupations` collection (that we loaded from the SQLite.NET database) to fill in the columns of our table via the `GetViewForItem` method override.

Populating the table

With all of the pieces in place, let's populate our table when it is inflated from the .xib file by overriding the

`AwakeFromNib` method and making it look like the following:

```
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

    // Get database connection
    Conn = GetDatabaseConnection ();

    // Create the Occupation Table Data Source and populate it
    var DataSource = new TableORMDatasource (Conn);

    // Populate the Product Table
    OccupationTable.DataSource = DataSource;
    OccupationTable.Delegate = new TableORMDelegate (DataSource);
}
```

First, we gain access to our SQLite.NET database, creating and populating it if it doesn't already exist. Next, we create a new instance of our custom Table Data Source, pass in our database connection and we attach it to the Table. Finally, we create a new instance of our custom Table Delegate, pass in our Data Source and attach it to the table.

Summary

This article has taken a detailed look at working with data binding and key-value coding with SQLite databases in a Xamarin.Mac application. First, it looked at exposing a C# class to Objective-C by using key-value coding (KVC) and key-value observing (KVO). Next, it showed how to use a KVO compliant class and Data Bind it to UI elements in Xcode's Interface Builder. The article also covered working with SQLite data via the SQLite.NET ORM and displaying that data in a Table View.

Related Links

- [MacDatabase \(sample\)](#)
- [Hello, Mac](#)
- [Data binding and key-value coding](#)
- [Standard controls](#)
- [Table views](#)
- [Outline views](#)
- [Collection views](#)
- [Key-Value Coding Programming Guide](#)
- [Introduction to Cocoa Bindings Programming Topics](#)
- [Introduction to Cocoa Bindings Reference](#)
- [NSCollectionView](#)
- [macOS Human Interface Guidelines](#)

Copy and paste in Xamarin.Mac

11/2/2020 • 31 minutes to read • [Edit Online](#)

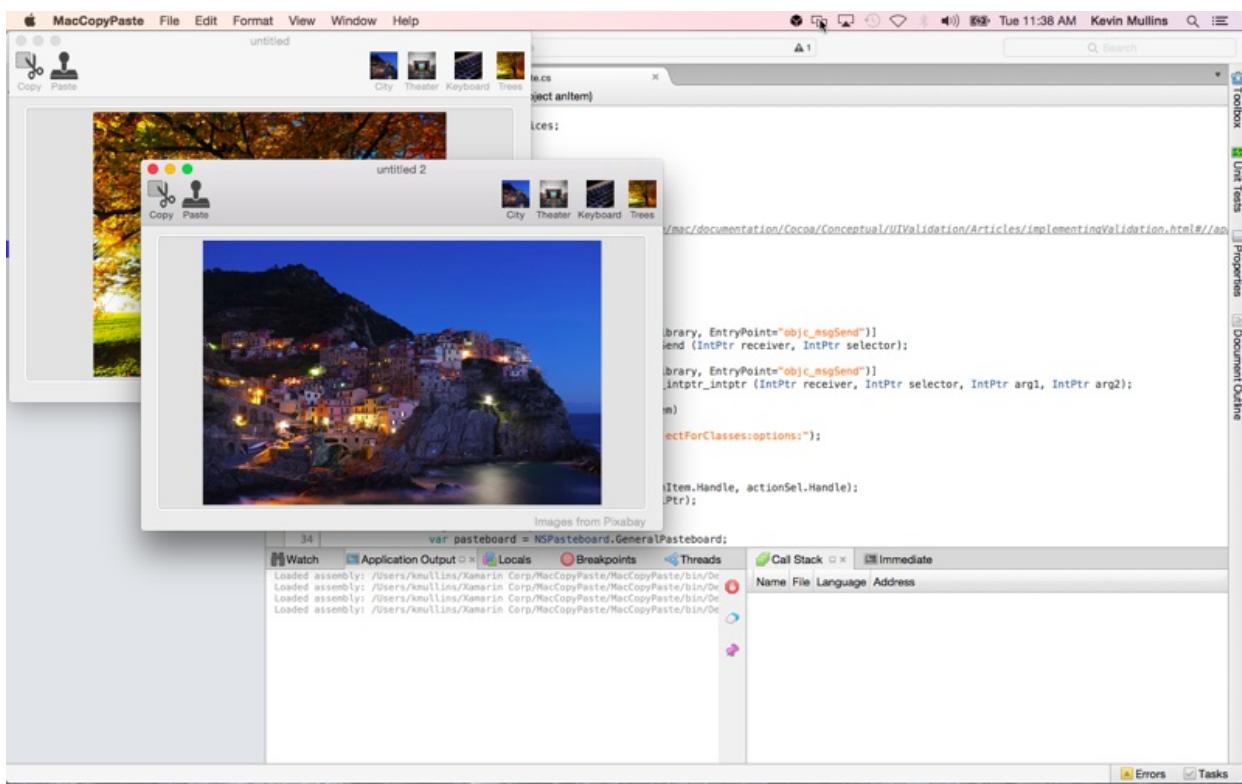
This article covers working with the pasteboard to provide copy and paste in a Xamarin.Mac application. It shows how to work with standard data types that can be shared between multiple apps and how to support custom data within a given app.

Overview

When working with C# and .NET in a Xamarin.Mac application, you have access to the same pasteboard (copy and paste) support that a developer working in Objective-C has.

In this article we will be covering the two main ways to use the pasteboard in a Xamarin.Mac app:

- Standard Data Types** - Since pasteboard operations are typically carried out between two unrelated apps, neither app knows the types of data that the other supports. To maximize the potential for sharing, the pasteboard can hold multiple representations of a given item (using a standard set of common data types), this allow the consuming app to pick the version that is best suited for its needs.
- Custom Data** - To support the copying and pasting of complex data within your Xamarin.Mac you can define a custom data type that will be handled by the pasteboard. For example, a vector drawing app that allows the user to copy and paste complex shapes that are composed of multiple data types and points.



In this article, we'll cover the basics of working with the pasteboard in a Xamarin.Mac application to support copy and paste operations. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` attributes used to wire up your C# classes to Objective-C objects and UI elements.

Getting started with the pasteboard

The pasteboard presents a standardized mechanism for exchanging data within a given application or between applications. The typical use for a pasteboard in a Xamarin.Mac application is to handle copy and paste operations, however a number of other operations are also supported (such as Drag & Drop and Application Services).

To get you off the ground quickly, we are going to start with a simple, practical introduction to using pasteboards in a Xamarin.Mac app. Later, we'll provide an in-depth explanation of how the pasteboard works and the methods used.

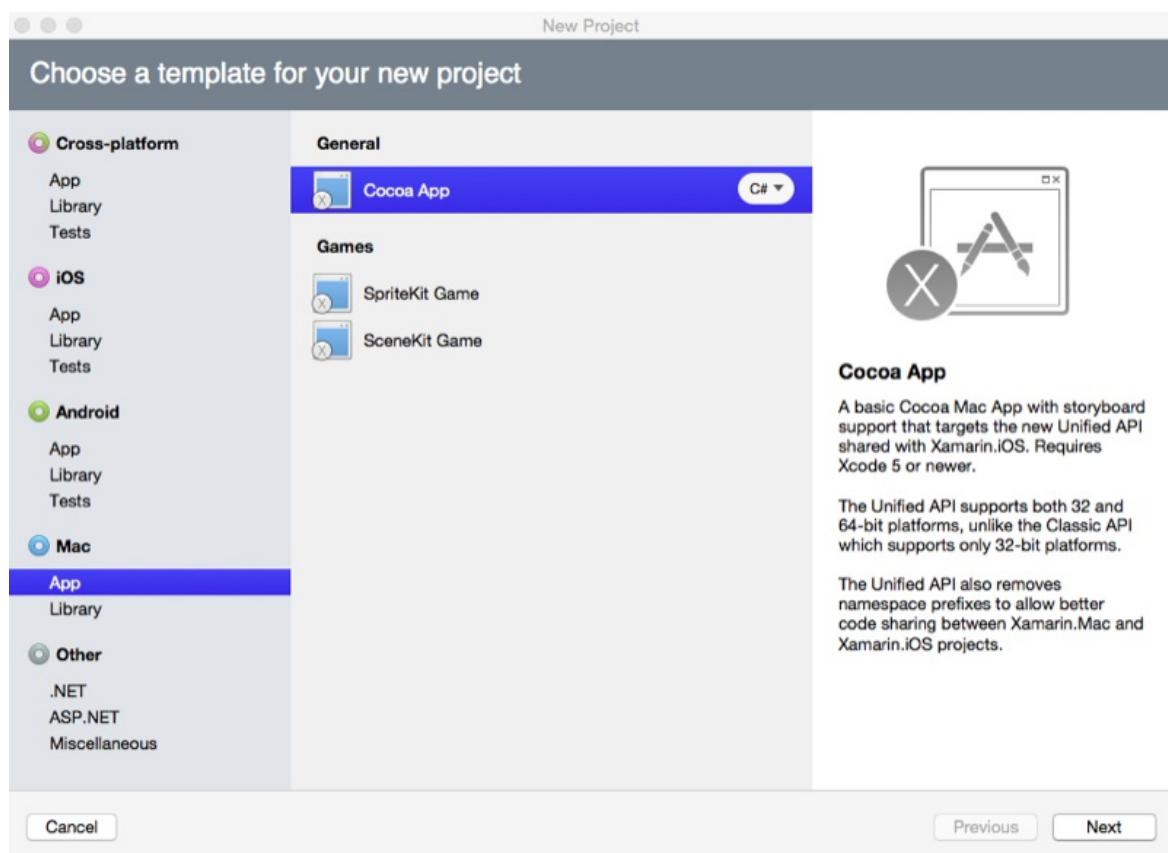
For this example, we will be creating a simple document based application that manages a window containing an image view. The user will be able to copy and paste images between documents in the app and to or from other apps or multiple windows inside the same app.

Creating the Xamarin project

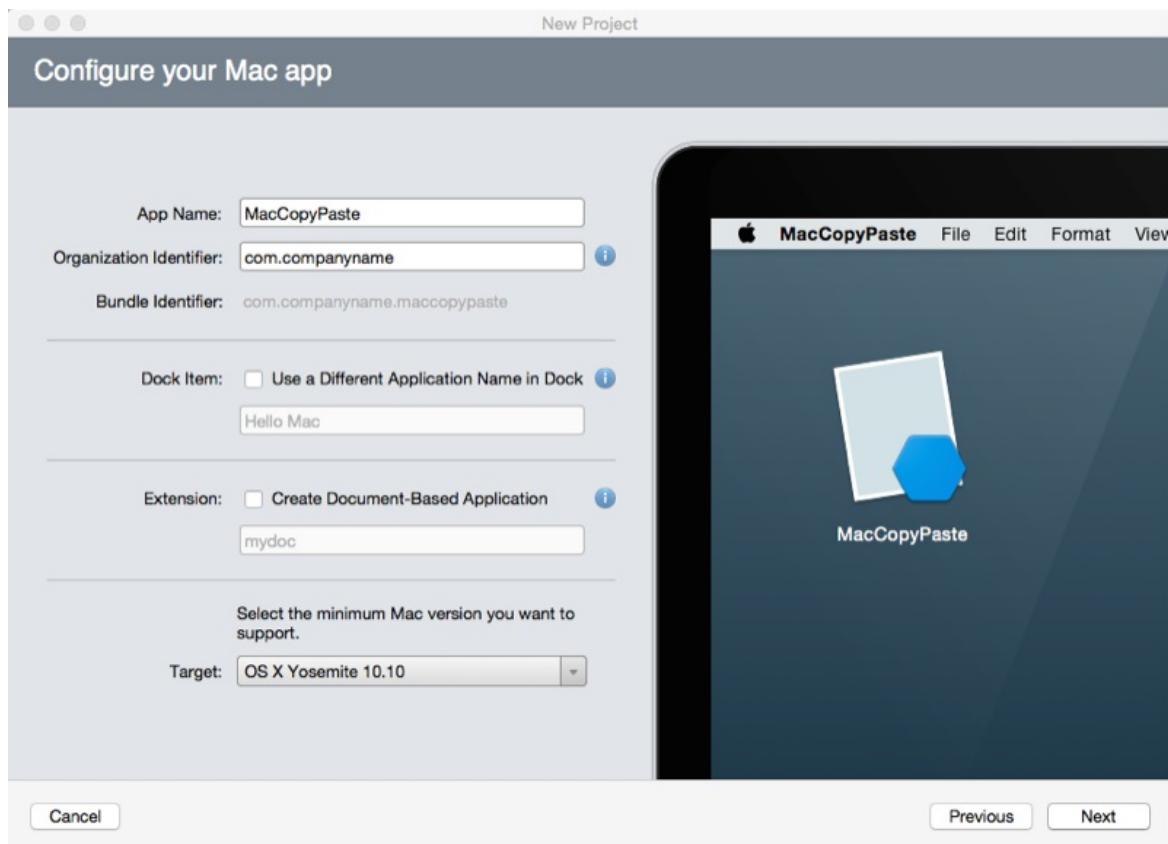
First, we are going to create a new document based Xamarin.Mac app that we will be adding copy and paste support for.

Do the following:

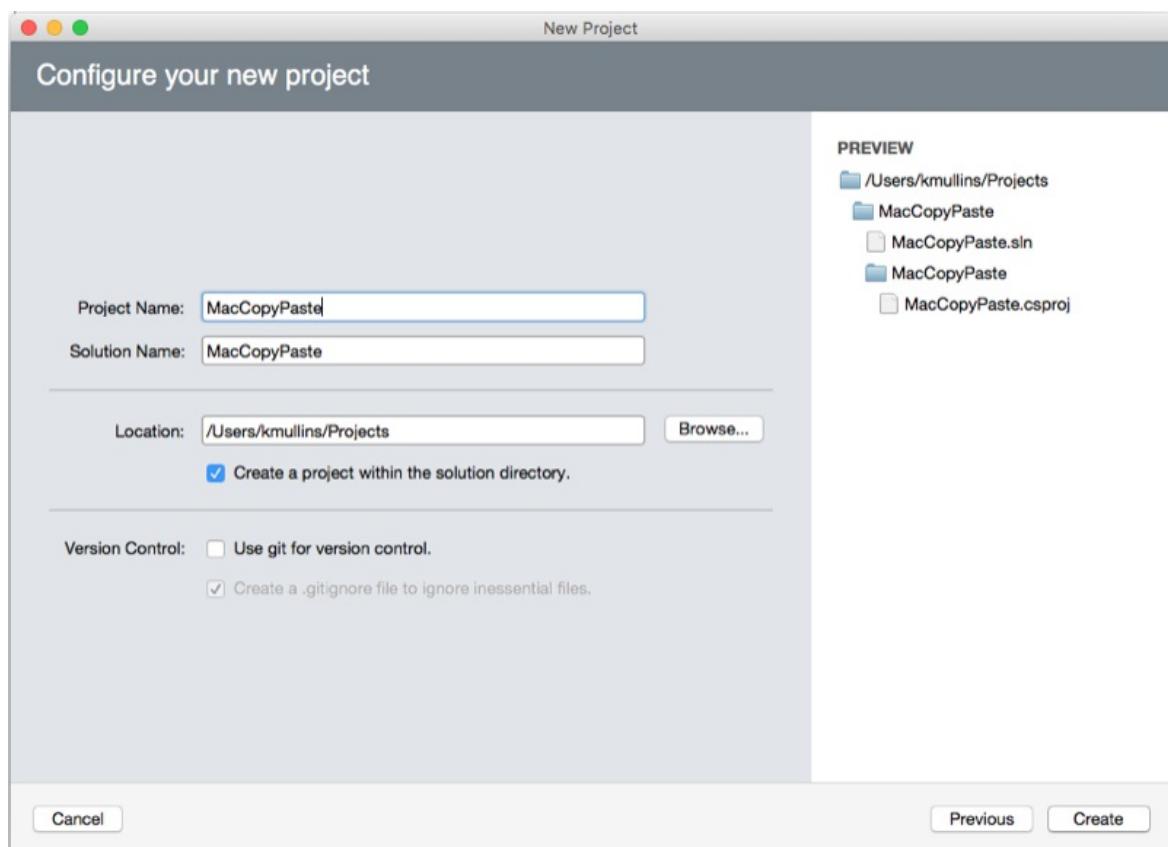
1. Start Visual Studio for Mac and click the **New Project...** link.
2. Select **Mac > App > Cocoa App**, then click the **Next** button:



3. Enter `MacCopyPaste` for the **Project Name** and keep everything else as default. Click **Next**:



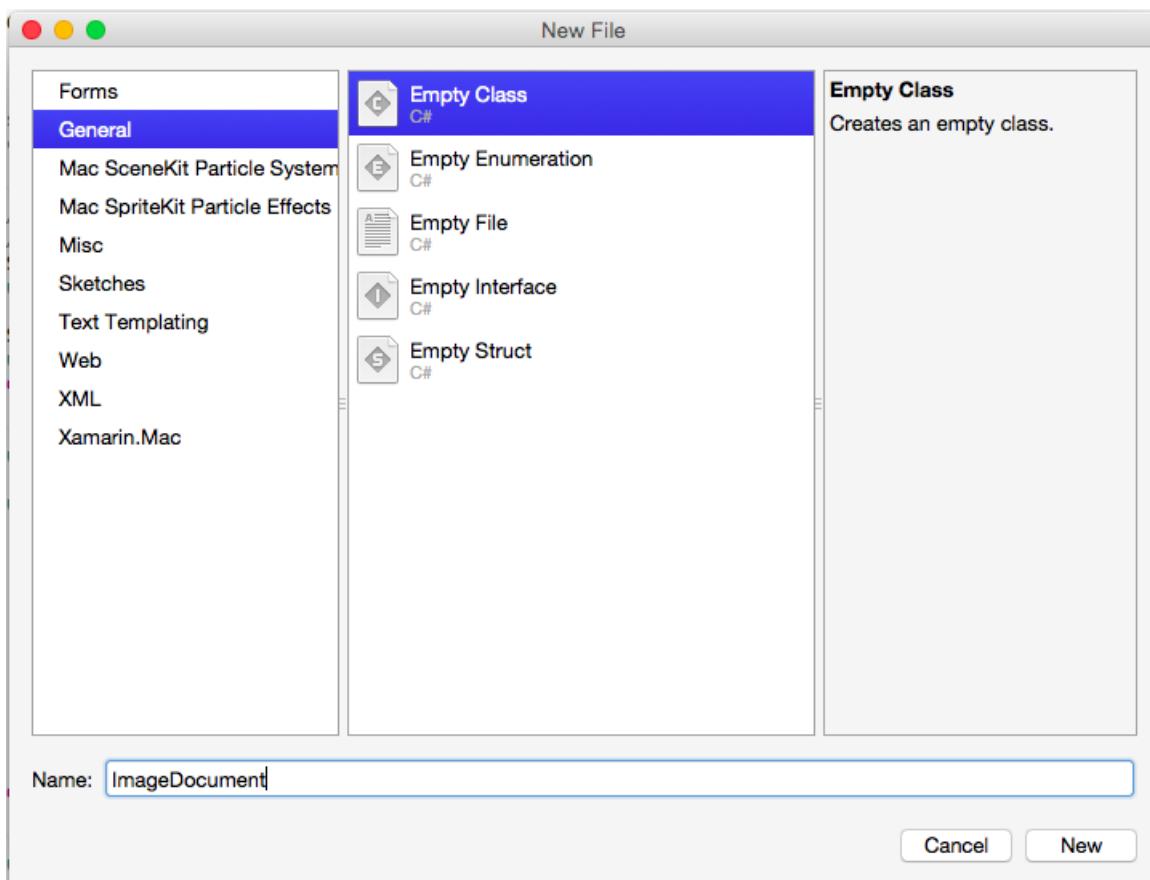
4. Click the **Create** button:



Add an NSDocument

Next we will add custom `NSDocument` class that will act as the background storage for the application's user interface. It will contain a single Image View and know how to copy an image from the view into the default pasteboard and how to take an image from the default pasteboard and display it in the Image View.

Right-click on the Xamarin.Mac project in the **Solution Pad** and select **Add > New File..**:



Enter `ImageDocument` for the **Name** and click the **New** button. Edit the `ImageDocument.cs` class and make it look like the following:

```

using System;
using AppKit;
using Foundation;
using ObjCRuntime;

namespace MacCopyPaste
{
    [Register("ImageDocument")]
    public class ImageDocument : NSDocument
    {
        #region Computed Properties
        public NSImageView ImageView {get; set;}
        public ImageInfo Info { get; set; } = new ImageInfo();

        public bool ImageAvailableOnPasteboard {
            get {
                // Initialize the pasteboard
                NSPasteboard pasteboard = NSPasteboard.GeneralPasteboard;
                Class [] classArray = { new Class ("NSImage") };

                // Check to see if an image is on the pasteboard
                return pasteboard.CanReadObjectForClasses (classArray, null);
            }
        }
        #endregion

        #region Constructor
        public ImageDocument ()
        {
        }
        #endregion

        #region Public Methods
    }
}

```

```

[Export("CopyImage")]
public void CopyImage(NSObject sender) {

    // Grab the current image
    var image = ImageView.Image;

    // Anything to process?
    if (image != null) {
        // Get the standard pasteboard
        var pasteboard = NSPasteboard.GeneralPasteboard;

        // Empty the current contents
        pasteboard.ClearContents();

        // Add the current image to the pasteboard
        pasteboard.WriteObjects (new NSImage[] {image});

        // Save the custom data class to the pasteboard
        pasteboard.WriteObjects (new ImageInfo[] { Info });

        // Using a Pasteboard Item
        NSPasteboardItem item = new NSPasteboardItem();
        string[] writableTypes = {"public.text"};

        // Add a data provider to the item
        ImageInfoDataProvider dataProvider = new ImageInfoDataProvider (Info.Name, Info.ImageType);
        var ok = item.SetDataProviderForTypes (dataProvider, writableTypes);

        // Save to pasteboard
        if (ok) {
            pasteboard.WriteObjects (new NSPasteboardItem[] { item });
        }
    }
}

[Export("PasteImage")]
public void PasteImage(NSObject sender) {

    // Initialize the pasteboard
    NSPasteboard pasteboard = NSPasteboard.GeneralPasteboard;
    Class [] classArray = { new Class ("NSImage") };

    bool ok = pasteboard.CanReadObjectForClasses (classArray, null);
    if (ok) {
        // Read the image off of the pasteboard
        NSObject [] objectsToPaste = pasteboard.ReadObjectsForClasses (classArray, null);
        NSImage image = (NSImage)objectsToPaste[0];

        // Display the new image
        ImageView.Image = image;
    }

    Class [] classArray2 = { new Class ("ImageInfo") };
    ok = pasteboard.CanReadObjectForClasses (classArray2, null);
    if (ok) {
        // Read the image off of the pasteboard
        NSObject [] objectsToPaste = pasteboard.ReadObjectsForClasses (classArray2, null);
        ImageInfo info = (ImageInfo)objectsToPaste[0];

    }
}
#endifregion
}
}

```

Let's take a look at some of the code in detail below.

The following code provides a property to test for the existence of image data on the default pasteboard, if an image is available, `true` is returned else `false`:

```
public bool ImageAvailableOnPasteboard {
    get {
        // Initialize the pasteboard
        NSPasteboard pasteboard = NSPasteboard.GeneralPasteboard;
        Class [] classArray = { new Class ("NSImage") };

        // Check to see if an image is on the pasteboard
        return pasteboard.CanReadObjectForClasses (classArray, null);
    }
}
```

The following code copies an image from the attached image view into the default pasteboard:

```
[Export("CopyImage")]
public void CopyImage(NSObject sender) {

    // Grab the current image
    var image = ImageView.Image;

    // Anything to process?
    if (image != null) {
        // Get the standard pasteboard
        var pasteboard = NSPasteboard.GeneralPasteboard;

        // Empty the current contents
        pasteboard.ClearContents();

        // Add the current image to the pasteboard
        pasteboard.WriteObjects (new NSImage[] {image});

        // Save the custom data class to the pasteboard
        pasteboard.WriteObjects (new ImageInfo[] { Info });

        // Using a Pasteboard Item
        NSPasteboardItem item = new NSPasteboardItem();
        string[] writableTypes = {"public.text"};

        // Add a data provider to the item
        ImageInfoDataProvider dataProvider = new ImageInfoDataProvider (Info.Name, Info.ImageType);
        var ok = item.SetDataProviderForTypes (dataProvider, writableTypes);

        // Save to pasteboard
        if (ok) {
            pasteboard.WriteObjects (new NSPasteboardItem[] { item });
        }
    }
}
```

And the following code pastes an image from the default pasteboard and displays it in the attached image view (if the pasteboard contains a valid image):

```

[Export("PasteImage")]
public void PasteImage(NSObject sender) {

    // Initialize the pasteboard
    NSPasteboard pasteboard = NSPasteboard.GeneralPasteboard;
    Class [] classArray = { new Class ("NSImage") };

    bool ok = pasteboard.CanReadObjectForClasses (classArray, null);
    if (ok) {
        // Read the image off of the pasteboard
        NSObject [] objectsToPaste = pasteboard.ReadObjectsForClasses (classArray, null);
        NSImage image = (NSImage)objectsToPaste[0];

        // Display the new image
        ImageView.Image = image;
    }

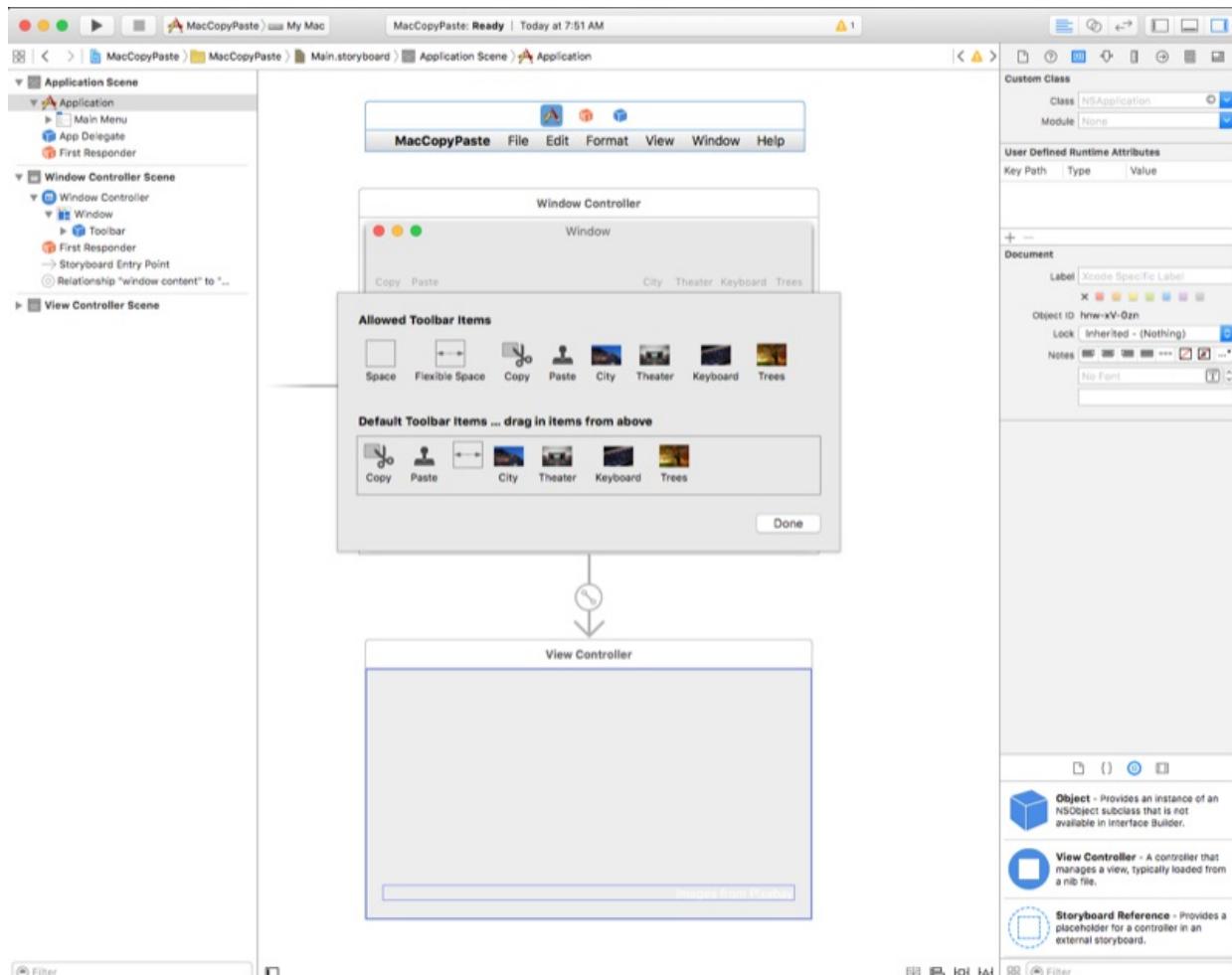
    Class [] classArray2 = { new Class ("ImageInfo") };
    ok = pasteboard.CanReadObjectForClasses (classArray2, null);
    if (ok) {
        // Read the image off of the pasteboard
        NSObject [] objectsToPaste = pasteboard.ReadObjectsForClasses (classArray2, null);
        ImageInfo info = (ImageInfo)objectsToPaste[0]
    }
}

```

With this document in place, we'll create the user interface for the Xamarin.Mac app.

Building the user interface

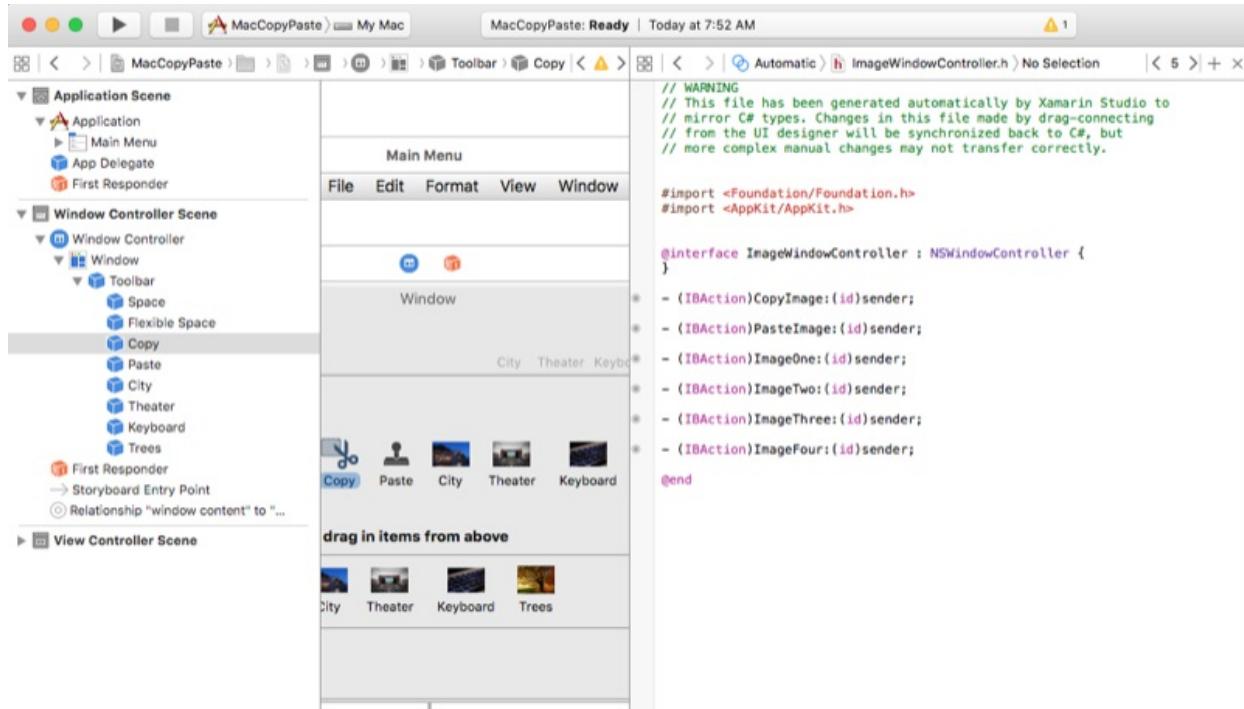
Double-click the **Main.storyboard** file to open it in Xcode. Next, add a toolbar and an image well and configure them as follows:



Add a copy and paste **Image Toolbar Item** to the left side of the toolbar. We'll be using those as shortcuts to copy and paste from the Edit menu. Next, add four **Image Toolbar Items** to the right side of the toolbar. We'll use these to populate the image well with some default images.

For more information on working with toolbars, please see our [Toolbars](#) documentation.

Next, let's expose the following outlets and actions for our toolbar items and the image well:



For more information on working with outlets and actions, please see the [Outlets and Actions](#) section of our [Hello, Mac](#) documentation.

Enabling the user interface

With our user interface created in Xcode and our UI element exposed via outlets and actions, we need to add the code to enable the UI. Double-click the **ImageWindow.cs** file in the **Solution Pad** and make it look like the following:

```
using System;
using Foundation;
using AppKit;

namespace MacCopyPaste
{
    public partial class ImageWindow : NSWindow
    {
        #region Private Variables
        ImageDocument document;
        #endregion

        #region Computed Properties
        [Export ("Document")]
        public ImageDocument Document {
            get {
                return document;
            }
            set {
                WillChangeValue ("Document");
                document = value;
                DidChangeValue ("Document");
            }
        }
    }
}
```

```

public ViewController ImageViewController {
    get { return ContentViewController as ViewController; }
}

public NSImage Image {
    get {
        return ImageViewController.Image;
    }
    set {
        ImageViewController.Image = value;
    }
}
#endregion

#region Constructor
public ImageWindow (IntPtr handle) : base (handle)
{
}
#endregion

#region Override Methods
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

    // Create a new document instance
    Document = new ImageDocument ();

    // Attach to image view
    Document.ImageView = ImageViewController.ContentView;
}
#endregion

#region Public Methods
public void CopyImage (NSObject sender)
{
    Document.CopyImage (sender);
}

public void PasteImage (NSObject sender)
{
    Document.PasteImage (sender);
}

public void ImageOne (NSObject sender)
{
    // Load image
    Image = NSImage.ImageNamed ("Image01.jpg");

    // Set image info
    Document.Info.Name = "city";
    Document.Info.ImageType = "jpg";
}

public void ImageTwo (NSObject sender)
{
    // Load image
    Image = NSImage.ImageNamed ("Image02.jpg");

    // Set image info
    Document.Info.Name = "theater";
    Document.Info.ImageType = "jpg";
}

public void ImageThree (NSObject sender)
{
    // Load image
    Image = NSImage.ImageNamed ("Image03.jpg");
}

```

```

        // Set image info
        Document.Info.Name = "keyboard";
        Document.Info.ImageType = "jpg";
    }

    public void ImageFour (NSObject sender)
    {
        // Load image
        Image = NSImage.ImageNamed ("Image04.jpg");

        // Set image info
        Document.Info.Name = "trees";
        Document.Info.ImageType = "jpg";
    }
    #endregion
}
}

```

Let's take a look at this code in detail below.

First, we expose an instance of the `ImageDocument` class that we created above:

```

private ImageDocument _document;
...

[Export ("Document")]
public ImageDocument Document {
    get { return _document; }
    set {
        WillChangeValue ("Document");
        _document = value;
        DidChangeValue ("Document");
    }
}

```

By using `Export`, `WillChangeValue` and `DidChangeValue`, we have setup the `Document` property to allow for Key-Value Coding and Data Binding in Xcode.

We also expose the Image from the image well we added to our UI in Xcode with the following property:

```

public ViewController ImageViewController {
    get { return ContentViewController as ViewController; }
}

public NSImage Image {
    get {
        return ImageViewController.Image;
    }
    set {
        ImageViewController.Image = value;
    }
}

```

When the Main Window is loaded and displayed, we create an instance of our `ImageDocument` class and attach the UI's image well to it with the following code:

```

public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

    // Create a new document instance
    Document = new ImageDocument ();

    // Attach to image view
    Document.ImageView = ImageViewController.ContentView;
}

```

Finally, in response to the user clicking on the copy and paste toolbar items, we call the instance of the `ImageDocument` class to do the actual work:

```

partial void CopyImage (NSObject sender) {
    Document.CopyImage(sender);
}

partial void PasteImage (Foundation.NSObject sender) {
    Document.PasteImage(sender);
}

```

Enabling the File and Edit menus

The last thing we need to do is enable the **New** menu item from the **File** menu (to create new instances of our main window) and to enable the **Cut**, **Copy** and **Paste** menu items from the **Edit** menu.

To enable the **New** menu item, edit the `AppDelegate.cs` file and add the following code:

```

public int UntitledWindowCount { get; set;} =1;
...

[Export ("newDocument:")]
void NewDocument (NSObject sender) {
    // Get new window
    var storyboard = NSStoryboard.FromName ("Main", null);
    var controller = storyboard.InstantiateControllerWithIdentifier ("MainWindow") as NSWindowController;

    // Display
    controller.ShowWindow(this);

    // Set the title
    controller.Window.Title = (++UntitledWindowCount == 1) ? "untitled" : string.Format ("untitled {0}", UntitledWindowCount);
}

```

For more information, please see the [Working with Multiple Windows](#) section of our [Windows](#) documentation.

To enable the **Cut**, **Copy** and **Paste** menu items, edit the `AppDelegate.cs` file and add the following code:

```

[Export("copy")]
void CopyImage (NSObject sender)
{
    // Get the main window
    var window = NSApplication.SharedApplication.KeyWindow as ImageWindow;

    // Anything to do?
    if (window == null)
        return;

    // Copy the image to the clipboard
    window.Document.CopyImage (sender);
}

[Export("cut")]
void CutImage (NSObject sender)
{
    // Get the main window
    var window = NSApplication.SharedApplication.KeyWindow as ImageWindow;

    // Anything to do?
    if (window == null)
        return;

    // Copy the image to the clipboard
    window.Document.CopyImage (sender);

    // Clear the existing image
    window.Image = null;
}

[Export("paste")]
void PasteImage (NSObject sender)
{
    // Get the main window
    var window = NSApplication.SharedApplication.KeyWindow as ImageWindow;

    // Anything to do?
    if (window == null)
        return;

    // Paste the image from the clipboard
    window.Document.PasteImage (sender);
}

```

For each menu item, we get the current, topmost, key window and cast it to our `ImageWindow` class:

```
var window = NSApplication.SharedApplication.KeyWindow as ImageWindow;
```

From there we call the `ImageDocument` class instance of that window to handle the copy and paste actions. For example:

```
window.Document.CopyImage (sender);
```

We only want **Cut**, **Copy** and **Paste** menu items to be accessible if there is image data on the default pasteboard or in the image well of the current active window.

Let's add a `EditMenuDelegate.cs` file to the `Xamarin.Mac` project and make it look like the following:

```

using System;
using AppKit;

namespace MacCopyPaste
{
    public class EditMenuDelegate : NSMenuDelegate
    {
        #region Override Methods
        public override void MenuWillHighlightItem (NSMenu menu, NSMenuItem item)
        {
        }

        public override void NeedsUpdate (NSMenu menu)
        {
            // Get list of menu items
            NSMenuItem[] Items = menu.ItemArray ();

            // Get the key window and determine if the required images are available
            var window = NSApplication.SharedApplication.KeyWindow as ImageWindow;
            var hasImage = (window != null) && (window.Image != null);
            var hasImageOnPasteboard = (window != null) && window.Document.ImageAvailableOnPasteboard;

            // Process every item in the menu
            foreach(NSMenuItem item in Items) {
                // Take action based on the menu title
                switch (item.Title) {
                    case "Cut":
                    case "Copy":
                    case "Delete":
                        // Only enable if there is an image in the view
                        item.Enabled = hasImage;
                        break;
                    case "Paste":
                        // Only enable if there is an image on the pasteboard
                        item.Enabled = hasImageOnPasteboard;
                        break;
                    default:
                        // Only enable the item if it has a sub menu
                        item.Enabled = item.HasSubmenu;
                        break;
                }
            }
        }
        #endregion
    }
}

```

Again, we get the current, topmost window and use its `ImageDocument` class instance to see if the required image data exists. Then we use the `MenuWillHighlightItem` method to enable or disable each item based on this state.

Edit the `AppDelegate.cs` file and make the `DidFinishLaunching` method look like the following:

```

public override void DidFinishLaunching (NSNotification notification)
{
    // Disable automatic item enabling on the Edit menu
    EditMenu.AutoEnablesItems = false;
    EditMenu.Delegate = new EditMenuDelegate ();
}

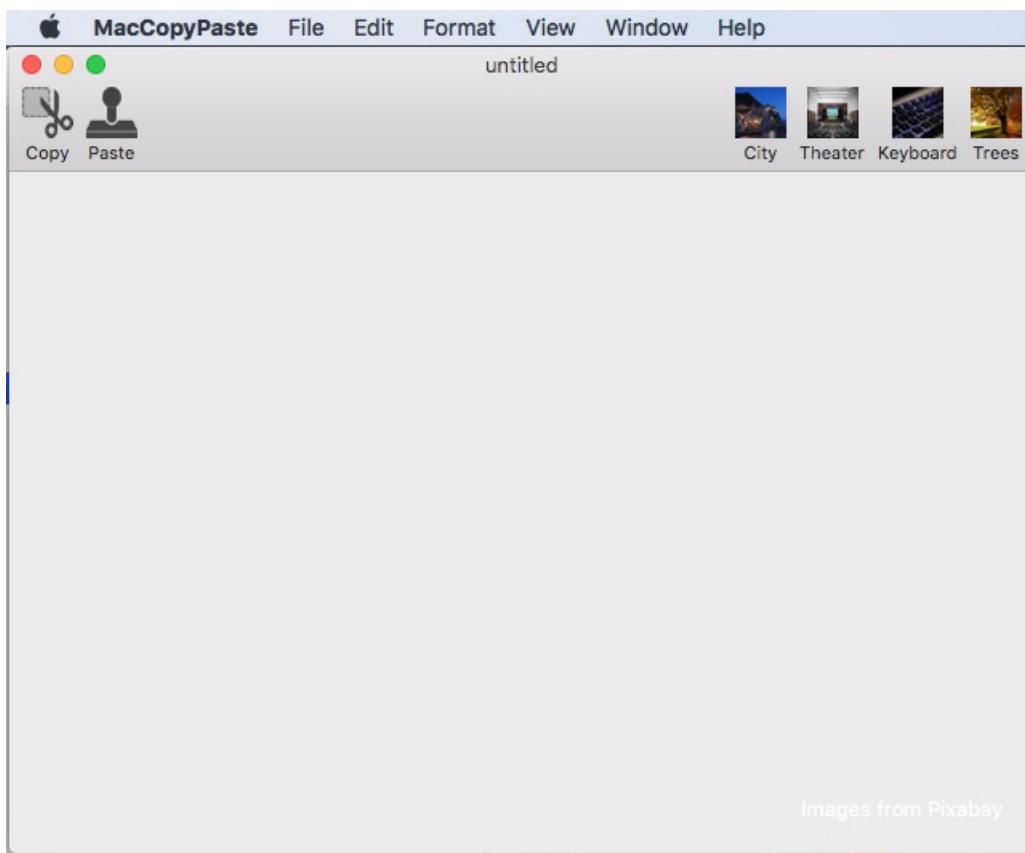
```

First, we disable the automatic enabling and disabling of menu items in the Edit menu. Next, we attach an instance of the `EditMenuDelegate` class that we created above.

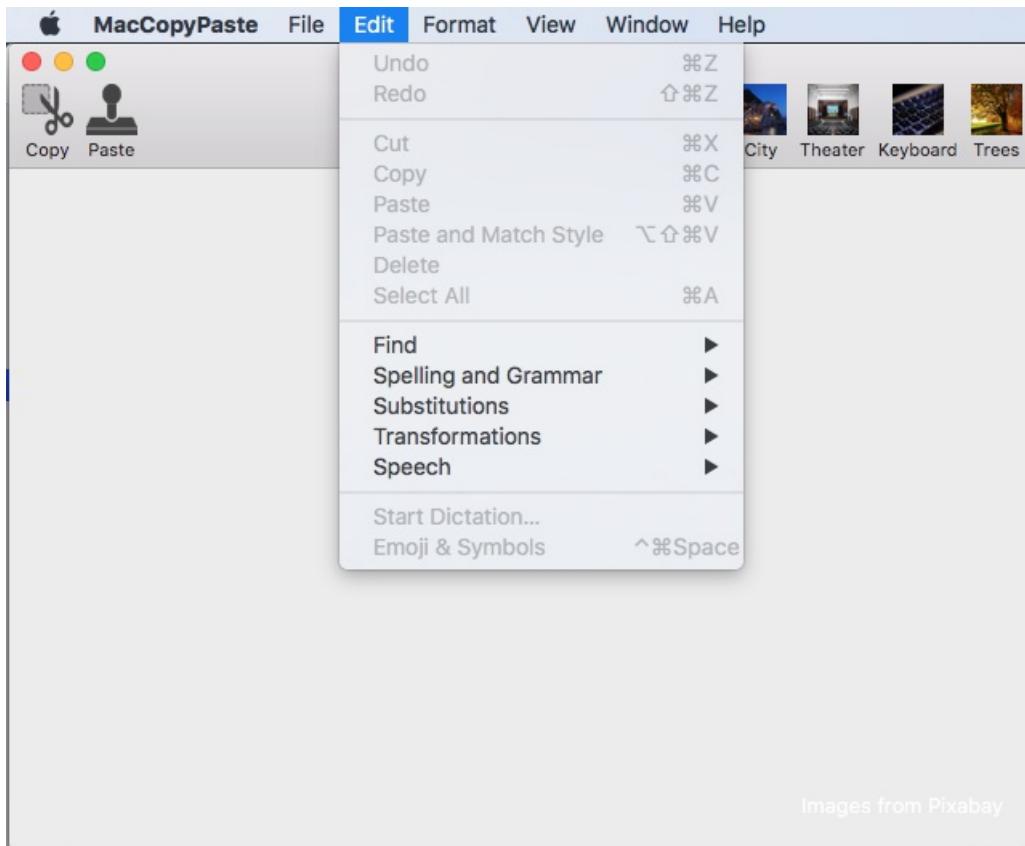
For more information, please see our [Menus](#) documentation.

Testing the app

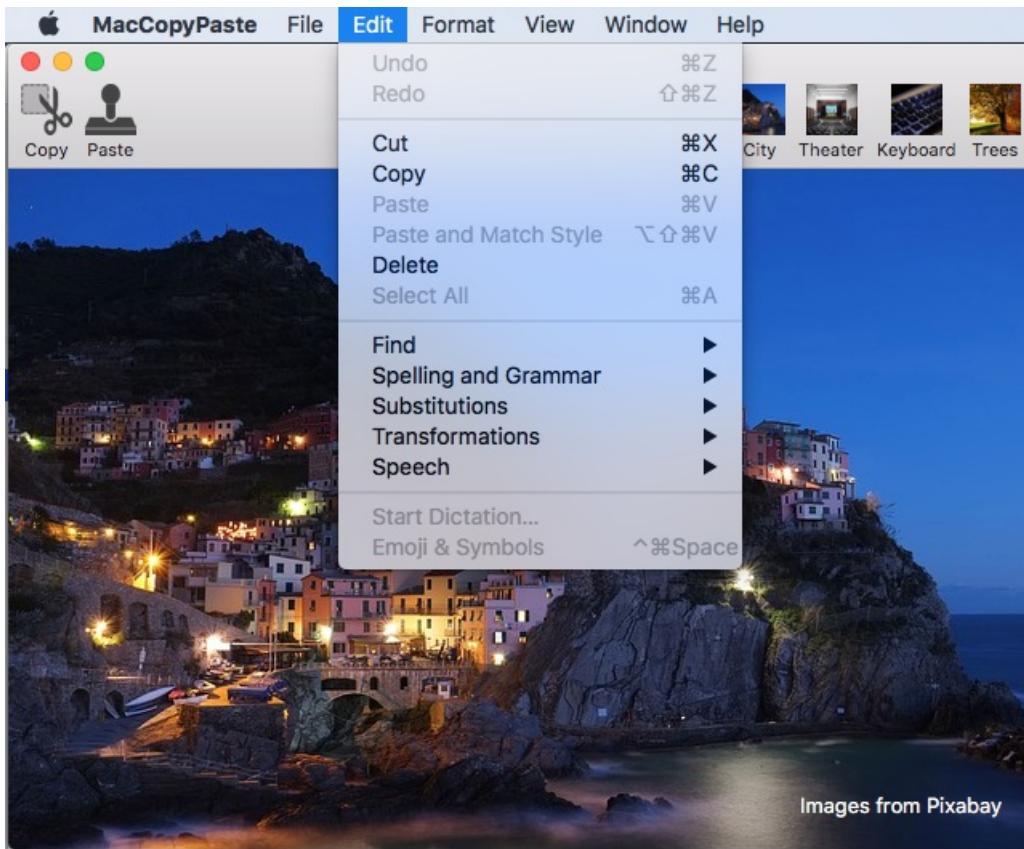
With everything in place, we are ready to test the application. Build and run the app and the main interface is displayed:



If you open the Edit menu, note that **Cut**, **Copy** and **Paste** are disabled because there is no image in the image well or in the default pasteboard:



If you add an image to the image well and reopen the Edit menu, the items will now be enabled:



If you copy the image and select **New** from the file menu, you can paste that image into the new window:



In the following sections, we'll take a detailed look at working with the pasteboard in a Xamarin.Mac application.

About the pasteboard

In macOS (formerly known as OS X) the pasteboard (`NSPasteboard`) provide support for several server processes such as Copy & Paste, Drag & Drop and Application Services. In the following sections, we'll take a closer look at several key pasteboard concepts.

What is a pasteboard?

The `NSPasteboard` class provides a standardized mechanism for exchanging information between applications or within a given app. The main function of a pasteboard is for handling copy and paste operations:

1. When the user selects an item in an app and uses the **Cut** or **Copy** menu item, one or more representations of the selected item are placed on the pasteboard.
2. When the user uses the **Paste** menu item (within the same app or a different one), the version of the data that it can handle is copied from the pasteboard and added to the app.

Less obvious pasteboard uses include find, drag, drag and drop, and application services operations:

- When the user initiates a drag operation, the drag data is copied to the pasteboard. If the drag operation ends with a drop onto another app, that app copies the data from the pasteboard.
- For translation services, the data to be translated is copied to the pasteboard by the requesting app. The application service, retrieves the data from pasteboard, does the translation, then pastes the data back on the pasteboard.

In their simplest form, pasteboards are used to move data inside a given app or between apps and therefore exist in a special global memory area outside of the app's process. While the concepts of the pasteboards are easily grasps, there are several more complex details that must be considered. These will be covered in detail below.

Named pasteboards

A pasteboard can be public or private and may be used for a variety of purposes within an application or between multiple apps. macOS provides several standard pasteboards, each with a specific, well-defined usage:

- `NSGeneralPboard` - The default pasteboard for **Cut**, **Copy** and **Paste** operations.
- `NSRulerPboard` - Supports **Cut**, **Copy** and **Paste** operations on **Rulers**.
- `NSFontPboard` - Supports **Cut**, **Copy** and **Paste** operations on `NSFont` objects.
- `NSFindPboard` - Supports application-specific find panels that can share search text.
- `NSDragPboard` - Supports **Drag & Drop** operations.

For most situations, you'll use one of the system defined pasteboards. But there might be situations that require you to create your own pasteboards. In these situations, you can use the `FromName (string name)` method of the `NSPasteboard` class to create a custom pasteboard with the given name.

Optionally, you can call the `CreateWithUniqueName` method of the `NSPasteboard` class to create a uniquely named pasteboard.

Pasteboard items

Each piece of data that an application writes to a pasteboard is considered a *Pasteboard Item* and a pasteboard can hold multiple items at the same time. In this way, an app can write multiple versions of the data being copied to a pasteboard (for example, plain text and formatted text) and the retrieving app can read off only the data that it can process (such as the plain text only).

Data representations and uniform type identifiers

Pasteboard operations typically take place between two (or more) applications that have no knowledge of each other or the types of data that each can handle. As stated in the section above, to maximize the potential for sharing information, a pasteboard can hold multiple representations of the data being copied and pasted.

Each representation is identified via a Uniform Type Identifier (UTI), which is nothing more than a simple string that uniquely identifies the type of date being presented (for more information, please see Apple's [Uniform Type](#)

[Identifiers Overview](#) documentation).

If you are creating a custom data type (for example, a drawing object in a vector drawing app), you can create your own UTI to uniquely identify it in copy and paste operations.

When an app prepares to paste data copied from a pasteboard, it must find the representation that best fits its abilities (if any exists). Typically this will be the richest type available (for example formatted text for a word processing app), falling back to the simplest forms available as required (plain text for a simple text editor).

Promised data

Generally speaking, you should provide as many representations of the data being copied as possible to maximize sharing between apps. However, because of time or memory constraints, it might be impractical to actually write each data type into the pasteboard.

In this situation, you can place the first data representation on the pasteboard and the receiving app can request a different representation, that can be generated on-the-fly just before the paste operation.

When you place the initial item in the pasteboard, you will specify that one or more of the other representations available are provided by an object that conforms to the `NSPasteboardItemDataProvider` interface. These objects will provide the extra representations on demand, as requested by the receiving app.

Change count

Each pasteboard maintains a *Change Count* that increments each time a new owner is declared. An app can determine if the pasteboard's contents have changed since the last time it examined it by checking the value of the Change Count.

Use the `ChangeCount` and `ClearContents` methods of the `NSPasteboard` class to modify a given pasteboard's Change Count.

Copying data to a pasteboard

You perform a copy operation by first accessing a pasteboard, clearing any existing contents and writing as many representations of the data as are required to the pasteboard.

For example:

```
// Get the standard pasteboard
var pasteboard = NSPasteboard.GeneralPasteboard;

// Empty the current contents
pasteboard.ClearContents();

// Add the current image to the pasteboard
pasteboard.WriteObjects (new NSImage[] {image});
```

Typically, you'll just be writing to the general pasteboard, as we have done in the example above. Any object that you send to the `WriteObjects` method *must* conform to the `INSPasteboardWriting` interface. Several, built-in classes (such as `NSString`, `NSImage`, `NSURL`, `NSColor`, `NSAttributedString`, and `NSPasteboardItem`) automatically conform to this interface.

If you are writing a custom data class to the pasteboard it must conform to the `INSPasteboardWriting` interface or be wrapped in an instance of the `NSPasteboardItem` class (see the [Custom Data Types](#) section below).

Reading data from a pasteboard

As stated above, to maximize the potential for sharing data between apps, multiple representations of the copied

data may be written to the pasteboard. It is up to the receiving app to select the richest version possible for its capabilities (if any exists).

Simple paste operation

You read data from the pasteboard using the `ReadObjectsForClasses` method. It will require two parameters:

1. An array of `NSObject` based class types that you want to read from the pasteboard. You should order this with the most desired data type first, with any remaining types in decreasing preference.
2. A dictionary containing additional constraints (such as limiting to specific URL content types) or an empty dictionary if no further constraints are required.

The method returns an array of items that meet the criteria that we passed in and therefore contains at most the same number of data types that are requested. It is also possible that none of the requested types are present and an empty array will be returned.

For example, the following code checks to see if an `NSImage` exists in the general pasteboard and displays it in an image well if it does:

```
[Export("PasteImage")]
public void PasteImage(NSObject sender) {

    // Initialize the pasteboard
    NSPasteboard pasteboard = NSPasteboard.GeneralPasteboard;
    Class [] classArray = { new Class ("NSImage") };

    bool ok = pasteboard.CanReadObjectForClasses (classArray, null);
    if (ok) {
        // Read the image off of the pasteboard
        NSObject [] objectsToPaste = pasteboard.ReadObjectsForClasses (classArray, null);
        NSImage image = (NSImage)objectsToPaste[0];

        // Display the new image
        ImageView.Image = image;
    }

    Class [] classArray2 = { new Class ("ImageInfo") };
    ok = pasteboard.CanReadObjectForClasses (classArray2, null);
    if (ok) {
        // Read the image off of the pasteboard
        NSObject [] objectsToPaste = pasteboard.ReadObjectsForClasses (classArray2, null);
        ImageInfo info = (ImageInfo)objectsToPaste[0];
    }
}
```

Requesting multiple data types

Based on the type of Xamarin.Mac application being created, it may be able to handle multiple representations of the data being pasted. In this situation, there are two scenarios for retrieving data from the pasteboard:

1. Make a single call to the `ReadObjectsForClasses` method and providing an array of all of the representations that you desire (in the preferred order).
2. Make multiple calls to the `ReadObjectsForClasses` method asking for a different array of types each time.

See the **Simple Paste Operation** section above for more details on retrieving data from a pasteboard.

Checking for existing data types

There are times that you might want to check if a pasteboard contains a given data representation without actually reading the data from the pasteboard (such as enabling the **Paste** menu item only when valid data exists).

Call the `CanReadObjectForClasses` method of the pasteboard to see if it contains a given type.

For example, the following code determines if the general pasteboard contains a `NSImage` instance:

```
public bool ImageAvailableOnPasteboard {
    get {
        // Initialize the pasteboard
        NSPasteboard pasteboard = NSPasteboard.GeneralPasteboard;
        Class [] classArray = { new Class ("NSImage") };

        // Check to see if an image is on the pasteboard
        return pasteboard.CanReadObjectForClasses (classArray, null);
    }
}
```

Reading urls from the pasteboard

Based on the function of a given Xamarin.Mac app, it may be required to read URLs from a pasteboard, but only if they meet a given set of criteria (such as pointing to files or URLs of a specific data type). In this situation, you can specify additional search criteria using the second parameter of the `CanReadObjectForClasses` or `ReadObjectsForClasses` methods.

Custom data types

There are times when you will need to save your own custom types to the pasteboard from a Xamarin.Mac app. For example, a vector drawing app that allows the user to copy and paste drawing objects.

In this situation, you'll need to design your data custom class so that it inherits from `NSObject` and it conforms to a few interfaces (`INSCoding`, `INSPasteboardWriting` and `INSPasteboardReading`). Optionally, you can use a `NSPasteboardItem` to encapsulate the data to be copied or pasted.

Both of these options will be covered in detail below.

Using a custom class

In this section we will be expanding on the simple example app that we created at the start of this document and adding a custom class to track information about the image that we are copying and pasting between windows.

Add a new class to the project and call it `ImageInfo.cs`. Edit the file and make it look like the following:

```
using System;
using AppKit;
using Foundation;

namespace MacCopyPaste
{
    [Register("ImageInfo")]
    public class ImageInfo : NSObject, INSCoding, INSPasteboardWriting, INSPasteboardReading
    {
        #region Computed Properties
        [Export("name")]
        public string Name { get; set; }

        [Export("imageType")]
        public string ImageType { get; set; }
        #endregion

        #region Constructors
        [Export ("init")]
        public ImageInfo ()
        {
        }
    }
}
```

```

public ImageInfo (IntPtr p) : base (p)
{
}

[Export ("initWithCoder:")]
public ImageInfo(NSCoder decoder) {

    // Decode data
    NSString name = decoder.DecodeObject("name") as NSString;
    NSString type = decoder.DecodeObject("imageType") as NSString;

    // Save data
    Name = name.ToString();
    ImageType = type.ToString ();
}
#endregion

#region Public Methods
[Export ("encodeWithCoder:")]
public void EncodeTo (NSCoder encoder) {

    // Encode data
    encoder.Encode(new NSString(Name), "name");
    encoder.Encode(new NSString(ImageType), "imageType");
}

[Export ("writableTypesForPasteboard:")]
public virtual string[] GetWritableTypesForPasteboard (NSPasteboard pasteboard) {
    string[] writableTypes = {"com.xamarin.image-info", "public.text"};
    return writableTypes;
}

[Export ("pasteboardPropertyListForType:")]
public virtual NSObject GetPasteboardPropertyListForType (string type) {

    // Take action based on the requested type
    switch (type) {
        case "com.xamarin.image-info":
            return NSKeyedArchiver.ArchivedDataWithRootObject(this);
        case "public.text":
            return new NSString(string.Format("{0}.{1}", Name, ImageType));
    }

    // Failure, return null
    return null;
}

[Export ("readableTypesForPasteboard:")]
public static string[] GetReadableTypesForPasteboard (NSPasteboard pasteboard){
    string[] readableTypes = {"com.xamarin.image-info", "public.text"};
    return readableTypes;
}

[Export ("readingOptionsForType:pasteboard:")]
public static NSPasteboardReadingOptions GetReadingOptionsForType (string type, NSPasteboard
pasteboard) {

    // Take action based on the requested type
    switch (type) {
        case "com.xamarin.image-info":
            return NSPasteboardReadingOptions.AsKeyedArchive;
        case "public.text":
            return NSPasteboardReadingOptionsAsString;
    }

    // Default to property list
    return NSPasteboardReadingOptions.AsPropertyList;
}

```

```

[Export ("initWithPasteboardPropertyList:ofType:")]
public NSObject InitWithPasteboardPropertyList (NSObject propertyList, string type) {

    // Take action based on the requested type
    switch (type) {
        case "com.xamarin.image-info":
            return new ImageInfo();
        case "public.text":
            return new ImageInfo();
    }

    // Failure, return null
    return null;
}
#endregion
}
}

```

In the following sections we'll take a detailed look at this class.

Inheritance and interfaces

Before a custom data class can be written to or read from a pasteboard, it must conform to the `INSPastebaordWriting` and `INSPasteboardReading` interfaces. In addition, it must inherit from `NSObject` and also conform to the `INSCoding` interface:

```

[Register("ImageInfo")]
public class ImageInfo : NSObject, NSCoding, INSPasteboardWriting, INSPasteboardReading
...

```

The class must also be exposed to Objective-C using the `Register` directive and it must expose any required properties or methods using `Export`. For example:

```

[Export("name")]
public string Name { get; set; }

[Export("imageType")]
public string ImageType { get; set; }

```

We are exposing the two fields of data that this class will contain - the image's name and its type (jpg, png, etc.).

For more information, see the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) documentation, it explains the `Register` and `Export` attributes used to wire up your C# classes to Objective-C objects and UI elements.

Constructors

Two constructors (properly exposed to Objective-C) will be required for our custom data class so that it can be read from a pasteboard:

```

[Export ("init")]
public ImageInfo ()
{
}

[Export ("initWithCoder:")]
public ImageInfo(NSCoder decoder) {

    // Decode data
    NSString name = decoder.DecodeObject("name") as NSString;
    NSString type = decoder.DecodeObject("imageType") as NSString;

    // Save data
    Name = name.ToString();
    ImageType = type.ToString ();
}

```

First, we expose the *empty* constructor under the default Objective-C method of `init`.

Next, we expose a `NSCoding` compliant constructor that will be used to create a new instance of the object from the pasteboard when pasting under the exported name of `initWithCoder`.

This constructor takes an `NSCoder` (as created by a `NSKeyedArchiver` when written to the pasteboard), extracts the key/value paired data and saves it to the property fields of the data class.

Writing to the pasteboard

By conforming to the `INSPasteboardWriting` interface, we need to expose two methods, and optionally a third method, so that the class can be written to the pasteboard.

First, we need to tell the pasteboard what data type representations that the custom class can be written to:

```

[Export ("writableTypesForPasteboard:")]
public virtual string[] GetWritableTypesForPasteboard (NSPasteboard pasteboard) {
    string[] writableTypes = {"com.xamarin.image-info", "public.text"};
    return writableTypes;
}

```

Each representation is identified via a Uniform Type Identifier (UTI), which is nothing more than a simple string that uniquely identifies the type of data being presented (for more information, please see Apple's [Uniform Type Identifiers Overview](#) documentation).

For our custom format, we are creating our own UTI: "com.xamarin.image-info" (note that is in reverse notation just like an App Identifier). Our class is also capable of writing a standard string to the pasteboard (`public.text`).

Next, we need to create the object in the requested format that actually gets written to the pasteboard:

```
[Export ("pasteboardPropertyListForType")]
public virtual NSObject GetPasteboardPropertyListForType (string type) {

    // Take action based on the requested type
    switch (type) {
        case "com.xamarin.image-info":
            return NSKeyedArchiver.ArchivedDataWithRootObject(this);
        case "public.text":
            return new NSString(string.Format("{0}.{1}", Name, ImageType));
    }

    // Failure, return null
    return null;
}
```

For the `public.text` type, we are returning a simple, formatted `NSString` object. For the custom `com.xamarin.image-info` type, we are using a `NSKeyedArchiver` and the `NSCoder` interface to encode the custom data class to a key/value paired archive. We will need to implement the following method to actually handle the encoding:

```
[Export ("encodeWithCoder")]
public void EncodeTo (NSCoder encoder) {

    // Encode data
    encoder.Encode(new NSString(Name),"name");
    encoder.Encode(new NSString(ImageType),"imageType");
}
```

The individual key/value pairs are written to the encoder and will be decoded using the second constructor that we added above.

Optionally, we can include the following method to define any options when writing data to the pasteboard:

```
[Export ("writingOptionsForType:pasteboard:"), CompilerGenerated]
public virtual NSPasteboardWritingOptions GetWritingOptionsForType (string type, NSPasteboard pasteboard) {
    return NSPasteboardWritingOptions.WritingPromised;
}
```

Currently only the `WritingPromised` option is available and should be used when a given type is only promised and not actually written to the pasteboard. For more information, please see the [Promised Data](#) section above.

With these methods in place, the following code can be used to write our custom class to the pasteboard:

```
// Get the standard pasteboard
var pasteboard = NSPasteboard.GeneralPasteboard;

// Empty the current contents
pasteboard.ClearContents();

// Add info to the pasteboard
pasteboard.WriteObjects (new ImageInfo[] { Info });
```

Reading from the pasteboard

By conforming to the `INSPasteboardReading` interface, we need to expose three methods so that the custom data class can be read from the pasteboard.

First, we need to tell the pasteboard what data type representations that the custom class can read from the clipboard:

```
[Export ("readableTypesForPasteboard")]
public static string[] GetReadableTypesForPasteboard (NSPasteboard pasteboard){
    string[] readableTypes = {"com.xamarin.image-info", "public.text"};
    return readableTypes;
}
```

Again, these are defined as simple UTIs and are the same types that we defined in the [Writing to the Pasteboard](#) section above.

Next, we need to tell the pasteboard *how* each of the UTI types will be read using the following method:

```
[Export ("readingOptionsForType:pasteboard")]
public static NSPasteboardReadingOptions GetReadingOptionsForType (string type, NSPasteboard pasteboard) {

    // Take action based on the requested type
    switch (type) {
        case "com.xamarin.image-info":
            return NSPasteboardReadingOptions.AsKeyedArchive;
        case "public.text":
            return NSPasteboardReadingOptions.AsString;
    }

    // Default to property list
    return NSPasteboardReadingOptions.AsPropertyList;
}
```

For the `com.xamarin.image-info` type, we are telling the pasteboard to decode the key/value pair that we created with the `NSKeyedArchiver` when writing the class to the pasteboard by calling the `initWithCoder:` constructor that we added to the class.

Finally, we need to add the following method to read the other UTI data representations from the pasteboard:

```
[Export ("initWithPasteboardPropertyList:fType")]
public NSObject InitWithPasteboardPropertyList (NSObject propertyList, string type) {

    // Take action based on the requested type
    switch (type) {
        case "public.text":
            return new ImageInfo();
    }

    // Failure, return null
    return null;
}
```

With all these methods in place, the custom data class can be read from the pasteboard using the following code:

```

// Initialize the pasteboard
NSPasteboard pasteboard = NSPasteboard.GeneralPasteboard;
var classArrayPtrs = new [] { Class.GetHandle (typeof(ImageInfo)) };
NSArray classArray = NSArray.FromIntPtrs (classArrayPtrs);

// NOTE: Sending messages directly to the base Objective-C API because of this defect:
// https://bugzilla.xamarin.com/show_bug.cgi?id=31760
// Check to see if image info is on the pasteboard
ok = bool_objc_msgSend_IntPtr_IntPtr (pasteboard.Handle, Selector.GetHandle
("canReadObjectForClasses:options:"), classArray.Handle, IntPtr.Zero);

if (ok) {
    // Read the image off of the pasteboard
    NSObject [] objectsToPaste = NSArray.ArrayFromHandle<Foundation.NSObject>
(IntPtr_objc_msgSend_IntPtr_IntPtr (pasteboard.Handle, Selector.GetHandle
("readObjectsForClasses:options:"), classArray.Handle, IntPtr.Zero));
    ImageInfo info = (ImageInfo)objectsToPaste[0];
}

```

Using a `NSPasteboardItem`

There might be times when you need to write custom items to the pasteboard that do not warrant the creation of a custom class or you want to provide data in a common format, only as required. For these situations, you can use a `NSPasteboardItem`.

A `NSPasteboardItem` provides fine-grained control over the data that is written to the pasteboard and is designed for temporary access - it should be disposed of after it has been written to the pasteboard.

Writing data

To write your custom data to an `NSPasteboardItem` you'll need to provide a custom `NSPasteboardItemDataProvider`. Add a new class to the project and call it `ImageInfoDataProvider.cs`. Edit the file and make it look like the following:

```

using System;
using AppKit;
using Foundation;

namespace MacCopyPaste
{
    [Register("ImageInfoDataProvider")]
    public class ImageInfoDataProvider : NSPasteboardItemDataProvider
    {
        #region Computed Properties
        public string Name { get; set; }
        public string ImageType { get; set; }
        #endregion

        #region Constructors
        [Export ("init")]
        public ImageInfoDataProvider ()
        {
        }

        public ImageInfoDataProvider (string name, string imageType)
        {
            // Initialize
            this.Name = name;
            this.ImageType = imageType;
        }

        protected ImageInfoDataProvider (NSObjectFlag t){
        }

        protected internal ImageInfoDataProvider (IntPtr handle){

        }
        #endregion

        #region Override Methods
        [Export ("pasteboardFinishedWithDataProvider:")]
        public override void FinishedWithDataProvider (NSPasteboard pasteboard)
        {

        }

        [Export ("pasteboard:item:provideDataForType:")]
        public override void ProvideDataForType (NSPasteboard pasteboard, NSPasteboardItem item, string type)
        {

            // Take action based on the type
            switch (type) {
                case "public.text":
                    // Encode the data to string
                    item.SetStringForType(string.Format("{0}.{1}", Name, ImageType),type);
                    break;
            }
        }
        #endregion
    }
}

```

As we did with the custom data class, we need to use the `Register` and `Export` directives to expose it to Objective-C. The class must inherit from `NSPasteboardItemDataProvider` and must implement the `FinishedWithDataProvider` and `ProvideDataForType` methods.

Use the `ProvideDataForType` method to provide the data that will be wrapped in the `NSPasteboardItem` as

follows:

```
[Export ("pasteboard:item:provideDataForType:")]
public override void ProvideDataForType (NSPasteboard pasteboard, NSPasteboardItem item, string type)
{

    // Take action based on the type
    switch (type) {
        case "public.text":
            // Encode the data to string
            item.SetStringForType(string.Format("{0}.{1}", Name, ImageType),type);
            break;
    }

}
```

In this case, we are storing two pieces of information about our image (Name and ImageType) and writing those to a simple string (`public.text`).

Type write the data to the pasteboard, use the following code:

```
// Get the standard pasteboard
var pasteboard = NSPasteboard.GeneralPasteboard;

// Using a Pasteboard Item
NSPasteboardItem item = new NSPasteboardItem();
string[] writableTypes = {"public.text"};

// Add a data provider to the item
ImageInfoDataProvider dataProvider = new ImageInfoDataProvider (Info.Name, Info.ImageType);
var ok = item.SetDataProviderForTypes (dataProvider, writableTypes);

// Save to pasteboard
if (ok) {
    pasteboard.WriteObjects (new NSPasteboardItem[] { item });
}
```

Reading data

To read the data back from the pasteboard, use the following code:

```

// Initialize the pasteboard
NSPasteboard pasteboard = NSPasteboard.GeneralPasteboard;
Class [] classArray = { new Class ("NSImage") };

bool ok = pasteboard.CanReadObjectForClasses (classArray, null);
if (ok) {
    // Read the image off of the pasteboard
    NSObject [] objectsToPaste = pasteboard.ReadObjectsForClasses (classArray, null);
    NSImage image = (NSImage)objectsToPaste[0];

    // Do something with data
    ...
}

Class [] classArray2 = { new Class ("ImageInfo") };
ok = pasteboard.CanReadObjectForClasses (classArray2, null);
if (ok) {
    // Read the image off of the pasteboard
    NSObject [] objectsToPaste = pasteboard.ReadObjectsForClasses (classArray2, null);

    // Do something with data
    ...
}

```

Summary

This article has taken a detailed look at working with the pasteboard in a Xamarin.Mac application to support copy and paste operations. First, it introduced a simple example to get you familiar with standard pasteboards operations. Next, it took a detailed look at the pasteboard and how to read and write data from it. Finally, it looked at using a custom data type to support the copying and pasting of complex data types within an app.

Related Links

- [MacCopyPaste \(sample\)](#)
- [Hello, Mac](#)
- [Pasteboard Programming Guide](#)
- [macOS Human Interface Guidelines](#)

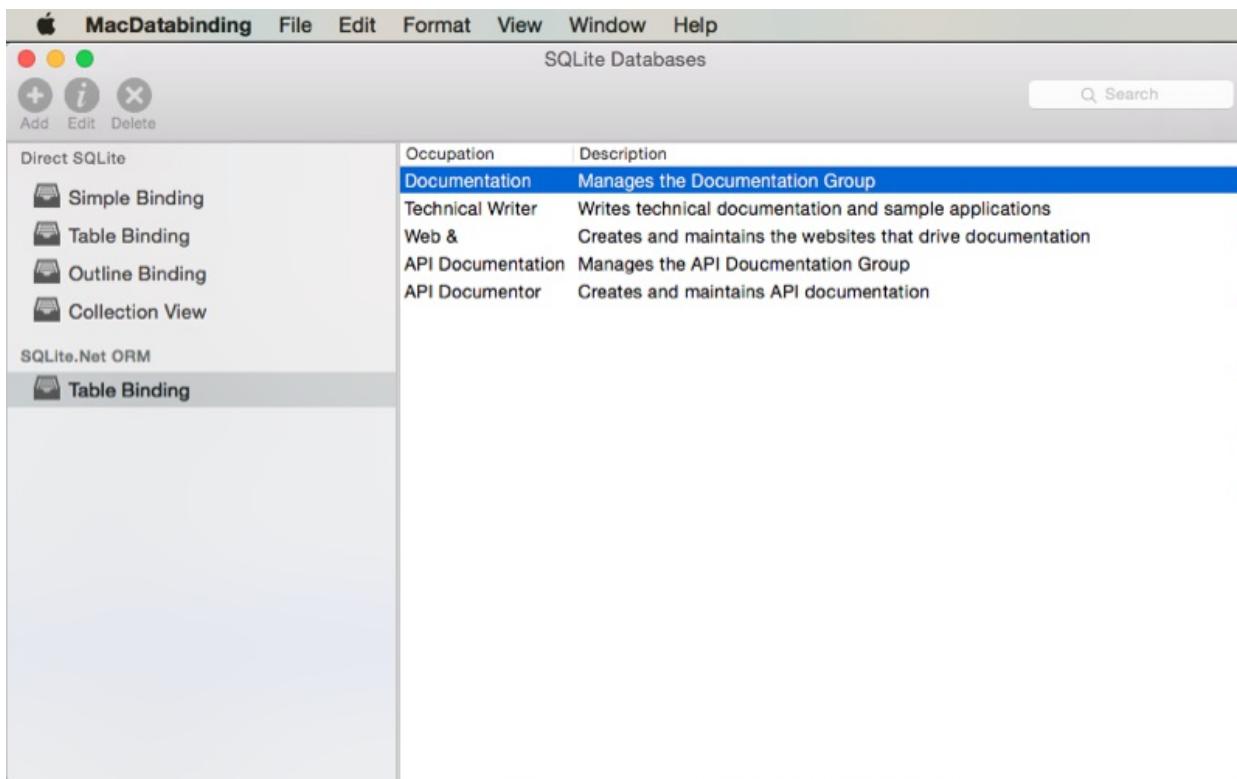
Sandboxing a Xamarin.Mac app

11/2/2020 • 27 minutes to read • [Edit Online](#)

This article covers sandboxing a Xamarin.Mac application for release on the App Store. It covers all of the elements that go into sandboxing, such as container directories, entitlements, user-determined permissions, privilege separation, and kernel enforcement.

Overview

When working with C# and .NET in a Xamarin.Mac application, you have the same ability to sandbox an application as you do when working with Objective-C or Swift.



In this article, we'll cover the basics of working with sandboxing in a Xamarin.Mac application and all of the elements that go into sandboxing: container directories, entitlements, user-determined permissions, privilege separation, and kernel enforcement. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` attributes used to wire up your C# classes to Objective-C objects and UI elements.

About the App Sandbox

The App Sandbox provides strong defense against damage that can be caused by a malicious application being run on a Mac by limiting the access that an application has to system resources.

A non-sandboxed application has the full rights of the user that is running the app and can access or do anything that the user can. If the app contains security holes (or any framework that it's using), a hacker can potentially exploit those weaknesses and use the app to take control of the Mac that it is running on.

By limiting access to resources on a per-application basis, a sandboxed app provides a line of defense against theft, corruption or malicious intent on the part of an application running on the user's machine.

The App Sandbox is an access control technology built into macOS (enforced at the kernel level) that provides a twofold strategy:

1. The App Sandbox enables the developer to describe *how* an application will interact with the OS and, in this way, it is granted only the access rights that are required to get the job done, and no more.
2. The App Sandbox allows the user to seamlessly grant further access to the system via the Open and Save dialog boxes, drag and drop operations and other, common user interactions.

Preparing to implement the App Sandbox

The elements of the App Sandbox that will be discussed in detail in the article are as follows:

- Container Directories
- Entitlements
- User-Determined Permissions
- Privilege Separation
- Kernel Enforcement

After you understand these details, you'll be able to create a plan for adopting the App Sandbox in your Xamarin.Mac application.

First, you will need to determine if your application is a good candidate for sandboxing (most applications are). Next, you will need to resolve any API incompatibilities and determine which elements of the App Sandbox you will require. Finally, look into using Privilege Separation to maximize your application's defense level.

When adopting the App Sandbox, some file system locations that your application uses will be different. Particularly, your application will have a Container Directory that will be used for application support files, databases, caches and any other files that are not user documents. Both macOS and Xcode provide support to migrate these type of files from their legacy locations to the container.

Sandboxing quick start

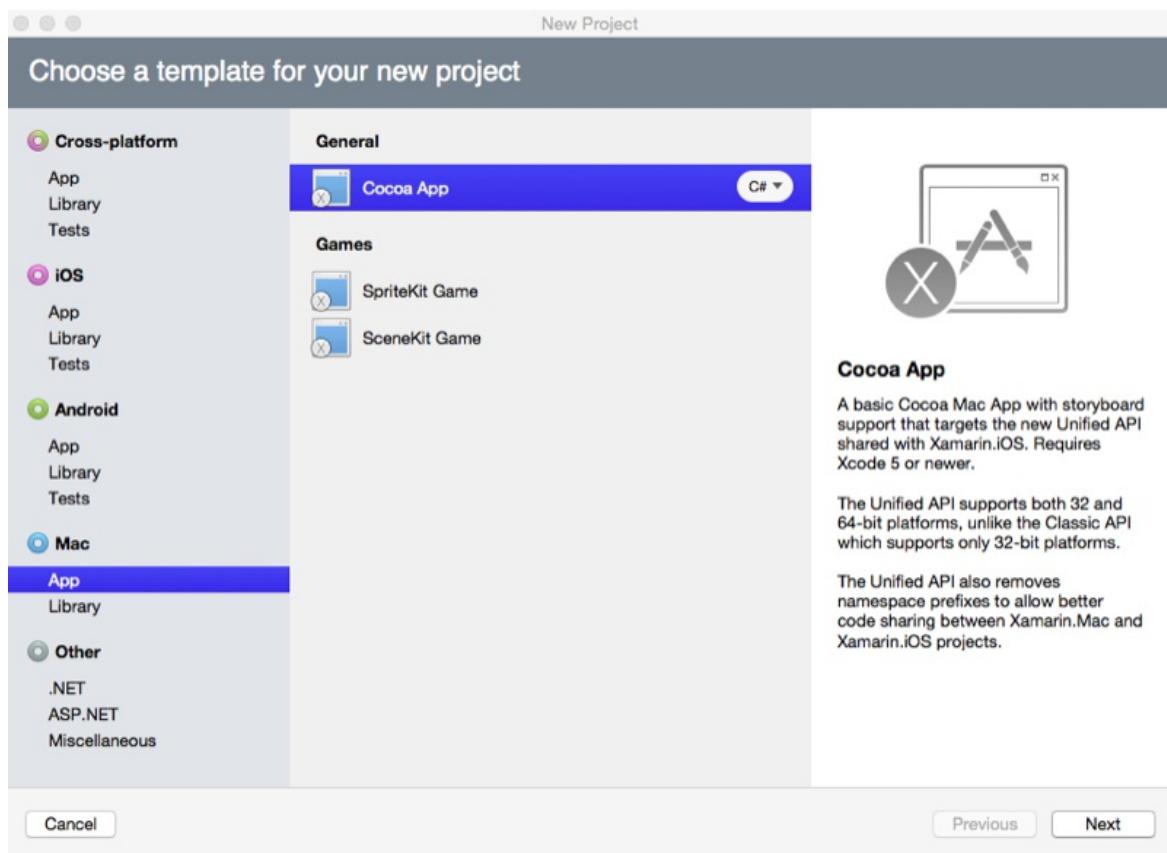
In this section, we'll create a simple Xamarin.Mac app that uses a Web View (which requires a network connection that is restricted under sandboxing unless specifically requested) as an example of getting started with the App Sandbox.

We'll verify that the application is actually sandboxed and learn how to troubleshoot and resolve common App Sandbox errors.

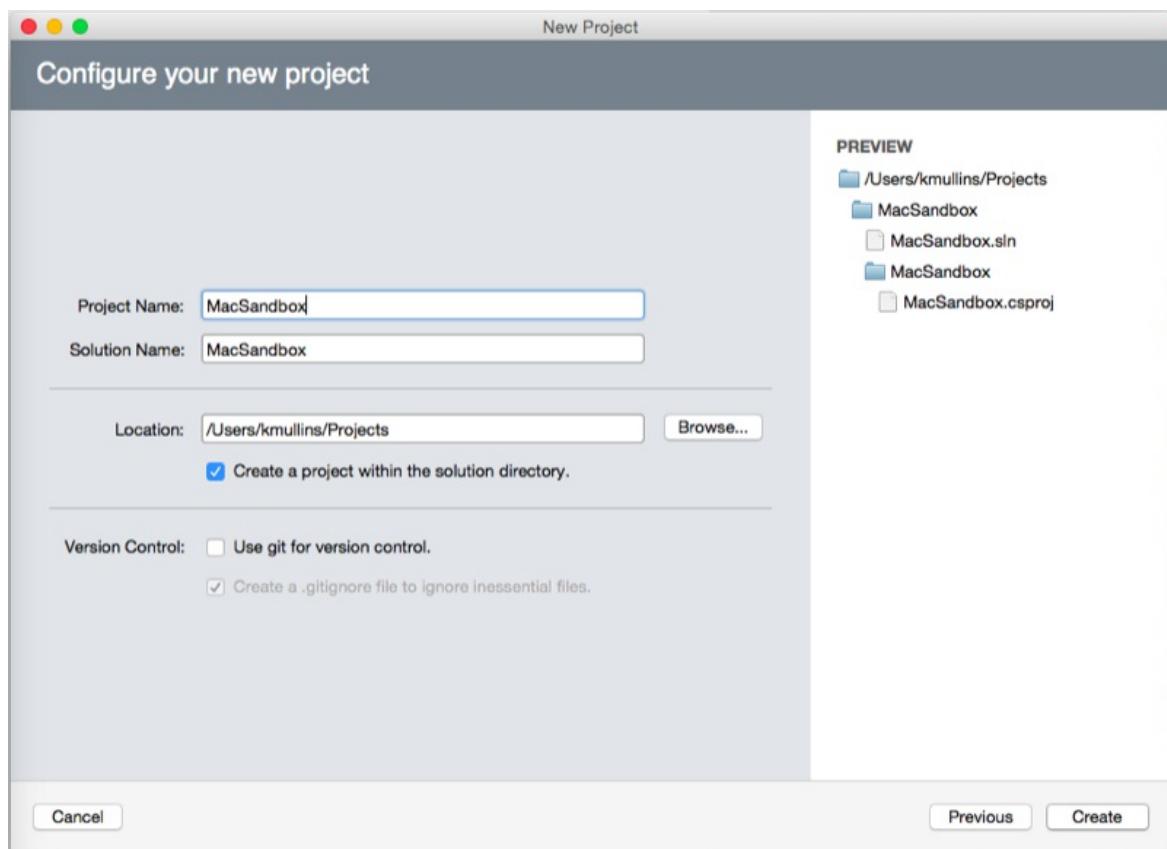
Creating the Xamarin.Mac project

Let's do the following to create our sample project:

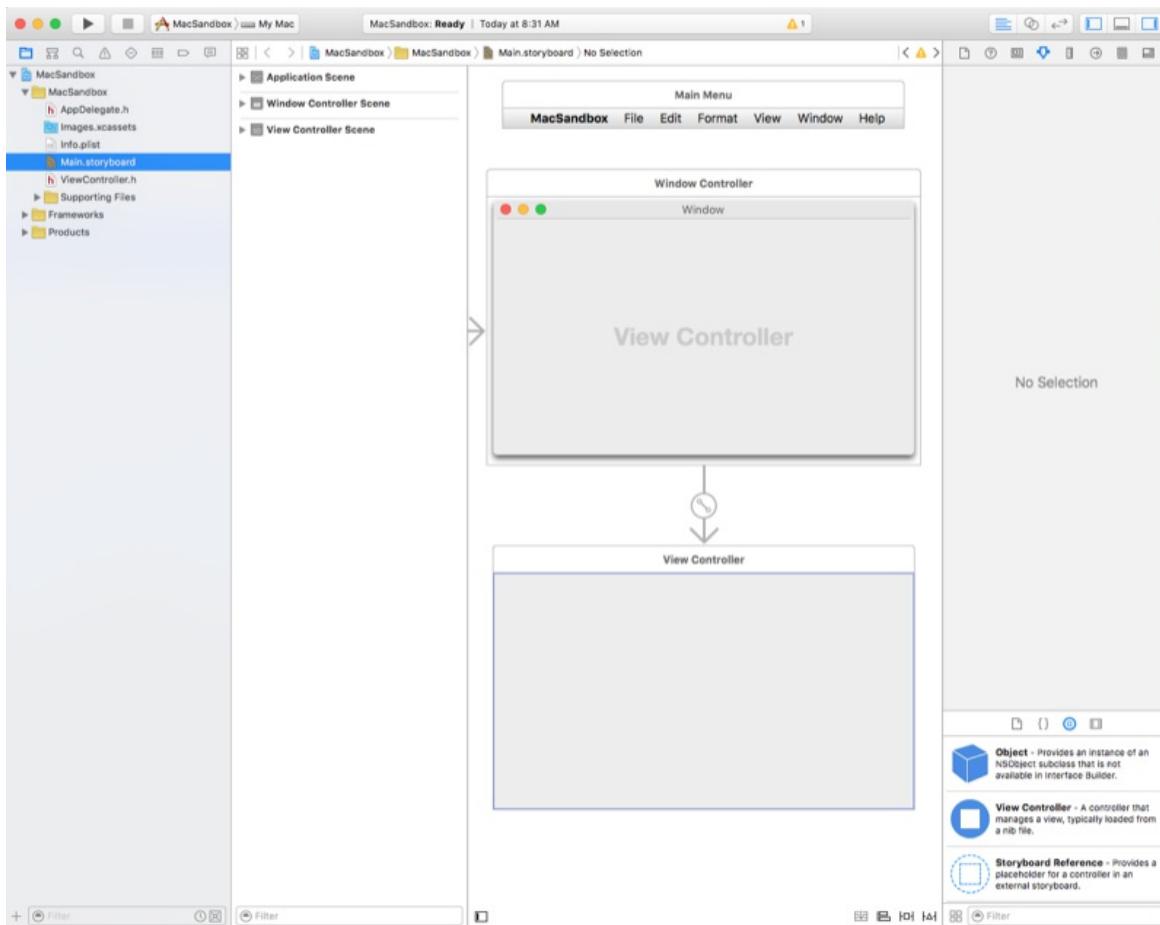
1. Start Visual Studio for Mac and click the **New Solution..** link.
2. From the **New Project** dialog box, select **Mac > App > Cocoa App**:



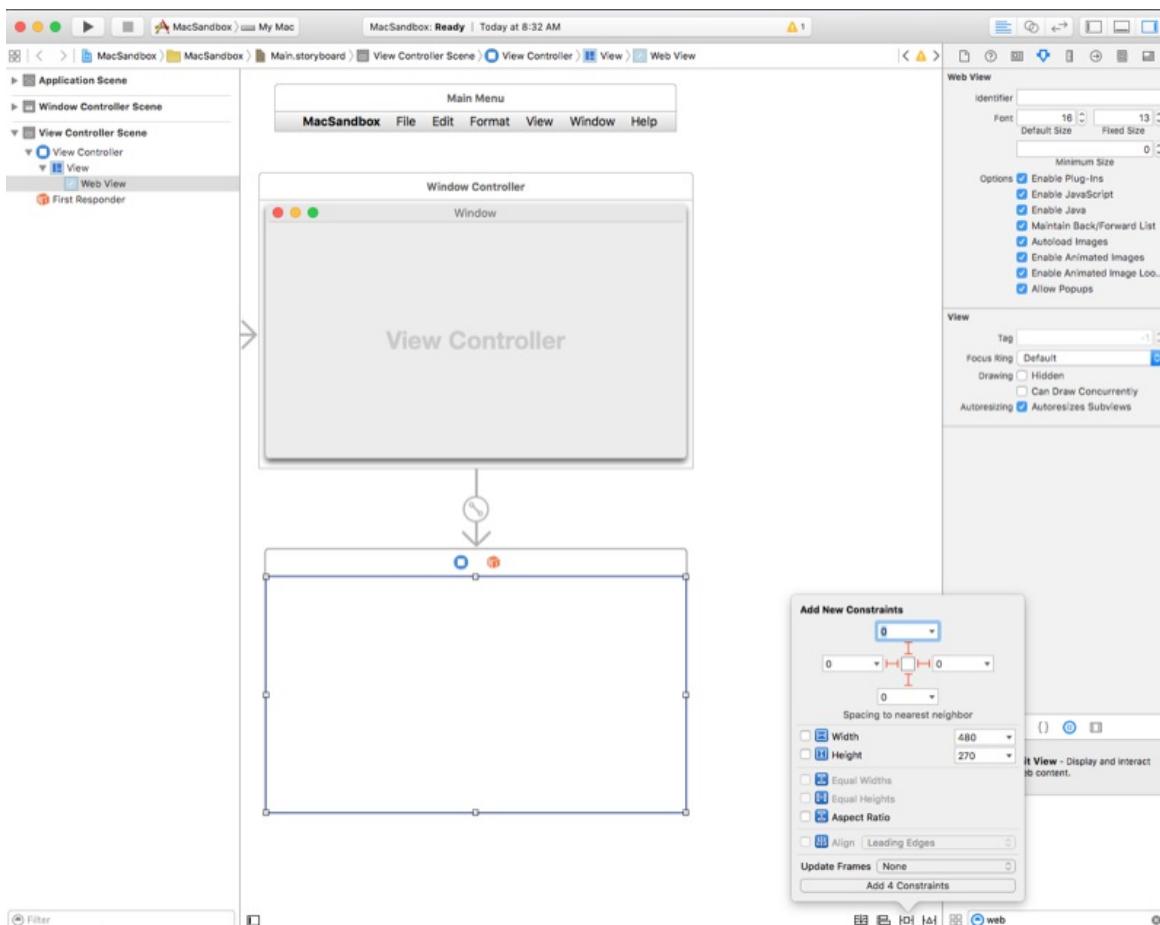
3. Click the **Next** button, enter `MacSandbox` for the project name and click the **Create** button:



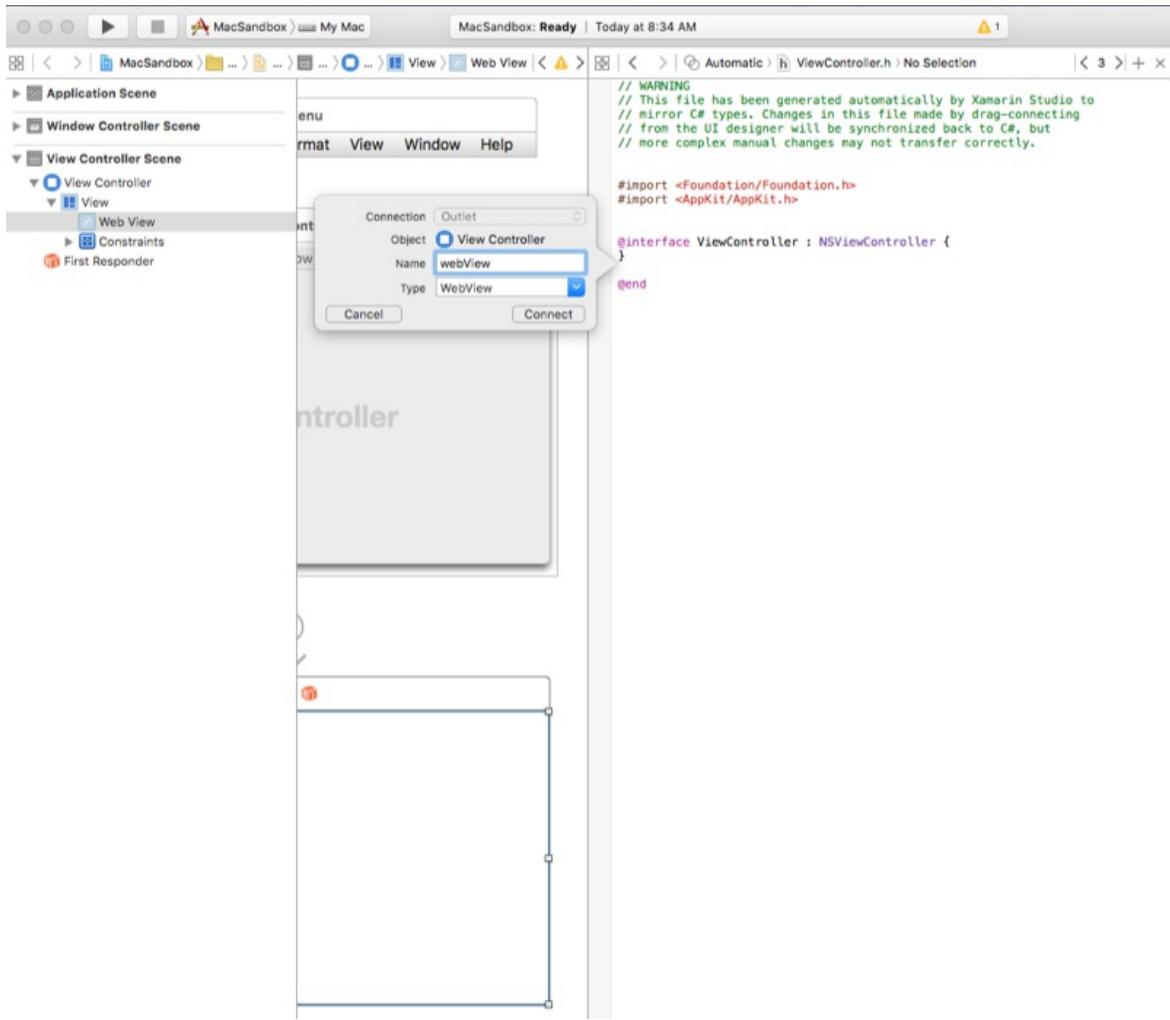
4. In the **Solution Pad**, double-click the **Main.storyboard** file to open it for editing in Xcode:



5. Drag a **Web View** onto the Window, size it to fill the content area and set it to grow and shrink with the window:



6. Create an outlet for the web view called `webView`:

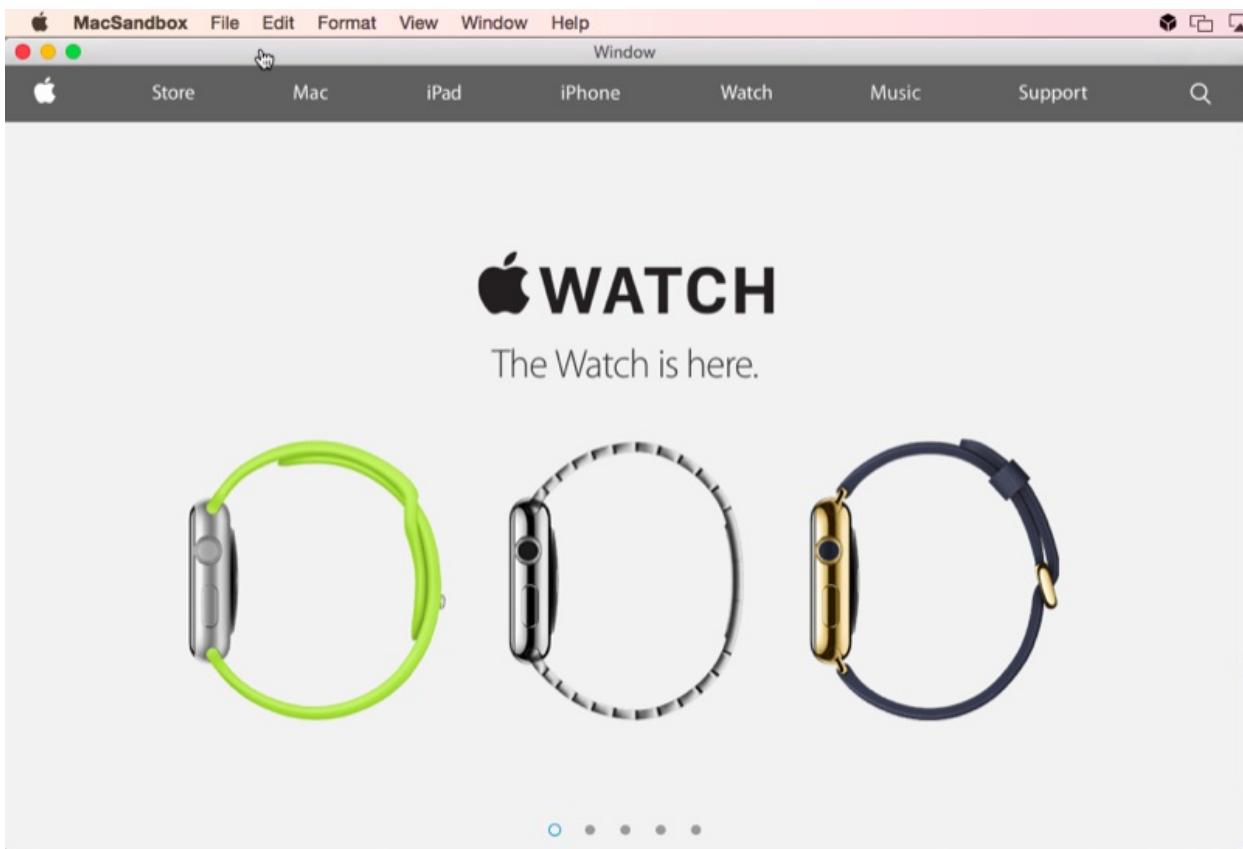


7. Return to Visual Studio for Mac and double-click the **ViewController.cs** file in the **Solution Pad** to open it for editing.
8. Add the following using statement: `using WebKit;`
9. Make the `ViewDidLoad` method look like the following:

```
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();
    webView.MainFrame.LoadRequest(new NSUrlRequest(new NSUrl("http://www.apple.com")));
}
```

10. Save your changes.

Run your application and ensure that the Apple Website is displayed in the window as follows:



Signing and provisioning the app

Before we can enable the App Sandbox, we first need to provision and sign our Xamarin.Mac application.

Let do the following:

1. Log into the Apple Developer Portal:

A screenshot of the Apple Developer portal homepage. The top navigation bar includes links for "Developer", "Member Center", "People", "Programs & Add-ons", and "Your Account". The user "Kevin Mullins" is logged in. Below the navigation, there are several sections: "SDKs" (with a link to download), "Forums" (with a link to find answers and discuss), "Certificates, Identifiers & Profiles" (with a link to manage), "Bug Reporting" (with a link to submit bugs or request enhancements), "iTunes Connect" (with a link to manage apps published on the App Store and Mac App Store), and "Technical Support" (with a link to request technical support). Each section has a brief description and a "Read more" link.

2. Select Certificates, Identifiers & Profiles:

The screenshot shows the 'Certificates, Identifiers & Profiles' section of the Apple Developer website. It features three main categories:

- iOS Apps**: Includes links for Certificates, Identifiers, Devices, and Provisioning Profiles, along with a 'Learn More' section for App Distribution Guide.
- Mac Apps**: Includes links for Certificates, Identifiers, Devices, and Provisioning Profiles, along with a 'Learn More' section for App Distribution Guide.
- Safari Extensions**: Includes links for Certificates and a 'Learn More' section for Safari Extensions Development Guide and Reference.

3. Under Mac Apps, select Identifiers:

The screenshot shows the 'Certificates, Identifiers & Profiles' section under the 'Mac Apps' category. The left sidebar shows navigation options for Certificates, Identifiers (with 'App IDs' selected), Devices, and Provisioning Profiles. The main panel displays the 'Mac App IDs' section with the following data:

Name	ID
CloudKit Test Mac	com.appracatappra.cloudtestmac
MacWriter	com.appracatappra.macwriter

4. Create a new ID for the application:

Developer

Technologies Resources Programs Support Member Center Search Developer Kevin Mullins

Certificates, Identifiers & Profiles

Mac Apps

Certificates
All Pending Development Production

Identifiers
App IDs Website Push IDs iCloud Containers

Devices All

Provisioning Profiles All Development Distribution

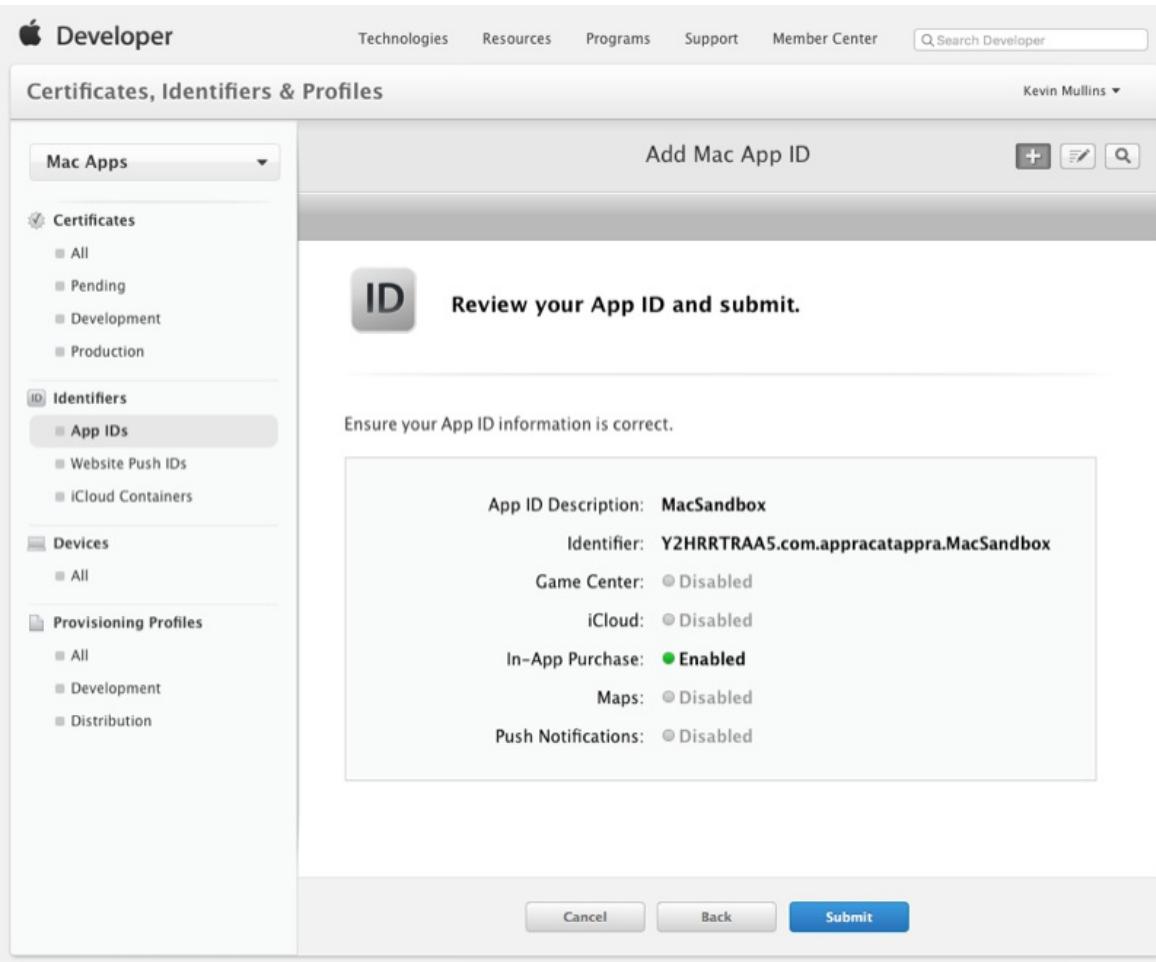
Add Mac App ID

ID Review your App ID and submit.

Ensure your App ID information is correct.

App ID Description: MacSandbox
Identifier: Y2HRRTRAAS.com.appracatappra.MacSandbox
Game Center: Disabled
iCloud: Disabled
In-App Purchase: Enabled
Maps: Disabled
Push Notifications: Disabled

Cancel Back Submit



5. Under Provisioning Profiles, select Development:

Developer

Technologies Resources Programs Support Member Center Search Developer Kevin Mullins

Certificates, Identifiers & Profiles

Mac Apps

Certificates
All Pending Development Production

Identifiers
App IDs Website Push IDs iCloud Containers

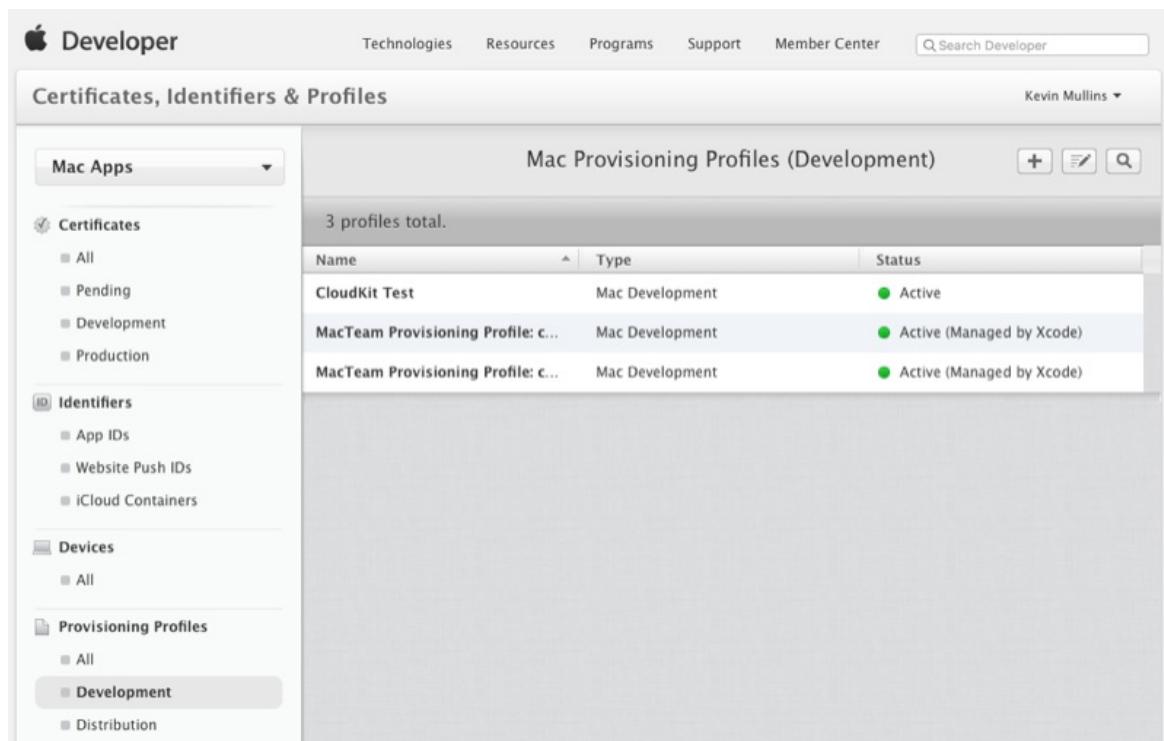
Devices All

Provisioning Profiles
All **Development** Distribution

Mac Provisioning Profiles (Development)

3 profiles total.

Name	Type	Status
CloudKit Test	Mac Development	<input checked="" type="radio"/> Active
MacTeam Provisioning Profile: c...	Mac Development	<input checked="" type="radio"/> Active (Managed by Xcode)
MacTeam Provisioning Profile: c...	Mac Development	<input checked="" type="radio"/> Active (Managed by Xcode)



6. Create a new profile and select Mac App Development:



Technologies Resources Programs Support Member Center

Search Developer

Kevin Mullins ▾

Certificates, Identifiers & Profiles

Mac Apps

Certificates

- All
- Pending
- Development
- Production

Identifiers

- App IDs
- Website Push IDs
- iCloud Containers

Devices

- All

Provisioning Profiles

- All
- Development**
- Distribution

Add Mac Provisioning Profile

[+] [Edit] [Search]

Select Type

Configure

Generate

Download



What type of provisioning profile do you need?

Development

Mac App Development

Create a provisioning profile to install development apps on test devices.

Distribution

Mac App Store

Create a distribution provisioning profile to submit your app to the Mac App Store.

Cancel

Continue

7. Select the App ID we created above:



Technologies Resources Programs Support Member Center

Search Developer

Kevin Mullins ▾

Certificates, Identifiers & Profiles

Mac Apps

Certificates

- All
- Pending
- Development
- Production

Identifiers

- App IDs
- Website Push IDs
- iCloud Containers

Devices

- All

Provisioning Profiles

- All
- Development**
- Distribution

Add Mac Provisioning Profile

[+] [Edit] [Search]

Select Type

Configure

Generate

Download



Select App ID.

If you plan to use services such as Game Center, In-App Purchase, and Push Notifications, or want a Bundle ID unique to a single app, use an explicit App ID. If you want to create one provisioning profile for multiple apps or don't need a specific Bundle ID, select a wildcard App ID. Wildcard App IDs use an asterisk (*) as the last digit in the Bundle ID field. Please note that iOS App IDs and Mac App IDs cannot be used interchangeably.

App ID CloudKit Test Mac (Y2HRRTRA5.com.appracatappra.cloudtestmac)

MacSandbox (Y2HRRTRA5.com.appracatappra.MacSandbox)

MacWriter (Y2HRRTRA5.com.appracatappra.macwriter)

Cancel

Back

Continue

8. Select the Developers for this profile:

Developer

Technologies Resources Programs Support Member Center Search Developer Kevin Mullins ▾

Certificates, Identifiers & Profiles

Mac Apps Select Type Configure Generate Download

Select certificates.

Select the certificates you wish to include in this provisioning profile. To use this profile to install an app, the certificate the app was signed with must be included.

Select All 1 of 1 item(s) selected
 Kevin Mullins (Mac Development)

Cancel Back Continue

Certificates
All Pending Development Production

Identifiers
App IDs Website Push IDs iCloud Containers

Devices
All

Provisioning Profiles
All Development Distribution

Add Mac Provisioning Profile + ⌂ ⌂

9. Select the computers for this profile:

Developer

Technologies Resources Programs Support Member Center Search Developer Kevin Mullins ▾

Certificates, Identifiers & Profiles

Mac Apps Select Type Configure Generate Download

Select devices.

Select the devices you wish to include in this provisioning profile. To install an app signed with this profile on a device, the device must be included.

Select All 1 of 1 item(s) selected
 Europa (Mac Mini)

Cancel Back Continue

Certificates
All Pending Development Production

Identifiers
App IDs Website Push IDs iCloud Containers

Devices
All

Provisioning Profiles
All Development Distribution

Add Mac Provisioning Profile + ⌂ ⌂

10. Give the profile a Name:

Certificates, Identifiers & Profiles

Mac Apps

Certificates

- All
- Pending
- Development
- Production

Identifiers

- App IDs
- Website Push IDs
- iCloud Containers

Devices

- All

Provisioning Profiles

- All
- **Development**
- Distribution

Add Mac Provisioning Profile



Select Type

Configure

Generate

Download



Your provisioning profile is ready.

Download and Install

Download and double click the following file to install your Provisioning Profile.



Name: MacSandbox Development
Type: Mac Development
App ID: Y2HRRTRAAS.com.appracatappra.MacSandbox
Expires: Jun 18, 2016

[Download](#)

Documentation

For more information on using and managing your Provisioning Profile read:

[App Distribution Guide](#)[Add Another](#)[Done](#)

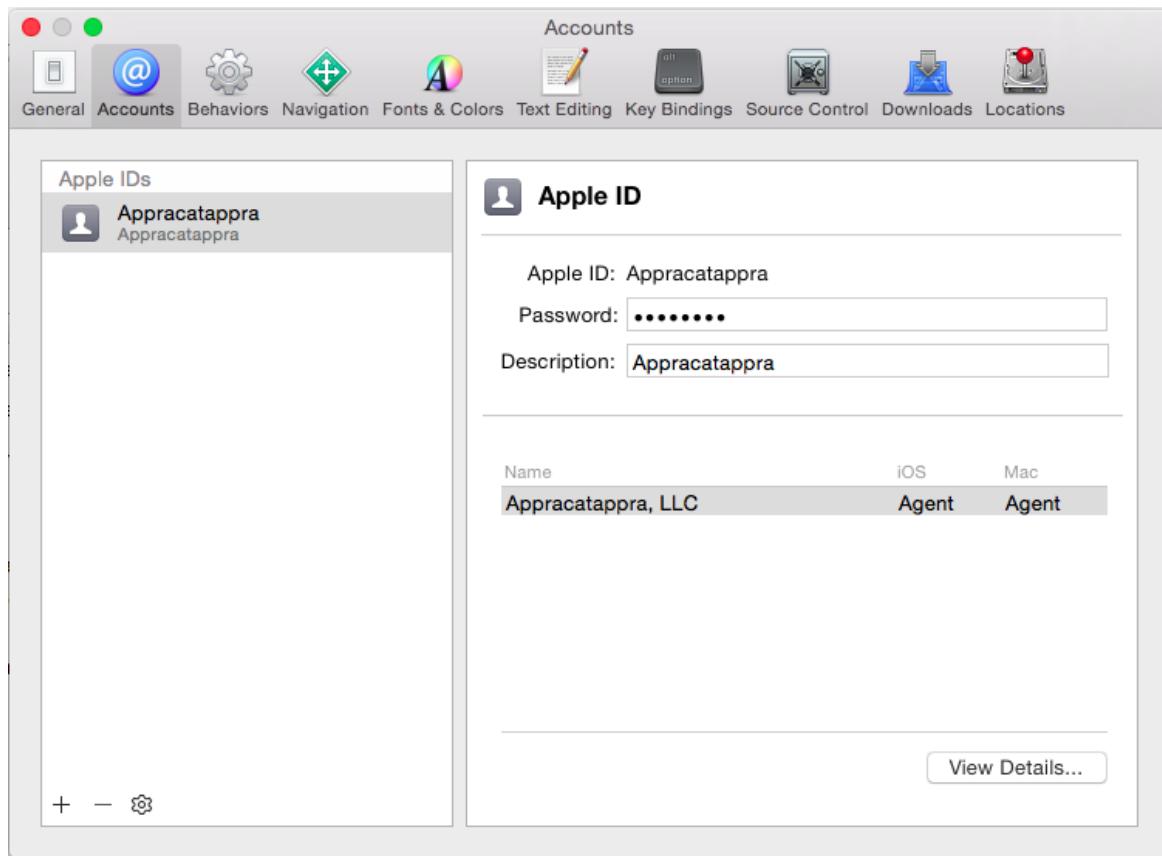
11. Click the **Done** button.

IMPORTANT

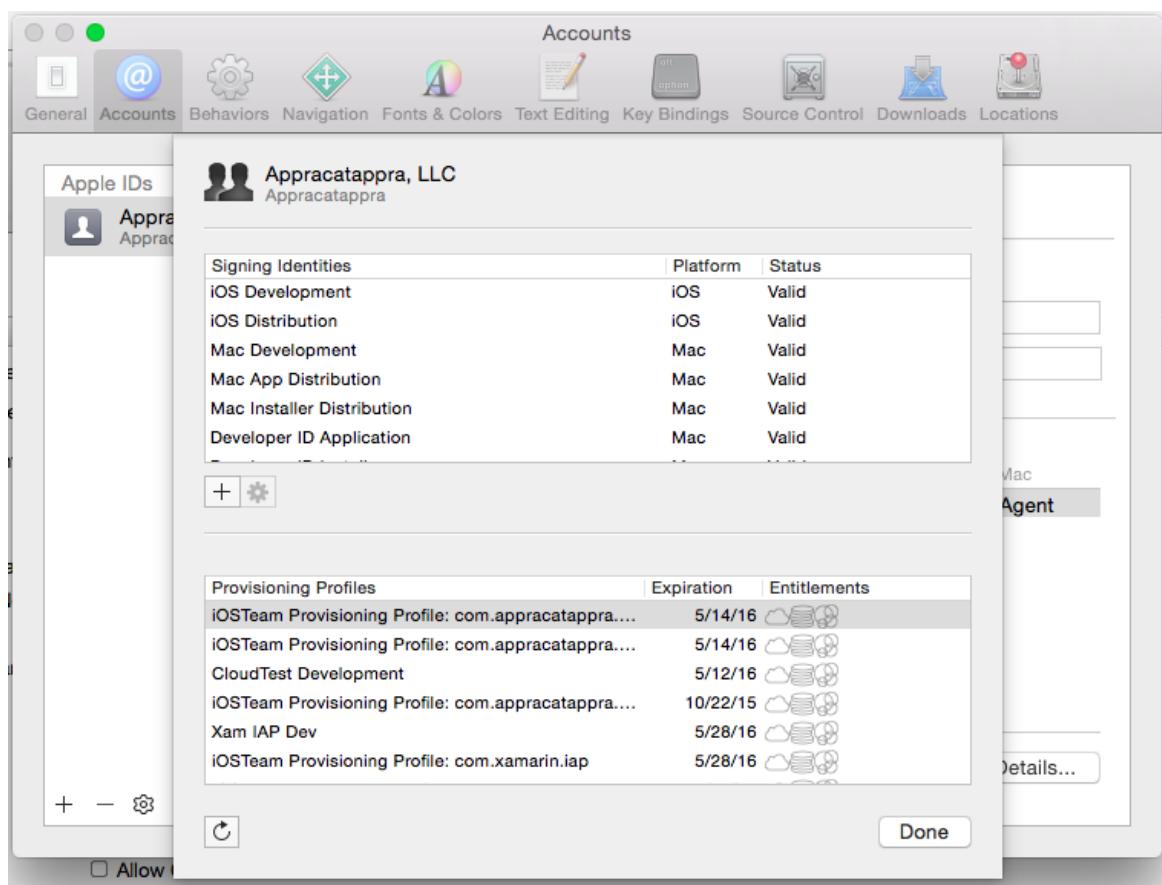
In some instances you might need to download the new Provisioning Profile from Apple's Developer Portal directly and double-click it to install. You might also need to stop and restart Visual Studio for Mac before it will be able to access the new profile.

Next, we need to load the new App ID and Profile on the development machine. Let's do the following:

1. Start Xcode and select **Preferences** from the **Xcode** menu:



2. Click on the View Details... button:

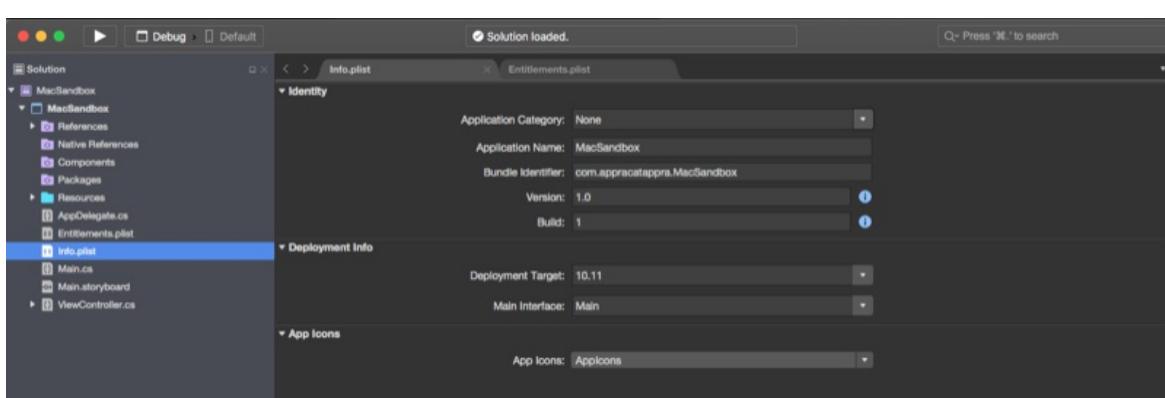


3. Click on the Refresh button (in the lower left hand corner).

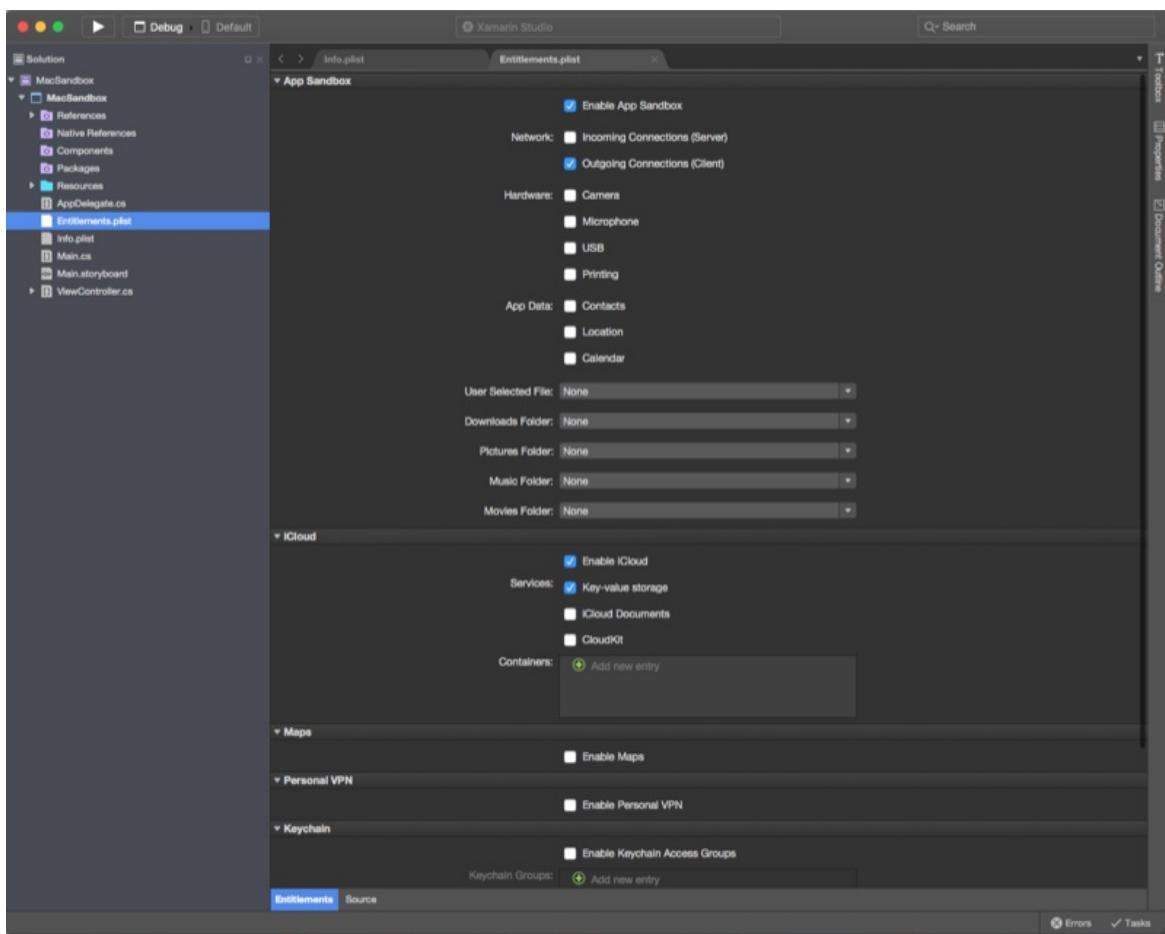
4. Click on the Done button.

Next, we need to select the new App ID and Provisioning Profile in our Xamarin.Mac project. Let's do the following:

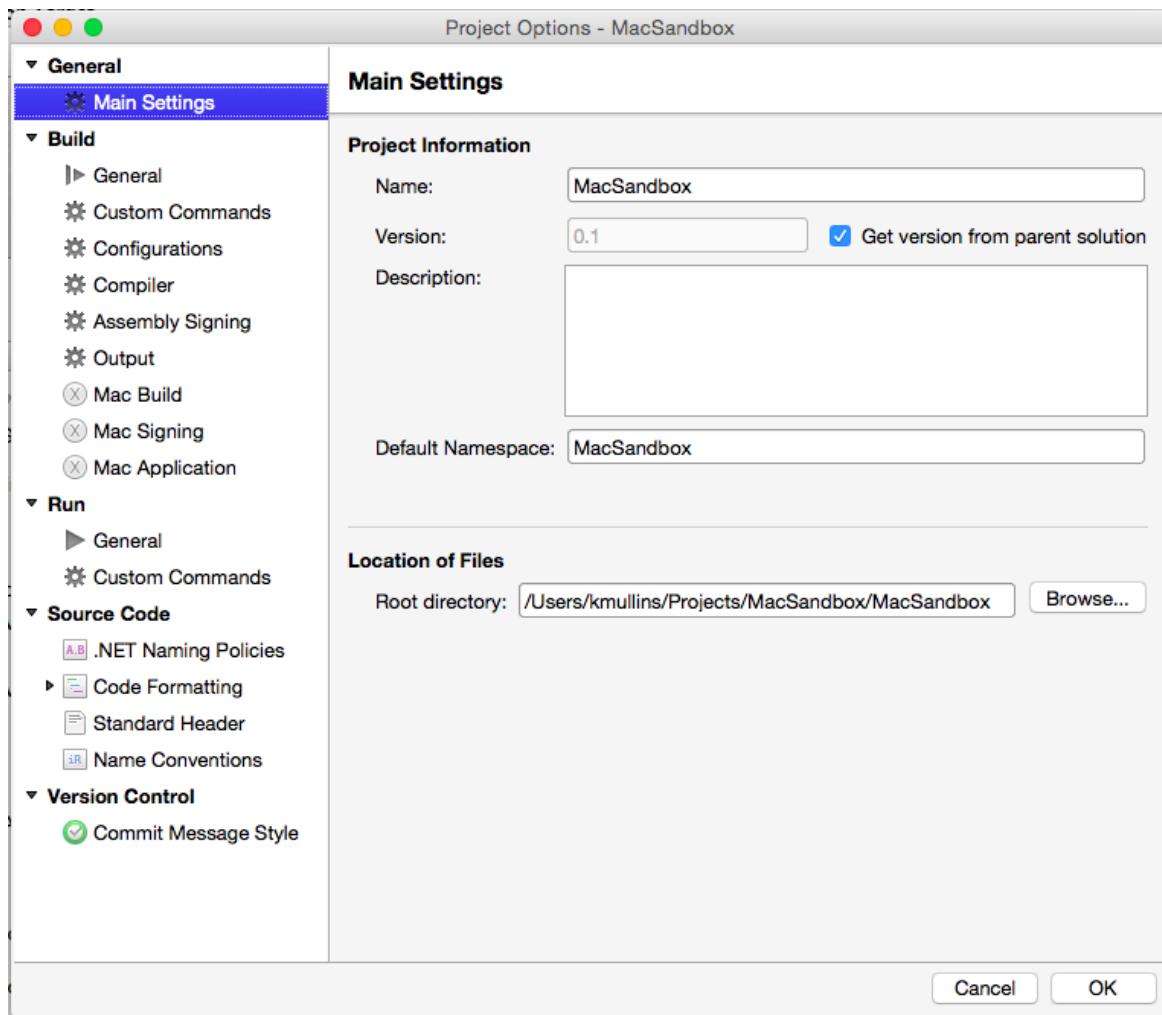
1. In the **Solution Pad**, double-click the **Info.plist** file to open it for editing.
2. Ensure that the **Bundle Identifier** matches our App ID we created above (example: `com.appracatappra.MacSandbox`):



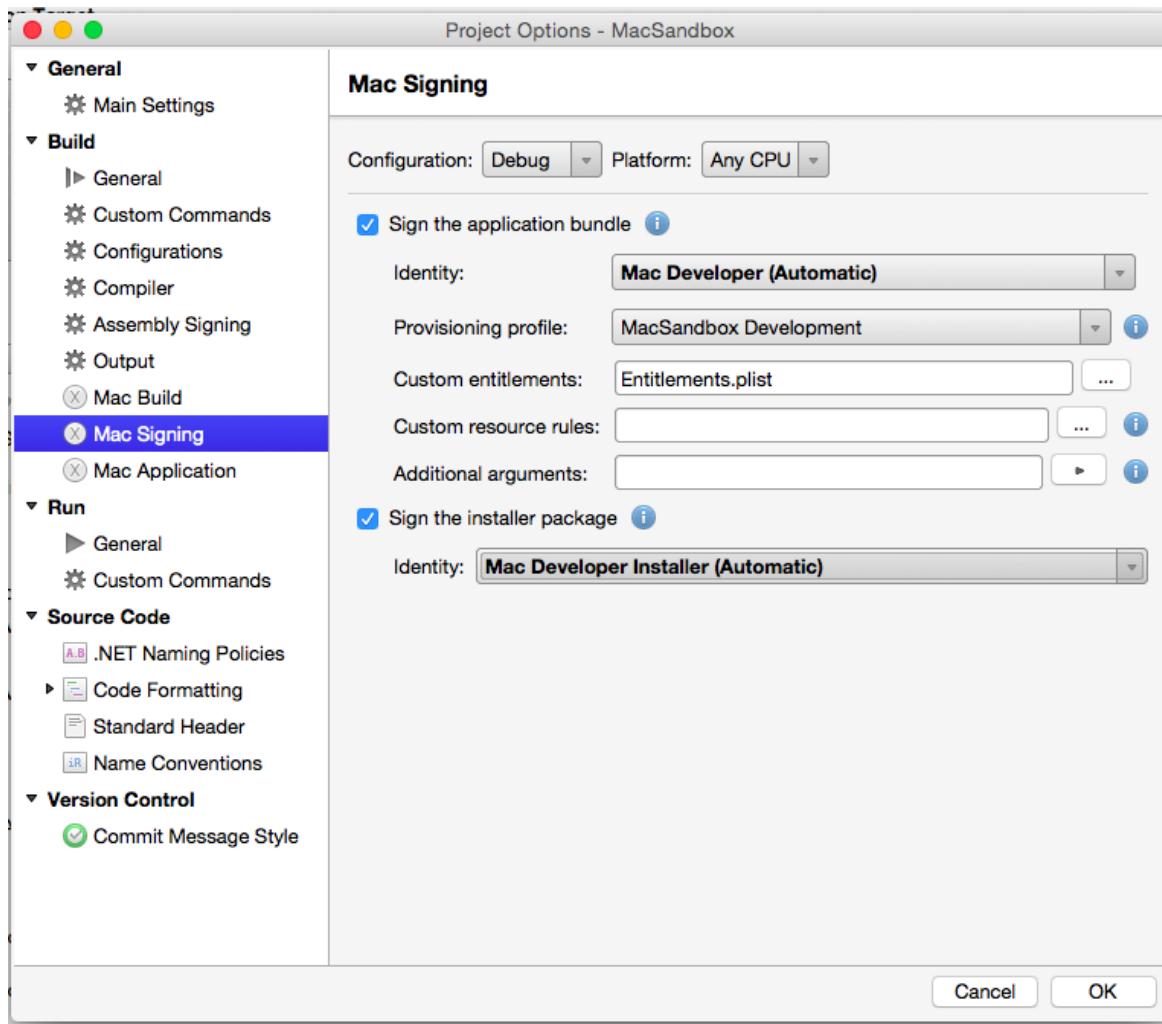
3. Next, double-click the **Entitlements.plist** file and ensure our **iCloud Key-Value Store** and the **iCloud Containers** all match our App ID we created above (example: `com.appracatappra.MacSandbox`):



4. Save your changes.
5. In the **Solution Pad**, double-click the project file to open its Options for editing:



6. Select **Mac Signing**, then check **Sign the application bundle** and **Sign the installer package**.
Under **Provisioning profile**, select the one we created above:



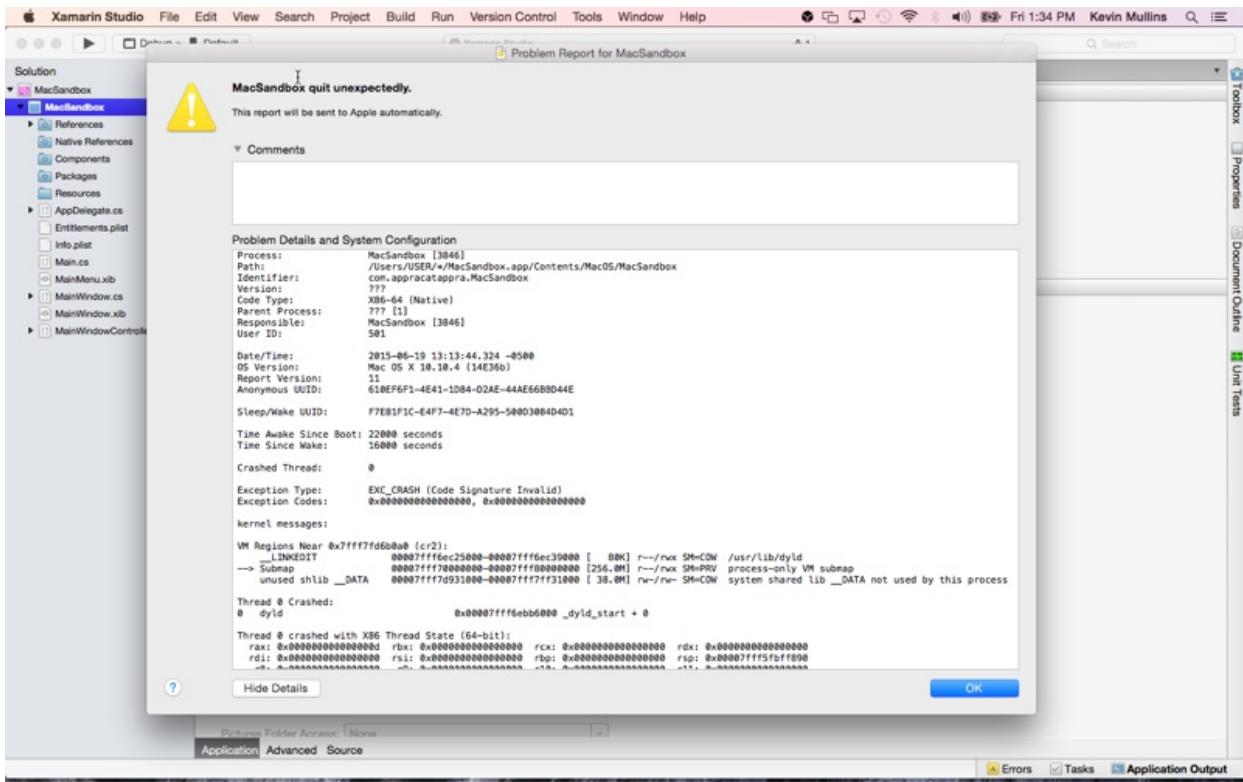
7. Click the **Done** button.

IMPORTANT

You might have to quit and restart Visual Studio for Mac to get it to recognize the new App ID and Provisioning Profile that was installed by Xcode.

Troubleshooting provisioning issues

At this point you should try to run the application and make sure that everything is signed and provisioned correctly. If the app still runs as before, everything is good. In the event of a failure, you might get a dialog box like the following one:



Here are the most common causes of provisioning and signing issues:

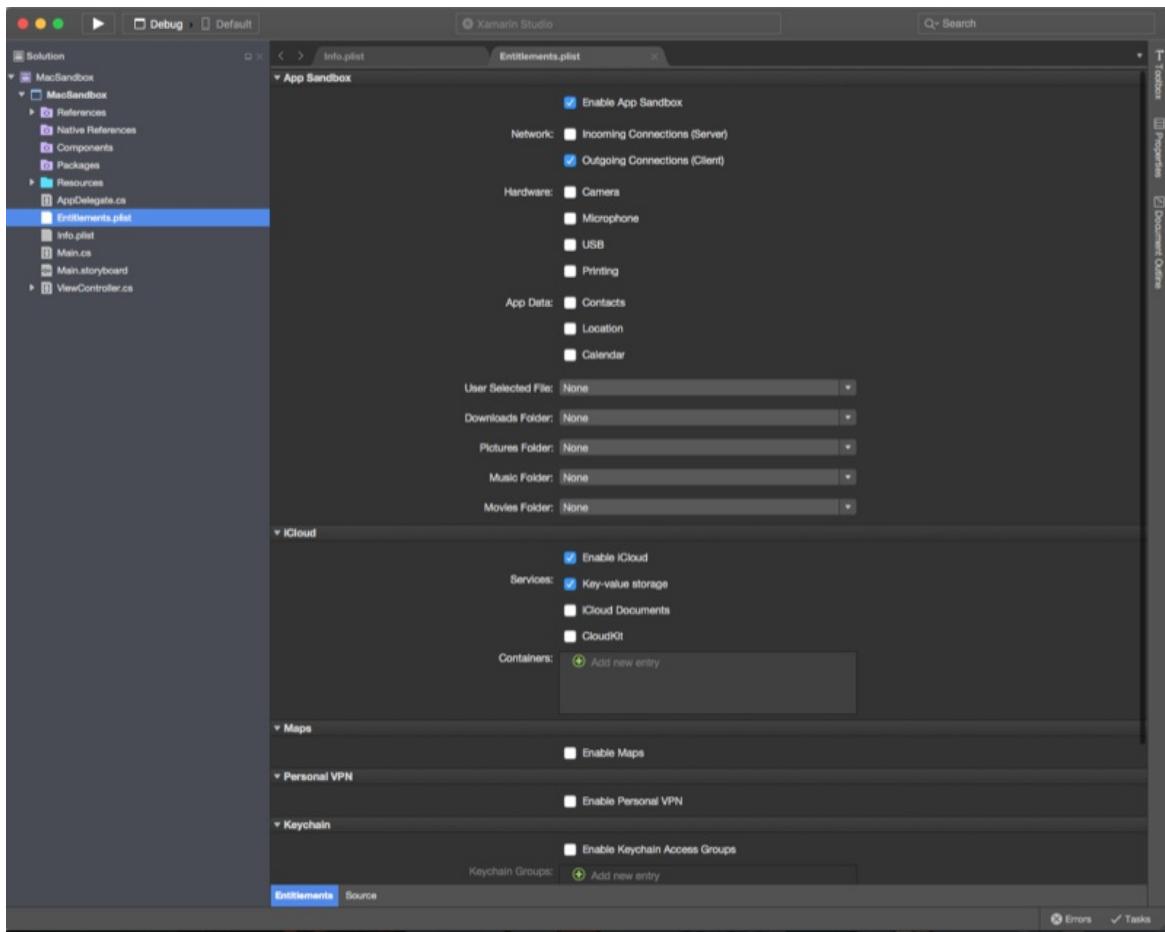
- The App Bundle ID doesn't match the App ID of the selected profile.
- The Developer ID doesn't match the Developer ID of the selected profile.
- The UUID of the Mac being tested on isn't registered as part of the selected profile.

In the case of an issue, correct the problem on the Apple Developer Portal, refresh the profiles in Xcode and do a clean build in Visual Studio for Mac.

Enable the App Sandbox

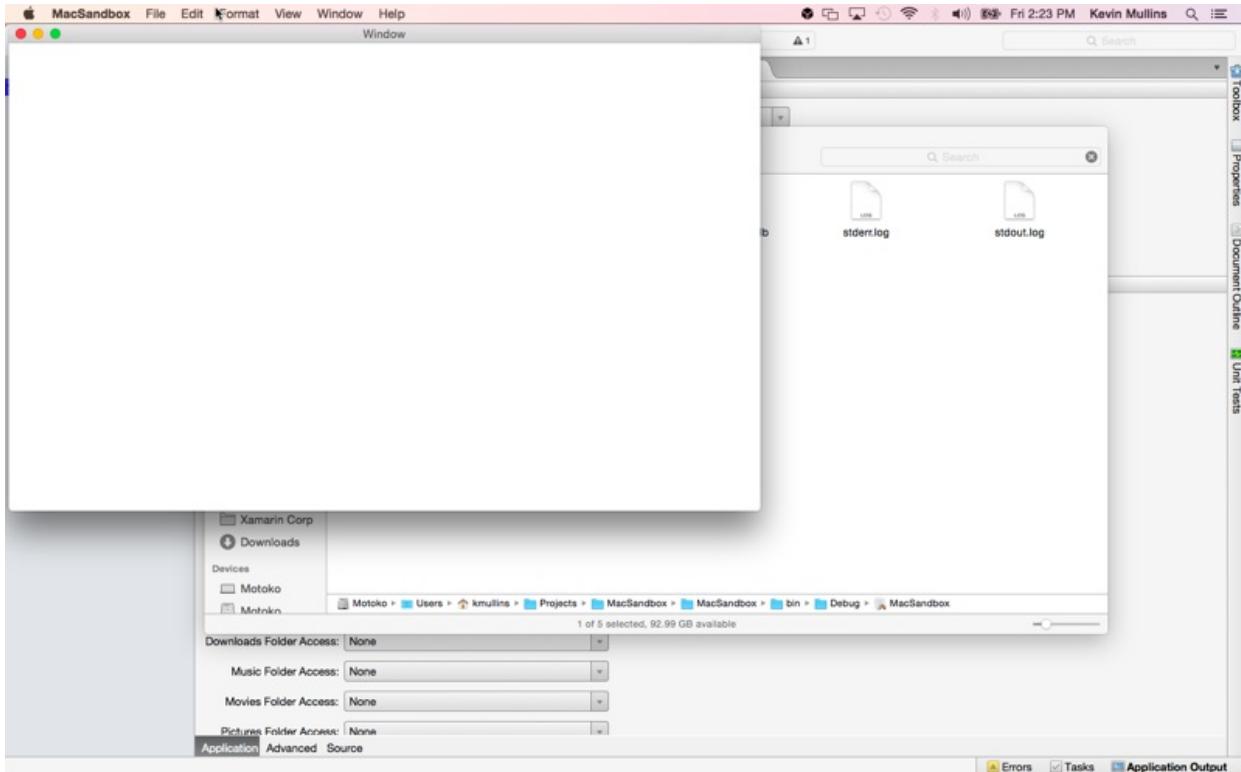
You enable the App Sandbox by selecting a checkbox in your projects options. Do the following:

1. In the **Solution Pad**, double-click the **Entitlements.plist** file to open it for editing.
2. Check both **Enable Entitlements** and **Enable App Sandboxing**:



3. Save your changes.

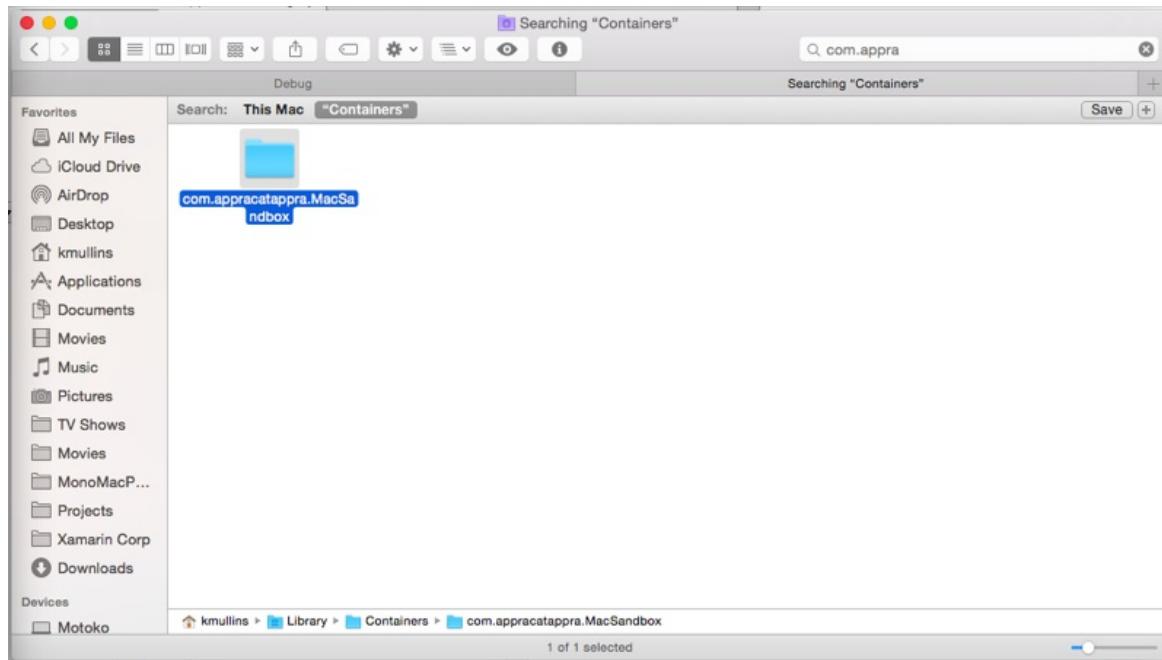
At this point, you have enabled the App Sandbox but you have not provided the required network access for the Web View. If you run the application now, you should get a blank window:



Verifying that the app is sandboxed

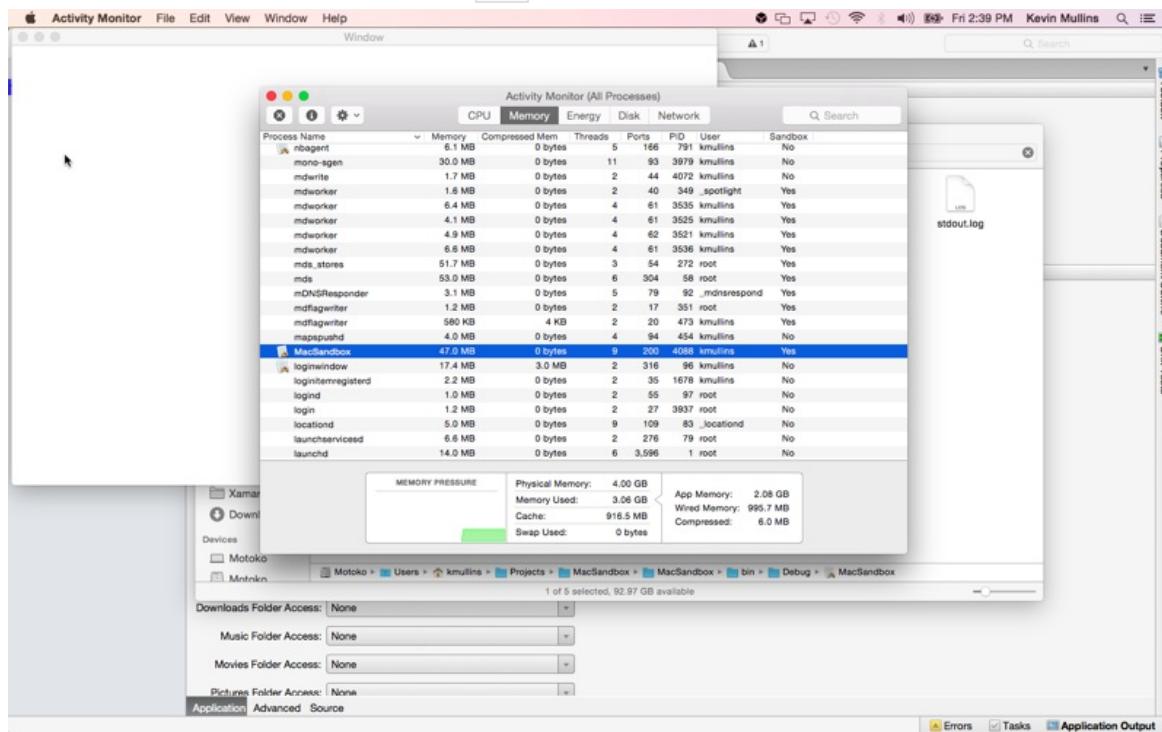
Aside from the resource blocking behavior, there are three main ways to tell if a Xamarin.Mac application has been successfully sandboxed:

1. In Finder, check the contents of the `~/Library/Containers/` folder - If the app is sandboxed, there will be a folder named like your app's Bundle Identifier (example: `com.appracatappra.MacSandbox`):



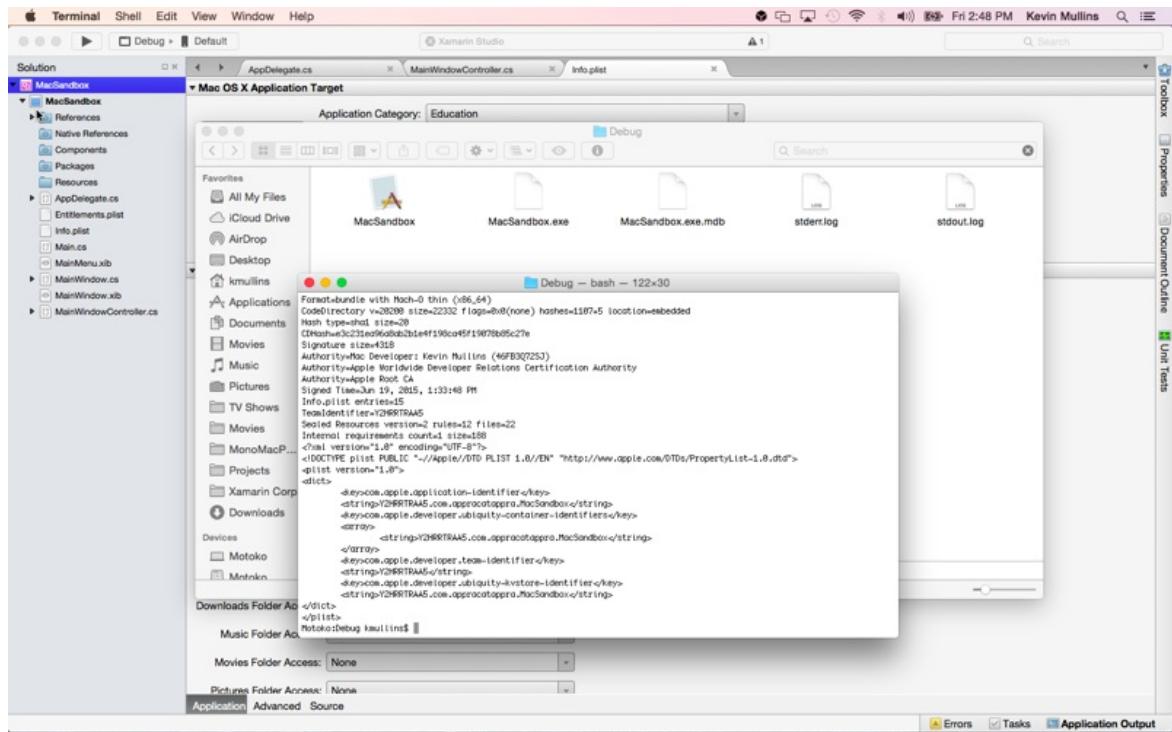
2. The system sees the app as sandboxed in the Activity Monitor:

- Launch Activity Monitor (under `/Applications/Utilities`).
- Choose **View > Columns** and ensure that the **Sandbox** menu item is checked.
- Ensure that the Sandbox column reads `Yes` for your application:



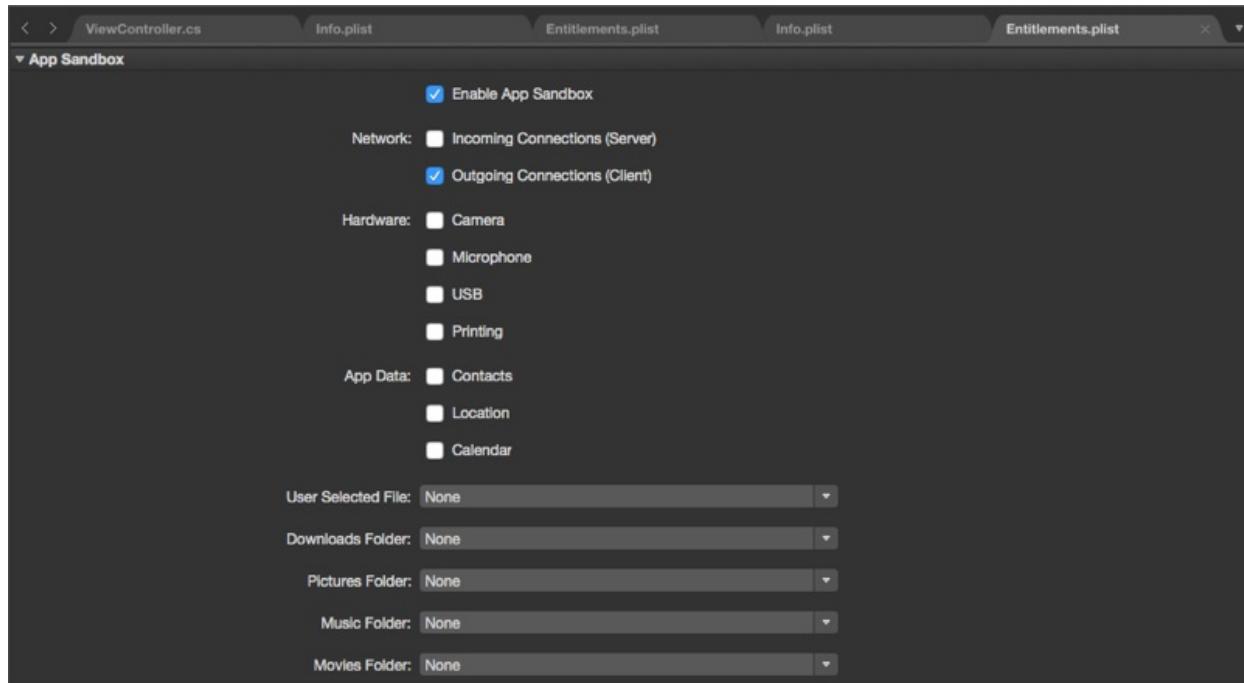
3. Check that the app binary is sandboxed:

- Start the Terminal app.
- Navigate to the applications `bin` directory.
- Issue this command: `codesign -dvvv --entitlements :- executable_path` (where `executable_path` is the path to your application):



Debugging a sandboxed app

The debugger connects to Xamarin.Mac apps through TCP, which means that by default when you enable sandboxing, it is unable to connect to the app, so if you try to run the app without the proper permissions enabled, you get an error *"Unable to connect to the debugger"*.



The **Allow Outgoing Network Connections (Client)** permission is the one required for the debugger, enabling this one will allow debugging normally. Since you can't debug without it, we have updated the `CompileEntitlements` target for `msbuild` to automatically add that permission to the entitlements for any app that is sandboxed for debug builds only. Release builds should use the entitlements specified in the entitlements file, unmodified.

Resolving an App Sandbox violation

An App Sandbox Violation occurs if a sandboxed Xamarin.Mac application tried to access a resource that has not be explicitly allowed. For example, our Web View no longer being able to display the Apple Website.

The most common source of App Sandbox Violations occurs when the Entitlement settings specified in Visual

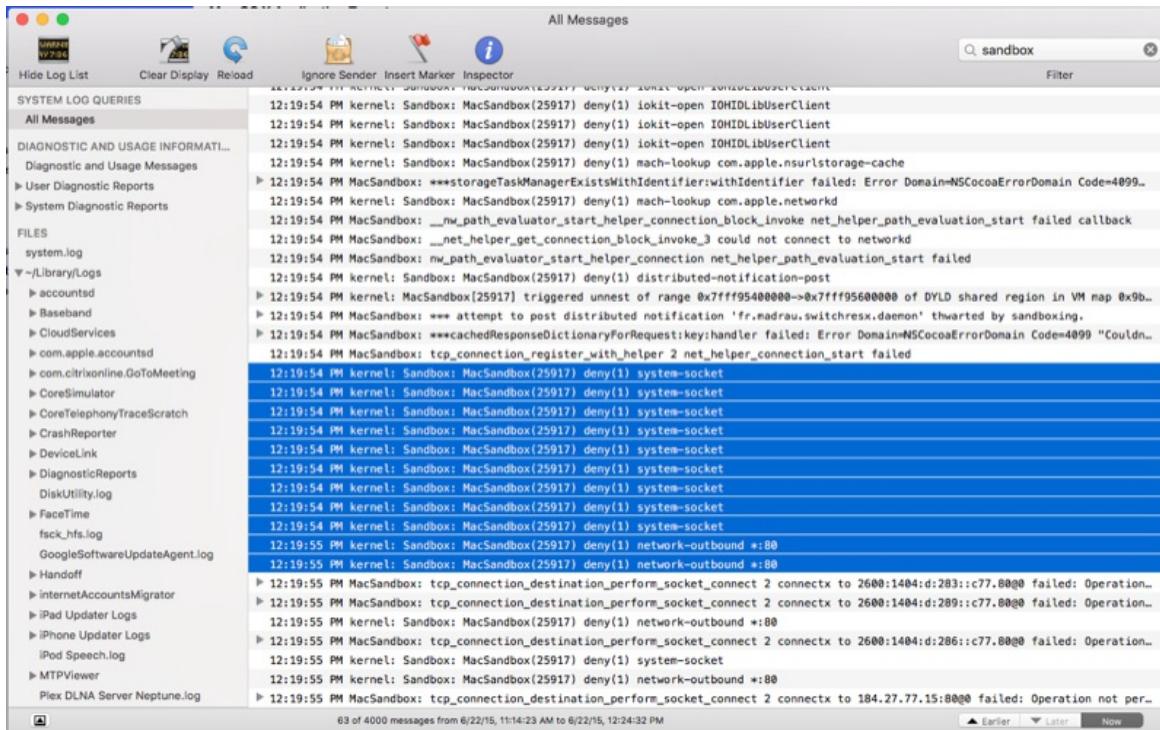
Studio for Mac don't match the requirements of the application. Again, back to our example, the missing Network Connection entitlements that keep the Web View from working.

Discovering App Sandbox violations

If you suspect an App Sandbox Violation is occurring in your Xamarin.Mac application, the quickest way to discover the issue is by using the **Console** app.

Do the following:

1. Compile the app in question and run it from Visual Studio for Mac.
2. Open the **Console** application (from `/Applications/Utilities/`).
3. Select **All Messages** in the sidebar and enter `sandbox` in the search:



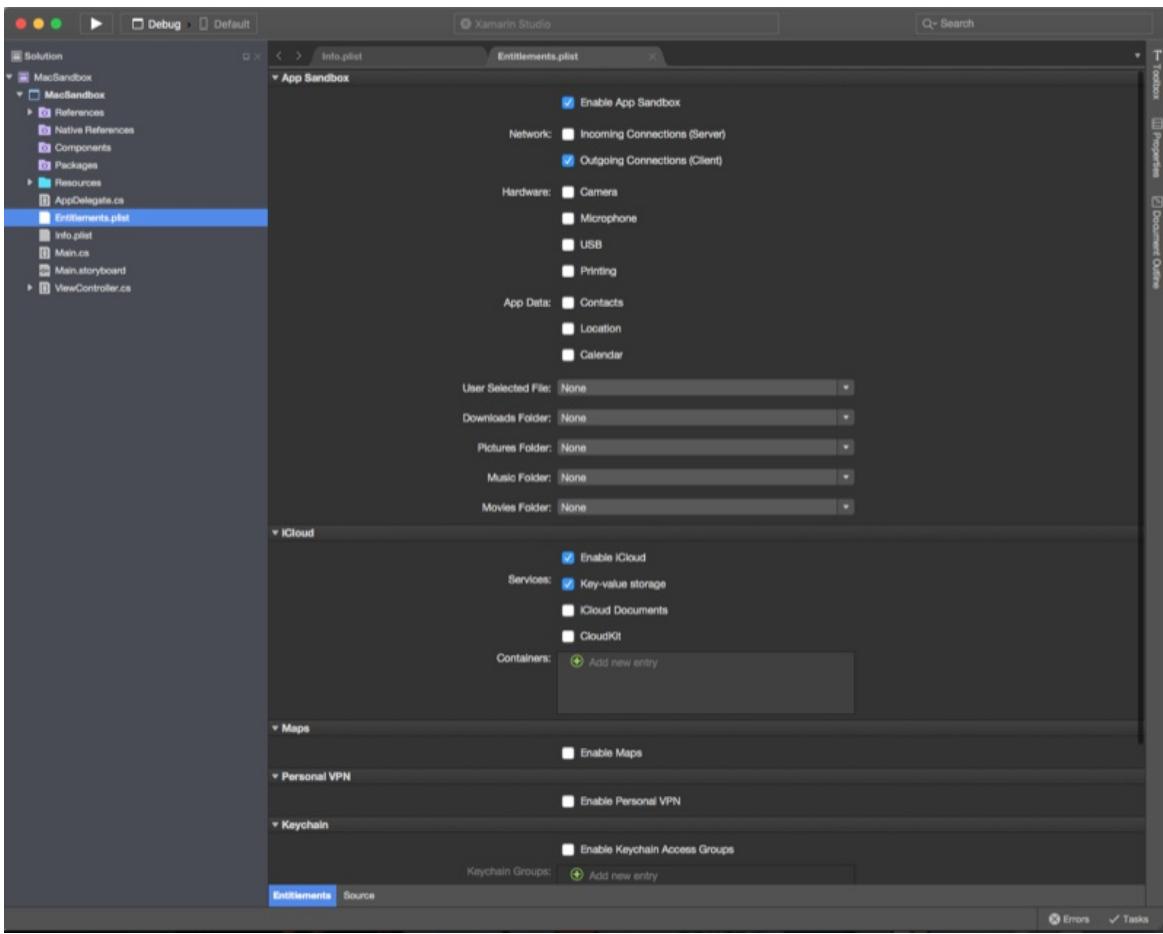
For our example app above, you can see that the Kernel is blocking the `network-outbound` traffic because of the App Sandbox, since we have not requested that right.

Fixing App Sandbox violations with entitlements

Now that we have seen how to find App Sandboxing Violations, let's see how they can be solved by adjusting our application's Entitlements.

Do the following:

1. In the **Solution Pad**, double-click the **Entitlements.plist** file to open it for editing.
2. Under the **Entitlements** section, check the **Allow Outgoing Network Connections (Client)** checkbox:



3. Save the changes to the application.

If we do the above for our example app, then build and run it, the web content will now be displayed as expected.

The App Sandbox in-depth

The access control mechanisms provided by the App Sandbox are few and easy to understand. However, way that the App Sandbox will be adopted by each app is unique and based on the requirements of the app.

Given your best effort to protect your Xamarin.Mac application from being exploited by malicious code, there need be only a single vulnerability in either the app (or one of the libraries or frameworks it consumes) to gain control of the app's interactions with the system.

The App Sandbox was designed to prevent the takeover (or limit the damage it can cause) by allowing you to specify the application's intended interactions with the system. The system will only grant access to the resource that your application requires to get its job done and nothing more.

When designing for the App Sandbox, you are designing for a worst-case scenario. If the application does become compromised by malicious code, it is limited to accessing only the files and resources in the app's sandbox.

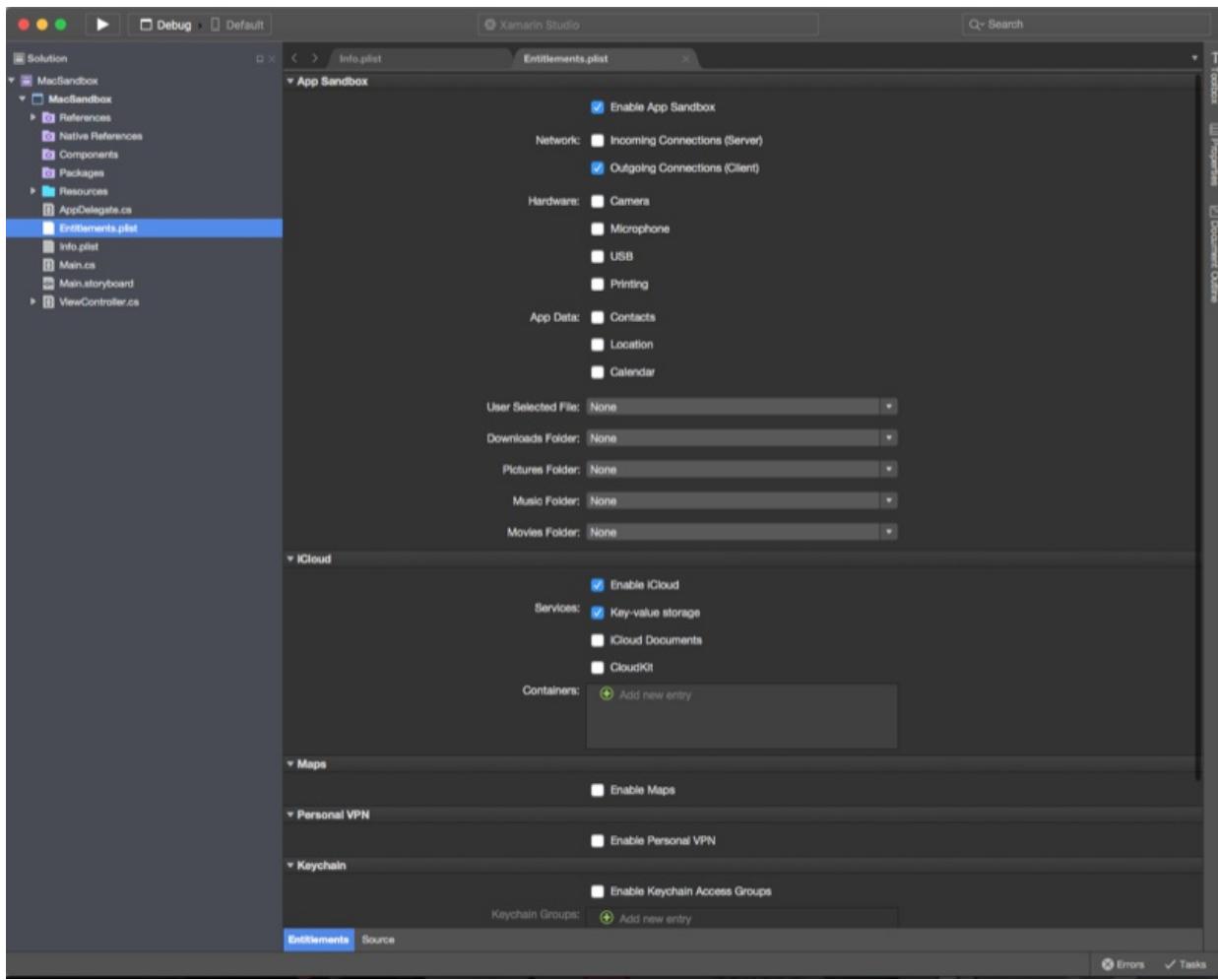
Entitlements and system resource access

As we saw above, a Xamarin.Mac application that has not been sandboxed is granted the full rights and access of the user that is running the app. If compromised by malicious code, a non-protected app can act as an agent for hostile behavior, with a wide ranging potential for doing harm.

By enabling the App Sandbox, you remove all but a minimal set of privileges, which you then re-enable on a need-only basis using your Xamarin.Mac app's entitlements.

You modify your application's App Sandbox Resources by editing its **Entitlements.plist** file and checking or

selecting the rights required from the editors dropdown boxes:



Container directories and file system access

When your Xamarin.Mac application adopts the App Sandbox, it has access to the following locations:

- **The App Container Directory** - On first run, the OS creates a special *Container Directory* where all of its resources go, that only it can access. The app will have full read/write access to this directory.
- **App Group Container Directories** - Your app can be granted access to one or more *Group Containers* that are shared amongst apps in the same group.
- **User-Specified Files** - Your application automatically obtains access to files that are explicitly opened or dragged and dropped onto the application by the user.
- **Related Items** - With the appropriate entitlements, your application can have access to a file with the same name but a different extension. For example, a document that has been saved as both a `.txt` file and a `.pdf`.
- **Temporary directories, command-line tool directories, and specific world-readable locations** - Your app has varying degrees of access to files in other well-defined locations as specified by the system.

The app container directory

A Xamarin.Mac application's App Container Directory has the following characteristics:

- It is in a hidden location in the user's Home directory (usually `~/Library/Containers`) and can be accessed with the `NSHomeDirectory` function (see below) within your application. Because it is in the Home directory, each user will get their own container for the app.
- The app has unrestricted read/write access to the Container Directory and all of its sub-directories and files within it.
- Most of macOS's path-finding APIs are relative to the app's container. For example, the container will have its own **Library** (accessed via `NSLibraryDirectory`), **Application Support** and **Preferences** subdirectories.

- macOS establishes and enforces the connection between app and its container via code signing. Even if another app tries to spoof the app by using its **Bundle Identifier**, it will not be able to access the Container because of code signing.
- The Container is not for user generated files. Instead it is for files that your application uses such as databases, caches or other specific types of data.
- For *shoebox* types of apps (like Apple's Photo app), the user's content will go into the Container.

IMPORTANT

Unfortunately, Xamarin.Mac does not have 100% API coverage yet (unlike Xamarin.iOS), as a result the `NSHomeDirectory` API has not been mapped in the current version of Xamarin.Mac.

As a temporary workaround, you can use the following code:

```
[System.Runtime.InteropServices.DllImport("/System/Library/Frameworks/Foundation.framework/Foundation")]
public static extern IntPtr NSHomeDirectory();

public static string ContainerDirectory {
    get {
        return ((NSString)ObjCRuntime.Runtime.GetNSObject(NSHomeDirectory())).ToString ();
    }
}
```

The app group container directory

Starting with Mac macOS 10.7.5 (and greater), an application can use the `com.apple.security.application-groups` entitlement to access a shared container that is common to all the applications in the group. You can use this shared container for non-user facing content such as database or other types of support files (such as caches).

The Group Containers are automatically added to each app's Sandbox Container (if they are part of a group) and are stored at `~/Library/Group Containers/<application-group-id>`. The Group ID *must* begin with your Development Team ID and a period, for example:

```
<team-id>.com.company.<group-name>
```

For more information, please see Apple's [Adding an Application to an Application Group](#) in [Entitlement Key Reference](#).

Powerbox and file system access outside of the app container

A sandboxed Xamarin.Mac application can access file system locations outside of its container in the following ways:

- At the specific direction of the user (via Open and Save Dialogs or other methods such as drag and drop).
- By using entitlements for specific file system locations (such as `/bin` or `/usr/lib`).
- When the file system location is in certain directories that are world readable (such as Sharing).

Powerbox is the macOS security technology that interacts with the user to expand your sandboxed Xamarin.Mac app's file access rights. Powerbox has no API, but is activated transparently when the app calls a `NSOpenPanel` or `NSSavePanel`. Powerbox access is enabled via the Entitlements that you set for your Xamarin.Mac application.

When a sandboxed app displays an Open or Save dialog, the window is presented by Powerbox (and not AppKit) and therefore has access to any file or directory that the user has access to.

When a user selects a file or directory from the Open or Save dialog (or drags either onto the App's icon), Powerbox adds the associated path to the app's sandbox.

Additionally, the system automatically permits the following to a sandboxed app:

- Connection to a system input method.
- Call services selected by the user from the **Services** menu (only for services marked as *safe for sandbox apps* by the service provider).
- Open files chosen by the user from the **Open Recent** menu.
- Use Copy & Paste between other applications.
- Read files from the following world-readable locations:
 - `/bin`
 - `/sbin`
 - `/usr/bin`
 - `/usr/lib`
 - `/usr/sbin`
 - `/usr/share`
 - `/System`
- Read and write files in the directories created by `NSTemporaryDirectory`.

By default, files opened or saved by a sandboxed Xamarin.Mac app remain accessible until the app terminates (unless the file was still open when the app quits). Open files will be automatically restored to the app's sandbox via the macOS Resume feature the next time the app is started.

To provide persistence to files located outside of a Xamarin.Mac App's container, use Security-Scoped bookmarks (see below).

Related items

The App Sandbox allows an app to access related items that have the same filename, but different extensions. The feature has two parts: a) a list of related extensions in the app's `Info.plist` file, b) code to tell the sandbox what the app will be doing with these files.

There are two scenarios where this makes sense:

1. The app needs to be able to save a different version of the file (with a new extension). For example, exporting a `.txt` file to a `.pdf` file. To handle this situation, you must use a `NSFileCoordinator` to access the file. You'll call the `WillMove(fromURL, toURL)` method first, move the file to the new extension and then call `ItemMoved(fromURL, toURL)`.
2. The app needs to open a main file with one extension and several supporting files with different extensions. For example, a Movie and a Subtitle file. Use an `NSFilePresenter` to gain access to the secondary file. Provide the main file to the `PrimaryPresentedItemURL` property and the secondary file to the `PresentedItemURL` property. When the main file is opened, call the `AddFilePresenter` method of the `NSFileCoordinator` class to register the secondary file.

In both scenarios, the app's `Info.plist` file must declare the Document Types that the app can open. For any file type, add the `NSIsRelatedItemType` (with a value of `YES`) to its entry in the `CFBundleDocumentTypes` array.

Open and save dialog behavior with sandboxed apps

The following limits are placed on the `NSOpenPanel` and `NSSavePanel` when calling them from a sandboxed Xamarin.Mac app:

- You cannot programmatically invoke the OK button.
- You cannot programmatically alter a user's selection in a `NSOpenSavePanelDelegate`.

Additionally, the following inheritance modifications are in place:

- Non-Sandboxed App - `NSOpenPanel`

Security-scoped bookmarks and persistent resource access

As stated above, a Sandboxed Xamarin.Mac application can access a file or resource outside of its container by way of direct user interaction (as provided by PowerBox). However, access to these resources does not automatically persist across app launches or system restarts.

By using *Security-Scoped Bookmarks*, a Sandboxed Xamarin.Mac application can preserve user intent and maintain access to given resources after an app restart.

Security-scoped bookmark types

When working with Security-Scoped Bookmarks and Persistent Resource Access, there are two distinct use cases:

- **An App-Scoped Bookmark provides persistent access to a user-specified file or folder.**

For example, if the sandboxed Xamarin.Mac application allows to use to open an external document for editing (using a `NSOpenPanel`), the app can create an App-Scoped Bookmark so that it can access the same file again in the future.

- **A Document-Scoped Bookmark provides a specific document persistent access to a sub-file.**

For example, a video editing app that creates a project file that has access to the individual images, video clips and sound files that will later be combined into a single movie.

When the user imports a resource file into the project (via a `NSOpenPanel`), the app creates a Document-Scoped Bookmark for the item that is stored in the project, so that the file is always accessible to the app.

A Document-Scoped Bookmark can be resolved by any application that can open the bookmark data and the document itself. This supports portability, allowing the user to send the project files to another user and having all of the bookmarks work for them as well.

IMPORTANT

A Document-Scoped Bookmark can *only* point to a single file and not a folder and that file cannot be in a location used by the system (such as `/private` or `/Library`).

Using security-scoped bookmarks

Using either type of Security-Scoped Bookmark, requires you to perform the following steps:

1. **Set the appropriate Entitlements in the Xamarin.Mac app that needs to use Security-Scoped Bookmarks** - For App-Scoped Bookmarks, set the `com.apple.security.files.bookmarks.app-scope` Entitlement key to `true`. For Document-Scoped Bookmarks, set the `com.apple.security.files.bookmarks.document-scope` Entitlement key to `true`.
2. **Create a Security-Scoped Bookmark** - You'll do this for any file or folder that the user has provided access to (via `NSOpenPanel` for example), that the app will need persistent access to. Use the

```
public virtual NSData CreateBookmarkData (NSURLBookmarkCreationOptions options, string[] resourceValues, NSURL relativeURL, out NSError error)
```

 method of the `NSURL` class to create the bookmark.
3. **Resolve the Security-Scoped Bookmark** - When the app needs to access the resource again (for example, after restart) it will need to resolve the bookmark to a security-scoped URL. Use the

```
public static NSURL FromBookmarkData (NSData data, NSURLBookmarkResolutionOptions options, NSURL relativeToURL, out bool isStale, out NSError error)
```

 method of the `NSURL` class to resolve the bookmark.
4. **Explicitly notify the System that you want to access to file from the Security-Scoped URL** - This step needs to be done immediately after obtaining the Security-Scoped URL above or, when you later want to regain access to the resource after having relinquished your access to it. Call the

`StartAccessingSecurityScopedResource ()` method of the `NSURL` class to start accessing a Security-Scope URL.

5. Explicitly notify the System that you are done accessing the file from the Security-Scope URL -

As soon as possible, you should inform the System when the app no longer needs access to the file (for example, if the user closes it). Call the `StopAccessingSecurityScopedResource ()` method of the `NSURL` class to stop accessing a Security-Scope URL.

After you relinquish access to a resource, you'll need to return to step 4 again to re-establish access. If the Xamarin.Mac app is restarted, you must return to step 3 and re-resolve the bookmark.

IMPORTANT

Failure to release access to Security-Scope URL resources will cause a Xamarin.Mac app to leak Kernel resources. As a result, the app will no longer be able to add file system locations to its Container until it is restarted.

The App Sandbox and code signing

After you enable the App Sandbox and enable the specific requirements for your Xamarin.Mac app (via Entitlements), you must code sign the project for the sandboxing to take effect. You must perform code signing because the Entitlements required for App Sandboxing are linked to the app's signature.

macOS enforces a link between an app's Container and its code signature, in this way no other application can access that container, even if it is spoofing the app's Bundle ID. This mechanism works as follows:

1. When the System creates the app's Container, it sets an *Access Control List* (ACL) on that Container. The initial access control entry in the list contains the app's *Designated Requirement* (DR), which describes how future versions of the app can be recognized (when it has been upgraded).
2. Each time an app with the same Bundle ID launches, the system checks that the app's code signature matches the Designated Requirements specified in one of the entries in the container's ACL. If the system does not find a match, it prevents the app from launching.

Code Signing works the following ways:

1. Before creating the Xamarin.Mac project, obtain a Development Certificate, a Distribution Certificate and a Developer ID Certificate from the Apple Developer Portal.
2. When the Mac App Store distributes the Xamarin.Mac app, it is signed with an Apple code signature.

When testing and debugging, you'll be using a version of the Xamarin.Mac application that you signed (which will be used to create the App Container). Later, if you wish to test or install the version from the Apple App Store, it will be signed with the Apple signature and will fail to launch (since it doesn't have the same code signature as the original App Container). In this situation, you will get a crash report similar to the following:

```
Exception Type: EXC_BAD_INSTRUCTION (SIGILL)
```

To fix this, you'll need to adjust the ACL entry to point to the Apple signed version of the app.

For more information on creating and downloading the Provisioning Profiles required for Sandboxing, please see the [Signing and Provisioning the App](#) section above.

Adjusting the ACL entry

To allow the Apple signed version of the Xamarin.Mac app to run, do the following:

1. Open the Terminal app (in `/Applications/Utilities`).
2. Open a Finder window to the Apple signed version of the Xamarin.Mac app.
3. Type `asctl container acl add -file` in the Terminal window.

4. Drag the Xamarin.Mac app's icon from the Finder window and drop it on the Terminal window.
5. The full path to the file will be added to the command in Terminal.
6. Press **Enter** to execute the command.

The container's ACL now contains the designated code requirements for both versions of the Xamarin.Mac app and macOS will now allow either version to run.

Display a list of ACL code requirements

You can view a list of the code requirements in a container's ACL by doing the following:

1. Open the Terminal app (in `/Applications/Utilities`).
2. Type `asctl container acl list -bundle <container-name>`.
3. Press **Enter** to execute the command.

The `<container-name>` is usually the Bundle Identifier for the Xamarin.Mac application.

Designing a Xamarin.Mac app for the App Sandbox

There is a common workflow that should be followed when designing a Xamarin.Mac app for the App Sandbox. That said, the specifics of implementing sandboxing in an app are going to be unique to the given app's functionality.

Six steps for adopting the App Sandbox

Designing a Xamarin.Mac app for the App Sandbox typically consists of the following steps:

1. Determine if the app is suitable for sandboxing.
2. Design a development and distribution strategy.
3. Resolve any API incompatibilities.
4. Apply the required App Sandbox Entitlements to the Xamarin.Mac project.
5. Add privilege separation using XPC.
6. Implement a migration strategy.

IMPORTANT

You must not only sandbox the main executable in your app bundle, but also every included helper app or tool in that bundle. This is required for any app distributed from the Mac App Store and, if possible, should be done for any other form of app distribution.

For a list of all executable binaries in a Xamarin.Mac app's bundle, type the following command in Terminal:

```
find -H [Your-App-Bundle].app -print0 | xargs -0 file | grep "Mach-O .*executable"
```

Where `[Your-App-Bundle]` is the name and path to the application's bundle.

Determine whether a Xamarin.Mac app is suitable for sandboxing

Most Xamarin.Mac apps are fully compatible with the App Sandbox and therefore, suitable for sandboxing. If the app requires a behavior that the App Sandbox doesn't allow, you should consider an alternative approach.

If your app requires one of the following behaviors, it is incompatible with the App Sandbox:

- **Authorization Services** - With the App Sandbox, you cannot work with the functions described in [Authorization Services C Reference](#).
- **Accessibility APIs** - You cannot sandbox assistive apps such as screen readers or apps that control other applications.

- **Send Apple Events to Arbitrary Apps** - If the app requires sending Apple events to an unknown, arbitrary app, it cannot be sandboxed. For a known list of called apps, the app can still be sandboxed and the Entitlements will need to include the called app list.
- **Send User Info Dictionaries in Distributed Notifications to other Tasks** - With the App Sandbox, you cannot include a `userInfo` dictionary when posting to an `NSDistributedNotificationCenter` object for messaging other tasks.
- **Load Kernel Extensions** - The loading of Kernel Extensions is prohibited by the App Sandbox.
- **Simulating User Input in Open and Save Dialogs** - Programmatically manipulating Open or Save dialogs to simulate or alter user input is prohibited by the App Sandbox.
- **Accessing or Setting Preferences on other Apps** - Manipulating the settings of other apps is prohibited by the App Sandbox.
- **Configuring Network Settings** - Manipulating Network Settings is prohibited by the App Sandbox.
- **Terminating other Apps** - The App Sandbox prohibits using `NSRunningApplication` to terminate other apps.

Resolving API incompatibilities

When designing a Xamarin.Mac app for the App Sandbox, you may encounter incompatibilities with the usage of some macOS APIs.

Here are a few common issues and things that you can do to solve them:

- **Opening, Saving and Tracking Documents** - If you are managing documents using any technology other than `NSDocument`, you should switch to it because of the built in support for the App Sandbox. `NSDocument` automatically works with PowerBox and provides support for keeping documents within your sandbox if the user moves them in Finder.
- **Retain Access to File System Resources** - If the Xamarin.Mac app depends on persistent access to resources outside of its container, use security-scoped bookmarks to maintain access.
- **Create a Login Item for an App** - With the App Sandbox, you cannot create a login item using `LSSharedFileList` nor can you manipulate the state of launch services using `LSRegisterURL`. Use the `SMLLoginItemSetEnabled` function as described in Apples [Adding Login Items Using the Service Management Framework](#) documentation.
- **Accessing User Data** - If you are using POSIX functions such as `getpwuid` to obtain the user's home directory from directory services, consider using Cocoa or Core Foundation symbols such as `NSHomeDirectory`.
- **Accessing the Preferences of Other Apps** - Because the App Sandbox directs path-finding APIs to the app's container, modifying preferences takes place within that container and accessing another apps preferences is not allowed.
- **Using HTML5 Embedded Video in Web Views** - If the Xamarin.Mac app uses WebKit to play embedded HTML5 videos, you must also link the app against the AV Foundation framework. The App Sandbox will prevent CoreMedia from play these videos otherwise.

Applying required App Sandbox entitlements

You will need to edit the Entitlements for any Xamarin.Mac application that you want to run in the App Sandbox and check the **Enable App Sandboxing** checkbox.

Based on the functionality of the app, you might need to enable other Entitlements to access OS features or resources. App Sandboxing works best when you minimize the entitlements you request to the bare minimum required to run your app so do just randomly enable entitlements.

To determine which entitlements a Xamarin.Mac app requires, do the following:

1. Enable the App Sandbox and run the Xamarin.Mac app.
2. Run through the features of the App.
3. Open the Console app (available in `/Applications/Utilities`) and look for `sandboxd` violations in the All

Messages log.

4. For each `sandboxd` violation, resolve the issue either by using the app container instead of other file system locations or apply App Sandbox Entitlements to enable access to restricted OS features.
5. Re-run and test all Xamarin.Mac app features again.
6. Repeat until all `sandboxd` violations have been resolved.

Add privilege separation using XPC

When developing a Xamarin.Mac app for the App Sandbox, look at the app's behaviors in the terms of privileges and access, then consider separating high-risk operations into their own XPC services.

For more information, see Apple's [Creating XPC Services](#) and [Daemons and Services Programming Guide](#).

Implement a migration strategy

If you are releasing a new, sandboxed version of a Xamarin.Mac application that was not previously sandboxed, you'll need to ensure that current users have a smooth upgrade path.

For details on how to implement a container migration manifest, read Apple's [Migrating an App to a Sandbox](#) documentation.

Summary

This article has taken a detailed look at sandboxing a Xamarin.Mac application. First, we created a simply Xamarin.Mac app to show the basics of the App Sandbox. Next, we showed how to resolve sandbox violations. Then, we took an in-depth look at the App Sandbox and finally, we looked at designing a Xamarin.Mac app for the App Sandbox.

Related Links

- [Publishing to the App Store](#)
- [About App Sandbox](#)
- [Common App Sandboxing issues](#)

Playing sound with AVAudioPlayer in Xamarin.Mac

10/28/2019 • 2 minutes to read • [Edit Online](#)

About the AVAudioPlayer

The `AVAudioPlayer` class is used to playback audio data from either memory or a file. Apple recommends using this class to play audio in your app unless you are doing network streaming or require low latency audio I/O.

You can use the `AVAudioPlayer` class to do the following:

- Play sounds of any duration with optional looping.
- Play multiple sounds at the same time with optional synchronization.
- Control volume, playback rate and stereo positioning for each sounds playing.
- Support features such as fast forward or rewind.
- Obtain playback level metering data.

`AVAudioPlayer` supports sounds in any audio format provided by iOS, tvOS and macOS such as .aif, .wav or .mp3.

Playing sounds in macOS

Because macOS supports the same Audio Toolbox classes as iOS, please see our iOS [Playing Sound with AVAudioPlayer](#) documentation for the full details of playing audio in a Xamarin.Mac app.

Related Links

- [AVAudioPlayer Reference](#)

macOS user interface controls in Xamarin.Mac

11/2/2020 • 3 minutes to read • [Edit Online](#)

This article links to guides that describe various macOS UI controls.

When working with C# and .NET in a Xamarin.Mac application, you have access to the same user interface controls that a developer working in *Objective-C* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, you can use Xcode's *Interface Builder* to create and maintain your user interfaces (or optionally create them directly in C# code).

The guides listed below give detailed information about working with macOS UI elements in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in every article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, as it explains the `Register` and `Export` attributes used to wire-up your C# classes to Objective-C objects and UI elements.

Windows

This article covers working with windows and panels in a Xamarin.Mac application. It covers creating and maintaining windows and panels in Xcode and Interface Builder, loading windows and panels from .storyboard or .xib files, using windows, and responding to windows in C# code.

Dialogs

This article covers working with dialogs and modal windows in a Xamarin.Mac application. It covers creating and maintaining modal windows in Xcode and Interface Builder, working with standard dialogs, and displaying and responding to windows in C# code.

Alerts

This article covers working with alerts in a Xamarin.Mac application. It covers creating and displaying alerts from C# code and responding to alerts.

Menus

Menus are used in various parts of a Mac application's user interface; from the application's main menu at the top of the screen to pop-up menus and contextual menus that can appear anywhere in a window. Menus are an integral part of a Mac application's user experience. This article covers working with Cocoa menus in a Xamarin.Mac application.

Standard controls

Working with the standard AppKit controls such as buttons, labels, text fields, check boxes, and segmented controls in a Xamarin.Mac application. This guide covers adding them to a user interface design in Xcode's Interface Builder, exposing them to code through outlets and actions, and working with AppKit controls in C# code.

Toolbars

This article covers working with toolbars in a Xamarin.Mac application. It covers creating and maintaining toolbars in Xcode and Interface Builder, how to expose the toolbar items to code using outlets and actions, enabling and disabling toolbar items, and finally responding to Toolbar items in C# code.

Table views

This article covers working with table views in a Xamarin.Mac application. It covers creating and maintaining table views in Xcode and Interface Builder, how to expose the table view items to code using outlets and actions, populating table views, and responding to table view items in C# code.

Outline views

This article covers working with outline views in a Xamarin.Mac application. It covers creating and maintaining outline views in Xcode and Interface Builder, how to expose the outline view items to code using outlets and actions, populating outline views, and responding to outline view items in C# code.

Source lists

This article covers working with source lists in a Xamarin.Mac application. It covers creating and maintaining source lists in Xcode and Interface Builder, how to expose source list items to code using outlets and actions, populating source lists, and responding to source list items in C# code.

Collection views

This article covers working with collection views in a Xamarin.Mac application. It covers creating and maintaining collection views in Xcode and Interface Builder, how to expose the collection view items to code using outlets and actions, populating collection views, and responding to collection views in C# code.

Creating custom controls

This article covers creating custom user interface controls (by inheriting from `NSControl`), drawing a custom interface for the control, and creating custom actions that can be used with Xcode's Interface Builder.

Mac samples gallery

We also suggest taking a look at the [Mac Samples Gallery](#). It includes a wealth of ready-to-use code that can help you get a Xamarin.Mac project off the ground quickly.

Related links

- [macOS Human Interface Guidelines](#)

Windows in Xamarin.Mac

11/2/2020 • 25 minutes to read • [Edit Online](#)

This article covers working with windows and panels in a Xamarin.Mac application. It describes creating windows and panels in Xcode and Interface Builder, loading them from storyboards and .xib files, and working with them programmatically.

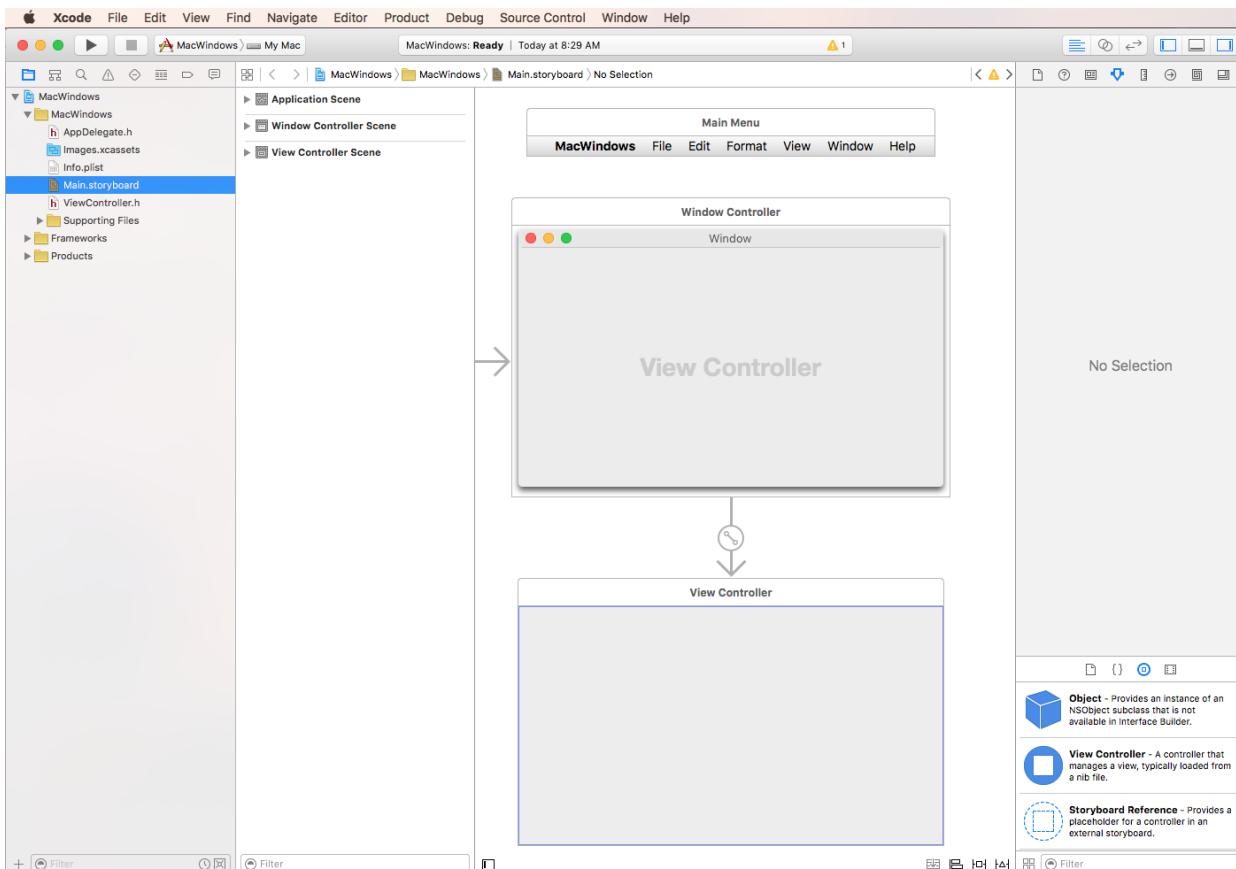
When working with C# and .NET in a Xamarin.Mac application, you have access to the same Windows and Panels that a developer working in *Objective-C* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, you can use Xcode's *Interface Builder* to create and maintain your Windows and Panels (or optionally create them directly in C# code).

Based on its purpose, a Xamarin.Mac application can present one or more Windows on screen to manage and coordinate the information it displays and works with. The principal functions of a window are:

1. To provide an area in which Views and Controls can be placed and managed.
2. To accept and respond to events in response to user interaction with both the keyboard and mouse.

Windows can be used in a Modeless state (such as a text editor that can have multiple documents open at once) or Modal (such as an Export dialog that must be dismissed before the application can continue).

Panels are a special kind of Window (a subclass of the base `NSWindow` class), that typically serve an auxiliary function in an application, such as utility windows like Text format inspectors and system Color Picker.



In this article, we'll cover the basics of working with Windows and Panels in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` commands used to wire-up your C# classes to Objective-C objects and UI Elements.

Introduction to windows

As stated above, a Window provides an area in which Views and Controls can be placed and managed and responds to events based on user interaction (either via keyboard or mouse).

According to Apple, there are five main types of Windows in a macOS App:

- **Document Window** - A document window contains file-based user data such as a spreadsheet or a text document.
- **App Window** - An app window is the main window of an application that is not document-based (like the Calendar app on a Mac).
- **Panel** - A panel floats above other windows and provides tools or controls that users can work with while documents are open. In some cases, a panel can be translucent (such as when working with large graphics).
- **Dialog** - A dialog appears in response to a user action and typically provides ways users can complete the action. A dialog requires a response from the user before it can be closed. (See [Working with Dialogs](#))
- **Alerts** - An alert is a special type of dialog that appears when a serious problem occurs (such as an error) or as a warning (such as preparing to delete a file). Because an alert is a dialog, it also requires a user response before it can be closed. (See [Working with Alerts](#))

For more information, see the [About Windows](#) section of Apple's [macOS design themes](#).

Main, key, and inactive windows

Windows in a Xamarin.Mac application can look and behave differently based on how the user is currently interacting with them. The foremost Document or App Window that is currently focus of the user's attention is called the *Main Window*. In most instances this Window will also be the *Key Window* (the window that is currently accepting user input). But this isn't always the case, for example, a Color Picker could be open and be the Key window that the user is interacting with to change the state of an item in the Document Window (which would still be the Main Window).

The Main and Key Windows (if they are separate) are always active, *Inactive Windows* are open windows that are not in the foreground. For example, a text editor application could have more than one document open at a time, only the Main Window would be active, all others would be inactive.

For more information, see the [About Windows](#) section of Apple's [macOS design themes](#).

Naming windows

A Window can display a Title Bar and when the Title is displayed, it's usually the name of the application, the name of the document being worked on or the function of the window (such as Inspector). Some applications don't display a Title Bar because they are recognizable by sight and don't work with documents.

Apple suggest the following guidelines:

- Use your application name for the title of a main, non-document window.
- Name a new document window `untitled`. For the first new document, don't append a number to the Title (such as `untitled 1`). If the user creates another new document before saving and titling the first, call that window `untitled 2`, `untitled 3`, etc.

For more information, see the [Naming Windows](#) section of Apple's [macOS design themes](#).

Full-screen windows

In macOS, an application's window can go full screen hiding everything including the Application Menu Bar

(which can be revealed by moving the cursor to the top of the screen) to provide distraction free interaction with its content.

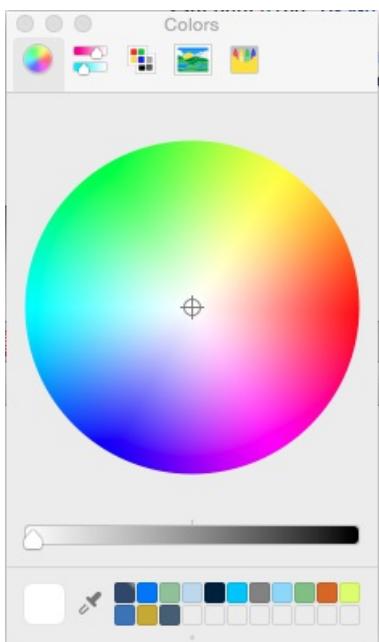
Apple suggests the following guidelines:

- Determine whether it makes sense for a window to go full screen. Applications that provide brief interactions (such as a Calculator) shouldn't provide a full screen mode.
- Show the toolbar if the full-screen task requires it. Typically the toolbar is hidden while in full screen mode.
- The full-screen window should have all the features users need to complete the task.
- If possible, avoid Finder interaction while the user is in a full-screen window.
- Take advantage of the increased screen space without shifting the focus away from the main task.

For more information, see the [Full-Screen Windows](#) section of Apple's [macOS design themes](#).

Panels

A Panel is an auxiliary window that contains controls and options that affect the active document or selection (such as the system Color Picker):



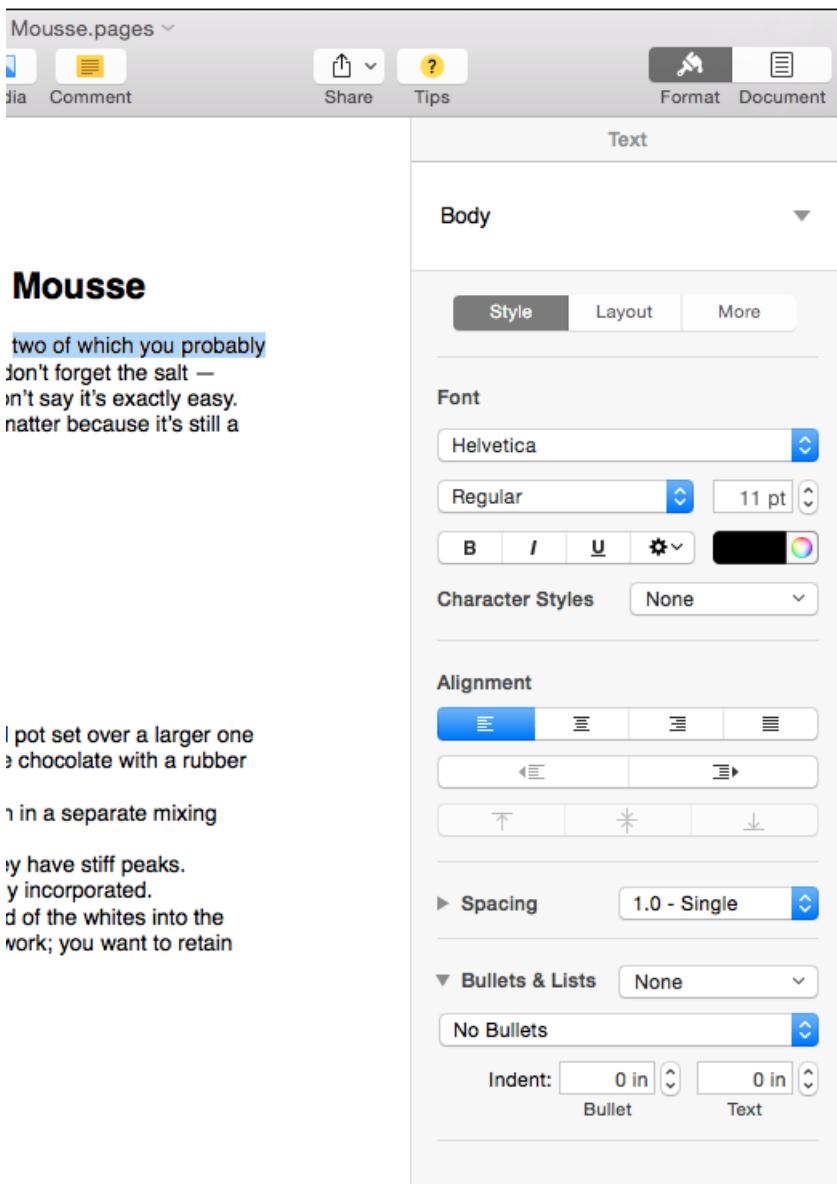
Panels can be either *App-Specific* or *Systemwide*. App-Specific Panels float over the top of the application's document windows and disappear when the application is in the background. Systemwide Panels (such as the **Fonts** panel), float on top of all open windows no matter the application.

Apple suggests the following guidelines:

- In general, use a standard panel, transparent panels should only be used sparingly and for graphically intensive tasks.
- Consider using a panel to give users easy access to important controls or information that directly affects their task.
- Hide and show panels as required.
- Panels should always include title bar.
- Panels should not include an active minimize button.

Inspectors

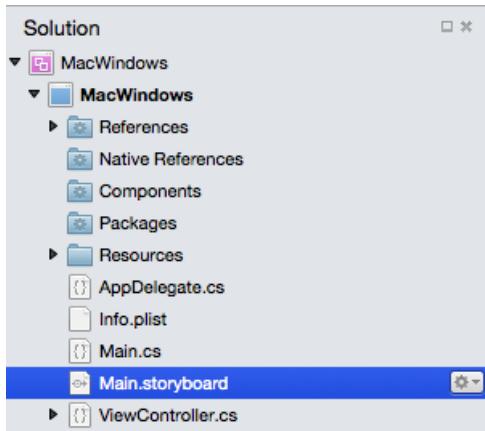
Most modern macOS applications present auxiliary controls and options that affect the active document or selection as *Inspectors* that are part of the Main Window (like the **Pages** app shown below), instead of using Panel Windows:



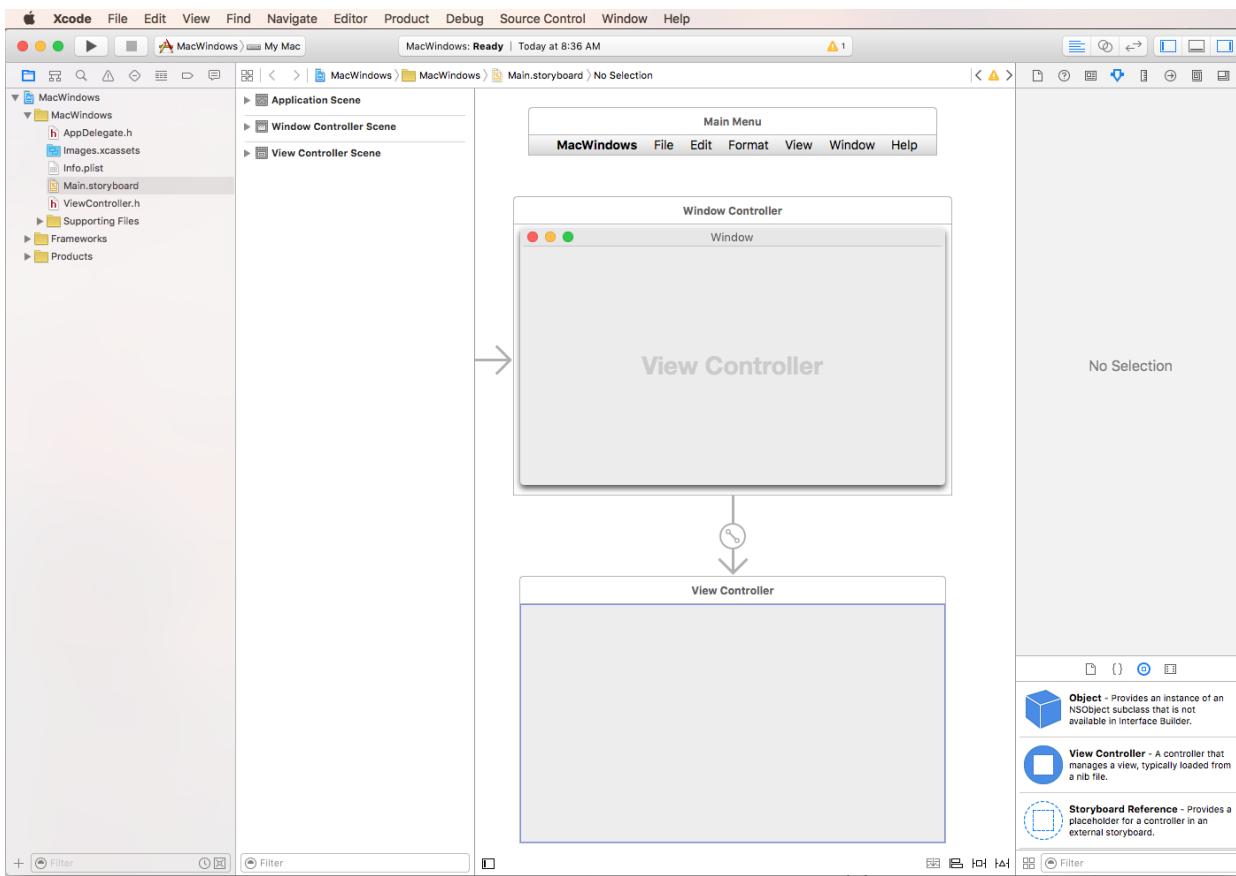
For more information, see the [Panels](#) section of Apple's [macOS design themes](#) and our [MacInspector](#) sample app for a full implementation of an [Inspector Interface](#) in a Xamarin.Mac app.

Creating and maintaining windows in Xcode

When you create a new Xamarin.Mac Cocoa application, you get a standard blank, window by default. This window is defined in a `.Storyboard` file automatically included in the project. To edit your windows design, in the **Solution Explorer**, double click the `Main.storyboard` file:



This will open the window design in Xcode's Interface Builder:



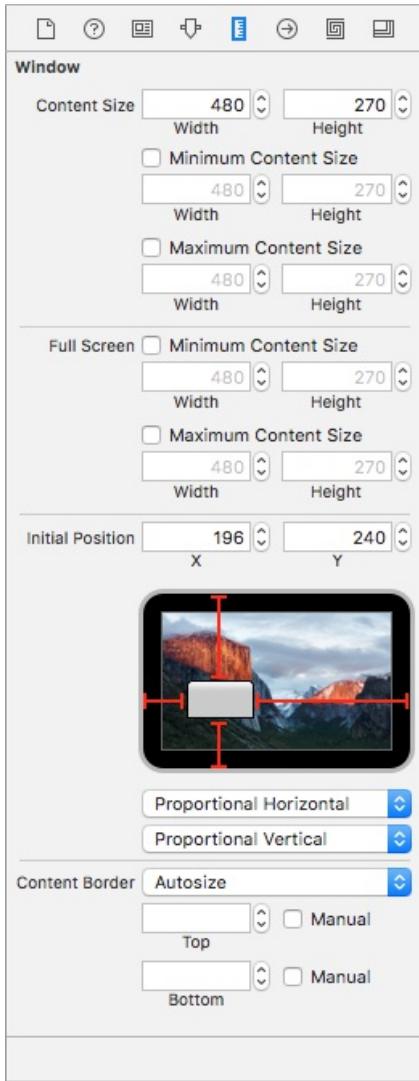
In the **Attribute Inspector**, there are several properties that you can use to define and control your window:

- **Title** - This is the text that will be displayed in the window's titlebar.
- **Autosave** - This is the *key* that will be used to ID the window when its position and settings are automatically saved.
- **Title Bar** - Does the window display a title bar.
- **Unified Title and Toolbar** - If the window includes a Toolbar, should it be part of the title bar.
- **Full Sized Content View** - Allows the content area of the window to be under the Title bar.
- **Shadow** - Does the window have a shadow.
- **Textured** - Textured windows can use effects (like vibrancy) and can be moved around by dragging anywhere on their body.
- **Close** - Does the window have a close button.
- **Minimize** - Does the window have a minimize button.
- **Resize** - Does the window have a resize control.
- **Toolbar Button** - Does the window have a hide/show toolbar button.
- **Restorable** - Is the window's position and settings automatically saved and restored.
- **Visible At Launch** - Is the window automatically shown when the `.xib` file is loaded.
- **Hide On Deactivate** - Is the window hidden when the application enters the background.
- **Release When Closed** - Is the window purged from memory when it is closed.
- **Always Display Tooltips** - Are the tooltips constantly displayed.
- **Recalculates View Loop** - Is the view order recalculated before the window is drawn.
- **Spaces, Exposé and Cycling** - All define how the window behaves in those macOS environments.
- **Full Screen** - Determines if this window can enter the full screen mode.
- **Animation** - Controls the type of animation available for the window.
- **Appearance** - Controls the appearance of the window. For now there is only one appearance, Aqua.

See Apple's [Introduction to Windows](#) and [NSWindow](#) documentation for more details.

Setting the default size and location

To set the initial position of your window and to control its size, switch to the **Size Inspector**:



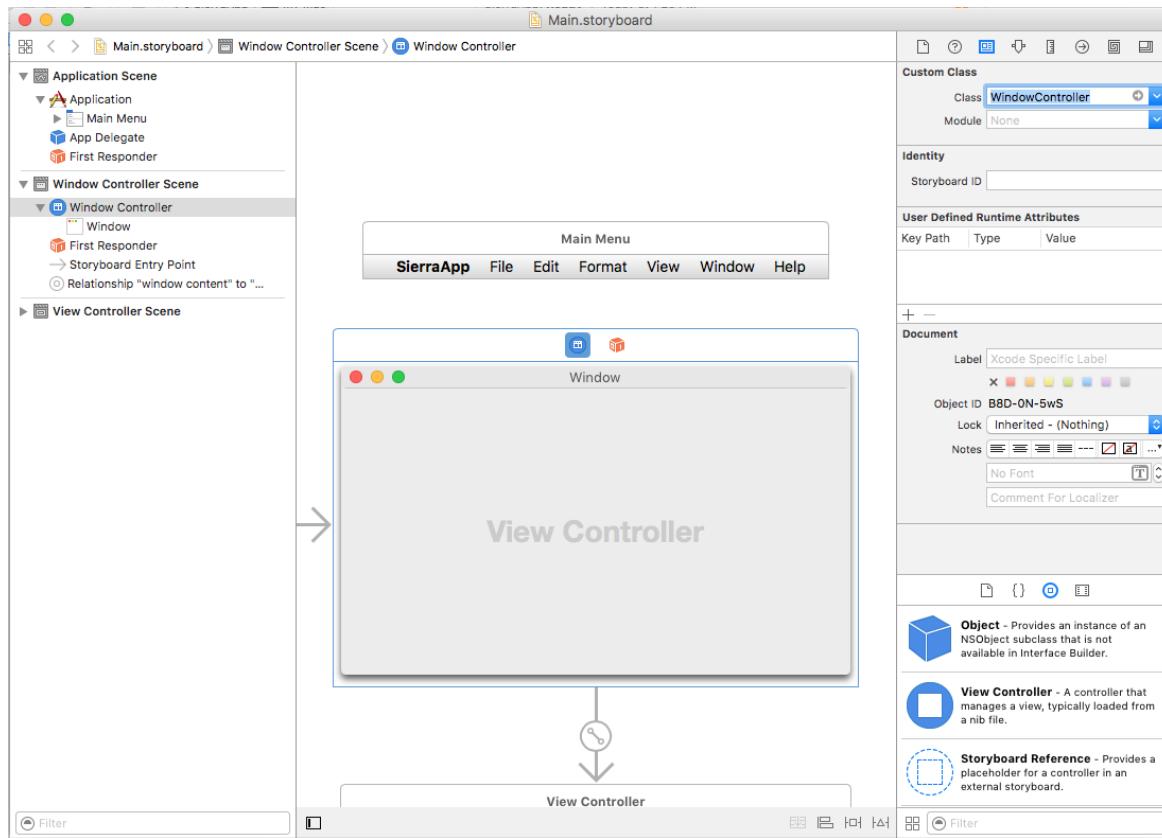
From here you can set the initial size of the window, give it a minimum and maximum size, set the initial location on the screen and control the borders around the window.

Setting a custom main window controller

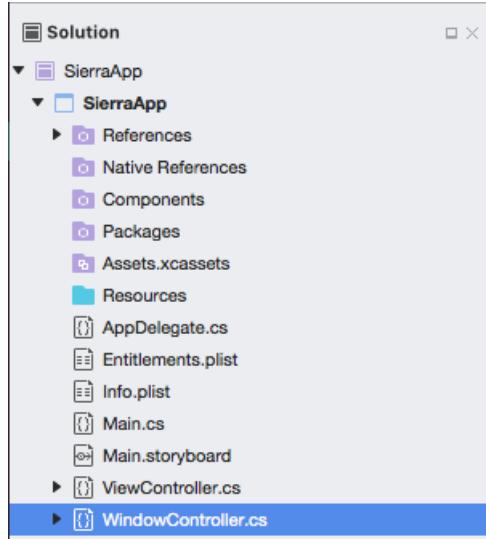
To be able to create Outlets and Actions to expose UI elements to C# code, the Xamarin.Mac app will need to be using a Custom Window Controller.

Do the following:

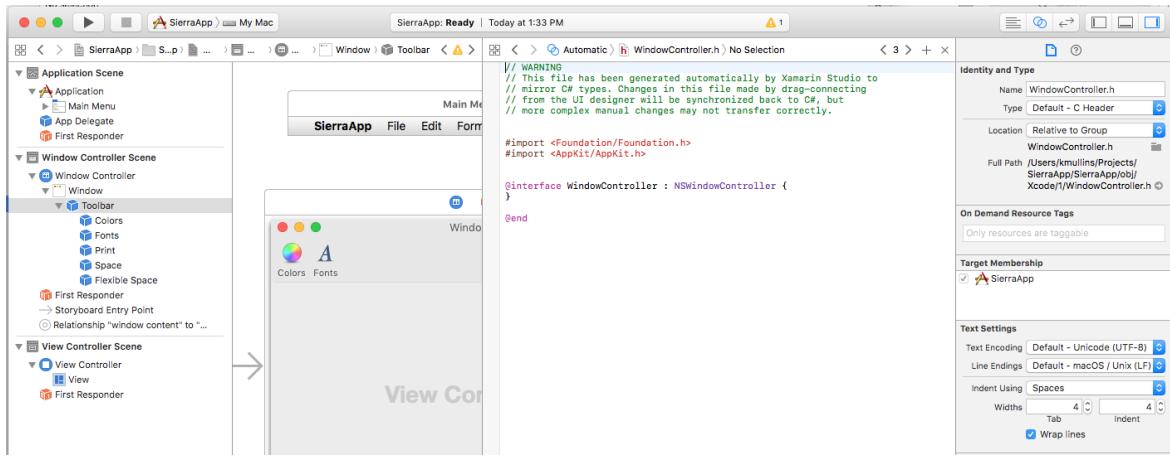
1. Open the app's Storyboard in Xcode's Interface Builder.
2. Select the `NSWindowController` in the Design Surface.
3. Switch to the **Identity Inspector** view and enter `WindowController` as the **Class Name**:



4. Save your changes and return to Visual Studio for Mac to sync.
5. A `WindowController.cs` file will be added to your Project in the Solution Explorer in Visual Studio for Mac:



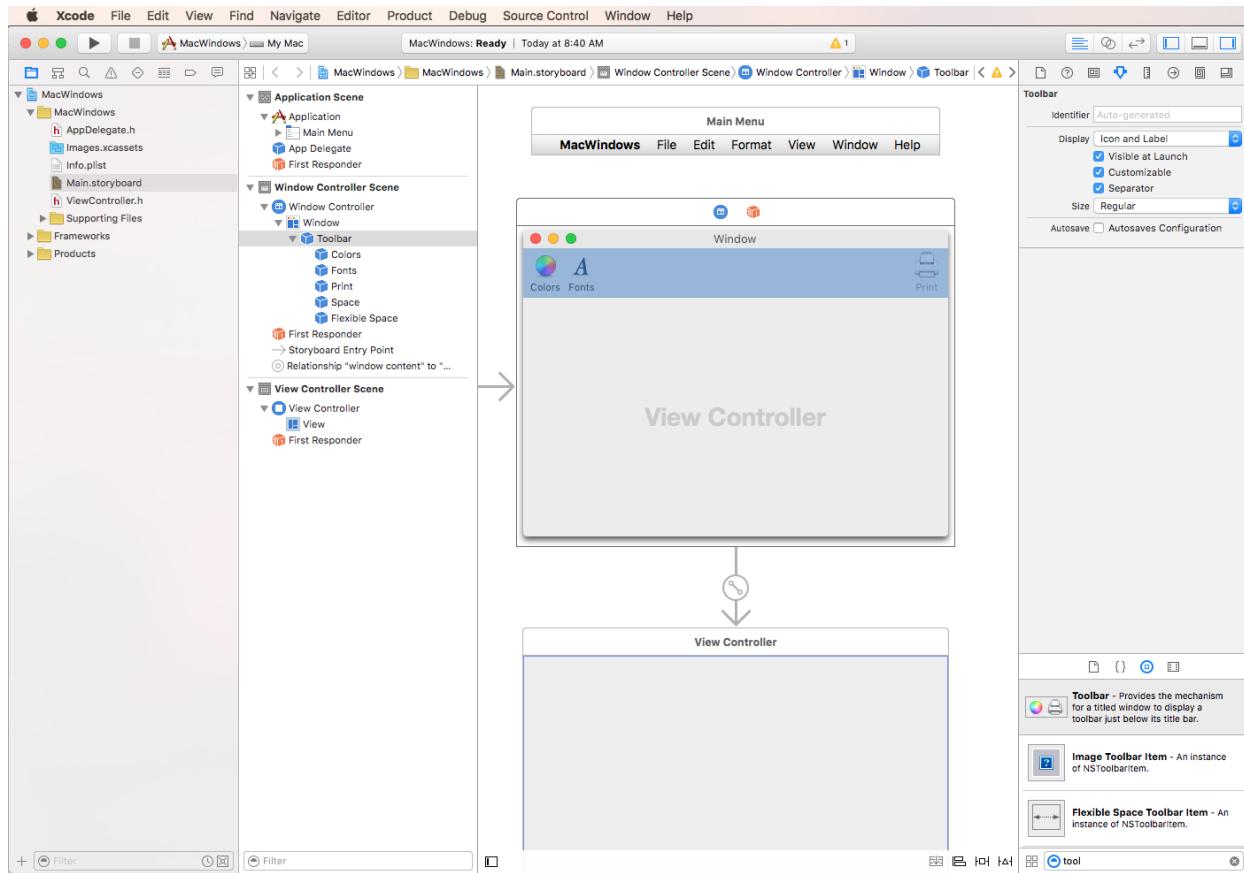
6. Reopen the Storyboard in Xcode's Interface Builder.
7. The `WindowController.h` file will be available for use:



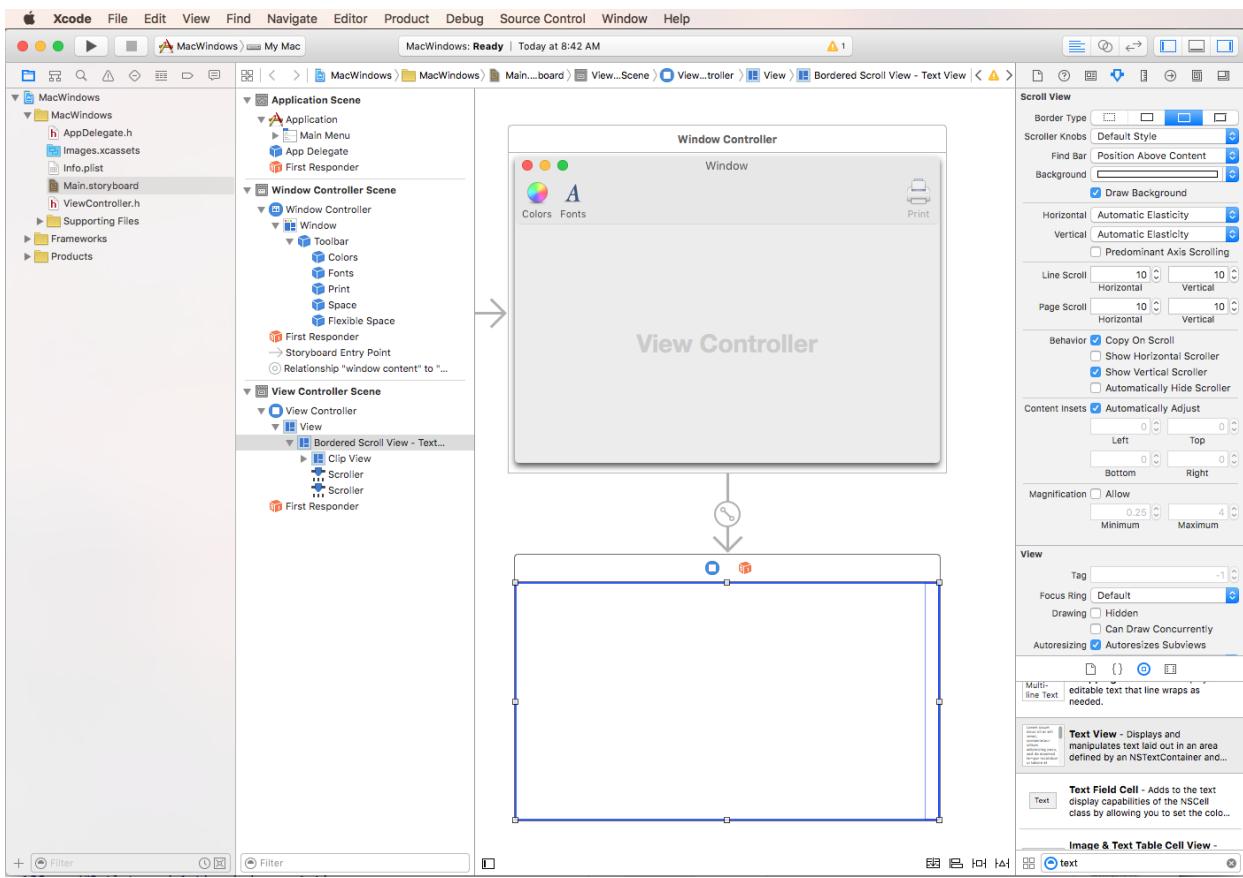
Adding UI elements

To define the content of a window, drag controls from the **Library Inspector** onto the **Interface Editor**. Please see our [Introduction to Xcode and Interface Builder](#) documentation for more information about using Interface Builder to create and enable controls.

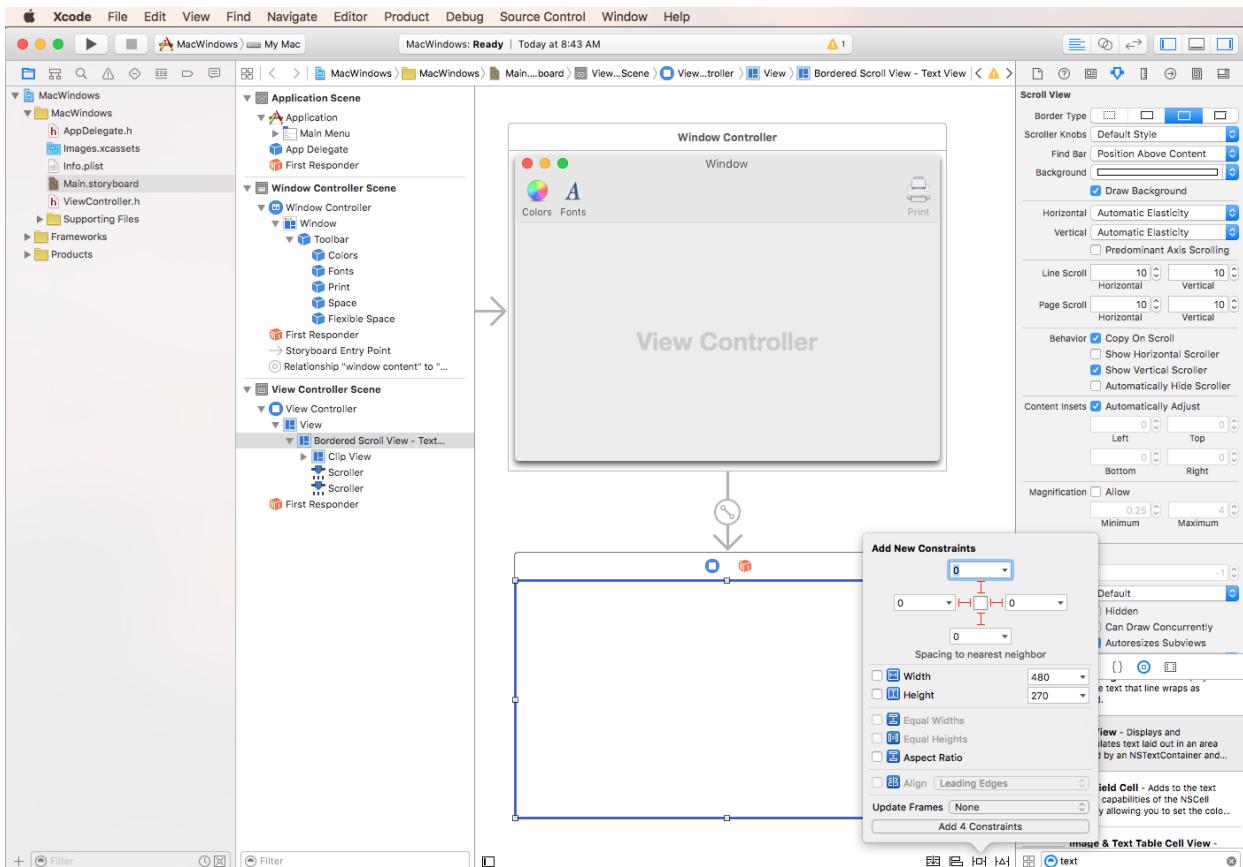
As an example, let's drag a Toolbar from the **Library Inspector** onto the window in the **Interface Editor**:



Next, drag in a **Text View** and size it to fill the area under the toolbar:

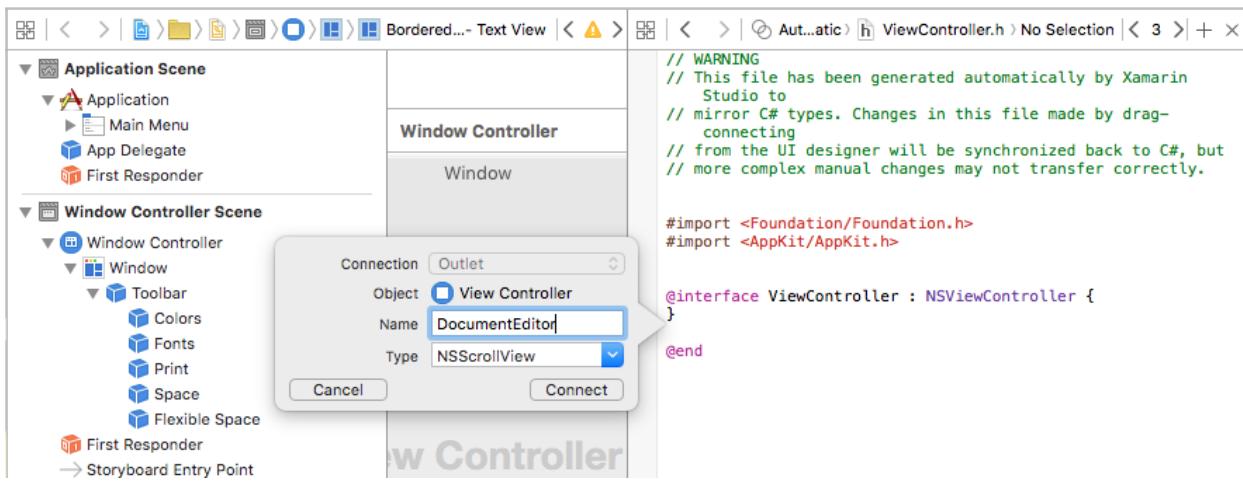


Since we want the **Text View** to shrink and grow as the window's size changes, let's switch to the **Constraint Editor** and add the following constraints:



By clicking the four **Red I-Beams** at the top of the editor and clicking **Add 4 Constraints**, we are telling the text view to stick to the given X,Y coordinates and grow or shrink horizontally and vertically as the window is resized.

Finally, expose the **Text View** to code using an **Outlet** (making sure to select the `ViewController.h` file):



Save your changes and switch back to Visual Studio for Mac to sync with Xcode.

For more information about working with **Outlets** and **Actions**, please see our [Outlet and Action](#) documentation.

Standard window workflow

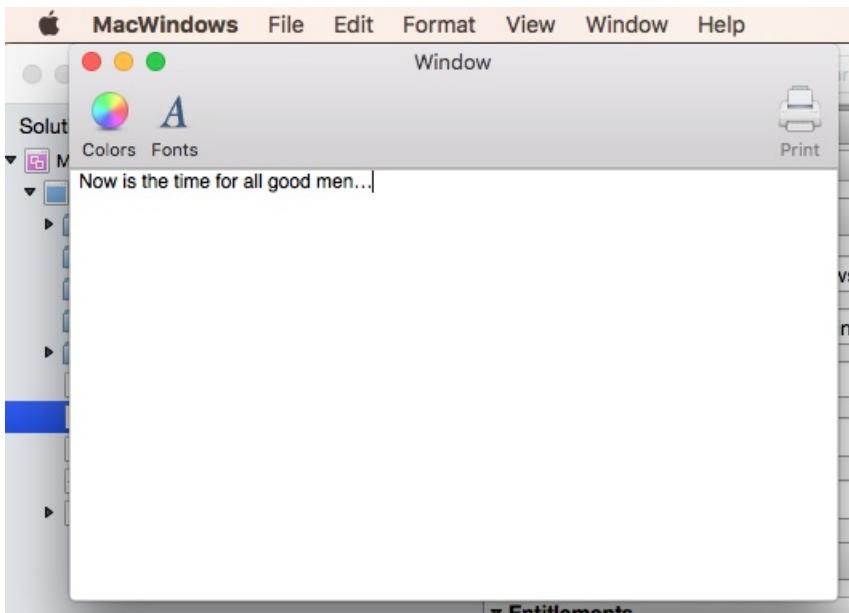
For any window that you create and work with in your Xamarin.Mac application, the process is basically the same as what we have just done above:

1. For new windows that are not the default added automatically to your project, add a new window definition to the project. This will be discussed in detail below.
2. Double-click the `Main.storyboard` file to open the window design for editing in Xcode's Interface Builder.
3. Drag a new Window into the User Interface's design and hook the window into Main Window using *Segues* (for more information see the [Segues](#) section of our [Working with Storyboards](#) documentation).
4. Set any required window properties in the **Attribute Inspector** and the **Size Inspector**.
5. Drag in the controls required to build your interface and configure them in the **Attribute Inspector**.
6. Use the **Size Inspector** to handle the resizing for your UI Elements.
7. Expose the window's UI elements to C# code via **Outlets** and **Actions**.
8. Save your changes and switch back to Visual Studio for Mac to sync with Xcode.

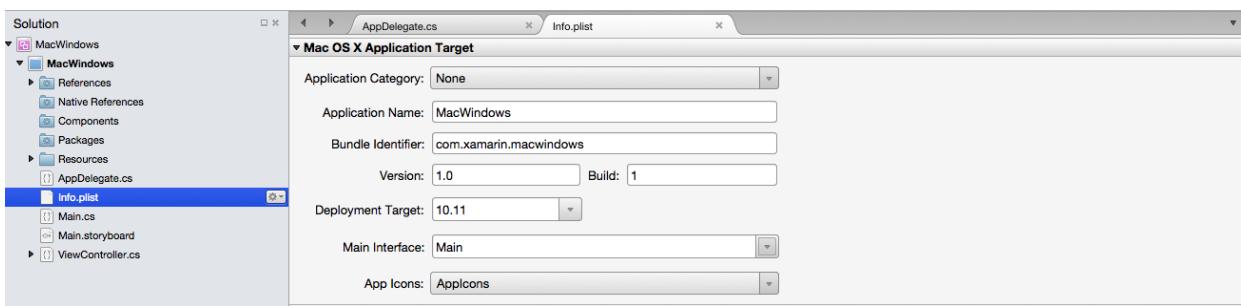
Now that we have a basic window created, we'll look at the typical processes a Xamarin.Mac application does when working with windows.

Displaying the default window

By default, a new Xamarin.Mac application will automatically display the window defined in the `MainWindow.xib` file when it is started:

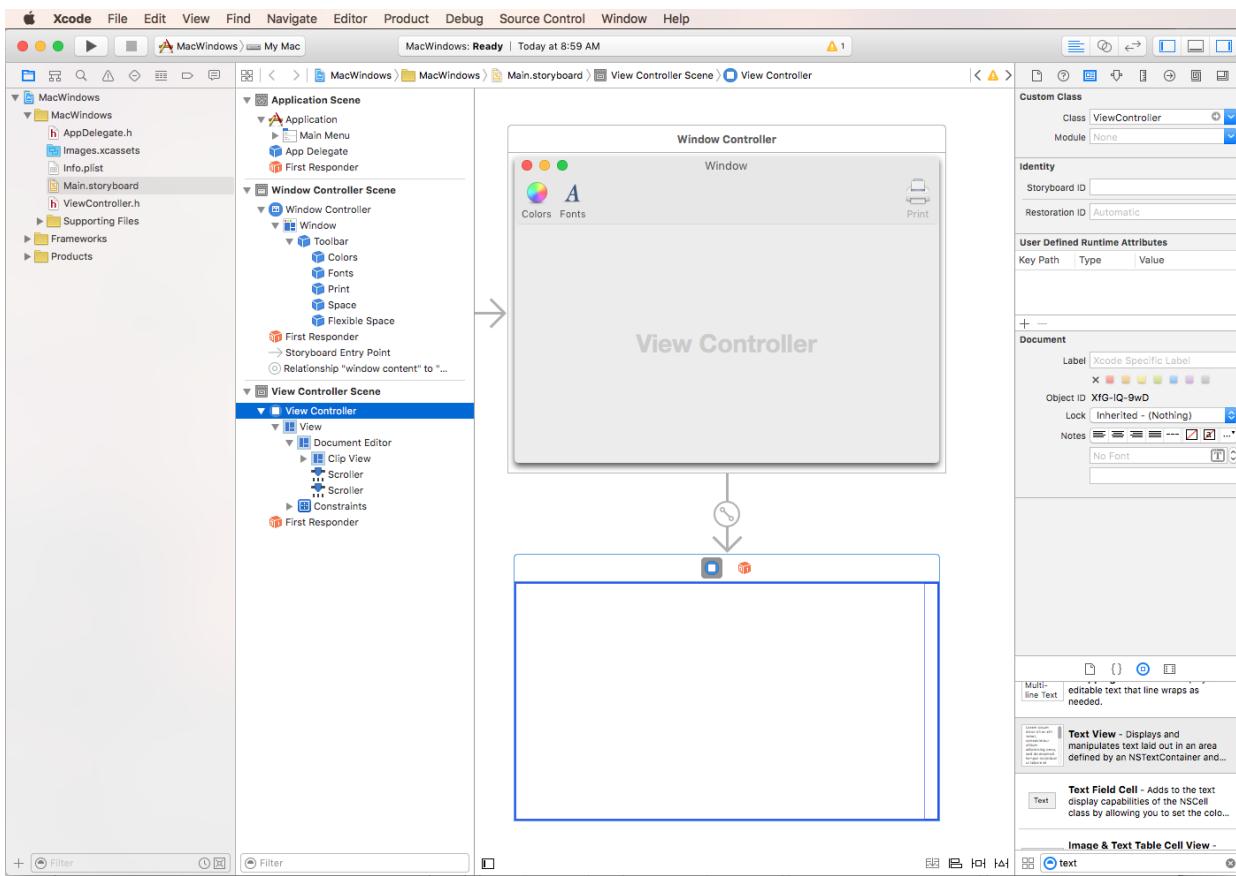


Since we modified the design of that window above, it now includes a default Toolbar and **Text View** control. The following section in the `Info.plist` file is responsible for displaying this window:



The **Main Interface** dropdown is used to select the Storyboard that will be used as the main app UI (in this case `Main.storyboard`).

A View Controller is automatically added to the project to control that Main Windows that is displayed (along with its primary View). It is defined in the `ViewController.cs` file and attached to the **File's Owner** in Interface Builder under the **Identity Inspector**:



For our window, we'd like it to have a title of `untitled` when it first opens so let's override the `ViewWillAppear` method in the `viewController.cs` to look like the following:

```
public override void ViewWillAppear ()
{
    base.ViewWillAppear ();

    // Set Window Title
    this.View.Window.Title = "untitled";
}
```

NOTE

The window's `Title` property is set in the `ViewWillAppear` method instead of the `ViewDidLoad` method because, while the view might be loaded into memory, it is not yet fully instantiated. Accessing the `Title` property in the `ViewDidLoad` method we will get a `null` exception since the window hasn't been constructed and wired-up to the property yet.

Programmatically closing a window

There might be times that you wish to programmatically close a window in a Xamarin.Mac application, other than having the user click the window's **Close** button or using a menu item. macOS provides two different ways to close an `NSWindow` programmatically: `PerformClose` and `Close`.

PerformClose

Calling the `PerformClose` method of an `NSWindow` simulates the user clicking the window's **Close** button by momentarily highlighting the button and then closing the window.

If the application implements the `NSWindow`'s `WillClose` event, it will be raised before the window is closed. If the event returns `false`, then the window will not be closed. If the window does not have a **Close** button or

cannot be closed for any reason, the OS will emit the alert sound.

For example:

```
MyWindow.PerformClose(this);
```

Would attempt to close the `MyWindow` `NSWindow` instance. If it was successful, the window will be closed, else the alert sound will be emitted and the window will stay open.

Close

Calling the `close` method of an `NSWindow` does not simulate the user clicking the window's **Close** button by momentarily highlighting the button, it simply closes the window.

A window does not have to be visible to be closed and an `NSNotification` notification will be posted to the default Notification Center for the window being closed.

The `close` method differs in two important ways from the `PerformClose` method:

1. It does not attempt to raise the `willClose` event.
2. It does not simulate the user clicking the **Close** button by momentarily highlighting the button.

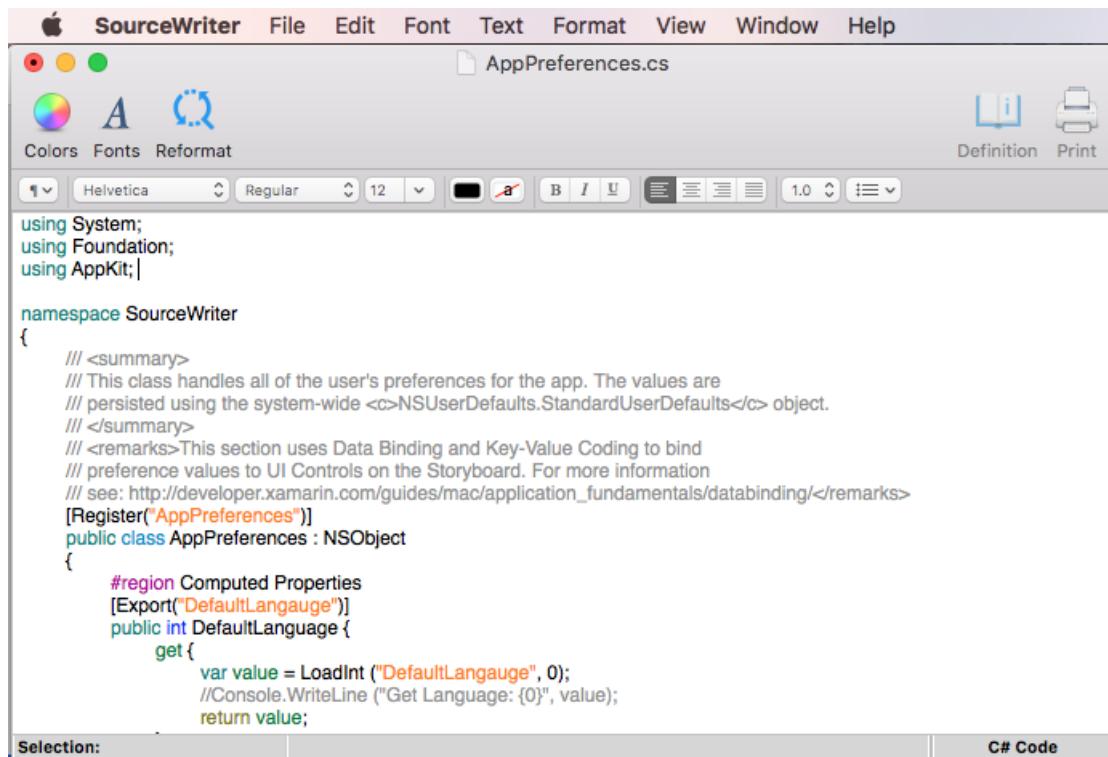
For example:

```
MyWindow.Close();
```

Would close the `MyWindow` `NSWindow` instance.

Modified windows content

In macOS, Apple has provided a way to inform the user that the contents of a Window (`NSWindow`) has been modified by the user and needs to be saved. If the Window contains modified content, a small black dot will be displayed in its **Close** widget:



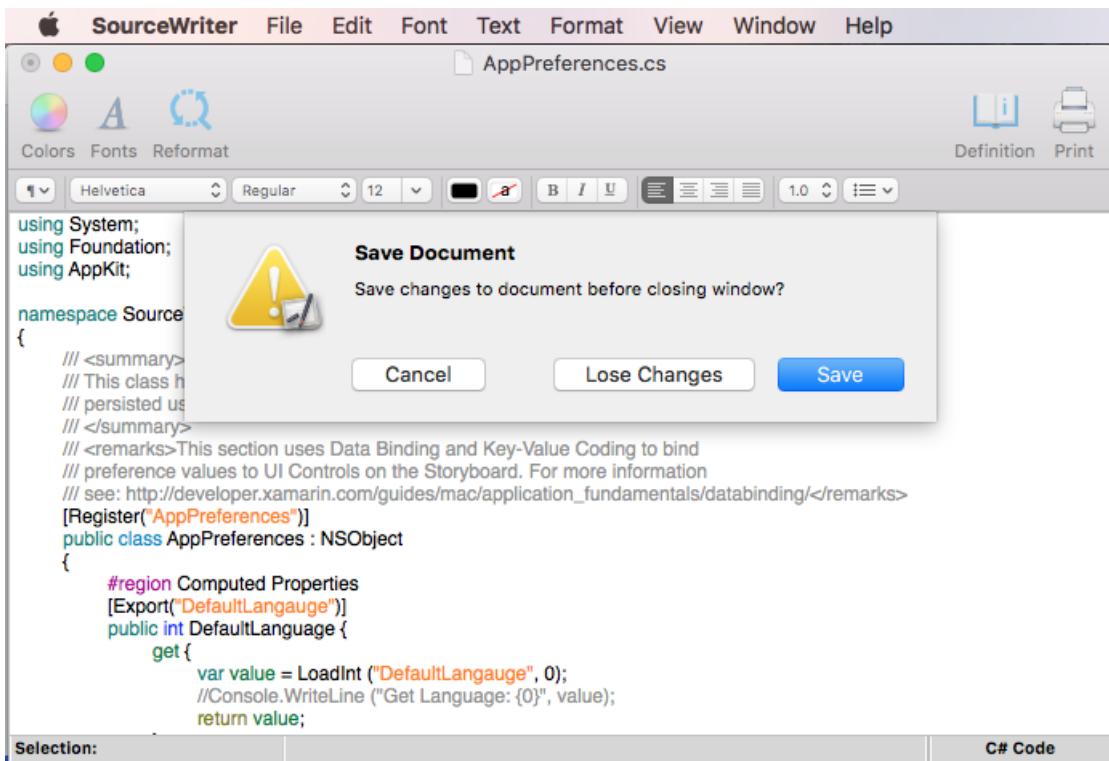
The screenshot shows the SourceWriter application interface. The menu bar includes Apple, SourceWriter, File, Edit, Font, Text, Format, View, Window, and Help. The toolbar includes standard Mac OS X icons for Colors, Fonts, Reformat, Definition, and Print. The main editor area displays the following C# code:

```
using System;
using Foundation;
using AppKit;

namespace SourceWriter
{
    /// <summary>
    /// This class handles all of the user's preferences for the app. The values are
    /// persisted using the system-wide <>c>NSUserDefaults.StandardUserDefaults</c> object.
    /// </summary>
    /// <remarks>This section uses Data Binding and Key-Value Coding to bind
    /// preference values to UI Controls on the Storyboard. For more information
    /// see: http://developer.xamarin.com/guides/mac/application\_fundamentals/databinding/</remarks>
    [Register("AppPreferences")]
    public class AppPreferences : NSObject
    {
        #region Computed Properties
        [Export("DefaultLangauge")]
        public int DefaultLanguage {
            get {
                var value = LoadInt ("DefaultLangauge", 0);
                //Console.WriteLine ("Get Language: {0}", value);
                return value;
            }
        }
    }
}
```

If the user attempts to close the Window or quit the Mac App while there are unsaved changes to the Window's

content, you should present a [Dialog Box](#) or [Modal Sheet](#) and allow the user to save their changes first:



Marking a window as modified

To mark a Window as having modified content, use the following code:

```
// Mark Window content as modified  
Window.DocumentEdited = true;
```

And once the change has been saved, clear the modified flag using:

```
// Mark Window content as not modified  
Window.DocumentEdited = false;
```

Saving changes before closing a window

To watch for the user closing a Window and allowing them to save modified content beforehand, you will need to create a subclass of `NSWindowDelegate` and override its `WindowShouldClose` method. For example:

```
using System;  
using AppKit;  
using System.IO;  
using Foundation;  
  
namespace SourceWriter  
{  
    public class EditorWindowDelegate : NSWindowDelegate  
    {  
        #region Computed Properties  
        public NSWindow Window { get; set; }  
        #endregion  
  
        #region Constructors  
        public EditorWindowDelegate (NSWindow window)  
        {  
            // Initialize  
            this.Window = window;  
        }  
    }  
}
```

```

        }

        #endregion

        #region Override Methods
        public override bool WindowShouldClose (Foundation.NSObject sender)
        {
            // Is the window dirty?
            if (Window.DocumentEdited) {
                var alert = new NSAlert () {
                    AlertStyle = NSAlertStyle.Critical,
                    InformativeText = "Save changes to document before closing window?",
                    MessageText = "Save Document",
                };
                alert.AddButton ("Save");
                alert.AddButton ("Lose Changes");
                alert.AddButton ("Cancel");
                var result = alert.RunSheetModal (Window);

                // Take action based on result
                switch (result) {
                    case 1000:
                        // Grab controller
                        var viewController = Window.ContentViewController as ViewController;

                        // Already saved?
                        if (Window.RepresentedUrl != null) {
                            var path = Window.RepresentedUrl.Path;

                            // Save changes to file
                            File.WriteAllText (path, viewController.Text);
                            return true;
                        } else {
                            var dlg = new NSSavePanel ();
                            dlg.Title = "Save Document";
                            dlg.BeginSheet (Window, (rslt) => {
                                // File selected?
                                if (rslt == 1) {
                                    var path = dlg.Url.Path;
                                    File.WriteAllText (path, viewController.Text);
                                    Window.DocumentEdited = false;
                                    viewController.View.WindowSetTitleWithRepresentedFilename
                                        (Path.GetFileName(path));
                                    viewController.View.Window.RepresentedUrl = dlg.Url;
                                    Window.Close();
                                }
                            });
                            return true;
                        }
                    return false;
                }
                case 1001:
                    // Lose Changes
                    return true;
                case 1002:
                    // Cancel
                    return false;
            }
        }

        return true;
    }
    #endregion
}

```

Use the following code to attach an instance of this delegate to the window:

```
// Set delegate  
Window.Delegate = new EditorWindowDelegate(Window);
```

Saving changes before closing the app

Finally, your Xamarin.Mac App should check to see if any of its Windows contain modified content and allow the user to save the changes before quitting. To do this, edit your `AppDelegate.cs` file, override the `ApplicationShouldTerminate` method and make it look like the following:

```
public override NSApplicationTerminateReply ApplicationShouldTerminate (NSApplication sender)  
{  
    // See if any window needs to be saved first  
    foreach (NSWindow window in NSApplication.SharedApplication.Windows) {  
        if (window.Delegate != null && !window.Delegate.WindowShouldClose (this)) {  
            // Did the window terminate the close?  
            return NSApplicationTerminateReply.Cancel;  
        }  
    }  
  
    // Allow normal termination  
    return NSApplicationTerminateReply.Now;  
}
```

Working with multiple windows

Most document based Mac applications can edit multiple documents at the same time. For example, a text editor can have multiple text files open for edit at the same time. By default, a new Xamarin.Mac application has a **File** menu with a **New** item automatically wired-up to the `newDocument: Action`.

The code below will activate this new item and allow the user to open multiple copies of the Main Window to edit multiple documents at once.

Edit the `AppDelegate.cs` file and add the following computed property:

```
public int UntitledWindowCount { get; set;} =1;
```

Use this to track the number of unsaved files so we can give feedback to the user (per Apple's guidelines as discussed above).

Next, add the following method:

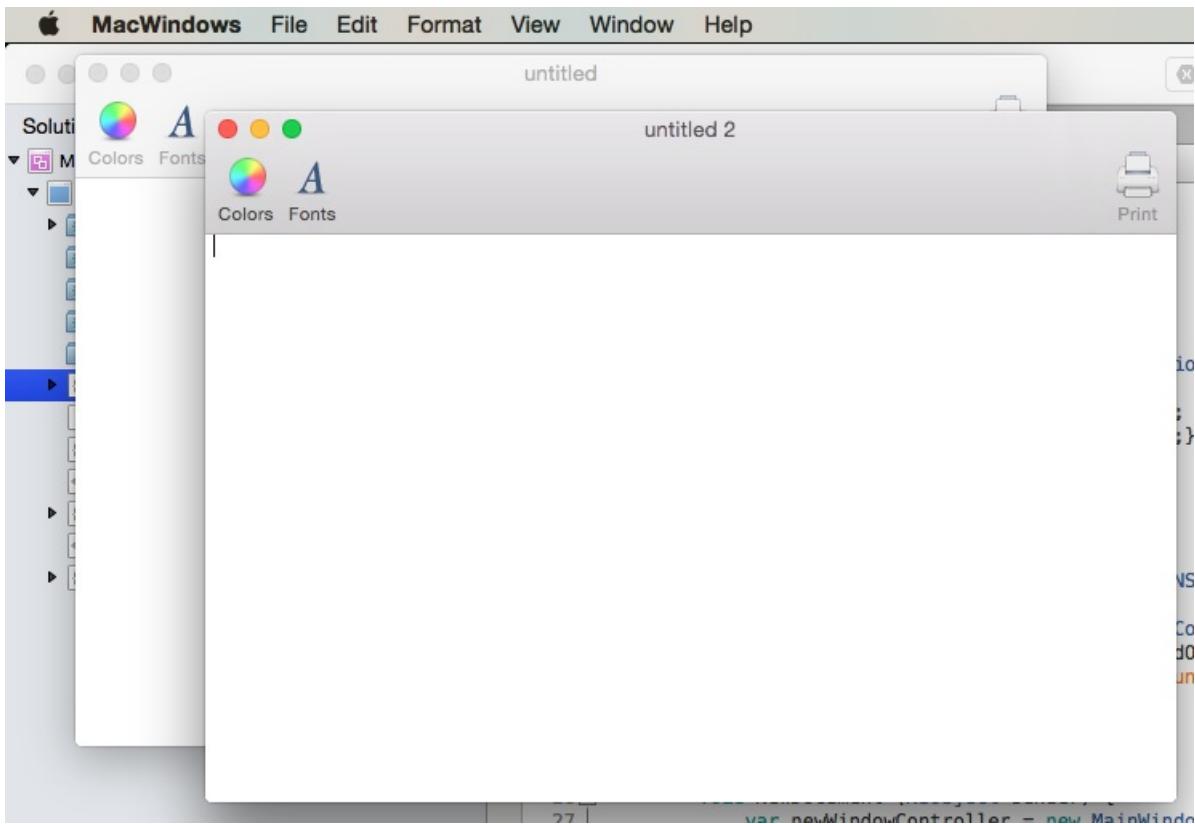
```
[Export ("newDocument:")]
void NewDocument (NSObject sender) {
    // Get new window
    var storyboard = NSSStoryboard.FromName ("Main", null);
    var controller = storyboard.InstantiateControllerWithIdentifier ("MainWindow") as NSWindowController;

    // Display
    controller.ShowWindow(this);

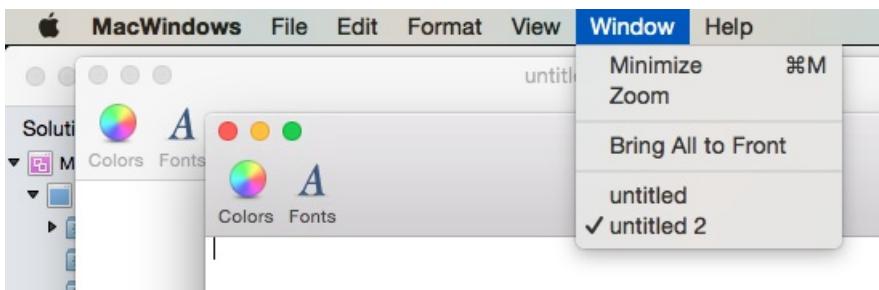
    // Set the title
    controller.Window.Title = (++UntitledWindowCount == 1) ? "untitled" : string.Format ("untitled {0}",
    UntitledWindowCount);
}
```

This code creates a new version of our Window Controller, loads the new Window, makes it the Main and Key Window, and sets its title. Now if we run our application, and select **New** from the **File** menu a new editor

window will be opened and displayed:



If we open the **Windows** menu, you can see the application is automatically tracking and handling our open windows:



For more information on working with Menus in a Xamarin.Mac application, please see our [Working with Menus](#) documentation.

Getting the currently active window

In a Xamarin.Mac application that can open multiple windows (documents), there are times when you will need to get the current, topmost window (the key window). The following code will return the key window:

```
var window = NSApplication.SharedApplication.KeyWindow;
```

It can be called in any class or method that needs to access the current, key window. If no window is currently open, it will return `null`.

Accessing all app windows

There might be times where you need to access all of the windows that your Xamarin.Mac app currently has open. For example, to see if a file that the user wants to open is already open in an existing window.

The `NSApplication.SharedApplication` maintains a `Windows` property that contains an array of all open windows in your app. You can iterate over this array to access all of the app's current windows. For example:

```

// Is the file already open?
for(int n=0; n<NSApplication.SharedApplication.Windows.Length; ++n) {
    var content = NSApplication.SharedApplication.Windows[n].ContentViewController as ViewController;
    if (content != null && path == content.FilePath) {
        // Bring window to front
        NSApplication.SharedApplication.Windows[n].MakeKeyAndOrderFront(this);
        return true;
    }
}

```

In the example code we are casting each returned window to the custom `ViewController` class in our app and testing the value of a custom `Path` property against the path of a file the user wants to open. If the file is already open, we are bringing that window to the front.

Adjusting the window size in code

There are times when the application needs to resize a window in code. To resize and reposition a window, you adjust its `Frame` property. When adjusting a window's size, you usually need to also adjust its origin, to keep the window in the same location because macOS's coordinate system.

Unlike iOS where the upper left hand corner represents (0,0), macOS uses a mathematic coordinate system where the lower left hand corner of the screen represents (0,0). In iOS the coordinates increase as you move downward towards the right. In macOS, the coordinates increase in value upwards to the right.

The following example code resizes a window:

```

nfloat y = 0;

// Calculate new origin
y = Frame.Y - (768 - Frame.Height);

// Resize and position window
CGRect frame = new CGRect (Frame.X, y, 1024, 768);
SetFrame (frame, true);

```

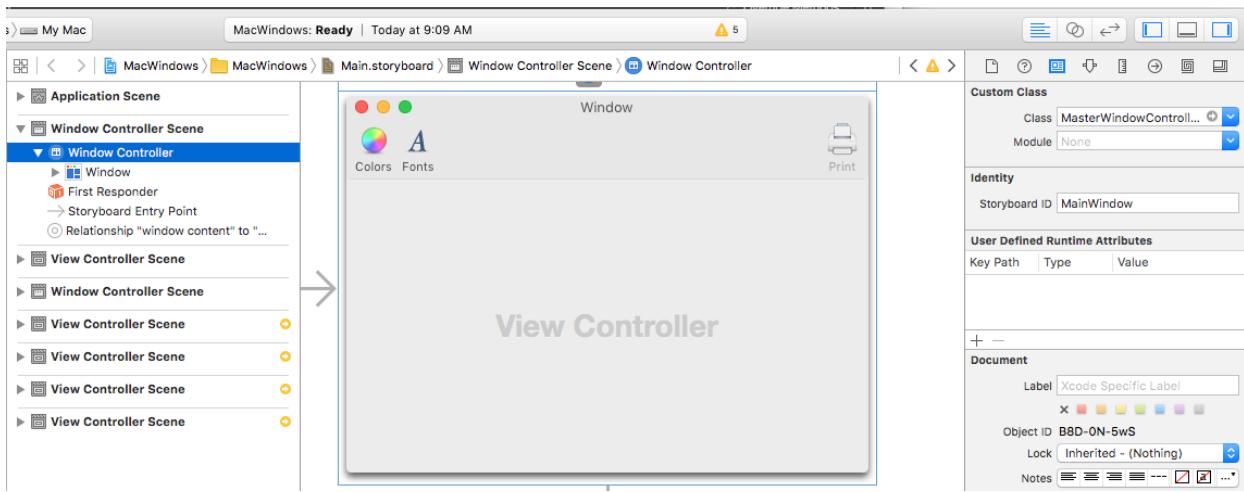
IMPORTANT

When you adjust a windows size and location in code, you need to make sure you respect the minimum and maximum sizes that you have set in Interface Builder. This will not be automatically honored and you will be able to make the window bigger or smaller than these limits.

Monitoring window size changes

There might be times where you need to monitor changes in a Window's size inside of your Xamarin.Mac app. For example, to redraw content to fit the new size.

To monitor size changes, first ensure that you have assigned a custom class for the Window Controller in Xcode's Interface Builder. For example, `MasterWindowController` in the following:



Next, edit the custom Window Controller class and monitor the `DidResize` event on the Controller's Window to be notified of live size changes. For example:

```
public override void WindowDidLoad ()
{
    base.WindowDidLoad ();

    Window.DidResize += (sender, e) => {
        // Do something as the window is being live resized
    };
}
```

Optionally, you can use the `DidEndLiveResize` event to only be notified after the user has finished changing the Window's size. For Example:

```
public override void WindowDidLoad ()
{
    base.WindowDidLoad ();

    Window.DidEndLiveResize += (sender, e) => {
        // Do something after the user's finished resizing
        // the window
    };
}
```

Setting a window's title and represented file

When working with windows that represent documents, `NSWindow` has a `DocumentEdited` property that if set to `true` displays a small dot in the Close Button to give the user an indication that the file has been modified and should be saved before closing.

Let's edit our `ViewController.cs` file and make the following changes:

```

public bool DocumentEdited {
    get { return View.Window.DocumentEdited; }
    set { View.Window.DocumentEdited = value; }
}

...

public override void ViewWillAppear ()
{
    base.ViewWillAppear ();

    // Set Window Title
    this.View.Window.Title = "untitled";

    View.Window.WillClose += (sender, e) => {
        // is the window dirty?
        if (DocumentEdited) {
            var alert = new NSAlert () {
                AlertStyle = NSAlertStyle.Critical,
                InformativeText = "We need to give the user the ability to save the document here...",
                MessageText = "Save Document",
            };
            alert.RunModal ();
        }
    };
}

public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

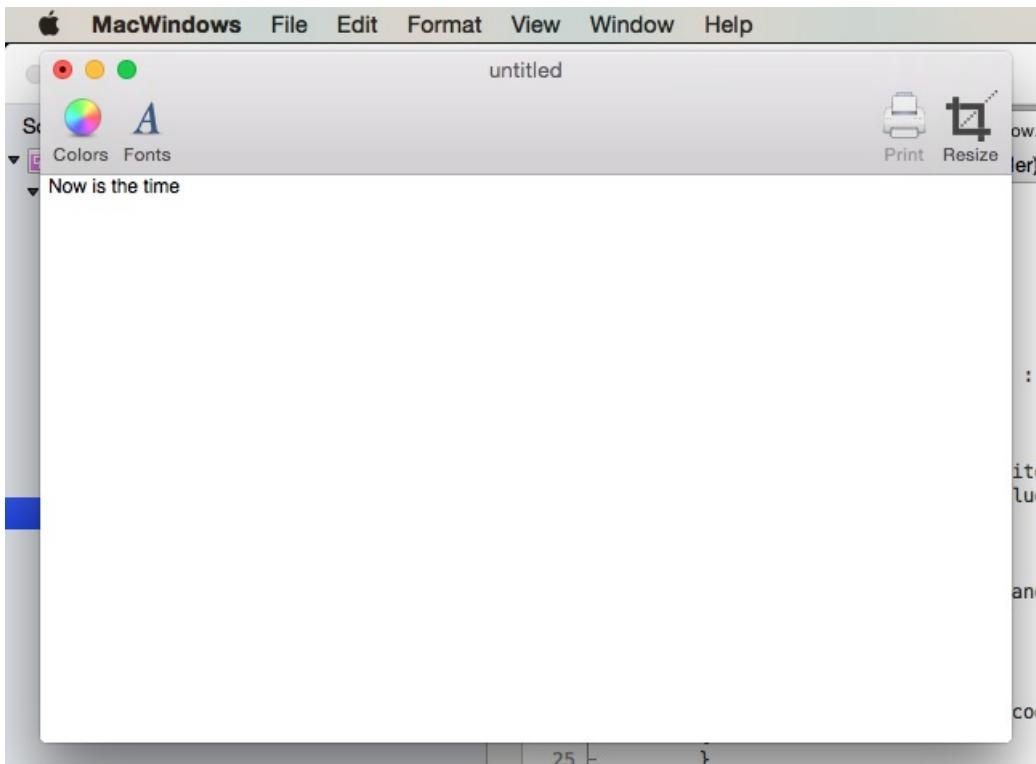
    // Show when the document is edited
    DocumentEditor.TextDidChange += (sender, e) => {
        // Mark the document as dirty
        DocumentEdited = true;
    };

    // Overriding this delegate is required to monitor the TextDidChange event
    DocumentEditor.ShouldChangeTextInRanges += (NSTextView view, NSValue[] values, string[] replacements) =>
{
    return true;
};

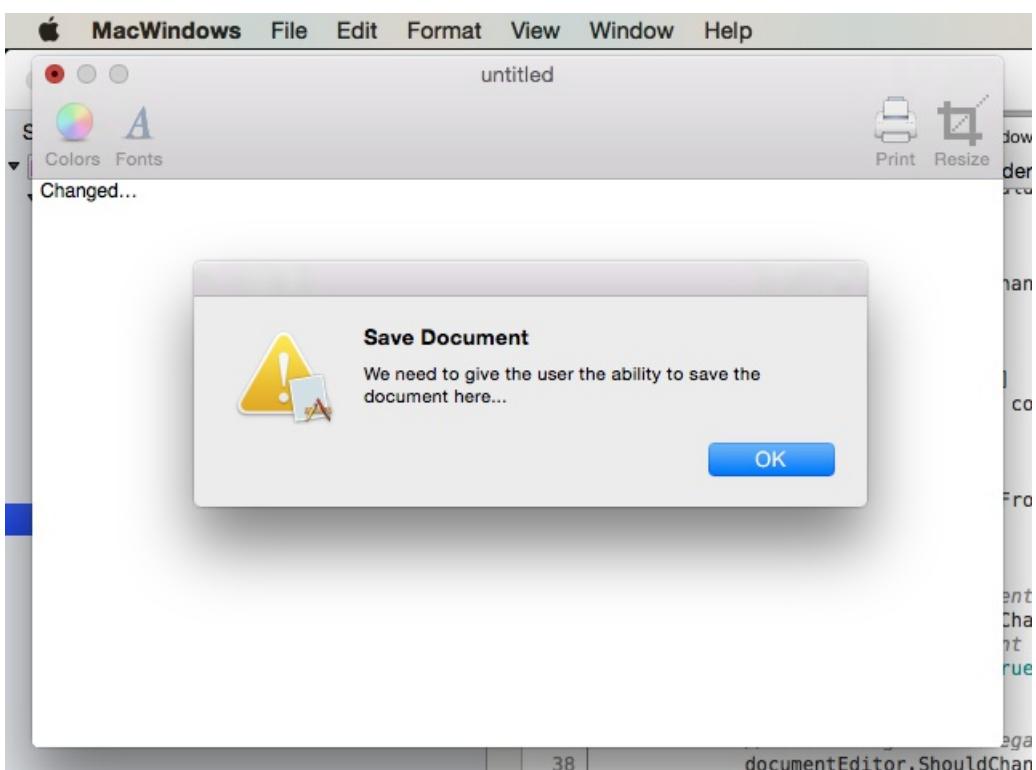
}

```

We are also monitoring the `WillClose` event on the window and checking the state of the `DocumentEdited` property. If it is `true` we need to give the user the ability to save the changes to the file. If we run our app and enter some text, the dot will be displayed:



If you try to close the window, you get an alert:



If you are loading a document from a file, set the title of the window to the file's name using the `window.SetTitleWithRepresentedFilename (Path.GetFileName(path));` method (given that `path` is a string representing the file being opened). Additionally, you can set the URL of the file using the `window.RepresentedUrl = url;` method.

If the URL is pointing to a file type known by the OS, its icon will be displayed in the title bar. If the user right clicks on the icon, the path to the file will be shown.

Edit the `AppDelegate.cs` file and add the following method:

```

[Export ("openDocument")]
void OpenDialog (NSObject sender)
{
    var dlg = NSOpenPanel.OpenPanel;
    dlg.CanChooseFiles = true;
    dlg.CanChooseDirectories = false;

    if (dlg.RunModal () == 1) {
        // Nab the first file
        var url = dlgUrls [0];

        if (url != null) {
            var path = url.Path;

            // Get new window
            var storyboard = NSSStoryboard.FromName ("Main", null);
            var controller = storyboard.InstantiateControllerWithIdentifier ("MainWindow") as
NSWindowController;

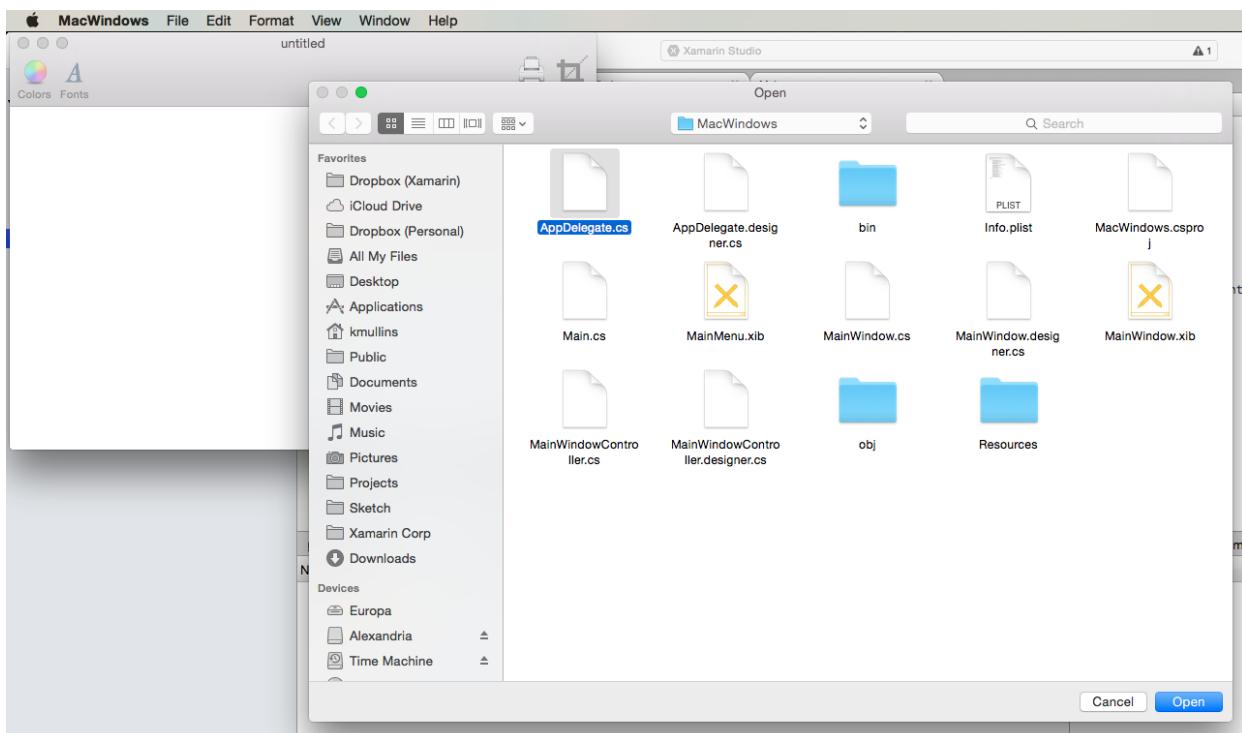
            // Display
            controller.ShowWindow(this);

            // Load the text into the window
            var viewController = controller.Window.ContentViewController as ViewController;
            viewController.Text = File.ReadAllText(path);
                viewController.View.WindowSetTitleWithRepresentedFilename (Path.GetFileName(path));
            viewController.View.Window.RepresentedUrl = url;

        }
    }
}

```

Now if we run our app, select **Open...** from the **File** menu, select a text file from the **Open** Dialog box and open it:



The file will be displayed and the title will be set with the icon of the file:

```

using System;
using Foundation;
using AppKit;
using System.IO;

namespace MacWindows
{
    public partial class AppDelegate : NSApplicationDelegate
    {
        MainWindowController mainWindowController;
        public int UntitledWindowCount { get; set;} =1;

        public AppDelegate ()
        {

        }

        public override void DidFinishLaunching (NSNotification notification)
        {
            mainWindowController = new MainWindowController ();
            mainWindowController.Window.MakeKeyAndOrderFront (this);
            mainWindowController.Window.Title = "untitled";
        }

        #region Menu Handlers
    }
}

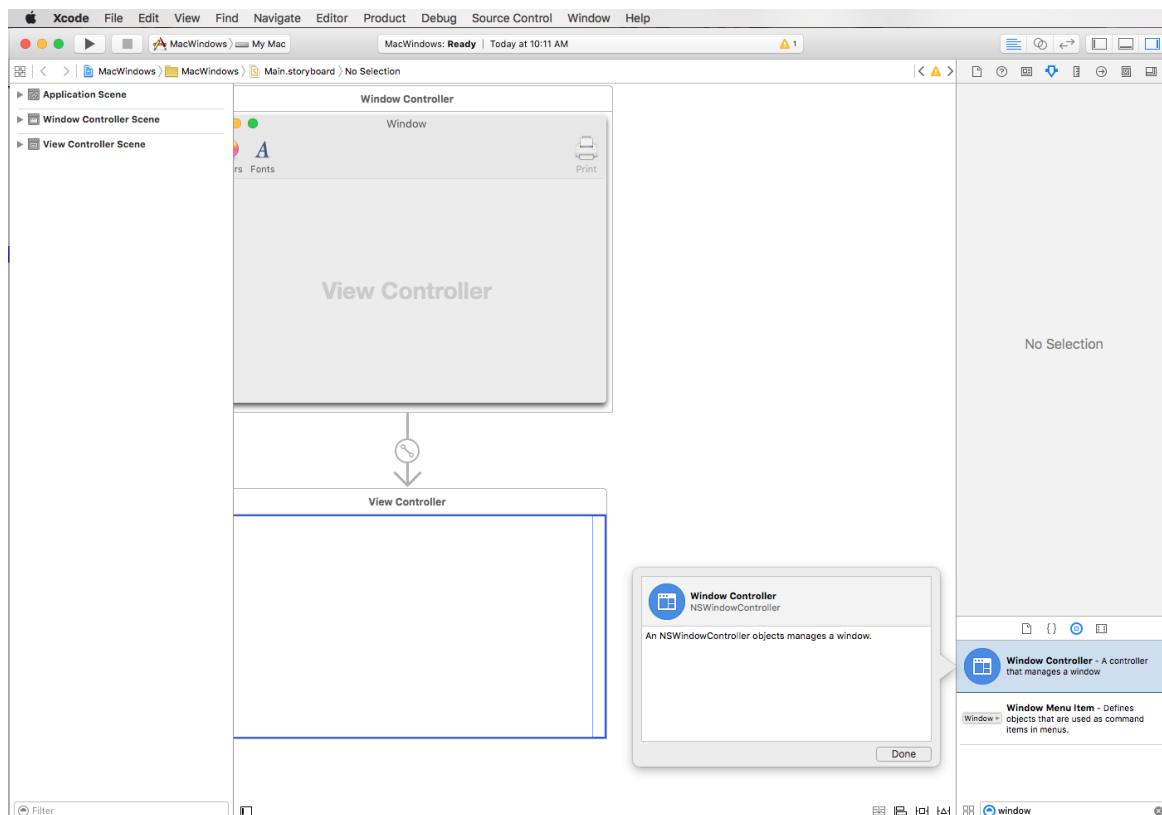
```

Adding a new window to a project

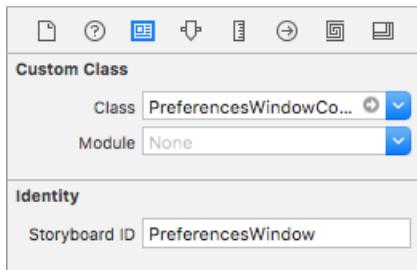
Aside from the main document window, a Xamarin.Mac application might need to display other types of windows to the user, such as Preferences or Inspector Panels.

To add a new window, do the following:

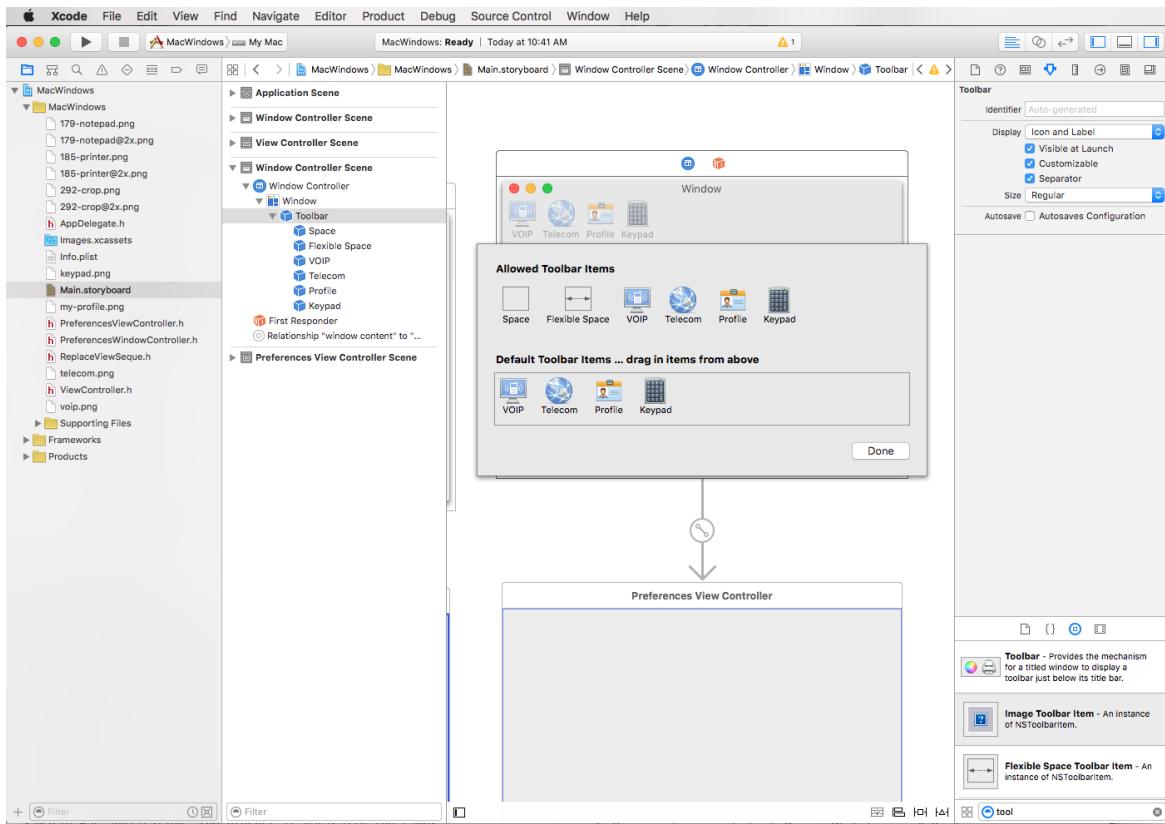
1. In the **Solution Explorer**, double-click the `Main.storyboard` file to open it for editing in Xcode's Interface Builder.
2. Drag a new **Window Controller** from the **Library** and drop it on the **Design Surface**:



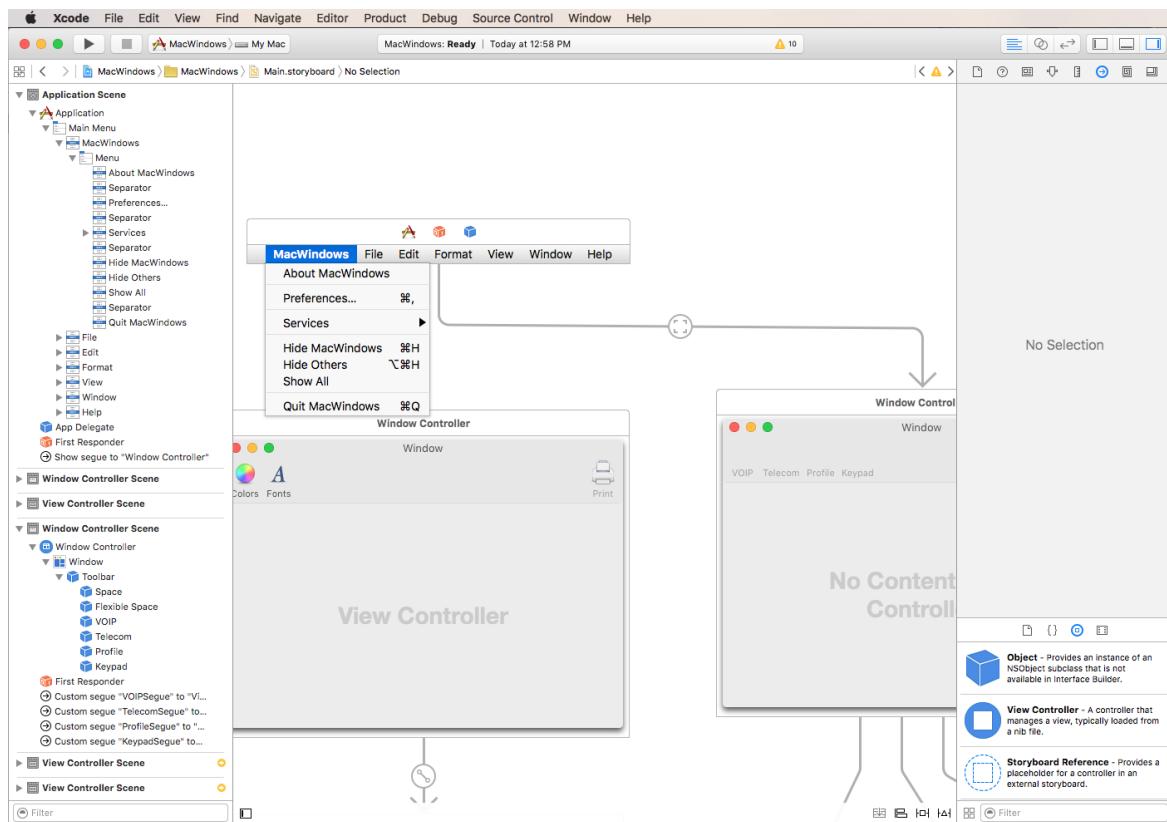
3. In the Identity Inspector, enter `PreferencesWindow` for the Storyboard ID:



4. Design your interface:



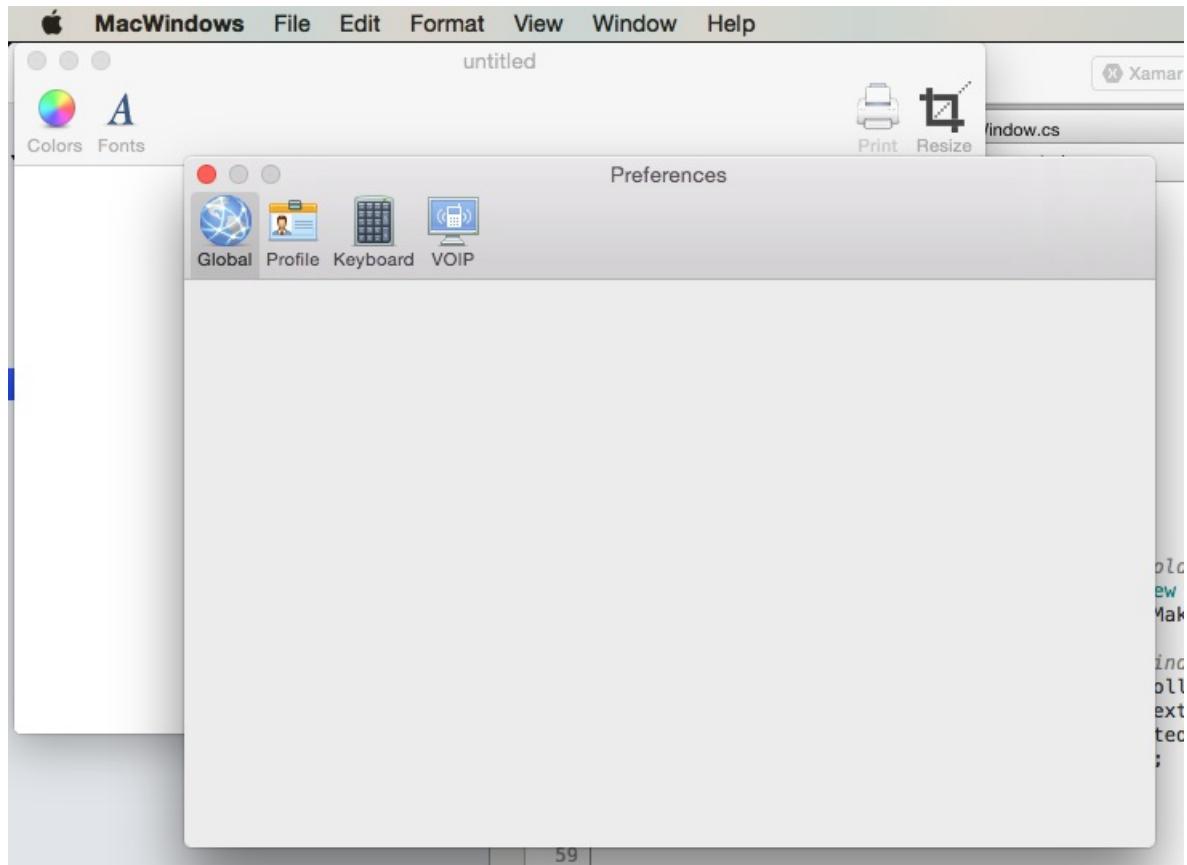
5. Open the App Menu (MacWindows), select Preferences..., Control-Click and drag to the new window:



6. Select **Show** from the popup menu.

7. Save your changes and return to Visual Studio for Mac to sync with Xcode.

If we run the code and select the **Preferences...** from the **Application Menu**, the window will be displayed:



Working with panels

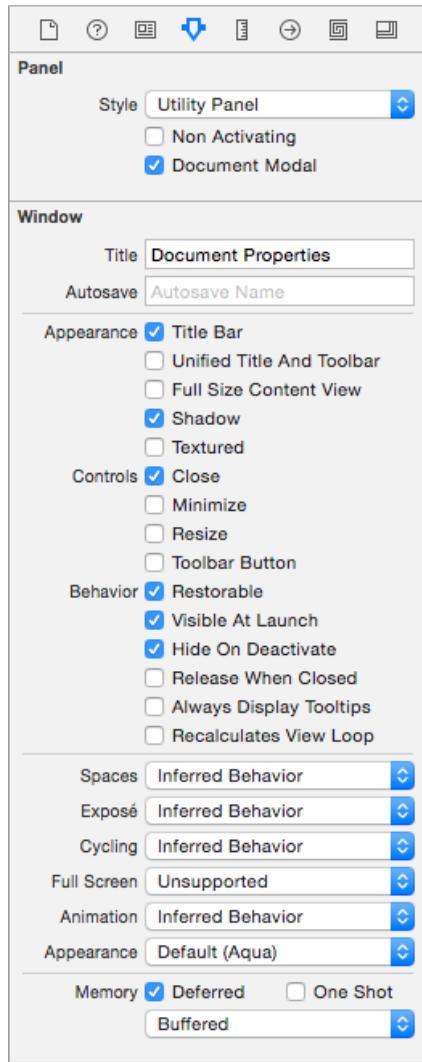
As stated at the start of this article, a panel floats above other windows and provides tools or controls that users

can work with while documents are open.

Just like any other type of window that you create and work with in your Xamarin.Mac application, the process is basically the same:

1. Add a new window definition to the project.
2. Double-click the `.xib` file to open the window design for editing in Xcode's Interface Builder.
3. Set any required window properties in the **Attribute Inspector** and the **Size Inspector**.
4. Drag in the controls required to build your interface and configure them in the **Attribute Inspector**.
5. Use the **Size Inspector** to handle the resizing for your UI Elements.
6. Expose the window's UI elements to C# code via **Outlets** and **Actions**.
7. Save your changes and switch back to Visual Studio for Mac to sync with Xcode.

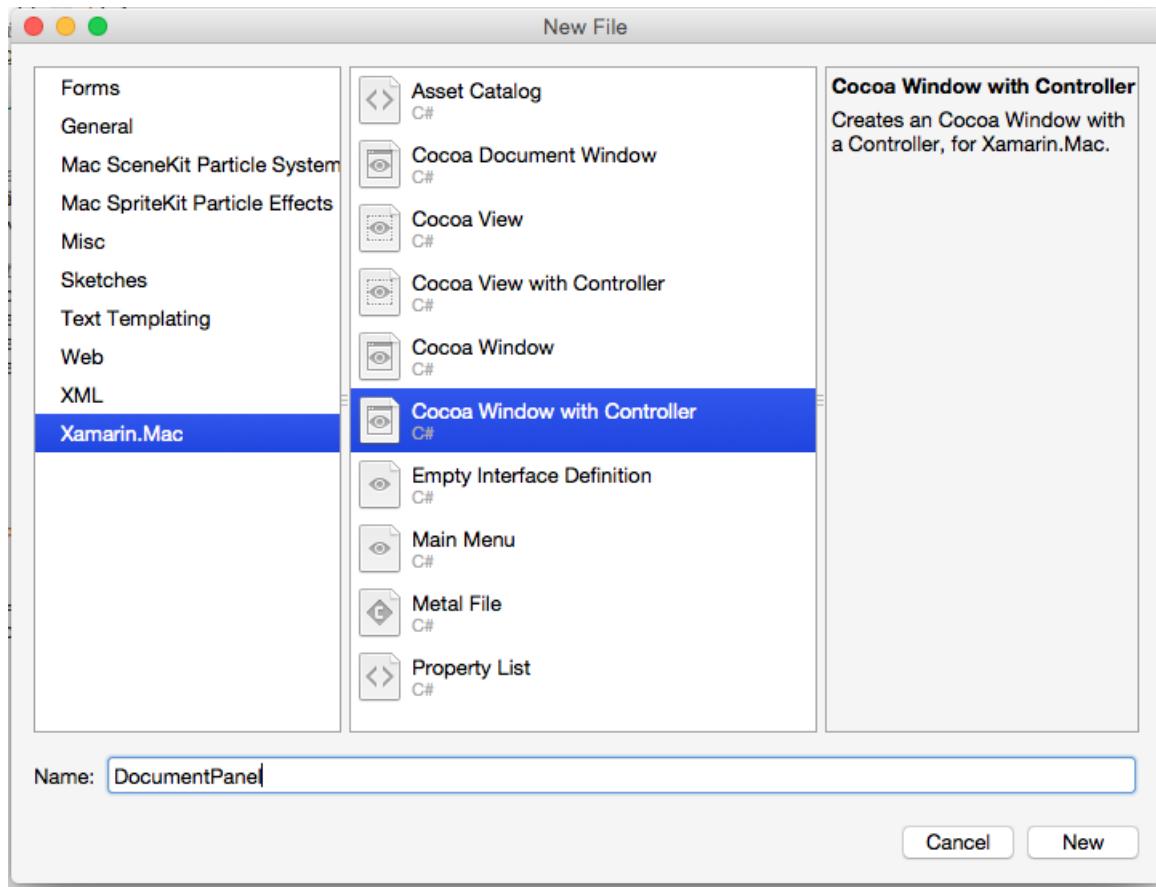
In the **Attribute Inspector**, you have the following options specific to Panels:



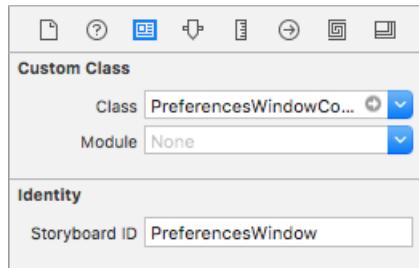
- **Style** - Allow you to adjust the style of the panel from: Regular Panel (looks like a standard window), Utility Panel (has a smaller Title bar), HUD Panel (is translucent and the title bar is part of the background).
- **Non Activating** - Determines in the panel becomes the key window.
- **Document Modal** - If Document Modal, the panel will only float above the application's windows, else it floats above all.

To add a new Panel, do the following:

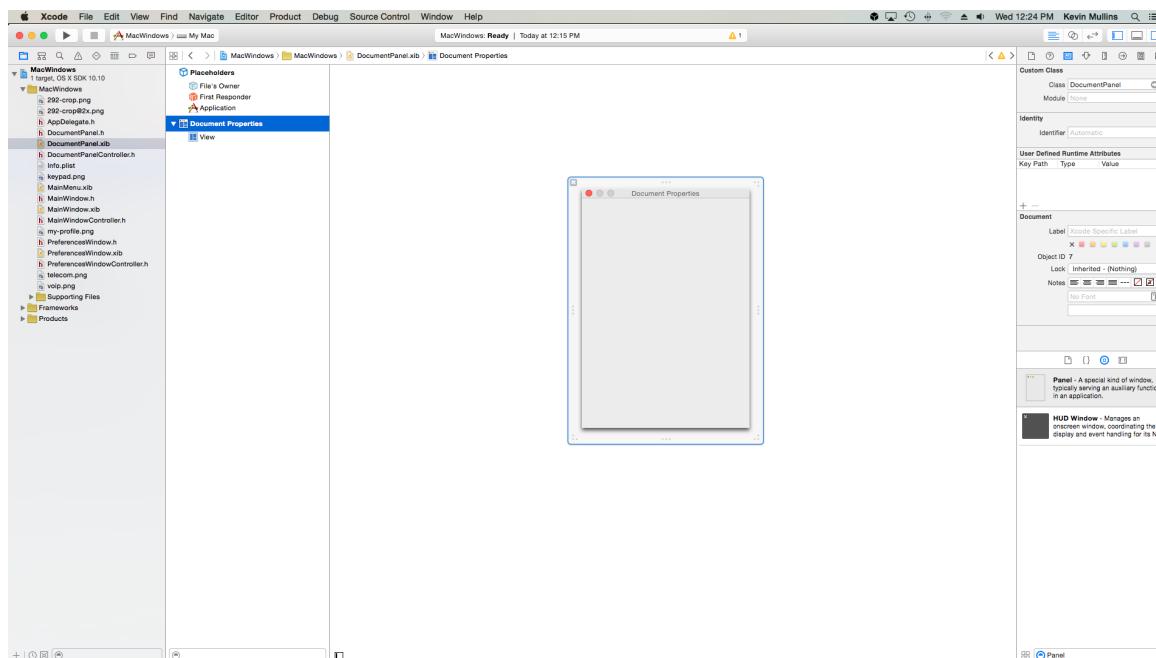
1. In the **Solution Explorer**, right-click on the Project and select **Add > New File....**
2. In the New File dialog box, select **Xamarin.Mac > Cocoa Window with Controller**:



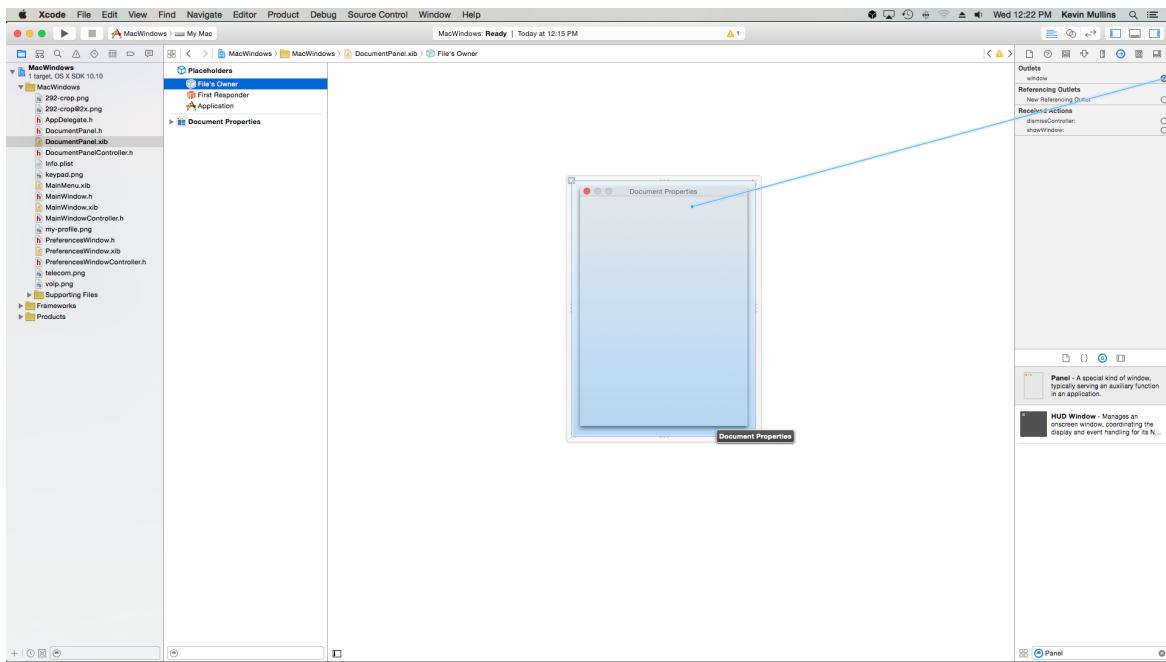
3. Enter `DocumentPanel` for the Name and click the New button.
4. Double-click the `DocumentPanel.xib` file to open it for editing in Interface Builder:



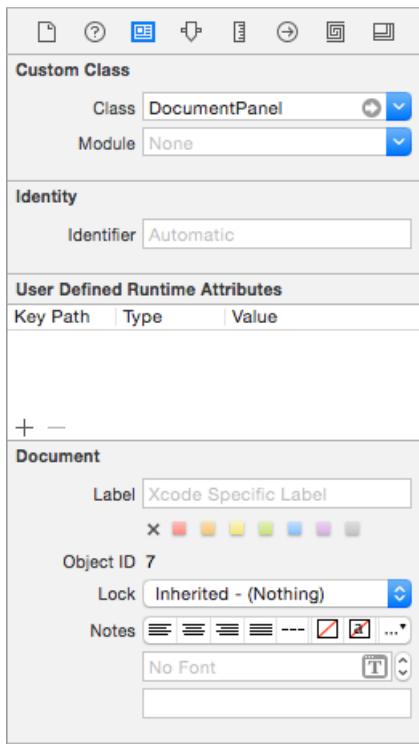
5. Delete the existing Window and drag a Panel from the Library Inspector in the Interface Editor:



6. Hook the panel up to the File's Owner - window - Outlet:



7. Switch to the Identity Inspector and set the Panel's class to `DocumentPanel`:



8. Save your changes and return to Visual Studio for Mac to sync with Xcode.

9. Edit the `DocumentPanel.cs` file and change the class definition to the following:

```
public partial class DocumentPanel : NSPanel
```

10. Save the changes to the file.

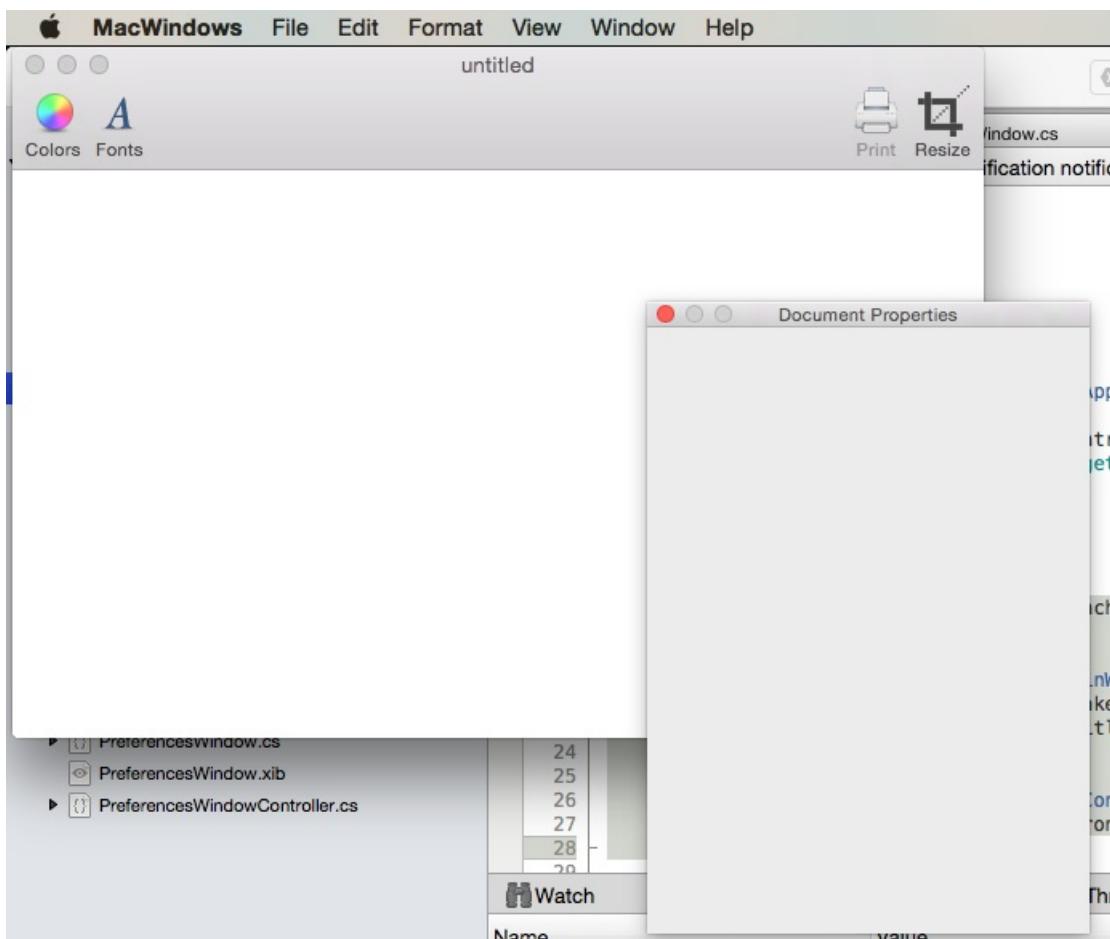
Edit the `AppDelegate.cs` file and make the `DidFinishLaunching` method look like the following:

```

public override void DidFinishLaunching (NSNotification notification)
{
    // Display panel
    var panel = new DocumentPanelController ();
    panel.Window.MakeKeyAndOrderFront (this);
}

```

If we run our application, the panel will be displayed:



IMPORTANT

Panel Windows have been deprecated by Apple and should be replaced with **Inspector Interfaces**. For a full example of creating an **Inspector** in a Xamarin.Mac app, please see our [MacInspector](#) sample app.

Summary

This article has taken a detailed look at working with Windows and Panels in a Xamarin.Mac application. We saw the different types and uses of Windows and Panels, how to create and maintain Windows and Panels in Xcode's Interface Builder and how to work with Windows and Panels in C# code.

Related links

- [MacWindows \(sample\)](#)
- [MacInspector \(sample\)](#)
- [Hello, Mac](#)
- [Working with Menus](#)
- [macOS design themes \(Apple\)](#)

- Windows, Panels, and Screens (Apple)

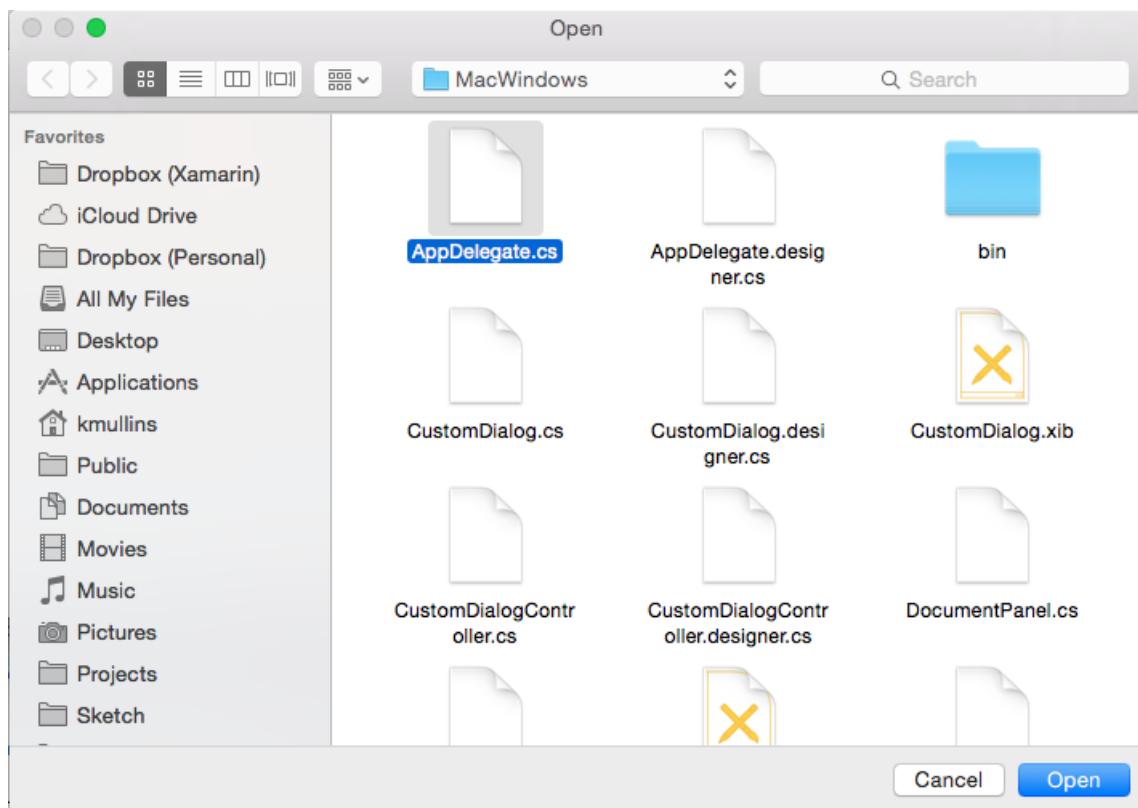
Dialogs in Xamarin.Mac

3/5/2021 • 21 minutes to read • [Edit Online](#)

When working with C# and .NET in a Xamarin.Mac application, you have access to the same Dialogs and Modal Windows that a developer working in *Objective-C* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, you can use Xcode's *Interface Builder* to create and maintain your Modal Windows (or optionally create them directly in C# code).

A dialog appears in response to a user action and typically provides ways users can complete the action. A dialog requires a response from the user before it can be closed.

Windows can be used in a Modeless state (such as a text editor that can have multiple documents open at once) or Modal (such as an Export dialog that must be dismissed before the application can continue).



In this article, we'll cover the basics of working with Dialogs and Modal Windows in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode](#) and [Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` commands used to wire-up your C# classes to Objective-C objects and UI Elements.

Introduction to Dialogs

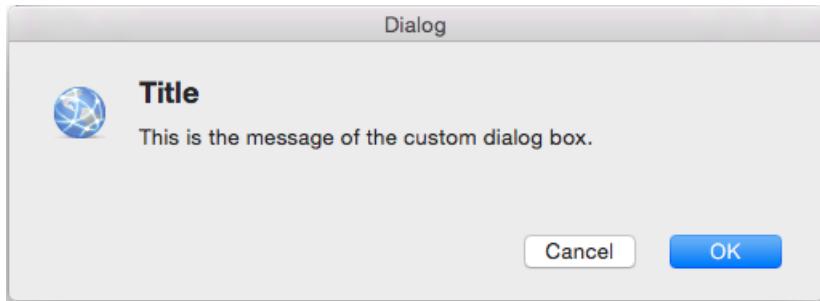
A dialog appears in response to a user action (such as saving a file) and provides a way for users to complete that action. A dialog requires a response from the user before it can be closed.

According to Apple, there are three ways to present a Dialog:

- **Document Modal** - A Document Modal dialog prevents the user from doing anything else within a given document until it is dismissed.
- **App Modal** - An App Modal dialog prevents the user from interacting with the application until it is dismissed.
- **Modeless** A Modeless Dialog enables users to change settings in the dialog while still interacting with the document window.

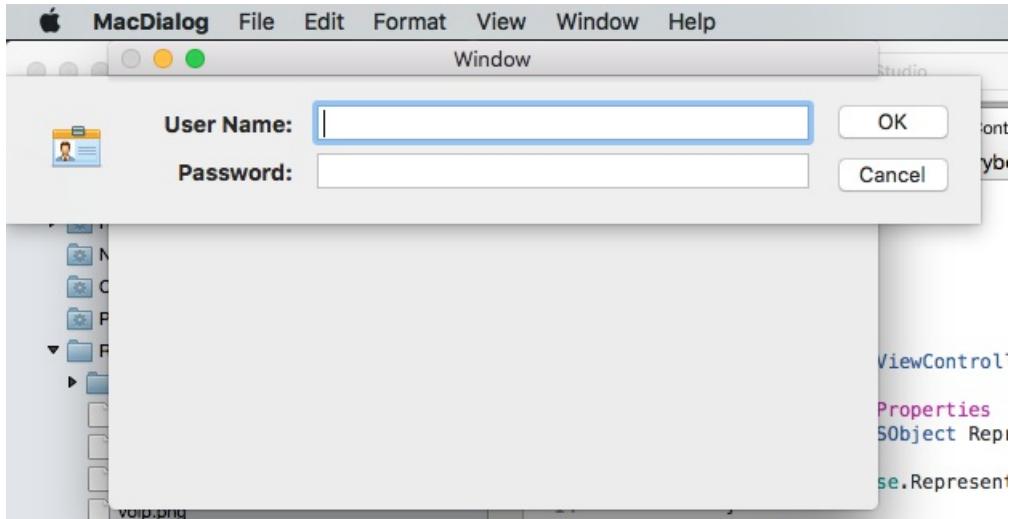
Modal Window

Any standard `NSWindow` can be used as a customized dialog by displaying it modally:



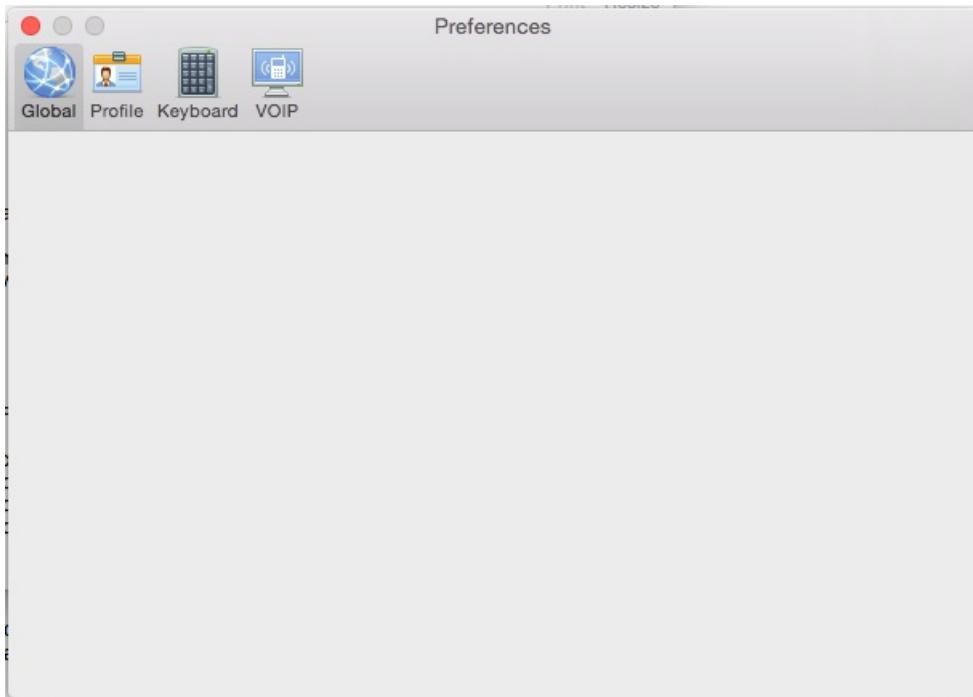
Document Modal Dialog Sheets

A *Sheet* is a modal dialog that is attached to a given document window, preventing users from interacting with the window until they dismiss the dialog. A Sheet is attached to the window from which it emerges and only one sheet can be open for a window at any one time.



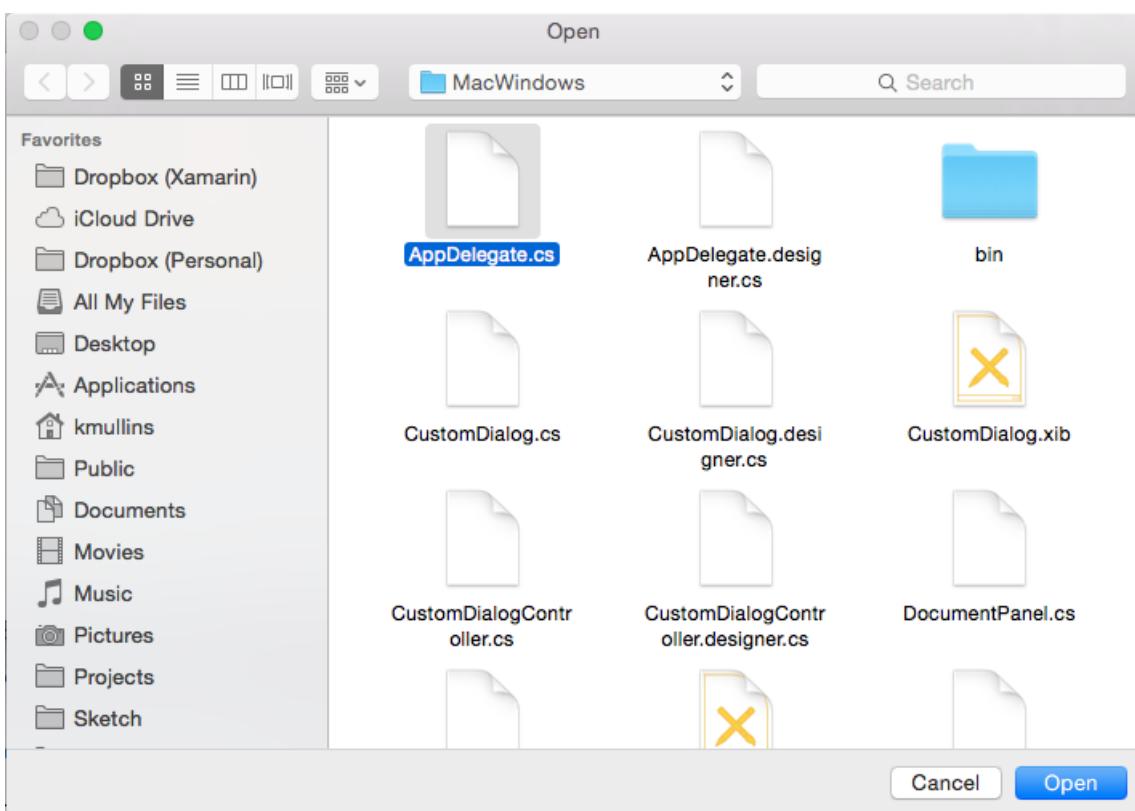
Preferences Windows

A Preferences Window is a modeless dialog that contains the application's settings that the user changes infrequently. Preferences Windows often include a Toolbar that allows the user to switch between different groups of settings:



Open Dialog

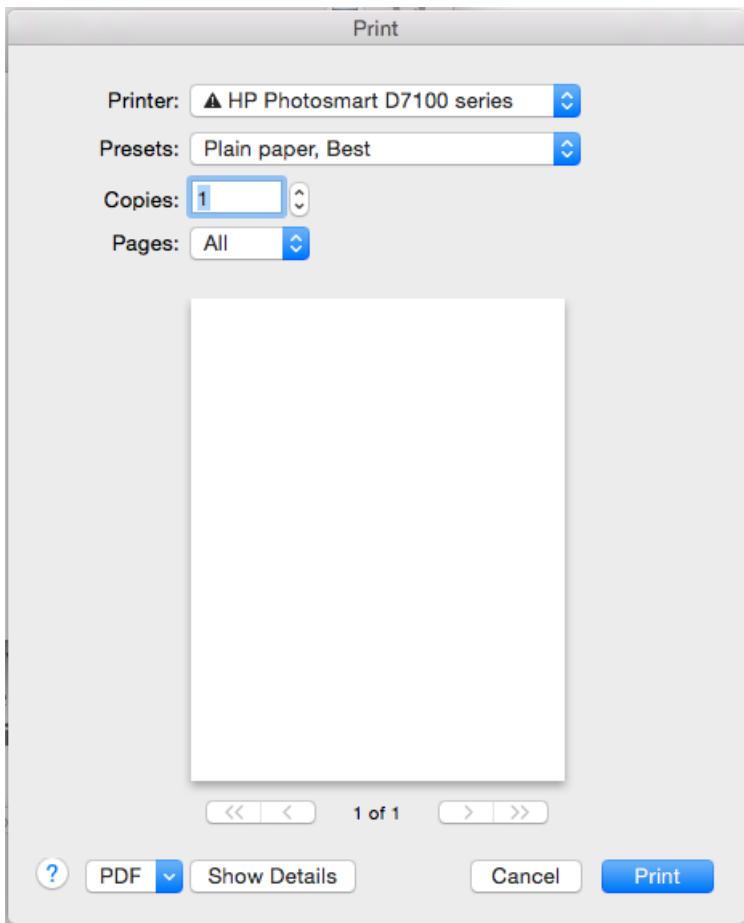
The Open Dialog gives users a consistent way to find and open an item in an application:



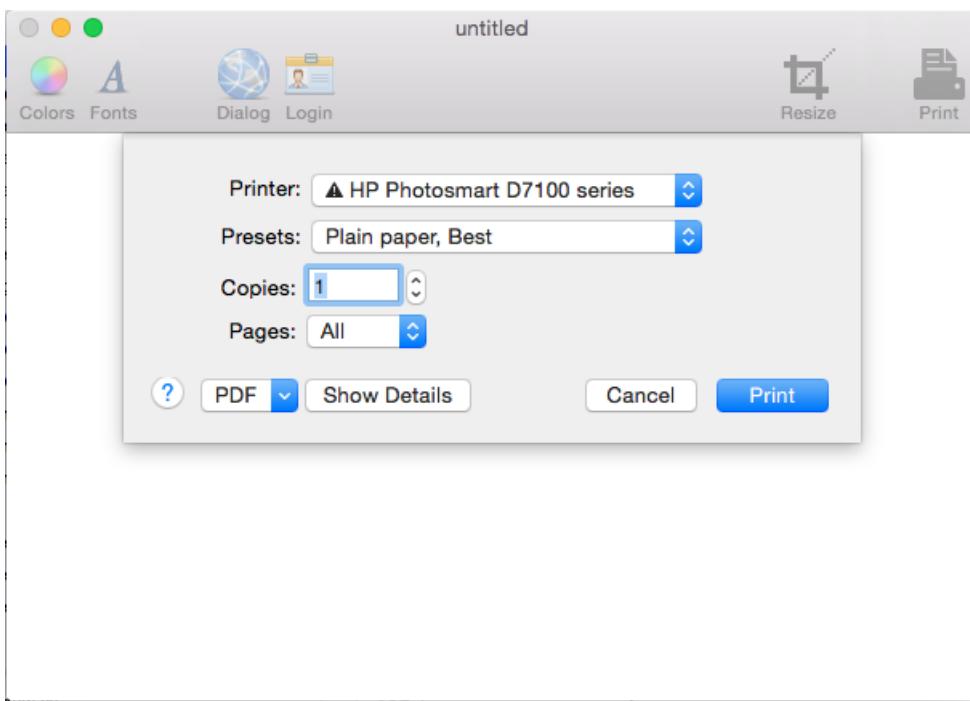
Print and Page Setup Dialogs

macOS provides standard Print and Page Setup Dialogs that your application can display so that users can have a consistent printing experience in every application they use.

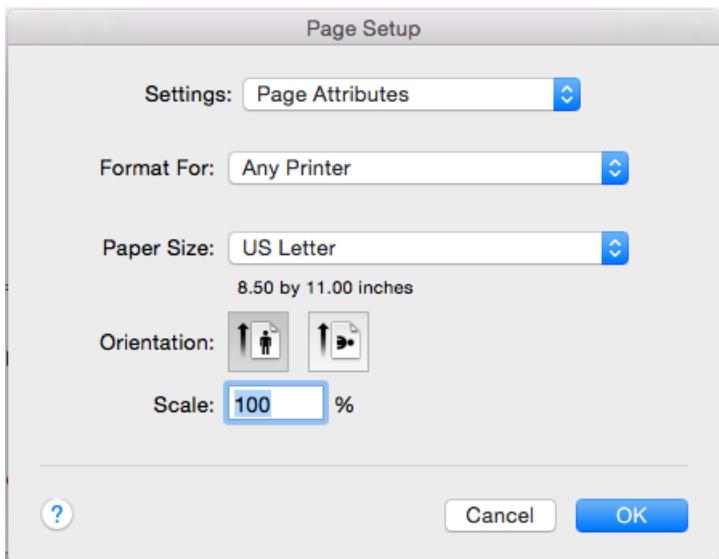
The Print Dialog can be displayed as both a free floating dialog box:



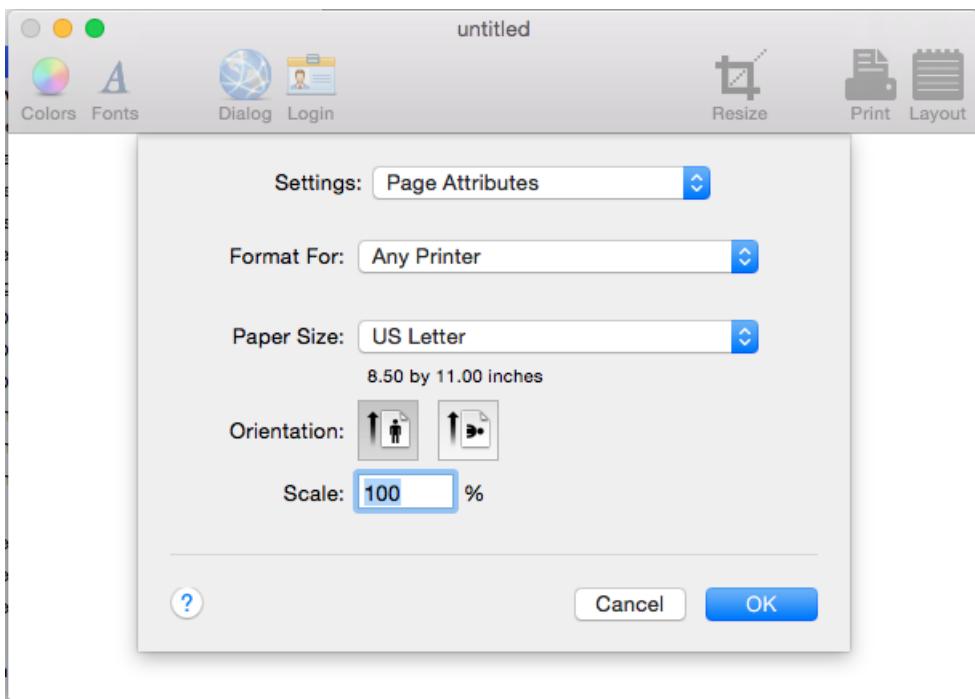
Or it can be displayed as a Sheet:



The Page Setup Dialog can be displayed as both a free floating dialog box:

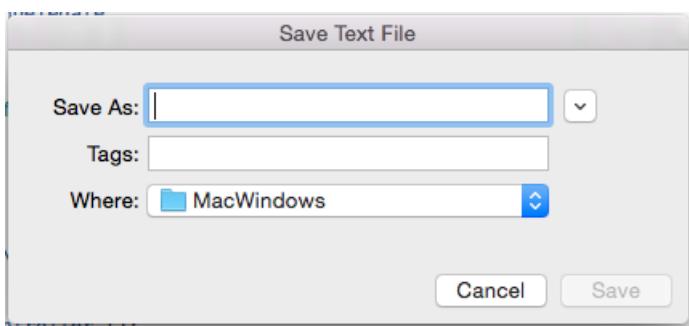


Or it can be displayed as a Sheet:

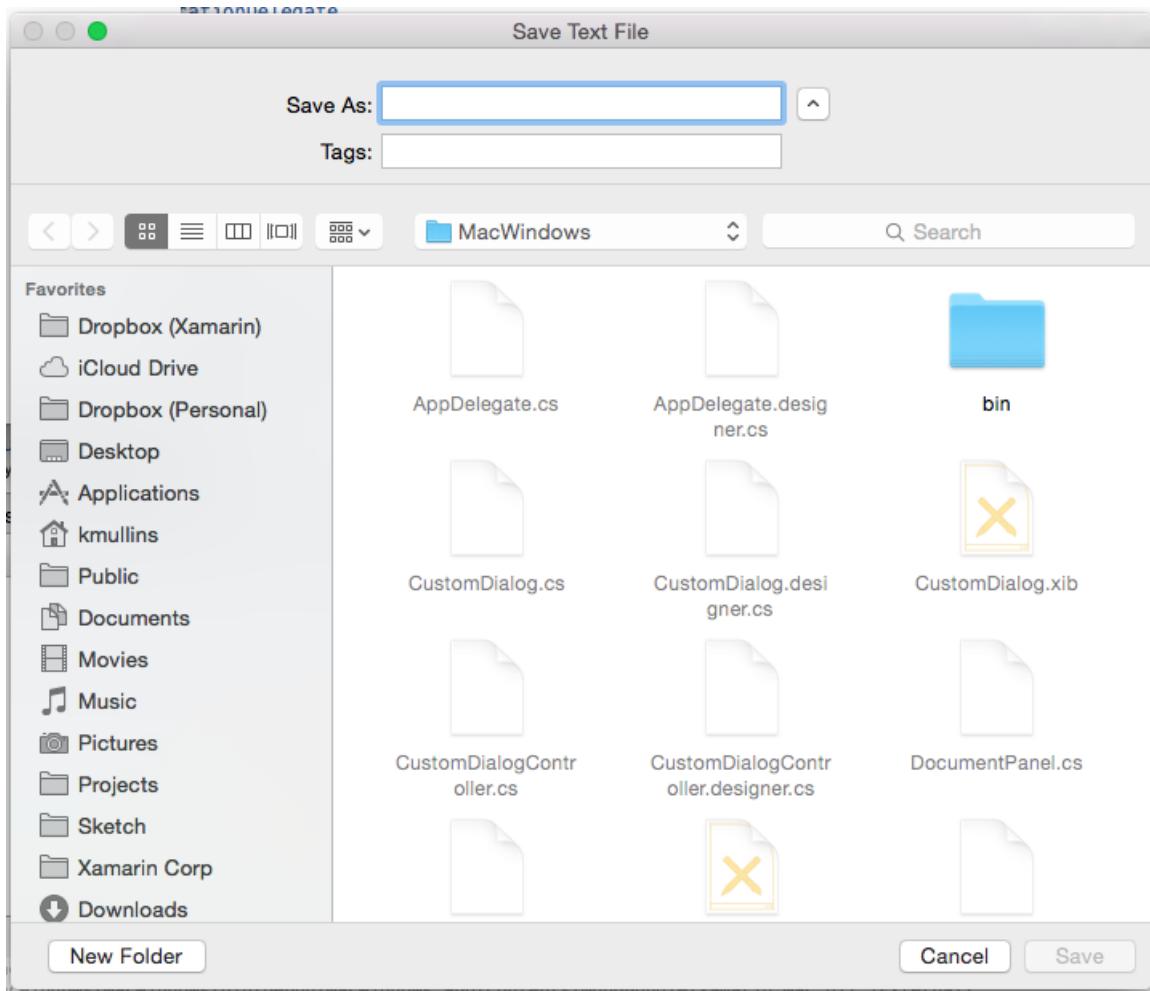


Save Dialogs

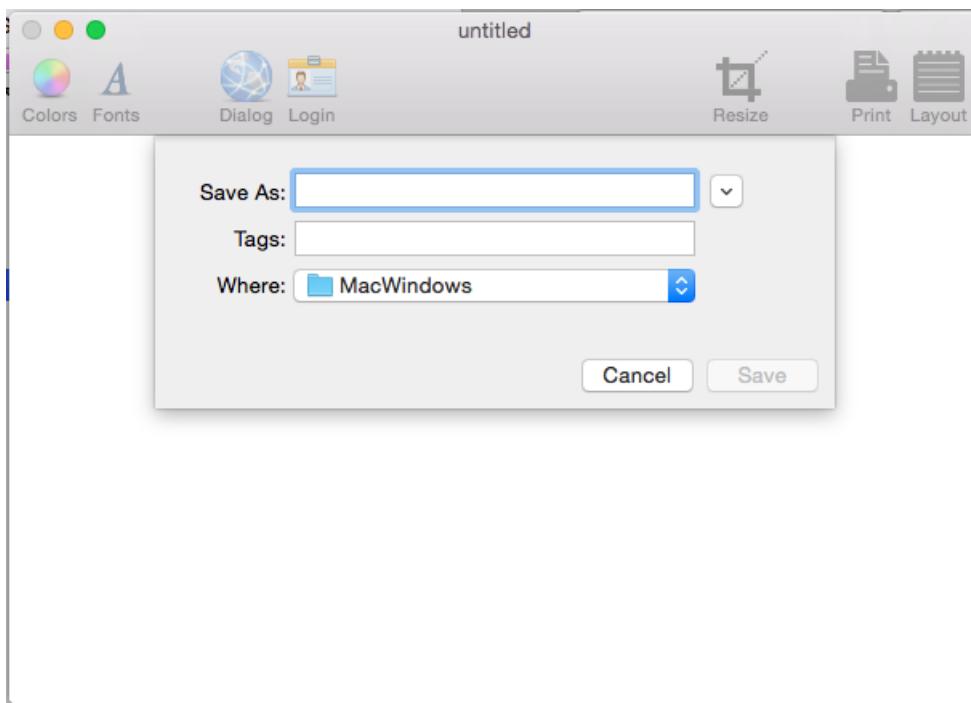
The Save Dialog gives users a consistent way to save an item in an application. The Save Dialog has two states:
Minimal (also known as Collapsed):



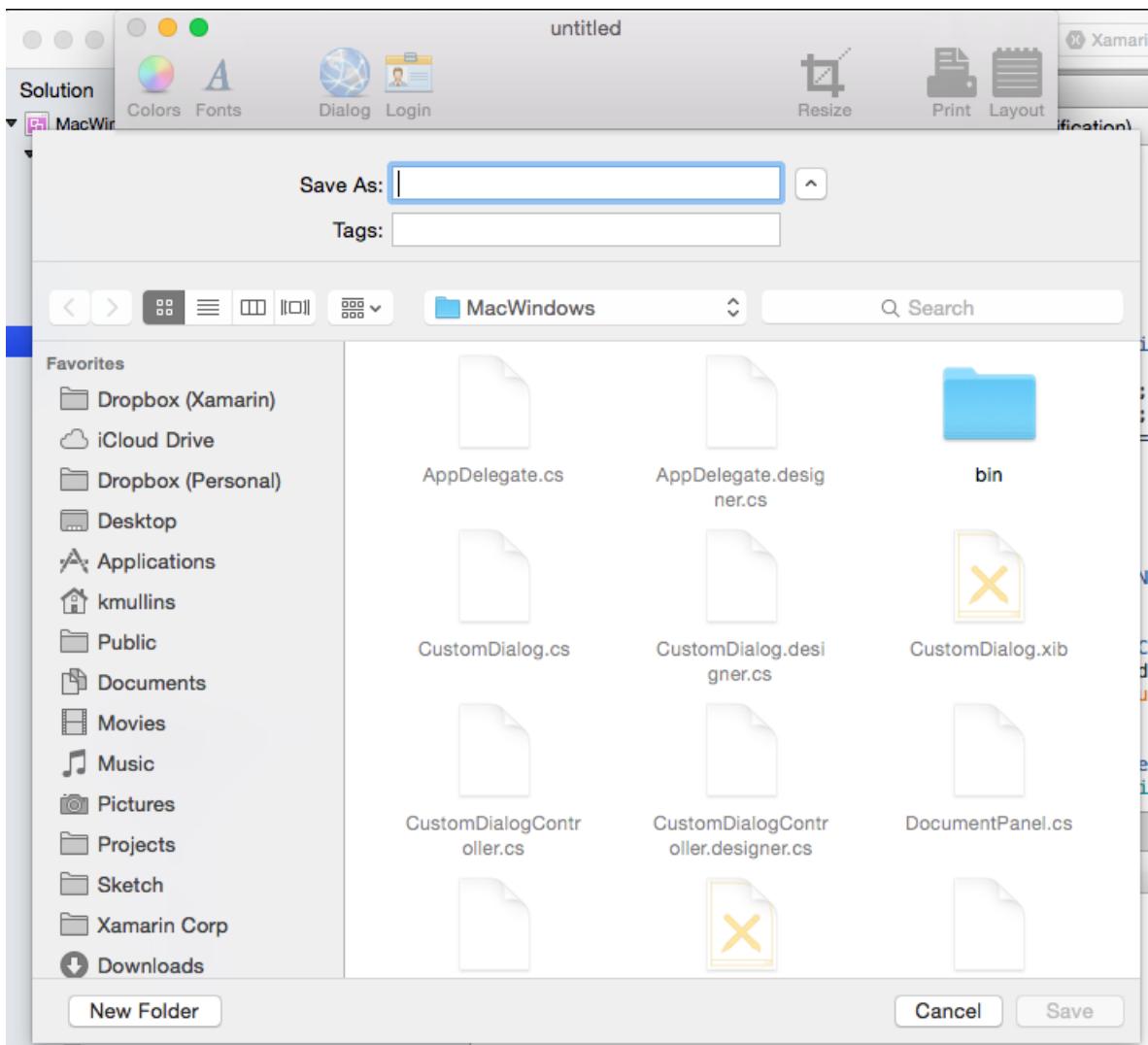
And the **Expanded** state:



The **Minimal** Save Dialog can also be displayed as a Sheet:



As can the **Expanded** Save Dialog:



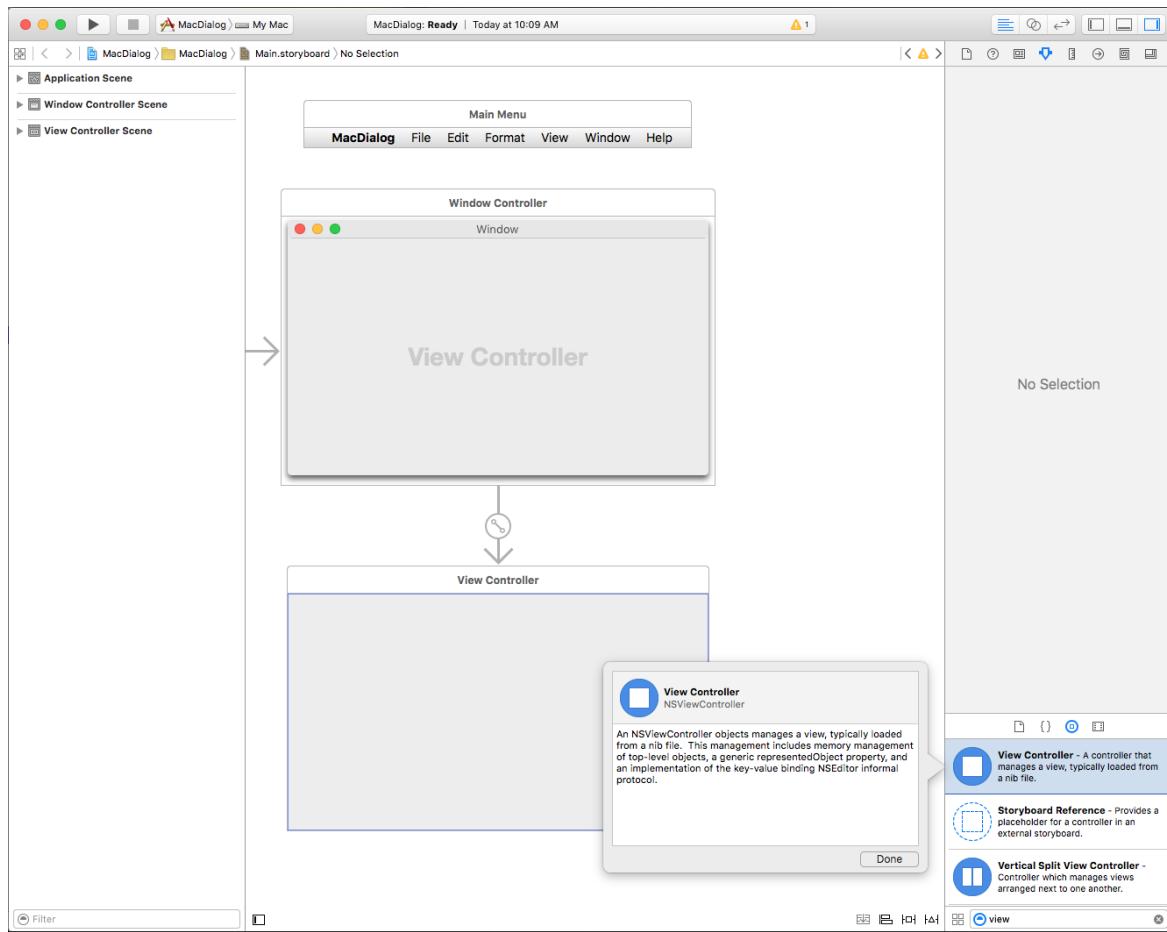
For more information, see the [Dialogs](#) section of Apple's OS X Human Interface Guidelines

Adding a Modal Window to a Project

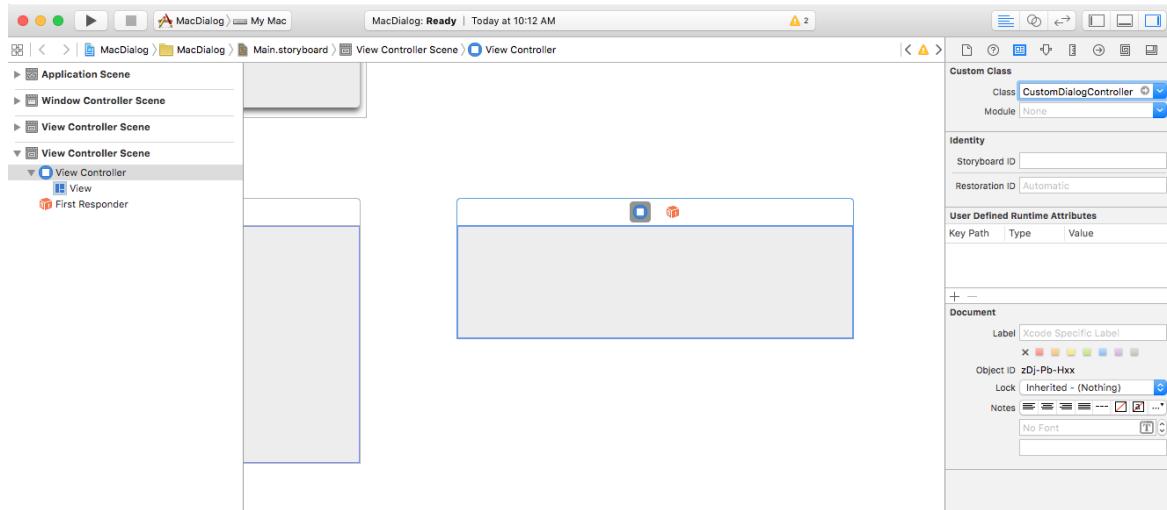
Aside from the main document window, a Xamarin.Mac application might need to display other types of windows to the user, such as Preferences or Inspector Panels.

To add a new window, do the following:

1. In the **Solution Explorer**, open the `Main.storyboard` file for editing in Xcode's Interface Builder.
2. Drag a new **View Controller** into the Design Surface:

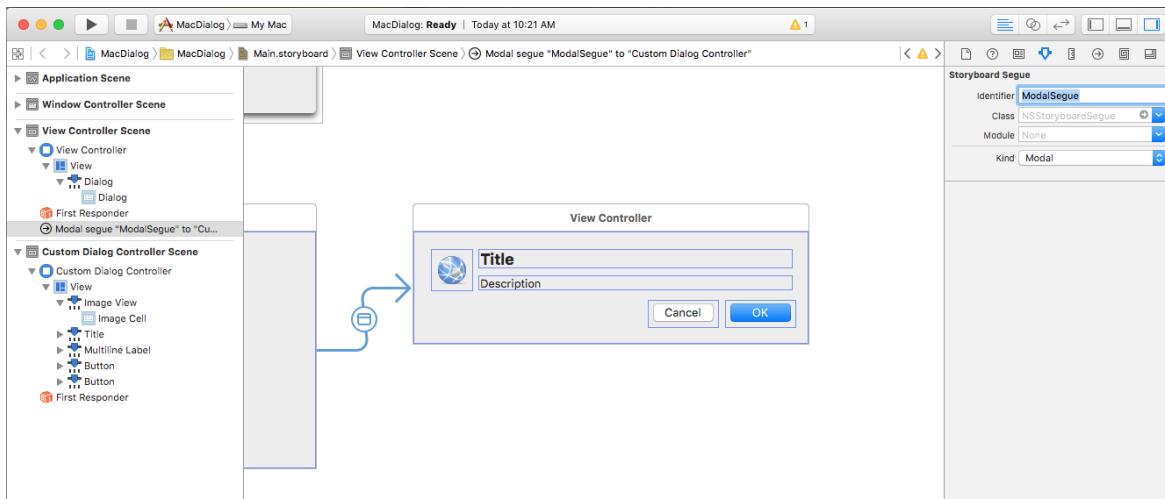


3. In the Identity Inspector, enter `CustomDialogController` for the Class Name:



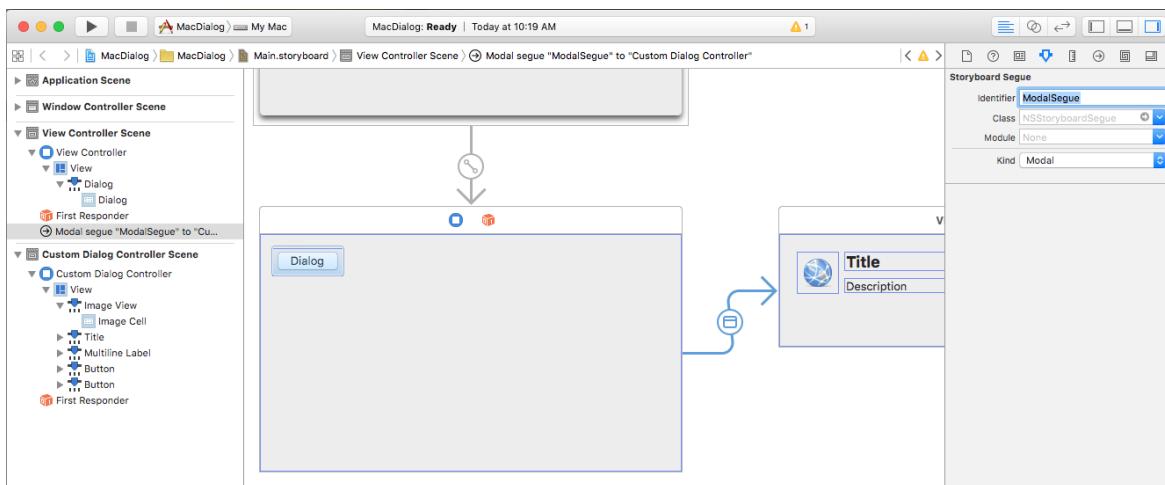
4. Switch back to Visual Studio for Mac, allow it to sync with Xcode and create the `CustomDialogController.h` file.

5. Return to Xcode and design your interface:

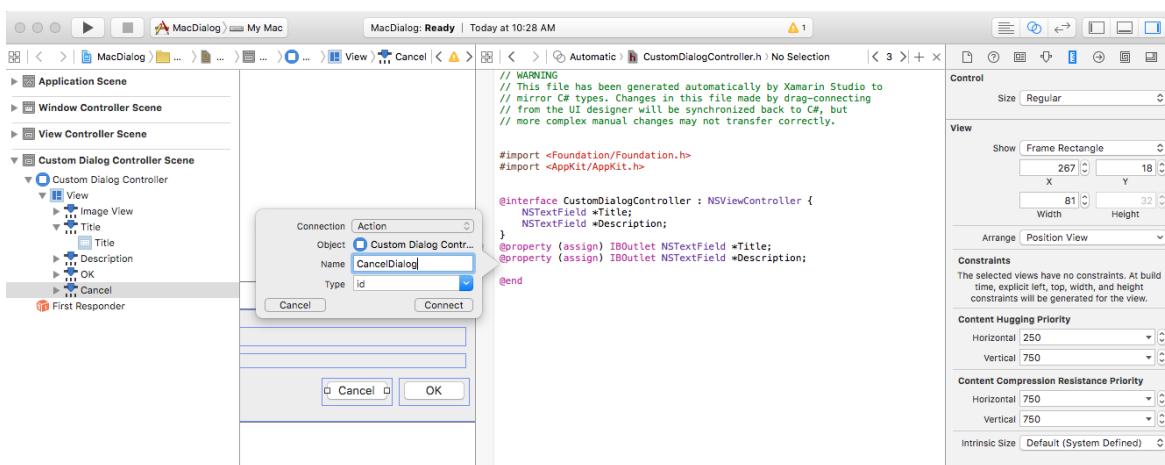


6. Create a **Modal Segue** from the Main Window of your app to the new View Controller by control-dragging from the UI element that will open the dialog to the dialog's window. Assign the **Identifier**

`ModalSegue` :



7. Wire-up any **Actions and Outlets**:



8. Save your changes and return to Visual Studio for Mac to sync with Xcode.

Make the `CustomDialogController.cs` file look like the following:

```
using System;
using Foundation;
using AppKit;

namespace MacDialog
{
    public partial class CustomDialogController : NSViewController
```

```

public partial class CustomDialogController : NSViewController
{
    #region Private Variables
    private string _dialogTitle = "Title";
    private string _dialogDescription = "Description";
    private NSViewController _presentor;
    #endregion

    #region Computed Properties
    public string DialogTitle {
        get { return _dialogTitle; }
        set { _dialogTitle = value; }
    }

    public string DialogDescription {
        get { return _dialogDescription; }
        set { _dialogDescription = value; }
    }

    public NSViewController Presentor {
        get { return _presentor; }
        set { _presentor = value; }
    }
    #endregion

    #region Constructors
    public CustomDialogController (IntPtr handle) : base (handle)
    {
    }
    #endregion

    #region Override Methods
    public override void ViewWillAppear ()
    {
        base.ViewWillAppear ();

        // Set initial title and description
        Title.StringValue = DialogTitle;
        Description.StringValue = DialogDescription;
    }
    #endregion

    #region Private Methods
    private void CloseDialog() {
        Presentor.DismissViewController (this);
    }
    #endregion

    #region Custom Actions
    partial void AcceptDialog (Foundation.NSObject sender) {
        RaiseDialogAccepted();
        CloseDialog();
    }

    partial void CancelDialog (Foundation.NSObject sender) {
        RaiseDialogCanceled();
        CloseDialog();
    }
    #endregion

    #region Events
    public EventHandler DialogAccepted;

    internal void RaiseDialogAccepted() {
        if (this.DialogAccepted != null)
            this.DialogAccepted (this, EventArgs.Empty);
    }

    public EventHandler DialogCanceled;

```

```

internal void RaiseDialogCanceled() {
    if (this.DialogCanceled != null)
        this.DialogCanceled (this, EventArgs.Empty);
}
#endregion
}

```

This code exposes a few properties to set the title and the description of the dialog and a few events to react to the dialog being canceled or accepted.

Next, edit the `viewController.cs` file, override the `PrepareForSegue` method and make it look like the following:

```

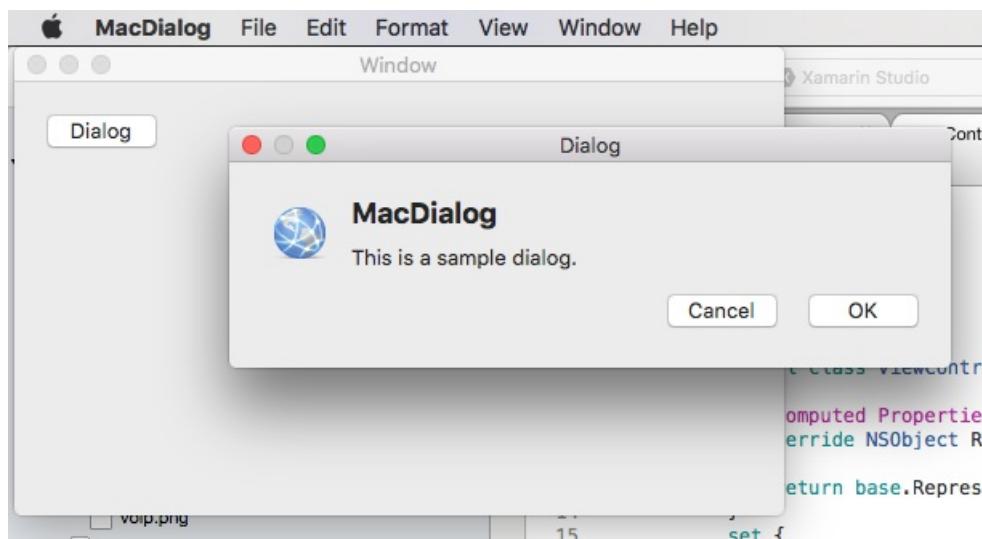
public override void PrepareForSegue (UIStoryboardSegue segue, NSObject sender)
{
    base.PrepareForSegue (segue, sender);

    // Take action based on the segue name
    switch (segue.Identifier) {
        case "ModalSegue":
            var dialog = segue.DestinationController as CustomDialogController;
            dialog.DialogTitle = "MacDialog";
            dialog.DialogDescription = "This is a sample dialog.";
            dialog.DialogAccepted += (s, e) => {
                Console.WriteLine ("Dialog accepted");
                DismissViewController (dialog);
            };
            dialog.Presentor = this;
            break;
    }
}

```

This code initializes the segue that we defined in Xcode's Interface Builder to our dialog and sets up the title and description. It also handles the choice the user makes in the dialog box.

We can run our application and display the custom dialog:



For more information about using windows in a Xamarin.Mac application, please see our [Working with Windows](#) documentation.

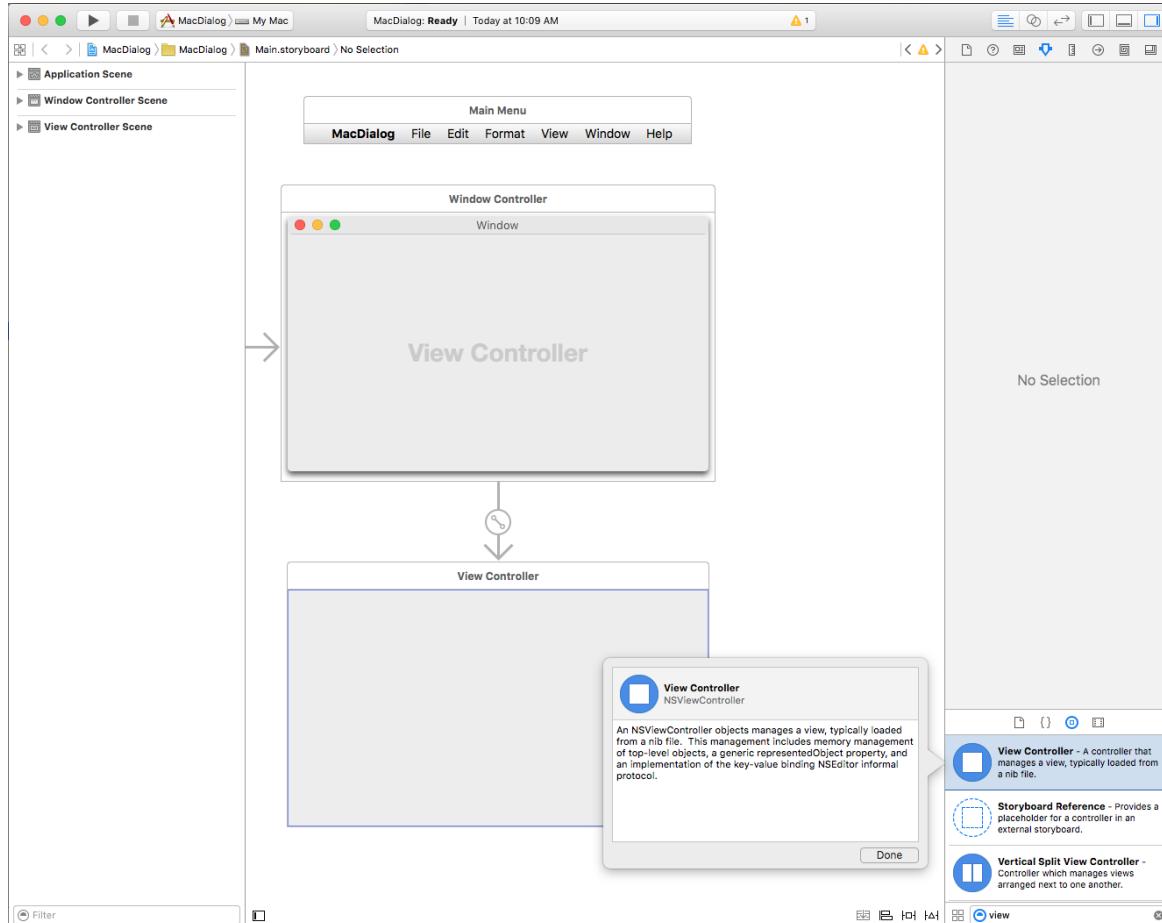
Creating a Custom Sheet

A *Sheet* is a modal dialog that is attached to a given document window, preventing users from interacting with

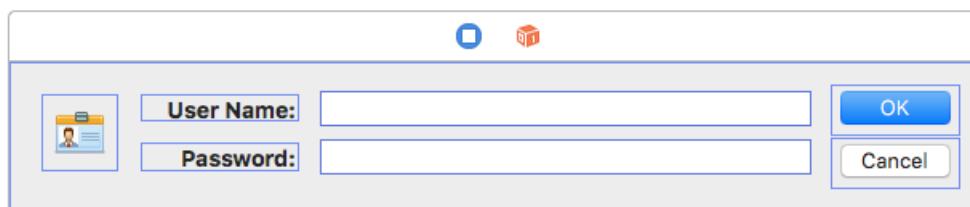
the window until they dismiss the dialog. A Sheet is attached to the window from which it emerges and only one sheet can be open for a window at any one time.

To create a Custom Sheet in Xamarin.Mac, let's do the following:

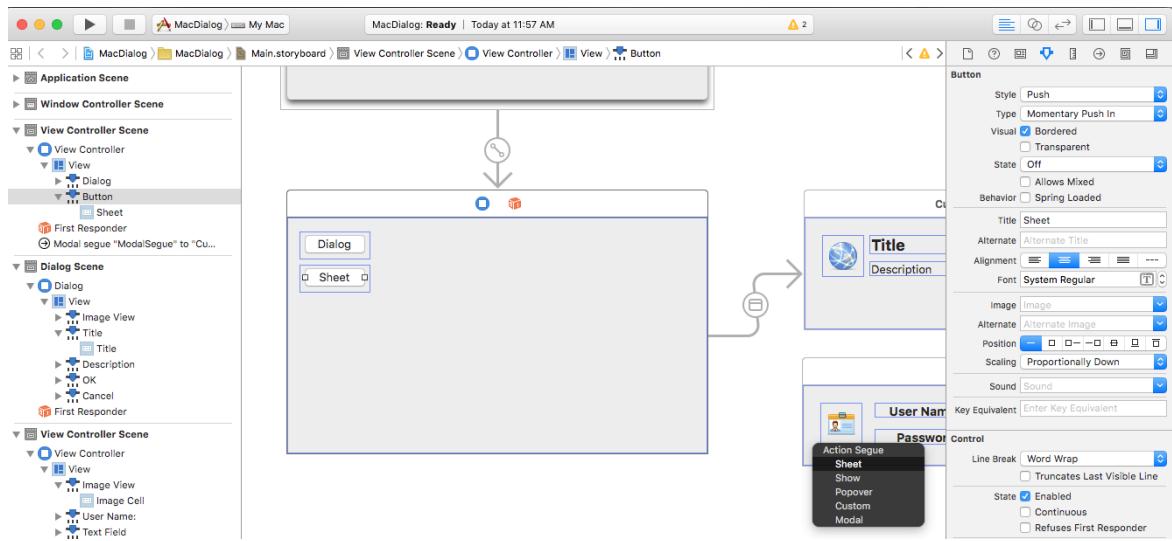
1. In the **Solution Explorer**, open the `Main.storyboard` file for editing in Xcode's Interface Builder.
2. Drag a new **View Controller** into the Design Surface:



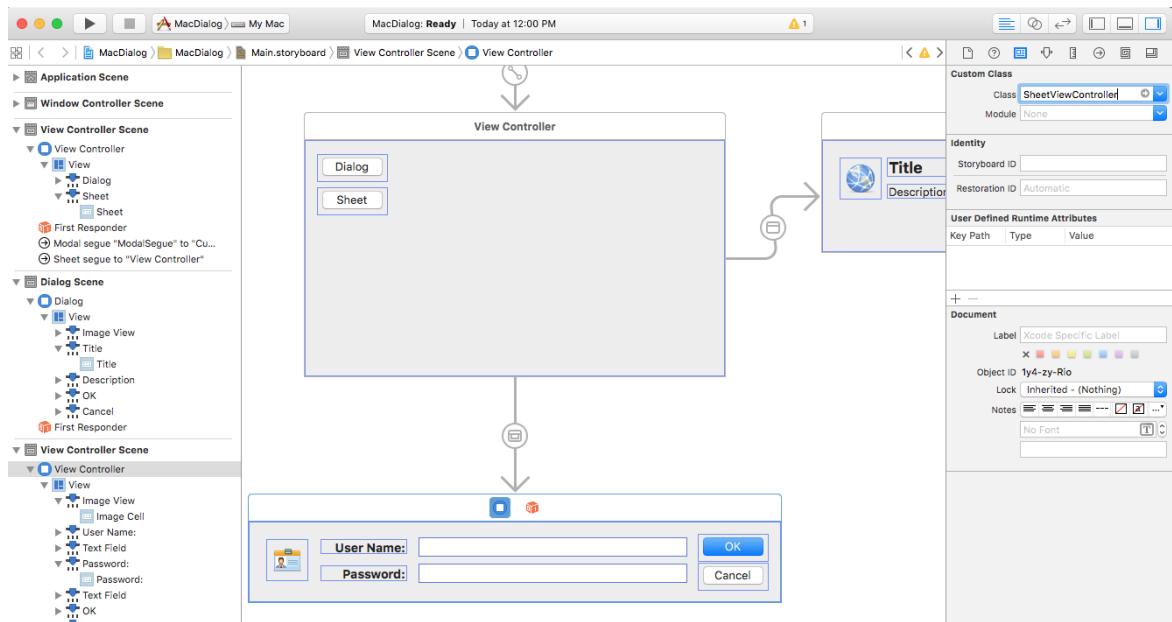
3. Design your user interface:



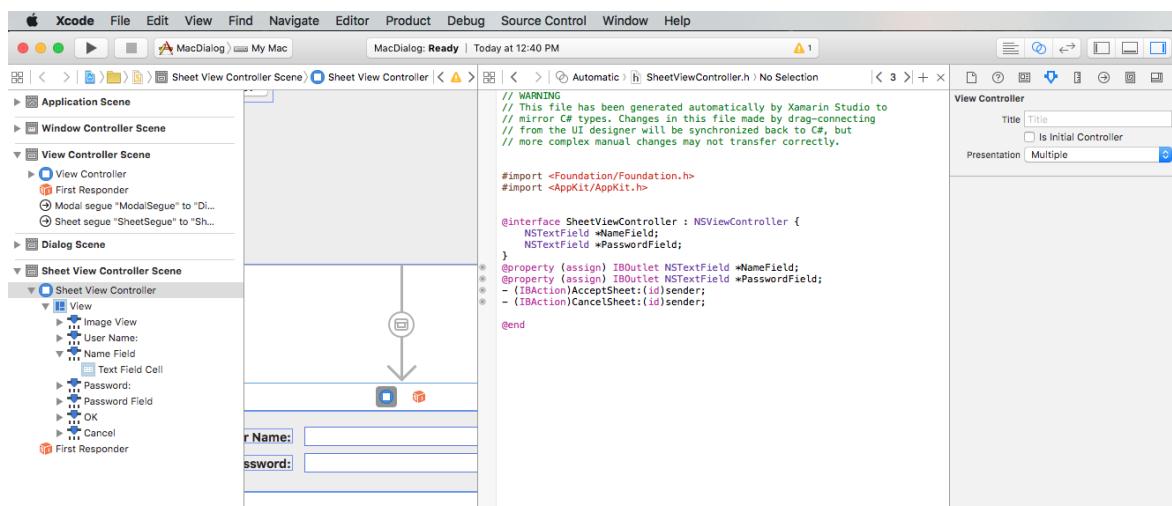
4. Create a **Sheet Segue** from your Main Window to the new View Controller:



5. In the Identity Inspector, name the View Controller's Class `SheetViewController`:



6. Define any needed Outlets and Actions:



7. Save your changes and return to Visual Studio for Mac to sync.

Next, edit the `SheetViewController.cs` file and make it look like the following:

```

using System;
using Foundation;

```

```

using AppKit;

namespace MacDialog
{
    public partial class SheetViewController : NSViewController
    {
        #region Private Variables
        private string _userName = "";
        private string _password = "";
        private NSViewController _presentor;
        #endregion

        #region Computed Properties
        public string UserName {
            get { return _userName; }
            set { _userName = value; }
        }

        public string Password {
            get { return _password; }
            set { _password = value; }
        }

        public NSViewController Presentor {
            get { return _presentor; }
            set { _presentor = value; }
        }
        #endregion

        #region Constructors
        public SheetViewController (IntPtr handle) : base (handle)
        {
        }
        #endregion

        #region Override Methods
        public override void ViewWillAppear ()
        {
            base.ViewWillAppear ();

            // Set initial values
            NameField.StringValue = UserName;
            PasswordField.StringValue = Password;

            // Wireup events
            NameField.Changed += (sender, e) => {
                UserName = NameField.StringValue;
            };
            PasswordField.Changed += (sender, e) => {
                Password = PasswordField.StringValue;
            };
        }
        #endregion

        #region Private Methods
        private void CloseSheet() {
            Presentor.DismissViewController (this);
        }
        #endregion

        #region Custom Actions
        partial void AcceptSheet (Foundation.NSObject sender) {
            RaiseSheetAccepted();
            CloseSheet();
        }

        partial void CancelSheet (Foundation.NSObject sender) {
            RaiseSheetCanceled();
            CloseSheet();
        }
    }
}

```

```

    }

    #endregion

    #region Events
    public EventHandler SheetAccepted;

    internal void RaiseSheetAccepted() {
        if (this.SheetAccepted != null)
            this.SheetAccepted (this, EventArgs.Empty);
    }

    public EventHandler SheetCanceled;

    internal void RaiseSheetCanceled() {
        if (this.SheetCanceled != null)
            this.SheetCanceled (this, EventArgs.Empty);
    }
    #endregion
}
}

```

Next, edit the `viewController.cs` file, edit the `PrepareForSegue` method and make it look like the following:

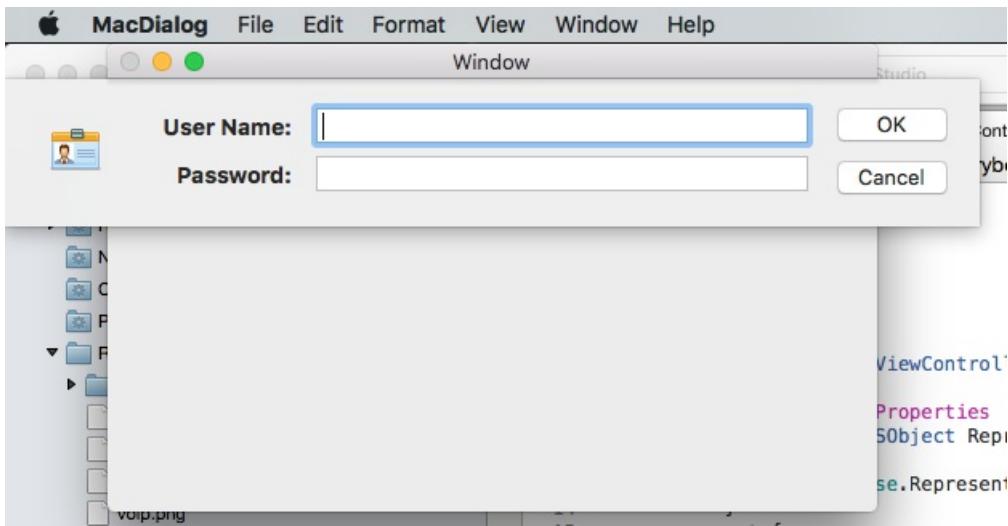
```

public override void PrepareForSegue (UIStoryboardSegue segue, NSObject sender)
{
    base.PrepareForSegue (segue, sender);

    // Take action based on the segue name
    switch (segue.Identifier) {
        case "ModalSegue":
            var dialog = segue.DestinationController as CustomDialogController;
            dialog.DialogTitle = "MacDialog";
            dialog.DialogDescription = "This is a sample dialog.";
            dialog.DialogAccepted += (s, e) => {
                Console.WriteLine ("Dialog accepted");
                DismissViewController (dialog);
            };
            dialog.Presentor = this;
            break;
        case "SheetSegue":
            var sheet = segue.DestinationController as SheetViewController;
            sheet.SheetAccepted += (s, e) => {
                Console.WriteLine ("User Name: {0} Password: {1}", sheet.UserName, sheet.Password);
            };
            sheet.Presentor = this;
            break;
    }
}

```

If we run our application and open the Sheet, it will be attached to the window:



Creating a Preferences Dialog

Before we lay out the Preference View in Interface Builder, we'll need to add a custom segue type to handle switching out the preferences. Add a new class to your project and call it `ReplaceViewSegue`. Edit the class and make it look like the following:

```

using System;
using AppKit;
using Foundation;

namespace MacWindows
{
    [Register("ReplaceViewSeque")]
    public class ReplaceViewSeque : NSStoryboardSegue
    {
        #region Constructors
        public ReplaceViewSeque() {

        }

        public ReplaceViewSeque (string identifier, NSObject sourceController, NSObject
destinationController) : base(identifier,sourceController,destinationController) {

        }

        public ReplaceViewSeque (IntPtr handle) : base(handle) {

        }

        public ReplaceViewSeque (NSObjectFlag x) : base(x) {
        }
        #endregion

        #region Override Methods
        public override void Perform ()
        {
            // Cast the source and destination controllers
            var source = SourceController as NSViewController;
            var destination = DestinationController as NSViewController;

            // Is there a source?
            if (source == null) {
                // No, get the current key window
                var window = NSApplication.SharedApplication.KeyWindow;

                // Swap the controllers
                window.ContentViewController = destination;

                // Release memory
                window.ContentViewController?.RemoveFromParentViewController ();
            } else {
                // Swap the controllers
                source.View.Window.ContentViewController = destination;

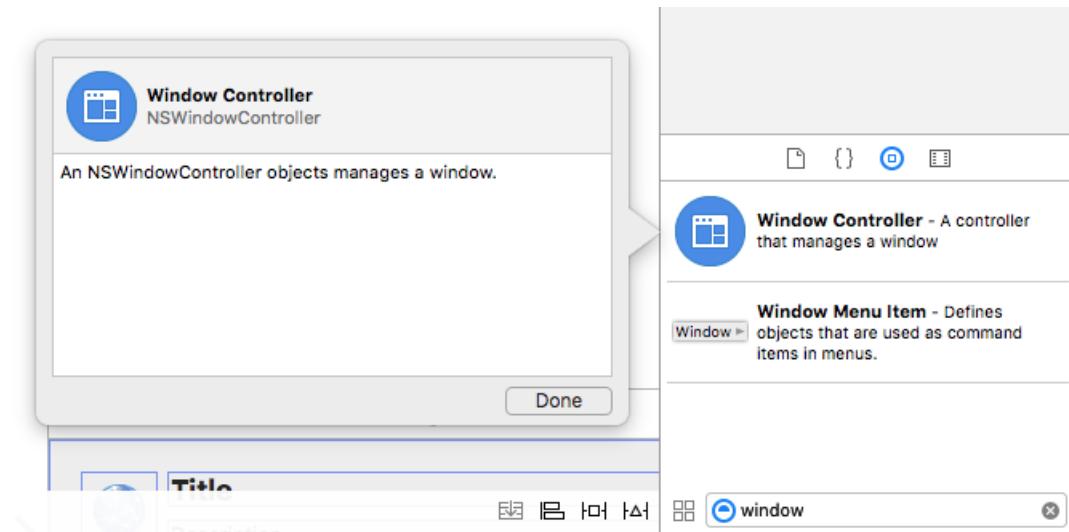
                // Release memory
                source.RemoveFromParentViewController ();
            }
        }
        #endregion
    }
}

```

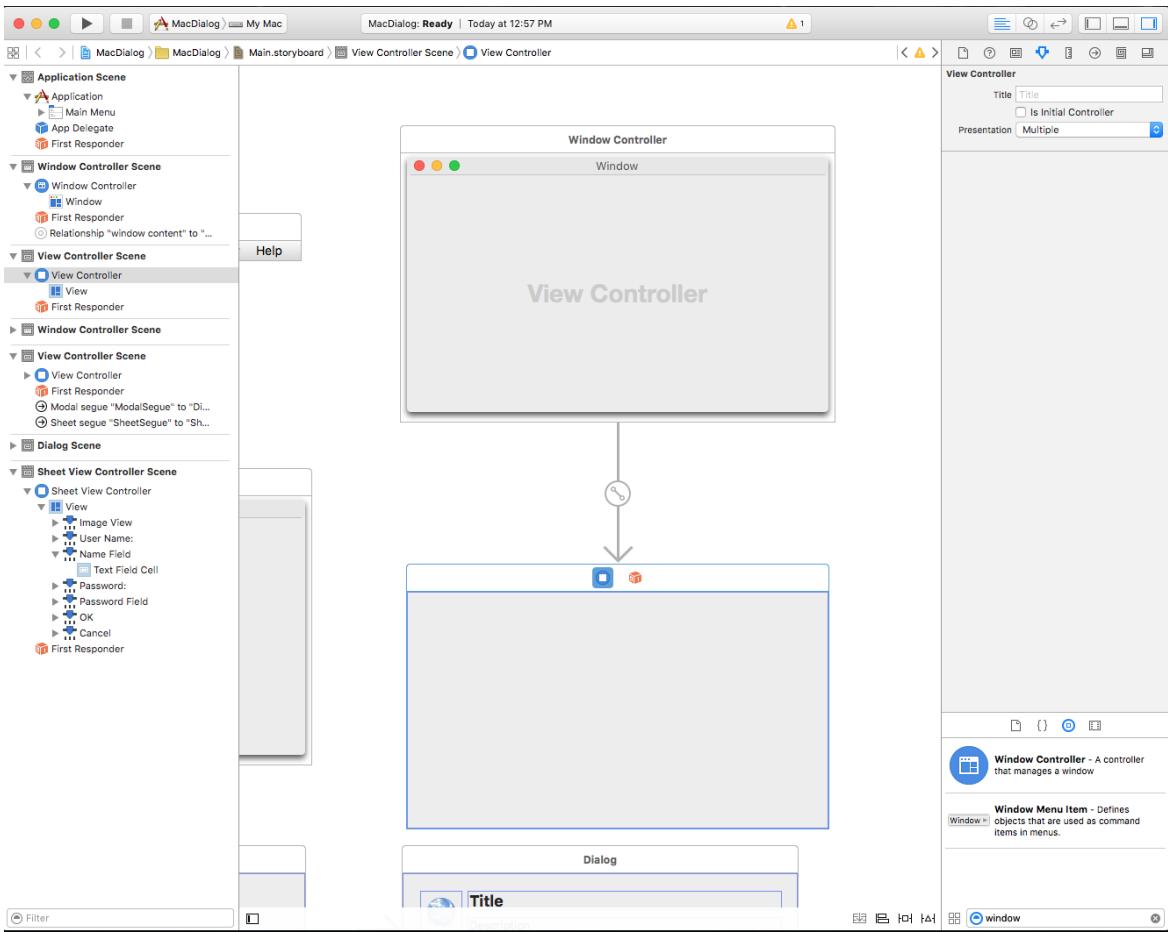
With the custom segue created, we can add a new window in Xcode's Interface Builder to handle our preferences.

To add a new window, do the following:

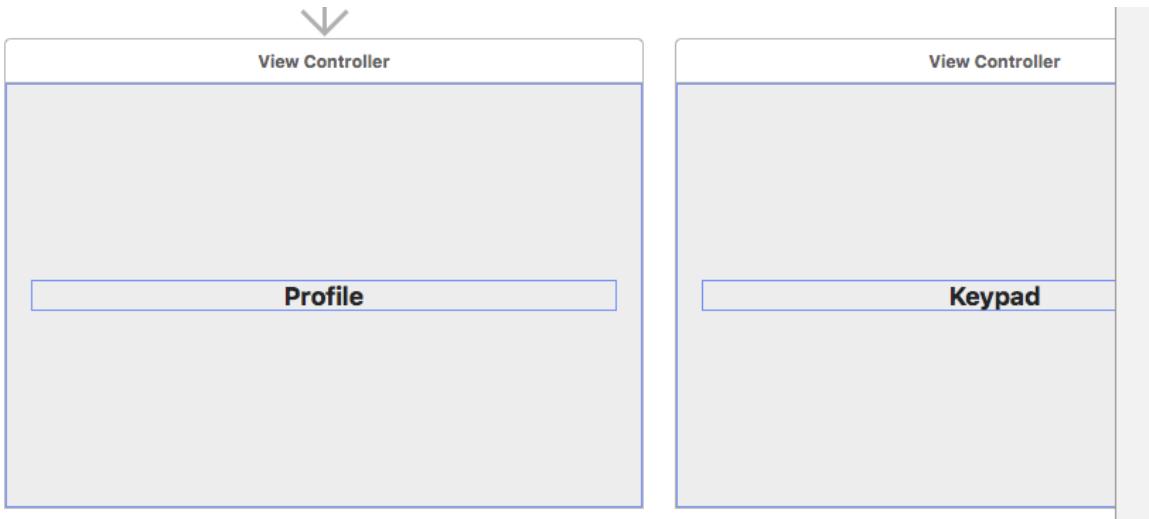
1. In the **Solution Explorer**, open the `Main.storyboard` file for editing in Xcode's Interface Builder.
2. Drag a new **Window Controller** into the Design Surface:



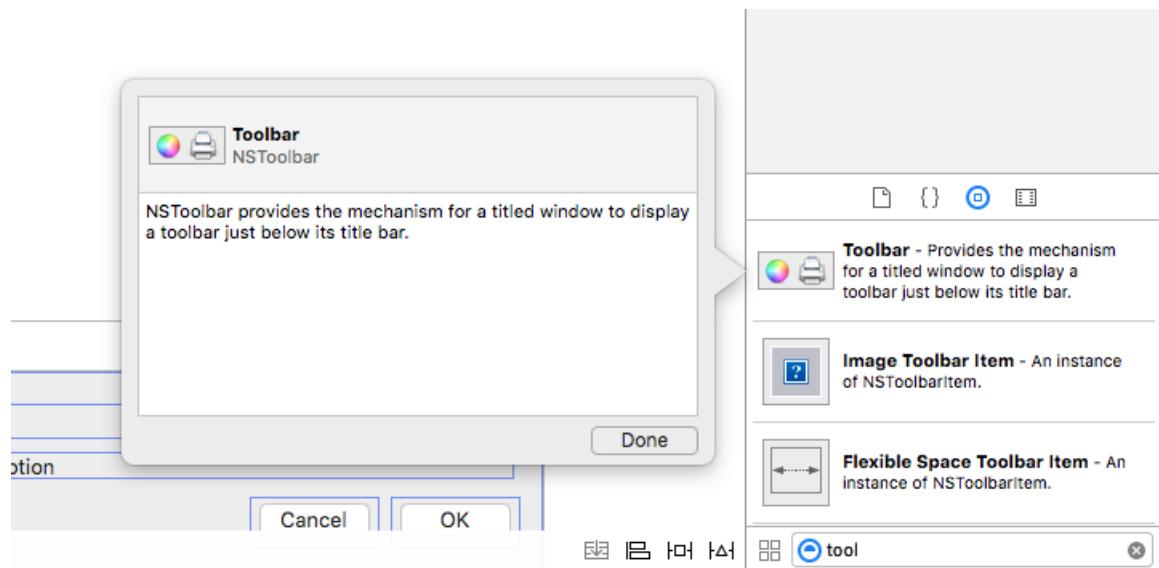
3. Arrange the Window near the **Menu Bar** designer:



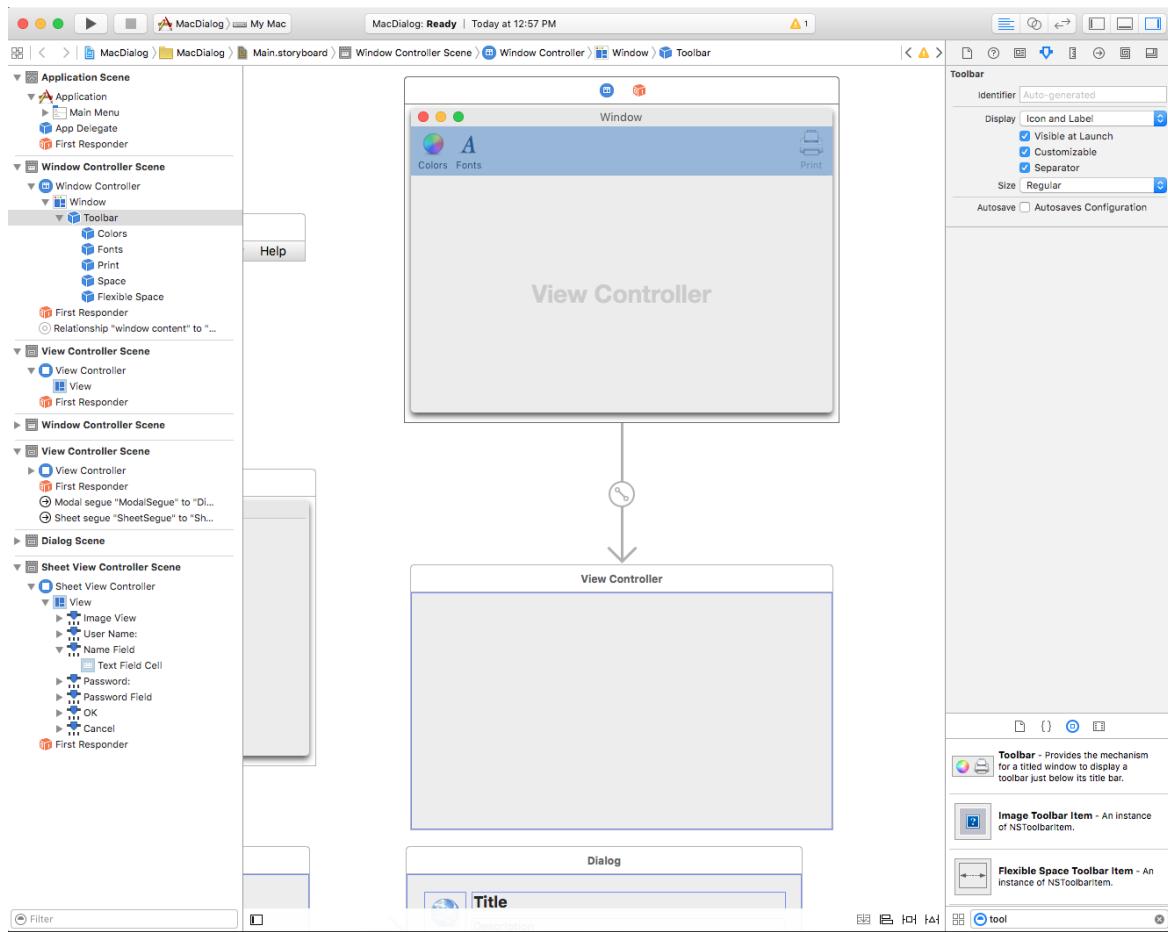
4. Create copies of the attached View Controller as there will be tabs in your preference view:



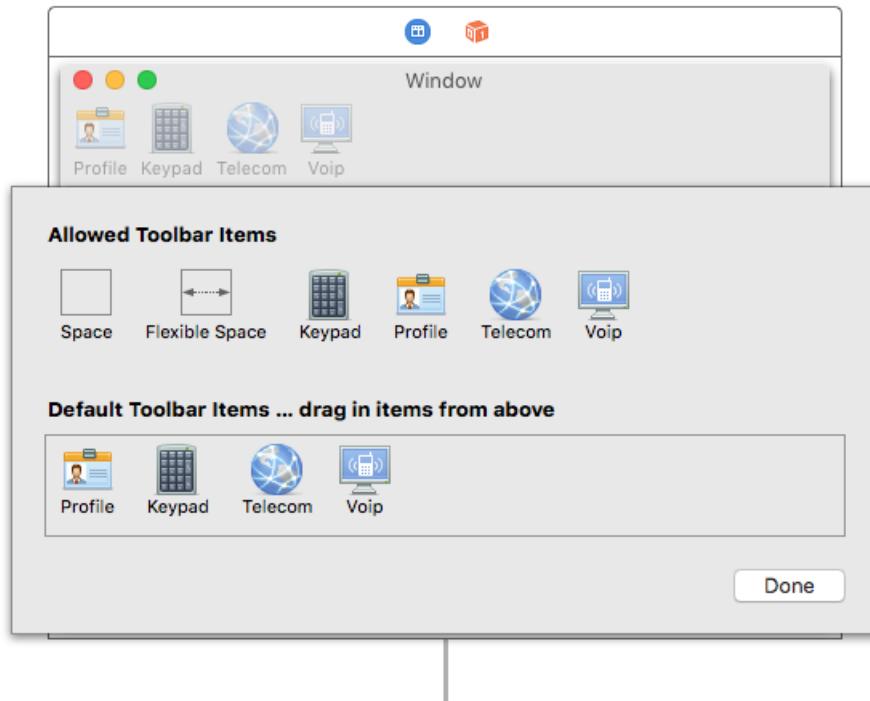
5. Drag a new Toolbar Controller from the Library:



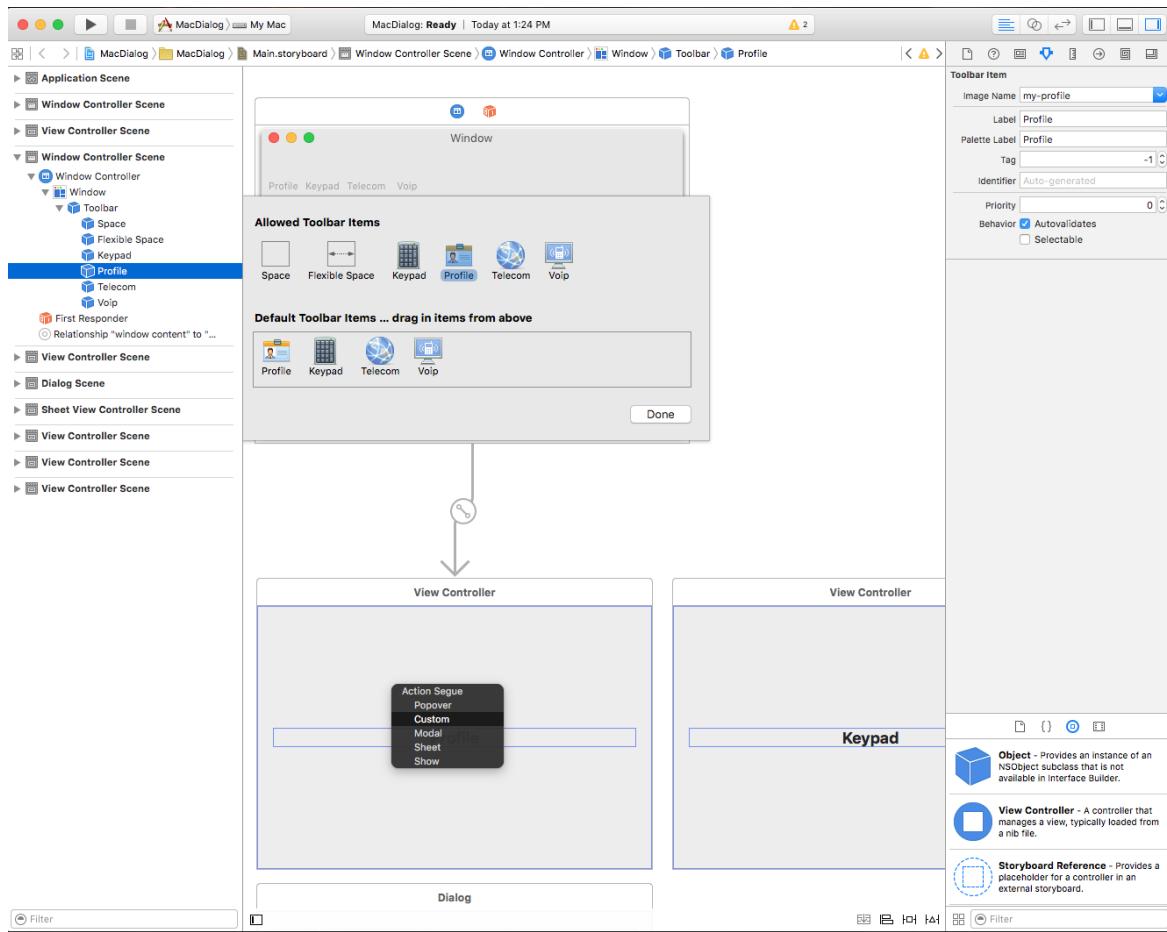
6. And drop it on the Window in the Design Surface:



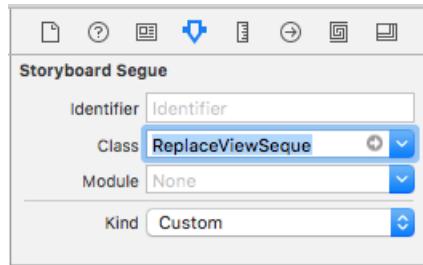
7. Layout the design of your toolbar:



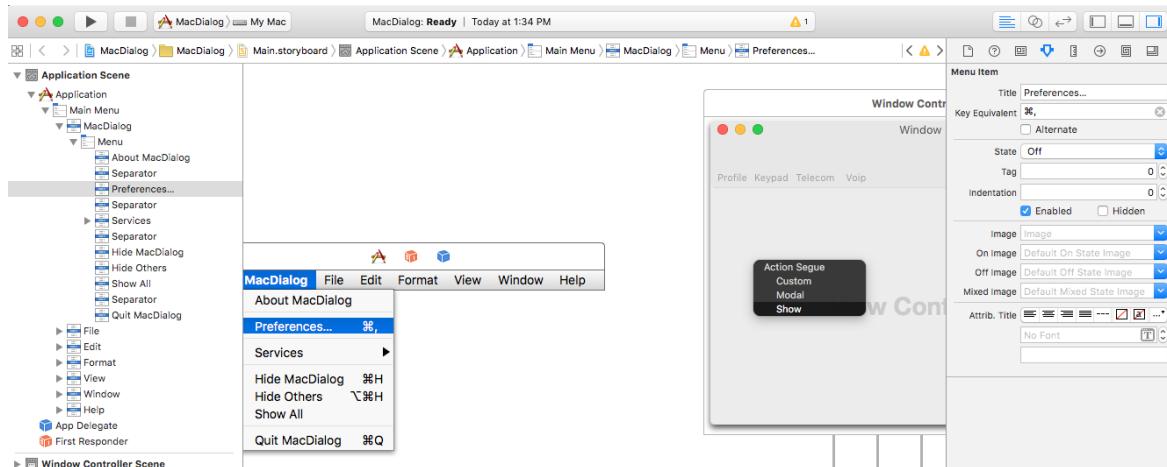
8. Control-Click and drag from each **Toolbar Button** to the Views you created above. Select a **Custom** segue type:



9. Select the new Segue and set the Class to `ReplaceStoryboardSegue` :

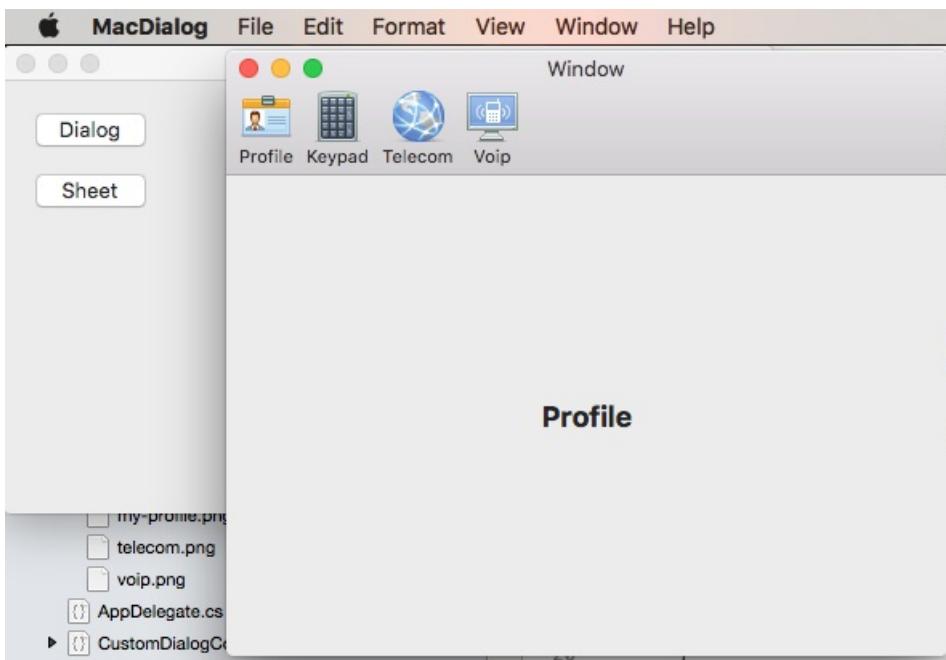


10. In the Menubar Designer on the Design Surface, from the Application Menu select Preferences..., control-click and drag to the Preferences Window to create a Show segue:



11. Save your changes and return to Visual Studio for Mac to sync.

If we run the code and select the Preferences... from the Application Menu, the window will be displayed:



For more information on working with Windows and Toolbars, please see our [Windows](#) and [Toolbars](#) documentation.

Saving and Loading Preferences

In a typical macOS App, when the user makes changes to any of the App's User Preferences, those changes are saved automatically. The easiest way to handle this in a Xamarin.Mac app, is to create a single class to manage all of the user's preferences and share it system-wide.

First, add a new `AppPreferences` class to the project and inherit from `NSObject`. The preferences will be designed to use [Data Binding and Key-Value Coding](#) which will make the process of creating and maintaining the preference forms much simpler. Since the Preferences will consist of a small amount of simple datatypes, use the built in `NSUserDefaults` to store and retrieve values.

Edit the `AppPreferences.cs` file and make it look like the following:

```
using System;
using Foundation;
using AppKit;

namespace SourceWriter
{
    [Register("AppPreferences")]
    public class AppPreferences : NSObject
    {
        #region Computed Properties
        [Export("DefaultLanguage")]
        public int DefaultLanguage {
            get {
                var value = LoadInt ("DefaultLanguage", 0);
                return value;
            }
            set {
                WillChangeValue ("DefaultLanguage");
                SaveInt ("DefaultLanguage", value, true);
                DidChangeValue ("DefaultLanguage");
            }
        }

        [Export("SmartLinks")]
        public bool SmartLinks {
            get { return LoadBool ("SmartLinks", true); }
            set {
                WillChangeValue ("SmartLinks");
                SaveBool ("SmartLinks", value, true);
                DidChangeValue ("SmartLinks");
            }
        }
    }
}
```

```

    ...
        WillChangeValue ("SmartLinks");
        SaveBool ("SmartLinks", value, true);
        DidChangeValue ("SmartLinks");
    }
}

// Define any other required user preferences in the same fashion
...

[Export("EditorBackgroundColor")]
public NSColor EditorBackgroundColor {
    get { return LoadColor("EditorBackgroundColor", NSColor.White); }
    set {
        WillChangeValue ("EditorBackgroundColor");
        SaveColor ("EditorBackgroundColor", value, true);
        DidChangeValue ("EditorBackgroundColor");
    }
}
#endregion

#region Constructors
public AppPreferences ()
{
}
#endregion

#region Public Methods
public int LoadInt(string key, int defaultValue) {
    // Attempt to read int
    var number = NSUserDefaults.StandardUserDefaults.IntForKey(key);

    // Take action based on value
    if (number == null) {
        return defaultValue;
    } else {
        return (int)number;
    }
}

public void SaveInt(string key, int value, bool sync) {
    NSUserDefaults.StandardUserDefaults.SetInt(value, key);
    if (sync) NSUserDefaults.StandardUserDefaults.Synchronize ();
}

public bool LoadBool(string key, bool defaultValue) {
    // Attempt to read int
    var value = NSUserDefaults.StandardUserDefaults.BoolForKey(key);

    // Take action based on value
    if (value == null) {
        return defaultValue;
    } else {
        return value;
    }
}

public void SaveBool(string key, bool value, bool sync) {
    NSUserDefaults.StandardUserDefaults.SetBool(value, key);
    if (sync) NSUserDefaults.StandardUserDefaults.Synchronize ();
}

public string NSColorToString(NSColor color, bool withAlpha) {
    // Break color into pieces
    nfloat red=0, green=0, blue=0, alpha=0;
    color.GetRgba (out red, out green, out blue, out alpha);

    // Adjust to byte
    alpha *= 255;
    red *= 255;
}

```

```

    red *= 255;
    green *= 255;
    blue *= 255;

    //With the alpha value?
    if (withAlpha) {
        return String.Format ("#{0:X2}{1:X2}{2:X2}{3:X2}", (int)alpha, (int)red, (int)green,
(int)blue);
    } else {
        return String.Format ("#{0:X2}{1:X2}{2:X2}", (int)red, (int)green, (int)blue);
    }
}

public NSColor NSColorFromHexString (string hexValue)
{
    var colorString = hexValue.Replace ("#", "");
    float red, green, blue, alpha;

    // Convert color based on length
    switch (colorString.Length) {
        case 3 : // #RGB
            red = Convert.ToInt32(string.Format("{0}{0}", colorString.Substring(0, 1)), 16) / 255f;
            green = Convert.ToInt32(string.Format("{0}{0}", colorString.Substring(1, 1)), 16) / 255f;
            blue = Convert.ToInt32(string.Format("{0}{0}", colorString.Substring(2, 1)), 16) / 255f;
            return NSColor.FromRgba(red, green, blue, 1.0f);
        case 6 : // #RRGGBB
            red = Convert.ToInt32(colorString.Substring(0, 2), 16) / 255f;
            green = Convert.ToInt32(colorString.Substring(2, 2), 16) / 255f;
            blue = Convert.ToInt32(colorString.Substring(4, 2), 16) / 255f;
            return NSColor.FromRgba(red, green, blue, 1.0f);
        case 8 : // #AARRGGBB
            alpha = Convert.ToInt32(colorString.Substring(0, 2), 16) / 255f;
            red = Convert.ToInt32(colorString.Substring(2, 2), 16) / 255f;
            green = Convert.ToInt32(colorString.Substring(4, 2), 16) / 255f;
            blue = Convert.ToInt32(colorString.Substring(6, 2), 16) / 255f;
            return NSColor.FromRgba(red, green, blue, alpha);
        default :
            throw new ArgumentOutOfRangeException(string.Format("Invalid color value '{0}'. It should be
a hex value of the form #RGB, #RRGGBB or #AARRGGBB", hexValue));
    }
}

public NSColor LoadColor(string key, NSColor defaultValue) {

    // Attempt to read color
    var hex = NSUserDefaults.StandardUserDefaults.StringForKey(key);

    // Take action based on value
    if (hex == null) {
        return defaultValue;
    } else {
        return NSColorFromHexString (hex);
    }
}

public void SaveColor(string key, NSColor color, bool sync) {
    // Save to default
    NSUserDefaults.StandardUserDefaults.SetString(NSColorToHexString(color,true), key);
    if (sync) NSUserDefaults.StandardUserDefaults.Synchronize ();
}

#endifregion
}
}

```

This class contains a few helper routines such as `SaveInt`, `LoadInt`, `SaveColor`, `LoadColor`, etc. to make working with `NSUserDefaults` easier. Also, since `NSUserDefaults` does not have a built-in way to handle `NSColors`, the `NSColorToHexString` and `NSColorFromHexString` methods are used to convert colors to web-based

hex strings (#RRGGBBAA where AA is the alpha transparency) that can be easily stored and retrieved.

In the `AppDelegate.cs` file, create an instance of the `AppPreferences` object that will be used app-wide:

```
using AppKit;
using Foundation;
using System.IO;
using System;

namespace SourceWriter
{
    [Register ("AppDelegate")]
    public class AppDelegate : NSApplicationDelegate
    {
        #region Computed Properties
        public int NewWindowNumber { get; set;} = -1;

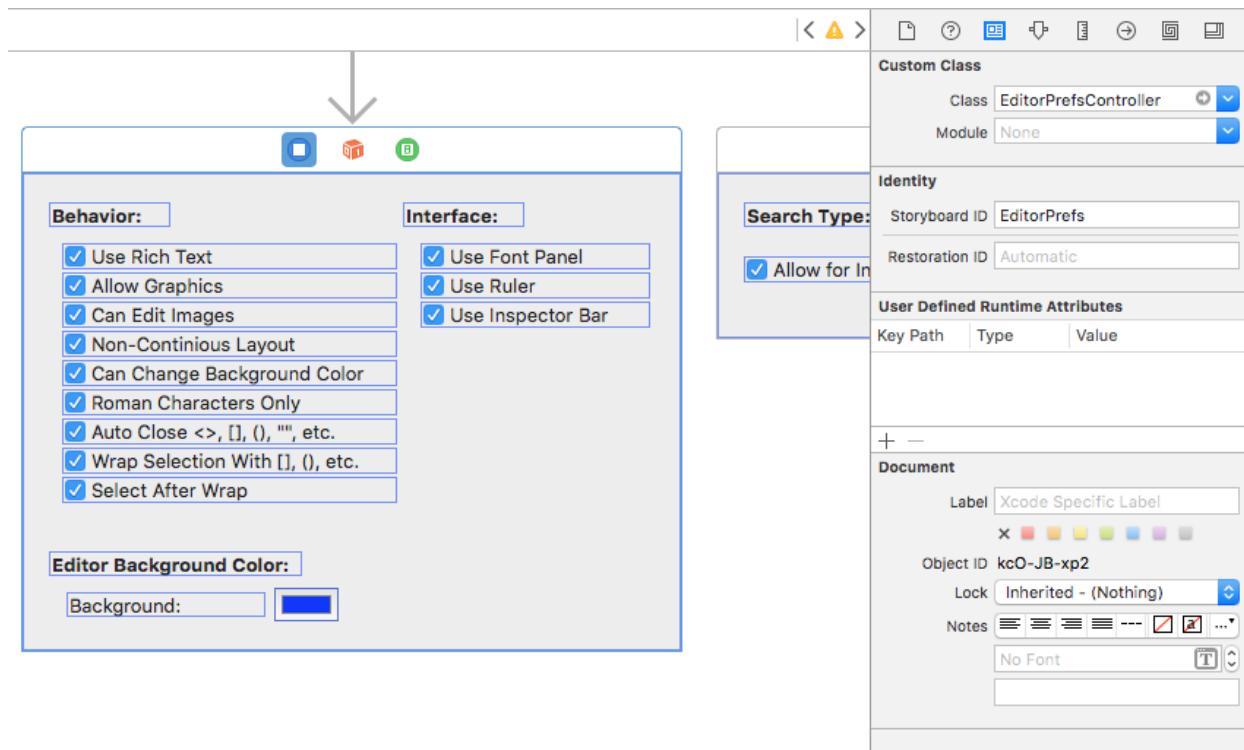
        public AppPreferences Preferences { get; set; } = new AppPreferences();
        #endregion

        #region Constructors
        public AppDelegate ()
        {
        }
        #endregion

        ...
    }
}
```

Wiring Preferences to Preference Views

Next, connect Preference class to UI elements on the Preference Window and Views created above. In Interface Builder, select a Preference View Controller and switch to the **Identity Inspector**, create a custom class for the controller:



Switch back to Visual Studio for Mac to sync your changes and open the newly created class for editing. Make the class look like the following:

```

using System;
using Foundation;
using AppKit;

namespace SourceWriter
{
    public partial class EditorPrefsController : NSViewController
    {
        #region Application Access
        public static AppDelegate App {
            get { return (AppDelegate)NSApplication.SharedApplication.Delegate; }
        }
        #endregion

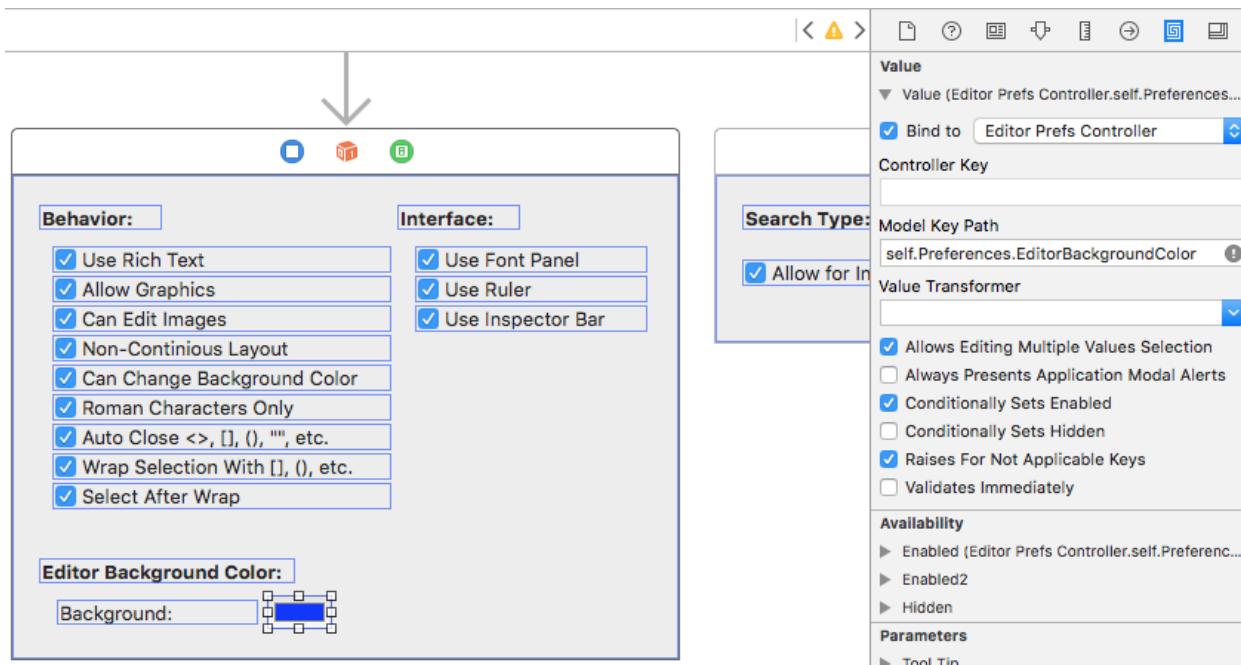
        #region Computed Properties
        [Export("Preferences")]
        public AppPreferences Preferences {
            get { return App.Preferences; }
        }
        #endregion

        #region Constructors
        public EditorPrefsController (IntPtr handle) : base (handle)
        {
        }
        #endregion
    }
}

```

Notice that this class has done two things here: First, there is a helper `App` property to make accessing the `AppDelegate` easier. Second, the `Preferences` property exposes the global `AppPreferences` class for data binding with any UI controls placed on this View.

Next, double click the Storyboard file to re-open it in Interface Builder (and see the changes just made above). Drag any UI controls required to build the preferences interface into the View. For each control, switch to the **Binding Inspector** and bind to the individual properties of the `AppPreference` class:



Repeat the above steps for all of the panels (View Controllers) and Preference Properties required.

Applying Preference Changes to All Open Windows

As stated above, in a typical macOS App, when the user makes changes to any of the App's User Preferences,

those changes are saved automatically and applied to any windows the user might have open in the application.

Careful planning and design of your app's preferences and windows will allow this process to happen smoothly and transparently to the end user, with a minimal amount of coding work.

For any Window that will be consuming App Preferences, add the following helper property to its Content View Controller to make accessing our **AppDelegate** easier:

```
#region Application Access
public static AppDelegate App {
    get { return (AppDelegate)NSApplication.SharedApplication.Delegate; }
}
#endregion
```

Next, add a class to configure the contents or behavior based on the user's preferences:

```
public void ConfigureEditor() {

    // General Preferences
    TextEditor.AutomaticLinkDetectionEnabled = App.Preferences.SmartLinks;
    TextEditor.AutomaticQuoteSubstitutionEnabled = App.Preferences.SmartQuotes;
    ...

}
```

You need to call the configuration method when the Window is first opened to make sure it conforms to the user's preferences:

```
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();

    // Configure editor from user preferences
    ConfigureEditor ();
    ...
}
```

Next, edit the `AppDelegate.cs` file and add the following method to apply any preference changes to all open windows:

```
public void UpdateWindowPreferences() {

    // Process all open windows
    for(int n=0; n<NSApplication.SharedApplication.Windows.Length; ++n) {
        var content = NSApplication.SharedApplication.Windows[n].ContentViewController as ViewController;
        if (content != null ) {
            // Reformat all text
            content.ConfigureEditor ();
        }
    }
}
```

Next, add a `PreferenceWindowDelegate` class to the project and make it look like the following:

```

using System;
using AppKit;
using System.IO;
using Foundation;

namespace SourceWriter
{
    public class PreferenceWindowDelegate : NSWindowDelegate
    {
        #region Application Access
        public static AppDelegate App {
            get { return (AppDelegate)NSApplication.SharedApplication.Delegate; }
        }
        #endregion

        #region Computed Properties
        public NSWindow Window { get; set; }
        #endregion

        #region Constructors
        public PreferenceWindowDelegate (NSWindow window)
        {
            // Initialize
            this.Window = window;
        }
        #endregion

        #region Override Methods
        public override bool WindowShouldClose (Foundation.NSObject sender)
        {

            // Apply any changes to open windows
            App.UpdateWindowPreferences();

            return true;
        }
        #endregion
    }
}

```

This will cause any preference changes to be sent to all open Windows when the preference Window closes.

Finally, edit the Preference Window Controller and add the delegate created above:

```

using System;
using Foundation;
using AppKit;

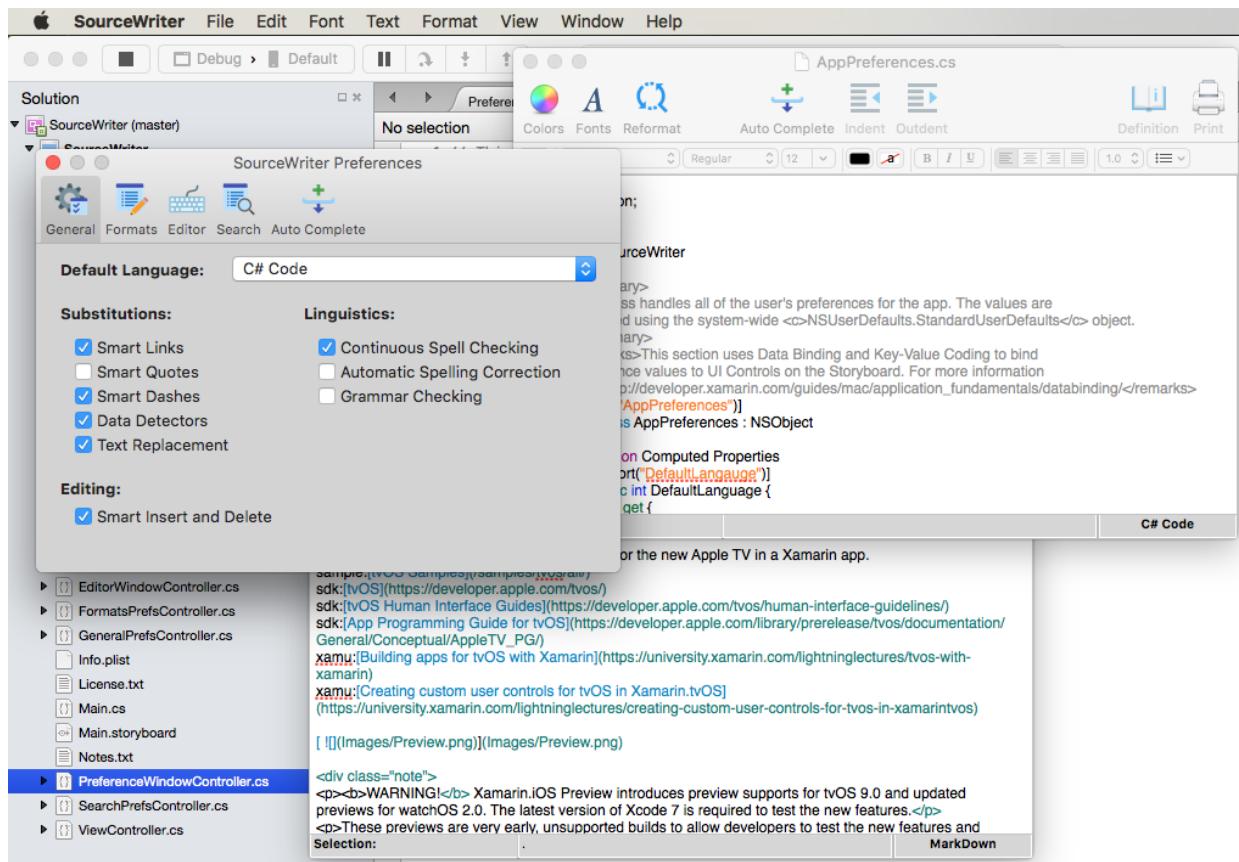
namespace SourceWriter
{
    public partial class PreferenceWindowController : NSWindowController
    {
        #region Constructors
        public PreferenceWindowController (IntPtr handle) : base (handle)
        {
        }
        #endregion

        #region Override Methods
        public override void WindowDidLoad ()
        {
            base.WindowDidLoad ();

            // Initialize
            Window.Delegate = new PreferenceWindowDelegate(Window);
            Toolbar.SelectedItemIdentifier = "General";
        }
        #endregion
    }
}

```

With all these changes in place, if the user edits the App's Preferences and closes the Preference Window, the changes will be applied to all open Windows:



The Open Dialog

The Open Dialog gives users a consistent way to find and open an item in an application. To display an Open Dialog in a Xamarin.Mac application, use the following code:

```

var dlg = NSOpenPanel.OpenPanel;
dlg.CanChooseFiles = true;
dlg.CanChooseDirectories = false;
dlg.AllowedFileTypes = new string[] { "txt", "html", "md", "css" };

if (dlg.RunModal () == 1) {
    // Nab the first file
    var url = dlgUrls [0];

    if (url != null) {
        var path = url.Path;

        // Create a new window to hold the text
        var newWindowController = new MainWindowController ();
        newWindowController.Window.MakeKeyAndOrderFront (this);

        // Load the text into the window
        var window = newWindowController.Window as MainWindow;
        window.Text = File.ReadAllText(path);
        windowSetTitleWithRepresentedFilename (Path.GetFileName(path));
        window.RepresentedUrl = url;

    }
}

```

In the above code, we are opening a new document window to display the contents of the file. You'll need to replace this code with functionality required by your application.

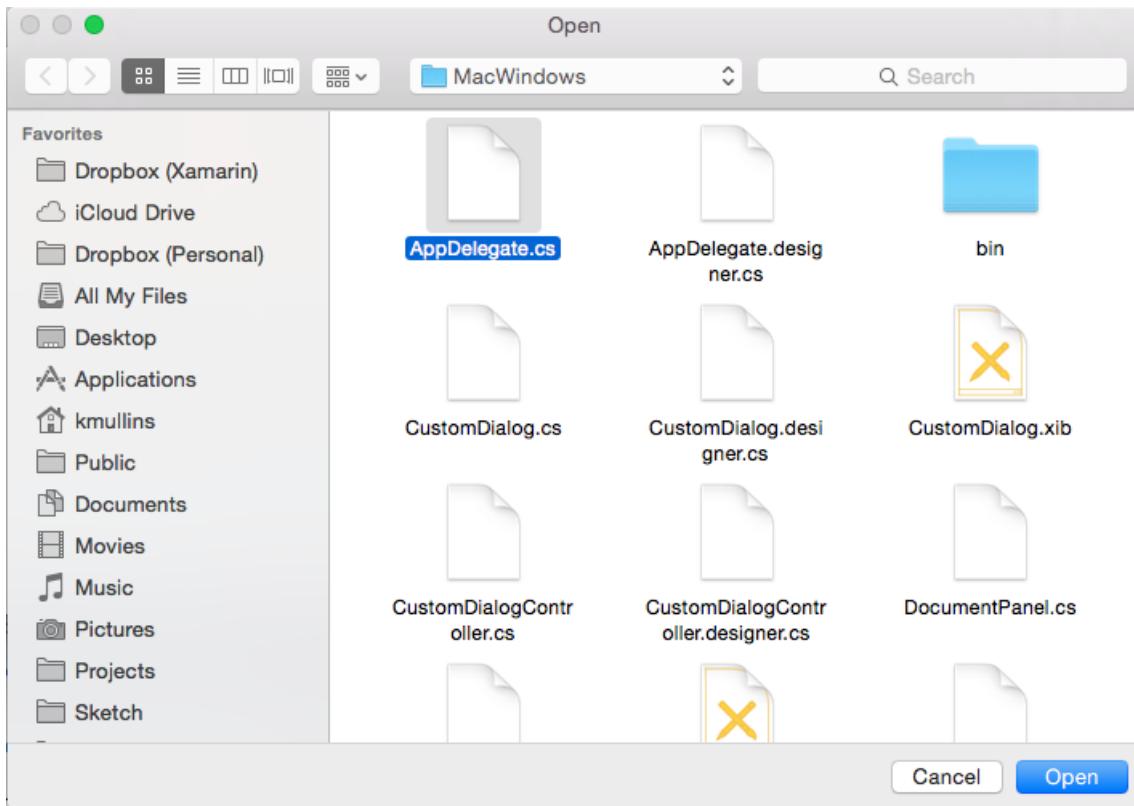
The following properties are available when working with a `NSOpenPanel`:

- **CanChooseFiles** - If `true` the user can select files.
- **CanChooseDirectories** - If `true` the user can select directories.
- **AllowsMultipleSelection** - If `true` the user can select more than one file at a time.
- **ResolveAliases** - If `true` selecting an alias, resolves it to the original file's path.
- **AllowedFileTypes** - Is a string array of file types that the user can select as either an extension or *UTI*. The default value is `null`, which allows any file to be opened.

The `RunModal ()` method displays the Open Dialog and allow the user to select files or directories (as specified by the properties) and returns `1` if the user clicks the **Open** button.

The Open Dialog returns the user's selected files or directories as an array of URLs in the `URL` property.

If we run the program and select the **Open...** item from the **File** menu, the following is displayed:



The Print and Page Setup Dialogs

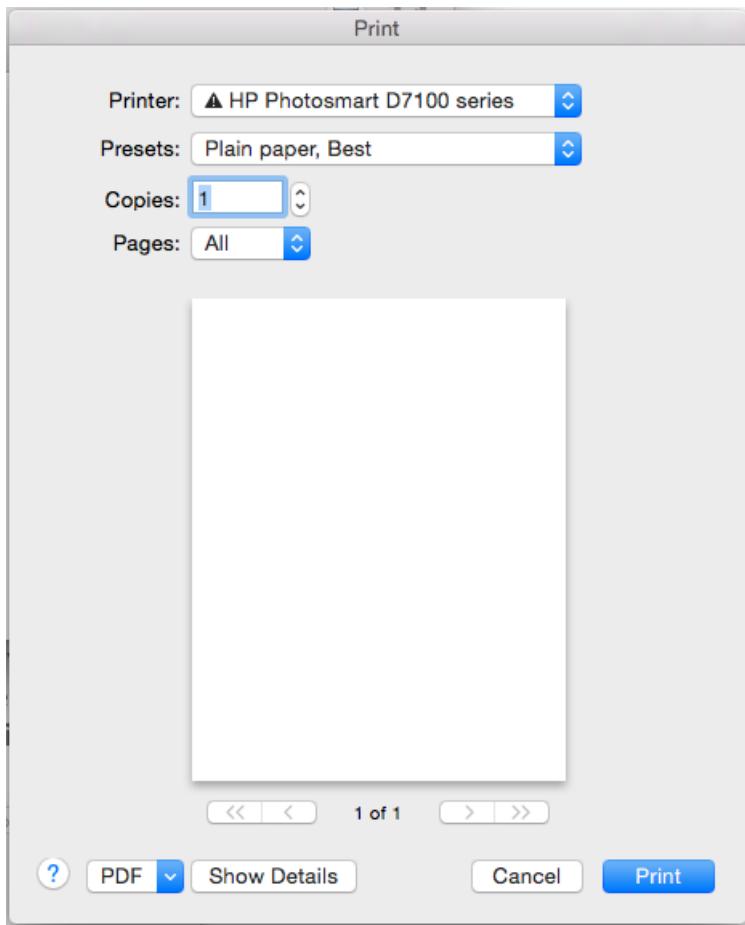
macOS provides standard Print and Page Setup Dialogs that your application can display so that users can have a consistent printing experience in every application they use.

The following code will show the standard Print Dialog:

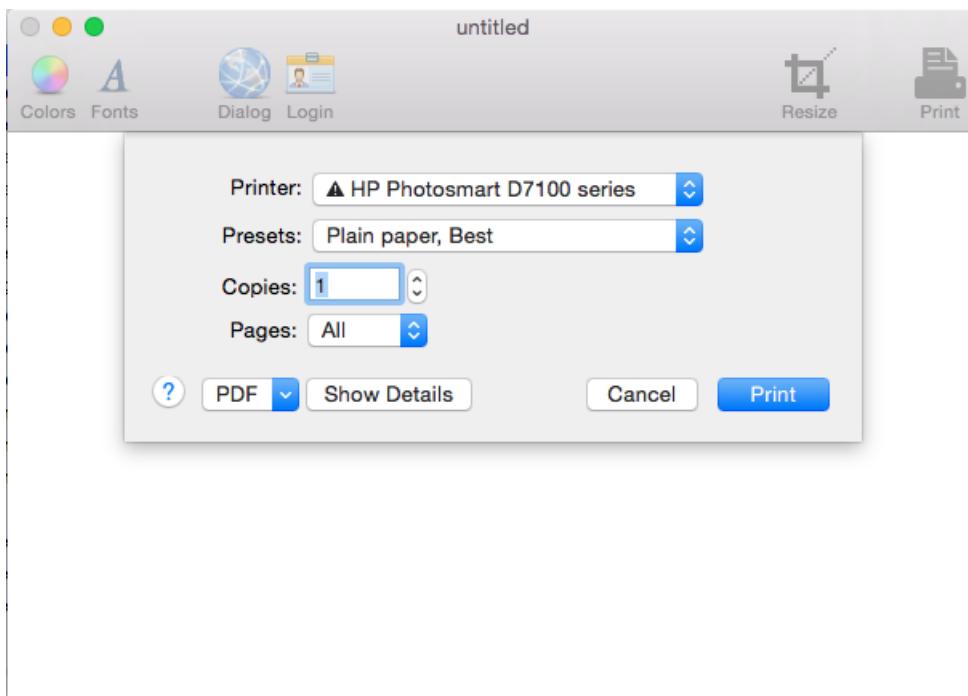
```
public bool ShowPrintAsSheet { get; set;} = true;
...
[Export ("showPrinter")]
void ShowDocument (NSObject sender) {
    var dlg = new NSPrintPanel();

    // Display the print dialog as dialog box
    if (ShowPrintAsSheet) {
        dlg.BeginSheet(new NSPrintInfo(),this,this,null,new IntPtr());
    } else {
        if (dlg.RunModalWithPrintInfo(new NSPrintInfo()) == 1) {
            var alert = new NSAlert () {
                AlertStyle = NSAlertStyle.Critical,
                InformativeText = "We need to print the document here...",
                MessageText = "Print Document",
            };
            alert.RunModal ();
        }
    }
}
```

If we set the `ShowPrintAsSheet` property to `false`, run the application and display the print dialog, the following will be displayed:



If set the `ShowPrintAsSheet` property to `true`, run the application and display the print dialog, the following will be displayed:

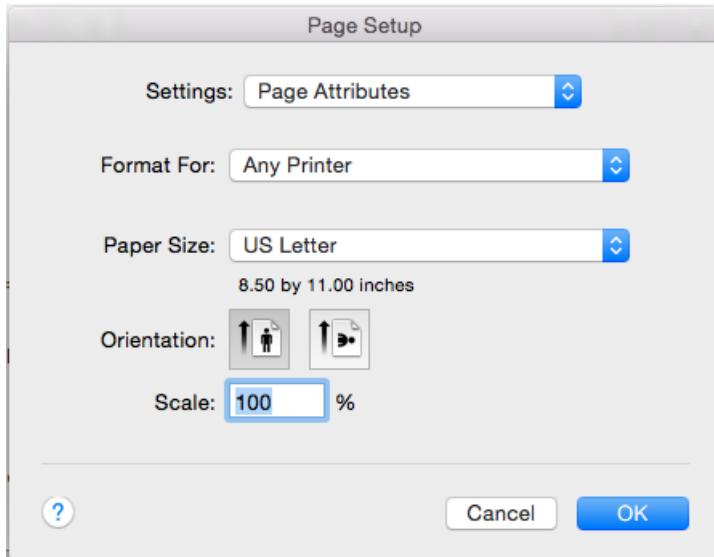


The following code will display the Page Layout Dialog:

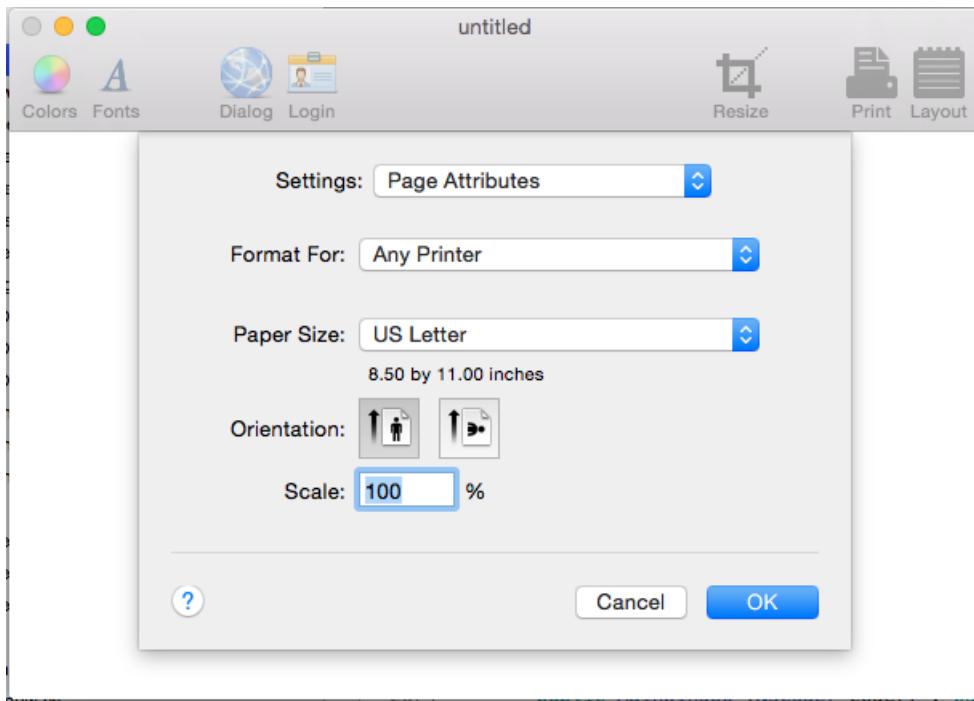
```
[Export ("showLayout")]
void ShowLayout (NSObject sender) {
    var dlg = new NSPageLayout();

    // Display the print dialog as dialog box
    if (ShowPrintAsSheet) {
        dlg.BeginSheet (new NSPrintInfo (), this);
    } else {
        if (dlg.RunModal () == 1) {
            var alert = new NSAlert () {
                AlertStyle = NSAlertStyle.Critical,
                InformativeText = "We need to print the document here...",
                MessageText = "Print Document",
            };
            alert.RunModal ();
        }
    }
}
```

If we set the `ShowPrintAsSheet` property to `false`, run the application and display the print layout dialog, the following will be displayed:



If set the `ShowPrintAsSheet` property to `true`, run the application and display the print layout dialog, the following will be displayed:



For more information about working with the Print and Page Setup Dialogs, please see Apple's [NSPrintPanel](#) and [NSPageLayout](#) documentation.

The Save Dialog

The Save Dialog gives users a consistent way to save an item in an application.

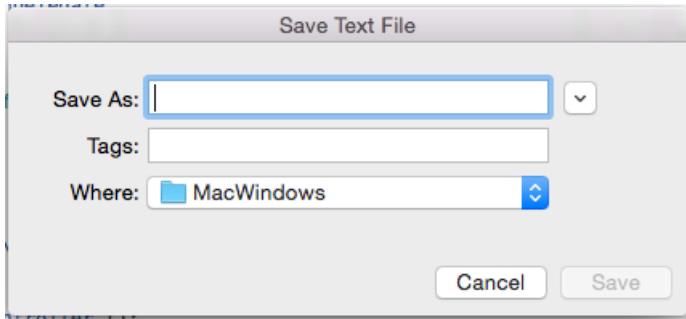
The following code will show the standard Save Dialog:

```
public bool ShowSaveAsSheet { get; set;} = true;
...
[Export("saveDocumentAs")]
void ShowSaveAs (NSObject sender)
{
    var dlg = new NSSavePanel ();
    dlg.Title = "Save Text File";
    dlg.AllowedFileTypes = new string[] { "txt", "html", "md", "css" };

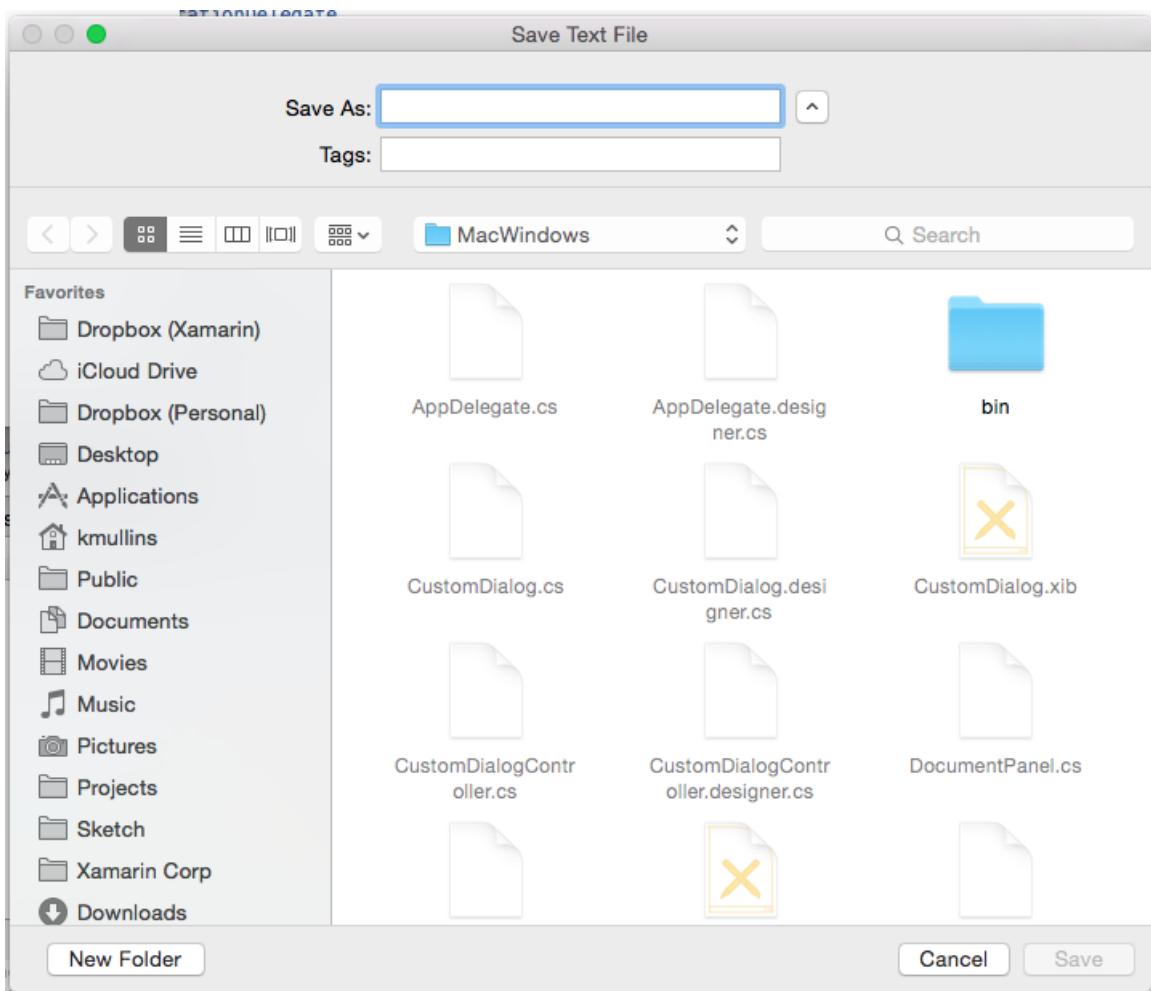
    if (ShowSaveAsSheet) {
        dlg.BeginSheet(mainWindowController.Window,(result) => {
            var alert = new NSAlert () {
                AlertStyle = NSAlertStyle.Critical,
                InformativeText = "We need to save the document here...",
                MessageText = "Save Document",
            };
            alert.RunModal ();
        });
    } else {
        if (dlg.RunModal () == 1) {
            var alert = new NSAlert () {
                AlertStyle = NSAlertStyle.Critical,
                InformativeText = "We need to save the document here...",
                MessageText = "Save Document",
            };
            alert.RunModal ();
        }
    }
}
```

The `AllowedFileTypes` property is a string array of file types that the user can select to save the file as. The file type can be either specified as an extension or *UTI*. The default value is `null`, which allows any file type to be used.

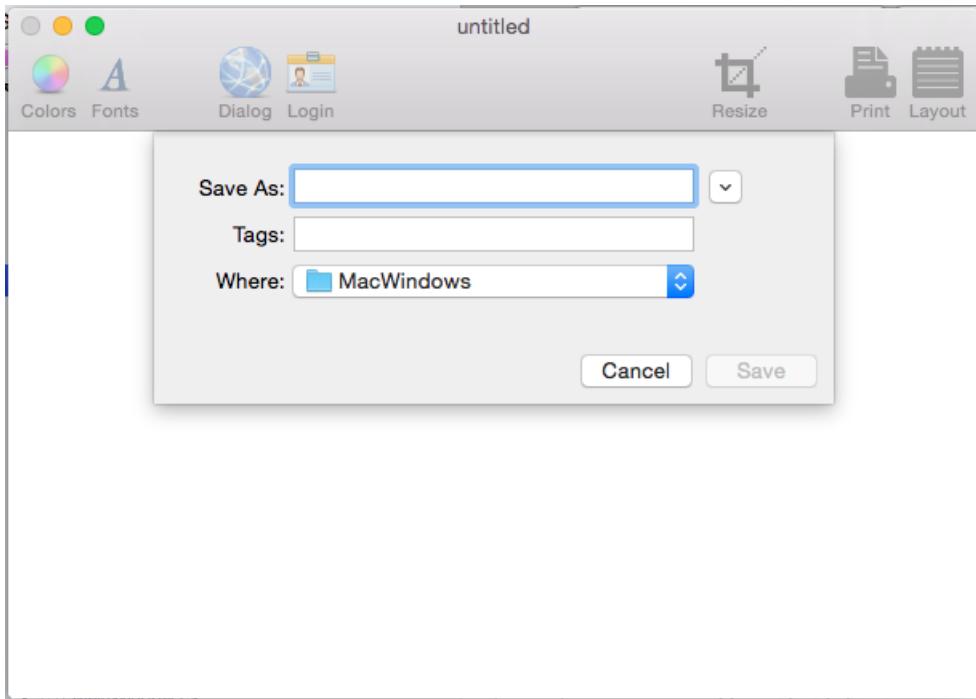
If we set the `ShowSaveAsSheet` property to `false`, run the application and select **Save As...** from the **File** menu, the following will be displayed:



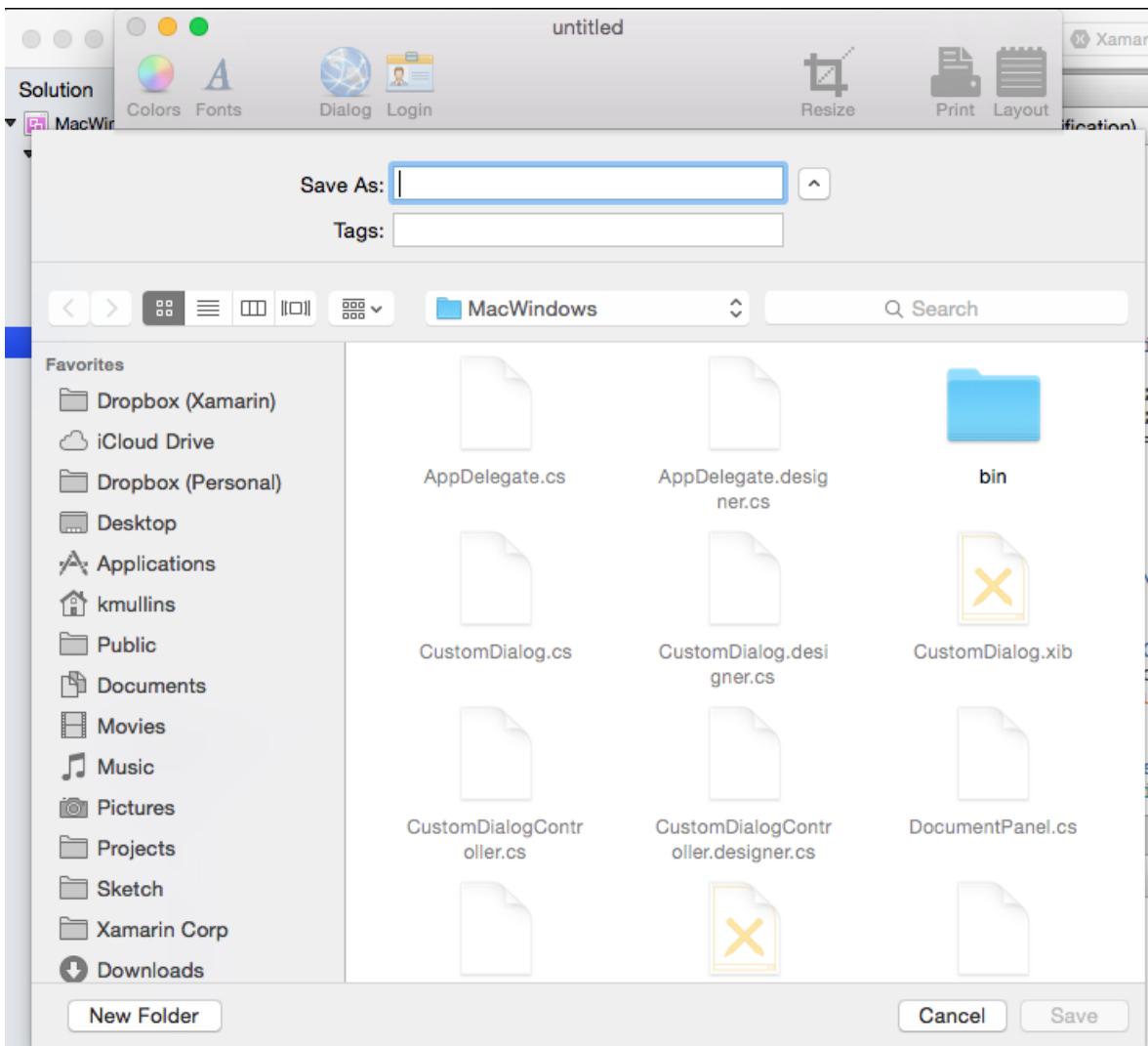
The user can expand the dialog:



If we set the `ShowSaveAsSheet` property to `true`, run the application and select **Save As...** from the **File** menu, the following will be displayed:



The user can expand the dialog:



For more information on working with the Save Dialog, please see Apple's [NSSavePanel](#) documentation.

Summary

This article has taken a detailed look at working with Modal Windows, Sheets and the standard system Dialog Boxes in a Xamarin.Mac application. We saw the different types and uses of Modal Windows, Sheets and Dialogs, how to create and maintain Modal Windows and Sheets in Xcode's Interface Builder and how to work with Modal Windows, Sheets and Dialogs in C# code.

Related Links

- [MacWindows \(sample\)](#)
- [Hello, Mac](#)
- [Menus](#)
- [Windows](#)
- [Toolbars](#)
- [OS X Human Interface Guidelines](#)
- [Introduction to Windows](#)
- [Introduction to Sheets](#)
- [Introduction to Printing](#)

Alerts in Xamarin.Mac

11/2/2020 • 7 minutes to read • [Edit Online](#)

This article covers working with alerts in a Xamarin.Mac application. It describes creating and displaying alerts from C# code and responding to user interactions.

When working with C# and .NET in a Xamarin.Mac application, you have access to the same Alerts that a developer working in *Objective-C* and *Xcode* does.

An alert is a special type of dialog that appears when a serious problem occurs (such as an error) or as a warning (such as preparing to delete a file). Because an alert is a dialog, it also requires a user response before it can be closed.



In this article, we'll cover the basics of working with Alerts in a Xamarin.Mac application.

Introduction to Alerts

An alert is a special type of dialog that appears when a serious problem occurs (such as an error) or as a warning (such as preparing to delete a file). Because Alerts disrupt the user since they must be dismissed before the user can continue on with their task, avoid displaying an alert unless it's absolutely necessary.

Apple suggest the following guidelines:

- Don't use an alert merely to give users information.
- Do not display an alert for common, undoable actions. Even if that situation might cause data loss.
- If a situation is worthy of an alert, avoid using any other UI element or method to display it.
- The Caution icon should be used sparingly.
- Describe the alert situation clearly and succinctly in the alert message.
- The Default Button name should correspond to the action you describe in your alert message.

For more information, see the [Alerts](#) section of Apple's [OS X Human Interface Guidelines](#)

Anatomy of an Alert

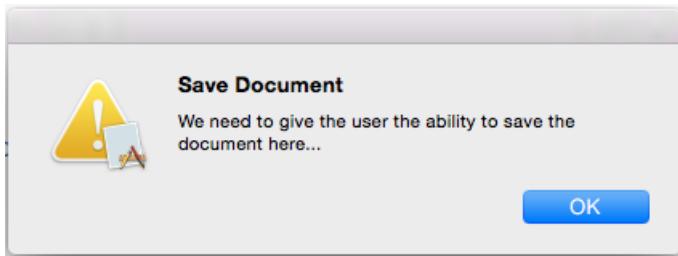
As stated above, alerts should be shown to your application's user when a serious problem occurs or as a warning to potential data loss (such as closing an unsaved file). In Xamarin.Mac, an alert is created in C# code, for example:

```

var alert = new NSAlert () {
    AlertStyle = NSAlertStyle.Critical,
    InformativeText = "We need to save the document here...",
    MessageText = "Save Document",
};
alert.RunModal ();

```

The code above displays an alert with the applications icon superimposed on the warning icon, a title, a warning message and a single OK button:



Apple provides several properties that can be used to customize an alert:

- **AlertStyle** defines the type of an alert as one of the following:
 - **Warning** - Used to warn the user a current or impending event that is not critical. This is the default style.
 - **Informational** - Used to warn the user about a current or impending event. Currently, there is no visible difference between a **Warning** and a **Informational**
 - **Critical** - Used to warn the user about severe consequences of an impending event (such as deleting a file). This type of alert should be used sparingly.
- **MessageText** - This is the main message or title of the alert and should quickly define the situation to the user.
- **InformativeText** - This is the body of the alert where you should define the situation clearly and present workable options to the user.
- **Icon** - Allows a custom icon to be displayed to the user.
- **HelpAnchor & ShowsHelp** - Allows the alert to be tied into the application HelpBook and display help for the alert.
- **Buttons** - By default, an alert only has the **OK** button but the **Buttons** collection allows you to add more choices as needed.
- **ShowsSuppressionButton** - If `true` displays a checkbox that the user can use to suppress the alert for subsequent occurrences of the event that triggered it.
- **AccessoryView** - Allows you to attach another subview to the alert to provide extra information, such as adding a **Text Field** for data entry. If you set a new **AccessoryView** or modify an existing one, you need to call the `Layout()` method to adjust the visible layout of the alert.

Displaying an Alert

There are two different ways that an alert can be displayed, Free-Floating or as a Sheet. The following code displays an alert as free-floating:

```

var alert = new NSAlert () {
    AlertStyle = NSAlertStyle.Informational,
    InformativeText = "This is the body of the alert where you describe the situation and any actions to correct it.",
    MessageText = "Alert Title",
};
alert.RunModal ();

```

If this code is run, the following is displayed:



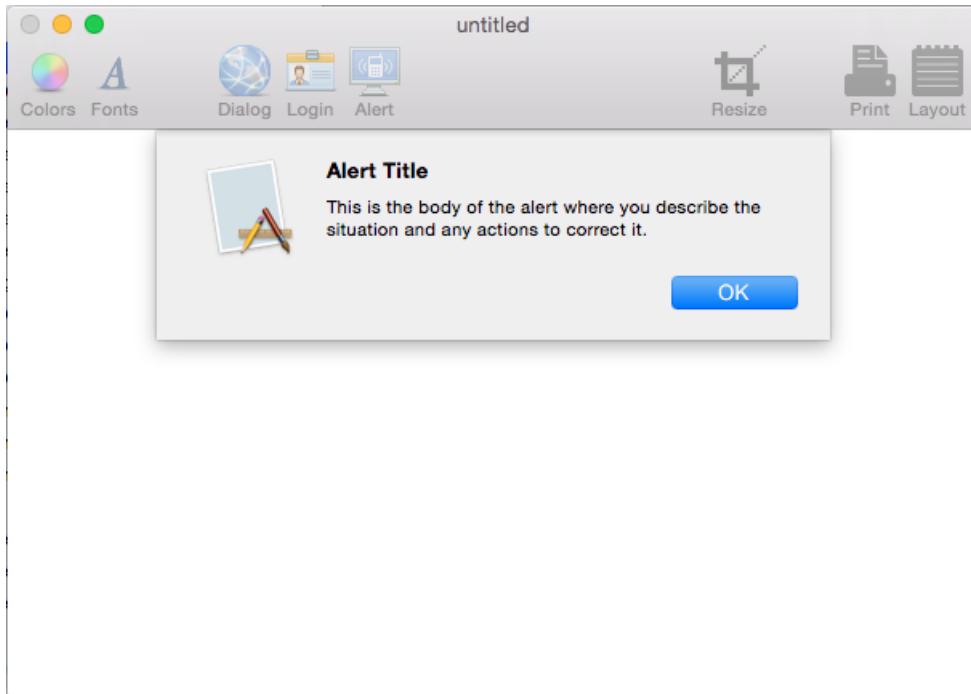
The following code displays the same alert as a Sheet:

```

var alert = new NSAlert () {
    AlertStyle = NSAlertStyle.Informational,
    InformativeText = "This is the body of the alert where you describe the situation and any actions to correct it.",
    MessageText = "Alert Title",
};
alert.BeginSheet (this);

```

If this code is run, the following will be displayed:



Working with Alert Buttons

By default, an Alert displays only the **OK** button. However, you are not limited to that, you can create extra buttons by appending them to the **Buttons** collection. The following code creates a free-floating alert with a **OK**, **Cancel** and **Maybe** button:

```

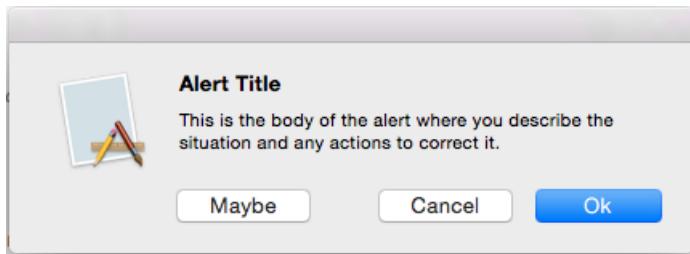
var alert = new NSAlert () {
    AlertStyle = NSAlertStyle.Informational,
    InformativeText = "This is the body of the alert where you describe the situation and any actions to
correct it.",
    MessageText = "Alert Title",
};
alert.AddButton ("Ok");
alert.AddButton ("Cancel");
alert.AddButton ("Maybe");
var result = alert.RunModal ();

```

The very first button added will be the *Default Button* that will be activated if the user presses the Enter key. The returned value will be an integer representing which button the user pressed. In our case the following values will be returned:

- **OK** - 1000.
- **Cancel** - 1001.
- **Maybe** - 1002.

If we run the code , the following will be displayed:



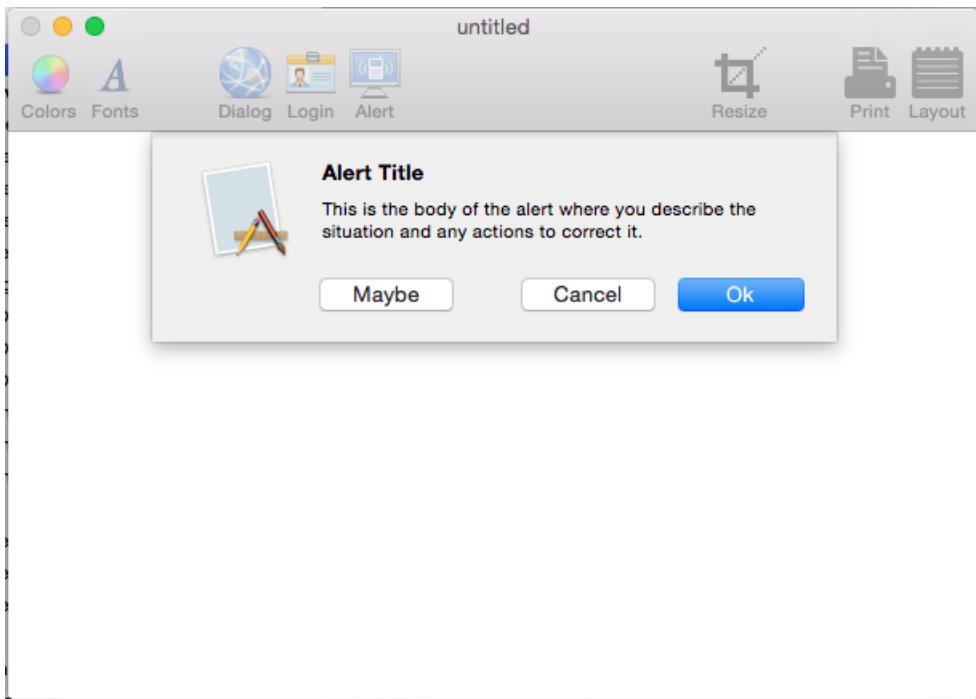
Here is the code for the same alert as a Sheet:

```

var alert = new NSAlert () {
    AlertStyle = NSAlertStyle.Informational,
    InformativeText = "This is the body of the alert where you describe the situation and any actions to
correct it.",
    MessageText = "Alert Title",
};
alert.AddButton ("Ok");
alert.AddButton ("Cancel");
alert.AddButton ("Maybe");
alert.BeginSheetForResponse (this, (result) => {
    Console.WriteLine ("Alert Result: {0}", result);
});

```

If this code is run, the following will be displayed:



IMPORTANT

You should never add more than three buttons to an alert.

Showing the Suppress Button

If the Alert's `ShowSuppressButton` property is `true`, the alert displays a checkbox that the user can use to suppress the alert for subsequent occurrences of the event that triggered it. The following code displays a free-floating alert with a suppress button:

```
var alert = new NSAlert () {
    AlertStyle = NSAlertStyle.Informational,
    InformativeText = "This is the body of the alert where you describe the situation and any actions to correct it.",
    MessageText = "Alert Title",
};
alert.AddButton ("Ok");
alert.AddButton ("Cancel");
alert.AddButton ("Maybe");
alert.ShowsSuppressionButton = true;
var result = alert.RunModal ();
Console.WriteLine ("Alert Result: {0}, Suppress: {1}", result, alert.SuppressionButton.State ==
NSCellStateValue.On);
```

If the value of the `alert.SuppressionButton.State` is `NSCellStateValue.On`, the user has checked the Suppress checkbox, else they have not.

If the code is run, the following will be displayed:

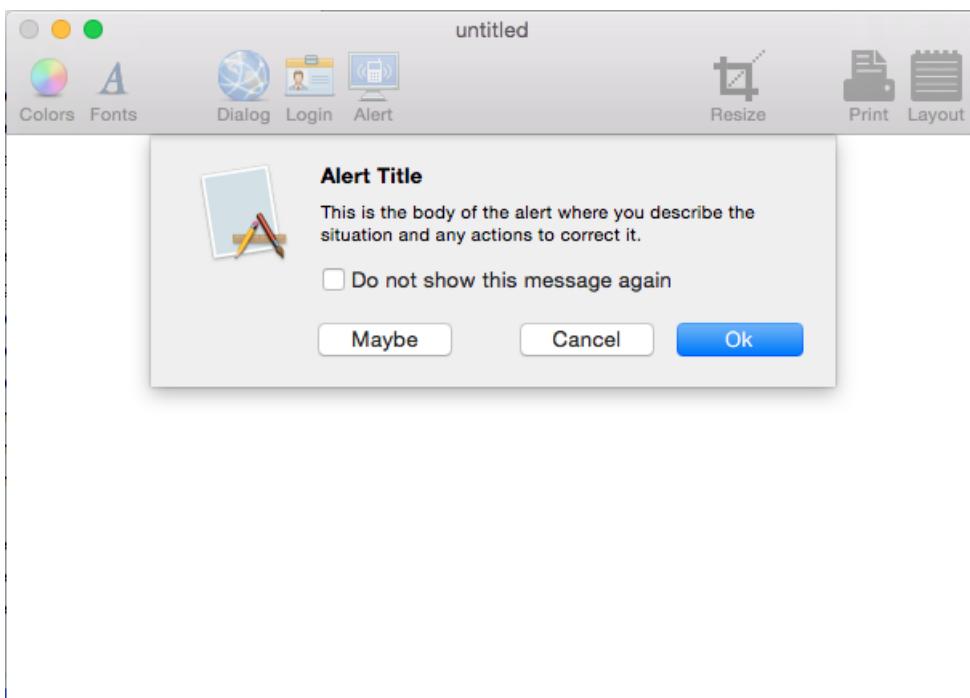


Here is the code for the same alert as a Sheet:

```
var alert = new NSAlert () {
    AlertStyle = NSAlertStyle.Informational,
    InformativeText = "This is the body of the alert where you describe the situation and any actions to correct it.",
    MessageText = "Alert Title",
};

alert.AddButton ("Ok");
alert.AddButton ("Cancel");
alert.AddButton ("Maybe");
alert.ShowsSuppressionButton = true;
alert.BeginSheetForResponse (this, (result) => {
    Console.WriteLine ("Alert Result: {0}, Suppress: {1}", result, alert.SuppressionButton.State ==
NSCellStateValue.On);
});
```

If this code is run, the following will be displayed:



Adding a Custom SubView

Alerts have an `AccessoryView` property that can be used to customize the alert further and add things like a **Text Field** for user input. The following code creates a free-floating alert with an added text input field:

```

var input = new NSTextField (new CGRect (0, 0, 300, 20));

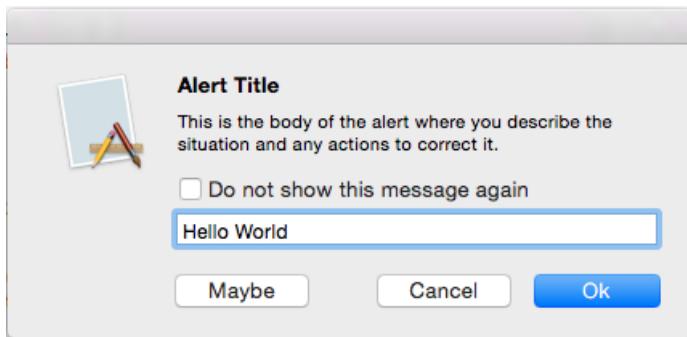
var alert = new NSAlert () {
    AlertStyle = NSAlertStyle.Informational,
    InformativeText = "This is the body of the alert where you describe the situation and any actions to correct it.",
    MessageText = "Alert Title",
};

alert.AddButton ("Ok");
alert.AddButton ("Cancel");
alert.AddButton ("Maybe");
alert.ShowsSuppressionButton = true;
alert.AccessoryView = input;
alert.Layout ();
var result = alert.RunModal ();
Console.WriteLine ("Alert Result: {0}, Suppress: {1}", result, alert.SuppressionButton.State ==
NSCellStateValue.On);

```

The key lines here are `var input = new NSTextField (new CGRect (0, 0, 300, 20));` which creates a new **Text Field** that we will be adding the alert. `alert.AccessoryView = input;` which attaches the **Text Field** to the alert and the call to the `Layout()` method, which is required to resize the alert to fit in the new subview.

If we run the code, the following will be displayed:



Here is the same alert as a sheet:

```

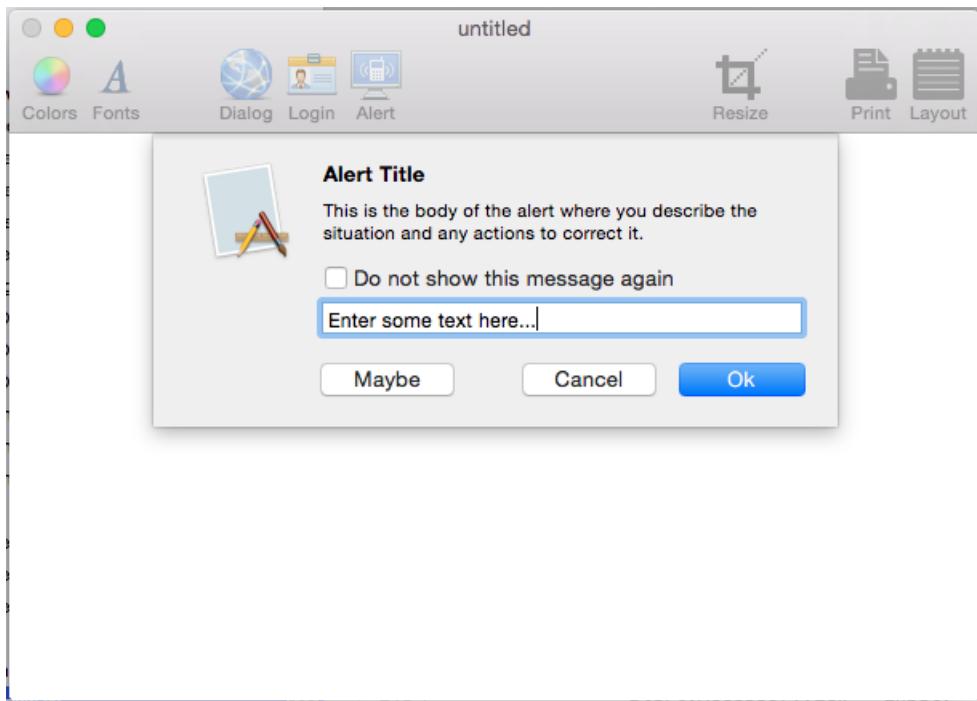
var input = new NSTextField (new CGRect (0, 0, 300, 20));

var alert = new NSAlert () {
    AlertStyle = NSAlertStyle.Informational,
    InformativeText = "This is the body of the alert where you describe the situation and any actions to correct it.",
    MessageText = "Alert Title",
};

alert.AddButton ("Ok");
alert.AddButton ("Cancel");
alert.AddButton ("Maybe");
alert.ShowsSuppressionButton = true;
alert.AccessoryView = input;
alert.Layout ();
alert.BeginSheetForResponse (this, (result) => {
    Console.WriteLine ("Alert Result: {0}, Suppress: {1}", result, alert.SuppressionButton.State ==
NSCellStateValue.On);
});

```

If we run this code, the following will be displayed:



Summary

This article has taken a detailed look at working with Alerts in a Xamarin.Mac application. We saw the different types and uses of Alerts, how to create and customize Alerts and how to work with Alerts in C# code.

Related Links

- [MacWindows \(sample\)](#)
- [Hello, Mac](#)
- [Working with Windows](#)
- [OS X Human Interface Guidelines](#)
- [Introduction to Windows](#)
- [NSAlert](#)

Menus in Xamarin.Mac

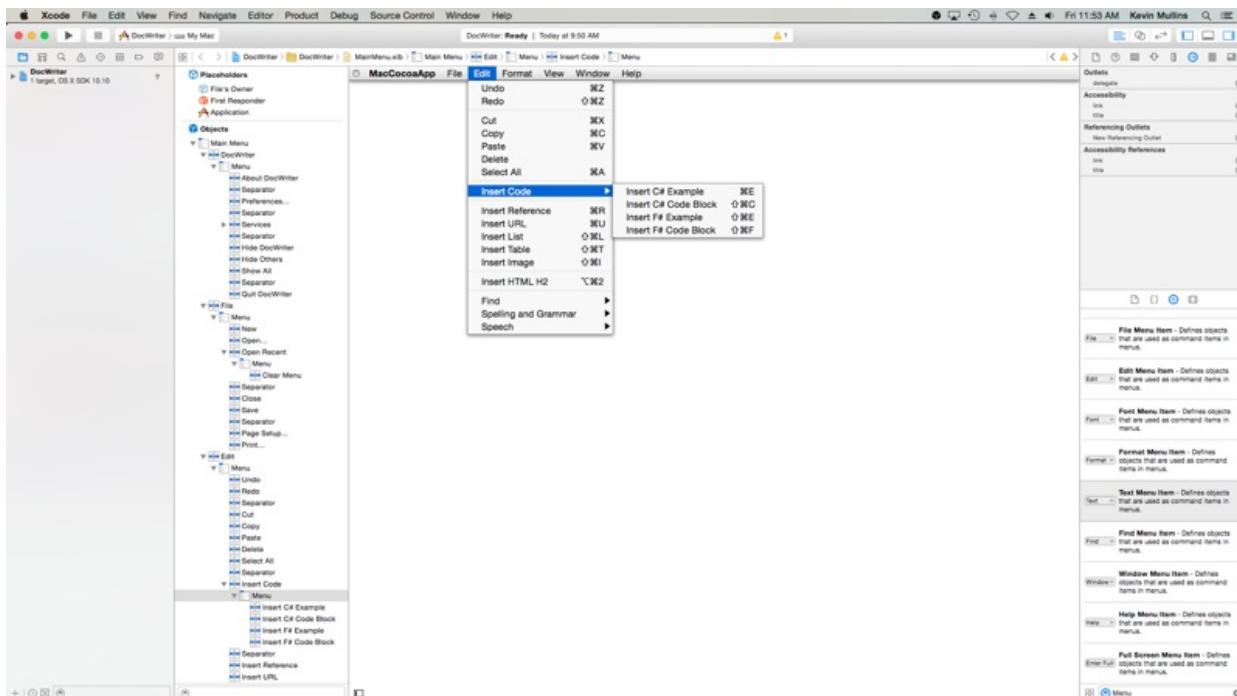
3/5/2021 • 30 minutes to read • [Edit Online](#)

This article covers working with menus in a Xamarin.Mac application. It describes creating and maintaining menus and menu items in Xcode and Interface Builder and working with them programmatically.

When working with C# and .NET in a Xamarin.Mac application, you have access to the same Cocoa menus that a developer working in Objective-C and Xcode does. Because Xamarin.Mac integrates directly with Xcode, you can use Xcode's Interface Builder to create and maintain your menu bars, menus, and menu items (or optionally create them directly in C# code).

Menus are an integral part of a Mac application's user experience and commonly appear in various parts of the user interface:

- **The application's menu bar** - This is the main menu that appears at the top of the screen for every Mac application.
 - **Contextual menus** - These appear when the user right-clicks or control-clicks an item in a window.
 - **The status bar** - This is the area at the far right side of the application menu bar that appears at the top of the screen (to the left of the menu bar clock) and grows to the left as items are added to it.
 - **Dock menu** - The menu for each application in the dock that appears when the user right-clicks or control-clicks the application's icon, or when the user left-clicks the icon and holds the mouse button down.
 - **Pop-up button and pull-down lists** - A pop-up button displays a selected item and presents a list of options to select from when clicked by the user. A pull-down list is a type of pop-up button usually used for selecting commands specific to the context of the current task. Both can appear anywhere in a window.



In this article, we'll cover the basics of working with Cocoa menu bars, menus, and menu items in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` attributes used to wire-up your C# classes to

Objective-C objects and UI elements.

The application's menu bar

Unlike applications running on the Windows OS where every window can have its own menu bar attached to it, every application running on macOS has a single menu bar that runs along the top of the screen that's used for every window in that application:



Items on this menu bar are activated or deactivated based on the current context or state of the application and its user interface at any given moment. For example: if the user selects a text field, items on the **Edit** menu will become enabled such as **Copy** and **Cut**.

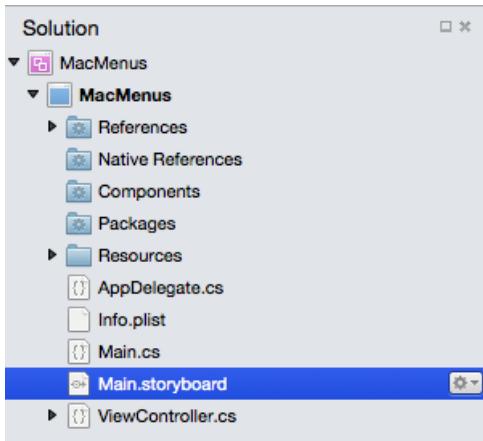
According to Apple and by default, all macOS applications have a standard set of menus and menu items that appear in the application's menu bar:

- **Apple menu** - This menu provides access to system wide items that are available to the user at all times, regardless of what application is running. These items cannot be modified by the developer.
- **App menu** - This menu displays the application's name in bold and helps the user identify what application is currently running. It contains items that apply to the application as a whole and not a given document or process such as quitting the application.
- **File menu** - Items used to create, open, or save documents that your application works with. If your application is not document-based, this menu can be renamed or removed.
- **Edit menu** - Holds commands such as **Cut**, **Copy**, and **Paste** which are used to edit or modify elements in the application's user interface.
- **Format menu** - If the application works with text, this menu holds commands to adjust the formatting of that text.
- **View menu** - Holds commands that affect how content is displayed (viewed) in the application's user interface.
- **Application-specific menus** - These are any menus that are specific to your application (such as a bookmarks menu for a web browser). They should appear between the **View** and **Window** menus on the bar.
- **Window menu** - Contains commands for working with windows in your application, as well as a list of current open windows.
- **Help menu** - If your application provides onscreen help, the Help menu should be the right-most menu on the bar.

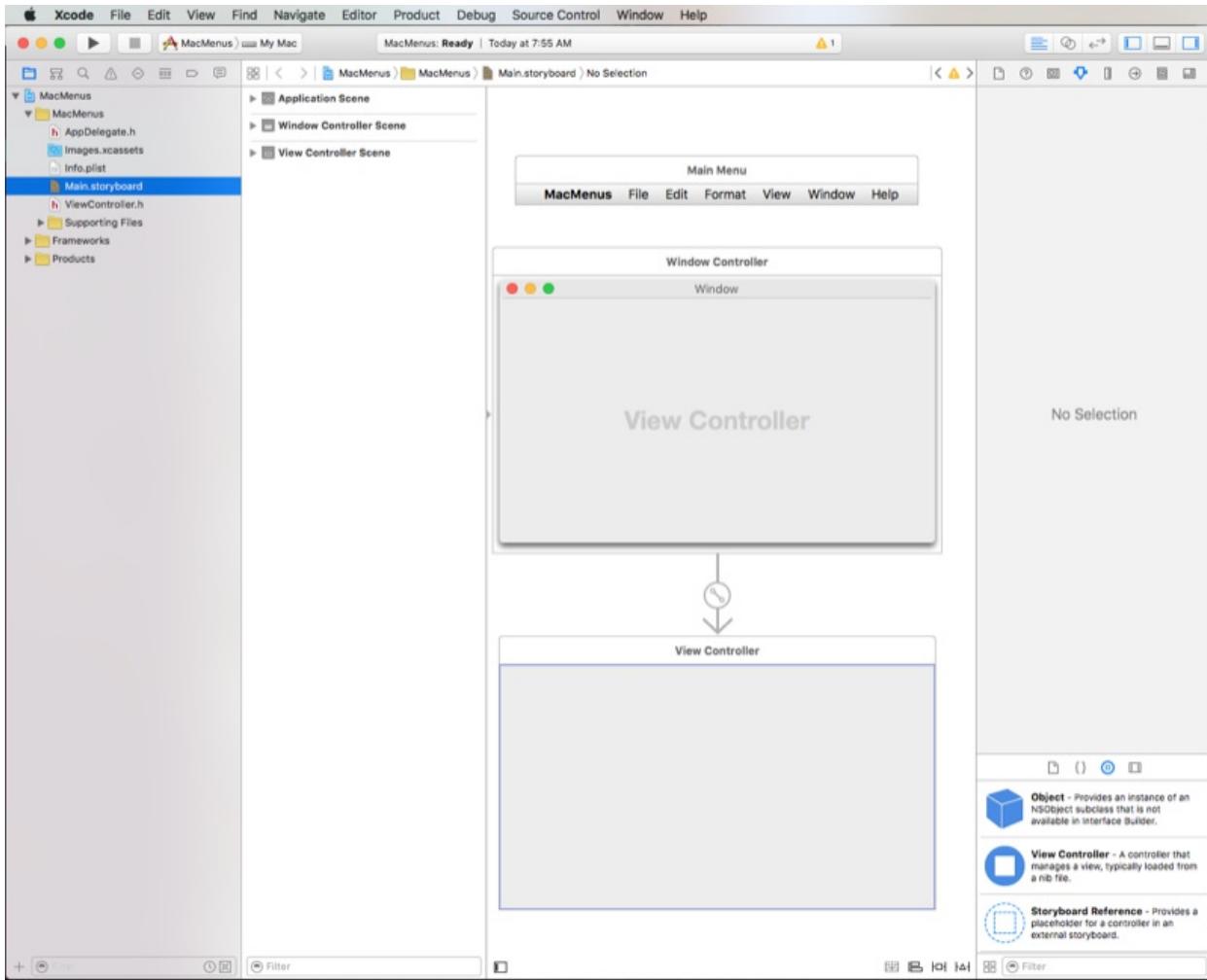
For more information about the application menu bar and standard menus and menu items, please see Apple's [Human Interface Guidelines](#).

The default application menu bar

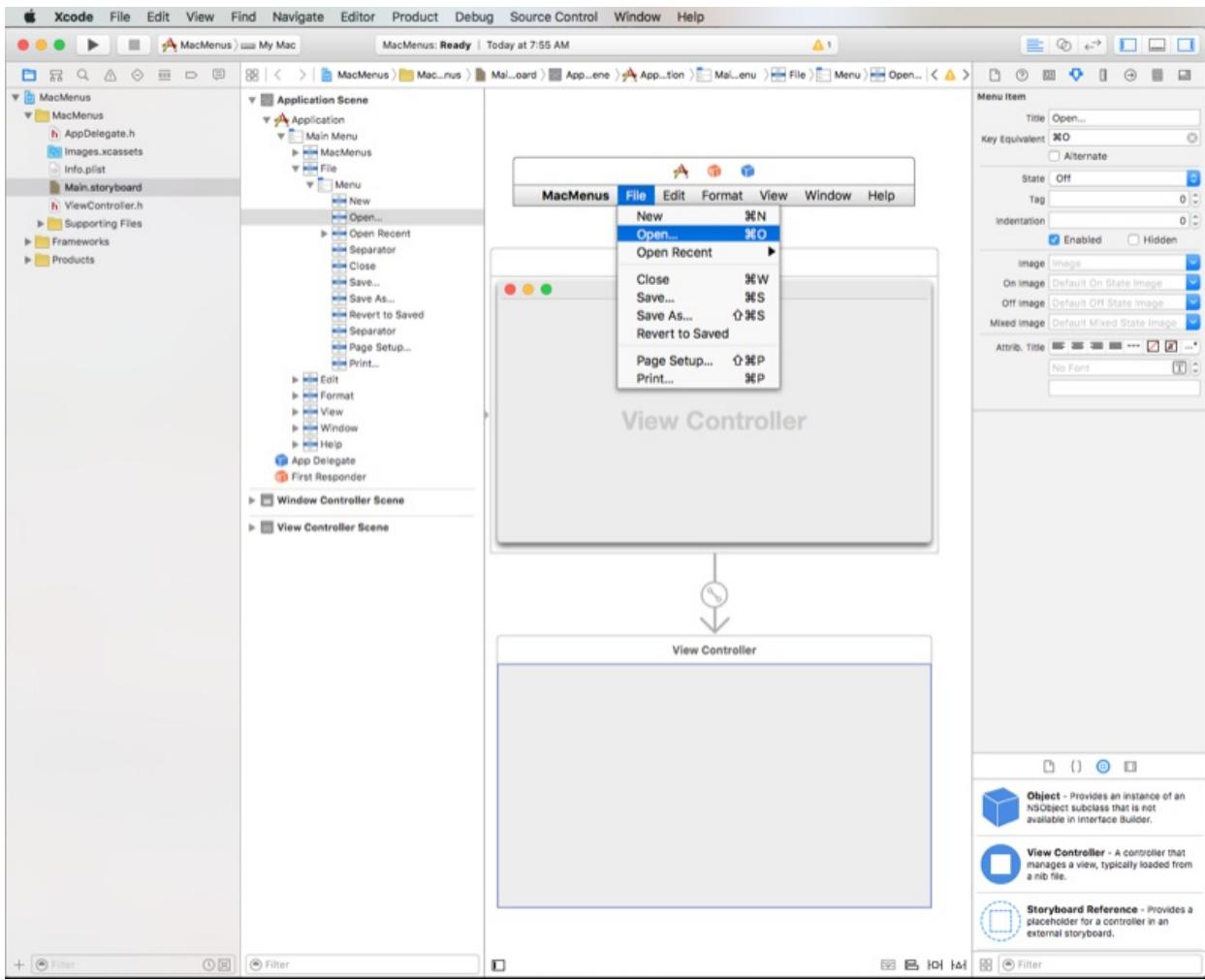
Whenever you create a new Xamarin.Mac project, you automatically get a standard, default application menu bar that has the typical items that a macOS application would normally have (as discussed in the section above). Your application's default menu bar is defined in the **Main.storyboard** file (along with the rest of your app's UI) under the project in the **Solution Pad**:



Double-click the **Main.storyboard** file to open it for editing in Xcode's Interface Builder and you'll be presented with the menu editor interface:

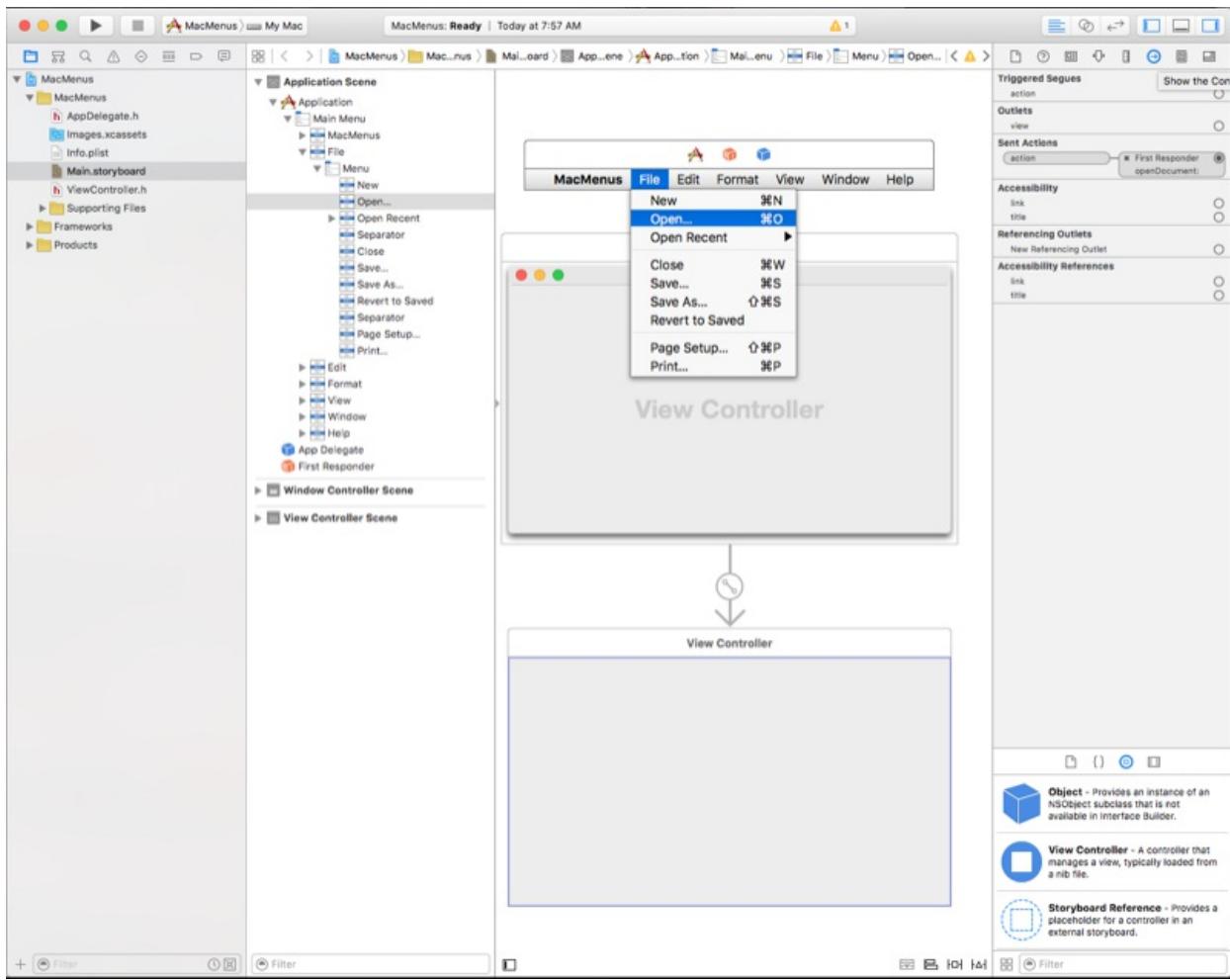


From here we can click on items such as the **Open** menu item in the **File** menu and edit or adjust its properties in the **Attributes Inspector**:

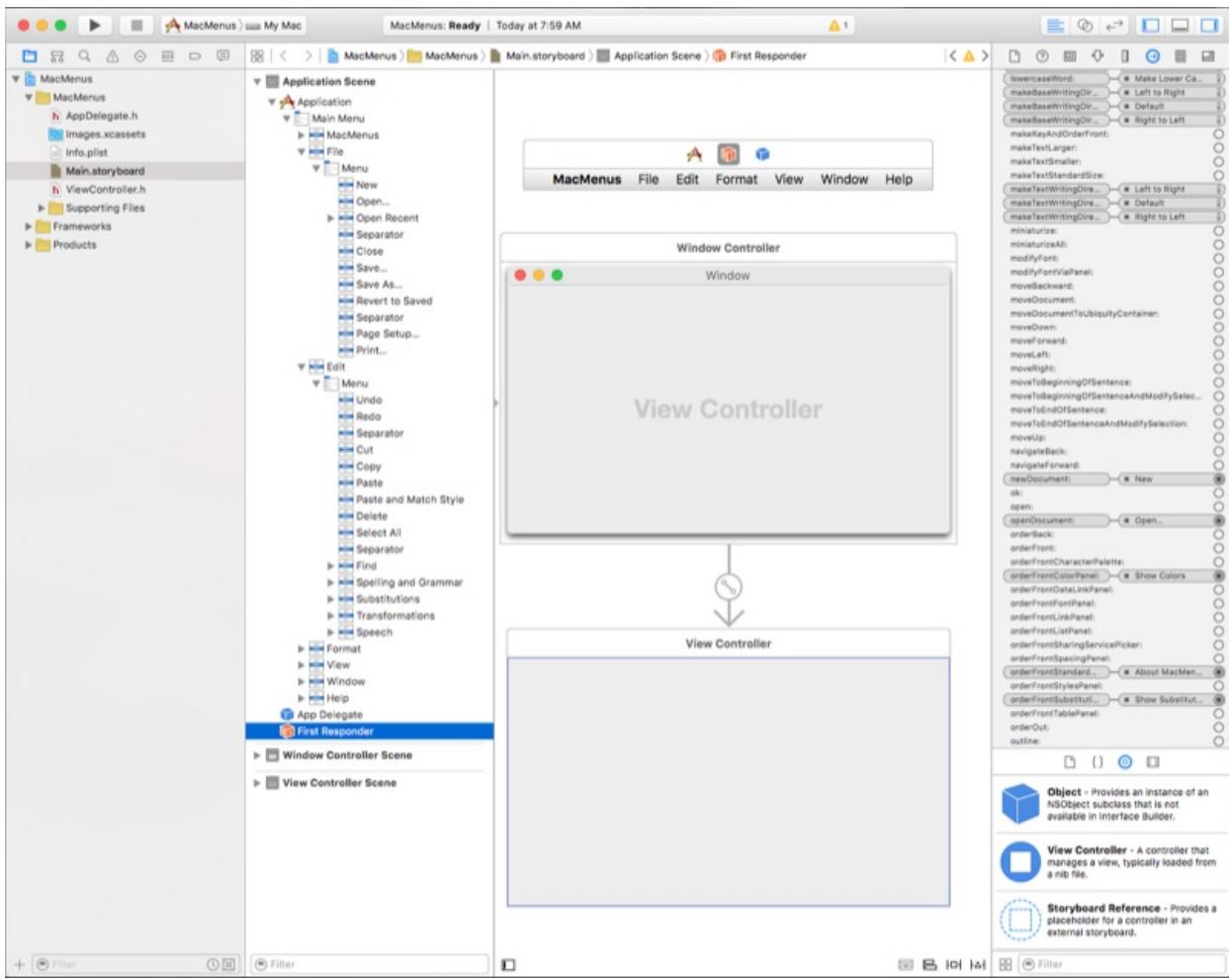


We'll get into adding, editing, and deleting menus and items later in this article. For now we just want to see what menus and menu items are available by default and how they have been automatically exposed to code via a set of predefined outlets and actions (for more information see our [Outlets and Actions](#) documentation).

For example, if we click on the **Connection Inspector** for the **Open** menu item we can see it is automatically wired up to the `openDocument:` action:



If you select the **First Responder** in the **Interface Hierarchy** and scroll down in the **Connection Inspector**, and you will see the definition of the `openDocument:` action that the **Open** menu item is attached to (along with several other default actions for the application that are and are not automatically wired up to controls):

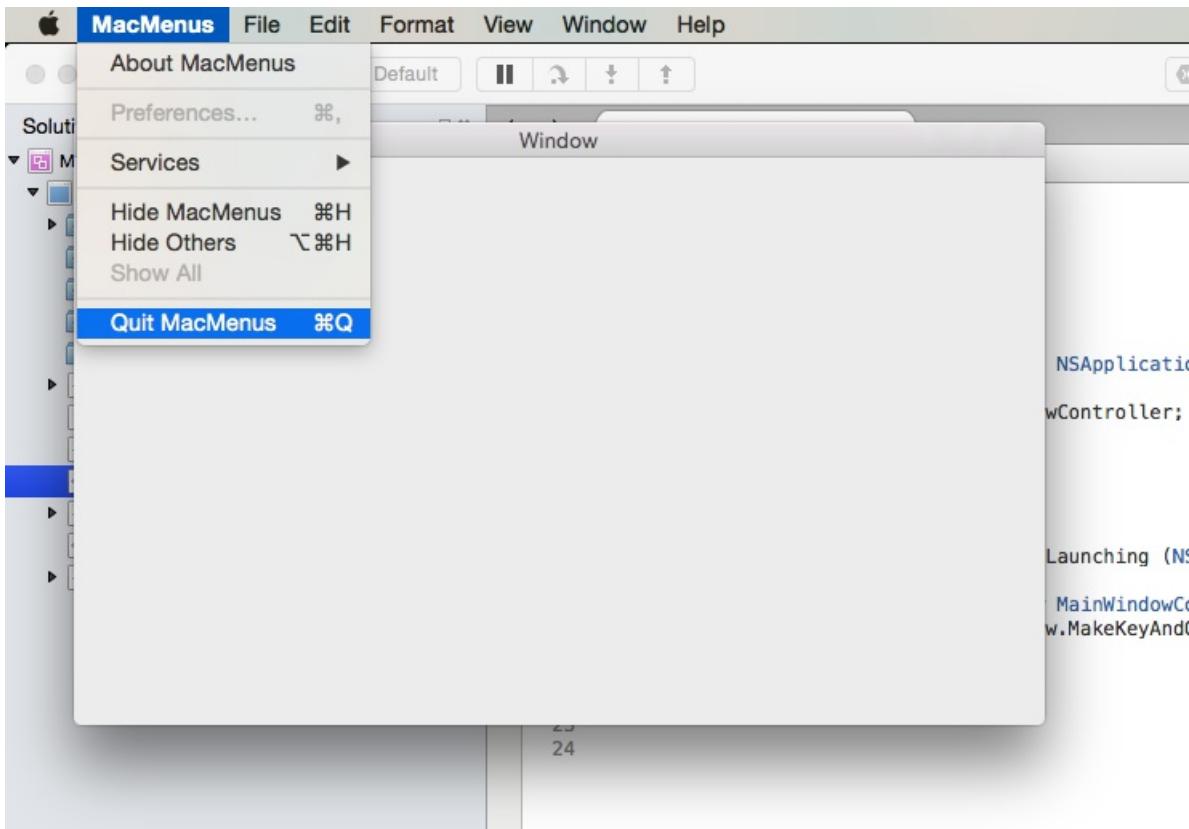


Why is this important? In the next section will see how these automatically-defined actions work with other Cocoa user interface elements to automatically enable and disable menu items, as well as, provide built-in functionality for the items.

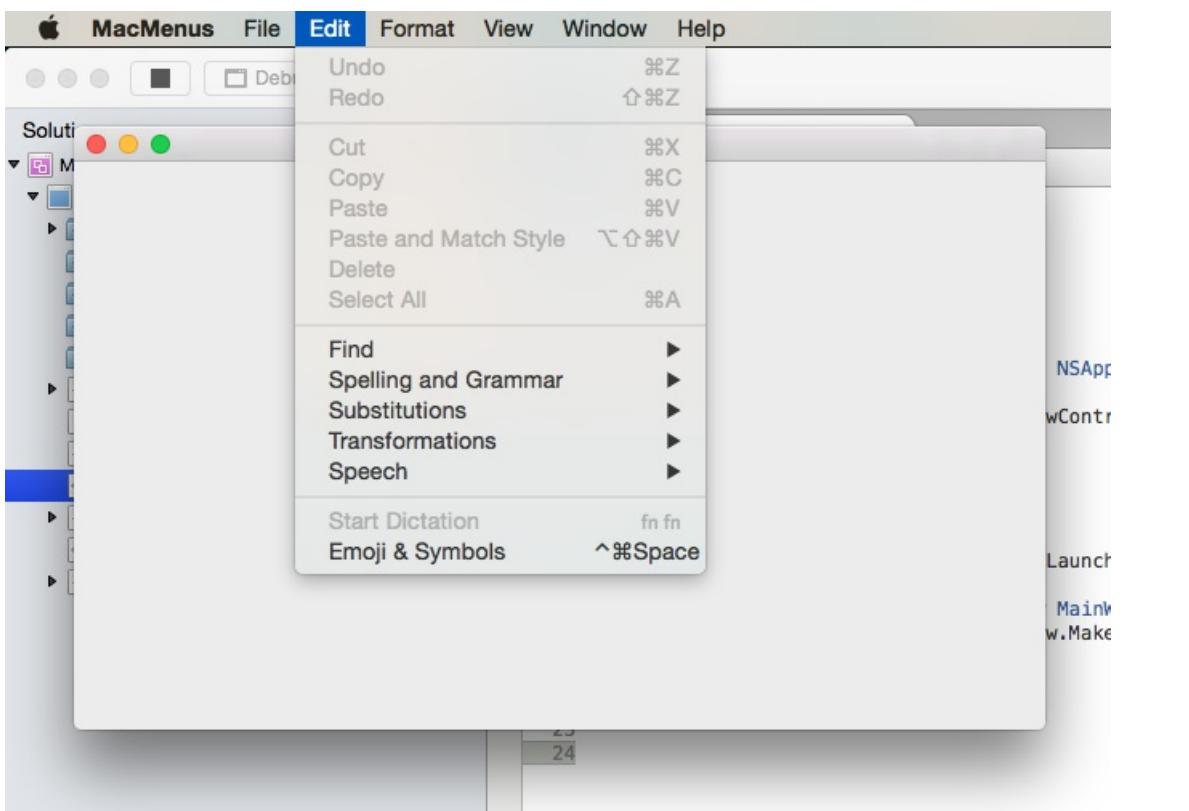
Later we'll be using these built-in actions to enable and disable items from code and provide our own functionality when they are selected.

Built-in menu functionality

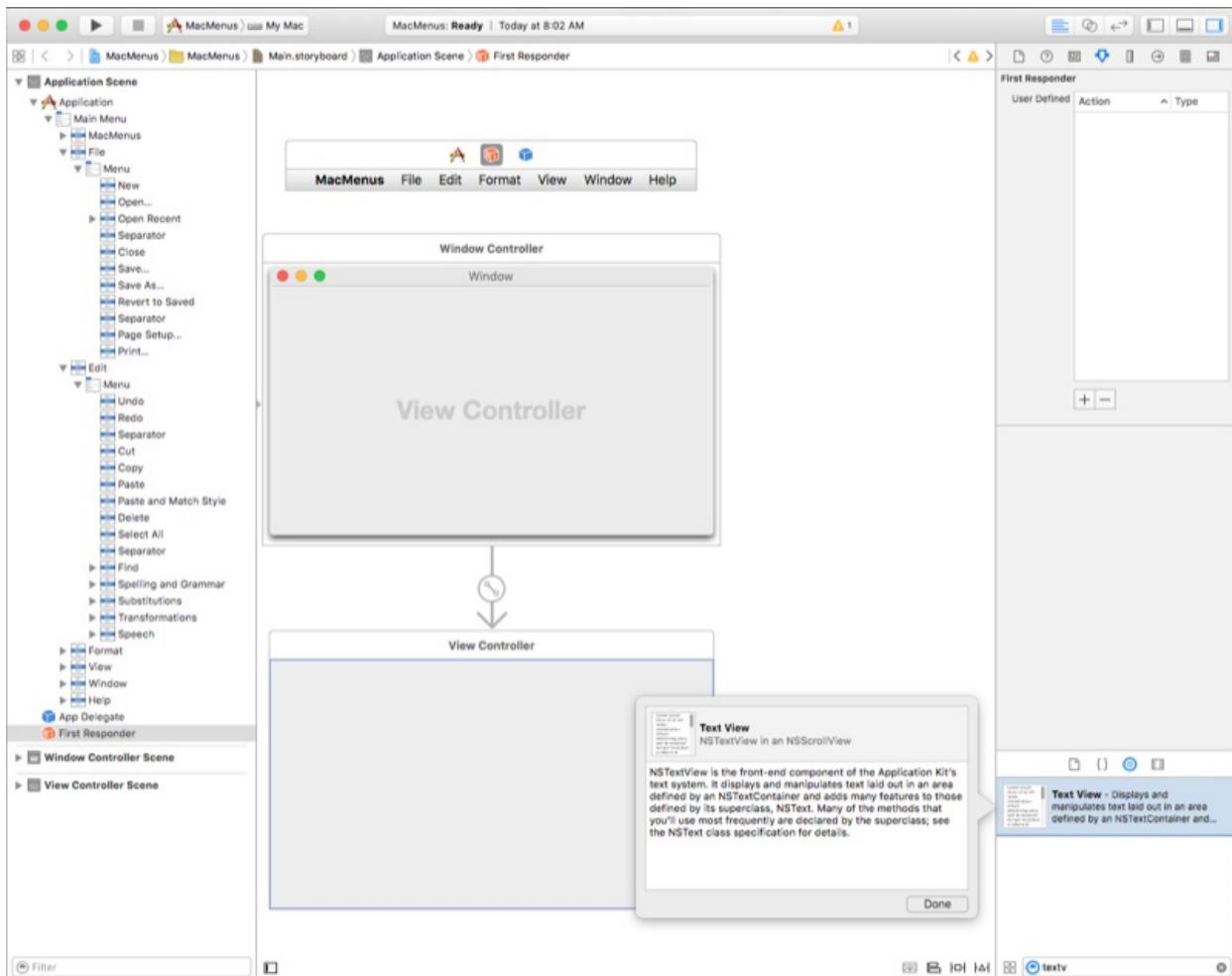
If you were to run a newly created Xamarin.Mac application before adding any UI items or code, you'll notice that some items are automatically wired-up and enabled for you (with fully functionality automatically built-in), such as the **Quit** item in the **App** menu:



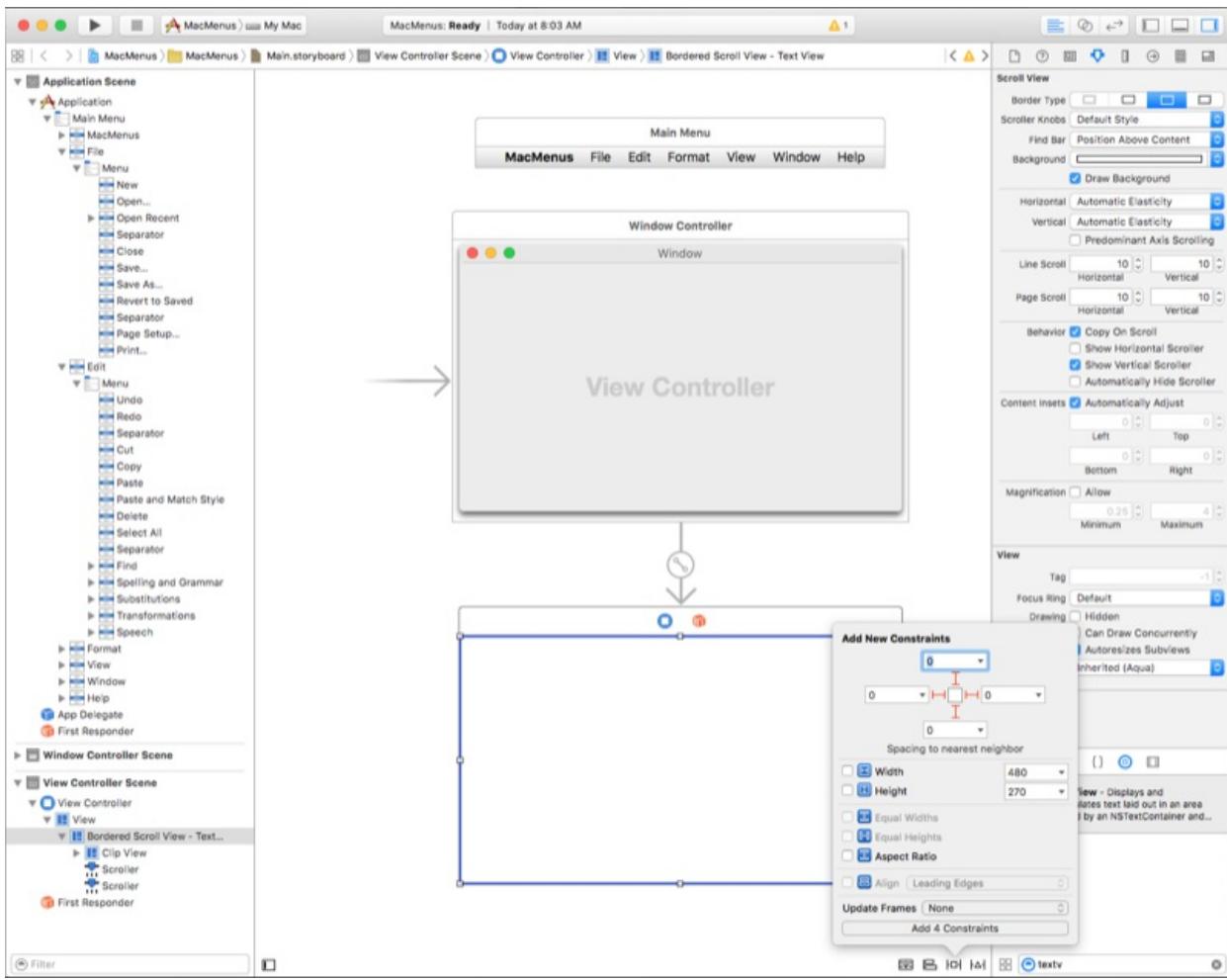
While other menu items, such as Cut, Copy, and Paste are not:



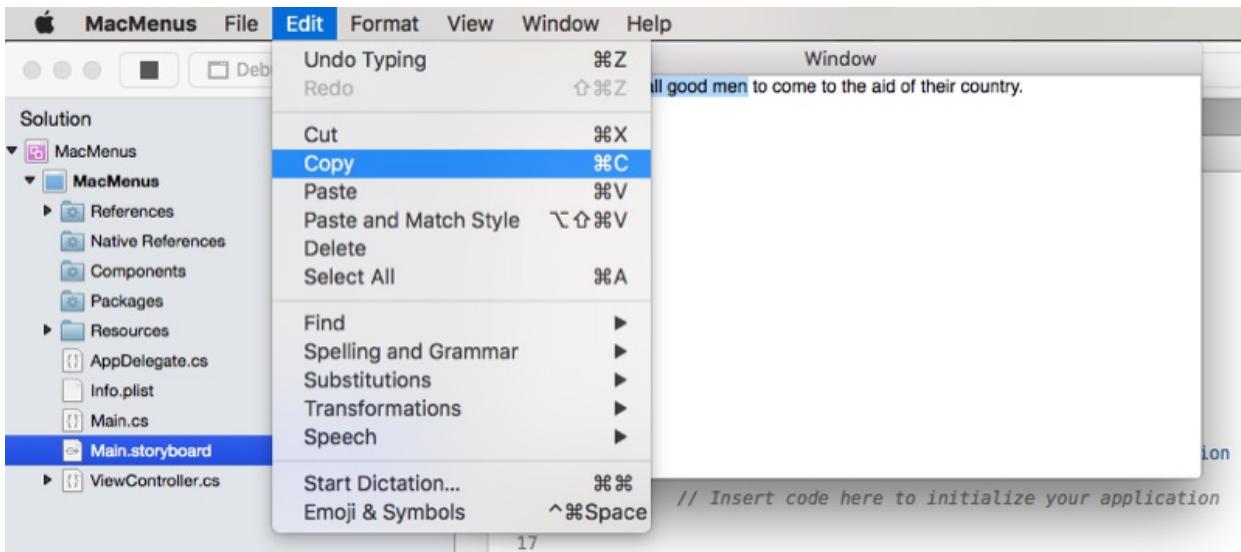
Let's stop the application and double-click the **Main.storyboard** file in the Solution Pad to open it for editing in Xcode's Interface Builder. Next, drag a **Text View** from the Library onto the window's view controller in the Interface Editor:



In the **Constraint Editor** let's pin the text view to the window's edges and set it where it grows and shrinks with the window by clicking all four red I-beams at the top of the editor and clicking the **Add 4 Constraints** button:



Save your changes to the user interface design and switch back the Visual Studio for Mac to synchronize the changes with your Xamarin.Mac project. Now start the application, type some text into the text view, select it, and open the **Edit** menu:



Notice how the **Cut**, **Copy**, and **Paste** items are automatically enabled and fully functional, all without writing a single line of code.

What's going on here? Remember the built-in predefined actions that come wired up to the default menu items (as presented above), most of the Cocoa user interface elements that are part of macOS have built in hooks to specific actions (such as `copy:`). So when they are added to a window, active, and selected, the corresponding menu item or items attached to that action are automatically enabled. If the user selects that menu item, the functionality built into the UI element is called and executed, all without developer intervention.

Enabling and disabling menus and items

By default, every time a user event occurs, `NSMenu` automatically enables and disables each visible menu and menu item based on the context of the application. There are three ways to enable/disable an item:

- **Automatic menu enabling** - A menu item is enabled if `NSMenu` can find an appropriate object that responds to the action that the item is wired-up to. For example, the text view above that had a built-in hook to the `copy:` action.
- **Custom actions and validateMenuItem:** - For any menu item that is bound to a [window or view controller custom action](#), you can add the `validateMenuItem:` action and manually enable or disable menu items.
- **Manual menu enabling** - You manually set the `Enabled` property of each `NSMenuItem` to enable or disable each item in a menu individually.

To choose a system, set the `AutoEnablesItems` property of a `NSMenu`. `true` is automatic (the default behavior) and `false` is manual.

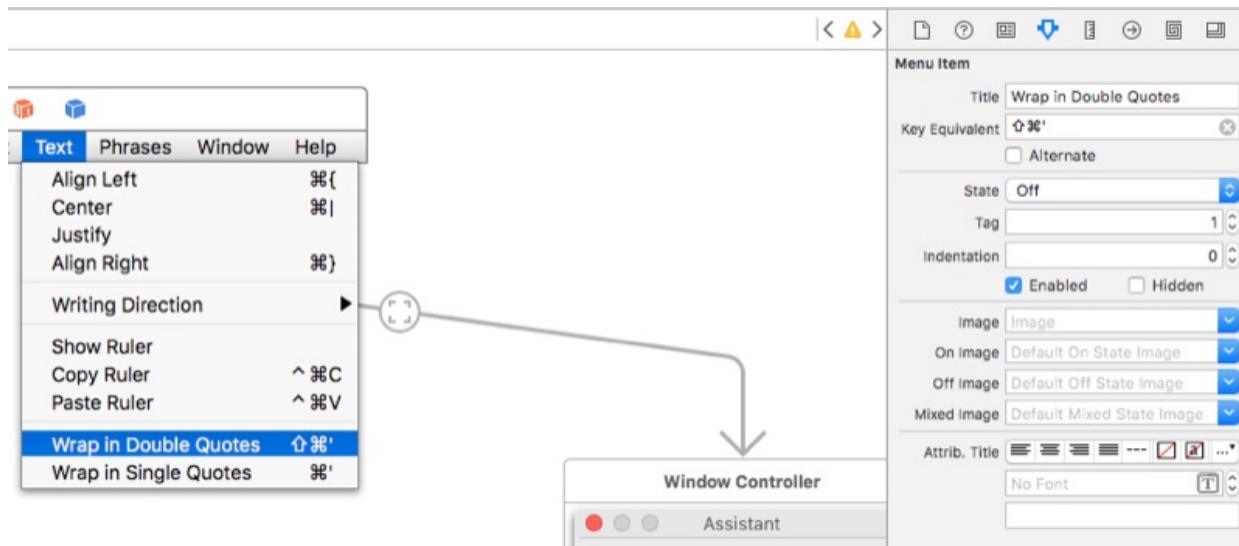
IMPORTANT

If you choose to use manual menu enabling, none of the menu items, even those controlled by AppKit classes like `NSTextView`, are updated automatically. You will be responsible for enabling and disabling all items by hand in code.

Using validateMenuItem

As stated above, for any menu item that is bound to a [Window or View Controller Custom Action](#), you can add the `validateMenuItem:` action and manually enable or disable menu items.

In the following example, the `Tag` property will be used to decide the type of menu item that will be enabled/disabled by the `validateMenuItem:` action based on the state of selected text in a `NSTextView`. The `Tag` property has been set in Interface Builder for each menu item:



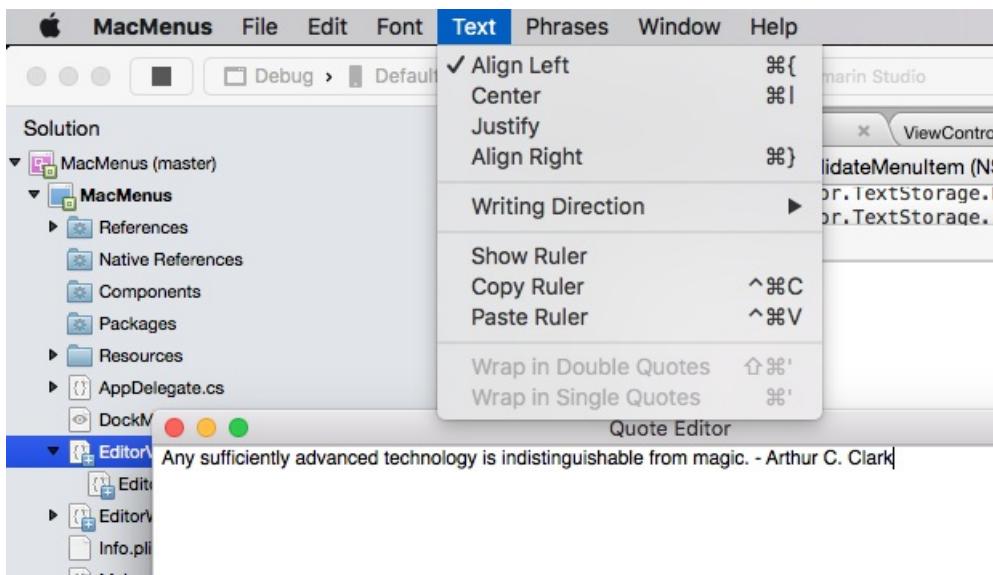
And the following code added to the View Controller:

```
[Action("validateMenuItem")]
public bool ValidateMenuItem (NSMenuItem item) {

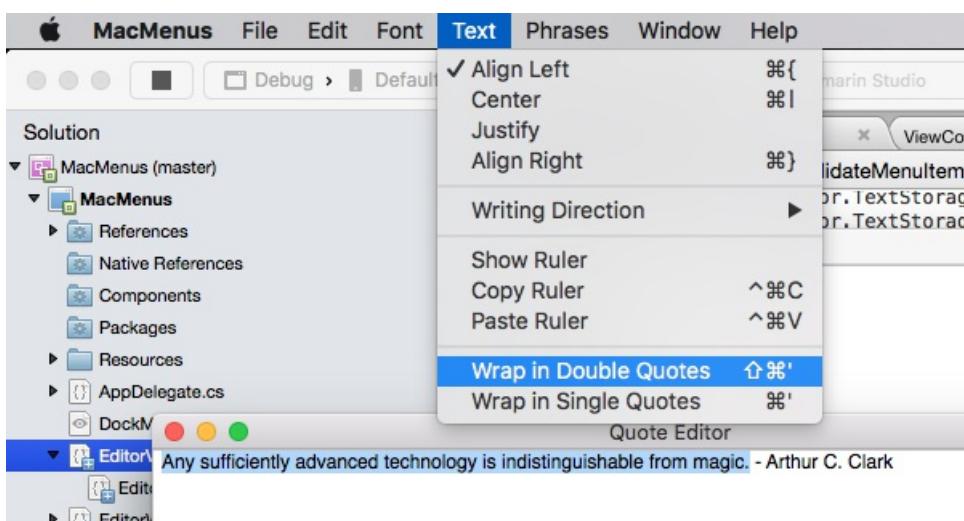
    // Take action based on the menu item type
    // (As specified in its Tag)
    switch (item.Tag) {
        case 1:
            // Wrap menu items should only be available if
            // a range of text is selected
            return (TextEditor.SelectedRange.Length > 0);
        case 2:
            // Quote menu items should only be available if
            // a range is NOT selected.
            return (TextEditor.SelectedRange.Length == 0);
    }

    return true;
}
```

When this code is run, and no text is selected in the `NSTextView`, the two wrap menu items are disabled (even though they are wired to actions on the view controller):



If a section of text is selected and the menu reopened, the two wrap menu items will be available:



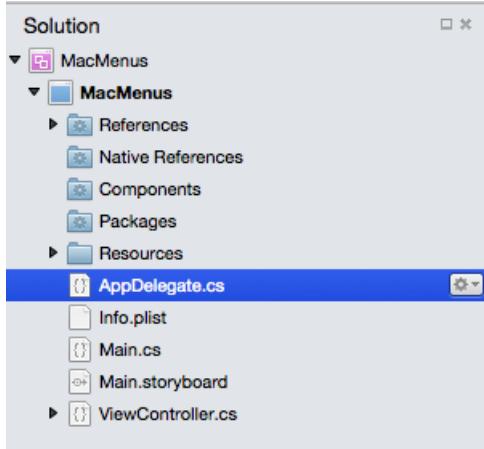
Enabling and responding to menu items in code

As we have seen above, just by adding specific Cocoa user interface elements to our UI design (such as a text

field), several of the default menu items will be enabled and function automatically, without having to write any code. Next let's look at adding our own C# code to our Xamarin.Mac project to enable a menu item and provide functionality when the user selects it.

For example, let say we want the user to be able to use the **Open** item in the **File** menu to select a folder. Since we want this to be an application-wide function and not limited to a give window or UI element, we're going to add the code to handle this to our application delegate.

In the **Solution Pad**, double-click the `AppDelegate.cs` file to open it for editing:

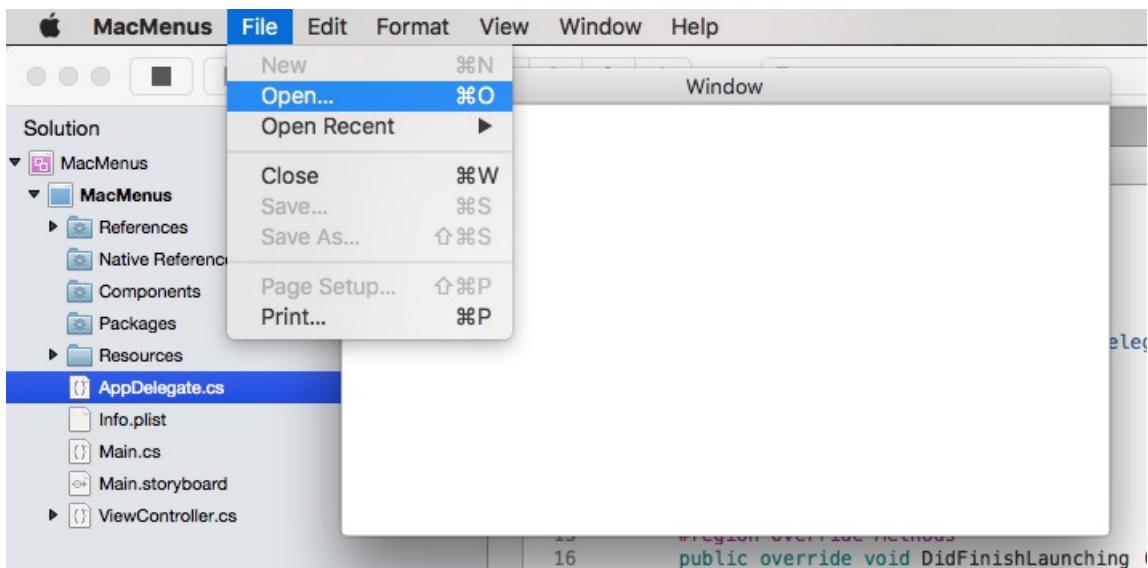


Add the following code below the `DidFinishLaunching` method:

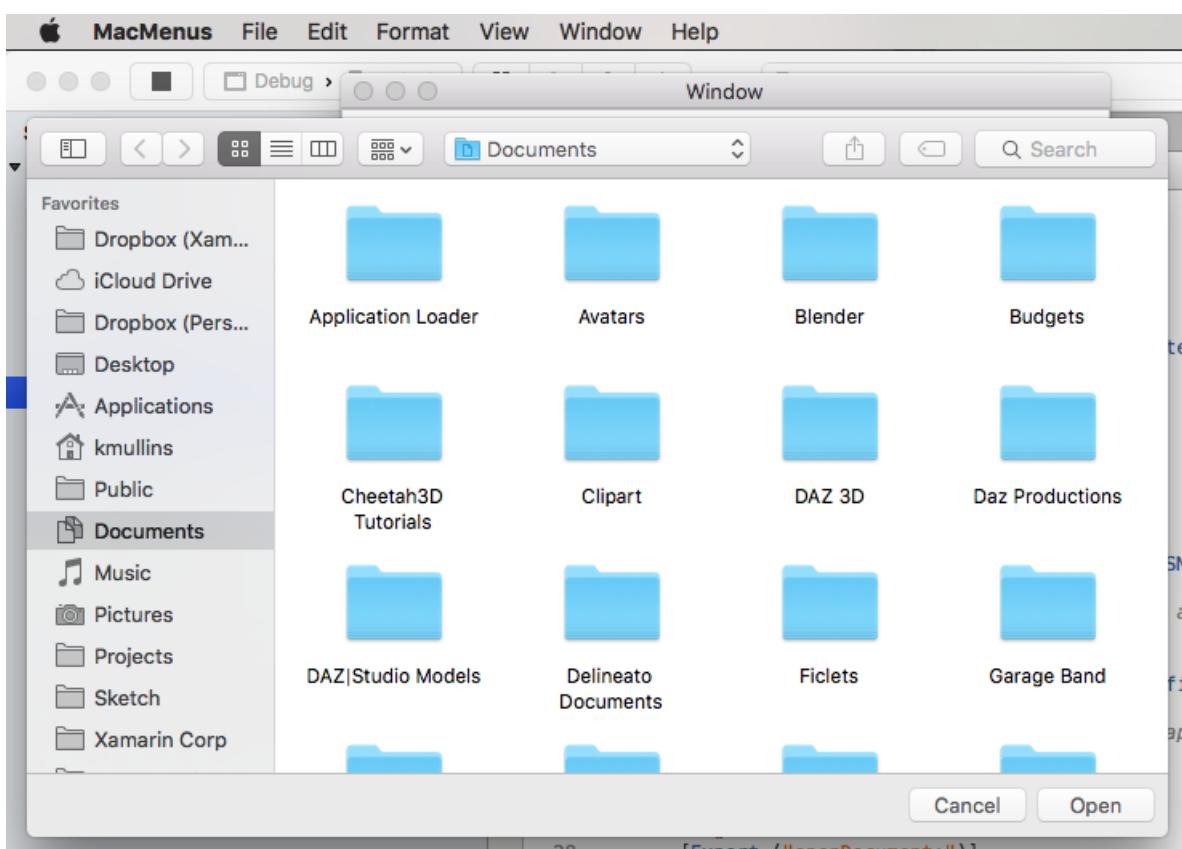
```
[Export ("openDocument:")]
void OpenDialog (NSObject sender)
{
    var dlg = NSOpenPanel.OpenPanel;
    dlg.CanChooseFiles = false;
    dlg.CanChooseDirectories = true;

    if (dlg.RunModal () == 1) {
        var alert = new NSAlert ();
        AlertStyle = NSAlertStyle.Informational,
        InformativeText = "At this point we should do something with the folder that the user just
selected in the Open File Dialog box...",
        MessageText = "Folder Selected"
    };
    alert.RunModal ();
}
```

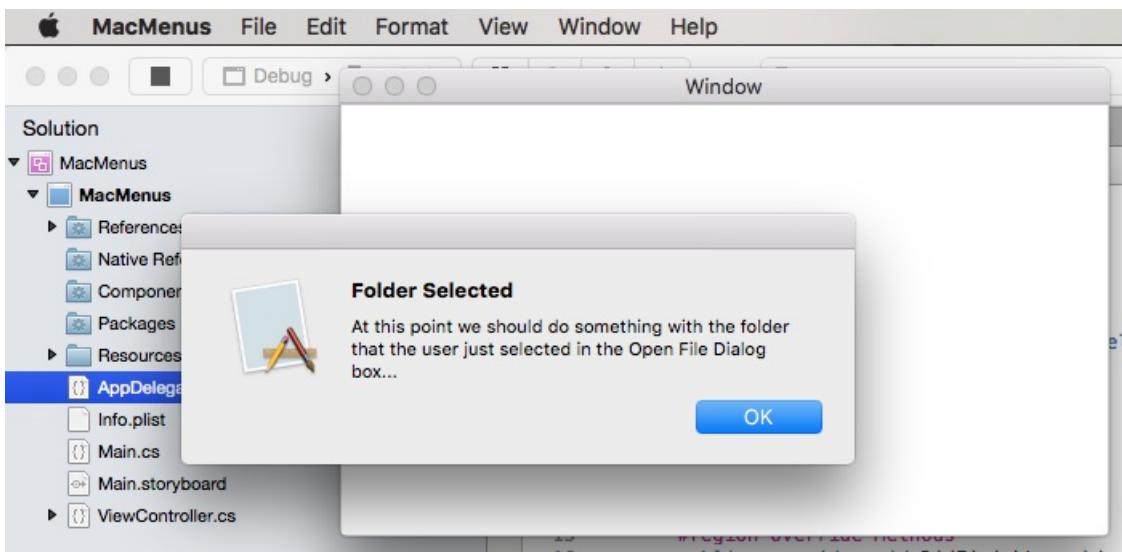
Let's run the application now and open the **File** menu:



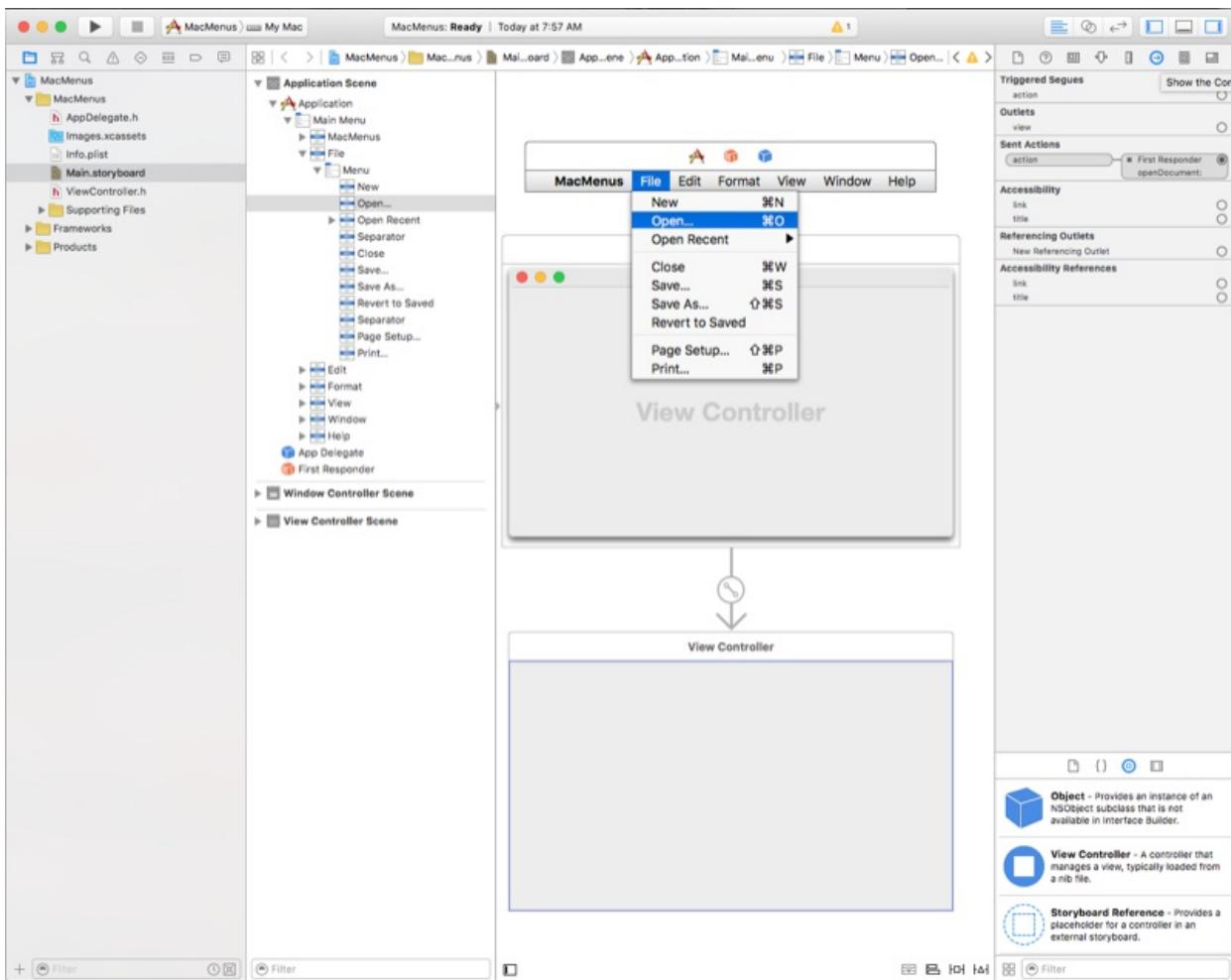
Notice that the **Open** menu item is now enabled. If we select it, the open dialog will be displayed:



If we click the **Open** button, our alert message will be displayed:



The key line here was `[Export ("openDocument:")]`, it tells `NSMenu` that our `AppDelegate` has a method `void OpenDialog (NSObject sender)` that responds to the `openDocument:` action. If you'll remember from above, the **Open** menu item is automatically wired-up to this action by default in Interface Builder:



Next let's look at creating our own menu, menu items, and actions and responding to them in code.

Working with the open recent menu

By default, the **File** menu contains an **Open Recent** item that keeps track of the last several files that the user has opened with your app. If you are creating a `NSDocument` based Xamarin.Mac app, this menu will be handled for you automatically. For any other type of Xamarin.Mac app, you will be responsible for managing and responding to this menu item manually.

To manually handle the **Open Recent** menu, you will first need to inform it that a new file has been opened or

saved using the following:

```
// Add document to the Open Recent menu  
NSDocumentController.SharedDocumentController.NoteNewRecentDocumentURL(url);
```

Even though your app is not using `NSDocuments`, you still use the `NSDocumentController` to maintain the **Open Recent** menu by sending a `NSURL` with the location of the file to the `NoteNewRecentDocumentURL` method of the `SharedDocumentController`.

Next, you need to override the `OpenFile` method of the app delegate to open any file that the user selects from the **Open Recent** menu. For example:

```
public override bool OpenFile (NSApplication sender, string filename)  
{  
    // Trap all errors  
    try {  
        filename = filename.Replace (" ", "%20");  
        var url = new NSURL ("file://" +filename);  
        return OpenFile(url);  
    } catch {  
        return false;  
    }  
}
```

Return `true` if the file can be opened, else return `false` and a built-in warning will be displayed to the user that the file could not be opened.

Because the filename and path returned from the **Open Recent** menu, might include a space, we need to properly escape this character before creating a `NSURL` or we will get an error. We do that with the following code:

```
filename = filename.Replace (" ", "%20");
```

Finally, we create a `NSURL` that points to the file and use a helper method in the app delegate to open a new window and load the file into it:

```
var url = new NSURL ("file://" +filename);  
return OpenFile(url);
```

To pull everything together, let's take a look at an example implementation in an `AppDelegate.cs` file:

```
using AppKit;  
using Foundation;  
using System.IO;  
using System;  
  
namespace MacHyperlink  
{  
    [Register ("AppDelegate")]  
    public class AppDelegate : NSApplicationDelegate  
    {  
        #region Computed Properties  
        public int NewWindowNumber { get; set; } = -1;  
        #endregion  
  
        #region Constructors  
        public AppDelegate ()  
        {
```

```

    }

    #endregion

    #region Override Methods
    public override void DidFinishLaunching (NSNotification notification)
    {
        // Insert code here to initialize your application
    }

    public override void WillTerminate (NSNotification notification)
    {
        // Insert code here to tear down your application
    }

    public override bool OpenFile (NSApplication sender, string filename)
    {
        // Trap all errors
        try {
            filename = filename.Replace (" ", "%20");
            var url = new NSUrl ("file://" +filename);
            return OpenFile(url);
        } catch {
            return false;
        }
    }
    #endregion

    #region Private Methods
    private bool OpenFile(NSUrl url) {
        var good = false;

        // Trap all errors
        try {
            var path = url.Path;

            // Is the file already open?
            for(int n=0; n<NSApplication.SharedApplication.Windows.Length; ++n) {
                var content = NSApplication.SharedApplication.Windows[n].ContentViewController as
ViewController;
                if (content != null && path == content.FilePath) {
                    // Bring window to front
                    NSApplication.SharedApplication.Windows[n].MakeKeyAndOrderFront(this);
                    return true;
                }
            }

            // Get new window
            var storyboard = NSStoryboard.FromName ("Main", null);
            var controller = storyboard.InstantiateControllerWithIdentifier ("MainWindow") as
NSWindowController;

            // Display
            controller.ShowWindow(this);

            // Load the text into the window
            var viewController = controller.Window.ContentViewController as ViewController;
            viewController.Text = File.ReadAllText(path);
            viewController.SetLanguageFromPath(path);
            viewController.View.WindowSetTitleWithRepresentedFilename (Path.GetFileName(path));
            viewController.View.Window.RepresentedUrl = url;

            // Add document to the Open Recent menu
            NSDocumentController.SharedDocumentController.NoteNewRecentDocumentURL(url);

            // Make as successful
            good = true;
        } catch {
            // Mark as bad file on error
            good = false;
        }
    }
}

```

```

        }

        // Return results
        return good;
    }
#endregion

#region actions
[Export ("openDocument:")]
void OpenDialog (NSObject sender)
{
    var dlg = NSOpenPanel.OpenPanel;
    dlg.CanChooseFiles = true;
    dlg.CanChooseDirectories = false;

    if (dlg.RunModal () == 1) {
        // Nab the first file
        var url = dlgUrls [0];

        if (url != null) {
            // Open the document in a new window
            OpenFile (url);
        }
    }
}
#endregion
}

```

Based on the requirements of your app, you might not want the user to open the same file in more than one window at the same time. In our example app, if the user chooses a file that is already open (either from the **Open Recent** or **Open..** menu items), the window that contains the file is brought to the front.

To accomplish this, we used the following code in our helper method:

```

var path = url.Path;

// Is the file already open?
for(int n=0; n<NSApplication.SharedApplication.Windows.Length; ++n) {
    var content = NSApplication.SharedApplication.Windows[n].ContentViewController as ViewController;
    if (content != null && path == content.FilePath) {
        // Bring window to front
        NSApplication.SharedApplication.Windows[n].MakeKeyAndOrderFront(this);
        return true;
    }
}

```

We designed our `ViewController` class to hold the path to the file in its `Path` property. Next, we loop through all currently open windows in the app. If the file is already open in one of the windows, it is brought to the front of all other windows using:

```
NSApplication.SharedApplication.Windows[n].MakeKeyAndOrderFront(this);
```

If no match is found, a new window is opened with the file loaded and the file is noted in the **Open Recent** menu:

```

// Get new window
var storyboard = NSStoryboard.FromName ("Main", null);
var controller = storyboard.InstantiateControllerWithIdentifier ("MainWindow") as NSWindowController;

// Display
controller.ShowWindow(this);

// Load the text into the window
var viewController = controller.Window.ContentViewController as ViewController;
viewController.Text = File.ReadAllText(path);
viewController.SetLanguageFromPath(path);
viewController.View.WindowSetTitleWithRepresentedFilename (Path.GetFileName(path));
viewController.View.Window.RepresentedUrl = url;

// Add document to the Open Recent menu
NSDocumentController.SharedDocumentController.NoteNewRecentDocumentURL(url);

```

Working with custom window actions

Just like the built-in **First Responder** actions that come pre-wired to standard menu items, you can create new, custom actions and wire them to menu items in Interface Builder.

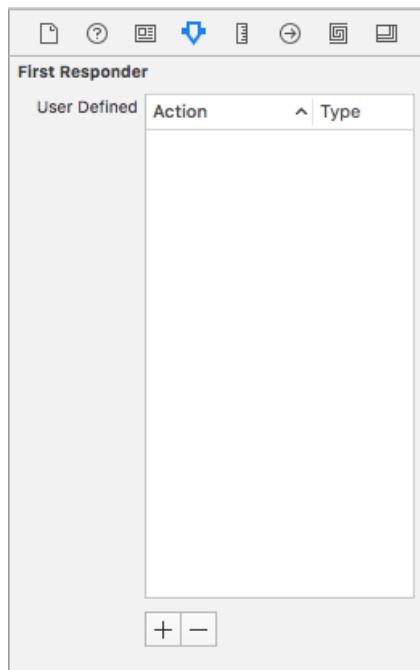
First, define a custom action on one of your app's window controllers. For example:

```

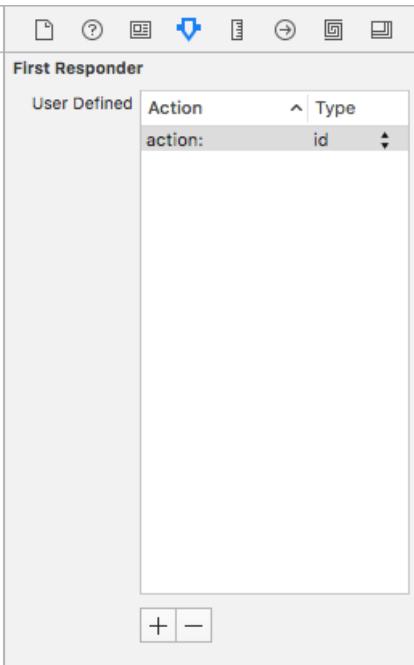
[Action("defineKeyword")]
public void defineKeyword (NSObject sender) {
    // Preform some action when the menu is selected
    Console.WriteLine ("Request to define keyword");
}

```

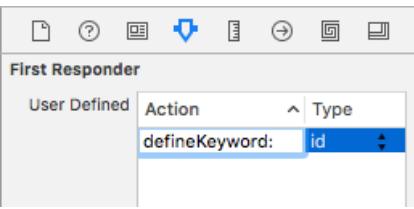
Next, double-click the app's storyboard file in the **Solution Pad** to open it for editing in Xcode's Interface Builder. Select the **First Responder** under the **Application Scene**, then switch to the **Attributes Inspector**:



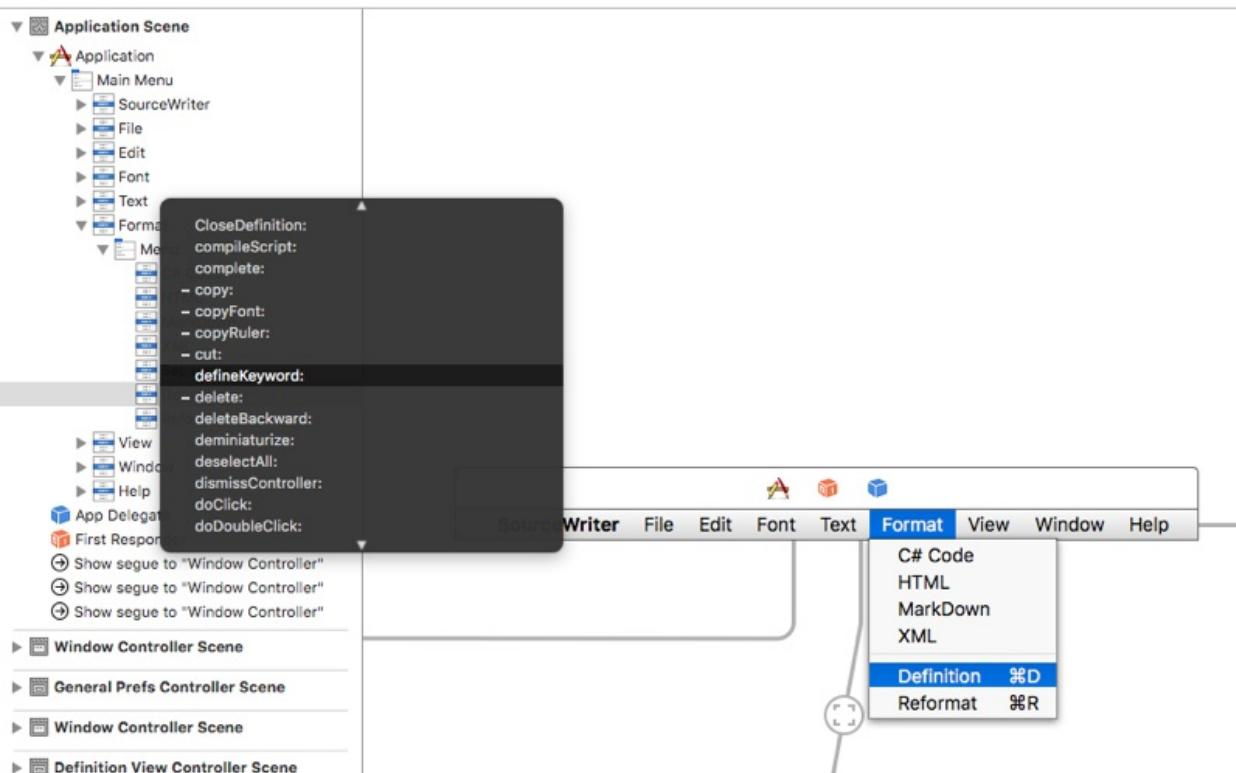
Click the **+** button at the bottom of the **Attributes Inspector** to add a new custom action:



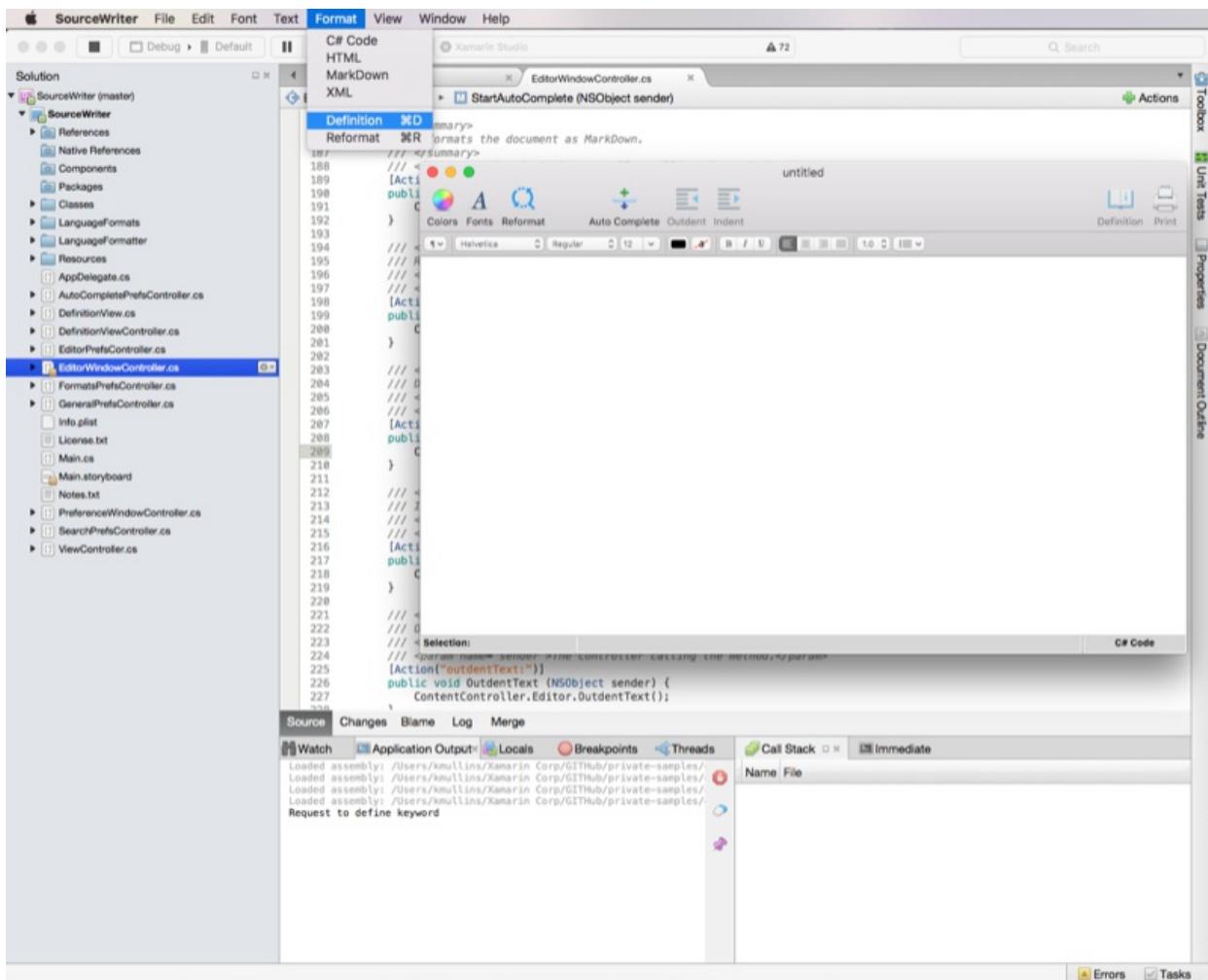
Give it the same name as the custom action that you created on your window controller:



Control-click and drag from a menu item to the **First Responder** under the **Application Scene**. From the popup list, select the new action you just created (`defineKeyword:` in this example):



Save the changes to the storyboard and return to Visual Studio for Mac to sync the changes. If you run the app, the menu item that you connected the custom action to will automatically be enabled/disabled (based on the window with the action being open) and selecting the menu item will fire off the action:

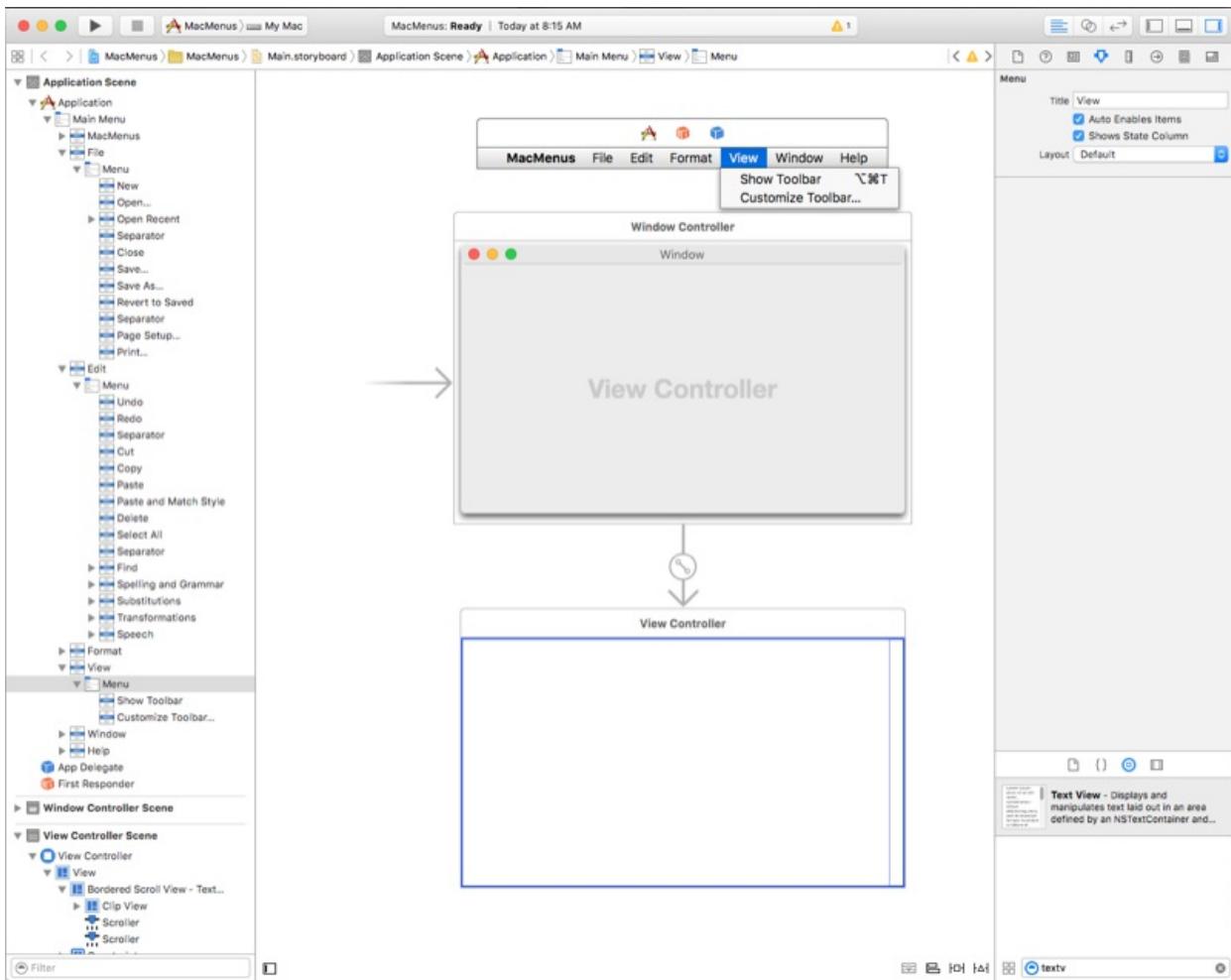


Adding, editing, and deleting menus

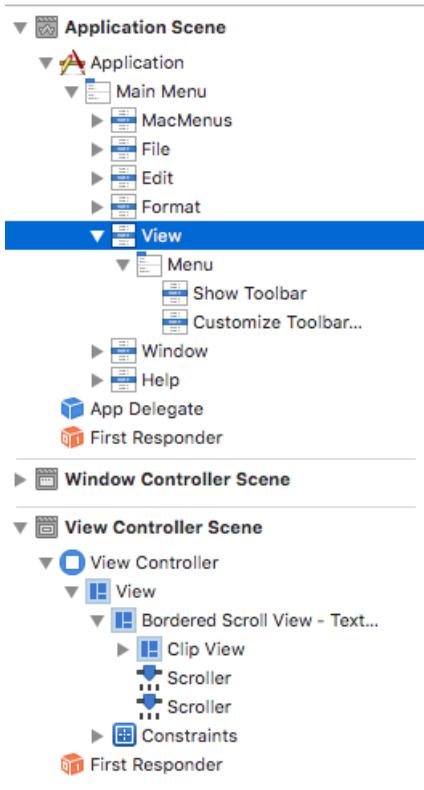
As we have seen in the previous sections, a Xamarin.Mac application comes with a preset number of default menus and menu items that specific UI controls will automatically activate and respond to. We have also seen how to add code to our application that will also enable and respond to these default items.

In this section we will look at removing menu items that we don't need, reorganizing menus and adding new menus, menu items and actions.

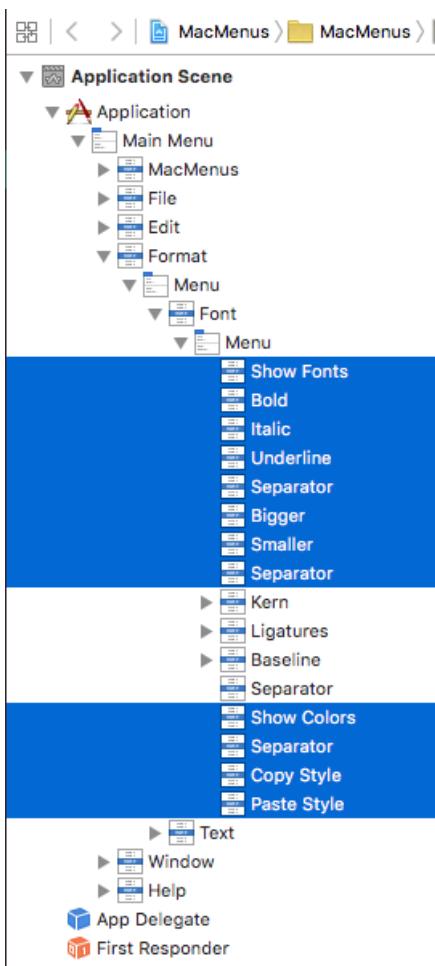
Double-click the **Main.storyboard** file in the **Solution Pad** to open it for editing:



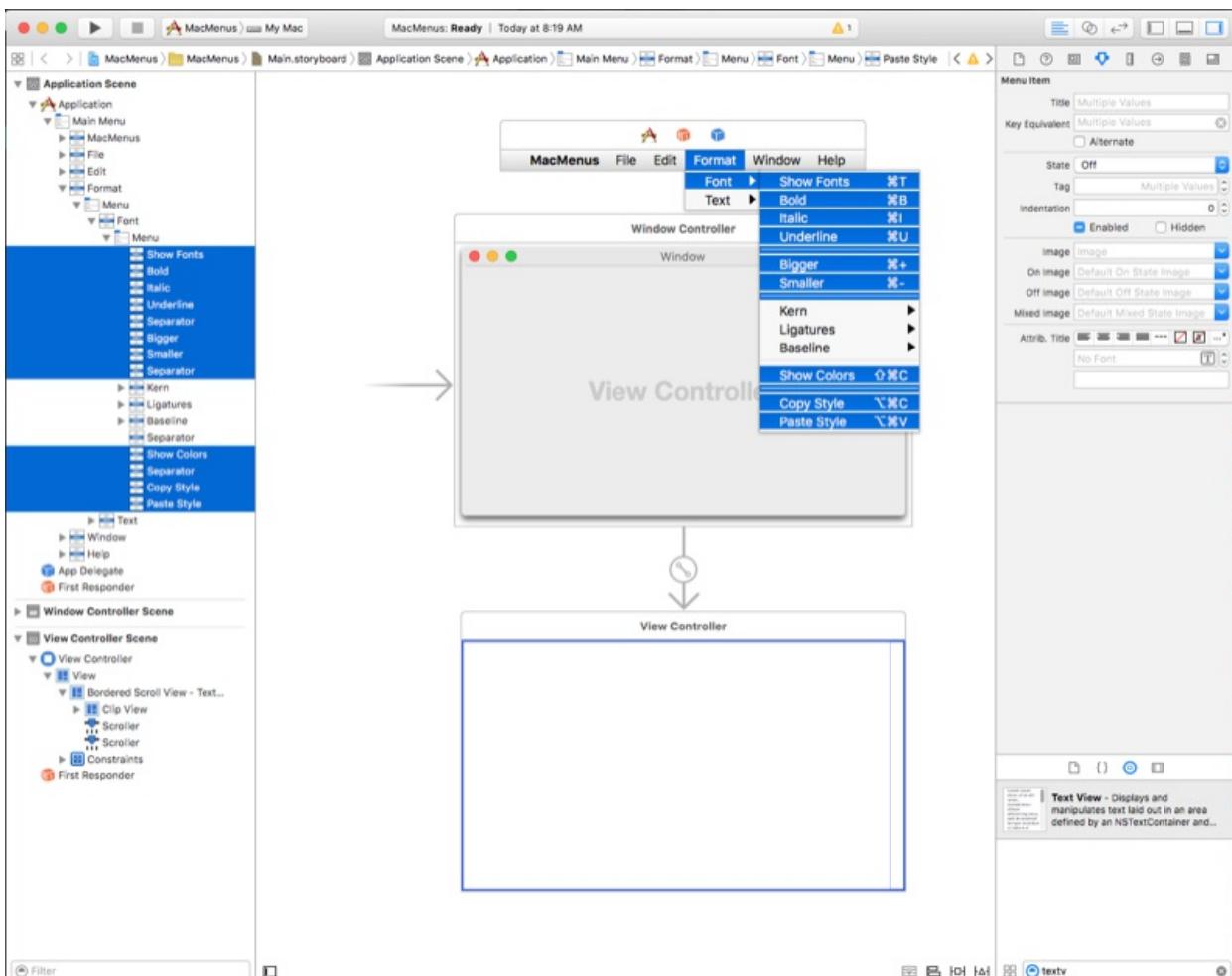
For our specific Xamarin.Mac application we are not going to be using the default **View** menu so we are going to remove it. In the **Interface Hierarchy** select the **View** menu item that is a part of the main menu bar:



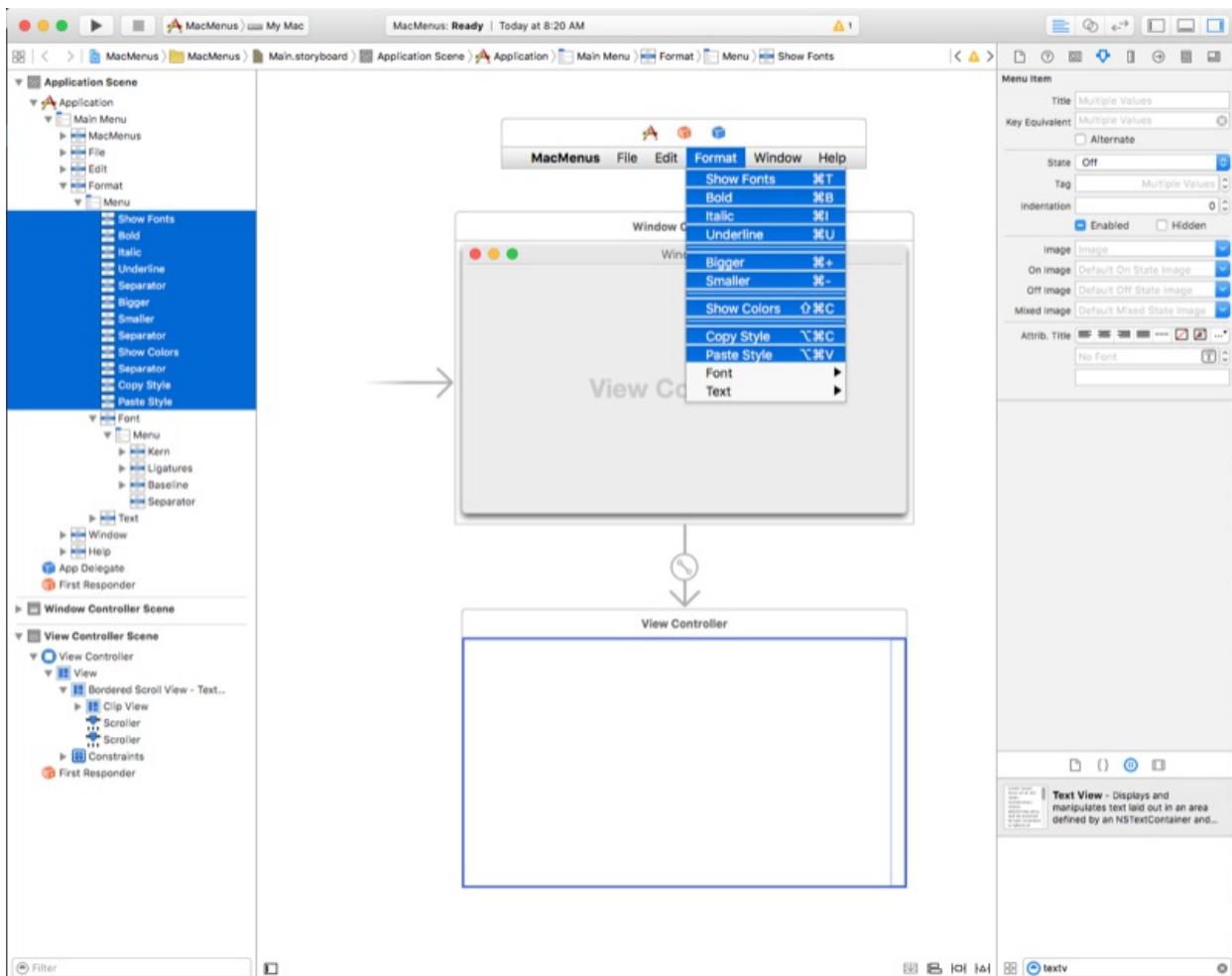
Press delete or backspace to delete the menu. Next, we aren't going to be using all of the items in the **Format** menu and we want to move the items we are going to use out from under the sub menus. In the **Interface Hierarchy** select the following menu items:



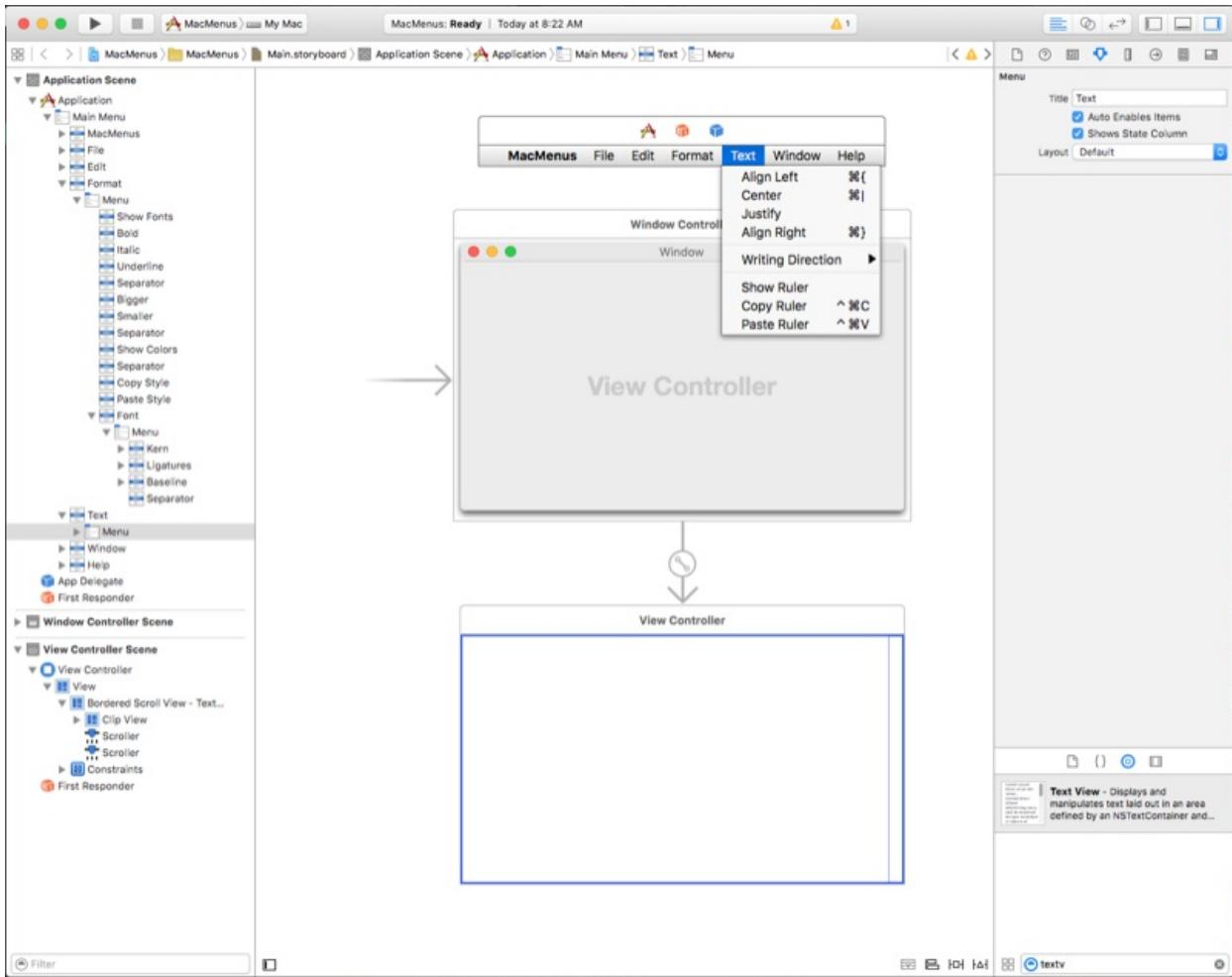
Drag the items under the parent **Menu** from the sub-menu where they currently are:



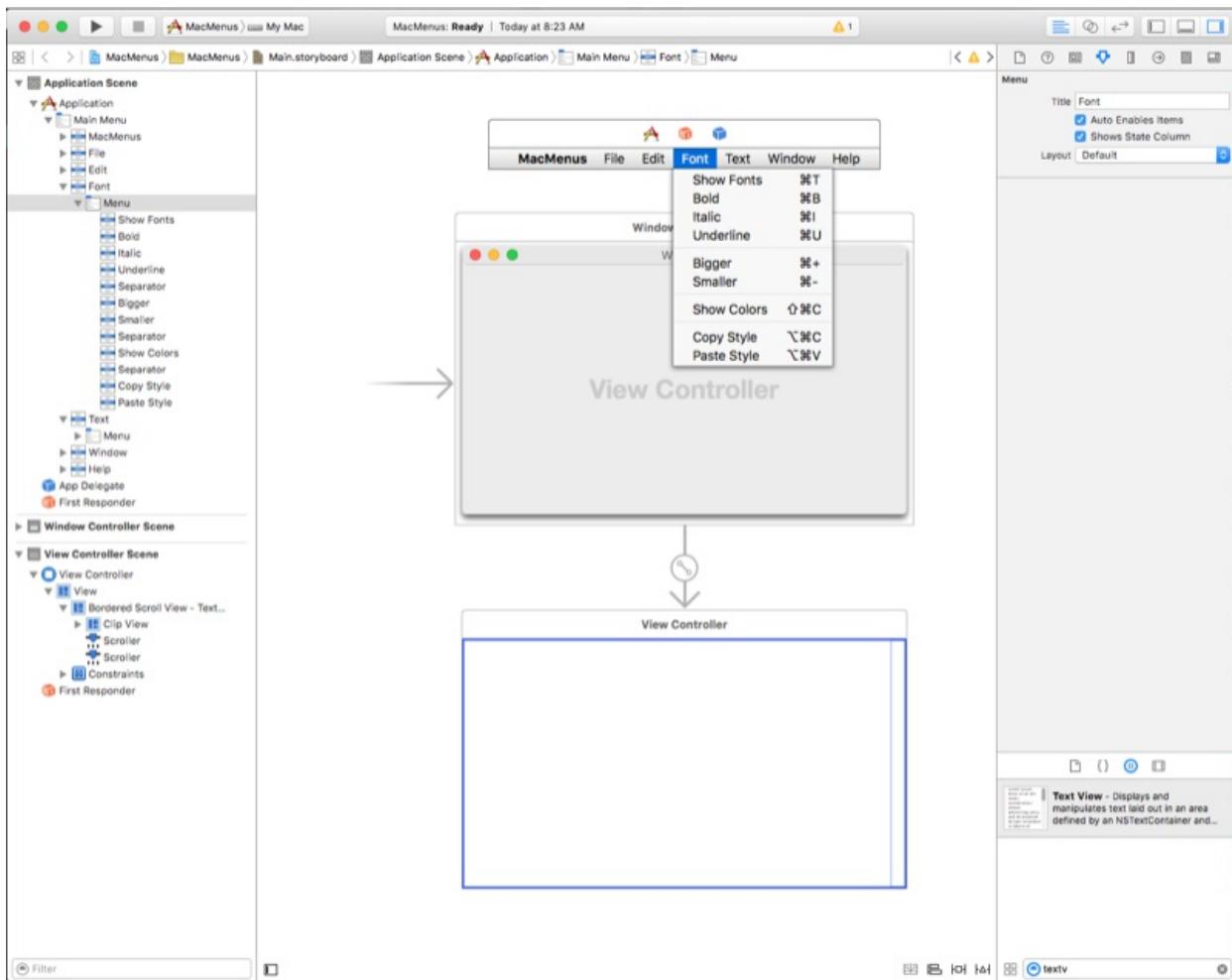
Your menu should now look like:



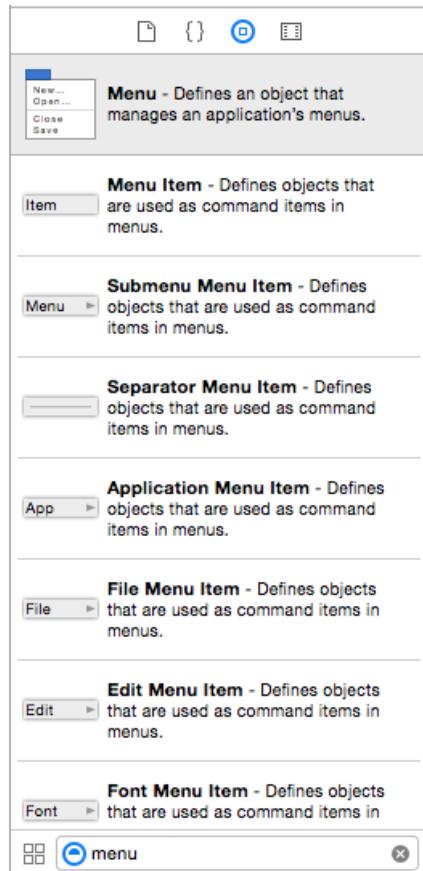
Next let's drag the **Text** sub-menu out from under the **Format** menu and place it on the main menu bar between the **Format** and **Window** menus:



Let's go back under the **Format** menu and delete the **Font** sub-menu item. Next, select the **Format** menu and rename it "Font":

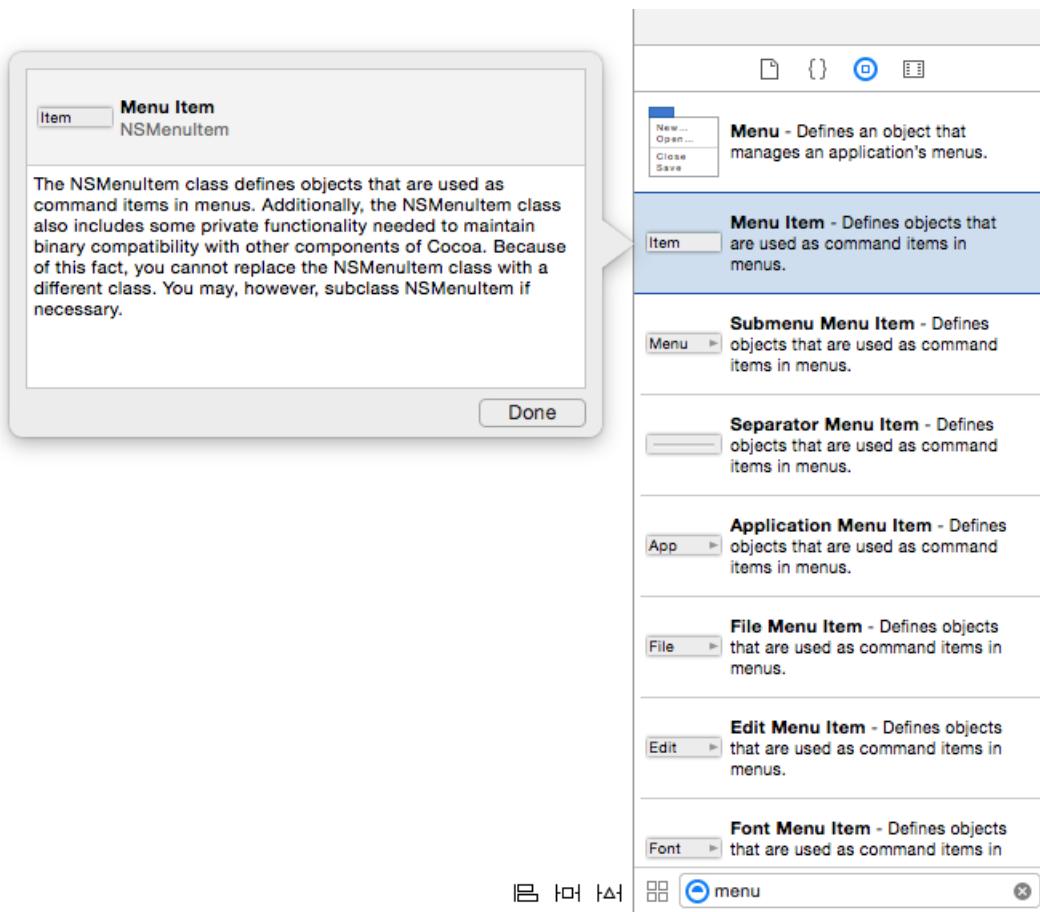


Next, let's create a custom menu of predefine phrases that will automatically get appended to the text in the text view when they are selected. In the search box at the bottom on the **Library Inspector** type in "menu." This will make it easier to find and work with all of the menu UI elements:

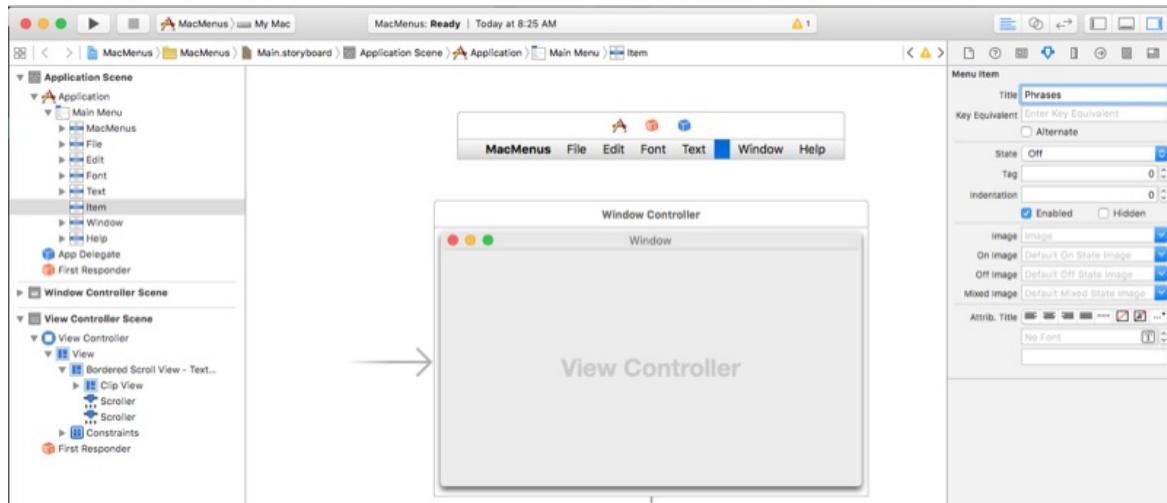


Now let's do the following to create our menu:

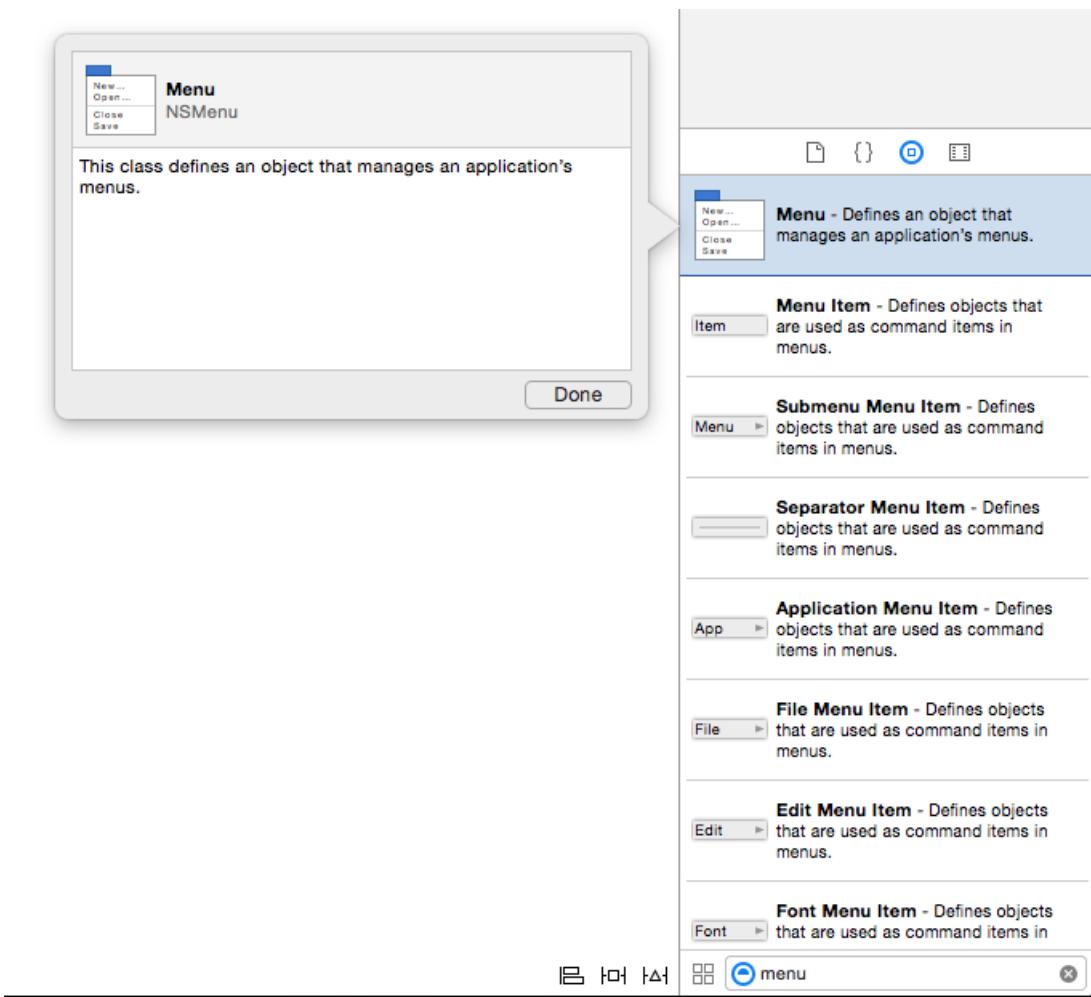
1. Drag a **Menu Item** from the **Library Inspector** onto the menu bar between the **Text** and **Window** menus:



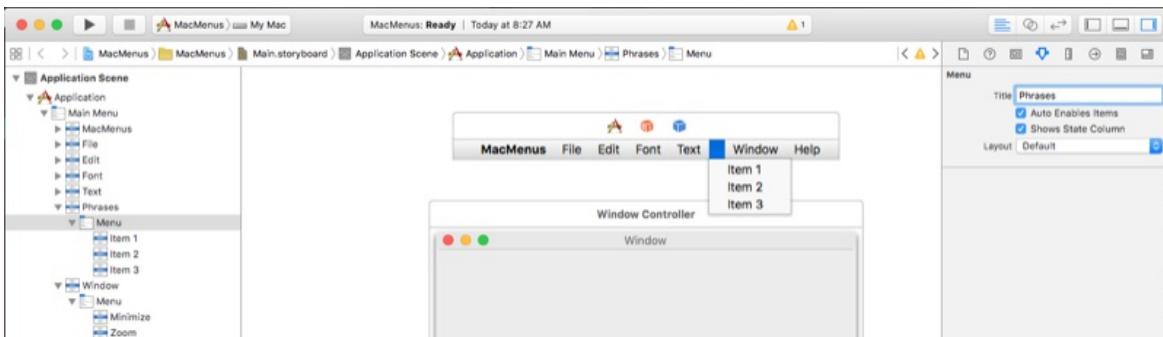
2. Rename the item "Phrases":



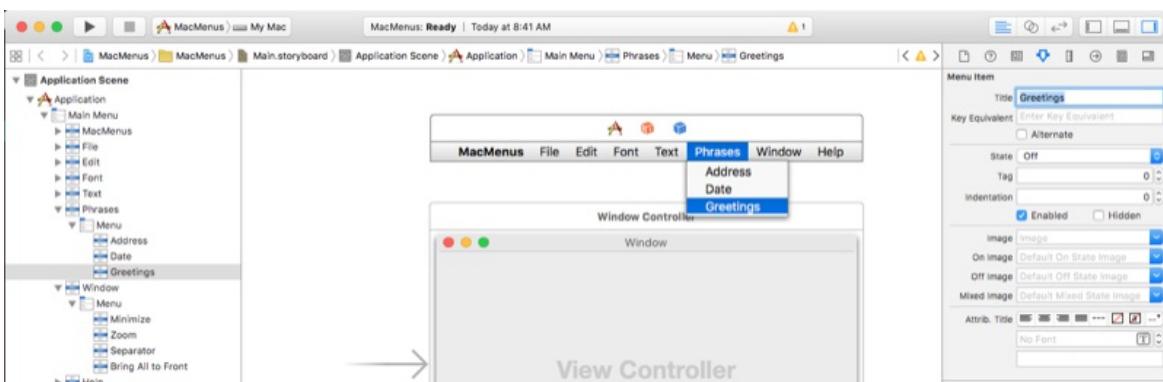
3. Next drag a **Menu** from the **Library Inspector**:



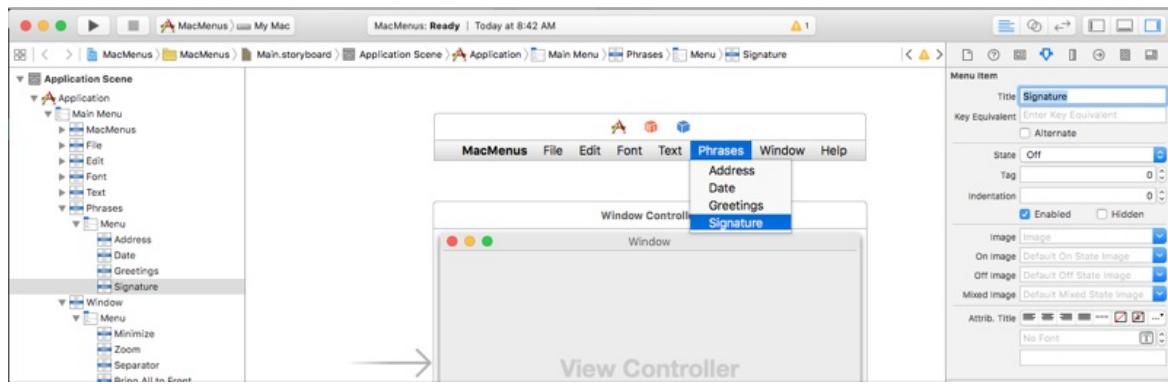
4. Drop then **Menu** on the new **Menu Item** we just created and change its name to "Phrases":



5. Now let's rename the three default **Menu Items** "Address", "Date," and "Greeting":

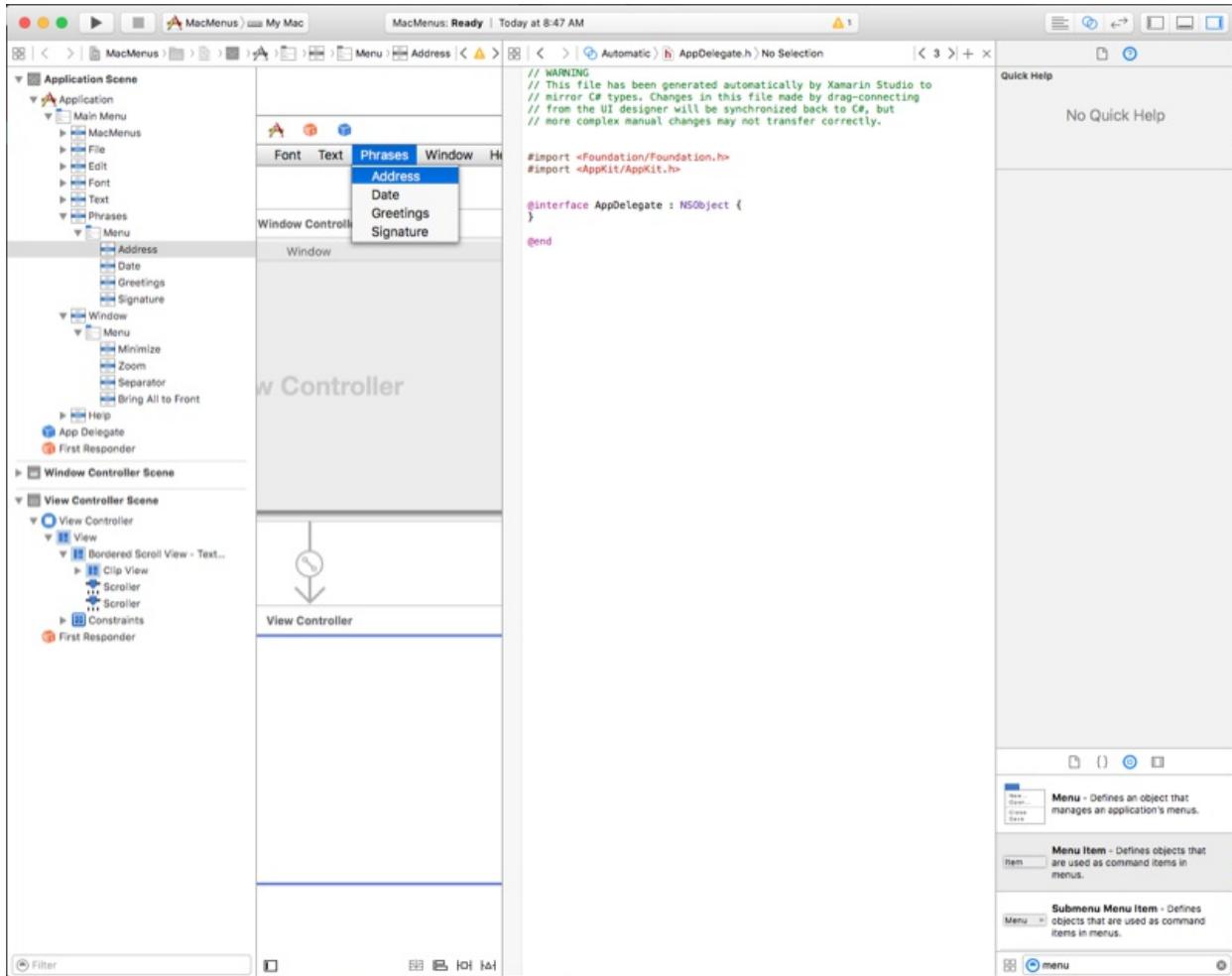


6. Let's add a fourth **Menu Item** by dragging a **Menu Item** from the **Library Inspector** and calling it "Signature":



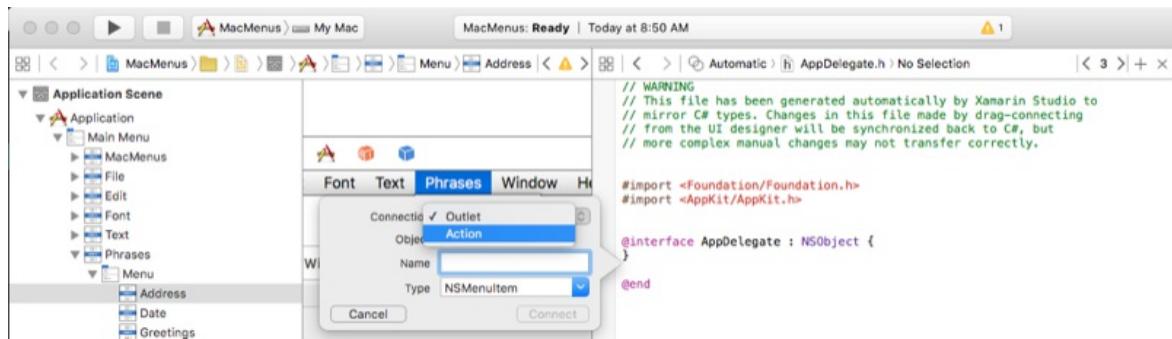
7. Save the changes to the menu bar.

Now let's create a set of custom actions so that our new menu items are exposed to C# code. In Xcode let's switch to the **Assistant** view:

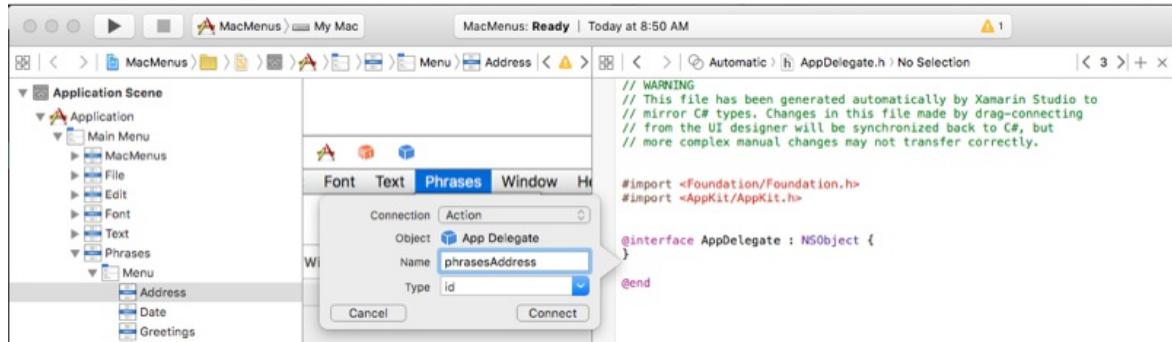


Let's do the following:

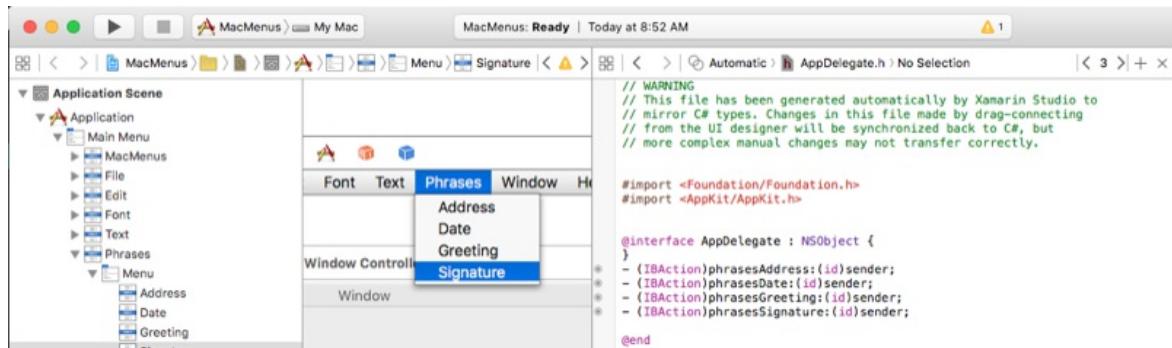
1. Control-drag from the Address menu item to the **AppDelegate.h** file.
2. Switch the **Connection type** to **Action**:



3. Enter a Name of "phraseAddress" and press the Connect button to create the new action:

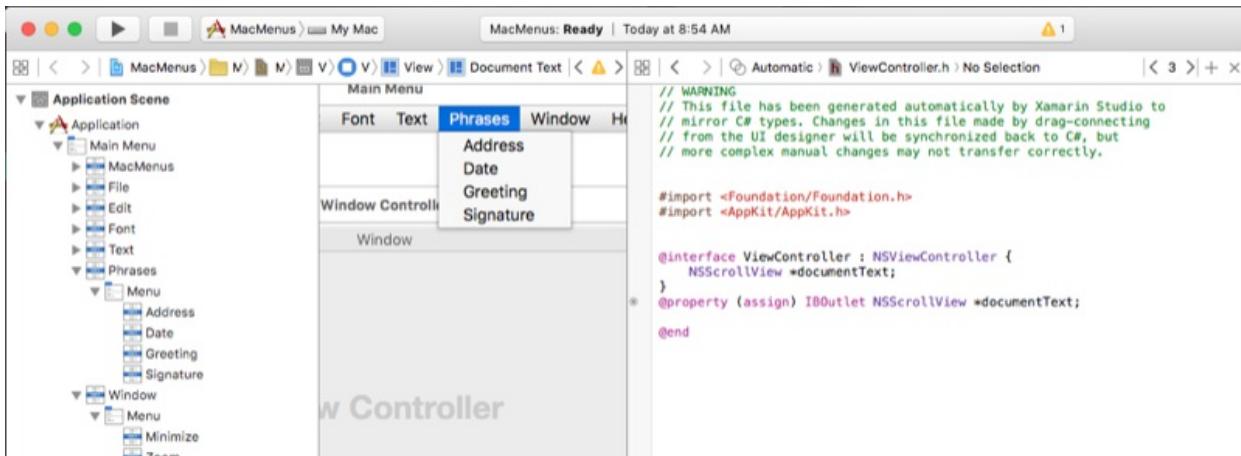


4. Repeat the above steps for the Date, Greeting, and Signature menu items:



5. Save the changes to the menu bar.

Next we need to create an outlet for our text view so that we can adjust its content from code. Select the **ViewController.h** file in the **Assistant Editor** and create a new outlet called `documentText`:



Return to Visual Studio for Mac to sync the changes from Xcode. Next edit the **ViewController.cs** file and make it look like the following:

```

using System;

using AppKit;
using Foundation;

namespace MacMenus
{
    public partial class ViewController : NSViewController
    {
        #region Application Access
        public static AppDelegate App {
            get { return (AppDelegate)NSApplication.SharedApplication.Delegate; }
        }
        #endregion

        #region Computed Properties
        public override NSObject RepresentedObject {
            get {
                return base.RepresentedObject;
            }
            set {
                base.RepresentedObject = value;
                // Update the view, if already loaded.
            }
        }
        #endregion

        public string Text {
            get { return documentText.Value; }
            set { documentText.Value = value; }
        }
        #endregion

        #region Constructors
        public ViewController (IntPtr handle) : base (handle)
        {
        }
        #endregion

        #region Override Methods
        public override void ViewDidLoad ()
        {
            base.ViewDidLoad ();

            // Do any additional setup after loading the view.
        }

        public override void ViewWillAppear ()
        {
            base.ViewWillAppear ();

            App.textEditor = this;
        }

        public override void ViewWillDisappear ()
        {
            base.ViewWillDisappear ();

            App.textEditor = null;
        }
        #endregion
    }
}

```

This exposes the text of our text view outside of the `ViewController` class and informs the app delegate when the window gains or loses focus. Now edit the `AppDelegate.cs` file and make it look like the following:

```

using AppKit;
using Foundation;
using System;

namespace MacMenus
{
    [Register ("AppDelegate")]
    public partial class AppDelegate : NSApplicationDelegate
    {
        #region Computed Properties
        public ViewController textEditor { get; set;} = null;
        #endregion

        #region Constructors
        public AppDelegate ()
        {
        }
        #endregion

        #region Override Methods
        public override void DidFinishLaunching (NSNotification notification)
        {
            // Insert code here to initialize your application
        }

        public override void WillTerminate (NSNotification notification)
        {
            // Insert code here to tear down your application
        }
        #endregion

        #region Custom actions
        [Export ("openDocument:")]
        void OpenDialog (NSObject sender)
        {
            var dlg = NSOpenPanel.OpenPanel;
            dlg.CanChooseFiles = false;
            dlg.CanChooseDirectories = true;

            if (dlg.RunModal () == 1) {
                var alert = new NSAlert ();
                AlertStyle = NSAlertStyle.Informational,
                InformativeText = "At this point we should do something with the folder that the user
just selected in the Open File Dialog box...",
                MessageText = "Folder Selected"
            };
            alert.RunModal ();
        }
    }

    partial void phrasesAddress (Foundation.NSObject sender) {

        textEditor.Text += "Xamarin HQ\n394 Pacific Ave, 4th Floor\nSan Francisco CA 94111\n\n";
    }

    partial void phrasesDate (Foundation.NSObject sender) {

        textEditor.Text += DateTime.Now.ToString("D");
    }

    partial void phrasesGreeting (Foundation.NSObject sender) {

        textEditor.Text += "Dear Sirs,\n\n";
    }

    partial void phrasesSignature (Foundation.NSObject sender) {

        textEditor.Text += "Sincerely,\n\nKevin Mullins\nXamarin, Inc.\n";
    }
}

```

```

        }
        #endregion
    }
}

```

Here we've made the `AppDelegate` a partial class so that we can use the actions and outlets that we defined in Interface Builder. We also expose a `textEditor` to track which window is currently in focus.

The following methods are used to handle our custom menu and menu items:

```

partial void phrasesAddress (Foundation.NSObject sender) {

    if (textEditor == null) return;
    textEditor.Text += "Xamarin HQ\n394 Pacific Ave, 4th Floor\nSan Francisco CA 94111\n\n";
}

partial void phrasesDate (Foundation.NSObject sender) {

    if (textEditor == null) return;
    textEditor.Text += DateTime.Now.ToString("D");
}

partial void phrasesGreeting (Foundation.NSObject sender) {

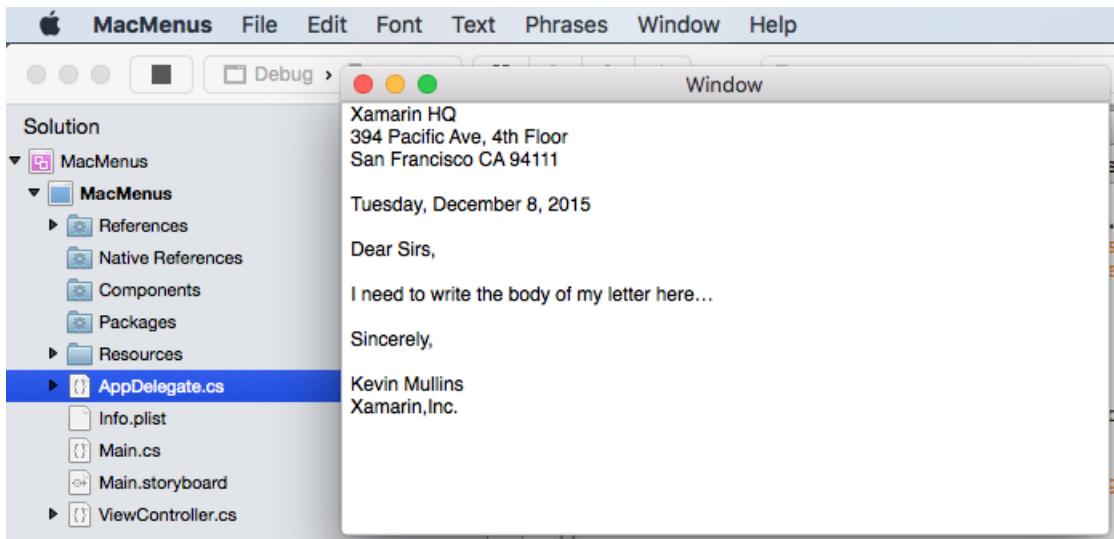
    if (textEditor == null) return;
    textEditor.Text += "Dear Sirs,\n\n";
}

partial void phrasesSignature (Foundation.NSObject sender) {

    if (textEditor == null) return;
    textEditor.Text += "Sincerely,\n\nKevin Mullins\nXamarin, Inc.\n";
}

```

Now if we run our application, all of the items in the **Phrase** menu will be active and will add the give phrase to the text view when selected:



Now that we have the basics of working with the application menu bar down, let's look at creating a custom contextual menu.

Creating menus from code

In addition to creating menus and menu items with Xcode's Interface Builder, there might be times when a Xamarin.Mac app needs to create, modify, or remove a menu, sub-menu, or menu item from code.

In the following example, a class is created to hold the information about the menu items and sub-menus that

will be dynamically created on-the-fly:

```
using System;
using System.Collections.Generic;
using Foundation;
using AppKit;

namespace AppKit.TextKit.Formatter
{
    public class LanguageFormatCommand : NSObject
    {
        #region Computed Properties
        public string Title { get; set; } = "";
        public string Prefix { get; set; } = "";
        public string Postfix { get; set; } = "";
        public List<LanguageFormatCommand> SubCommands { get; set; } = new List<LanguageFormatCommand>();
        #endregion

        #region Constructors
        public LanguageFormatCommand () {

        }

        public LanguageFormatCommand (string title)
        {
            // Initialize
            this.Title = title;
        }

        public LanguageFormatCommand (string title, string prefix)
        {
            // Initialize
            this.Title = title;
            this.Prefix = prefix;
        }

        public LanguageFormatCommand (string title, string prefix, string postfix)
        {
            // Initialize
            this.Title = title;
            this.Prefix = prefix;
            this.Postfix = postfix;
        }
        #endregion
    }
}
```

Adding menus and items

With this class defined, the following routine will parse a collection of `LanguageFormatCommand` objects and recursively build new menus and menu items by appending them to the bottom of the existing menu (created in Interface Builder) that has been passed in:

```

private void AssembleMenu(NSMenu menu, List<LanguageFormatCommand> commands) {
    NSMenuItem menuItem;

    // Add any formatting commands to the Formatting menu
    foreach (LanguageFormatCommand command in commands) {
        // Add separator or item?
        if (command.Title == "") {
            menuItem = NSMenuItem.SeparatorItem;
        } else {
            menuItem = new NSMenuItem (command.Title);

            // Submenu?
            if (command.SubCommands.Count > 0) {
                // Yes, populate submenu
                menuItem.Submenu = new NSMenu (command.Title);
                AssembleMenu (menuItem.Submenu, command.SubCommands);
            } else {
                // No, add normal menu item
                menuItem.Activated += (sender, e) => {
                    // Apply the command on the selected text
                    TextEditor.PerformFormattingCommand (command);
                };
            }
        }
        menu.AddItem (menuItem);
    }
}

```

For any `LanguageFormatCommand` object that has a blank `Title` property, this routine creates a **Separator menu item** (a thin gray line) between menu sections:

```
menuItem = NSMenuItem.SeparatorItem;
```

If a title is provided, a new menu item with that title is created:

```
menuItem = new NSMenuItem (command.Title);
```

If the `LanguageFormatCommand` object contains child `LanguageFormatCommand` objects, a sub-menu is created and the `AssembleMenu` method is recursively called to build out that menu:

```
menuItem.Submenu = new NSMenu (command.Title);
AssembleMenu (menuItem.Submenu, command.SubCommands);
```

For any new menu item that does not have sub-menus, code is added to handle the menu item being selected by the user:

```
menuItem.Activated += (sender, e) => {
    // Do something when the menu item is selected
    ...
};
```

Testing the menu creation

With all of the above code in place, if the following collection of `LanguageFormatCommand` objects were created:

```

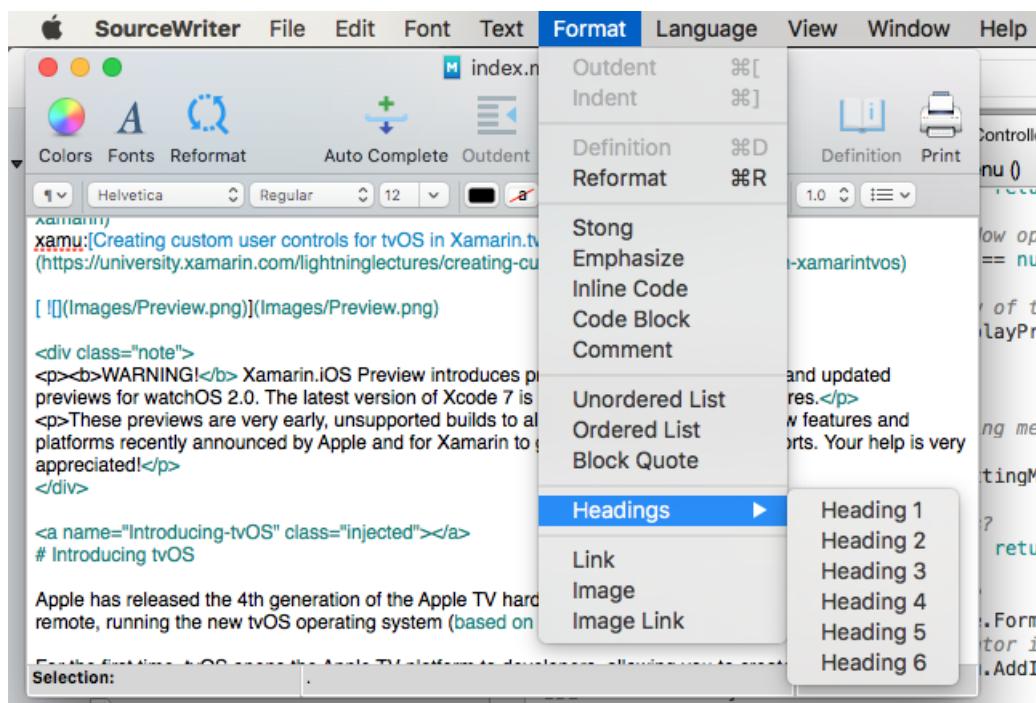
// Define formatting commands
FormattingCommands.Add(new LanguageFormatCommand("Strong","**","**"));
FormattingCommands.Add(new LanguageFormatCommand("Emphasize","_","_"));
FormattingCommands.Add(new LanguageFormatCommand("Inline Code","`","`"));
FormattingCommands.Add(new LanguageFormatCommand("Code Block","```\n","`"));
FormattingCommands.Add(new LanguageFormatCommand("Comment",<!--,-->));
FormattingCommands.Add (new LanguageFormatCommand ());
FormattingCommands.Add(new LanguageFormatCommand("Unordered List","* "));
FormattingCommands.Add(new LanguageFormatCommand("Ordered List","1. "));
FormattingCommands.Add(new LanguageFormatCommand("Block Quote",> ));
FormattingCommands.Add (new LanguageFormatCommand ());

var Headings = new LanguageFormatCommand ("Headings");
Headings.SubCommands.Add(new LanguageFormatCommand("Heading 1","# "));
Headings.SubCommands.Add(new LanguageFormatCommand("Heading 2","## "));
Headings.SubCommands.Add(new LanguageFormatCommand("Heading 3","### "));
Headings.SubCommands.Add(new LanguageFormatCommand("Heading 4","#### "));
Headings.SubCommands.Add(new LanguageFormatCommand("Heading 5","##### "));
Headings.SubCommands.Add(new LanguageFormatCommand("Heading 6","##### "));
FormattingCommands.Add (Headings);

FormattingCommands.Add(new LanguageFormatCommand ());
FormattingCommands.Add(new LanguageFormatCommand("Link","[,""]()"));
FormattingCommands.Add(new LanguageFormatCommand("Image","![](),""));
FormattingCommands.Add(new LanguageFormatCommand("Image Link","![,](LinkImageHere)"));

```

And that collection passed to the `AssembleMenu` function (with the Format Menu set as the base), the following dynamic menus and menu items would be created:



Removing menus and items

If you need to remove any menu or menu item from the app's user interface, you can use the `RemoveItemAt` method of the `NSMenuItem` class simply by giving it the zero based index of the item to remove.

For example, to remove the menus and menu items created by the routine above, you could use the following code:

```

public void UnpopulateFormattingMenu(NSMenu menu) {

    // Remove any additional items
    for (int n = (int)menu.Count - 1; n > 4; --n) {
        menu.RemoveItemAt (n);
    }
}

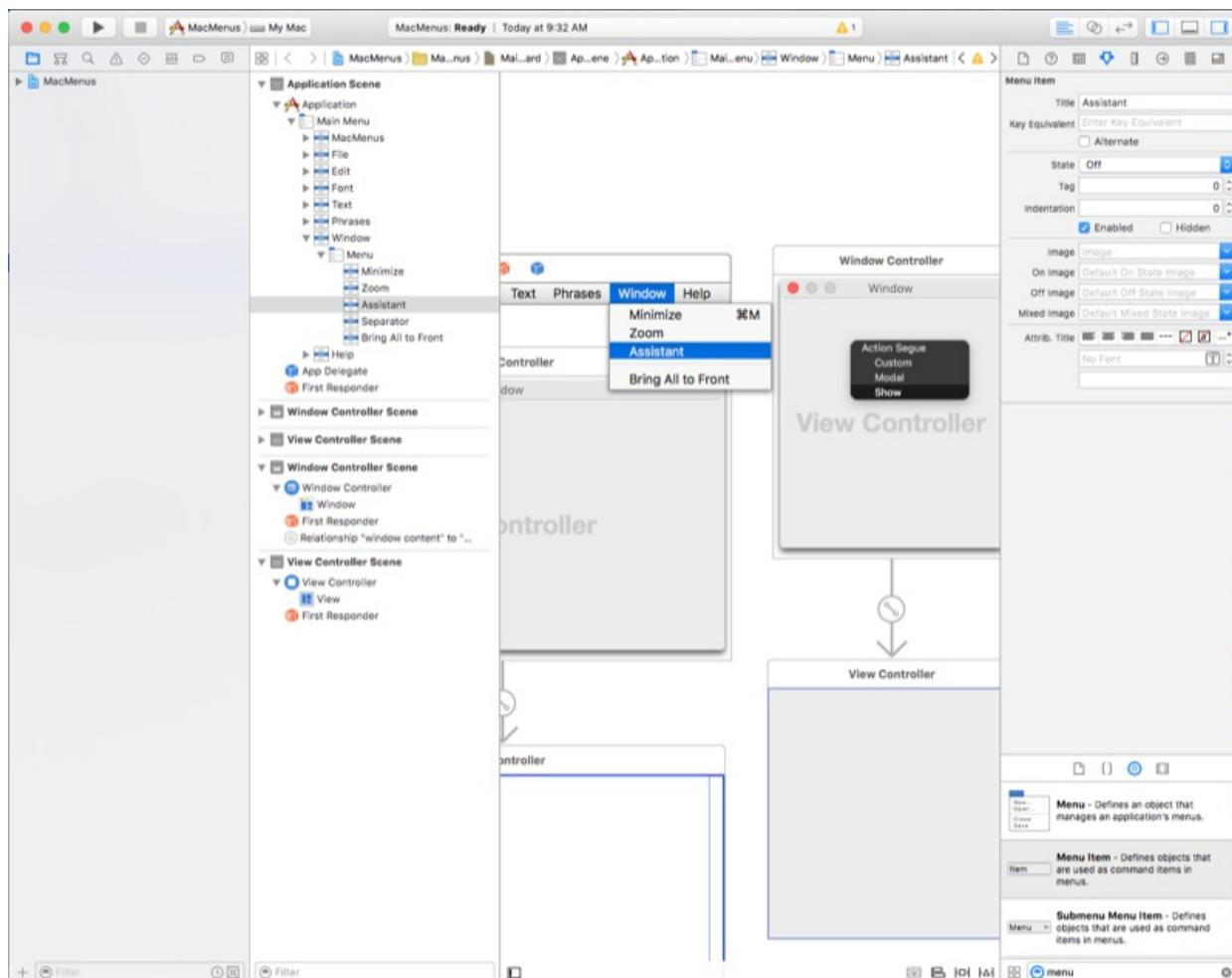
```

In the case of the code above, the first four menu items are created in Xcode's Interface Builder and always available in the app, so they are not removed dynamically.

Contextual menus

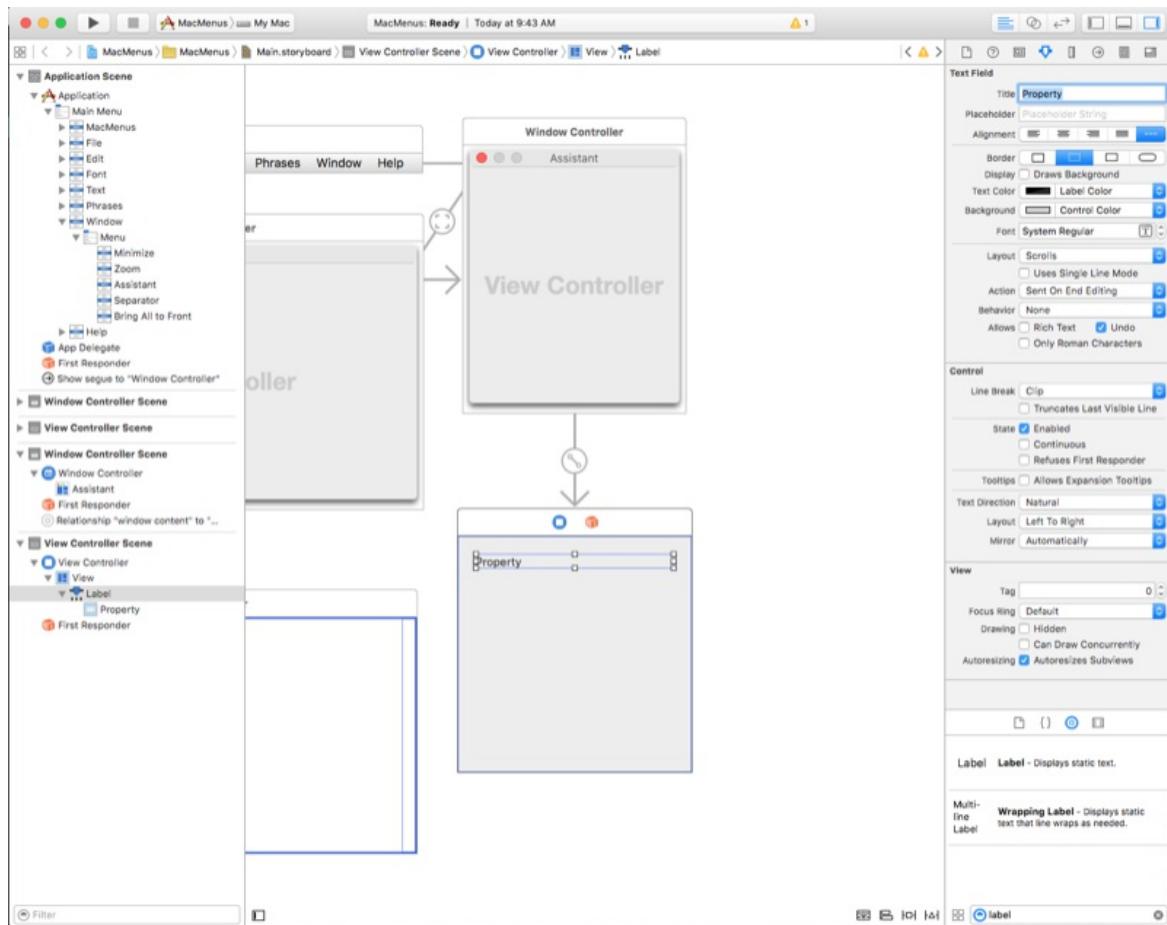
Contextual menus appear when the user right-clicks or control-clicks an item in a window. By default, several of the UI elements built into macOS already have contextual menus attached to them (such as the text view). However, there might be times when we want to create our own custom contextual menus for a UI element that we have added to a window.

Let's edit our **Main.storyboard** file in Xcode and add a **Window** window to our design, set its **Class** to "**NSPanel**" in the **Identity Inspector**, add a new **Assistant** item to the **Window** menu, and attach it to the new window using a **Show Segue**:

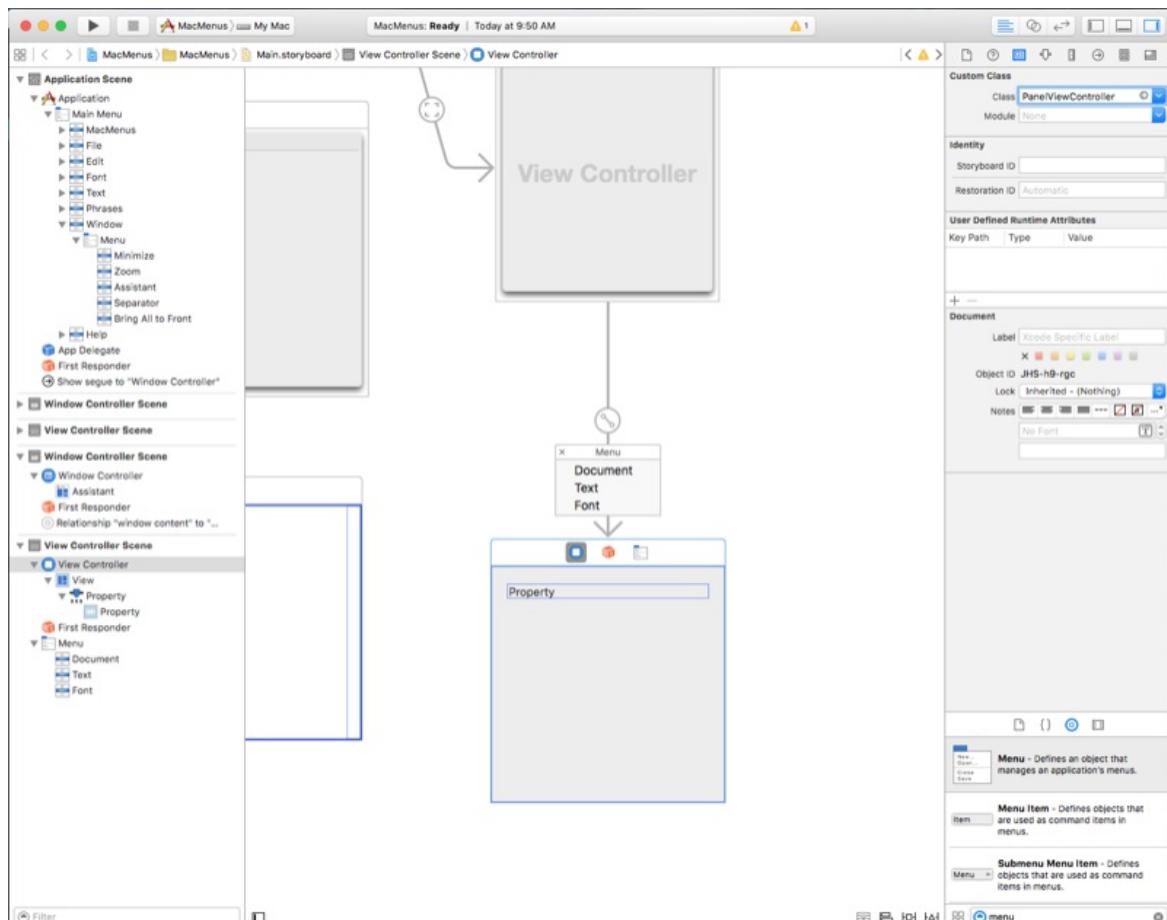


Let's do the following:

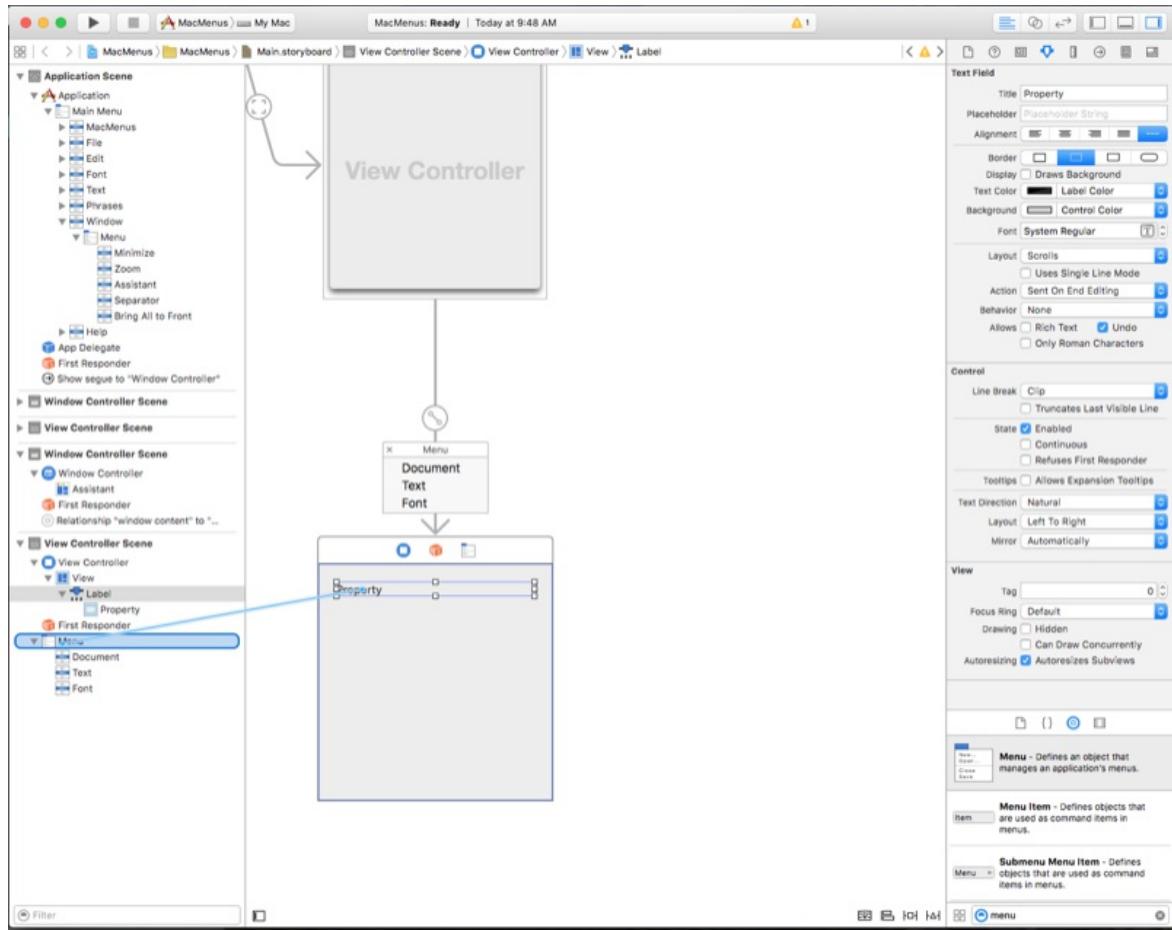
1. Drag a **Label** from the **Library Inspector** onto the **Panel** window and set its text to "Property":



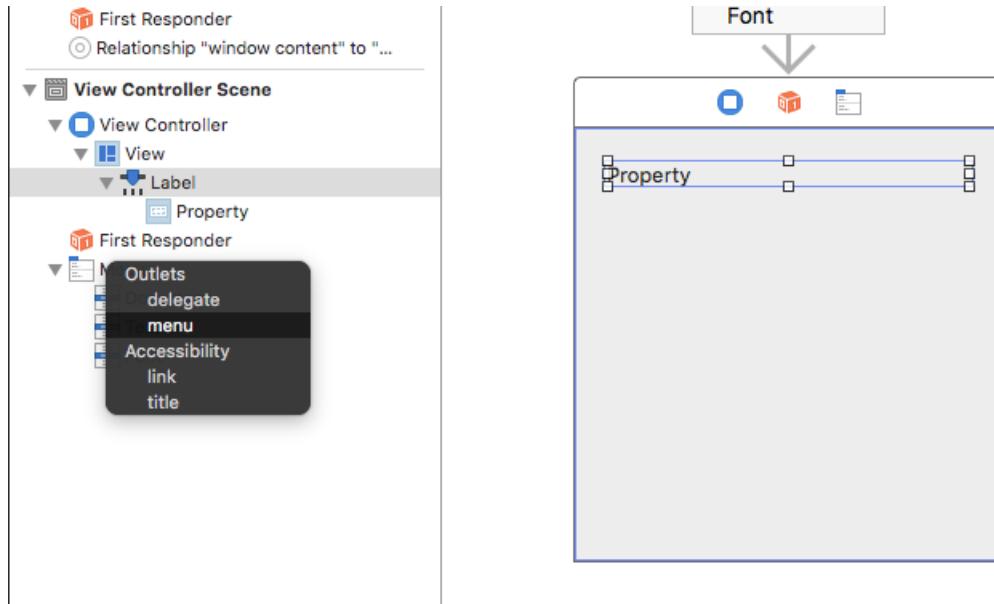
2. Next drag a **Menu** from the **Library Inspector** onto the View Controller in the View Hierarchy and rename the three default menu items **Document**, **Text** and **Font**:



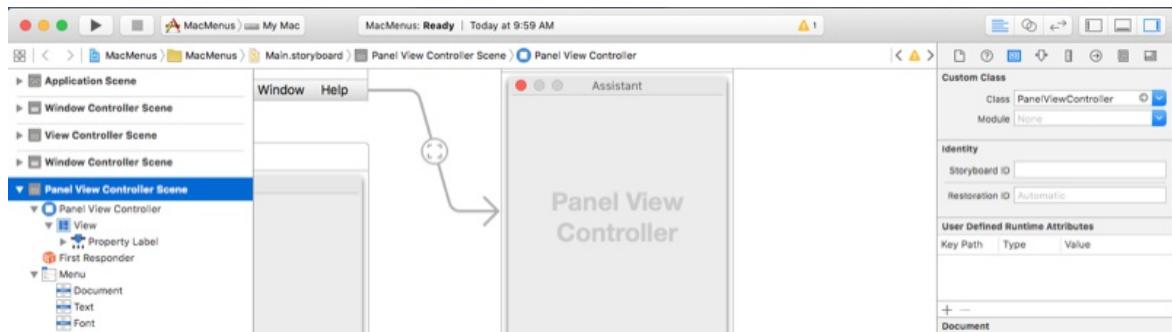
3. Now control-drag from the **Property Label** onto the **Menu**:



4. From the popup dialog, select **Menu**:



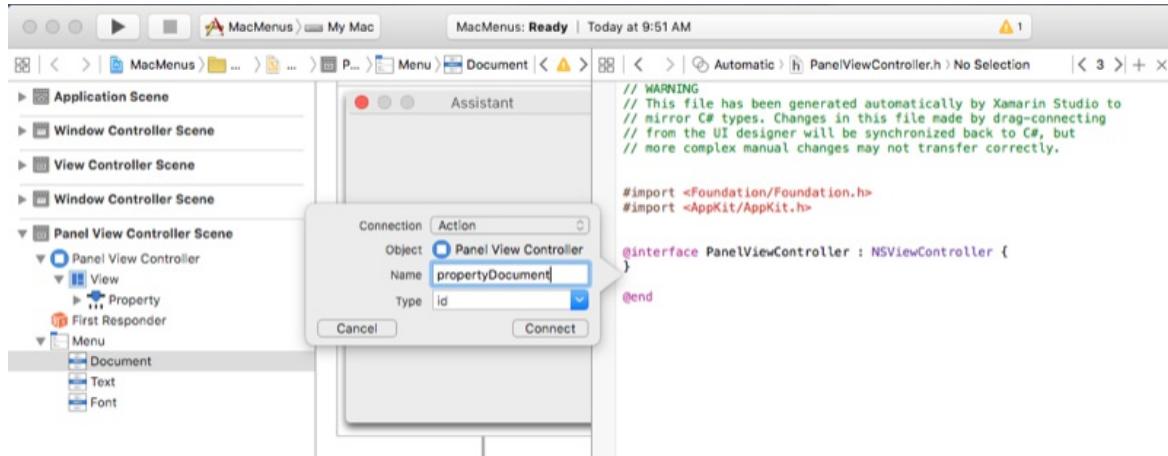
5. From the **Identity Inspector**, set the View Controller's class to "PanelViewController":



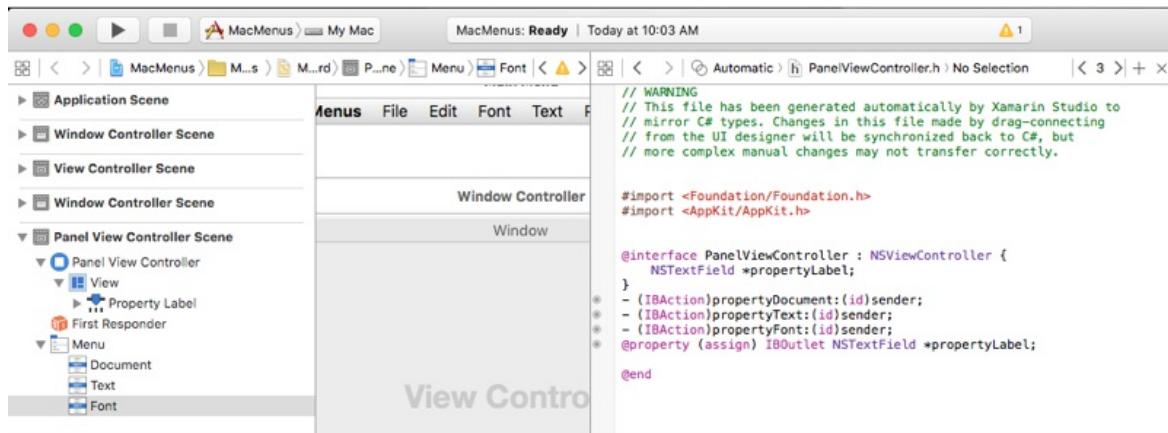
6. Switch back to Visual Studio for Mac to sync, then return to Interface Builder.

7. Switch to the Assistant Editor and select the `PanelViewController.h` file.

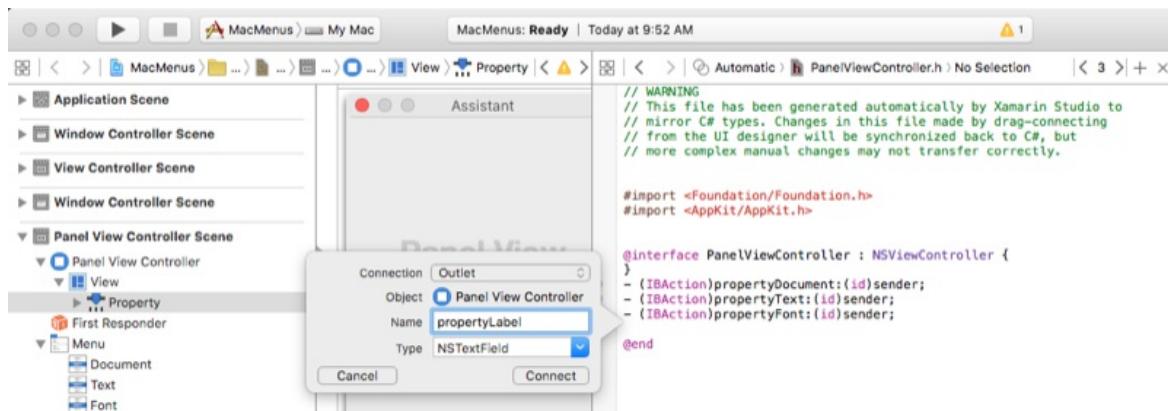
8. Create an action for the Document menu item called `propertyDocument`:



9. Repeat creating actions for the remaining menu items:



10. Finally create an outlet for the Property Label called `propertyLabel`:



11. Save your changes and return to Visual Studio for Mac to sync with Xcode.

Edit the `PanelViewController.cs` file and add the following code:

```

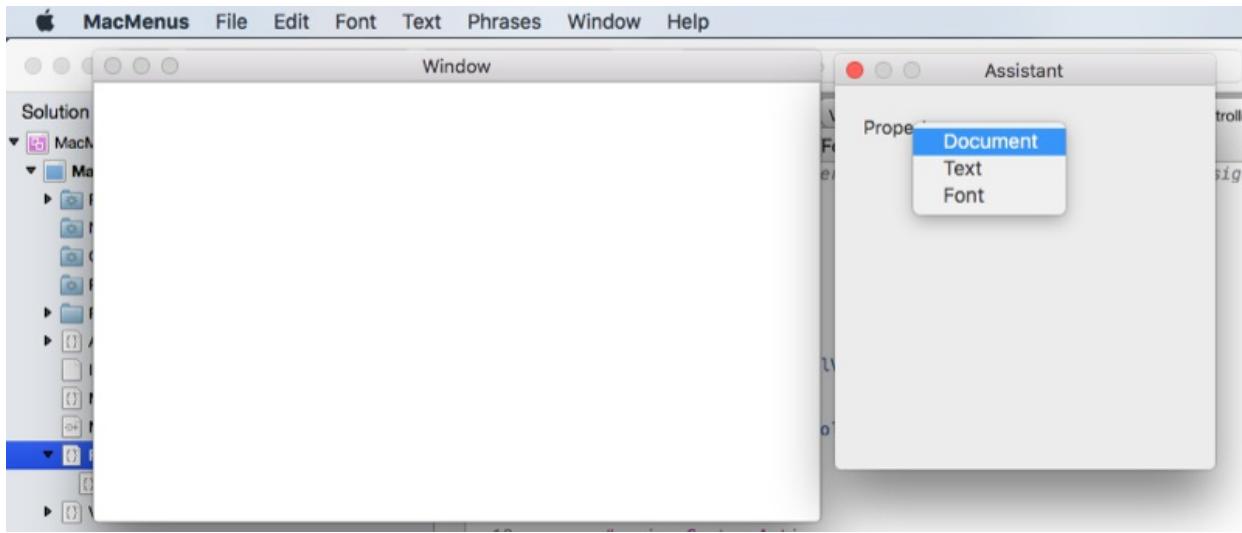
partial void propertyDocument (Foundation.NSObject sender) {
    propertyLabel.StringValue = "Document";
}

partial void propertyFont (Foundation.NSObject sender) {
    propertyLabel.StringValue = "Font";
}

partial void propertyText (Foundation.NSObject sender) {
    propertyLabel.StringValue = "Text";
}

```

Now if we run the application and right-click on the property label in the panel, we'll see our custom contextual menu. If we select an item from the menu, the label's value will change:



Next let's look at creating status bar menus.

Status bar menus

Status bar menus display a collection of status menu items that provide interaction with or feedback to the user, such as a menu or an image reflecting an application's state. An application's status bar menu is enabled and active even if the application is running in the background. The system-wide status bar resides at the right side of the application menu bar and is the only Status Bar currently available in macOS.

Let's edit our `AppDelegate.cs` file and make the `DidFinishLaunching` method look like the following:

```

public override void DidFinishLaunching (NSNotification notification)
{
    // Create a status bar menu
    NSSStatusBar statusBar = NSSStatusBar.SystemStatusBar;

    var item = statusBar.CreateStatusItem (NSStatusItemLength.Variable);
    item.Title = "Text";
    item.HighlightMode = true;
    item.Menu = new NSMenu ("Text");

    var address = new NSMenuItem ("Address");
    address.Activated += (sender, e) => {
        PhraseAddress(address);
    };
    item.Menu.AddItem (address);

    var date = new NSMenuItem ("Date");
    date.Activated += (sender, e) => {
        PhraseDate(date);
    };
    item.Menu.AddItem (date);

    var greeting = new NSMenuItem ("Greeting");
    greeting.Activated += (sender, e) => {
        PhraseGreeting(greeting);
    };
    item.Menu.AddItem (greeting);

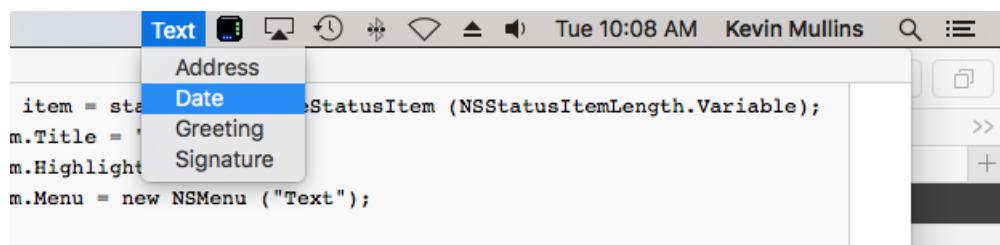
    var signature = new NSMenuItem ("Signature");
    signature.Activated += (sender, e) => {
        PhraseSignature(signature);
    };
    item.Menu.AddItem (signature);
}

```

`NSSStatusBar statusBar = NSSStatusBar.SystemStatusBar;` gives us access to the system-wide status bar.

`var item = statusBar.CreateStatusItem (NSStatusItemLength.Variable);` creates a new status bar item. From there we create a menu and a number of menu items and attach the menu to the status bar item we just created.

If we run the application, the new status bar item will be displayed. Selecting an item from the menu will change the text in the text view:



Next, let's look at creating custom dock menu items.

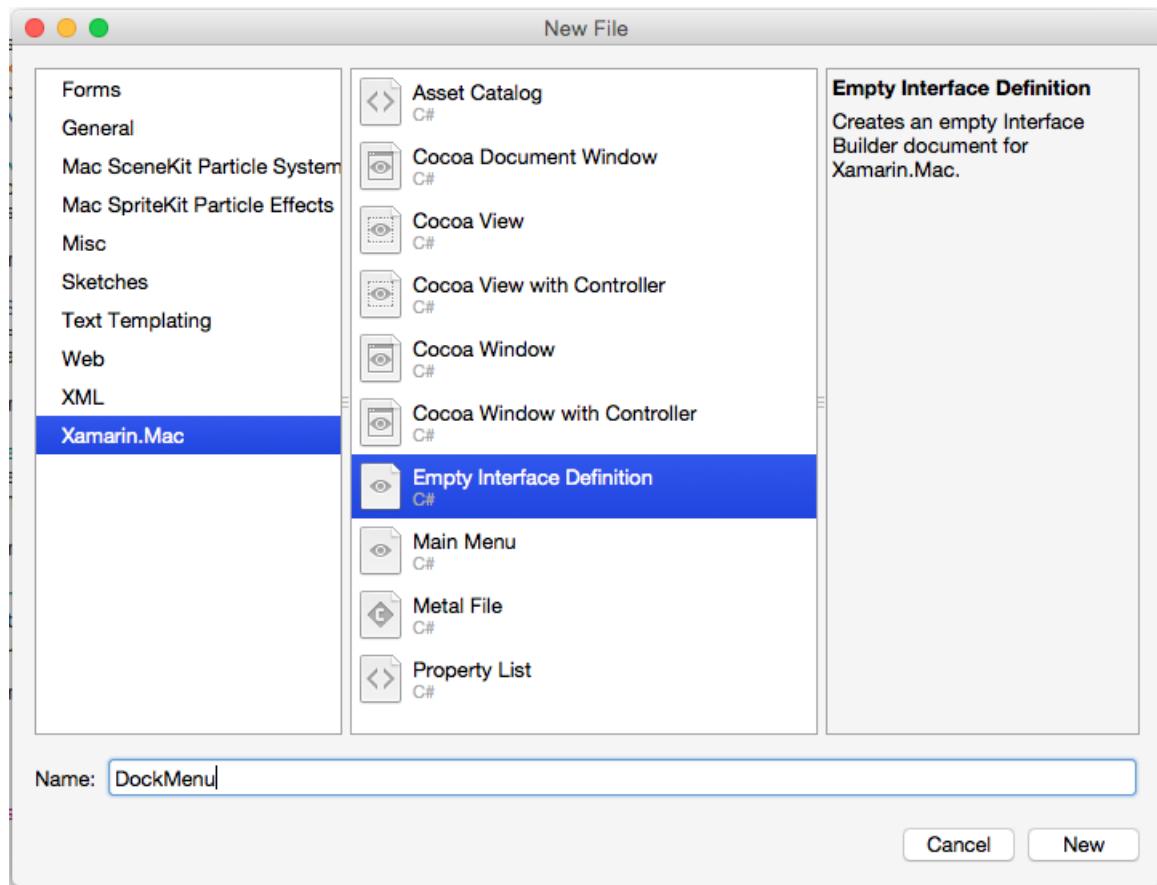
Custom dock menus

The dock menu appears for your Mac application when the user right-clicks or control-clicks the application's icon in the dock:

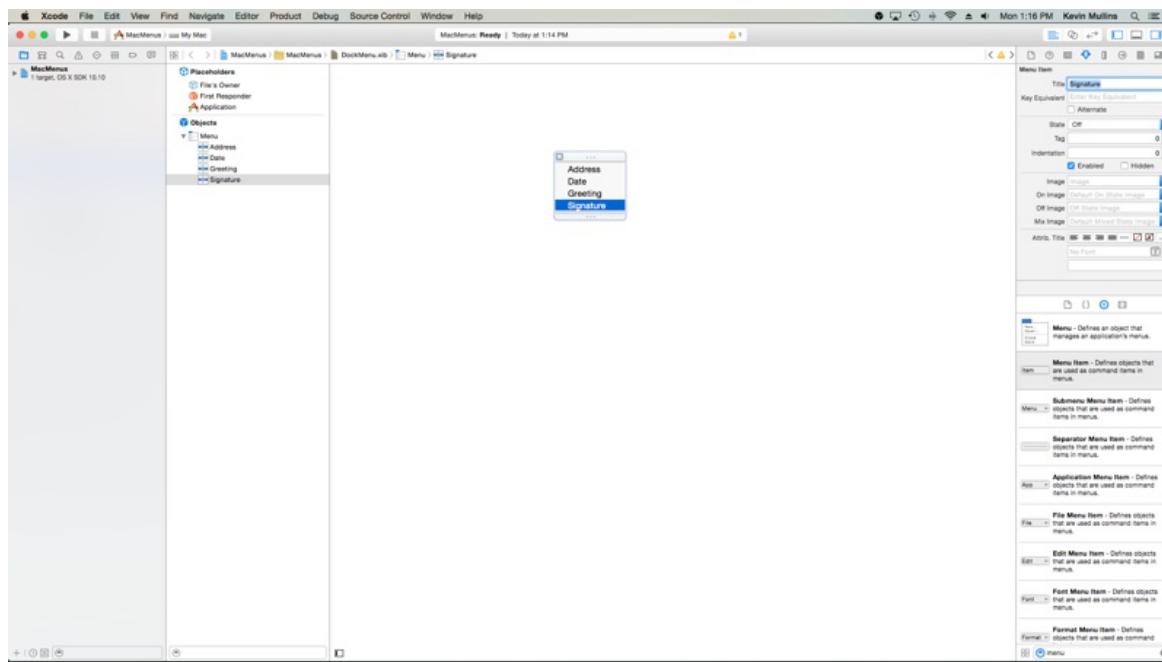


Let's create a custom dock menu for our application by doing the following:

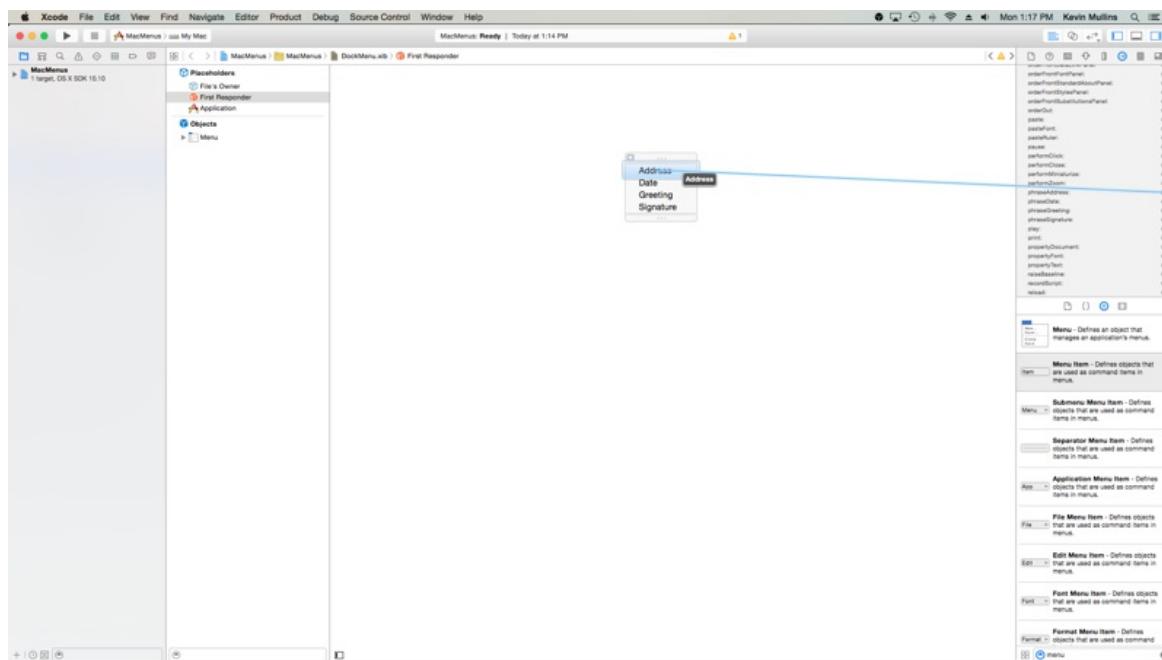
1. In Visual Studio for Mac, right-click on the application's project and select **Add > New File...**. From the new file dialog, select **Xamarin.Mac > Empty Interface Definition**, use "DockMenu" for the **Name** and click the **New** button to create the new **DockMenu.xib** file:



2. In the **Solution Pad**, double-click the **DockMenu.xib** file to open it for editing in Xcode. Create a new **Menu** with the following items: **Address**, **Date**, **Greeting**, and **Signature**



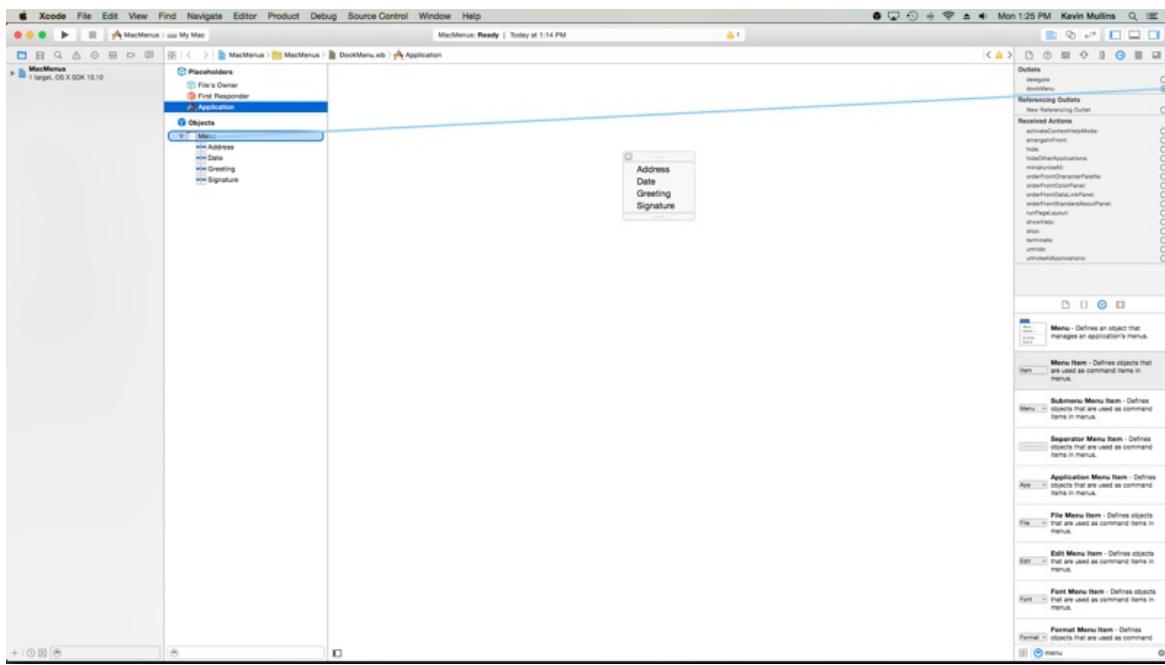
3. Next, let's connect our new menu items to our existing actions that we created for our custom menu in the [Adding, Editing and Deleting Menus](#) section above. Switch to the **Connection Inspector** and select the **First Responder** in the **Interface Hierarchy**. Scroll down and find the `phraseAddress:` action. Drag a line from the circle on that action to the **Address** menu item:



4. Repeat for all of the other menu items attaching them to their corresponding actions:

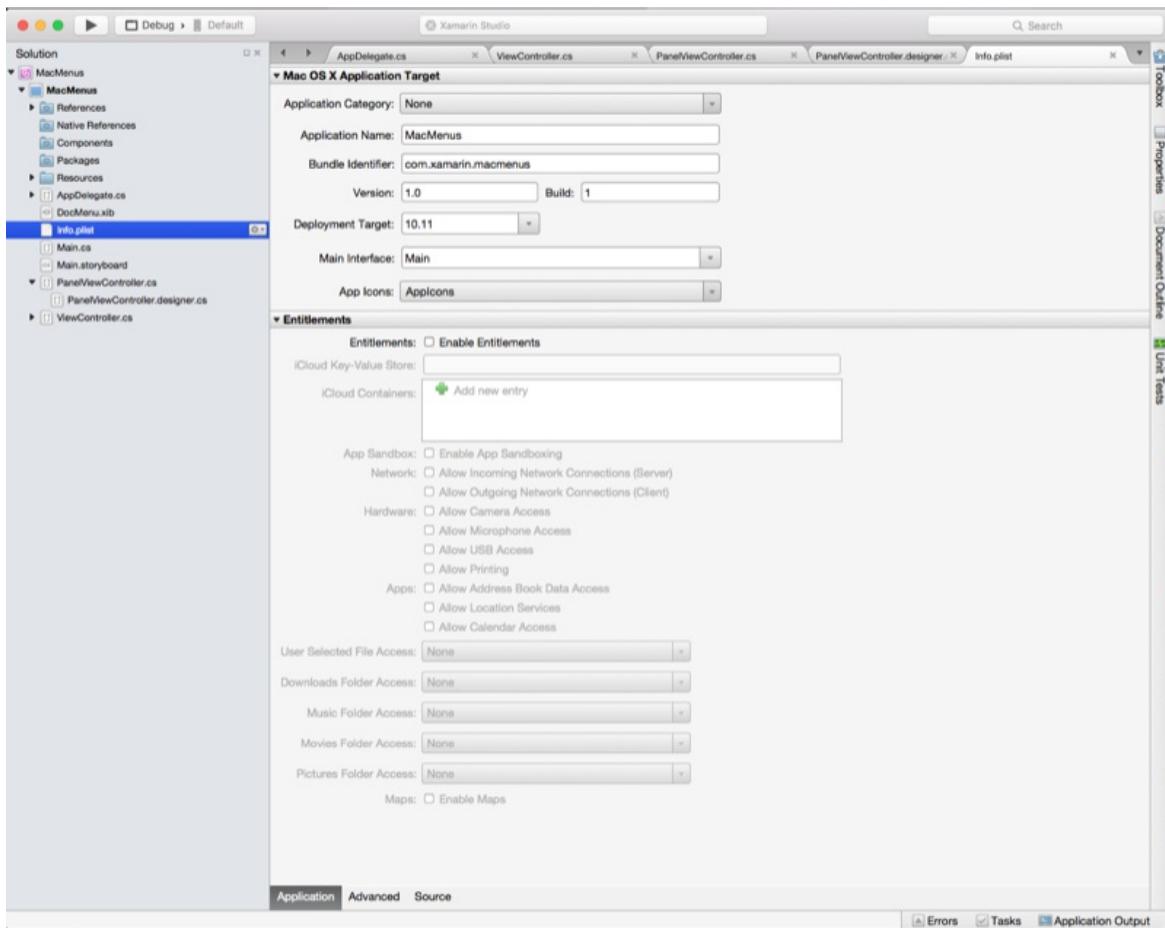


5. Next, select the **Application** in the **Interface Hierarchy**. In the **Connection Inspector**, drag a line from the circle on the `dockMenu` outlet to the menu we just created:

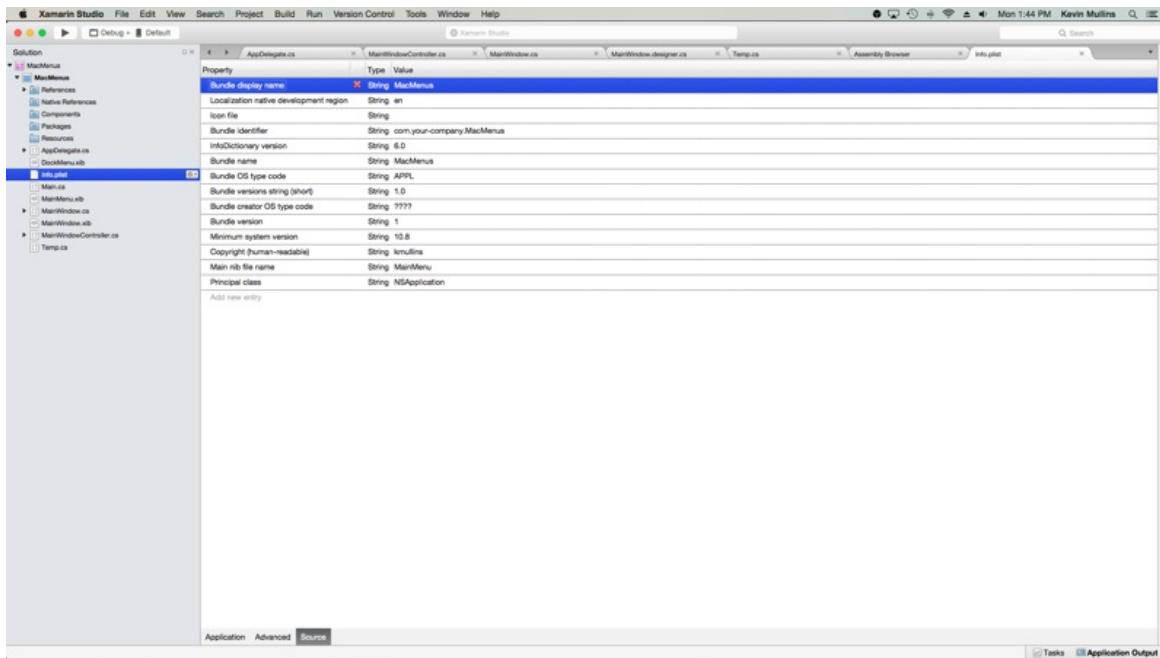


6. Save your changes and switch back to Visual Studio for Mac to sync with Xcode.

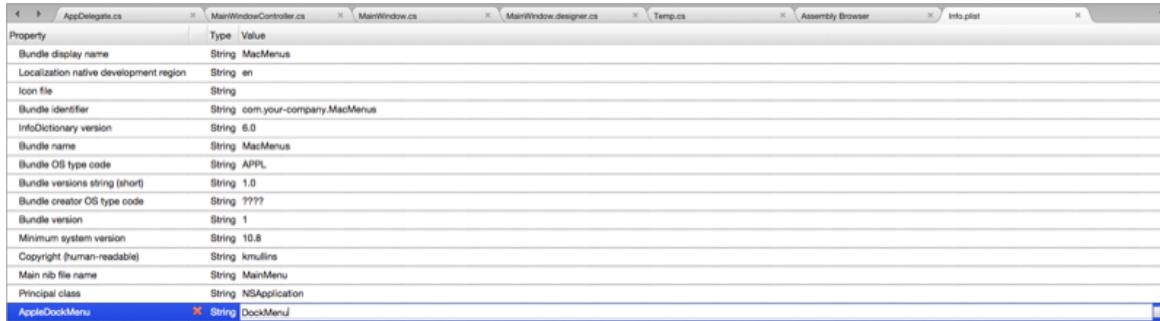
7. Double-click the **Info.plist** file to open it for editing:



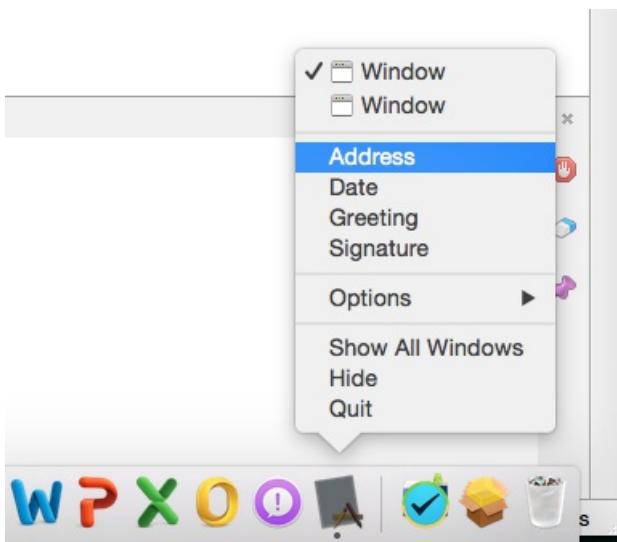
8. Click the **Source** tab at the bottom of the screen:



9. Click **Add new entry**, click the green plus button, set the property name to "AppleDockMenu" and the value to "DockMenu" (the name of our new .xib file without the extension):



Now if we run our application and right-click on its icon in the Dock, our new menu items will be displayed:



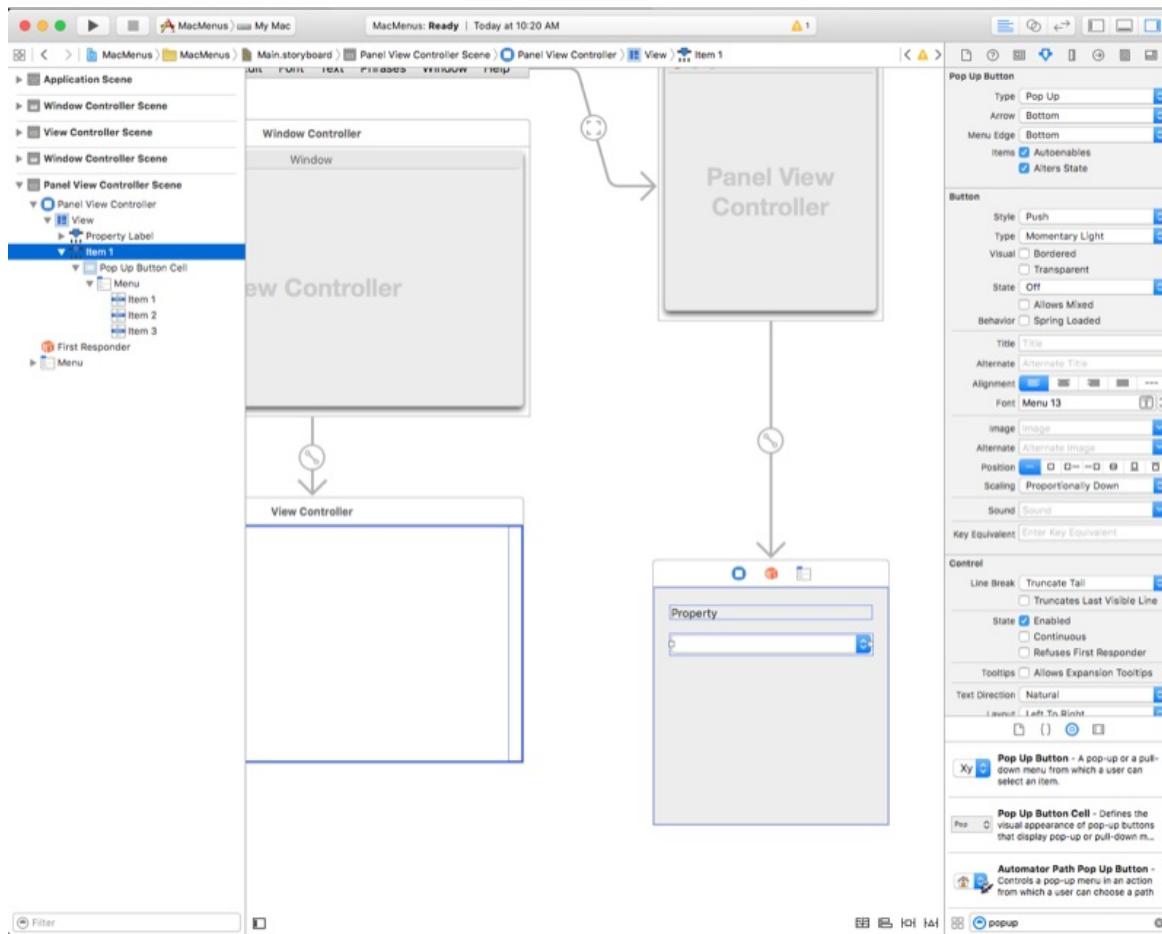
If we select one of the custom items from the menu, the text in our text view will be modified.

Pop-up button and pull-down lists

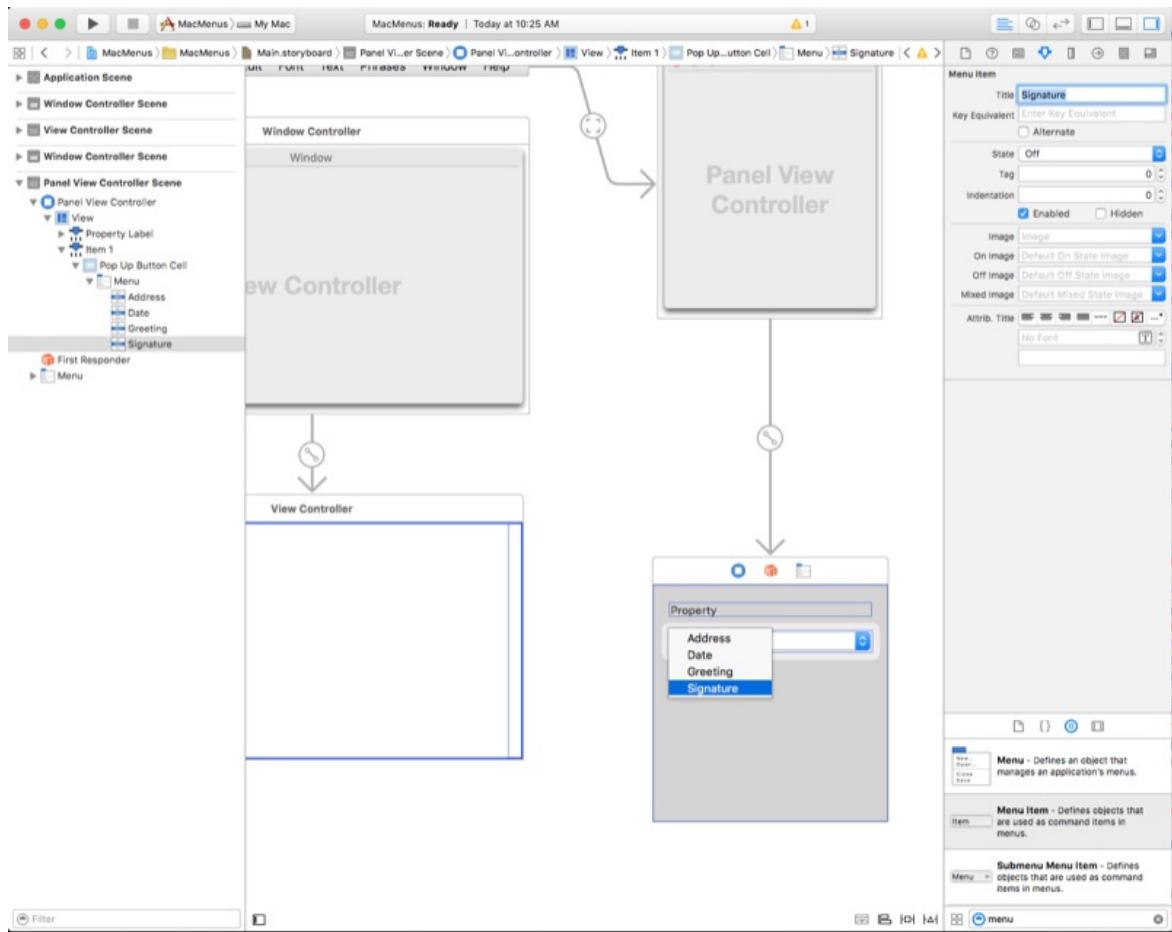
A pop-up button displays a selected item and presents a list of options to select from when clicked by the user. A pull-down list is a type of pop-up button usually used for selecting commands specific to the context of the current task. Both can appear anywhere in a window.

Let's create a custom pop-up button for our application by doing the following:

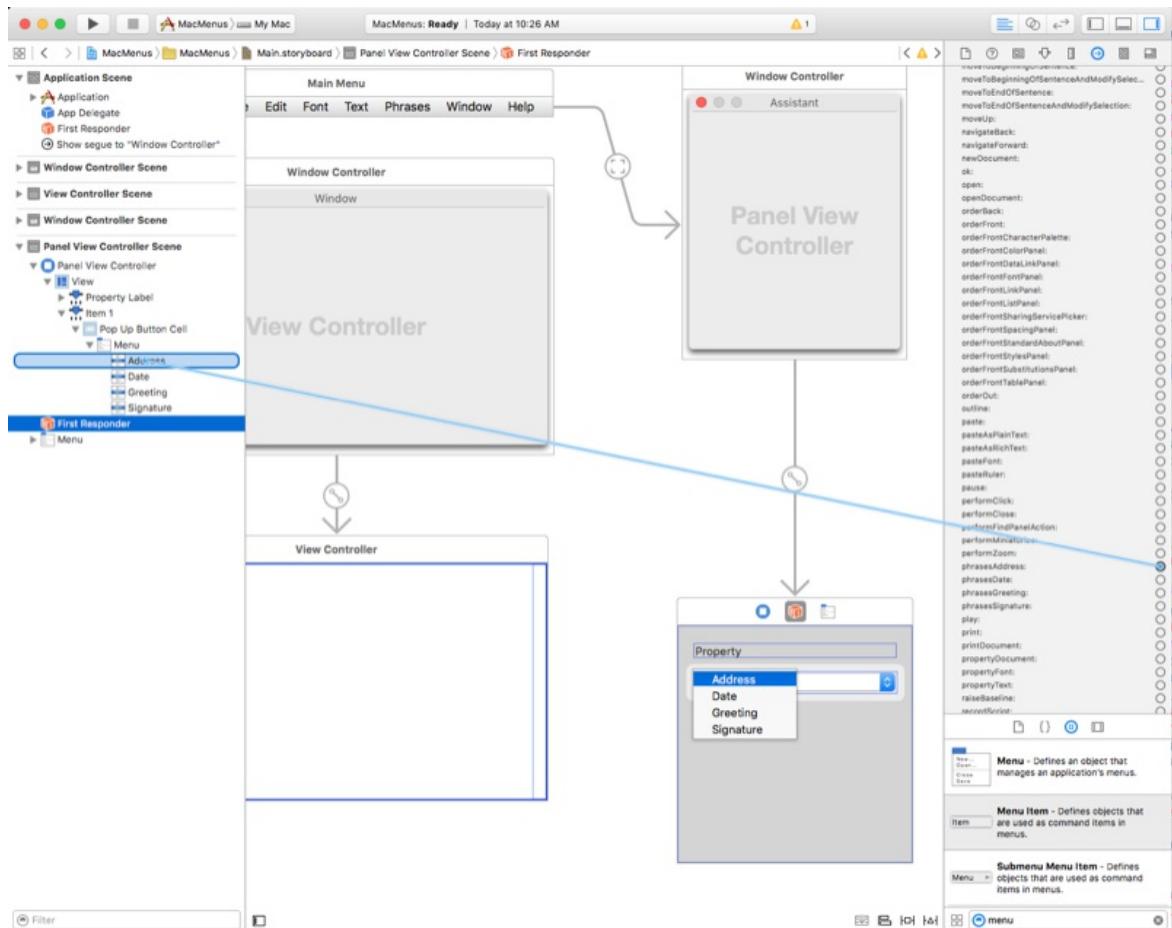
1. Edit the **Main.storyboard** file in Xcode and drag a **Popup Button** from the **Library Inspector** onto the **Panel** window we created in the **Contextual Menus** section:



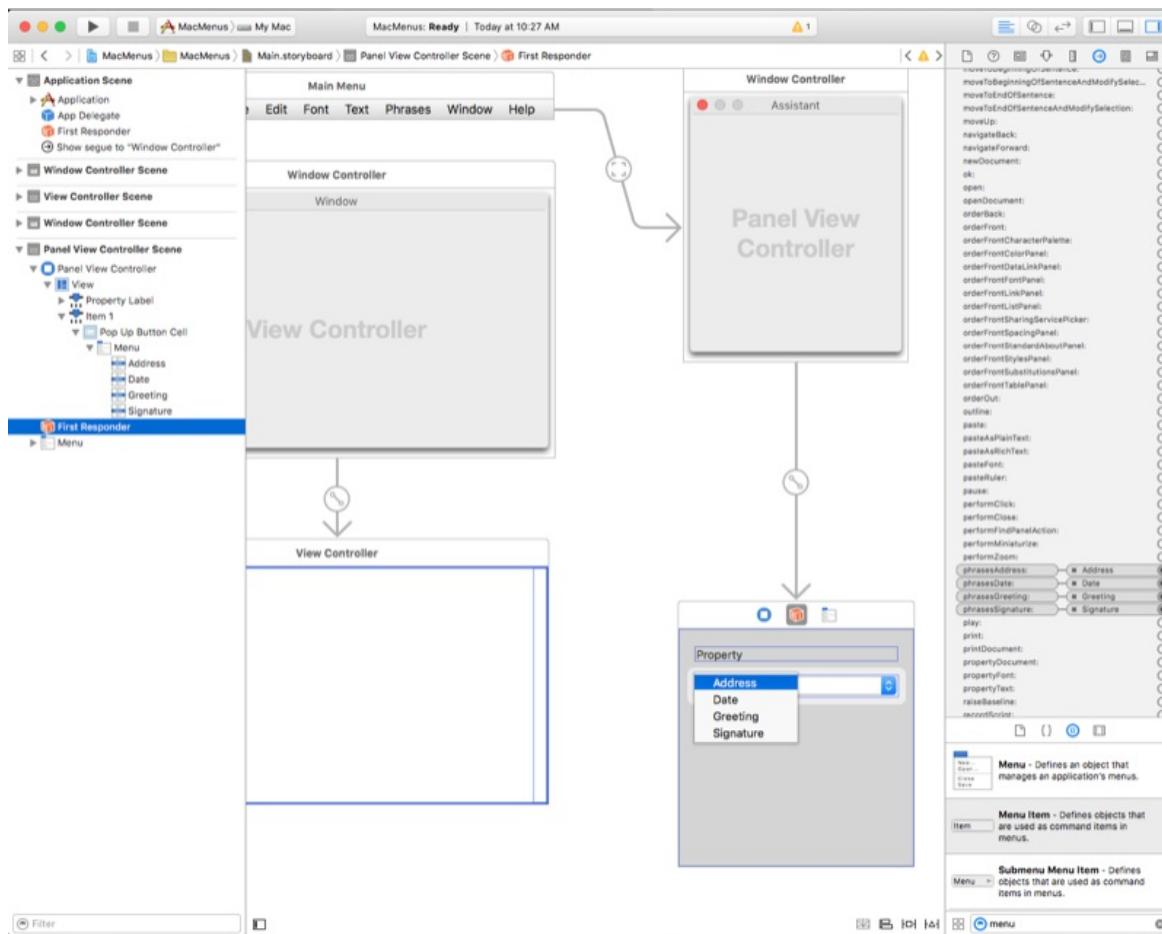
2. Add a new menu item and set the titles of the Items in the Popup to: **Address**, **Date**, **Greeting**, and **Signature**



3. Next, let's connect our new menu items to the existing actions that we created for our custom menu in the [Adding, Editing and Deleting Menus](#) section above. Switch to the **Connection Inspector** and select the **First Responder** in the **Interface Hierarchy**. Scroll down and find the `phraseAddress:` action. Drag a line from the circle on that action to the **Address** menu item:

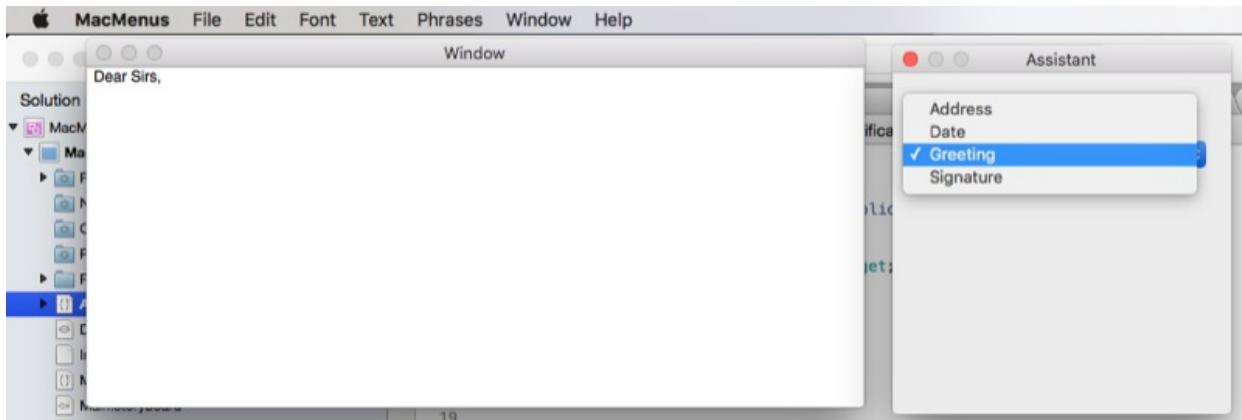


4. Repeat for all of the other menu items attaching them to their corresponding actions:



5. Save your changes and switch back to Visual Studio for Mac to sync with Xcode.

Now if we run our application and select an item from the popup, the text in our text view will change:



You can create and work with pull-down lists in the exact same way as pop-up buttons. Instead of attaching to existing action, you could create your own custom actions just like we did for our contextual menu in the [Contextual Menus](#) section.

Summary

This article has taken a detailed look at working with menus and menu items in a Xamarin.Mac application. First we examined the application's menu bar, then we looked at creating contextual menus, next we examined status bar menus and custom dock menus. Finally, we covered pop-up menus and pull-down Lists.

Related Links

- [MacMenus \(sample\)](#)
- [Hello, Mac](#)
- [Human Interface Guidelines - Menus](#)
- [Introduction to Application Menus and Pop-up Lists](#)

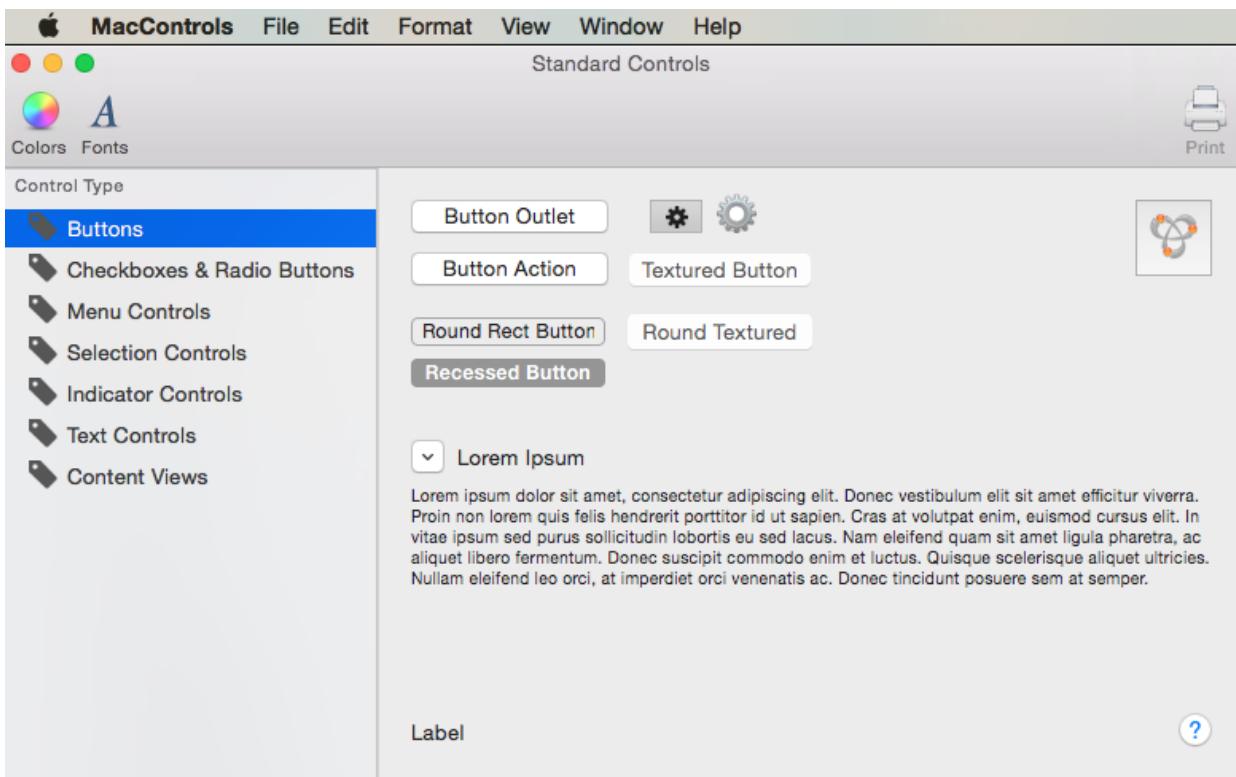
Standard controls in Xamarin.Mac

11/2/2020 • 19 minutes to read • [Edit Online](#)

This article covers working with the standard AppKit controls such as buttons, labels, text fields, check boxes, and segmented controls in a Xamarin.Mac application. It describes adding them to an interface with Interface Builder and interacting with them in code.

When working with C# and .NET in a Xamarin.Mac application, you have access to the same AppKit Controls that a developer working in *Objective-C* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, you can use Xcode's *Interface Builder* to create and maintain your Appkit Controls (or optionally create them directly in C# code).

AppKit Controls are the UI Elements that are used to create the User Interface of your Xamarin.Mac application. They consist of elements such as Buttons, Labels, Text Fields, Check Boxes and Segmented Controls and cause instant actions or visible results when a user manipulates them.



In this article, we'll cover the basics of working with AppKit Controls in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` commands used to wire-up your C# classes to Objective-C objects and UI Elements.

Introduction to Controls and Views

macOS (formerly known as Mac OS X) provides a standard set of User Interface controls via the AppKit Framework. They consist of elements such as Buttons, Labels, Text Fields, Check Boxes and Segmented Controls and cause instant actions or visible results when a user manipulates them.

All of the AppKit Controls have a standard, built-in appearance that will be appropriate for most uses, some specify an alternate appearance for use in a window frame area or in a *Vibrance Effect* context, such as in a Sidebar area or in a Notification Center widget.

Apple suggest the following guidelines when working with AppKit Controls:

- Avoid mixing control sizes in the same view.
- In general, avoid resizing controls vertically.
- Use the system font and the proper text size within a control.
- Use the proper spacing between controls.

For more information, please see the [About Controls and Views](#) section of Apple's [OS X Human Interface Guidelines](#).

Using Controls in a Window Frame

There are a subset of AppKit Controls that include a display style that allows them to be included in a Window's Frame area. For an example, see the Mail app's toolbar:



- **Round Textured Button** - A `NSButton` with a style of `NSTexturedRoundedBezelStyle`.
- **Textured Rounded Segmented Control** - A `NSSegmentedControl` with a style of `NSSegmentStyleTexturedRounded`.
- **Textured Rounded Segmented Control** - A `NSSegmentedControl` with a style of `NSSegmentStyleSeparated`.
- **Round Textured Pop-Up Menu** - A `NSPopUpButton` with a style of `NSTexturedRoundedBezelStyle`.
- **Round Textured Drop-Down Menu** - A `NSPopUpButton` with a style of `NSTexturedRoundedBezelStyle`.
- **Search Bar** - A `NSSearchField`.

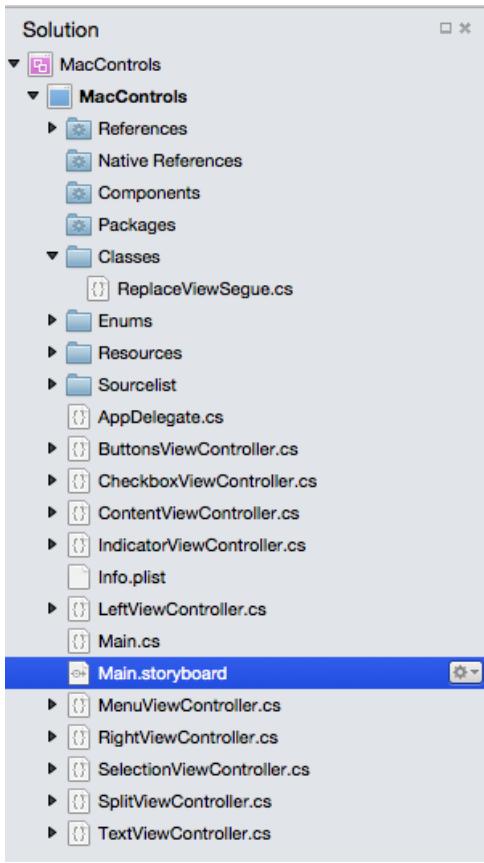
Apple suggest the following guidelines when working with AppKit Controls in a Window Frame:

- Don't use Window Frame specific control styles in the Window Body.
- Don't use Window Body controls or styles in the Window Frame.

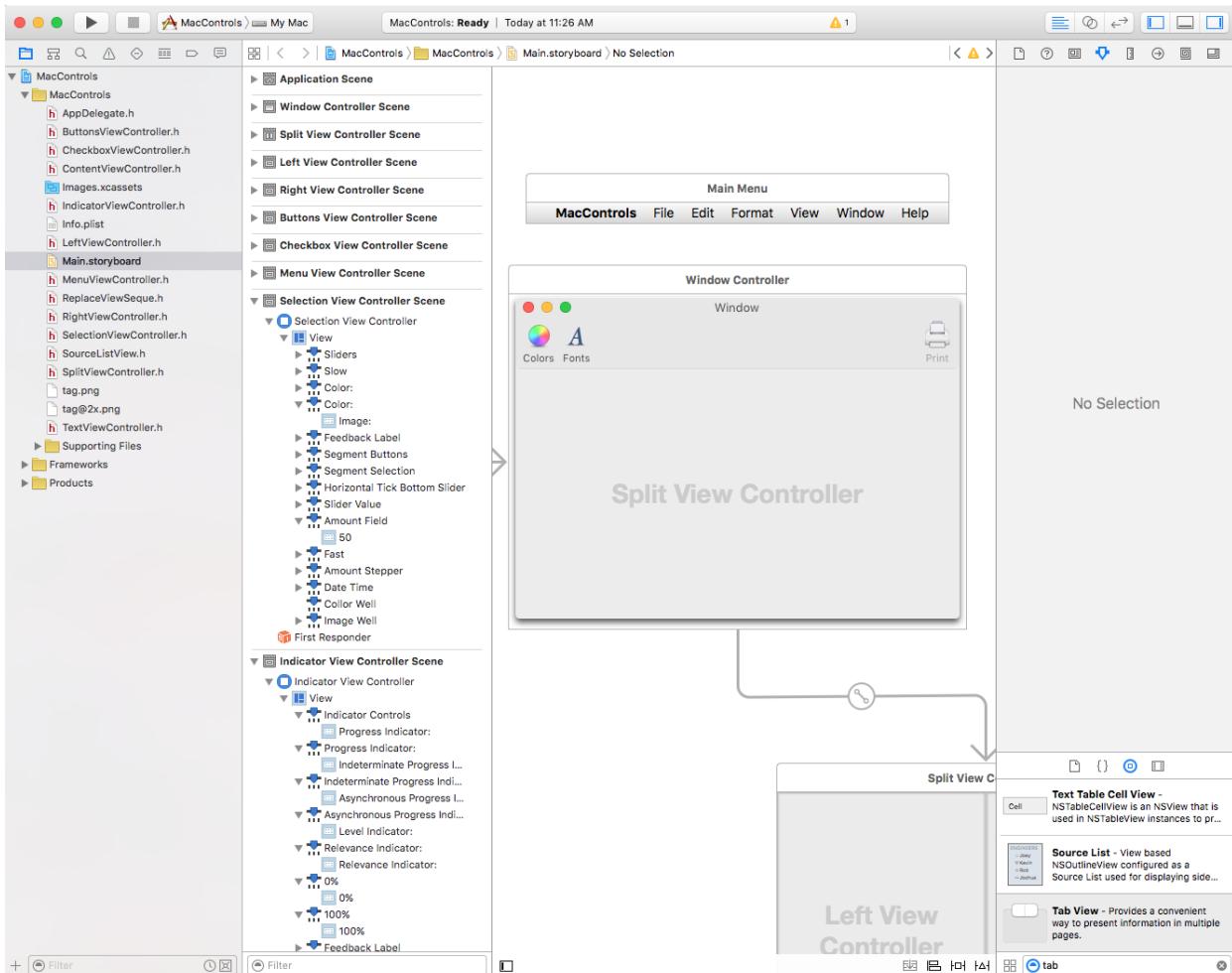
For more information, please see the [About Controls and Views](#) section of Apple's [OS X Human Interface Guidelines](#).

Creating a User Interface in Interface Builder

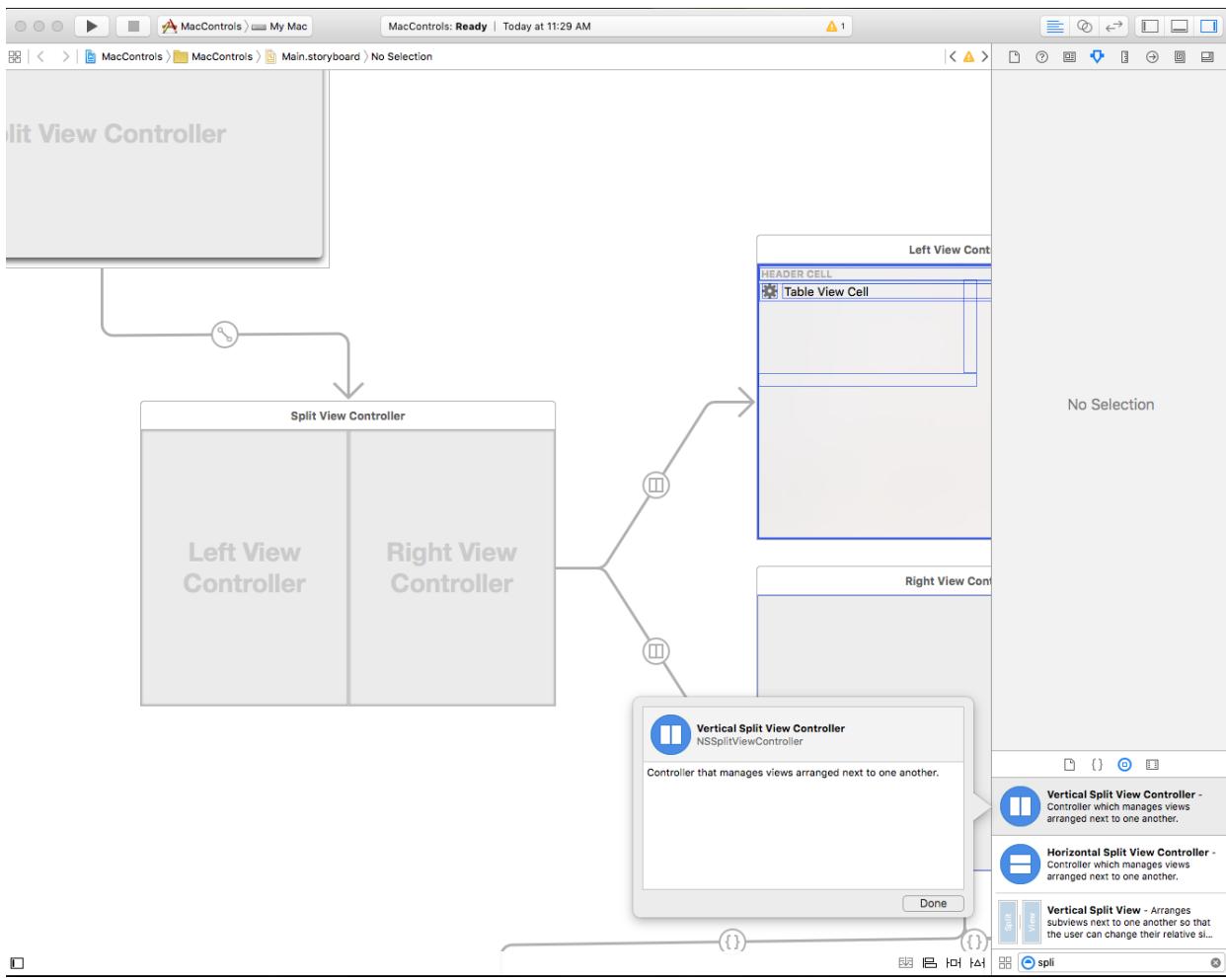
When you create a new Xamarin.Mac Cocoa application, you get a standard blank, window by default. This windows is defined in a `.Storyboard` file automatically included in the project. To edit your windows design, in the **Solution Explorer**, double click the `Main.storyboard` file:



This will open the window design in Xcode's Interface Builder:



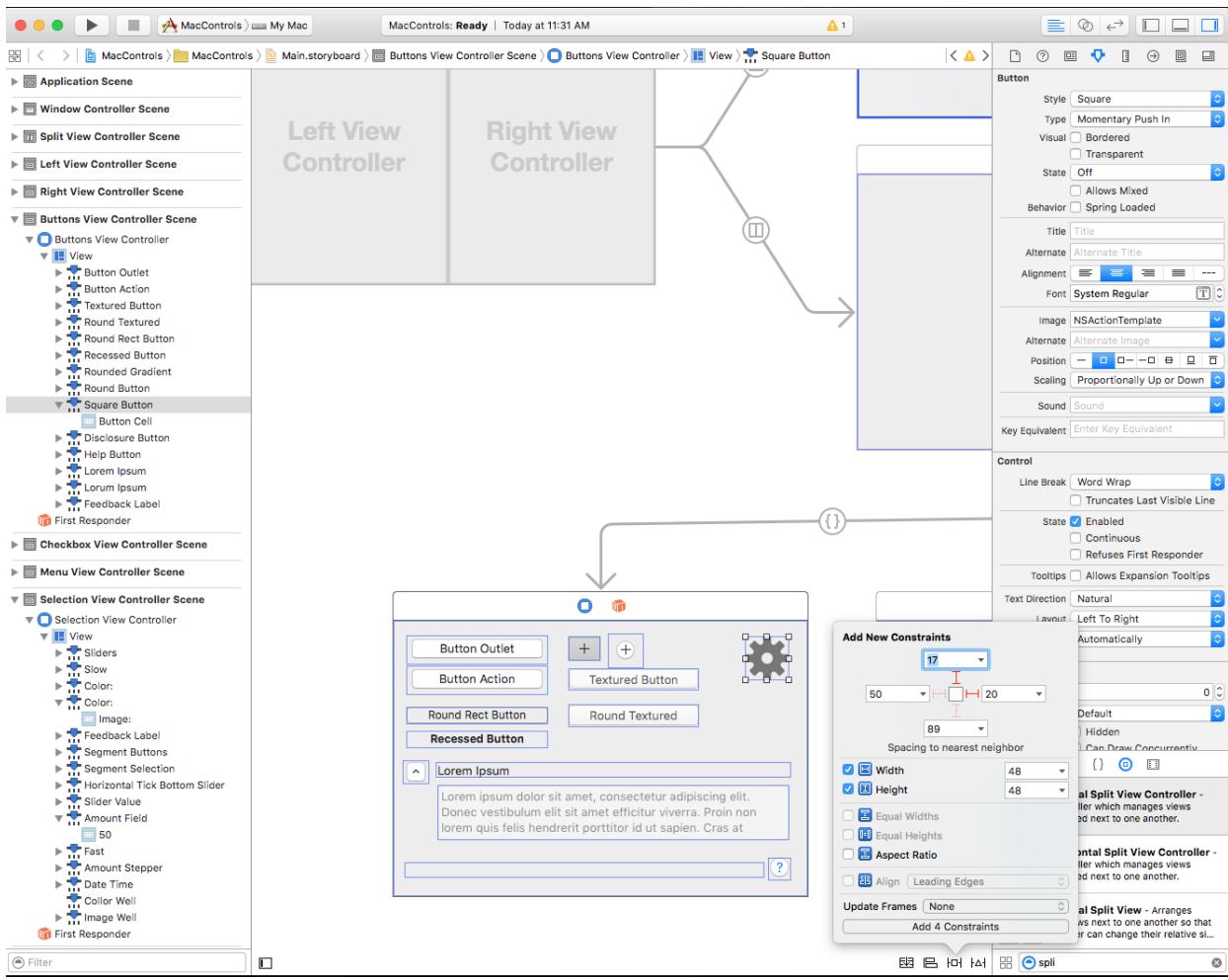
To create your User Interface, you'll drag UI Elements (AppKit Controls) from the **Library Inspector** to the **Interface Editor** in Interface Builder. In the example below, a **Vertical Split View** control has been drug from the **Library Inspector** and placed on the Window in the **Interface Editor**:



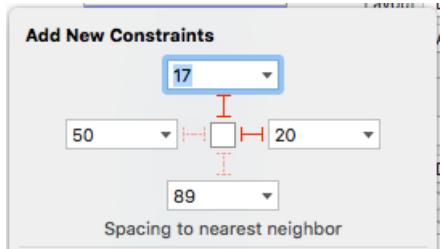
For more information on creating a User Interface in Interface Builder, please see our [Introduction to Xcode and Interface Builder](#) documentation.

Sizing and Positioning

Once a control has been included in the User Interface, use the **Constraint editor** to set its location and size by entering values manually and control how the control is automatically positioned and sized when the parent Window or View is resized:

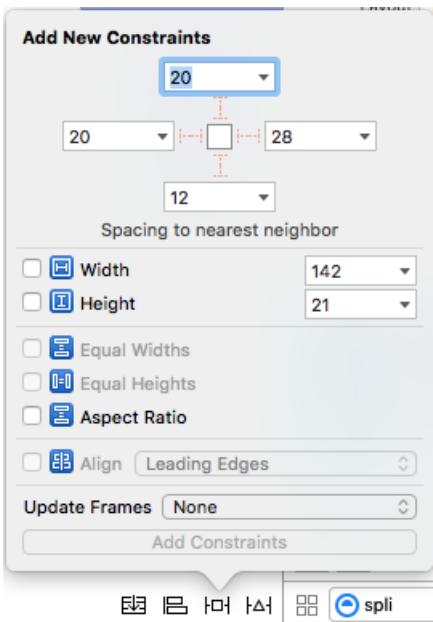


Use the Red I-Beams around the outside of the Autoresizing box to *stick* a control to a given (x,y) location. For example:

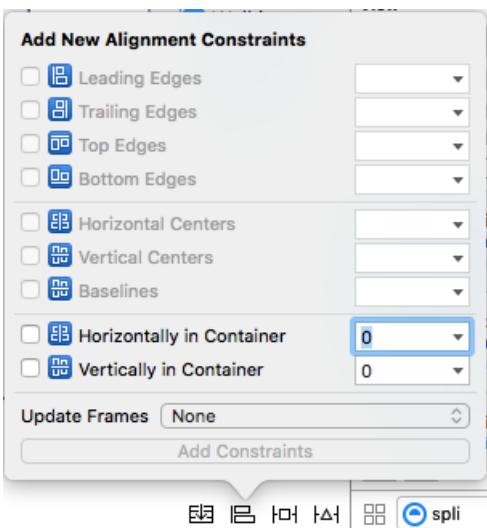


Specifies that the selected control (in the **Hierarchy View & Interface Editor**) will be stuck to the top and right location of the Window or View as it is resized or moved.

Other elements of the editor control properties such as Height and Width:



You can also control the alignment of elements with constraints using the **Alignment Editor**:



IMPORTANT

Unlike iOS where (0,0) is the upper left hand corner of the screen, in macOS (0,0) is the lower left hand corner. This is because macOS uses a mathematical coordinate system with the number values increasing in value upward and to the right. You need to take this into consideration when placing AppKit controls on a User Interface.

Setting a Custom Class

There are times when working with AppKit Controls that you will need to subclass an existing control and create your own custom version of that class. For example, defining a custom version of the Source List:

```
using System;
using AppKit;
using Foundation;

namespace AppKit
{
    [Register("SourceListView")]
    public class SourceListView : NSOutlineView
    {
        #region Computed Properties
        public SourceListDataSource Data {
            get {return (SourceListDataSource)this.DataSource; }
        }
    }
}
```

```

        }

    #endregion

    #region Constructors
    public SourceListView ()
    {

    }

    public SourceListView (IntPtr handle) : base(handle)
    {

    }

    public SourceListView (NSCoder coder) : base(coder)
    {

    }

    public SourceListView (NSObjectFlag t) : base(t)
    {

    }
    #endregion

    #region Override Methods
    public override void AwakeFromNib ()
    {
        base.AwakeFromNib ();
    }
    #endregion

    #region Public Methods
    public void Initialize() {

        // Initialize this instance
        this.DataSource = new SourceListDataSource (this);
        this.Delegate = new SourceListDelegate (this);

    }

    public void AddItem(SourceListItem item) {
        if (Data != null) {
            Data.Items.Add (item);
        }
    }
    #endregion

    #region Events
    public delegate void ItemSelectedDelegate(SourceListItem item);
    public event ItemSelectedDelegate ItemSelected;

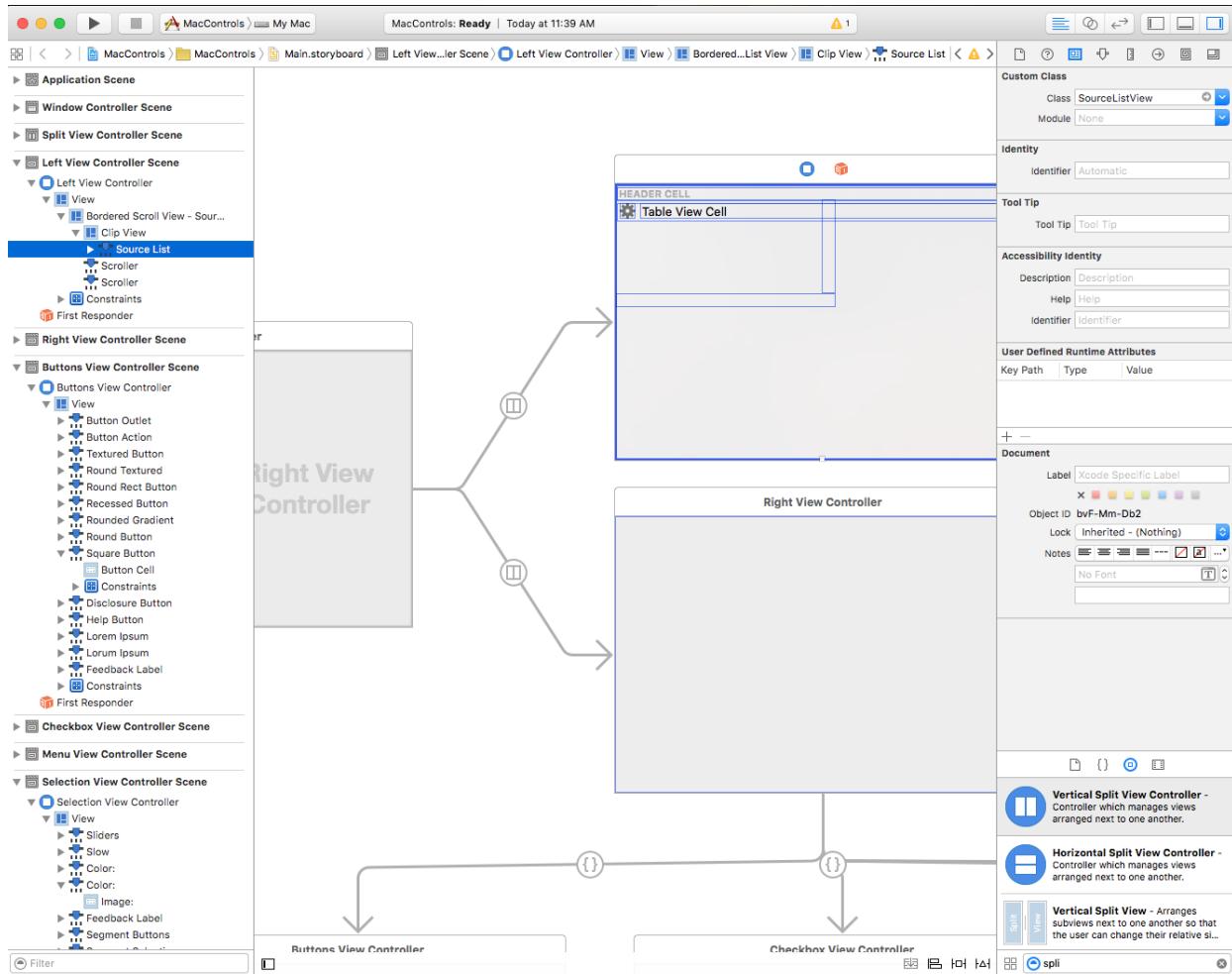
    internal void RaiseItemSelected(SourceListItem item) {
        // Inform caller
        if (this.ItemSelected != null) {
            this.ItemSelected (item);
        }
    }
    #endregion
}
}

```

Where the `[Register("SourceListView")]` instruction exposes the `SourceListView` class to Objective-C so that it can be used in Interface Builder. For more information, please see the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document, it explains the `Register` and `Export` commands

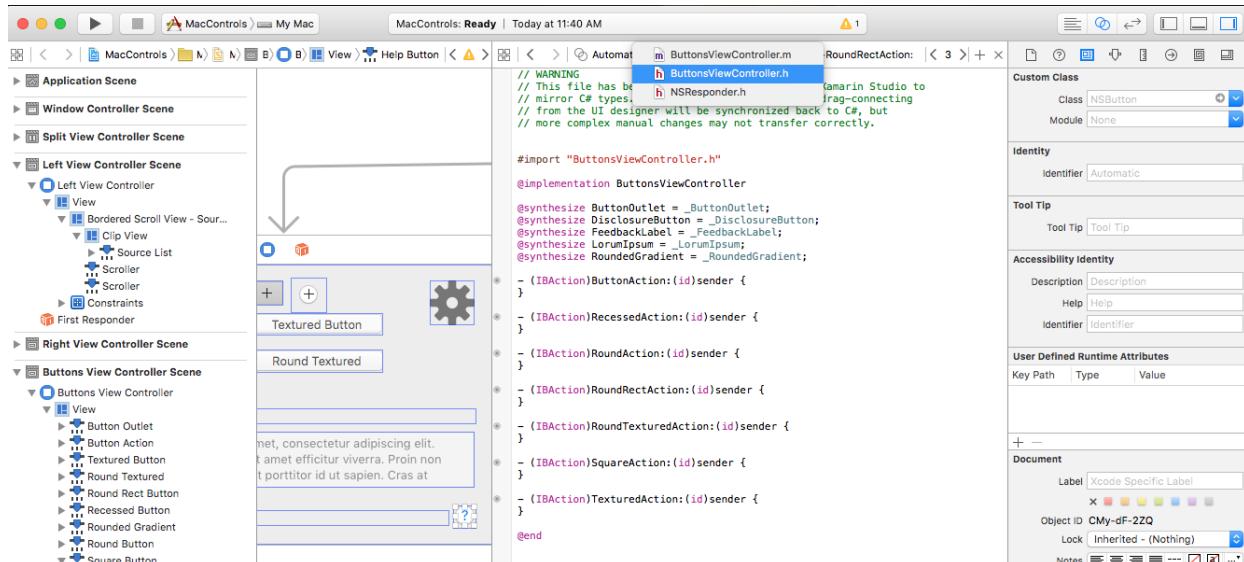
used to wire-up your C# classes to Objective-C objects and UI Elements.

With the above code in place, you can drag an AppKit Control, of the base type that you are extending, onto the design surface (in the example below, a **Source List**), switch to the **Identity Inspector** and set the **Custom Class** to the name that you exposed to Objective-C (example `SourceListView`):

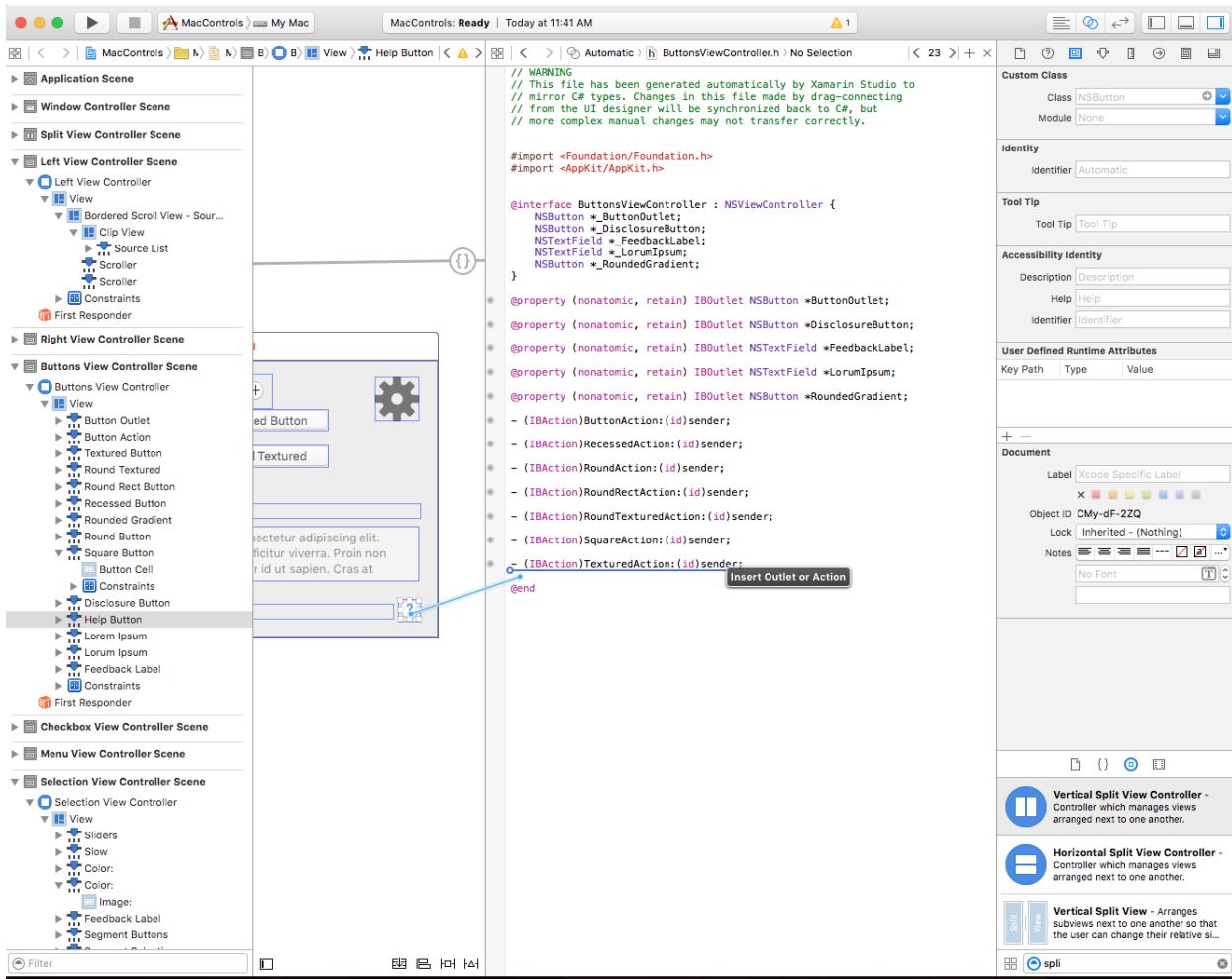


Exposing Outlets and Actions

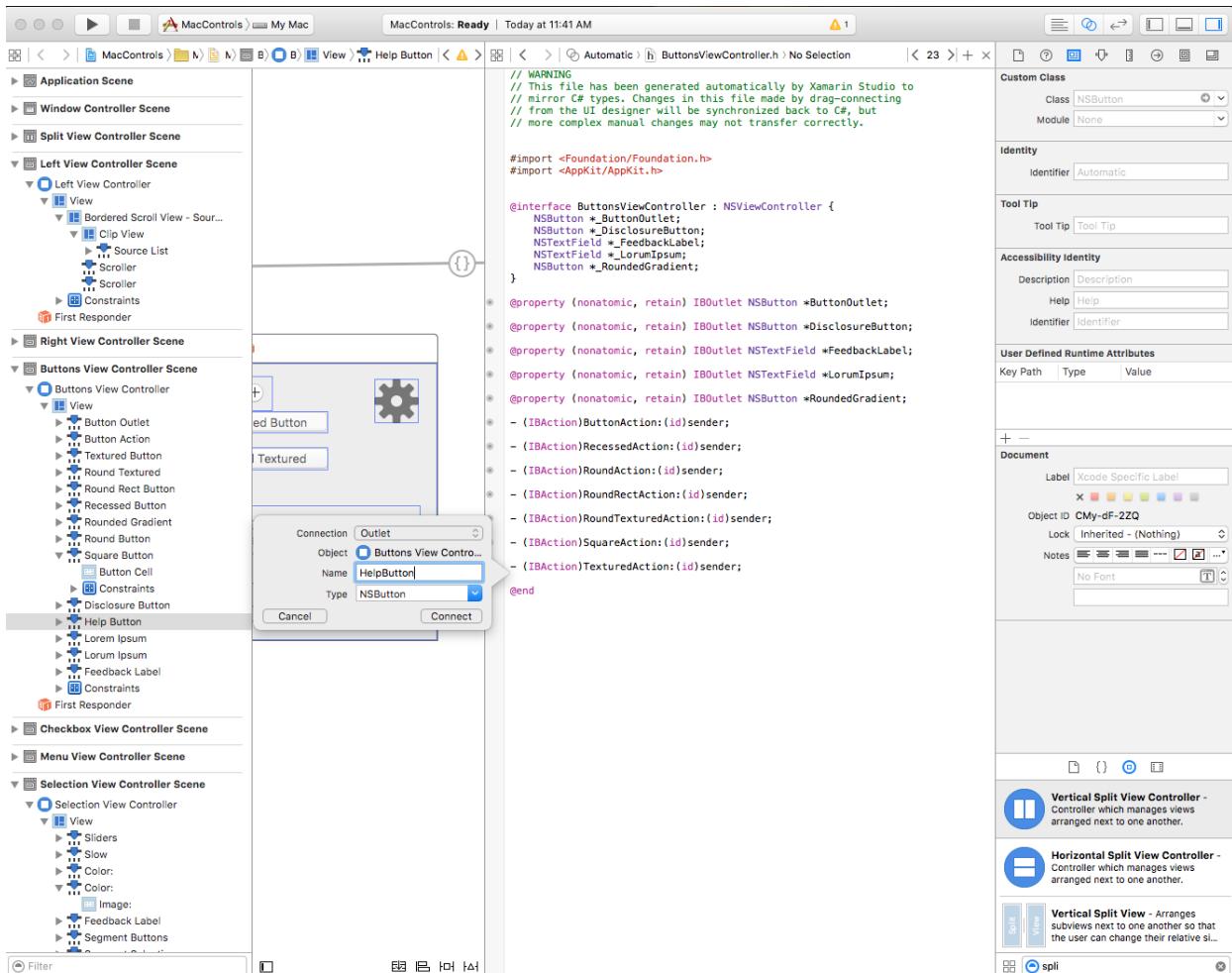
Before an AppKit Control can be accessed in C# code, it needs to be exposed as either an **Outlet** or an **Action**. To do this select the given control in either the **Interface Hierarchy** or the **Interface Editor** and switch to the **Assistant View** (ensure that you have the `.h` of your Window selected for editing):



Control-drag from the AppKit control onto the give `.h` file to start creating an **Outlet** or **Action**:



Select the type of exposure to create and give the **Outlet** or **Action** a Name:

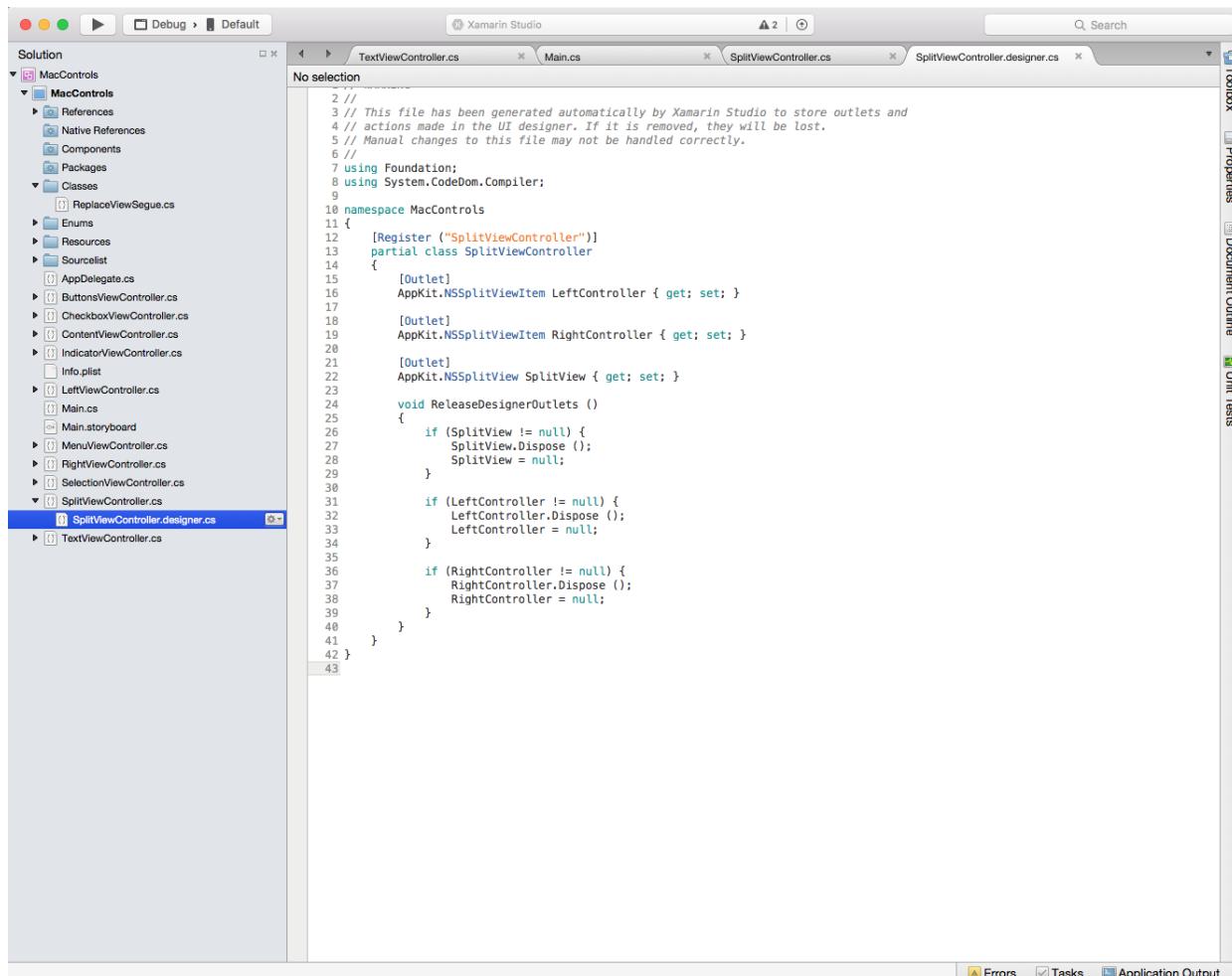


For more information on working with **Outlets** and **Actions**, please see the [Outlets and Actions](#) section of our [Introduction to Xcode and Interface Builder](#) documentation.

Synchronizing Changes with Xcode

When you switch back to Visual Studio for Mac from Xcode, any changes that you have made in Xcode will automatically be synchronized with your Xamarin.Mac project.

If you select the `SplitViewController.designer.cs` in the **Solution Explorer** you'll be able to see how your **Outlet** and **Action** have been wired up in our C# code:



Notice how the definition in the `SplitViewController.designer.cs` file:

```
[Outlet]
AppKit.NSSplitViewItem LeftController { get; set; }

[Outlet]
AppKit.NSSplitViewItem RightController { get; set; }

[Outlet]
AppKit.NSSplitView SplitView { get; set; }
```

Line up with the definition in the `MainWindow.h` file in Xcode:

```

@interface SplitViewController : NSSplitViewController {
    NSSplitViewItem *_LeftController;
    NSSplitViewItem *_RightController;
    NSSplitView *_SplitView;
}

@property (nonatomic, retain) IBOutlet NSSplitViewItem *LeftController;

@property (nonatomic, retain) IBOutlet NSSplitViewItem *RightController;

@property (nonatomic, retain) IBOutlet NSSplitView *SplitView;

```

As you can see, Visual Studio for Mac listens for changes to the `.h` file, and then automatically synchronizes those changes in the respective `.designer.cs` file to expose them to your application. You may also notice that `SplitViewController.designer.cs` is a partial class, so that Visual Studio for Mac doesn't have to modify `SplitViewController.cs` which would overwrite any changes that we have made to the class.

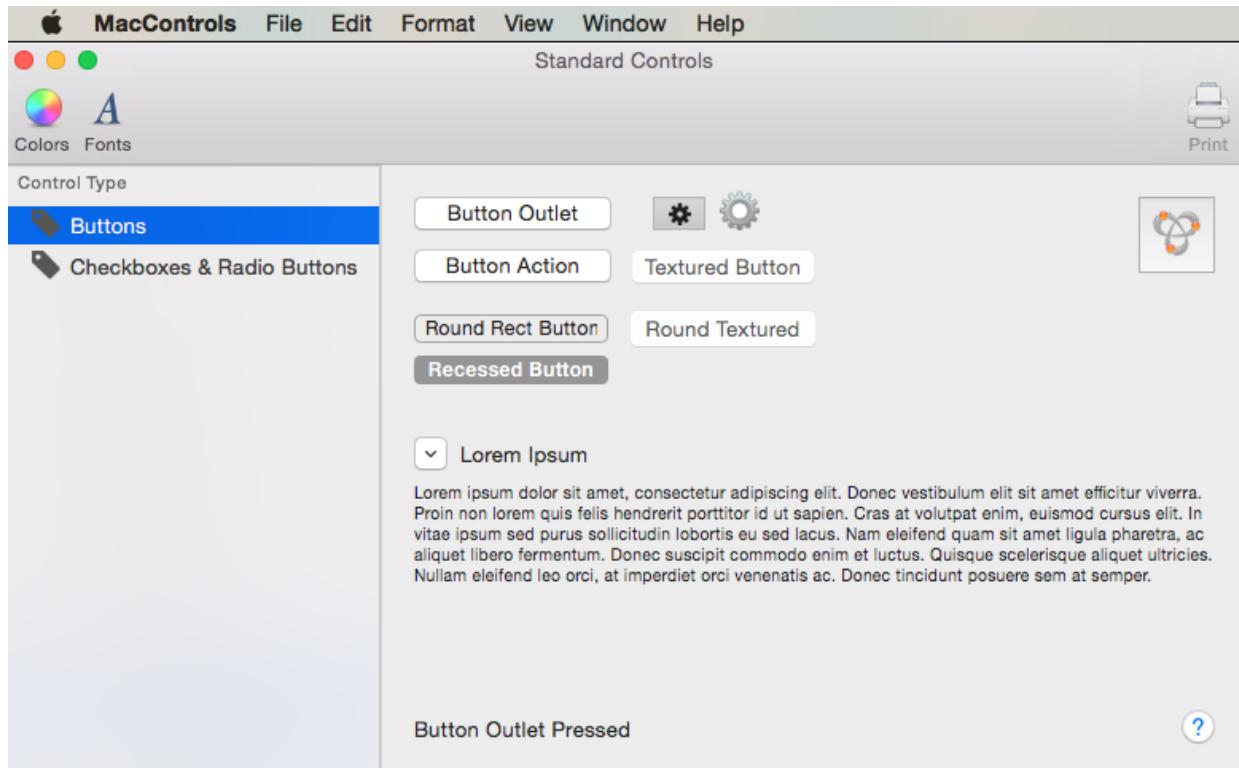
You normally will never need to open the `SplitViewController.designer.cs` yourself, it was presented here for educational purposes only.

IMPORTANT

In most situations, Visual Studio for Mac will automatically see any changes made in Xcode and sync them to your Xamarin.Mac project. In the off occurrence that synchronization doesn't automatically happen, switch back to Xcode and them back to Visual Studio for Mac again. This will normally kick off a synchronization cycle.

Working with Buttons

AppKit provides several types of button that can be used in your User Interface Design. For more information, please see the [Buttons](#) section of Apple's [OS X Human Interface Guidelines](#).



If a button has been exposed via an **Outlet**, the following code will respond to it being pressed:

```
ButtonOutlet.Activated += (sender, e) => {
    FeedbackLabel.StringValue = "Button Outlet Pressed";
};
```

For buttons that have been exposed via **Actions**, a `public partial` method will automatically be created for you with the name that you chose in Xcode. To respond to the **Action**, complete the partial method in the class that the **Action** was defined on. For example:

```
partial void ButtonAction (Foundation.NSObject sender) {
    // Do something in response to the Action
    FeedbackLabel.StringValue = "Button Action Pressed";
}
```

For buttons that have a state (like **On** and **Off**), the state can be checked or set with the `State` property against the `NSCellStateValue` enum. For example:

```
DisclosureButton.Activated += (sender, e) => {
    LorumIpsum.Hidden = (DisclosureButton.State == NSCellStateValue.On);
};
```

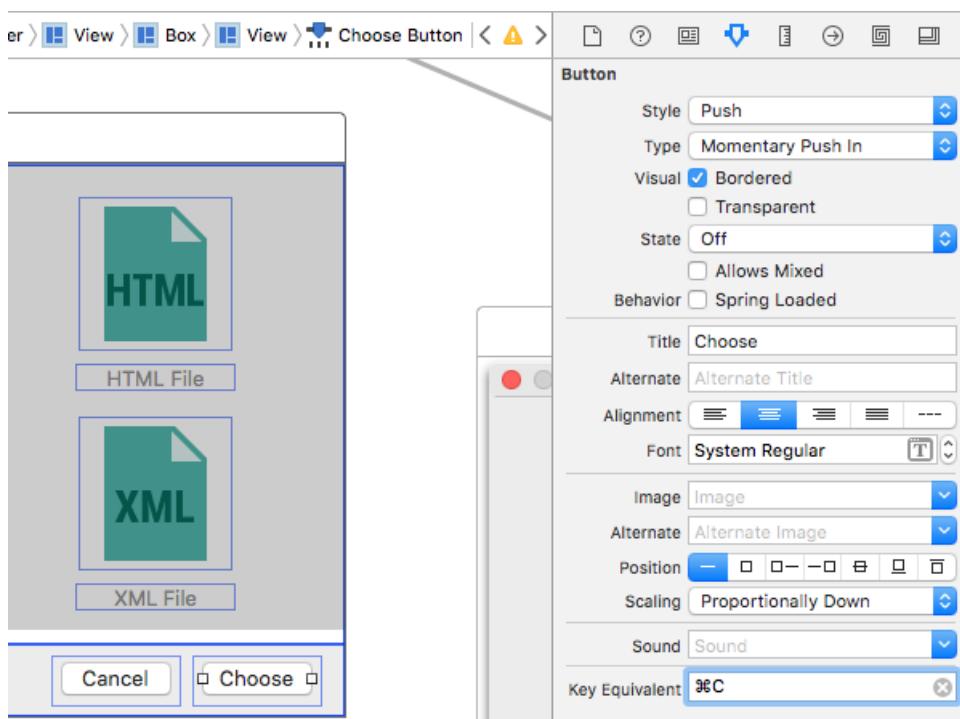
Where `NSCellStateValue` can be:

- **On** - The button is pushed or the control is selected (such as a check in a Check Box).
- **Off** - The button is not pushed or the control is not selected.
- **Mixed** - A mixture of On and Off states.

Mark a Button as Default and Set Key Equivalent

For any button that you have added to a user interface design, you can mark that button as the *Default* button that will be activated when the user presses the **Return/Enter** key on the keyboard. In macOS, this button will receive a blue background color by default.

To set a button as default, select it in Xcode's Interface Builder. Next, in the **Attribute Inspector**, select the **Key Equivalent** field and press the **Return/Enter** key:

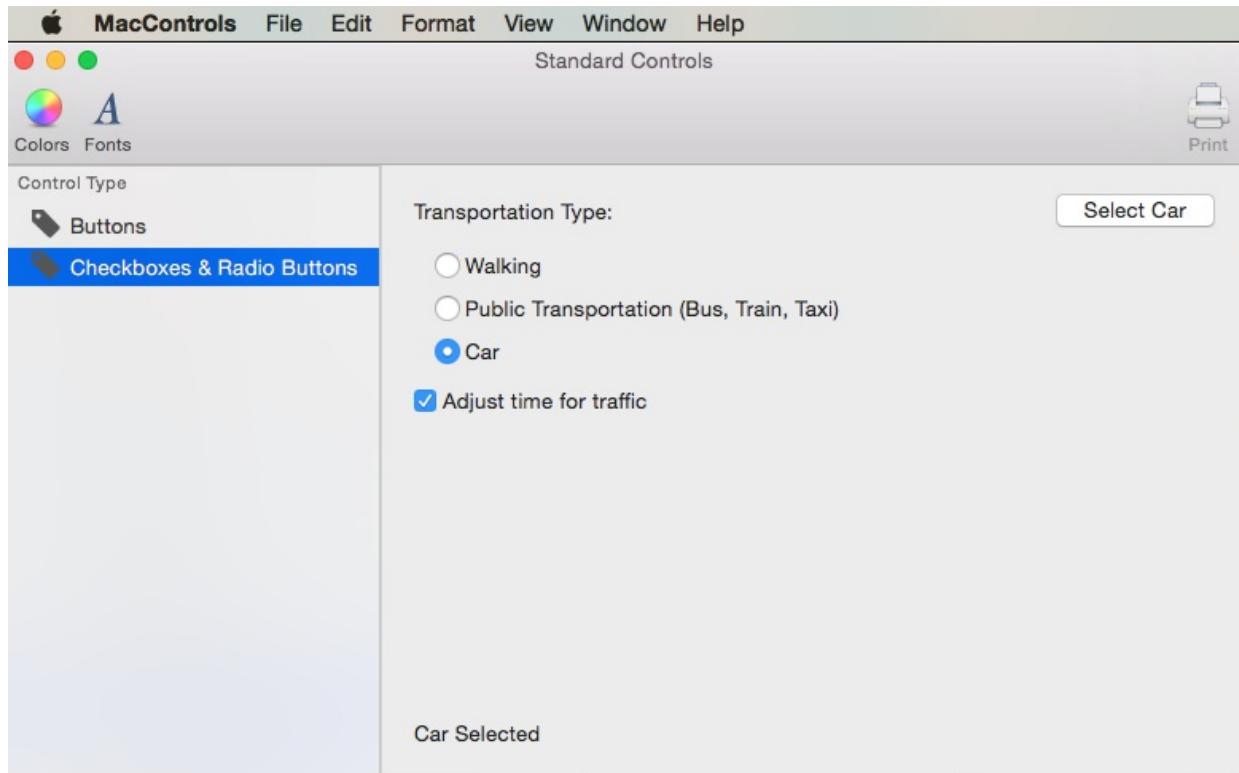


Equally, you can assign any key sequence that can be used to activate the button using the keyboard instead of the mouse. For example, by pressing the Command-C keys in the image above.

When the app is run and the Window with the Button is Key and Focused, if the user presses Command-C, the Action for the button will be activated (as-if the user had clicked on the button).

Working with Checkboxes and Radio Buttons

AppKit provides several types of Checkboxes and Radio Button Groups that can be used in your User Interface Design. For more information, please see the [Buttons](#) section of Apple's [OS X Human Interface Guidelines](#).



Checkboxes and Radio Buttons (exposed via **Outlets**) have a state (like **On** and **Off**), the state can be checked or set with the `State` property against the `NSCellStateValue` enum. For example:

```
AdjustTime.Activated += (sender, e) => {
    FeedbackLabel.StringValue = string.Format("Adjust Time: {0}", AdjustTime.State == NSCellStateValue.On);
};
```

Where `NSCellStateValue` can be:

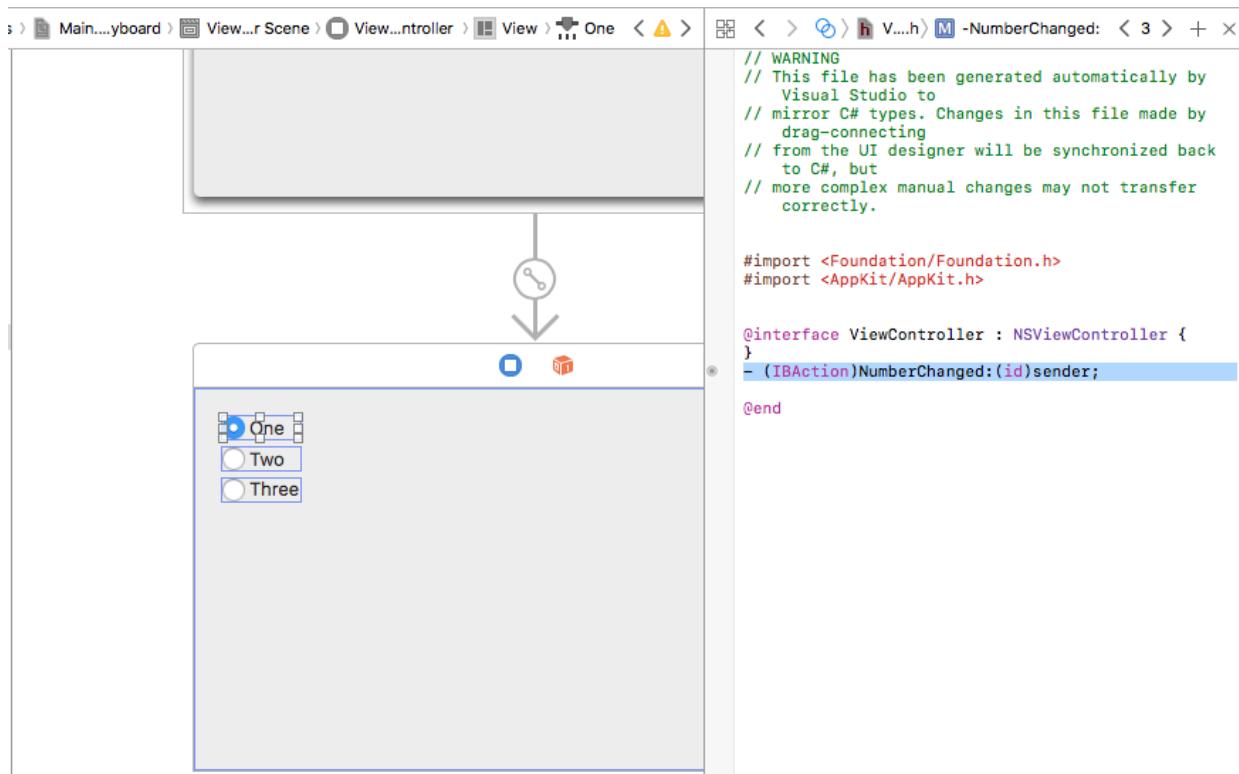
- **On** - The button is pushed or the control is selected (such as a check in a Check Box).
- **Off** - The button is not pushed or the control is not selected.
- **Mixed** - A mixture of On and Off states.

To select a button in a Radio Button Group, expose the Radio Button to select as an **Outlet** and set its `State` property. For Example:

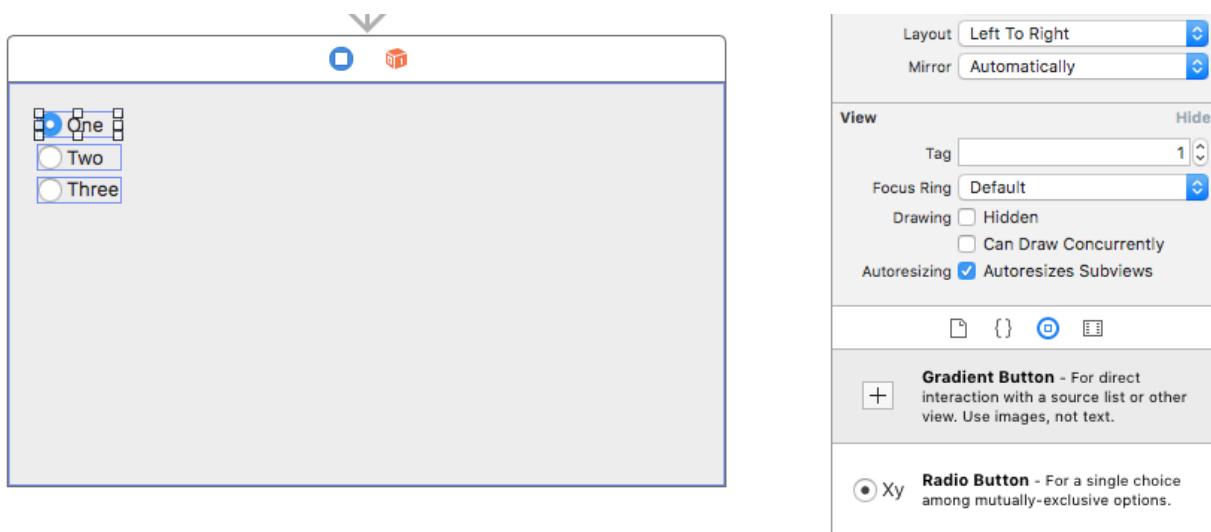
```
partial void SelectCar (Foundation.NSObject sender) {
    TransportationCar.State = NSCellStateValue.On;
    FeedbackLabel.StringValue = "Car Selected";
}
```

To get a collection of radio buttons to act as a group and automatically handle the selected state, create a new

Action and attach every button in the group to it:



Next, assign a unique `Tag` to each radio button in the **Attribute Inspector**:



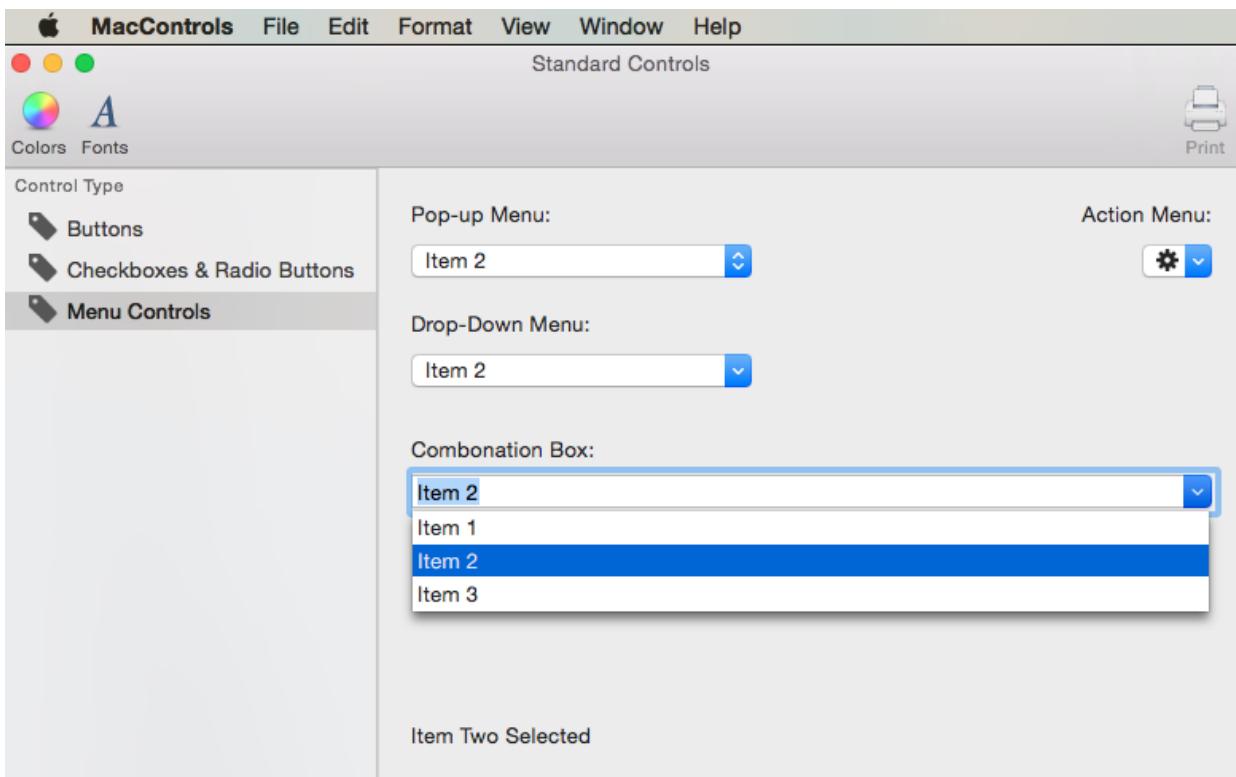
Save your changes and return to Visual Studio for Mac, add the code to handle the **Action** that all of the radio buttons are attached to:

```
partial void NumberChanged(Foundation.NSObject sender)
{
    var check = sender as NSButton;
    Console.WriteLine("Changed to {0}", check.Tag);
}
```

You can use the `Tag` property to see which radio button was selected.

Working with Menu Controls

AppKit provides several types of Menu Controls that can be used in your User Interface Design. For more information, please see the [Menu Controls](#) section of Apple's [OS X Human Interface Guidelines](#).



Providing Menu Control Data

The Menu Controls available to macOS can be set to populate the dropdown list either from an internal list (that can be pre-defined in Interface Builder or populated via code) or by providing your own custom, external data source.

Working with Internal Data

In addition to defining items in Interface Builder, Menu Controls (such as `NSComboBox`), provide a complete set of methods that allow you to Add, Edit or Delete the items from the internal list that they maintain:

- `Add` - Adds a new item to the end of the list.
- `GetItem` - Returns the item at the given index.
- `Insert` - Inserts a new item in the list at the given location.
- `IndexOf` - Returns the index of the given item.
- `Remove` - Removes the given item from the list.
- `RemoveAll` - Removes all items from the list.
- `RemoveAt` - Removes the item at the given index.
- `Count` - Returns the number of items in the list.

IMPORTANT

If you are using an Extern Data Source (`UsesDataSource = true`), calling any of the above methods will throw an exception.

Working with an External Data Source

Instead of using the built-in Internal Data to provide the rows for your Menu Control, you can optionally use an External Data Source and provide your own backing store for the items (such as a SQLite database).

To work with an External Data Source, you'll create an instance of the Menu Control's Data Source (`NSComboBoxDataSource` for example) and override several methods to provide the necessary data:

- `ItemCount` - Returns the number of items in the list.
- `ObjectValueForItem` - Returns the value of the item for a given index.
- `IndexOfItem` - Returns the index for the give item value.
- `CompletedString` - Returns the first matching item value for the partially typed item value. This method is only called if Autocomplete has been enabled (`Completes = true`).

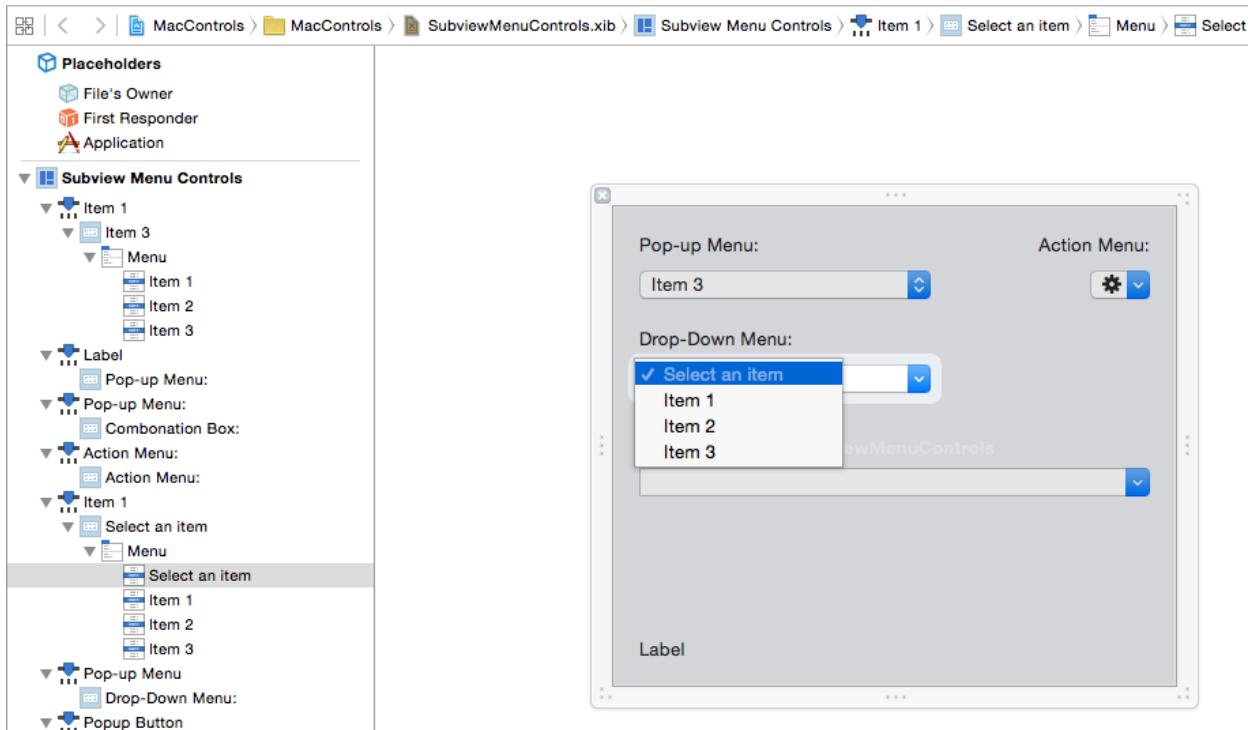
Please see the [Databases and ComboBoxes](#) section of the [Working with Databases](#) document for more details.

Adjusting the List's Appearance

The following methods are available to adjust the Menu Control's appearance:

- `HasVerticalScroller` - If `true`, the control will display a vertical scrollbar.
- `VisibleItems` - Adjust the number of items displayed when the control is opened. The default value is five (5).
- `InterCellSpacing` - Adjust the amount of space around a given item by providing a `NSSize` where the `Width` specifies the left and right margins and the `Height` specifies the space before and after an item.
- `ItemHeight` - Specifies the height of each item in the list.

For Drop-Down types of `NSPopUpButtons`, the first Menu Item provides the title for the control. For Example:



To change the title, expose this item as an **Outlet** and use code like the following:

```
DropDownSelected.Title = "Item 1";
```

Manipulating the Selected Items

The following methods and properties allow you to manipulate the selected items in the Menu Control's list:

- `SelectItem` - Selects the item at the given index.
- `Select` - Select the given item value.
- `DeselectItem` - Deselects the item at the given index.
- `SelectedIndex` - Returns the index of the currently selected item.
- `SelectedValue` - Returns the value of the currently selected item.

Use the `ScrollItemAtIndexToTop` to present the item at the given index at the top of the list and the `ScrollItemAtIndexToVisible` to scroll to list until the item at the given index is visible.

Responding to Events

Menu Controls provide the following events to respond to user interaction:

- `SelectionChanged` - Is called when the user has selected a value from the list.
- `SelectionIsChanging` - Is called before the new user selected item becomes the active selection.
- `WillPopup` - Is called before the dropdown list of items is displayed.
- `WillDismiss` - Is called before the dropdown list of items is closed.

For `NSComboBox` controls, they include all of the same events as the `NSTextField`, such as the `Changed` event that is called whenever the user edits the value of the text in the Combo Box.

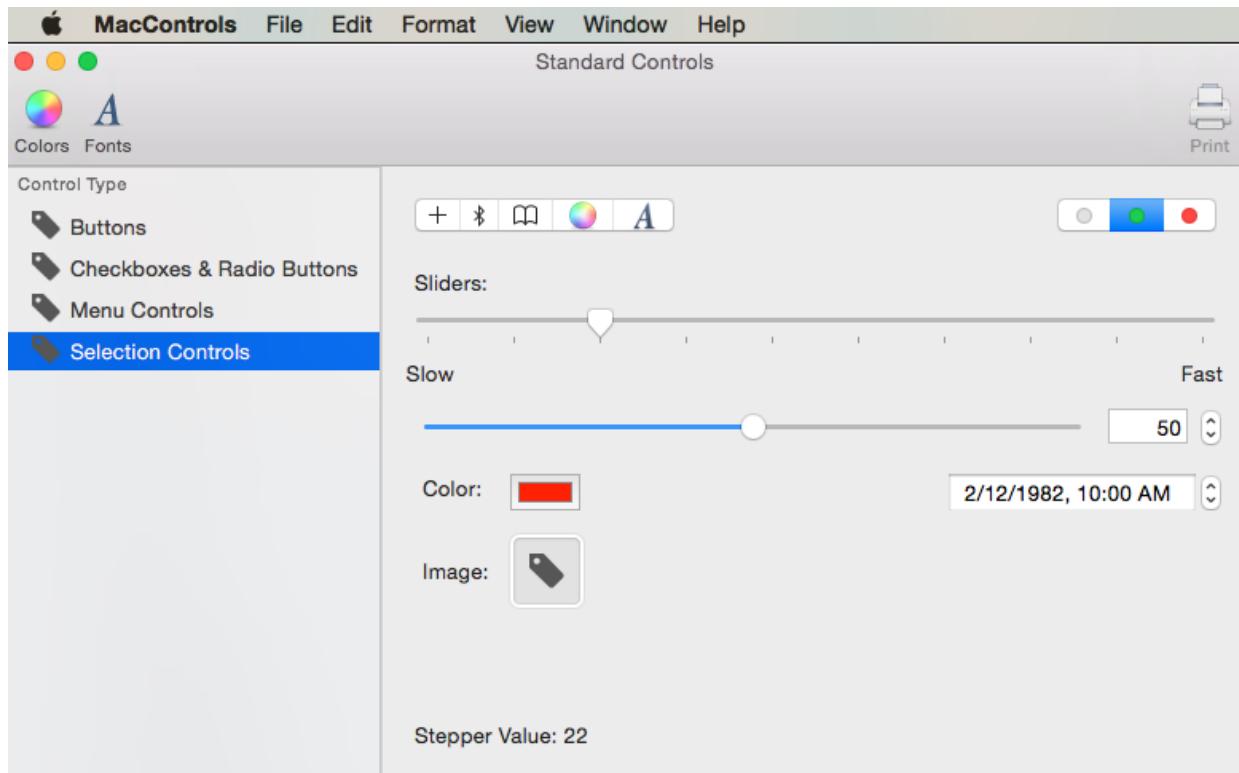
Optionally, you can respond to the Internal Data Menu Items defined in Interface Builder being selected by attaching the item to an **Action** and use code like the following to respond to **Action** being triggered by the user:

```
partial void ItemOne (Foundation.NSObject sender) {
    DropDownSelected.Title = "Item 1";
    FeedbackLabel.StringValue = "Item One Selected";
}
```

For more information on working with Menus and Menu Controls, please see our [Menus](#) and [Pop-up Button and Pull-Down Lists](#) documentation.

Working with Selection Controls

AppKit provides several types of Selection Controls that can be used in your User Interface Design. For more information, please see the [Selection Controls](#) section of Apple's [OS X Human Interface Guidelines](#).



There are two ways to track when a Selection Control has user interaction, by exposing it as an **Action**. For

example:

```
partial void SegmentButtonPressed (Foundation.NSObject sender) {
    FeedbackLabel.StringValue = string.Format("Button {0} Pressed",SegmentButtons.SelectedSegment);
}
```

Or by attaching a **Delegate** to the `Activated` event. For example:

```
TickedSlider.Activated += (sender, e) => {
    FeedbackLabel.StringValue = string.Format("Stepper Value: {0:###}",TickedSlider.IntValue);
};
```

To set or read the value of a Selection Control, use the `IntValue` property. For example:

```
FeedbackLabel.StringValue = string.Format("Stepper Value: {0:###}",TickedSlider.IntValue);
```

The specialty controls (such as Color Well and Image Well) have specific properties for their value types. For Example:

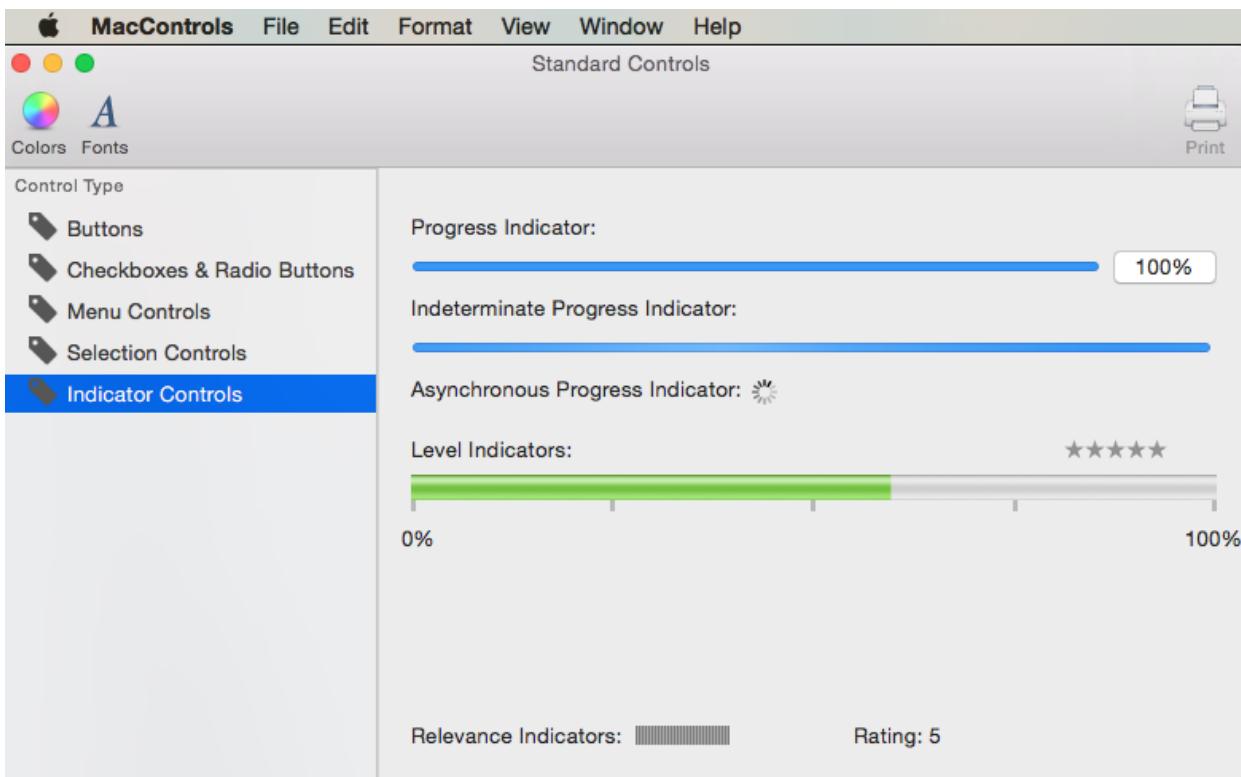
```
ColorWell.Color = NSColor.Red;
ImageWell.Image = NSImage.ImageNamed ("tag.png");
```

The `NSDatePicker` has the following properties for working directly with Date and Time:

- **DateValue** - The current date and time value as a `NSDate`.
- **Local** - The user's location as a `NSLocal`.
- **TimeInterval** - The time value as a `Double`.
- **TimeZone** - The user's time zone as a `NSTimeZone`.

Working with Indicator Controls

AppKit provides several types of Indicator Controls that can be used in your User Interface Design. For more information, please see the [Indicator Controls](#) section of Apple's [OS X Human Interface Guidelines](#).



There are two ways to track when a Indicator Control has user interaction, either by exposing it as an **Action** or an **Outlet** and attaching a **Delegate** to the `Activated` event. For example:

```
LevelIndicator.Activated += (sender, e) => {
    FeedbackLabel.StringValue = string.Format("Level: {0:###}",LevelIndicator.DoubleValue);
};
```

To read or set the value of the Indicator Control, use the `DoubleValue` property. For example:

```
FeedbackLabel.StringValue = string.Format("Rating: {0:###}",Rating.DoubleValue);
```

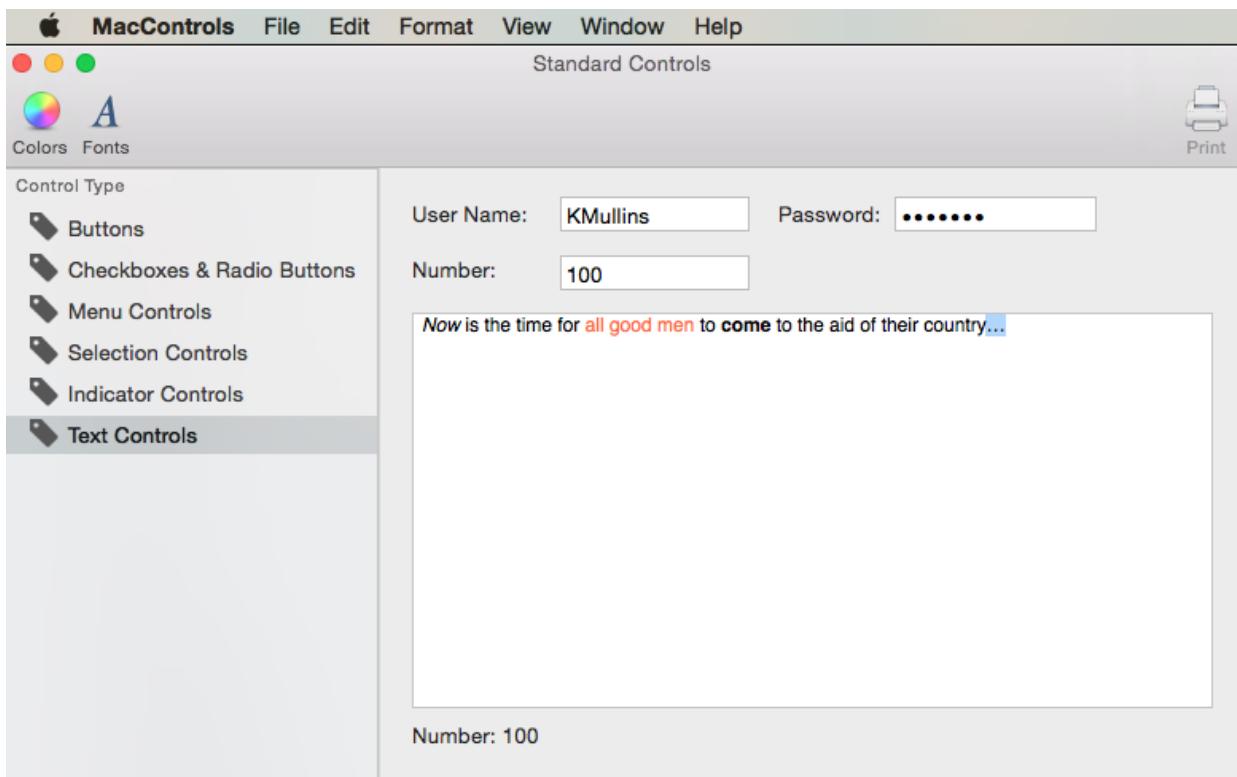
The Indeterminate and Asynchronous Progress Indicators should be animated when displayed. Use the `StartAnimation` method to start the animation when they are displayed. For example:

```
Indeterminate.StartAnimation (this);
AsyncProgress.StartAnimation (this);
```

Calling the `StopAnimation` method will stop the animation.

Working with Text Controls

AppKit provides several types of Text Controls that can be used in your User Interface Design. For more information, please see the [Text Controls](#) section of Apple's [OS X Human Interface Guidelines](#).



For Text Fields (`NSTextField`), the following events can be used to track user interaction:

- **Changed** - Is fired any time the user changes the value of the field. For example, on every character typed.
- **EditingBegan** - Is fired when the user selects the field for editing.
- **EditingEnded** - When the user presses the Enter key in the field or leaves the field.

Use the `StringValue` property to read or set the field's value. For example:

```
FeedbackLabel.StringValue = string.Format("User ID: {0}", UserField.StringValue);
```

For fields that display or edit numerical values, you can use the `IntValue` property. For example:

```
FeedbackLabel.StringValue = string.Format("Number: {0}", NumberField.IntValue);
```

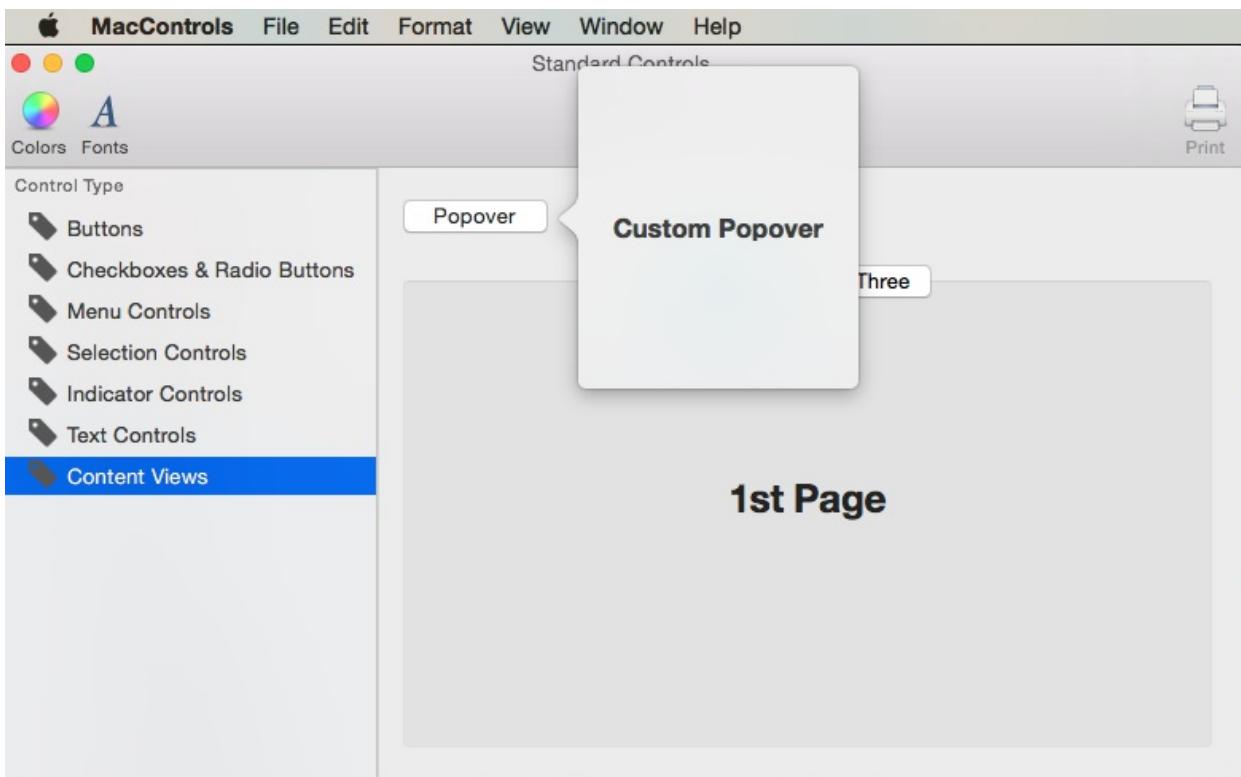
An `NSTextView` provides a full featured text edit and display area with built-in formatting. Like a `NSTextField`, use the `StringValue` property to read or set the area's value.

For an example of a complex example of working with Text Views in a Xamarin.Mac app, please see the [SourceWriter Sample App](#). SourceWriter is a simple source code editor that provides support for code completion and simple syntax highlighting.

The SourceWriter code has been fully commented and, where available, links have been provided from key technologies or methods to relevant information in the Xamarin.Mac Guides Documentation.

Working with Content Views

AppKit provides several types of Content Views that can be used in your User Interface Design. For more information, please see the [Content Views](#) section of Apple's [OS X Human Interface Guidelines](#).

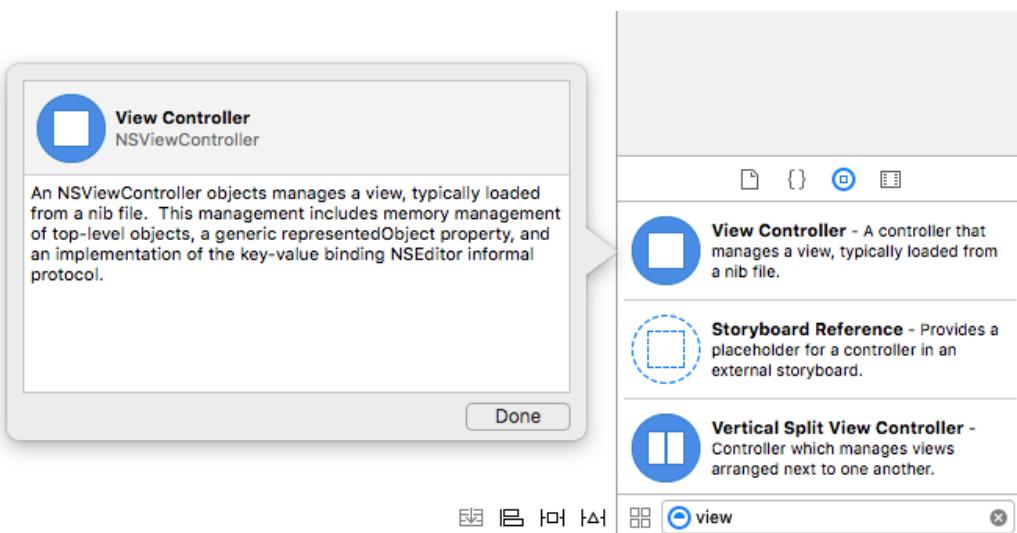


Popovers

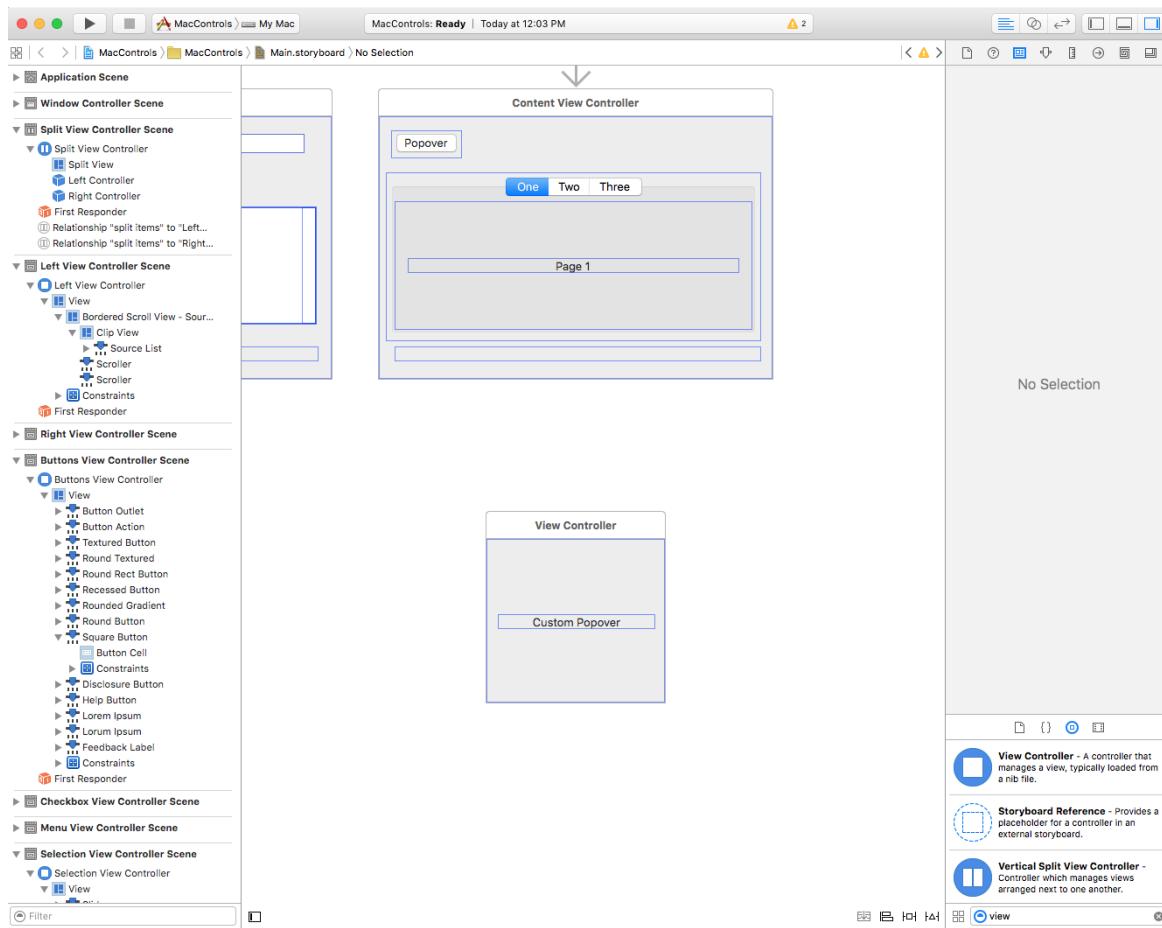
A popover is a transient UI element that provides functionality that is directly related to a specific control or an onscreen area. A popover floats above the window that contains the control or area that it's related to, and its border includes an arrow to indicate the point from which it emerged.

To create a popover, do the following:

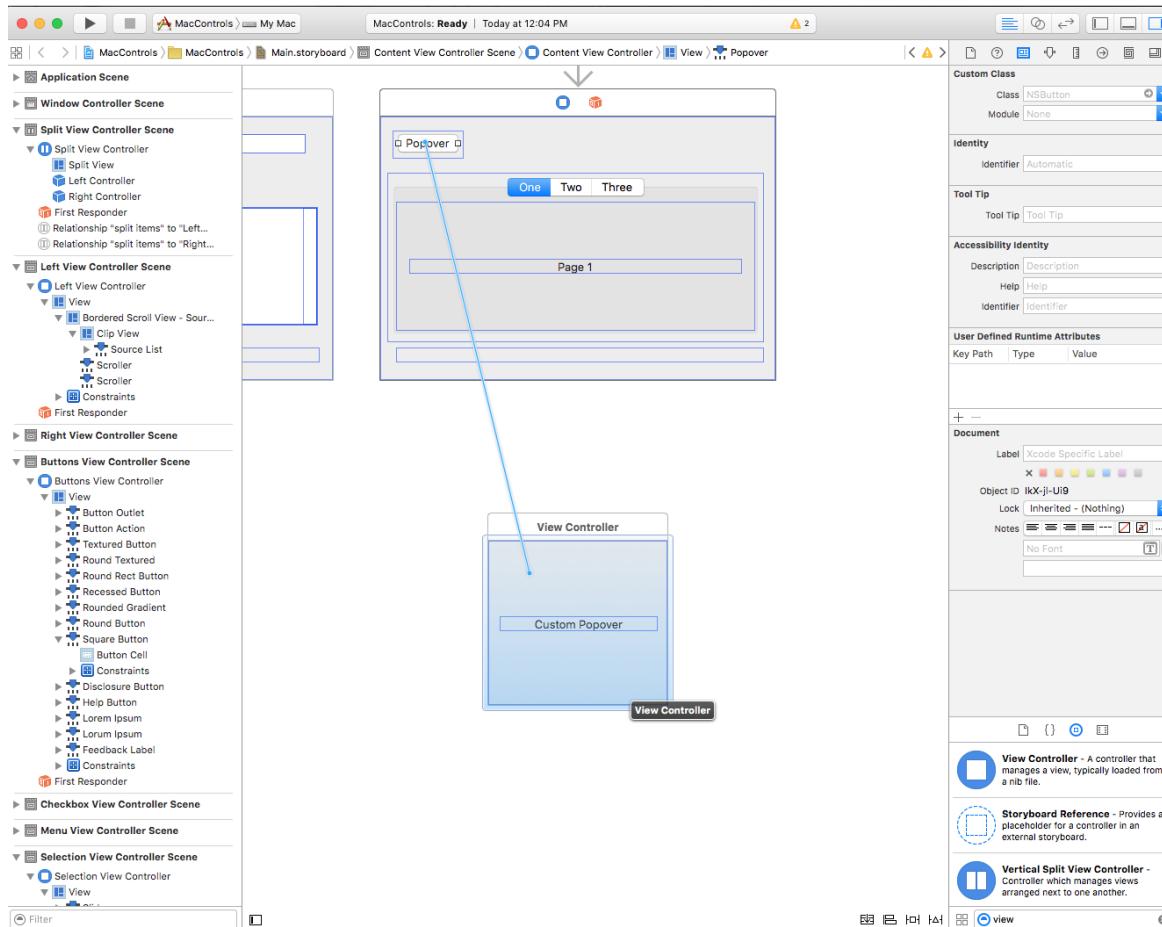
1. Open the `.storyboard` file of the window that you want to add a popover to by double-clicking it in the **Solution Explorer**
2. Drag a **View Controller** from the **Library Inspector** onto the **Interface Editor**:



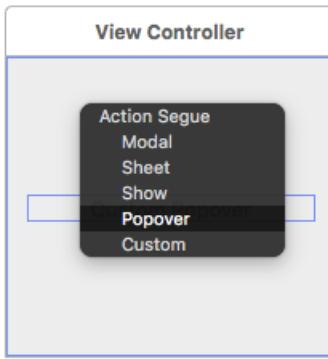
3. Define the size and the layout of the **Custom View**:



4. Control-click and drag from the source of the popup onto the View Controller:



5. Select Popover from the popup menu:

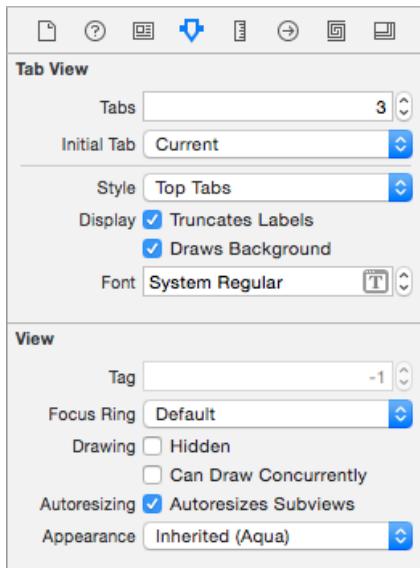


6. Save your changes and return to Visual Studio for Mac to sync with Xcode.

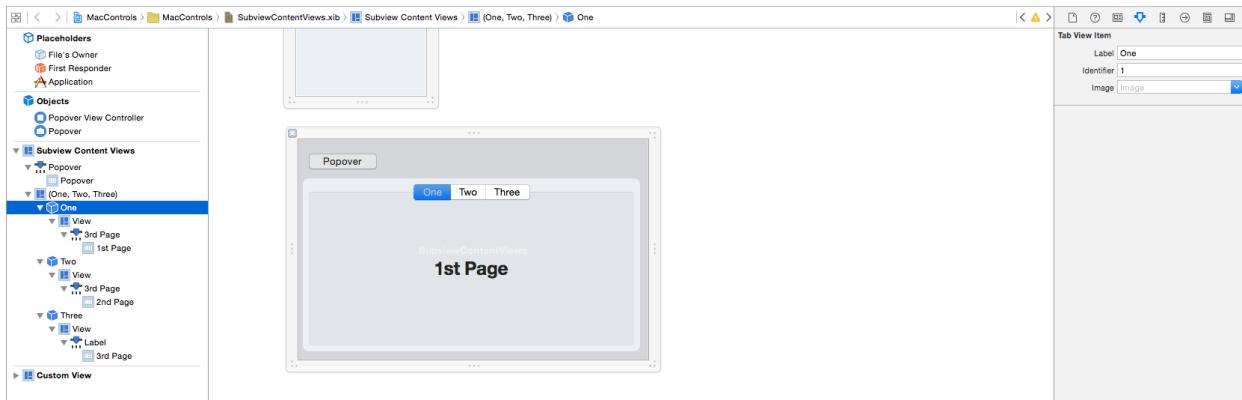
Tab Views

Tab Views consists of a Tab List (which looks similar to a Segmented Control) combined with a set of views that are called *Panes*. When the user selects a new Tab, the Pane that is attached to it will be displayed. Each Pane contains its own set of controls.

When working with a Tab View in Xcode's Interface Builder, use the **Attribute Inspector** to set the number of Tabs:



Select each Tab in the **Interface Hierarchy** to set its **Title** and add UI Elements to its **Pane**:



Data Binding AppKit Controls

By using Key-Value Coding and Data Binding techniques in your Xamarin.Mac application, you can greatly decrease the amount of code that you have to write and maintain to populate and work with UI elements. You also have the benefit of further decoupling your backing data (*Data Model*) from your front end User Interface (*Model-View-Controller*), leading to easier to maintain, more flexible application design.

Key-Value Coding (KVC) is a mechanism for accessing an object's properties indirectly, using keys (specially formatted strings) to identify properties instead of accessing them through instance variables or accessor methods (`get/set`). By implementing Key-Value Coding compliant accessors in your Xamarin.Mac application, you gain access to other macOS features such as Key-Value Observing (KVO), Data Binding, Core Data, Cocoa bindings, and scriptability.

For more information, please see the [Simple Data Binding](#) section of our [Data Binding and Key-Value Coding](#) documentation.

Summary

This article has taken a detailed look at working with the standard AppKit controls such as Buttons, Labels, Text Fields, Check Boxes and Segmented Controls in a Xamarin.Mac application. It covered adding them to a User Interface Design in Xcode's Interface Builder, exposing them to code through Outlets and Actions and working with AppKit Controls in C# Code.

Related Links

- [MacControls \(sample\)](#)
- [Hello, Mac](#)
- [Windows](#)
- [Data Binding and Key-Value Coding](#)
- [OS X Human Interface Guidelines](#)
- [About Controls and Views](#)

Toolbars in Xamarin.Mac

11/2/2020 • 10 minutes to read • [Edit Online](#)

This article describes working with toolbars in a Xamarin.Mac application. It covers creating and maintaining toolbars in Xcode and Interface Builder, exposing them to code, and working with them programmatically.

Xamarin.Mac developers working with Visual Studio for Mac have access to the same UI controls available to macOS developers working with Xcode, including the toolbar control. Because Xamarin.Mac integrates directly with Xcode, it's possible to use Xcode's Interface Builder to create and maintain toolbar items. These toolbar items can also be created in C#.

Toolbars in macOS are added to the top section of a window and provide easy access to commands related to its functionality. Toolbars can be hidden, shown, or customized by an application's users, and they can present toolbar items in various ways.

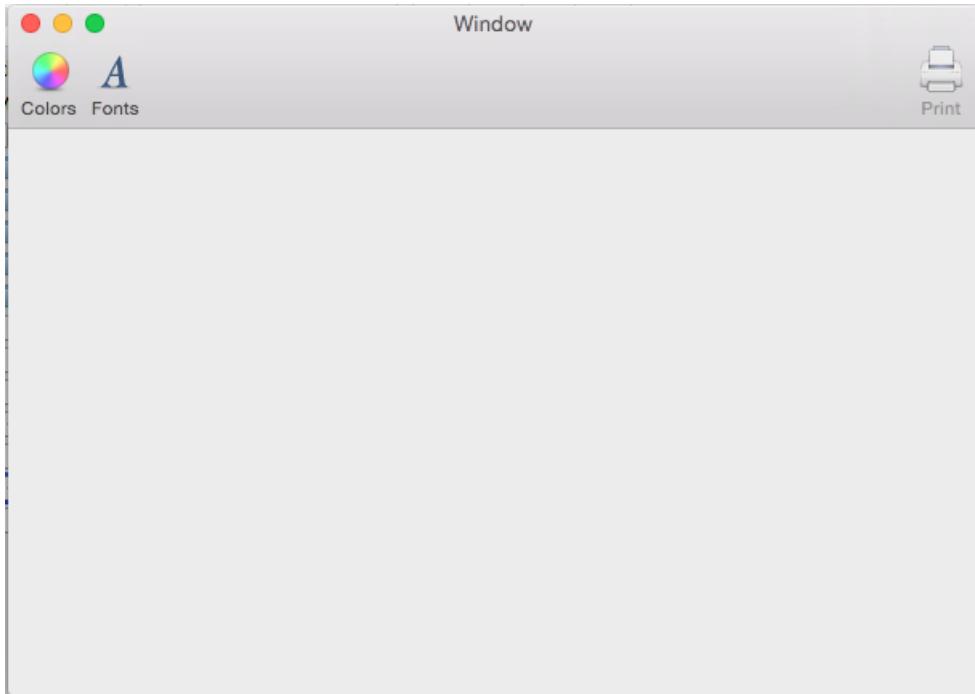
This article covers the basics of working with toolbars and toolbar items in a Xamarin.Mac application.

Before continuing, read through the [Hello, Mac](#) article — specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections — as it covers key concepts and techniques that will be used throughout this guide.

Also take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document. It explains the `Register` and `Export` attributes used to connect C# classes to Objective-C classes.

Introduction to toolbars

Any window in a macOS application can include a toolbar:



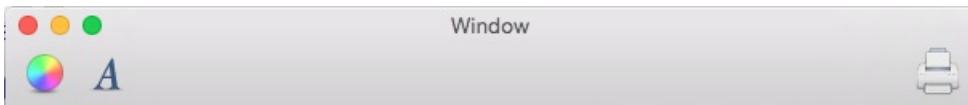
Toolbars provide an easy way for your application's users to quickly access important or commonly-used features. For example, a document-editing application might provide toolbar items for setting the text color, changing the font, or printing the current document.

Toolbars can display items in three ways:

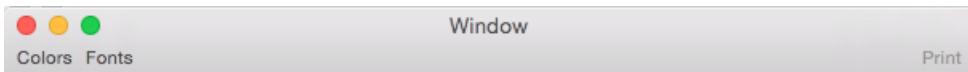
1. Icon and Text



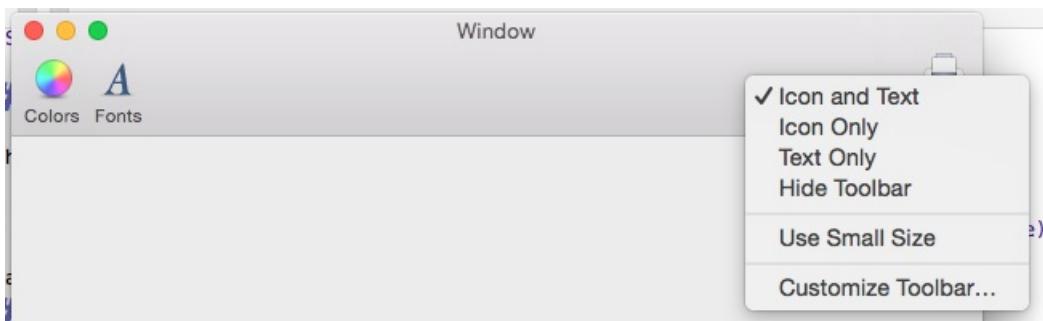
2. Icon Only



3. Text Only



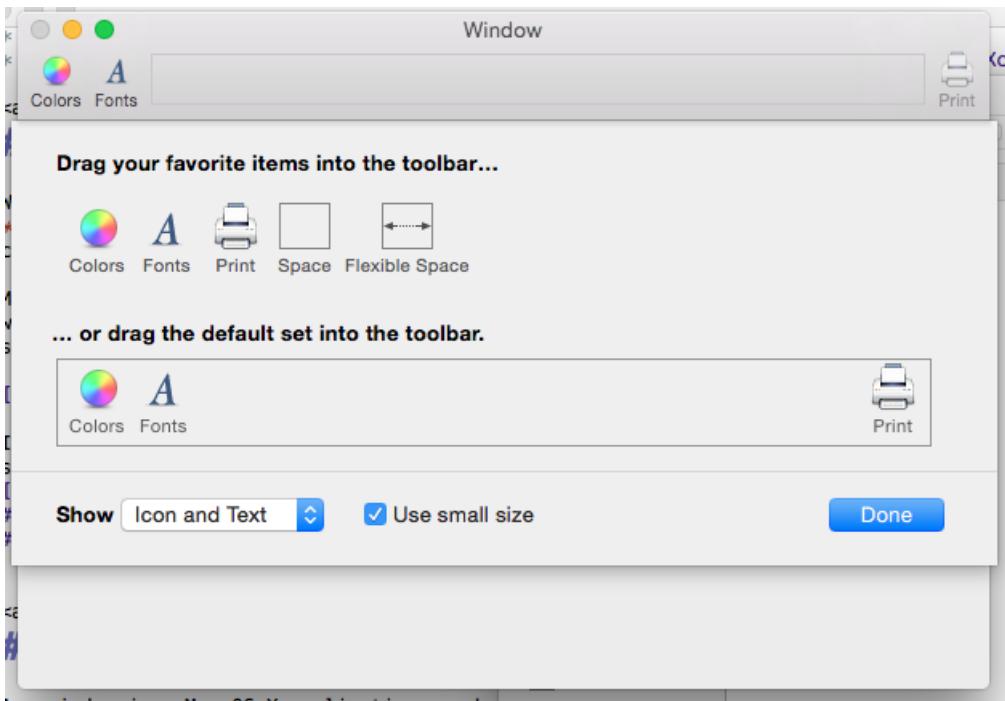
Switch between these modes by right-clicking the toolbar and selecting a display mode from the contextual menu:



Use the same menu to display the toolbar at a smaller size:



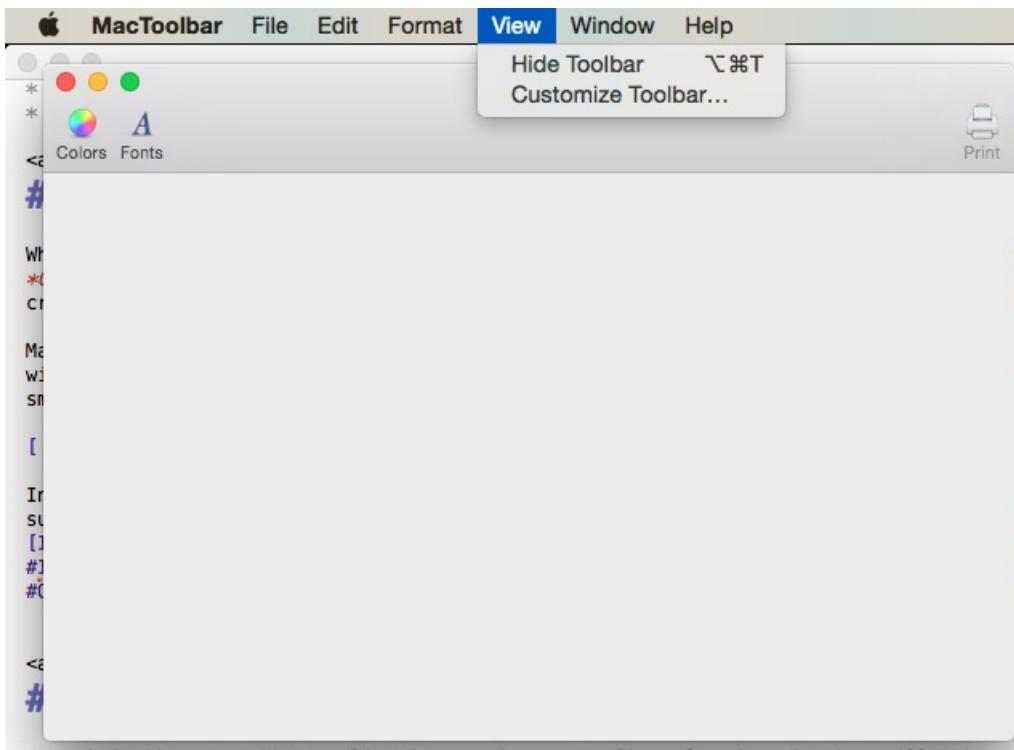
The menu also allows for customizing the toolbar:



When setting up a toolbar in Xcode's Interface Builder, a developer can provide extra toolbar items that are not

part of its default configuration. Users of the application can then customize the toolbar, adding and removing these pre-defined items as necessary. Of course, the toolbar can be reset to its default configuration.

The toolbar automatically connects to the **View** menu, which allows users to hide it, show it, and customize it:



See the [Built-In Menu Functionality](#) documentation for more details.

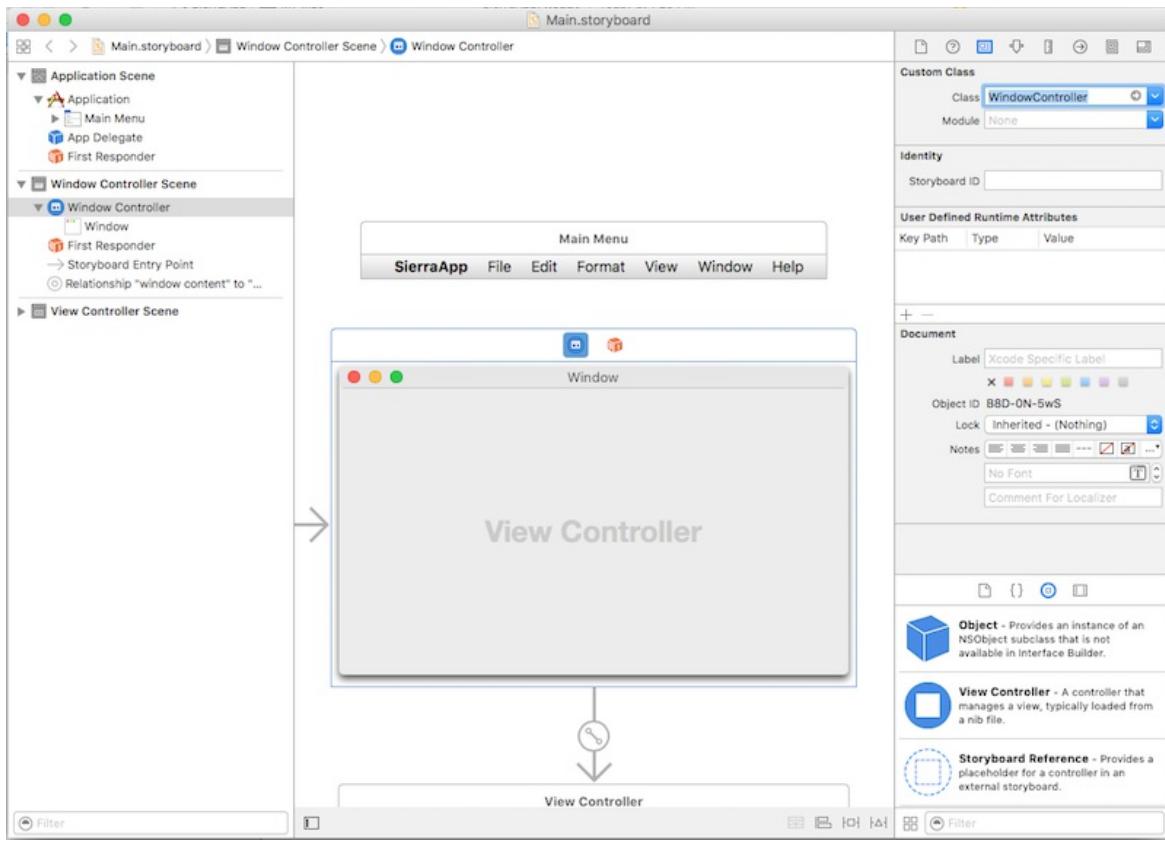
Additionally, if the toolbar is properly configured in Interface Builder, the application will automatically persist toolbar customizations across multiple launches of the application.

The next sections of this guide describe how to create and maintain toolbars with Xcode's Interface Builder and how to work with them in code.

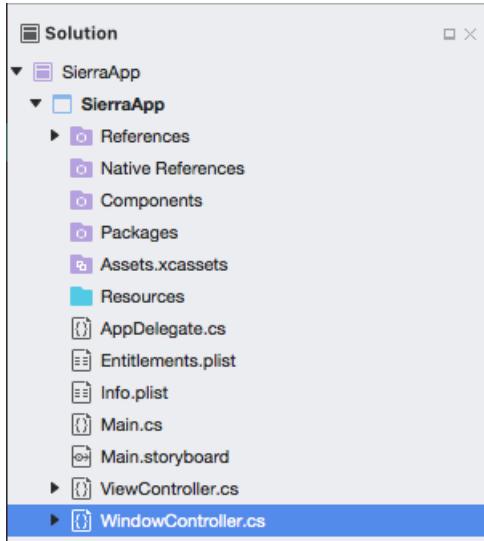
Setting a custom main window controller

To expose UI elements to C# code through outlets and actions, the Xamarin.Mac app must use a custom window controller:

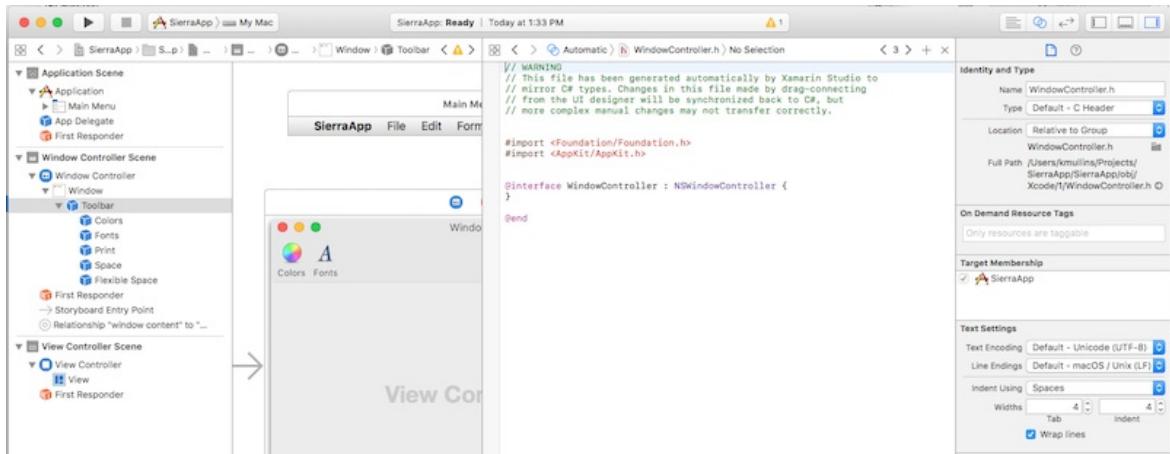
1. Open the app's storyboard in Xcode's Interface Builder.
2. Select the window controller on the design surface.
3. Switch to the **Identity Inspector** and enter "WindowController" as the **Class Name**:



4. Save your changes and return to Visual Studio for Mac to sync.
5. A **WindowController.cs** file will be added to your project in the **Solution Pad** in Visual Studio for Mac:

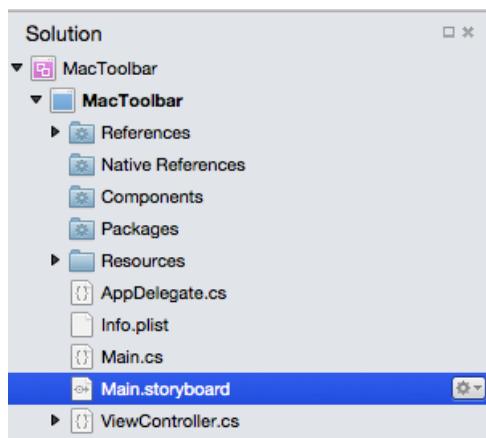


6. Reopen the storyboard in Xcode's Interface Builder.
7. The **WindowController.h** file will be available for use:

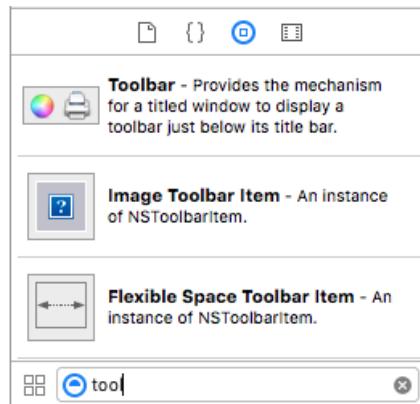


Creating and maintaining toolbars in Xcode

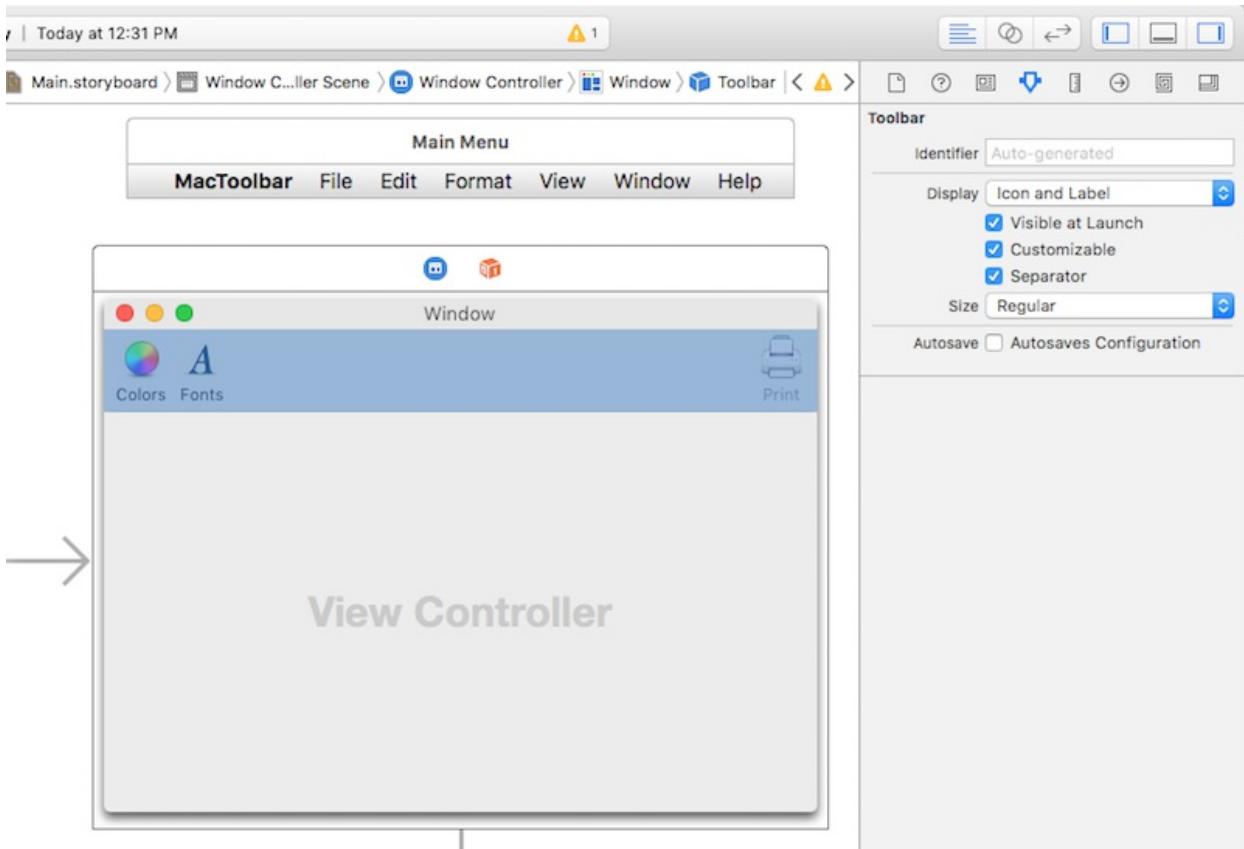
Toolbars are created and maintained with Xcode's Interface Builder. To add a toolbar to an application, edit the app's primary storyboard (in this case **Main.storyboard**) by double-clicking it in the Solution Pad:



In the **Library Inspector**, enter "tool" in the **Search Box** to make it easier to see all of the available toolbar items:



Drag a toolbar onto the window in the **Interface Editor**. With the toolbar selected, configure its behavior by setting properties in the **Attributes Inspector**:

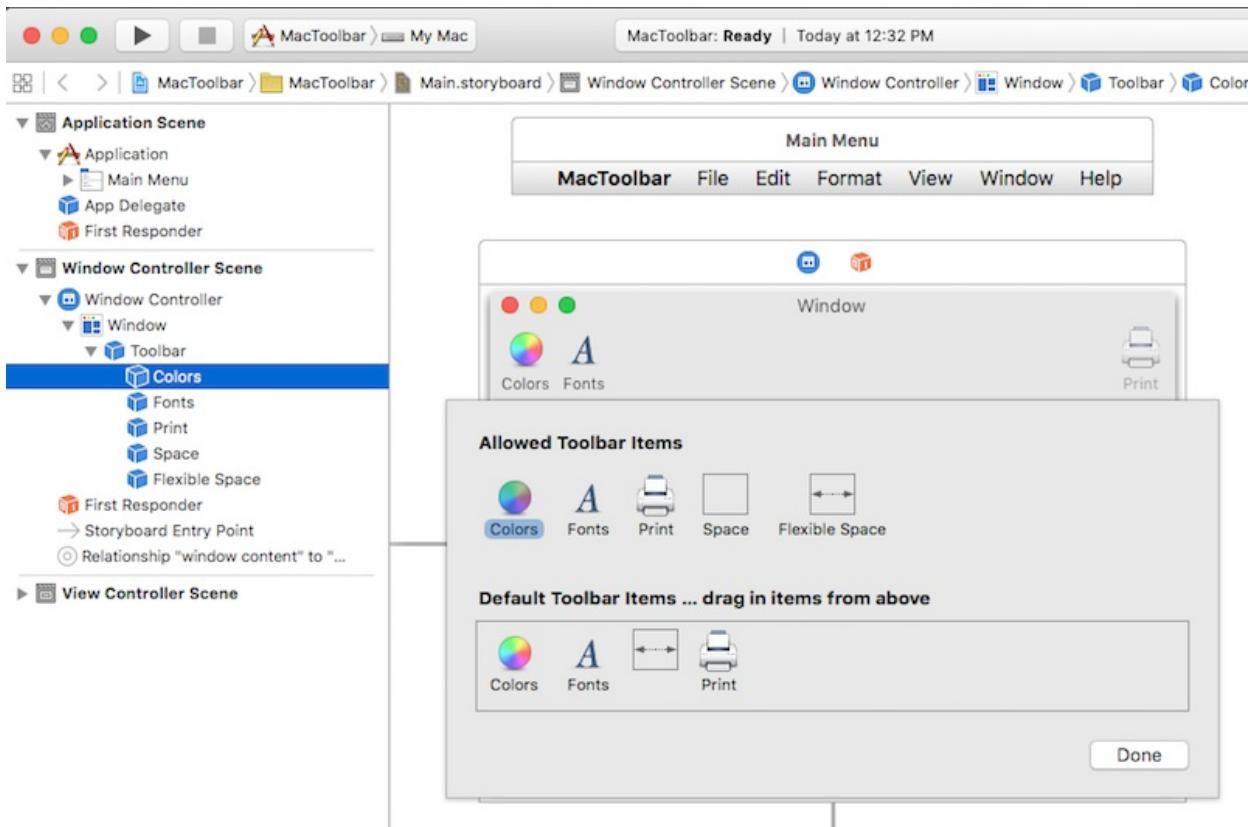


The following properties are available:

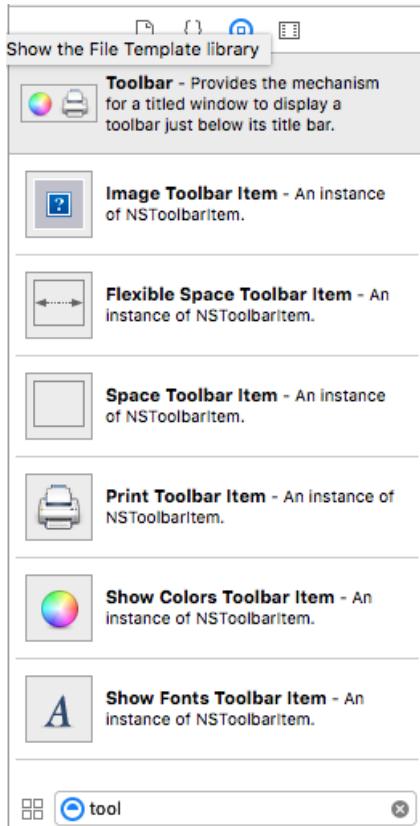
1. **Display** - Controls whether the toolbar displays icons, text, or both.
2. **Visible at Launch** - If selected, the toolbar is visible by default.
3. **Customizable** - If selected, users can edit and customize the toolbar.
4. **Separator** - If selected, a thin horizontal line separates the toolbar from the window's contents.
5. **Size** - Sets the size of the toolbar.
6. **Autosave** - If selected, the application will persist a user's toolbar configuration changes across application launches.

Select the **Autosave** option and leave all the other properties at their default settings.

After opening the toolbar in the **Interface Hierarchy**, bring up the customization dialog by selecting a toolbar item:



Use this dialog to set properties for items that are already part of the toolbar, to design the default toolbar for the application, and to provide extra toolbar items for a user to select when customizing the toolbar. To add items to the toolbar, drag them from the **Library Inspector**:



The following toolbar items can be added:

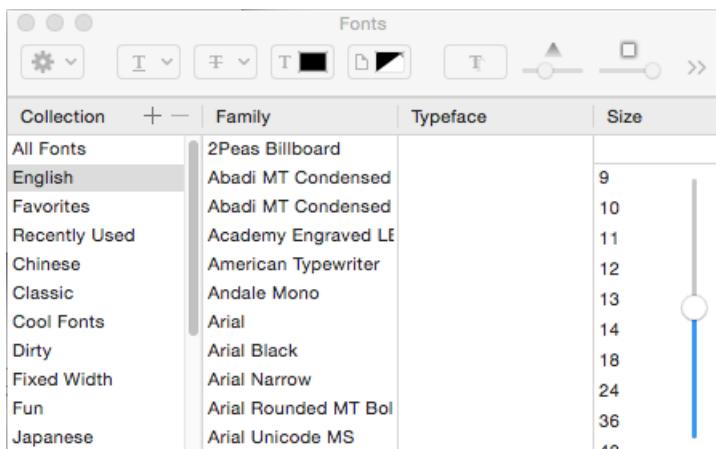
- **Image Toolbar Item** - A toolbar item with a custom image as an icon.
- **Flexible Space Toolbar Item** - Flexible space used to justify subsequent toolbar items. For example, one or more toolbar items followed by a flexible space toolbar item and another toolbar item would pin

the last item to the right side of the toolbar.

- **Space Toolbar Item** - Fixed space between items on the toolbar
- **Separator Toolbar Item** - A visible separator between two or more toolbar items, for grouping
- **Customize Toolbar Item** - Allows users to customize the toolbar
- **Print Toolbar Item** - Allows users to print the open document
- **Show Colors Toolbar Item** - Displays the standard system color picker:



- **Show Font Toolbar Item** - Displays the standard system font dialog:

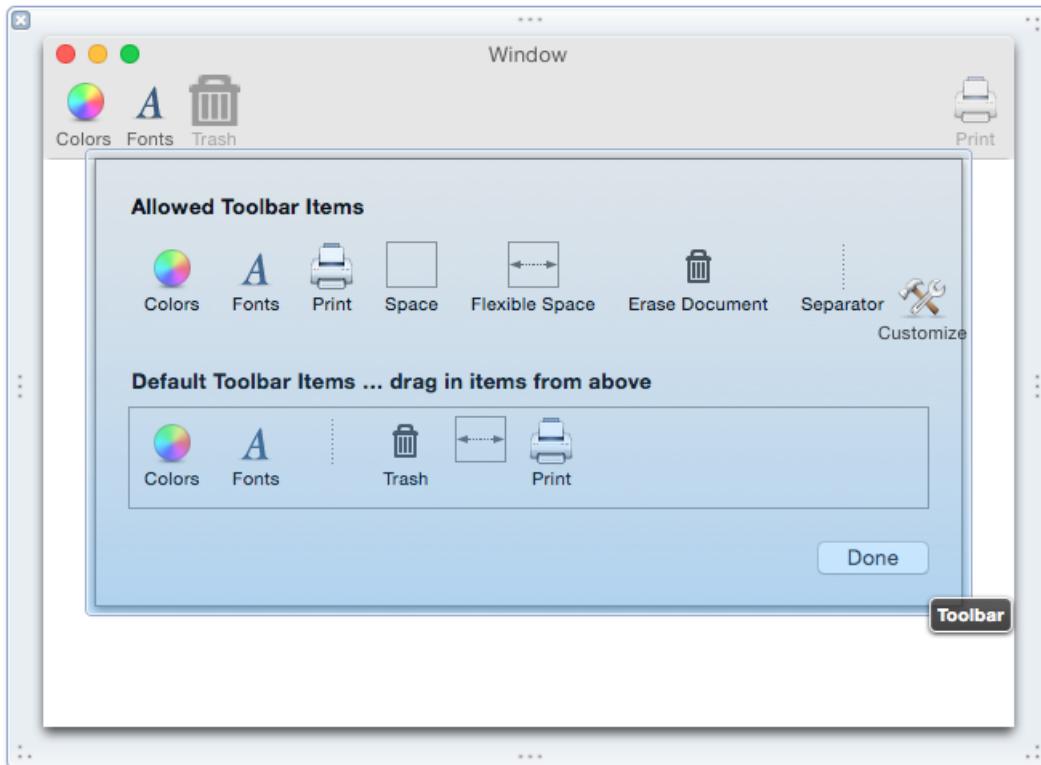


IMPORTANT

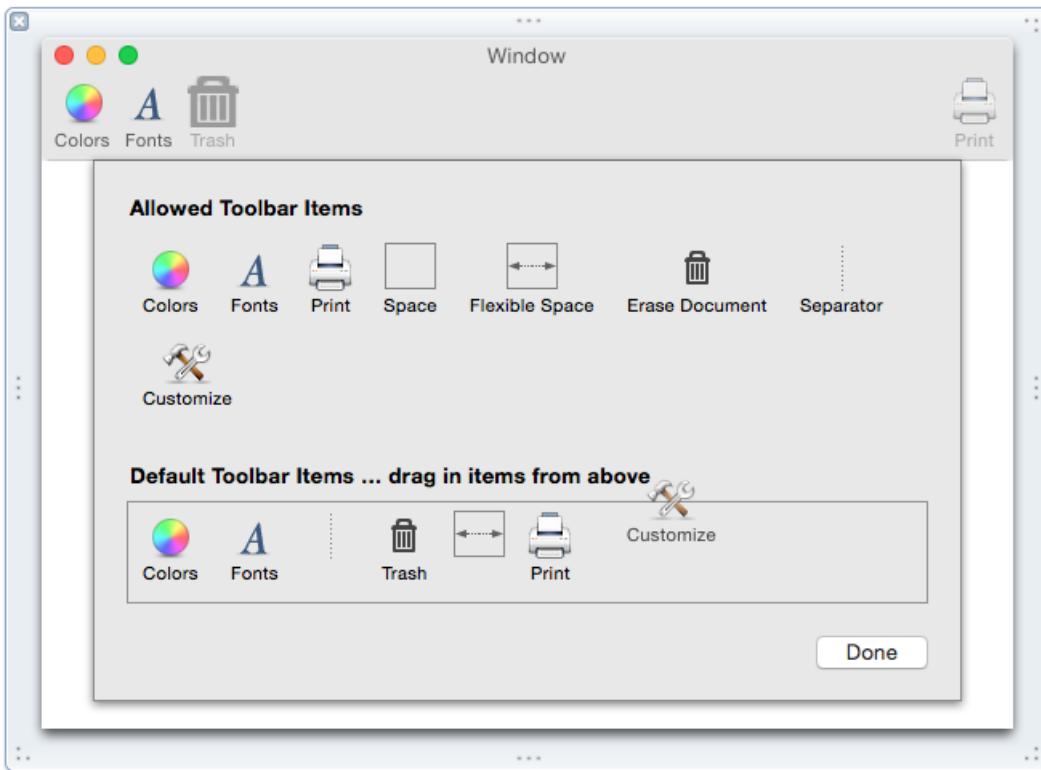
As will be seen later, many standard Cocoa UI controls such as search fields, segmented controls, and horizontal sliders can also be added to a toolbar.

Adding an item to a toolbar

To add an item to a toolbar, select the toolbar in the **Interface Hierarchy** and click one of its items, causing the customization dialog to appear. Next, drag a new item from the **Library Inspector** to the **Allowed Toolbar Items** area:

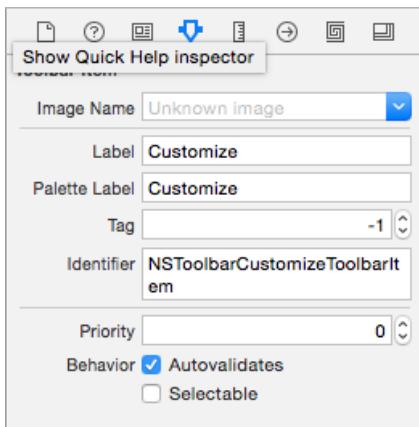


To make sure that a new item is part of the default toolbar, drag it to the **Default Toolbar Items** area:



To reorder default toolbar items, drag them left or right.

Next, use the **Attributes Inspector** to set default properties for the item:



The following properties are available:

- **Image Name** - Image to use as an icon for the item
- **Label** - Text to display for the item in the toolbar
- **Palette Label** - Text to display for the item in the **Allowed Toolbar Items** area
- **Tag** - An optional, unique identifier that helps identify the item in code.
- **Identifier** - Defines the toolbar item type. A custom value can be used to select a toolbar item in code.
- **Selectable** - If checked, the item will act like an on/off button.

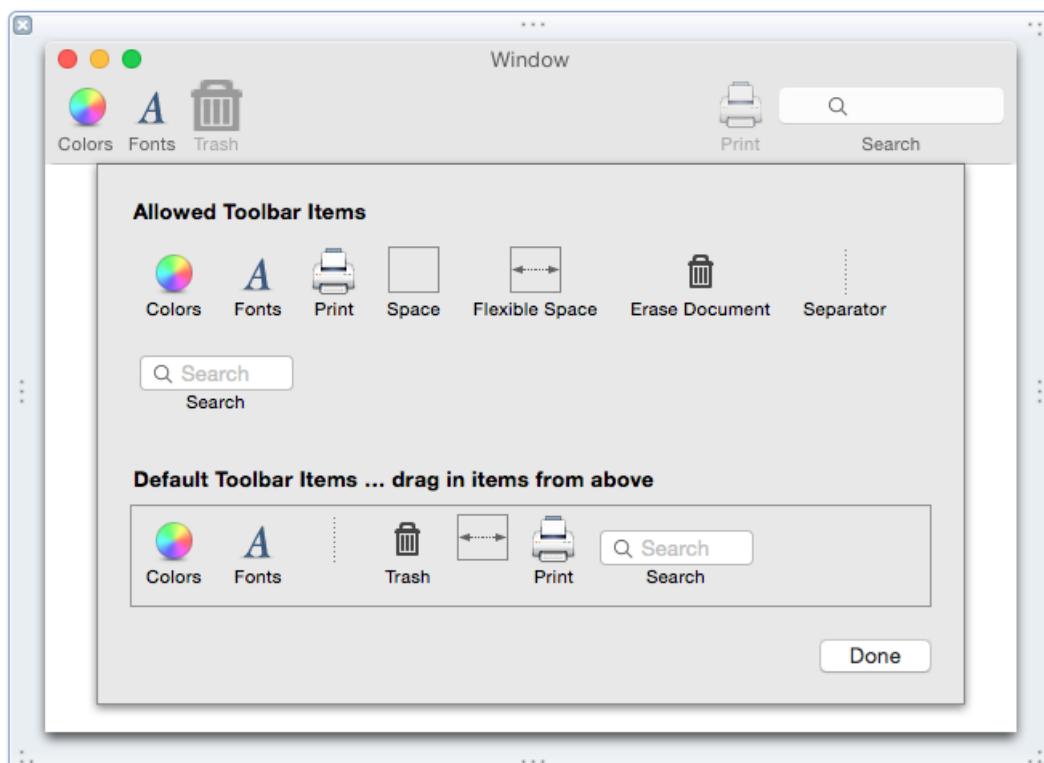
IMPORTANT

Add an item to the **Allowed Toolbar Items** area but not the default toolbar to provide customization options for users.

Adding other UI controls to a toolbar

Several Cocoa UI elements such as search fields and segmented controls can also be added to a toolbar.

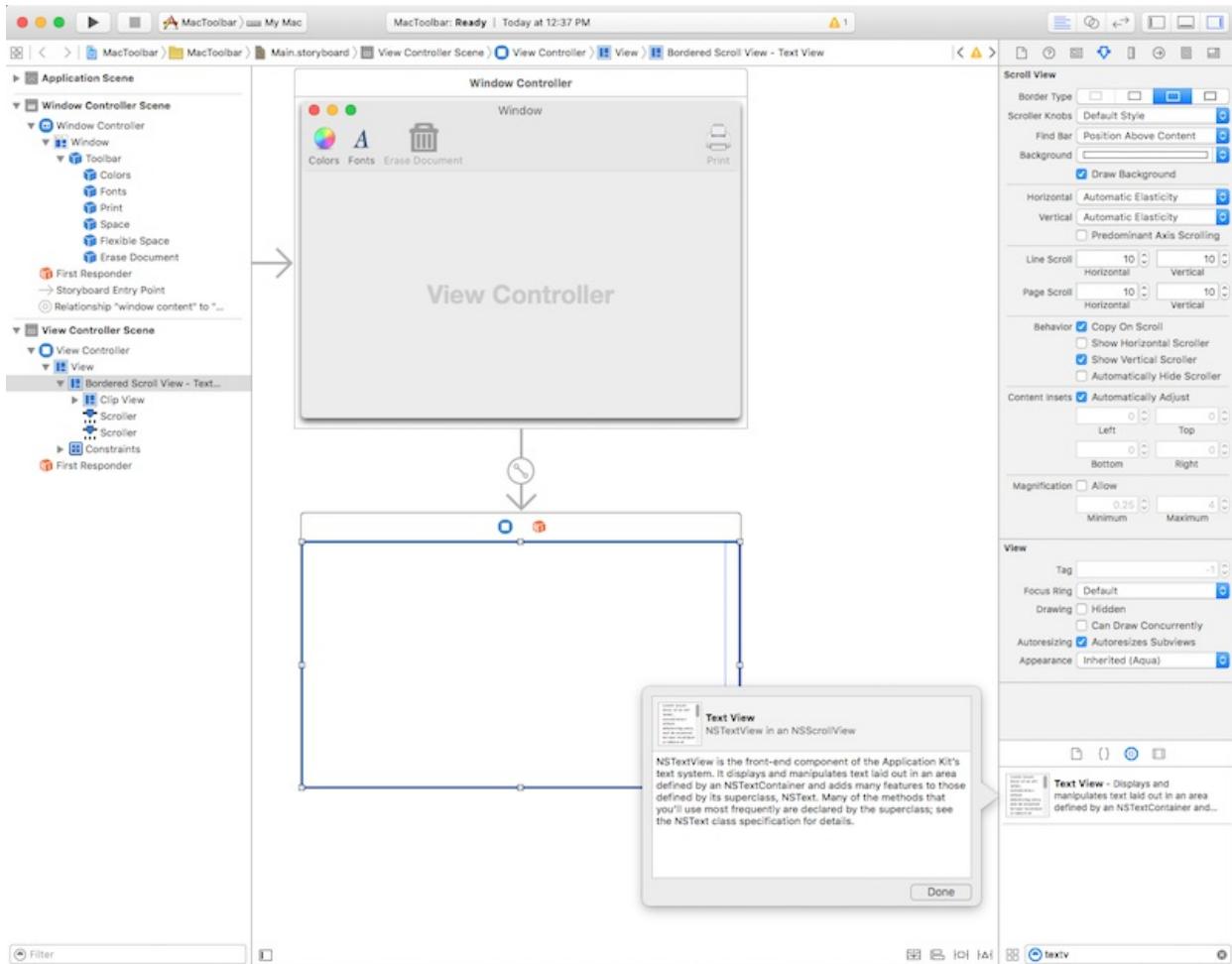
To try this, open the toolbar in the **Interface Hierarchy** and select a toolbar item to open the customization dialog. Drag a **Search Field** from the **Library Inspector** to the **Allowed Toolbar Items** area:



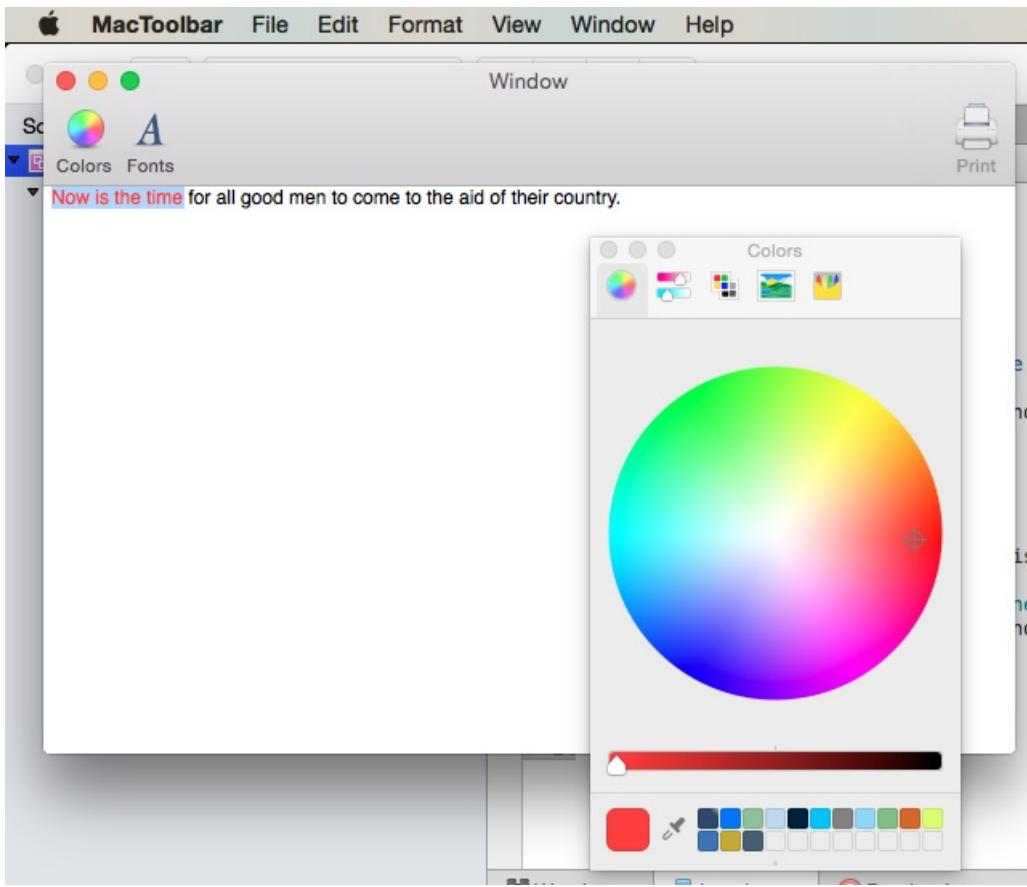
From here, use Interface Builder to configure the search field and expose it to code through an action or outlet.

Built-in toolbar item support

Several Cocoa UI elements interact with standard toolbar items by default. For example, drag a **Text View** onto the application's window and position it to fill the content area:



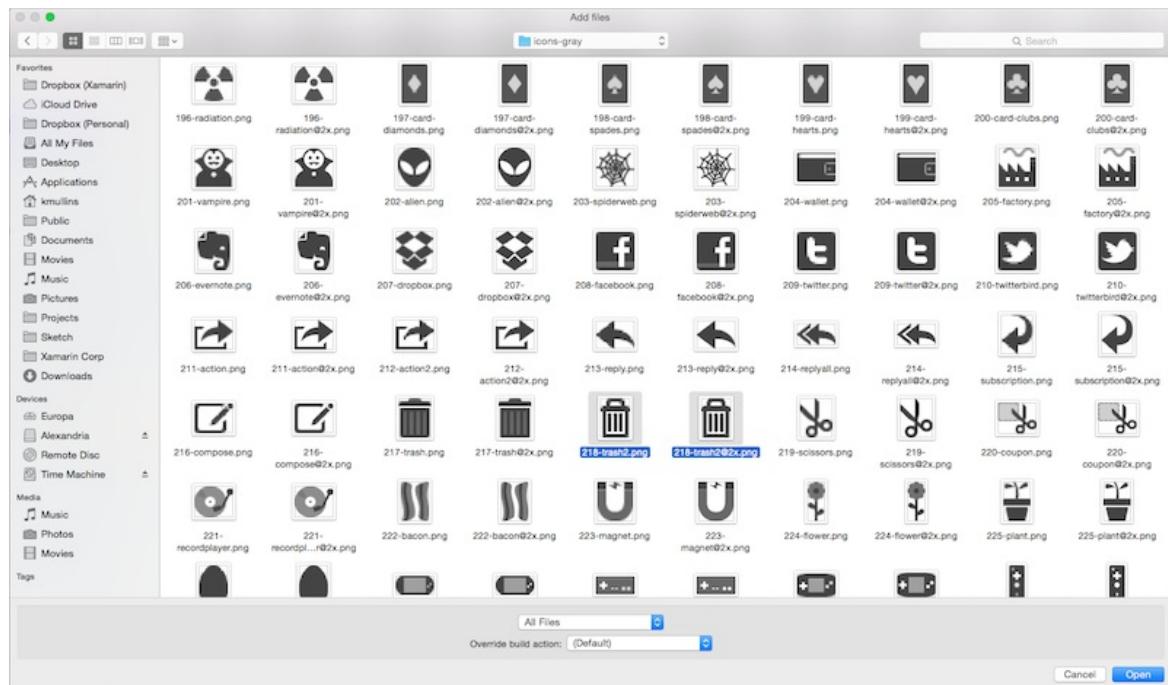
Save the document, return to Visual Studio for Mac to sync with Xcode, run the application, enter some text, select it, and click the **Colors** toolbar item. Notice that the text view automatically works with the color picker:



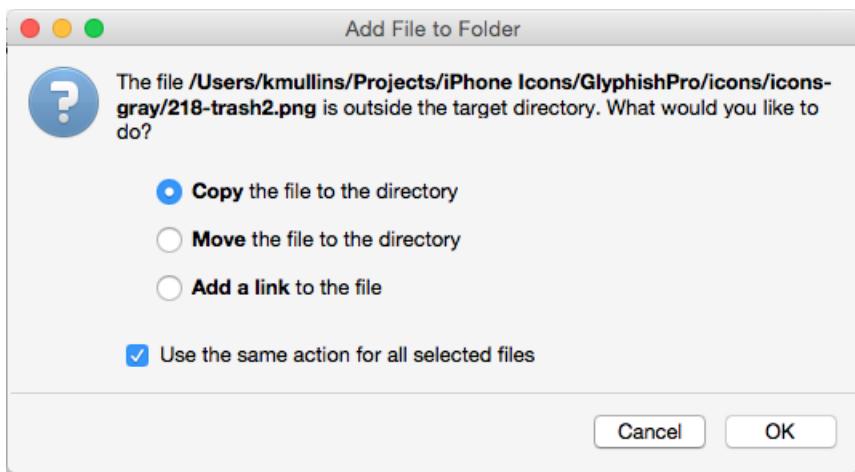
Using images with toolbar items

Using an **Image Toolbar Item**, any bitmap image added to the **Resources** folder (and given a build action of **Bundle Resource**) can be displayed on the toolbar as an icon:

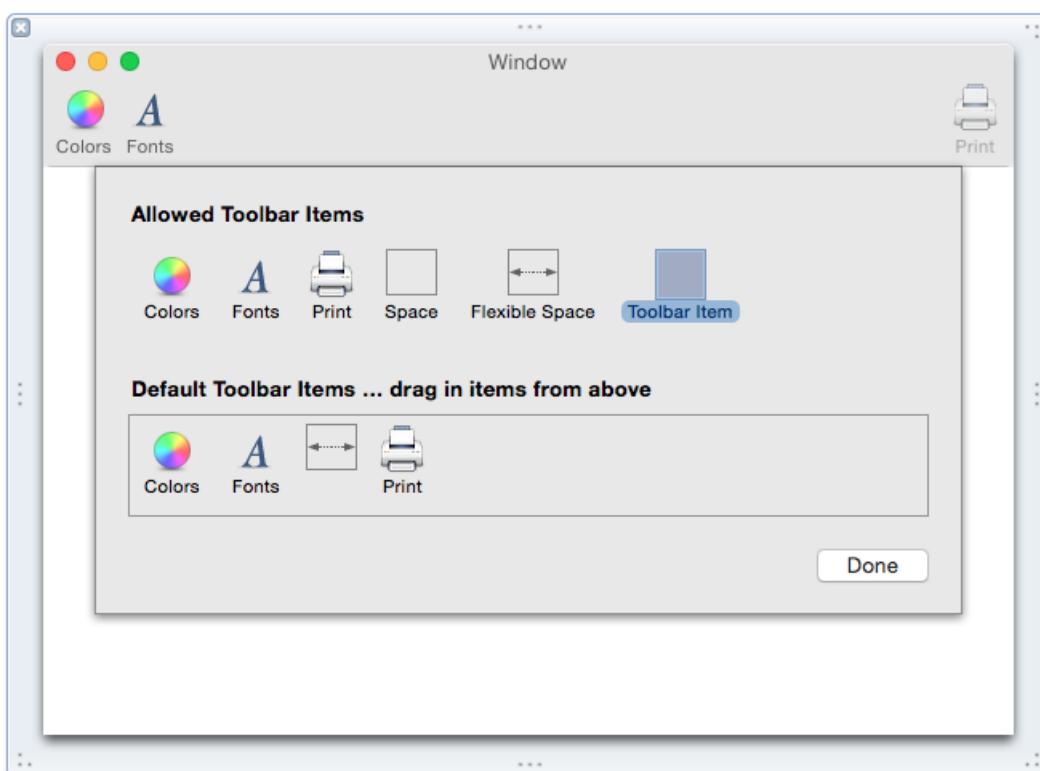
1. In Visual Studio for Mac, in the **Solution Pad**, right-click the **Resources** folder and select **Add > Add Files**.
2. From the **Add Files** dialog box, navigate to the desired images, select them and click the **Open** button:



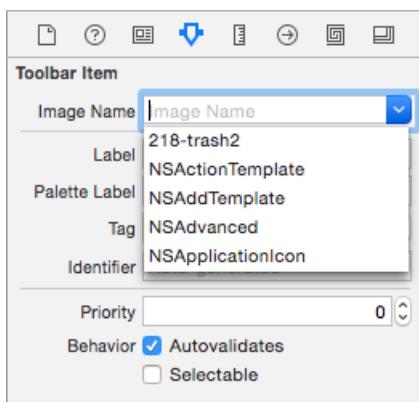
3. Select **Copy**, check **Use the same action for all selected files**, and click **OK**:



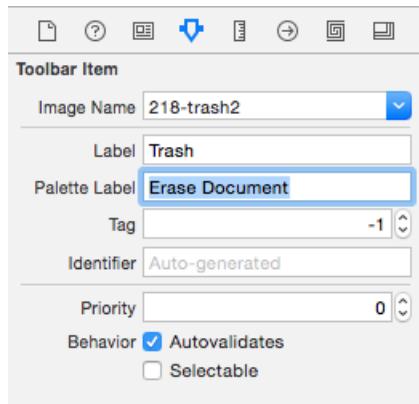
4. In the **Solution Pad**, double-click **MainWindow.xib** to open it in Xcode.
5. Select the toolbar in the **Interface Hierarchy** and click one of its items to open the customization dialog.
6. Drag an **Image Toolbar Item** from the **Library Inspector** to the toolbar's **Allowed Toolbar Items** area:



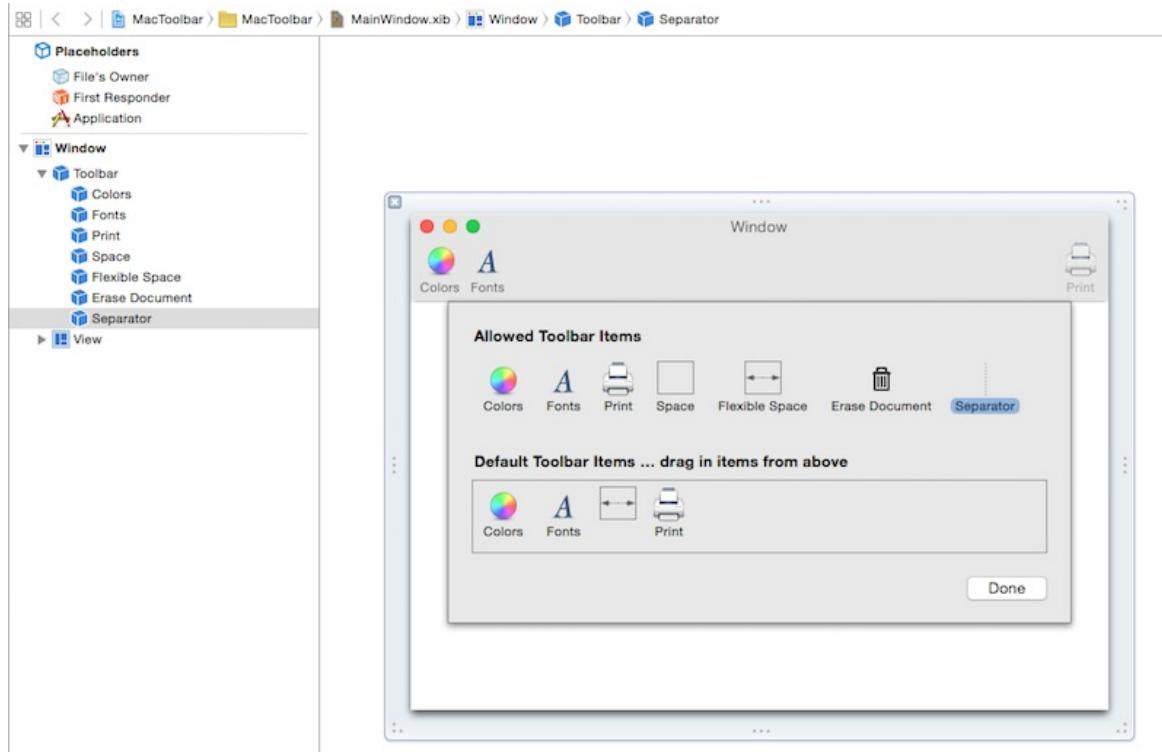
7. In the **Attributes Inspector**, select the image that was just added in Visual Studio for Mac:



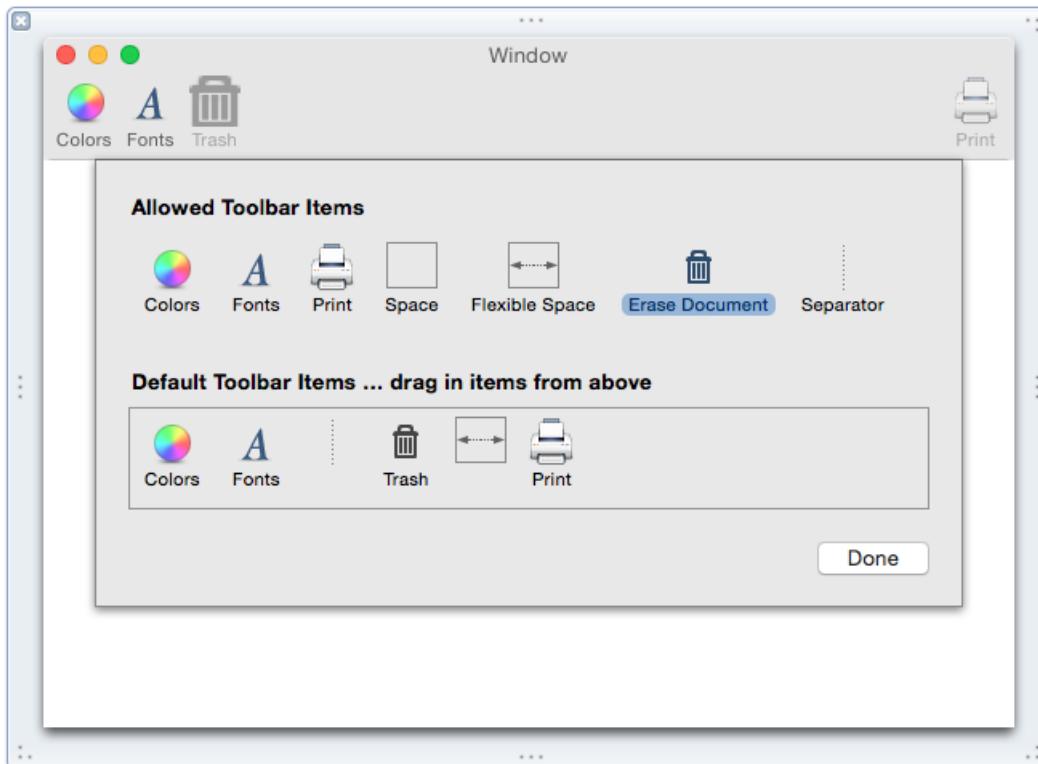
8. Set the Label to "Trash" and the Palette Label to "Erase Document":



9. Drag a Separator Toolbar Item from the Library Inspector to the toolbar's Allowed Toolbar Items area:

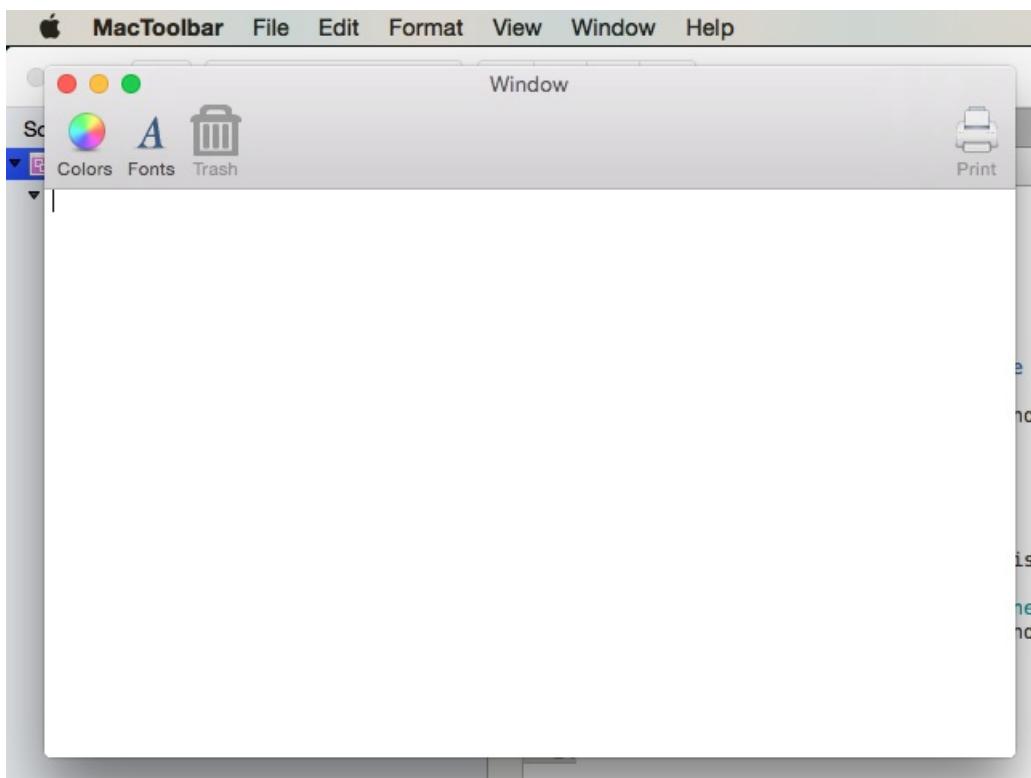


10. Drag the separator item and the "Trash" item to the Default Toolbar Items area and set the order of the toolbar items from left to right as follows (Colors, Fonts, Separator, Trash, Flexible Space, Print):



11. Save changes and return to Visual Studio for Mac to sync with Xcode.

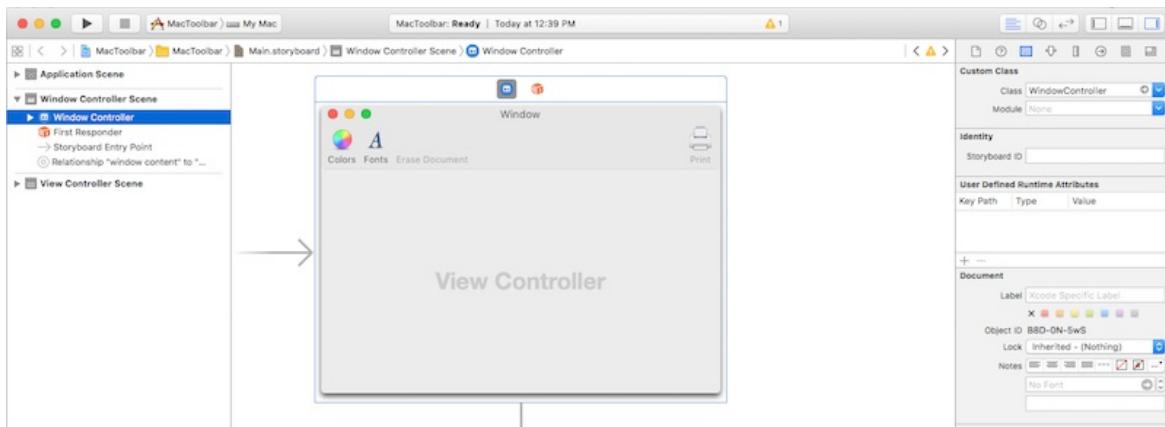
Run the application to verify that the new toolbar is displayed by default:



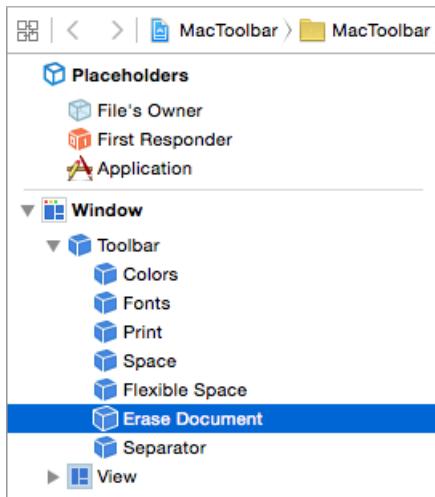
Exposing toolbar items with outlets and actions

To access a toolbar or toolbar item in code, it must be attached to an outlet or an action:

1. In the **Solution Pad**, double-click **Main.storyboard** to open it in Xcode.
2. Ensure that the custom class "WindowController" has been assigned to the main window controller in the **Identity Inspector**:

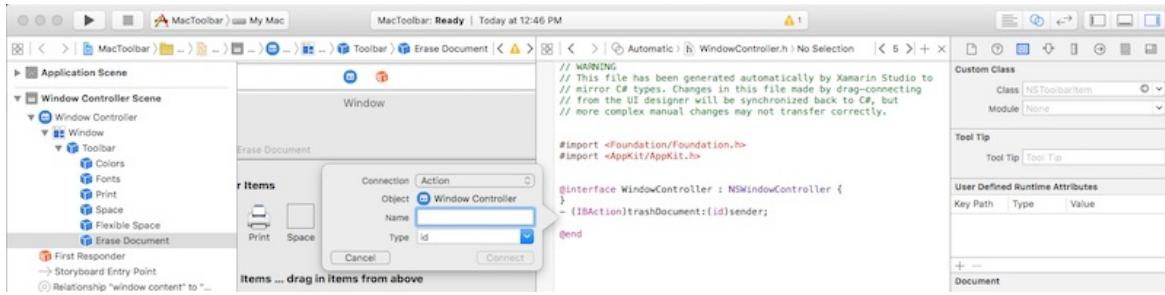


3. Next, select the toolbar item in the **Interface Hierarchy**:

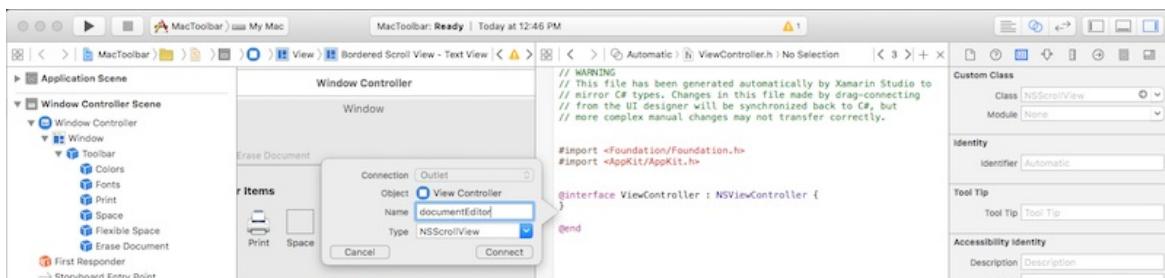


4. Open the Assistant View, select the **WindowController.h** file, and control-drag from the toolbar item to the **WindowController.h** file.

5. Set the **Connection type** to **Action**, enter "trashDocument" for the **Name**, and click the **Connect** button:



6. Expose the **Text View** as an outlet called "documentEditor" in the **ViewController.h** file:



7. Save your changes and return to Visual Studio for Mac to sync with Xcode.

In Visual Studio for Mac, edit the **ViewController.cs** file and add the following code:

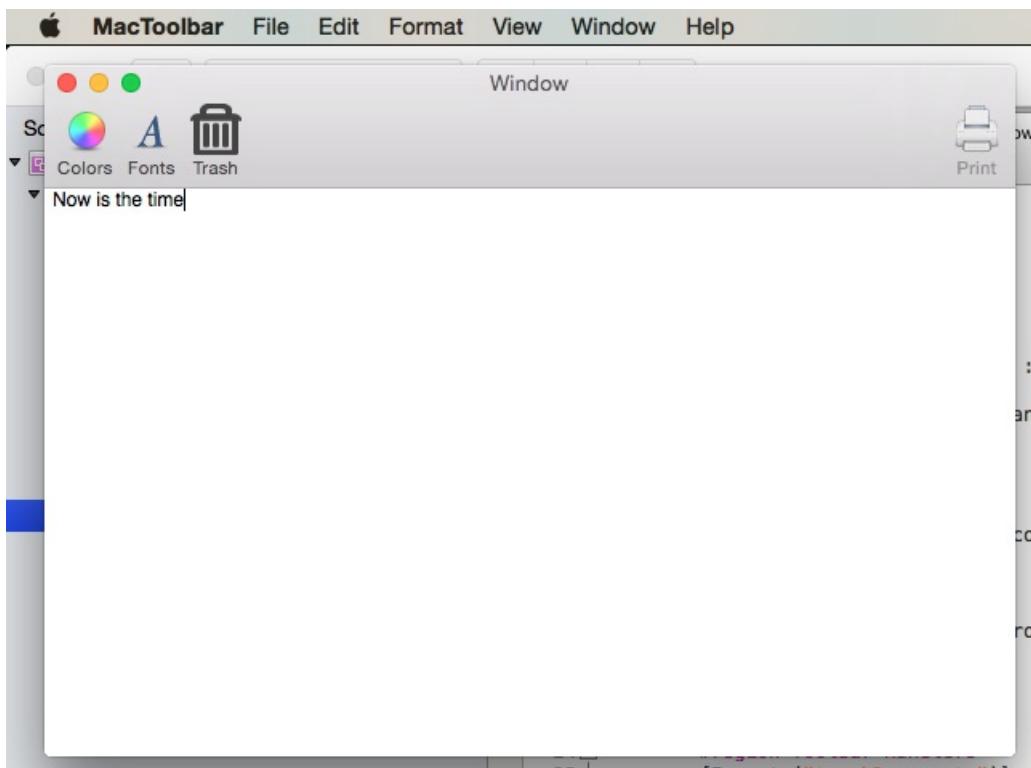
```
public void EraseDocument() {
    documentEditor.Value = "";
}
```

Next, edit the `WindowController.cs` file and add the following code to the bottom of the `WindowController` class:

```
[Export ("trashDocument")]
void TrashDocument (NSObject sender) {

    var controller = ContentViewController as ViewController;
    controller.EraseDocument ();
}
```

When running the application, the **Trash** toolbar item will be active:

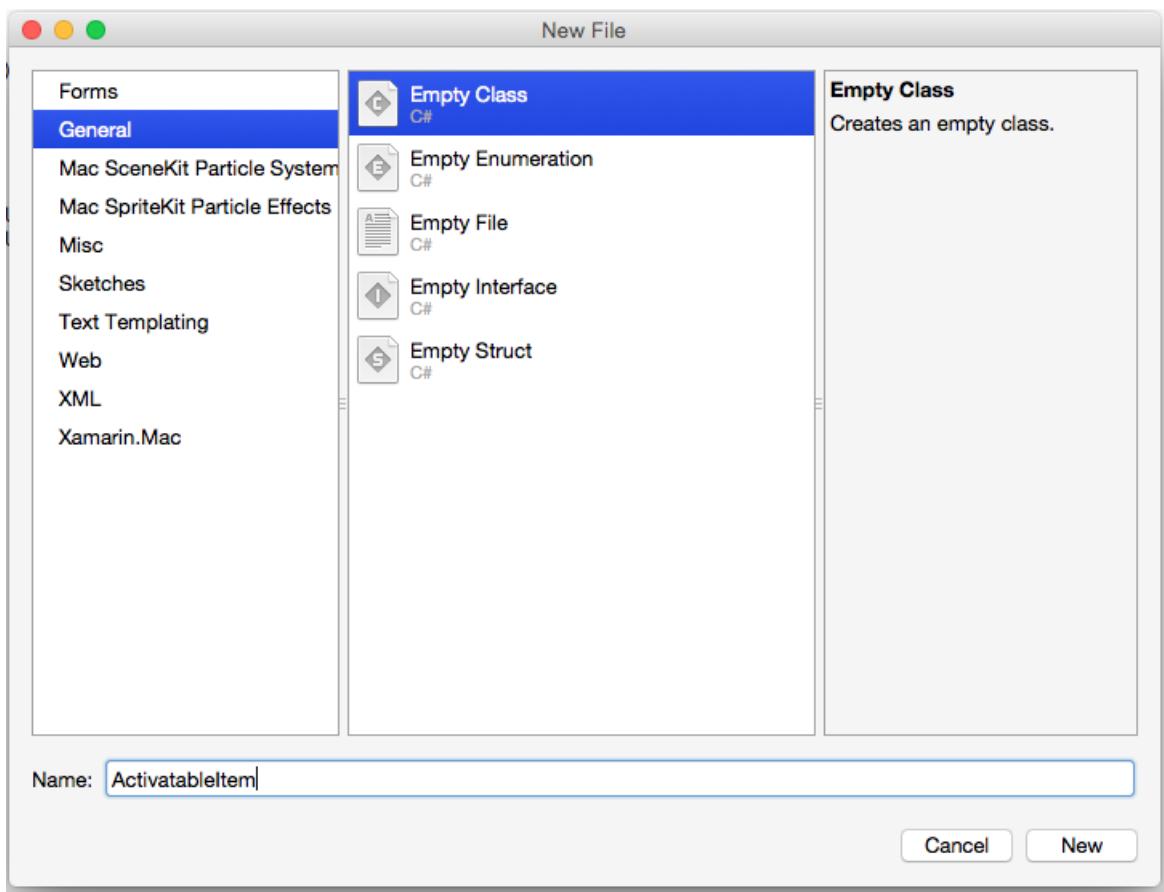


Notice that the **Trash** toolbar item can now be used to delete text.

Disabling toolbar items

To disable an item on a toolbar, create a custom `NSToolbarItem` class and override the `Validate` method. Then, in Interface Builder, assign the custom type to the item that you want to enable/disable.

To create a custom `NSToolbarItem` class, right-click on the project and select **Add > New File...**. Select **General > Empty Class**, enter "ActivatableItem" for the **Name**, and click the **New** button:



Next, edit the `ActivatableItem.cs` file to read as follows:

```

using System;

using Foundation;
using AppKit;

namespace MacToolbar
{
    [Register("ActivatableItem")]
    public class ActivatableItem : NSToolBarItem
    {
        public bool Active { get; set; } = true;

        public ActivatableItem ()
        {
        }

        public ActivatableItem (IntPtr handle) : base (handle)
        {
        }

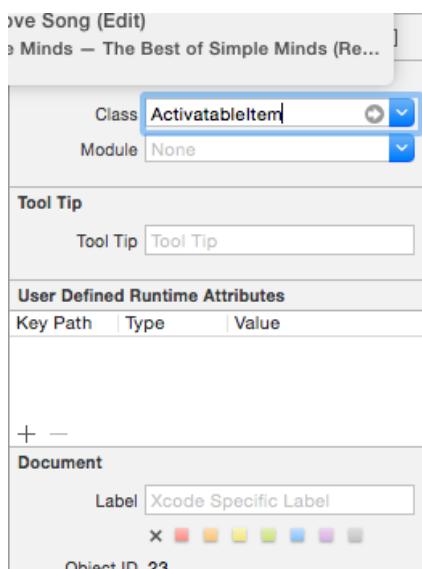
        public ActivatableItem (NSObjectFlag t) : base (t)
        {
        }

        public ActivatableItem (string title) : base (title)
        {
        }

        public override void Validate ()
        {
            base.Validate ();
            Enabled = Active;
        }
    }
}

```

Double-click **Main.storyboard** to open it in Xcode. Select the **Trash** toolbar item created above and change its class to "ActivatableItem" in the **Identity Inspector**:

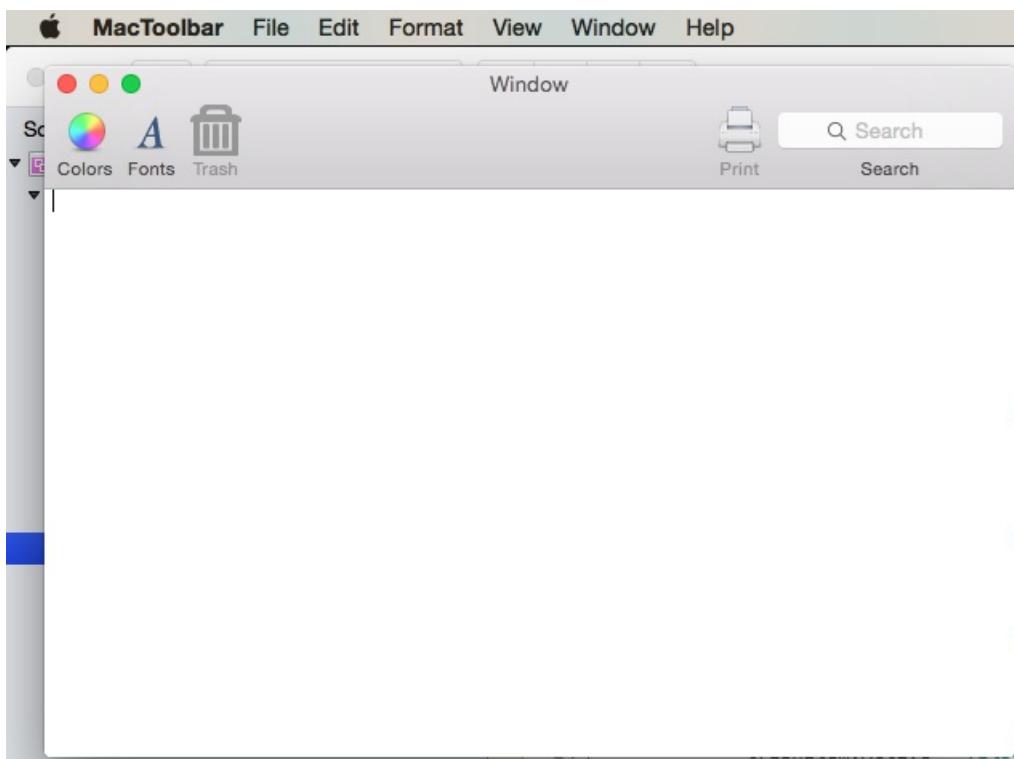


Create an outlet called `trashItem` for the **Trash** toolbar item. Save changes and return to Visual Studio for Mac to sync with Xcode. Finally, open **MainWindow.cs** and update the `AwakeFromNib` method to read as follows:

```
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

    // Disable trash
    trashItem.Active = false;
}
```

Run the application and note that the **Trash** item is now disabled in the toolbar:



Summary

This article has taken a detailed look at working with toolbars and toolbar items in a Xamarin.Mac application. It described how to create and maintain toolbars in Xcode's Interface Builder, how some UI controls automatically work with toolbar items, how to work with toolbars in C# code, and how to enable and disable toolbar items.

Related Links

- [MacToolbar \(sample\)](#)
- [Hello, Mac](#)
- [Human Interface Guidelines for Toolbars](#)
- [Introduction to Toolbars](#)

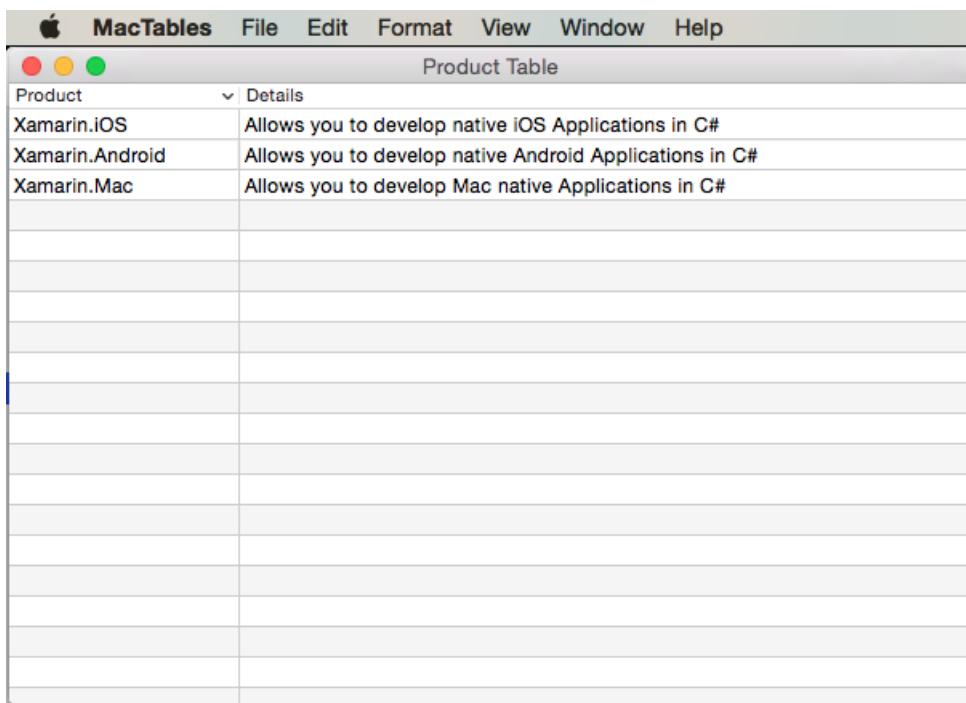
Table Views in Xamarin.Mac

3/5/2021 • 25 minutes to read • [Edit Online](#)

This article covers working with table views in a Xamarin.Mac application. It describes creating table views in Xcode and Interface Builder and interacting with them in code.

When working with C# and .NET in a Xamarin.Mac application, you have access to the same Table Views that a developer working in *Objective-C* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, you can use Xcode's *Interface Builder* to create and maintain your Table Views (or optionally create them directly in C# code).

A Table View displays data in a tabular format containing one or more columns of information in multiple rows. Based on the type of Table View being created, the user can sort by column, reorganize columns, add columns, remove columns or edit the data contained within the table.



In this article, we'll cover the basics of working with Table Views in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` commands used to wire-up your C# classes to Objective-C objects and UI Elements.

Introduction to Table Views

A Table View displays data in a tabular format containing one or more columns of information in multiple rows. Table Views are displayed inside of Scroll Views (`NSScrollView`) and starting with macOS 10.7, you can use any `NSView` instead of Cells (`NSCell`) to display both rows and columns. That said, you can still use `NSCell` however, you'll typically subclass `NSTableCellView` and create your custom rows and columns.

A Table View does not store its own data, instead it relies on a Data Source (`NITableViewDataSource`) to provide

both the rows and columns required, on a as-needed basis.

A Table View's behavior can be customized by providing a subclass of the Table View Delegate (`NSTableViewDelegate`) to support table column management, type to select functionality, row selection and editing, custom tracking, and custom views for individual columns and rows.

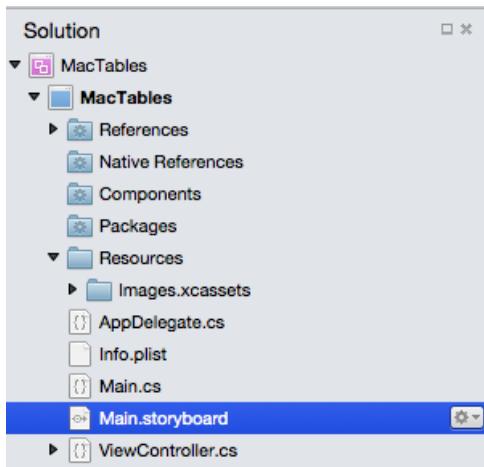
When creating Table Views, Apple suggests the following:

- Allow the user to sort the table by clicking on a Column Headers.
- Create Column Headers that are nouns or short noun phrases that describe the data being shown in that column.

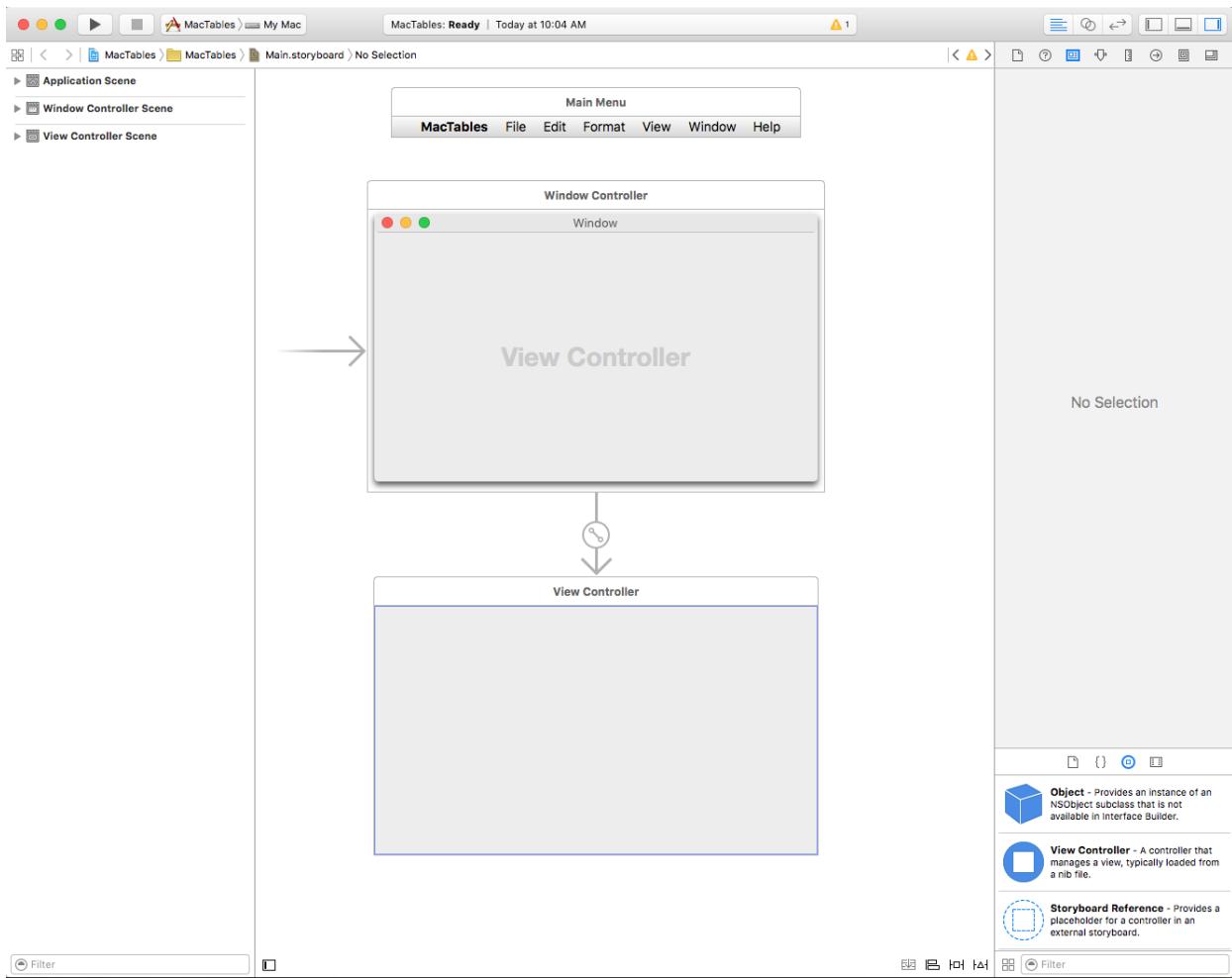
For more information, please see the [Content Views](#) section of Apple's [OS X Human Interface Guidelines](#).

Creating and Maintaining Table Views in Xcode

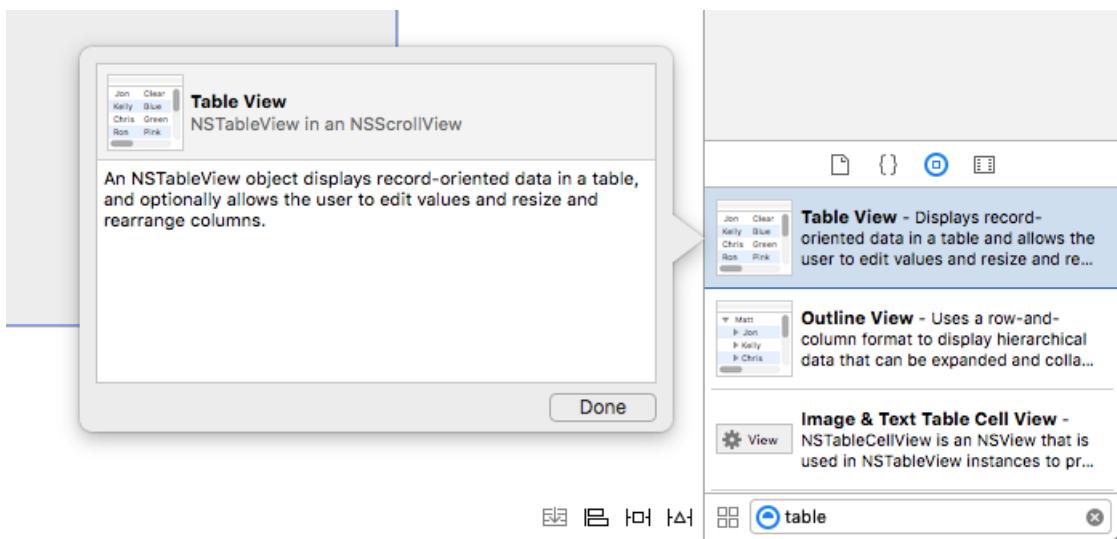
When you create a new Xamarin.Mac Cocoa application, you get a standard blank, window by default. This windows is defined in a `.Storyboard` file automatically included in the project. To edit your windows design, in the **Solution Explorer**, double click the `Main.storyboard` file:



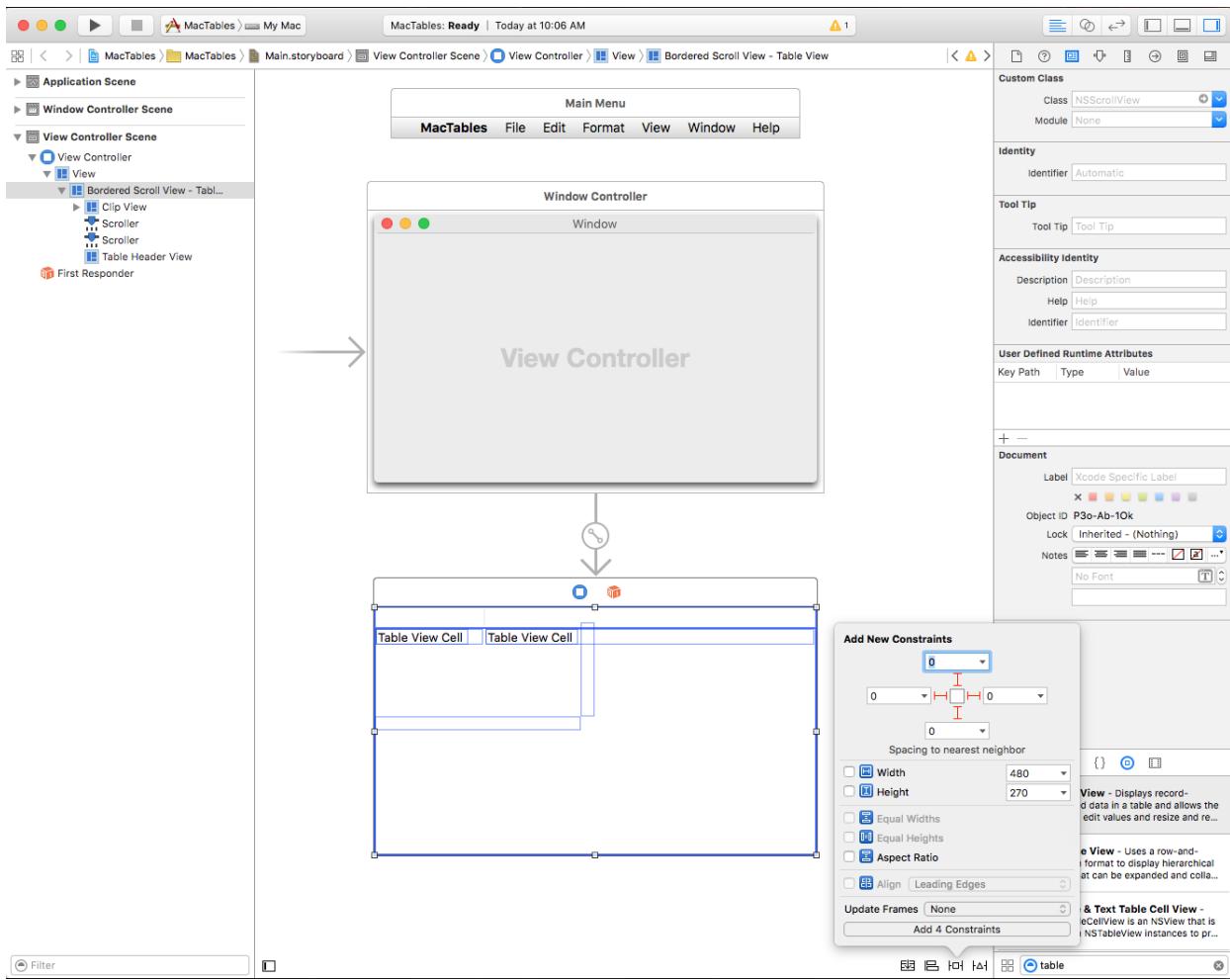
This will open the window design in Xcode's Interface Builder:



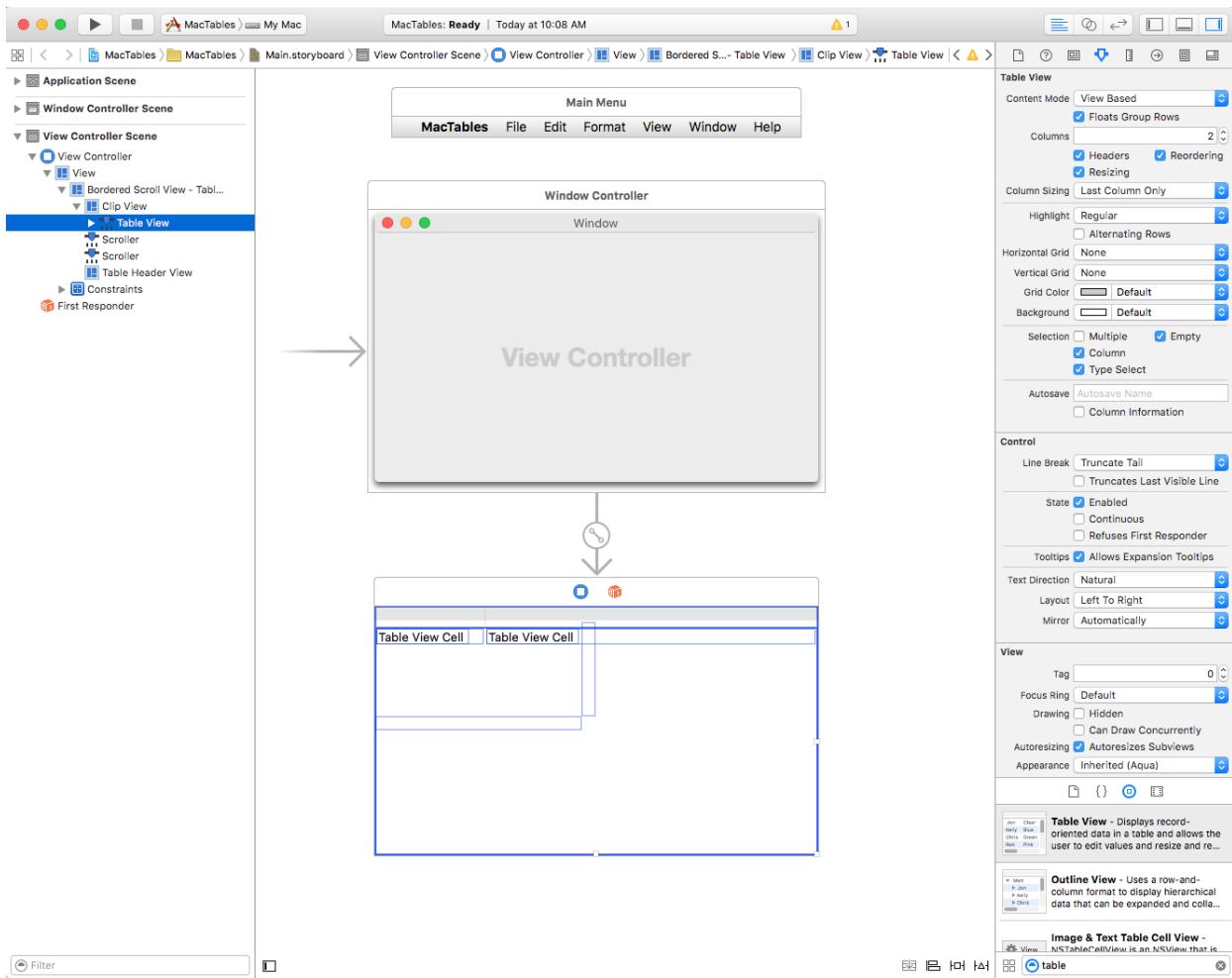
Type `table` into the Library Inspector's Search Box to make it easier to find the Table View controls:



Drag a Table View onto the View Controller in the Interface Editor, make it fill the content area of the View Controller and set it to where it shrinks and grows with the window in the Constraint Editor:



Select the Table View in the **Interface Hierarchy** and the following properties are available in the **Attribute Inspector**:



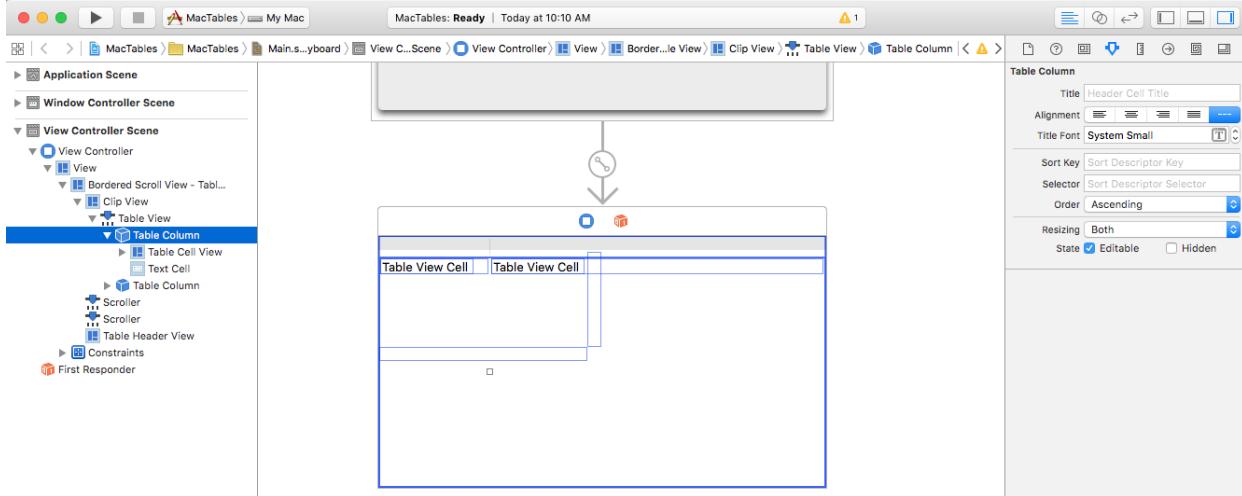
- **Content Mode** - Allows you to use either Views (`NSView`) or Cells (`NSCell`) to display the data in the rows and columns. Starting with macOS 10.7, you should use Views.
- **Floats Group Rows** - If `true`, the Table View will draw grouped cells as if they are floating.
- **Columns** - Defines the number of columns displayed.
- **Headers** - If `true`, the columns will have Headers.
- **Reordering** - If `true`, the user will be able to drag reorder the columns in the table.
- **Resizing** - If `true`, the user will be able to drag column Headers to resize columns.
- **Column Sizing** - Controls how the table will auto size columns.
- **Highlight** - Controls the type of highlighting the table uses when a cell is selected.
- **Alternate Rows** - If `true`, ever other row will have a different background color.
- **Horizontal Grid** - Selects the type of border drawn between cells horizontally.
- **Vertical Grid** - Selects the type of border drawn between cells vertically.
- **Grid Color** - Sets the cell border color.
- **Background** - Sets the cell background color.
- **Selection** - Allow you to control how the user can select cells in the table as:
 - **Multiple** - If `true`, the user can select multiple rows and columns.
 - **Column** - If `true`, the user can select columns.
 - **Type Select** - If `true`, the user can type a character to select a row.
 - **Empty** - If `true`, the user is not required to select a row or column, the table allows for no selection at all.
- **Autosave** - The name that the tables format is automatically save under.
- **Column Information** - If `true`, the order and width of the columns will be automatically saved.
- **Line Breaks** - Select how the cell handles line breaks.

- **Truncates Last Visible Line** - If `true`, the cell will be truncated in the data can not fit inside it's bounds.

IMPORTANT

Unless you are maintaining a legacy Xamarin.Mac application, `NSView` based Table Views should be used over `NSCell` based Table Views. `NSCell` is considered legacy and may not be supported going forward.

Select a Table Column in the **Interface Hierarchy** and the following properties are available in the **Attribute Inspector**:

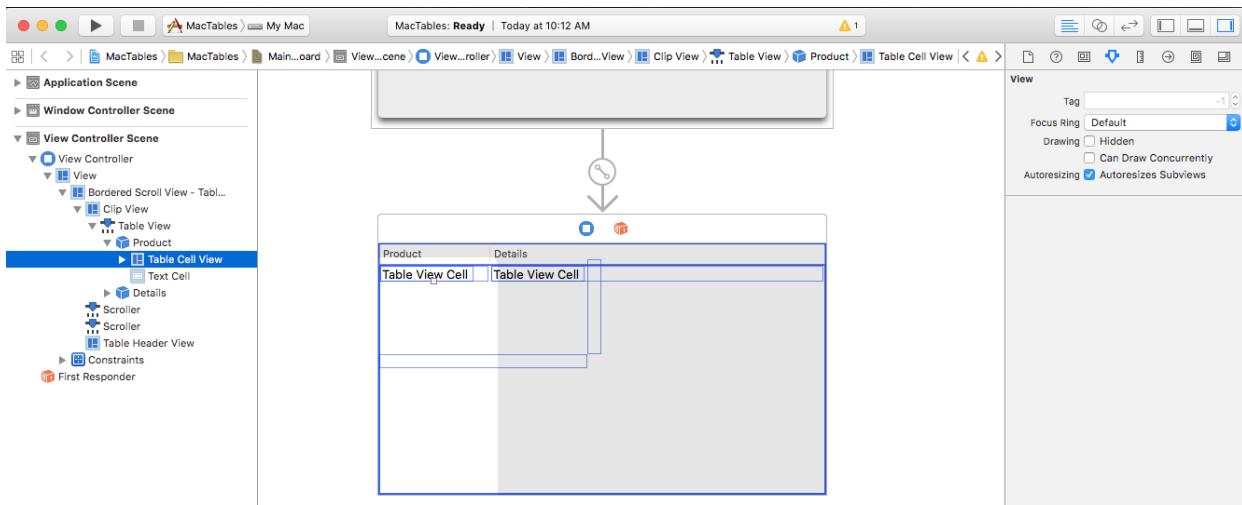


- **Title** - Sets the title of the column.
- **Alignment** - Set the alignment of the text within the cells.
- **Title Font** - Selects the font for the cell's Header text.
- **Sort Key** - Is the key used to sort data in the column. Leave blank if the user cannot sort this column.
- **Selector** - Is the **Action** used to perform the sort. Leave blank if the user cannot sort this column.
- **Order** - Is the sort order for the columns data.
- **Resizing** - Selects the type of resizing for the column.
- **Editable** - If `true`, the user can edit cells in a cell based table.
- **Hidden** - If `true`, the column is hidden.

You can also resize the column by dragging it's handle (vertically centered on the column's right side) left or right.

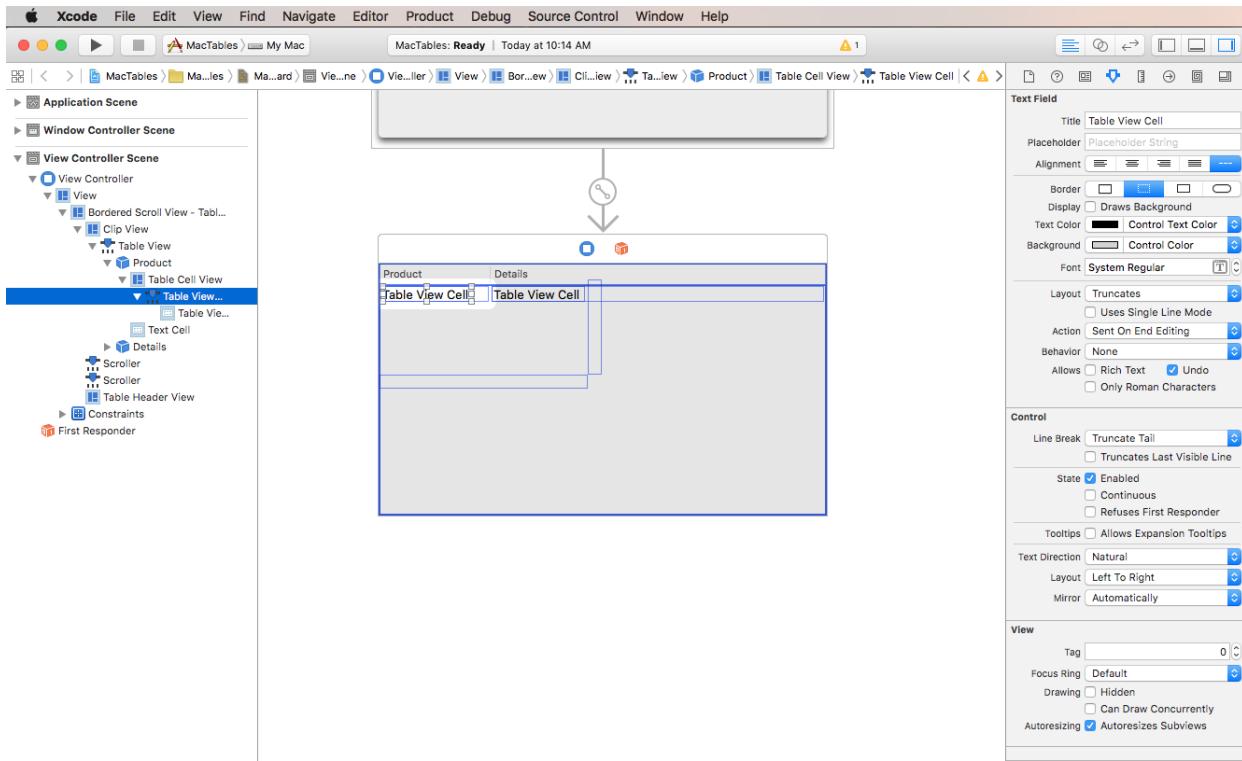
Let's select the each Column in our Table View and give the first column a **Title** of `Product` and the second one `Details`.

Select a Table Cell View (`NSTableViewController`) in the **Interface Hierarchy** and the following properties are available in the **Attribute Inspector**:



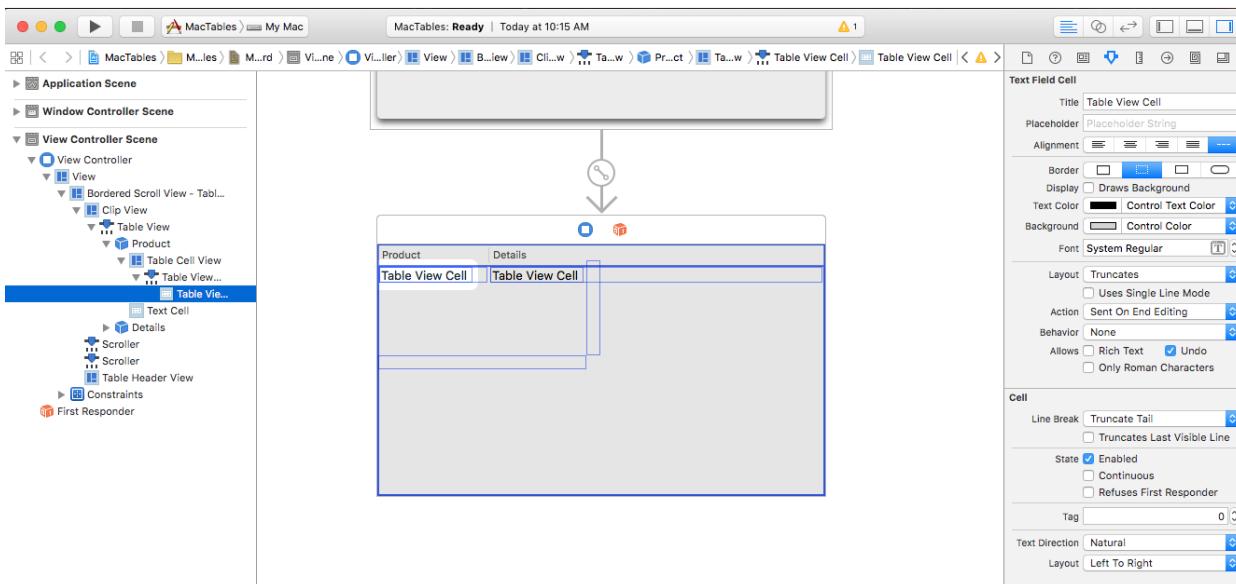
These are all of the properties of a standard View. You also have the option of resizing the rows for this column here.

Select a Table View Cell (by default, this is a `NSTextField`) in the **Interface Hierarchy** and the following properties are available in the **Attribute Inspector**:



You'll have all the properties of a standard text field to set here. By default, a standard Text Field is used to display data for a cell in a column.

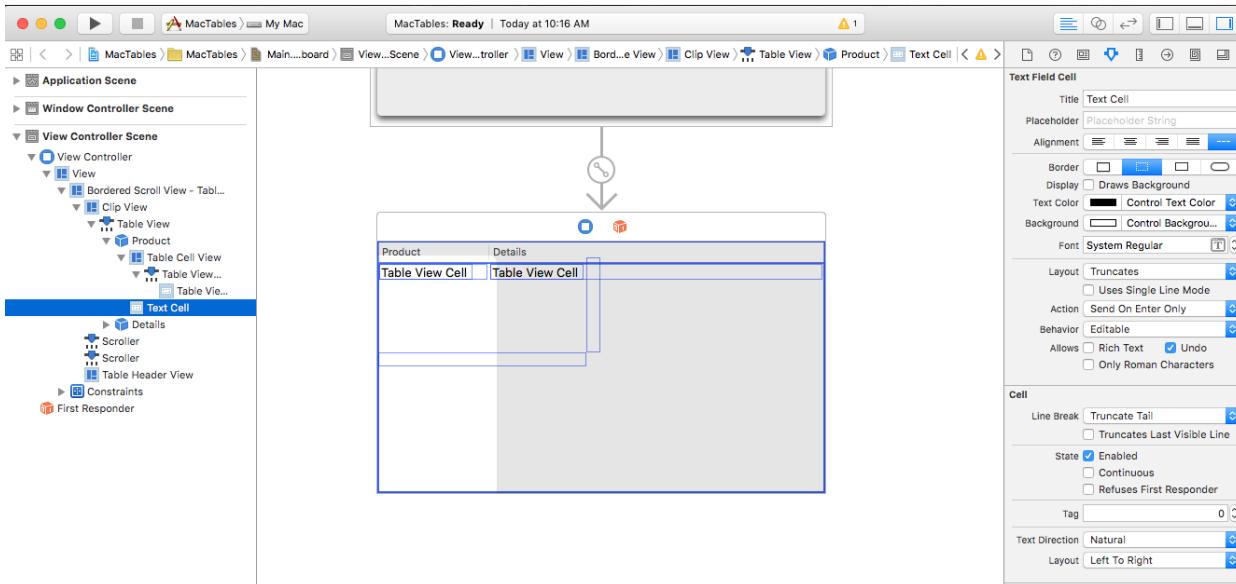
Select a Table Cell View (`NSTableFieldCell`) in the **Interface Hierarchy** and the following properties are available in the **Attribute Inspector**:



The most important settings here are:

- **Layout** - Select how cells in this column are laid out.
- **Uses Single Line Mode** - If `true`, the cell is limited to a single line.
- **First Runtime Layout Width** - If `true`, the cell will prefer the width set for it (either manually or automatically) when it is displayed the first time the application is run.
- **Action** - Controls when the **Edit Action** is sent for the cell.
- **Behavior** - Defines if a cell is selectable or editable.
- **Rich Text** - If `true`, the cell can display formatted and styled text.
- **Undo** - If `true`, the cell assumes responsibility for its undo behavior.

Select the Table Cell View (`NSTextFieldCell`) at the bottom of a Table Column in the **Interface Hierarchy**:



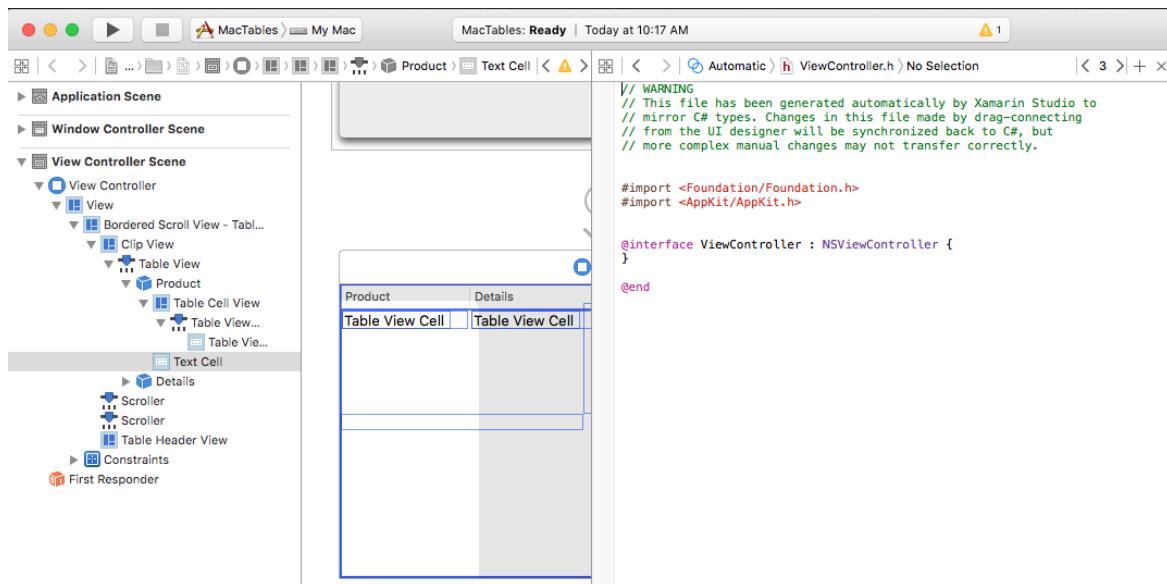
This allows you to edit the Table Cell View used as the base *Pattern* for all cells created for the given column.

Adding Actions and Outlets

Just like any other Cocoa UI control, we need to expose our Table View and its columns and cells to C# code using **Actions** and **Outlets** (based on the functionality required).

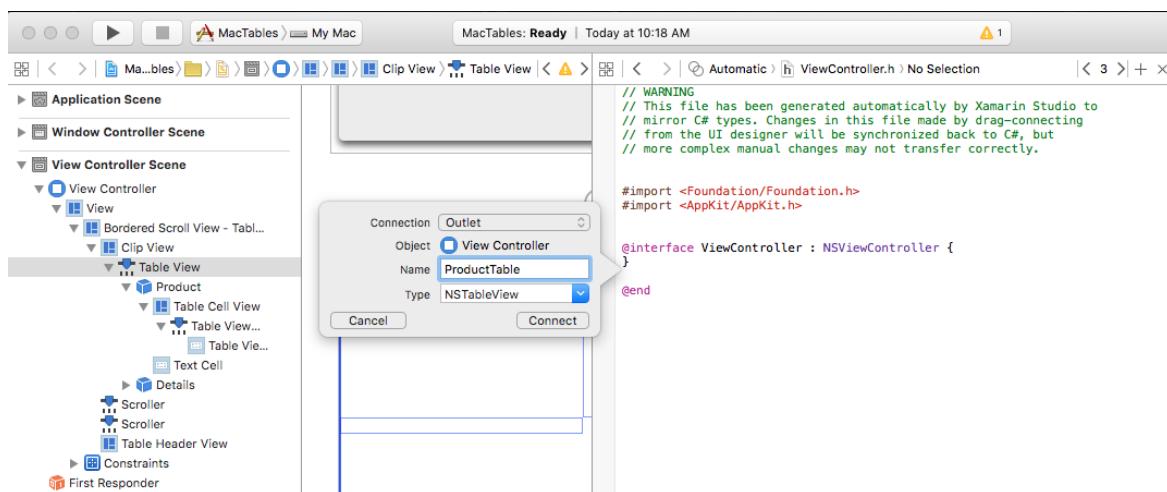
The process is the same for any Table View element that we want to expose:

1. Switch to the **Assistant Editor** and ensure that the `ViewController.h` file is selected:

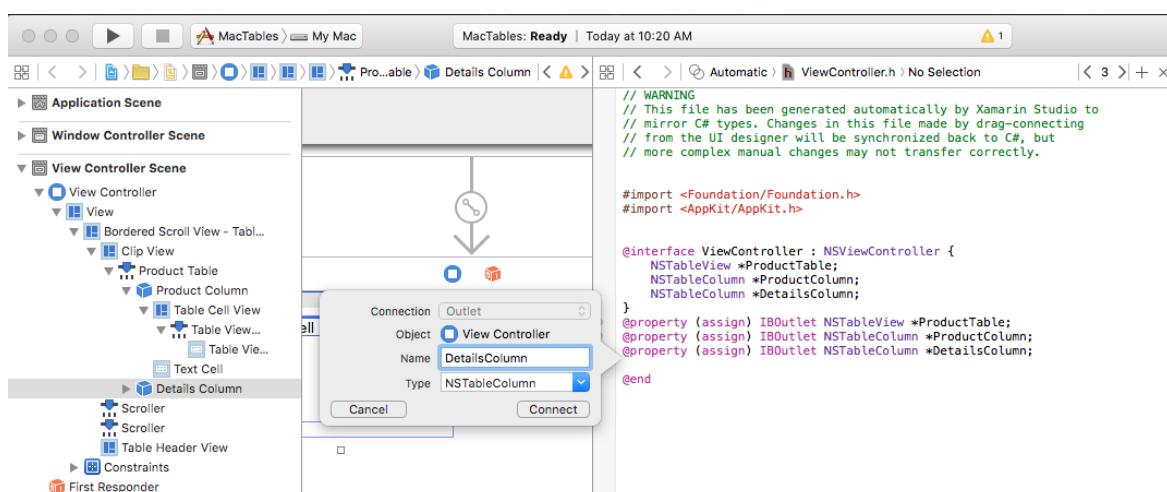


2. Select the Table View from the **Interface Hierarchy**, control-click and drag to the `ViewController.h` file.

3. Create an **Outlet** for the Table View called `ProductTable`:



4. Create **Outlets** for the tables columns as well called `ProductColumn` and `DetailsColumn`:



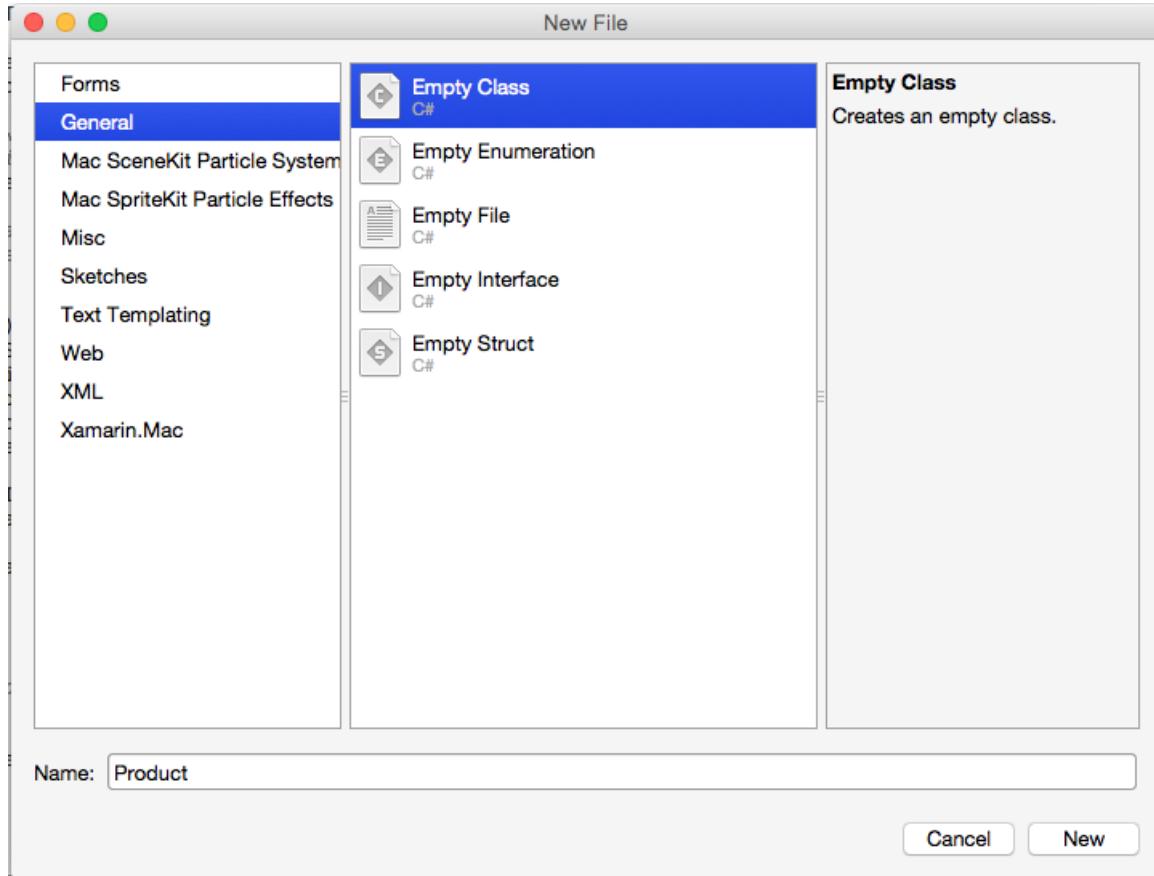
5. Save your changes and return to Visual Studio for Mac to sync with Xcode.

Next, we'll write the code to display some data for the table when the application is run.

Populating the Table View

With our Table View designed in Interface Builder and exposed via an **Outlet**, next we need to create the C# code to populate it.

First, let's create a new **Product** class to hold the information for the individual rows. In the **Solution Explorer**, right-click the Project and select **Add > New File...**. Select **General > Empty Class**, enter **Product** for the **Name** and click the **New** button:



Make the **Product.cs** file look like the following:

```
using System;

namespace MacTables
{
    public class Product
    {
        #region Computed Properties
        public string Title { get; set; } = "";
        public string Description { get; set; } = "";
        #endregion

        #region Constructors
        public Product ()
        {
        }

        public Product (string title, string description)
        {
            this.Title = title;
            this.Description = description;
        }
        #endregion
    }
}
```

Next, we need to create a subclass of `NSTableViewDataSource` to provide the data for our table as it is requested. In the **Solution Explorer**, right-click the Project and select **Add > New File...**. Select **General > Empty Class**, enter `ProductTableDataSource` for the **Name** and click the **New** button.

Edit the `ProductTableDataSource.cs` file and make it look like the following:

```
using System;
using AppKit;
using CoreGraphics;
using Foundation;
using System.Collections;
using System.Collections.Generic;

namespace MacTables
{
    public class ProductTableDataSource : NSTableViewDataSource
    {
        #region Public Variables
        public List<Product> Products = new List<Product>();
        #endregion

        #region Constructors
        public ProductTableDataSource ()
        {
        }
        #endregion

        #region Override Methods
        public override nint GetRowCount (NSTableView tableView)
        {
            return Products.Count;
        }
        #endregion
    }
}
```

This class has storage for our Table View's items and overrides the `GetRowCount` to return the number of rows in the table.

Finally, we need to create a subclass of `NSTableDelegate` to provide the behavior for our table. In the **Solution Explorer**, right-click the Project and select **Add > New File...**. Select **General > Empty Class**, enter `ProductTableDelegate` for the **Name** and click the **New** button.

Edit the `ProductTableDelegate.cs` file and make it look like the following:

```

using System;
using AppKit;
using CoreGraphics;
using Foundation;
using System.Collections;
using System.Collections.Generic;

namespace MacTables
{
    public class ProductTableDelegate: NSTableViewDelegate
    {
        #region Constants
        private const string CellIdentifier = "ProdCell";
        #endregion

        #region Private Variables
        private ProductTableDataSource DataSource;
        #endregion

        #region Constructors
        public ProductTableDelegate (ProductTableDataSource datasource)
        {
            this.DataSource = datasource;
        }
        #endregion

        #region Override Methods
        public override NSView GetViewForItem (NSTableView tableView, NSTableColumn TableColumn, nint row)
        {
            // This pattern allows you reuse existing views when they are no-longer in use.
            // If the returned view is null, you instance up a new view
            // If a non-null view is returned, you modify it enough to reflect the new data
            NSTextField view = (NSTextField)tableView.MakeView (CellIdentifier, this);
            if (view == null) {
                view = new NSTextField ();
                view.Identifier = CellIdentifier;
                view.BackgroundColor = NSColor.Clear;
                view.Bordered = false;
                view.Selectable = false;
                viewEditable = false;
            }

            // Setup view based on the column selected
            switch (TableColumn.Title) {
                case "Product":
                    view.StringValue = DataSource.Products [(int)row].Title;
                    break;
                case "Details":
                    view.StringValue = DataSource.Products [(int)row].Description;
                    break;
            }

            return view;
        }
        #endregion
    }
}

```

When we create an instance of the `ProductTableDelegate`, we also pass in an instance of the `ProductTableDataSource` that provides the data for the table. The `GetViewForItem` method is responsible for returning a view (data) to display the cell for a give column and row. If possible, an existing view will be reused to display the cell, if not a new view must be created.

To populate the table, let's edit the `viewController.cs` file and make the `AwakeFromNib` method look like the following:

```
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

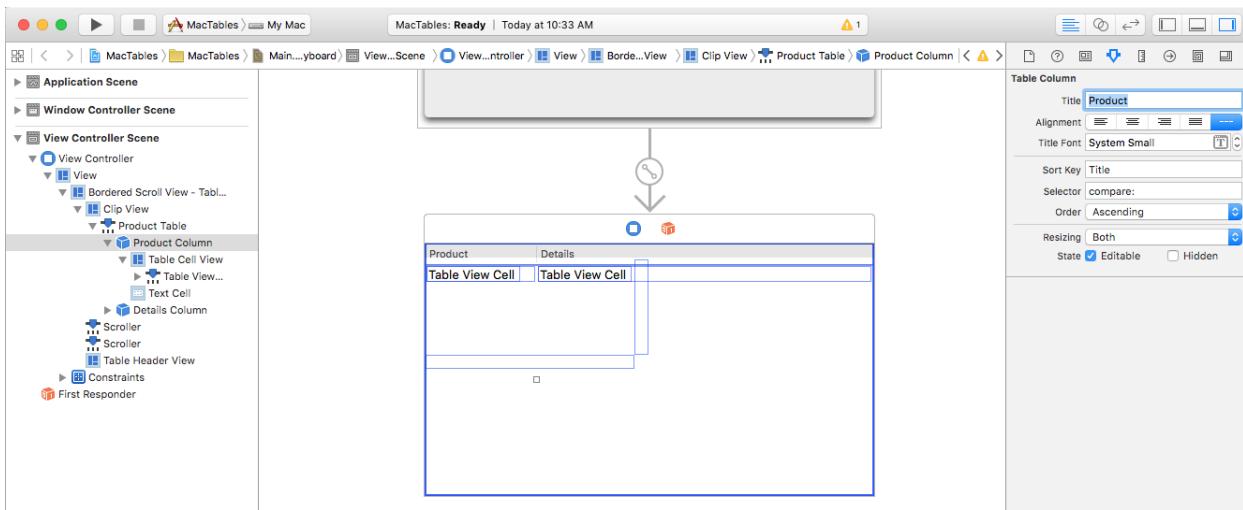
    // Create the Product Table Data Source and populate it
    var DataSource = new ProductTableDataSource ();
    DataSource.Products.Add (new Product ("Xamarin.iOS", "Allows you to develop native iOS Applications in C#"));
    DataSource.Products.Add (new Product ("Xamarin.Android", "Allows you to develop native Android Applications in C#"));
    DataSource.Products.Add (new Product ("Xamarin.Mac", "Allows you to develop Mac native Applications in C#"));

    // Populate the Product Table
    ProductTable.DataSource = DataSource;
    ProductTable.Delegate = new ProductTableDelegate (DataSource);
}
```

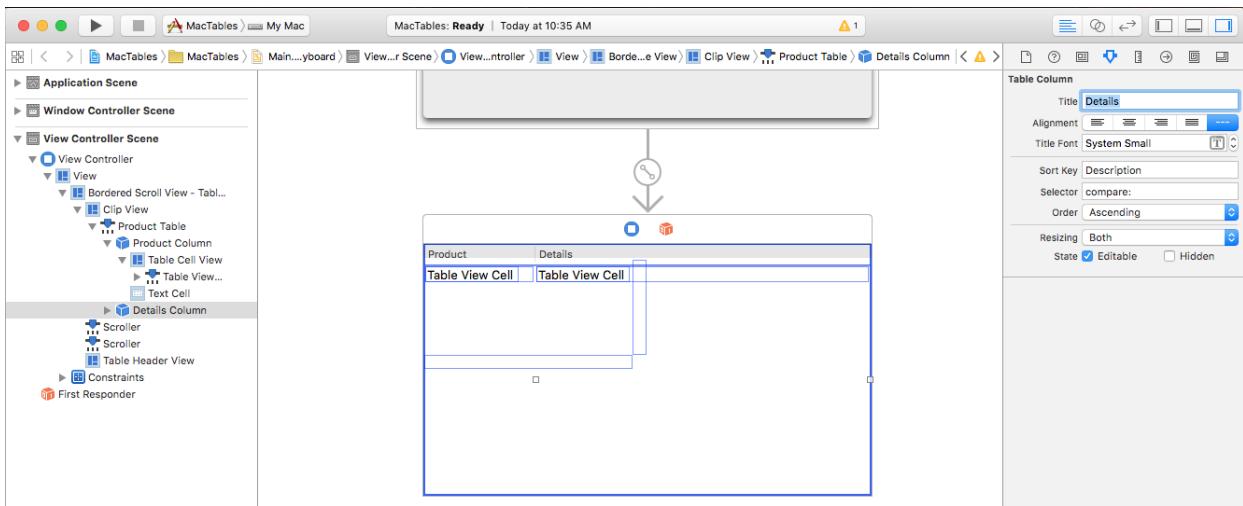
If we run the application, the following is displayed:

Sorting by Column

Let's allow the user to sort the data in the table by clicking on a Column Header. First, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Select the `Product` column, enter `Title` for the `Sort Key`, `compare:` for the `Selector` and select `Ascending` for the `Order`:



Select the `Details` column, enter `Description` for the `Sort Key`, `compare:` for the `Selector` and select `Ascending` for the `Order`:



Save your changes and return to Visual Studio for Mac to sync with Xcode.

Now let's edit the `ProductTableDataSource.cs` file and add the following methods:

```

public void Sort(string key, bool ascending) {

    // Take action based on key
    switch (key) {
        case "Title":
            if (ascending) {
                Products.Sort ((x, y) => x.Title.CompareTo (y.Title));
            } else {
                Products.Sort ((x, y) => -1 * x.Title.CompareTo (y.Title));
            }
            break;
        case "Description":
            if (ascending) {
                Products.Sort ((x, y) => x.Description.CompareTo (y.Description));
            } else {
                Products.Sort ((x, y) => -1 * x.Description.CompareTo (y.Description));
            }
            break;
    }

}

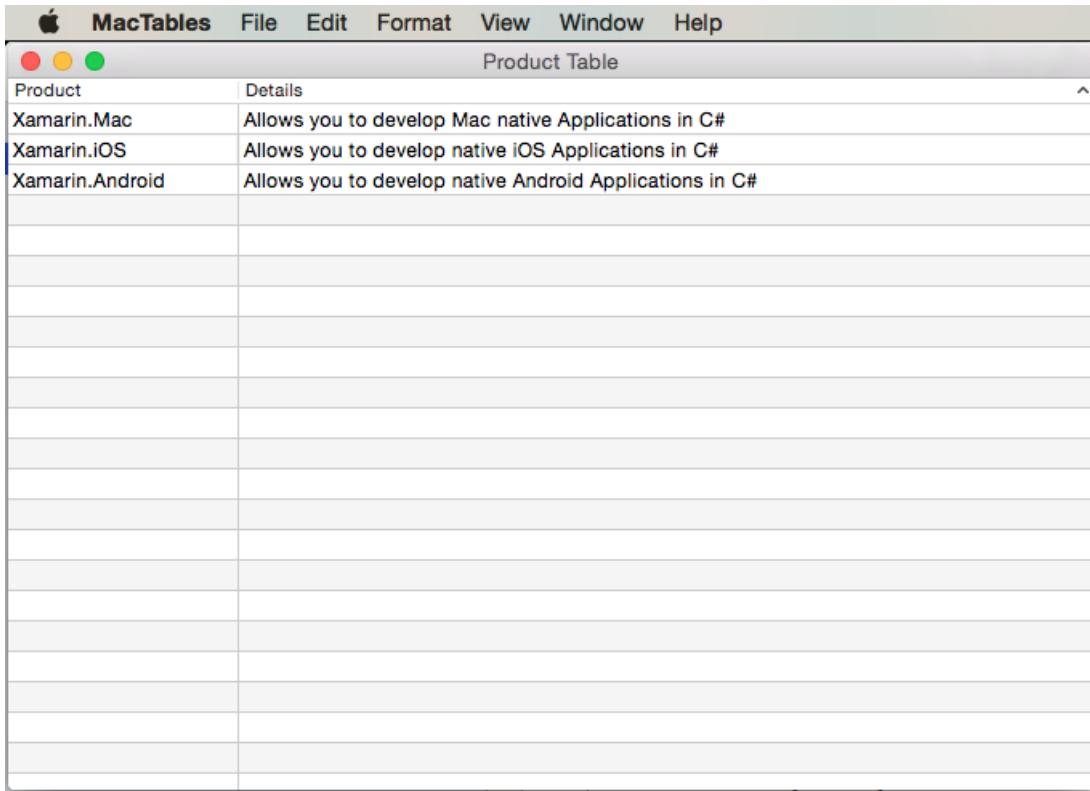
public override void SortDescriptorsChanged (NSTableView tableView, NSSortDescriptor[] oldDescriptors)
{
    // Sort the data
    if (oldDescriptors.Length > 0) {
        // Update sort
        Sort (oldDescriptors [0].Key, oldDescriptors [0].Ascending);
    } else {
        // Grab current descriptors and update sort
        NSSortDescriptor[] tbSort = tableView.SortDescriptors;
        Sort (tbSort[0].Key, tbSort[0].Ascending);
    }

    // Refresh table
    tableView.ReloadData ();
}

```

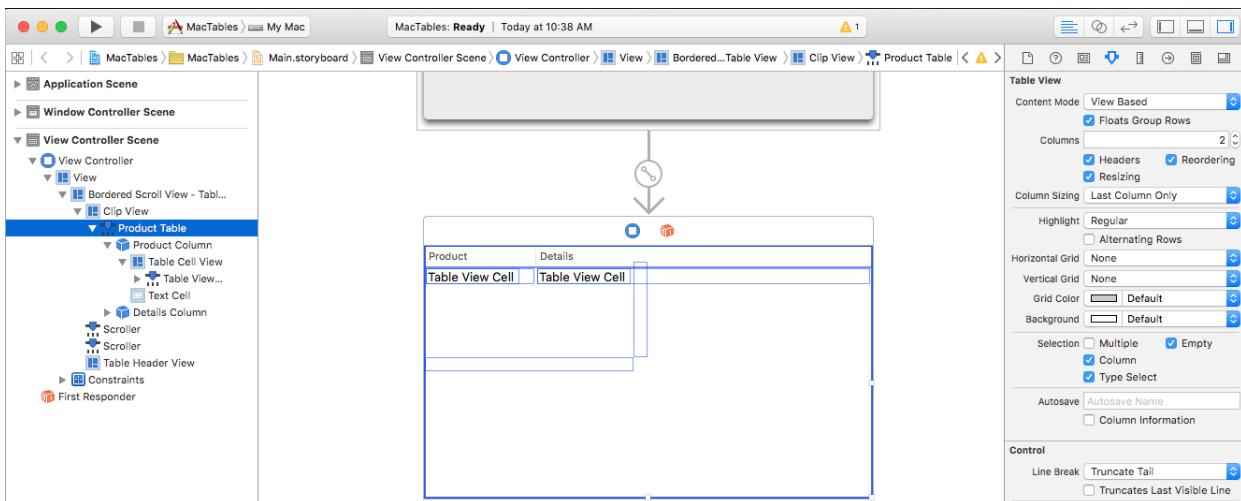
The `Sort` method allow us to sort the data in the Data Source based on a given `Product` class field in either ascending or descending order. The overridden `SortDescriptorsChanged` method will be called every time the user clicks on a Column Heading. It will be passed the `Key` value that we set in Interface Builder and the sort order for that column.

If we run the application and click in the Column Headers, the rows will be sorted by that column:



Row Selection

If you want to allow the user to select a single row, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Select the Table View in the **Interface Hierarchy** and uncheck the **Multiple** checkbox in the **Attribute Inspector**:



Save your changes and return to Visual Studio for Mac to sync with Xcode.

Next, edit the `ProductTableDelegate.cs` file and add the following method:

```
public override bool ShouldSelectRow (NSTableView tableView, nint row)
{
    return true;
}
```

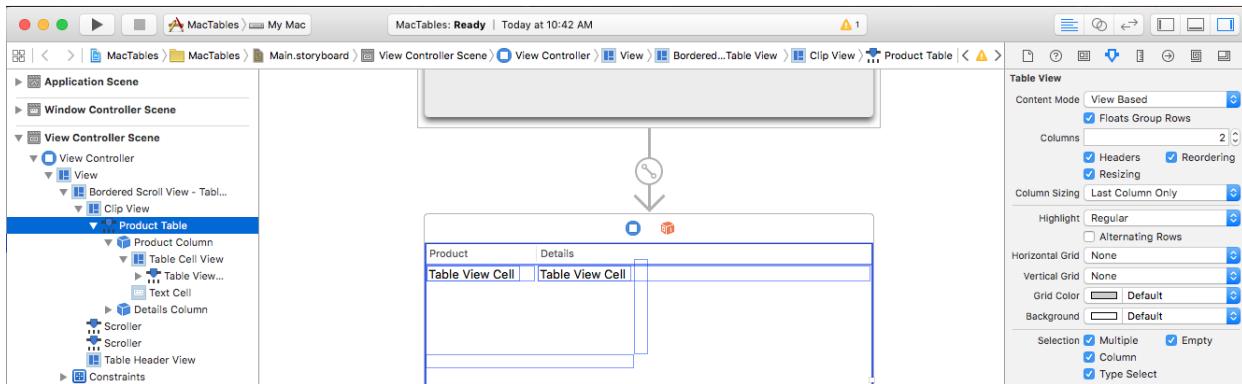
This will allow the user to select any single row in the Table View. Return `false` for the `ShouldSelectRow` for any row that you don't want the user to be able to select or `false` for every row if you don't want the user to be able to select any rows.

The Table View (`NSTableView`) contains the following methods for working with row selection:

- `DeselectRow(nint)` - Deselects the given row in the table.
- `SelectRow(nint, bool)` - Selects the given row. Pass `false` for the second parameter to select only one row at a time.
- `SelectedRow` - Returns the current row selected in the table.
- `IsRowSelected(nint)` - Returns `true` if the given row is selected.

Multiple Row Selection

If you want to allow the user to select a multiple rows, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Select the Table View in the **Interface Hierarchy** and check the **Multiple** checkbox in the **Attribute Inspector**:



Save your changes and return to Visual Studio for Mac to sync with Xcode.

Next, edit the `ProductTableDelegate.cs` file and add the following method:

```
public override bool ShouldSelectRow (NSTableView tableView, nint row)
{
    return true;
}
```

This will allow the user to select any single row in the Table View. Return `false` for the `ShouldSelectRow` for any row that you don't want the user to be able to select or `false` for every row if you don't want the user to be able to select any rows.

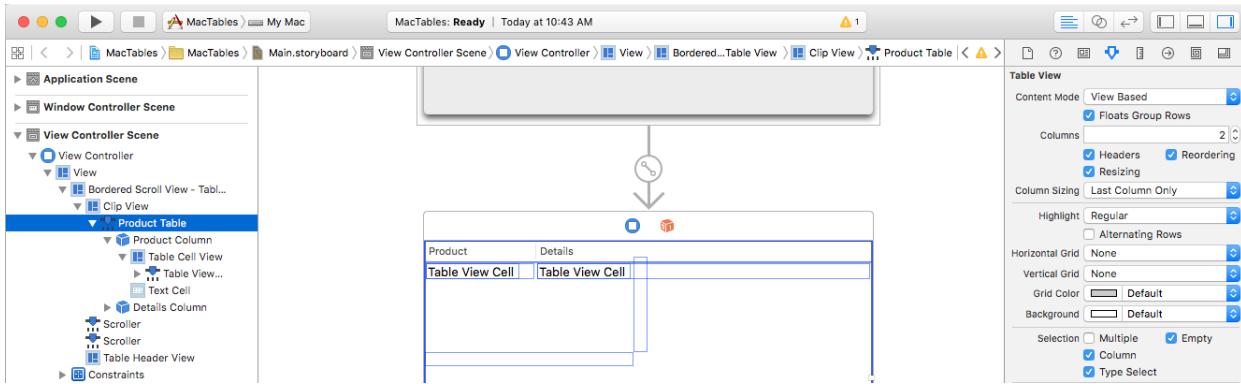
The Table View (`NSTableView`) contains the following methods for working with row selection:

- `DeselectAll(NSObject)` - Deselects all rows in the table. Use `this` for the first parameter to send in the object doing the selecting.
- `DeselectRow(nint)` - Deselects the given row in the table.
- `SelectAll(NSobject)` - Selects all rows in the table. Use `this` for the first parameter to send in the object doing the selecting.
- `SelectRow(nint, bool)` - Selects the given row. Pass `false` for the second parameter clear the selection and select only a single row, pass `true` to extend the selection and include this row.
- `SelectRows(NSIndexSet, bool)` - Selects the given set of rows. Pass `false` for the second parameter clear the selection and select only a these rows, pass `true` to extend the selection and include these rows.
- `SelectedRow` - Returns the current row selected in the table.
- `SelectedRows` - Returns a `NSIndexSet` containing the indexes of the selected rows.
- `SelectedRowCount` - Returns the number of selected rows.

- `IsRowSelected(nint)` - Returns `true` if the given row is selected.

Type to Select Row

If you want to allow the user to type a character with the Table View selected and select the first row that has that character, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Select the Table View in the **Interface Hierarchy** and check the **Type Select** checkbox in the **Attribute Inspector**:



Save your changes and return to Visual Studio for Mac to sync with Xcode.

Now let's edit the `ProductTableDelegate.cs` file and add the following method:

```
public override nint GetNextTypeSelectMatch (NSTableView tableView, nint startRow, nint endRow, string searchString)
{
    nint row = 0;
    foreach(Product product in DataSource.Products) {
        if (product.Title.Contains(searchString)) return row;

        // Increment row counter
        ++row;
    }

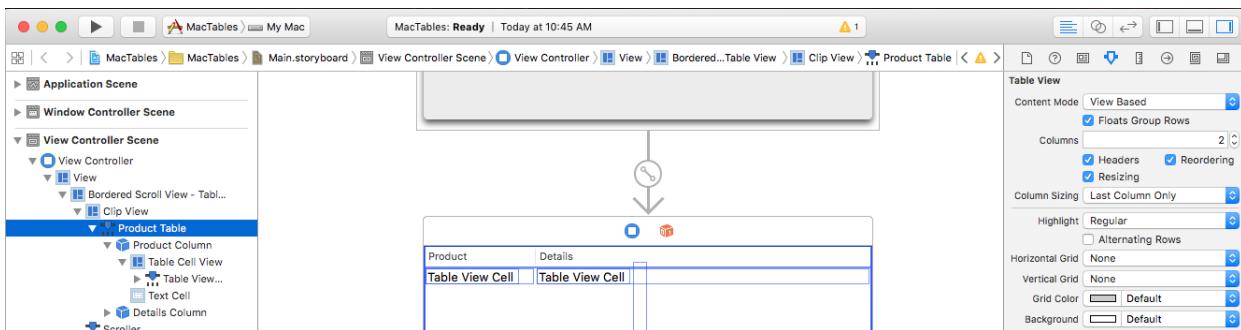
    // If not found select the first row
    return 0;
}
```

The `GetNextTypeSelectMatch` method takes the given `searchString` and returns the row of the first `Product` that has that string in its `Title`.

If we run the application and type a character, a row is selected:

Reordering Columns

If you want to allow the user to drag reorder columns in the Table View, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Select the Table View in the **Interface Hierarchy** and check the **Reordering** checkbox in the **Attribute Inspector**:



If we give a value for the **Autosave** property and check the **Column Information** field, any changes we make to the table's layout will automatically be saved for us and restored the next time the application is run.

Save your changes and return to Visual Studio for Mac to sync with Xcode.

Now let's edit the `ProductTableDelegate.cs` file and add the following method:

```
public override bool ShouldReorder (NSTableView tableView, nint columnIndex, nint newColumnIndex)
{
    return true;
}
```

The `ShouldReorder` method should return `true` for any column that it wants to allow to be drag reordered into the `newColumnIndex`, else return `false`:

If we run the application, we can drag Column Headers around to reorder our columns:

Editing Cells

If you want to allow the user to edit the values for a given cell, edit the `ProductTableDelegate.cs` file and change the `GetViewForItem` method as follows:

```

public override NSView GetViewForItem (NSTableView tableView, NSTableColumn tableColumn, nint row)
{
    // This pattern allows you reuse existing views when they are no-longer in use.
    // If the returned view is null, you instance up a new view
    // If a non-null view is returned, you modify it enough to reflect the new data
    NSTextField view = (NSTextField)tableView.MakeView (tableColumn.Title, this);
    if (view == null) {
        view = new NSTextField ();
        view.Identifier = tableColumn.Title;
        view.BackgroundColor = NSColor.Clear;
        view.Bordered = false;
        view.Selectable = false;
        viewEditable = true;

        view.EditingEnded += (sender, e) => {

            // Take action based on type
            switch(view.Identifier) {
                case "Product":
                    DataSource.Products [(int)view.Tag].Title = view.StringValue;
                    break;
                case "Details":
                    DataSource.Products [(int)view.Tag].Description = view.StringValue;
                    break;
            }
        };
    }

    // Tag view
    view.Tag = row;

    // Setup view based on the column selected
    switch (tableColumn.Title) {
        case "Product":
            view.StringValue = DataSource.Products [(int)row].Title;
            break;
        case "Details":
            view.StringValue = DataSource.Products [(int)row].Description;
            break;
    }

    return view;
}

```

Now if we run the application, the user can edit the cells in the Table View:

Using Images in Table Views

To include an image as part of the cell in a `NSTableView`, you'll need to change how the data is returned by the Table View's `NSTableViewDelegate`'s `GetViewForItem` method to use a `NTableCellView` instead of the typical `NTextField`. For example:

```

public override NSView GetViewForItem (NSTableView tableView, NSTableColumn tableColumn, nint row)
{

    // This pattern allows you reuse existing views when they are no-longer in use.
    // If the returned view is null, you instance up a new view
    // If a non-null view is returned, you modify it enough to reflect the new data
    NSTableCellView view = (NSTableCellView)tableView.MakeView (tableColumn.Title, this);
    if (view == null) {
        view = new NSTableCellView ();
        if (tableColumn.Title == "Product") {
            view.ImageView = new NSImageView (new CGRect (0, 0, 16, 16));
            view.AddSubview (view.ImageView);
            view.TextField = new NSTextField (new CGRect (20, 0, 400, 16));
        } else {
            view.TextField = new NSTextField (new CGRect (0, 0, 400, 16));
        }
        view.TextField.AutoresizingMask = NSViewResizingMask.WidthSizable;
        view.AddSubview (view.TextField);
        view.Identifier = tableColumn.Title;
        view.TextField.BackgroundColor = NSColor.Clear;
        view.TextField.Bordered = false;
        view.TextField.Selectable = false;
        view.TextField.Editable = true;

        view.TextField.EditingEnded += (sender, e) => {

            // Take action based on type
            switch(view.Identifier) {
                case "Product":
                    DataSource.Products [(int)view.TextField.Tag].Title = view.TextField.StringValue;
                    break;
                case "Details":
                    DataSource.Products [(int)view.TextField.Tag].Description = view.TextField.StringValue;
                    break;
            };
        };
    }

    // Tag view
    view.TextField.Tag = row;

    // Setup view based on the column selected
    switch (tableColumn.Title) {
        case "Product":
            view.ImageView.Image = NSImage.ImageNamed ("tags.png");
            view.TextField.StringValue = DataSource.Products [(int)row].Title;
            break;
        case "Details":
            view.TextField.StringValue = DataSource.Products [(int)row].Description;
            break;
    }

    return view;
}

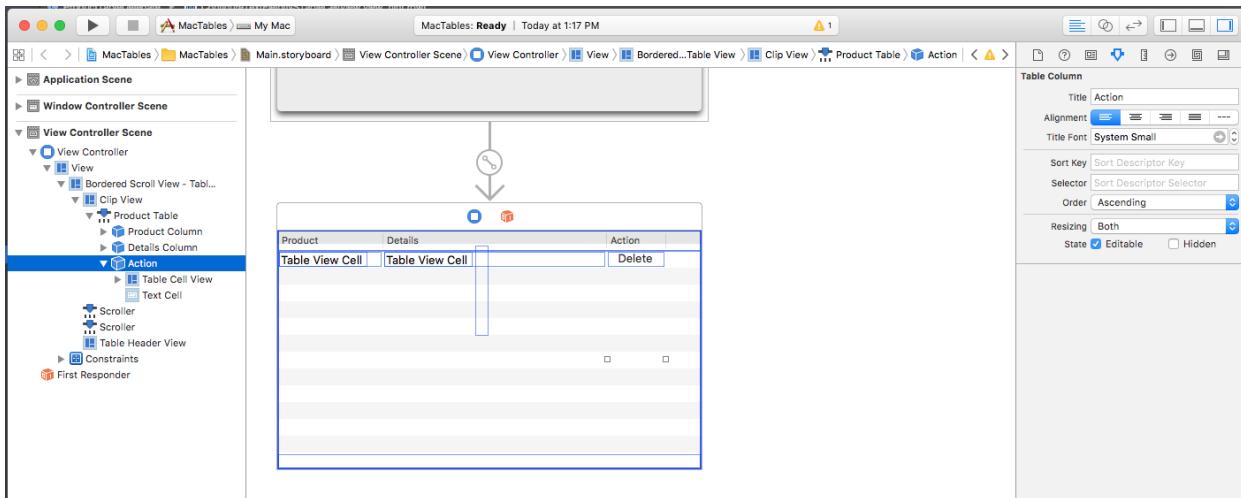
```

For more information, please see the [Using Images with Table Views](#) section of our [Working with Image](#) documentation.

Adding a Delete Button to a Row

Based on the requirements of your app, there might be occasions where you need to supply an action button for each row in the table. As an example of this, let's expand the Table View example created above to include a **Delete** button on each row.

First, edit the `Main.storyboard` in Xcode's Interface Builder, select the Table View and increase the number of columns to three (3). Next, change the Title of the new column to `Action`:



Save the changes to the Storyboard and return to Visual Studio for Mac to sync the changes.

Next, edit the `viewController.cs` file and add the following public method:

```
public void ReloadTable ()  
{  
    ProductTable.ReloadData ();  
}
```

In the same file, modify the creation of the new Table View Delegate inside of `ViewDidLoad` method as follows:

```
// Populate the Product Table  
ProductTable.DataSource = DataSource;  
ProductTable.Delegate = new ProductTableDelegate (this, DataSource);
```

Now, edit the `ProductTableDelegate.cs` file to include a private connection to the View Controller and to take the controller as a parameter when creating a new instance of the delegate:

```
#region Private Variables  
private ProductTableDataSource DataSource;  
private ViewController Controller;  
#endregion  
  
#region Constructors  
public ProductTableDelegate (ViewController controller, ProductTableDataSource datasource)  
{  
    this.Controller = controller;  
    this.DataSource = datasource;  
}  
#endregion
```

Next, add the following new private method to the class:

```

private void ConfigureTextField (NSTableCellView view, nint row)
{
    // Add to view
    view.TextField.AutoresizingMask = NSViewResizingMask.WidthSizable;
    view.AddSubview (view.TextField);

    // Configure
    view.TextField.BackgroundColor = NSColor.Clear;
    view.TextField.Bordered = false;
    view.TextField.Selectable = false;
    view.TextFieldEditable = true;

    // Wireup events
    view.TextField.EditingEnded += (sender, e) => {

        // Take action based on type
        switch (view.Identifier) {
            case "Product":
                DataSource.Products [(int)view.TextField.Tag].Title = view.TextField.StringValue;
                break;
            case "Details":
                DataSource.Products [(int)view.TextField.Tag].Description = view.TextField.StringValue;
                break;
        }
    };

    // Tag view
    view.TextField.Tag = row;
}

```

This takes all of the Text View configurations that were previously being done in the `GetViewForItem` method and places them in a single, callable location (since the last column of the table does not include a Text View but a Button).

Finally, edit the `GetViewForItem` method and make it look like the following:

```

public override NSView GetViewForItem (NSTableView tableView, NSTableColumn TableColumn, nint row)
{

    // This pattern allows you reuse existing views when they are no-longer in use.
    // If the returned view is null, you instance up a new view
    // If a non-null view is returned, you modify it enough to reflect the new data
    NSTableCellView view = (NSTableCellView)tableView.MakeView (TableColumn.Title, this);
    if (view == null) {
        view = new NSTableCellView ();

        // Configure the view
        view.Identifier = TableColumn.Title;

        // Take action based on title
        switch (TableColumn.Title) {
            case "Product":
                view.ImageView = new NSImageView (new CGRect (0, 0, 16, 16));
                view.AddSubview (view.ImageView);
                view.TextField = new NSTextField (new CGRect (20, 0, 400, 16));
                ConfigureTextField (view, row);
                break;
            case "Details":
                view.TextField = new NSTextField (new CGRect (0, 0, 400, 16));
                ConfigureTextField (view, row);
                break;
            case "Action":
                // Create new button
                var button = new NSButton (new CGRect (0, 0, 81, 16));
                button.SetButtonType (NSButtonType.MomentaryPushIn);
        }
    }
}

```

```

button.Title = "Delete";
button.Tag = row;

// Wireup events
button.Activated += (sender, e) => {
    // Get button and product
    var btn = sender as NSButton;
    var product = DataSource.Products [(int)btn.Tag];

    // Configure alert
    var alert = new NSAlert () {
        AlertStyle = NSAlertStyle.Informational,
        InformativeText = $"Are you sure you want to delete {product.Title}? This operation cannot be undone.",
        MessageText = $"Delete {product.Title}?",
    };
    alert.AddButton ("Cancel");
    alert.AddButton ("Delete");
    alert.BeginSheetForResponse (Controller.View.Window, (result) => {
        // Should we delete the requested row?
        if (result == 1001) {
            // Remove the given row from the dataset
            DataSource.Products.RemoveAt((int)btn.Tag);
            Controller.ReloadTable ();
        }
    });
};

// Add to view
view.AddSubview (button);
break;
}

}

// Setup view based on the column selected
switch (tableColumn.Title) {
case "Product":
    view.ImageView.Image = NSImage.ImageNamed ("tag.png");
    view.TextField.StringValue = DataSource.Products [(int)row].Title;
    view.TextField.Tag = row;
    break;
case "Details":
    view.TextField.StringValue = DataSource.Products [(int)row].Description;
    view.TextField.Tag = row;
    break;
case "Action":
    foreach (NSView subview in view.Subviews) {
        var btn = subview as NSButton;
        if (btn != null) {
            btn.Tag = row;
        }
    }
    break;
}

return view;
}

```

Let's look at several sections of this code in more detail. First, if a new `NSTableviewCell` is being created action is taken based on the name of the Column. For the first two columns (**Product** and **Details**), the new `ConfigureTextField` method is called.

For the **Action** column, a new `NSButton` is created and added to the Cell as a Sub View:

```

// Create new button
var button = new NSButton (new CGRect (0, 0, 81, 16));
button.SetButtonType (NSButtonType.MomentaryPushIn);
button.Title = "Delete";
button.Tag = row;
...

// Add to view
view.AddSubview (button);

```

The Button's `Tag` property is used to hold the number of the Row that is currently being processed. This number will be used later when the user requests a row to be deleted in the Button's `Activated` event:

```

// Wireup events
button.Activated += (sender, e) => {
    // Get button and product
    var btn = sender as NSButton;
    var product = DataSource.Products [(int)btn.Tag];

    // Configure alert
    var alert = new NSAlert () {
        AlertStyle = NSAlertStyle.Informational,
        InformativeText = $"Are you sure you want to delete {product.Title}? This operation cannot be undone.",
        MessageText = $"Delete {product.Title}",
    };
    alert.AddButton ("Cancel");
    alert.AddButton ("Delete");
    alert.BeginSheetForResponse (Controller.View.Window, (result) => {
        // Should we delete the requested row?
        if (result == 1001) {
            // Remove the given row from the dataset
            DataSource.Products.RemoveAt((int)btn.Tag);
            Controller.ReloadTable ();
        }
    });
};

```

At the start of the event handler, we get the button and the product that is on the given table row. Then an Alert is presented to the user confirming the row deletion. If the user chooses to delete the row, the given row is removed from the Data Source and the table is reloaded:

```

// Remove the given row from the dataset
DataSource.Products.RemoveAt((int)btn.Tag);
Controller.ReloadTable ();

```

Finally, if the Table View Cell is being reused instead of being created new, the following code configures it based on the Column being processed:

```

// Setup view based on the column selected
switch (tableColumn.Title) {
case "Product":
    view.ImageView.Image = NSImage.ImageNamed ("tag.png");
    view.TextField.StringValue = DataSource.Products [(int)row].Title;
    view.TextField.Tag = row;
    break;
case "Details":
    view.TextField.StringValue = DataSource.Products [(int)row].Description;
    view.TextField.Tag = row;
    break;
case "Action":
    foreach (NSView subview in view.Subviews) {
        var btn = subview as NSButton;
        if (btn != null) {
            btn.Tag = row;
        }
    }
    break;
}

```

For the **Action** column, all Sub Views are scanned until the `NSButton` is found, then it's `Tag` property is updated to point at the current Row.

With these changes in place, when the app is run each row will have a **Delete** button:

Product	Details	Action
HockeyApp	Bring Mobile DevOps to your apps and reliability to your users.	Delete
Visual Studio Community	A free, full-featured and extensible IDE for Windows users to create Android and iOS apps with Xamarin, as	Delete
Visual Studio Enterprise	End-to-end solution for teams of any size with demanding quality and scale needs	Delete
Visual Studio Professional	Professional developer tools and services for individual developers or small teams.	Delete
Xamarin Studio Community	A free, full-featured IDE for Mac users to create Android and iOS apps using Xamarin.	Delete
Xamarin Test Cloud	Automatically test your app on thousands of mobile devices.	Delete
Xamarin University	Take your mobile strategy and apps to the next level.	Delete
Xamarin.Android	Allows you to develop native Android Applications in C#	Delete
Xamarin.iOS	Allows you to develop native iOS Applications in C#	Delete
Xamarin.Mac	Allows you to develop Mac native Applications in C#	Delete
Xamarin.tvOS	Allows you to develop Apple TV native Applications in C#	Delete

When the user clicks a **Delete** button, an alert will be displayed asking them to delete the given Row:

Product	Details	Action
HockeyApp	Bring Mobile DevOps to your apps and reliability to your users.	Delete
Visual Studio Community	A free, full-featured and extensible IDE for Windows users to create Android and iOS apps with Xamarin, as	Delete
Visual Studio Enterprise	End-to-end solution for teams of any size with demanding quality and scale needs	Delete
Visual Studio Professional	Professional developer tools and services for individual developers or small teams.	Delete
Xamarin Studio Community	A free, full-featured IDE for Mac users to create Android and iOS apps using Xamarin.	Delete
Xamarin Test Cloud	Automatically test your app on thousands of mobile devices.	Delete
Xamarin University	Take your mobile strategy and apps to the next level.	Delete
Xamarin.Android	Allows you to develop native Android Applications in C#	Delete
Xamarin.iOS	Allows you to develop native iOS Applications in C#	Delete
Xamarin.Mac	Allows you to develop Mac native Applications in C#	Delete
Xamarin.tvOS	Allows you to develop Apple TV native Applications in C#	Delete

Delete HockeyApp?

Are you sure you want to delete HockeyApp? This operation cannot be undone.

Delete **Cancel**

If the user chooses delete, the row will be removed and the table will be redrawn:

Product	Details	Action
Visual Studio Community	A free, full-featured and extensible IDE for Windows users to create Android and iOS apps with Xamarin, as	Delete
Visual Studio Enterprise	End-to-end solution for teams of any size with demanding quality and scale needs	Delete
Visual Studio Professional	Professional developer tools and services for individual developers or small teams.	Delete
Xamarin Studio Community	A free, full-featured IDE for Mac users to create Android and iOS apps using Xamarin.	Delete
Xamarin Test Cloud	Automatically test your app on thousands of mobile devices.	Delete
Xamarin University	Take your mobile strategy and apps to the next level.	Delete
Xamarin.Android	Allows you to develop native Android Applications in C#	Delete
Xamarin.iOS	Allows you to develop native iOS Applications in C#	Delete
Xamarin.Mac	Allows you to develop Mac native Applications in C#	Delete
Xamarin.tvOS	Allows you to develop Apple TV native Applications in C#	Delete

Data Binding Table Views

By using Key-Value Coding and Data Binding techniques in your Xamarin.Mac application, you can greatly decrease the amount of code that you have to write and maintain to populate and work with UI elements. You also have the benefit of further decoupling your backing data (*Data Model*) from your front end User Interface (*Model-View-Controller*), leading to easier to maintain, more flexible application design.

Key-Value Coding (KVC) is a mechanism for accessing an object's properties indirectly, using keys (specially formatted strings) to identify properties instead of accessing them through instance variables or accessor methods (`get/set`). By implementing Key-Value Coding compliant accessors in your Xamarin.Mac application, you gain access to other macOS features such as Key-Value Observing (KVO), Data Binding, Core Data, Cocoa bindings, and scriptability.

For more information, please see the [Table View Data Binding](#) section of our [Data Binding and Key-Value Coding](#) documentation.

Summary

This article has taken a detailed look at working with Table Views in a Xamarin.Mac application. We saw the different types and uses of Table Views, how to create and maintain Table Views in Xcode's Interface Builder and how to work with Table Views in C# code.

Related Links

- [MacTables \(sample\)](#)
- [MacImages \(sample\)](#)
- [Hello, Mac](#)
- [Outline Views](#)
- [Source Lists](#)
- [Data Binding and Key-Value Coding](#)
- [OS X Human Interface Guidelines](#)
- [NSTableView](#)
- [NSTableViewDelegate](#)
- [NSTableViewDataSource](#)

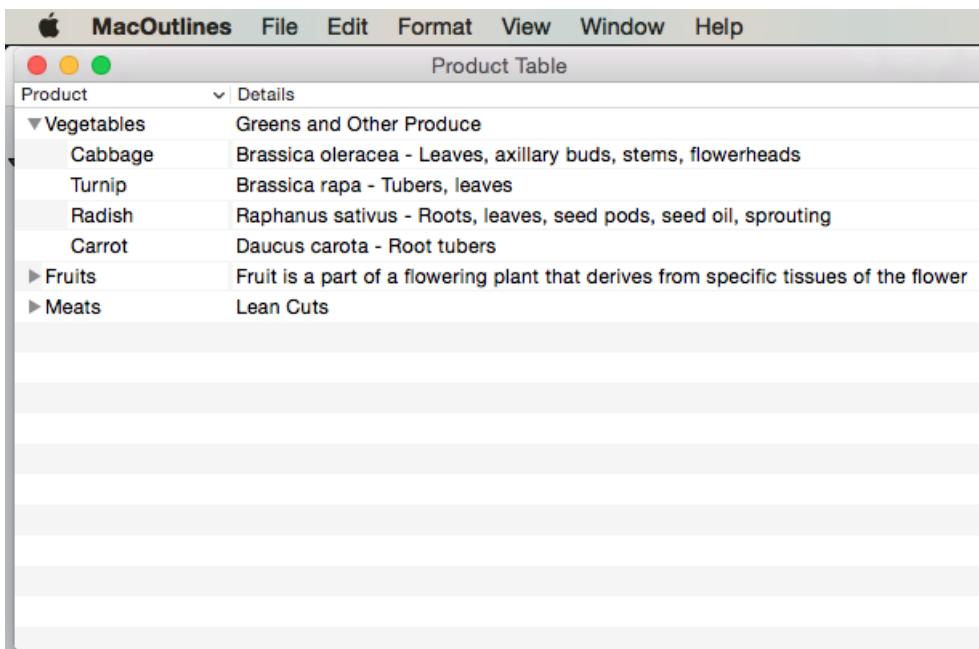
Outline views in Xamarin.Mac

3/5/2021 • 20 minutes to read • [Edit Online](#)

This article covers working with outline views in a Xamarin.Mac application. It describes creating and maintaining outline views in Xcode and Interface Builder and working with them programmatically.

When working with C# and .NET in a Xamarin.Mac application, you have access to the same Outline Views that a developer working in *Objective-C* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, you can use Xcode's *Interface Builder* to create and maintain your Outline Views (or optionally create them directly in C# code).

An Outline View is a type of Table that allows the user expand or collapse rows of hierarchical data. Like a Table View, an Outline View displays data for a set of related items, with rows representing individual items and columns representing the attributes of those items. Unlike a Table View, items in an Outline View are not in a flat list, they are organized in a hierarchy, like files and folders on a hard drive.



In this article, we'll cover the basics of working with Outline Views in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` commands used to wire-up your C# classes to Objective-C objects and UI Elements.

Introduction to Outline Views

An Outline View is a type of Table that allows the user expand or collapse rows of hierarchical data. Like a Table View, an Outline View displays data for a set of related items, with rows representing individual items and columns representing the attributes of those items. Unlike a Table View, items in an Outline View are not in a flat list, they are organized in a hierarchy, like files and folders on a hard drive.

If an item in an Outline View contains other items, it can be expanded or collapsed by the user. An expandable

item displays a disclosure triangle, which points to the right when the item is collapsed and points down when the item is expanded. Clicking on the disclosure triangle causes the item to expand or collapse.

The Outline View (`NSOutlineView`) is a subclass of the Table View (`NSTableView`) and as such, inherits much of its behavior from its parent class. As a result, many operations supported by a Table View, such as selecting rows or columns, repositioning columns by dragging Column Headers, etc., are also supported by an Outline View. A Xamarin.Mac application has control of these features, and can configure the Outline View's parameters (either in code or Interface Builder) to allow or disallow certain operations.

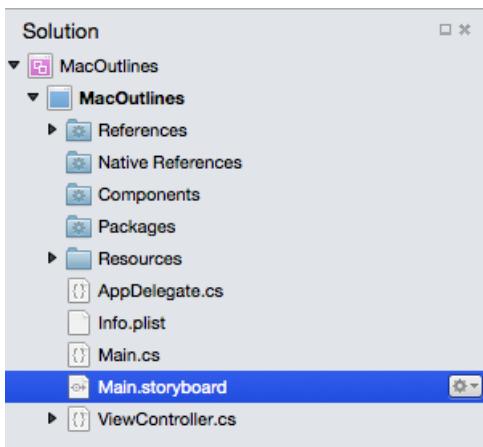
An Outline View does not store its own data, instead it relies on a Data Source (`NSOutlineViewDataSource`) to provide both the rows and columns required, on a as-needed basis.

An Outline View's behavior can be customized by providing a subclass of the Outline View Delegate (`NSOutlineViewDelegate`) to support Outline column management, type to select functionality, row selection and editing, custom tracking, and custom views for individual columns and rows.

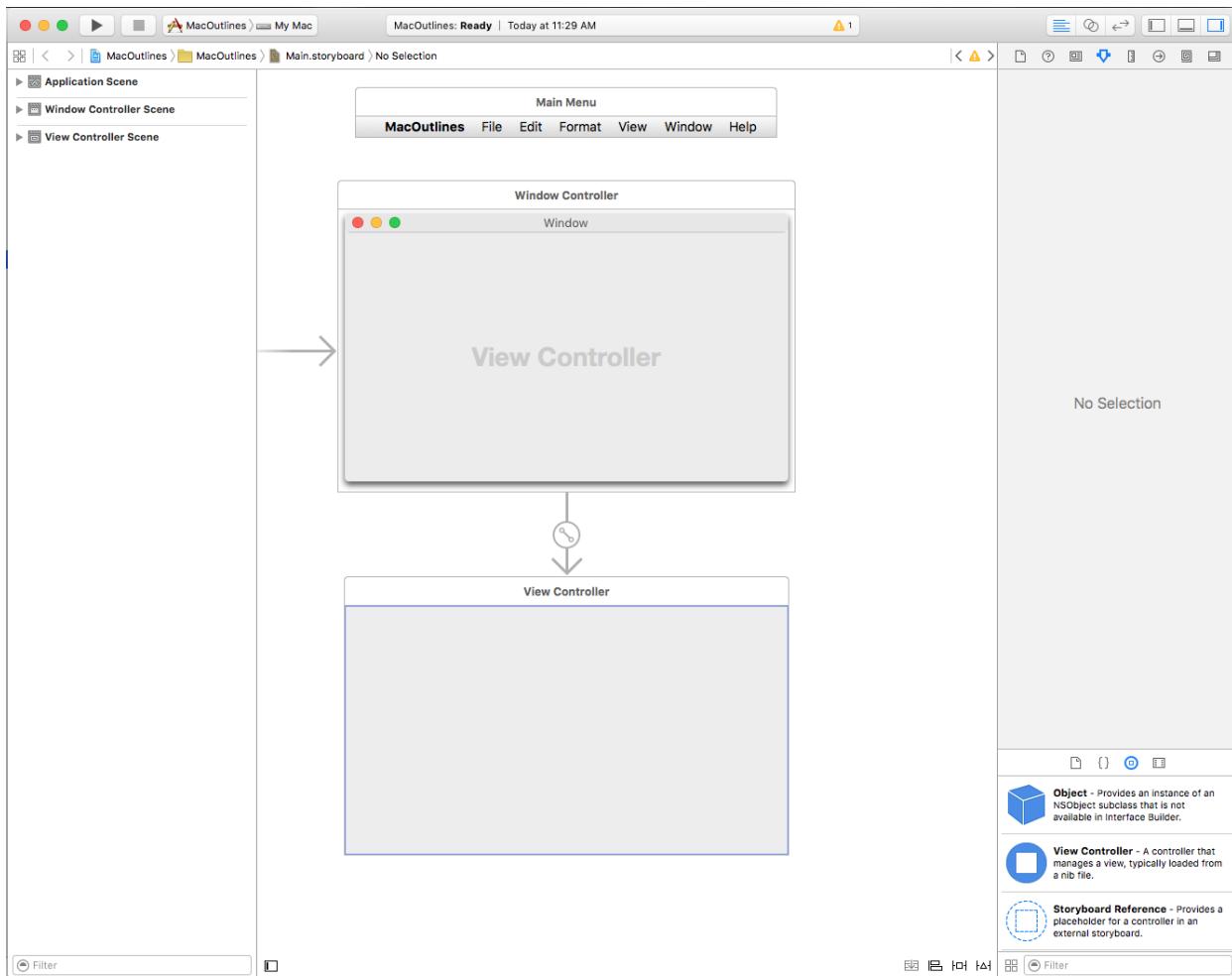
Since an Outline View shares much of its behavior and functionality with a Table View, you might want to go through our [Table Views](#) documentation before continuing with this article.

Creating and Maintaining Outline Views in Xcode

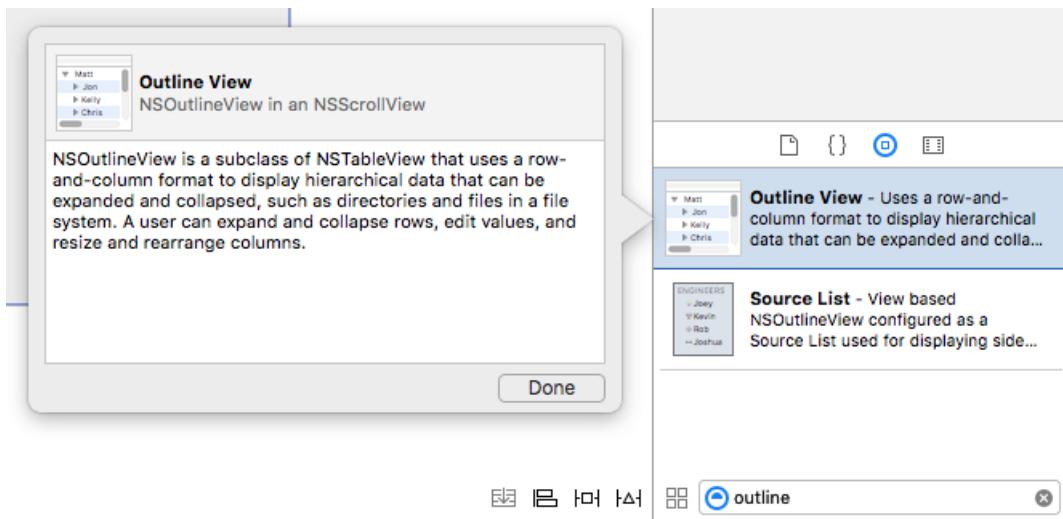
When you create a new Xamarin.Mac Cocoa application, you get a standard blank, window by default. This windows is defined in a `.Storyboard` file automatically included in the project. To edit your windows design, in the **Solution Explorer**, double click the `Main.storyboard` file:



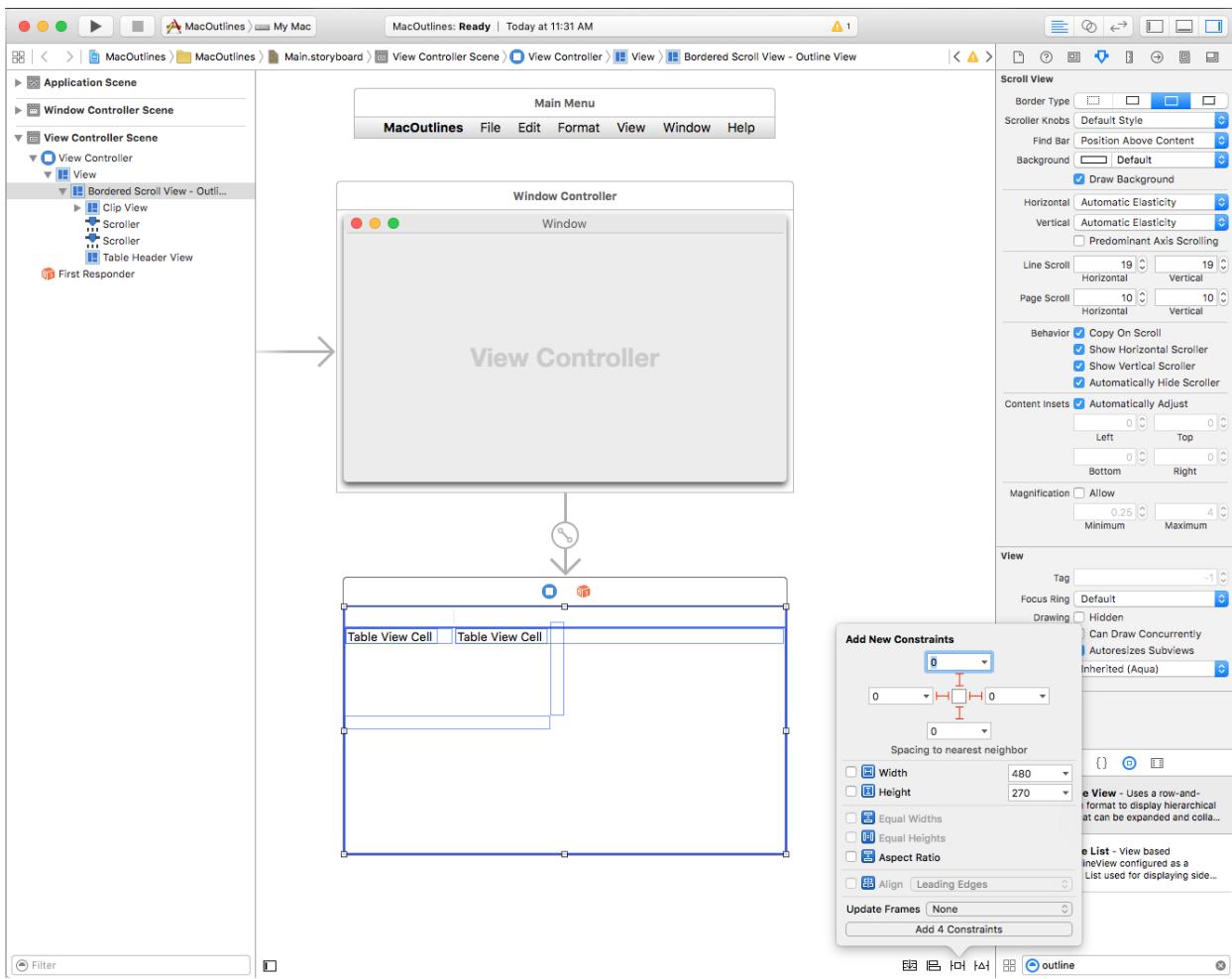
This will open the window design in Xcode's Interface Builder:



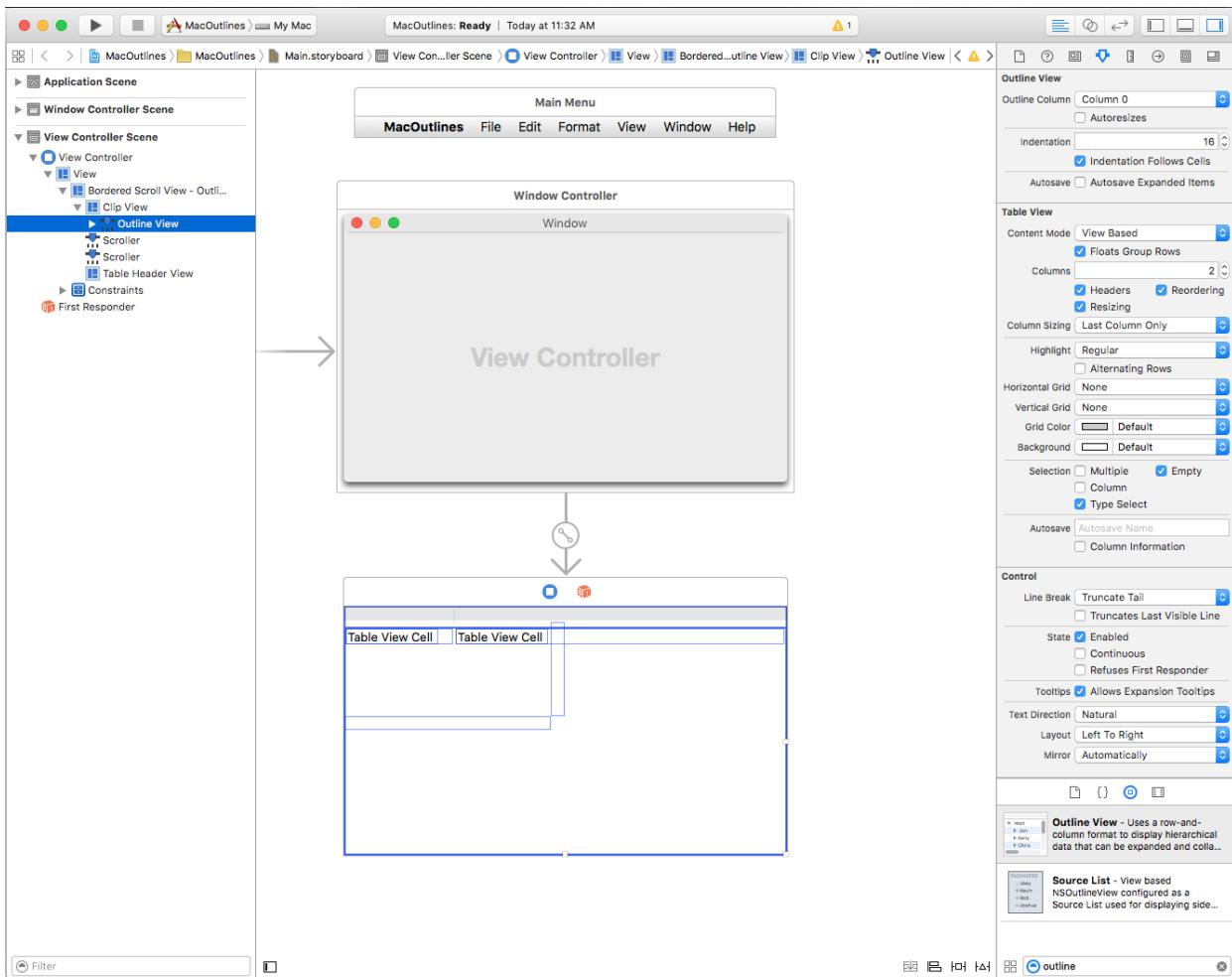
Type `outline` into the Library Inspector's Search Box to make it easier to find the Outline View controls:



Drag a Outline View onto the View Controller in the Interface Editor, make it fill the content area of the View Controller and set it to where it shrinks and grows with the window in the Constraint Editor:



Select the Outline View in the **Interface Hierarchy** and the following properties are available in the **Attribute Inspector**:



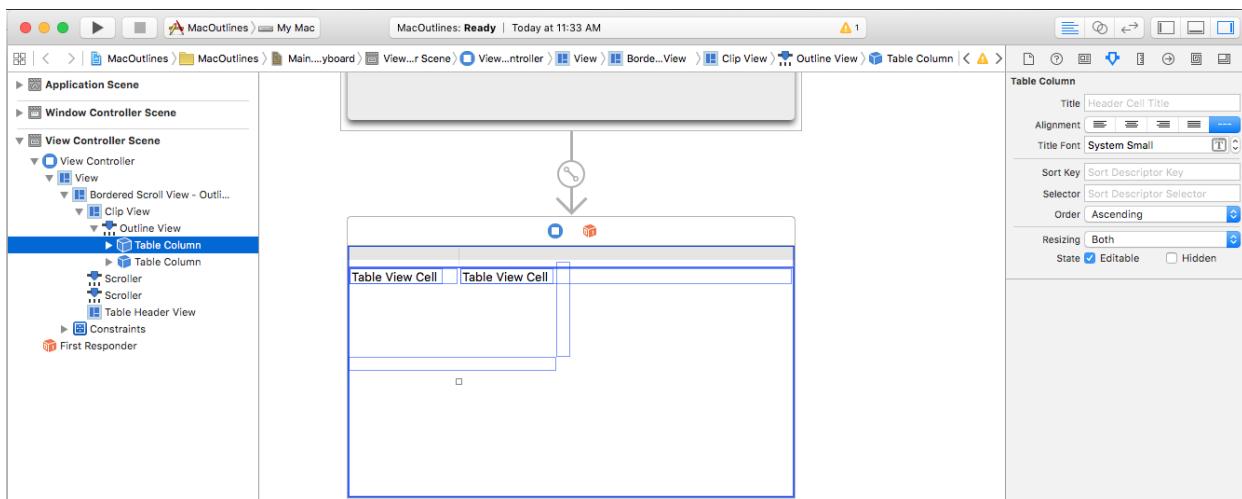
- **Outline Column** - The Table Column in which the Hierarchical data is displayed.
- **Autosave Outline Column** - If `true`, the Outline Column will be automatically saved and restored between application runs.
- **Indentation** - The amount to indent columns under an expanded item.
- **Indentation Follows Cells** - If `true`, the Indentation Mark will be indented along with the cells.
- **Autosave Expanded Items** - If `true`, the expanded/collapsed state of the items will be automatically saved and restored between application runs.
- **Content Mode** - Allows you to use either Views (`NSView`) or Cells (`NSCell`) to display the data in the rows and columns. Starting with macOS 10.7, you should use Views.
- **Floating Group Rows** - If `true`, the Table View will draw grouped cells as if they are floating.
- **Columns** - Defines the number of columns displayed.
- **Headers** - If `true`, the columns will have Headers.
- **Reordering** - If `true`, the user will be able to drag reorder the columns in the table.
- **Resizing** - If `true`, the user will be able to drag column Headers to resize columns.
- **Column Sizing** - Controls how the table will auto size columns.
- **Highlight** - Controls the type of highlighting the table uses when a cell is selected.
- **Alternate Rows** - If `true`, every other row will have a different background color.
- **Horizontal Grid** - Selects the type of border drawn between cells horizontally.
- **Vertical Grid** - Selects the type of border drawn between cells vertically.
- **Grid Color** - Sets the cell border color.
- **Background** - Sets the cell background color.
- **Selection** - Allow you to control how the user can select cells in the table as:
 - **Multiple** - If `true`, the user can select multiple rows and columns.
 - **Column** - If `true`, the user can select columns.

- **Type Select** - If `true`, the user can type a character to select a row.
- **Empty** - If `true`, the user is not required to select a row or column, the table allows for no selection at all.
- **Autosave** - The name that the tables format is automatically save under.
- **Column Information** - If `true`, the order and width of the columns will be automatically saved.
- **Line Breaks** - Select how the cell handles line breaks.
- **Truncates Last Visible Line** - If `true`, the cell will be truncated in the data can not fit inside its bounds.

IMPORTANT

Unless you are maintaining a legacy Xamarin.Mac application, `NSView` based Outline Views should be used over `NSCell` based Table Views. `NSCell` is considered legacy and may not be supported going forward.

Select a Table Column in the **Interface Hierarchy** and the following properties are available in the **Attribute Inspector**:

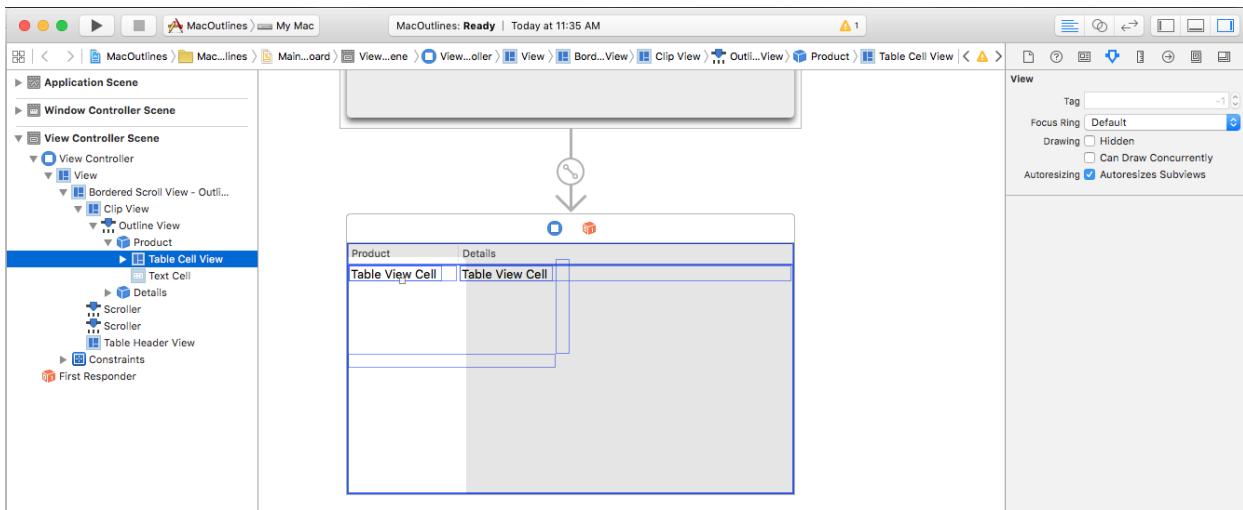


- **Title** - Sets the title of the column.
- **Alignment** - Set the alignment of the text within the cells.
- **Title Font** - Selects the font for the cell's Header text.
- **Sort Key** - Is the key used to sort data in the column. Leave blank if the user cannot sort this column.
- **Selector** - Is the Action used to perform the sort. Leave blank if the user cannot sort this column.
- **Order** - Is the sort order for the columns data.
- **Resizing** - Selects the type of resizing for the column.
- **Editable** - If `true`, the user can edit cells in a cell based table.
- **Hidden** - If `true`, the column is hidden.

You can also resize the column by dragging its handle (vertically centered on the column's right side) left or right.

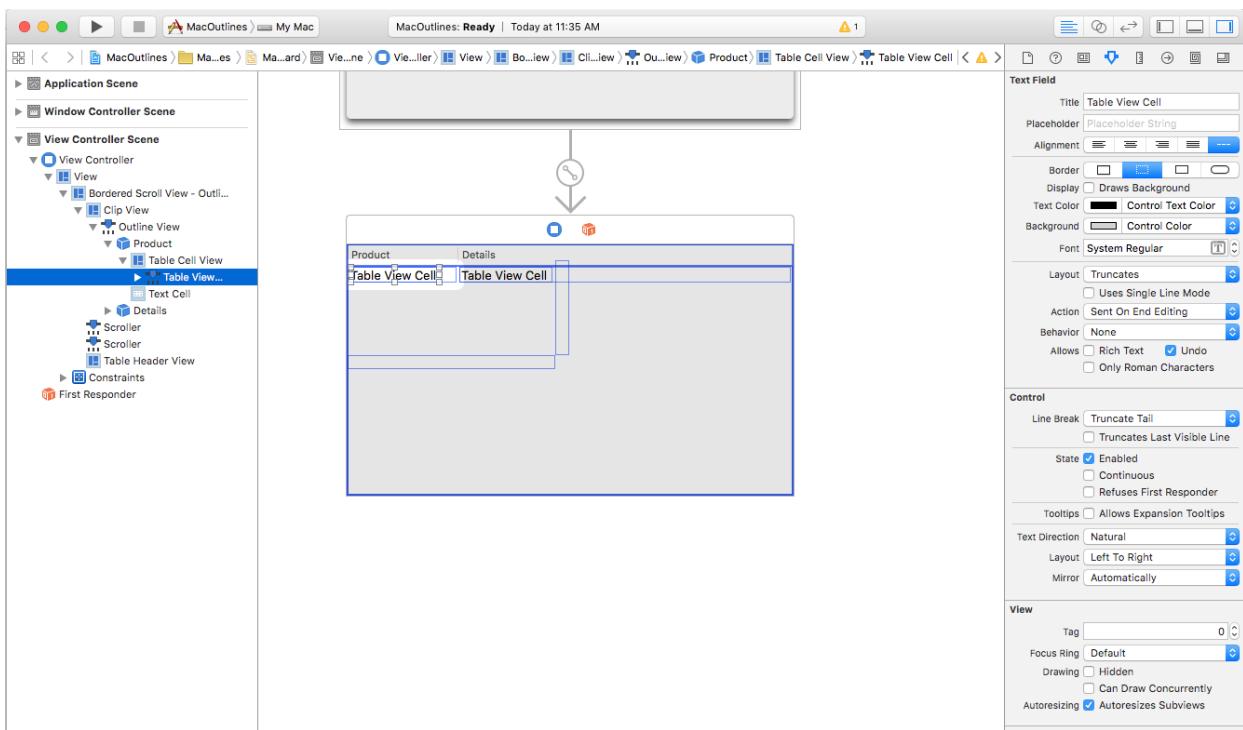
Let's select the each Column in our Table View and give the first column a **Title** of `Product` and the second one `Details`.

Select a Table Cell View (`NSTableViewCell`) in the **Interface Hierarchy** and the following properties are available in the **Attribute Inspector**:



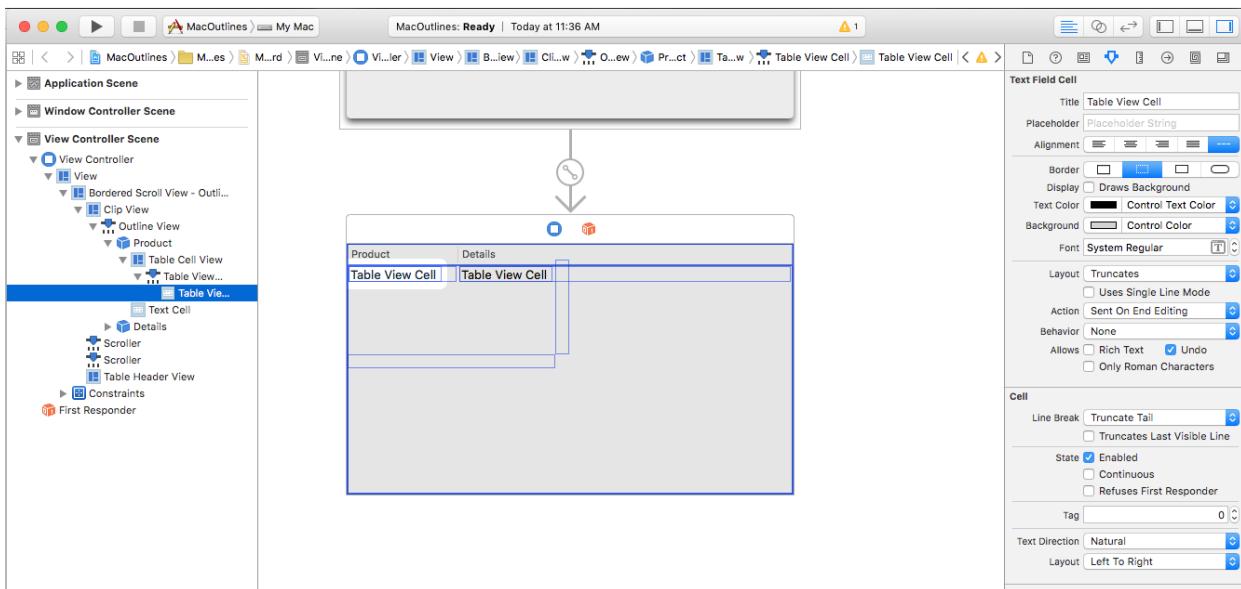
These are all of the properties of a standard View. You also have the option of resizing the rows for this column here.

Select a Table View Cell (by default, this is a `NSTextField`) in the **Interface Hierarchy** and the following properties are available in the **Attribute Inspector**:



You'll have all the properties of a standard Text Field to set here. By default, a standard Text Field is used to display data for a cell in a column.

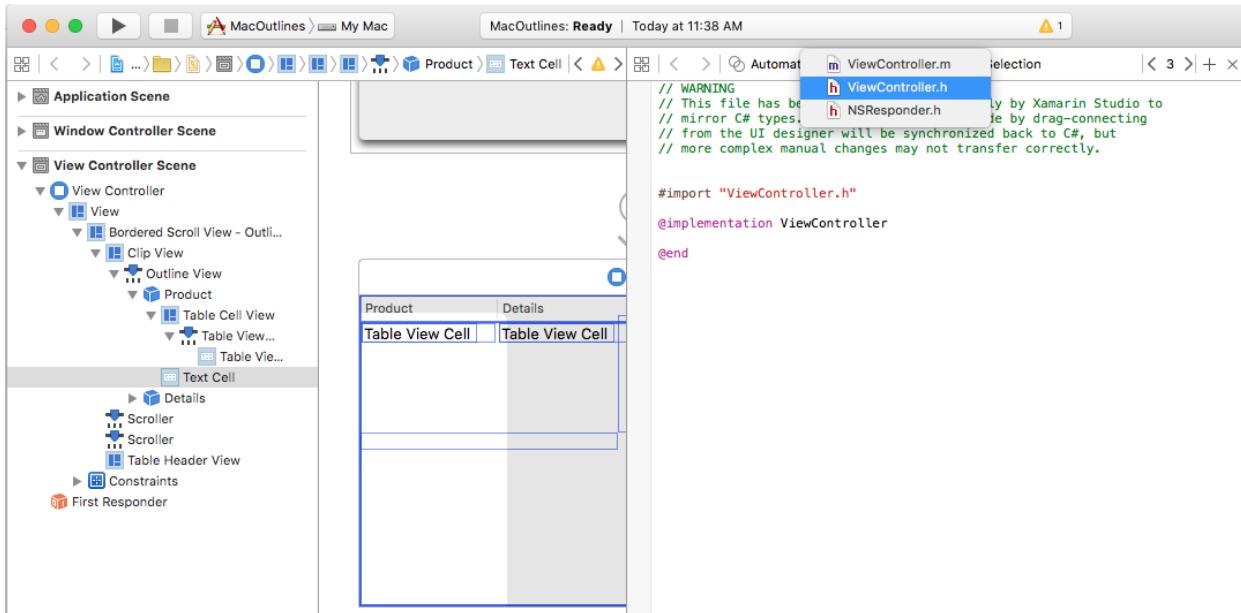
Select a Table Cell View (`NSTextFieldCell`) in the **Interface Hierarchy** and the following properties are available in the **Attribute Inspector**:



The most important settings here are:

- **Layout** - Select how cells in this column are laid out.
- **Uses Single Line Mode** - If `true`, the cell is limited to a single line.
- **First Runtime Layout Width** - If `true`, the cell will prefer the width set for it (either manually or automatically) when it is displayed the first time the application is run.
- **Action** - Controls when the **Edit Action** is sent for the cell.
- **Behavior** - Defines if a cell is selectable or editable.
- **Rich Text** - If `true`, the cell can display formatted and styled text.
- **Undo** - If `true`, the cell assumes responsibility for its undo behavior.

Select the Table Cell View (`NSTextFieldCell`) at the bottom of a Table Column in the **Interface Hierarchy**:



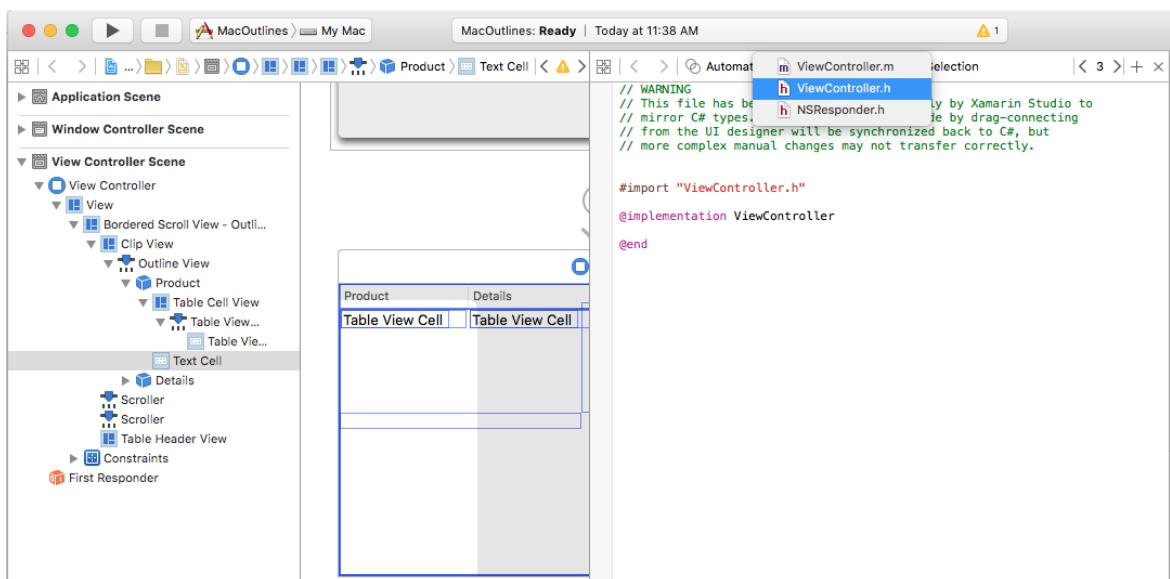
This allows you to edit the Table Cell View used as the base *Pattern* for all cells created for the given column.

Adding Actions and Outlets

Just like any other Cocoa UI control, we need to expose our Outline View and its columns and cells to C# code using **Actions** and **Outlets** (based on the functionality required).

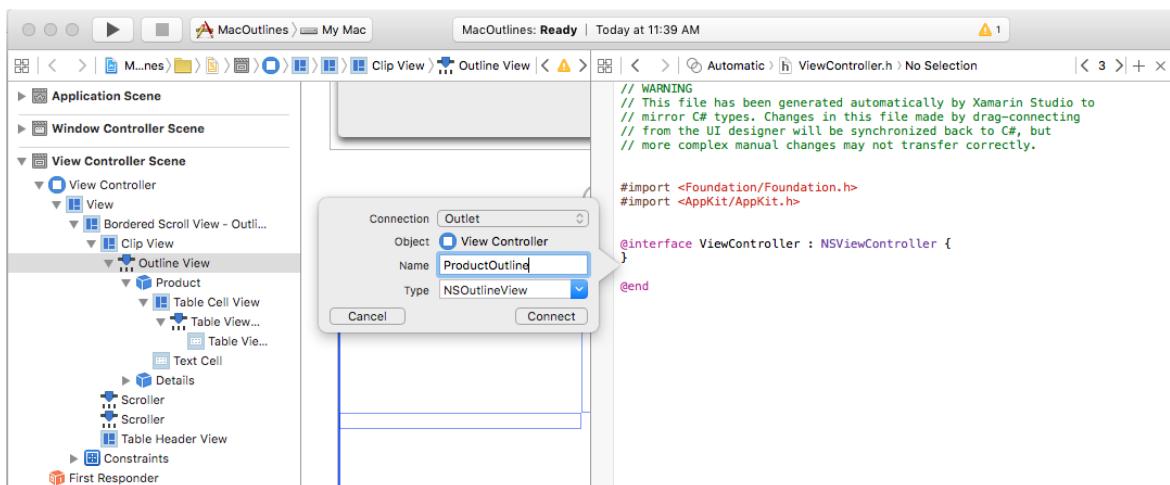
The process is the same for any Outline View element that we want to expose:

1. Switch to the Assistant Editor and ensure that the `ViewController.h` file is selected:

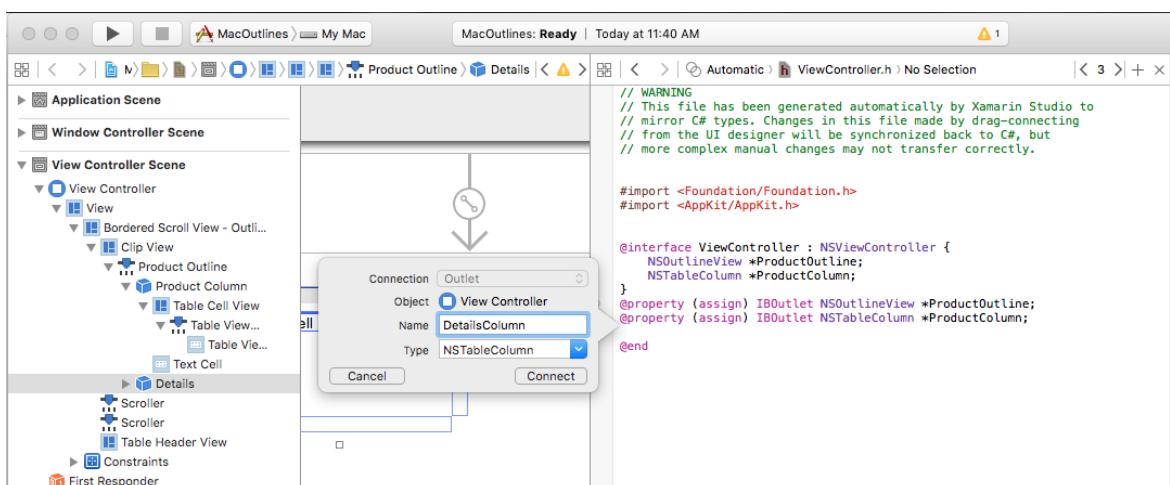


2. Select the Outline View from the **Interface Hierarchy**, control-click and drag to the `ViewController.h` file.

3. Create an **Outlet** for the Outline View called `ProductOutline`:



4. Create **Outlets** for the tables columns as well called `ProductColumn` and `DetailsColumn`:



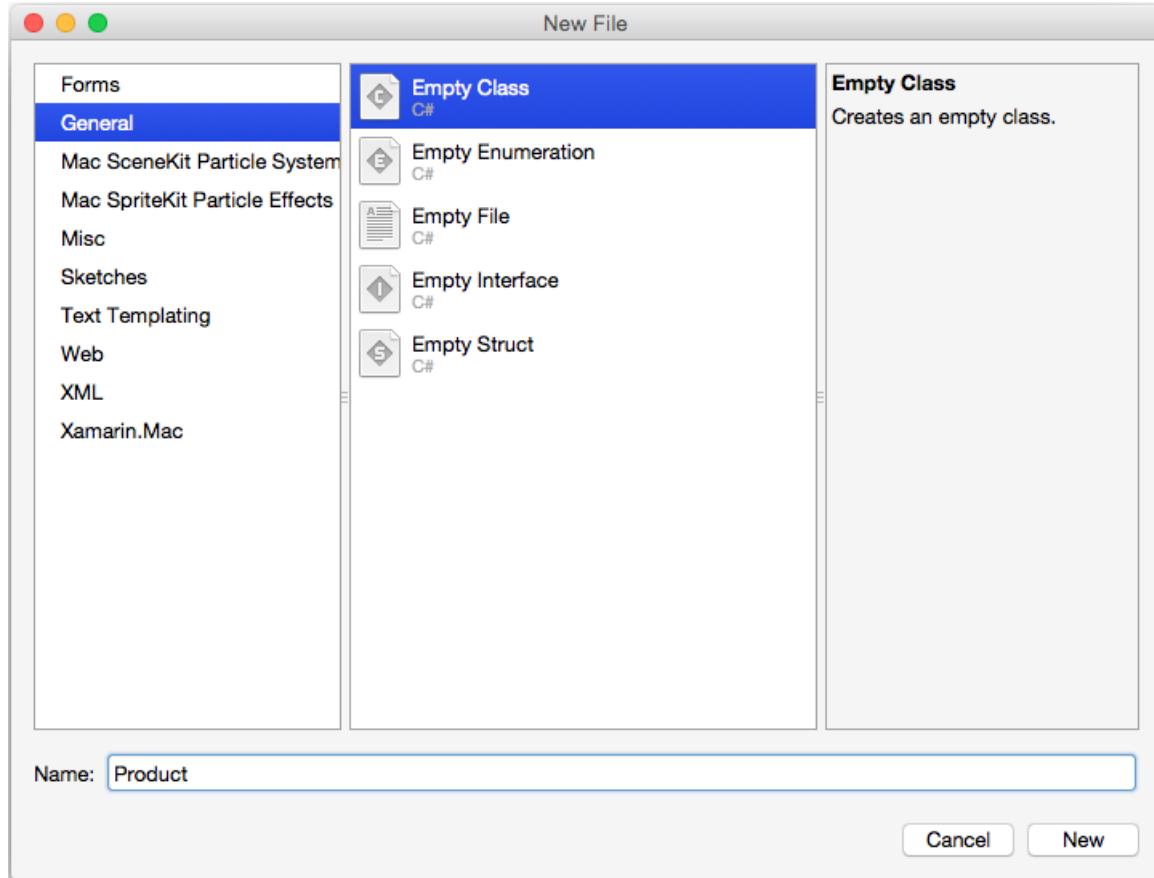
5. Save your changes and return to Visual Studio for Mac to sync with Xcode.

Next, we'll write the code to display some data for the outline when the application is run.

Populating the Outline View

With our Outline View designed in Interface Builder and exposed via an **Outlet**, next we need to create the C# code to populate it.

First, let's create a new `Product` class to hold the information for the individual rows and groups of sub products. In the **Solution Explorer**, right-click the Project and select **Add > New File...** Select **General > Empty Class**, enter `Product` for the **Name** and click the **New** button:



Make the `Product.cs` file look like the following:

```

using System;
using Foundation;
using System.Collections.Generic;

namespace MacOutlines
{
    public class Product : NSObject
    {
        #region Public Variables
        public List<Product> Products = new List<Product>();
        #endregion

        #region Computed Properties
        public string Title { get; set; } = "";
        public string Description { get; set; } = "";
        public bool IsProductGroup {
            get { return (Products.Count > 0); }
        }
        #endregion

        #region Constructors
        public Product ()
        {
        }

        public Product (string title, string description)
        {
            this.Title = title;
            this.Description = description;
        }
        #endregion
    }
}

```

Next, we need to create a subclass of `NSOutlineDataSource` to provide the data for our outline as it is requested. In the **Solution Explorer**, right-click the Project and select **Add > New File...**. Select **General > Empty Class**, enter `ProductOutlineDataSource` for the **Name** and click the **New** button.

Edit the `ProductTableDataSource.cs` file and make it look like the following:

```

using System;
using AppKit;
using CoreGraphics;
using Foundation;
using System.Collections;
using System.Collections.Generic;

namespace MacOutlines
{
    public class ProductOutlineDataSource : NSOutlineViewDataSource
    {
        #region Public Variables
        public List<Product> Products = new List<Product>();
        #endregion

        #region Constructors
        public ProductOutlineDataSource ()
        {
        }
        #endregion

        #region Override Methods
        public override nint GetChildrenCount (NSOutlineView outlineView, NSObject item)
        {
            if (item == null) {
                return Products.Count;
            } else {
                return ((Product)item).Products.Count;
            }
        }

        public override NSObject GetChild (NSOutlineView outlineView, nint childIndex, NSObject item)
        {
            if (item == null) {
                return Products [childIndex];
            } else {
                return ((Product)item).Products [childIndex];
            }
        }

        public override bool ItemExpandable (NSOutlineView outlineView, NSObject item)
        {
            if (item == null) {
                return Products [0].IsProductGroup;
            } else {
                return ((Product)item).IsProductGroup;
            }
        }
        #endregion
    }
}

```

This class has storage for our Outline View's items and overrides the `GetChildrenCount` to return the number of rows in the table. The `GetChild` returns a specific parent or child item (as requested by the Outline View) and the `ItemExpandable` defines the specified item as either a parent or a child.

Finally, we need to create a subclass of `NSOutlineDelegate` to provide the behavior for our outline. In the **Solution Explorer**, right-click the Project and select **Add > New File...** Select **General > Empty Class**, enter `ProductOutlineDelegate` for the **Name** and click the **New** button.

Edit the `ProductOutlineDelegate.cs` file and make it look like the following:

```

using System;
using AppKit;
using CoreGraphics;
using Foundation;
using System.Collections;
using System.Collections.Generic;

namespace MacOutlines
{
    public class ProductOutlineDelegate : NSOutlineViewDelegate
    {
        #region Constants
        private const string CellIdentifier = "ProdCell";
        #endregion

        #region Private Variables
        private ProductOutlineDataSource DataSource;
        #endregion

        #region Constructors
        public ProductOutlineDelegate (ProductOutlineDataSource datasource)
        {
            this.DataSource = datasource;
        }
        #endregion

        #region Override Methods

        public override NSView GetView (NSOutlineView outlineView, NSTableColumn TableColumn, NSObject item)
        {
            // This pattern allows you reuse existing views when they are no-longer in use.
            // If the returned view is null, you instance up a new view
            // If a non-null view is returned, you modify it enough to reflect the new data
            NSTextField view = (NSTextField)outlineView.MakeView (CellIdentifier, this);
            if (view == null) {
                view = new NSTextField ();
                view.Identifier = CellIdentifier;
                view.BackgroundColor = NSColor.Clear;
                view.Bordered = false;
                view.Selectable = false;
                viewEditable = false;
            }

            // Cast item
            var product = item as Product;

            // Setup view based on the column selected
            switch (TableColumn.Title) {
                case "Product":
                    view.StringValue = product.Title;
                    break;
                case "Details":
                    view.StringValue = product.Description;
                    break;
            }

            return view;
        }
        #endregion
    }
}

```

When we create an instance of the `ProductOutlineDelegate`, we also pass in an instance of the `ProductOutlineDataSource` that provides the data for the outline. The `GetView` method is responsible for returning a view (data) to display the cell for a give column and row. If possible, an existing view will be reused.

to display the cell, if not a new view must be created.

To populate the outline, let's edit the `MainWindow.cs` file and make the `AwakeFromNib` method look like the following:

```
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

    // Create data source and populate
    var DataSource = new ProductOutlineDataSource ();

    var Vegetables = new Product ("Vegetables", "Greens and Other Produce");
    Vegetables.Products.Add (new Product ("Cabbage", "Brassica oleracea - Leaves, axillary buds, stems, flowerheads"));
    Vegetables.Products.Add (new Product ("Turnip", "Brassica rapa - Tubers, leaves"));
    Vegetables.Products.Add (new Product ("Radish", "Raphanus sativus - Roots, leaves, seed pods, seed oil, sprouting"));
    Vegetables.Products.Add (new Product ("Carrot", "Daucus carota - Root tubers"));
    DataSource.Products.Add (Vegetables);

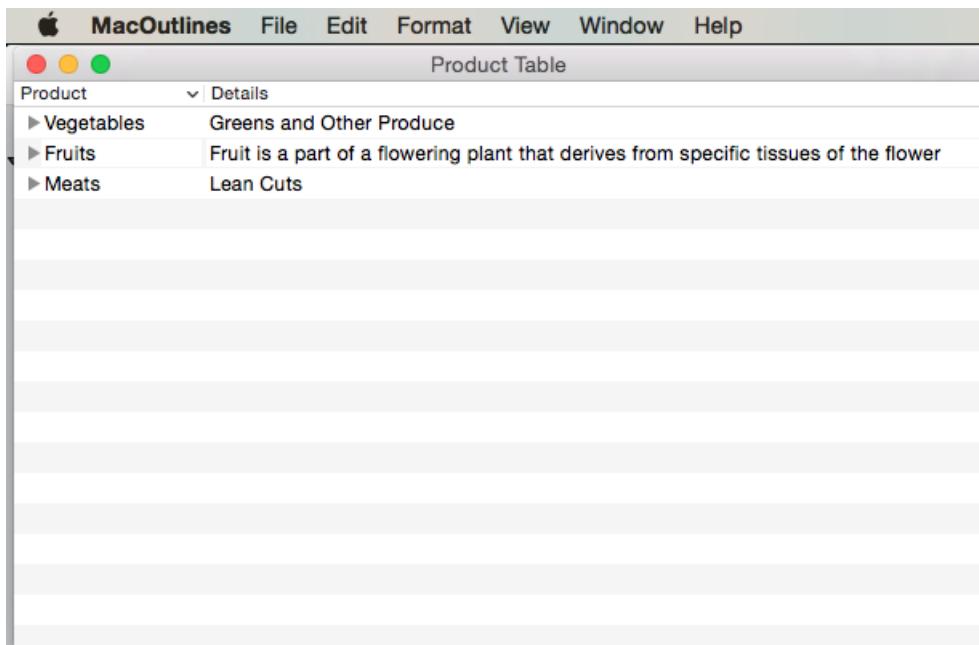
    var Fruits = new Product ("Fruits", "Fruit is a part of a flowering plant that derives from specific tissues of the flower");
    Fruits.Products.Add (new Product ("Grape", "True Berry"));
    Fruits.Products.Add (new Product ("Cucumber", "Pepo"));
    Fruits.Products.Add (new Product ("Orange", "Hesperidium"));
    Fruits.Products.Add (new Product ("Blackberry", "Aggregate fruit"));
    DataSource.Products.Add (Fruits);

    var Meats = new Product ("Meats", "Lean Cuts");
    Meats.Products.Add (new Product ("Beef", "Cow"));
    Meats.Products.Add (new Product ("Pork", "Pig"));
    Meats.Products.Add (new Product ("Veal", "Young Cow"));
    DataSource.Products.Add (Meats);

    // Populate the outline
    ProductOutline.DataSource = DataSource;
    ProductOutline.Delegate = new ProductOutlineDelegate (DataSource);
}

}
```

If we run the application, the following is displayed:



If we expand a node in the Outline View, it will look like the following:

The screenshot shows the MacOutlines application window. The menu bar includes Apple, MacOutlines, File, Edit, Format, View, Window, and Help. The main area displays an outline titled "Product Table". The outline structure is as follows:

- Product** (details expanded)
 - Vegetables** (details expanded)
 - Cabbage: Brassica oleracea - Leaves, axillary buds, stems, flowerheads
 - Turnip: Brassica rapa - Tubers, leaves
 - Radish: Raphanus sativus - Roots, leaves, seed pods, seed oil, sprouting
 - Carrot: Daucus carota - Root tubers
 - Fruits** (details collapsed)
 - Meats** (details collapsed)

Sorting by Column

Let's allow the user to sort the data in the outline by clicking on a Column Header. First, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Select the `Product` column, enter `Title` for the **Sort Key**, `compare:` for the **Selector** and select `Ascending` for the **Order**:

The screenshot shows the Interface Builder environment. The left sidebar lists scenes: Application Scene, Window Controller Scene, and View Controller Scene. Under View Controller Scene, there is a View Controller object with a View child. The View contains a Bordered Scroll View - Outline, which has a Clip View child. Inside the Clip View is a Product Outline, which contains a Product Column. The Product Column is selected, and its properties are shown in the Attributes Inspector on the right.

Table Column
Title: Product
Alignment: <input checked="" type="checkbox"/> Left <input type="checkbox"/> Center <input type="checkbox"/> Right
Title Font: System Small
Sort Key: Title
Selector: compare:
Order: Ascending
Resizing: Both <input checked="" type="checkbox"/> Editable <input type="checkbox"/> Hidden

Save your changes and return to Visual Studio for Mac to sync with Xcode.

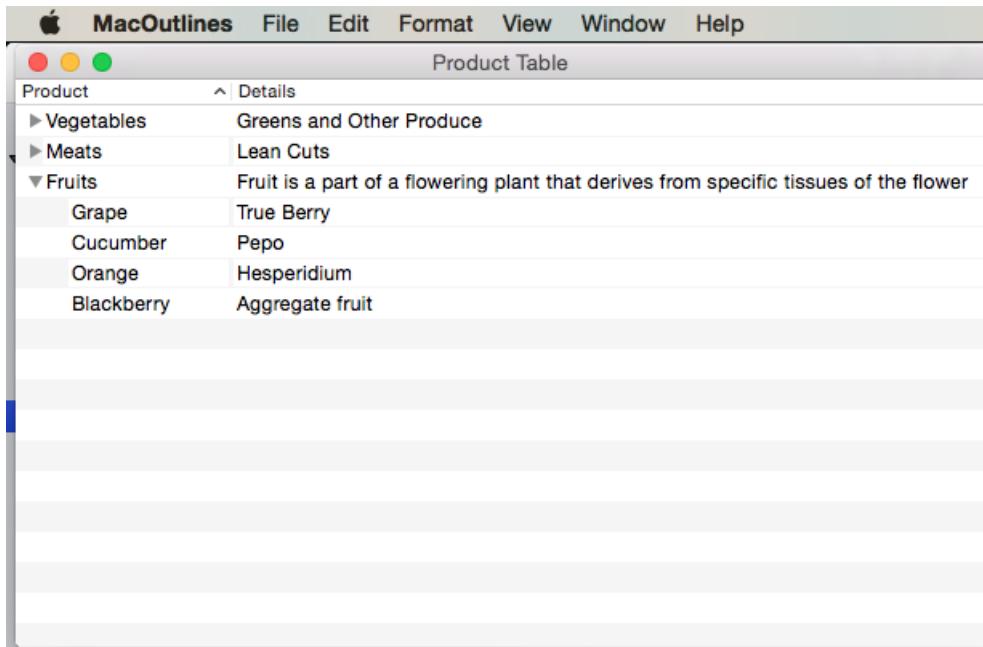
Now let's edit the `ProductOutlineDataSource.cs` file and add the following methods:

```
public void Sort(string key, bool ascending) {
    // Take action based on key
    switch (key) {
        case "Title":
            if (ascending) {
                Products.Sort ((x, y) => x.Title.CompareTo (y.Title));
            } else {
                Products.Sort ((x, y) => -1 * x.Title.CompareTo (y.Title));
            }
            break;
    }
}

public override void SortDescriptorsChanged (NSOutlineView outlineView, NSSortDescriptor[] oldDescriptors)
{
    // Sort the data
    Sort (oldDescriptors [0].Key, oldDescriptors [0].Ascending);
    outlineView.ReloadData ();
}
```

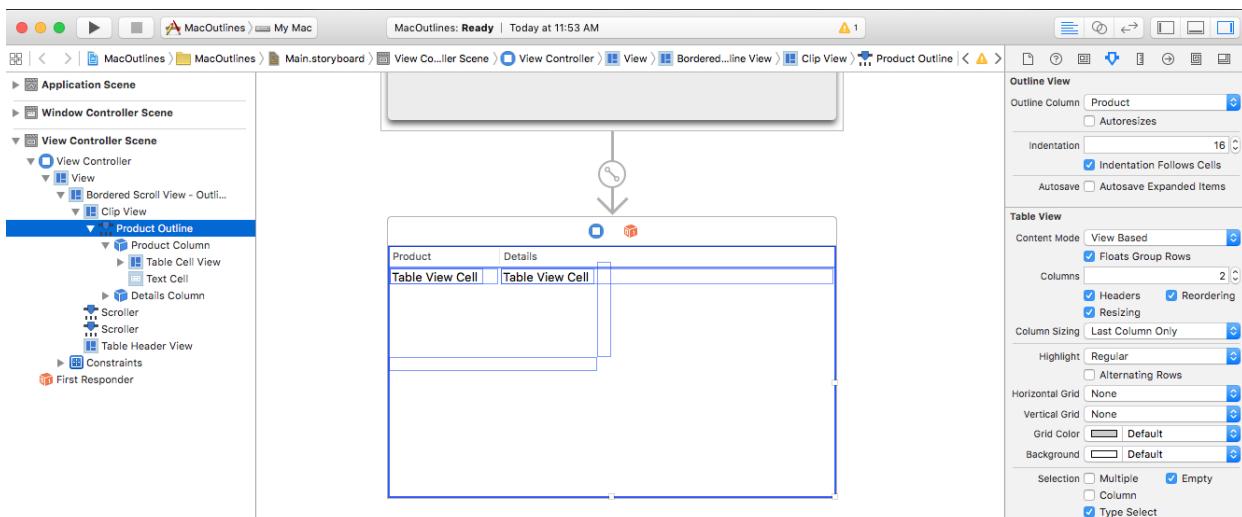
The `Sort` method allow us to sort the data in the Data Source based on a given `Product` class field in either ascending or descending order. The overridden `SortDescriptorsChanged` method will be called every time the user clicks on a Column Heading. It will be passed the `Key` value that we set in Interface Builder and the sort order for that column.

If we run the application and click in the Column Headers, the rows will be sorted by that column:



Row Selection

If you want to allow the user to select a single row, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Select the Outline View in the **Interface Hierarchy** and uncheck the **Multiple** checkbox in the **Attribute Inspector**:



Save your changes and return to Visual Studio for Mac to sync with Xcode.

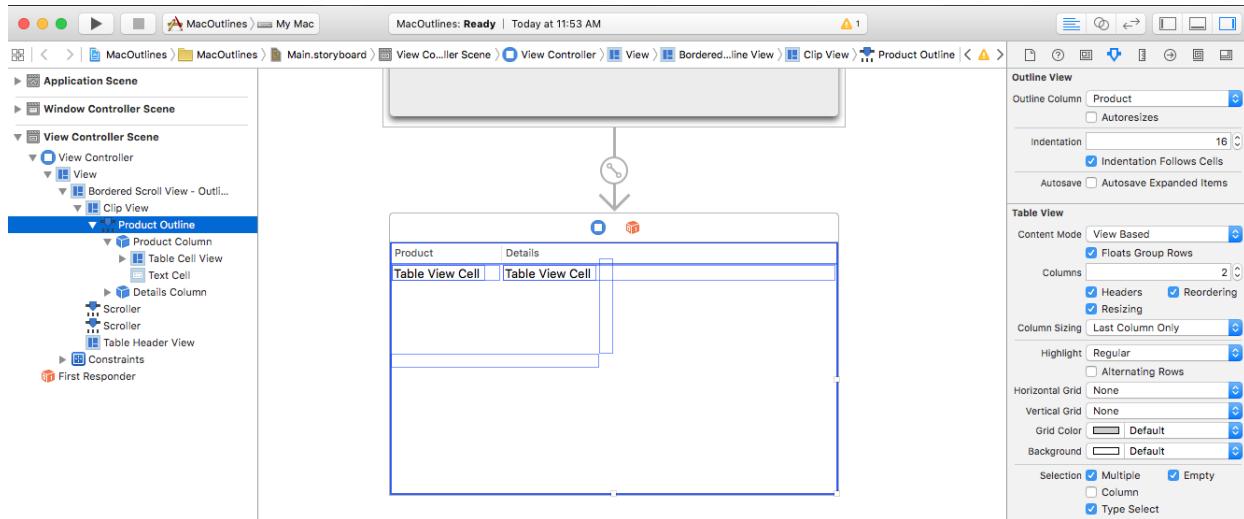
Next, edit the `ProductOutlineDelegate.cs` file and add the following method:

```
public override bool ShouldSelectItem (NSOutlineView outlineView, NSObject item)
{
    // Don't select product groups
    return !((Product)item).IsProductGroup;
}
```

This will allow the user to select any single row in the Outline View. Return `false` for the `ShouldSelectItem` for any item that you don't want the user to be able to select or `false` for every item if you don't want the user to be able to select any items.

Multiple Row Selection

If you want to allow the user to select a multiple rows, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Select the Outline View in the **Interface Hierarchy** and check the **Multiple** checkbox in the **Attribute Inspector**:



Save your changes and return to Visual Studio for Mac to sync with Xcode.

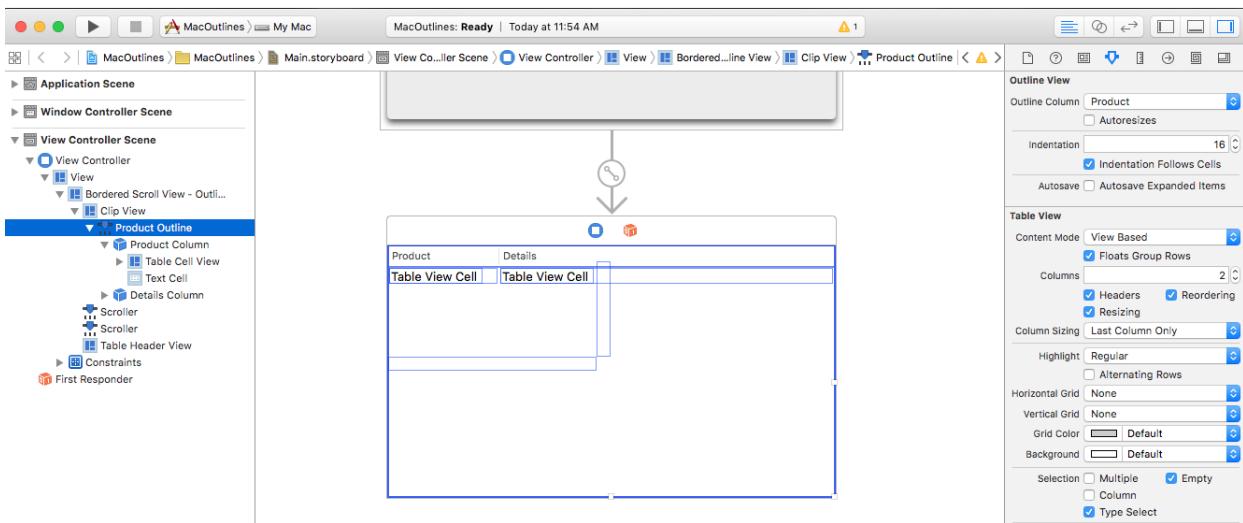
Next, edit the `ProductOutlineDelegate.cs` file and add the following method:

```
public override bool ShouldSelectItem (NSOutlineView outlineView, NSObject item)
{
    // Don't select product groups
    return !((Product)item).IsProductGroup;
}
```

This will allow the user to select any single row in the Outline View. Return `false` for the `ShouldSelectRow` for any item that you don't want the user to be able to select or `false` for every item if you don't want the user to be able to select any items.

Type to Select Row

If you want to allow the user to type a character with the Outline View selected and select the first row that has that character, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Select the Outline View in the **Interface Hierarchy** and check the **Type Select** checkbox in the **Attribute Inspector**:



Save your changes and return to Visual Studio for Mac to sync with Xcode.

Now let's edit the `ProductOutlineDelegate.cs` file and add the following method:

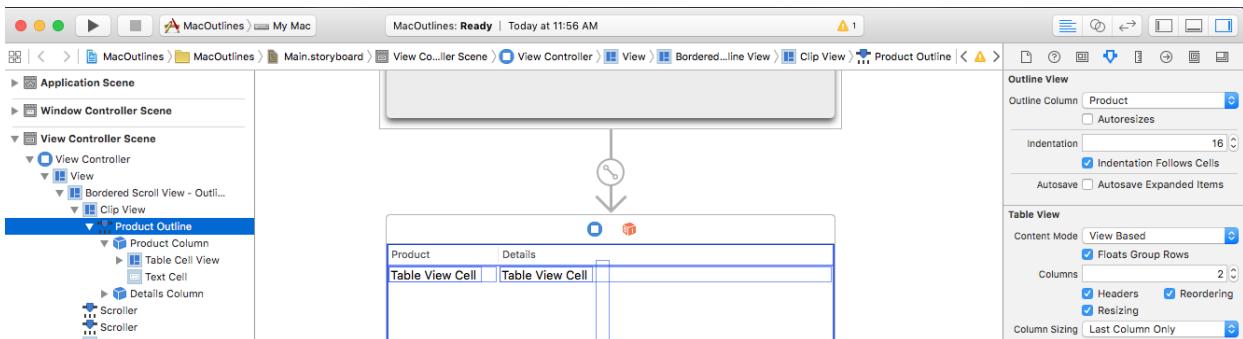
```
public override NSObject GetNextTypeSelectMatch (NSOutlineView outlineView, NSObject startItem, NSObject endItem, string searchString)
{
    foreach(Product product in DataSource.Products) {
        if (product.Title.Contains (searchString)) {
            return product;
        }
    }

    // Not found
    return null;
}
```

The `GetNextTypeSelectMatch` method takes the given `searchString` and returns the item of the first `Product` that has that string in its `Title`.

Reordering Columns

If you want to allow the user to drag reorder columns in the Outline View, double-click the `Main.storyboard` file to open it for editing in Interface Builder. Select the Outline View in the **Interface Hierarchy** and check the **Reordering** checkbox in the **Attribute Inspector**:



If we give a value for the **Autosave** property and check the **Column Information** field, any changes we make to the table's layout will automatically be saved for us and restored the next time the application is run.

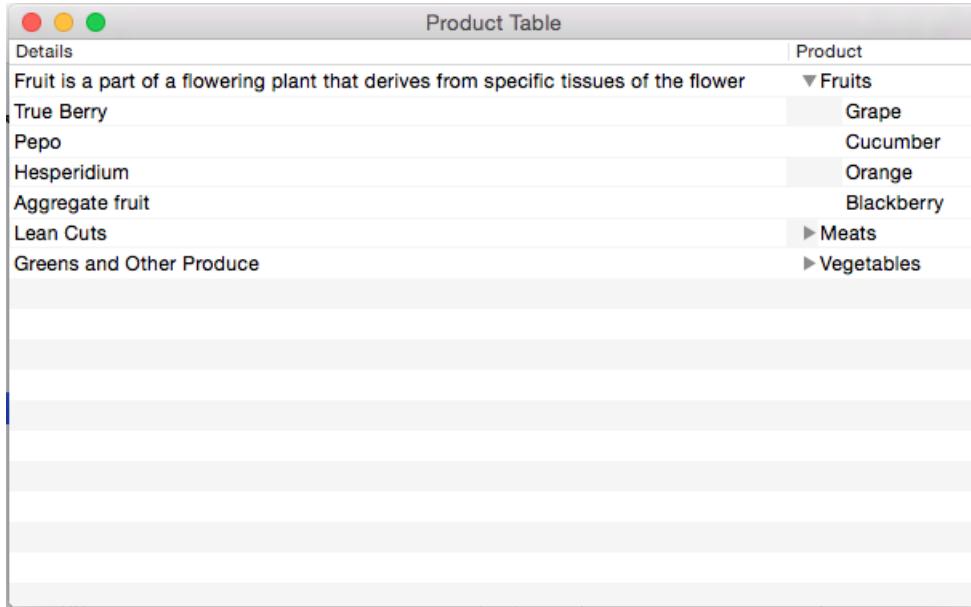
Save your changes and return to Visual Studio for Mac to sync with Xcode.

Now let's edit the `ProductOutlineDelegate.cs` file and add the following method:

```
public override bool ShouldReorder (NSOutlineView outlineView, nint columnIndex, nint newColumnIndex)
{
    return true;
}
```

The `ShouldReorder` method should return `true` for any column that it wants to allow to be drag reordered into the `newColumnIndex`, else return `false`;

If we run the application, we can drag Column Headers around to reorder our columns:



Editing Cells

If you want to allow the user to edit the values for a given cell, edit the `ProductOutlineDelegate.cs` file and change the `GetViewForItem` method as follows:

```

public override NSView GetView (NSOutlineView outlineView, NSTableColumn tableColumn, NSObject item) {
    // Cast item
    var product = item as Product;

    // This pattern allows you reuse existing views when they are no-longer in use.
    // If the returned view is null, you instance up a new view
    // If a non-null view is returned, you modify it enough to reflect the new data
    NSTextField view = (NSTextField)outlineView.MakeView (tableColumn.Title, this);
    if (view == null) {
        view = new NSTextField ();
        view.Identifier = tableColumn.Title;
        view.BackgroundColor = NSColor.Clear;
        view.Bordered = false;
        view.Selectable = false;
        view.Editable = !product.IsProductGroup;
    }

    // Tag view
    view.Tag = outlineView.RowForItem (item);

    // Allow for edit
    view.Ending += (sender, e) => {

        // Grab product
        var prod = outlineView.ItemAtRow(view.Tag) as Product;

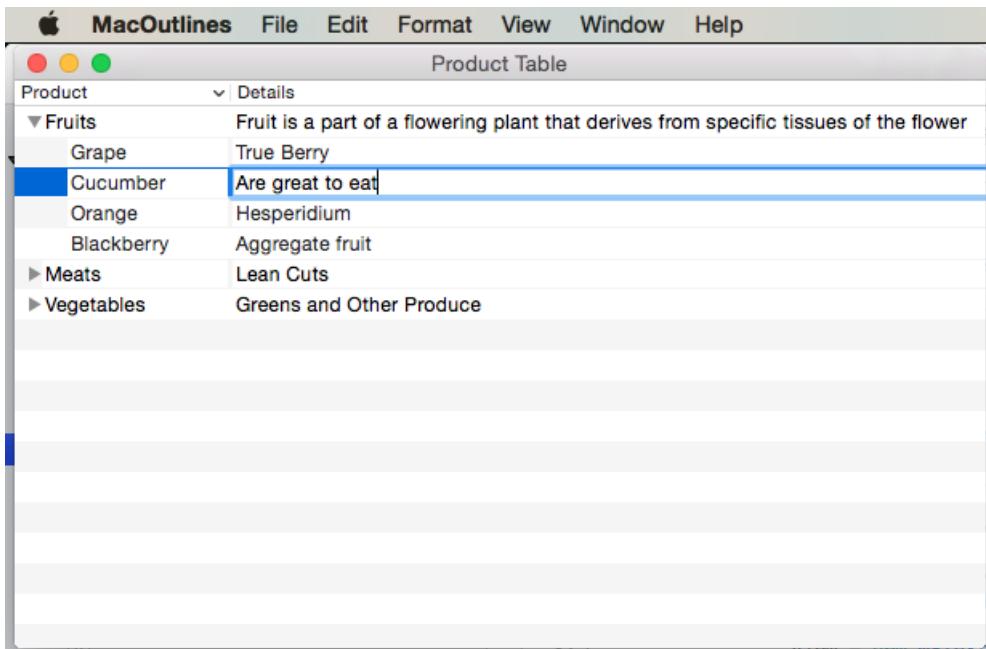
        // Take action based on type
        switch(view.Identifier) {
            case "Product":
                prod.Title = view.StringValue;
                break;
            case "Details":
                prod.Description = view.StringValue;
                break;
        }
    };

    // Setup view based on the column selected
    switch (tableColumn.Title) {
        case "Product":
            view.StringValue = product.Title;
            break;
        case "Details":
            view.StringValue = product.Description;
            break;
    }

    return view;
}

```

Now if we run the application, the user can edit the cells in the Table View:



Using Images in Outline Views

To include an image as part of the cell in a `NSOutlineView`, you'll need to change how the data is returned by the Outline View's `NSTableViewDelegate`'s `GetView` method to use a `NSTableCellView` instead of the typical `NTextField`. For example:

```

public override NSView GetView (NSOutlineView outlineView, NSTableColumn tableColumn, NSObject item) {
    // Cast item
    var product = item as Product;

    // This pattern allows you reuse existing views when they are no-longer in use.
    // If the returned view is null, you instance up a new view
    // If a non-null view is returned, you modify it enough to reflect the new data
    NSTableCellView view = (NSTableCellView)outlineView.MakeView (tableColumn.Title, this);
    if (view == null) {
        view = new NSTableCellView ();
        if (tableColumn.Title == "Product") {
            view.ImageView = new NSImageView (new CGRect (0, 0, 16, 16));
            view.AddSubview (view.ImageView);
            view.TextField = new NSTextField (new CGRect (20, 0, 400, 16));
        } else {
            view.TextField = new NSTextField (new CGRect (0, 0, 400, 16));
        }
        view.TextField.AutoresizingMask = NSViewResizingMask.WidthSizable;
        view.AddSubview (view.TextField);
        view.Identifier = tableColumn.Title;
        view.TextField.BackgroundColor = NSColor.Clear;
        view.TextField.Bordered = false;
        view.TextField.Selectable = false;
        view.TextField.Editable = !product.IsProductGroup;
    }

    // Tag view
    view.TextField.Tag = outlineView.RowForItem (item);

    // Allow for edit
    view.TextField.Ending += (sender, e) => {

        // Grab product
        var prod = outlineView.ItemAtRow(view.Tag) as Product;

        // Take action based on type
        switch(view.Identifier) {
        case "Product":
            prod.Title = view.TextField.StringValue;
            break;
        case "Details":
            prod.Description = view.TextField.StringValue;
            break;
        }
    };

    // Setup view based on the column selected
    switch (tableColumn.Title) {
    case "Product":
        view.ImageView.Image = NSImage.ImageNamed (product.IsProductGroup ? "tags.png" : "tag.png");
        view.TextField.StringValue = product.Title;
        break;
    case "Details":
        view.TextField.StringValue = product.Description;
        break;
    }

    return view;
}

```

For more information, please see the [Using Images with Outline Views](#) section of our [Working with Image](#) documentation.

Data Binding Outline Views

By using Key-Value Coding and Data Binding techniques in your Xamarin.Mac application, you can greatly decrease the amount of code that you have to write and maintain to populate and work with UI elements. You also have the benefit of further decoupling your backing data (*Data Model*) from your front end User Interface (*Model-View-Controller*), leading to easier to maintain, more flexible application design.

Key-Value Coding (KVC) is a mechanism for accessing an object's properties indirectly, using keys (specially formatted strings) to identify properties instead of accessing them through instance variables or accessor methods (`get/set`). By implementing Key-Value Coding compliant accessors in your Xamarin.Mac application, you gain access to other macOS features such as Key-Value Observing (KVO), Data Binding, Core Data, Cocoa bindings, and scriptability.

For more information, please see the [Outline View Data Binding](#) section of our [Data Binding and Key-Value Coding](#) documentation.

Summary

This article has taken a detailed look at working with Outline Views in a Xamarin.Mac application. We saw the different types and uses of Outline Views, how to create and maintain Outline Views in Xcode's Interface Builder and how to work with Outline Views in C# code.

Related Links

- [MacOutlines \(sample\)](#)
- [MacImages \(sample\)](#)
- [Hello, Mac](#)
- [Table Views](#)
- [Source Lists](#)
- [Data Binding and Key-Value Coding](#)
- [OS X Human Interface Guidelines](#)
- [Introduction to Outline Views](#)
- [NSOutlineView](#)
- [NSOutlineViewDataSource](#)
- [NSOutlineViewDelegate](#)

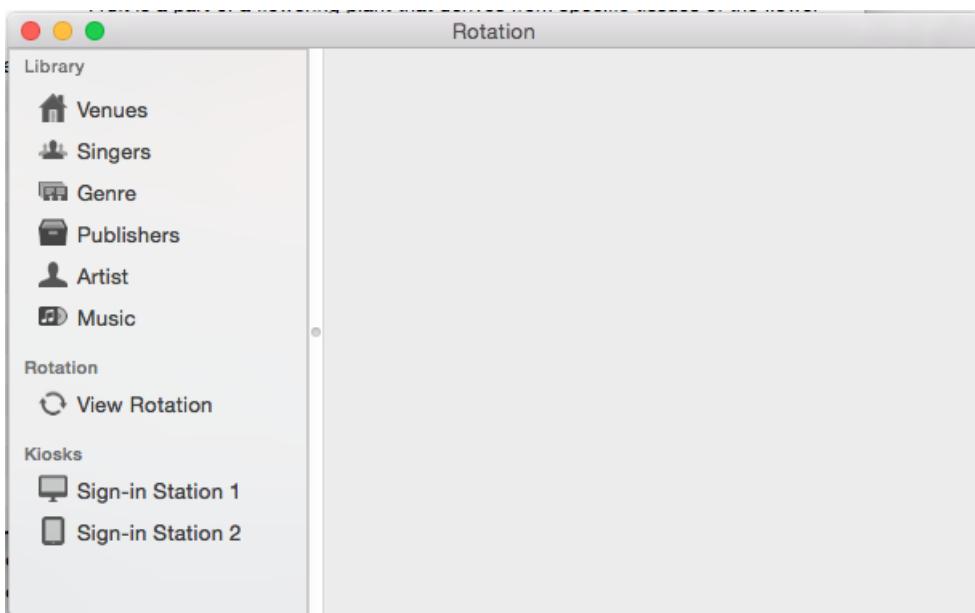
Source lists in Xamarin.Mac

3/5/2021 • 11 minutes to read • [Edit Online](#)

This article covers working with source lists in a Xamarin.Mac application. It describes creating and maintaining source lists in Xcode and Interface Builder and interacting with them in C# code.

When working with C# and .NET in a Xamarin.Mac application, you have access to the same Source Lists that a developer working in *Objective-C* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, you can use Xcode's *Interface Builder* to create and maintain your Source Lists (or optionally create them directly in C# code).

A Source List is a special type of Outline View used to show the source of an action, like the side bar in Finder or iTunes.



In this article, we'll cover the basics of working with Source Lists in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` commands used to wire-up your C# classes to Objective-C objects and UI Elements.

Introduction to Source Lists

As stated above, a Source List is a special type of Outline View used to show the source of an action, like the side bar in Finder or iTunes. A Source List is a type of Table that allows the user expand or collapse rows of hierarchical data. Unlike a Table View, items in an Source List are not in a flat list, they are organized in a hierarchy, like files and folders on a hard drive. If an item in an Source List contains other items, it can be expanded or collapsed by the user.

The Source List is a specially styled Outline View (`NSOutlineView`), which itself is a subclass of the Table View (`NSTableView`) and as such, inherits much of its behavior from its parent class. As a result, many operations supported by a Outline View, are also supported by an Source List. A Xamarin.Mac application has control of

these features, and can configure the Source List's parameters (either in code or Interface Builder) to allow or disallow certain operations.

A Source List does not store its own data, instead it relies on a Data Source (`NSOutlineViewDataSource`) to provide both the rows and columns required, on a as-needed basis.

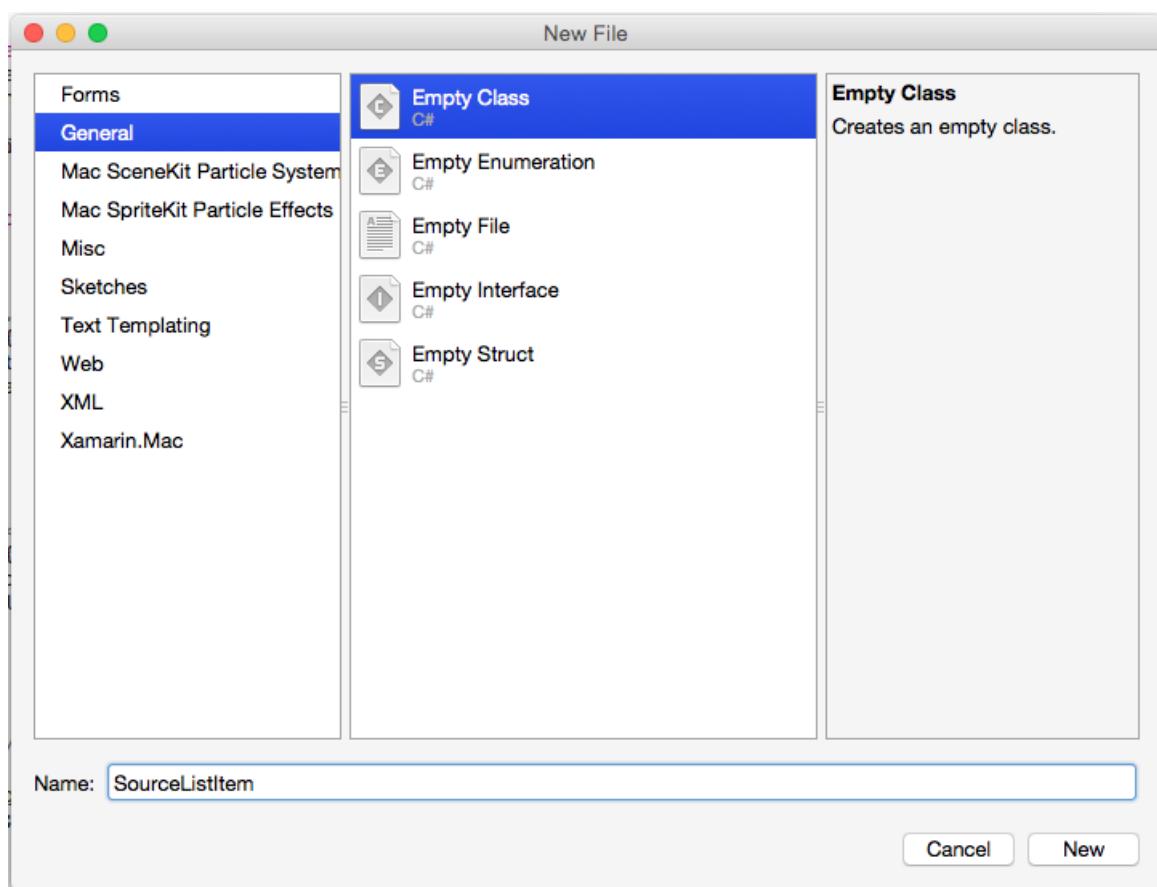
A Source List's behavior can be customized by providing a subclass of the Outline View Delegate (`NSOutlineViewDelegate`) to support Outline type to select functionality, item selection and editing, custom tracking, and custom views for individual items.

Since a Source List shares much of its behavior and functionality with a Table View and an Outline View, you might want to go through our [Table Views](#) and [Outline Views](#) documentation before continuing with this article.

Working with Source Lists

A Source List is a special type of Outline View used to show the source of an action, like the side bar in Finder or iTunes. Unlike Outline Views, before we define our Source List in Interface Builder, let's create the backing classes in Xamarin.Mac.

First, let's create a new `SourceListItem` class to hold the data for our Source List. In the **Solution Explorer**, right-click the Project and select **Add > New File...**. Select **General > Empty Class**, enter `SourceListItem` for the **Name** and click the **New** button:



Make the `SourceListItem.cs` file look like the following:

```
using System;
using System.Collections;
using System.Collections.Generic;
using AppKit;
using Foundation;

namespace MacOutlines
{
```

```

public class SourceListItem: NSObject, I Enumerator, IEnumerable
{
    #region Private Properties
    private string _title;
    private NSImage _icon;
    private string _tag;
    private List<SourceListItem> _items = new List<SourceListItem> ();
    #endregion

    #region Computed Properties
    public string Title {
        get { return _title; }
        set { _title = value; }
    }

    public NSImage Icon {
        get { return _icon; }
        set { _icon = value; }
    }

    public string Tag {
        get { return _tag; }
        set { _tag = value; }
    }
    #endregion

    #region Indexer
    public SourceListItem this[int index]
    {
        get
        {
            return _items[index];
        }

        set
        {
            _items[index] = value;
        }
    }

    public int Count {
        get { return _items.Count; }
    }

    public bool HasChildren {
        get { return (Count > 0); }
    }
    #endregion

    #region Enumerable Routines
    private int _position = -1;

    public I Enumerator GetEnumerator()
    {
        _position = -1;
        return (I Enumerator) this;
    }

    public bool MoveNext()
    {
        _position++;
        return (_position < _items.Count);
    }

    public void Reset()
    {_position = -1; }

    public object Current
    {

```

```

    }

    get
    {
        try
        {
            return _items[_position];
        }

        catch (IndexOutOfRangeException)
        {
            throw new InvalidOperationException();
        }
    }
}

#endregion

#region Constructors
public SourceListItem ()
{
}

public SourceListItem (string title)
{
    // Initialize
    this._title = title;
}

public SourceListItem (string title, string icon)
{
    // Initialize
    this._title = title;
    this._icon = NSImage.ImageNamed (icon);
}

public SourceListItem (string title, string icon, ClickedDelegate clicked)
{
    // Initialize
    this._title = title;
    this._icon = NSImage.ImageNamed (icon);
    this.Clicked = clicked;
}

public SourceListItem (string title, NSImage icon)
{
    // Initialize
    this._title = title;
    this._icon = icon;
}

public SourceListItem (string title, NSImage icon, ClickedDelegate clicked)
{
    // Initialize
    this._title = title;
    this._icon = icon;
    this.Clicked = clicked;
}

public SourceListItem (string title, NSImage icon, string tag)
{
    // Initialize
    this._title = title;
    this._icon = icon;
    this._tag = tag;
}

public SourceListItem (string title, NSImage icon, string tag, ClickedDelegate clicked)
{
    // Initialize
    this._title = title;
    ...
}

```

```

        this._icon = icon;
        this._tag = tag;
        this.Clicked = clicked;
    }
}

#region Public Methods
public void AddItem(SourceListItem item) {
    _items.Add (item);
}

public void AddItem(string title) {
    _items.Add (new SourceListItem (title));
}

public void AddItem(string title, string icon) {
    _items.Add (new SourceListItem (title, icon));
}

public void AddItem(string title, string icon, ClickedDelegate clicked) {
    _items.Add (new SourceListItem (title, icon, clicked));
}

public void AddItem(string title, NSImage icon) {
    _items.Add (new SourceListItem (title, icon));
}

public void AddItem(string title, NSImage icon, ClickedDelegate clicked) {
    _items.Add (new SourceListItem (title, icon, clicked));
}

public void AddItem(string title, NSImage icon, string tag) {
    _items.Add (new SourceListItem (title, icon, tag));
}

public void AddItem(string title, NSImage icon, string tag, ClickedDelegate clicked) {
    _items.Add (new SourceListItem (title, icon, tag, clicked));
}

public void Insert(int n, SourceListItem item) {
    _items.Insert (n, item);
}

public void RemoveItem(SourceListItem item) {
    _items.Remove (item);
}

public void RemoveItem(int n) {
    _items.RemoveAt (n);
}

public void Clear() {
    _items.Clear ();
}
#endregion

#region Events
public delegate void ClickedDelegate();
public event ClickedDelegate Clicked;

internal void RaiseClickedEvent() {
    // Inform caller
    if (this.Clicked != null)
        this.Clicked ();
}
#endregion
}
}

```

In the Solution Explorer, right-click the Project and select Add > New File... Select General > Empty Class, enter `SourceListDataSource` for the Name and click the New button. Make the `SourceListDataSource.cs` file look like the following:

```
using System;
using System.Collections;
using System.Collections.Generic;
using AppKit;
using Foundation;

namespace MacOutlines
{
    public class SourceListDataSource : NSOutlineViewDataSource
    {
        #region Private Variables
        private SourceListView _controller;
        #endregion

        #region Public Variables
        public List<SourceListItem> Items = new List<SourceListItem>();
        #endregion

        #region Constructors
        public SourceListDataSource (SourceListView controller)
        {
            // Initialize
            this._controller = controller;
        }
        #endregion

        #region Override Properties
        public override nint GetChildrenCount (NSOutlineView outlineView, Foundation.NSObject item)
        {
            if (item == null) {
                return Items.Count;
            } else {
                return ((SourceListItem)item).Count;
            }
        }

        public override bool ItemExpandable (NSOutlineView outlineView, Foundation.NSObject item)
        {
            return ((SourceListItem)item).HasChildren;
        }

        public override NSObject GetChild (NSOutlineView outlineView, nint childIndex, Foundation.NSObject item)
        {
            if (item == null) {
                return Items [(int)childIndex];
            } else {
                return ((SourceListItem)item) [(int)childIndex];
            }
        }
    }

    public override NSObject GetObjectValue (NSOutlineView outlineView, NSTableColumn TableColumn,
    NSObject item)
    {
        return new NSString (((SourceListItem)item).Title);
    }
    #endregion

    #region Internal Methods
    internal SourceListItem ItemForRow(int row)
    {
        int index = 0;

        // Look at each group
    }
}
```

```

        foreach (SourceListItem item in Items) {
            // Is the row inside this group?
            if (row >= index && row <= (index + item.Count)) {
                return item [row - index - 1];
            }

            // Move index
            index += item.Count + 1;
        }

        // Not found
        return null;
    }
    #endregion
}
}

```

This will provide the data for our Source List.

In the **Solution Explorer**, right-click the Project and select **Add > New File...** Select **General > Empty Class**, enter `SourceListDelegate` for the **Name** and click the **New** button. Make the `SourceListDelegate.cs` file look like the following:

```

using System;
using AppKit;
using Foundation;

namespace MacOutlines
{
    public class SourceListDelegate : NSOutlineViewDelegate
    {
        #region Private variables
        private SourceListView _controller;
        #endregion

        #region Constructors
        public SourceListDelegate (SourceListView controller)
        {
            // Initialize
            this._controller = controller;
        }
        #endregion

        #region Override Methods
        public override bool ShouldEditTableColumn (NSOutlineView outlineView, NSTableColumn TableColumn,
Foundation.NSObject item)
        {
            return false;
        }

        public override NSCell GetCell (NSOutlineView outlineView, NSTableColumn TableColumn,
Foundation.NSObject item)
        {
            nint row = outlineView.RowForItem (item);
            return TableColumn.DataCellForRow (row);
        }

        public override bool IsGroupItem (NSOutlineView outlineView, Foundation.NSObject item)
        {
            return ((SourceListItem)item).HasChildren;
        }

        public override NSView GetView (NSOutlineView outlineView, NSTableColumn TableColumn, NSObject item)
        {
            NSTableCellView view = null;

```

```

        // Is this a group item?
        if (((SourceListItem)item).HasChildren) {
            view = (NSTableCellView)outlineView.MakeView ("HeaderCell", this);
        } else {
            view = (NSTableCellView)outlineView.MakeView ("DataCell", this);
            view.ImageView.Image = ((SourceListItem)item).Icon;
        }

        // Initialize view
        view.TextField.StringValue = ((SourceListItem)item).Title;

        // Return new view
        return view;
    }

    public override bool ShouldSelectItem (NSOutlineView outlineView, Foundation.NSObject item)
    {
        return (outlineView.GetParent (item) != null);
    }

    public override void SelectionDidChange (NSNotification notification)
    {
        NSIndexSet selectedIndexes = _controller.SelectedRows;

        // More than one item selected?
        if (selectedIndexes.Count > 1) {
            // Not handling this case
        } else {
            // Grab the item
            var item = _controller.Data.ItemForRow ((int)selectedIndexes.FirstIndex);

            // Was an item found?
            if (item != null) {
                // Fire the clicked event for the item
                item.RaiseClickedEvent ();

                // Inform caller of selection
                _controller.RaiseItemSelected (item);
            }
        }
    }
}

```

This will provide the behavior of our Source List.

Finally, in the **Solution Explorer**, right-click the Project and select **Add > New File...** Select **General > Empty Class**, enter `SourceListView` for the **Name** and click the **New** button. Make the `SourceListView.cs` file look like the following:

```

using System;
using AppKit;
using Foundation;

namespace MacOutlines
{
    [Register("SourceListView")]
    public class SourceListView : NSOutlineView
    {
        #region Computed Properties
        public SourceListDataSource Data {
            get {return (SourceListDataSource)this.DataSource; }
        }
        #endregion

        #region Constructors

```

```

...->...->...
public SourceListView ()
{
}

public SourceListView (IntPtr handle) : base(handle)
{
}

public SourceListView (NSCoder coder) : base(coder)
{
}

public SourceListView (NSObjectFlag t) : base(t)
{
}

}
#endifregion

#region Override Methods
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();
}
#endifregion

#region Public Methods
public void Initialize() {

    // Initialize this instance
    this.DataSource = new SourceListDataSource (this);
    this.Delegate = new SourceListDelegate (this);

}

public void AddItem(SourceListItem item) {
    if (Data != null) {
        Data.Items.Add (item);
    }
}
#endifregion

#region Events
public delegate void ItemSelectedDelegate(SourceListItem item);
public event ItemSelectedDelegate ItemSelected;

internal void RaiseItemSelected(SourceListItem item) {
    // Inform caller
    if (this.ItemSelected != null) {
        this.ItemSelected (item);
    }
}
#endifregion
}
}

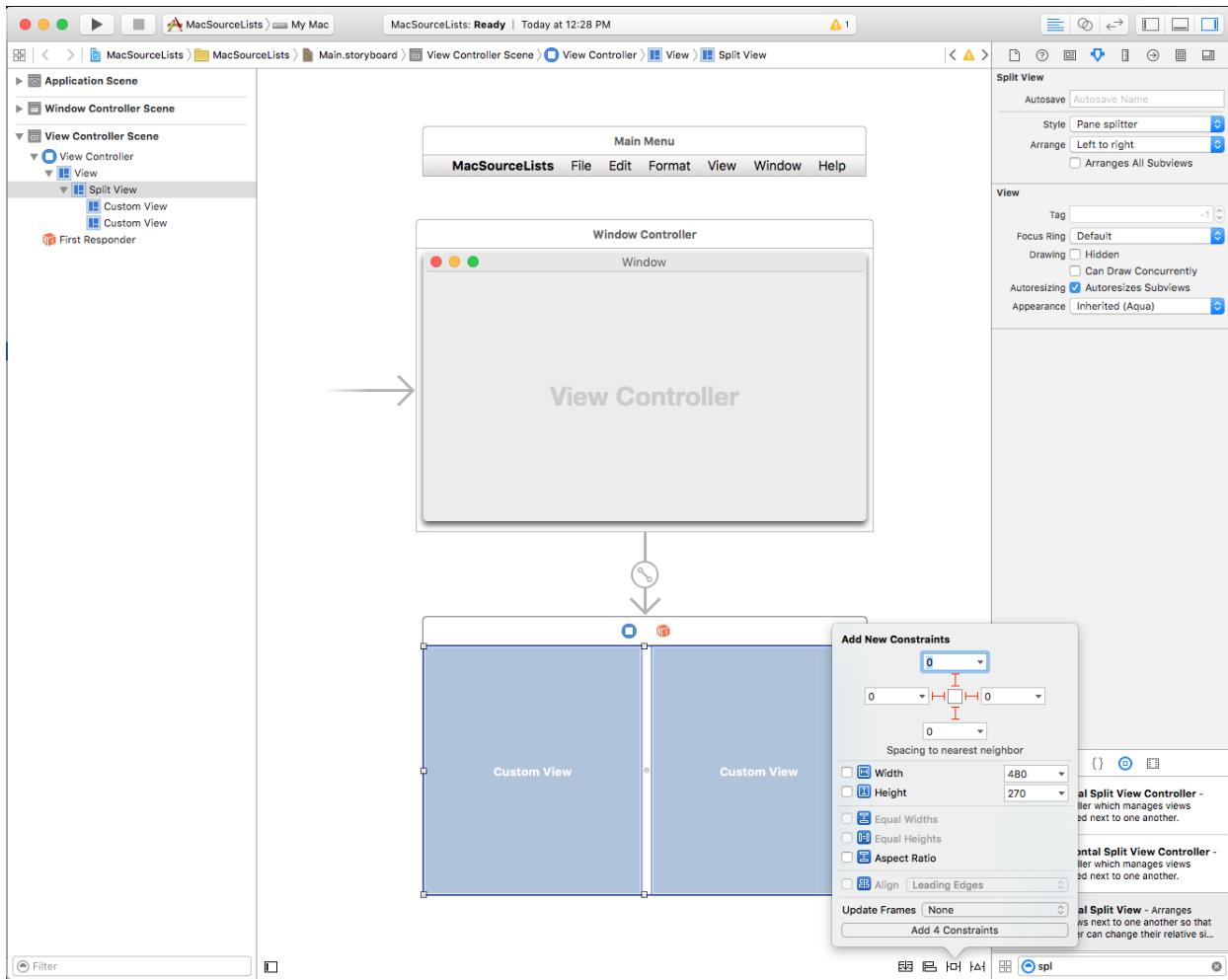
```

This creates a custom, reusable subclass of `NSOutlineView` (`SourceListView`) that we can use to drive the Source List in any Xamarin.Mac application that we make.

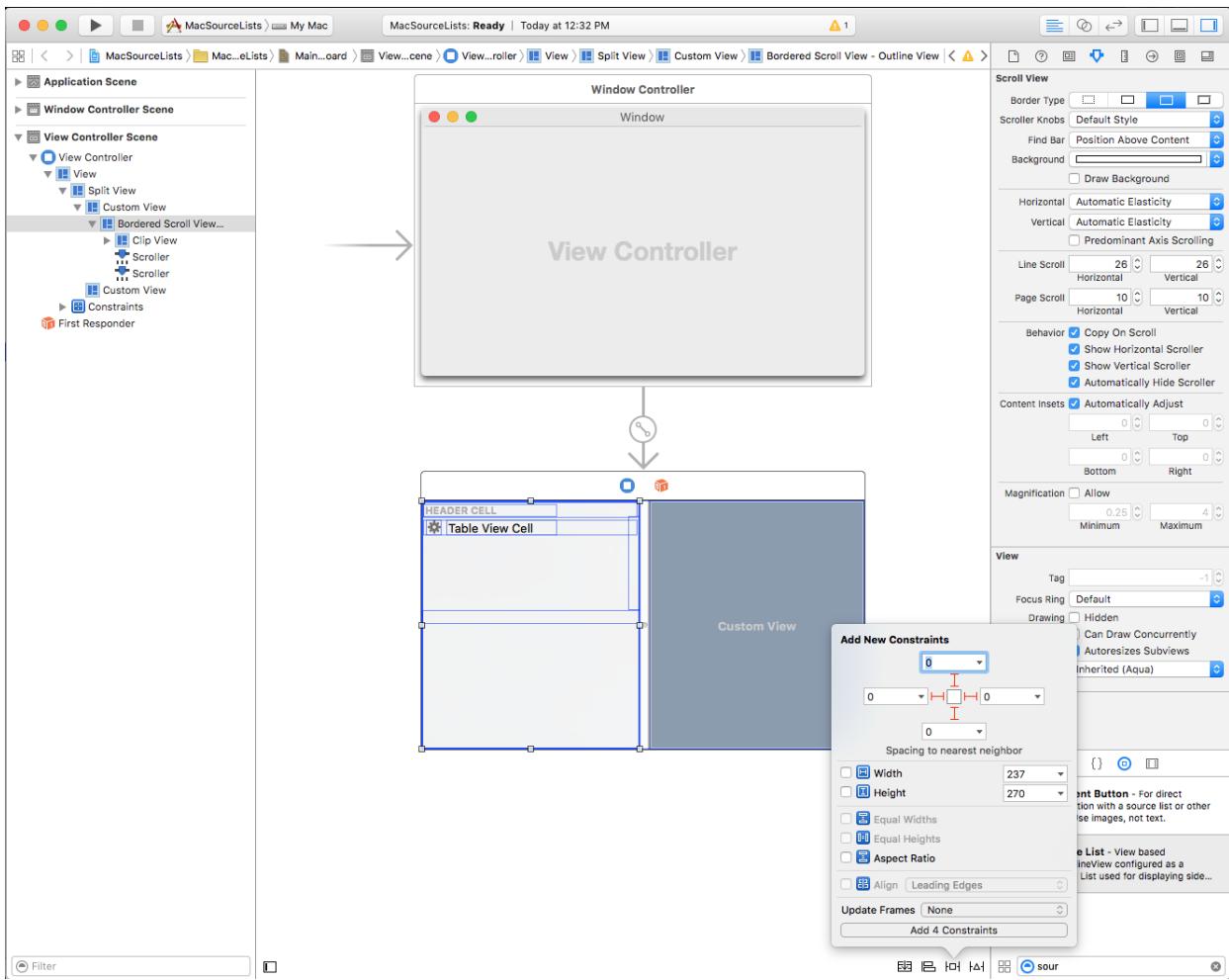
Creating and Maintaining Source Lists in Xcode

Now, let's design our Source List in Interface Builder. Double-click the `Main.storyboard` file to open it for editing in Interface Builder and drag a Split View from the **Library Inspector**, add it to the View Controller and set it to

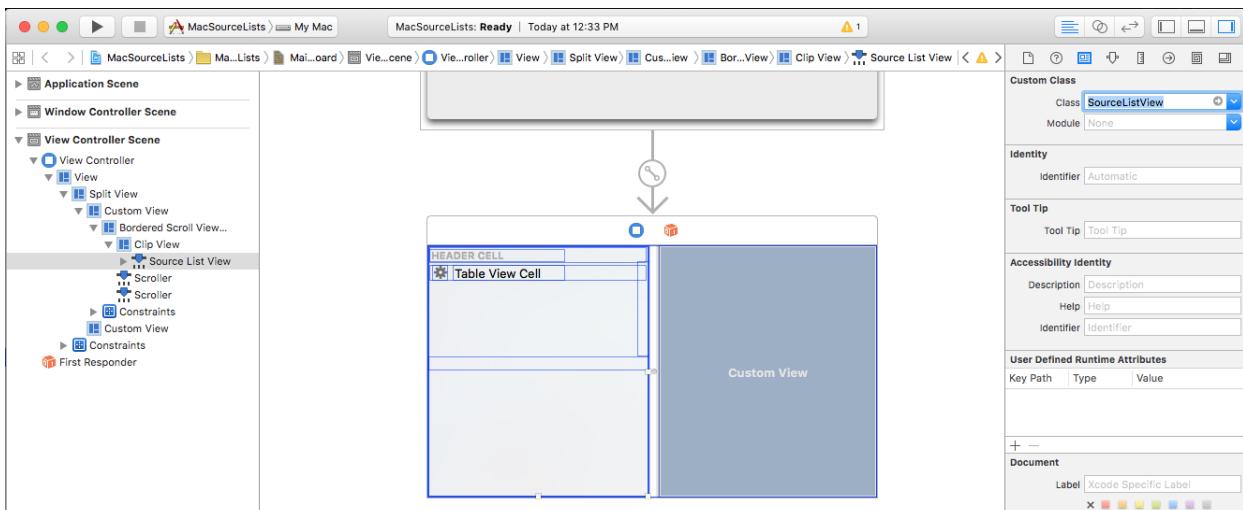
resize with the View in the Constraints Editor:



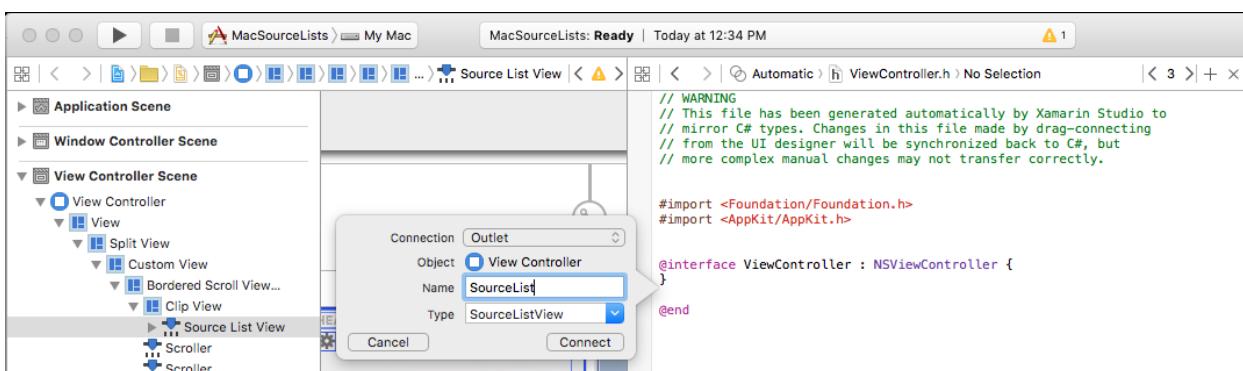
Next, drag a Source List from the **Library Inspector**, add it to the left side of the Split View and set it to resize with the View in the **Constraints Editor**:



Next, switch to the **Identity View**, select the Source List, and change its **Class** to `SourceListView`:



Finally, create an **Outlet** for our Source List called `SourceList` in the `ViewController.h` file:



Save your changes and return to Visual Studio for Mac to sync with Xcode.

Populating the Source List

Let's edit the `RotationWindow.cs` file in Visual Studio for Mac and make it's `AwakeFromNib` method look like the following:

```
public override void AwakeFromNib ()
{
    base.AwakeFromNib ();

    // Populate source list
    SourceList.Initialize ();

    var library = new SourceListItem ("Library");
    library.AddItem ("Venues", "house.png", () => {
        Console.WriteLine("Venue Selected");
    });
    library.AddItem ("Singers", "group.png");
    library.AddItem ("Genre", "cards.png");
    library.AddItem ("Publishers", "box.png");
    library.AddItem ("Artist", "person.png");
    library.AddItem ("Music", "album.png");
    SourceList.AddItem (library);

    // Add Rotation
    var rotation = new SourceListItem ("Rotation");
    rotation.AddItem ("View Rotation", "redo.png");
    SourceList.AddItem (rotation);

    // Add Kiosks
    var kiosks = new SourceListItem ("Kiosks");
    kiosks.AddItem ("Sign-in Station 1", "imac");
    kiosks.AddItem ("Sign-in Station 2", "ipad");
    SourceList.AddItem (kiosks);

    // Display side list
    SourceList.ReloadData ();
    SourceList.ExpandItem (null, true);
}
```

The `Initialize()` method need to be called against our Source List's `Outlet` before any items are added to it. For each group of items, we create a parent item and then add the sub items to that group item. Each group is then added to the Source List's collection `SourceList.AddItem(...)`. The last two lines load the data for the Source List and expands all groups:

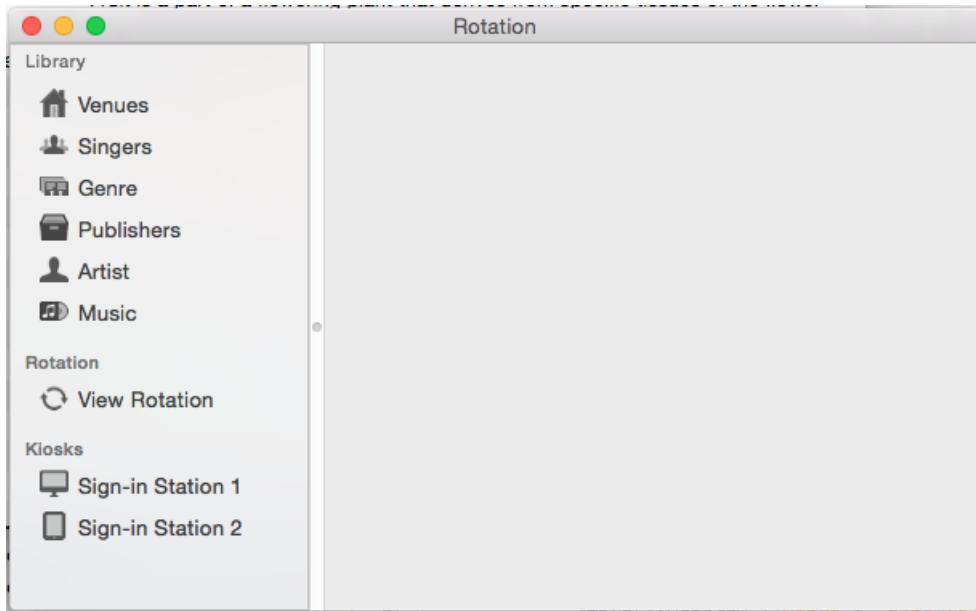
```
// Display side list
SourceList.ReloadData ();
SourceList.ExpandItem (null, true);
```

Finally, edit the `AppDelegate.cs` file and make the `DidFinishLaunching` method look like the following:

```
public override void DidFinishLaunching (NSNotification notification)
{
    mainWindowController = new MainWindowController ();
    mainWindowController.Window.MakeKeyAndOrderFront (this);

    var rotation = new RotationWindowController ();
    rotation.Window.MakeKeyAndOrderFront (this);
}
```

If we run our application, the following will be displayed:



Summary

This article has taken a detailed look at working with Source Lists in a Xamarin.Mac application. We saw how to create and maintain Source Lists in Xcode's Interface Builder and how to work with Source Lists in C# code.

Related Links

- [MacOutlines \(sample\)](#)
- [Hello, Mac](#)
- [Table Views](#)
- [Outline Views](#)
- [OS X Human Interface Guidelines](#)
- [Introduction to Outline Views](#)
- [NSOutlineView](#)
- [NSOutlineViewDataSource](#)
- [NSOutlineViewDelegate](#)

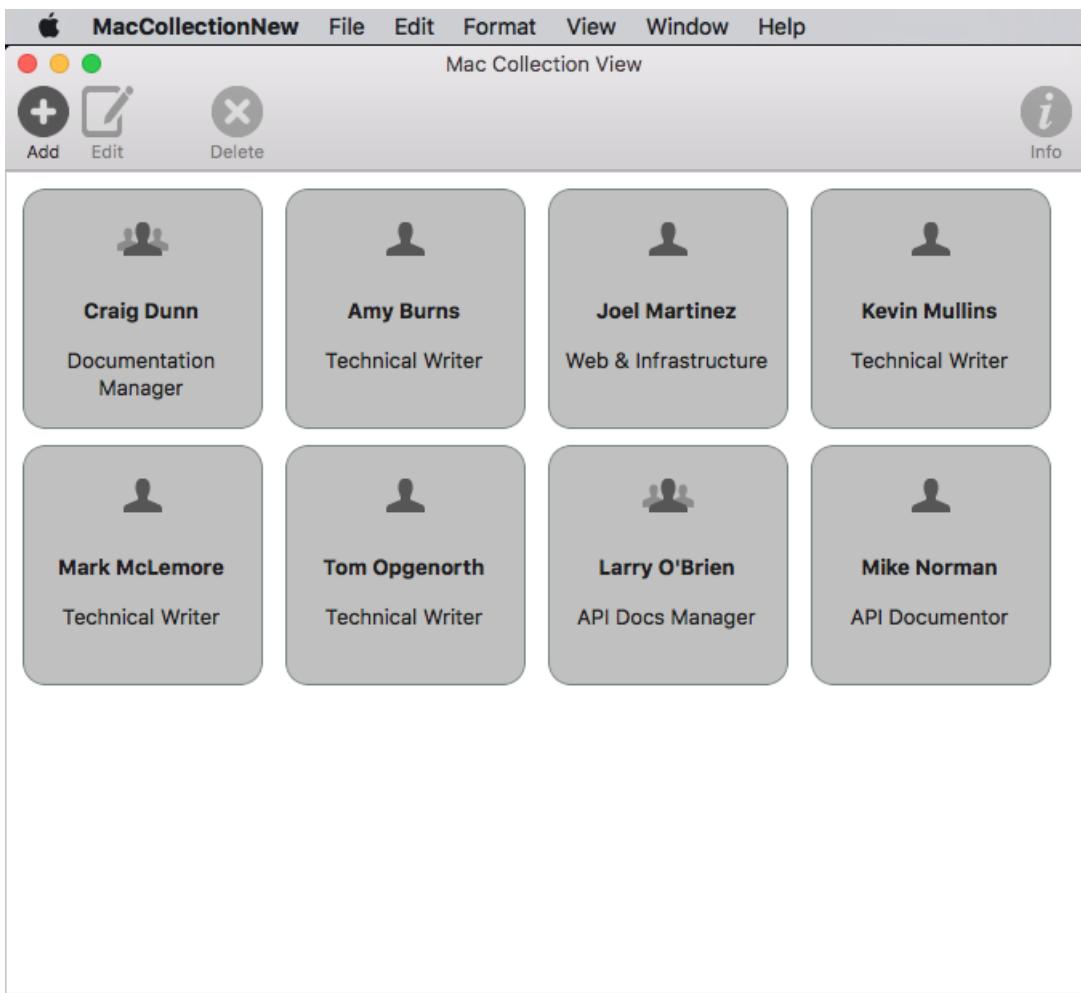
Collection Views in Xamarin.Mac

3/5/2021 • 17 minutes to read • [Edit Online](#)

This article describes working with collection views in a Xamarin.Mac app. It covers creating and maintaining collection views in Xcode and Interface Builder and working with them programmatically.

When working with C# and .NET in a Xamarin.Mac app, the developer has access to the same AppKit Collection View controls that a developer working in *Objective-C* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, the developer uses Xcode's *Interface Builder* to create and maintain Collection Views.

A `NSCollectionView` displays a grid of subviews organized using a `NSCollectionViewLayout`. Each subview in the grid is represented by a `NSCollectionViewItem` which manages the loading of the view's content from a `.xib` file.



This article covers the basics of working with Collection Views in a Xamarin.Mac app. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that are used throughout this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` commands used to wire-up your C# classes to Objective-C objects and UI Elements.

About Collection Views

The main goal of a Collection View (`NSCollectionView`) is to visually arrange a group of objects in an organized fashion using a Collection View Layout (`NSCollectionViewLayout`), with each individual object (`NSCollectionViewItem`) getting its own View in the larger collection. Collection Views work via Data Binding and Key-Value Coding techniques and as such, you should read the [Data Binding and Key-Value Coding](#) documentation before continuing with this article.

The Collection View has no standard, built-in Collection View Item (like an Outline or Table View does), so the developer is responsible for designing and implementing a *Prototype View* using other AppKit controls such as Image Fields, Text Fields, Labels, etc. This Prototype View will be used to display and work with each item being managed by the Collection View and is stored in a `.xib` file.

Because the developer is responsible for the look and feel of a Collection View Item, the Collection View has no built-in support for highlighting a selected item in the grid. Implementing this feature will be covered in this article.

Defining the Data Model

Before Data Binding a Collection View in Interface Builder, a Key-Value Coding (KVC)/Key-Value Observing (KVO) compliant class must be defined in the Xamarin.Mac app to act as the *Data Model* for the binding. The Data Model provides all of the data that will be displayed in the collection and receives any modifications to the data that the user makes in the UI while running the application.

Take the example of an app that manages a group of employees, the following class could be used to define the Data Model:

```
using System;
using Foundation;
using AppKit;

namespace MacDatabinding
{
    [Register("PersonModel")]
    public class PersonModel : NSObject
    {
        #region Private Variables
        private string _name = "";
        private string _occupation = "";
        private bool _isManager = false;
        private NSMutableArray _people = new NSMutableArray();
        #endregion

        #region Computed Properties
        [Export("Name")]
        public string Name {
            get { return _name; }
            set {
                WillChangeValue ("Name");
                _name = value;
                DidChangeValue ("Name");
            }
        }
        #endregion

        [Export("Occupation")]
        public string Occupation {
            get { return _occupation; }
            set {
                WillChangeValue ("Occupation");
                _occupation = value;
                DidChangeValue ("Occupation");
            }
        }
    }
}
```

```

[Export("isManager")]
public bool isManager {
    get { return _isManager; }
    set {
        WillChangeValue ("isManager");
        WillChangeValue ("Icon");
        _isManager = value;
        DidChangeValue ("isManager");
        DidChangeValue ("Icon");
    }
}

[Export("isEmployee")]
public bool isEmployee {
    get { return (NumberOfEmployees == 0); }
}

[Export("Icon")]
public NSImage Icon
{
    get
    {
        if (isManager)
        {
            return NSImage.ImageNamed("IconGroup");
        }
        else
        {
            return NSImage.ImageNamed("IconUser");
        }
    }
}

[Export("personModelArray")]
public NSArray People {
    get { return _people; }
}

[Export("NumberOfEmployees")]
public nint NumberOfEmployees {
    get { return (nint)_people.Count; }
}
#endregion

#region Constructors
public PersonModel ()
{
}

public PersonModel (string name, string occupation)
{
    // Initialize
    this.Name = name;
    this.Occupation = occupation;
}

public PersonModel (string name, string occupation, bool manager)
{
    // Initialize
    this.Name = name;
    this.Occupation = occupation;
    this.isManager = manager;
}
#endregion

#region Array Controller Methods
[Export("addObject:")]
public void AddPerson(PersonModel person) {
}

```

```

        WillChangeValue ("personModelArray");
        isManager = true;
        _people.Add (person);
        DidChangeValue ("personModelArray");
    }

    [Export("insertObject:inPersonModelArrayAtIndex:")]
    public void InsertPerson(PersonModel person, nint index) {
        WillChangeValue ("personModelArray");
        _people.Insert (person, index);
        DidChangeValue ("personModelArray");
    }

    [Export("removeObjectFromPersonModelArrayAtIndex:")]
    public void RemovePerson(nint index) {
        WillChangeValue ("personModelArray");
        _people.RemoveObject (index);
        DidChangeValue ("personModelArray");
    }

    [Export("setPersonModelArray:")]
    public void SetPeople(NSMutableArray array) {
        WillChangeValue ("personModelArray");
        _people = array;
        DidChangeValue ("personModelArray");
    }
    #endregion
}
}

```

The `PersonModel` Data Model will be used throughout the rest of this article.

Working with a Collection View

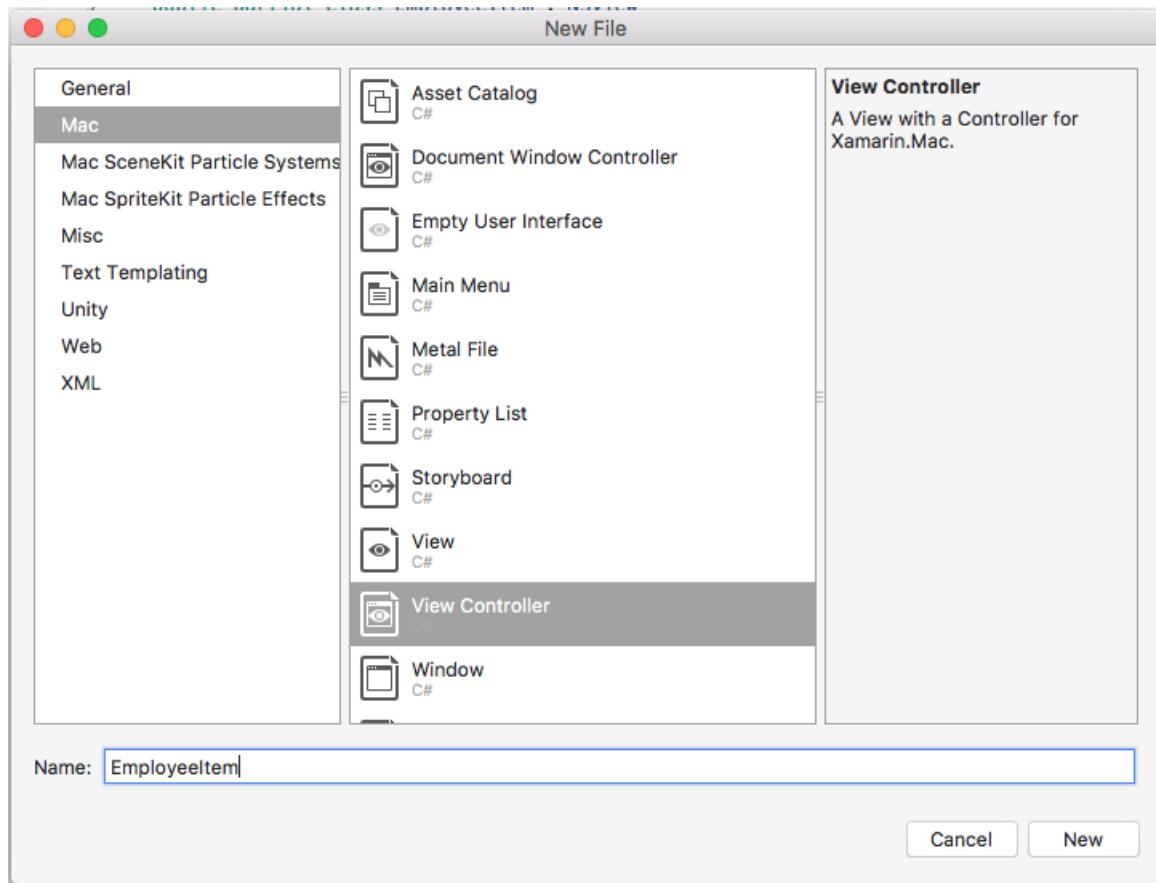
Data Binding with a Collection View is very much like binding with a Table View, as `NSCollectionViewDataSource` is used to provide data for the collection. Since the collection view doesn't have a preset display format, more work is required to provide user interaction feedback and to track user selection.

Creating the Cell Prototype

Since the Collection View does not include a default cell prototype, the developer will need to add one or more `.xib` files to the Xamarin.Mac app to define the layout and content of the individual cells.

Do the following:

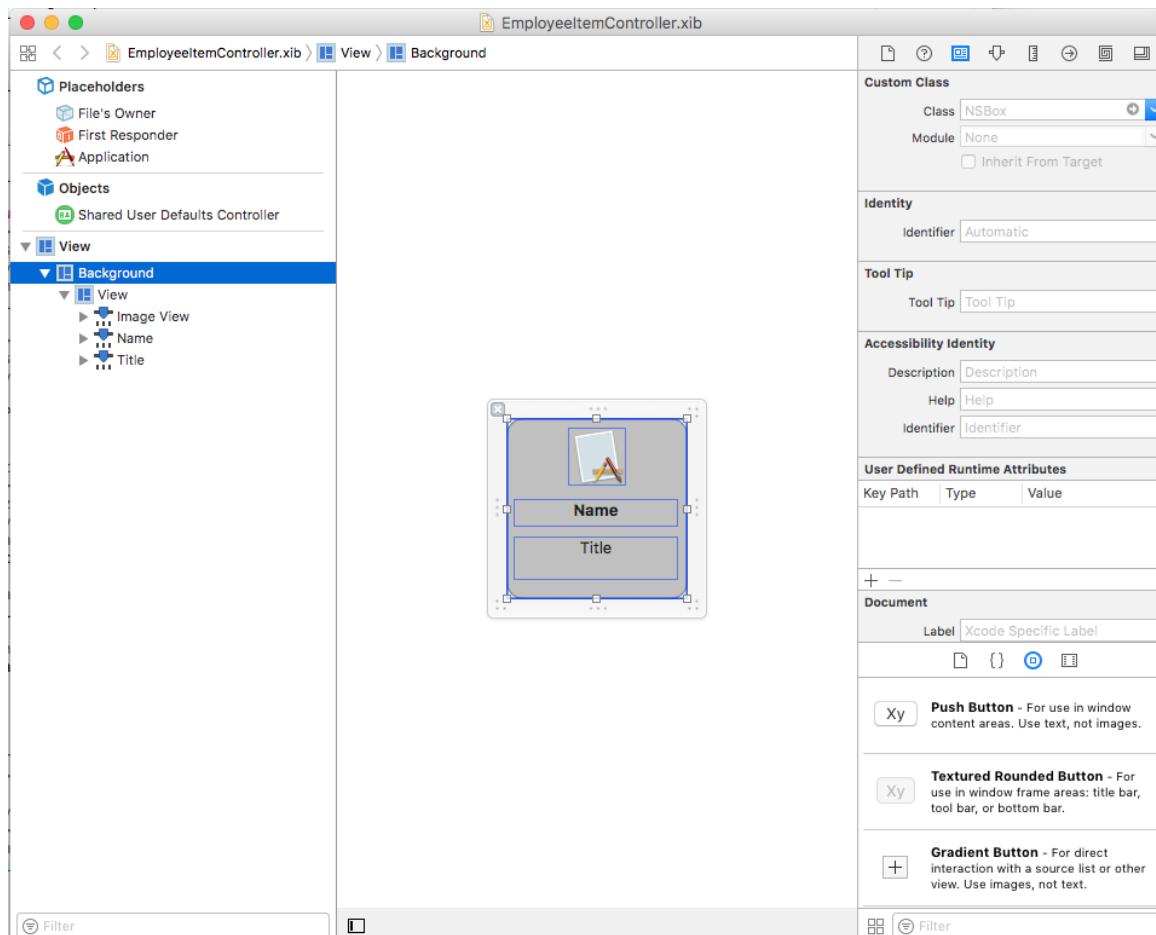
1. In the **Solution Explorer**, right-click on the project name and select **Add > New File...**
2. Select **Mac > View Controller**, give it a name (such as `EmployeeItem` in this example) and click the **New** button to create:



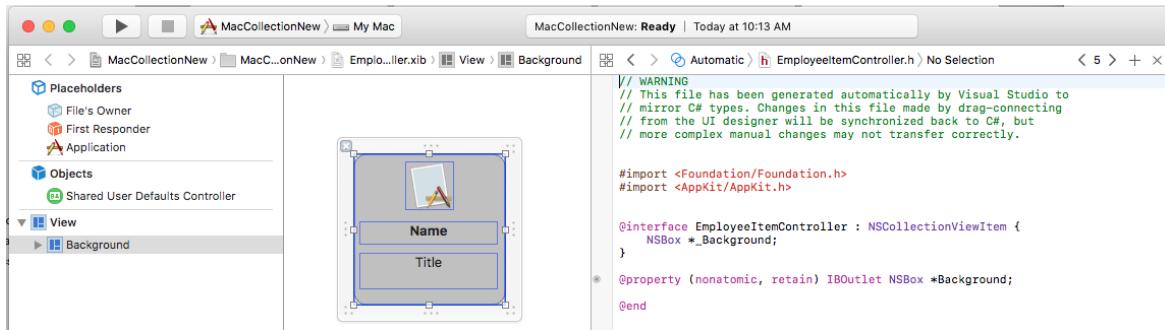
This will add an `EmployeeItem.cs`, `EmployeeItemController.cs` and `EmployeeItemController.xib` file to the project's solution.

3. Double-click the `EmployeeItemController.xib` file to open it for editing in Xcode's Interface Builder.

4. Add an `NSBox`, `NSImageView` and two `NSLabel` controls to the View and lay them out as follows:

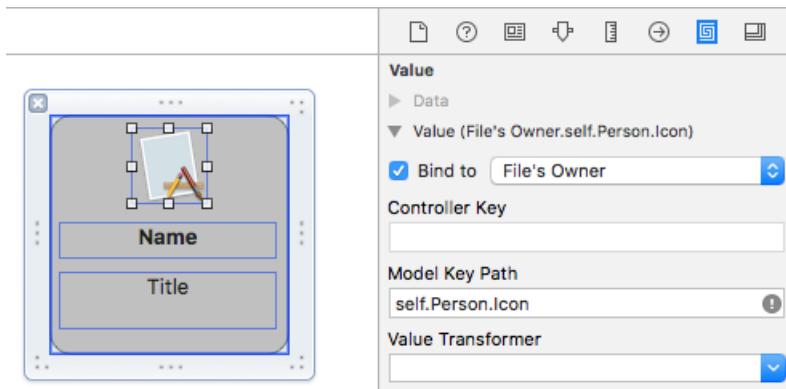


5. Open the Assistant Editor and create an Outlet for the `NSBox` so that it can be used to indicate the selection state of a cell:

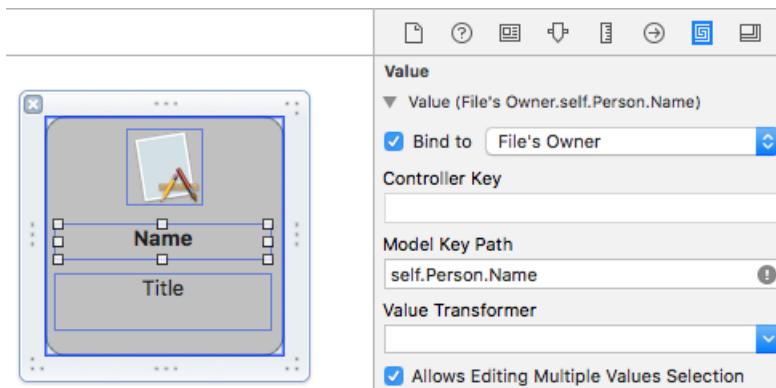


6. Return to the Standard Editor and select the Image View.

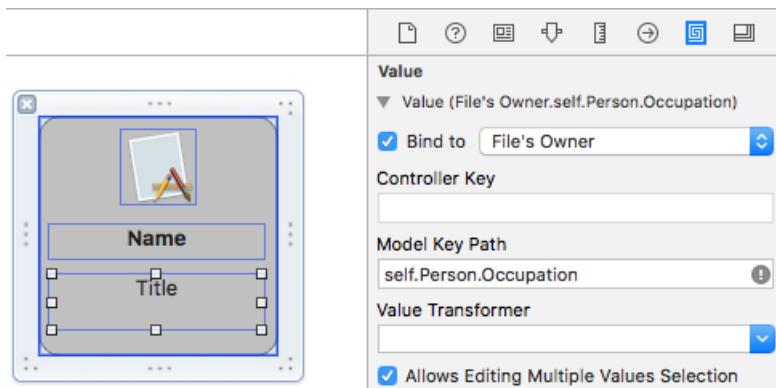
7. In the Binding Inspector, select Bind To > File's Owner and enter a Model Key Path of `self.Person.Icon`:



8. Select the first Label and in the Binding Inspector, select Bind To > File's Owner and enter a Model Key Path of `self.Person.Name`:



9. Select the second Label and in the Binding Inspector, select Bind To > File's Owner and enter a Model Key Path of `self.Person.Occupation`:



10. Save the changes to the `.xib` file and return to Visual Studio to sync the changes.

Edit the `EmployeeItemController.cs` file and make it look like the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Foundation;
using AppKit;

namespace MacCollectionNew
{
    /// <summary>
    /// The Employee item controller handles the display of the individual items that will
    /// be displayed in the collection view as defined in the associated .XIB file.
    /// </summary>
    public partial class EmployeeItemController : NSCollectionViewItem
    {
        #region Private Variables
        /// <summary>
        /// The person that will be displayed.
        /// </summary>
        private PersonModel _person;
        #endregion

        #region Computed Properties
        // strongly typed view accessor
        public new EmployeeItem View
        {
            get
            {
                return (EmployeeItem)base.View;
            }
        }

        /// <summary>
        /// Gets or sets the person.
        /// </summary>
        /// <value>The person that this item belongs to.</value>
        [Export("Person")]
        public PersonModel Person
        {
            get { return _person; }
            set
            {
                WillChangeValue("Person");
                _person = value;
                DidChangeValue("Person");
            }
        }

        /// <summary>
        /// Gets or sets the color of the background for the item.
        /// </summary>
        /// <value>The color of the background.</value>
        public NSColor BackgroundColor {
            get { return Background.FillColor; }
            set { Background.FillColor = value; }
        }

        /// <summary>
        /// Gets or sets a value indicating whether this <see
        cref="T:MacCollectionNew.EmployeeItemController"/> is selected.
        /// </summary>
        /// <value><c>true</c> if selected; otherwise, <c>false</c>.</value>
        /// <remarks>This also changes the background color based on the selected state
        /// of the item.</remarks>
        public override bool Selected
    }
}
```

```

public override void Selected
{
    get
    {
        return base.Selected;
    }
    set
    {
        base.Selected = value;

        // Set background color based on the selection state
        if (value) {
            BackgroundColor = NSColor.DarkGray;
        } else {
            BackgroundColor = NSColor.LightGray;
        }
    }
}
#endregion

#region Constructors
// Called when created from unmanaged code
public EmployeeItemController(IntPtr handle) : base(handle)
{
    Initialize();
}

// Called when created directly from a XIB file
[Export("initWithCoder:")]
public EmployeeItemController(NSCoder coder) : base(coder)
{
    Initialize();
}

// Call to load from the XIB/NIB file
public EmployeeItemController() : base("EmployeeItem", NSBundle.MainBundle)
{
    Initialize();
}

// Added to support loading from XIB/NIB
public EmployeeItemController(string nibName, NSBundle nibBundle) : base(nibName, nibBundle) {

    Initialize();
}

// Shared initialization code
void Initialize()
{
}
#endregion
}
}

```

Looking at this code in detail, the class inherits from `NSCollectionViewItem` so it can act as a prototype for a Collection View cell. The `Person` property exposes the class that was used to data bind to the Image View and Labels in Xcode. This is an instance of the `PersonModel` created above.

The `BackgroundColor` property is a shortcut to the `NSBox` control's `FillColor` that will be used to show the selection status of a cell. By overriding the `Selected` property of the `NSCollectionViewItem`, the following code sets or clears this selection state:

```

public override bool Selected
{
    get
    {
        return base.Selected;
    }
    set
    {
        base.Selected = value;

        // Set background color based on the selection state
        if (value) {
            BackgroundColor = NSColor.DarkGray;
        } else {
            BackgroundColor = NSColor.LightGray;
        }
    }
}

```

Creating the Collection View Data Source

A Collection View Data Source (`NSCollectionViewDataSource`) provides all of the data for a Collection View and creates and populates a Collection View Cell (using the `.xib` prototype) as required for each item in the collection.

Add a new class the project, call it `collectionViewDataSource` and make it look like the following:

```

using System;
using System.Collections.Generic;
using AppKit;
using Foundation;

namespace MacCollectionNew
{
    /// <summary>
    /// Collection view data source provides the data for the collection view.
    /// </summary>
    public class CollectionViewDataSource : NSCollectionViewDataSource
    {
        #region Computed Properties
        /// <summary>
        /// Gets or sets the parent collection view.
        /// </summary>
        /// <value>The parent collection view.</value>
        public NSCollectionView ParentCollectionView { get; set; }

        /// <summary>
        /// Gets or sets the data that will be displayed in the collection.
        /// </summary>
        /// <value>A collection of PersonModel objects.</value>
        public List<PersonModel> Data { get; set; } = new List<PersonModel>();
        #endregion

        #region Constructors
        /// <summary>
        /// Initializes a new instance of the <see cref="T:MacCollectionNew.CollectionViewDataSource"/>
        class.
        /// </summary>
        /// <param name="parent">The parent collection that this datasource will provide data for.</param>
        public CollectionViewDataSource(NSCollectionView parent)
        {
            // Initialize
            ParentCollectionView = parent;

            // Attach to collection view
        }
    }
}

```

```

        parent.DataSource = this;

    }

#endregion

#region Override Methods
/// <summary>
/// Gets the number of sections.
/// </summary>
/// <returns>The number of sections.</returns>
/// <param name="collectionView">The parent Collection view.</param>
public override nint GetNumberOfSections(NSCollectionView collectionView)
{
    // There is only one section in this view
    return 1;
}

/// <summary>
/// Gets the number of items in the given section.
/// </summary>
/// <returns>The number of items.</returns>
/// <param name="collectionView">The parent Collection view.</param>
/// <param name="section">The Section number to count items for.</param>
public override nint GetNumberofItems(NSCollectionView collectionView, nint section)
{
    // Return the number of items
    return Data.Count;
}

/// <summary>
/// Gets the item for the give section and item index.
/// </summary>
/// <returns>The item.</returns>
/// <param name="collectionView">The parent Collection view.</param>
/// <param name="indexPath">Index path specifying the section and index.</param>
public override NSCollectionViewItem GetItem(NSCollectionView collectionView, NSIndexPath indexPath)
{
    var item = collectionView.MakeItem("EmployeeCell", indexPath) as EmployeeItemController;
    item.Person = Data[(int)indexPath.Item];

    return item;
}
#endregion
}

```

Looking at this code in detail, the class inherits from `NSCollectionViewDataSource` and exposes a List of `PersonModel` instances through its `Data` property.

Since this collection only has one section, the code overrides the `GetNumberOfSections` method and always returns `1`. Additionally, the `GetNumberofItems` method is overridden as it returns the number of items in the `Data` property list.

The `GetItem` method is called whenever a new cell is required and looks like the following:

```

public override NSCollectionViewItem GetItem(NSCollectionView collectionView, NSIndexPath indexPath)
{
    var item = collectionView.MakeItem("EmployeeCell", indexPath) as EmployeeItemController;
    item.Person = Data[(int)indexPath.Item];

    return item;
}

```

The `MakeItem` method of the Collection View is called to create or return a reusable instance of the

`EmployeeItemController` and its `Person` property is set to item being displayed in the requested cell.

The `EmployeeItemController` must be registered with the Collection View Controller beforehand using the following code:

```
EmployeeCollection.RegisterClassForItem(typeof(EmployeeItemController), "EmployeeCell");
```

The `Identifier` (`EmployeeCell`) used in the `MakeItem` call *must* match the name of the View Controller that was registered with the Collection View. This step will be covered in detail below.

Handling Item Selection

To handle the selection and deselection of items in the collection, a `NSCollectionViewDelegate` will be required.

Since this example will be using the built in `NSCollectionViewFlowLayout` layout type, a

`NSCollectionViewDelegateFlowLayout` specific version of this delegate will be required.

Add a new class to the project, call it `CollectionViewDelegate` and make it look like the following:

```

using System;
using Foundation;
using AppKit;

namespace MacCollectionNew
{
    /// <summary>
    /// Collection view delegate handles user interaction with the elements of the
    /// collection view for the Flow-Based layout type.
    /// </summary>
    public class CollectionViewDelegate : NSCollectionViewDelegateFlowLayout
    {
        #region Computed Properties
        /// <summary>
        /// Gets or sets the parent view controller.
        /// </summary>
        /// <value>The parent view controller.</value>
        public ViewController ParentViewController { get; set; }
        #endregion

        #region Constructors
        /// <summary>
        /// Initializes a new instance of the <see cref="T:MacCollectionNew.CollectionViewDelegate"/> class.
        /// </summary>
        /// <param name="parentViewController">Parent view controller.</param>
        public CollectionViewDelegate(ViewController parentViewController)
        {
            // Initialize
            ParentViewController = parentViewController;
        }
        #endregion

        #region Override Methods
        /// <summary>
        /// Handles one or more items being selected.
        /// </summary>
        /// <param name="collectionView">The parent Collection view.</param>
        /// <param name="indexPaths">The Index paths of the items being selected.</param>
        public override void ItemsSelected(NSCollectionView collectionView, NSSet indexPaths)
        {
            // Dereference path
            var paths = indexPaths.ToArray<NSIndexPath>();
            var index = (int)paths[0].Item;

            // Save the selected item
            ParentViewController.PersonSelected = ParentViewController.DataSource.Data[index];
        }

        /// <summary>
        /// Handles one or more items being deselected.
        /// </summary>
        /// <param name="collectionView">The parent Collection view.</param>
        /// <param name="indexPaths">The Index paths of the items being deselected.</param>
        public override void ItemsDeselected(NSCollectionView collectionView, NSSet indexPaths)
        {
            // Dereference path
            var paths = indexPaths.ToArray<NSIndexPath>();
            var index = paths[0].Item;

            // Clear selection
            ParentViewController.PersonSelected = null;
        }
        #endregion
    }
}

```

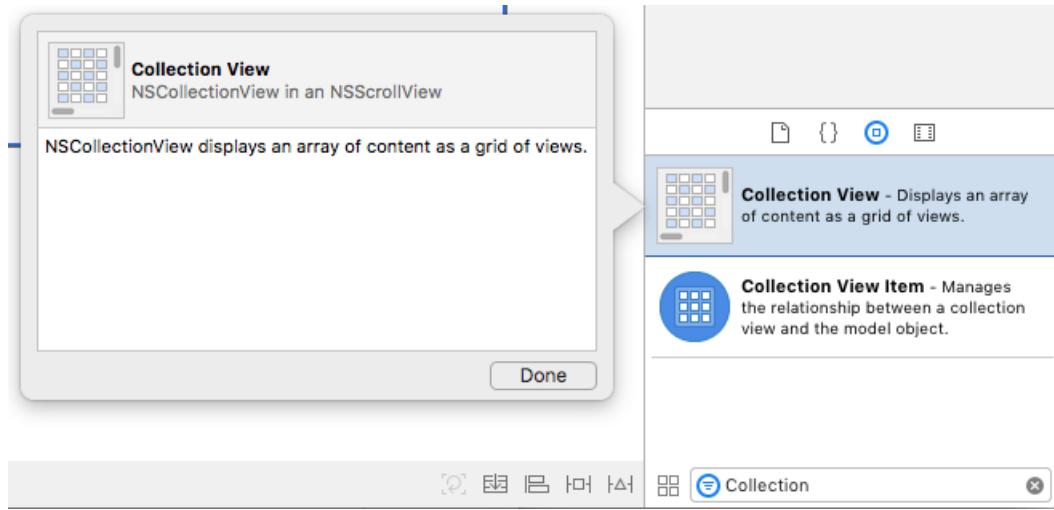
The `ItemSelected` and `ItemsDeselected` methods are overridden and used to set or clear the `PersonSelected` property of the View Controller that is handling the Collection View when the user selects or deselects an item. This will be shown in detail below.

Creating the Collection View in Interface Builder

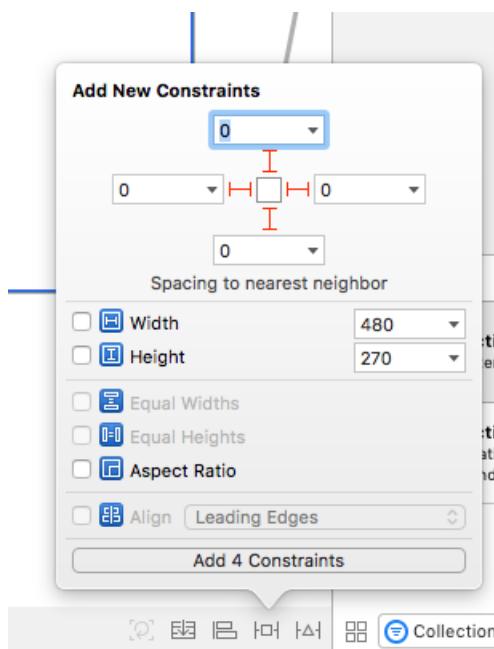
With all of the required supporting pieces in place, the main storyboard can be edited and a Collection View added to it.

Do the following:

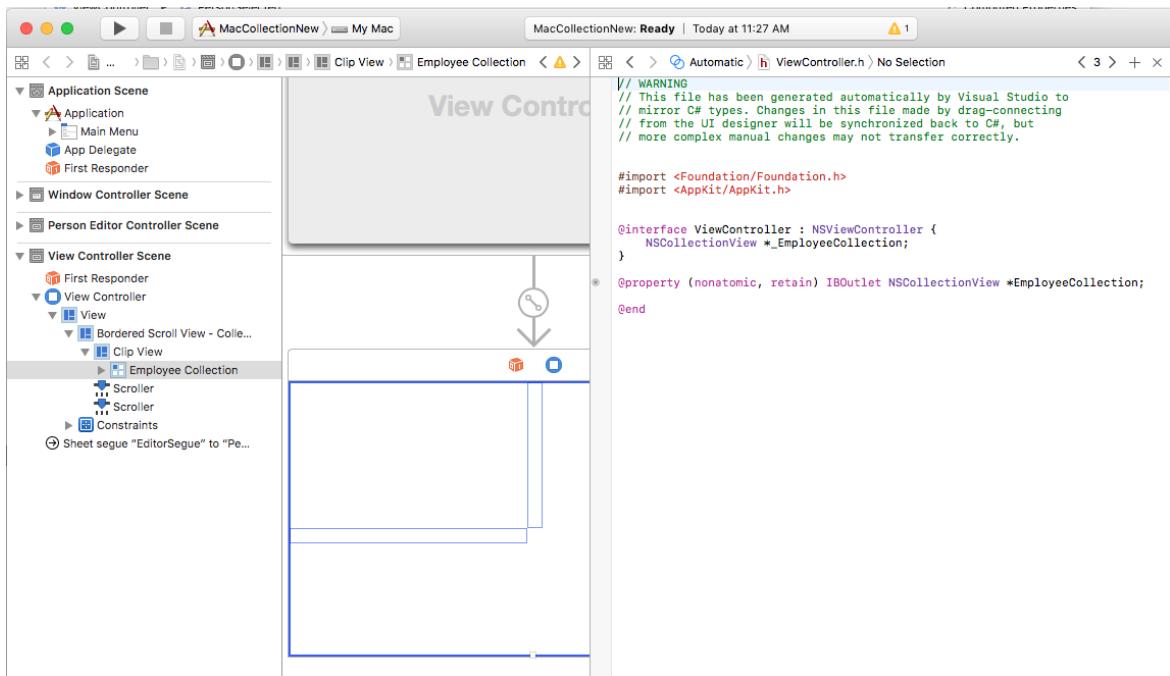
1. Double-click the `MainStoryboard` file in the **Solution Explorer** to open it for editing in Xcode's Interface Builder.
2. Drag a Collection View into the Main View and resize it to fill the View:



3. With the Collection View selected, use the Constraint Editor to pin it to the View when it is resized:



4. Ensure that the Collection View is selected in the Design Surface (and not the Bordered Scroll View or Clip View that contains it), switch to the **Assistant Editor** and create an **Outlet** for the collection view:



5. Save the changes and return to Visual Studio to sync.

Bringing it all Together

All of the supporting pieces have now been put into place with a class to act as the data model (`PersonModel`), a `NSCollectionViewDataSource` has been added to supply data, a `NSCollectionViewDelegateFlowLayout` was created to handle item selection and a `NSCollectionView` was added to the Main Storyboard and exposed as an Outlet (`EmployeeCollection`).

The final step is to edit the View Controller that contains the Collection View and bring all of the pieces together to populate the collection and handle item selection.

Edit the `ViewController.cs` file and make it look like the following:

```

using System;
using AppKit;
using Foundation;
using CoreGraphics;

namespace MacCollectionNew
{
    /// <summary>
    /// The View controller controls the main view that houses the Collection View.
    /// </summary>
    public partial class ViewController : NSViewController
    {
        #region Private Variables
        private PersonModel _personSelected;
        private bool shouldEdit = true;
        #endregion

        #region Computed Properties
        /// <summary>
        /// Gets or sets the datasource that provides the data to display in the
        /// Collection View.
        /// </summary>
        /// <value>The datasource.</value>
        public CollectionViewDataSource Datasource { get; set; }

        /// <summary>
        /// Gets or sets the person currently selected in the collection view.
        /// </summary>
    }
}

```

```

/// </summary>
/// <value>The person selected or <c>null</c> if no person is selected.</value>
[Export("PersonSelected")]
public PersonModel PersonSelected
{
    get { return _personSelected; }
    set
    {
        WillChangeValue("PersonSelected");
        _personSelected = value;
        DidChangeValue("PersonSelected");
        RaiseSelectionChanged();
    }
}
#endregion

#region Constructors
/// <summary>
/// Initializes a new instance of the <see cref="T:MacCollectionNew.ViewController"/> class.
/// </summary>
/// <param name="handle">Handle.</param>
public ViewController(IntPtr handle) : base(handle)
{
}
#endregion

#region Override Methods
/// <summary>
/// Called after the view has finished loading from the Storyboard to allow it to
/// be configured before displaying to the user.
/// </summary>
public override void ViewDidLoad()
{
    base.ViewDidLoad();

    // Initialize Collection View
    ConfigureCollectionView();
    PopulateWithData();
}
#endregion

#region Private Methods
/// <summary>
/// Configures the collection view.
/// </summary>
private void ConfigureCollectionView()
{
    EmployeeCollection.RegisterClassForItem(typeof(EmployeeItemController), "EmployeeCell");

    // Create a flow layout
    var flowLayout = new NSCollectionViewFlowLayout()
    {
        ItemSize = new CGSize(150, 150),
        SectionInset = new NSEdgeInsets(10, 10, 10, 20),
        MinimumInteritemSpacing = 10,
        MinimumLineSpacing = 10
    };
    EmployeeCollection.WantsLayer = true;

    // Setup collection view
    EmployeeCollection.CollectionViewLayout = flowLayout;
    EmployeeCollection.Delegate = new CollectionViewDelegate(this);
}

/// <summary>
/// Populates the Datasource with data and attaches it to the collection view.
/// </summary>
private void PopulateWithData()

```

```

{
    // Make datasource
    Datasource = new CollectionViewDataSource(EmployeeCollection);

    // Build list of employees
    Datasource.Data.Add(new PersonModel("Craig Dunn", "Documentation Manager", true));
    Datasource.Data.Add(new PersonModel("Amy Burns", "Technical Writer"));
    Datasource.Data.Add(new PersonModel("Joel Martinez", "Web & Infrastructure"));
    Datasource.Data.Add(new PersonModel("Kevin Mullins", "Technical Writer"));
    Datasource.Data.Add(new PersonModel("Mark McLemore", "Technical Writer"));
    Datasource.Data.Add(new PersonModel("Tom Opgenorth", "Technical Writer"));
    Datasource.Data.Add(new PersonModel("Larry O'Brien", "API Docs Manager", true));
    Datasource.Data.Add(new PersonModel("Mike Norman", "API Documentor"));

    // Populate collection view
    EmployeeCollection.ReloadData();
}
#endifregion

#region Events
/// <summary>
/// Selection changed delegate.
/// </summary>
public delegate void SelectionChangedDelegate();

/// <summary>
/// Occurs when selection changed.
/// </summary>
public event SelectionChangedDelegate SelectionChanged;

/// <summary>
/// Raises the selection changed event.
/// </summary>
internal void RaiseSelectionChanged() {
    // Inform caller
    if (this.SelectionChanged != null) SelectionChanged();
}
#endifregion
}

```

Taking a look at this code in detail, a `Datasource` property is defined to hold an instance of the `CollectionViewDataSource` that will provide the data for the Collection View. A `PersonSelected` property is defined to hold the `PersonModel` representing the currently selected item in the Collection View. This property also raises the `SelectionChanged` event when the selection changes.

The `ConfigureCollectionView` class is used to register the View Controller that acts as the cell prototype with the Collection View using the following line:

```
EmployeeCollection.RegisterClassForItem(typeof(EmployeeItemController), "EmployeeCell");
```

Notice that the `Identifier` (`EmployeeCell`) used to register the prototype matches the one called in the `GetItem` method of the `CollectionViewDataSource` defined above:

```
var item = collectionView.MakeItem("EmployeeCell", indexPath) as EmployeeItemController;
...
```

Additionally, the type of the View Controller **must** match the name of the `.xib` file that defines the prototype **exactly**. In the case of this example, `EmployeeItemController` and `EmployeeItemController.xib`.

The actual layout of the items in the Collection View is controlled by a Collection View Layout class and can be

changed dynamically at runtime by assigning a new instance to the `CollectionViewLayout` property. Changing this property updates the Collection View appearance without animating the change.

Apple ships two built-in layout types with the Collection View that will handle most typical uses:

`NSCollectionViewFlowLayout` and `NSCollectionViewGridLayout`. If the developer required a custom format, such as laying the items out in a circle, they can create a custom instance of `NSCollectionViewLayout` and override the required methods to achieve the desired effect.

This example uses the default flow layout so it creates an instance of the `NSCollectionViewFlowLayout` class and configures it as follows:

```
var flowLayout = new NSCollectionViewFlowLayout()
{
    ItemSize = new CGSize(150, 150),
    SectionInset = new NSEdgeInsets(10, 10, 10, 20),
    MinimumInteritemSpacing = 10,
    MinimumLineSpacing = 10
};
```

The `ItemSize` property defines the size of each individual cell in the collection. The `SectionInset` property defines the insets from the edge of the collection that cells will be laid out in. `MinimumInteritemSpacing` defines the minimum spacing between items and `MinimumLineSpacing` defines the minimum spacing between lines in the collection.

The layout is assigned to the Collection View and an instance of the `CollectionViewDelegate` is attached to handle item selection:

```
// Setup collection view
EmployeeCollection.CollectionViewLayout = flowLayout;
EmployeeCollection.Delegate = new CollectionViewDelegate(this);
```

The `PopulateWithData` method creates a new instance of the `CollectionViewDataSource`, populates it with data, attaches it to the Collection View and calls the `ReloadData` method to display the items:

```
private void PopulateWithData()
{
    // Make datasource
    Datasource = new CollectionViewDataSource(EmployeeCollection);

    // Build list of employees
    Datasource.Data.Add(new PersonModel("Craig Dunn", "Documentation Manager", true));
    ...

    // Populate collection view
    EmployeeCollection.ReloadData();
}
```

The `ViewDidLoad` method is overridden and calls the `ConfigureCollectionView` and `PopulateWithData` methods to display the final Collection View to the user:

```
public override void ViewDidLoad()
{
    base.ViewDidLoad();

    // Initialize Collection View
    ConfigureCollectionView();
    PopulateWithData();
}
```

Summary

This article has taken a detailed look at working with Collection Views in a Xamarin.Mac application. First, it looked at exposing a C# class to Objective-C by using Key-Value Coding (KVC) and Key-Value Observing (KVO). Next, it showed how to use a KVO compliant class and Data Bind it to Collection Views in Xcode's Interface Builder. Finally, it showed how to interact with Collection Views in C# code.

Related Links

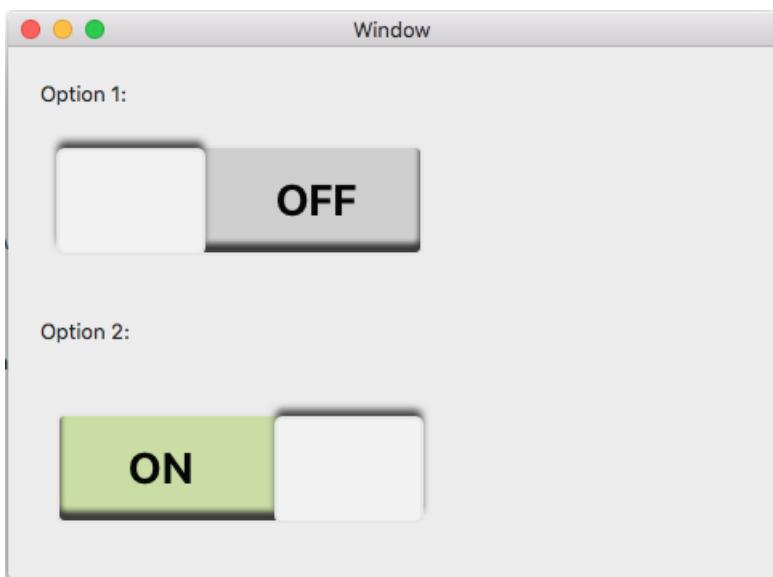
- [MacCollectionNew \(sample\)](#)
- [Hello, Mac](#)
- [Data Binding and Key-Value Coding](#)
- [NSCollectionView](#)
- [OS X Human Interface Guidelines](#)

Creating Custom Controls in Xamarin.Mac

11/2/2020 • 9 minutes to read • [Edit Online](#)

When working with C# and .NET in a Xamarin.Mac application, you have access to the same User Controls that a developer working in *Objective-C*, *Swift* and *Xcode* does. Because Xamarin.Mac integrates directly with Xcode, you can use Xcode's *Interface Builder* to create and maintain your User Controls (or optionally create them directly in C# code).

While macOS provides a wealth of built-in User Controls, there might be times that you need to create a custom control to provide functionality not provided out-of-the-box or to match a custom UI theme (such as a game interface).



In this article, we'll cover the basics of creating a reusable Custom User Interface Control in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` commands used to wire-up your C# classes to Objective-C objects and UI Elements.

Introduction to Custom Controls

As stated above, there might be times when you need to create a reusable, Custom User Interface Control to provide unique functionality for your Xamarin.Mac app's UI or to create a custom UI theme (such as a game interface).

In these situations, you can easily inherit from `NSControl` and create a custom tool that can either be added to your app's UI via C# code or through Xcode's Interface Builder. By inheriting from `NSControl` your custom control will automatically have all of the standard features that a built-in User Interface Control has (such as `NSButton`).

If your custom User Interface control just displays information (like a custom charting and graphic tool), you might want to inherit from `NSView` instead of `NSControl`.

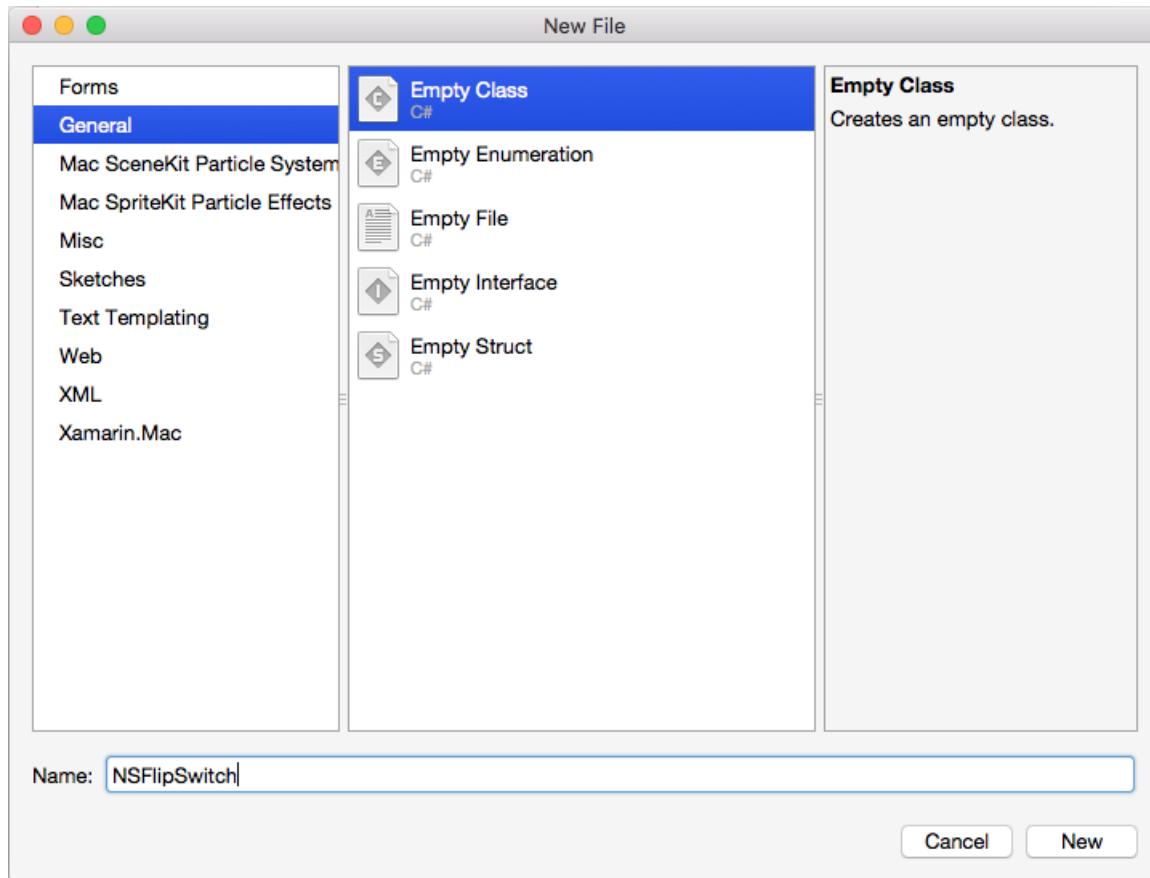
No matter which base class is used, the basic steps for creating a custom control is the same.

In this article will, create a custom Flip Switch component that provides a unique User Interface Theme and an example of building a fully functional Custom User Interface Control.

Building the Custom Control

Since the custom control we are creating will be responding to user input (left mouse button clicks), we are going to inherit from `NSControl`. In this way, our custom control will automatically have all of the standard features that a built-in User Interface Control has and respond like a standard macOS control.

In Visual Studio for Mac, open the `Xamarin.Mac` project that you want to create a Custom User Interface Control for (or create a new one). Add a new class and call it `NSFlipSwitch`:



Next, edit the `NSFlipSwitch.cs` class and make it look like the following:

```
using Foundation;
using System;
using System.CodeDom.Compiler;
using AppKit;
using CoreGraphics;

namespace MacCustomControl
{
    [Register("NSFlipSwitch")]
    public class NSFlipSwitch : NSControl
    {
        #region Private Variables
        private bool _value = false;
        #endregion

        #region Computed Properties
        public bool Value {
            get { return _value; }
            set { }
        }
    }
}
```

```

    set _value;
    // Save value and force a redraw
    _value = value;
    NeedsDisplay = true;
}
}

#endregion

#region Constructors
public NSFlipSwitch ()
{
    // Init
    Initialize();
}

public NSFlipSwitch (IntPtr handle) : base (handle)
{
    // Init
    Initialize();
}

[Export ("initWithFrame")]
public NSFlipSwitch (CGRect frameRect) : base(frameRect) {
    // Init
    Initialize();
}

private void Initialize() {
    this.WantsLayer = true;
    this.LayerContentsRedrawPolicy = NSViewLayerContentsRedrawPolicy.OnSetNeedsDisplay;
}
#endregion

#region Draw Methods
public override void DrawRect (CGRect dirtyRect)
{
    base.DrawRect (dirtyRect);

    // Use Core Graphic routines to draw our UI
    ...
}

#endregion

#region Private Methods
private void FlipSwitchState() {
    // Update state
    Value = !Value;
}
#endregion

}
}

```

The first thing to notice about our custom class is that we are inheriting from `NSControl` and using the `Register` command to expose this class to Objective-C and Xcode's Interface Builder:

```

[Register("NSFlipSwitch")]
public class NSFlipSwitch : NSControl

```

In the following sections, we'll take a look at the rest of the above code in detail.

Tracking the Control's State

Since our Custom Control is a switch, we need a way to track the On/Off state of the switch. We handle that with

the following code in `NSFlipSwitch`:

```
private bool _value = false;
...
public bool Value {
    get { return _value; }
    set {
        // Save value and force a redraw
        _value = value;
        NeedsDisplay = true;
    }
}
```

When the state of the switch changes, we need a way to update the UI. We do that by forcing the control to redraw its UI with `NeedsDisplay = true`.

If our control required more than a single On/Off state (for example a multi-state switch with 3 positions), we could have used an `Enum` to track the state. For our example, a simple `bool` will do.

We also added a helper method to swap the state of the switch between On and Off:

```
private void FlipSwitchState() {
    // Update state
    Value = !Value;
}
```

Later, we'll expand this helper class to inform the caller when the switch's state has changed.

Drawing the Control's Interface

We are going to use Core Graphic drawing routines to draw our custom control's User Interface at runtime. Before we can do this, we need to turn on layers for our control. We do this with the following private method:

```
private void Initialize() {
    this.WantsLayer = true;
    this.LayerContentsRedrawPolicy = NSViewLayerContentsRedrawPolicy.OnSetNeedsDisplay;
}
```

This method gets called from each of the control's constructors to ensure that the control is properly configured. For example:

```
public NSFlipSwitch (IntPtr handle) : base (handle)
{
    // Init
    Initialize();
}
```

Next, we need to override the `DrawRect` method and add the Core Graphic routines to draw the control:

```

public override void DrawRect (CGRect dirtyRect)
{
    base.DrawRect (dirtyRect);

    // Use Core Graphic routines to draw our UI
    ...

}

```

We'll be adjusting the visual representation for the control when its state changes (such as going from **On** to **Off**). Any time the state changes, we can use the `NeedsDisplay = true` command to force the control to redraw for the new state.

Responding to User Input

There are two basic way that we can add user input to our custom control: **Override Mouse Handling Routines or Gesture Recognizers**. Which method we use, will be based on the functionality required by our control.

IMPORTANT

For any custom control you create, you should use either **Override Methods** *or* **Gesture Recognizers**, but not both at the same time as they can conflict with each other.

Handling User Input with Override Methods

Objects that inherit from `NSControl` (or `NSView`) have several override methods for handling mouse or keyboard input. For our example control, we want to flip the state of the switch between **On** and **Off** when the user clicks on the control with the left mouse button. We can add the following override methods to the `NSFlipSwitch` class to handle this:

```

#region Mouse Handling Methods
// -----
// Handle mouse with Override Methods.
// NOTE: Use either this method or Gesture Recognizers, NOT both!
// -----
public override void MouseDown (NSEvent theEvent)
{
    base.MouseDown (theEvent);

    FlipSwitchState ();
}

public override void MouseDragged (NSEvent theEvent)
{
    base.MouseDragged (theEvent);
}

public override void MouseUp (NSEvent theEvent)
{
    base.MouseUp (theEvent);
}

public override void MouseMoved (NSEvent theEvent)
{
    base.MouseMoved (theEvent);
}
##endregion

```

In the above code, we call the `FlipSwitchState` method (defined above) to flip the On/Off state of the switch in

the `MouseDown` method. This will also force the control to be redrawn to reflect the current state.

Handling User Input with Gesture Recognizers

Optionally, you can use Gesture Recognizers to handle the user interacting with the control. Remove the overrides added above, edit the `Initialize` method and make it look like the following:

```
private void Initialize() {
    this.WantsLayer = true;
    this.LayerContentsRedrawPolicy = NSViewLayerContentsRedrawPolicy.OnSetNeedsDisplay;

    // -----
    // Handle mouse with Gesture Recognizers.
    // NOTE: Use either this method or the Override Methods, NOT both!
    // -----
    var click = new NSClickGestureRecognizer (() => {
        FlipSwitchState();
    });
    AddGestureRecognizer (click);
}
```

Here, we are creating a new `NSClickGestureRecognizer` and calling our `FlipSwitchState` method to change the switch's state when the user clicks on it with the left mouse button. The `AddGestureRecognizer (click)` method adds the Gesture Recognizer to the control.

Again, which method we use depends on what we are trying to accomplish with our custom control. If we need low level access to the user interaction, use the Override Methods. If we need predefined functionality, such as mouse clicks, use Gesture Recognizers.

Responding to State Change Events

When the user changes the state of our custom control, we need a way to respond to the state change in code (such as doing something when clicks on a custom button).

To provide this functionality, edit the `NSFlipSwitch` class and add the following code:

```
#region Events
public event EventHandler ValueChanged;

internal void RaiseValueChanged() {
    if (this.ValueChanged != null)
        this.ValueChanged (this, EventArgs.Empty);

    // Perform any action bound to the control from Interface Builder
    // via an Action.
    if (this.Action != null)
        NSApplication.SharedApplication.SendAction (this.Action, this.Target, this);
}

## endregion
```

Next, edit the `FlipSwitchState` method and make it look like the following:

```
private void FlipSwitchState() {
    // Update state
    Value = !Value;
    RaiseValueChanged ();
}
```

First, we provide a `ValueChanged` event that we can add a handler to in C# code so that we can perform an action when the user changes the state of the switch.

Second, because our custom control inherits from `NSControl`, it automatically has an **Action** that can be assigned in Xcode's Interface Builder. To call this **Action** when the state changes, we use the following code:

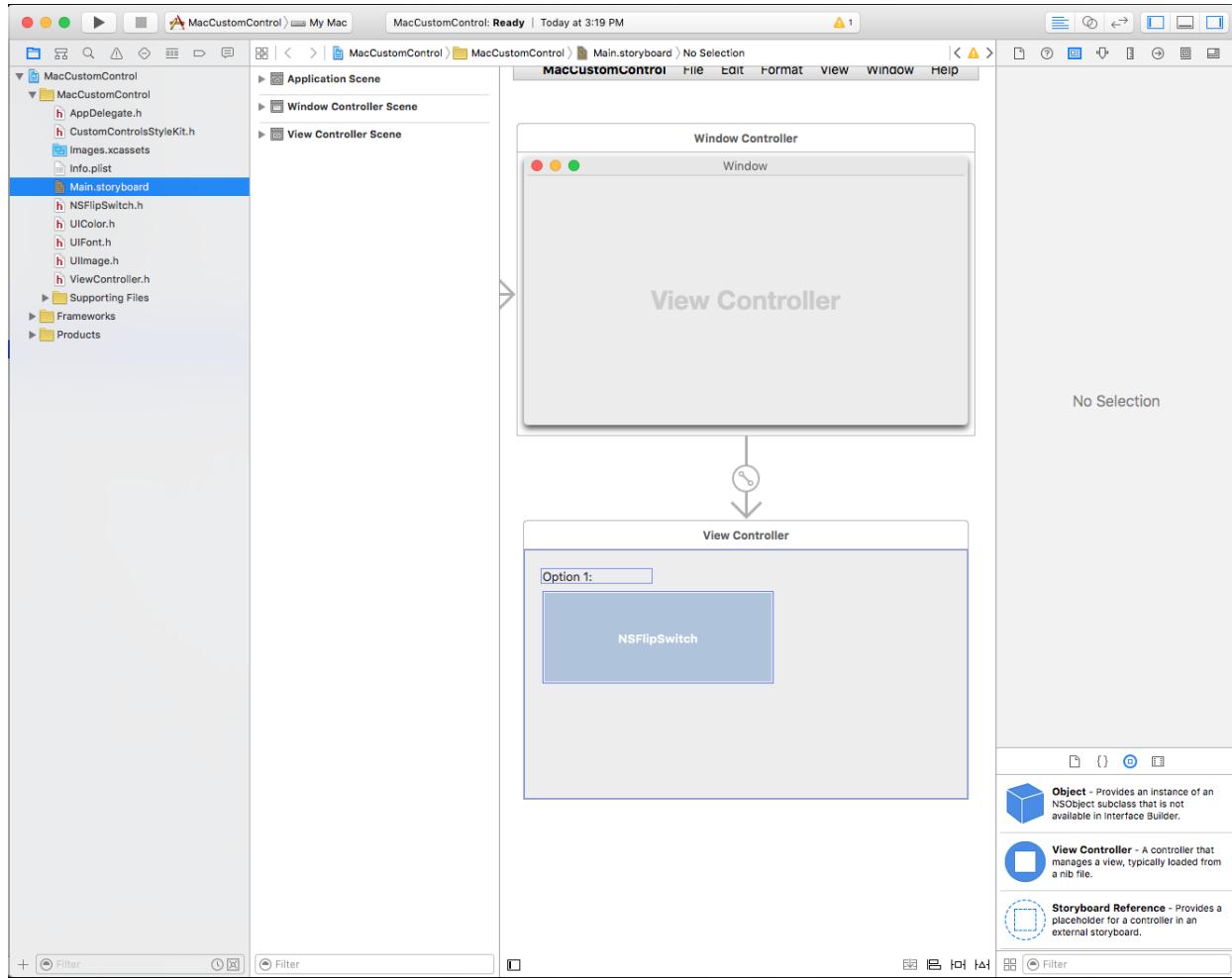
```
if (this.Action != null)
    NSApplication.SharedApplication.SendAction (this.Action, this.Target, this);
```

First, we check to see if an **Action** has been assigned to the control. Next, we call the **Action** if it has been defined.

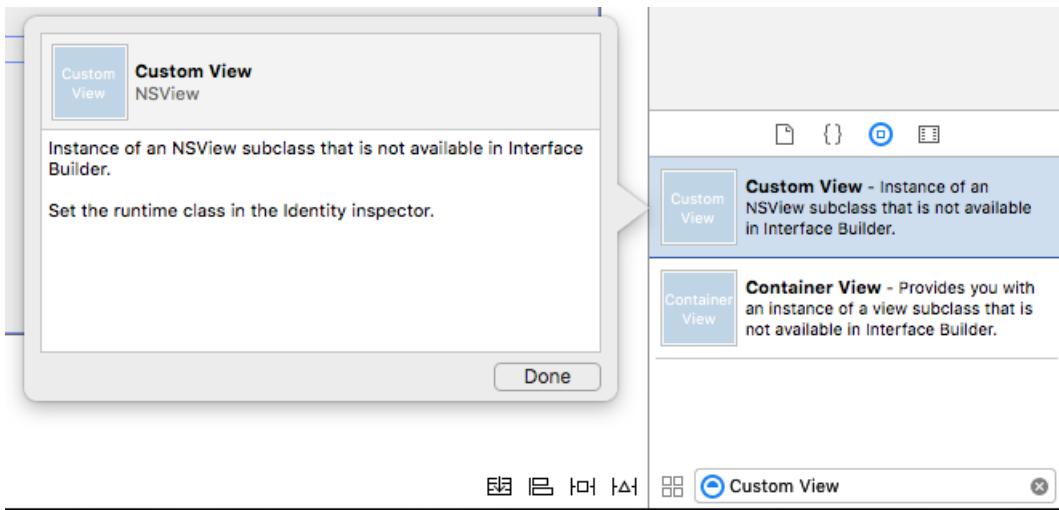
Using the Custom Control

With our custom control fully defined, we can either add it to our Xamarin.Mac app's UI using C# code or in Xcode's Interface Builder.

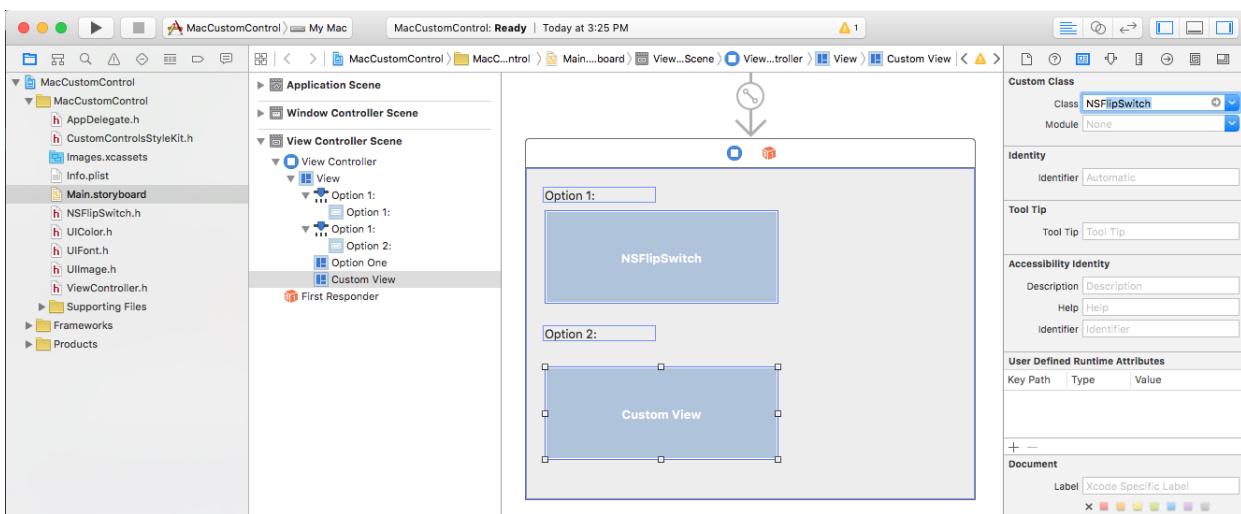
To add the control using Interface Builder, first do a clean build of the Xamarin.Mac project, then double-click the `Main.storyboard` file to open it in Interface Builder for edit:



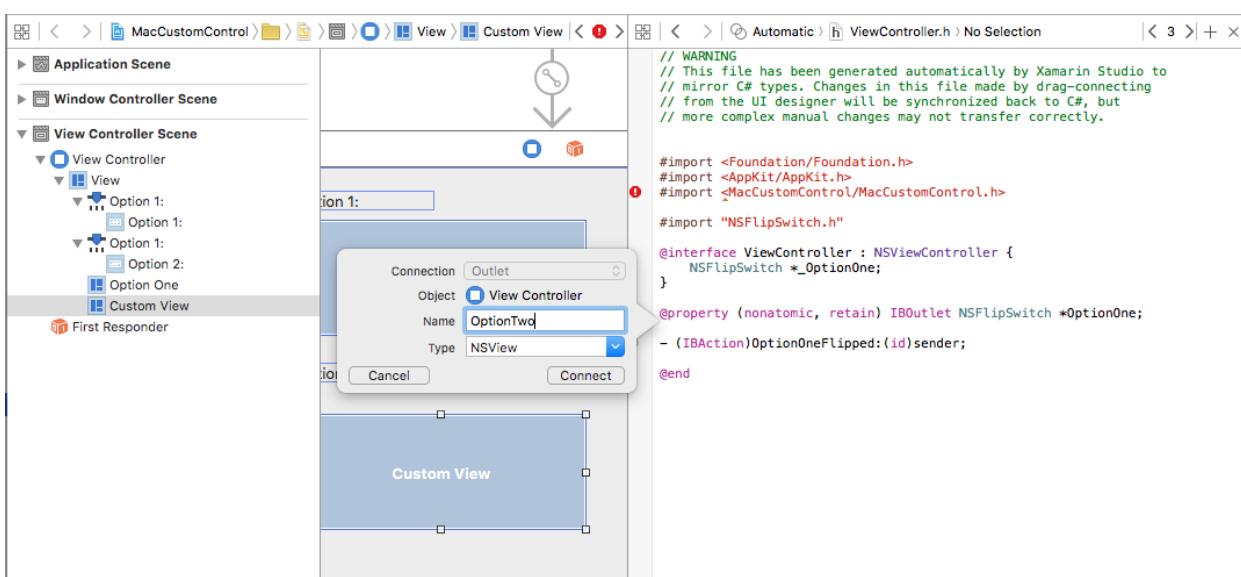
Next, drag a `Custom View` into the User Interface design:



With the Custom View still selected, switch to the **Identity Inspector** and change the view's **Class** to **NSFlipSwitch**:



Switch to the **Assistant Editor** and create an **Outlet** for the custom control (making sure to bind it in the **ViewController.h** file and not the **.m** file):



Save your changes, return to Visual Studio for Mac and allow the changes to sync. Edit the **ViewController.cs** file and make the **ViewDidLoad** method look like the following:

```

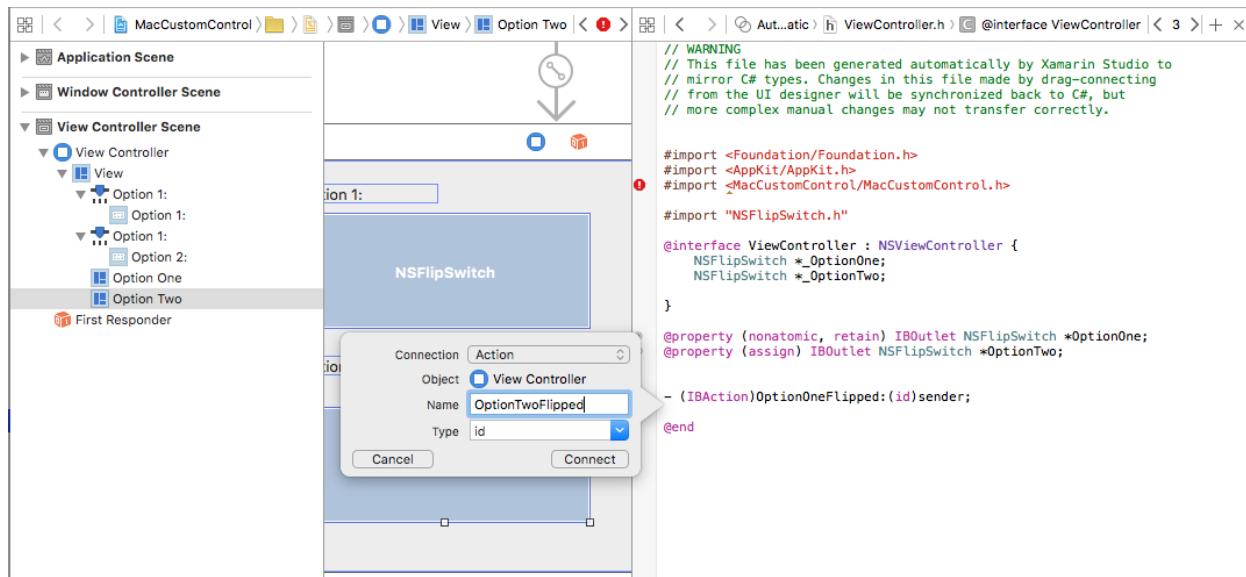
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();

    // Do any additional setup after loading the view.
    OptionTwo.ValueChanged += (sender, e) => {
        // Display the state of the option switch
        Console.WriteLine("Option Two: {0}", OptionTwo.Value);
    };
}

```

Here, we respond to the `ValueChanged` event we defined above on the `NSFlipSwitch` class and write out the current **Value** when the user clicks on the control.

Optionally, we could return to Interface Builder and define an **Action** on the control:



Again, edit the `ViewController.cs` file and add the following method:

```

partial void OptionTwoFlipped (Foundation.NSObject sender) {
    // Display the state of the option switch
    Console.WriteLine("Option Two: {0}", OptionTwo.Value);
}

```

IMPORTANT

You should use either the **Event** or define an **Action** in Interface Builder, but you should not use both methods at the same time or they can conflict with each other.

Summary

This article has taken a detailed look at creating a reusable Custom User Interface Control in a Xamarin.Mac application. We saw how to draw the custom controls UI, the two main ways to respond to mouse and user input and how to expose the new control to Actions in Xcode's Interface Builder.

Related Links

- [MacCustomControl \(sample\)](#)
- [Hello, Mac](#)

- Data Binding and Key-Value Coding
- OS X Human Interface Guidelines
- Handling Mouse Events

macOS platform features

10/28/2019 • 2 minutes to read • [Edit Online](#)

Documents in this section cover working with key, platform-specific features of macOS in a Xamarin.Mac application.

Introduction to macOS Mojave

This document provides a high-level overview of new and updated features in macOS Mojave that are available for use when building Xamarin.Mac applications.

Introduction to macOS High Sierra

This document describes new and enhanced features in macOS High Sierra.

Introduction to macOS Sierra

macOS Sierra is the latest incarnation of Apple's desktop operating system for Mac. This document covers the changes from Mac OS X El Capitan and how to implement them in a Xamarin.Mac app.

Binding Objective-C libraries for Mac

Learn how to bind Objective-C Mac libraries for use in Xamarin.Mac projects. Refer to the [Troubleshooting page](#) to resolve any issues.

Introduction to OpenTK

OpenTK (The Open Toolkit) is an advanced, low-level C# library that makes working with OpenGL, OpenCL and OpenAL easier. OpenTK can be used for games, scientific applications or other projects that require 3D graphics, audio or computational functionality. This article gives a brief introduction to using OpenTK in a Xamarin.Mac app.

Introduction to storyboards

Storyboards allow you to develop a User Interface for your Xamarin.Mac app that not only includes the window definitions and controls, but also contains the links between different windows (via segues) and view states. This article gives an introduction to using Storyboards in a Xamarin.Mac app.

Xamarin.Mac extension support

In Xamarin.Mac 2.10 preview support was added for multiple macOS extension points:

- Finder
- Share
- Today

Target frameworks

This article covers the types of Target Frameworks (Base Class Libraries) that are available in Xamarin.Mac and the implications of selecting a specific target for your Xamarin.Mac application.

Introduction to macOS Mojave

11/2/2020 • 2 minutes to read • [Edit Online](#)

This document provides a high-level overview of new and updated features in macOS Mojave.

To get started building macOS Mojave apps with Xamarin, take a look at the [getting started guide for Xamarin.Mac 5.0](#).

Dark Mode

Dark mode is a system-wide dark theme in macOS Mojave that uses a dynamic, dark grey color scheme to display user interface elements. It also introduces new accent colors, color effects, and content tint colors to help third-party apps look good no matter the user's color settings.

User Notifications framework

The User Notifications framework is included in macOS Mojave, changing the APIs that Mac apps use to work with user notifications.

Natural Language framework

The Natural Language framework enables applications to perform various types of language analysis. For example, it can be used to identify parts of speech and determine the language represented by a block of text.

Vision framework

The Vision framework includes an improved face detector that can detect faces in various orientations. Also, request revisions can now be used to select a specific Vision framework algorithm revision.

Network framework

Network framework, the network stack underlying the `NSURLSession` APIs commonly used in iOS applications, is now available as a standalone framework, making it easier to work with TCP, UDP, TLS, IPv4/IPv6, and more.

Deprecations

With macOS Mojave, Apple has deprecated OpenGL ES and OpenCL, [encouraging developers](#) to adopt Metal and Metal Performance Shaders.

Related links

- [Xamarin.Mac samples](#)
- [macOS – Apple Developer](#)
- [Xamarin.Mac 5.0 release notes](#)

Get started with macOS Mojave

11/2/2020 • 2 minutes to read • [Edit Online](#)

This document describes how to get set up to build macOS Mojave apps with Xamarin.Mac. It discusses how to download Xcode 10 and update Visual Studio for Mac.

Download and install

1. **Install the latest Xcode 10 beta** – Registered Apple developers can download and install the latest version of Xcode 10 from the [Apple Developer Portal](#).
2. **Run Xcode 10** – Run Xcode 10 before updating and running Visual Studio for Mac; it installs some tools that Xamarin requires.
3. **Update Visual Studio for Mac** – Use the latest stable version of Visual Studio for Mac, with [Xamarin.Mac 5.0](#) or newer.
4. *(optional)* **Install the macOS Mojave on your Mac** –

TIP

Even if your app does not use any new macOS Mojave APIs, be sure to build it with the macOS Mojave SDK and test it to make sure that everything works as expected. If an app doesn't call any new APIs, you can recompile it with the macOS Mojave SDK and test it without upgrading your Mac's operating system.

Before upgrading your Mac to macOS Mojave to build and test Xamarin.Mac applications that call new macOS Mojave APIs:

- Read [Apple's release notes](#) for the operating system update.

Related links

- [Download Xcode 10](#)
- [Xamarin.Mac 5.0 release notes](#)

Introduction to macOS High Sierra

10/28/2019 • 2 minutes to read • [Edit Online](#)

macOS High Sierra introduced new features and updates to macOS, including:

- HEVC (High Efficiency Video Coding)
- Apple File System
- Metal 2

Related links

- [macOS High Sierra](#)

Introduction to macOS Sierra

11/2/2020 • 6 minutes to read • [Edit Online](#)

With the new macOS Sierra, the developer can take advantage of new APIs that allow the end user to interact with their apps and websites in previously unavailable ways. For example, Apple now allows websites to give customers the option of paying securely via Apple Pay and enhancements to the Metal framework boost an app's graphics and computing potential.

For more information on macOS Sierra, please see Apple's [macOS + Apps](#) documentation.

What's New in macOS Sierra

Apple has added several new APIs and services in macOS Sierra along with many enhancements to existing features, including:

Apple File System

With macOS Sierra, Apple has released the new Apple File System as a modern file system for iOS, macOS, tvOS and watchOS. The Apple File System was optimized for Flash and SSD storage and provides the following features: strong encryption, copy-on-write metadata, space sharing, cloning for files and directories, snapshots, fast directory sizing and atomic safe-save primitives.

For more information, please see Apple's [Apple File System Guide](#).

Apple Pay Enhancements

Apple has made several enhancements to Apple Pay in macOS Sierra that allow the user to make secure payments from websites.

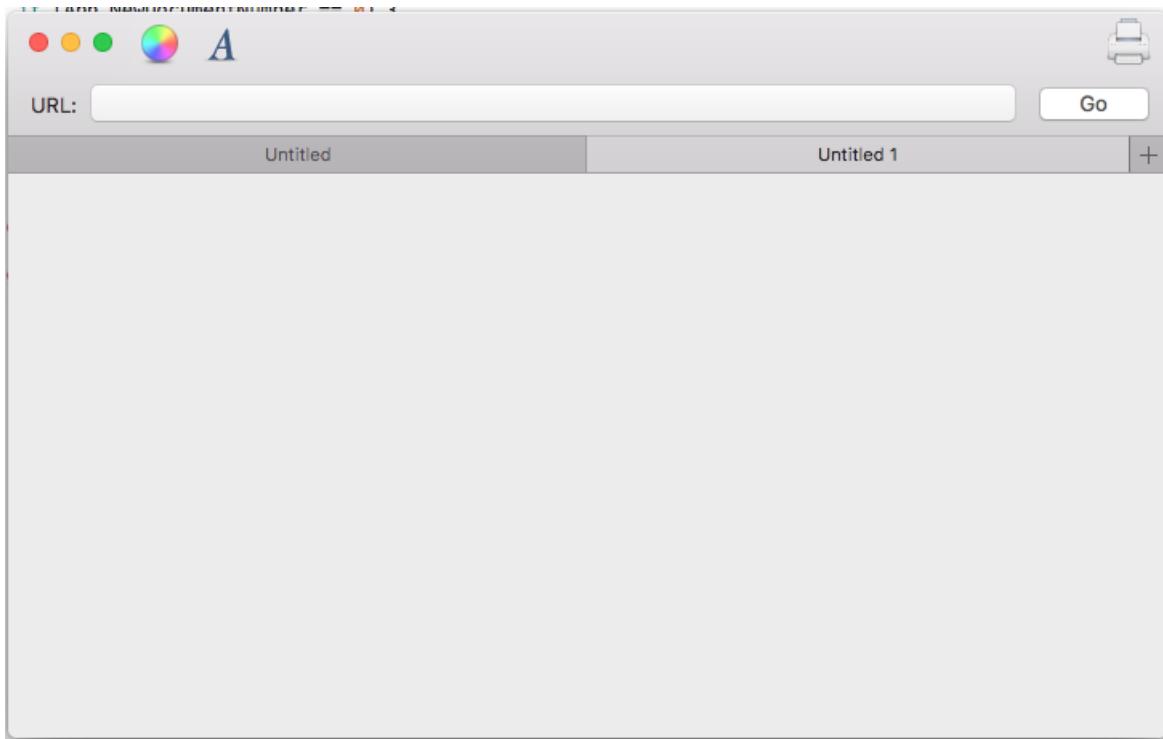
With macOS Sierra, several new APIs have been added that work with macOS Sierra, iOS and watchOS to support dynamic payment networks and a new sandbox test environment.

macOS Sierra includes the new ApplePay Javascript framework that allows the developer to incorporate Apple Pay directly into iOS and macOS Safari-based websites. For websites that support Apple Pay, the user can authorize payment using either their iPhone or Apple Watch.

For more information, please see Apple's [ApplePay JS Framework](#) reference.

Building Modern macOS Apps

Modern macOS apps such as Apple's Safari web browser, Pages word processor and Numbers spread sheet use many new technologies to present a unified, context sensitive User Interface that does away with traditional UI elements such as floating panels and multiple open windows.



Our [Building Modern macOS Apps](#) guide covers several tips, features and techniques a developer can use to build a modern macOS app in Xamarin.Mac.

CloudKit Data Sharing

The CloudKit framework has been expanded in macOS Sierra to allow user to quickly and easily share records or record sets from their private iCloud databases.

CloudKit provides a complete UI for sending and accepting shared record invitations and the user has complete read/write control over the people that have access to the records.

For more information, please see Apple's [CloudKit Framework Reference](#) and [CloudKit JS Framework Reference](#).

IMPORTANT

Apple [provides tools](#) to help developers properly handle the European Union's General Data Protection Regulation (GDPR).

Safari App Extensions Support

Safari App Extensions allow the app to extend the behavior of the Safari web browser while being tightly integrated with macOS Sierra. Since macOS Safari App Extensions work similar to iOS Safari App Extensions, they are easy to port from one system to another.

For more information, please see Apple's [Safari App Extension Programming Guide](#).

Security and Privacy Enhancements

Apple has made several enhancements to both security and privacy in macOS Sierra that will help the app improve the security of the app and ensure the end user's privacy including the following:

- The new `NSAllowsArbitraryLoadsInWebContent` key can be add to the app's `Info.plist` file and will allow web pages to load correctly while Apple Transport Security (ATS) protection is still enabled for the rest of the app.
- The Common Data Security Architecture (CDSA) API has been deprecated and should be replaced with the SecKey API to generate asymmetric keys.
- For all SSL/TLS connections, the RC4 symmetric cipher is now disabled by default. Additionally, the Secure

Transport API no longer supports SSLv3 and it is recommended that the app stop using SHA-1 and 3DES cryptography as soon as possible.

- Because the new Clipboard in iOS 10 and macOS Sierra allows the user to copy and paste between devices, the API has been expanded to allow a clipboard to be limited to a specific device and be timestamped to be cleared automatically at a given point. Additionally, named pasteboards are no longer persisted and should be replaced with the shared pasteboard containers.
- If the app accesses protected data (such as the user's Calendar), it *must* declare that intent with the correct purpose string value key in its `Info.plist` file (`NSCalendarUsageDescription` in the case of the Calendar).
- Developer Signed apps that are not delivered via the Mac App Store can now take advantage of CloudKit, iCloud Keychain, iCloud Drive, remote push notifications, MapKit and VPN entitlements.
- macOS Sierra no longer supports delivering external code or data along with the code-signer app in its zip archive or unsigned disk image as the runtime path is not known before runtime.

Additionally, apps running on macOS Sierra (or later) must statically declare their intent to access specific features or user information by entering one or more Privacy Specific Keys in their `Info.plist` files that explain to the user why the app wishes to gain access.

Since macOS Sierra shares these changes with iOS 10, please see our [iOS 10 Security and Privacy Enhancements](#) guide for more information.

Smart Card Driver Extension Support

With macOS Sierra, the app can create `NSExtension` based smart card drivers that allows read-only access to the content from certain types of smart cards. This information is then presented inside the system keychain (replacing the deprecated Common Data Security Architecture method).

for more information, Please see Apple's [CryptoTokenKit Framework Reference](#).

Unified Logging

Unified Logging provides the app with a single API for efficient messaging across all levels of the system. With Unified Logging the app has fine-grained control over multiple levels of logging that include privacy controls and activity tracking for easier debugging.

Logging provides automatic message correlation when activity tracking and logging are used together.

macOS Sierra includes a new Console App (in Applications/Utilities) that is able to display log data from multiple sources including connected devices. It also supports tokenized and saved searches and displays connections between related messages across multiple processes.

Additionally, log messages can be viewed and maintained using command line tools.

For more information, please see Apple's [Logging Reference](#).

Wide Color

macOS Sierra extends the support for extended-range pixel formats and wide-gamut color spaces throughout the system including frameworks such as Core Graphics, Core Image, Metal and AVFoundation. Support for devices with wide color displays is further eased by providing this behavior throughout the entire graphics stack.

Additionally, `AppKit` has been modified to work in the new extended sRGB colorspace, making it easier to mix colors in wide color gamuts without significant performance loss.

Apple offers the following best practices when working with wide colors:

- `NSColor` now uses the sRGB color space and will no longer clamp values to the `0.0` to `1.0` range. If the app relies on the previous clamp behavior, it will need to be modified for macOS Sierra.

- When using a low-level API such as Core Graphics or Metal to provide image processing, the app should use an extended range color space and pixel format that supports 16-bit floating point values. Where necessary, the app will have to manually clamp color component values.
- Core Graphics, Core Image and Metal Performance Shaders all provide new methods for converting between the two color spaces.

To find out more, please see our [Introduction to Wide Color](#) guide.

Additional Framework Changes

In addition to the major framework changes and additions listed above, Apple has made many additional minor framework changes in macOS Sierra.

To find out more, please see our [Additional Framework Changes](#) guide.

Deprecated APIs

The following APIs have been deprecated in macOS Sierra:

- The HFS Standard File System is no longer supported.

See Apple's [macOS v10.12 API Diffs](#) documentation for a complete list of deprecations and changes.

Related Links

- [Mac Samples](#)
- [What's new in macOS 10.12](#)

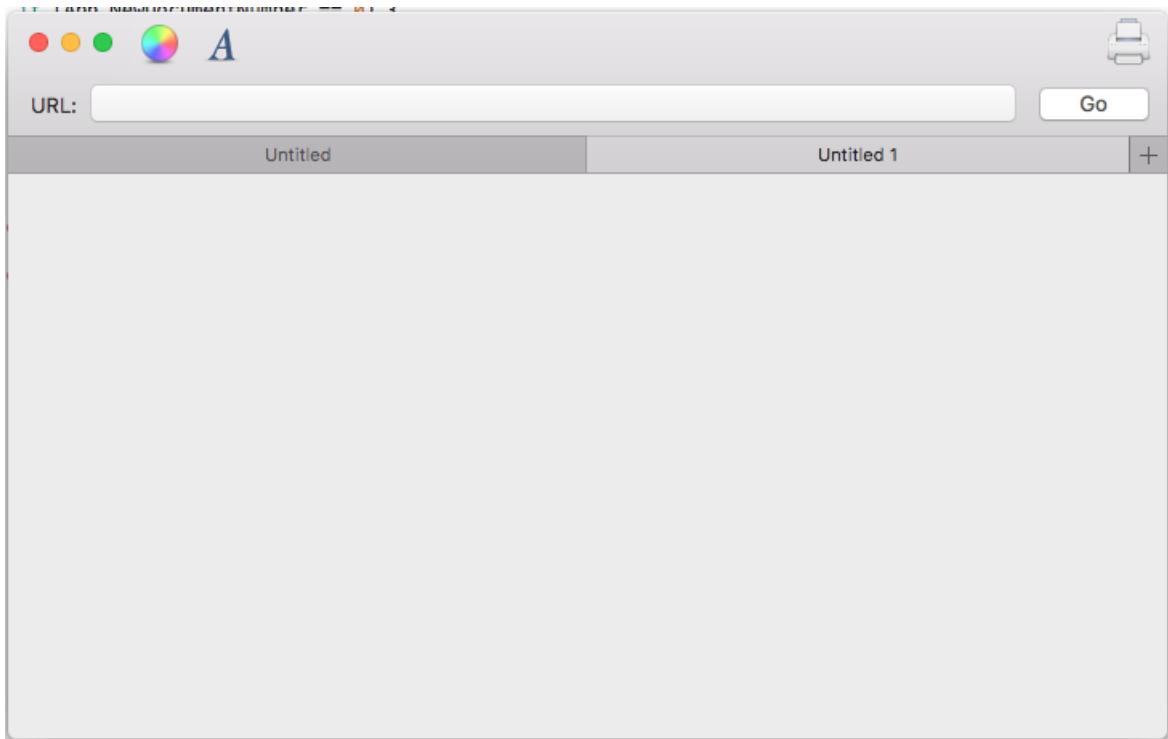
Building Modern macOS Apps

11/2/2020 • 24 minutes to read • [Edit Online](#)

This article covers several tips, features and techniques a developer can use to build a modern macOS app in Xamarin.Mac.

Building Modern Looks with Modern Views

A modern look will include a modern Window and Toolbar appearance such as the example app shown below:



Enabling Full Sized Content Views

To achieve this looks in a Xamarin.Mac app, the developer will want to use a *Full Size Content View*, meaning the content extends under the Tool and Title Bar areas and will be automatically blurred by macOS.

To enable this feature in code, create a custom class for the `NSWindowController` and make it look like the following:

```

using System;
using Foundation;
using AppKit;

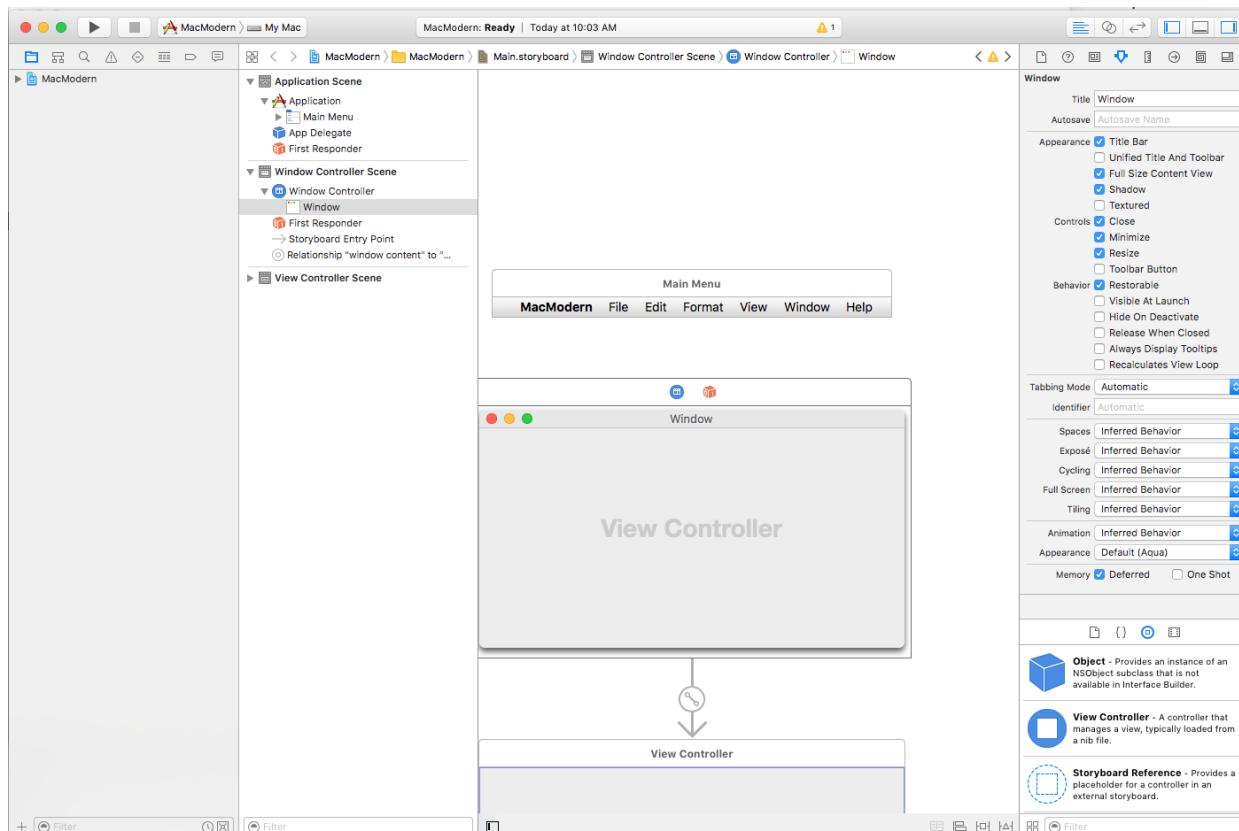
namespace MacModern
{
    public partial class MainWindowController : NSWindowController
    {
        #region Constructor
        public MainWindowController (IntPtr handle) : base (handle)
        {
        }
        #endregion

        #region Override Methods
        public override void WindowDidLoad ()
        {
            base.WindowDidLoad ();

            // Set window to use Full Size Content View
            Window.StyleMask = NSWindowStyle.FullSizeContentView;
        }
        #endregion
    }
}

```

This feature can also be enabled in Xcode's Interface Builder by selecting the Window and checking **Full Sized Content View**:



When using a Full Size Content View, the developer may need to offset the content beneath the title and tool bar areas so that specific content (such as labels) doesn't slide under them.

To complicate this issue, the Title and Tool Bar areas can have a dynamic height based on the action that the user is currently performing, the version of macOS the user has installed and/or the Mac hardware that the app is running on.

As a result, simply hard coding the offset when laying out the User Interface will not work. The developer will

need to take a dynamic approach.

Apple has included the [Key-Value Observable](#) `ContentLayoutRect` property of the `NSWindow` class to get the current Content Area in code. The developer can use this value to manually position the required elements when the Content Area changes.

The better solution is to use Auto Layout and Size Classes to position the UI elements in either code or Interface Builder.

Code like the following example can be used to position UI elements using AutoLayout and Size Classes in the app's View Controller:

```
using System;
using AppKit;
using Foundation;

namespace MacModern
{
    public partial class ViewController : NSViewController
    {
        #region Computed Properties
        public NSLayoutConstraint topConstraint { get; set; }
        #endregion

        ...

        #region Override Methods
        public override void UpdateViewConstraints ()
        {
            // Has the constraint already been set?
            if (topConstraint == null) {
                // Get the top anchor point
                var contentLayoutGuide = ItemTitle.Window?.ContentLayoutGuide as NSLayoutGuide;
                var topAnchor = contentLayoutGuide.TopAnchor;

                // Found?
                if (topAnchor != null) {
                    // Assemble constraint and activate it
                    topConstraint = topAnchor.ConstraintEqualToAnchor (topAnchor, 20);
                    topConstraint.Active = true;
                }
            }

            base.UpdateViewConstraints ();
        }
        #endregion
    }
}
```

This code creates storage for a top constraint that will be applied to a Label (`ItemTitle`) to ensure that it doesn't slip under the Title and Tool Bar area:

```
public NSLayoutConstraint topConstraint { get; set; }
```

By overriding the View Controller's `UpdateViewConstraints` method, the developer can test to see if the needed constraint has already been built and create it if needed.

If a new constraint needs to be built, the `ContentLayoutGuide` property of the Window the control that needs to be constrained is accessed and cast into a `NSLayoutGuide`:

```
var contentLayoutGuide = ItemTitle.Window?.ContentLayoutGuide as NSLayoutGuide;
```

The TopAnchor property of the `NSLayoutGuide` is accessed and if it is available, it is used to build a new constraint with the desired offset amount and the new constraint is made active to apply it:

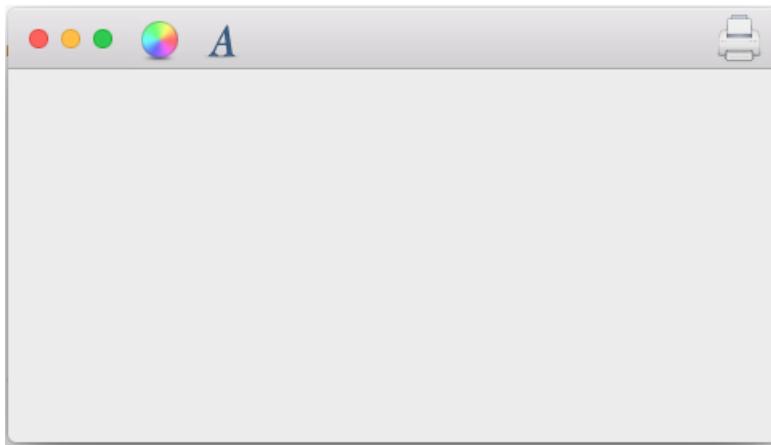
```
// Assemble constraint and activate it
topConstraint = topAnchor.ConstraintEqualToAnchor (topAnchor, 20);
topConstraint.Active = true;
```

Enabling Streamlined Toolbars

A normal macOS Window includes a standard Title Bar at runs along to top edge of the Window. If the Window also includes a Tool Bar, it will be displayed under this Title Bar area:



When using a Streamlined Toolbar, the Title Area disappears and the Tool Bar moves up into the Title Bar's position, in-line with the Window Close, Minimize and Maximize buttons:



The Streamlined Toolbar is enabled by overriding the `ViewWillAppear` method of the `NSViewController` and making it look like the following:

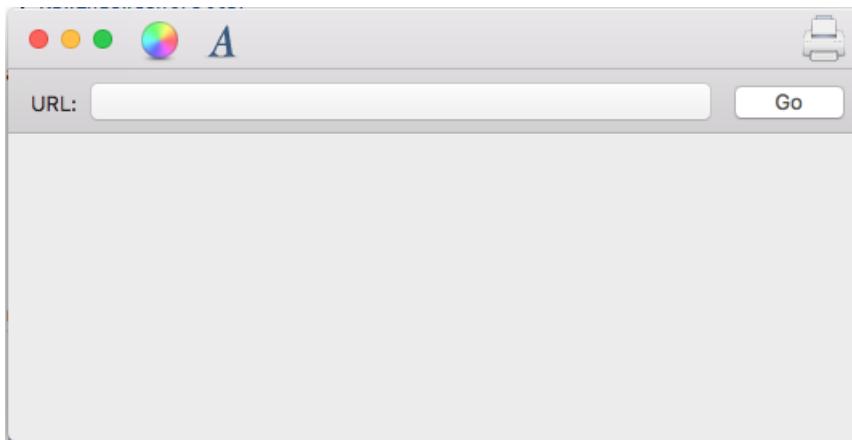
```
public override void ViewWillAppear ()
{
    base.ViewWillAppear ();

    // Enable streamlined Toolbars
    View.Window.TitleVisibility = NSWindowTitleVisibility.Hidden;
}
```

This effect is typically used for *Shoebox Applications* (one window apps) like Maps, Calendar, Notes and System Preferences.

Using Accessory View Controllers

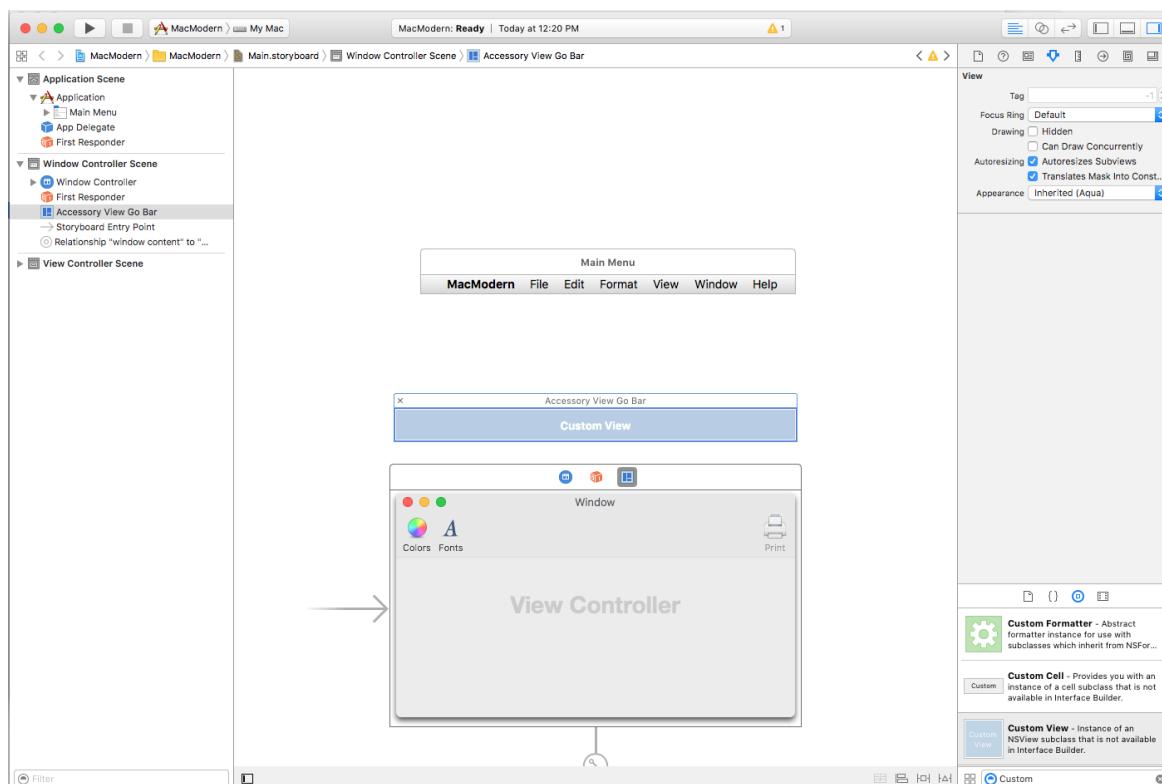
Depending on the design of the app, the developer might also want to complement the Title Bar area with an Accessory View Controller that appears right below the Title/Tool Bar area to provide context sensitive controls to the user based on the activity they are currently engaged in:



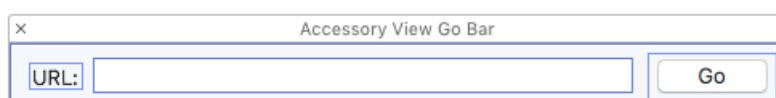
The Accessory View controller will automatically be blurred and resized by the system without developer intervention.

To add an Accessory View Controller, do the following:

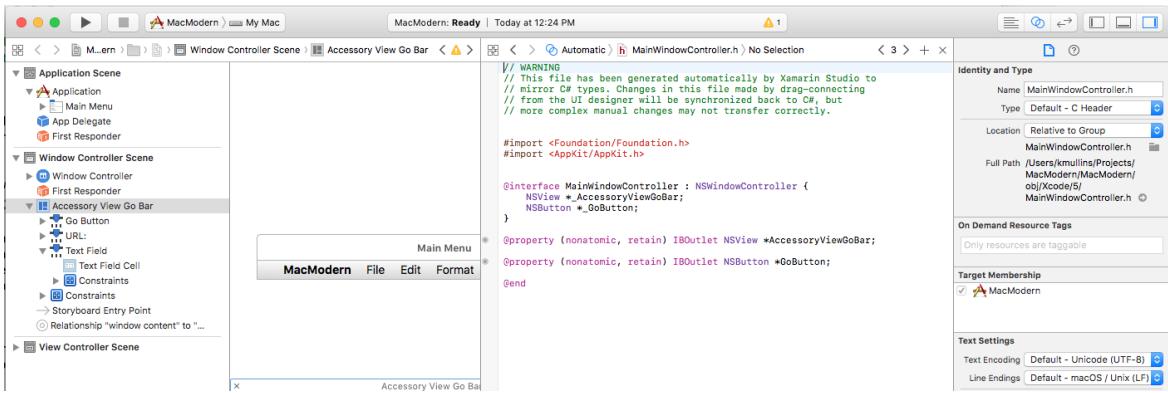
1. In the **Solution Explorer**, double-click the `Main.storyboard` file to open it for editing.
2. Drag a **Custom View Controller** into the Window's hierarchy:



3. Layout the Accessory View's UI:



4. Expose the Accessory View as an **Outlet** and any other **Actions** or **Outlets** for its UI:



5. Save the changes.

6. Return to Visual Studio for Mac to sync the changes.

Edit the `MainWindowController` and make it look like the following:

```

using System;
using Foundation;
using AppKit;

namespace MacModern
{
    public partial class MainWindowController : NSWindowController
    {
        #region Constructor
        public MainWindowController (IntPtr handle) : base (handle)
        {
        }
        #endregion

        #region Override Methods
        public override void WindowDidLoad ()
        {
            base.WindowDidLoad ();

            // Create a title bar accessory view controller and attach
            // the view created in Interface Builder
            var accessoryView = new NSTitlebarAccessoryViewController ();
            accessoryView.View = AccessoryViewGoBar;

            // Set the location and attach the accessory view to the
            // titlebar to be displayed
            accessoryView.LayoutAttribute = NSLayoutAttribute.Bottom;
            Window.AddTitlebarAccessoryViewController (accessoryView);

        }
        #endregion
    }
}

```

The key points of this code are where the View is set to the custom View that was defined in Interface Builder and exposed as an **Outlet**:

```
accessoryView.View = AccessoryViewGoBar;
```

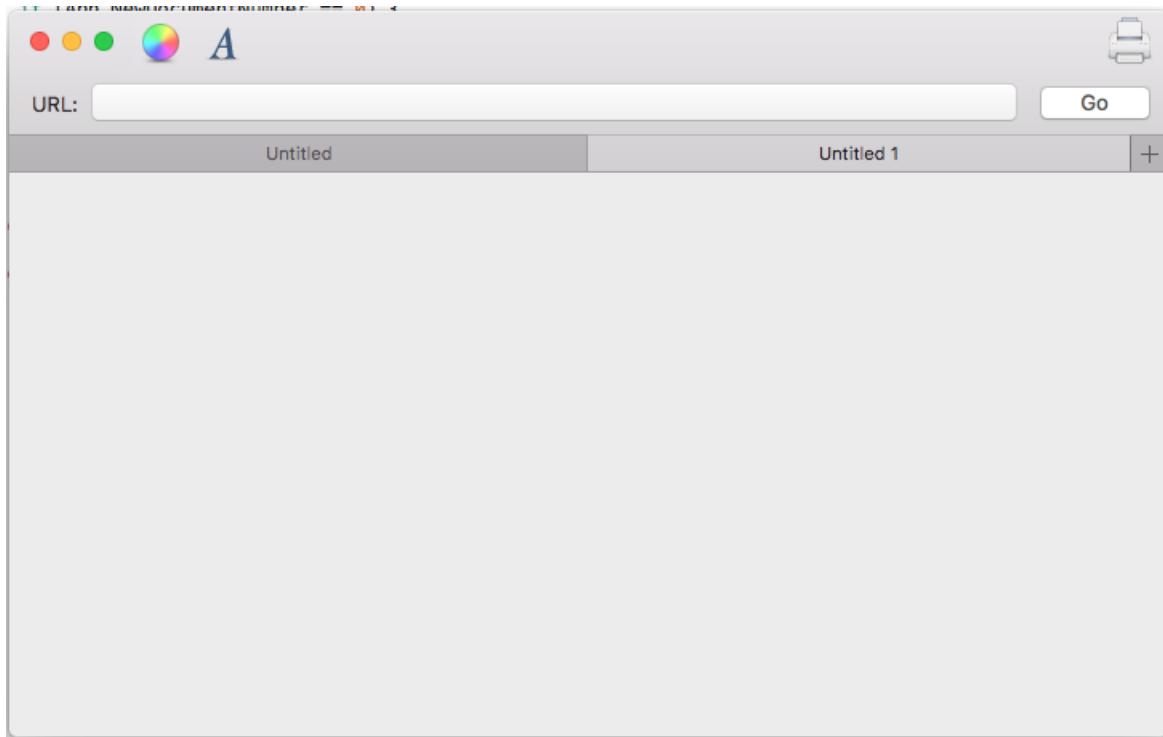
And the `LayoutAttribute` that defines *where* the accessory will be displayed:

```
accessoryView.LayoutAttribute = NSLayoutAttribute.Bottom;
```

Because macOS is now fully localized, the `Left` and `Right` `NSLayoutAttribute` properties have been deprecated and should be replaced with `Leading` and `Trailing`.

Using Tabbed Windows

Additionally, the macOS system might add Accessory View Controllers to the app's Window. For example, to create Tabbed Windows where several of the App's Windows are merged into one virtual Window:



Typically, the developer will need to take limited action use Tabbed Windows in their Xamarin.Mac apps, the system will handle them automatically as follows:

- Windows will automatically be Tabbed when the `OrderFront` method is invoked.
- Windows will automatically be Untabbed when the `orderOut` method is invoked.
- In code all Tabbed windows are still considered "visible", however any non-frontmost Tabs are hidden by the system using CoreGraphics.
- Use the `TabbingIdentifier` property of `NSWindow` to group Windows together into Tabs.
- If it is an `NSDocument` based app, several of these features will be enabled automatically (such as the plus button being added to the Tab Bar) without any developer action.
- Non-`NSDocument` based apps can enable the "plus" button in the Tab Group to add a new document by overriding the `GetNewWindowForTab` method of the `NSWindowsController`.

Bringing all of the pieces together, the `AppDelegate` of an app that wanted to use system based Tabbed Windows could look like the following:

```

using AppKit;
using Foundation;

namespace MacModern
{
    [Register ("AppDelegate")]
    public class AppDelegate : NSApplicationDelegate
    {
        #region Computed Properties
        public int NewDocumentNumber { get; set; } = 0;
        #endregion

        #region Constructors
        public AppDelegate ()
        {
        }
        #endregion

        #region Override Methods
        public override void DidFinishLaunching (NSNotification notification)
        {
            // Insert code here to initialize your application
        }

        public override void WillTerminate (NSNotification notification)
        {
            // Insert code here to tear down your application
        }
        #endregion

        #region Custom Actions
        [Export ("newDocument:")]
        public void NewDocument (NSObject sender)
        {
            // Get new window
            var storyboard = NSStoryboard.FromName ("Main", null);
            var controller = storyboard.InstantiateControllerWithIdentifier ("MainWindow") as
NSWindowController;

            // Display
            controller.ShowWindow (this);
        }
        #endregion
    }
}

```

Where the `NewDocumentNumber` property keeps track of the number of new documents created and the `NewDocument` method creates a new document and displays it.

The `NSWindowController` could then look like:

```

using System;
using Foundation;
using AppKit;

namespace MacModern
{
    public partial class MainWindowController : NSWindowController
    {
        #region Application Access
        /// <summary>
        /// A helper shortcut to the app delegate.
        /// </summary>
        /// <value>The app.</value>
        public static AppDelegate App {
            get { return (AppDelegate)NSApplication.SharedApplication.Delegate; }
        }
    }
}

```

```

        get { return (AppDelegate)NSApplication.SharedApplication.Delegate; }
    }

#endregion

#region Constructor
public MainWindowController (IntPtr handle) : base (handle)
{
}
#endregion

#region Public Methods
public void SetDefaultDocumentTitle ()
{
    // Is this the first document?
    if (App.NewDocumentNumber == 0) {
        // Yes, set title and increment
        Window.Title = "Untitled";
        ++App.NewDocumentNumber;
    } else {
        // No, show title and count
        Window.Title = $"Untitled {App.NewDocumentNumber++}";
    }
}
#endregion

#region Override Methods
public override void WindowDidLoad ()
{
    base.WindowDidLoad ();

    // Prefer Tabbed Windows
    Window.TabbingMode = NSWindowTabbingMode.Preferred;
    Window.TabbingIdentifier = "Main";

    // Set default window title
    SetDefaultDocumentTitle ();

    // Set window to use Full Size Content View
    // Window.StyleMask = NSWindowStyle.FullSizeContentView;

    // Create a title bar accessory view controller and attach
    // the view created in Interface Builder
    var accessoryView = new NSTitlebarAccessoryViewController ();
    accessoryView.View = AccessoryViewGoBar;

    // Set the location and attach the accessory view to the
    // titlebar to be displayed
    accessoryView.LayoutAttribute = NSLayoutAttribute.Bottom;
    Window.AddTitlebarAccessoryViewController (accessoryView);

}

public override void GetNewWindowForTab (NSObject sender)
{
    // Ask app to open a new document window
    App.NewDocument (this);
}
#endregion
}
}

```

Where the static `App` property provides a shortcut to get to the `AppDelegate`. The `SetDefaultDocumentTitle` method sets a new documents title based on the number of new documents created.

The following code, tells macOS that the app prefers to use tabs and provides a string that allows the app's Windows to be grouped into Tabs:

```
// Prefer Tabbed Windows
Window.TABBINGMode = NSWindowTabbingMode.Preferred;
Window.TABBINGIdentifier = "Main";
```

And the following override method adds a plus button to the Tab Bar that will create a new document when clicked by the user:

```
public override void GetNewWindowForTab (NSObject sender)
{
    // Ask app to open a new document window
    App.NewDocument (this);
}
```

Using Core Animation

Core Animation is a high powered graphics rendering engine that is built into macOS. Core Animation has been optimized to take advantage of the GPU (Graphics Processing Unit) available in modern macOS hardware as opposed to running the graphics operations on the CPU, which can slow down the machine.

The `CALayer`, provided by Core Animation, can be used for tasks such as fast and fluid scrolling and animations. An app's User Interface should be composed of multiple subviews and layers to fully take advantage of Core Animation.

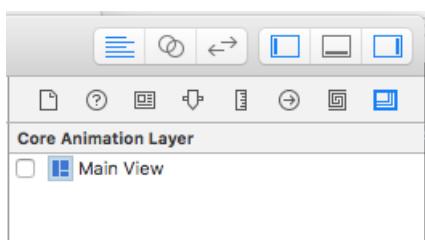
A `CALayer` object provides several properties that allow the developer to control what is presented onscreen to the user such as:

- `Content` - Can be a `NSImage` or `CGImage` that provides the contents of the layer.
- `BackgroundColor` - Sets the background color of the layer as a `CGColor`
- `BorderWidth` - Sets the border width.
- `BorderColor` - Sets the border color.

To utilize Core Graphics in the app's UI, it must be using *Layer Backed Views*, which Apple suggests that the developer should always enable in the Window's Content View. This way, all child views will automatically inherit Layer Backing as well.

Additionally, Apple suggests using Layer Backed Views as opposed to adding a new `CALayer` as a sublayer because the system will automatically handle several of the required settings (such as those required by a Retina Display).

Layer Backing can be enabled by setting the `wantsLayer` of a `NSView` to `true` or inside of Xcode's Interface Builder under the **View Effects Inspector** by checking **Core Animation Layer**:



Redrawing Views with Layers

Another important step when using Layer Backed Views in a Xamarin.Mac app is setting the `LayerContentsRedrawPolicy` of the `NSView` to `OnSetNeedsDisplay` in the `NSViewController`. For example:

```
public override void ViewWillAppear ()
{
    base.ViewWillAppear ();

    // Set the content redraw policy
    View.LayerContentsRedrawPolicy = NSViewLayerContentsRedrawPolicy.OnSetNeedsDisplay;
}
```

If the developer doesn't set this property, the View will be redrawn whenever its frame origin changes, which is not desired for performance reasons. With this property set to `OnSetNeedsDisplay` the developer will manually have to set `NeedsDisplay` to `true` to force the content to redraw, however.

When a View is marked as dirty, the system checks the `WantsUpdateLayer` property of the View. If it returns `true` then the `UpdateLayer` method is called, else the `DrawRect` method of the View is called to update the View's contents.

Apple has the following suggestions for updating a Views contents when required:

- Apple prefers using `UpdateLater` over `DrawRect` whenever possible as it provides a significant performance boost.
- Use the same `layer.Contents` for UI elements that look similar.
- Apple also prefers the developer to compose their UI using standard views such as `NSTextField`, again whenever possible.

To use `UpdateLayer`, create a custom class for the `NSView` and make the code look like the following:

```

using System;
using Foundation;
using AppKit;

namespace MacModern
{
    public partial class MainView : NSView
    {
        #region Computed Properties
        public override bool WantsLayer {
            get { return true; }
        }

        public override bool WantsUpdateLayer {
            get { return true; }
        }
        #endregion

        #region Constructor
        public MainView (IntPtr handle) : base (handle)
        {
        }
        #endregion

        #region Override Methods
        public override void DrawRect (CoreGraphics.CGRect dirtyRect)
        {
            base.DrawRect (dirtyRect);

        }

        public override void UpdateLayer ()
        {
            base.UpdateLayer ();

            // Draw view
            Layer.BackgroundColor = NSColor.Red.CGColor;
        }
        #endregion
    }
}

```

Using Modern Drag and Drop

To present a modern Drag and Drop experience for the user, the developer should adopt *Drag Flocking* in their app's Drag and Drop operations. Drag Flocking is where each individual file or item being dragged initially appears as an individual element that flocks (group together under the cursor with a count of the number of items) as the user continues the drag operation.

If the user terminates the Drag operation, the individual elements will Unflock and return to their original locations.

The following example code enables Drag Flocking on a custom View:

```

using System;
using System.Collections.Generic;
using Foundation;
using AppKit;

namespace MacModern
{
    public partial class MainView : NSView, INSDraggingSource, INSDraggingDestination
    {
        #region Constructor
        public MainView (IntPtr handle) : base (handle)
        {
        }
        #endregion

        #region Override Methods
        public override void MouseDragged (NSEvent theEvent)
        {
            // Create group of string to be dragged
            var string1 = new NSDraggingItem ((NSString)"Item 1");
            var string2 = new NSDraggingItem ((NSString)"Item 2");
            var string3 = new NSDraggingItem ((NSString)"Item 3");

            // Drag a cluster of items
            BeginDraggingSession (new [] { string1, string2, string3 }, theEvent, this);
        }
        #endregion
    }
}

```

The flocking effect was achieved by sending each item being dragged to the `BeginDraggingSession` method of the `NSView` as a separate element in an array.

When working with a `NSTableView` or `NSOutlineView`, use the `PasteboardWriterForRow` method of the `NSTableViewDataSource` class to start the Dragging operation:

```

using System;
using System.Collections.Generic;
using Foundation;
using AppKit;

namespace MacModern
{
    public class ContentsTableDataSource: NSTableViewDataSource
    {
        #region Constructors
        public ContentsTableDataSource ()
        {
        }
        #endregion

        #region Override Methods
        public override INSPasteboardWriting GetPasteboardWriterForRow (NSTableView tableView, nint row)
        {
            // Return required pasteboard writer
            ...

            // Pasteboard writer failed
            return null;
        }
        #endregion
    }
}

```

This allows the developer to provide an individual `NSDraggingItem` for every item in the table that is being dragged as opposed to the older method `WriteRowsWith` that write all of the rows as a single group to the pasteboard.

When working with `NSCollectionViews`, again use the `PasteboardWriterForItemAt` method as opposed to the `WriteItemsAt` method when Dragging begins.

The developer should always avoid putting large files on the pasteboard. New to macOS Sierra, *File Promises* allow the developer to place references to given files on the pasteboard that will later be fulfilled when the user finishes the Drop operation using the new `NSFilePromiseProvider` and `NSFilePromiseReceiver` classes.

Using Modern Event Tracking

For a User Interface element (such as a `NSButton`) that has been added to a Title or Tool Bar area, the user should be able to click the element and have it fire an event as normal (such as displaying a popup window). However, since the item is also in the Title or Tool Bar area, the user should be able to click and drag the element to move the window as well.

To accomplish this in code, create a custom class for the element (such as `NSButton`) and override the `MouseDown` event as follows:

```
public override void MouseDown (NSEvent theEvent)
{
    var shouldCallSuper = false;

    Window.TrackEventsMatching (NSEventMask.LeftMouseUp, 2000, NSRunLoop.NSRunLoopEventTracking, (NSEvent
evt, ref bool stop) => {
        // Handle event as normal
        stop = true;
        shouldCallSuper = true;
    });

    Window.TrackEventsMatching(NSEventMask.LeftMouseDragged, 2000, NSRunLoop.NSRunLoopEventTracking,
(NSEvent evt, ref bool stop) => {
        // Pass drag event to window
        stop = true;
        Window.PerformWindowDrag (evt);
    });

    // Call super to handle mousedown
    if (shouldCallSuper) {
        base.MouseDown (theEvent);
    }
}
```

This code uses the `TrackEventsMatching` method of the `NSWindow` that the UI element is attached to intercept the `LeftMouseUp` and `LeftMouseDragged` events. For a `LeftMouseUp` event, the UI element responds as normal. For the `LeftMouseDragged` event, the event is passed to the `NSWindow`'s `PerformWindowDrag` method to move the window on screen.

Calling the `PerformWindowDrag` method of the `NSWindow` class provides the following benefits:

- It allows for the Window to move, even if the app is hung (such as when processing a deep loop).
- Space switching will work as expected.
- The Spaces Bar will be displayed as normal.
- Window snapping and alignment work as normal.

Using Modern Container View Controls

macOS Sierra provides many modern improvements to the existing Container View Controls available in previous version of the OS.

Table View Enhancements

The developer should always use the new `NSView` based version of Container View Controls such as `NSTableView`. For example:

```
using System;
using System.Collections.Generic;
using Foundation;
using AppKit;

namespace MacModern
{
    public class ContentsTableDelegate : NSTableViewDelegate
    {
        #region Constructors
        public ContentsTableDelegate ()
        {
        }
        #endregion

        #region Override Methods
        public override NSView GetViewForItem (NSTableView tableView, NSTableColumn TableColumn, nint row)
        {
            // Build new view
            var view = new NSView ();
            ...

            // Return the view representing the item to display
            return view;
        }
        #endregion
    }
}
```

This allows for custom Table Row Actions to be attached to given rows in the table (such as swiping right to delete the row). To enable this behavior, override the `RowActions` method of the `NSTableViewDelegate`:

```

using System;
using System.Collections.Generic;
using Foundation;
using AppKit;

namespace MacModern
{
    public class ContentsTableDelegate : NSTableViewDelegate
    {
        #region Constructors
        public ContentsTableDelegate ()
        {
        }
        #endregion

        #region Override Methods
        public override NSView GetViewForItem (NSTableView tableView, NSTableColumn TableColumn, nint row)
        {
            // Build new view
            var view = new NSView ();
            ...

            // Return the view representing the item to display
            return view;
        }

        public override NSTableViewRowAction [] RowActions (NSTableView tableView, nint row,
NSTableRowActionEdge edge)
        {
            // Take action based on the edge
            if (edge == NSTableRowActionEdge.Trailing) {
                // Create row actions
                var editAction = NSTableViewRowAction.FromStyle (NSTableViewRowActionStyle.Regular, "Edit",
(action, rowNum) => {
                    // Handle row being edited
                    ...
                });

                var deleteAction = NSTableViewRowAction.FromStyle (NSTableViewRowActionStyle.Destructive,
"Delete", (action, rowNum) => {
                    // Handle row being deleted
                    ...
                });
            }

            // Return actions
            return new [] { editAction, deleteAction };
        } else {
            // No matching actions
            return null;
        }
    }
    #endregion
}
}

```

The static `NSTableViewRowAction.FromStyle` is used to create a new Table Row Action of the following styles:

- `Regular` - Performs a standard non-destructive action such as edit the row's contents.
- `Destructive` - Performs a destructive action such as delete the row from the table. These actions will be rendered with a red background.

ScrollView Enhancements

When using a Scroll View (`NSScrollView`) directly, or as part of another control (such as `NSTableView`), the contents of the Scroll View can slide under the Title and Tool Bar areas in a Xamarin.Mac app using a Modern Look and Views.

As a result, the first item in the Scroll View content area can be partially obscured by the Title and Tool Bar area.

To correct this issue, Apple has added two new properties to the `NSScrollView` class:

- `ContentInsets` - Allows the developer to provide a `NSEdgeInsets` object defining the offset that will be applied to the top of the Scroll View.
- `AutomaticallyAdjustsContentInsets` - If `true` the Scroll View will automatically handle the `ContentInsets` for the developer.

By using the `ContentInsets` the developer can adjust the start of the Scroll View to allow for the inclusion of accessories such as:

- A Sort indicator like the one shown in the Mail app.
- A Search Field.
- A Refresh or Update button.

Auto Layout and Localization in Modern Apps

Apple has included several technologies in Xcode that allow the developer to easily create an internationalized macOS app. Xcode now allows the developer to separate user-facing text from the app's User Interface design in its Storyboard files and provides tools to maintain this separation if the UI changes.

For more information, please see Apple's [Internationalization and Localization Guide](#).

Implementing Base Internationalization

By implementing Base Internationalization, the developer can provide a single Storyboard file to represent the app's UI and separate out all of the user-facing strings.

When the developer is creating the initial Storyboard file (or files) that define the app's User Interface, they will be built in the Base Internationalization (the language that the developer speaks).

Next, the developer can export localizations and the Base Internationalization strings (in the Storyboard UI design) that can be translated into multiple languages.

Later, these localizations can be imported and Xcode will generate the language-specific string files for the Storyboard.

Implementing Auto Layout to Support Localization

Because localized versions of string values can have vastly different sizes and/or reading direction, the developer should use Auto Layout to position and size the app's User Interface in a Storyboard file.

Apple suggest doing the following:

- **Remove Fixed Width Constraints** - All text-based Views should be allowed to resize based on their content. Fixed width View may crop their content in specific languages.
- **Use Intrinsic Content Sizes** - By default text-based Views will auto-size to fit their content. For text-based View that are not sizing correctly, select them in Xcode's Interface Builder then choose **Edit > Size To Fit Content**.
- **Apply Leading and Trailing Attributes** - Because the direction of the text can change based on the user's language, use the new `Leading` and `Trailing` constraint attributes as opposed to the existing `Right` and `Left` attributes. `Leading` and `Trailing` will automatically adjust based on the languages direction.

- **Pin Views to Adjacent Views** - This allows the Views to reposition and resize as the Views around them change in response to the language selected.
- **Don't Set a Windows Minimum and/or Maximum Sizes** - Allow Windows to change size as the language selected resizes their content areas.
- **Test Layout Changes Constantly** - During development an app should be tested constantly in different languages. See Apple's [Testing Your Internationalized app](#) documentation for more details.
- **Use NSStackViews to Pin Views Together** - `NSStackViews` allows their contents to shift and grow in predictable ways and the content change size based on the language selected.

Localizing in Xcode's Interface Builder

Apple has provided several features in Xcode's Interface Builder that the developer can use when designing or editing an app's UI to support localization. The **Text Direction** section of the **Attribute Inspector** allows the developer to provide hints on how direction should be used and updated on a select Text-Based View (such as `NSTextField`):



There are three possible values for the **Text Direction**:

- **Natural** - The layout is based on the string assigned to the control.
- **Left to Right** - The layout is always forced to left to right.
- **Right to Left** - The layout is always forced to right to left.

There are two possible values for the **Layout**:

- **Left to Right** - The layout is always left to right.
- **Right to Left** - The layout is always right to left.

Typically these should not be changed unless a specific alignment is required.

The **Mirror** property tells the system to flip specific control properties (such as the Cell Image Position). It has three possible values:

- **Automatically** - The position will automatically change based on the direction of the language selected.
- **In Right To Left Interface** - The position will only be changed in right to left based languages.
- **Never** - The position will never change.

If the developer has specified **Center**, **Justify** or **Full** alignment on the content of a text-based View, these will never be flipped based on language selected.

Prior to macOS Sierra, controls created in code would not be mirrored automatically. The developer had to use code like the following to handle mirroring:

```

public override void ViewDidLoad ()
{
    base.ViewDidLoad ();

    // Setting a button's mirroring based on the layout direction
    var button = new NSButton ();
    if (button.UserInterfaceLayoutDirection == NSUserInterfaceLayoutDirection.LeftToRight) {
        button.Alignment = NSTextAlignment.Right;
        button.ImagePosition = NSCellImagePosition.ImageLeft;
    } else {
        button.Alignment = NSTextAlignment.Left;
        button.ImagePosition = NSCellImagePosition.ImageRight;
    }
}

```

Where the `Alignment` and `ImagePosition` are being set based on the `UserInterfaceLayoutDirection` of the control.

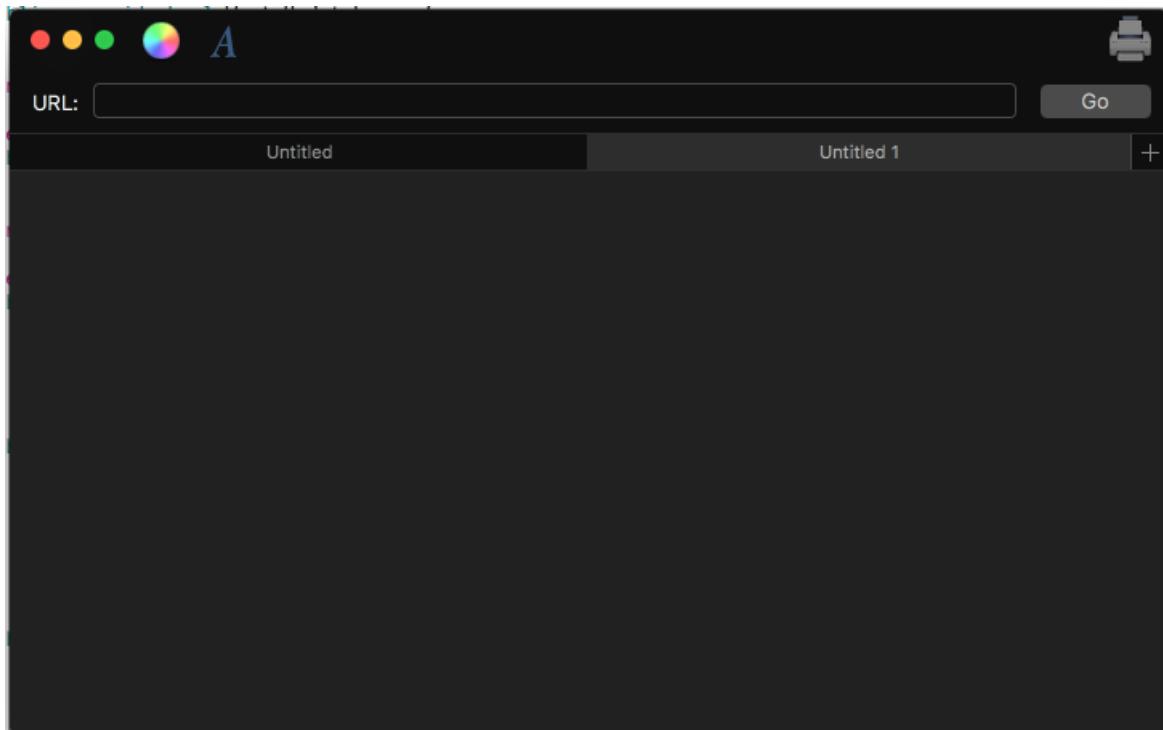
macOS Sierra adds several new convenience constructors (via the static `CreateButton` method) that take several parameters (such as Title, Image and Action) and will automatically mirror correctly. For example:

```

var button2 = NSButton.CreateButton (myTitle, myImage, () => {
    // Take action when the button is pressed
    ...
});
```

Using System Appearances

Modern macOS apps can adopt a new Dark Interface Appearance that works well for image creation, editing or presentation apps:



This can be done by adding one line of code before the Window is presented. For example:

```

using System;
using AppKit;
using Foundation;

namespace MacModern
{
    public partial class ViewController : NSViewController
    {
        ...
        ...

        #region Override Methods
        public override void ViewWillAppear ()
        {
            base.ViewWillAppear ();

            // Apply the Dark Interface Appearance
            View.Window.Appearance = NSAppearance.GetAppearance (NSAppearance.NameVibrantDark);

            ...
        }
        #endregion
    }
}

```

The static `GetAppearance` method of the `NSAppearance` class is used to get a named appearance from the system (in this case `NSAppearance.NameVibrantDark`).

Apple has the following suggestions for using System Appearances:

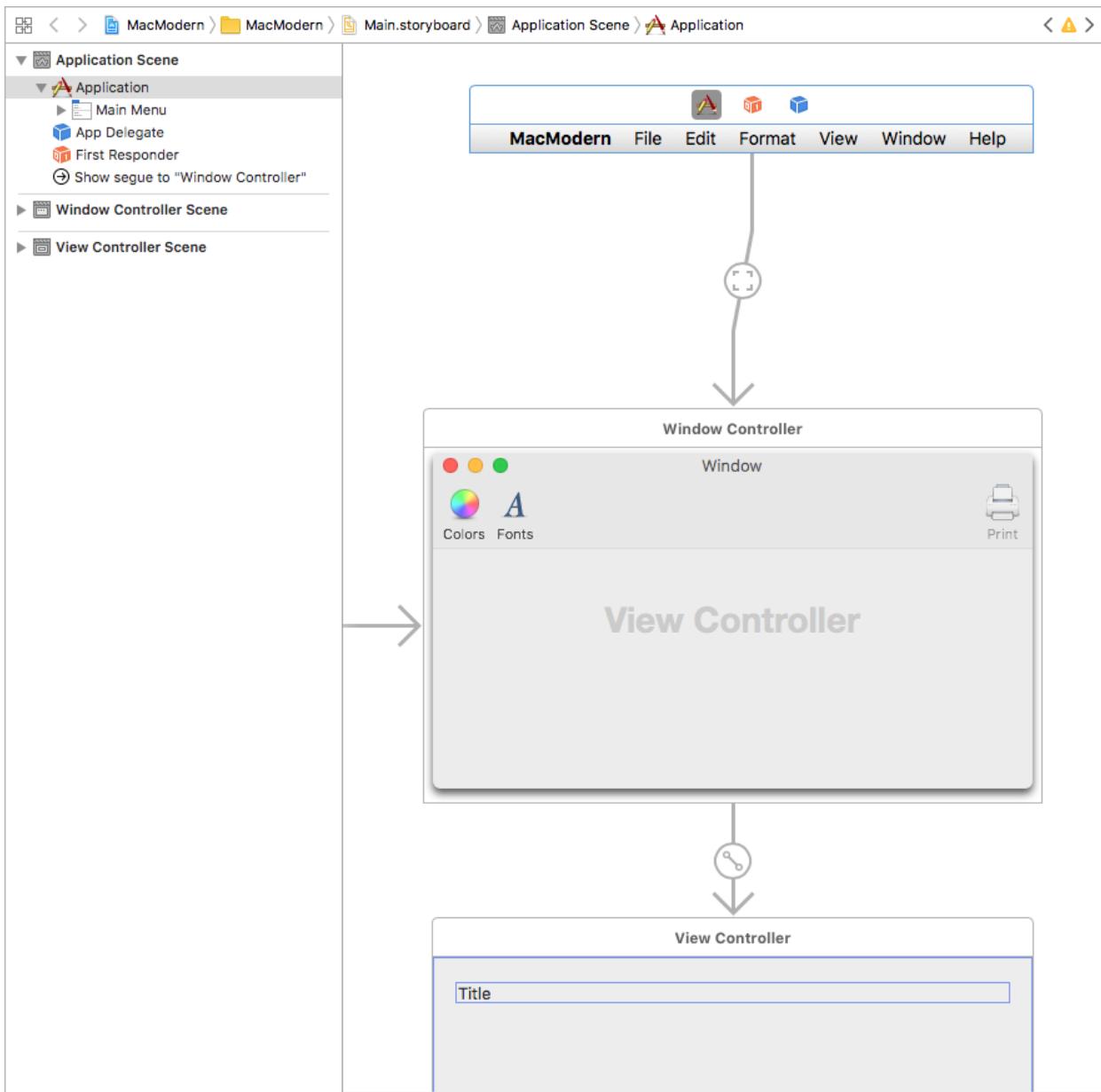
- Prefer named colors over hardcoded values (such as `LabelColor` and `SelectedControlColor`).
- Use system standard control style where possible.

A macOS app that uses the System Appearances will automatically work correctly for users that have enabled Accessibility features from the System Preferences app. As a result, Apple suggests that the developer should always use System Appearances in their macOS apps.

Designing UIs with Storyboards

Storyboards allow the developer to not only design the individual elements that make up an app's User Interface, but to visualize and design the UI flow and the hierarchy of the given elements.

Controllers allow the developer to collect elements into a unit of composition and Segues abstract and remove the typical "glue code" required to move throughout the View Hierarchy:



For more information, please see our [Introduction to Storyboards](#) documentation.

There are many instances where a given scene defined in a storyboard will require data from a previous scene in the View Hierarchy. Apple has the following suggestions for passing information between scenes:

- Data dependancies should always cascade downwards through the hierarchy.
- Avoid hardcoding UI structural dependancies, as this limits UI flexibility.
- Use C# Interfaces to provide generic data dependancies.

The View Controller that is acting as the source of the Segue, can override the `PrepareForSegue` method and do any initialization required (such as passing data) before the Segue is executed to display the target View Controller. For example:

```

public override void PrepareForSegue (UIStoryboardSegue segue, NSObject sender)
{
    base.PrepareForSegue (segue, sender);

    // Take action based on Segue ID
    switch (segue.Identifier) {
        case "MyNamedSegue":
            // Prepare for the segue to happen
            ...
            break;
    }
}

```

For more information, please see our [Segues](#) documentation.

Propagating Actions

Based on the design of the macOS app, there might be times when the best handler for an Action on a UI control might be in elsewhere in the UI Hierarchy. This is typically true of Menus and Menu Items that live in their own scene, separate from the rest of the app's UI.

To handle this situation, the developer can create a Custom Action and pass the Action up the responder chain. For more information please see our [Working with Custom Window Actions](#) documentation.

Modern Mac Features

Apple has included several user-facing features in macOS Sierra that allow the developer to make the most of the Mac platform, such as:

- **NSUserActivity** - This allows the app to describe the activity that the user is currently involved in. `NSUserActivity` was initially created to support HandOff, where an activity started on one of the user's devices could be picked up and continued on another device. `NSUserActivity` works the same in macOS as it does in iOS so please see our [Introduction to Handoff](#) iOS documentation for more details.
- **Siri on Mac** - Siri uses the Current Activity (`NSUserActivity`) to provide context to the commands a user can issue.
- **State Restoration** - When the user quits an app on macOS and then later relaunches it, the app will automatically be returned to its previous state. The developer can use the State Restoration API to encode and restore transient UI states before the User Interface is displayed to the user. If the app is `NSDocument` based, State Restoration is handled automatically. To enable State Restoration for non-`NSDocument` based apps, set the `Restorable` of the `NSWindow` class to `true`.
- **Documents in the Cloud** - Prior to macOS Sierra, an app had to explicitly opt-in to working with documents in the user's iCloud Drive. In macOS Sierra the user's **Desktop** and **Documents** folders may be synced with their iCloud Drive automatically by the system. As a result, local copies of documents may be deleted to free up space on the user's machine. `NSDocument` based apps will automatically handle this change. All other app types will need to use a `NSFileCoordinator` to sync reading and writing of documents.

Summary

This article has covered several tips, features and techniques a developer can use to build a modern macOS app in Xamarin.Mac.

Related Links

- macOS Samples

Additional macOS Sierra Framework Changes

11/2/2020 • 8 minutes to read • [Edit Online](#)

Accelerate Framework Enhancements

The following enhancement have been made to the Accelerate Framework for macOS Sierra:

- Added Quadrature (integral calculus).
- Added Basic functions for constructing neural networks.
- Added Geometric predicate functions to test for things like the intersection of two geometric objects.

AppKit Framework Enhancements

The following enhancement have been made to the AppKit Framework for macOS Sierra:

- Several enhancements to `NSCollectionView` such as:
 - **Collapsible Sections** - Allows the user to collapse a Collection View section into a single horizontal row.
 - **Floating Headers** - Headers and Footers can now be floated (in a flow layout) using the same API as `UICollectionView` in iOS.
 - **Scrollable Background Views** - A collection Views background can now be set to scroll along with the content.
- The deferred view layout pass has been optimized and extended.
- The drag-and-drop API now includes the new `NSFilePromiseProvider` and `NSFilePromiseReceiver` classes to support drag flocking.
- Several convenience constructors have been added to existing controls:
 - `NSButton` includes new constructors for creating push buttons, checkboxes and radio buttons.
 - `NSTextField` includes new constructors for creating wrapping and non-wrapping labels, attributed labels and editable text fields.
 - `NSSegmentedControl` includes new constructors for creating segmented controls from a group of labels or images.
 - `NSSlider` includes new constructors for creating horizontal linear sliders.
 - `NSImageView` includes new constructors for creating non-editable image views from a given `NSImage`.
- The new `NSGridView` has been added to auto layout a collection of sub views into a grid with variable sized rows and columns that can be dynamically hidden or shown.

AVFoundation Framework Enhancements

The following enhancement have been made to the AVFoundation Framework for macOS Sierra:

- In macOS, the app no longer has to implement different `AVPlayerItem` behaviors based on content type. Simply set the `Rate` property and AVFoundation will determine when enough content is available for playback without stalling.
- The new `AVPlayerLooper` class makes it easier to loop a given piece of media during playback.
- The `AVAssetDownloadURLSession` class allows for the downloading and later playback of FairPlay encrypted HLS streams.

Core Data Framework Enhancements

The following enhancement have been made to the Core Data Framework for macOS Sierra:

- Root `NSManagedObjectContext` objects supports concurrent faulting and fetching without serialization.
- The `NSPersistentStoreCoordinator` class maintains a pool of SQLite data stores.
- The `NSManagedObjectContext` objects with SQLite data stores in the WAL Journal Mode support the new query generation feature where Managed Object Contexts (MOC) can be pinned to specific database versions for future fetching and faulting transactions.
- Using the high-level `NSPersistenceContainer` to reference the `NSPersistentStoreCoordinator`, `NSManagedObjectModel` and other Core Data configuration resources.
- Several new convenience methods have been added to `NSManagedObject` making it easier to perform fetches and create subclasses.

For more information, please see Apple's [Core Data Framework Reference](#).

Core Image Framework Enhancements

The following enhancement have been made to the Core Image Framework for macOS Sierra:

- The `ImageWithExtent` method of the `CIFilter` class can be used to insert custom processing into the filter operation. Core Image will invoke the given callback between filters when processing an image for output or display.
- The app can now process images in a color space outside of the Core Image context's working color space by converting in and out of the color space before and after processing.
- The Core Image kernel can now request a specific pixel output format.
- The following new image filters have been added: `CINinePartTitled`, `CINinePartStretched`, `CIHueSaturationValueGradient`, `CIEdgePreserveUpsampleFilter` and `CIClamp`.

Foundation Framework Enhancements

The following enhancement have been made to the Foundation Framework for macOS Sierra:

- Use the `NSDimentions` API for representing, converting and displaying many of the most common physical units such as mass, length, speed, duration and temperature.
- Use the `NSISO8601DateFormatter` class for parsing and generating ISO 8601 formatted dates.
- Use the new `NSDateInterval` class to make date and time interval calculations such as durations, for comparing intervals and testing for interval intersections.
- Use the `NSPersonNameComponentsFormatter` class to parse the elements of a person's name from a string.
- Use the new `NSURLSessionTaskMetrics` class to obtain metrics for a URL networking session.

For more information, please see Apple's [Foundation Release Notes for OS X v10.12 and iOS 10](#).

GameKit Framework Enhancements

The following enhancement have been made to the GameKit Framework for macOS Sierra:

- The **Game Center App** has been deprecated and removed from macOS. If the app uses GameKit, it *must* present its own interface to display GameKit features such as leaderboards, etc.
- A new iCloud-only account type has been implemented by the `GKCloudPlayer` class.
- The new `GKGameSession` class provides a generalized solution for managing persistent data storage on Game Center. `GKGameSession` maintains a list of players and the app is responsible for implementing how

and when participant date is stored, retrieved or exchanged between players. In many instances Game Sessions can replace existing turn-based matches, real-time matches or persistent game save methods.

GamePlayKit Framework Enhancements

The following enhancement have been made to the GamePlayKit Framework for macOS Sierra:

- Procedural noise generation has been added and can be used to enhance the realism in natural-looking textures, add realism to camera movements and help generate rich game worlds.
- Use Spatial Partitioning to partition the game world data for efficient searching.
- A new Monte Carlo strategist ([GKMonteCarloStrategist](#)) has been added for exhaustive possible move computation.
- A new Decision Tree API has been added ([GKDecisionTree](#) and [GKDecisionNode](#)) to enhance the game-building AI.
- 3D support has been added to existing agent and path-finding behaviors using the new [GKAgent3D](#) and [GKGraphNode3D](#) classes.
- Use the new [GKMeshGraph](#) class to provide high-performance, natural-looking paths.
- The new [GKScene](#) and [GSKNodeComponent](#) classes make combining GameplayKit and SpriteKit easier than ever.

Metal Framework Enhancements

The following enhancement have been made to the Metal Framework for macOS Sierra:

- 3D apps and games can now use *Tessellation* to efficiently render complex scenes and geometry via the GPU.
- Use Function Specialization to create a highly-optimized collection of material and light combination functions for a scene.
- Provide fine-grained control of resource allocation to optimize performance of Metal based apps using Resource Heaps and Memoryless Render Targets.

To learn more, please see Apple's [Metal Programming Guide](#).

Model I/O Framework Enhancements

The following enhancement have been made to the Model I/O Framework for macOS Sierra:

- The USD file format is now supported.
- Use the new [MDLMaterialPropertyGraph](#) class to easily support runtime changes to models.
- Signed Distance Field support has been added to the [MDLVoxelArray](#) class.
- Use the new [MDLLightProbeIrradianceDataSource](#) class to assist in Light Probe placement.

Photos Framework Enhancements

The following enhancement have been made to the Photos Framework for macOS Sierra:

- Live Photo editing is now available for apps that support the Photos framework and to photo editing extensions (for use inside of the Photos and Camera apps).
- Use the new [PHLivePhotoEditingContext](#) class to apply edits to both the video and still content of Live Photos.
- Use the [CILImageProcessorInput](#) and [CILImageProcessorOutput](#) classes to take advantage of the new Core Image processor feature to perform edits.

- To support Live Photos, the `PHLivePhoto` and `PHLivePhotoView` classes have been ported from iOS to macOS.

SceneKit Framework Enhancements

The following enhancement have been made to the SceneKit Framework for macOS Sierra:

- Now includes a new Physically Based Rendering (PBR) system for more realistic results with simpler asset authoring.
- Use the new `SCNLightingModelPhysicallyBased` shading model to produce a wide range of realistic shading effects while requiring only three fundamental properties (`Diffuse`, `Metalness` and `Roughness`).
- Since PBR shading works best with environment-based lighting, use the `LightingEnvironment` property to assign image-based lighting to an entire scene.
- Use the `IESprofileURL` property to import real-world light fixtures that define lighting base on real-world values such as intensity (in lumens) and color temperature (in degrees Kelvin).
- The `SCNCamera` class can provide greater realism by using HDR features and effects. Use adaptive exposure to create automatic effects or use vignetting, color fringing and color grading to add filmatic effects to the game.
- Both PBR and HDR camera features provide better results than traditional rendering techniques and, as a result, SceneKit now performs all color calculations in a linear color space (using P3 color gamut on wide-color device displays).
- SceneKit now matches all colors by reading the color profile information.
- SceneKit interprets color component values in a linear RGB color space for all shader types.
- Since SceneKit reads and adjust for color profile information in texture images, use Asset Catalogs for all images to ensure this information is provided.
- Both linear color space rendering and wide-color can be disabled by specifying the `SCNDisableLinearSpaceRendering` and `SCNDisableWideGamut` keys in the app's `Info.plist`.
- Build arbitrary polygon primitives (either loaded from files or generated programmatically) to specify geometry with the new `SCNGeometryPrimitiveTypePolygon` class.

Security Framework Enhancements

The following enhancement have been made to the Security Framework for macOS Sierra:

- The `SecKey` interface has been modernized and unified across all platforms (iOS, tvOS, watchOS and macOS).

SpriteKit Framework Enhancements

The following enhancement have been made to the SpriteKit Framework for macOS Sierra:

- Tilemaps now support square, hexagonal and isometric tile shapes for 2D, 2.5D and side-scrolling games using the `SKTileMapMode`, `SKTileGroup`, `SKTileGroupRule` and `SKTileSet` classes.
- Use the new `SKWarpGeometry` class to stretch or distort `SKSpriteNode` or `SKEffectNode` rendering. The new `SKAction` class can be used to animate transitions between warp effects.
- Custom shaders can provide attributes (`skAttribute`) that can be configured separately by each node that uses the shader by supplying an Attribute Value (`skAttributeValue`).
- The `SKView` class provides several new methods to give fine-grained control over when and how a scene is rendered.

New Frameworks

The following frameworks have been added to macOS Sierra:

- **Intents Framework** - This framework allow the app to examine interactions (such as location or user actions), and take action based on that information.
- **SafariServices Framework** - This framework allow the app to develop app extensions for Safari (such as content blockers) for both macOS and iOS.

Related Links

- [Mac Samples](#)
- [What's new in OS X 10.12](#)

Xamarin.Mac - macOS Sierra Troubleshooting

11/2/2020 • 2 minutes to read • [Edit Online](#)

This article provides several troubleshooting tips for working with macOS Sierra in Xamarin.Mac apps.

The following sections list some known issues that can occur when using macOS Sierra with Xamarin.Mac and the solution to those issues:

- [App Store](#)
- [Apple Pay](#)
- [Binary Compatibility](#)
- [CFNetwork HTTP Protocol](#)
- [CloudKit](#)
- [Core Image](#)
- [Notifications](#)
- [NSUserActivity](#)
- [Safari](#)

App Store

Known Issues:

- When testing In-App Purchases in the sandbox environment, the authentication dialog may appear twice.
- When testing In-App Purchases with hosted content in the sandbox environment, the password dialog will appear every time the app is brought to the foreground until the content download completes.

Apple Pay

If an incorrect expiration date or security code (CW) is entered when adding a new payment card to Apple Pay, the card provisioning process will be terminated.

Binary Compatibility

Known Issues:

- Calling `NSObject.ValueForKey` will a `null` key will result in an exception.
- Both `NSURLSession` and `NSURLConnection` no longer RC4 cipher suites during the TLS handshake for `http://` URLs.
- Apps can hang if they modify a superview's geometry in either the `ViewWillLayoutSubviews` or `LayoutSubviews` methods.
- For all SSL/TLS connections, the RC4 symmetric cipher is now disabled by default. Additionally, the Secure Transport API no longer supports SSLv3 and it is recommended that the app stop using SHA-1 and 3DES cryptography as soon as possible.

CFNetwork HTTP Protocol

The `HTTPBodyStream` property of the `NSMutableURLRequest` class must be set to an unopened stream since

`NSURLConnection` and `NSURLSession` now strictly enforce this requirement.

CloudKit

Long running operations will return a *"You don't have permission to save the file."* error.

Core Image

The `CIIImageProcessor` API now supports an arbitrary input image count. `CIIImageProcessor` API that was included in macOS Sierra beta 1 will be removed.

Notifications

When working with Notification Content Extensions, View Controllers are not being correctly released and might result in a crash when Extension memory limits are reached.

NSUserActivity

After a Handoff operation, the `UserInfo` property of a `NSUserActivity` object might be empty. Explicitly call `BecomeCurrent` on `NSUserActivity` object as a current workaround.

Safari

WebGeolocation requires a secure (`https://`) URL to work on both iOS 10 and macOS Sierra to prevent malicious use of location data.

Related Links

- [Mac Samples](#)
- [What's new in macOS 10.12](#)

Binding Mac libraries for Xamarin.Mac

10/28/2019 • 2 minutes to read • [Edit Online](#)

Follow these links to learn about binding Objective-C libraries on Xamarin.Mac:

- [Overview](#) - Describes how binding works.
- [Binding Objective-C Libraries](#) - Instructions on how to bind Objective-C libraries for use in Xamarin projects.
- [Type Definition Reference Guide](#) - Describes all of the attributes available to binding authors to drive the binding generation process.

Objective Sharpie

Objective Sharpie is a command-line tool to help bootstrap the first pass of a binding. It works by parsing the header files of a native library to map the public API into the [binding definition](#) (a process that is otherwise done manually). Objective Sharpie does not create a binding by itself, but it can help get you started!

Examples

Refer to the [XMBindingExample Mac sample](#) to learn how to create a Mac binding using binding projects.

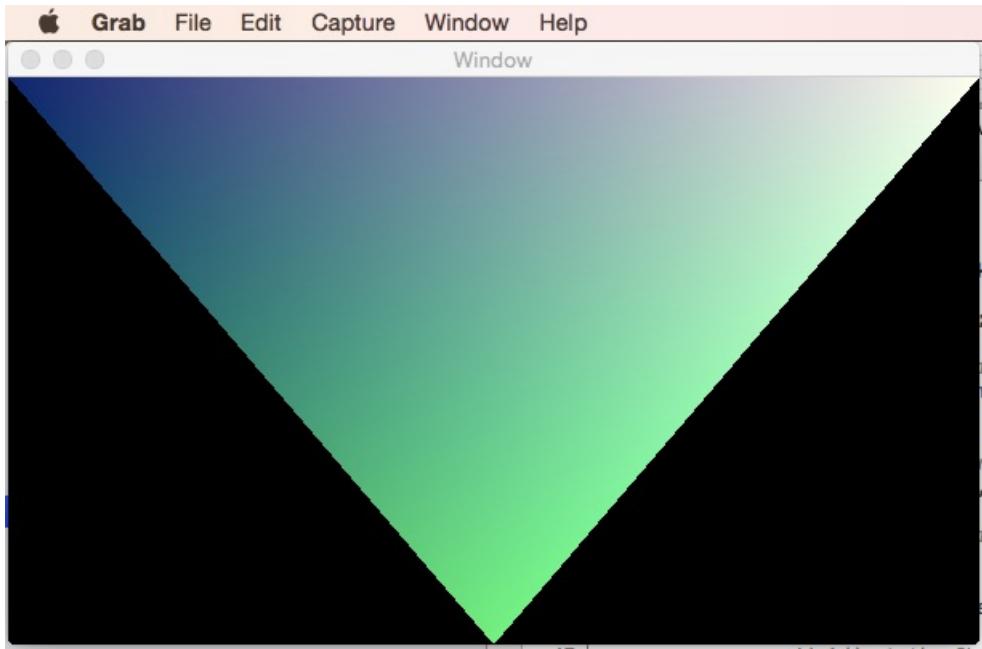
Related Links

- [Binding Objective-C](#)
- [Binding iOS libraries](#)

Introduction to OpenTK in Xamarin.Mac

11/2/2020 • 8 minutes to read • [Edit Online](#)

OpenTK (The Open Toolkit) is an advanced, low-level C# library that makes working with OpenGL, OpenCL and OpenAL easier. OpenTK can be used for games, scientific applications or other projects that require 3D graphics, audio or computational functionality. This article gives a brief introduction to using OpenTK in a Xamarin.Mac app.



In this article, we'll cover the basics of OpenTK in a Xamarin.Mac application. It is highly suggested that you work through the [Hello, Mac](#) article first, specifically the [Introduction to Xcode and Interface Builder](#) and [Outlets and Actions](#) sections, as it covers key concepts and techniques that we'll be using in this article.

You may want to take a look at the [Exposing C# classes / methods to Objective-C](#) section of the [Xamarin.Mac Internals](#) document as well, it explains the `Register` and `Export` commands used to wire-up your C# classes to Objective-C objects and UI Elements.

About OpenTK

As stated above, OpenTK (The Open Toolkit) is an advanced, low-level C# library that makes working with OpenGL, OpenCL and OpenAL easier. Using OpenTK in a Xamarin.Mac app provides the following features:

- **Rapid Development** - OpenTK provides strong data types and inline documentation to improve your coding workflow and catch errors easier and sooner.
- **Easy Integration** - OpenTK was designed to easily integrate with .NET applications.
- **Permissive License** - OpenTK is distributed under the MIT/X11 Licenses and is totally free.
- **Rich, Type-Safe Bindings** - OpenTK supports the latest versions of OpenGL, OpenGL|ES, OpenAL and OpenCL with automatic extension loading, error checking and inline documentation.
- **Flexible GUI Options** - OpenTK provides the native, high-performance Game Window designed specifically for games and Xamarin.Mac.
- **Fully Managed, CLS-Compliant Code** - OpenTK supports 32-bit and 64-bit versions of macOS with no unmanaged libraries.
- **3D Math Toolkit** OpenTK supplies `Vector`, `Matrix`, `Quaternion` and `Bezier` structs via its 3D Math Toolkit.

OpenTK can be used for games, scientific applications or other projects that require 3D graphics, audio or computational functionality.

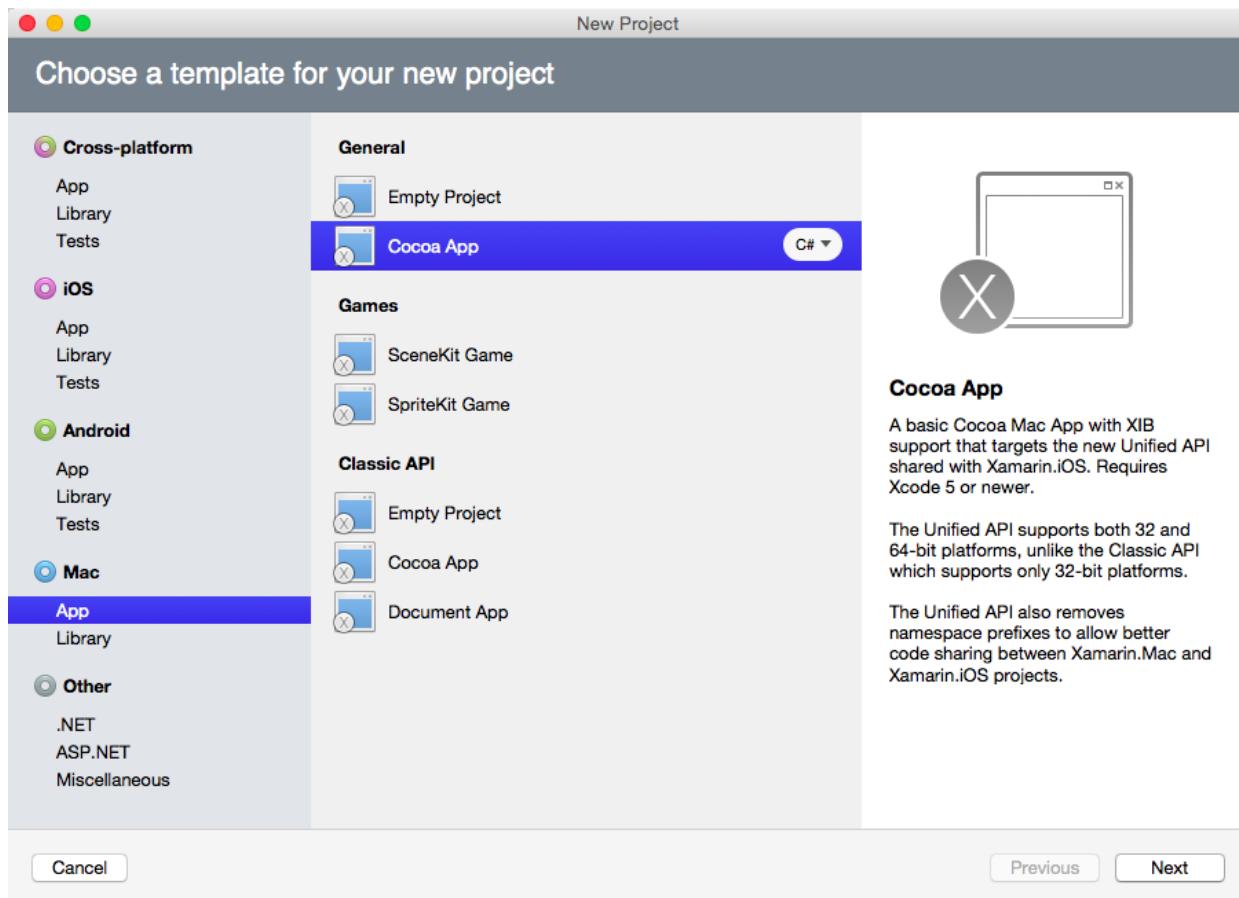
For more information, please see [The Open Toolkit](#) website.

OpenTK Quickstart

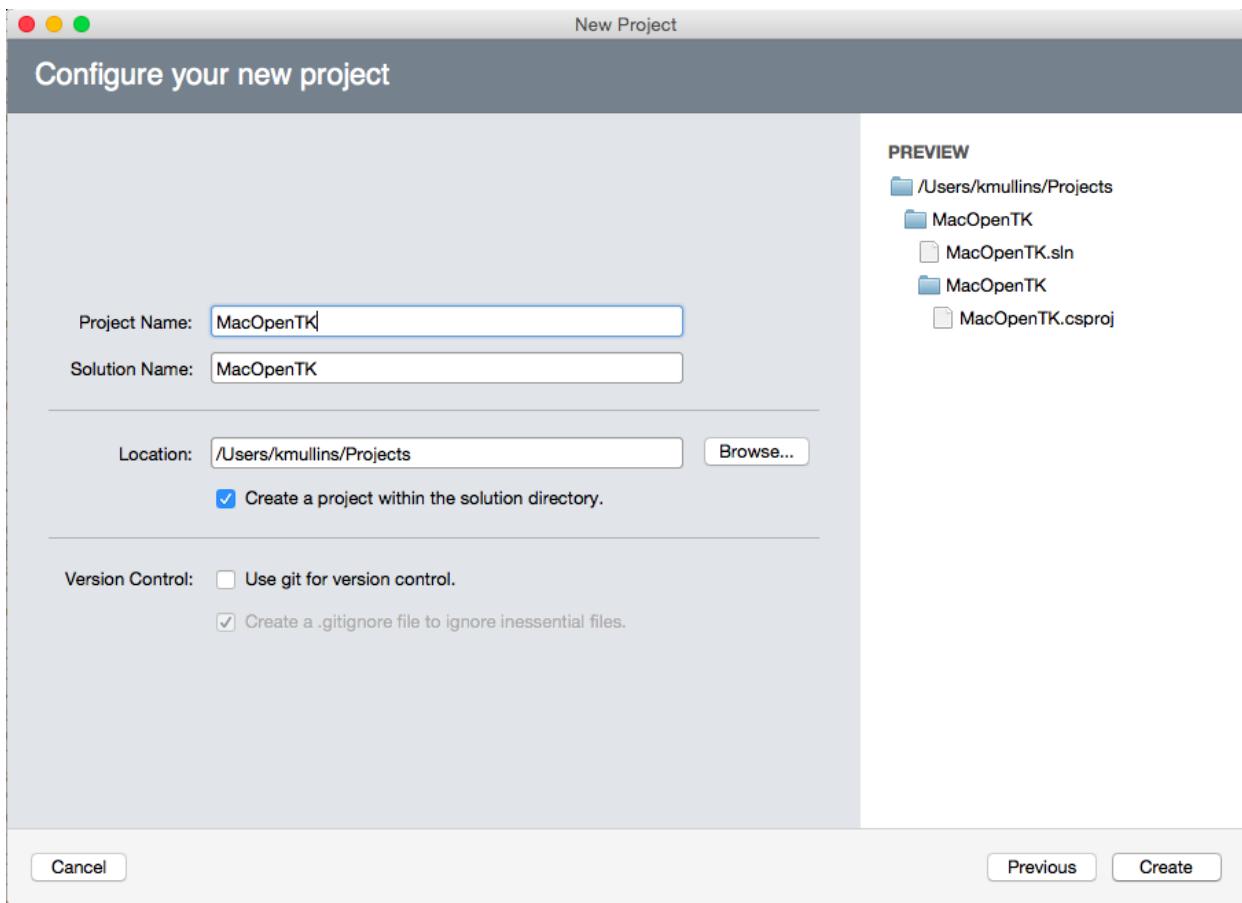
As a quick introduction to using OpenTK in a Xamarin.Mac app, we are going to create a simple application that opens a Game View, renders a simple triangle in that view and attaches the Game View to the Mac app's Main Window to display the triangle to the user.

Starting a New Project

Start Visual Studio for Mac and create a new Xamarin.Mac solution. Select **Mac > App > General > Cocoa App**:



Enter `MacOpenTK` for the Project Name:

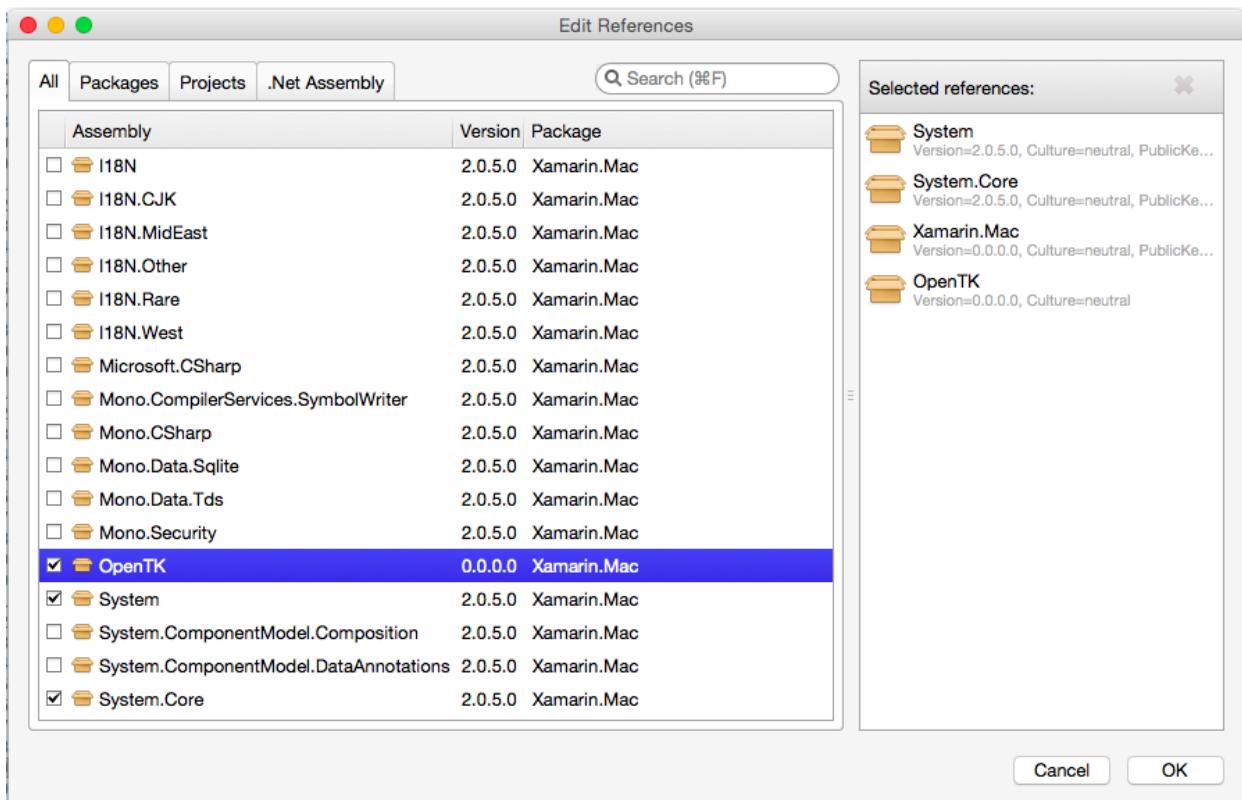


Click the Create button to build the new project.

Including OpenTK

Before you can use Open TK in a Xamarin.Mac application, you need to include a reference to the OpenTK assembly. In the **Solution Explorer**, right-click the **References** folder and select **Edit References....**.

Place a check by **OpenTK** and click the **OK** button:



Using OpenTK

With the new project created, double-click the `MainWindow.cs` file in the **Solution Explorer** to open it for editing. Make the `MainWindow` class look like the following:

```
using System;
using System.Drawing;
using OpenTK;
using OpenTK.Graphics;
using OpenTK.Graphics.OpenGL;
using OpenTK.Platform.MacOS;
using Foundation;
using AppKit;
using CoreGraphics;

namespace MacOpenTK
{
    public partial class MainWindow : NSWindow
    {
        #region Computed Properties
        public MonoMacGameView Game { get; set; }
        #endregion

        #region Constructors
        public MainWindow (IntPtr handle) : base (handle)
        {
        }

        [Export ("initWithCoder:")]
        public MainWindow (NSCoder coder) : base (coder)
        {
        }
        #endregion

        #region Override Methods
        public override void AwakeFromNib ()
        {
            base.AwakeFromNib ();

            // Create new Game View and replace the window content with it
            Game = new MonoMacGameView(ContentView.Frame);
            ContentView = Game;

            // Wire-up any required Game events
            Game.Load += (sender, e) =>
            {
                // TODO: Initialize settings, load textures and sounds here
            };

            Game.Resize += (sender, e) =>
            {
                // Adjust the GL view to be the same size as the window
                GL.Viewport(0, 0, Game.Size.Width, Game.Size.Height);
            };

            Game.UpdateFrame += (sender, e) =>
            {
                // TODO: Add any game logic or physics
            };

            Game.RenderFrame += (sender, e) =>
            {
                // Setup buffer
                GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);
                GL.MatrixMode(MatrixMode.Projection);

                // Draw a simple triangle
                GLLoadIdentity();
            };
        }
    }
}
```

```

        GL.Ortho(-1.0, 1.0, -1.0, 1.0, 0.0, 4.0);
        GL.Begin(BeginMode.Triangles);
        GL.Color3(Color.MidnightBlue);
        GL.Vertex2(-1.0f, 1.0f);
        GL.Color3(Color.SpringGreen);
        GL.Vertex2(0.0f, -1.0f);
        GL.Color3(Color.Ivory);
        GL.Vertex2(1.0f, 1.0f);
        GL.End();

    };

    // Run the game at 60 updates per second
    Game.Run(60.0);
}
#endifregion
}
}

```

Let's go over this code in detail below.

Required APIs

Several references are required to use OpenTK in a Xamarin.Mac class. At the start of the definition we have included the following `using` statements:

```

using System;
using System.Drawing;
using OpenTK;
using OpenTK.Graphics;
using OpenTK.Graphics.OpenGL;
using OpenTK.Platform.MacOS;
using Foundation;
using CoreGraphics;

```

This minimal set will be required for any class using OpenTK.

Adding the Game View

Next we need to create a Game View to contain all of our interaction with OpenTK and display the results. We used the following code:

```

public MonoMacGameView Game { get; set; }
...

// Create new Game View and replace the window content with it
Game = new MonoMacGameView(ContentView.Frame);
ContentView = Game;

```

Here we've made the Game View the same size as our Main Mac Window and replaced the Content View of the window with the new `MonoMacGameView`. Because we replaced the existing window content, our Game View will be automatically resized when the Main Windows is resized.

Responding to Events

There are several default events that each Game View should respond to. In this section will cover the main events required.

The Load Event

The `Load` event is the place to load resources from disk such as images, textures or music. For our simple, test app, we are not using the `Load` event, but have included it for reference:

```
Game.Load += (sender, e) =>
{
    // TODO: Initialize settings, load textures and sounds here
};
```

The Resize Event

The `Resize` event should be called every time the Game View is resized. For our sample app, we are making the GL Viewport the same size as our Game View (which is be auto resized by the Mac Main Window) with the following code:

```
Game.Resize += (sender, e) =>
{
    // Adjust the GL view to be the same size as the window
    GL.Viewport(0, 0, Game.Size.Width, Game.Size.Height);
};
```

The UpdateFrame Event

The `UpdateFrame` event is used to handle user input, update object positions, run physics or AI calculations. For our simple, test app, we are not using the `UpdateFrame` event, but have included it for reference:

```
Game.UpdateFrame += (sender, e) =>
{
    // TODO: Add any game logic or physics
};
```

IMPORTANT

The Xamarin.Mac implementation of OpenTK does not include the `Input API`, so you will need to use the Apple provided APIs to add keyboard and Mouse support. Optionally you can create a custom instance of the `MonoMacGameView` and override the `KeyDown` and `KeyUp` methods.

The RenderFrame Event

The `RenderFrame` event contains the code that is used to render (draw) your graphics. For our example app, we are filling the Game View with a simple triangle:

```

Game.RenderFrame += (sender, e) =>
{
    // Setup buffer
    GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);
    GL.MatrixMode(MatrixMode.Projection);

    // Draw a simple triangle
    GL.LoadIdentity();
    GL.Ortho(-1.0, 1.0, -1.0, 1.0, 0.0, 4.0);
    GL.Begin(BeginMode.Triangles);
    GL.Color3(Color.MidnightBlue);
    GL.Vertex2(-1.0f, 1.0f);
    GL.Color3(Color.SpringGreen);
    GL.Vertex2(0.0f, -1.0f);
    GL.Color3(Color.Ivory);
    GL.Vertex2(1.0f, 1.0f);
    GL.End();

};


```

Typically the render code will begin with a call to `GL.Clear` to remove any existing elements before drawing the new elements.

IMPORTANT

For the Xamarin.Mac version of OpenTK do **not** call the `SwapBuffers` method of your `MonoMacGameView` instance at the end of your rendering code. Doing so will cause the Game View to strobe rapidly instead of displaying your rendered view.

Running the Game View

With all of the required events defined and the Game View attached to the Main Mac Window of our app, we are ready to run the Game View and display our graphics. Use the following code:

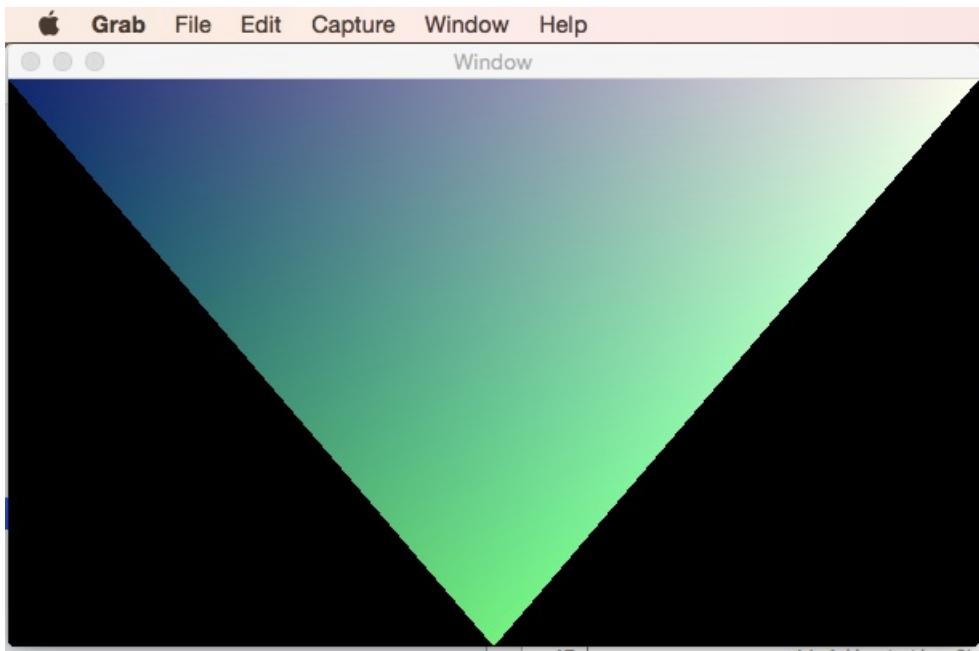
```

// Run the game at 60 updates per second
Game.Run(60.0);

```

We pass in the desired frame rate that we want the Game View to update at, for our example we have chosen `60` frames per second (the same refresh rate as normal TV).

Let's run our app and see the output:



If we resize our window, the Game View will also be resized and the triangle will be resized and updated real-time as well.

Where to Next?

With the basics of working with OpenTK in a Xamarin.mac application done, here are some suggestions of what to try out next:

- Try changing the color of the triangle and the background color of the Game View in the `Load` and `RenderFrame` events.
- Make the triangle change color when the user press a key in the `UpdateFrame` and `RenderFrame` events or make your own custom `MonoMacGameView` class and override the `KeyUp` and `KeyDown` methods.
- Make the triangle move across the screen using the aware keys in the `UpdateFrame` event. Hint: use the `Matrix4.CreateTranslation` method to create a translation matrix and call the `GL.LoadMatrix` method to load it in the `RenderFrame` event.
- Use a `for` loop to render several triangles in the `RenderFrame` event.
- Rotate the camera to give a different view of the triangle in 3D space. Hint: use the `Matrix4.CreateTranslation` method to create a translation matrix and call the `GL.LoadMatrix` method to load it. You can also use the `Vector2`, `Vector3`, `Vector4` and `Matrix4` classes for camera manipulations.

For more examples, please see the [OpenTK Samples GitHub](#) repo. It contains an official list of examples of using OpenTK. You'll have to adapt these examples for using with the Xamarin.Mac version of OpenTK.

For a more complex Xamarin.Mac example of an OpenTK implementation, please see our [MonoMacGameView](#) sample.

Summary

This article has taken a quick look at working with OpenTK in a Xamarin.Mac application. We saw how to create a Game Window, how to attach the Game Window to a Mac Window and how to render a simple shape in the Game Window.

Related Links

- [MacOpenTK \(sample\)](#)
- [MonoMacGameView \(sample\)](#)

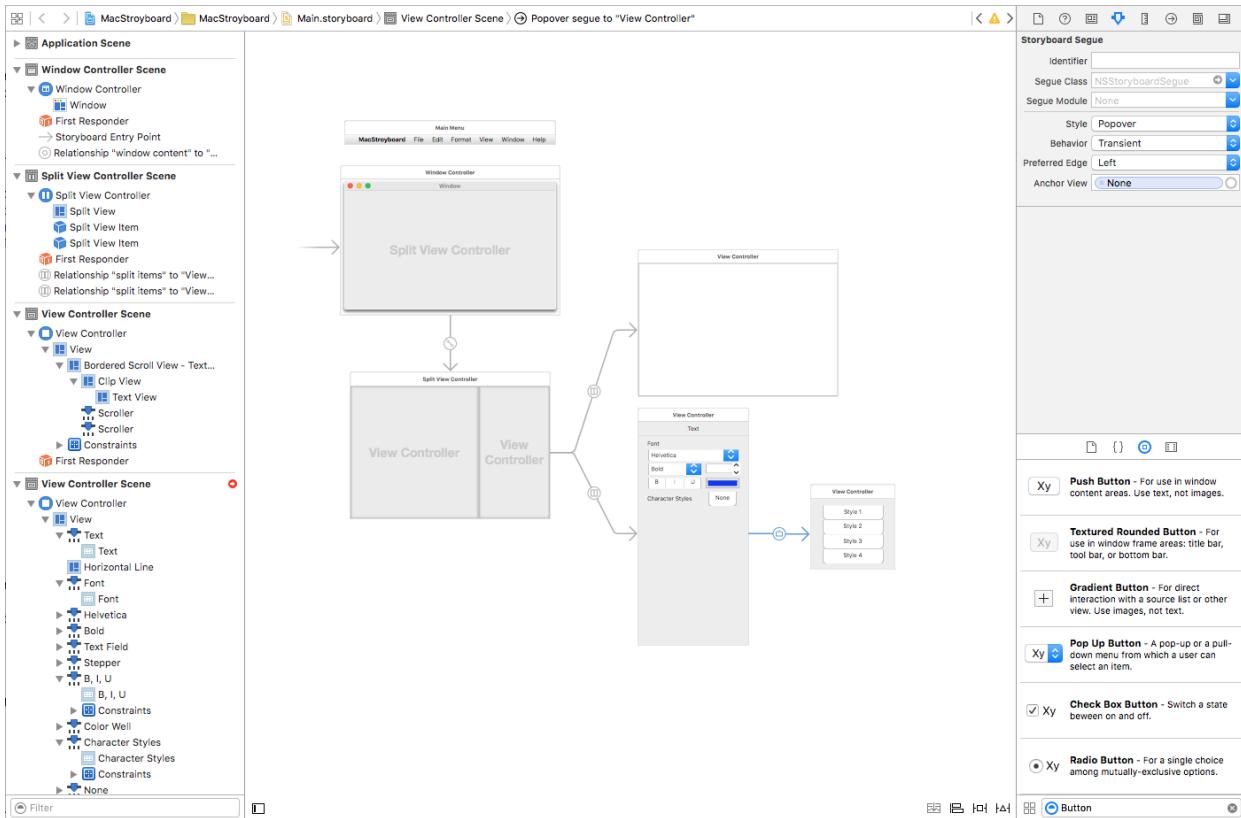
- [Hello, Mac](#)
- [Working with Windows](#)
- [The Open Toolkit](#)
- [OS X Human Interface Guidelines](#)
- [Introduction to Windows](#)

Introduction to Storyboards in Xamarin.Mac

11/2/2020 • 4 minutes to read • [Edit Online](#)

This article provides an introduction to working with Storyboards in a Xamarin.Mac app. It covers creating and maintaining the app's UI using storyboards and Xcode's Interface Builder.

Storyboards allow you to develop a User Interface for your Xamarin.Mac app that not only includes the window definitions and controls, but also contains the links between different windows (via segues) and view states.



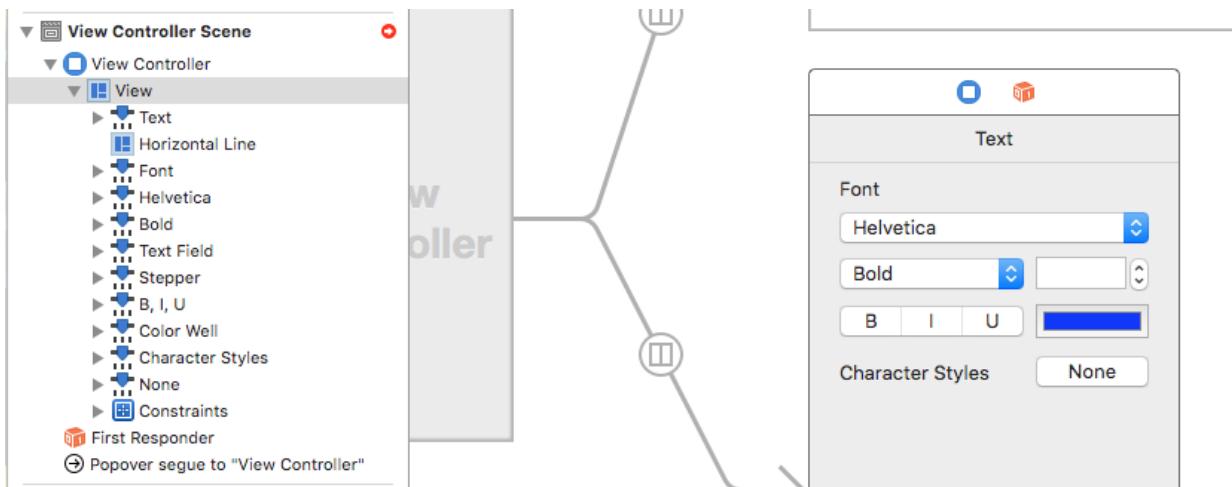
This article will provide an introduction to using Storyboards to define a Xamarin.Mac app's user Interface.

What are Storyboards?

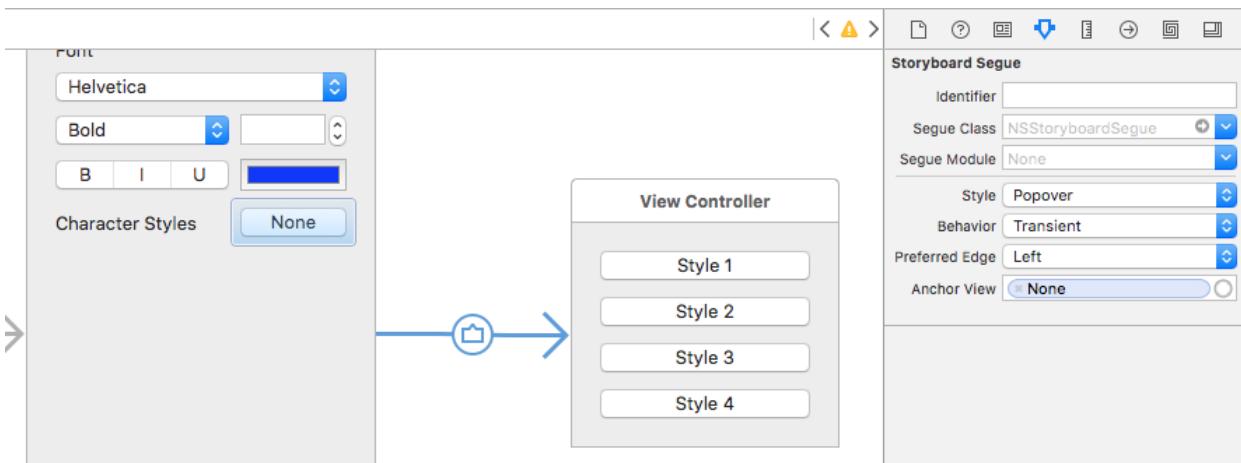
By using Storyboards, all of a Xamarin.Mac app's UI can be defined in a single location with all of the navigation between its individual elements and user interfaces. Storyboards for Xamarin.Mac, work in a very similar fashion to Storyboards for Xamarin.iOS. However, they contain a different set of *Segue Types* because of the different interface idioms.

Working with Scenes

As stated above, a Storyboard defines all of the UI for a given app broken down into a functional overview of its *View Controllers*. In Xcode's Interface Builder, each of these controllers lives in its own *Scene*.



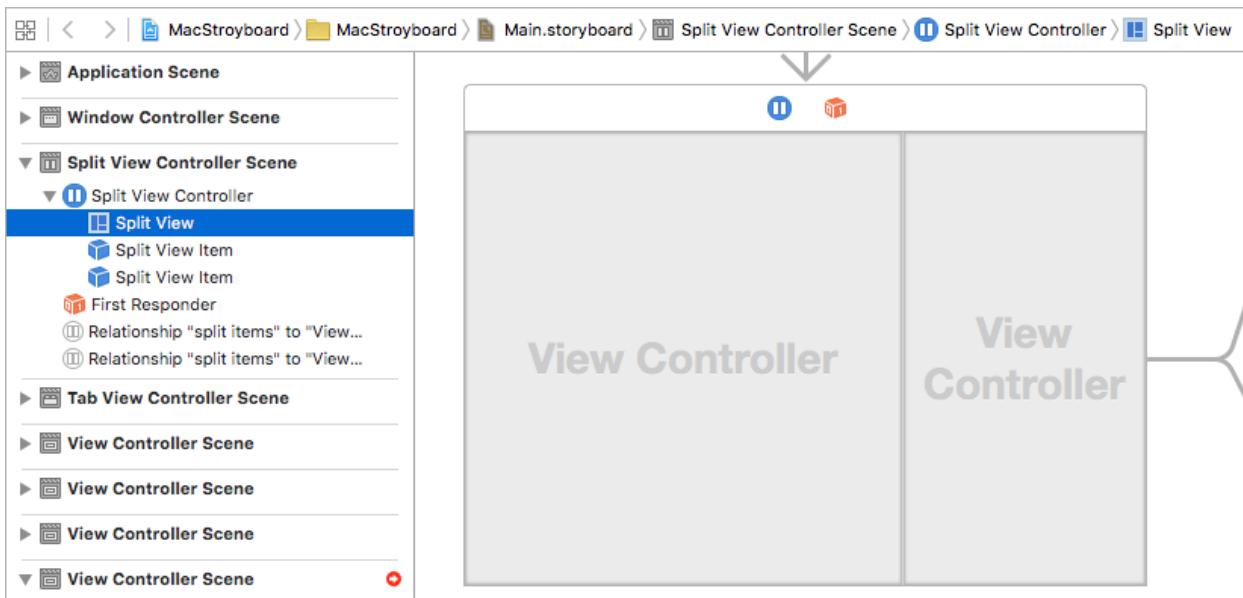
Each Scene represents a given View and View Controller Pair with a set of lines (called Segues) that connect each Scene in the UI, thus showing their relationships. Some Segues define how one View Controller contains one or more child Views or View Controllers. Other Segues, define transitions between View Controller (such as displaying a popover or dialog box).



The most important thing to note is that each Segue represents the flow of some form of data between the given element of the app's UI.

Working with View Controllers

View Controllers define the relationships between a given View of information within a Mac app and the data model that provides that information. Each top level scene in the Storyboard represents one View Controller in the Xamarin.Mac app's code.



In this way, each View Controller is a self-contained, reusable pairing of both the information's visual representation (View) and the logic to present and control that information.

Within a given Scene, you can do all of the things that would normally have been handled by individual `.xib` files:

- Place subviews and controls (such as buttons and text boxes).
- Define element positions and auto layout constraints.
- Wire-up Actions and Outlets to expose UI elements to code.

Working with Segues

As stated above, Segues provide the relationships between all of the Scenes that define your app's UI. If you are familiar with working in Storyboards in iOS, you know that Segues for iOS usually define transitions between full screen views. This differs from macOS, when Segues usually define "containment" (where one Scene is the child of a parent Scene).

In macOS, most apps tend to group their views together within the same window using UI elements such as Split Views and Tabs. Unlike iOS, where views need to be transitioned on and off screen, due to limited physical display space.

Given macOS's tendencies towards containment, there are situations where *Presentation Segues* are used, such as Modal Windows, Sheet Views and Popovers.

When using Presentation Segues, you can override the `PrepareForSegue` method of the parent View Controller for presentation to initialize and variables and provide any data to the View Controller being presented.

Design and Run Times

At Design time (when layout out the UI in Xcode's Interface Builder), each element of the app's UI is broken down into its constituent items:

- **Scenes** - Which are composed of:
 - **View Controller** - That define the relationships between Views and the data that support them.
 - **Views and Subviews** - The actual elements that make up the user interface.
 - **Containment Segues** - That define the parent-child relationships between Scenes.
- **Presentation Segues** - That define individual presentation modes.

By defining each element in this way, it allows for the lazy-loading of each element only as it is needed during runtime. In macOS, the entire process was designed to allow the developer to create complex, flexible User

Interfaces that require a bare minimum of backing code to make them work, all while being as efficient with system resources as possible.

Storyboard Quick Start

In the [Storyboard Quick Start](#) guide, we'll create a simple Xamarin.Mac app that introduces the key concepts of working with storyboards to create a User Interface. The sample app will consist of a Split View containing a *Content Area* and an *Inspector Area* and it will present a simple Preferences Dialog window. We'll be using Segues to tie all of the User Interface elements together.

Working with Storyboards

This section covers the in-depth details of [Working with Storyboards](#) in a Xamarin.Mac app. We take an in-depth look at Scenes and how they are composed of View Controllers and View. Then, we'll take a look at how Scenes are tied together with Segues. Finally, we'll take a look at working with custom Segue types.

Complex Storyboard Example

For an example of a complex example of working with Storyboards in a Xamarin.Mac app, please see the [SourceWriter Sample App](#). SourceWriter is a simple source code editor that provides support for code completion and simple syntax highlighting.

The SourceWriter code has been fully commented and, where available, links have been provided from key technologies or methods to relevant information in the Xamarin.Mac Guides Documentation.

Summary

This article has taken a quick look at working with Storyboards in a Xamarin.Mac app. We saw how to create a new app using storyboards and how to define a user interface. We also saw how to navigate between different windows and view states using segues.

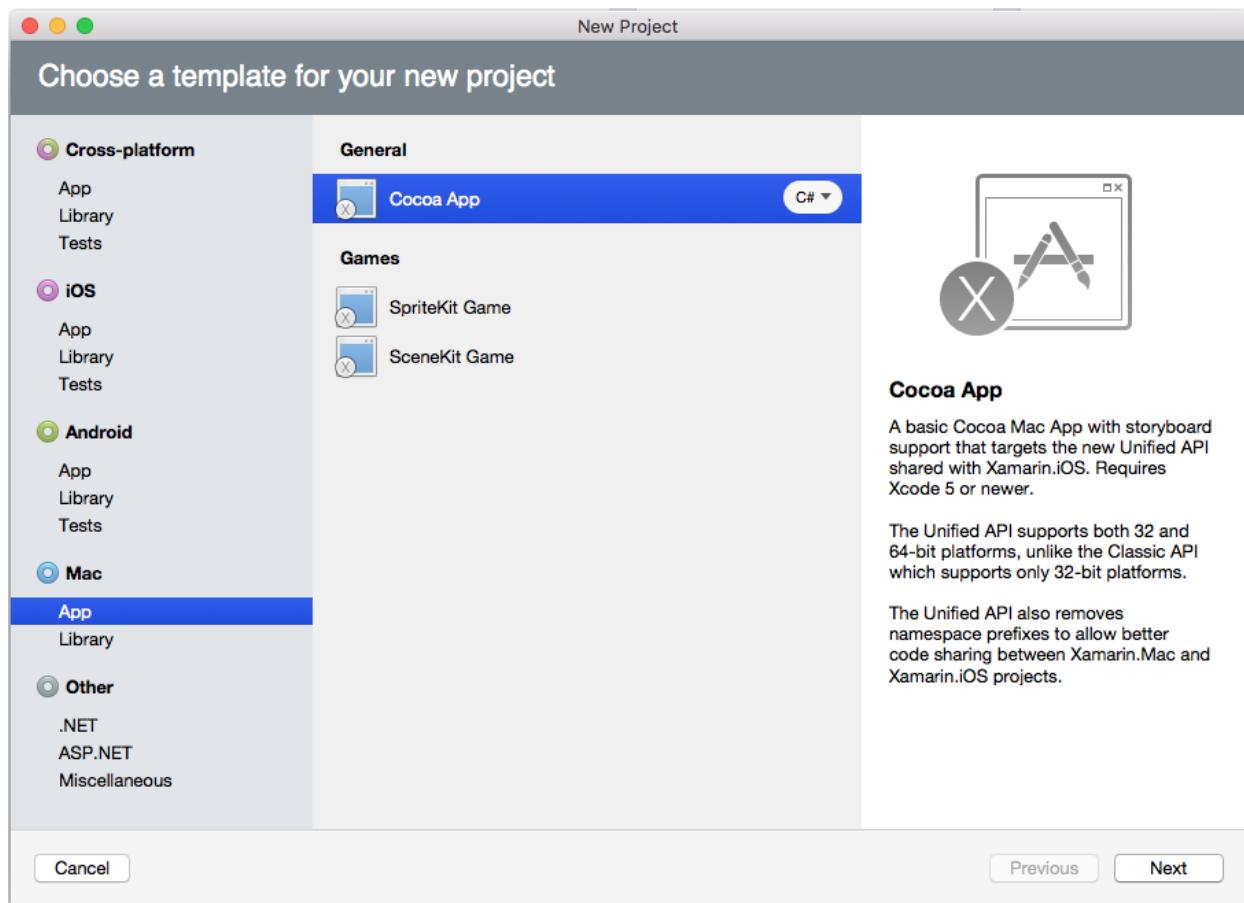
Related Links

- [Hello, Mac \(sample\)](#)
- [Hello, Mac](#)
- [Working with Windows](#)
- [OS X Human Interface Guidelines](#)
- [Introduction to Windows](#)

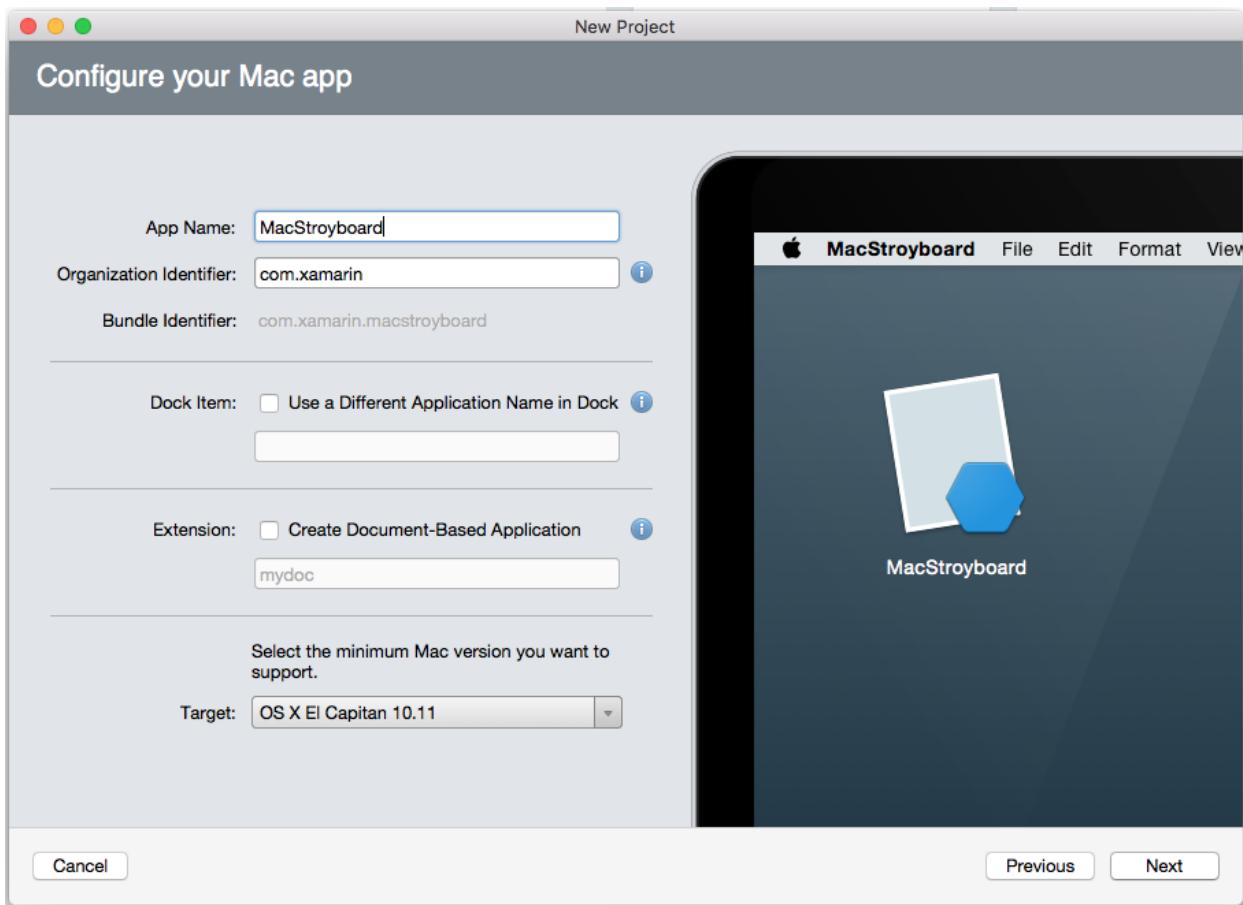
Storyboards in Xamarin.Mac – Quick Start

3/5/2021 • 4 minutes to read • [Edit Online](#)

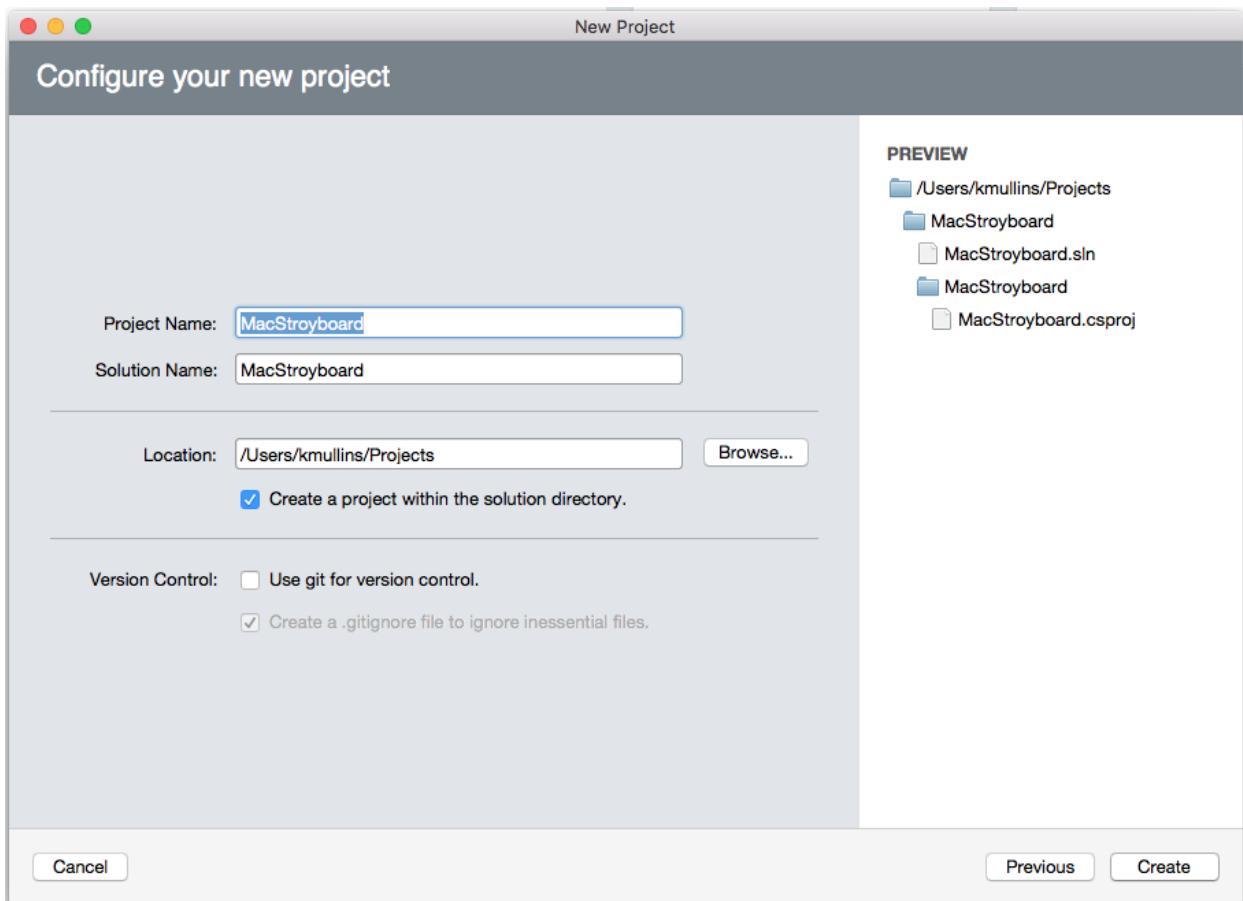
As a quick introduction to using Storyboards to define a Xamarin.Mac app's User Interface, let's start a new Xamarin.Mac project. Select **Mac > App > Cocoa App** and click the **Next** button:



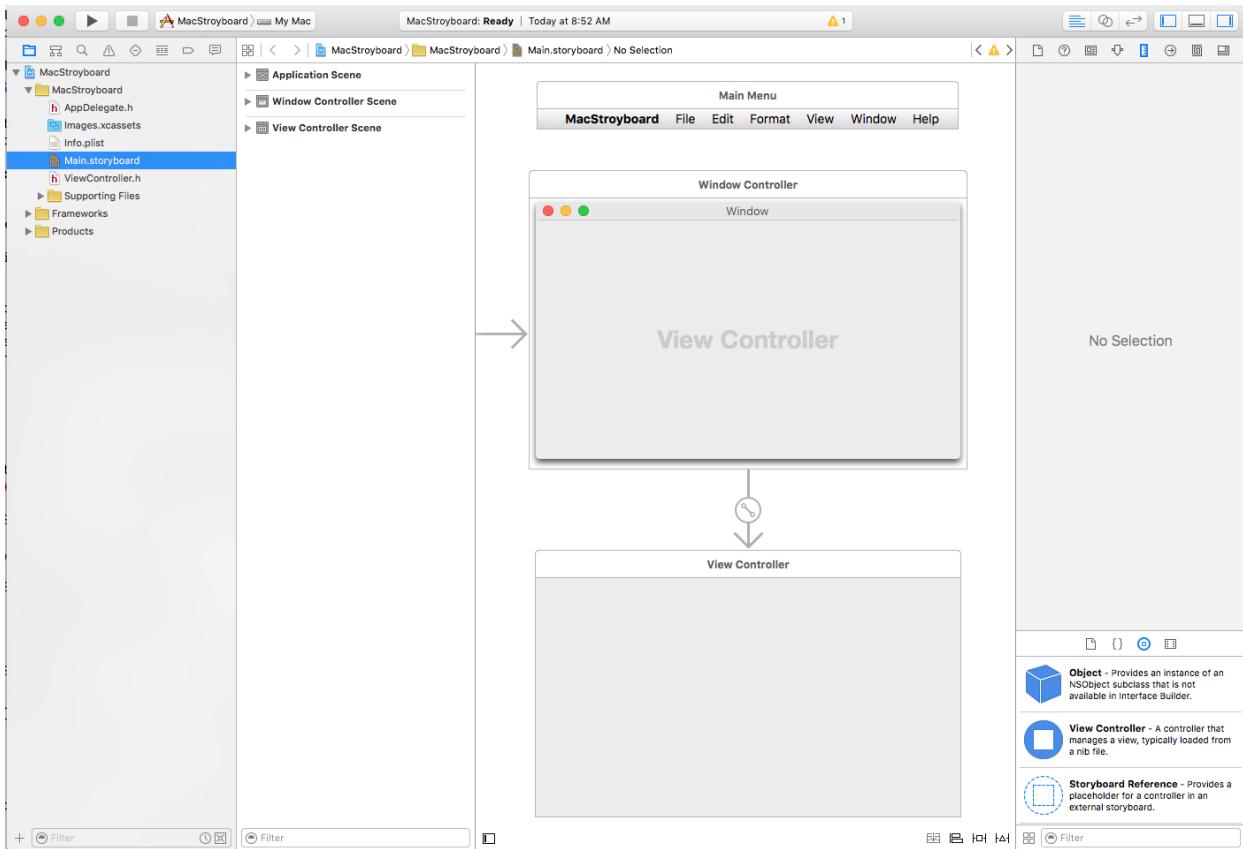
Use the **App Name** of `MacStoryboard` and click the **Next** button:



Use the default Project Name and Solution Name and click the Create button:

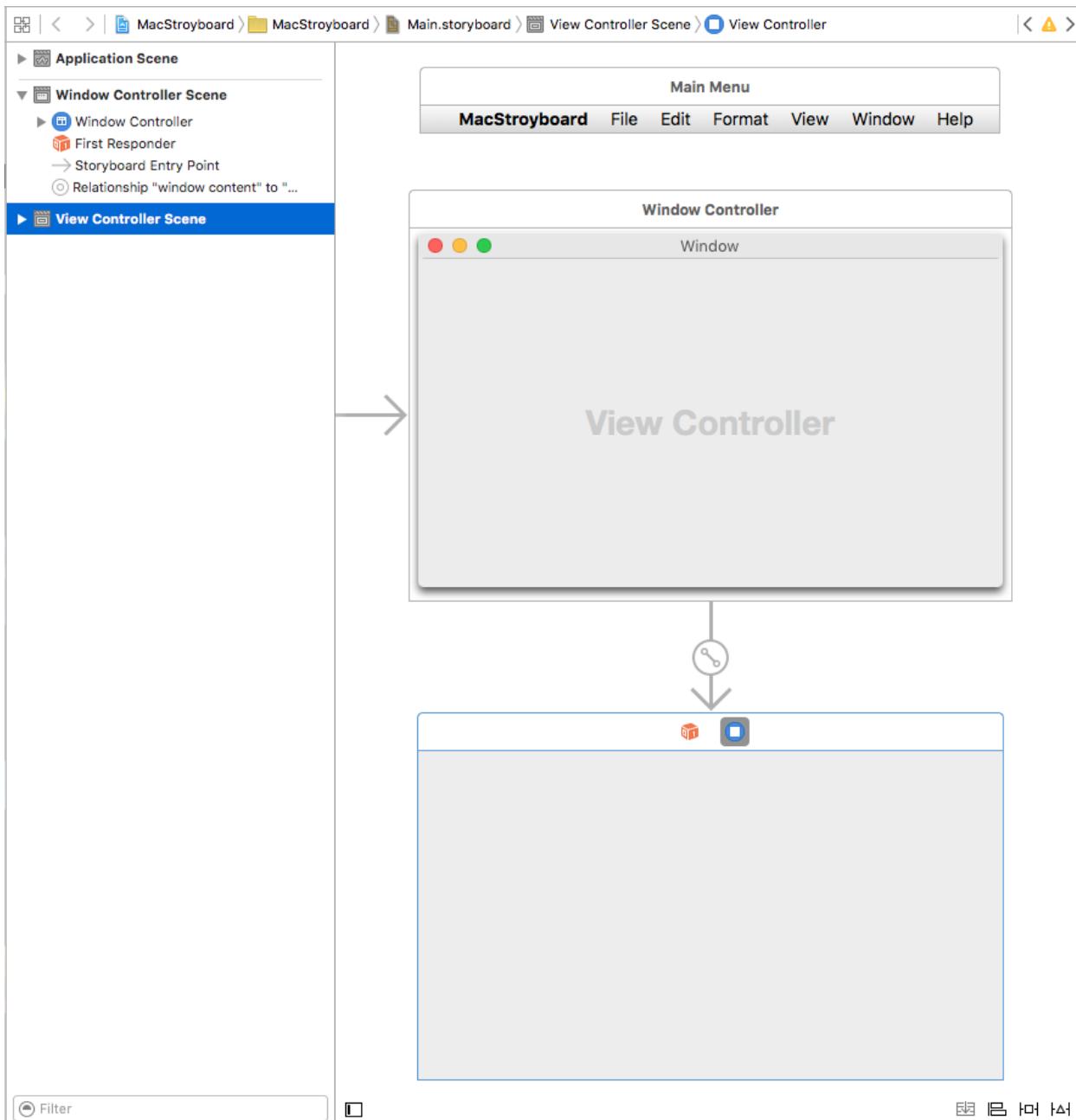


In the Solution Explorer, double-click the `Main.storyboard` file to open it for editing in Xcode's Interface Builder:

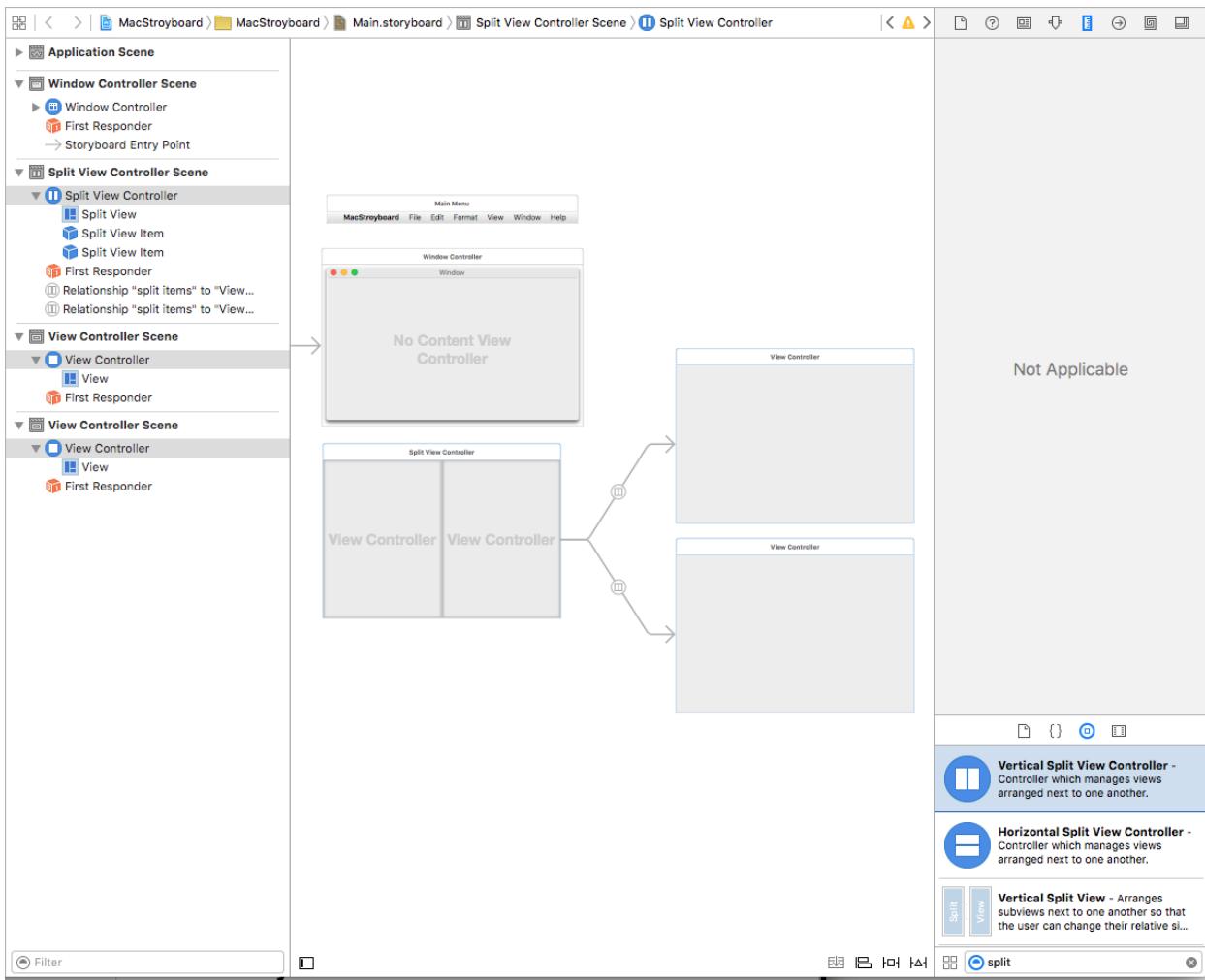


As you can see above, the default Storyboard defines both the app's Menu Bar and its main Window with its View Controller and View. For our sample app, we are going to be creating a UI that has a main *Content View* on one side and an *Inspector View* in the second.

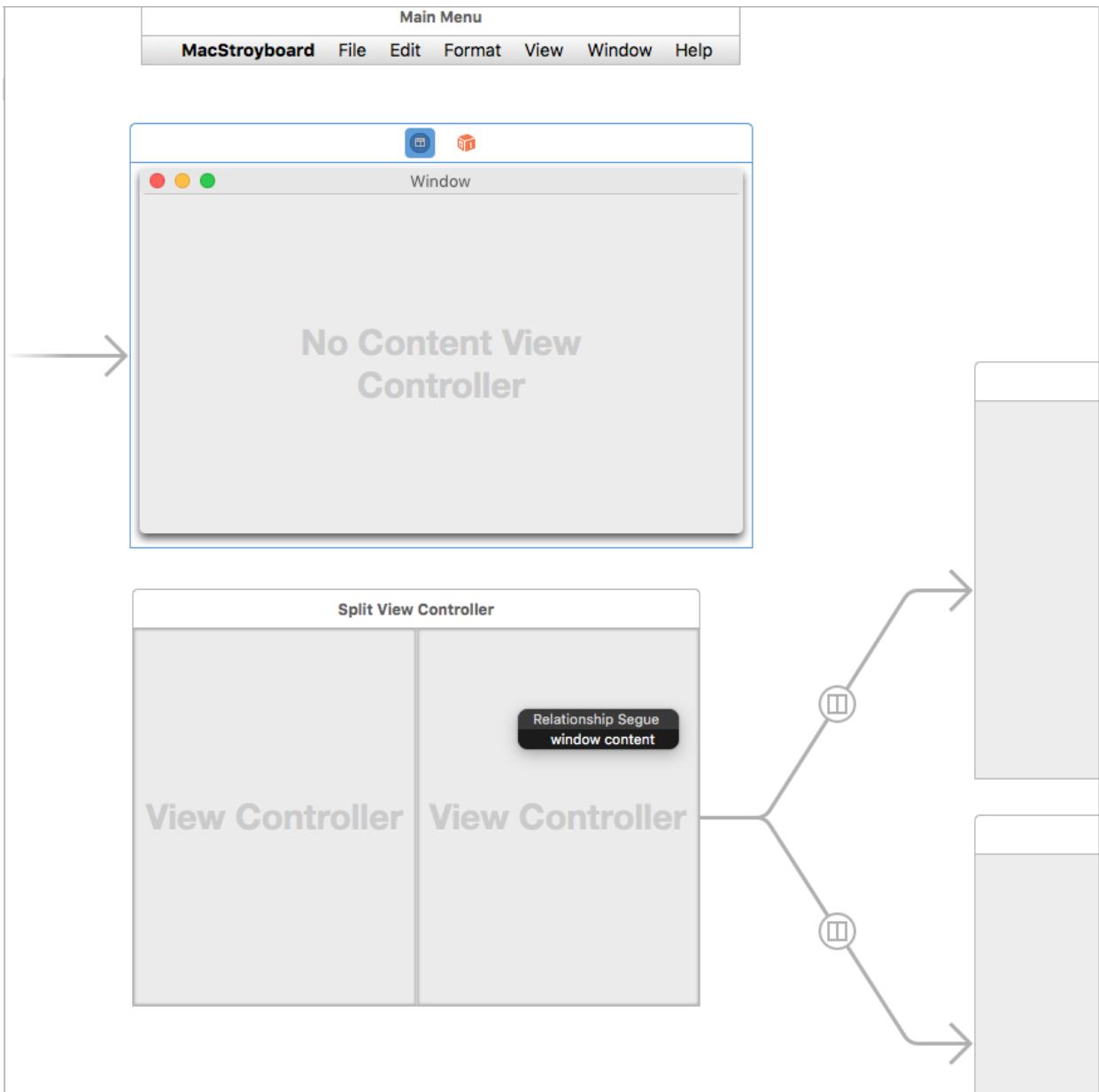
To do this, we will need to first remove the default View Controller and View that comes with the Storyboard by select it in Interface Builder and pressing the **Delete** key:



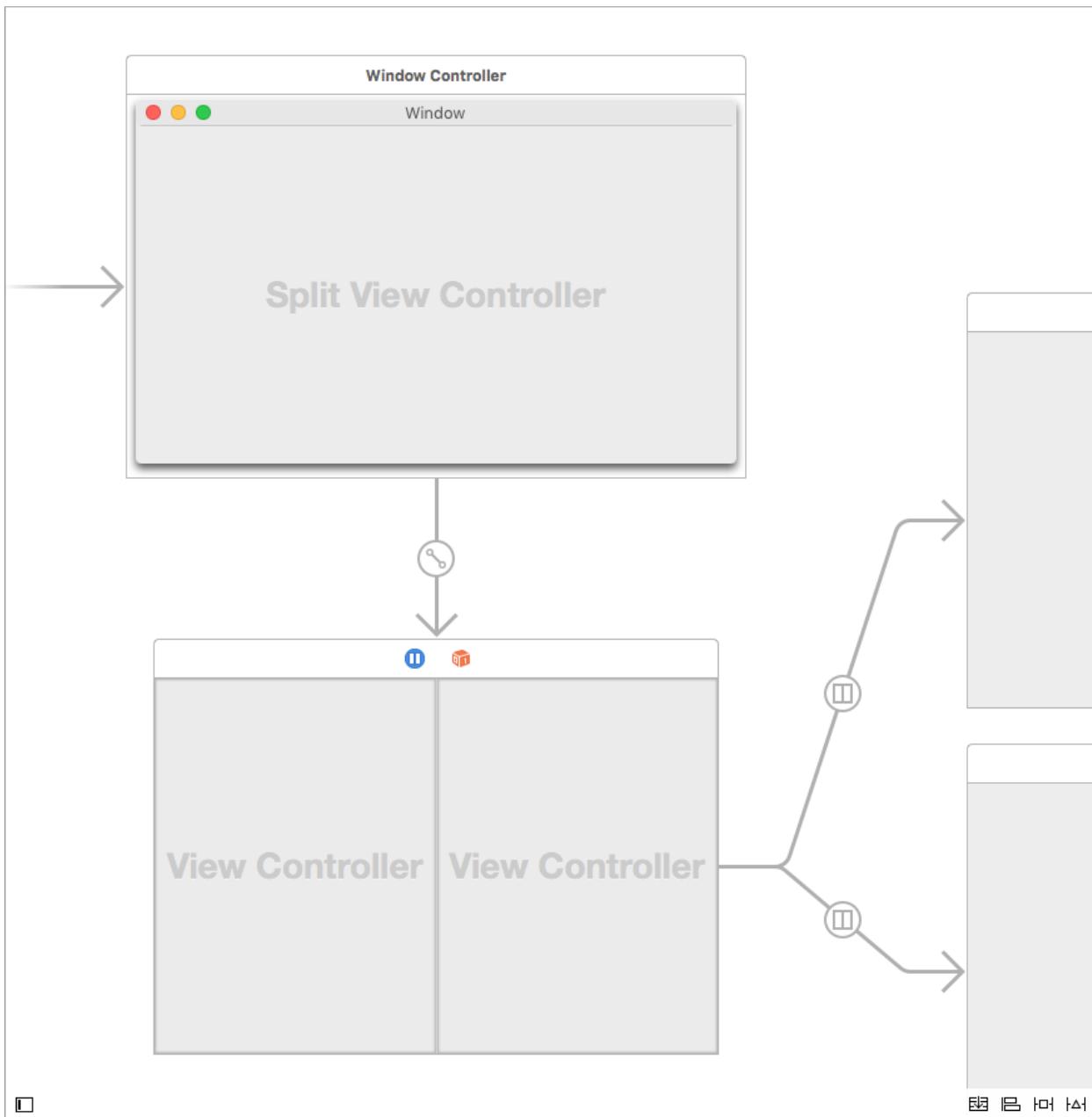
Next, type `split` into the **Filter** area, select the Vertical Split View Controller and drag it onto the *Design Surface*:



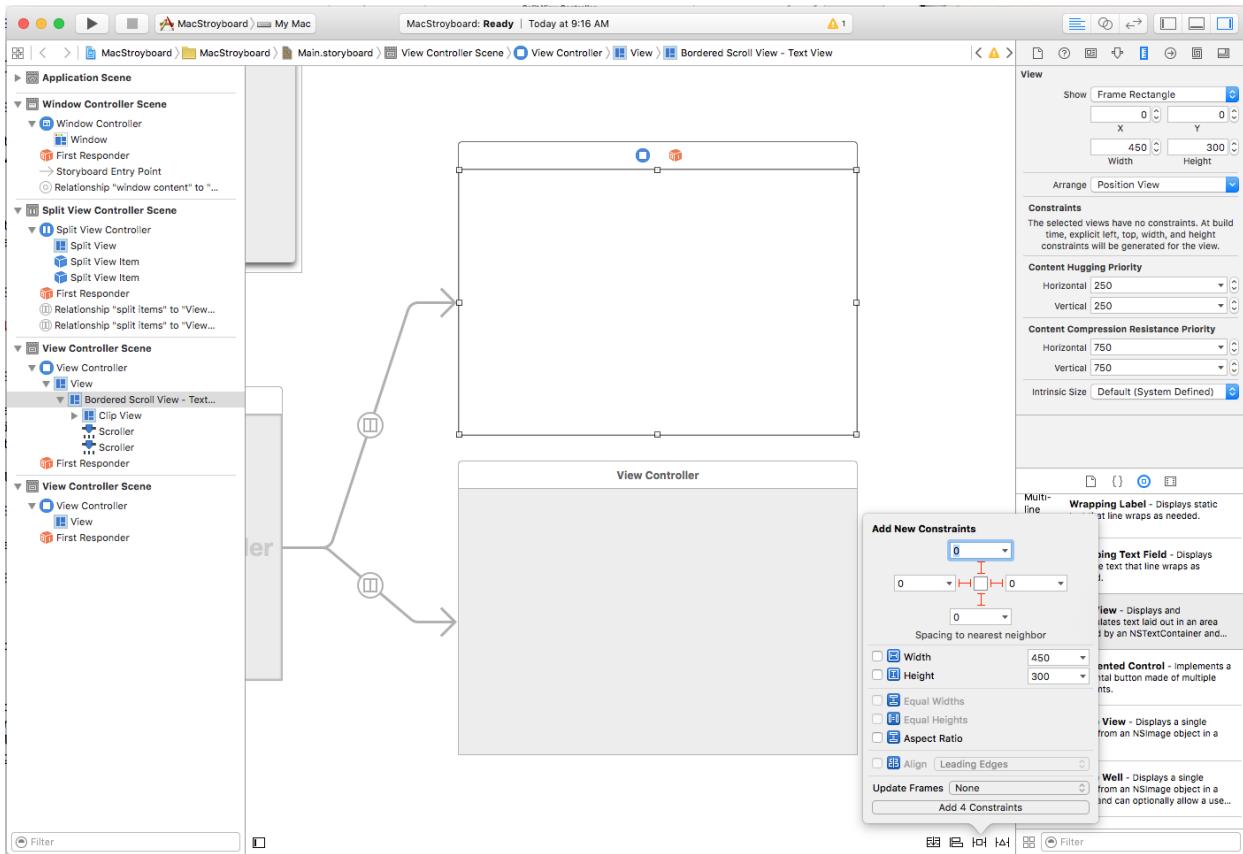
Notice that the controller automatically included two child View Controllers (and their related views), wired-up to the left and right sides of the split view. To tie the split view to its parent window, press the **Control** key, click on the Window Controller (the blue circle in the Window Controller's frame) and drag a line to the Split View Controller. Select **window content** from the popup:



This will tie the two interface elements together using a Segue:

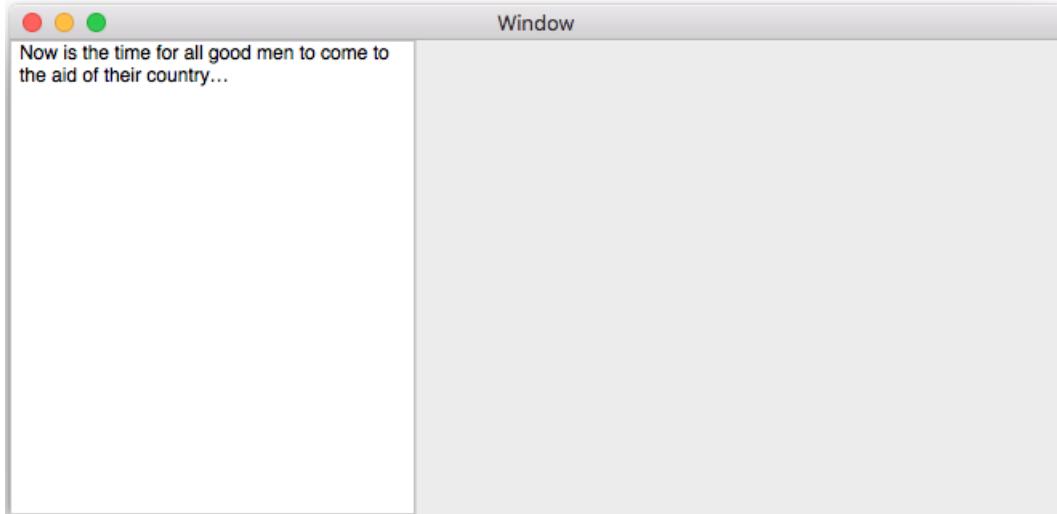


We want to place a Text View in the left side of the Split View and have it automatically fill the available area when either the Window or the Split View is resized. Drag a Text View onto the top View Controller attached to the Split View and click the **Pin** auto layout constraint (the second icon from the right at the bottom of the Design Surface).

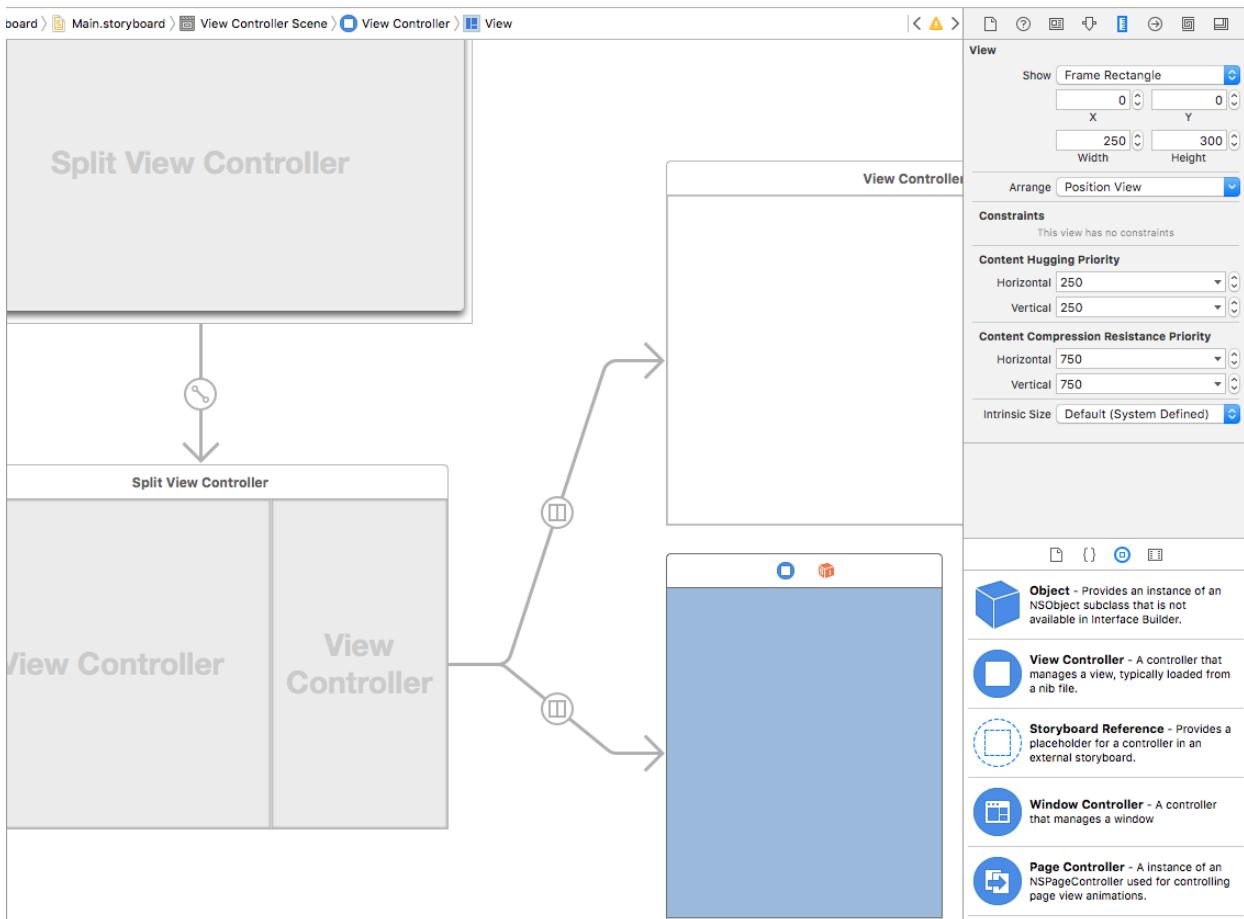


From here we will click all four of the I-Beam icons around the bounding box at the top of the Constraints Popover and click the **Add 4 Constraints** button at the bottom to add the required constraints.

If we return to Visual Studio for Mac and run the project, notice that the Text View automatically resizes to fill the left side of the Split View as the Window or the split are resized:



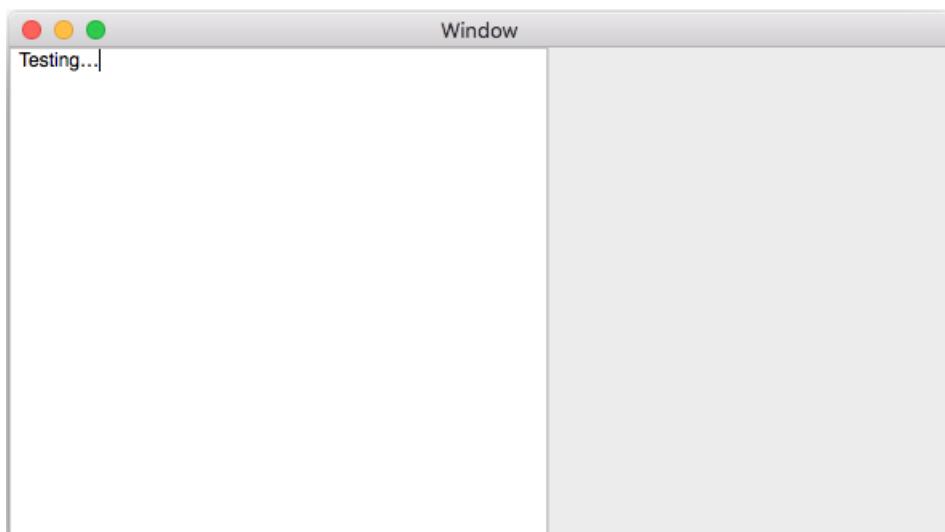
Since we are going to be using the right hand side of the split view as an Inspector area, we want it to have a smaller size and allow it to be collapsed. Return to Xcode and edit the View for the right side by selecting it in the Design Surface and clicking on the **Size Inspector**. From here enter a **Width of 250**:



Next select the Split Item that represents the right side, set a higher **Holding Priority** and click the **User Can Collapse** checkbox:



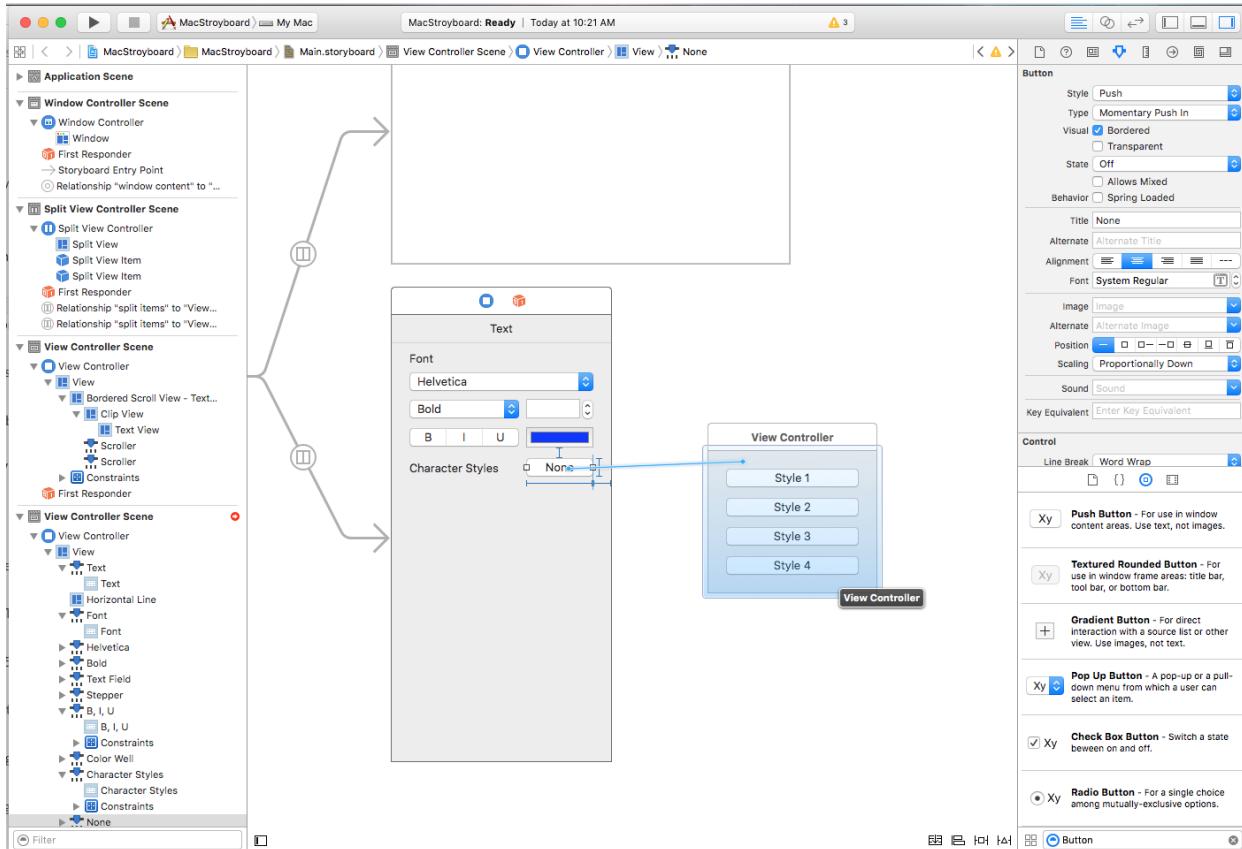
If we return to Visual Studio for Mac and run the project now, notice that the right side keeps its smaller size and the window is resized:



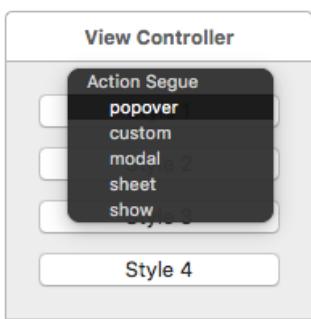
Defining a Presentation Segue

We are going to layout the right hand side of the Split View to act as an Inspector for the selected text's properties. We'll drag some controls onto the bottom view to layout the UI of the inspector. For the last control, we want to display a popover that allows the user to select from four preset character styles.

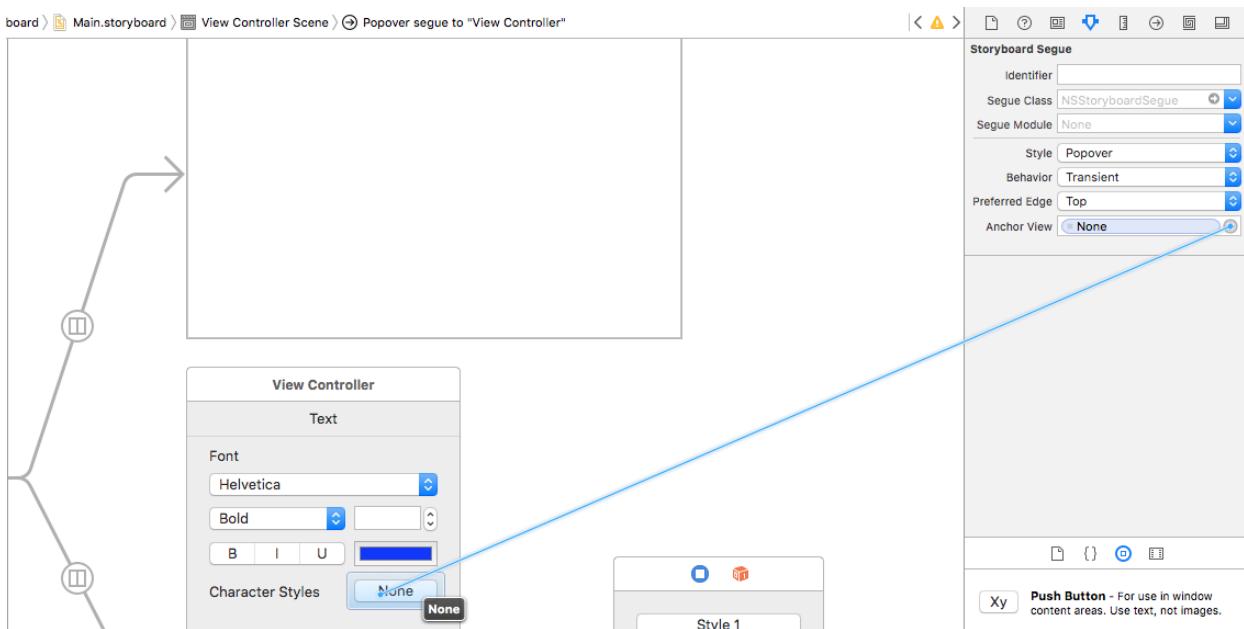
We'll add a Button to the Inspector and a View Controller to the Design Surface. We'll resize the View Controller to be the size that we want our Popover to be and add four Buttons to it. Next we'll **Control** key-click on the button in the Inspector View and drag to the View Controller that will represent our popover:



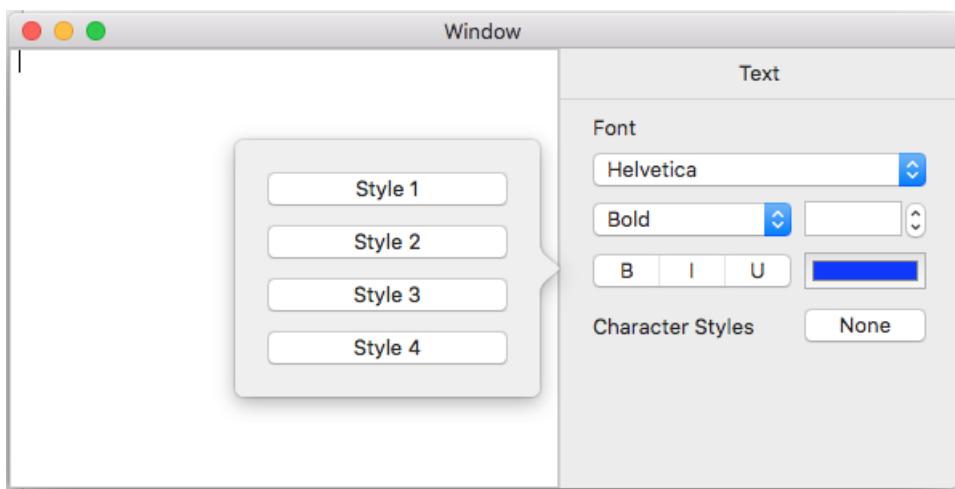
From the popup menu, we'll select **Popover**:



Finally, we'll select the Segue in the Design Surface and set the **Preferred Edge** to **Left**. Then, we'll drag a line from the **Anchor View** to the Button that we want the popover to be attached to:



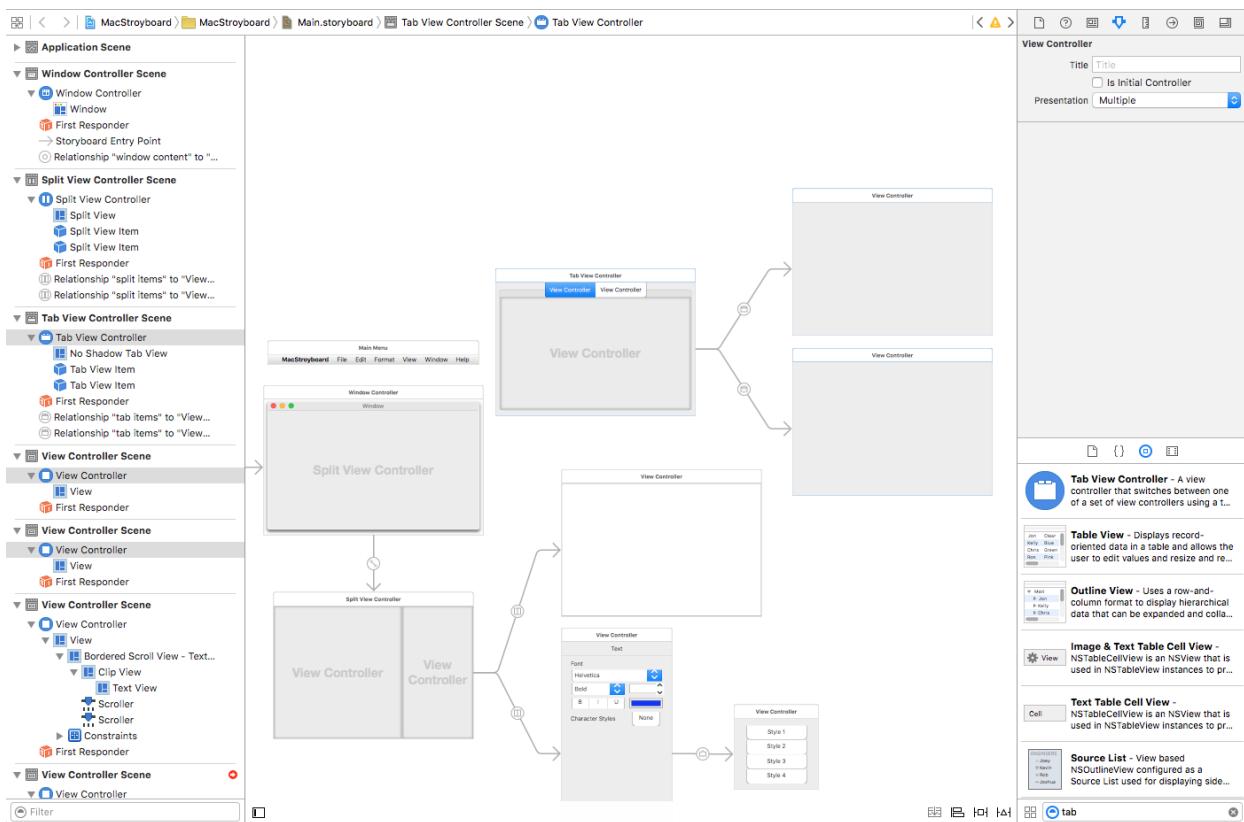
If we return to Visual Studio for Mac, run the app and click on the **None** button in the Inspector, the popover will be displayed:



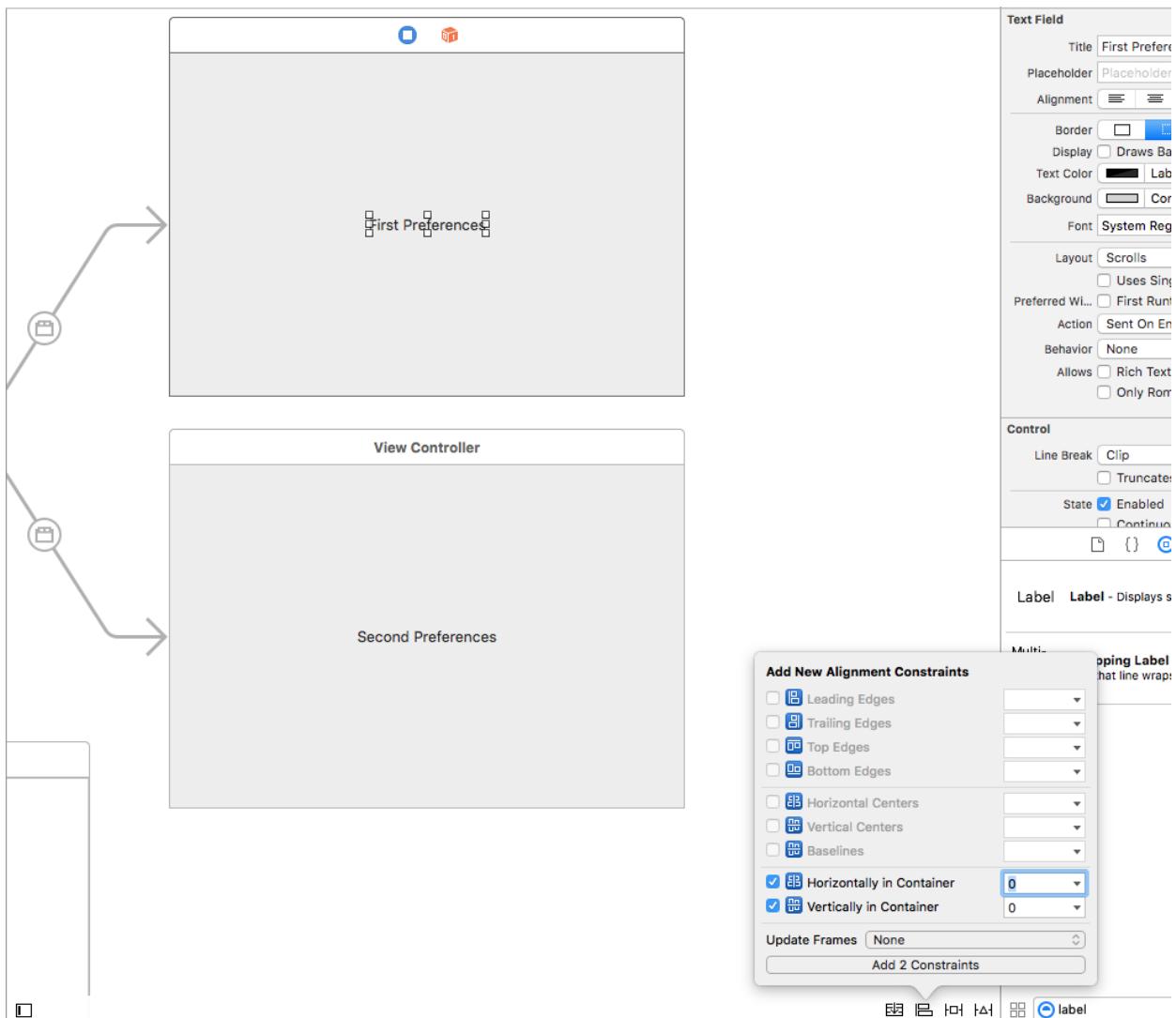
Creating App Preferences

Most standard macOS apps provide a *Preference Dialog* that allows the user to define several options that control various aspects of the app, such as appearance or user accounts.

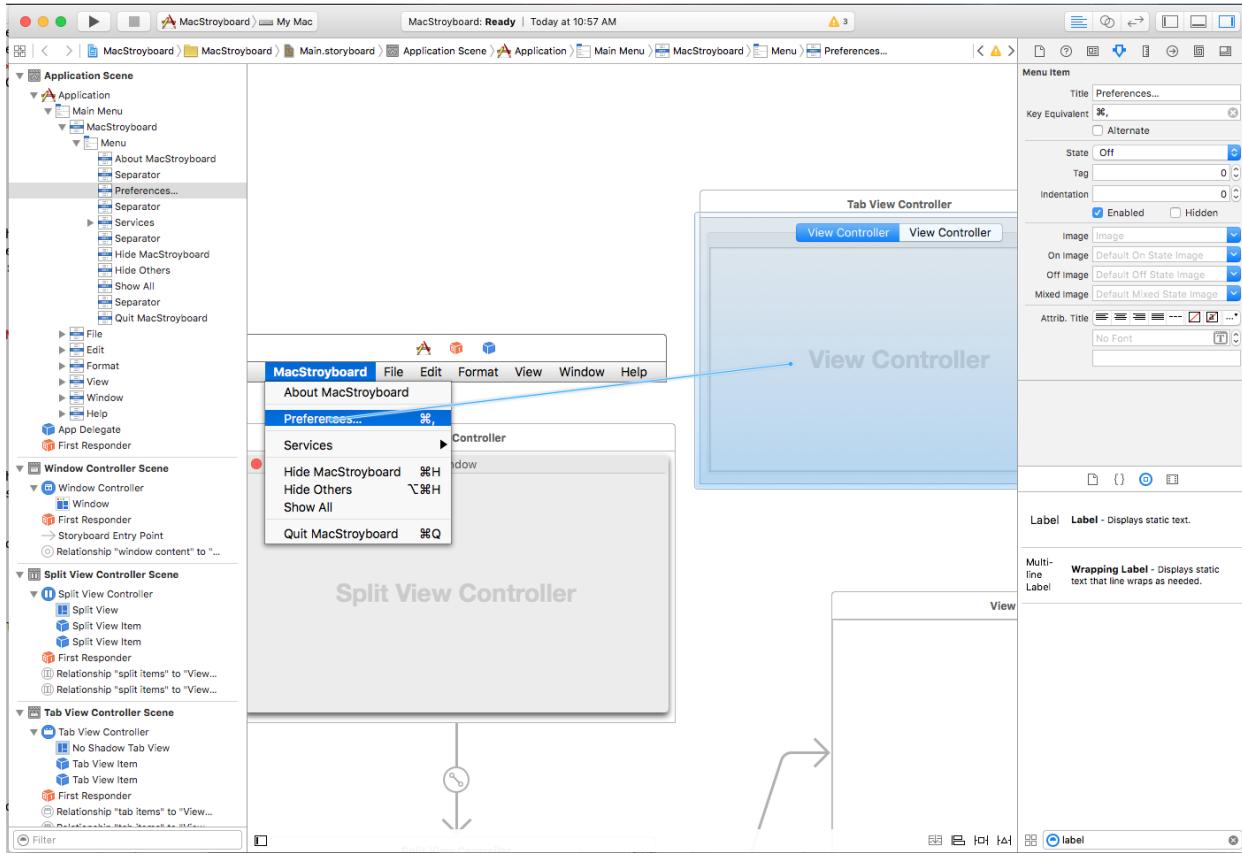
To define a standard Preference Dialog Window, first drag a Tab View Controller onto the Design Surface:



Again, this will automatically come with two child View Controllers attached. For example sake, we'll add a label to each view that will center inside of it:



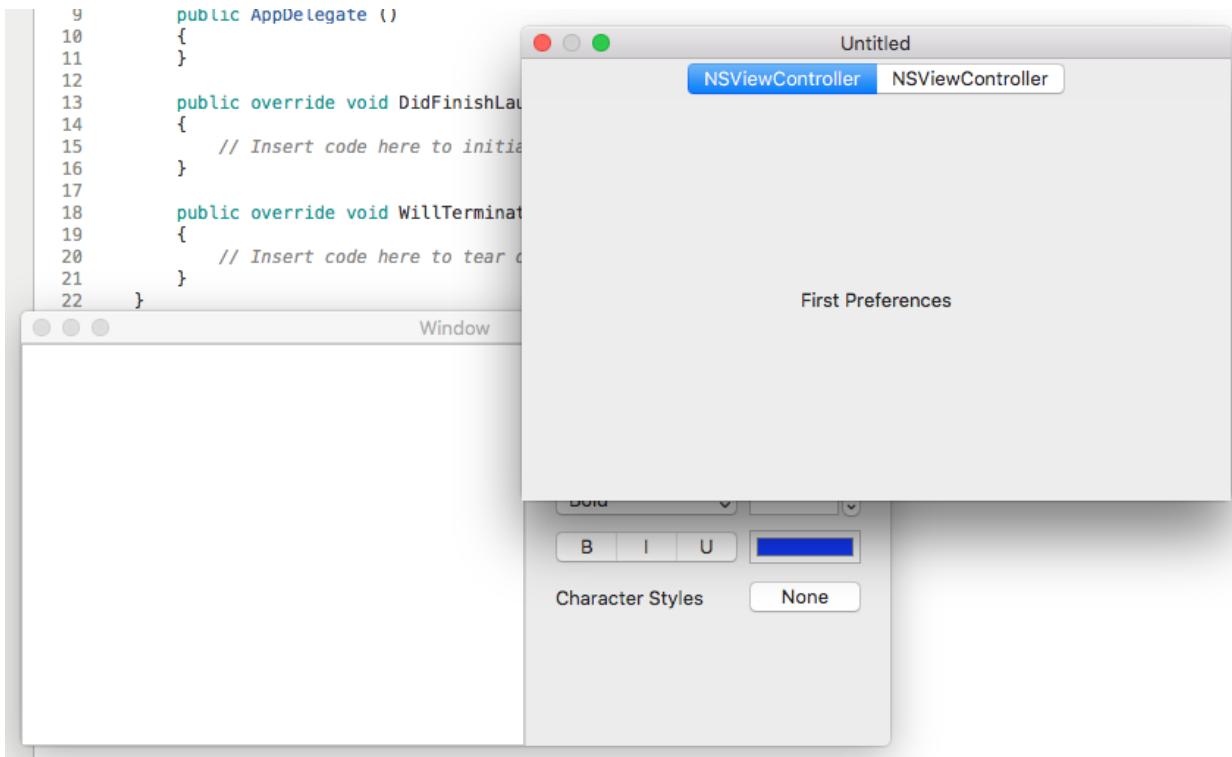
Next, we want to display the Preferences window when the user selects the **Preferences...** menu item. From the Menu Bar, select the preferences menu item, **Control** key-click and drag a line to our Tab View Controller:



From the popup, we'll select **Modal** to show this window as a Modal Dialog:

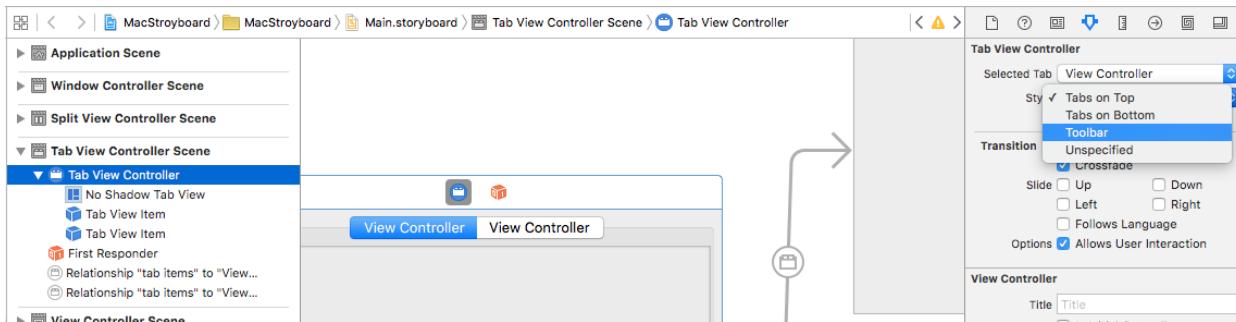


If we save our changes, return to Visual Studio for Mac, run the app and select the **Preferences...** menu item, our new Preferences dialog will be displayed:

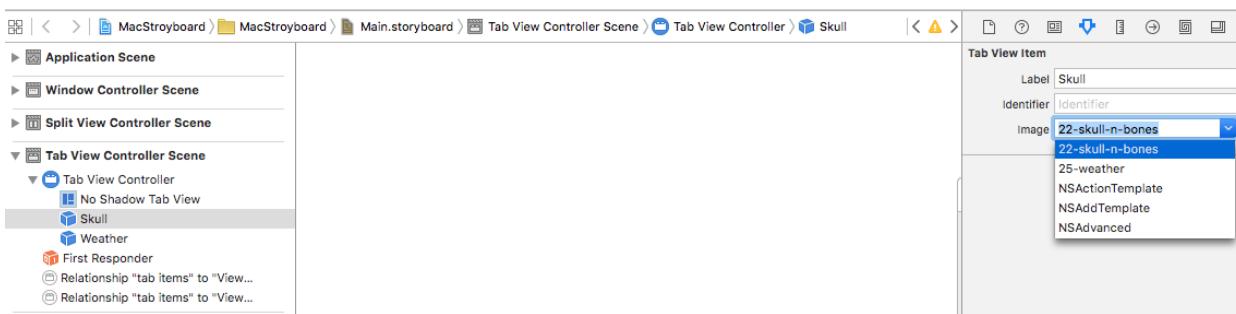


You might notice that this doesn't look like a standard macOS app Preference Dialog Window. To fix this, include two image files in the Xamarin.Mac app's `Resources` folder in the **Solution Explorer** and return to Xcode's Interface Builder.

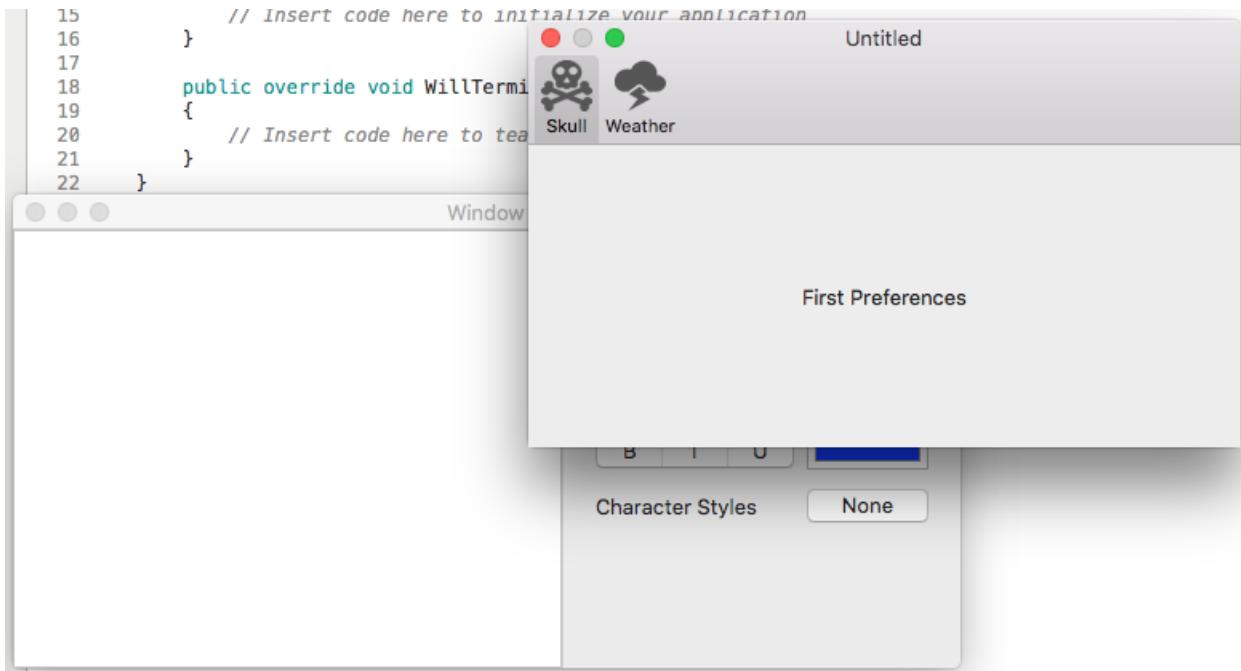
Select the Tab View Controller and switch its Style to Toolbar:



Select each Tab and give it a Label and select one of the images to represent it:



If we save our changes, return to Visual Studio for Mac, run the app and select the **Preferences...** menu item, the dialog will now be displayed like a standard macOS app:



For more information, please see our [Working with Images](#), [Menus](#), [Windows](#) and [Dialogs](#) documentation.

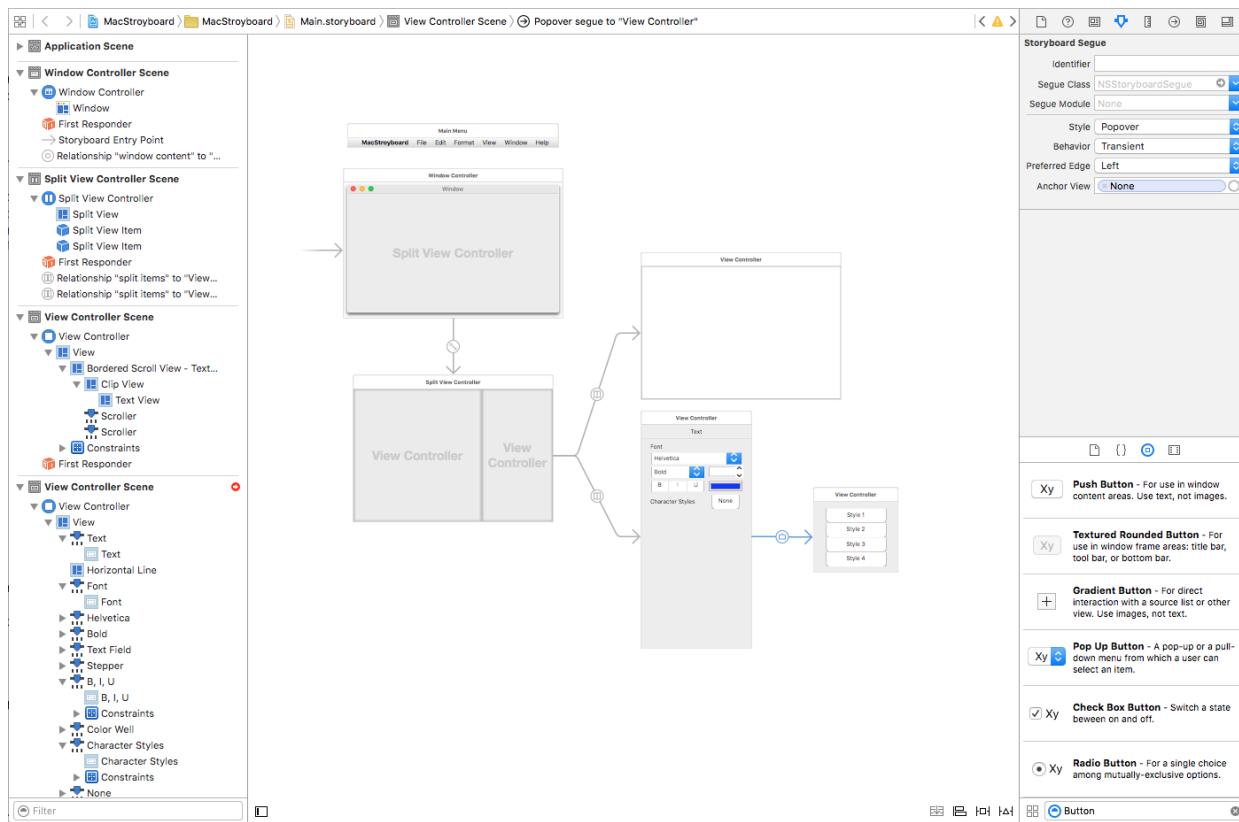
Related Links

- [Hello, Mac](#)
- [Working with Windows](#)
- [macOS Human Interface Guidelines](#)
- [Introduction to Windows](#)

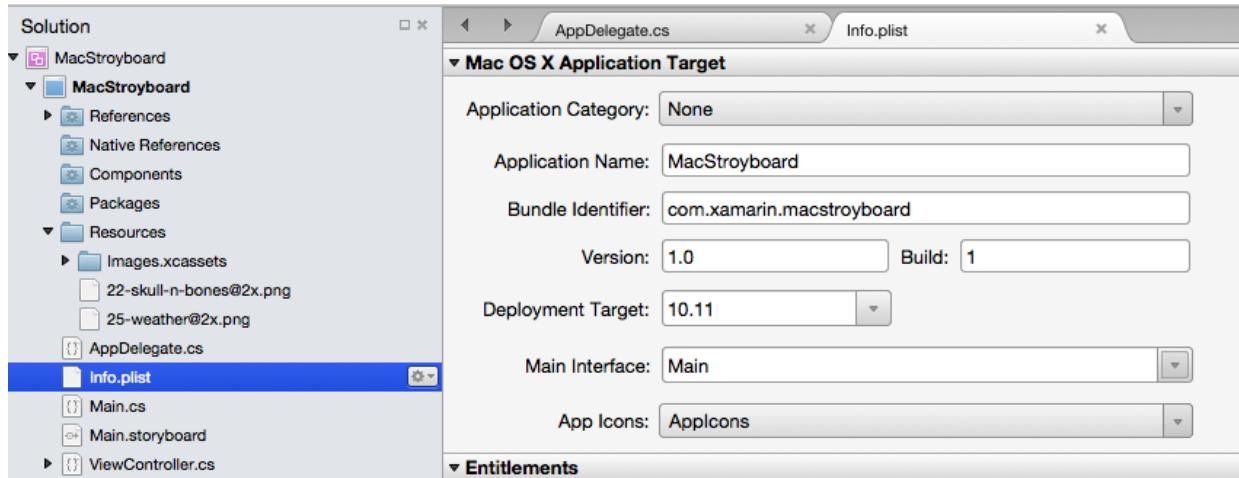
Working with Storyboards in Xamarin.Mac

3/5/2021 • 12 minutes to read • [Edit Online](#)

A storyboard defines all of the UI for a given app broken down into a functional overview of its view controllers. In Xcode's Interface Builder, each of these controllers lives in its own Scene.



The storyboard is a resource file (with the extensions of `.storyboard`) that gets included in the Xamarin.Mac app's bundle when it is compiled and shipped. To define the starting Storyboard for your app, edit its `Info.plist` file and select the **Main Interface** from the dropdown box:



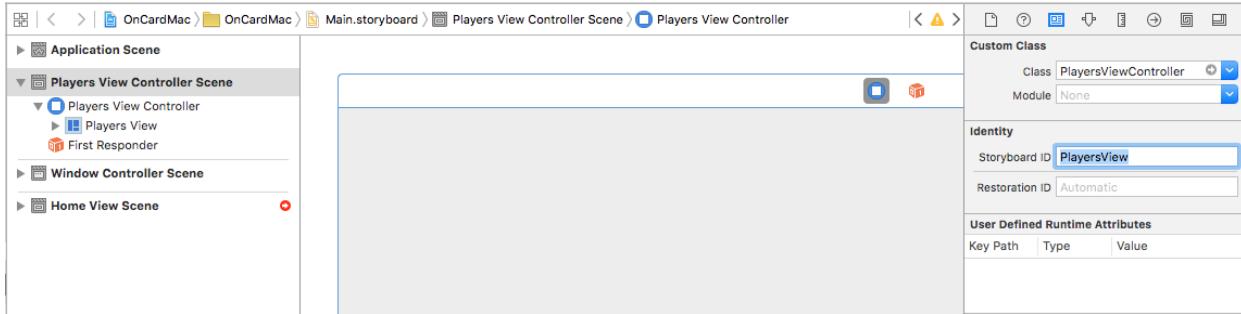
Loading from Code

There might be times when you need to load a specific Storyboard from code and create a View Controller manually. You can use the following code to perform this action:

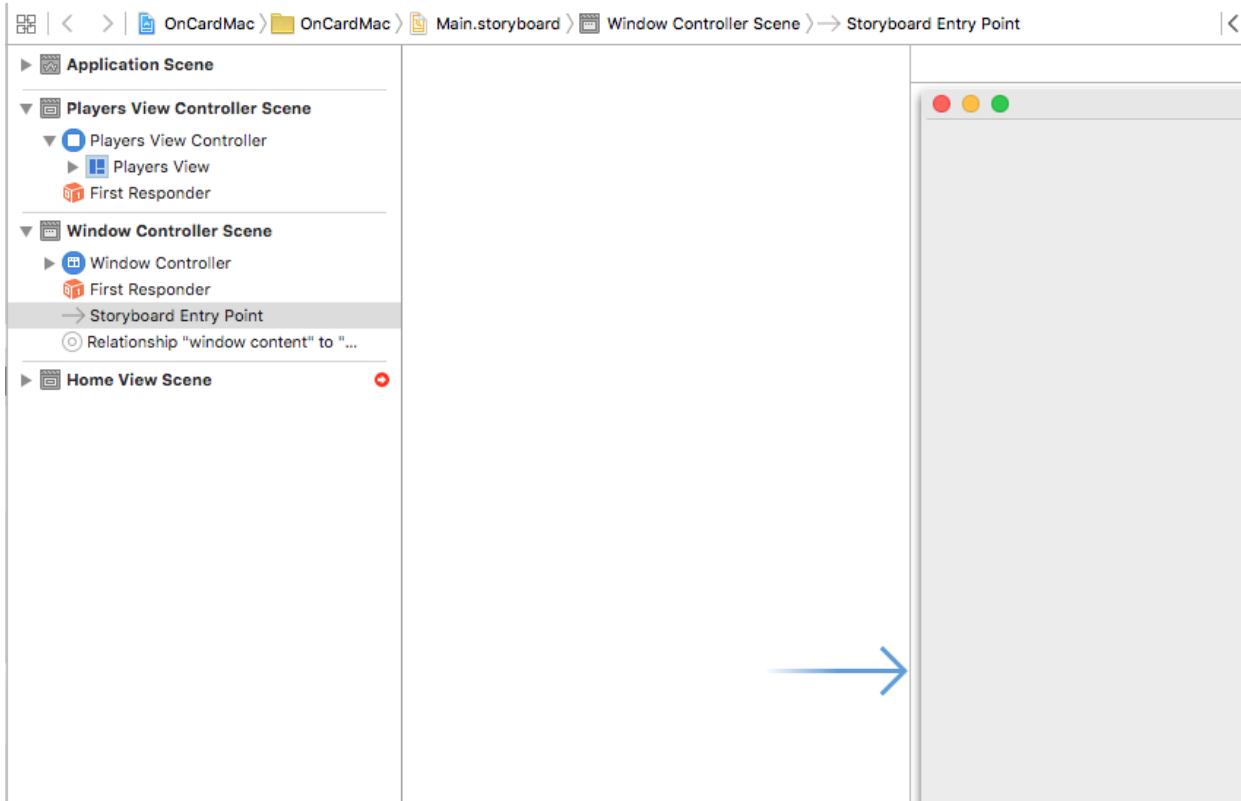
```
// Get new window
var storyboard = NSStoryboard.FromName ("Main", null);
var controller = storyboard.InstantiateControllerWithIdentifier ("MainWindow") as NSWindowController;

// Display
controller.ShowWindow(this);
```

The `FromName` loads the Storyboard file with the given name that has been included in the app's bundle. The `InstantiateControllerWithIdentifier` creates an instance of the View Controller with the given Identity. You set the Identity in Xcode's Interface Builder when designing the UI:



Optionally, you can use the `InstantiateInitialController` method to load the View Controller that has been assigned the Initial Controller in Interface Builder:



It's marked by the **Storyboard Entry Point** and the open ended arrow above.

View Controllers

View Controllers define the relationships between a given View of information within a Mac app and the data model that provides that information. Each top level scene in the Storyboard represents one View Controller in the Xamarin.Mac app's code.

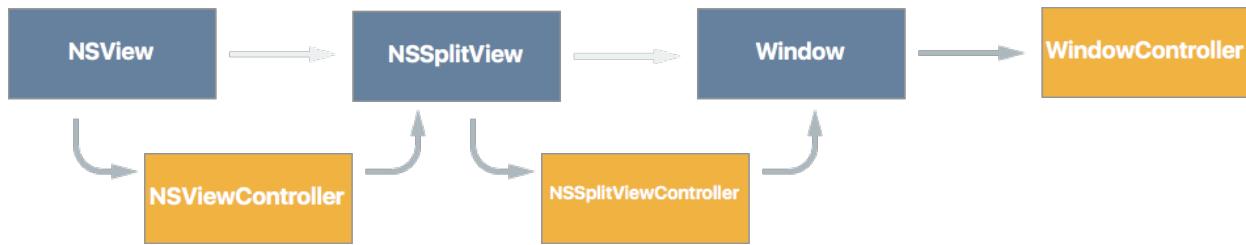
The View Controller Lifecycle

Several new methods have been added to the `NSViewController` class to support Storyboards in macOS. Most importantly are the follow methods use to respond to the lifecycle of the View being controlled by the given View Controller:

- `ViewDidLoad` - This method is called when the view is loaded from the Storyboard file.
- `ViewWillAppear` - This method is called just before the view is displayed on screen.
- `ViewDidAppear` - This method is called directly after the view has been displayed on screen.
- `ViewWillDisappear` - This method is called just before the view is removed from the screen.
- `ViewDidDisappear` - This method is called directly after the view has been removed from the screen.
- `UpdateViewConstraints` - This method is called when the constraints that define a view auto layout position and size need to be updated.
- `ViewWillLayout` - This method is called just before the subviews of this view are laid out on screen.
- `ViewDidLayout` - This method is called directly after the subviews of view are laid out on screen.

The Responder Chain

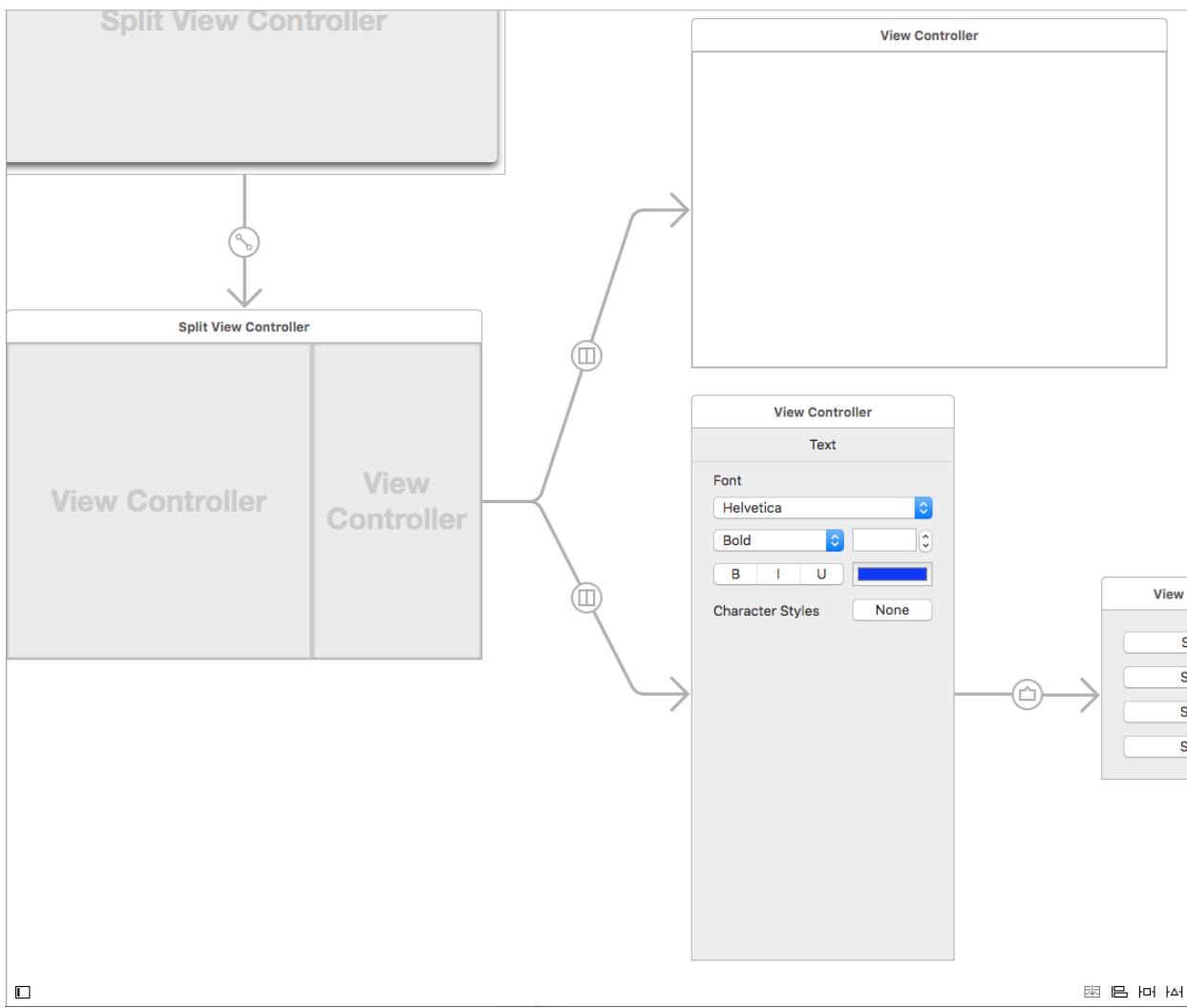
Additionally, `NSViewControllers` are now part of the Window's *Responder Chain*:



And as such they are wired-up to receive and respond to events such as Cut, Copy and Paste menu item selections. This automatic View Controller wire-up only occurs on apps running on macOS Sierra (10.12) and greater.

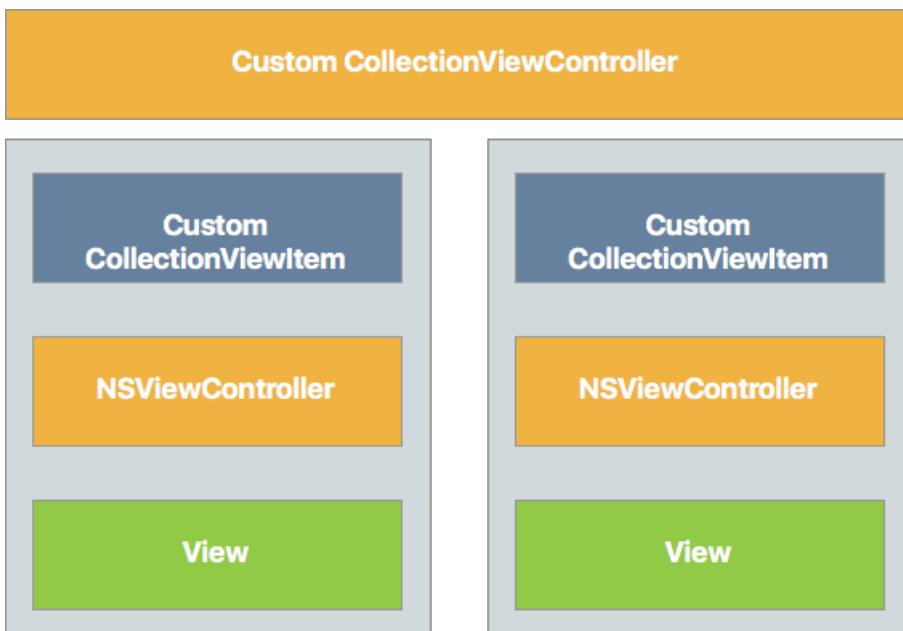
Containment

In Storyboards, View Controllers (such as the Split View Controller and the Tab View Controller) can now implement *Containment*, such that they can "contain" other sub View Controllers:



Child View Controllers contain methods and properties to tie them back to their Parent View Controller and to work with displaying and removing Views from the screen.

All Container View Controllers built into macOS have a specific layout which Apple suggest that you follow if creating your own custom Container View Controllers:



The Collection View Controller contains an array of Collection View Items, each of which contain one or more View Controllers that contain their own Views.

Segues

Segues provide the relationships between all of the Scenes that define your app's UI. If you are familiar with working in Storyboards in iOS, you know that Segues for iOS usually define transitions between full screen views. This differs from macOS, when Segues usually define "[Containment](#)", where one Scene is the child of a parent Scene.

In macOS, most apps tend to group their views together within the same window using UI elements such as Split Views and Tabs. Unlike iOS, where views need to be transitioned on and off screen, due to limited physical display space.

Presentation Segues

Given macOS's tendencies towards containment, there are situations where *Presentation Segues* are used, such as Modal Windows, Sheet Views and Popovers. macOS Provides the following built-in segue types:

- **Show** - Displays the target of the Segue as a non-modal window. For example, use this type of Segue to present another instance of a Document Window in your app.
- **Modal** - Presents the target of the Segue as a modal window. For example, use this type of Segue to present the Preferences Window for your app.
- **Sheet** - Presents the target of the Segue as a Sheet attached to the parent window. For example, use this type of segue to present a Find and Replace Sheet.
- **Popover** - Presents the target of the Segue as in a popover window. For Example, use this Segue type to present options when a UI element is clicked by the user.
- **Custom** - Presents the target of the Segue using a custom Segue Type defined by the developer. See the [Creating Custom Segues](#) section below for more details.

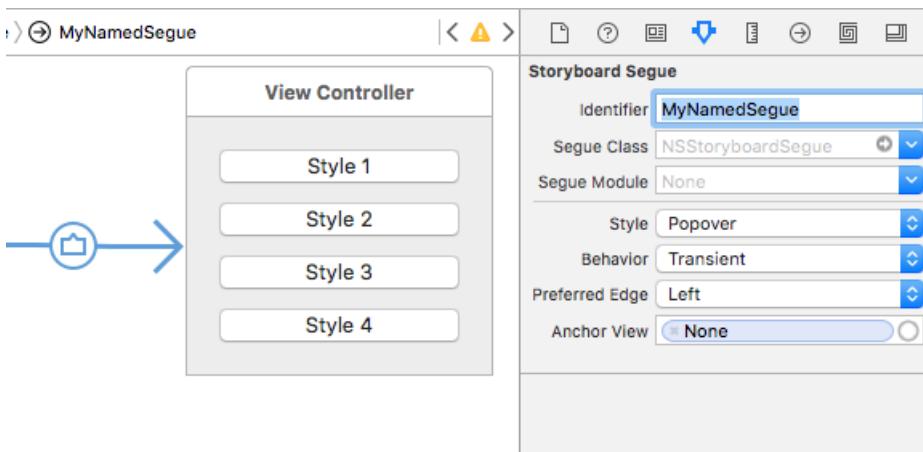
When using Presentation Segues, you can override the `PrepareForSegue` method of the parent View Controller for presentation to initialize and variables and provide any data to the View Controller being presented.

Triggered Segues

Triggered Segues allow you to specify named Segues (via their **Identifier** property in Interface Builder) and have them triggered by events such as the user clicking a button or by calling the `PerformSegue` method in code:

```
// Display the Scene defined by the given Segue ID
PerformSegue("MyNamedSegue", this);
```

The Segue ID is defined inside of Xcode's Interface Builder when you are laying out the app's UI:



In the View Controller that is acting as the source of the Segue, you should override the `PrepareForSegue` method and do any initialization required before the Segue is executed and the specified View Controller is

displayed:

```
public override void PrepareForSegue (UIStoryboardSegue segue, NSObject sender)
{
    base.PrepareForSegue (segue, sender);

    // Take action based on Segue ID
    switch (segue.Identifier) {
        case "MyNamedSegue":
            // Prepare for the segue to happen
            ...
            break;
    }
}
```

Optionally, you can override the `ShouldPerformSegue` method and control whether or not the Segue is actually executed via C# code. For manually presented View Controllers, call their `DismissController` method to remove them from display when they are no longer needed.

Creating Custom Segues

There might be times when your app requires a Segue type not provided by the build-in Segues defined in macOS. If this is the case, you can create a Custom Segue that can be assigned in Xcode's Interface Builder when laying out your app's UI.

For example, to create a new Segue type that replaces the current View Controller inside a Window (instead of opening the target Scene in a new window), we can use the following code:

```

using System;
using AppKit;
using Foundation;

namespace OnCardMac
{
    [Register("ReplaceViewSegue")]
    public class ReplaceViewSegue : NSSStoryboardSegue
    {
        #region Constructors
        public ReplaceViewSegue() {

        }

        public ReplaceViewSegue (string identifier, NSObject sourceController, NSObject destinationController) : base(identifier,sourceController,destinationController) {

        }

        public ReplaceViewSegue (IntPtr handle) : base(handle) {

        }

        public ReplaceViewSegue (NSObjectFlag x) : base(x) {
        }
        #endregion

        #region Override Methods
        public override void Perform ()
        {
            // Cast the source and destination controllers
            var source = SourceController as NSViewController;
            var destination = DestinationController as NSViewController;

            // Swap the controllers
            source.Window.ContentViewController = destination;

            // Release memory
            source.RemoveFromParentViewController ();
        }
        #endregion
    }
}

```

A couple of things to note here:

- We are using the `Register` attribute to expose this class to Objective-C/macOS.
- We are overriding the `Perform` method to actually perform the action of our custom Segue.
- We are replacing the Window's `ContentViewController` controller with the one defined by the target (destination) of the Segue.
- We are removing the original View Controller to free up memory using the `RemoveFromParentViewController` method.

To use this new Segue type in Xcode's Interface Builder, we need to compile the app first, then switch to Xcode and add a new Segue between two scenes. Set the **Style** to **Custom** and the **Segue Class** to `ReplaceViewSegue` (the name of our custom Segue class):



Window Controllers

Window Controllers contain and control the different Window types that your macOS app can create. For Storyboards, they have the following features:

1. They must provide a Content View Controller. This will be the same Content View Controller that the child Window has.
2. The `Storyboard` property will contain the Storyboard that the Window Controller was loaded from, else `null` if not loaded from a Storyboard.
3. You can call the `DismissController` method to close the given Window and remove it from view.

Like View Controllers, Window Controllers implement the `PerformSegue`, `PrepareForSegue` and the `ShouldPerformSegue` methods and can be used as the source of a Segue operation.

Window Controller are responsible for the following features of a macOS app:

- They manage a specific Window.
- They manage the Window's Title Bar and Toolbar (if available).
- They manage the Content View Controller to display the contents of the Window.

Gesture Recognizers

Gesture Recognizers for macOS are nearly identical to their counterparts in iOS and allow the developer to easily add gestures (such as clicking a mouse button) to elements in your app's UI.

However, where gestures in iOS are determined by the app's design (such as tapping the screen with two fingers), most gestures in macOS are determined by hardware.

By using Gesture Recognizers, you can greatly reduce the amount of code required to add custom interactions to an item in the UI. As they can automatically determine between double and single clicks, click and drag events, etc.

Instead of overriding the `MouseDown` event in your View Controller, you should be using a Gesture Recognizer to handle the user input event when working with Storyboards.

The following Gesture Recognizers are available in macOS:

- `NSClickGestureRecognizer` - Register mouse down and up events.
- `NSPanGestureRecognizer` - Registers mouse button down, drag and release events.

- `NSPressGestureRecognizer` - Registers holding a mouse button down for a given amount of time event.
- `NSMagnificationGestureRecognizer` - Registers a magnification event from trackpad hardware.
- `NSRotationGestureRecognizer` - Registers a rotation event from trackpad hardware.

Using Storyboard References

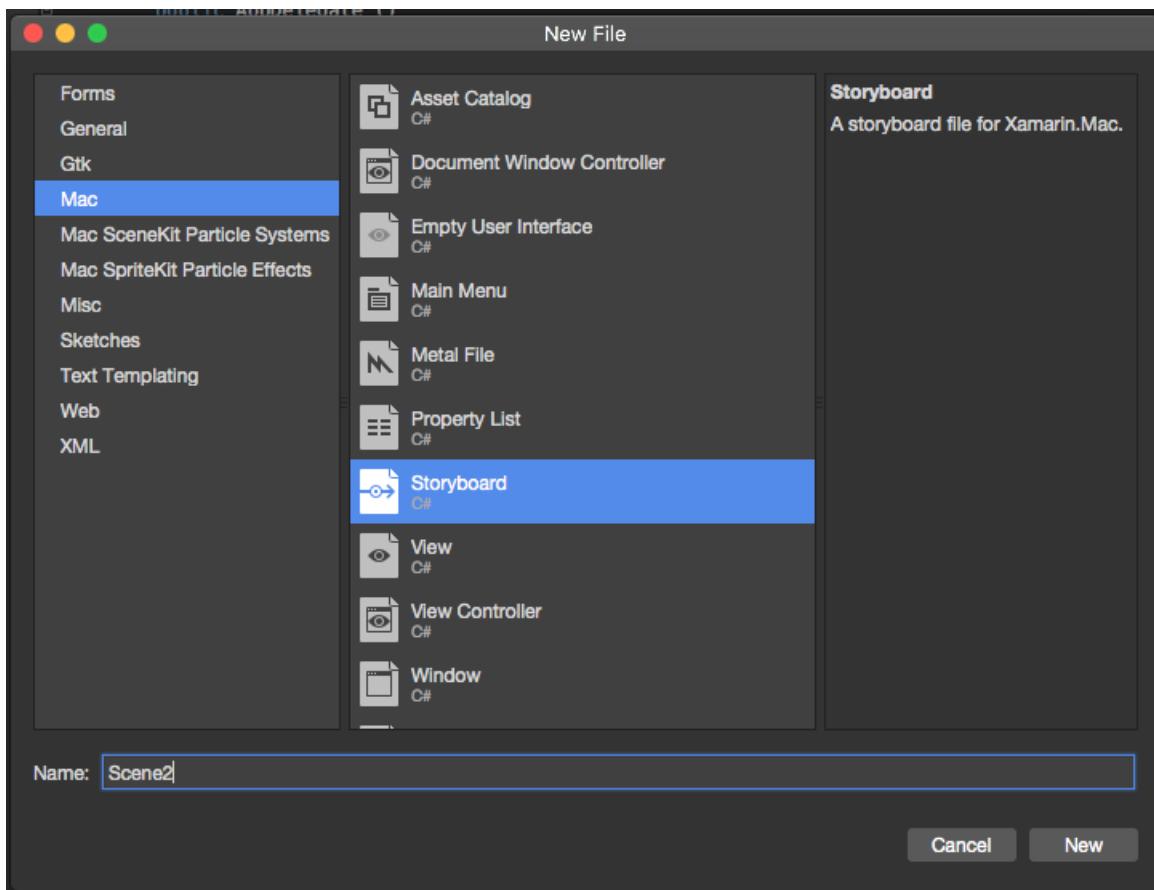
A Storyboard Reference allows you to take a large, complex Storyboard design and break it into smaller Storyboards that get referenced from the original, thus removing complexity and making the resulting individual Storyboards easier to design and maintain.

Additionally, a Storyboard Reference can provide an *anchor* to another scene within the same Storyboard or a specific scene on a different one.

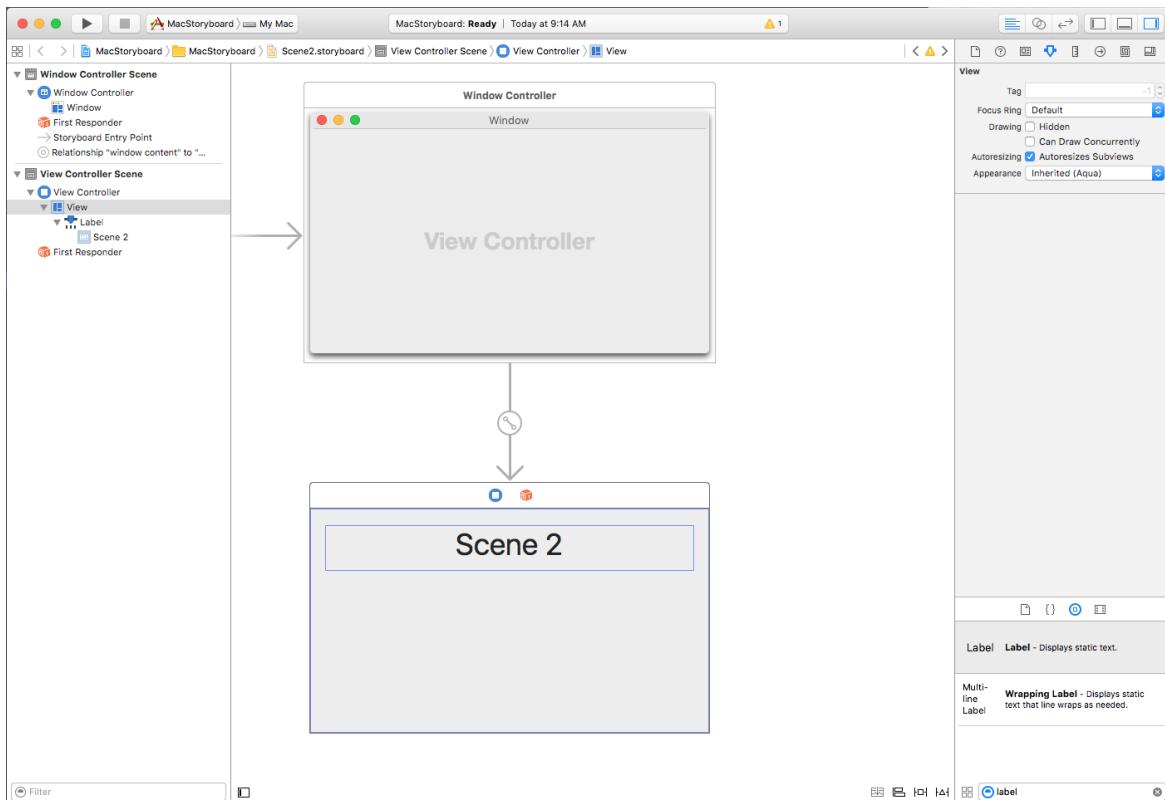
Referencing an External Storyboard

To add a reference to an external Storyboard, do the following:

1. In the **Solution Explorer**, right-click on the Project Name and select **Add > New File... > Mac > Storyboard**. Enter a **Name** for the new Storyboard and click the **New** button:

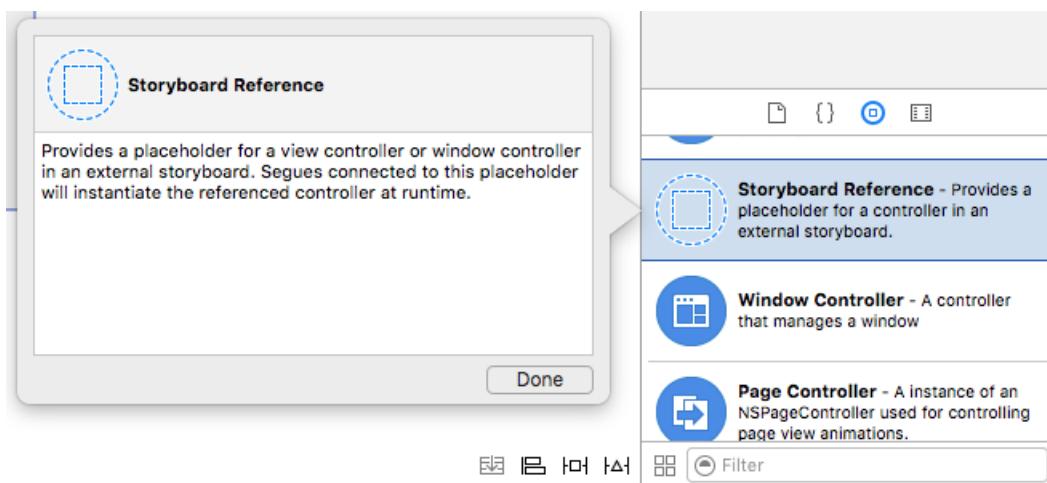


2. In the **Solution Explorer**, double-click the new Storyboard name to open it for editing in Xcode's Interface Builder.
3. Design the layout of the new Storyboard's scenes as you normally would and save your changes:

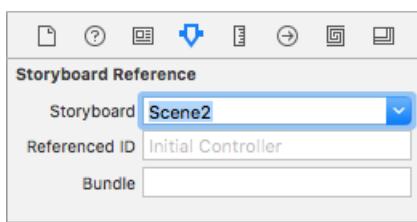


4. Switch to the Storyboard that you are going to be adding the reference to in the Interface Builder.

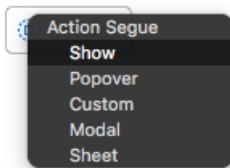
5. Drag a **Storyboard Reference** from the **Object Library** onto the Design Surface:



6. In the **Attribute Inspector**, select the name of the **Storyboard** that you created above:



7. Control-click on a UI Widget (like a Button) on an existing Scene and create a new Segue to the **Storyboard Reference** that you just created. From the popup menu select **Show** to complete the Segue:



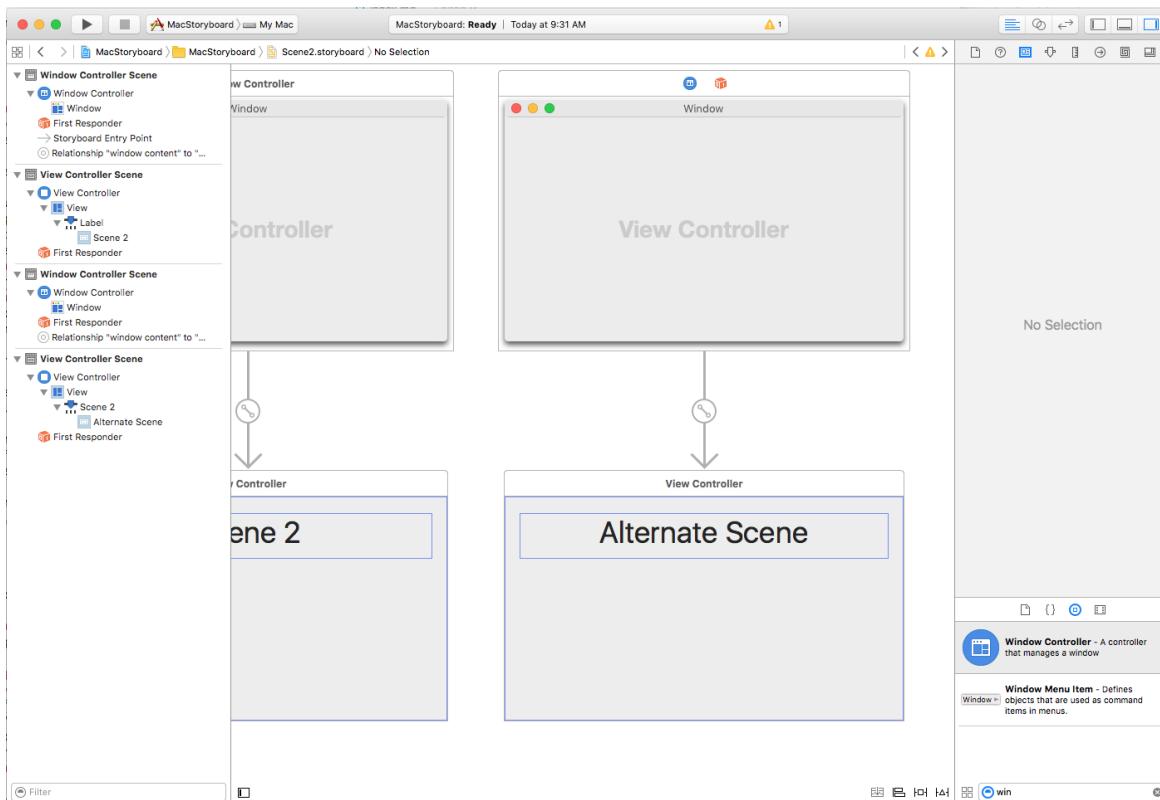
8. Save your changes to the Storyboard.
9. Return to Visual Studio for Mac to sync your changes.

When the app is run and the user clicks on the UI element that you created the Segue from, the Initial Window Controller from the External Storyboard specified in the Storyboard Reference will be displayed.

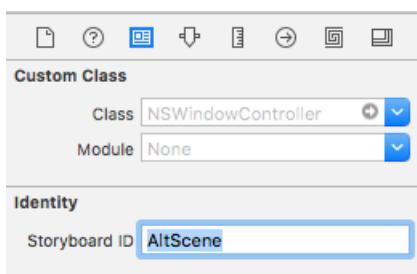
Referencing a Specific Scene in an External Storyboard

To add a reference to a specific Scene in an external Storyboard (and not the Initial Window Controller), do the following:

1. In the **Solution Explorer**, double-click the external Storyboard to open it for editing in Xcode's Interface Builder.
2. Add a new Scene and design its layout as you normally would:

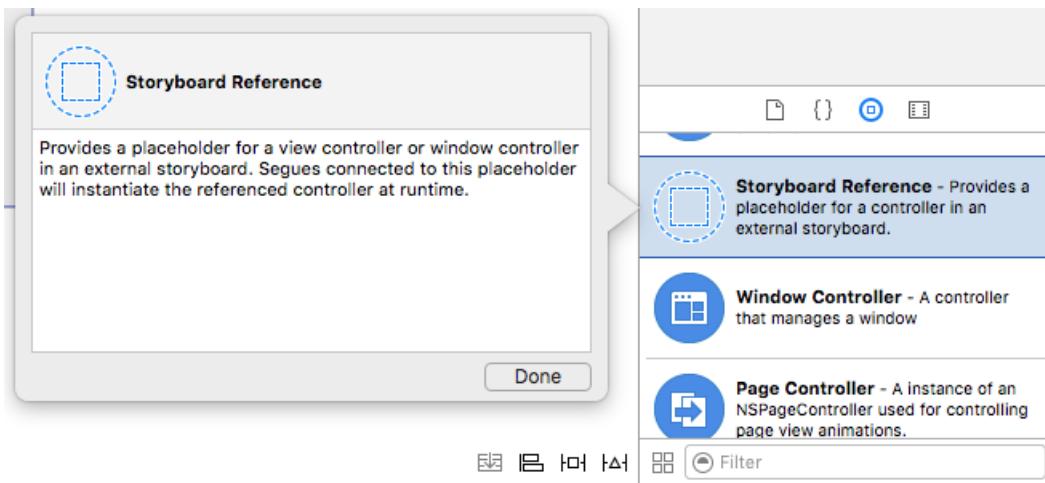


3. In the **Identity Inspector**, enter a **Storyboard ID** for the new Scene's Window Controller:

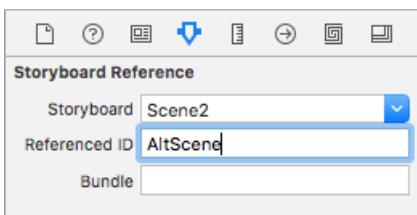


4. Open the Storyboard that you are going to be adding the reference to in Interface Builder.

5. Drag a **Storyboard Reference** from the Object Library onto the Design Surface:



6. In the **Identity Inspector**, select the name of the **Storyboard** and the **Reference ID** (Storyboard ID) of the Scene that you created above:



7. Control-click on a UI Widget (like a Button) on an existing Scene and create a new Segue to the **Storyboard Reference** that you just created. From the popup menu select **Show** to complete the Segue:



8. Save your changes to the Storyboard.

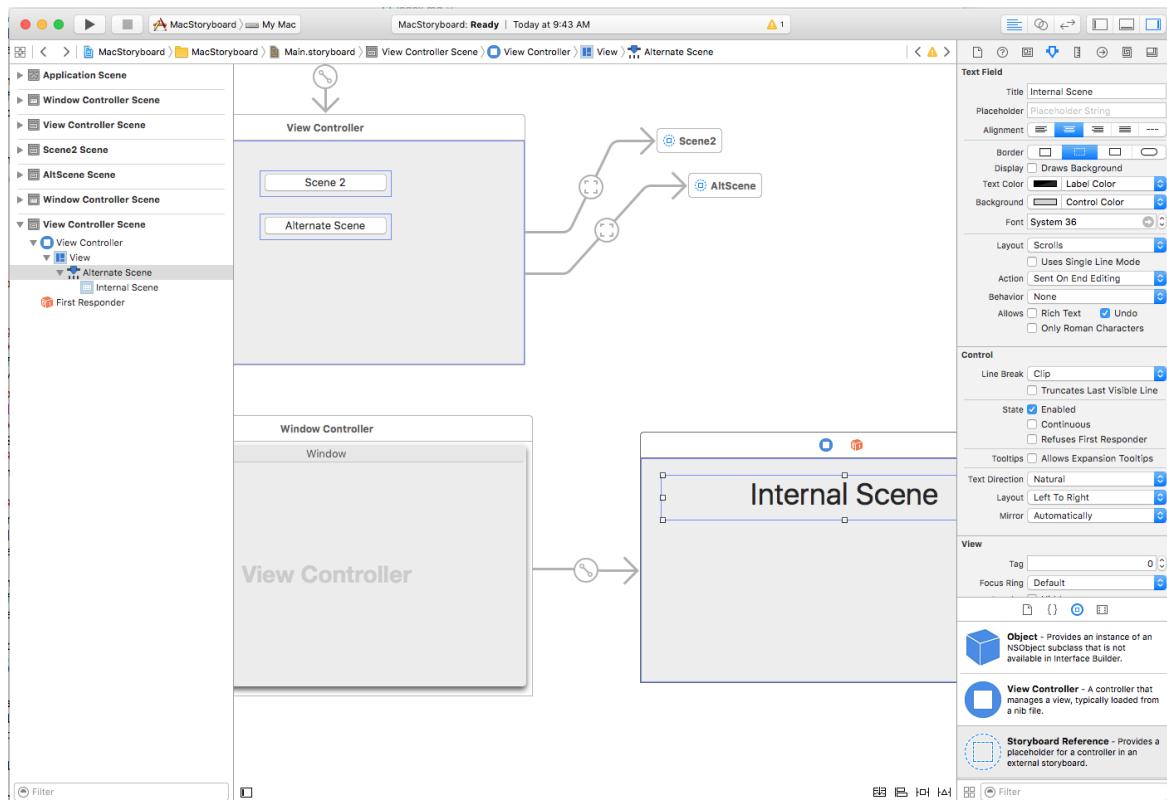
9. Return to Visual Studio for Mac to sync your changes.

When the app is run and the user clicks on the UI element that you created the Segue from, the Scene with the given **Storyboard ID** from the External Storyboard specified in the Storyboard Reference will be displayed.

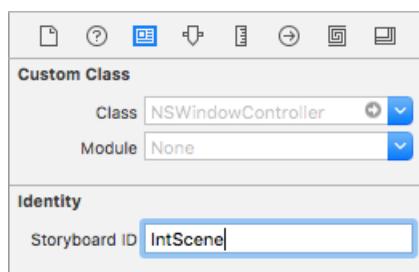
Referencing a Specific Scene in the Same Storyboard

To add a reference to a specific Scene in the same Storyboard, do the following:

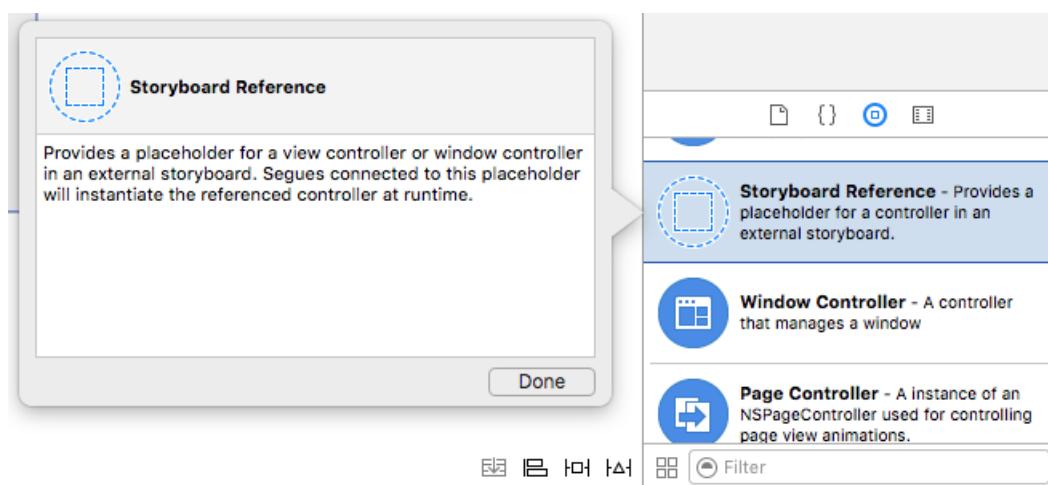
1. In the **Solution Explorer**, double-click the Storyboard to open it for editing.
2. Add a new Scene and design its layout as you normally would:



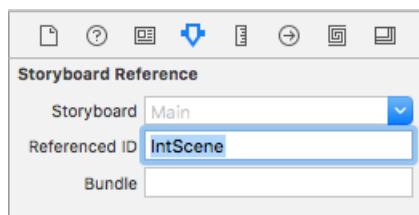
3. In the **Identity Inspector**, enter a **Storyboard ID** for the new Scene's Window Controller:



4. Drag a **Storyboard Reference** from the **Toolbox** onto the Design Surface:



5. In **Attribute Inspector**, select **Reference ID** (Storyboard ID) of the Scene that you created above:



6. Control-click on a UI Widget (like a Button) on an existing Scene and create a new Segue to the **Storyboard Reference** that you just created. From the popup menu select **Show** to complete the Segue:



7. Save your changes to the Storyboard.
8. Return to Visual Studio for Mac to sync your changes.

When the app is run and the user clicks on the UI element that you created the Segue from, the Scene with the given **Storyboard ID** in the same Storyboard specified in the Storyboard Reference will be displayed.

Complex Storyboard Example

For a complex example of working with Storyboards in a Xamarin.Mac app, please see the [SourceWriter Sample App](#). SourceWriter is a simple source code editor that provides support for code completion and simple syntax highlighting.

The SourceWriter code has been fully commented and, where available, links have been provided from key technologies or methods to relevant information in the Xamarin.Mac Guides Documentation.

Related Links

- [Hello, Mac](#)
- [Working with Windows](#)
- [OS X Human Interface Guidelines](#)
- [Introduction to Windows](#)

Xamarin.Mac Extension Support

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Xamarin.Mac 2.10 support was added for multiple macOS extension points:

- Finder
- Share
- Today

Limitations and Known Issues

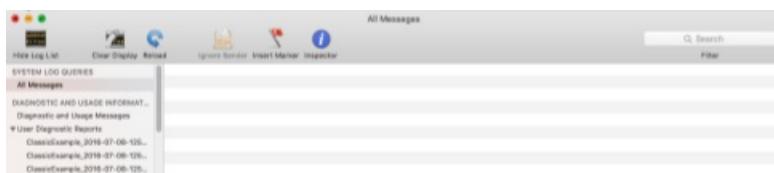
The following are the limitations and known issues that can occur when developing extensions in Xamarin.Mac:

- There is currently no debugging support in Visual Studio for Mac. All debugging will need to be done via `NSLog` and the **Console**. See the tips section below for details.
- Extensions must be contained in a host application, which when run one time with register with the system. They must then be enabled in the **Extension** section of **System Preferences**.
- Some extension crashes may destabilize the host application and cause strange behavior. In particular, **Finder** and the **Today** section of the **Notification Center** may become "jammed" and become unresponsive. This has been experienced in extension projects in Xcode as well, and currently appears unrelated to Xamarin.Mac. Often this can be seen in the system log (via **Console**, see **Tips** for details) printing repeated error messages. Restarting macOS appears to fix this.

Tips

The following tips can be helpful when working with extensions in Xamarin.Mac:

- As Xamarin.Mac currently does not support debugging extensions, the debugging experience will primarily depend on execution and `printf` like statements. However, extensions run in a sandbox process, thus `Console.WriteLine` will not act as it does in other Xamarin.Mac applications. Invoking `NSLog` directly will output debugging messages to the System Log.
- Any uncaught exceptions will crash the extension process, providing only a small amount of useful information in the **System Log**. Wrapping troublesome code in a `try/catch` (Exception) block that `NSLog`'s before re-throwing may be useful.
- The **System Log** can be accessed from the **Console** app under **Applications > Utilities**:



- As noted above, running the extension host application will register it with the system. Deleting the application bundle will unregister it.
- If "stray" versions of an app's extensions are registered, use the following command to locate them (so they can be deleted): `plugin kit -mv`

Walkthrough and Sample App

Since the developer will create and work with Xamarin.Mac extensions in the same way as Xamarin.iOS extensions, please refer to our [Introduction to Extensions](#) documentation for more details.

An example Xamarin.Mac project containing small, working samples of each extension type can be found [here](#).

Summary

This article has taken a quick look at working with extensions in a Xamarin.Mac version 2.10 (and greater) app.

Related Links

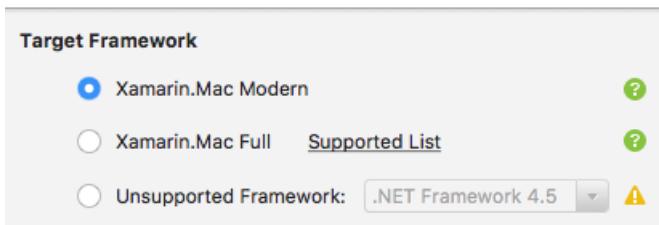
- [Hello, Mac](#)
- [ExtensionSamples](#)
- [macOS Human Interface Guidelines](#)

Target Framework for Xamarin.Mac

10/28/2019 • 2 minutes to read • [Edit Online](#)

This article covers the target frameworks (Base Class Libraries) available for Xamarin.Mac, and the implications of using them in your Xamarin.Mac project.

General



Background

Every .NET program or library depends on functionality provided by the Base Class Library (BCL). This BCL includes assemblies such as mscorelib, System, System.Net.Http, and System.Xml that provide the common functionality built into all .NET languages.

Over the years, there have developed multiple different versions of this BCL, optimized for different use cases. The "desktop" BCL includes a richer set of libraries which might be too heavyweight for other use cases, while mobile focuses on ensuring APIs are safe for linking, which removes unused code to reduce application footprint.

One of the more important repercussions of these different Target Frameworks, is that all of the assemblies in a given program *must* target compatible BCL assemblies. If this was not the case, you could have two assemblies linked against different versions of the `System.dll` disagreeing about the signature of a given type. A shared library can either target [.NET Standard 2](#), which is the common subset of the Target Frameworks, or a specific target framework.

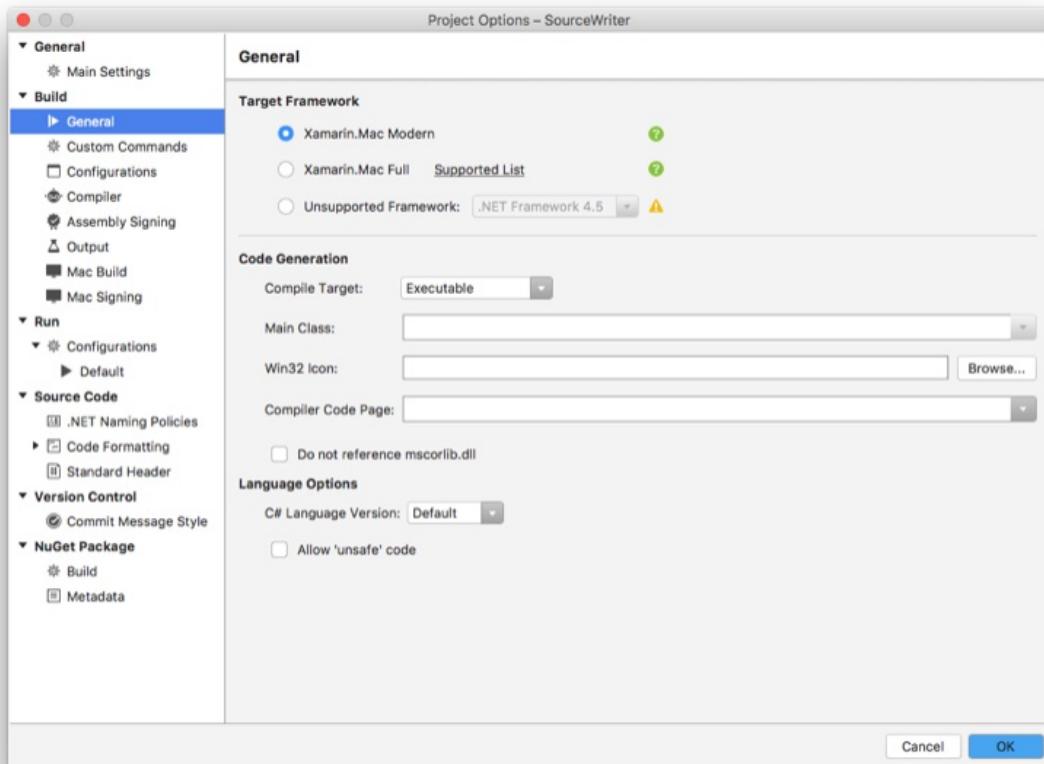
There are three Target Framework options available for Xamarin.Mac, each with different advantages and tradeoffs:

- **Modern** (called Mobile in older documentation) – A very similar subset to what powers Xamarin.iOS, highly tuned for performance and size. This Target Framework is linker safe, so these projects can have their final footprint drastically reduced by removing unused code.
- **Full** (called XM 4.5 in older documentation) – A very similar subset to the "desktop" BCL, with a few small removals. As the Target Framework is almost identical to net45 (and later), it can easily consume many nugets that do not provide either netstandard2 or specific Xamarin.Mac builds. However, due to `System.Configuration` usage it is incompatible with linking.
- **Unsupported** (called System in older documentation) – Instead of linking to a BCL provided by Xamarin.Mac, use the current system installed mono. This provides the fullest set of assemblies, including some known to be problematic (`System.Drawing` for example). This option exists only as a "last resort" and it is strongly suggested to exhaust other options before using it. As the name implies, usage is unsupported by official support channels.

Setting the target framework

To change to the Target Framework type for a Xamarin.Mac project, do the following:

1. Open the Xamarin.Mac project in Visual Studio for Mac.
2. In the **Solution Explorer**, double-click the project file to open the **Project Options** dialog box.
3. From the **General** tab, select the type of **Target Framework** that suits your application's needs:



4. Click the **OK** button to save your changes.

You should **Clean** and then **Rebuild** your Xamarin.Mac project after switching the Target Framework type.

Summary

This article has briefly covered the different types of Target Frameworks (Base Class Libraries) available to a Xamarin.Mac application and when each type of framework should be used.

Related Links

- [iOS and Mac code sharing](#)
- [Unified API](#)
- [Portable Class Libraries](#)
- [Assemblies](#)
- [Updating existing Mac apps](#)

Deploying and Testing Xamarin.Mac Apps

10/28/2019 • 2 minutes to read • [Edit Online](#)

Application icon

This article covers creating the images required for a Xamarin.Mac application's icon, bundling the images into a .icns file, and including the icon in the Xamarin.Mac project.

Performance

Tip to improve your Xamarin.Mac application's performance.

Publishing to the App Store

This guide walks through deploying a Xamarin.Mac application using Visual Studio for Mac. It explains how to set up the developer's Mac Developer Account, walks through the process of creating certificates for code signing and shows how to use them in Visual Studio for Mac to build Mac apps that can be distributed directly or via the Mac App Store.

Application icon for Xamarin.Mac apps

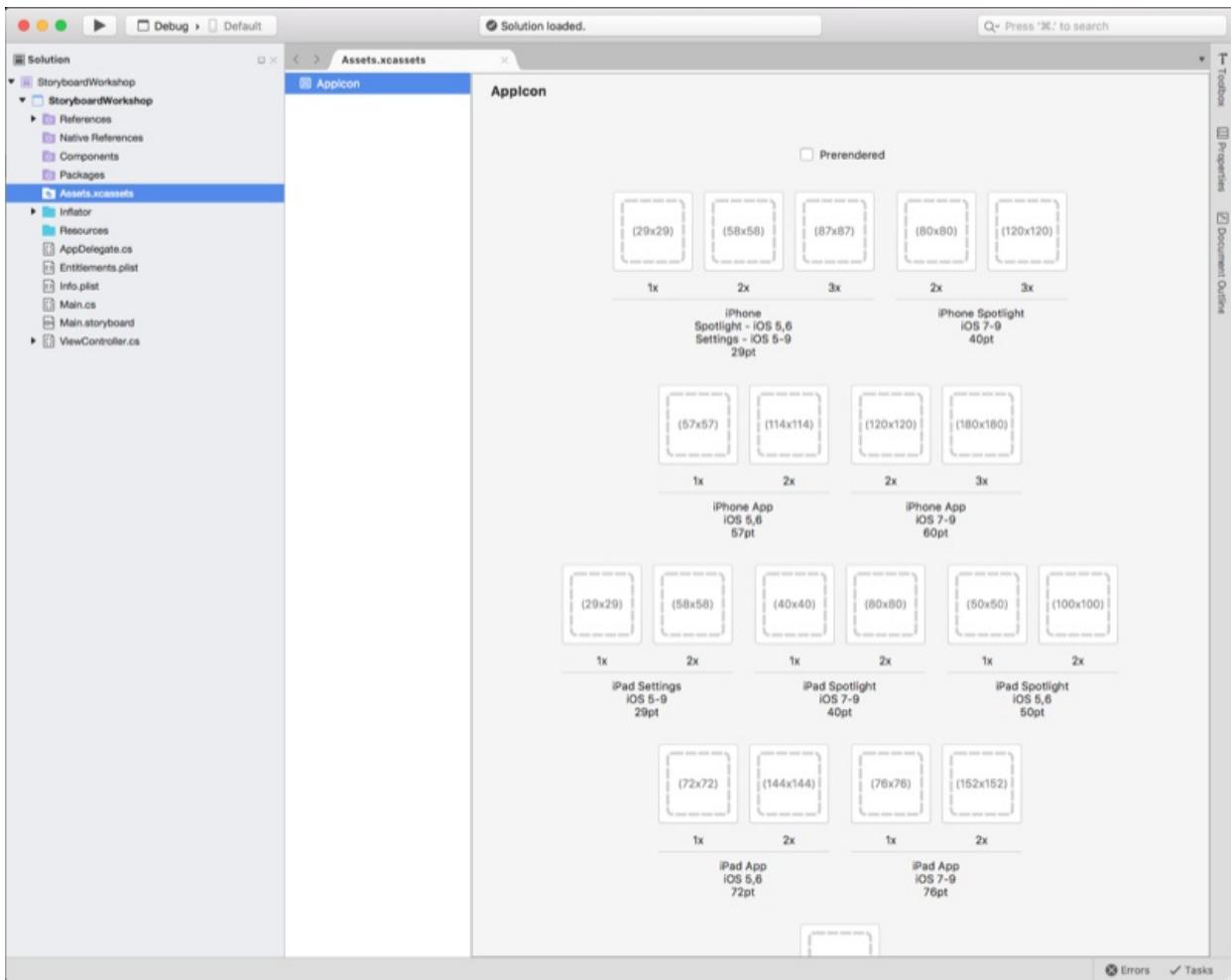
11/2/2020 • 3 minutes to read • [Edit Online](#)

This article covers creating the images required for a Xamarin.Mac application's icon, bundling the images into a .icns file, and including the icon in the Xamarin.Mac project.

Overview

When working with C# and .NET in a Xamarin.Mac application, a developer has access to the same Image and Icon tools that a developer working in *Objective-C* and *Xcode* does.

A great Icon should convey the main purpose of a Xamarin.Mac app and hint experience the user should expect when using the app. This article covers all of the steps necessary to create the Image Assets required for an Icon, packaging those assets into a `AppIcon.appiconset` file and consuming that file in a Xamarin.Mac app.



Application icon

A great Icon should convey the main purpose of a Xamarin.Mac app and hint experience the user should expect when using an app. Every macOS app must include several sizes of its Icon for display in the Finder, Dock, Launchpad, and other locations throughout the computer.

Designing the icon

Apple suggests the following tips when designing an application's icon:

- Consider giving the icon a realistic and unique shape.
- If the macOS app has an iOS counterpart, don't reuse the iOS app's icon.
- Use universal imagery that people can easily recognize.
- Strive for simplicity.
- Use color and shadow sparingly to help the icon tell the app's story.
- Avoid mixing actual text with *greeked* text or lines to suggest text.
- Create an idealized version of the icon's subject rather than using an actual photo.
- Avoid using macOS UI elements in the icons.
- Don't use replicas of Apple icons in the icons.

Please read the [App Icon Gallery](#) and [Designing App Icons](#) sections of Apple's OS X Human Interface Guidelines before designing a Xamarin.Mac app's icon.

Required image sizes and filenames

Like any other Image Resource that the developer is going to use in a Xamarin.Mac app, the app icon needs to provide both a Standard and Retina Resolution version. Again, like any other image, use a `@2x` format when naming the icon files:

- **Standard-Resolution** - `/ImageName.filename-extension` (Example: `icon_512x512.png`)
- **High-Resolution** - `/ImageName@2x.filename-extension` (Example: `icon_512x512@2x.png`)

For example, to supply the 512 x 512 version of the app's icon, the file would be named `icon_512x512.png` and `icon_512x512@2x.png`.

To ensure that the icon looks great in all the places that users see it, provide resources in the sizes listed below:

FILENAME	SIZE IN PIXELS
<code>icon_512x512@2x.png</code>	1024 x 1024
<code>icon_512x512.png</code>	512 x 512
<code>icon_256x256@2x.png</code>	512 x 512
<code>icon_256x256.png</code>	256 x 256
<code>icon_128x128@2x.png</code>	256 x 256
<code>icon_128x128.png</code>	128 x 128
<code>icon_32x32@2x.png</code>	64 x 64
<code>icon_32x32.png</code>	32 x 32
<code>icon_16x16@2x.png</code>	32 x 32
<code>icon_16x16.png</code>	16 x 16

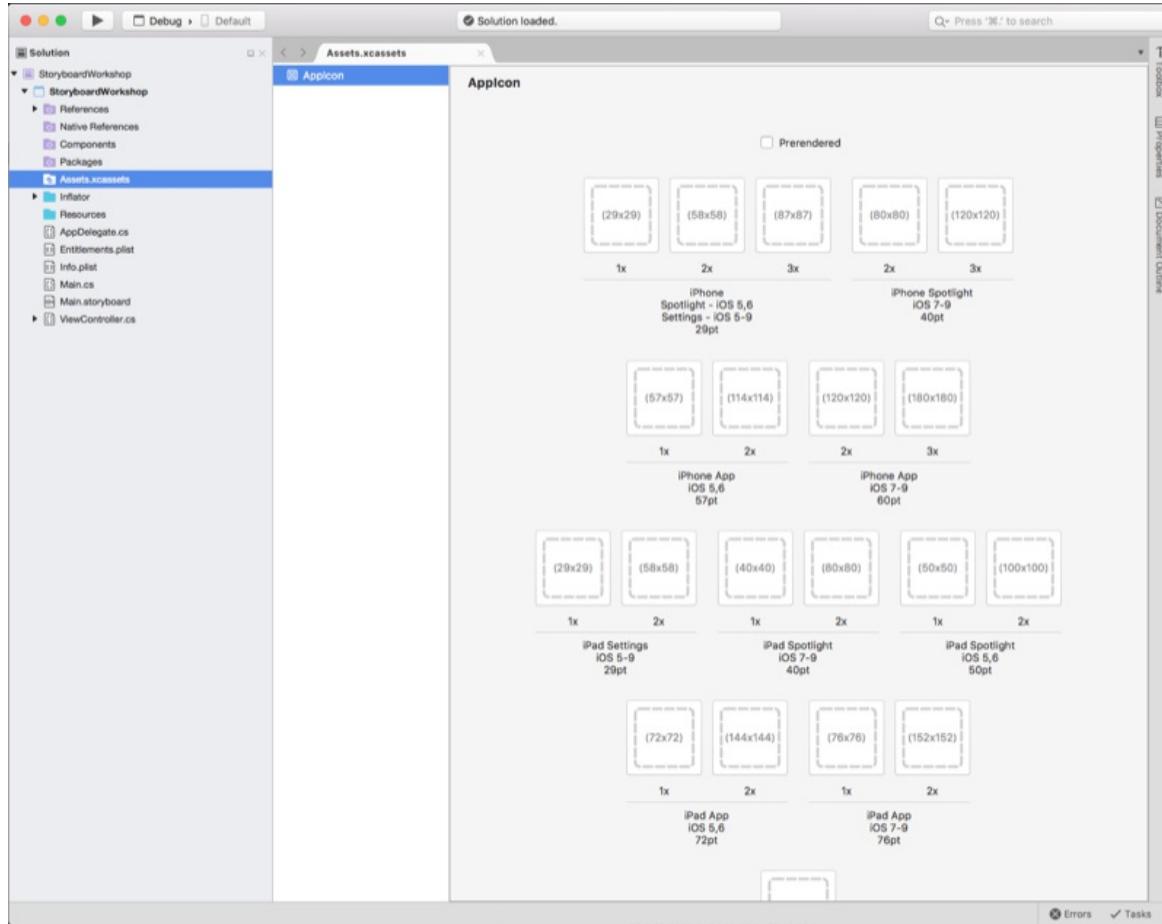
For more information, see Apple's [Provide High-Resolution Versions of All App Graphics Resources](#) documentation.

Packaging the icon resources

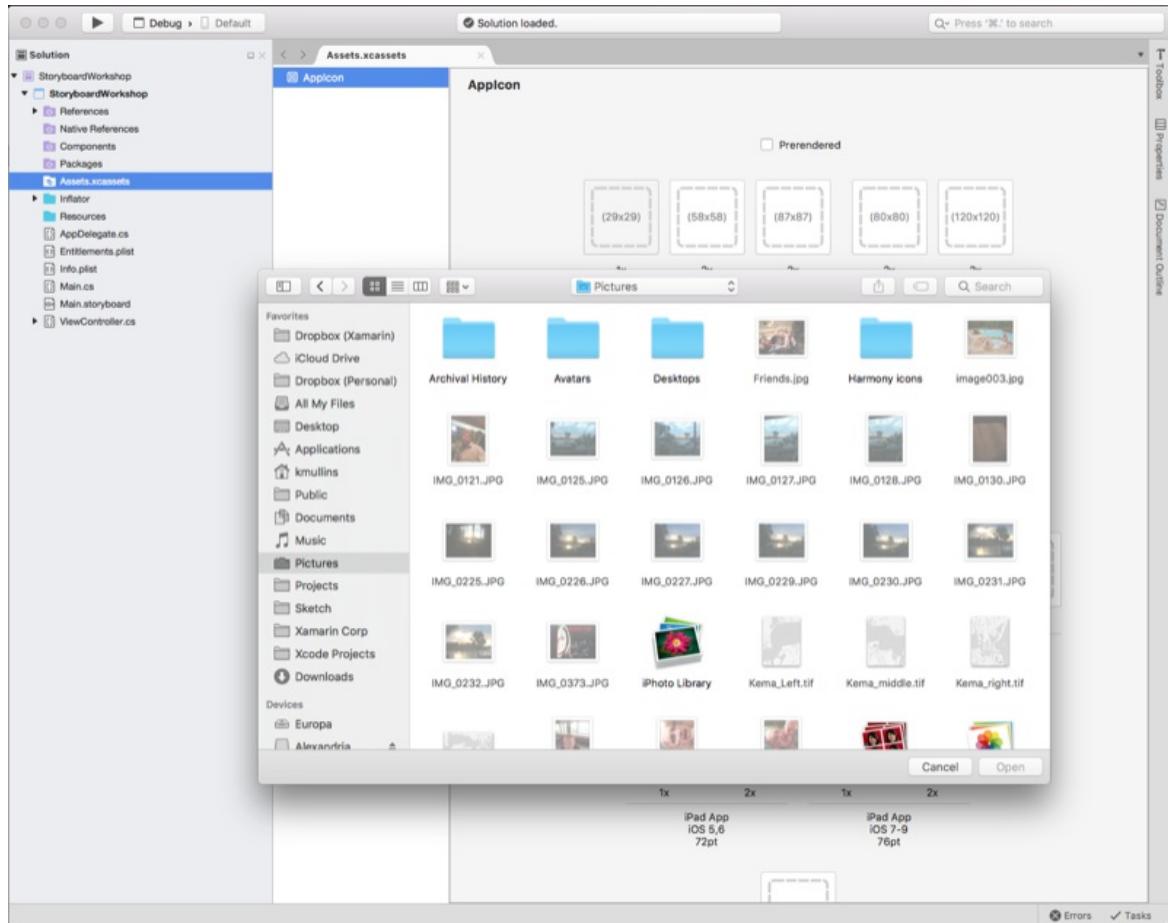
With the icon designed and saved out to the required file sizes and names, Visual Studio for Mac makes it easy to assign them to the image assets for use in Xamarin.Mac.

Do the following:

1. In the Solution Pad, open Assets.xcassets > AppIcons.appiconset:



2. For each icon size required, click the icon and select the corresponding image file that were created above:



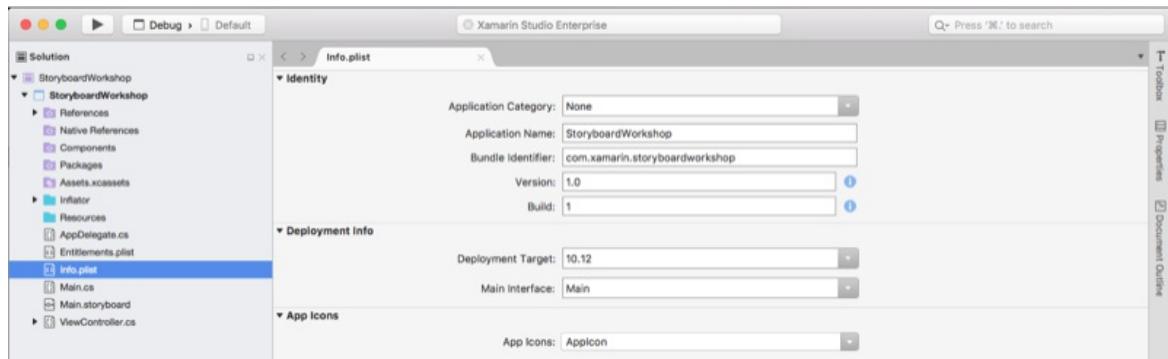
3. Save your changes.

Using the icon

Once the `AppIcon.appiconset` file has been built, it will need to assign it to the Xamarin.Mac project in Visual Studio for Mac.

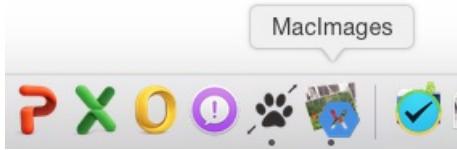
Do the following:

1. Double-click the **Info.plist** in the **Solution Pad** to open the **Project Options**.
 2. In the **Mac OS X Application Target** section and click the **App Icons** to select the **AppIcon.appiconset** file:



3. Save the changes.

When the app is run, the new icon will be displayed in the dock:



Summary

This article has taken a detailed look at working with Images required to create an macOS app Icon, packaging an Icon and including an Icon in a Xamarin.Mac project.

Related Links

- [MacImages \(sample\)](#)
- [Hello, Mac](#)
- [Working with Images](#)
- [macOS Human Interface Guidelines - Icons and Images](#)
- [About High Resolution for OS X](#)
- [Icns Builder](#)

Debugging a native crash in a Xamarin.Mac app

1/23/2020 • 8 minutes to read • [Edit Online](#)

Overview

Sometimes programming errors can cause crashes in the native Objective-C runtime. Unlike C# exceptions, these don't point to a specific line in your code you can look to fix. Sometimes they can be trivial to find and fix, and other times they can be extremely difficult to track down.

Let's walk through a few real native crash examples and take a look.

Example 1: Assertion failure

Here is the first few lines of a crash in a simple test application (this information will be in the **Application Output Pad**):

```
2014-10-15 16:18:02.364 NSOutlineViewHottness[79111:1304993] *** Assertion failure in -[NSTableView _uncachedRectHeightOfRow:], /SourceCache/AppKit/AppKit-1343.13/TableView.subproj/NSTableView.m:1855
2014-10-15 16:18:02.364 NSOutlineViewHottness[79111:1304993] NSTableView variable rowHeight error: The value must be > 0 for row 0, but the delegate <NSOutlineViewHottness_HotnessViewDelegate: 0xaa01860> gave -1.000.
2014-10-15 16:18:02.378 NSOutlineViewHottness[79111:1304993] *** Assertion failure in -[NSTableView _uncachedRectHeightOfRow:], /SourceCache/AppKit/AppKit-1343.13/TableView.subproj/NSTableView.m:1855
2014-10-15 16:18:02.378 NSOutlineViewHottness[79111:1304993] NSTableView variable rowHeight error: The value must be > 0 for row 0, but the delegate <NSOutlineViewHottness_HotnessViewDelegate: 0xaa01860> gave -1.000.
2014-10-15 16:18:02.381 NSOutlineViewHottness[79111:1304993] (
 0  CoreFoundation                      0x91888343 __raiseError + 195
 1  libobjc.A.dylib                     0x9a5e6a2a objc_exception_throw + 276
 2  CoreFoundation                      0x918881ca +[NSEception raise:format:arguments:] + 138
 3  Foundation                          0x950742b1 -[NSAssertionHandler
handleFailureInMethod:object:file:lineNumber:description:] + 118
 4  AppKit                             0x975db476 -[NSTableView _uncachedRectHeightOfRow:] + 373
 5  AppKit                             0x975db2f8 -[_NSTableRowHeightStorage _uncachedRectHeightOfRow:] +
143
 6  AppKit                             0x975db206 -[_NSTableRowHeightStorage _cacheRowHeights] + 167
 7  AppKit                             0x975db130 -[_NSTableRowHeightStorage _createRowHeightsArray] +
226
 8  AppKit                             0x975b5851 -[_NSTableRowHeightStorage _ensureRowHeights] + 73
 9  AppKit                             0x975b5790 -[_NSTableRowHeightStorage
computeTableHeightForNumberOfRows:] + 89
10  AppKit                            0x975b4c38 -[NSTableView _totalHeightOfTableView] + 220
```

The lines prefixed with numbers is the native stack trace. From that you can see the crash occurred somewhere inside `NSTableView` handling the row heights. Then `NSAssertionHandler` fires an `NSEException (objc_exception_throw)` and we see the Assertion failure:

```
rowHeight error: The value must be > 0 for row 0, but the delegate
<NSOutlineView_ViewDelegate: 0xaa01860> gave -1.000
```

Once you see this, it's pretty clear that some `NSOutlineViewDelegate` method is returning a negative number. This was the problem:

```

public override nfloat GetRowHeight (NSTableView tableView, nint row)
{
    return -1;
}

```

Example 2: Callback jumped into middle of nowhere

Stacktrace:

```

at <unknown> <0xffffffff>
at (wrapper managed-to-native) MonoMac.AppKit.NSApplication.NSApplicationMain (int,string[]) <IL 0x000a4, 0xffffffff>
at MonoMac.AppKit.NSApplication.Main (string[]) [0x00041] in /Users/donblas/Programming/xamcore-master/src/AppKit/NSApplication.cs:107
at NSOutlineViewHottness.MainClass.Main (string[]) [0x00007] in /Users/donblas/Programming/Local/NSOutlineViewHottness/NSOutlineViewHottness/Main.cs:14
at (wrapper runtime-invoke) <Module>.runtime_invoke_void_object (object,intptr,intptr,intptr) <IL 0x00050, 0xffffffff>

```

Native stacktrace:

Debug info from gdb:

```

(lldb) command source -s 0 '/tmp/mono-gdb-commands.qrH1lW'
Executing commands in '/private/tmp/mono-gdb-commands.qrH1lW'.
(lldb) process attach --pid 79229
Process 79229 stopped
Executable module set to
"/Users/donblas/Programming/Local/NSOutlineViewHottness/NSOutlineViewHottness/bin/Debug/NSOutlineViewHottness.app/Contents/MacOS/NSOutlineViewHottness".
Architecture set to: i386-apple-macosx.
(lldb) thread list
Process 79229 stopped
* thread #1: tid = 0x142776, 0x9af75e1a libsystem_kernel.dylib`__wait4 + 10, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
  thread #2: tid = 0x142790, 0x9af768d2 libsystem_kernel.dylib`kevent64 + 10, queue = 'com.apple.libdispatch-manager'
  thread #3: tid = 0x142792, 0x9af75e6e libsystem_kernel.dylib`__workq_kernreturn + 10
  thread #4: tid = 0x142794, 0x9af6fa6a libsystem_kernel.dylib`semaphore_wait_trap + 10
  thread #5: tid = 0x142795, 0x9af75772 libsystem_kernel.dylib`__recvfrom + 10
  thread #6: tid = 0x142799, 0x9af75e6e libsystem_kernel.dylib`__workq_kernreturn + 10
  thread #7: tid = 0x14279a, 0x9af75e6e libsystem_kernel.dylib`__workq_kernreturn + 10
  thread #8: tid = 0x14279b, 0x9af75e6e libsystem_kernel.dylib`__workq_kernreturn + 10
  thread #9: tid = 0x1427f8, 0x9af75e6e libsystem_kernel.dylib`__workq_kernreturn + 10
  thread #10: tid = 0x1427fe, 0x9af6fa2e libsystem_kernel.dylib`mach_msg_trap + 10
(lldb) thread backtrace all
* thread #1: tid = 0x142776, 0x9af75e1a libsystem_kernel.dylib`__wait4 + 10, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
  * frame #0: 0x9af75e1a libsystem_kernel.dylib`__wait4 + 10
    frame #1: 0x986bfb25 libsystem_c.dylib`waitpid$UNIX2003 + 48
    frame #2: 0x028ba36d libmono-2.0.dylib`mono_handle_native_sigsegv(signal=11, ctx=0x03115fe0) + 541 at mini-exceptions.c:2323
    frame #3: 0x0290a8bb libmono-2.0.dylib`mono_arch_handle_altstack_exception(sigctx=<unavailable>, fault_addr=<unavailable>, stack_ovf=0) + 155 at exceptions-x86.c:1159
    frame #4: 0x0280b4fd libmono-2.0.dylib`mono_sigsegv_signal_handler(_dummy=<unavailable>, info=<unavailable>, context=<unavailable>) + 445 at mini.c:6861
    frame #5: 0x91ef403b libsystem_platform.dylib`_sigtramp + 43
    frame #6: 0x9a5dd0bd libobjc.A.dylib`objc_msgSend + 45
    frame #7: 0x96bcec03 libsystem_trace.dylib`_os_activity_initiate + 89
    frame #8: 0x9773ba91 AppKit`-[NSApplication sendAction:to:from:] + 548
    frame #9: 0x9773b82d AppKit`-[NSControl sendAction:to:] + 102
    frame #10: 0x97934d36 AppKit`__26-[NSCell _sendActionFrom:]_block_invoke + 176
    frame #11: 0x96bcec03 libsystem_trace.dylib`_os_activity_initiate + 89
    frame #12: 0x97787975 AppKit`-[NSCell _sendActionFrom:] + 161
    frame #13: 0x979188ea AppKit`-[NSButtonCell _sendActionFrom:] + 55

```

```
frame #14: 0x979366e6 AppKit`__48-[NSCell trackMouse:inRect:ofView:untilMouseUp:]_block_invoke965 + 43
frame #15: 0x96bcec03 libsystem_trace.dylib`_os_activity_initiate + 89
frame #16: 0x977a3d00 AppKit`-[NSCell trackMouse:inRect:ofView:untilMouseUp:] + 2815
frame #17: 0x977a2df4 AppKit`-[NSButtonCell trackMouse:inRect:ofView:untilMouseUp:] + 524
frame #18: 0x977a233b AppKit`-[NSControl mouseDown:] + 762
frame #19: 0x97cbc112 AppKit`[_NSThemeWidget mouseDown:] + 378
frame #20: 0x97d36d74 AppKit`-[NSWindow _reallySendEvent:] + 12353
frame #21: 0x977201f9 AppKit`-[NSWindow sendEvent:] + 409
frame #22: 0x976cdc67 AppKit`-[NSApplication sendEvent:] + 4679
frame #23: 0x9754807c AppKit`-[NSApplication run] + 1003
```

This is an issue that was much more difficult to track down. When you see `at <unknown> <0xffffffff>` or `MonoMac.ObjCRuntime.Runtime.GetObject (IntPtr ptr)` at the top of the managed stack trace, it suggests we are trying to execute some managed code with an object that has been garbage collected. The native stack trace shows `trackMouse:inRect:ofView:untilMouseUp` into `NSCell _sendActionFrom` so we are somewhere handling a click event, trying to call back into C# and dying.

In general, errors like this are difficult to track down. I added `GC.Collect(2)` to a button handler to help track down this issue (forcing a garbage collection) to make the issue reproducible. After bisecting the example for a while, removing sections of code until the issue disappeared, it turned out to be [this bug](#):

```
mainWindowController.Window.StandardWindowButton (NSWindowButton.CloseButton).Activated += HandleActivated;
```

The `NSButton` returned by `StandardWindowButton()` was being collected even though an event was registered to it (that's the bug). When we try to call that event by clicking, if the button has been garbage collected we crash.

Although it wasn't the root cause of this particular issue, stack traces like this can also be caused by incorrect method signatures in functions `[Export]` ed to Objective-C. For example, if a method expects a parameter to be an `out string` and you type it as `string`, we can crash in the same way.

Example 3: Callbacks and managed objects

Many Cocoa APIs involve being "called back" by the library when some event occurs, giving you a chance to respond, or when some data is needed to carry out a task. Though you may think primarily of the **Delegate** and **DataSource** patterns, there are a multitude of APIs that work this way. For example, when you override the methods of an `NSView` and then insert it into the visual tree, you expect AppKit to call you back when certain events occur.

In almost all cases, Xamarin.Mac will correctly prevent the managed object target of these callbacks from being Garbage Collected while they still could be called back. However, rarely bugs in the binding can disrupt this. When this happens, you can see unpleasant crashes similar to this:

```

Thread 0 Crashed:: Dispatch queue: com.apple.main-thread

0  libsystem_kernel.dylib          0x98c2f69a __pthread_kill + 10
1  libsystem_pthread.dylib        0x90341f19 pthread_kill + 101
2  libsystem_c.dylib              0x9453feee abort + 156
3  libmonosgen-2.0.dylib         0x020bfba5 mono_handle_native_sigsegv + 757
4  libmonosgen-2.0.dylib         0x0210b812 mono_arch_handle_altstack_exception + 162
5  libmonosgen-2.0.dylib         0x0200c55e mono_sigsegv_signal_handler + 446
6  libsystem_platform.dylib       0x9513003b _sigtramp + 43
7  ???                           0xffffffff 0 + 4294967295
8  libmonosgen-2.0.dylib         0x0200c3a0 mono_sigill_signal_handler + 48
9  com.apple.AppKit               0x99f76041 -[NSView setFrame:] + 448
10 com.apple.AppKit               0x9a1fd4ea -[NSToolbarView adjustToWindow:attachedToEdge:] + 198
11 com.apple.AppKit               0x9a1fd414 -[NSToolbar _adjustViewToWindow] + 68
12 com.apple.AppKit               0x9a01eb0d -[NSToolbar _windowWillShowToolbar] + 79

```

This guide will help you track down bugs of this nature if they crop up, correctly report them so they can be fixed, and work around them in your code until then.

Locating

In almost every case with bugs of this nature, the primary symptom is native crashes, normally with something similar to `mono_sigsegv_signal_handler`, or `_sigtrap` in the top frames of the stack. Cocoa is attempting to call back into your C# code, hitting a garbage collected object, and crashing. However, not every crash with these symbols is caused by a binding issue like this, you'll need to do some additional digging to confirm this is the problem.

What makes these bugs difficult to track down is that they only occur **after** a garbage collection has disposed of the object in question. If you believe you've hit one of these bugs, add the following code somewhere in your startup sequence:

```

new System.Threading.Thread (() =>
{
    while (true) {
        System.Threading.Thread.Sleep (1000);
        GC.Collect ();
    }
}).Start ();

```

This will force your application to run the garbage collector every second. Re-run your application and try to reproduce the bug. If you crash immediately, or consistently instead of randomly, you are on the right track.

Reporting

The next step is to report the issue to Xamarin so the binding can be fixed for future releases. If you are a business or enterprise license holder, open a ticket at

visualstudio.microsoft.com/vs/support/

Otherwise, search for an existing issue:

- Check the [Xamarin.Mac Forums](#)
- Search the [issue repository](#)
- Before switching to GitHub issues, Xamarin issues were tracked on [Bugzilla](#). Please search there for matching issues.
- If you cannot find a matching issue, please file a new issue in the [GitHub issue repository](#).

GitHub issues are all public. It's not possible to hide comments or attachments.

Please include as much of the following as possible:

- A simple example reproducing the issue. This is **invaluable** where possible.
- The full stack trace of the crash.
- The C# code surrounding the crash.

Working around

Once you've tracked down the issue, patching the issue with a work around until the binding can be fixed can be simple. The aim is to prevent the object (**View**, **Delegate**, **DataSource**) that is incorrectly being disposed from leaving memory by keeping an open reference.

For simple cases where there is only a single instance of the object, change the code from this:

```
void AddObject ()  
{  
    item.View = new MyView ();  
    ...  
}
```

To using a static variable such as this:

```
static NSObject view;  
...  
  
void AddObject ()  
{  
    view = new MyView ();  
    item.View = view;  
    ...  
}
```

In cases where multiple instances may be created, a static `HashSet` can be employed:

```
static HashSet<NSObject> collection = new HashSet<NSObject> ();  
...  
  
void AddObject ()  
{  
    item.View = new MyView ();  
    collection.Add (item.View );  
    ...  
}
```

Once the binding has been fixed and you've upgrade to the version of Xamarin.Mac that includes the fix, the work-around code can be removed.

Exception bubbling and Objective-C

You should never allow a C# Exception to "escape" managed code to the calling Objective-C method. If you do, the results are undefined but generally involve crashing. In general, we do everything we can to bubble up useful information for both native and managed crashes to help you solve your issues quickly.

Without getting too bogged down with the technical reasons why, setting up the infrastructure to catch managed exceptions at every managed/native boundary is non-trivially expensive and there are a *lot* of transitions that happen in many common operations. Many operations, specifically ones that involve the UI thread must finish quickly or your app will stutter and have unacceptable performance characteristics. Many of those callbacks do very simple things that rarely have the possibility of throwing, so this overhead would both

be expensive and unnecessary in those cases.

Thus, we'd don't set up those try / catches for you. For places where you code does non-trivial things (beyond say returning a booleans or simple math), you can try catch yourself.

Xamarin.Mac linker options

10/28/2019 • 2 minutes to read • [Edit Online](#)

Linking is a powerful optimization tool that reduces the size of your application by removing unused code.

Overview

Based upon the [Target Framework](#) your project uses, the linker options available may be limited. This is due to the fact that linking requires the creation of an object graph of every type used by your application and this is not possible in Full (or Unsupported) due to `System.Configuration`.

There are four options available:

- **None** – Disable all linking. Default in Debug configuration in Modern and all configurations in Full.
- **SDK** – Links all SDK assembly, excluding user assemblies. Default in Release configuration in Modern.
Unavailable on Full.
- **Full** – Link all assemblies. This requires user code to be linker safe, see the [notes](#) for more information.
Unavailable on Full.
- **Platform** – Link just `Xamarin.Mac.dll`. See below for details.

Platform linking

Linking applications using the Full [Target Framework](#) is generally unsafe, but there are a number of scenarios where a very limited form of linking is required.

For example, Applications submitted to the macOS App Store must not reference a number of deprecated APIs (such as `QTKit`), some of which Xamarin.Mac contains bindings of. Even if an application does not call those bindings, the invocation will exist in the SDK and be rejected.

Platform linking assumes the application and the BCL are linker unsafe and just remove unused code from `Xamarin.Mac.dll`.

Any applications not reflecting on `Xamarin.Mac.dll` types will see a minor startup improvement from the removal of unnecessary types.

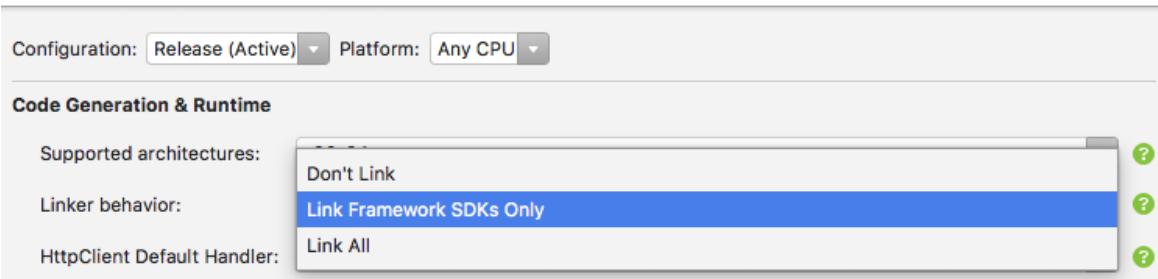
Platform linking is generally only useful for applications using the Full target framework, as Modern application can use the more powerful SDK option.

Setting the linker configuration

To change to the linker configuration for a Xamarin.Mac project, do the following:

1. Open the Xamarin.Mac project in Visual Studio for Mac.
2. In the **Solution Explorer**, double-click the project file to open the **Project Options** dialog box.
3. From the **Mac Build** tab, select the type of **Linker Behavior** that suits your application's needs:

Mac Build



4. Platform linking for Full Target Frameworks will not appear in the IDE until a future update. Until then, add `--linkplatform` to the **Additional mmp arguments** instead.
5. Click the OK button to save your changes.

Related Links

- [Linking on iOS](#)

Xamarin.Mac performance

11/2/2020 • 2 minutes to read • [Edit Online](#)

Overview

Xamarin.Mac applications are similar to Xamarin.iOS, and many of the same performance suggestions are applicable:

- [Xamarin.iOS performance](#)
- [Cross-platform performance](#)

but there are a number of macOS specific suggestions that may be helpful.

Prefer modern target framework

There are multiple [Target Frameworks](#) available to Xamarin.Mac application with different performance characteristics and features.

When possible, prefer Modern and work with dependent libraries to add support. Only the Modern Target Framework allows linking which can drastically reduce assembly sizes. This becomes especially important when enabling AOT, as AOT compilation of Full assemblies can produce large final bundles.

Enable the linker

Startup time, both in load and "Just In Time" (JIT), scales somewhat linearly with the size of your final binaries. The easiest way to improve this is by removing dead code with the [linker](#).

While this suggestion primarily applies to Modern Target Framework users, use of [Platform Linking](#) can provide a limited performance boost as well.

Enable AOT when appropriate

Another facet of startup performance is the JIT compilation of assemblies into machine code. Ahead of Time (AOT) compilation can significantly reduce startup time, but comes with a number of tradeoffs covered in the [AOT documentation](#).

Ensure performant delegates

Many Xamarin.Mac applications are centered around Cocoa views such as `NSCollectionView`, `NSOutlineView`, or `NSTableView`. Often these views are powered by `Delegate` and `DataSource` classes you provide to Cocoa, answering questions on what to display.

Many of these entry points are invoked often, sometimes multiple times per second when scrolling.

Make sure these functions return values that are easily calculated or use information already cached, to prevent the user interface from blocking.

Use Cocoa-provided APIs for reusing views

Many Cocoa views that contain many child views or cells (such as `NSCollectionView`, `NSOutlineView`, and `NSTableView`) provide APIs for creating and reusing views. These create pools of shared items and prevent performance issues when scrolling through the views quickly.

Use `async` and do not block the UI

Desktop applications often process large quantities of data and it is very easy to block the UI thread waiting on a synchronous operation.

Whenever possible, use `async` and threads to prevent blocking the UI.

For long running operations consider using [NSProgressIndicator](#) or other options noted in Apple's [HIG](#) to notify users.

Related Links

- [Cross-platform performance](#)
- [Xamarin.iOS performance](#)

Publishing Xamarin.Mac Apps to the Mac App Store

1/23/2020 • 2 minutes to read • [Edit Online](#)

Overview

Xamarin.Mac apps can be distributed in two different ways:

- **Developer ID** – Applications signed with a Developer ID can be distributed outside of the App Store but are recognized by GateKeeper and allowed to install.
- **Mac App Store** – Apps must have an installer package, and both the app and the installer must be signed, for submission to the Mac App Store.

This document explains how to use Visual Studio for Mac and Xcode to setup a Apple Developer account and configure a Xamarin.Mac project for each deployment type.

Mac developer program

When you join the [Mac Developer Program](#) the developer will be offered a choice to join as an Individual or a Company, as shown in the screenshot below:

Enter Account Info Select Program Review & Submit Agree to License Purchase Program Activate Program

Are you enrolling as an individual or company?

Individual

Select this option if you are an individual or sole proprietor/single person company.



Individual Development Only

You are the only one allowed access to program resources.



App Store Distribution

Your name will appear as the "seller" for apps you distribute on the App Store.

[View example](#)



You will need:

- Credit card billing information.
- A valid credit card for purchase.
We may also require additional personal documentation to verify your identity.

Company

Select this option if you are a company, non-profit organization, joint venture, partnership, or government organization.



Development Team

You can add additional developers to your team who can access program resources. Companies who have hired a contractor to create apps for distribution on the App Store should enroll with their company name and add the contractors to their team.



App Store Distribution

Your legal entity name will appear as the "seller" for apps you distribute on the App Store.

[View example](#)



You will need:

- The legal authority to bind your company/organization to Apple Developer Program legal agreements.
- An address for the company's principal place of business or corporate headquarters.
- A D-U-N-S® Number assigned to a legal entity. D-U-N-S Numbers, available from D&B for free in most jurisdictions, are unique nine-digit numbers widely used as standard business identifiers. To learn more, read our [FAQs](#). Before enrolling, check to see if D&B has assigned you a D-U-N-S Number. If not, please request one. [Check now ▶](#)
- Note: We do not accept DBAs, Fictitious Business, or Trade names at this time.
- A valid credit card for purchase.

[Individual](#)

[Company](#)

Choose the correct enrollment type for your situation.

NOTE

The choices made here will affect the way some screens appear when configuring a developer account. The descriptions and screenshots in this document are done from the perspective of an **Individual** developer account. In a **Company**, some options will only be available to **Team Admin** users.

Certificates and identifiers

This guide walks through creating the necessary Certificates and Identifiers that will be required to publish a Xamarin.Mac app.

Create provisioning profile

This guide walks through creating the necessary Provisioning Profiles that will be required to publish a Xamarin.Mac app.

Mac app configuration

This guide walks through configuring a Xamarin.Mac app for publication.

Sign with Developer ID

This guide walks through signing a Xamarin.Mac app with a Developer ID for publication.

Bundle for Mac App Store

This guide walks through bundling a Xamarin.Mac app for publication to the Mac App Store.

Upload to Mac App Store

This guide walks through uploading a Xamarin.Mac app for publication to the Mac App Store.

Related links

- [Installation](#)
- [Hello, Mac sample](#)
- [Developer ID and GateKeeper](#)

Certificates and identifiers in Xamarin.Mac

3/5/2021 • 4 minutes to read • [Edit Online](#)

This guide walks through creating the necessary certificates and identifiers that will be required to publish a Xamarin.Mac app.

Setup

Visit the [Apple Developer Member Center](#) to configure the Mac for development. Click on the **Account** link and sign-in. The main menu is shown below:

The screenshot shows the Apple Developer Member Center interface. At the top, there's a navigation bar with the Apple logo and the word "Developer". Below it, the "Account" tab is selected. On the left, a sidebar menu lists "Program Resources" with options like "Overview", "Membership", "Certificates, IDs & Profiles", "App Store Connect", "CloudKit Dashboard", and "Code-Level Support". Under "Additional Resources", there are links for "iTunes Connect", "iCloud.com", "App Store Connect", and "Xcode". The main content area is titled "Apple Developer Program" and contains two sections: "Certificates, Identifiers & Profiles" (with a circular icon) and "App Store Connect" (with a stylized 'A' icon). The "Certificates, Identifiers & Profiles" section has a sub-description: "Manage the certificates, identifiers, profiles, and devices you need to develop and distribute apps."

Click on the **Certificates, Identifiers & Profiles** button (or the plus button near the **Certificates** heading):

The screenshot shows the "Certificates, Identifiers & Profiles" page. At the top, there's a header with "Certificates" and a plus sign, "Identifiers", "Devices", "Profiles", "Keys", and "More". To the right is a search bar with "All Types" and a magnifying glass icon. Below the header, there's a section titled "Getting Started with Certificates" with a sub-description: "Certificates identify who signed an app or is accessing a service. You'll need to set up various Apple-issued digital certificates to develop and distribute apps and connect to app services." At the bottom of this section is a blue "Create a certificate" button.

Select a certificate type and click **Continue**:

Intermediate Certificates

To use your certificates, you must have the intermediate signing certificate in your system keychain. This is automatically installed by Xcode. However, if you need to reinstall the intermediate signing certificate click the link below:

[Worldwide Developer Relations Certificate Authority >](#)

From here you can download the **Intermediate Certificates** (Worldwide Developer Relations Certificate Authority and Developer ID Certificate Authority) if required (last item at the bottom of the page). However, these should automatically be setup for the developer by Xcode.

The remainder of this section walks through the sections relevant to Mac developers:

- **Register Mac App ID** – The developer will need to follow these steps for each application they write.
- **Register macOS Systems** – This is only required when adding computers to test with.
- **Create Certificates** – Only required once when setting-up the certificates, and later when renewing them.
- **Create Provisioning Profile** – The developer will need to follow these steps for each new application written, and when adding new systems.

Register Mac App ID

You need to register an App ID for every application. Follow the steps below to create an entry:

1. Press the "+" (plus sign) or **Register an App ID**:

Certificates, Identifiers & Profiles

The screenshot shows the Apple Developer portal's "Certificates, Identifiers & Profiles" page. The left sidebar has tabs for Certificates, Identifiers (which is selected and highlighted in grey), Devices, Profiles, Keys, and More. The main content area is titled "Identifiers" with a blue circular icon containing a plus sign. Below it is a search bar with the placeholder "App IDs". A large central box is titled "Getting Started with App IDs" and contains the text: "Register an App ID to enable your app to access available services and identify your app in a provisioning profile. You can enable app services when you create an App ID or modify these settings later." At the bottom of this box is a blue "Register an App ID" button.

2. Choose App IDs

Register a New Identifier

[Continue](#)

App IDs

Register an App ID to enable your app to access available services and identify your app in a provisioning profile. You can enable app services when you create an App ID or modify these settings later.

Services IDs

For each website that uses Sign In with Apple, register a service identifier (Service ID), configure your

3. Enter a **Description**, and select any **App Services** that the application will require:
a. Platform should be **macOS**
a. Choose a **Description** (used only in this portal)
a. Enter the **Bundle ID**, which should match your **Info.plist**
a. Select the capabilities that your app requires

Register an App ID

[Back](#) [Continue](#)

Platform

iOS, tvOS, watchOS macOS

App ID Prefix

7852235SN6 (Team ID)

Description

sourcewriter appid

You cannot use special characters such as @, &, *, ', "

Bundle ID

Explicit

Wildcard

com.xamarin.sourcewriter

We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).

Capabilities

ENABLED NAME

 Associated Domains

 Custom Network Protocol

 Game Center

 iCloud

Include CloudKit support (requires Xcode 6)

Compatible with Xcode 5

Press **Continue** to review your selections.

4. If the information is correct, click **Register** to complete the setup:

Confirm your App ID

[Back](#) [Register](#)

Platform

macOS

App ID Prefix

7852235SN6 (Team ID)

Description

sourcewriter appid

Bundle ID

com.xamarin.sourcewriter (explicit)

5. Verify the information and click the **Submit** button:



Certificates, Identifiers & Profiles

Certificates

Identifiers

 App IDs 

Identifiers

Devices

Profiles

Keys

More

NAME	IDENTIFIER	PLATFORM
sourcewriter appid	com.xamarin.sourcewriter	macOS

Some App Services might require further configuration (for example, iCloud). If that is the case, select the new App ID just created and click the **Edit** button:

Edit your App ID Configuration

[Remove](#) [Save](#)

Platform	macOS
Description	<input type="text" value="sourcewriter appid"/> You cannot use special characters such as @, &, *, ', "

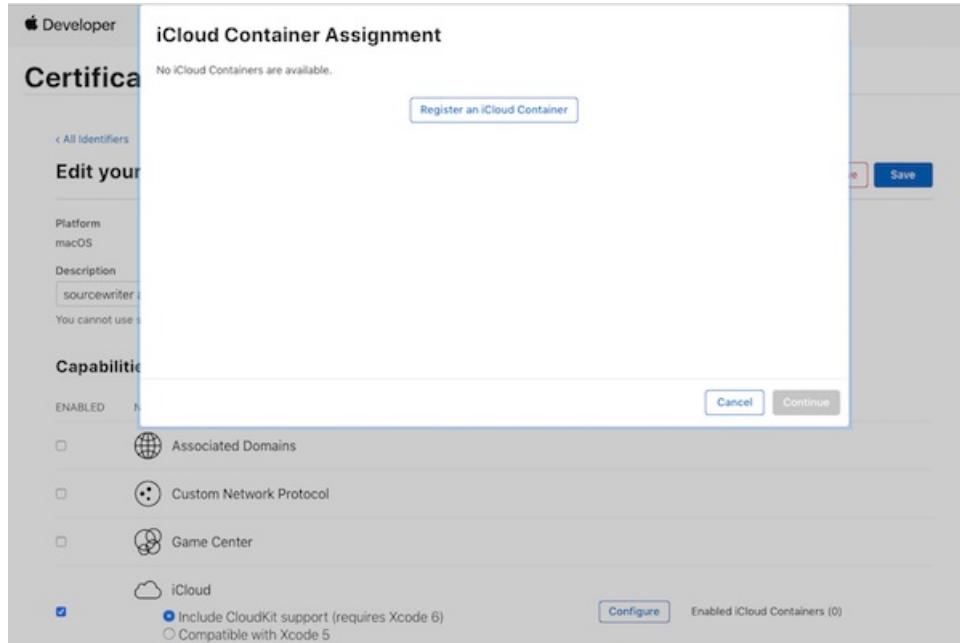
App ID Prefix
7852235SN6 (Team ID)
Bundle ID
com.xamarin.sourcewriter (explicit)

You cannot use special characters such as @, &, *, ', "

Capabilities

ENABLED	NAME
<input type="checkbox"/>	Associated Domains
<input type="checkbox"/>	Custom Network Protocol
<input type="checkbox"/>	Game Center
<input checked="" type="checkbox"/>	iCloud
	<input checked="" type="radio"/> Include CloudKit support (requires Xcode 6) <input type="radio"/> Compatible with Xcode 5
	Configure Enabled iCloud Containers (0)

To configure the iCloud services, for example, click the Edit button:



The screenshot shows the 'iCloud Container Assignment' dialog box. It lists the 'iCloud' capability with the 'Include CloudKit support (requires Xcode 6)' option checked. There are 'Cancel' and 'Continue' buttons at the bottom.

Register macOS devices

To create a provisioning profile for testing, the developer will need to have their Mac computers registered. A maximum of 100 computers can be registered for testing.

1. In the Mac Developer Center, select All from the Devices section and click the + button:

Certificates, Identifiers & Profiles

Certificates	Devices +		Edit
Identifiers	NAME	IDENTIFIER	TYPE
Devices	iPad Pro	68d0da166d979e339b6b0d77a6d24h1234567890	iPad
Profiles			
Keys			
More			

2. Enter a Name and the UUID of the computer to add and click the Continue button. Review the

information and the click **Register** button:

Register a New Device

Continue

Register Devices

To create a provisioning profile for app testing and ad hoc distribution, you'll need to specify registered devices. If you use automatic signing, Xcode registers connected devices for you. Xcode Server can also be configured to register connected devices.

Note: If you remove a registered device from your account, it will continue to count against your device limit. At the start of your new membership year, Account Holders and Admins will be presented with the option to remove listed devices and restore the available device count.

Register a Device

Name your device and enter its Unique Device Identifier (UDID).

Platform

macOS

Device Name

My Mac

Device ID (UUID)

|

Register Multiple Devices

Upload a file containing the devices you wish to register. Please note that a maximum of 100 devices can be included in your file and it may take a few minutes to process.

[Download sample files >](#)

Device List

3. Review and confirm the data entered:

Register a New Device

Back

Register

Confirm the device information is correct. Once this device is registered, you will not be able to edit the UDID and can only edit the name or disable it.

Name: My Mac

UUID: E63B0E5F-3333-5555-8888-AA1234567890

Test Device Details

You have the following number of devices available for registration. You may reset your device list at the start of your next membership year.

Apple TV	iPad	iPod
100	99	100
Apple Watch	iPhone	Mac
100	100	100

Create certificates

Use the Certificates section to create several different types of certificates that will be used to sign Mac Applications:

Create a New Certificate

Continue

Software

- Apple Development**
Sign development versions of your iOS, macOS, tvOS, and watchOS apps. For use in Xcode 11 or later.
- Apple Distribution**
Sign your apps for submission to the App Store or for Ad Hoc distribution. For use with Xcode 11 or later.
- iOS App Development**
Sign development versions of your iOS app.
- iOS Distribution (App Store and Ad Hoc)**
Sign your iOS app for submission to the App Store or for Ad Hoc distribution.
- Mac Development**
Sign development versions of your Mac app.
- Mac App Distribution**
This certificate is used to code sign your app and configure a Distribution Provisioning Profile for submission to the Mac App Store.

There are five main types of certificate relevant to macOS development:

- **Mac Development** – Optional for general app development, but required if the developer plans to use features like iCloud or push notifications. The developer will need a Development Certificate before they can create Provisioning Profiles that allow them to access those features.
- **Mac App Distribution** – The developer will need a certificate for their app and another certificate for the installer.
- **Mac Installer Distribution** – The developer will need a certificate for their app and another certificate for the installer.
- **Developer ID Installer** – Certificates for the installer to distribute outside the Mac App Store.
- **Developer ID Application** – Certificates for the app to distribute outside the Mac App Store.

The following sections will provide examples of creating some of these certificate types.

Mac development certificate

As mentioned previously, Mac Development certificate isn't required unless macOS features like iCloud or push notifications are being used.

Do the following to created a new Development Certificate:

1. Select the **Mac Development** radio button and click **Continue**:

Create a New Certificate

Continue

Software

Apple Development

Sign development versions of your iOS, macOS, tvOS, and watchOS apps. For use in Xcode 11 or later.

Apple Distribution

Sign your apps for submission to the App Store or for Ad Hoc distribution. For use with Xcode 11 or later.

iOS App Development

Sign development versions of your iOS app.

iOS Distribution (App Store and Ad Hoc)

Sign your iOS app for submission to the App Store or for Ad Hoc distribution.

Mac Development

Sign development versions of your Mac app.

2. Upload a *certificate signing request*. The certificate request file (extension `.certSigningRequest`) will be saved locally on the Mac. Click **Choose file** to select the certificate request, then press **Continue**.

Create a New Certificate

Back

Continue

Upload a Certificate Signing Request

To manually generate a Certificate, you need a Certificate Signing Request (CSR) file from your Mac.

[Learn more >](#)

[Choose File](#)

Follow the [Learn more >](#) link for instructions on how to use **Keychain Access** to create a certificate request file.

3. Press **Download** to get the certificate file, and double-click it to install:

Download Your Certificate

[Download](#)

Certificate Details

Certificate Name
Craig

Certificate Type
Mac Development

Download your certificate to your Mac, then double click the .cer file to install in Keychain Access. Make sure to save a backup copy of your private and public keys somewhere secure.

Expiration Date
2020/12/17

Created By
Craig

As previously mentioned the Developer certificate is not always required unless the developer is implementing macOS features like iCloud and push notifications. It is also required to create a **Development Provisioning Profile**, which will be needed to test Mac App Store apps.

Mac App Store certificates

To release an app on the App Store, you'll need two certificates:

- **Mac App Distribution** certificate that will be used to sign the application; and
- **Mac Installer Distribution** certificate, to sign the installer.

TIP

Be careful when naming the certificate requests for these keys: use descriptive names that include the text **Application** and **Installer** so they can be distinguished later.

First, create the installer certificate:

1. Select **Mac Installer Distribution** as the certificate type and click the **Continue** button:

 Mac App Distribution

This certificate is used to code sign your app and configure a Distribution Provisioning Profile for submission to the Mac App Store.

 Mac Installer Distribution

This certificate is used to sign your app's Installer Package for submission to the Mac App Store.

2. The next page explains how to use **Keychain Access** to generate a certificate request file. Follow the instructions:

Create a New Certificate

[Back](#)[Continue](#)**Upload a Certificate Signing Request**

To manually generate a Certificate, you need a Certificate Signing Request (CSR) file from your Mac.

[Learn more >](#)[Choose File](#)

Follow the [Learn more >](#) link for instructions on how to use **Keychain Access** to create a certificate request file. Remember to choose a certificate name that reflects the *type* of certificate (Application or Installer).

3. Click **Download** to get your certificate and double-click to install it in the **Keychain**:

Download Your Certificate

[Download](#)**Certificate Details**

Certificate Name
Craig

Certificate Type
Mac Installer Distribution

Download your certificate to your Mac, then double click the .cer file to install in Keychain Access. Make sure to save a backup copy of your private and public keys somewhere secure.

Expiration Date
2020/12/17

Created By
Craig

Follow the same steps for the Mac App Distribution certificate.

**Mac App Distribution**

This certificate is used to code sign your app and configure a Distribution Provisioning Profile for submission to the Mac App Store.

**Mac Installer Distribution**

This certificate is used to sign your app's Installer Package for submission to the Mac App Store.

Developer ID certificates

To self-release a Xamarin.Mac application (not release via the Apple App Store), you'll need two certificates:

- **Developer ID Installer** certificate that will be used to sign the application; and
- **Developer ID Application** certificate, to sign the installer.

TIP

Be careful when naming the certificate requests for these keys: use descriptive names that include the text `Application` and `Installer` so they can be distinguished later.

Once you have created, downloaded, and installed certificates, they'll be visible in **Keychain Access**:

[Keychain Access certificate list](#)

Related links

- [Installation](#)
- [Hello, Mac sample](#)
- [Distribute your apps on the Mac App Store](#)
- [Developer ID and GateKeeper](#)

Provisioning profiles for Xamarin.Mac apps

3/5/2021 • 2 minutes to read • [Edit Online](#)

Provisioning profiles allow a developer to incorporate several macOS (formerly known as Mac OS X) specific features (such as iCloud and Push Notifications) into their Xamarin.Mac apps. They must create, download and install a Mac Provisioning Profile for each application they are developing that use these features.

The screenshot shows the Apple Developer portal interface. The top navigation bar includes links for Technologies, Resources, Programs, Support, Member Center, and a search bar. The main area is titled 'Certificates, Identifiers & Profiles' and is specifically set to 'Mac Apps'. On the left, there are filters for Certificates (All, Pending, Development, Production), Identifiers (App IDs, Website Push IDs, iCloud Containers), Devices (All), and Provisioning Profiles (All, Development, Distribution). The 'Development' filter is currently selected. In the center, a table titled 'Mac Provisioning Profiles (Development)' displays one profile: 'MacTeam Provisioning Profile: c...', which is a 'Mac Development' type profile, marked as 'Active (Managed by Xcode)'.

Development provisioning profile

A Development Provisioning Profile allows a Mac App Store-targeted app to be tested on the specific computers that have been set-up in the profile. This is particularly relevant when using macOS features like iCloud and Push Notifications.

NOTE

The developer must have already created a Mac Development Certificate before they can create a Development Provisioning Profile. Complete the details as shown on this screenshot to generate a **Development Provisioning Profile** that can be used to create builds. There must be a valid Mac Development Certificate available for selection in the **Certificate** box, and at least one system registered for testing.

Do the following:

1. Select the type of Provisioning Profile that to create and click the **Continue** button:

Developer

Technologies Resources Programs Support Member Center Search Developer

Certificates, Identifiers & Profiles Kevin Mullins ▾

Mac Apps

Select Type Configure Generate Download

What type of provisioning profile do you need?

Development

Mac App Development
Create a provisioning profile to install development apps on test devices.

Distribution

Mac App Store
Create a distribution provisioning profile to submit your app to the Mac App Store.

Cancel Continue

2. Select the ID of the Application to create the profile for and click the **Continue** button:

Developer

Technologies Resources Programs Support Member Center Search Developer

Certificates, Identifiers & Profiles Kevin Mullins ▾

Mac Apps

Select App ID.

If you plan to use services such as Game Center, In-App Purchase, and Push Notifications, or want a Bundle ID unique to a single app, use an explicit App ID. If you want to create one provisioning profile for multiple apps or don't need a specific Bundle ID, select a wildcard App ID. Wildcard App IDs use an asterisk (*) as the last digit in the Bundle ID field. Please note that iOS App IDs and Mac App IDs cannot be used interchangeably.

App ID: CloudKit Test Mac (Y2HRRTRAA5.com.appracatappra.cloudtestmac)

Cancel Back Continue

3. Select the developer ID used to sign the profile and click **Continue**:

Developer

Technologies Resources Programs Support Member Center Q Search Developer

Certificates, Identifiers & Profiles Kevin Mullins ▾

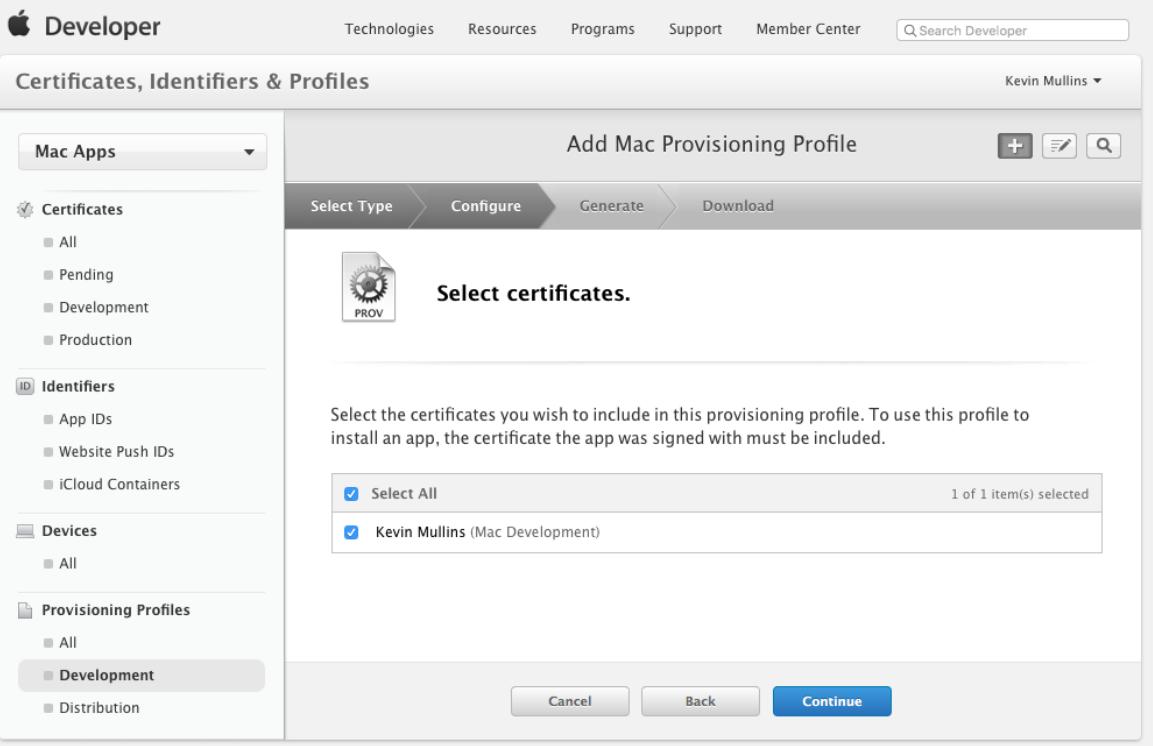
Mac Apps

Select certificates.

Select the certificates you wish to include in this provisioning profile. To use this profile to install an app, the certificate the app was signed with must be included.

Select All 1 of 1 item(s) selected
 Kevin Mullins (Mac Development)

Cancel Back Continue



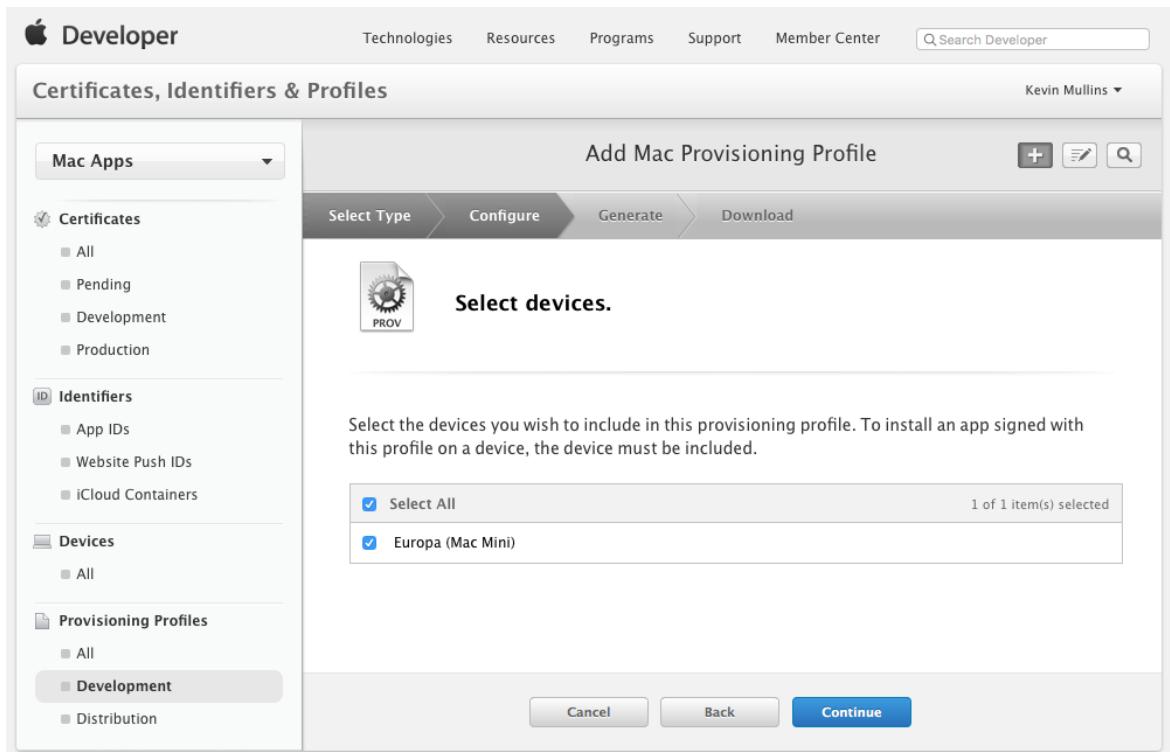
4. Select the computers that this profile can be used on and click **Continue**:

Select devices.

Select the devices you wish to include in this provisioning profile. To install an app signed with this profile on a device, the device must be included.

Select All 1 of 1 item(s) selected
 Europa (Mac Mini)

Cancel Back Continue



5. Now, enter a **Profile Name** and click the **Generate** button:

Developer

Technologies Resources Programs Support Member Center Search Developer

Certificates, Identifiers & Profiles Kevin Mullins ▾

Mac Apps

Certificates
All Pending Development Production

Identifiers
App IDs Website Push IDs iCloud Containers

Devices
All

Provisioning Profiles
All **Development** Distribution

Add Mac Provisioning Profile + ⌂ ⌂

Select Type Configure Generate Download

 Name this profile and generate.

The name you provide will be used to identify the profile in the portal.

Profile Name: CloudKit Test

Type: Mac Development

App ID: CloudKit Test Mac (Y2HRRTRAAS.com.appracatappra.cloudtestmac)

Certificates: 1 Included

Devices: 1 Included

Cancel Back Generate

6. Click the Download button to download the new profile:

Developer

Technologies Resources Programs Support Member Center Search Developer

Certificates, Identifiers & Profiles Kevin Mullins ▾

Mac Apps

Certificates
All Pending Development Production

Identifiers
App IDs Website Push IDs iCloud Containers

Devices
All

Provisioning Profiles
All **Development** Distribution

Add Mac Provisioning Profile + ⌂ ⌂

Select Type Configure Generate Download

 Your provisioning profile is ready.

Download and Install
Download and double click the following file to install your Provisioning Profile.

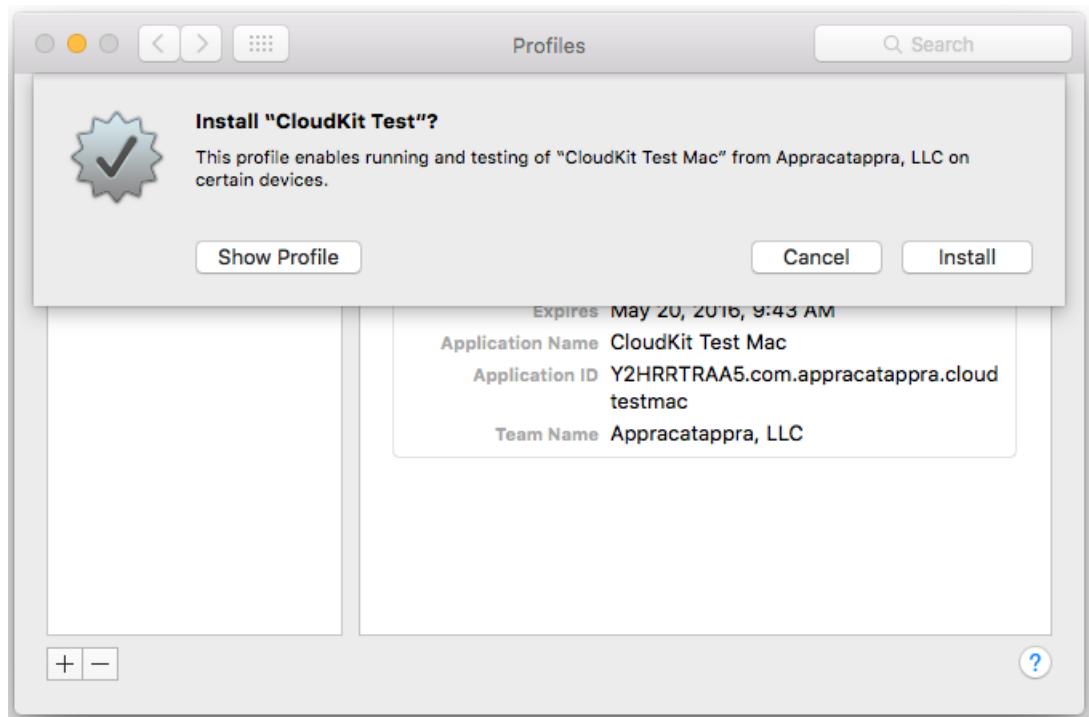
 Name: CloudKit Test
Type: Mac Development
App ID: Y2HRRTRAAS.com.appracatappra.cloudtestmac
Expires: Jun 14, 2016

Download

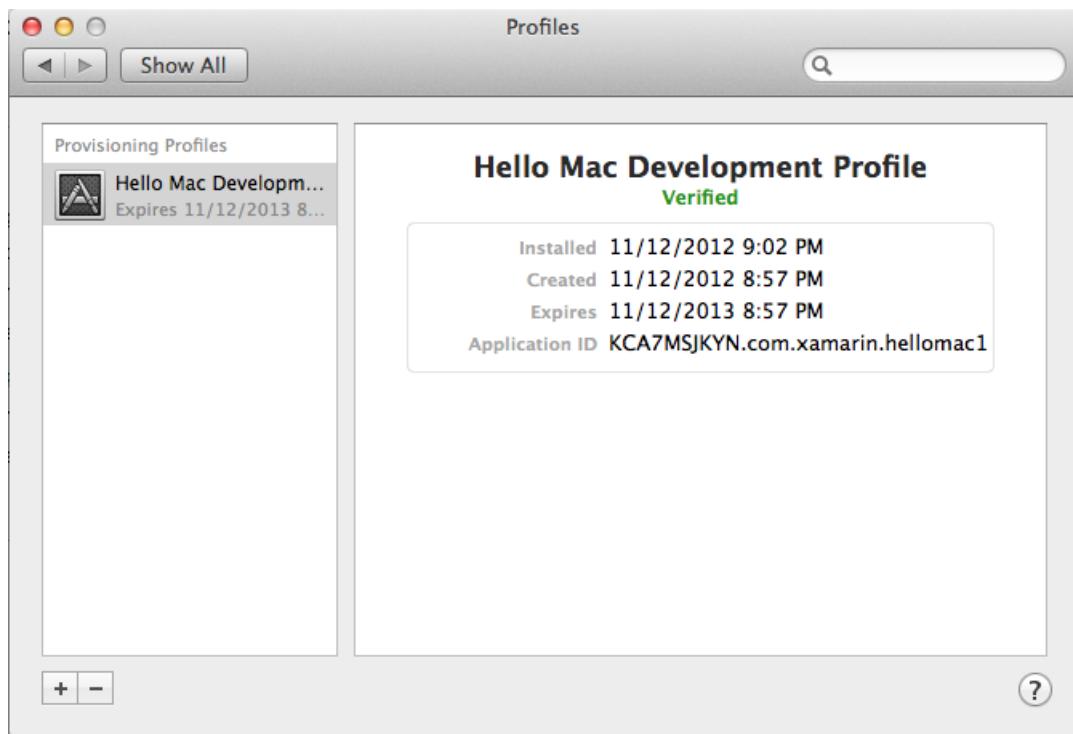
Documentation
For more information on using and managing your Provisioning Profile read:
[App Distribution Guide](#)

Add Another Done

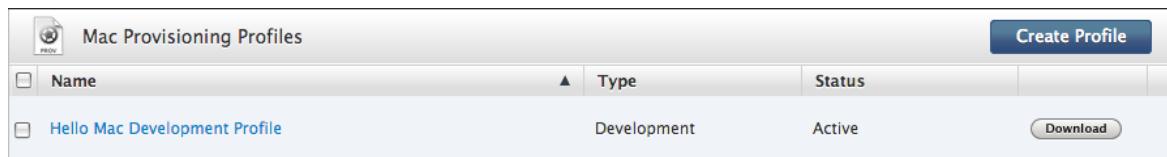
7. Development provisioning profiles are installed to the Profiles Preferences pane of the Mac's System Preferences application:



8. The Profile Preferences pane will show all installed profiles:



9. The profile will also appear in the Developer Certificate Utility in case it needs to be downloaded again:



A new Development Provisioning Profile will need to be created for each new app or when a new computer is being added to test on.

Production provisioning profile

Production provisioning profiles are required to build a package for submission to the Mac App Store.

Do the following:

1. Select the type of profile to create and click the **Continue** button:

The screenshot shows the Apple Developer portal's 'Certificates, Identifiers & Profiles' section. On the left, there's a sidebar with filters for 'Mac Apps' (selected), 'Certificates' (All, Pending, Development, Production), 'Identifiers' (App IDs, Website Push IDs, iCloud Containers), 'Devices' (All), and 'Provisioning Profiles' (All, Development, Distribution). The main area is titled 'Add Mac Provisioning Profile' with a progress bar: 'Select Type' (active), 'Configure', 'Generate', and 'Download'. A sub-section titled 'What type of provisioning profile do you need?' asks 'Development' or 'Distribution'. Under 'Distribution', the 'Mac App Store' option is selected, with the sub-instruction 'Create a distribution provisioning profile to submit your app to the Mac App Store.' At the bottom are 'Cancel' and 'Continue' buttons.

2. Select the ID of the app to create the profile for and click the **Continue** button:

This screenshot continues the 'Add Mac Provisioning Profile' wizard. The 'Select App ID' step is shown, with a note explaining that explicit App IDs are needed for services like Game Center, In-App Purchase, and Push Notifications. It also notes that wildcard App IDs can be used for multiple apps. Below this is a dropdown menu labeled 'App ID:' containing the value 'CloudKit Test Mac (Y2HRRTRAA5.com.appracatappra.cloudtestmac)'. At the bottom are 'Cancel', 'Back', and 'Continue' buttons.

3. Select the company ID to sign the profile and click the **Continue** button:

Developer

Technologies Resources Programs Support Member Center Search Developer Kevin Mullins ▾

Certificates, Identifiers & Profiles

Mac Apps ▾

Certificates

- All
- Pending
- Development
- Production

Identifiers

- App IDs
- Website Push IDs
- iCloud Containers

Devices

- All

Provisioning Profiles

- All
- Development
- Distribution

Add Mac Provisioning Profile

Select Type Configure Generate Download

Select certificates.

Select the certificates you wish to include in this provisioning profile. To use this profile to install an app, the certificate the app was signed with must be included.

Appracatappra, LLC (Mac App Distribution)
Nov 21, 2015

Cancel Back Continue

4. Enter a **Profile name** and click the **Generate** button:

Developer

Technologies Resources Programs Support Member Center Search Developer Kevin Mullins ▾

Certificates, Identifiers & Profiles

Mac Apps ▾

Certificates

- All
- Pending
- Development
- Production

Identifiers

- App IDs
- Website Push IDs
- iCloud Containers

Devices

- All

Provisioning Profiles

- All
- Development
- Distribution

Add Mac Provisioning Profile

Select Type Configure Generate Download

Name this profile and generate.

The name you provide will be used to identify the profile in the portal.

Profile Name: CloudKit App Store

Type: Mac Production

App ID: CloudKit Test Mac
(Y2HRRTRAA5.com.appracatappra.cloudtestmac)

Certificates: 1 Included

Cancel Back Generate

5. Click **Download** to get the provisioning profile file (extension `.provisionprofile`):

Developer

Technologies Resources Programs Support Member Center Search Developer

Certificates, Identifiers & Profiles Kevin Mullins ▾

Mac Apps

- Certificates**
 - All
 - Pending
 - Development
 - Production
- Identifiers**
 - App IDs
 - Website Push IDs
 - iCloud Containers
- Devices**
 - All
- Provisioning Profiles**
 - All
 - Development
 - Distribution

Add Mac Provisioning Profile

Select Type Configure Generate Download

Your provisioning profile is ready.

Download and Install
Download and double click the following file to install your Provisioning Profile.

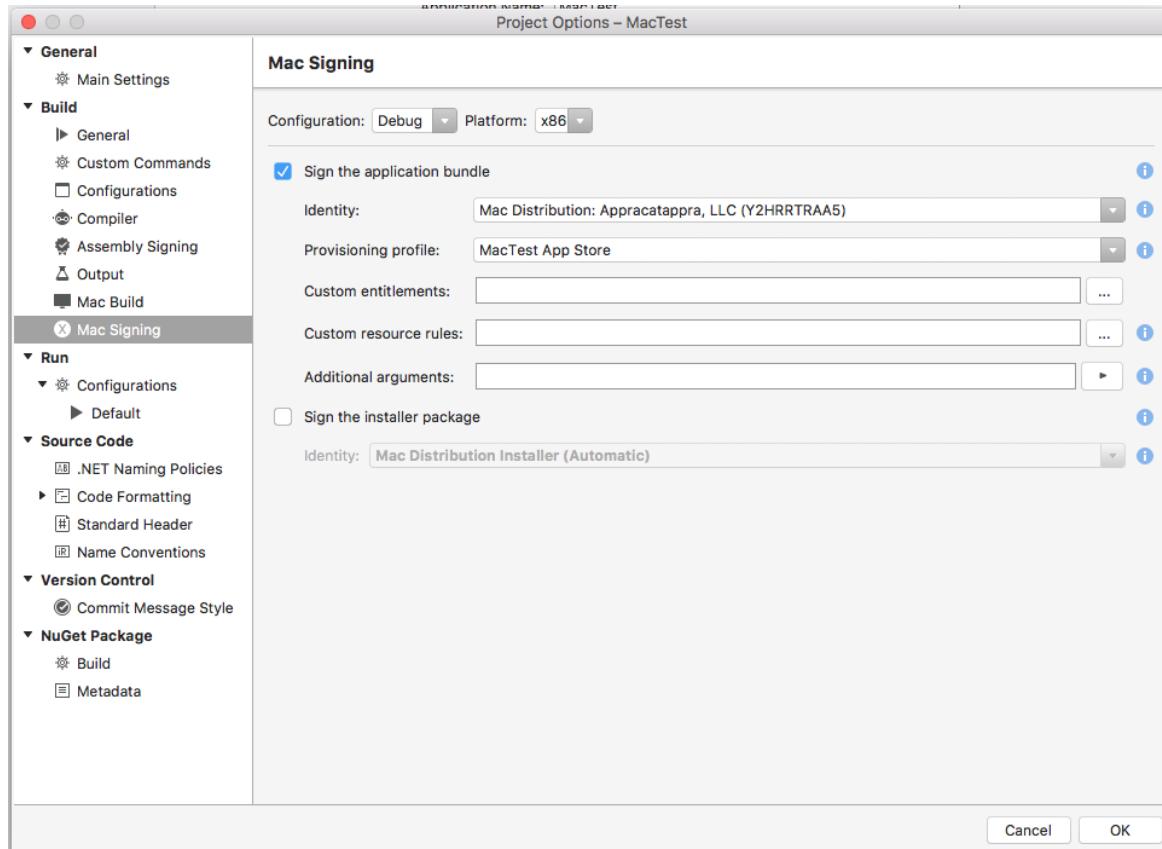
Name: CloudKit App Store
Type: Mac Distribution
App ID: Y2HRRTRAAS.com.appracatappr.a.cloudtestmac
Expires: Nov 21, 2015

Download

Documentation
For more information on using and managing your Provisioning Profile read:
[App Distribution Guide](#)

Add Another Done

6. Drag it into the **Xcode Organizer** or double-click it to install. The profile will then appear in the Xcode Organizer:



7. The provisioning profile will also appear in the list:

Certificates, Identifiers & Profiles

Mac Apps ▾

Certificates

- All
- Pending
- Development
- Production

Identifiers

- App IDs
- Website Push IDs
- iCloud Containers

Mac Provisioning Profiles



3 profiles total.

Name	Type	Status
CloudKit App Store	Mac Distribution	● Active
CloudKit Test	Mac Development	● Active
MacTeam Provisioning Profile: c...	Mac Development	● Active (Managed by Xcode)

If the developer ever changes the features being used by an App ID (eg. enabling iCloud or push notifications) then they should re-create the provision profiles for that App ID.

Related links

- [Installation](#)
- [Hello, Mac sample](#)
- [Distribute your apps on the Mac App Store](#)
- [Tools Guide : Code Signing Your App](#)
- [Developer ID and GateKeeper](#)

Mac app configuration

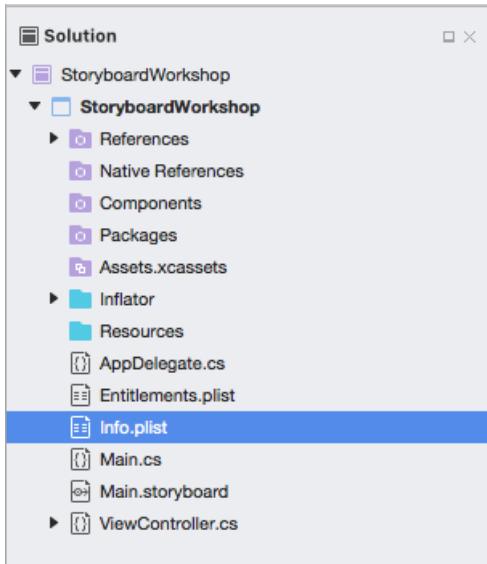
1/23/2020 • 2 minutes to read • [Edit Online](#)

Mac app configuration

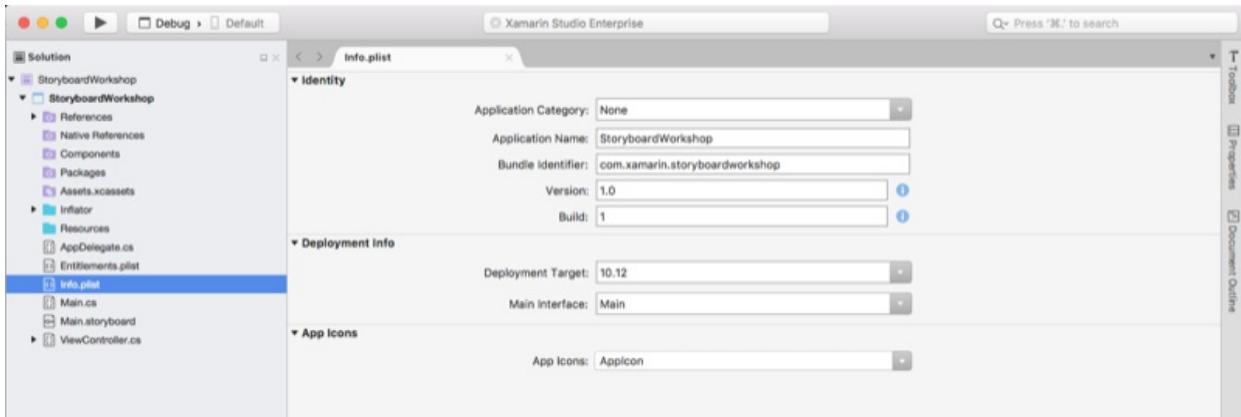
Right-click on the Mac application project in Visual Studio for Mac and choose **Options**.

Application settings

To change a Xamarin.Mac app's Application Settings, double-click the **Info.plist** file in the **Solution Pad**:



This will display the available options for the app:

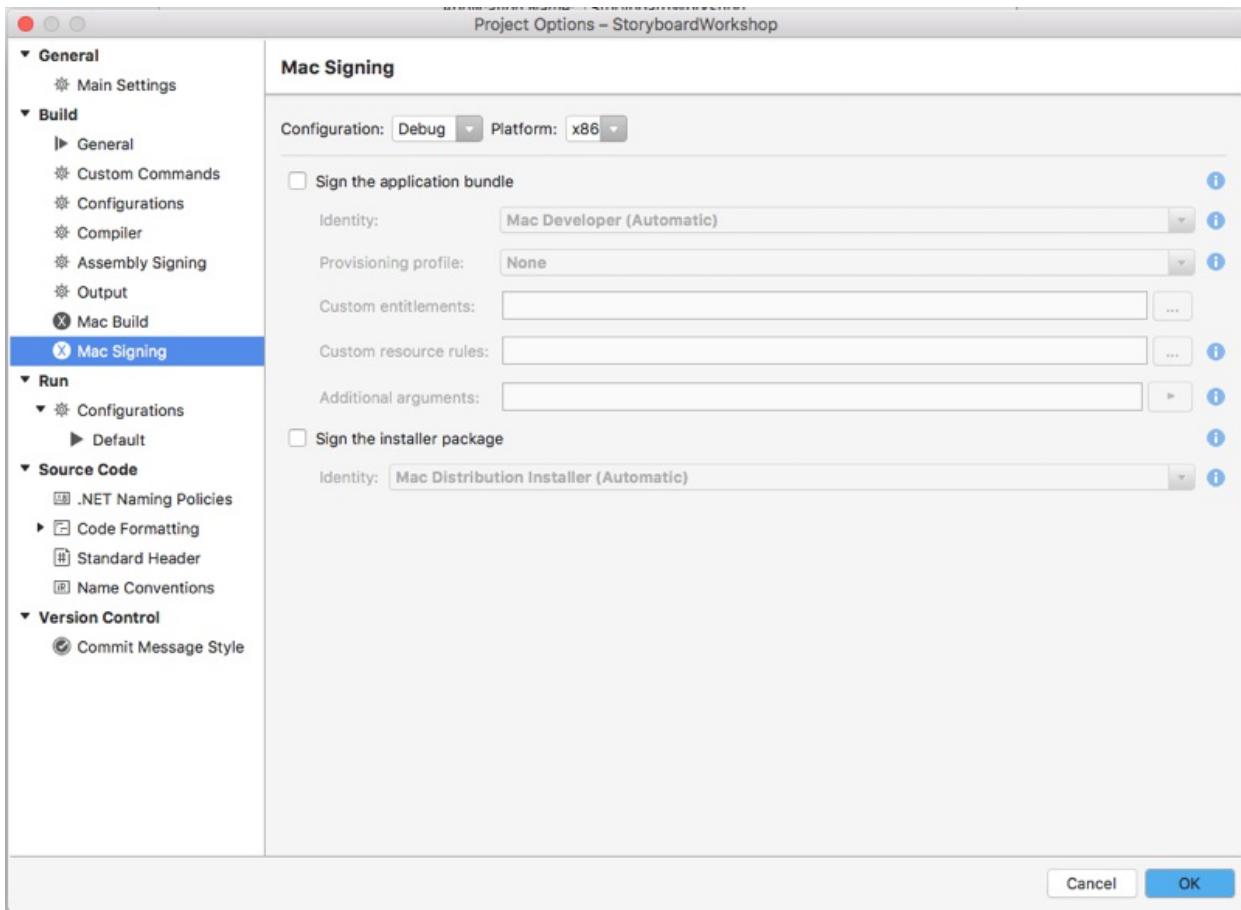


Running Mac applications created with Xamarin.Mac have the following system requirements:

- A Mac computer running Mac OS X 10.7 or greater.

Signing settings

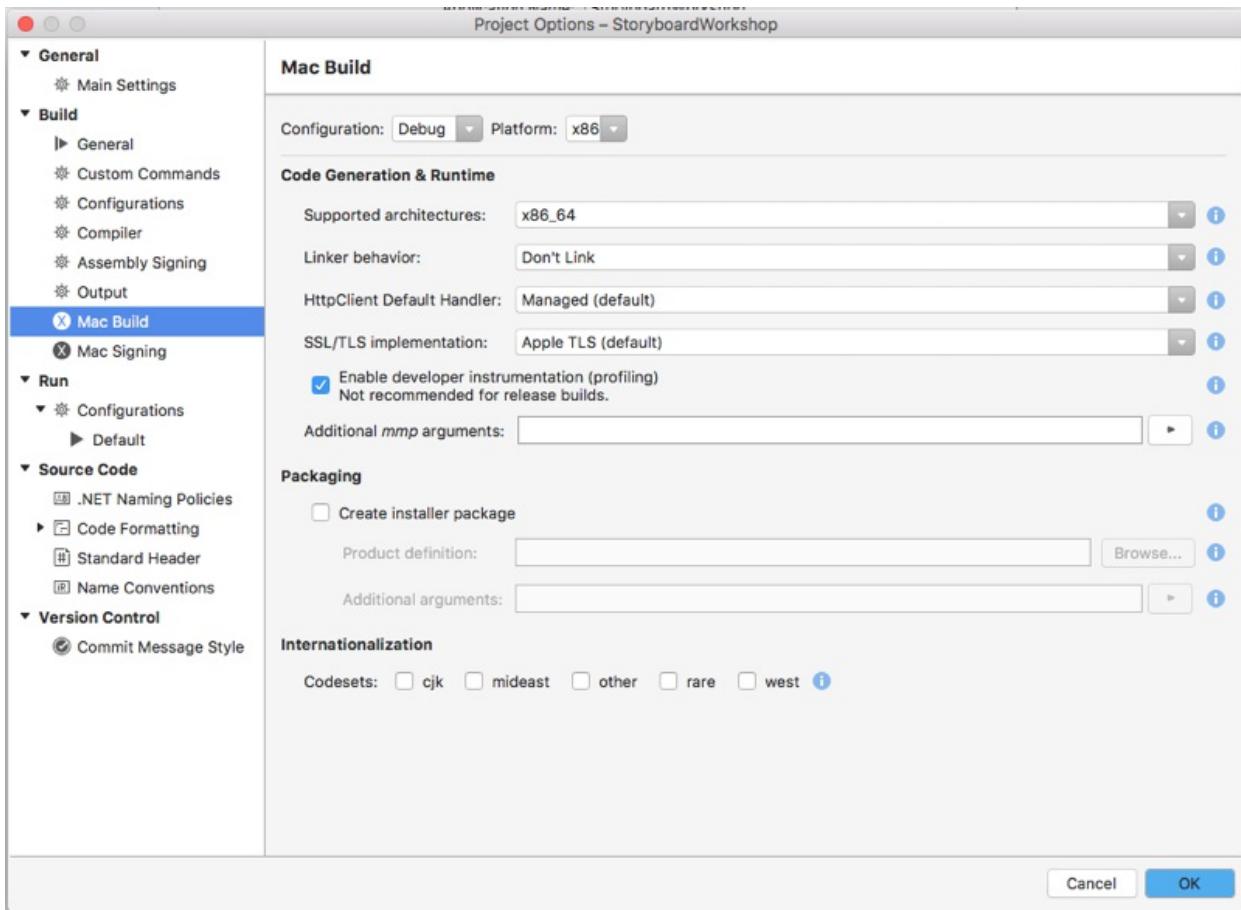
The **Mac Signing** section of the **Project Options** dialog allows the developer to sign a Xamarin.Mac app for testing, for self release or for release through the Apple App Store:



From here select the Identity, Provisioning Profile and any custom entitlements used to sign the app when it is compiled. The developer can optionally sign the Installer used to install the app on an other Mac.

Build settings

The **Mac Build** section of the **Project Options** dialog allows the developer to select the architecture for a `Xamarin.Mac` app, to control what version of macOS the app will support and to optionally create an install package when the app is successfully compiled:



Related links

- [Installation](#)
- [Hello, Mac sample](#)
- [Distribute your apps on the Mac App Store](#)
- [Developer ID and GateKeeper](#)

Signing Xamarin.Mac Apps with a Developer ID

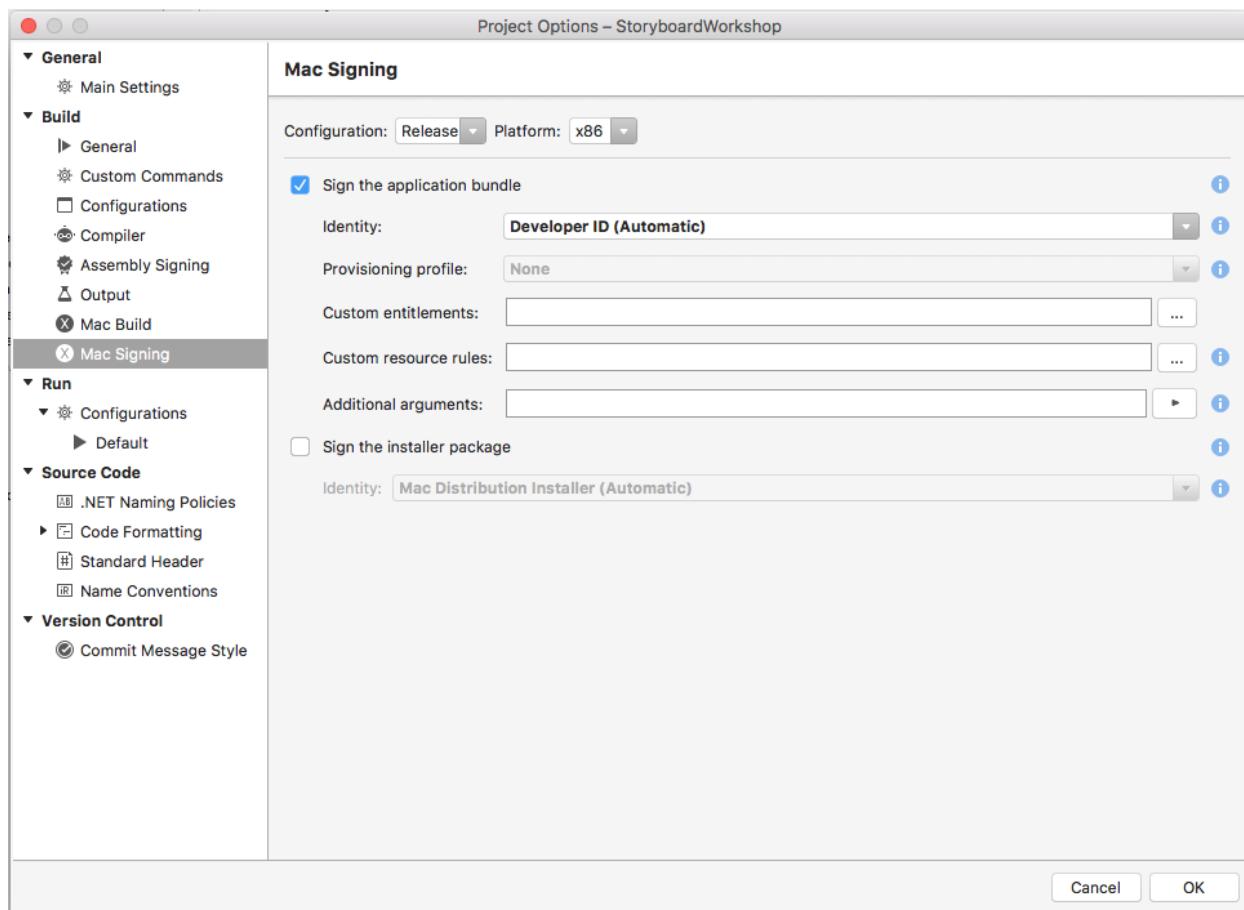
3/5/2021 • 2 minutes to read • [Edit Online](#)

If the developer plans to distribute an app directly to macOS users, Apple recommends that they code-sign it with their Developer ID so that it can be installed on macOS systems with **GateKeeper** enabled. If the app has not been signed, **GateKeeper** will prevent users from installing with an alert message (they can bypass this restricting by holding down the Control key while launching).

Read more about [Developer ID and GateKeeper](#) and [Distributing Outside the Mac App Store](#) on Apple's website.

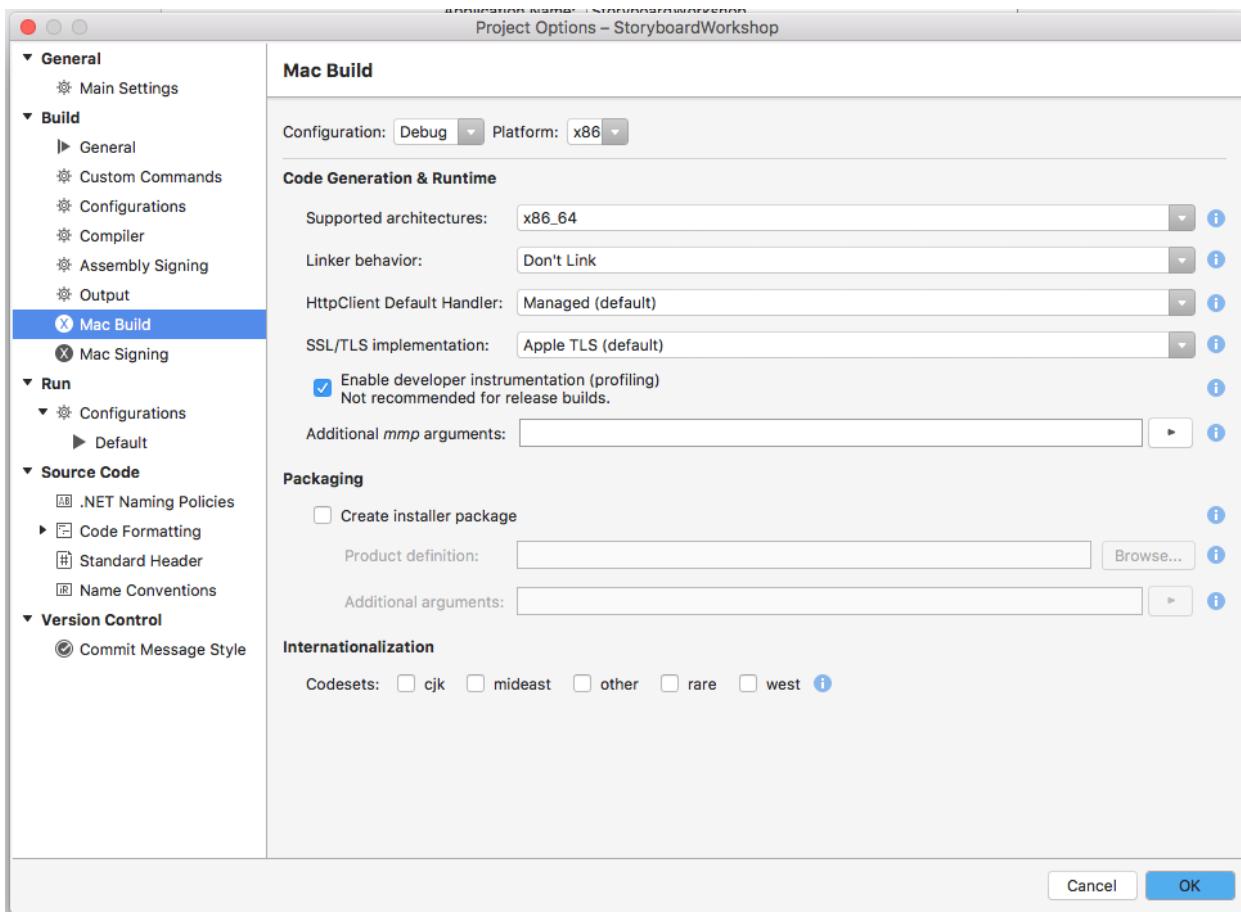
Code signing options

To build an app for deployment directly to users (NOT via the Mac App Store) set the **Signing Settings** to use the **Developer ID**. Ensure to edit the **Release** configuration.



Build

Before building, ensure to selected the correct configuration and select to create an install package in the **Mac Build** settings:

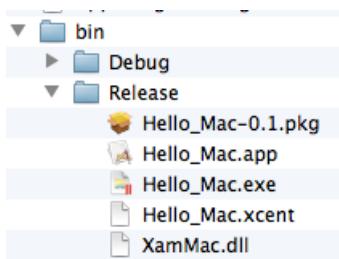


When building the app, the developer will be prompted to use both certificates:



After the application has been built, the developer can right-click on the project and choose **Open Containing Folder** to find the package file (in the `bin/Release` directory). This package file includes an installer for the

application, so it can be distributed to any macOS user for installation.



Related links

- [Installation](#)
- [Hello, Mac sample](#)
- [Distribute your apps on the Mac App Store](#)
- [Tools Guide : Code Signing Your App](#)
- [Developer ID and GateKeeper](#)

Bundling for the Mac App Store

1/23/2020 • 2 minutes to read • [Edit Online](#)

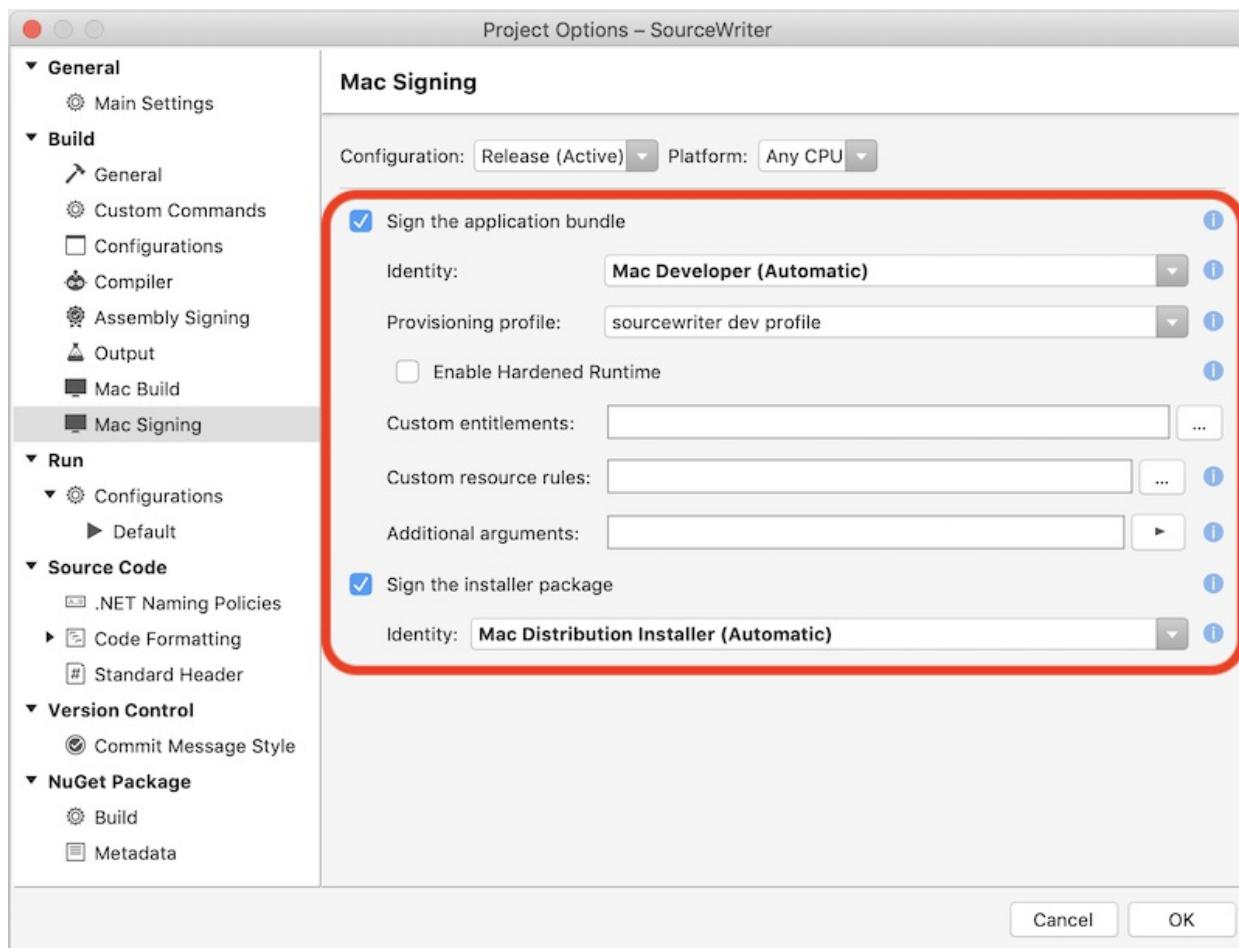
This section describes the basics of building an application for release in the Mac App Store using Visual Studio for Mac. Based on additional features (such as iCloud access and push notifications), further setup may be required that goes beyond the scope of this article.

NOTE

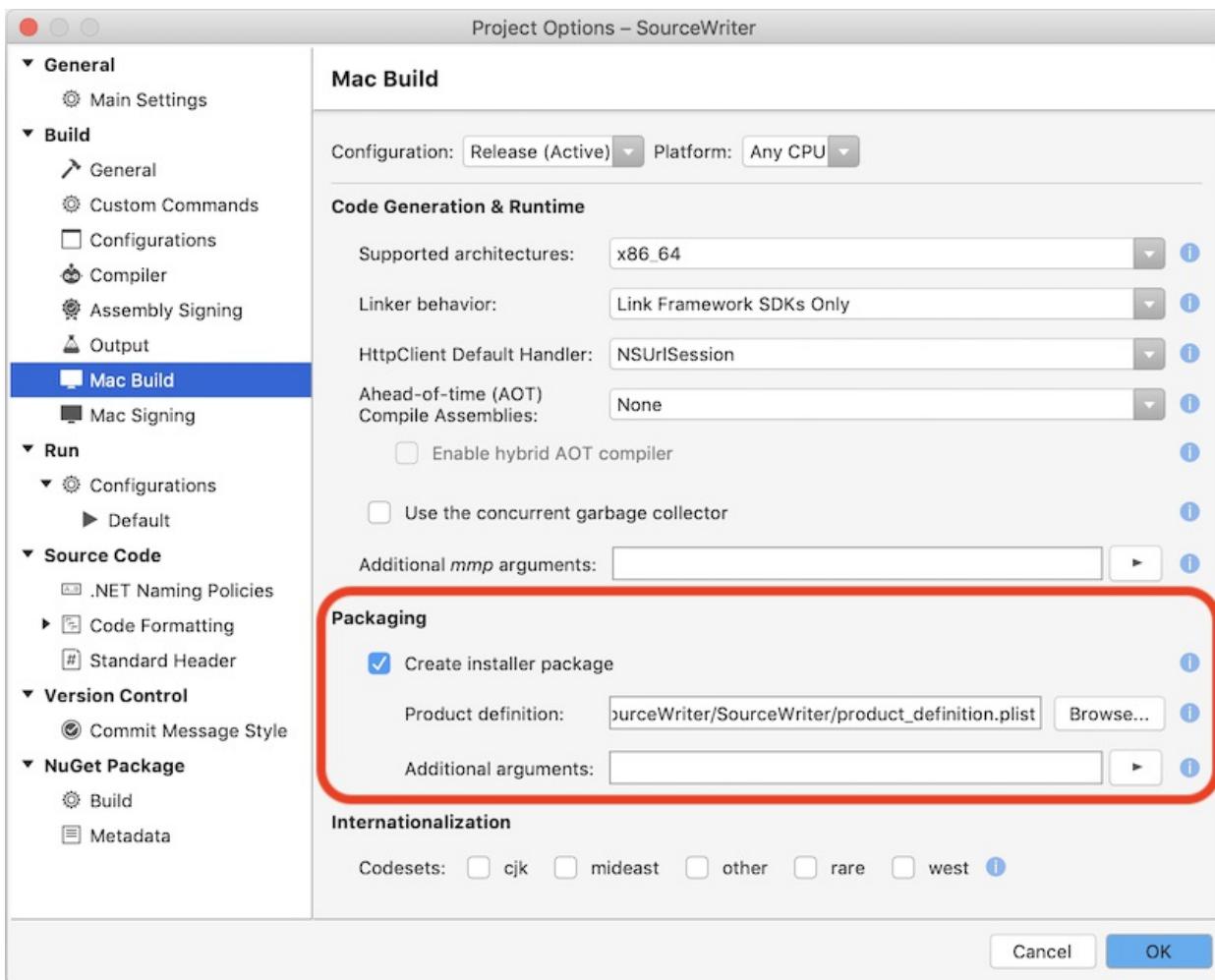
Before starting this section, the developer must have created a production provisioning profile to build for the Mac App Store. See the [profile instructions](#) for creating the required provisioning profiles.

Code signing options

Change the **Configuration** to **Release** before updating the code signing and packaging options. The developer needs to make sure that they use their company **Identity** and the provisioning profile that we created above when signing the application for release in the App Store.

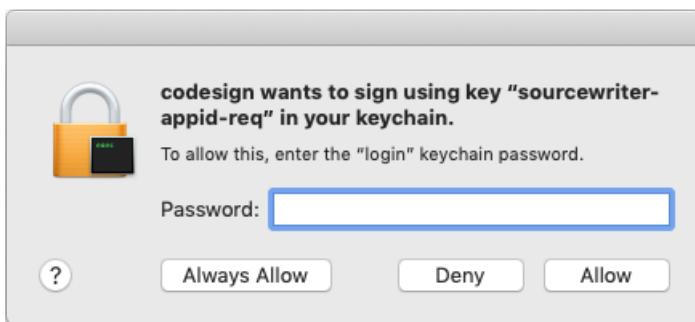


Ensure that the option to create an installer package has been checked in the **Mac Build** settings:

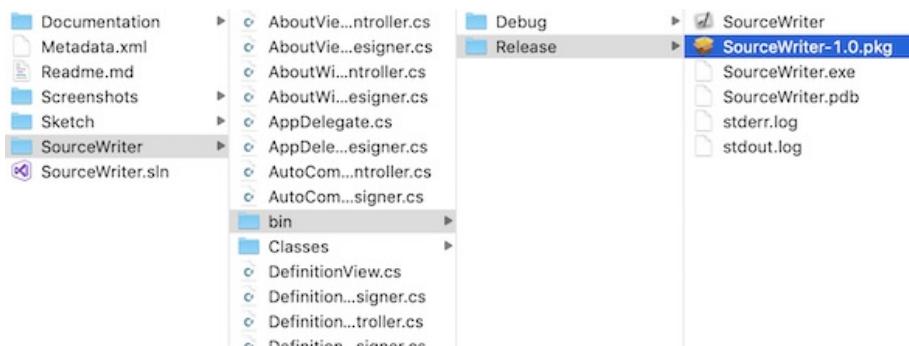


Build

Before building, ensure that the **Release** configuration has been selected. When the developer builds the app, they'll be prompted *twice* (to use both the application and installer certificates):



After the application has been built, the developer can right-click on the project and choose **Reveal in Finder** to find the package file (in the `bin/Release/AppStore` directory in the example shown below). This package file includes an installer for the app that can be submitted to Apple for inclusion in the Mac App Store.



SourceWriter-1.0.pkg
Installer package - 6.8 MB

Related links

- [Installation](#)
- [Hello, Mac sample](#)
- [Distribute your apps on the Mac App Store](#)
- [Developer ID and GateKeeper](#)

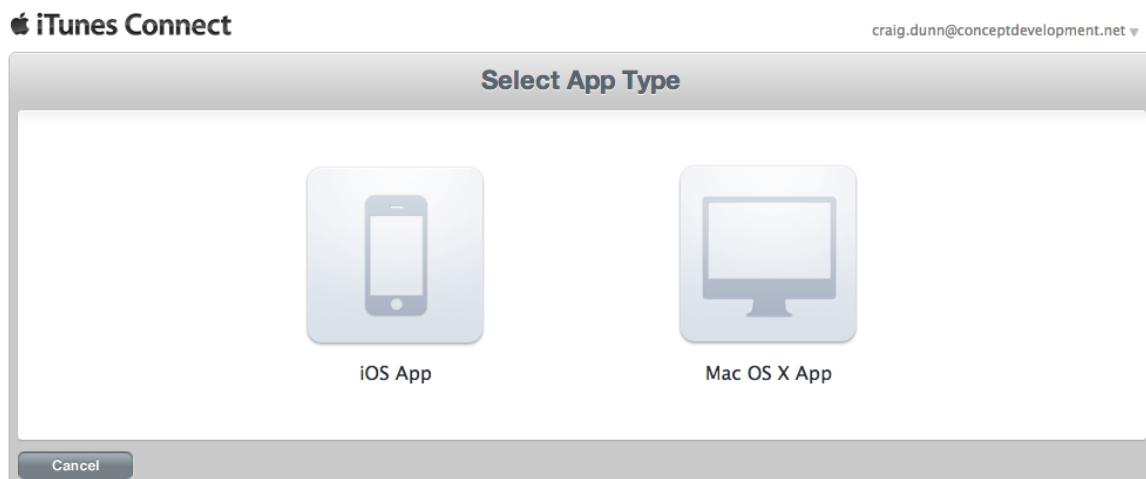
Upload to Mac App Store

11/2/2020 • 2 minutes to read • [Edit Online](#)

This guide walks through uploading a *Xamarin.Mac* app for publication to the Mac App Store.

Applications are submitted for Mac App Store approval via [iTunes Connect](#). You will also need the [Transporter](#) tool from the App Store.

1. Choose a macOS App to create:



2. Enter the application's name and other details. The developer can only choose from an existing **Bundle ID** that has been created previously:

A screenshot of the iTunes Connect 'App Information' dialog. At the top, it says 'iTunes Connect' and shows the email 'craig.dunn@conceptdevelopment.net'. Below that is the title 'App Information'. It says 'Enter the following information about your app.' There are four input fields: 'Default Language' (set to English), 'App Name' (set to Hello Mac), 'SKU Number' (set to HELOMAC), and 'Bundle ID' (set to Hello Mac Test App - com.xamarin.hellomac1). Below the bundle ID field is a note: 'You can register a new Bundle ID [here](#)'. A yellow warning box contains the text: '⚠ Note that the Bundle ID cannot be changed if the first version of your app has been approved or if you have enabled Game Center or the iAd Network.' At the bottom left is a 'Cancel' button and at the bottom right is a 'Continue' button.

3. Select the availability date and price. Regardless of the availability date the developer selects, the app will only become available for sale after it has been approved. This value can be set far in the future if the developer wants more control over the actual availability date:

Hello Mac

Select the availability date and price tier for your app.

Availability Date	<input type="text" value="12/Dec"/> <input type="button" value="11"/> <input type="button" value="2012"/>	?
Price Tier	<input type="text" value="Select"/>	?
View Pricing Matrix ▶		

Unless you select [specific stores](#), your app will be for sale in all App Stores worldwide. **

**Paid Apps will only be available for sale in territories covered by your current Mac OS X Paid Applications Contract.

[Go Back](#)
[Continue ▶](#)

4. Enter the app's information, including the App Store category it belongs in:

Hello Mac

Enter the following information in English.

Version Information

Version Number	<input type="text" value="1"/>	?
Copyright	<input type="text" value="Monkey"/>	?
Primary Category	<input type="text" value="Developer Tools"/>	?
Secondary Category (Optional)	<input type="text" value="Utilities"/>	?

Select the ratings that apply:

Rating

For each content description, choose the level of frequency that best describes your app.

[App Rating Details ▶](#)

Apps must not contain any obscene, pornographic, offensive or defamatory content or materials of any kind (text, graphics, images, photographs, etc.), or other content or materials that in Apple's reasonable judgment may be found objectionable.

Apple Content Descriptions

None Infrequent/Mild Frequent/Intense

Cartoon or Fantasy Violence

Realistic Violence

Sexual Content or Nudity

Profanity or Crude Humor

Alcohol, Tobacco, or Drug Use or References

Mature/Suggestive Themes

Simulated Gambling

Horror/Fear Themes

Prolonged Graphic or Sadistic Realistic Violence

Graphic Sexual Content and Nudity



App Rating

Description, keywords and contact URLs:

Metadata

Description	A welcome to Xamarin on the Mac	(?)
Keywords	xamarin, c#, mac	(?)
Support URL	http://xamarin.com	(?)
Marketing URL (Optional)	http://	(?)
Privacy Policy URL (Optional)	http://	(?)

Contact information and advice for the App Store reviewers:

App Review Information

Contact Information (?)

First Name	<input type="text"/>
Last Name	<input type="text"/>
Email Address	<input type="text"/>
Phone Number	<input type="text"/> Include your country code

Review Notes (Optional) (?)

Demo Account Information (Optional) (?)

Username	<input type="text"/>
Password	<input type="password"/>

App Sandbox Entitlement Usage Information (Optional)

Add Entitlement

Entitlement Key

Usage Information

To get started, provide your contact information above and click Add Entitlement. You must provide information for every temporary exception entitlement specified in your binary.

And finally, screenshots:

Uploads

Mac OS X App Screenshots (?)



Choose File

Go Back

Save

Screenshots should be in JPG, TIF or PNG format, 1280x800, 1440x900, 2880x1800 or 2560x1600 pixels in size. Press **Save** to finish.

5. The app information is shown for review. Click **View Details** to change the status:

Hello Mac

App Information [Edit](#)

Identifiers	Links	Rights and Pricing
SKU HELOMAC	View in App Store	Manage Game Center
Bundle ID com.xamarin.hellomac1		Newsstand Status
Apple ID 586830009		Delete Application
Type Mac OS X App		
Default Language English		

Versions

Current Version

	Version 1
	Status Prepare for Upload
	Date Created Dec 11, 2012
View Details	

[Done](#)

6. In the details view, click Ready to Upload Binary to submit The application package file:

Hello Mac (1)

[App Summary](#) [Ready to Upload Binary](#)

Version Information [Edit](#)

	Hello Mac	Links
	Version 1	Version Summary
	Copyright Monkey	Status History
	Primary Category Developer Tools	
	Secondary Category (Optional) Utilities	
	Rating 4+	
	Status Prepare for Upload	

Metadata and Uploads [Edit](#)

English (Default Language)	Choose Another Language: English (Default) ▾
Take advantage of the App Store's global audience by translating your app's metadata for the App Store for each of the countries in which you offer apps. Customers are more likely to read about your app if it's in their native language. View a list of third party vendors who can provide internationalization and localization services for your app.	
App Name Hello Mac Description A welcome to Xamarin on the Mac Keywords xamarin, c#, mac Support URL http://xamarin.com	

7. Answer the cryptography question:

Hello Mac (1) - Export Compliance

Export laws require that products containing encryption be properly authorized for export.
Failure to comply could result in severe penalties.
For further information, [click here](#).

Is your product designed to use cryptography or does it contain or
incorporate cryptography?

Yes No

[Cancel](#)

[Save](#)

8. The site will advise when it is ready to accept the application package file:

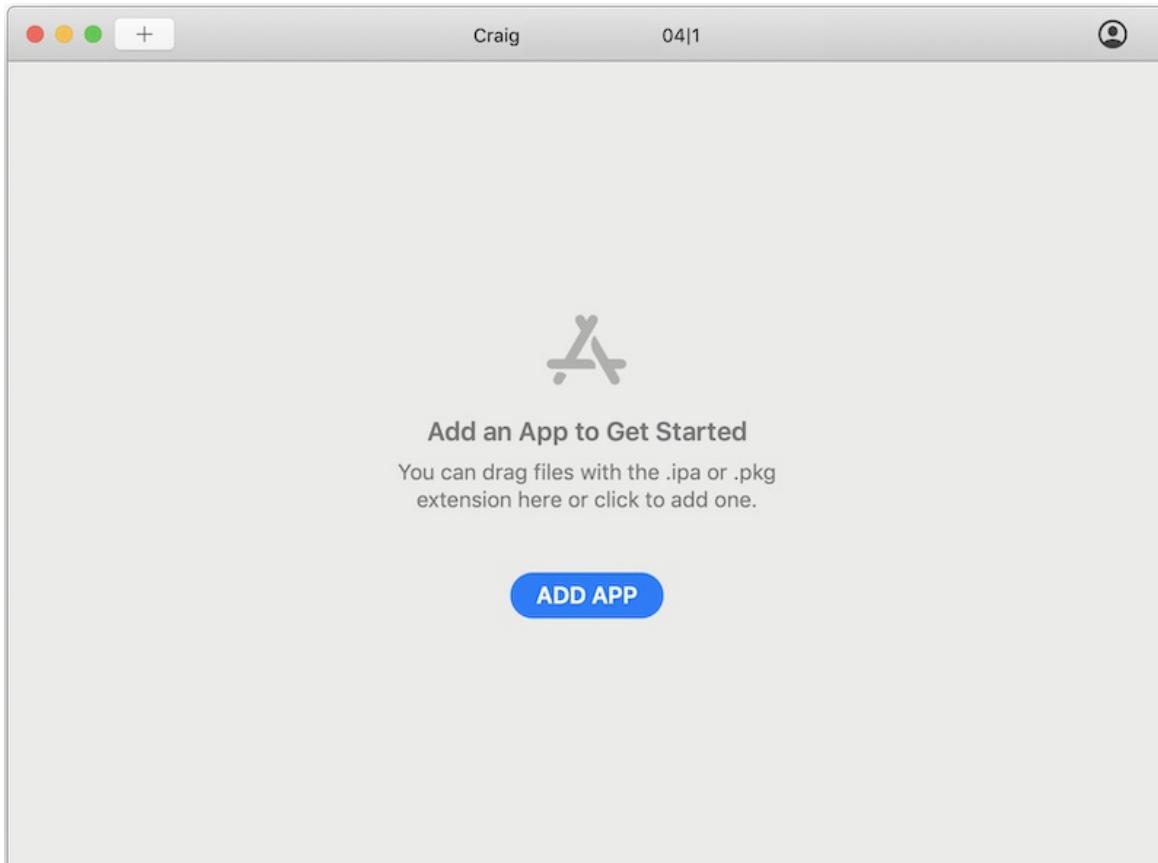
Hello Mac (1)

You are now ready to upload your binary using Application Loader. Application Loader can only be used when your app status is Waiting for Upload. Once the binary is uploaded, your app status will change first to Upload Received and then to Waiting for Review. If we encounter any issues with the binary itself, your app status will change to Invalid Binary and you will receive an email explaining the issues and the steps you can take to correct them.

Mac App Store submissions require Application Loader 1.4 or later. If you have downloaded Xcode 3.2.5 or later, you should already have Application Loader stored here: /Developer/Applications/Utilities/Application Loader.app (or in your equivalent custom install location). If you do not find it, download and install the [latest version of Application Loader](#).

[Continue](#)

9. Start **Transporter** and login with your Apple ID, then choose **ADD APP**:



Follow the instructions to upload your app package to iTunes Connect.

NOTE

Transporter replaces the **Application Loader** tool that was used with Xcode 10 and earlier. Application Loader is no longer available in Xcode 11 or newer.

When the application has been approved, it will be available for download or purchase from the Mac App Store.

Related links

- [Installation](#)
- [Hello, Mac sample](#)
- [Distribute your apps on the Mac App Store](#)
- [Tools Guide : Code Signing Your App](#)
- [Developer ID and GateKeeper](#)

Apple Platform (iOS and Mac)

10/28/2019 • 2 minutes to read • [Edit Online](#)

Code Sharing

For elements of your code that have no user interface elements the best way to share code between iOS and Mac is still the use of [Portable Class Libraries](#).

For code that has to do some user interface work and yet, you want to share, you should use [Shared Projects](#) which allow you to place code to share in a single project and have it compiled with both Mac and iOS when referenced.

Unified API

The Unified API for iOS and Mac projects uses the same namespaces for frameworks so that the same code file can be used across both platforms, for seamless code-sharing. It also enables both 32 and 64 bit builds. The Unified API has been the template default since early 2015, and is recommended for all new projects - *only* Unified API projects can be submitted to the App Store.

Classic APIs

NOTE

Classic Profile Deprecation: As new platforms are added in Xamarin.iOS we are starting to gradually deprecate features from the classic profile (monotouch.dll). For example, the non-NRC (new-ref-count) option was removed. NRC has always been enabled for all unified applications (i.e. non-NRC was never an option) and has no known issues. Future releases will remove the option of using Boehm as the garbage collector. This was also an option never available to unified applications. The complete removal of classic support is scheduled for fall 2016 with the release of Xamarin.iOS 10.0.

The original (non-Unified) Xamarin.iOS and Xamarin.Mac APIs made code-sharing more difficult because native frameworks had either `MonoTouch.` or `MonoMac.` namespace prefixes. We provided some empty namespaces that allows developers to share code by adding `using` statements that reference both MonoMac and MonoTouch namespaces on the same file, but this was a little ugly. The Classic API should only continue to be used in legacy apps that are internally distributed (upgrading to the Unified API is recommended).

Updating from Classic to the Unified API

There are detailed instructions for updating any application from the Classic to the Unified API.

Binding Objective-C Libraries

Xamarin lets you bring native libraries into your apps with bindings. This section explains:

- how bindings work,
- how to manually build a binding project that lets you bring Objective-C code into Xamarin, and
- how to use our **Objective Sharpie** tool to help automate the process.

Native References

Mac/iOS Native Types

To support 32 and 64 bit code transparently from C# and F#, we are introducing new data types. Learn about

them here.

Building 32 and 64 bit apps

What you need to know to support 32 and 64 bit applications.

Working with Native Types in Cross-Platform Apps

This article covers using the new iOS Unified API Native types (`nint`, `nuint`, `nfloat`) in a cross-platform application where code is shared with non-iOS devices such as Android or Windows Phone OSes. It provides insight into when the Native types should be used and provides several possible solutions to cases where the new type must be used with cross-platform code.

HttpClient Stack and SSL/TLS Implementation Selector

The new HttpClient Stack Selector controls which HttpClient implementation to use in your Xamarin.iOS, Xamarin.tvOS and Xamarin.Mac app. You can now switch to an implementation that uses iOS's, tvOS's or OS X's native transports (`NSURLSession` or `CFNetwork` depending on the OS).

SSL (Secure Socket Layer) and its successor, TLS (Transport Layer Security), provide support for HTTP and other network connections via `System.Net.Security.SslStream`. The new SSL/TLS implementation build option switches between Mono's own TLS stack, and one powered by Apple's TLS stack present in Mac and iOS.

Apple Account Management

11/2/2020 • 3 minutes to read • [Edit Online](#)

The Apple account management interface in Visual Studio provides a way to view information for development teams associated with an Apple ID. It allows you to do the following:

- Add Apple developer accounts
- View signing certificates and provisioning profiles
- Create new signing certificates
- Download existing provisioning profiles

IMPORTANT

Xamarin's tools for Apple account management only display information about paid Apple developer accounts. To learn how to test an app on a device without a paid Apple developer account, please see the [Free provisioning for Xamarin.iOS apps](#) guide.

Requirements

Apple account management is available on Visual Studio for Mac, Visual Studio 2019, and Visual Studio 2017 (Version 15.7 and higher). You must also have a paid Apple Developer account to use this feature. More information on Apple developer accounts is available in the [Device Provisioning](#) guide.

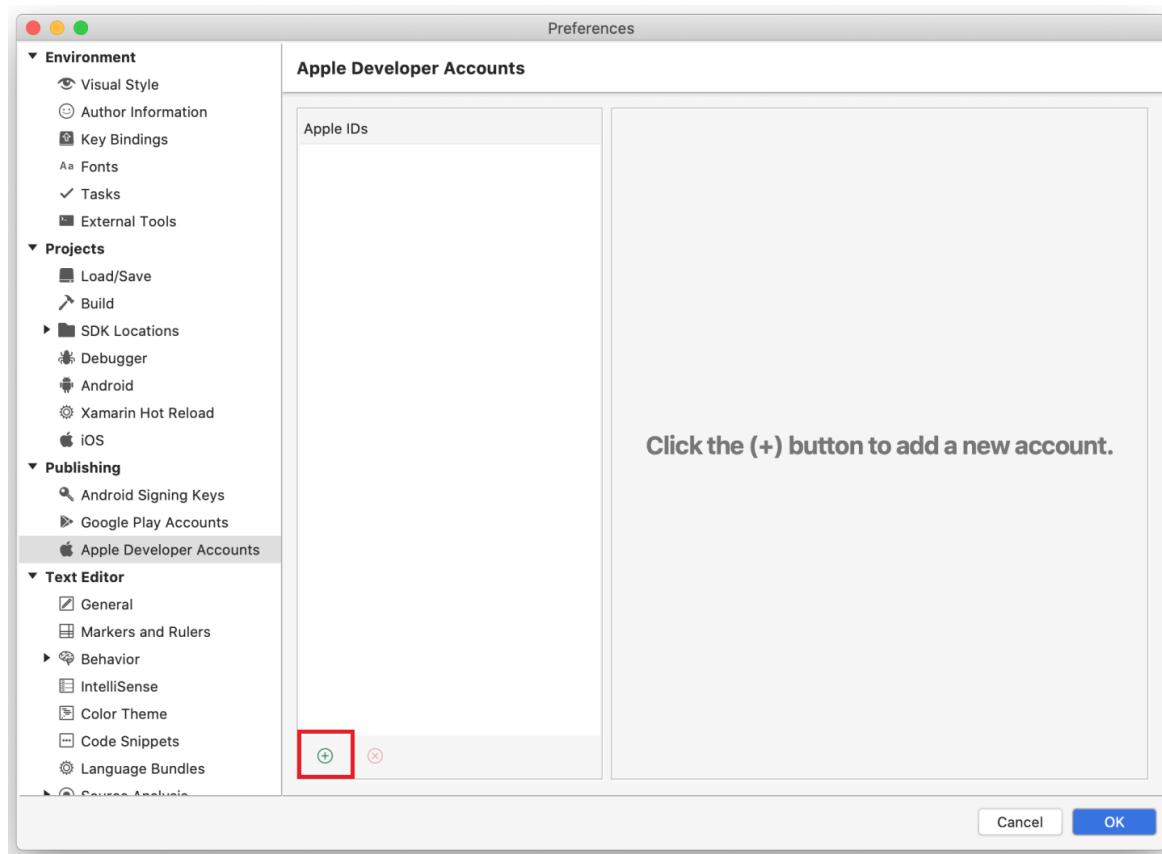
NOTE

Before you begin, be sure to first accept any user license agreements in the [Apple Developer portal](#).

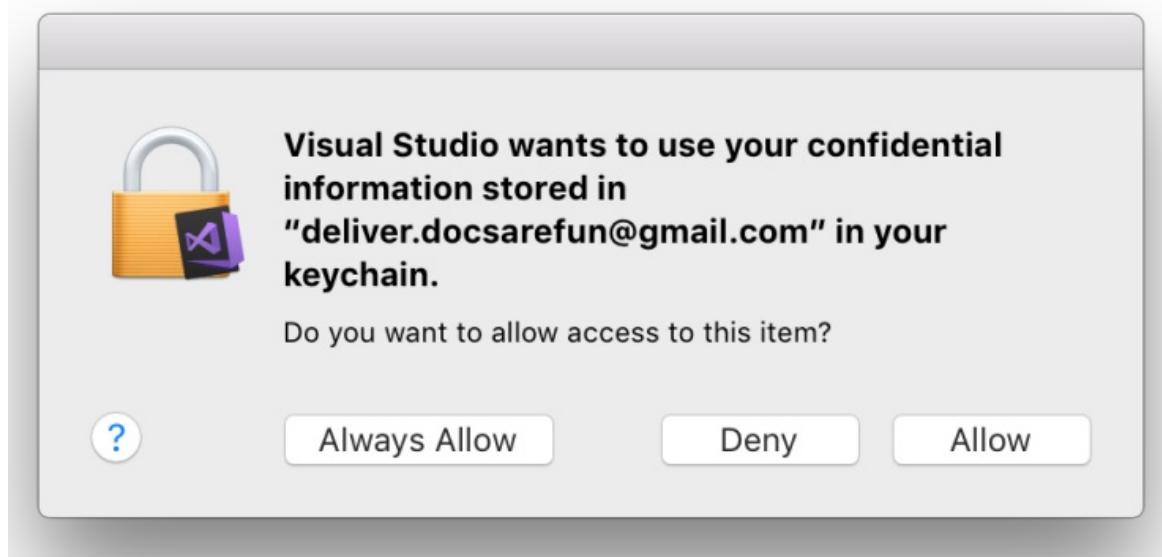
Add an Apple developer account

- [Visual Studio for Mac](#)
- [Visual Studio](#)

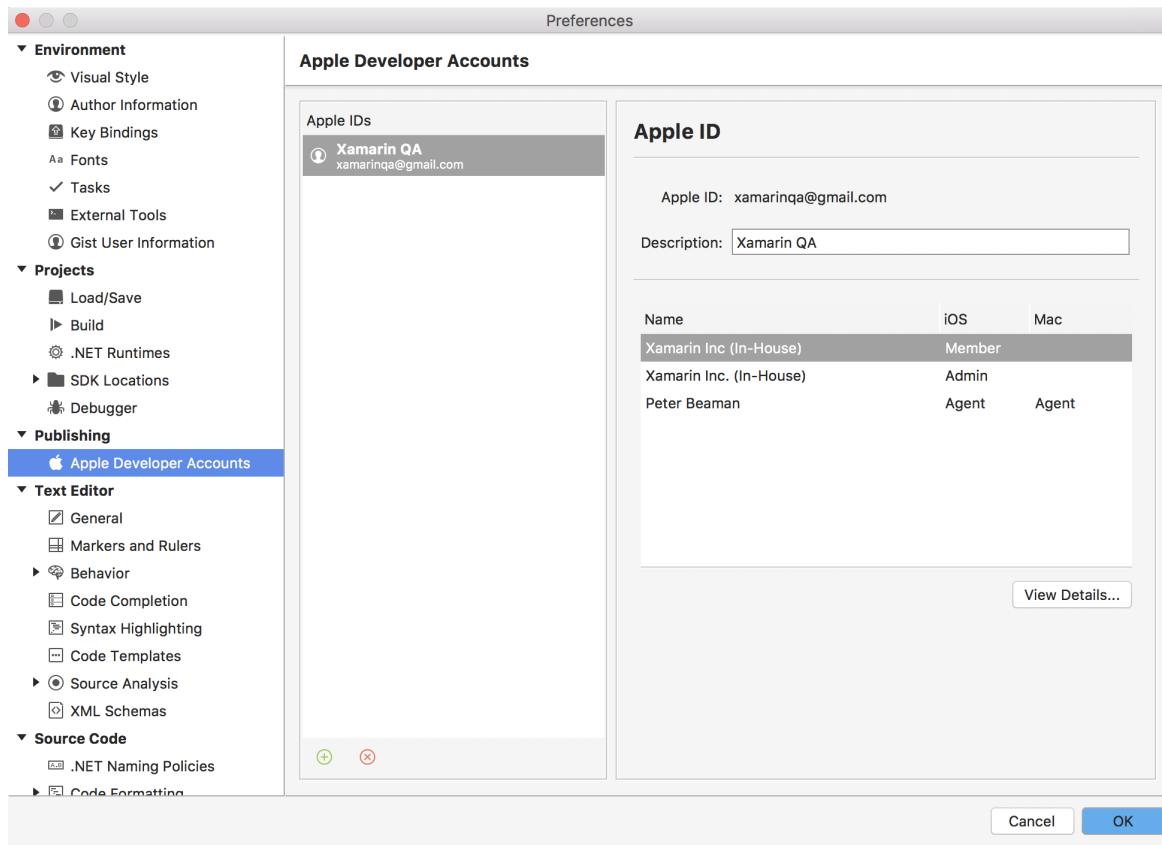
1. Go to **Visual Studio > Preferences > Apple Developer Account** and click the + button to open the sign in dialog:



2. Enter your Apple ID and password then click **Sign In**. This will save your credentials in the secure Keychain on this machine.
3. Select **Always Allow** on the alert dialog to allow Visual Studio to use your credentials:



4. Once your account has been added successfully, you will see your Apple ID and any teams that your Apple ID is part of:



View signing certificates and provisioning profiles

Select a team and click on **View Details...** to open a dialog that displays a list of signing identities and provisioning profiles that are installed on your machine.

The team details dialog displays a list of Signing Identities, organized by type. The **Status** column advises you if the certificate is:

- **Valid** – The signing identity (both the certificate and the private key) is installed on your machine and it has not expired.
- **Not in Keychain** – There is a valid signing identity on Apple's server. To install this on your machine, it must be exported from another machine. You cannot download the signing identity from the Apple Developer Portal as it will not contain the private key.
- **Private key is missing** – A Certificate with no private key is installed in the keychain.
- **Expired** – The Certificate is expired. You should remove this from your keychain.

Certificates	Type	Status
Amy Burns (amyb)	iOS Development	Valid

Create a signing certificate

To create a new signing identity, click **Create Certificate** to open the drop-down menu and select the [certificate type](#) that you want to create. If you have the correct permissions a new signing identity will appear after a few seconds.

If an option in the drop-down is greyed out and unselected, it means that you do not have the correct team permissions to create this type of certificate.

Download provisioning profiles

The team details dialog also displays a list of all provisioning profiles connected to your developer account. You can download all provisioning profiles to your local machine by clicking **Download all Profiles**.

Troubleshoot

- It may take several hours for a new Apple developer account to be approved. You will not be able to enable automatic provisioning until the account has been approved.
- If adding an Apple developer accounts fails with the message
`Authentication Error: Xcode 7.3 or later is required to continue developing with your Apple ID.`, make sure that the Apple ID you are using has an active paid membership to the Apple Developer Program. To use a paid Apple developer account, please see the [Free provisioning for Xamarin.iOS apps](#) guide.
- If attempting to create a new signing certificate fails with the error
`You have reached the limit for certificates of this type`, then the maximum number of certificates allowed have been generated. To fix this, browse to the [Apple Developer Center](#) and revoke one of the Production Certificates.
- If you are experiencing issues logging in your account on Visual Studio for Mac, a possible fix is to open the Keychain application and under **Category** select **Passwords**. Search for `deliver` and delete all entries that are found.
- If your signing certificate is revoked, it can be removed from the following path on Windows:
`C:\Users\<user>\AppData\Local\Xamarin\iOS\Provisioning\Certificates`.

Known Issues

- Distribution provisioning profiles by default will target App Store. In House or Ad Hoc profiles should be created manually.

Unified API for Xamarin.iOS and Xamarin.Mac

10/28/2019 • 2 minutes to read • [Edit Online](#)

Xamarin's Unified API makes it possible to share code between Mac and iOS and support 32 and 64-bit applications with the same binary. The Unified API is used by default in new Xamarin.iOS and Xamarin.Mac projects.

Overview

A description of Xamarin's Unified API, its features, and the changes that come along with its use

Update Existing Apps

General information about updating existing apps from the Classic API to the Unified API

Updating Existing iOS Apps

Information about updating existing iOS apps from the Classic API to the Unified API

Updating Existing Mac Apps

Information about updating existing Mac apps from the Classic API to the Unified API

Update Existing Xamarin.Forms Apps

Information about updating existing Xamarin.Forms apps from the Classic API to the Unified API

Migrating a Binding to the Unified API

A description of how to migrate a Xamarin.iOS or Xamarin.Mac binding project from the Classic API to the Unified API

Tips for Updating Code to the Unified API

Various tips that can be helpful when updating to the Unified API

Unified API Overview

11/2/2020 • 10 minutes to read • [Edit Online](#)

Xamarin's Unified API makes it possible to share code between Mac and iOS and support 32 and 64-bit applications with the same binary. The Unified API is used by default in new Xamarin.iOS and Xamarin.Mac projects.

IMPORTANT

The Xamarin Classic API, which preceded the Unified API, has been deprecated.

- The last version of Xamarin.iOS to support the Classic API (`monotouch.dll`) was Xamarin.iOS 9.10.
- Xamarin.Mac still supports the Classic API, but it is no longer updated. Since it is deprecated, developers should move their applications to the Unified API.

Updating Classic API-based Apps

Follow the relevant instructions for your platform:

- [Update Existing Apps](#)
- [Update Existing iOS Apps](#)
- [Update Existing Mac Apps](#)
- [Update Existing Xamarin.Forms Apps](#)
- [Migrating a Binding to the Unified API](#)

Tips for Updating Code to the Unified API

Regardless of what applications you are migrating, check out [these tips](#) to help you successfully update to the Unified API.

Library Split

From this point on, our APIs will be surfaced in two ways:

- **Classic API:** Limited to 32-bits (only) and exposed in the `monotouch.dll` and `XamMac.dll` assemblies.
- **Unified API:** Support both 32 and 64 bit development with a single API available in the `Xamarin.iOS.dll` and `Xamarin.Mac.dll` assemblies.

This means that for Enterprise developers (not targeting the App Store), you can continue using the existing Classic APIs, as we will keep maintaining them forever, or you can upgrade to the new APIs.

Namespace Changes

To reduce the friction to share code between our Mac and iOS products, we are changing the namespaces for the APIs in the products.

We are dropping the prefix "MonoTouch" from our iOS product and "MonoMac" from our Mac product on the data types.

This makes it simpler to share code between the Mac and iOS platforms without resorting to conditional

compilation and will reduce the noise at the top of your source code files.

- **Classic API:** Namespaces use `MonoTouch.` or `MonoMac.` prefix.
- **Unified API:** No namespace prefix

Runtime Defaults

The Unified API by default uses the **SGen** garbage collector and the [New Reference Counting](#) system for tracking object ownership. This same feature has been ported to Xamarin.Mac.

This solves a number of problems that developers faced with the old system and also ease [memory management](#).

Note that it is possible to enable New Refcount even for the Classic API, but the default is conservative and does not require users to make any changes. With the Unified API, we took the opportunity of changing the default and give developers all the improvements at the same time that they refactor and re-test their code.

API Changes

The Unified API removes deprecated methods and there are a few instances where there were typos in the API names when they were bound to the original MonoTouch and MonoMac namespaces in the Classic APIs. These instances have been corrected in the new Unified APIs and will need to be updated in your component, iOS and Mac applications. Here is a list of the most common ones you might run into:

CLASSIC API METHOD NAME	UNIFIED API METHOD NAME
<code>UINavigationController.PushViewControllerAnimated()</code>	<code>UINavigationController.PushViewController()</code>
<code>UINavigationController.PopViewControllerAnimated()</code>	<code>UINavigationController.PopViewController()</code>
<code>CGContext.SetRGBFillColor()</code>	<code>CGContext.SetFillColor()</code>
<code>NetworkReachability.SetCallback()</code>	<code>NetworkReachability.SetNotification()</code>
<code>CGContext.SetShadowWithColor</code>	<code>CGContext.SetShadow</code>
<code>UIView.StringSize</code>	<code>UIKit.UIStringDrawing.StringSize</code>

For a full list of changes when switching from the Classic to the Unified API, please see our [Classic \(monotouch.dll\) vs Unified \(Xamarin.iOS.dll\) API differences](#) documentation.

Updating to Unified

Several old/broken/deprecated API in **classic** are not available in the **Unified API**. It can be easier to fix the `CS0616` warnings before starting your (manual or automated) upgrade since you'll have the `[Obsolete]` attribute message (part of the warning) to guide you to the right API.

Note that we are publishing a [diff](#) of the classic vs unified API changes that can be used either before or after your project updates. Still fixing the obsoletes calls in Classic will often be a time saver (less documentation lookups).

Follow these instructions to [update existing iOS apps](#), or [Mac apps](#) to the Unified API. Review the remainder of this page, and [these tips](#) for additional information on migrating your code.

NuGet

NuGet packages that previously supported Xamarin.iOS via the Classic API published their assemblies using the **Monotouch10** platform moniker.

The Unified API introduces a new platform identifier for compatible packages - **Xamarin.iOS10**. Existing NuGet packages will need to be updated to add support for this platform, by building against the Unified API.

IMPORTANT

If you have an error in the form "*Error 3 Cannot include both 'monotouch.dll' and 'Xamarin.iOS.dll' in the same Xamarin.iOS project - 'Xamarin.iOS.dll' is referenced explicitly, while 'monotouch.dll' is referenced by 'xxx, Version=0.0.000, Culture=neutral, PublicKeyToken=null'*" after converting your application to the Unified APIs, it is typically due to having either a component or NuGet Package in the project that has not been updated to the Unified API. You'll need to remove the existing component/NuGet, update to a version that supports the Unified APIs and do a clean build.

The Road to 64 Bits

For background on supporting 32 and 64 bit applications and information about frameworks see the [32 and 64 bit Platform Considerations](#).

New Data Types

At the core of the difference, both Mac and iOS APIs use an architecture-specific data types that are always 32 bit on 32 bit platforms and 64 bit on 64 bit platforms.

For example, Objective-C maps the `NSInteger` data type to `int32_t` on 32 bit systems and to `int64_t` on 64 bit systems.

To match this behavior, on our Unified API, we are replacing the previous uses of `int` (which in .NET is defined as always being `System.Int32`) to a new data type: `System.nint`. You can think of the "n" as meaning "native", so the native integer type of the platform.

We are introducing `nint`, `nuint` and `nfloating` as well providing data types built on top of them where necessary.

To learn more about these data type changes, see the [Native Types](#) document.

How to detect the architecture of iOS apps

There might be situations where your application needs to know if it is running on a 32 bit or a 64 bit iOS system. The following code can be used to check the architecture:

```
if (IntPtr.Size == 4) {
    Console.WriteLine ("32-bit App");
} else if (IntPtr.Size == 8) {
    Console.WriteLine ("64-bit App");
}
```

Arrays and System.Collections.Generic

Because C# indexers expect a type of `int`, you'll have to explicitly cast `nint` values to `int` to access the elements in a collection or array. For example:

```

public List<string> Names = new List<string>();
...

public string GetName(nint index) {
    return Names[(int)index];
}

```

This is expected behavior, because the cast from `int` to `nint` is lossy on 64 bit, an implicit conversion is not done.

Converting DateTime to NSDate

When using the Unified APIs, the implicit conversion of `DateTime` to `NSDate` values is no longer performed. These values will need to be explicitly converted from one type to another. The following extension methods can be used to automate this process:

```

public static DateTime NSDate.ToDateTime(this NSDate date)
{
    // NSDate has a wider range than DateTime, so clip
    // the converted date to DateTime.Min|MaxValue.
    double secs = date.SecondsSinceReferenceDate;
    if (secs < -63113904000)
        return DateTime.MinValue;
    if (secs > 252423993599)
        return DateTime.MaxValue;
    return (DateTime) date;
}

public static NSDate DateTimeToNSDate(this DateTime date)
{
    if (date.Kind == DateTimeKind.Unspecified)
        date = DateTime.SpecifyKind (date, /* DateTimeKind.Local or DateTimeKind.Utc, this depends on each
app */)
    return (NSDate) date;
}

```

Deprecated APIs and Typos

Inside Xamarin.iOS classic API (`monotouch.dll`) the `[Obsolete]` attribute was used in two different ways:

- **Deprecated iOS API:** This is when Apple hints to you to stop using an API because it's being superseded by a newer one. The Classic API is still fine and often required (if you support the older version of iOS). Such API (and the `[Obsolete]` attribute) are included into the new Xamarin.iOS assemblies.
- **Incorrect API** Some API had typos on their names.

For the original assemblies (`monotouch.dll` and `XamMac.dll`) we kept the old code available for compatibility but they have been removed from the Unified API assemblies (`Xamarin.iOS.dll` and `Xamarin.Mac`)

NSObject subclasses .ctor(IntPtr)

Every `NSObject` subclass has a constructor that accepts an `IntPtr`. This is how we can instantiate a new managed instance from a native ObjC handle.

In classic this was a `public` constructor. However it was easy to misuse this feature in user code, e.g. creating several managed instances for a single ObjC instance *or* creating a managed instance that would lack the expected managed state (for subclasses).

To avoid those kind of problems the `IntPtr` constructors are now `protected` in unified API, to be used only for

subclassing. This will ensure the correct/safe API is used to create managed instance from handles, i.e.

```
var label = Runtime.GetObject<UILabel> (handle);
```

This API will return an existing managed instance (if it already exists) or will create a new one (when required). It is already available in both classic and unified API.

Note that the `.ctor(NSObjectFlag)` is now also `protected` but this one was rarely used outside of subclassing.

NSAction Replaced with Action

With the Unified APIs, `NSAction` has been removed in favor of the standard .NET `Action`. This is a big improvement because `Action` is a common .NET type, whereas `NSAction` was specific to Xamarin.iOS. They both do exactly the same thing, but they were distinct and incompatible types and resulted in more code having to be written to achieve the same result.

For example, if your existing Xamarin application included the following code:

```
UITapGestureRecognizer singleTap = new UITapGestureRecognizer (new NSAction (delegate() {
    ShowDropDownAnimated (tblDataView);
}));
```

It can now be replaced with a simple lambda:

```
UITapGestureRecognizer singleTap = new UITapGestureRecognizer (() => ShowDropDownAnimated(tblDataView));
```

Previously that would be a compiler error because an `Action` can't be assigned to `NSAction`, but since `UITapGestureRecognizer` now takes an `Action` instead of an `NSAction` it is valid in the Unified APIs.

Custom delegates replaced with Action<T>

In **unified** some simple (e.g. one parameter) .net delegates were replaced with `Action<T>`. E.g.

```
public delegate void NSNotificationHandler (NSNotification notification);
```

can now be used as an `Action<NSNotification>`. This promote code reuse and reduce code duplication inside both Xamarin.iOS and your own applications.

Task<bool> replaced with Task<Boolean, NSError>>

In **classic** there were some async APIs returning `Task<bool>`. However some of them where are to use when an `NSError` was part of the signature, i.e. the `bool` was already `true` and you had to catch an exception to get the `NSError`.

Since some errors are very common and the return value was not useful this pattern was changed in **unified** to return a `Task<Tuple<Boolean, NSError>>`. This allows you to check both the success and any error that could have happened during the async call.

NSString vs string

In a few cases some constants had to be changed from `string` to `NSString`, e.g. `UITableViewCell`

Classic

```
public virtual string ReuseIdentifier { get; }
```

Unified

```
public virtual NSString ReuseIdentifier { get; }
```

In general we prefer the .NET `System.String` type. However, despite Apple guidelines, some native API are comparing constant pointers (not the string itself) and this can only work when we expose the constants as `NSString`.

Objective-C Protocols

The original MonoTouch did not have full support for ObjC protocols and some, non-optimal, API were added to support the most common scenario. This limitation does not exists anymore but, for backward compatibility, several APIs are kept around inside `monotouch.dll` and `XamMac.dll`.

These limitations have been removed and cleaned up on the Unified APIs. Most changes will look like this:

Classic

```
public virtual AVAssetResourceLoaderDelegate Delegate { get; }
```

Unified

```
public virtual IAVAssetResourceLoaderDelegate Delegate { get; }
```

The `I` prefix means **unified** expose an interface, instead of a specific type, for the ObjC protocol. This will ease cases where you do not want to subclass the specific type that Xamarin.iOS provided.

It also allowed some API to be more precise and easy to use, e.g.:

Classic

```
public virtual void SelectionDidChange (NSObject uiTextInput);
```

Unified

```
public virtual void SelectionDidChange (UITextInput uiTextInput);
```

Such API are now easier to us, without referring to documentation, and your IDE code completion will provide you with more useful suggestions based on the protocol/interface.

NSCoding Protocol

Our original binding included an `.ctor(NSCoder)` for every type - even if it did not support the `NSCoding` protocol. A single `Encode(NSCoder)` method was present in the `NSObject` to encode the object. But this method would only work if the instance conformed to NSCoding protocol.

On the Unified API we have fixed this. The new assemblies will only have the `.ctor(NSCoder)` if the type conforms to `NSCoding`. Also such types now have an `Encode(NSCoder)` method which conforms to the `INSCoding` interface.

Low Impact: In most cases this change won't affect applications as the old, removed, constructors could not be used.

Further Tips

Additional changes to be aware of are listed in the [tips for updating apps to the Unified API](#).

Related Links

- [Updating iOS Apps](#)
- [Updating Mac Apps](#)
- [Updating Xamarin.Forms Apps](#)
- [Updating Bindings](#)
- [Updating Tips](#)
- [Classic vs Unified API differences](#)
- [Working with Native Types in Cross-Platform Apps](#)

Updating Existing Apps to the Unified API

10/28/2019 • 2 minutes to read • [Edit Online](#)

IMPORTANT

The Xamarin Classic API, which preceded the Unified API, has been deprecated.

- The last version of Xamarin.iOS to support the Classic API (`monotouch.dll`) was Xamarin.iOS 9.10.
- Xamarin.Mac still supports the Classic API, but it is no longer updated. Since it is deprecated, developers should move their applications to the Unified API.

How to Update Your Apps

There are three steps to update your apps:

1. Fix any compiler warnings in your existing code, particularly those relating to deprecated APIs.
2. Use the Migration Tool built in to Visual Studio for Mac to update your project files and namespaces.
3. Fix remaining compiler errors relating to the new [64-types](#) and [other APIs](#) that have changed. Check out [these tips](#) for additional information on manual updates that might be required.

There are specific guides available for each product to help you update your apps to the Unified API and 64-bit support:

Xamarin.iOS apps

Existing Xamarin.iOS apps can be updated to the Unified API using the automated migration tool built in to Visual Studio for Mac. Some additional fixes may then be required, as explained in [these instructions](#) and [tips](#).

Xamarin.Mac apps

Existing Xamarin.Mac apps can be updated to the Unified API using the automated migration tool built in to Visual Studio for Mac. Some additional fixes may then be required, as explained in [these instructions](#) and [tips](#).

Xamarin.Forms apps

Follow these instructions to update an existing Xamarin.Forms solution with an iOS project to use the Unified API. Unified API support is only available in Xamarin.Forms 1.3 and later, so [the instructions](#) also explain how to update your Xamarin.Forms app to version 1.3. These [tips](#) may help updating any native iOS code in custom renderers or dependency services.

Working with Native Types in Cross-Platform Apps

This article covers using the new iOS Unified API Native types (`nint`, `nuint`, `nfloat`) in a cross-platform application where code is shared with non-iOS devices such as Android or Windows Phone OSes. It provides insight into when the Native types should be used and provides several possible solutions to cases where the new type must be used with cross-platform code.

Update Bindings to the Unified API

Customers that have created bindings to Objective-C libraries will need to update the binding project to reflect changes in the underlying API (where some types will now be 64-bit). Follow these instructions to [update an existing Binding Project to support the Unified API](#).

Related Links

- [Updating iOS Apps](#)
- [Updating Mac Apps](#)
- [Updating Xamarin.Forms Apps](#)
- [Updating Bindings](#)
- [Updating Tips](#)
- [Classic vs Unified API differences](#)

Updating Existing iOS Apps

11/2/2020 • 4 minutes to read • [Edit Online](#)

Follow these steps to update an existing Xamarin.iOS app to use the Unified API.

Updating an existing app to use the Unified API requires changes to the project file itself as well as to the namespaces and APIs used in the application code.

The Road to 64 Bits

The new Unified APIs are required to support 64 bit device architectures from a Xamarin.iOS mobile application. As of February 1st, 2015 Apple requires that all new app submissions to the iTunes App Store support 64 bit architectures.

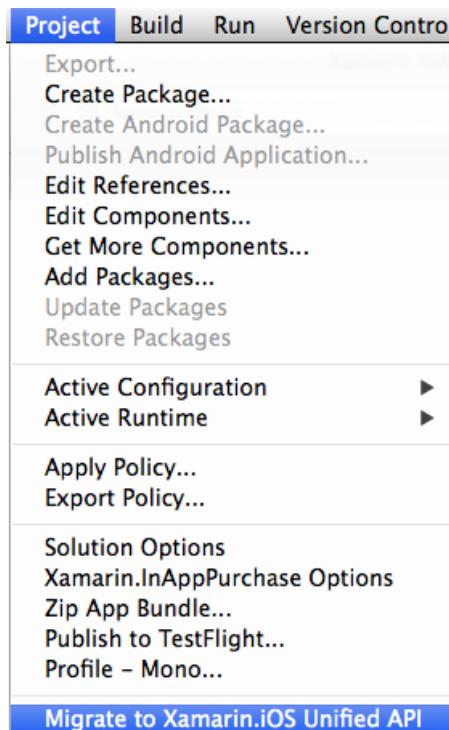
Xamarin provides tooling for both Visual Studio for Mac and Visual Studio to automate the migration process from the Classic API to the Unified API or you can convert the project files manually. While the using the automatic tooling is highly suggested, this article will cover both methods.

Before You Start...

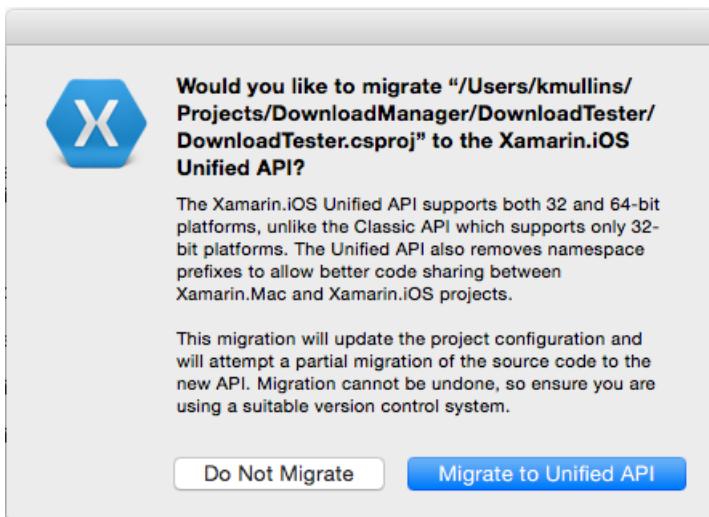
Before you update your existing code to the Unified API, it is highly recommended that you eliminate all *compilation warnings*. Many *warnings* in the Classic API will become errors once you migrate to Unified. Fixing them before you start is easier because the compiler messages from the Classic API often provide hints on what to update.

Automated Updating

Once the warnings have been fixed, select an existing iOS project in Visual Studio for Mac or Visual Studio and choose **Migrate to Xamarin.iOS Unified API** from the Project menu. For example:



You'll need to agree to this warning before the automated migration will run (obviously you should ensure you have backups/source control before embarking on this adventure):



The tool basically automates all the steps outlined in the **Update Manually** section presented below and is the suggested method of converting an existing Xamarin.iOS project to the Unified API.

Steps to Update Manually

Again, once the warnings have been fixed, follow these steps to manually update Xamarin.iOS apps to use the new Unified API:

1. Update Project Type & Build Target

Change the project flavor in your `csproj` files from `6BC8ED88-2882-458C-8E55-DFD12B67127B` to `FEACFB2D-3405-455C-9665-78FE426C6842`. Edit the `csproj` file in a text editor, replacing the first item in the `<ProjectTypeGuids>` element as shown:

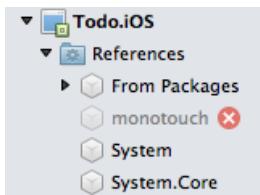
```
1  <?xml version="1.0" encoding="utf-8"?>
2  <Project DefaultTargets="Build" ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
3    <PropertyGroup>
4      <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
5      <Platform Condition=" '$(Platform)' == '' ">iPhoneSimulator</Platform>
6      <ProjectGuid>{1902913C-BB6F-4CCB-9B90-1D0CAEECAF12}</ProjectGuid>
7      <ProjectTypeGuids>{6BC8ED88-2882-458C-8E55-DFD12B67127B};{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}</ProjectTypeGuids>
8      <OutputType>Exe</OutputType>
9      <RootNamespace>Todo.iOS</RootNamespace>
10     <iPhoneResourcePrefix>Resources</iPhoneResourcePrefix>
11     <AssemblyName>TodoiOS</AssemblyName>
12   </PropertyGroup>
```

Change the `Import` element that contains `Xamarin.MonoTouch.CSharp.targets` to `Xamarin.iOS.CSharp.targets` as shown:

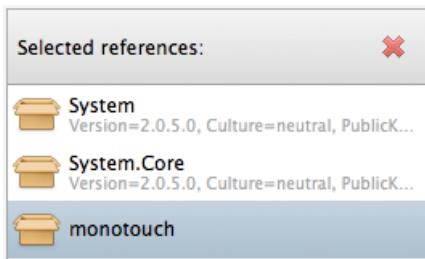
```
97  <Import Project="$(MSBuildExtensionsPath)\Xamarin\iOS\Xamarin.iOS.CSharp.targets" />
98 </Project>
```

2. Update Project References

Expand the iOS application project's **References** node. It will initially show a *broken- monotouch reference similar to this screenshot (because we just changed the project type):



Right-click on the iOS application project to **Edit References**, then click on the **monotouch** reference and delete it using the red "X" button.



Now scroll to the end of the references list and tick the `Xamarin.iOS` assembly.



Press OK to save the project references changes.

3. Remove MonoTouch from Namespaces

Remove the `MonoTouch` prefix from namespaces in `using` statements or wherever a classname has been fully qualified (eg. `MonoTouch.UIKit` becomes just `UIKit`).

4. Remap Types

[Native types](#) have been introduced which replace some Types that were previously used, such as instances of `System.Drawing.RectangleF` with `CoreGraphics.CGRect` (for example). The full list of types can be found on the [native types](#) page.

5. Fix Method Overrides

Some `UIKit` methods have had their signature changed to use the new [native types](#) (such as `nint`). If custom subclasses override these methods the signatures will no longer match and will result in errors. Fix these method overrides by changing the subclass to match the new signature using native types.

Examples include changing `public override int NumberOfSections (UITableView tableView)` to return `nint` and changing both return type and parameter types in

```
public override int RowsInSection (UITableView tableView, int section)
```

Considerations

The following considerations should be taken into account when converting an existing Xamarin.iOS project from the Classic API to the new Unified API if that app relies on one or more Component or NuGet Package.

Components

Any component that you have included in your application will also need to be updated to the Unified API or you will get a conflict when you try to compile. For any included component, replace the current version with a new version from the Xamarin Component Store that supports the Unified API and do a clean build. Any component that has not yet been converted by the author, will display a 32 bit only warning in the component store.

NuGet Support

While we contributed changes to NuGet to work with the Unified API support, there has not been a new release of NuGet, so we are evaluating how to get NuGet to recognize the new APIs.

Until that time, just like the components, you'll need to switch any NuGet Package you have included in your project to a version that supports the Unified APIs and do a clean build afterwards.

IMPORTANT

If you have an error in the form "*Error 3 Cannot include both 'monotouch.dll' and 'Xamarin.iOS.dll' in the same Xamarin.iOS project - 'Xamarin.iOS.dll' is referenced explicitly, while 'monotouch.dll' is referenced by 'xxx, Version=0.0.000, Culture=neutral, PublicKeyToken=null'*" after converting your application to the Unified APIs, it is typically due to having either a component or NuGet Package in the project that has not been updated to the Unified API. You'll need to remove the existing component/NuGet, update to a version that supports the Unified APIs and do a clean build.

Enabling 64 Bit Builds of Xamarin.iOS Apps

For a Xamarin.iOS mobile application that has been converted to the Unified API, the developer still needs to enable the building of the application for 64 bit machines from the app's Options. Please see the [Enabling 64 Bit Builds of Xamarin.iOS Apps](#) of the [32/64 bit Platform Considerations](#) document for detailed instructions on enabling 64 bit builds.

Finishing Up

Whether or not you choose to use the automatic or manual method to convert your Xamarin.iOS application from the Classic to the Unified APIs, there are several instances that will require further, manual intervention. Please see our [Tips for Updating Code to the Unified API](#) document for known issues and work arounds.

Related Links

- [Tips for Updating Code to the Unified API](#)
- [Working with Native Types in Cross-Platform Apps](#)
- [Classic vs Unified API differences](#)

Updating Existing Mac Apps

11/2/2020 • 5 minutes to read • [Edit Online](#)

Updating an existing app to use the Unified API requires changes to the project file itself as well as to the namespaces and APIs used in the application code.

The Road to 64 Bits

The new Unified APIs are required to support 64 bit device architectures from a Xamarin.Mac application. As of February 1st, 2015 Apple requires that all new app submissions to the Mac App Store support 64 bit architectures.

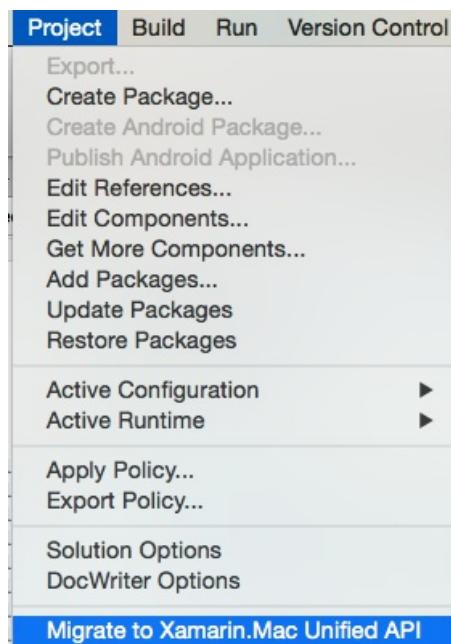
Xamarin provides tooling for both Visual Studio for Mac and Visual Studio to automate the migration process from the Classic API to the Unified API or you can convert the project files manually. While the using the automatic tooling is highly suggested, this article will cover both methods.

Before You Start...

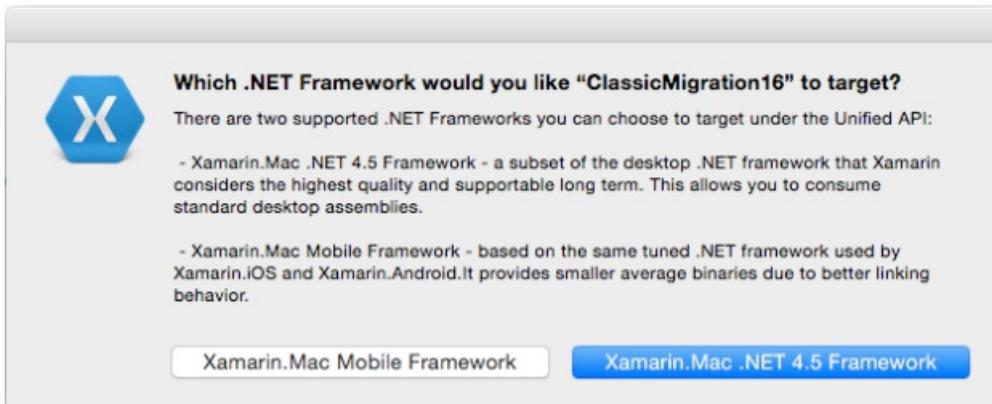
Before you update your existing code to the Unified API, it is highly recommended that you eliminate all *compilation warnings*. Many *warnings* in the Classic API will become errors once you migrate to Unified. Fixing them before you start is easier because the compiler messages from the Classic API often provide hints on what to update.

Automated Updating

Once the warnings have been fixed, select an existing Mac project in Visual Studio for Mac or Visual Studio and choose **Migrate to Xamarin.Mac Unified API** from the Project menu. For example:



You'll need to agree to this warning before the automated migration will run (obviously you should ensure you have backups/source control before embarking on this adventure):



There are two supported Target Framework types that can be selected when using the Unified API in a Xamarin.Mac application:

- **Xamarin.Mac Mobile Framework** - This is the same tuned .NET framework used by Xamarin.iOS and Xamarin.Android supporting a subset of the full **Desktop** framework. This is the recommended framework because it provides smaller average binaries due to superior linking behavior.
- **Xamarin.Mac .NET 4.5 Framework** - This framework is again, a subset of the **Desktop** framework. However, it trims off far less of the full **Desktop** framework than the **Mobile** framework and should *'just work'* with most NuGet Packages or 3rd party libraries. This allows the developer to consume standard **Desktop** assemblies while still using a supported framework, but this option produces larger application bundles. This is the recommended framework where 3rd party .NET assemblies are being used that are not compatible with the **Xamarin.Mac Mobile Framework**. For a list of supported assemblies, please see our [Assemblies](#) documentation.

For detailed information on Target Frameworks and the implications of selecting a specific target for your Xamarin.Mac application, please see our [Target Frameworks](#) documentation.

The tool basically automates all the steps outlined in the **Update Manually** section presented below and is the suggested method of converting an existing Xamarin.Mac project to the Unified API.

Steps to Update Manually

Again, once the warnings have been fixed, follow these steps to manually update Xamarin.Mac apps to use the new Unified API:

1. Update Project Type & Build Target

Change the project flavor in your `cproj` files from `42C0BB09-55CE-4FC1-8D90-A7348ABA0B23` to

`A3F8F2AB-B479-4A4A-A458-A89E7DC349F1`. Edit the `cproj` file in a text editor, replacing the first item in the

`<ProjectTypeGuids>` element as shown:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <Project DefaultTargets="Build" ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
3    <PropertyGroup>
4      <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
5      <Platform Condition=" '$(Platform)' == '' ">x86</Platform>
6      <ProductVersion>10.0.0</ProductVersion>
7      <SchemaVersion>2.0</SchemaVersion>
8      <ProjectGuid>{CAE4431E-ASBF-4F9C-9620-9B82D594B275}</ProjectGuid>
9      <ProjectTypeGuids>{A3F8F2AB-B479-4A4A-A458-A89E7DC349F1};{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}</ProjectTypeGuids>
10     <OutputType>Exe</OutputType>
11     <RootNamespace>Hello_Mac</RootNamespace>
12     <MonoMacResourcePrefix>Resources</MonoMacResourcePrefix>
13     <AssemblyName>Hello_Mac</AssemblyName>
14   </PropertyGroup>
15   <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|x86' ">
```

Change the `Import` element that contains `Xamarin.Mac.targets` to `Xamarin.Mac.CSharp.targets` as shown:

```
</ItemGroup>
<Import Project="$(MSBuildExtensionsPath)\Xamarin\Mac\Xamarin.Mac.CSharp.targets" />
</Project>
```

Add the following lines of code after the `<AssemblyName>` element:

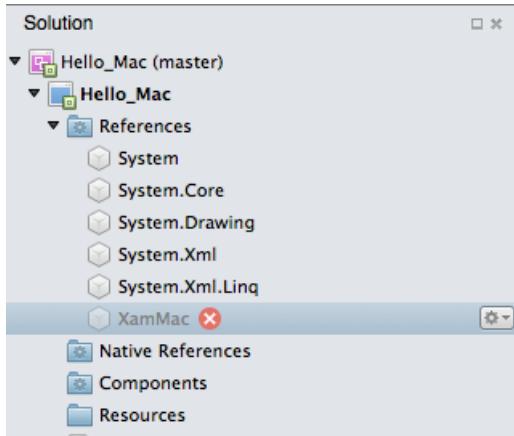
```
<TargetFrameworkVersion>v2.0</TargetFrameworkVersion>
<TargetFrameworkIdentifier>Xamarin.Mac</TargetFrameworkIdentifier>
```

Example:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <Project DefaultTargets="Build" ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
3   <PropertyGroup>
4     <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
5     <Platform Condition=" '$(Platform)' == '' ">x86</Platform>
6     <ProductVersion>8.0.30703</ProductVersion>
7     <SchemaVersion>2.0</SchemaVersion>
8     <ProjectGuid>{CAE4431E-A5BF-4F9C-9620-9B82D594B275}</ProjectGuid>
9     <ProjectTypeGuids>{A3FBF2AB-B479-4A4A-A458-A89E7DC349F1};{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}</ProjectTypeGuids>
10    <OutputType>Exe</OutputType>
11    <RootNamespace>Hello_Mac</RootNamespace>
12    <MonoMacResourcePrefix>Resources</MonoMacResourcePrefix>
13    <AssemblyName>Hello_Mac</AssemblyName>
14    <TargetFrameworkVersion>v2.0</TargetFrameworkVersion>
15    <TargetFrameworkIdentifier>Xamarin.Mac</TargetFrameworkIdentifier>
16  </PropertyGroup>
```

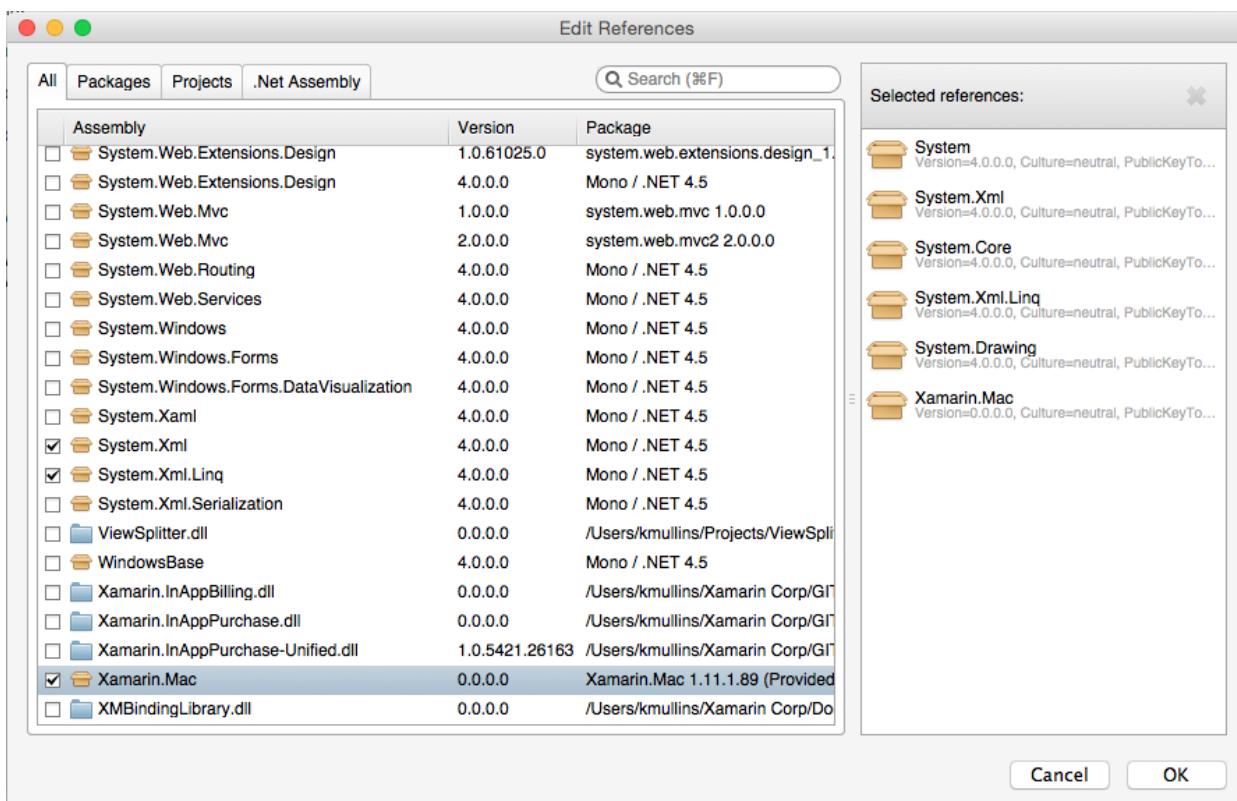
2. Update Project References

Expand the Mac application project's **References** node. It will initially show a *broken- **XamMac** reference similar to this screenshot (because we just changed the project type):



Click the Gear Icon beside the **XamMac** entry and select **Delete** to remove the broken reference.

Next, right-click on the **References** folder in the **Solution Explorer** and select **Edit References**. Scroll to the bottom of the list of references and place a check besides **Xamarin.Mac**.



Press OK to save the project references changes.

3. Remove MonoMac from Namespaces

Remove the `MonoMac` prefix from namespaces in `using` statements or wherever a classname has been fully qualified (eg. `MonoMac.AppKit` becomes just `AppKit`).

4. Remap Types

[Native types](#) have been introduced which replace some Types that were previously used, such as instances of `System.Drawing.RectangleF` with `CoreGraphics.CGRect` (for example). The full list of types can be found on the [native types](#) page.

5. Fix Method Overrides

Some `AppKit` methods have had their signature changed to use the new [native types](#) (such as `nint`). If custom subclasses override these methods the signatures will no longer match and will result in errors. Fix these method overrides by changing the subclass to match the new signature using native types.

Considerations

The following considerations should be taken into account when converting an existing Xamarin.Mac project from the Classic API to the new Unified API if that app relies on one or more Component or NuGet Package.

Components

Any component that you have included in your application will also need to be updated to the Unified API or you will get a conflict when you try to compile. For any included component, replace the current version with a new version from the Xamarin Component Store that supports the Unified API and do a clean build. Any component that has not yet been converted by the author, will display a 32 bit only warning in the component store.

NuGet Support

While we contributed changes to NuGet to work with the Unified API support, there has not been a new release of NuGet, so we are evaluating how to get NuGet to recognize the new APIs.

Until that time, just like the components, you'll need to switch any NuGet Package you have included in your

project to a version that supports the Unified APIs and do a clean build afterwards.

IMPORTANT

If you have an error in the form "*Error 3 Cannot include both 'monomac.dll' and 'Xamarin.Mac.dll' in the same Xamarin.Mac project - 'Xamarin.Mac.dll' is referenced explicitly, while 'monomac.dll' is referenced by 'xxx, Version=0.0.000, Culture=neutral, PublicKeyToken=null'*" after converting your application to the Unified APIs, it is typically due to having either a component or NuGet Package in the project that has not been updated to the Unified API. You'll need to remove the existing component/NuGet, update to a version that supports the Unified APIs and do a clean build.

Enabling 64 Bit Builds of Xamarin.Mac Apps

For a Xamarin.Mac mobile application that has been converted to the Unified API, the developer still needs to enable the building of the application for 64 bit machines from the app's Options. Please see the [Enabling 64 Bit Builds of Xamarin.Mac Apps](#) of the [32/64 bit Platform Considerations](#) document for detailed instructions on enabling 64 bit builds.

Finishing Up

Whether or not you choose to use the automatic or manual method to convert your Xamarin.Mac application from the Classic to the Unified APIs, there are several instances that will require further, manual intervention. Please see our [Tips for Updating Code to the Unified API](#) document for known issues and work arounds.

Related Links

- [Tips for Updating Code to the Unified API](#)
- [Working with Native Types in Cross-Platform Apps](#)
- [Classic vs Unified API differences](#)

Updating Existing Xamarin.Forms Apps

11/2/2020 • 8 minutes to read • [Edit Online](#)

Follow these steps to update an existing Xamarin.Forms app to use the Unified API and update to version 1.3.1

IMPORTANT

Because Xamarin.Forms 1.3.1 is the first release that supports the Unified API, the entire solution should be updated to use the latest version at the same time as migrating the iOS app to Unified. This means that in addition to updating the iOS project for Unified support, you'll also need to edit code in *all* the projects in the solution.

The update is performed in two steps:

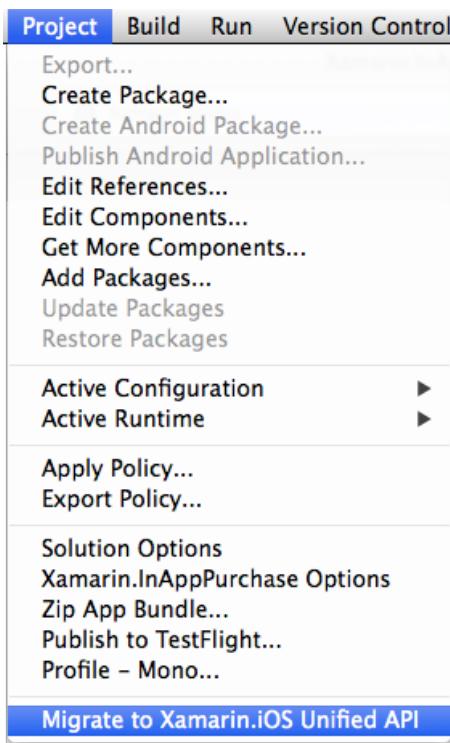
1. Migrate the iOS app to the Unified API using Visual Studio for Mac's build in migration tool.
 - Use the migration tool to automatically update the project.
 - Update iOS native APIs as outlined in the instructions to [update iOS apps](#) (specifically in custom renderer or dependency service code).
2. Update the entire solution to Xamarin.Forms version 1.3.
 - a. Install the Xamarin.Forms 1.3.1 NuGet package.
 - b. Update the `App` class in the shared code.
 - c. Update the `AppDelegate` in the iOS project.
 - d. Update the `MainActivity` in the Android project.
 - e. Update the `MainPage` in the Windows Phone project.

1. iOS App (Unified Migration)

Part of the migration requires upgrading Xamarin.Forms to version 1.3, which supports the Unified API. In order for the correct assembly references to be created, we first need to update the iOS project to use the Unified API.

Migration Tool

Click on the iOS project so that it's selected, then choose **Project > Migrate to Xamarin.iOS Unified API...** and agree to the warning message that appears.



This will automatically:

- Change the project type to support the Unified 64-bit API.
- Change the framework reference to **Xamarin.iOS** (replacing the old **monotouch** reference).
- Change the namespace references in the code to remove the `MonoTouch` prefix.
- Update the `csproj` file to use the correct build targets for the Unified API.

Clean and Build the project to ensure there are no other errors to fix. No further action should be required. These steps are explained in more detail in the [Unified API docs](#).

Update native iOS APIs (if required)

If you have added additional iOS native code (such as custom renderers or dependency services) you may need to perform additional manual code fixes. Re-compile your app and refer to the [Updating Existing iOS Apps instructions](#) for additional information on changes that may be required. [These tips](#) will also help identify changes that are required.

2. Xamarin.Forms 1.3.1 Update

Once the iOS app has been updated to the Unified API, the rest of the solution needs to be updated to Xamarin.Forms version 1.3.1. This includes:

- Updating the Xamarin.Forms NuGet package in each project.
- Changing the code to use the new Xamarin.Forms `Application`, `FormsApplicationDelegate` (iOS), `FormsApplicationActivity` (Android), and `FormsApplicationPage` (Windows Phone) classes.

These steps are explained below:

2.1 Update NuGet in all Projects

Update Xamarin.Forms to 1.3.1 pre-release using the NuGet Package Manager for all projects in the solution: PCL (if present), iOS, Android, and Windows Phone. It is recommended that you **delete and re-add** the Xamarin.Forms NuGet package to update to version 1.3.

NOTE

Xamarin.Forms version 1.3.1 is currently in *pre-release*. This means you must select the **pre-release** option in NuGet (via a tick-box in Visual Studio for Mac or a drop-down-list in Visual Studio) to see the latest pre-release version.

IMPORTANT

If you are using Visual Studio, ensure the latest version of the NuGet Package Manager is installed. Older versions of NuGet in Visual Studio will not correctly install the Unified version of Xamarin.Forms 1.3.1. Go to **Tools > Extensions and Updates...** and click on the **Installed** list to check that the **NuGet Package Manager for Visual Studio** is at least version 2.8.5. If it is older, click on the **Updates** list to download the latest version.

Once you've updated the NuGet package to Xamarin.Forms 1.3.1, make the following changes in each project to upgrade to the new `Xamarin.Forms.Application` class.

2.2 Portable Class Library (or Shared Project)

Change the `App.cs` file so that:

- The `App` class now inherits from `Application`.
- The `MainPage` property is set to the first content page you wish to display.

```
public class App : Application // superclass new in 1.3
{
    public App ()
    {
        // The root page of your application
        MainPage = new ContentPage {...}; // property new in 1.3
    }
}
```

We have completely removed the `GetMainPage` method, and instead set the `MainPage` *property* on the `Application` subclass.

This new `Application` base class also supports the `OnStart`, `OnSleep`, and `OnResume` overrides to help you manage your application's lifecycle.

The `App` class is then passed to a new `LoadApplication` method in each app project, as described below:

2.3 iOS App

Change the `AppDelegate.cs` file so that:

- The class inherits from `FormsApplicationDelegate` (instead of `UIApplicationDelegate` previously).
- `LoadApplication` is called with a new instance of `App`.

```
[Register ("AppDelegate")]
public partial class AppDelegate :
    global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate // superclass new in 1.3
{
    public override bool FinishedLaunching (UIApplication app, NSDictionary options)
    {
        global::Xamarin.Forms.Forms.Init ();

        LoadApplication (new App ()); // method is new in 1.3

        return base.FinishedLaunching (app, options);
    }
}
```

2.3 Android App

Change the **MainActivity.cs** file so that:

- The class inherits from `FormsApplicationActivity` (instead of `FormsActivity` previously).
- `LoadApplication` is called with a new instance of `App`

```
[Activity (Label = "YOURAPPNAME", Icon = "@drawable/icon", MainLauncher = true,
ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
public class MainActivity :
    global::Xamarin.Forms.Platform.Android.FormsApplicationActivity // superclass new in 1.3
{
    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        global::Xamarin.Forms.Forms.Init (this, bundle);

        LoadApplication (new App ()); // method is new in 1.3
    }
}
```

2.4 Windows Phone App

We need to update the **MainPage** - both the XAML and the codebehind.

Change the **MainPage.xaml** file so that:

- The root XAML element should be `winPhone:FormsApplicationPage`.
- The `xmlns:phone` attribute should be *changed to*

```
xmlns:winPhone="clr-namespace:Xamarin.Forms.Platform.WinPhone;assembly=Xamarin.Forms.Platform.WP8"
```

An updated example is shown below - you should only have to edit these things (the rest of the attributes should remain the same):

```
<winPhone:FormsApplicationPage
    ...
    xmlns:winPhone="clr-namespace:Xamarin.Forms.Platform.WinPhone;assembly=Xamarin.Forms.Platform.WP8"
    ...>
</winPhone:FormsApplicationPage>
```

Change the **MainPage.xaml.cs** file so that:

- The class inherits from `FormsApplicationPage` (instead of `PhoneApplicationPage` previously).
- `LoadApplication` is called with a new instance of the `Xamarin.Forms App` class. You may need to fully qualify this reference since Windows Phone has it's own `App` class already defined.

```

public partial class MainPage : global::Xamarin.Forms.Platform.WinPhone.FormsApplicationPage // superclass
new in 1.3
{
    public MainPage()
    {
        InitializeComponent();
        SupportedOrientations = SupportedPageOrientation.PortraitOrLandscape;

        global::Xamarin.Forms.Forms.Init();
        LoadApplication(new YOUR_APP_NAMESPACE.App()); // new in 1.3
    }
}

```

Troubleshooting

Occasionally you will see an error similar to this after updating the Xamarin.Forms NuGet package. It occurs when the NuGet updater does not completely remove references to older versions from your `csproj` files.

`YOUR_PROJECT.csproj`: Error: This project references NuGet package(s) that are missing on this computer. Enable NuGet Package Restore to download them. For more information, see <https://go.microsoft.com/fwlink/?LinkId=322105>. The missing file is `..\packages\Xamarin.Forms.1.2.3.6257\build\portable-win+net45+wp80+MonoAndroid10+MonoTouch10\Xamarin.Forms.targets`. (`YOUR_PROJECT`)

To fix these errors, open the `csproj` file in a text editor and look for `<Target>` elements that refer to older versions of Xamarin.Forms, such as the element shown below. You should manually delete this entire element from the `csproj` file and save the changes.

```

<Target Name="EnsureNuGetPackageBuildImports" BeforeTargets="PrepareForBuild">
    <PropertyGroup>
        <ErrorText>This project references NuGet package(s) that are missing on this computer. Enable NuGet Package Restore to download them. For more information, see https://go.microsoft.com/fwlink/?LinkId=322105. The missing file is {0}.</ErrorText>
    </PropertyGroup>
    <Error Condition="!Exists('..\..\packages\Xamarin.Forms.1.2.3.6257\build\portable-win+net45+wp80+MonoAndroid10+MonoTouch10\Xamarin.Forms.targets')"
Text="$([System.String]::Format('${ErrorText}', '..\..\packages\Xamarin.Forms.1.2.3.6257\build\portable-win+net45+wp80+MonoAndroid10+MonoTouch10\Xamarin.Forms.targets'))" />
</Target>

```

The project should build successfully once these old references are removed.

Considerations

The following considerations should be taken into account when converting an existing Xamarin.Forms project from the Classic API to the new Unified API if that app relies on one or more Component or NuGet Package.

Components

Any component that you have included in your application will also need to be updated to the Unified API or you will get a conflict when you try to compile. For any included component, replace the current version with a new version from the Xamarin Component Store that supports the Unified API and do a clean build. Any component that has not yet been converted by the author, will display a 32 bit only warning in the component store.

NuGet Support

While we contributed changes to NuGet to work with the Unified API support, there has not been a new release of NuGet, so we are evaluating how to get NuGet to recognize the new APIs.

Until that time, just like the components, you'll need to switch any NuGet Package you have included in your project to a version that supports the Unified APIs and do a clean build afterwards.

IMPORTANT

If you have an error in the form *"Error 3 Cannot include both 'monotouch.dll' and 'Xamarin.iOS.dll' in the same Xamarin.iOS project - 'Xamarin.iOS.dll' is referenced explicitly, while 'monotouch.dll' is referenced by 'xxx, Version=0.0.000, Culture=neutral, PublicKeyToken=null'"* after converting your application to the Unified APIs, it is typically due to having either a component or NuGet Package in the project that has not been updated to the Unified API. You'll need to remove the existing component/NuGet, update to a version that supports the Unified APIs and do a clean build.

Enabling 64 Bit Builds of Xamarin.iOS Apps

For a Xamarin.iOS mobile application that has been converted to the Unified API, the developer still needs to enable the building of the application for 64 bit machines from the app's Options. Please see the [Enabling 64 Bit Builds of Xamarin.iOS Apps](#) of the [32/64 bit Platform Considerations](#) document for detailed instructions on enabling 64 bit builds.

Summary

The Xamarin.Forms application should now be updated to version 1.3.1 and the iOS app migrated to the Unified API (which supports 64-bit architectures on the iOS platform).

As noted above, if your Xamarin.Forms app includes native code such as custom renderers or dependency services then these may also need updating to use the new Types [introduced in the Unified API](#).

Related Links

- [Updating iOS Apps](#)
- [Updating iOS Apps](#)
- [Working with Native Types in Cross-Platform Apps](#)
- [Updating Tips](#)
- [Classic vs Unified API differences](#)

Migrating a Binding to the Unified API

11/2/2020 • 6 minutes to read • [Edit Online](#)

This article covers the steps required to update an existing Xamarin Binding Project to support the Unified APIs for Xamarin.iOS and Xamarin.Mac applications.

Overview

Starting February 1st, 2015 Apple requires that all new submissions to the iTunes and the Mac App Store must be 64 bit applications. As a result, any new Xamarin.iOS or Xamarin.Mac application will need to be using the new Unified API instead of the existing Classic MonoTouch and MonoMac APIs to support 64 bit.

Additionally, any Xamarin Binding Project must also support the new Unified APIs to be included in a 64 bit Xamarin.iOS or Xamarin.Mac project. This article will cover the steps required to update an existing binding project to use the Unified API.

Requirements

The following is required to complete the steps presented in this article:

- **Visual Studio for Mac** - The latest version of Visual Studio for Mac installed and configured on the development computer.
- **Apple Mac** - An Apple mac is required to build Binding Projects for iOS and Mac.

Binding projects are not supported in Visual studio on a Windows machine.

Modify the Using Statements

The Unified APIs makes it easier than ever to share code between Mac and iOS as well as allowing you to support 32 and 64 bit applications with the same binary. By dropping the *MonoMac* and *MonoTouch* prefixes from the namespaces, simpler sharing is achieved across Xamarin.Mac and Xamarin.iOS application projects.

As a result we will need to modify any of our binding contracts (and other `.cs` files in our binding project) to remove the *MonoMac* and *MonoTouch* prefixes from our `using` statements.

For example, given the following using statements in a binding contract:

```
using System;
using System.Drawing;
using MonoTouch.Foundation;
using MonoTouch.UIKit;
using MonoTouch.ObjCRuntime;
```

We would strip off the `MonoTouch` prefix resulting in the following:

```
using System;
using System.Drawing;
using Foundation;
using UIKit;
using ObjCRuntime;
```

Again, we will need to do this for any `.cs` file in our binding project. With this change in place, the next step is

to update our binding project to use the new Native Data Types.

For more information on the Unified API, please see the [Unified API](#) documentation. For more background on supporting 32 and 64 bit applications and information about frameworks see the [32 and 64 bit Platform Considerations](#) documentation.

Update to Native Data Types

Objective-C maps the `NSInteger` data type to `int32_t` on 32 bit systems and to `int64_t` on 64 bit systems. To match this behavior, the new Unified API replaces the previous uses of `int` (which in .NET is defined as always being `System.Int32`) to a new data type: `System.nint`.

Along with the new `nint` data type, the Unified API introduces the `nuint` and `nfloat` types, for mapping to the `NSUInteger` and `CGFloat` types as well.

Given the above, we need to review our API and ensure that any instance of `NSInteger`, `NSUInteger` and `CGFloat` that we previously mapped to `int`, `uint` and `float` get updated to the new `nint`, `nuint` and `nfloat` types.

For example, given an Objective-C method definition of:

```
- (NSInteger) add:(NSInteger)operandUn and:(NSInteger) operandDeux;
```

If the previous binding contract had the following definition:

```
[Export("add:and:")]
int Add(int operandUn, int operandDeux);
```

We would update the new binding to be:

```
[Export("add:and:")]
nint Add(nint operandUn, nint operandDeux);
```

If we are mapping to a newer version 3rd party library than what we had initially linked to, we need to review the `.h` header files for the library and see if any exiting, explicit calls to `int`, `int32_t`, `unsigned int`, `uint32_t` or `float` have been upgraded to be an `NSInteger`, `NSUInteger` or a `CGFloat`. If so, the same modifications to the `nint`, `nuint` and `nfloat` types will need to be made to their mappings as well.

To learn more about these data type changes, see the [Native Types](#) document.

Update the CoreGraphics Types

The point, size and rectangle data types that are used with `CoreGraphics` use 32 or 64 bits depending on the device they are running on. When Xamarin originally bound the iOS and Mac APIs we used existing data structures that happened to match the data types in `System.Drawing` (`RectangleF` for example).

Because of the requirements to support 64 bits and the new native data types, the following adjustments will need to be made to existing code when calling `CoreGraphic` methods:

- `CGRect` - Use `CGRect` instead of `RectangleF` when defining floating point rectangular regions.
- `CGSize` - Use `CGSize` instead of `SizeF` when defining floating point sizes (width and height).
- `CGPoint` - Use `CGPoint` instead of `PointF` when defining a floating point location (X and Y coordinates).

Given the above, we will need to review our API and ensure that any instance of `CGRect`, `CGSize` or `CGPoint`

that was previously bound to `RectangleF`, `SizeF` or `PointF` be changed to the native type `CGRect`, `CGSize` or `CGPoint` directly.

For example, given an Objective-C initializer of:

```
- (id)initWithFrame:(CGRect)frame;
```

If our previous binding included the following code:

```
[Export ("initWithFrame")]
IntPtr Constructor (RectangleF frame);
```

We would update that code to:

```
[Export ("initWithFrame")]
IntPtr Constructor (CGRect frame);
```

With all of the code changes now in place, we need to modify our binding project or make file to bind against the Unified APIs.

Modify the Binding Project

As the final step to updating our binding project to use the Unified APIs, we need to either modify the `MakeFile` that we use to build the project or the Xamarin Project Type (if we are binding from within Visual Studio for Mac) and instruct `btouch` to bind against the Unified APIs instead of the Classic ones.

Updating a MakeFile

If we are using a makefile to build our binding project into a Xamarin .DLL, we will need to include the `--new-style` command line option and call `btouch-native` instead of `btouch`.

So given the following `MakeFile`:

```

BINDDIR=/src/binding
XBUILD=/Applications/Xcode.app/Contents/Developer/usr/bin/xcodebuild
PROJECT_ROOT=XMBindingLibrarySample
PROJECT=$(PROJECT_ROOT)/XMBindingLibrarySample.xcodeproj
TARGET=XMBindingLibrarySample
BTOUCH=/Developer/MonoTouch/usr/bin/btouch

all: XMBindingLibrary.dll

libXMBindingLibrarySample-i386.a:
$(XBUILD) -project $(PROJECT) -target $(TARGET) -sdk iphonesimulator -configuration Release clean build
-mv $(PROJECT_ROOT)/build/Release-iphonesimulator/lib$(TARGET).a $@

libXMBindingLibrarySample-arm64.a:
$(XBUILD) -project $(PROJECT) -target $(TARGET) -sdk iphoneos -arch arm64 -configuration Release clean
build
-mv $(PROJECT_ROOT)/build/Release-iphoneos/lib$(TARGET).a $@

libXMBindingLibrarySample-armv7.a:
$(XBUILD) -project $(PROJECT) -target $(TARGET) -sdk iphoneos -arch armv7 -configuration Release clean
build
-mv $(PROJECT_ROOT)/build/Release-iphoneos/lib$(TARGET).a $@

libXMBindingLibrarySampleUniversal.a: libXMBindingLibrarySample-armv7.a libXMBindingLibrarySample-i386.a
libXMBindingLibrarySample-arm64.a
lipo -create -output $@ $^

XMBindingLibrary.dll: AssemblyInfo.cs XMBindingLibrarySample.cs extras.cs
libXMBindingLibrarySampleUniversal.a
$(BTOUCH) -unsafe -out:$@ XMBindingLibrarySample.cs -x=AssemblyInfo.cs -x=extras.cs --link-
with=libXMBindingLibrarySampleUniversal.a,libXMBindingLibrarySampleUniversal.a

clean:
-rm -f *.a *.dll

```

We need to switch from calling `btouch` to `btouch-native`, so we would adjust our macro definition as follows:

```
BTOUCH=/Developer/MonoTouch/usr/bin/btouch-native
```

We would update the call to `btouch` and add the `--new-style` option as follows:

```

XMBindingLibrary.dll: AssemblyInfo.cs XMBindingLibrarySample.cs extras.cs
libXMBindingLibrarySampleUniversal.a
$(BTOUCH) -unsafe --new-style -out:$@ XMBindingLibrarySample.cs -x=AssemblyInfo.cs -x=extras.cs --link-
with=libXMBindingLibrarySampleUniversal.a,libXMBindingLibrarySampleUniversal.a

```

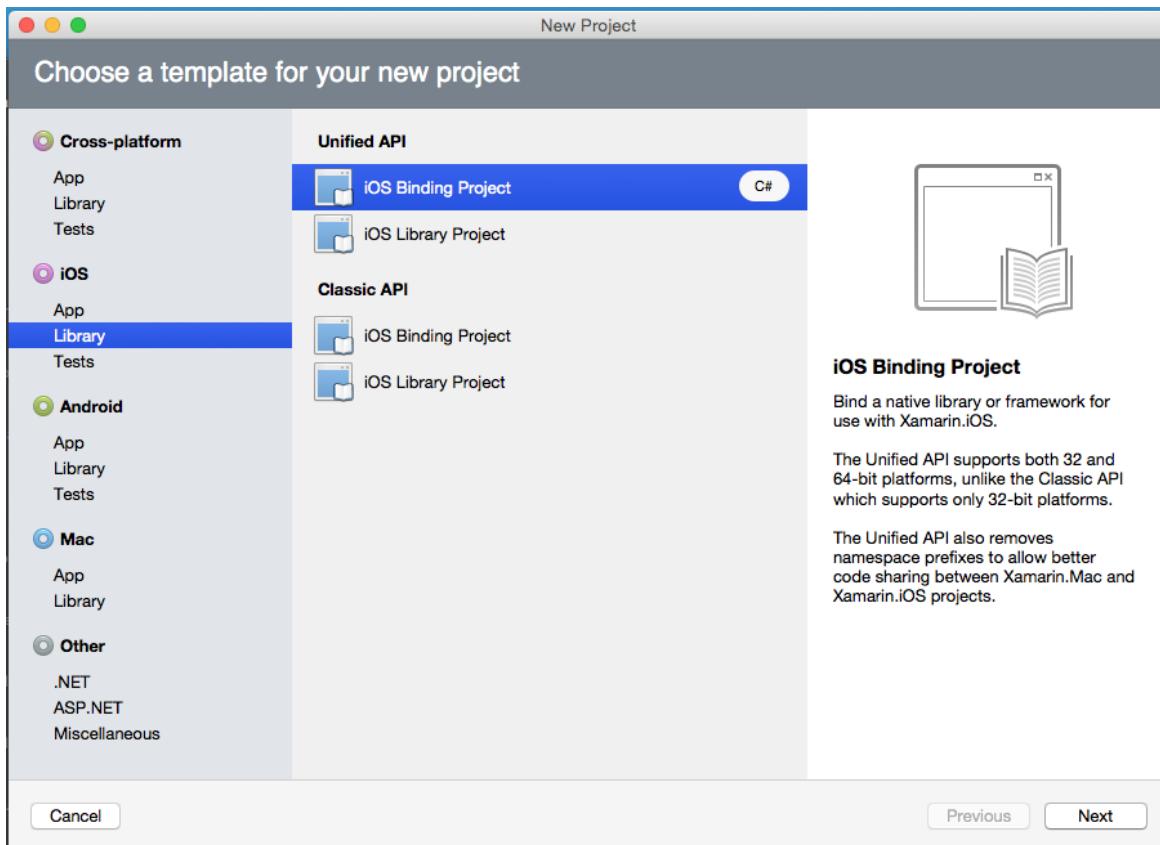
We can now execute our `Makefile` as normal to build the new 64 bit version of our API.

Updating a Binding Project Type

If we are using a Visual Studio for Mac Binding Project Template to build our API, we'll need to update to the new Unified API version of the Binding Project Template. The easiest way to do this is to start a new Unified API Binding Project and copy over all of the existing code and settings.

Do the following:

1. Start Visual Studio for Mac.
2. Select **File > New > Solution...**
3. In the New Solution Dialog Box, select **iOS > Unified API > iOS Binding Project**:



4. On 'Configure your new project' dialog enter a **Name** for the new binding project and click the **OK** button.
5. Include the 64 bit version of Objective-C library that you are going to be creating bindings for.
6. Copy over the source code from your existing 32 bit Classic API binding project (such as the `ApiDefinition.cs` and `StructsAndEnums.cs` files).
7. Make the above noted changes to the source code files.

With all of these changes in place, you can build the new 64 bit version of the API as you would the 32 bit version.

Summary

In this article we have shown the changes that need to be made to an existing Xamarin Binding Project to support the new Unified APIs and 64 bit devices and the steps required to build the new 64 bit compatible version of an API.

Related Links

- [Mac and iOS](#)
- [Unified API](#)
- [32/64 bit Platform Considerations](#)
- [Upgrading Existing iOS Apps](#)
- [Unified API](#)
- [BindingSample](#)

Tips for Updating Code to the Unified API

10/28/2019 • 5 minutes to read • [Edit Online](#)

When updating older Xamarin solutions to the Unified API, the following errors might be encountered.

NSInvalidArgumentException Could Not Find storyboard Error

There is a [bug](#) in the current version of Visual Studio for Mac that can occur after using the automated migration tool to convert your project to the Unified APIs. After the update, if you get an error message in the form:

```
Objective-C exception thrown. Name: NSInvalidArgumentException Reason: Could not find a storyboard named  
'xxx' in bundle NSBundle...
```

You can do the following to solve this issue, locate the following build target file:

```
/Library/Frameworks/Xamarin.iOS.framework/Versions/Current/lib/mono/2.1/Xamarin.iOS.Common.targets
```

In this file you need to find the following target declaration:

```
<Target Name="_CopyContentToBundle"  
      Inputs = "@(_BundleResourceWithLogicalName)"  
      Outputs = "@(_BundleResourceWithLogicalName -> '$(_AppBundlePath)%(_LogicalName)')" >
```

And add the `DependsOnTargets=_CollectBundleResources` attribute to it. Like this:

```
<Target Name="_CopyContentToBundle"  
      DependsOnTargets=_CollectBundleResources  
      Inputs = "@(_BundleResourceWithLogicalName)"  
      Outputs = "@(_BundleResourceWithLogicalName -> '$(_AppBundlePath)%(_LogicalName)')" >
```

Save the file, reboot Visual Studio for Mac, and do a clean & rebuild of your project. A fix for this issue should be released by Xamarin shortly.

Useful Tips

After using the migration tool, you may still get some compiler errors requiring manual intervention. Some things that might need to be manually fixed include:

- Comparing `enum`s might require an `(int)` cast.
- `NSDictionary.IntValue` now returns an `nint`, there is an `Int32Value` which can be used instead.
- `nfloat` and `nint` types cannot be marked `const`; `static readonly nint` is a reasonable alternative.
- Things that used to be directly in the `MonoTouch.` namespace are now generally in the `ObjCRuntime.` namespace (e.g.: `MonoTouch.Constants.Version` is now `ObjCRuntime.Constants.Version`).
- Code that serializes objects may break when attempting to serialize `nint` and `nfloat` types. Be sure to check your serialization code works as expected after migration.
- Sometimes the automated tool misses code inside `#if #else` conditional compiler directives. In this case

you'll need to make the fixes manually (see the common errors below).

- Manually exported methods using `[Export]` may not be automatically fixed by the migration tool, for example in this code snippet you must manually update the return type to `nfloat`:

```
[Export("tableView:heightForRowAtIndexPath:")]
public nfloat HeightForRow(UITableView tableView, NSIndexPath indexPath)
```

- The Unified API does not provide an implicit conversion between `NSDate` and `.NET DateTime` because it's not a lossless conversion. To prevent errors related to `DateTimeKind.Unspecified` convert the `.NET DateTime` to local or UTC before casting to `NSDate`.
- Objective-C category methods are now generated as extension methods in the Unified API. For example, code that previously used `UIView.DrawString` would now reference `NSString.DrawString` in the Unified API.
- Code using AVFoundation classes with `VideoSettings` should change to use the `WeakVideoSettings` property. This requires a `Dictionary`, which is available as a property on the settings classes, for example:

```
vidrec.WeakVideoSettings = new AVVideoSettings() { ... }.Dictionary;
```

- The `NSObject .ctor(IntPtr)` constructor has been changed from public to protected ([to prevent improper use](#)).
- `NSAction` has been [replaced](#) with the standard `.NET Action`. Some simple (single parameter) delegates have also been replaced with `Action<T>`.

Finally, refer to the [Classic v Unified API differences](#) to look up changes to APIs in your code. Searching [this page](#) will help find Classic APIs and what they've been updated to.

NOTE

The `MonoTouch.Dialog` namespace remains the same after migration. If your code uses `MonoTouch.Dialog` you should continue to use that namespace - do *not* change `MonoTouch.Dialog` to `Dialog`!

Common Compiler Errors

Other examples of common errors are listed below, along with the solution:

Error CS0012: The type 'MonoTouch.UIKit.UIView' is defined in an assembly that is not referenced.

Fix: This usually means the project references a component or NuGet package that has not been built with the Unified API. You should delete and re-add all Components and NuGet packages. If this does not fix the error, the external library may not yet support the Unified API.

Error MT0034: Cannot include both 'monotouch.dll' and 'Xamarin.iOS.dll' in the same Xamarin.iOS project - 'Xamarin.iOS.dll' is referenced explicitly, while 'monotouch.dll' is referenced by 'Xamarin.Mobile, Version=0.6.3.0, Culture=neutral, PublicKeyToken=null'.

Fix: Delete the component that is causing this error and re-add to the project.

Error CS0234: The type or namespace name 'Foundation' does not exist in the namespace 'MonoTouch'. Are you missing an assembly reference?

Fix: The automated migration tool in Visual Studio for Mac *should* update all `MonoTouch.Foundation` references to

`Foundation`, however in some instances these will need to be updated manually. Similar errors may appear for the other namespaces previously contained in `MonoTouch`, such as `UIKit`.

Error CS0266: Cannot implicitly convert type 'double' to 'System.float'

Fix: change type and cast to `nfloat`. This error may also occur for the other types with 64-bit equivalents (such as `nint`,)

```
nfloat scale = (nfloat)Math.Min(rect.Width, rect.Height);
```

Error CS0266: Cannot implicitly convert type 'CoreGraphics.CGRect' to 'System.Drawing.RectangleF'. An explicit conversion exists (are you missing a cast?)

Fix: Change instances to `RectangleF` to `CGRect`, `SizeF` to `CGSize`, and `PointF` to `CGPoint`. The namespace `using System.Drawing;` should be replaced with `using CoreGraphics;` (if it isn't already present).

error CS1502: The best overloaded method match for 'CoreGraphics.CGContext.SetLineDash(System.nfloat, System.nfloat[])' has some invalid arguments

Fix: Change array type to `nfloat[]` and explicitly cast `Math.PI`.

```
grphc.SetLineDash (0, new nfloat[] { 0, 3 * (nfloat)Math.PI });
```

Error CS0115: `WordsTableSource.RowsInSection(UIKit.UITableView, int)` is marked as an override but no suitable method found to override

Fix: Change the return value and parameter types to `nint`. This commonly occurs in method overrides such as those on `UITableViewSource`, including `RowsInSection`, `NumberOfSections`, `GetHeightForRow`, `TitleForHeader`, `GetViewForHeader`, etc.

```
public override nint RowsInSection (UITableView tableview, nint section) {
```

Error CS0508: `WordsTableSource.NumberOfSections(UIKit.UITableView)` : return type must be 'System.nint' to match overridden member `UIKit.UITableViewSource.NumberOfSections(UIKit.UITableView)`

Fix: When the return type is changed to `nint`, cast the return value to `nint`.

```
public override nint NumberOfSections (UITableView tableView)
{
    return (nint)navItems.Count;
}
```

Error CS1061: Type 'CoreGraphics.CGPath' does not contain a definition for 'AddEllipseInRect'

Fix: Correct spelling to `AddEllipseInRect`. Other name changes include:

- Change 'Color.Black' to `NSColor.Black`.
- Change MapKit 'AddAnnotation' to `AddAnnotations`.
- Change AVFoundation 'DataUsingEncoding' to `Encode`.
- Change AVFoundation 'AVMetadataObjectTypeQRCode' to `AVMetadataObjectType.QRCode`.
- Change AVFoundation 'VideoSettings' to `WeakVideoSettings`.
- Change PopViewControllerAnimated to `PopViewController`.

- Change CoreGraphics 'CGBitmapContext.SetRGBFillColor' to `SetFillColor`.

Error CS0546: cannot override because `MapKit.MKAnnotation.Coordinate' does not have an overridable set accessor (CS0546)

When creating a custom annotation by subclassing MKAnnotation the Coordinate field has no setter, only a getter.

Fix:

- Add a field to keep track of the coordinate
- return this field in the getter of the Coordinate property
- Override the SetCoordinate method and set your field
- Call SetCoordinate in your ctor with the passed in coordinate parameter

It should look similar to the following:

```
class BasicPinAnnotation : MKAnnotation
{
    private CLLocationCoordinate2D _coordinate;

    public override CLLocationCoordinate2D Coordinate
    {
        get
        {
            return _coordinate;
        }
    }

    public override void SetCoordinate(CLLocationCoordinate2D value)
    {
        _coordinate = value;
    }

    public BasicPinAnnotation (CLLocationCoordinate2D coordinate)
    {
        SetCoordinate(coordinate);
    }
}
```

Related Links

- [Updating Apps](#)
- [Updating iOS Apps](#)
- [Updating Mac Apps](#)
- [Updating Xamarin.Forms Apps](#)
- [Updating Bindings](#)
- [Working with Native Types in Cross-Platform Apps](#)
- [Classic vs Unified API differences](#)

Binding Objective-C

1/4/2020 • 2 minutes to read • [Edit Online](#)

This section includes a variety of documents that cover creating bindings to Objective-C libraries, so they can be called from C# applications created with Xamarin.iOS or Xamarin.Mac.

Overview

This document contains some of the internals of how a binding takes place. It is an advanced document with some technical information.

Binding Objective-C Libraries

This document describes the process used to create C# bindings of Objective-C APIs and how the idioms in Objective-C are mapped to the idioms used in .NET. If you are binding just C APIs, you should use the standard .NET mechanism for this, the P/Invoke framework.

Binding Definition Reference Guide

This is the reference guide that describes all of the attributes available to binding authors to drive the binding generation process.

Objective Sharpie

Objective Sharpie is a command line tool to help bootstrap the first pass of a binding. It works by parsing the header files of a native library to map the public API into the [binding definition](#) (a process that can also be done manually).

iOS

The [iOS binding page](#) links back to these common binding resources, in addition to the examples below.

Walkthrough: Binding an Objective-C Library

This article provides a step-by-step walkthrough of creating a binding project using the open source [InfColorPicker](#) Objective-C project as an example. The InfColorPicker library provides a reusable view controller that allow the user to select a color based on its HSB representation, making color selection more user-friendly. Objective Sharpie will be used to assist in the binding process.

Binding Samples

A collection of third-party bindings that can be used as reference when creating new Binding Projects.

Mac

Follow the [Mac binding](#) instructions to bind macOS libraries. You can create a new **Mac Bindings Library** from the **New Project** window:

RESIS

Android

App
Library
Tests

.NET Core

App
Library
Tests

Cloud

General

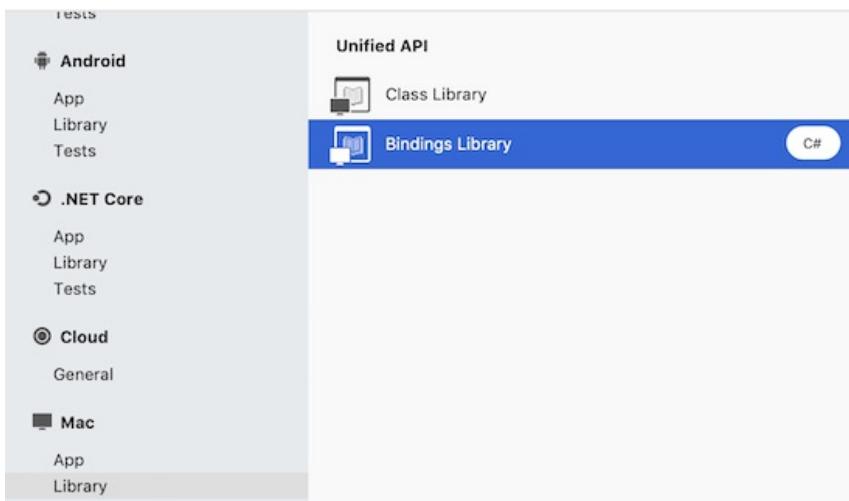
Mac

App
Library

Unified API

Class Library

Bindings Library C#



Bindings Library

Bind a native library or framework for use with Xamarin.Mac.

Related Links

- [iOS Binding](#)
- [Mac Binding](#)

Overview of Objective-C Bindings

11/2/2020 • 4 minutes to read • [Edit Online](#)

Details of how the binding process works

Binding an Objective-C library for use with Xamarin takes of three steps:

1. Write a C# "API definition" to describe how the native API is exposed in .NET, and how it maps to the underlying Objective-C. This is done using standard C# constructs like `interface` and various binding attributes (see this [simple example](#)).
2. Once you have written the "API definition" in C#, you compile it to produce a "binding" assembly. This can be done on the [command line](#) or using a [binding project](#) in Visual Studio for Mac or Visual Studio.
3. That "binding" assembly is then added to your Xamarin application project, so you can access the native functionality using the API you defined. The binding project is completely separate from your application projects.

NOTE

Step 1 can be automated with the assistance of [Objective Sharpie](#). It examines the Objective-C API and generates a proposed C# "API definition." You can customize the files created by Objective Sharpie and use them in a binding project (or on the command line) to create your binding assembly. Objective Sharpie does not create bindings by itself, it's merely an optional part of the larger process.

You can also read more technical details of [how it works](#), which will help you to write your bindings.

Command Line Bindings

You can use the `btouch-native` for Xamarin.iOS (or `bmac-native` if you are using Xamarin.Mac) to build bindings directly. It works by passing the C# API definitions that you've created by hand (or using Objective Sharpie) to the command line tool (`btouch-native` for iOS or `bmac-native` for Mac).

The general syntax for invoking these tools is:

```
# Use this for Xamarin.iOS:  
bash$ /Developer/MonoTouch/usr/bin/btouch-native -e cocos2d.cs -s:enums.cs -x:extensions.cs
```

```
# Use this for Xamarin.Mac:  
bash$ bmac-native -e cocos2d.cs -s:enums.cs -x:extensions.cs
```

The above command will generate the file `cocos2d.dll` in the current directory, and it will contain the fully bound library that you can use in your project. This is the tool that Visual Studio for Mac uses to create your bindings if you use a binding project (described [below](#)).

Binding Project

A binding project can be created in Visual Studio for Mac or Visual Studio (Visual Studio only supports iOS bindings) and makes it easier to edit and build API definitions for binding (versus using the command line).

Follow this [getting started guide](#) to see how to create and use a binding project to produce a binding.

Objective Sharpie

Objective Sharpie is another, separate command line tool that helps with the initial stages of creating a binding. It does not create a binding by itself, rather it automates the initial step of generating an API definition for the target native library.

Read the [Objective Sharpie docs](#) to learn how to parse native libraries, native frameworks, and CocoaPods into API definitions that can be built into bindings.

How Binding Works

It is possible to use the [\[Register\]](#) attribute, [\[Export\]](#) attribute, and [manual Objective-C selector invocation](#) together to manually bind new (previously unbound) Objective-C types.

First, find a type that you wish to bind. For discussion purposes (and simplicity), we'll bind the [NSEnumerator](#) type (which has already been bound in [Foundation.NSEnumerator](#); the implementation below is just for example purposes).

Second, we need to create the C# type. We'll likely want to place this into a namespace; since Objective-C doesn't support namespaces, we'll need to use the [\[Register\]](#) attribute to change the type name that Xamarin.iOS will register with the Objective-C runtime. The C# type must also inherit from [Foundation.NSObject](#):

```
namespace Example.Binding {
    [Register("NSEnumerator")]
    class NSEnumerator : NSObject
    {
        // see steps 3-5
    }
}
```

Third, review the Objective-C documentation and create [ObjCRuntime.Selector](#) instances for each selector you wish to use. Place these within the class body:

```
static Selector selInit      = new Selector("init");
static Selector selAllObjects = new Selector("allObjects");
static Selector selNextObject = new Selector("nextObject");
```

Fourth, your type will need to provide constructors. You *must* chain your constructor invocation to the base class constructor. The [\[Export\]](#) attributes permit Objective-C code to call the constructors with the specified selector name:

```
[Export("init")]
public NSEnumerator()
    : base(NSObjectFlag.Empty)
{
    Handle = Messaging.IntPtr_objc_msgSend(this.Handle, selInit.Handle);
}
```

```
// This constructor must be present so that Xamarin.iOS  
// can create instances of your type from Objective-C code.  
public NSEnumerator(IntPtr handle)  
    : base(handle)  
{  
}
```

Fifth, provide methods for each of the Selectors declared in Step 3. These will use `objc_msgSend()` to invoke the selector on the native object. Note the use of `Runtime.GetNSObject()` to convert an `IntPtr` into an appropriately typed `NSObject` (sub-)type. If you want the method to be callable from Objective-C code, the member *must* be **virtual**.

```
[Export("nextObject")]  
public virtual NSObject NextObject()  
{  
    return Runtime.GetNSObject(  
        Messaging.IntPtr_objc_msgSend(this.Handle, selNextObject.Handle));  
}
```

```
// Note that for properties, [Export] goes on the get/set method:  
public virtual NSArray AllObjects {  
    [Export("allObjects")]  
    get {  
        return (NSArray) Runtime.GetNSObject(  
            Messaging.IntPtr_objc_msgSend(this.Handle, selAllObjects.Handle));  
    }  
}
```

Putting it all together:

```
using System;
using Foundation;
using ObjCRuntime;

namespace Example.Binding {
    [Register("NSEnumerator")]
    class NSEnumerator : NSObject
    {
        static Selector selInit      = new Selector("init");
        static Selector selAllObjects = new Selector("allObjects");
        static Selector selNextObject = new Selector("nextObject");

        [Export("init")]
        public NSEnumerator()
            : base(NSObjectFlag.Empty)
        {
            Handle = Messaging.IntPtr_objc_msgSend(this.Handle,
                selInit.Handle);
        }

        public NSEnumerator(IntPtr handle)
            : base(handle)
        {
        }

        [Export("nextObject")]
        public virtual NSObject NextObject()
        {
            return Runtime.GetNSObject(
                Messaging.IntPtr_objc_msgSend(this.Handle,
                    selNextObject.Handle));
        }

        // Note that for properties, [Export] goes on the get/set method:
        public virtual NSArray AllObjects {
            [Export("allObjects")]
            get {
                return (NSArray) Runtime.GetNSObject(
                    Messaging.IntPtr_objc_msgSend(this.Handle,
                        selAllObjects.Handle));
            }
        }
    }
}
```

Binding Objective-C libraries

11/3/2020 • 42 minutes to read • [Edit Online](#)

When working with Xamarin.iOS or Xamarin.Mac, you might encounter cases where you want to consume a third-party Objective-C library. In those situations, you can use Xamarin Binding Projects to create a C# binding to the native Objective-C libraries. The project uses the same tools that we use to bring the iOS and Mac APIs to C#.

This document describes how to bind Objective-C APIs, if you are binding just C APIs, you should use the standard .NET mechanism for this, [the P/Invoke framework](#). Details on how to statically link a C library are available on the [Linking Native Libraries](#) page.

See our companion [Binding Types Reference Guide](#). Additionally, if you want to learn more about what happens under the hood, check our [Binding Overview](#) page.

Bindings can be built for both iOS and Mac libraries. This page describes how to work on an iOS binding, however Mac bindings are very similar.

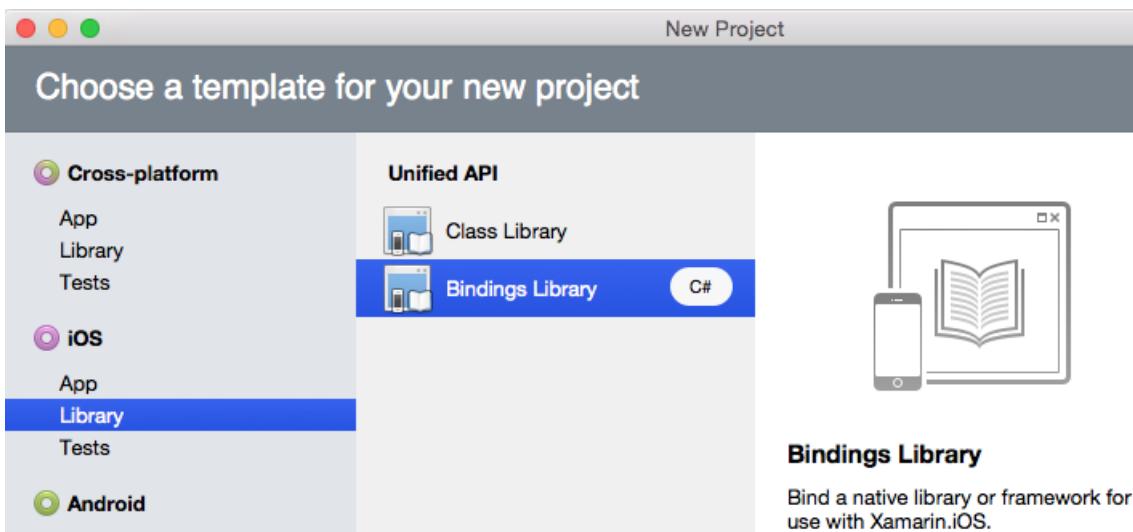
Sample Code for iOS

You can use the [iOS Binding Sample](#) project to experiment with bindings.

Getting started

- [Visual Studio for Mac](#)
- [Visual Studio](#)

The easiest way to create a binding is to create a Xamarin.iOS Binding Project. You can do this from Visual Studio for Mac by selecting the project type, **iOS > Library > Bindings Library**:



The generated project contains a small template that you can edit, it contains two files: `ApiDefinition.cs` and `StructsAndEnums.cs`.

The `ApiDefinition.cs` is where you will define the API contract, this is the file that describes how the underlying Objective-C API is projected into C#. The syntax and contents of this file are the main topic of discussion of this document and the contents of it are limited to C# interfaces and C# delegate declarations. The `StructsAndEnums.cs` file is the file where you will enter any definitions that are required by the interfaces and

delegates. This includes enumeration values and structures that your code might use.

Binding an API

To do a comprehensive binding, you will want to understand the Objective-C API definition and familiarize yourself with the .NET Framework Design guidelines.

To bind your library you will typically start with an API definition file. An API definition file is merely a C# source file that contains C# interfaces that have been annotated with a handful of attributes that help drive the binding. This file is what defines what the contract between C# and Objective-C is.

For example, this is a trivial api file for a library:

```
using Foundation;

namespace Cocos2D {
    [BaseType (typeof (NSObject))]
    interface Camera {
        [Static, Export ("getZEye")]
        nfloat ZEye { get; }

        [Export ("restore")]
        void Restore ();

        [Export ("locate")]
        void Locate ();

        [Export ("setEyeX:eyeY:eyeZ:")]
        void SetEyeXYZ (nfloat x, nfloat y, nfloat z);

        [Export ("setMode:")]
        void SetMode (CameraMode mode);
    }
}
```

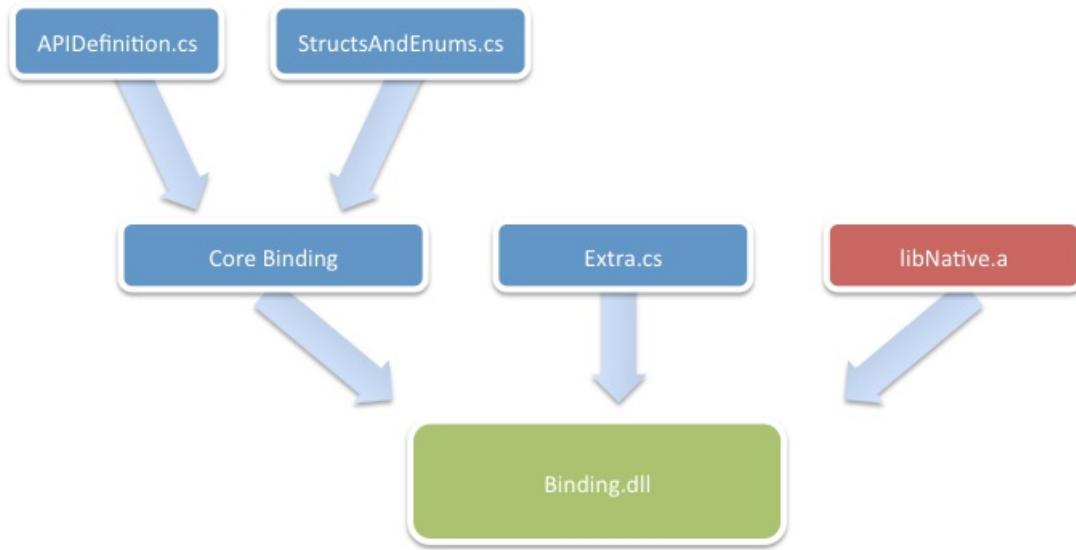
The above sample defines a class called `Cocos2D.Camera` that derives from the `NSObject` base type (this type comes from `Foundation.NSObject`) and which defines a static property (`ZEye`), two methods that take no arguments and a method that takes three arguments.

An in-depth discussion of the format of the API file and the attributes that you can use is covered in the [API definition file](#) section below.

To produce a complete binding, you will typically deal with four components:

- The API definition file (`ApiDefinition.cs` in the template).
- Optional: any enums, types, structs required by the API definition file (`StructsAndEnums.cs` in the template).
- Optional: extra sources that might expand the generated binding, or provide a more C# friendly API (any C# files that you add to the project).
- The native library that you are binding.

This chart shows the relationship between the files:



The API Definition file will only contain namespaces and interface definitions (with any members that an interface can contain), and should not contain classes, enumerations, delegates or structs. The API definition file is merely the contract that will be used to generate the API.

Any extra code that you need like enumerations or supporting classes should be hosted on a separate file, in the example above the "CameraMode" is an enumeration value that does not exist in the CS file and should be hosted in a separate file, for example `StructsAndEnums.cs` :

```

public enum CameraMode {
    FlyOver, Back, Follow
}

```

The `APIDefinition.cs` file is combined with the `StructsAndEnum` class and are used to generate the core binding of the library. You can use the resulting library as-is, but typically, you will want to tune the resulting library to add some C# features for the benefit of your users. Some examples include implementing a `Tostring()` method, provide C# indexers, add implicit conversions to and from some native types or provide strongly-typed versions of some methods. These improvements are stored in extra C# files. Merely add the C# files to your project and they will be included in this build process.

This shows how you would implement the code in your `Extra.cs` file. Notice that you will be using partial classes as these augment the partial classes that are generated from the combination of the `ApiDefinition.cs` and the `StructsAndEnums.cs` core binding:

```

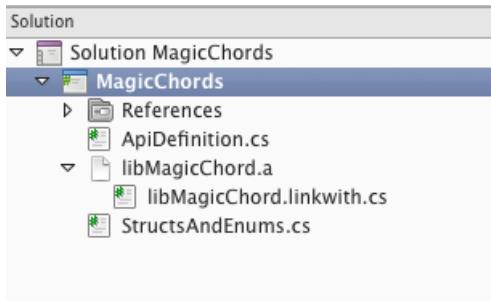
public partial class Camera {
    // Provide a ToString method
    public override string ToString ()
    {
        return String.Format ("ZEye: {0}", ZEye);
    }
}

```

Building the library will produce your native binding.

To complete this binding, you should add the native library to the project. You can do this by adding the native library to your project, either by dragging and dropping the native library from Finder onto the project in the solution explorer, or by right-clicking the project and choosing **Add > Add Files** to select the native library. Native libraries by convention start with the word "lib" and end with the extension ".a". When you do this, Visual Studio for Mac will add two files: the .a file and an automatically populated C# file that contains information

about what the native library contains:



The contents of the `libMagicChord.linkwith.cs` file has information about how this library can be used and instructs your IDE to package this binary into the resulting DLL file:

```
using System;
using ObjCRuntime;

[assembly: LinkWith ("libMagicChord.a", SmartLink = true, ForceLoad = true)]
```

Full details about how to use the `[LinkWith]` attribute are documented in the [Binding Types Reference Guide](#).

Now when you build the project you will end up with a `MagicChords.dll` file that contains both the binding and the native library. You can distribute this project or the resulting DLL to other developers for their own use.

Sometimes you might find that you need a few enumeration values, delegate definitions or other types. Do not place those in the API definitions file, as this is merely a contract

The API definition file

The API definition file consists of a number of interfaces. The interfaces in the API definition will be turned into a class declaration and they must be decorated with the `[BaseType]` attribute to specify the base class for the class.

You might be wondering why we did not use classes instead of interfaces for the contract definition. We picked interfaces because it allowed us to write the contract for a method without having to supply a method body in the API definition file, or having to supply a body that had to throw an exception or return a meaningful value.

But since we are using the interface as a skeleton to generate a class we had to resort to decorating various parts of the contract with attributes to drive the binding.

Binding methods

The simplest binding you can do is to bind a method. Just declare a method in the interface with the C# naming conventions and decorate the method with the `[Export]` attribute. The `[Export]` attribute is what links your C# name with the Objective-C name in the Xamarin.iOS runtime. The parameter of the `[Export]` attribute is the name of the Objective-C selector. Some examples:

```
// A method, that takes no arguments
[Export ("refresh")]
void Refresh ();

// A method that takes two arguments and return the result
[Export ("add:and:")]
nint Add (nint a, nint b);

// A method that takes a string
[Export ("draw:atColumn:andRow:")]
void Draw (string text, nint column, nint row);
```

The above samples show how you can bind instance methods. To bind static methods, you must use the [\[Static\]](#) attribute, like this:

```
// A static method, that takes no arguments
[Static, Export ("refresh")]
void Beep ();
```

This is required because the contract is part of an interface, and interfaces have no notion of static vs instance declarations, so it is necessary once again to resort to attributes. If you want to hide a particular method from the binding, you can decorate the method with the [\[Internal\]](#) attribute.

The `btouch-native` command will introduce checks for reference parameters to not be null. If you want to allow null values for a particular parameter, use the [\[NullAllowed\]](#) attribute on the parameter, like this:

```
[Export ("setText")]
string SetText ([NullAllowed] string text);
```

When exporting a reference type, with the [\[Export\]](#) keyword you can also specify the allocation semantics. This is necessary to ensure that no data is leaked.

Binding properties

Just like methods, Objective-C properties are bound using the [\[Export\]](#) attribute and map directly to C# properties. Just like methods, properties can be decorated with the [\[Static\]](#) and the [\[Internal\]](#) attributes.

When you use the [\[Export\]](#) attribute on a property under the covers btouch-native actually binds two methods: the getter and the setter. The name that you provide to export is the **basename** and the setter is computed by prepending the word "set", turning the first letter of the **basename** into upper case and making the selector take an argument. This means that `[Export ("label")]` applied on a property actually binds the "label" and "setLabel:" Objective-C methods.

Sometimes the Objective-C properties do not follow the pattern described above and the name is manually overwritten. In those cases you can control the way that the binding is generated by using the [\[Bind\]](#) attribute on the getter or setter, for example:

```
[Export ("menuVisible")]
bool MenuVisible { [Bind ("isMenuVisible")] get; set; }
```

This then binds "isMenuVisible" and "setMenuVisible:". Optionally, a property can be bound using the following syntax:

```
[Category, BaseType(typeof(UIView))]
interface UIView_MyIn
{
    [Export ("name")]
    string Name();

    [Export("setName:")]
    void SetName(string name);
}
```

Where the getter and setter are explicitly defined as in the `name` and `setName` bindings above.

In addition to support for static properties using `[Static]`, you can decorate thread-static properties with `[IsThreadStatic]`, for example:

```
[Export ("currentRunLoop")][Static][IsThreadStatic]
NSRunLoop Current { get; }
```

Just like methods allow some parameters to be flagged with `[NullAllowed]`, you can apply `[NullAllowed]` to a property to indicate that null is a valid value for the property, for example:

```
[Export ("text"), NullAllowed]
string Text { get; set; }
```

The `[NullAllowed]` parameter can also be specified directly on the setter:

```
[Export ("text")]
string Text { get; [NullAllowed] set; }
```

Caveats of binding custom controls

The following caveats should be considered when setting up the binding for a custom control:

- Binding properties must be static** - When defining the binding of properties, the `[Static]` attribute must be used.
- Property names must match exactly** - The name used to bind the property must match the name of the property in the custom control exactly.
- Property types must match exactly** - The variable type used to bind the property must match the type of the property in the custom control exactly.
- Breakpoints and the getter/setter** - Breakpoints placed in the getter or setter methods of the property will never be hit.
- Observe Callbacks** - You will need to use observation callbacks to be notified of changes in the property values of custom controls.

Failure to observe any of the above listed caveats can result in the binding silently failing at runtime.

Objective-C mutable pattern and properties

Objective-C frameworks use an idiom where some classes are immutable with a mutable subclass. For example `NSString` is the immutable version, while `NSMutableString` is the subclass that allows mutation.

In those classes it is common to see the immutable base class contain properties with a getter, but no setter. And for the mutable version to introduce the setter. Since this is not really possible with C#, we had to map this idiom into an idiom that would work with C#.

The way that this is mapped to C# is by adding both the getter and the setter on the base class, but flagging the

setter with a `[NotImplemented]` attribute.

Then, on the mutable subclass, you use the `[Override]` attribute on the property to ensure that the property is actually overriding the parent's behavior.

Example:

```
[BaseType (typeof (NSObject))]
interface MyTree {
    string Name { get; [NotImplemented] set; }
}

[BaseType (typeof (MyTree))]
interface MyMutableTree {
    [Override]
    string Name { get; set; }
}
```

Binding constructors

The `btouch-native` tool will automatically generate four constructors in your class, for a given class `Foo`, it generates:

- `Foo ()`: the default constructor (maps to Objective-C's "init" constructor)
- `Foo (NSCoder)`: the constructor used during deserialization of NIB files (maps to Objective-C's "initWithCoder:" constructor).
- `Foo (IntPtr handle)`: the constructor for handle-based creation, this is invoked by the runtime when the runtime needs to expose a managed object from an unmanaged object.
- `Foo (NSEmptyFlag)`: this is used by derived classes to prevent double initialization.

For constructors that you define, they need to be declared using the following signature inside the Interface definition: they must return an `IntPtr` value and the name of the method should be `Constructor`. For example to bind the `initWithFrame:` constructor, this is what you would use:

```
[Export ("initWithFrame")]
IntPtr Constructor (CGRect frame);
```

Binding protocols

As described in the API design document, in the section [discussing Models and Protocols](#), Xamarin.iOS maps the Objective-C protocols into classes that have been flagged with the `[Model]` attribute. This is typically used when implementing Objective-C delegate classes.

The big difference between a regular bound class and a delegate class is that the delegate class might have one or more optional methods.

For example consider the `UIKit` class `UIAccelerometerDelegate`, this is how it is bound in Xamarin.iOS:

```
[BaseType (typeof (NSObject))]
[Model][Protocol]
interface UIAccelerometerDelegate {
    [Export ("accelerometer:didAccelerate:")]
    void DidAccelerate (UIAccelerometer accelerometer, UIAcceleration acceleration);
}
```

Since this is an optional method on the definition for `UIAccelerometerDelegate` there is nothing else to do. But if

there was a required method on the protocol, you should add the `[Abstract]` attribute to the method. This will force the user of the implementation to actually provide a body for the method.

In general, protocols are used in classes that respond to messages. This is typically done in Objective-C by assigning to the "delegate" property an instance of an object that responds to the methods in the protocol.

The convention in Xamarin.iOS is to support both the Objective-C loosely-coupled style where any instance of an `NSObject` can be assigned to the delegate, and to also expose a strongly-typed version of it. For this reason, we typically provide both a `Delegate` property that is strongly typed and a `WeakDelegate` that is loosely typed. We usually bind the loosely-typed version with `[Export]`, and we use the `[Wrap]` attribute to provide the strongly-typed version.

This shows how we bound the `UIAccelerometer` class:

```
[BaseType (typeof (NSObject))]
interface UIAccelerometer {
    [Static] [Export ("sharedAccelerometer")]
    UIAccelerometer SharedAccelerometer { get; }

    [Export ("updateInterval")]
    double UpdateInterval { get; set; }

    [Wrap ("WeakDelegate")]
    UIAccelerometerDelegate Delegate { get; set; }

    [Export ("delegate", ArgumentSemantic.Assign)][NullAllowed]
    NSObject WeakDelegate { get; set; }
}
```

New in MonoTouch 7.0

Starting with MonoTouch 7.0 a new and improved protocol binding functionality has been incorporated. This new support makes it simpler to use Objective-C idioms for adopting one or more protocols in a given class.

For every protocol definition `MyProtocol` in Objective-C, there is now an `IMyProtocol` interface that lists all the required methods from the protocol, as well as an extension class that provides all the optional methods. The above, combined with new support in the Xamarin Studio editor allows developers to implement protocol methods without having to use the separate subclasses of the previous abstract model classes.

Any definition that contains the `[Protocol]` attribute will actually generate three supporting classes that vastly improve the way that you consume protocols:

```
// Full method implementation, contains all methods
class MyProtocol : IMyProtocol {
    public void Say (string msg);
    public void Listen (string msg);
}

// Interface that contains only the required methods
interface IMyProtocol: INativeObject, IDisposable {
    [Export ("say:")]
    void Say (string msg);
}

// Extension methods
static class IMyProtocol_Extensions {
    public static void Optional (this IMyProtocol this, string msg);
}
```

The **class implementation** provides a complete abstract class that you can override individual methods of and get full type safety. But due to C# not supporting multiple inheritance, there are scenarios where you might need to have a different base class, but you still want to implement an interface, that is where the

The generated **interface definition** comes in. It is an interface that has all the required methods from the protocol. This allows developers that want to implement your protocol to merely implement the interface. The runtime will automatically register the type as adopting the protocol.

Notice that the interface only lists the required methods and does not expose the optional methods. This means that classes that adopt the protocol will get full type checking for the required methods, but will have to resort to weak typing (manually using **[Export]** attributes and matching the signature) for the optional protocol methods.

To make it convenient to consume an API that uses protocols, the binding tool also will produce an extensions method class that exposes all of the optional methods. This means that as long as you are consuming an API, you will be able to treat protocols as having all the methods.

If you want to use the protocol definitions in your API, you will need to write skeleton empty interfaces in your API definition. If you want to use the `MyProtocol` in an API, you would need to do this:

```
[BaseType (typeof (NSObject))]
[Model, Protocol]
interface MyProtocol {
    // Use [Abstract] when the method is defined in the @required section
    // of the protocol definition in Objective-C
    [Abstract]
    [Export ("say")]
    void Say (string msg);

    [Export ("listen")]
    void Listen ();
}

interface IMyProtocol {}

[BaseType (typeof(NSObject))]
interface MyTool {
    [Export ("getProtocol")]
    IMyProtocol GetProtocol ();
}
```

The above is needed because at binding time the `IMyProtocol` would not exist, that is why you need to provide an empty interface.

Adopting protocol-generated interfaces

Whenever you implement one of the interfaces generated for the protocols, like this:

```
class MyDelegate : NSObject, IUITableViewDelegate {
    nint UITableViewDelegate.GetRowHeight (nint row) {
        return 1;
    }
}
```

The implementation for the required interface methods gets exported with the proper name, so it is equivalent to this:

```

class MyDelegate : NSObject, UITableViewDelegate {
    [Export ("getRowHeight")]
    nint UITableViewDelegate.GetRowHeight (nint row) {
        return 1;
    }
}

```

This will work for all required protocol members, but there is a special case with optional selectors to be aware of. Optional protocol members are treated identically when using the base class:

```

public class URLSessionDelegate : NSUrlSessionDownloadDelegate {
    public override void DidWriteData (NSUrlSession session, NSUrlSessionDownloadTask downloadTask, long bytesWritten, long totalBytesWritten, long totalBytesExpectedToWrite)
}

```

but when using the protocol interface it is required to add the [Export]. The IDE will add it via autocomplete when you add it starting with override.

```

public class URLSessionDelegate : NSObject, INSUrlSessionDownloadDelegate {
    [Export ("URLSession:downloadTask:didWriteData:totalBytesWritten:totalBytesExpectedToWrite:")]
    public void DidWriteData (NSUrlSession session, NSUrlSessionDownloadTask downloadTask, long bytesWritten, long totalBytesWritten, long totalBytesExpectedToWrite)
}

```

There is a slight behavior difference between the two at runtime.

- Users of the base class (NSUrlSessionDownloadDelegate in example) provides all required and optional selectors, returning reasonable default values.
- Users of the interface (INSUrlSessionDownloadDelegate in example) only respond to the exact selectors provided.

Some rare classes can behave differently here. In almost all cases however it is safe to use either.

Binding class extensions

In Objective-C it is possible to extend classes with new methods, similar in spirit to C#'s extension methods. When one of these methods is present, you can use the `[BaseType]` attribute to flag the method as being the receiver of the Objective-C message.

For example, in Xamarin.iOS we bound the extension methods that are defined on `NSString` when `UIKit` is imported as methods in the `NSStringDrawingExtensions`, like this:

```

[Category, BaseType (typeof (NSString))]
interface NSStringDrawingExtensions {
    [Export ("drawAtPoint:withFont:")]
    CGSize DrawString (CGPoint point, UIFont font);
}

```

Binding Objective-C argument lists

Objective-C supports variadic arguments. For example:

```

- (void) appendWorkers:(XWorker *) firstWorker, ...
    NS_REQUIRE_NIL_TERMINATION ;

```

To invoke this method from C# you will want to create a signature like this:

```
[Export ("appendWorkers"), Internal]
void AppendWorkers (Worker firstWorker, IntPtr workersPtr)
```

This declares the method as internal, hiding the above API from users, but exposing it to the library. Then you can write a method like this:

```
public void AppendWorkers(params Worker[] workers)
{
    if (workers == null)
        throw new ArgumentNullException ("workers");

    var pNativeArr = Marshal.AllocHGlobal(workers.Length * IntPtr.Size);
    for (int i = 1; i < workers.Length; ++i)
        Marshal.WriteIntPtr (pNativeArr, (i - 1) * IntPtr.Size, workers[i].Handle);

    // Null termination
    Marshal.WriteIntPtr (pNativeArr, (workers.Length - 1) * IntPtr.Size, IntPtr.Zero);

    // the signature for this method has gone from (IntPtr, IntPtr) to (Worker, IntPtr)
    WorkerManager.AppendWorkers(workers[0], pNativeArr);
    Marshal.FreeHGlobal(pNativeArr);
}
```

Binding fields

Sometimes you will want to access public fields that were declared in a library.

Usually these fields contain strings or integers values that must be referenced. They are commonly used as string that represent a specific notification and as keys in dictionaries.

To bind a field, add a property to your interface definition file, and decorate the property with the [\[Field\]](#) attribute. This attribute takes one parameter: the C name of the symbol to lookup. For example:

```
[Field ("NSSomeEventNotification")]
NSString NSSomeEventNotification { get; }
```

If you want to wrap various fields in a static class that does not derive from [NSObject](#), you can use the [\[Static\]](#) attribute on the class, like this:

```
[Static]
interface LonelyClass {
    [Field ("NSSomeEventNotification")]
    NSString NSSomeEventNotification { get; }
}
```

The above will generate a [LonelyClass](#) which does not derive from [NSObject](#) and will contain a binding to the [NSSomeEventNotification](#) [NSString](#) exposed as an [NSString](#).

The [\[Field\]](#) attribute can be applied to the following data types:

- [NSString](#) references (read-only properties only)
- [NSArray](#) references (read-only properties only)
- 32-bit ints ([System.Int32](#))
- 64-bit ints ([System.Int64](#))
- 32-bit floats ([System.Single](#))
- 64-bit floats ([System.Double](#))

- `System.Drawing.SizeF`
- `CGSize`

In addition to the native field name, you can specify the library name where the field is located, by passing the library name:

```
[Static]
interface LonelyClass {
    [Field ("SomeSharedLibrarySymbol", "SomeSharedLibrary")]
    NSString SomeSharedLibrarySymbol { get; }
}
```

If you are linking statically, there is no library to bind to, so you need to use the `__Internal` name:

```
[Static]
interface LonelyClass {
    [Field ("MyFieldFromALibrary", "__Internal")]
    NSString MyFieldFromALibrary { get; }
}
```

Binding enums

You can add `enum` directly in your binding files to makes it easier to use them inside API definitions - without using a different source file (that needs to be compiled in both the bindings and the final project).

Example:

```
[Native] // needed for enums defined as NSInteger in ObjC
enum MyEnum {}

interface MyType {
    [Export ("initWithEnum:")]
    IntPtr Constructor (MyEnum value);
}
```

It is also possible to create your own enums to replace `NSString` constants. In this case the generator will **automatically** create the methods to convert enums values and `NSString` constants for you.

Example:

```

enum NSRunLoopMode {

    [DefaultValue]
    [Field ("NSDefaultRunLoopMode")]
    Default,

    [Field ("NSRunLoopCommonModes")]
    Common,

    [Field (null)]
    Other = 1000
}

interface MyType {
    [Export ("performForMode:")]
    void Perform (NSString mode);

    [Wrap ("Perform (mode.GetConstant ())")]
    void Perform (NSRunLoopMode mode);
}

```

In the above example you could decide to decorate `void Perform (NSString mode);` with an [\[Internal\]](#) attribute. This will **hide** the constant-based API from your binding consumers.

However this would limit subclassing the type as the nicer API alternative uses a [\[Wrap\]](#) attribute. Those generated methods are not `virtual`, i.e. you won't be able to override them - which may, or not, be a good choice.

An alternative is to mark the original, `NSString`-based, definition as [\[Protected\]](#). This will allow subclassing to work, when required, and the wrap'ed version will still work and call the overridden method.

Binding `NSNumber`, `NSDate`, **and** `NSString` **to a better type**

The [\[BindAs\]](#) attribute allows binding `NSNumber`, `NSDate` and `NSString` (enums) into more accurate C# types. The attribute can be used to create better, more accurate, .NET API over the native API.

You can decorate methods (on return value), parameters and properties with [\[BindAs\]](#). The only restriction is that your member **must not** be inside a [\[Protocol\]](#) or [\[Model\]](#) interface.

For example:

```

@return: BindAs (typeof (bool?))
[Export ("shouldDrawAt:")]
NSNumber ShouldDraw ([BindAs (typeof (CGRect))] NSValue rect);

```

Would output:

```

[Export ("shouldDrawAt:")]
bool? ShouldDraw (CGRect rect) { ... }

```

Internally we will do the `bool? <-> NSNumber` and `CGRect <-> NSValue` conversions.

[\[BindAs\]](#) also supports arrays of `NSNumber`, `NSDate` and `NSString` (enums).

For example:

```
[BindAs (typeof (CAScroll []))]
[Export ("supportedScrollModes")]
NSString [] SupportedScrollModes { get; set; }
```

Would output:

```
[Export ("supportedScrollModes")]
CAScroll [] SupportedScrollModes { get; set; }
```

`CAScroll` is a `NSString` backed enum, we will fetch the right `NSString` value and handle the type conversion.

Please see the [\[BindAs\]](#) documentation to see supported conversion types.

Binding notifications

Notifications are messages that are posted to the `NSNotificationCenter.DefaultCenter` and are used as a mechanism to broadcast messages from one part of the application to another. Developers subscribe to notifications typically using the `NSNotificationCenter`'s `AddObserver` method. When an application posts a message to the notification center, it typically contains a payload stored in the `NSNotification.UserInfo` dictionary. This dictionary is weakly typed, and getting information out of it is error prone, as users typically need to read in the documentation which keys are available on the dictionary and the types of the values that can be stored in the dictionary. The presence of keys sometimes is used as a boolean as well.

The Xamarin.iOS binding generator provides support for developers to bind notifications. To do this, you set the `[Notification]` attribute on a property that has been also been tagged with a `[Field]` property (it can be public or private).

This attribute can be used without arguments for notifications that carry no payload, or you can specify a `System.Type` that references another interface in the API definition, typically with the name ending with "EventArgs". The generator will turn the interface into a class that subclasses `EventArgs` and will include all of the properties listed there. The `[Export]` attribute should be used in the EventArgs class to list the name of the key used to look up the Objective-C dictionary to fetch the value.

For example:

```
interface MyClass {
    [Notification]
    [Field ("MyClassDidStartNotification")]
    NSString DidStartNotification { get; }
}
```

The above code will generate a nested class `MyClass.Notifications` with the following methods:

```
public class MyClass {
    [...]
    public Notifications {
        public static NSObject ObserveDidStart (EventHandler<NSNotificationEventArgs> handler)
    }
}
```

Users of your code can then easily subscribe to notifications posted to the `NSNotificationCenter` by using code like this:

```

var token = MyClass.Notifications.ObserverDidStart ((notification) => {
    Console.WriteLine ("Observed the 'DidStart' event!");
});

```

The returned value from `ObserveDidStart` can be used to easily stop receiving notifications, like this:

```
token.Dispose ();
```

Or you can call `NSNotification.DefaultCenter.RemoveObserver` and pass the token. If your notification contains parameters, you should specify a helper `EventArgs` interface, like this:

```

interface MyClass {
    [Notification (typeof (MyScreenChangedEventArgs))]
    [Field ("MyClassScreenChangedNotification")]
    NSString ScreenChangedNotification { get; }

}

// The helper EventArgs declaration
interface MyScreenChangedEventArgs {
    [Export ("ScreenXKey")]
    nint ScreenX { get; set; }

    [Export ("ScreenYKey")]
    nint ScreenY { get; set; }

    [Export ("DidGoOffKey")]
    [ProbePresence]
    bool DidGoOff { get; }
}

```

The above will generate a `MyScreenChangedEventArgs` class with the `ScreenX` and `ScreenY` properties that will fetch the data from the `NSNotification.UserInfo` dictionary using the keys "ScreenXKey" and "ScreenYKey" respectively and apply the proper conversions. The `[ProbePresence]` attribute is used for the generator to probe if the key is set in the `UserInfo`, instead of trying to extract the value. This is used for cases where the presence of the key is the value (typically for boolean values).

This allows you to write code like this:

```

var token = MyClass.NotificationsObserveScreenChanged ((notification) => {
    Console.WriteLine ("The new screen dimensions are {0},{1}", notification.ScreenX, notification.ScreenY);
});

```

Binding categories

Categories are an Objective-C mechanism used to extend the set of methods and properties available in a class. In practice, they are used to either extend the functionality of a base class (for example `NSObject`) when a specific framework is linked in (for example `UIKit`), making their methods available, but only if the new framework is linked in. In some other cases, they are used to organize features in a class by functionality. They are similar in spirit to C# extension methods. This is what a category would look like in Objective-C:

```

@interface UIView (MyUIViewExtension)
-(void) makeBackgroundRed;
@end

```

The above example if found on a library would extend instances of `UIView` with the method `makeBackgroundRed`.

To bind those, you can use the `[Category]` attribute on an interface definition. When using the `[Category]` attribute, the meaning of the `[BaseType]` attribute changes from being used to specify the base class to extend, to be the type to extend.

The following shows how the `UIView` extensions are bound and turned into C# extension methods:

```
[BaseType (typeof (UIView))]
[Category]
interface MyUIViewExtension {
    [Export ("makeBackgroundRed")]
    void MakeBackgroundRed ();
}
```

The above will create a `MyUIViewExtension` a class that contains the `MakeBackgroundRed` extension method. This means that you can now call "MakeBackgroundRed" on any `UIView` subclass, giving you the same functionality you would get on Objective-C. In some other cases, categories are used not to extend a system class, but to organize functionality, purely for decoration purposes. Like this:

```
@interface SocialNetworking (Twitter)
- (void) postToTwitter:(Message *) message;
@end

@interface SocialNetworking (Facebook)
- (void) postToFacebook:(Message *) message andPicture: (UIImage*)
picture;
@end
```

Although you can use the `[Category]` attribute also for this decoration style of declarations, you might as well just add them all to the class definition. Both of these would achieve the same:

```
[BaseType (typeof (NSObject))]
interface SocialNetworking {

}

[Category]
[BaseType (typeof (SocialNetworking))]
interface Twitter {
    [Export ("postToTwitter:")]
    void PostToTwitter (Message message);
}

[Category]
[BaseType (typeof (SocialNetworking))]
interface Facebook {
    [Export ("postToFacebook:andPicture:")]
    void PostToFacebook (Message message, UIImage picture);
}
```

It is just shorter in these cases to merge the categories:

```
[BaseType (typeof (NSObject))]
interface SocialNetworking {
    [Export ("postToTwitter:")]
    void PostToTwitter (Message message);

    [Export ("postToFacebook:andPicture:")]
    void PostToFacebook (Message message, UIImage picture);
}
```

Binding blocks

Blocks are a new construct introduced by Apple to bring the functional equivalent of C# anonymous methods to Objective-C. For example, the `NSSet` class now exposes this method:

```
- (void) enumerateObjectsUsingBlock:(void (^)(id obj, BOOL *stop)) block
```

The above description declares a method called `enumerateObjectsUsingBlock:` that takes one argument named `block`. This block is similar to a C# anonymous method in that it has support for capturing the current environment (the "this" pointer, access to local variables and parameters). The above method in `NSSet` invokes the block with two parameters an `NSObject` (the `id obj` part) and a pointer to a boolean (the `BOOL *stop`) part.

To bind this kind of API with btouch, you need to first declare the block type signature as a C# delegate and then reference it from an API entry point, like this:

```
// This declares the callback signature for the block:  
delegate void NSSetEnumerator (NSObject obj, ref bool stop)  
  
// Later, inside your definition, do this:  
[Export ("enumerateObjectUsingBlock:")]  
void Enumerate (NSSetEnumerator enum)
```

And now your code can call your function from C#:

```
var myset = new NSMutableSet ();  
myset.Add (new NSString ("Foo"));  
  
s.Enumerate (delegate (NSObject obj, ref bool stop){  
    Console.WriteLine ("The first object is: {0} and stop is: {1}", obj, stop);  
});
```

You can also use lambdas if you prefer:

```
var myset = new NSMutableSet ();  
myset.Add (new NSString ("Foo"));  
  
s.Enumerate ((obj, stop) => {  
    Console.WriteLine ("The first object is: {0} and stop is: {1}", obj, stop);  
});
```

Asynchronous methods

The binding generator can turn a certain class of methods into async-friendly methods (methods that return a `Task` or `Task<T>`).

You can use the `[Async]` attribute on methods that return void and whose last argument is a callback. When you apply this to a method, the binding generator will generate a version of that method with the suffix `Async`. If the callback takes no parameters, the return value will be a `Task`, if the callback takes a parameter, the result will be a `Task<T>`. If the callback takes multiple parameters, you should set the `ResultType` or `ResultTypeName` to specify the desired name of the generated type which will hold all the properties.

Example:

```
[Export ("loadfile:completed:")]
[Async]
void LoadFile (string file, Action<string> completed);
```

The above code will generate both the LoadFile method, as well as:

```
[Export ("loadfile:completed:")]
Task<string> LoadFileAsync (string file);
```

Surfacing strong types for weak NSDictionary parameters

In many places in the Objective-C API, parameters are passed as weakly-typed `NSDictionary` APIs with specific keys and values, but these are error prone (you can pass invalid keys, and get no warnings; you can pass invalid values, and get no warnings) and frustrating to use as they require multiple trips to the documentation to lookup the possible key names and values.

The solution is to provide a strongly-typed version that provides the strongly-typed version of the API and behind the scenes maps the various underlying keys and values.

So for example, if the Objective-C API accepted an `NSDictionary` and it is documented as taking the key `XyzVolumeKey` which takes an `NSNumber` with a volume value from 0.0 to 1.0 and a `XyzCaptionKey` that takes a string, you would want your users to have a nice API that looks like this:

```
public class XyzOptions {
    public nullable float? Volume { get; set; }
    public string Caption { get; set; }
}
```

The `Volume` property is defined as nullable float, as the convention in Objective-C does not require these dictionaries to have the value, so there are scenarios where the value might not be set.

To do this, you need to do a few things:

- Create a strongly-typed class, that subclasses `DictionaryContainer` and provides the various getters and setters for each property.
- Declare overloads for the methods taking `NSDictionary` to take the new strongly-typed version.

You can create the strongly-typed class either manually, or use the generator to do the work for you. We first explore how to do this manually so you understand what is going on, and then the automatic approach.

You need to create a supporting file for this, it does not go into your contract API. This is what you would have to write to create your `XyzOptions` class:

```

public class XyzOptions : DictionaryContainer {
    # if !COREBUILD
        public XyzOptions () : base (new NSMutableDictionary ()) {}
        public XyzOptions (NSDictionary dictionary) : base (dictionary){}

        public nfloat? Volume {
            get { return GetFloatValue (XyzOptionsKeys.VolumeKey); }
            set { SetNumberValue (XyzOptionsKeys.VolumeKey, value); }
        }
        public string Caption {
            get { return GetStringValue (XyzOptionsKeys.CaptionKey); }
            set { SetStringValue (XyzOptionsKeys.CaptionKey, value); }
        }
    # endif
}

```

You then should provide a wrapper method that surfaces the high-level API, on top of the low-level API.

```

[BaseType (typeof (NSObject))]
interface XyzPanel {
    [Export ("playback:withOptions:")]
    void Playback (string fileName, [NullAllowed] NSDictionary options);

    [Wrap ("Playback (fileName, options == null ? null : options.Dictionary")]
    void Playback (string fileName, XyzOptions options);
}

```

If your API does not need to be overwritten, you can safely hide the NSDictionary-based API by using the [\[Internal\]](#) attribute.

As you can see, we use the [\[Wrap\]](#) attribute to surface a new API entry point, and we surface it using our strongly-typed `XyzOptions` class. The wrapper method also allows for null to be passed.

Now, one thing that we did not mention is where the `XyzOptionsKeys` values came from. You would typically group the keys that an API surfaces in a static class like `XyzOptionsKeys`, like this:

```

[Static]
class XyzOptionKeys {
    [Field ("kXyzVolumeKey")]
    NSString VolumeKey { get; }

    [Field ("kXyzCaptionKey")]
    NSString CaptionKey { get; }
}

```

Let us look at the automatic support for creating these strongly-typed dictionaries. This avoids plenty of the boilerplate, and you can define the dictionary directly in your API contract, instead of using an external file.

To create a strongly-typed dictionary, introduce an interface in your API and decorate it with the [StrongDictionary](#) attribute. This tells the generator that it should create a class with the same name as your interface that will derive from `DictionaryContainer` and will provide strong typed accessors for it.

The [\[StrongDictionary\]](#) attribute takes one parameter, which is the name of the static class that contains your dictionary keys. Then each property of the interface will become a strongly-typed accessor. By default, the code will use the name of the property with the suffix "Key" in the static class to create the accessor.

This means that creating your strongly-typed accessor no longer requires an external file, nor having to manually create getters and setters for every property, nor having to lookup the keys manually yourself.

This is what your entire binding would look like:

```
[Static]
class XyzOptionKeys {
    [Field ("kXyzVolumeKey")]
    NSString VolumeKey { get; }

    [Field ("kXyzCaptionKey")]
    NSString CaptionKey { get; }
}

[StrongDictionary ("XyzOptionKeys")]
interface XyzOptions {
    nfloat Volume { get; set; }
    string Caption { get; set; }
}

[BaseType (typeof (NSObject))]
interface XyzPanel {
    [Export ("playback:withOptions:")]
    void Playback (string fileName, [NullAllowed] NSDictionary options);

    [Wrap ("Playback (fileName, options == null ? null : options.Dictionary)")]
    void Playback (string fileName, XyzOptions options);
}
```

In case you need to reference in your `XyzOption` members a different field (that is not the name of the property with the suffix `key`), you can decorate the property with an `[Export]` attribute with the name that you want to use.

Type mappings

This section covers how Objective-C types are mapped to C# types.

Simple types

The following table shows how you should map types from the Objective-C and CocoaTouch world to the Xamarin.iOS world:

OBJECTIVE-C TYPE NAME	XAMARIN.IOS UNIFIED API TYPE
<code>BOOL</code> , <code>GLboolean</code>	<code>bool</code>
<code>NSInteger</code>	<code>nint</code>
<code>NSUInteger</code>	<code>nuint</code>
<code>CFTimeInterval</code> / <code>NSTimeInterval</code>	<code>double</code>
<code>NSString</code> (more on binding NSString)	<code>string</code>
<code>char *</code>	<code>string</code> (see also: [PlainString])
<code>CGRect</code>	<code>CGRect</code>
<code>CGPoint</code>	<code>CGPoint</code>

OBJECTIVE-C TYPE NAME	XAMARIN.IOS UNIFIED API TYPE
CGSize	CGSize
CGFloat , GLfloat	nfloat
CoreFoundation types (CF*)	CoreFoundation.CF*
GLint	nint
GLfloat	nfloat
Foundation types (NS*)	Foundation.NS*
id	Foundation.NSObject
NSGlyph	nint
NSSize	CGSize
NSTextAlignment	UITextAlignment
SEL	ObjCRuntime.Selector
dispatch_queue_t	CoreFoundation.DispatchQueue
CFTimeInterval	double
CFIndex	nint
NSGlyph	nuint

Arrays

The Xamarin.iOS runtime automatically takes care of converting C# arrays to `NSArray` and doing the conversion back, so for example the imaginary Objective-C method that returns an `NSArray` of `UIViews`:

```
// Get the peer views - untyped
- (NSArray *)getPeerViews ();

// Set the views for this container
- (void) setViews:(NSArray *) views
```

Is bound like this:

```
[Export ("getPeerViews")]
UIView [] GetPeerViews ();

[Export ("setViews:")]
void SetViews (UIView [] views);
```

The idea is to use a strongly-typed C# array as this will allow the IDE to provide proper code completion with the

actual type without forcing the user to guess, or look up the documentation to find out the actual type of the objects contained in the array.

In cases where you can not track down the actual most derived type contained in the array, you can use `NSObject []` as the return value.

Selectors

Selectors appear on the Objective-C API as the special type `SEL`. When binding a selector, you would map the type to `ObjCRuntime.Selector`. Typically selectors are exposed in an API with both an object, the target object, and a selector to invoke in the target object. Providing both of these basically corresponds to the C# delegate: something that encapsulates both the method to invoke as well as the object to invoke the method in.

This is what the binding looks like:

```
interface Button {
    [Export ("setTarget:selector:")]
    void SetTarget (NSObject target, Selector sel);
}
```

And this is how the method would typically be used in an application:

```
class DialogPrint : UIViewController {
    void HookPrintButton (Button b)
    {
        b.SetTarget (this, new Selector ("print"));
    }

    [Export ("print")]
    void ThePrintMethod ()
    {
        // This does the printing
    }
}
```

To make the binding nicer to C# developers, you typically will provide a method that takes an `NSAction` parameter, which allows C# delegates and lambdas to be used instead of the `Target+Selector`. To do this you would typically hide the `SetTarget` method by flagging it with an `[Internal]` attribute and then you would expose a new helper method, like this:

```
// API.cs
interface Button {
    [Export ("setTarget:selector:"), Internal]
    void SetTarget (NSObject target, Selector sel);
}

// Extensions.cs
public partial class Button {
    public void SetTarget (NSAction callback)
    {
        SetTarget (new NSActionDispatcher (callback), NSActionDispatcher.Selector);
    }
}
```

So now your user code can be written like this:

```

class DialogPrint : UIViewController {
    void HookPrintButton (Button b)
    {
        // First Style
        b.SetTarget (ThePrintMethod);

        // Lambda style
        b.SetTarget (() => { /* print here */ });
    }

    void ThePrintMethod ()
    {
        // This does the printing
    }
}

```

Strings

When you are binding a method that takes an `NSString`, you can replace that with a C# string type, both on return types and parameters.

The only case when you might want to use an `NSString` directly is when the string is used as a token. For more information about strings and `NSString`, read the [API Design on NSString](#) document.

In some rare cases, an API might expose a C-like string (`char *`) instead of an Objective-C string (`NSString *`). In those cases, you can annotate the parameter with the `[PlainString]` attribute.

out/ref parameters

Some APIs return values in their parameters, or pass parameters by reference.

Typically the signature looks like this:

```

- (void) something:(int) foo withError:(NSError **) retError
- (void) someString:(NSObject **) byref

```

The first example shows a common Objective-C idiom to return error codes, a pointer to an `NSError` pointer is passed, and upon return the value is set. The second method shows how an Objective-C method might take an object and modify its contents. This would be a pass by reference, rather than a pure output value.

Your binding would look like this:

```

[Export ("something:withError:")]
void Something (int foo, out NSError error);
[Export ("someString:")]
void SomeString (ref NSObject byref);

```

Memory management attributes

When you use the `[Export]` attribute and you are passing data that will be retained by the called method, you can specify the argument semantics by passing it as a second parameter, for example:

```
[Export ("method", ArgumentSemantic.Retain)]
```

The above would flag the value as having the "Retain" semantics. The semantics available are:

- Assign

- Copy
- Retain

Style guidelines

Using [Internal]

You can use the `[Internal]` attribute to hide a method from the public API. You might want to do this in cases where the exposed API is too low-level and you want to provide a high-level implementation in a separate file based on this method.

You can also use this when you run into limitations in the binding generator, for example some advanced scenarios might expose types that are not bound and you want to bind in your own way, and you want to wrap those types yourself in your own way.

Event handlers and callbacks

Objective-C classes typically broadcast notifications or request information by sending a message on a delegate class (Objective-C delegate).

This model, while fully supported and surfaced by Xamarin.iOS can sometimes be cumbersome. Xamarin.iOS exposes the C# event pattern and a method-callback system on the class that can be used in these situations. This allows code like this to run:

```
button.Clicked += delegate {
    Console.WriteLine ("I was clicked");
};
```

The binding generator is capable of reducing the amount of typing required to map the Objective-C pattern into the C# pattern.

Starting with Xamarin.iOS 1.4 it will be possible to also instruct the generator to produce bindings for a specific Objective-C delegates and expose the delegate as C# events and properties on the host type.

There are two classes involved in this process, the host class which will be the one that currently emits events and sends those into the `Delegate` or `WeakDelegate` and the actual delegate class.

Considering the following setup:

```
[BaseType (typeof (NSObject))]
interface MyClass {
    [Export ("delegate", ArgumentSemantic.Assign)][NullAllowed]
    NSObject WeakDelegate { get; set; }

    [Wrap ("WeakDelegate")][NullAllowed]
    MyClassDelegate Delegate { get; set; }
}

[BaseType (typeof (NSObject))]
interface MyClassDelegate {
    [Export ("loaded:bytes:")]
    void Loaded (MyClass sender, int bytes);
}
```

To wrap the class you must:

- In your host class, add to your `[BaseType]` declaration the type that is acting as its delegate and the C# name that you exposed. In our example above

those are `typeof (MyClassDelegate)` and `WeakDelegate` respectively.

- In your delegate class, on each method that has more than two parameters, you need to specify the type that you want to use for the automatically generated EventArgs class.

The binding generator is not limited to wrapping only a single event destination, it is possible that some Objective-C classes to emit messages to more than one delegate, so you will have to provide arrays to support this setup. Most setups will not need it, but the generator is ready to support those cases.

The resulting code will be:

```
[BaseType (typeof (NSObject),
    Delegates=new string [] {"WeakDelegate"},
    Events=new Type [] { typeof (MyClassDelegate) })]
interface MyClass {
    [Export ("delegate", ArgumentSemantic.Assign)][NullAllowed]
    NSObject WeakDelegate { get; set; }

    [Wrap ("WeakDelegate")][NullAllowed]
    MyClassDelegate Delegate { get; set; }
}

[BaseType (typeof (NSObject))]
interface MyClassDelegate {
    [Export ("loaded:bytes:"), EventArgs ("MyClassLoaded")]
    void Loaded (MyClass sender, int bytes);
}
```

The `EventArgs` is used to specify the name of the `EventArgs` class to be generated. You should use one per signature (in this example, the `EventArgs` will contain a `With` property of type `nint`).

With the definitions above, the generator will produce the following event in the generated MyClass:

```
public MyClassLoadedEventArgs : EventArgs {
    public MyClassLoadedEventArgs (nint bytes);
    public nint Bytes { get; set; }
}

public event EventHandler<MyClassLoadedEventArgs> Loaded {
    add; remove;
}
```

So you can now use the code like this:

```
MyClass c = new MyClass ();
c.Loaded += delegate (sender, args){
    Console.WriteLine ("Loaded event with {0} bytes", args.Bytes);
};
```

Callbacks are just like event invocations, the difference is that instead of having multiple potential subscribers (for example, multiple methods can hook into a `Clicked` event or a `DownloadFinished` event) callbacks can only have a single subscriber.

The process is identical, the only difference is that instead of exposing the name of the `EventArgs` class that will be generated, the `EventArgs` actually is used to name the resulting C# delegate name.

If the method in the delegate class returns a value, the binding generator will map this into a delegate method in the parent class instead of an event. In these cases you need to provide the default value that should be returned by the method if the user does not hook up to the delegate. You do this using the `[DefaultValue]` or

`[DefaultValueFromArgument]` attributes.

`[DefaultValue]` will hardcode a return value, while `[DefaultValueFromArgument]` is used to specify which input argument will be returned.

Enumerations and base types

You can also reference enumerations or base types that are not directly supported by the btouch interface definition system. To do this, put your enumerations and core types into a separate file and include this as part of one of the extra files that you provide to btouch.

Linking the dependencies

If you are binding APIs that are not part of your application, you need to make sure that your executable is linked against these libraries.

You need to inform Xamarin.iOS how to link your libraries, this can be done either by altering your build configuration to invoke the `mtouch` command with some extra build arguments that specify how to link with the new libraries using the `-gcc_flags` option, followed by a quoted string that contains all the extra libraries that are required for your program, like this:

```
-gcc_flags "-L${ProjectDir} -lMylibrary -force_load -lSystemLibrary -framework CFNetwork -ObjC"
```

The above example will link `libMyLibrary.a`, `libSystemLibrary.dylib` and the `CFNetwork` framework library into your final executable.

Or you can take advantage of the assembly-level `[LinkWithAttribute]`, that you can embed in your contract files (such as `AssemblyInfo.cs`). When you use the `[LinkWithAttribute]`, you will need to have your native library available at the time you make your binding, as this will embed the native library with your application. For example:

```
// Specify only the library name as a constructor argument and specify everything else with properties:  
[assembly: LinkWith ("libMyLibrary.a", LinkTarget = LinkTarget.ArmV6 | LinkTarget.ArmV7 |  
LinkTarget.Simulator, ForceLoad = true, IsCxx = true)]  
  
// Or you can specify library name *and* link target as constructor arguments:  
[assembly: LinkWith ("libMyLibrary.a", LinkTarget.ArmV6 | LinkTarget.ArmV7 | LinkTarget.Simulator, ForceLoad  
= true, IsCxx = true)]
```

You might be wondering, why do you need `-force_load` command, and the reason is that the `-ObjC` flag although it compiles the code in, it does not preserve the metadata required to support categories (the linker/compiler dead code elimination strips it) which you need at runtime for Xamarin.iOS.

Assisted references

Some transient objects like action sheets and alert boxes are cumbersome to keep track of for developers and the binding generator can help a little bit here.

For example if you had a class that showed a message and then generated a `Done` event, the traditional way of handling this would be:

```

class Demo {
    MessageBox box;

    void ShowError (string msg)
    {
        box = new MessageBox (msg);
        box.Done += { box = null; ... };
    }
}

```

In the above scenario the developer needs to keep the reference to the object himself and either leak or actively clear the reference for box on his own. While binding code, the generator supports keeping track of the reference for you and clear it when a special method is invoked, the above code would then become:

```

class Demo {
    void ShowError (string msg)
    {
        var box = new MessageBox (msg);
        box.Done += { ... };
    }
}

```

Notice how it is no longer necessary to keep the variable in an instance, that it works with a local variable and that it is not necessary to clear the reference when the object dies.

To take advantage of this, your class should have a Events property set in the [\[BaseType \]](#) declaration and also the [KeepUntilRef](#) variable set to the name of the method that is invoked when the object has completed its work, like this:

```

[BaseType (typeof (NSObject), KeepUntilRef="Dismiss"), Delegates=new string [] { "WeakDelegate" },
Events=new Type [] { typeof (SomeDelegate) } ]
class Demo {
    [Export ("show")]
    void Show (string message);
}

```

Inheriting protocols

As of Xamarin.iOS v3.2, we support inheriting from protocols that have been marked with the [\[Model \]](#) property. This is useful in certain API patterns, such as in [MapKit](#) where the [MKOverlay](#) protocol, inherits from the [MKAnnotation](#) protocol, and is adopted by a number of classes which inherit from [NSObject](#).

Historically we required copying the protocol to every implementation, but in these cases now we can have the [MKShape](#) class inherit from the [MKOverlay](#) protocol and it will generate all the required methods automatically.

Related links

- [Binding Sample](#)

Binding types reference guide

11/3/2020 • 57 minutes to read • [Edit Online](#)

This document describes the list of attributes that you can use to annotate your API contract files to drive the binding and the code generated

Xamarin.iOS and Xamarin.Mac API contracts are written in C# mostly as interface definitions that define the way that Objective-C code is surfaced to C#. The process involves a mix of interface declarations plus some basic type definitions that the API contract might require. For an introduction to binding types, see our companion guide [Binding Objective-C Libraries](#).

Type definitions

Syntax:

```
[ BaseType (typeof (BTYPENAME))
interface MyType : [Protocol1, Protocol2] {
    IntPtr Constructor (string foo);
}
```

Every interface in your contract definition that has the `[BaseType]` attribute declares the base type for the generated object. In the above declaration a `MyType` class C# type will be generated that binds to an Objective-C type called `MyType`.

If you specify any types after the typename (in the sample above `Protocol1` and `Protocol2`) using the interface inheritance syntax the contents of those interfaces will be inlined as if they had been part of the contract for `MyType`. The way that Xamarin.iOS surfaces that a type adopts a protocol is by inlining all of the methods and properties that were declared in the protocol into the type itself.

The following shows how the Objective-C declaration for `UITextField` would be defined in a Xamarin.iOS contract:

```
@interface UITextField : UIControl <UITextInput> {

}
```

Would be written like this as a C# API contract:

```
[BaseType (typeof (UIControl))]
interface UITextField : UITextInput {
```

You can control many other aspects of the code generation by applying other attributes to the interface as well as configuring the `[BaseType]` attribute.

Generating events

One feature of the Xamarin.iOS and Xamarin.Mac API design is that we map Objective-C delegate classes as C# events and callbacks. Users can choose in a per-instance basis whether they want to adopt the Objective-C programming pattern, by assigning to properties like `Delegate` an instance of a class that implements the various methods that the Objective-C runtime would call, or by choosing the C#-style events and properties.

Let us see one example of how to use the Objective-C model:

```
bool MakeDecision ()  
{  
    return true;  
}  
  
void Setup ()  
{  
    var scrollView = new UIScrollView (myRect);  
    scrollView.Delegate = new MyScrollViewDelegate ();  
    ...  
}  
  
class MyScrollViewDelegate : UIScrollViewDelegate {  
    public override void Scrolled (UIScrollView scrollView)  
    {  
        Console.WriteLine ("Scrolled");  
    }  
  
    public override bool ShouldScrollToTop (UIScrollView scrollView)  
    {  
        return MakeDecision ();  
    }  
}
```

In the above example, you can see that we have chosen to overwrite two methods, one a notification that a scrolling event has taken place, and the second that is a callback that should return a boolean value instructing the `scrollView` whether it should scroll to the top or not.

The C# model allows the user of your library to listen to notifications using the C# event syntax or the property syntax to hook up callbacks that are expected to return values.

This is how the C# code for the same feature looks like using lambdas:

```
void Setup ()  
{  
    var scrollView = new UIScrollView (myRect);  
    // Event connection, use += and multiple events can be connected  
    scrollView.Scrolled += (sender, eventArgs) { Console.WriteLine ("Scrolled"); }  
  
    // Property connection, use = only a single callback can be used  
    scrollView.ShouldScrollToTop = (sv) => MakeDecision ();  
}
```

Since events do not return values (they have a void return type) you can connect multiple copies. The `ShouldScrollToTop` is not an event, it is instead a property with the type `UIScrollViewCondition` which has this signature:

```
public delegate bool UIScrollViewCondition (UIScrollView scrollView);
```

It returns a `bool` value, in this case the lambda syntax allows us to just return the value from the `MakeDecision` function.

The binding generator supports generating events and properties that link a class like `UIScrollView` with its `UIScrollViewDelegate` (well call these the Model class), this is done by annotating your `[BaseType]` definition with the `Events` and `Delegates` parameters (described below). In addition to annotating the `[BaseType]` with those parameters it is necessary to inform the generator of a few more components.

For events that take more than one parameter (in Objective-C the convention is that the first parameter in a delegate class is the instance of the sender object) you must provide the name that you would like for the generated `EventArgs` class to be. This is done with the `[EventArgs]` attribute on the method declaration in your Model class. For example:

```
[BaseType (typeof (UINavigationControllerDelegate))]
[Model][Protocol]
public interface UIImagePickerControllerDelegate {
    [Export ("imagePickerController:didFinishPickingImage:editingInfo:"), EventArgs
    ("UIImagePickerControllerImagePicked")]
    void FinishedPickingImage (UIImagePickerController picker, UIImage image, NSDictionary editingInfo);
}
```

The above declaration will generate a `UIImagePickerControllerImagePickedEventArgs` class that derives from `EventArgs` and packs both parameters, the `UIImage` and the `NSDictionary`. The generator produces this:

```
public partial class UIImagePickerControllerImagePickedEventArgs : EventArgs {
    public UIImagePickerControllerImagePickedEventArgs (UIImage image, NSDictionary editingInfo);
    public UIImage Image { get; set; }
    public NSDictionary EditingInfo { get; set; }
}
```

It then exposes the following in the `UIImagePickerController` class:

```
public event EventHandler<UIImagePickerControllerImagePickedEventArgs> FinishedPickingImage { add; remove; }
```

Model methods that return a value are bound differently. Those require both a name for the generated C# delegate (the signature for the method) and also a default value to return in case the user does not provide an implementation himself. For example, the `ShouldScrollToTop` definition is this:

```
[BaseType (typeof (NSObject))]
[Model][Protocol]
public interface UIScrollViewDelegate {
    [Export ("scrollViewShouldScrollToTop:"), DelegateName ("UIScrollViewCondition"), DefaultValue ("true")]
    bool ShouldScrollToTop (UIScrollView scrollView);
}
```

The above will create a `UIScrollViewCondition` delegate with the signature that was shown above, and if the user does not provide an implementation, the return value will be true.

In addition to the `[DefaultValue]` attribute, you can also use the `[DefaultValueFromArgument]` attribute that directs the generator to return the value of the specified parameter in the call or the `[NoDefaultValue]` parameter that instructs the generator that there is no default value.

BaseTypeAttribute

Syntax:

```

public class BaseTypeAttribute : Attribute {
    public BaseTypeAttribute (Type t);

    // Properties
    public Type BaseType { get; set; }
    public string Name { get; set; }
    public Type [] Events { get; set; }
    public string [] Delegates { get; set; }
    public string KeepRefUntil { get; set; }
}

```

BaseType.Name

You use the `Name` property to control the name that this type will bind to in the Objective-C world. This is typically used to give the C# type a name that is compliant with the .NET Framework Design Guidelines, but which maps to a name in Objective-C that does not follow that convention.

Example, in the following case we map the Objective-C `NSURLConnection` type to `NSURLConnection`, as the .NET Framework Design Guidelines use "Url" instead of "URL":

```

[BaseType (typeof (NSObject), Name="NSURLConnection")]
interface NSURLConnection {
}

```

The specified name is used as the value for the generated `[Register]` attribute in the binding. If `Name` is not specified, the type's short name is used as the value for the `[Register]` attribute in the generated output.

BaseType.Events and BaseType.Delegates

These properties are used to drive the generation of C#-style events in the generated classes. They are used to link a given class with its Objective-C delegate class. You will encounter many cases where a class uses a delegate class to send notifications and events. For example a `BarcodeScanner` would have a companion `BarcodeScannerDelegate` class. The `BarcodeScanner` class would typically have a `Delegate` property that you would assign an instance of `BarcodeScannerDelegate` to, while this works, you might want to expose to your users a C#-like style event interface, and in those cases you would use the `Events` and `Delegates` properties of the `[BaseType]` attribute.

These properties are always set together and must have the same number of elements and be kept in sync. The `Delegates` array contains one string for each weakly-typed delegate that you want to wrap, and the `Events` array contains one type for each type that you want to associate with it.

```

[BaseType (typeof (NSObject),
    Delegates=new string [] { "WeakDelegate" },
    Events=new Type [] {typeof(UIAccelerometerDelegate)})]
public interface UIAccelerometer {
}

[BaseType (typeof (NSObject))]
[Model][Protocol]
public interface UIAccelerometerDelegate {
}

```

BaseType.KeepRefUntil

If you apply this attribute when new instances of this class are created, the instance of that object will be kept around until the method referenced by the `KeepRefUntil` has been invoked. This is useful to improve the usability of your APIs, when you do not want your user to keep a reference to an object around to use your code. The value of this property is the name of a method in the `Delegate` class, so you must use this in combination with the `Events` and `Delegates` properties as well.

The following example show how this is used by `[UIActionSheet]` in Xamarin.iOS:

```
[BaseType (typeof (NSObject), KeepRefUntil="Dismissed")]
[BaseType (typeof (UIView),
    KeepRefUntil="Dismissed",
    Delegates=new string [] { "WeakDelegate" },
    Events=new Type [] {typeof(UIActionSheetDelegate)})]
public interface UIActionSheet {

}

[BaseType (typeof (NSObject))]
[Model][Protocol]
public interface UIActionSheetDelegate {
    [Export ("actionSheet:didDismissWithButtonIndex:"), EventArgs ("UIButton")]
    void Dismissed (UIActionSheet actionSheet, nint buttonIndex);
}
```

DesignatedDefaultCtorAttribute

When this attribute is applied to the interface definition it will generate a `[DesignatedInitializer]` attribute on the default (generated) constructor, which maps to the `init` selector.

DisableDefaultCtorAttribute

When this attribute is applied to the interface definition it will prevent the generator from producing the default constructor.

Use this attribute when you need the object to be initialized with one of the other constructors in the class.

PrivateDefaultCtorAttribute

When this attribute is applied to the interface definition it will flag the default constructor as private. This means that you can still instantiate object of this class internally from your extension file, but it just wont be accessible to users of your class.

CategoryAttribute

Use this attribute on a type definition to bind Objective-C categories and to expose those as C# extension methods to mirror the way Objective-C exposes the functionality.

Categories are an Objective-C mechanism used to extend the set of methods and properties available in a class. In practice, they are used to either extend the functionality of a base class (for example `NSObject`) when a specific framework is linked in (for example `UIKit`), making their methods available, but only if the new framework is linked in. In some other cases, they are used to organize features in a class by functionality. They are similar in spirit to C# extension methods.

This is what a category would look like in Objective-C:

```
@interface UIView (MyUIViewExtension)
-(void) makeBackgroundRed;
@end
```

The above example is found on a library that would extend instances of `UIView` with the method `makeBackgroundRed`.

To bind those, you can use the `[Category]` attribute on an interface definition. When using the `[Category]` attribute, the meaning of the `[BaseType]` attribute changes from being used to specify the base class to extend, to being the type to extend.

The following shows how the `UIView` extensions are bound and turned into C# extension methods:

```
[BaseType (typeof (UIView))]
[Category]
interface MyUIViewExtension {
    [Export ("makeBackgroundRed")]
    void MakeBackgroundRed ();
}
```

The above will create a `MyUIViewExtension` a class that contains the `MakeBackgroundRed` extension method. This means that you can now call `MakeBackgroundRed` on any `UIView` subclass, giving you the same functionality you would get on Objective-C.

In some cases you will find **static** members inside categories like in the following example:

```
@interface FooObject (MyFooObjectExtension)
+ (BOOL)boolMethod:(NSRange *)range;
@end
```

This will lead to an **incorrect** Category C# interface definition:

```
[Category]
[BaseType (typeof (FooObject))]
interface FooObject_Extensions {

    // Incorrect Interface definition
    [Static]
    [Export ("boolMethod:")]
    bool BoolMethod (NSRange range);
}
```

This is incorrect because to use the `BoolMethod` extension you need an instance of `FooObject` but you are binding an ObjC **static** extension, this is a side effect due to the fact of how C# extension methods are implemented.

The only way to use the above definitions is by the following ugly code:

```
(null as FooObject).BoolMethod (range);
```

The recommendation to avoid this is to inline the `BoolMethod` definition inside the `FooObject` interface definition itself, this will allow you to call this extension like it is intended `FooObject.BoolMethod (range)`.

```
[BaseType (typeof (NSObject))]
interface FooObject {

    [Static]
    [Export ("boolMethod:")]
    bool BoolMethod (NSRange range);
}
```

We will issue a warning (BI1117) whenever we find a `[Static]` member inside a `[Category]` definition. If you really want to have `[Static]` members inside your `[Category]` definitions you can silence the warning by using `[Category (allowStaticMembers: true)]` or by decorating either your member or `[Category]` interface definition with `[Internal]`.

StaticAttribute

When this attribute is applied to a class it will just generate a static class, one that does not derive from `NSObject`, so the `[BaseType]` attribute is ignored. Static classes are used to host C public variables that you want to expose.

For example:

```
[Static]
interface CBAdvertisement {
    [Field ("CBAdvertisementDataServiceUUIDsKey")]
    NSString DataServiceUUIDsKey { get; }
```

Will generate a C# class with the following API:

```
public partial class CBAdvertisement {
    public static NSString DataServiceUUIDsKey { get; }
}
```

Protocol/Model definitions

Models are typically used by protocol implementation. They differ in that the runtime will only register with Objective-C the methods that actually have been overwritten. Otherwise, the method will not be registered.

This in general means that when you subclass a class that has been flagged with the `[ModelAttribute]`, you should not call the base method. Calling that method will throw the following exception:

`Foundation.You_Should_Not_Call_base_In_This_Method`. You are supposed to implement the entire behavior on your subclass for any methods you override.

AbstractAttribute

By default, members that are part of a protocol are not mandatory. This allows users to create a subclass of the `Model` object by merely deriving from the class in C# and overriding only the methods they care about.

Sometimes the Objective-C contract requires that the user provides an implementation for this method (those are flagged with the `@required` directive in Objective-C). In those cases, you should flag those methods with the `[Abstract]` attribute.

The `[Abstract]` attribute can be applied to either methods or properties and causes the generator to flag the generated member as abstract and the class to be an abstract class.

The following is taken from Xamarin.iOS:

```
[BaseType (typeof (NSObject))]
[Model][Protocol]
public interface UITableViewDataSource {
    [Export ("tableView:numberOfRowsInSection:")]
    [Abstract]
    nint RowsInSection (UITableView tableView, nint section);
}
```

DefaultValueAttribute

Specifies the default value to be returned by a model method if the user does not provide a method for this particular method in the Model object

Syntax:

```

public class DefaultValueAttribute : Attribute {
    public DefaultValueAttribute (object o);
    public object Default { get; set; }
}

```

For example, in the following imaginary delegate class for a `Camera` class, we provide a `ShouldUploadToServer` which would be exposed as a property on the `Camera` class. If the user of the `Camera` class does not explicitly set a the value to a lambda that can respond true or false, the default value return in this case would be false, the value that we specified in the `DefaultValue` attribute:

```

[BaseType (typeof (NSObject))]
[Model][Protocol]
interface CameraDelegate {
    [Export ("camera:shouldPromptForAction:"), DefaultValue (false)]
    bool ShouldUploadToServer (Camera camera, CameraAction action);
}

```

If the user sets a handler in the imaginary class, then this value would be ignored:

```

var camera = new Camera ();
camera.ShouldUploadToServer = (camera, action) => return SomeDecision ();

```

See also: [\[NoDefaultValue\]](#) , [\[DefaultValueFromArgument\]](#) .

DefaultValueFromArgumentAttribute

Syntax:

```

public class DefaultValueFromArgumentAttribute : Attribute {
    public DefaultValueFromArgumentAttribute (string argument);
    public string Argument { get; }
}

```

This attribute when provided on a method that returns a value on a model class will instruct the generator to return the value of the specified parameter if the user did not provide his own method or lambda.

Example:

```

[BaseType (typeof (NSObject))]
[Model][Protocol]
public interface NSAnimationDelegate {
    [Export ("animation:valueForProgress:"), DelegateName ("NSAnimationProgress"),
DefaultValueFromArgumentAttribute ("progress")]
    float ComputeAnimationCurve (NSAnimation animation, nfloat progress);
}

```

In the above case if the user of the `NSAnimation` class chose to use any of the C# events/properties, and did not set `NSAnimation.ComputeAnimationCurve` to a method or lambda, the return value would be the value passed in the progress parameter.

See also: [\[NoDefaultValue\]](#) , [\[DefaultValue\]](#)

IgnoredInDelegateAttribute

Sometimes it makes sense not to expose an event or delegate property from a Model class into the host class so adding this attribute will instruct the generator to avoid the generation of any method decorated with it.

```
[BaseType (typeof (UINavigationControllerDelegate))]
[Model][Protocol]
public interface UIImagePickerControllerDelegate {
    [Export ("imagePickerController:didFinishPickingImage:editingInfo:"), EventArgs
    ("UIImagePickerControllerPicked")]
    void FinishedPickingImage (UIImagePickerController picker, UIImage image, NSDictionary editingInfo);

    [Export ("imagePickerController:didFinishPickingImage:"), IgnoredInDelegate] // No event generated for
    this method
    void FinishedPickingImage (UIImagePickerController picker, UIImage image);
}
```

DelegateNameAttribute

This attribute is used in Model methods that return values to set the name of the delegate signature to use.

Example:

```
[BaseType (typeof (NSObject))]
[Model][Protocol]
public interface NSAnimationDelegate {
    [Export ("animation:valueForProgress:"), DelegateName ("NSAnimationProgress"),
    DefaultValueFromArgumentAttribute ("progress")]
    float ComputeAnimationCurve (NSAnimation animation, float progress);
}
```

With the above definition, the generator will produce the following public declaration:

```
public delegate float NSAnimationProgress (MonoMac.AppKit.NSAnimation animation, float progress);
```

DelegateApiNameAttribute

This attribute is used to allow the generator to change the name of the property generated in the host class. Sometimes it is useful when the name of the FooDelegate class method makes sense for the Delegate class, but would look odd in the host class as a property.

Also this is really useful (and needed) when you have two or more overload methods that makes sense to keep them named as is in the FooDelegate class but you want to expose them in the host class with a better given name.

Example:

```
[BaseType (typeof (NSObject))]
[Model][Protocol]
public interface NSAnimationDelegate {
    [Export ("animation:valueForProgress:"), DelegateApiName ("ComputeAnimationCurve"), DelegateName
    ("Func<NSAnimation, float, float>"), DefaultValueFromArgument ("progress")]
    float GetValueForProgress (NSAnimation animation, float progress);
}
```

With the above definition, the generator will produce the following public declaration in the host class:

```
public Func<NSAnimation, float, float> ComputeAnimationCurve { get; set; }
```

EventArgsAttribute

For events that take more than one parameter (in Objective-C the convention is that the first parameter in a delegate class is the instance of the sender object) you must provide the name that you would like for the

generated EventArgs class to be. This is done with the `[EventArgs]` attribute on the method declaration in your `Model` class.

For example:

```
[BaseType (typeof (UINavigationControllerDelegate))]
[Model][Protocol]
public interface UIImagePickerControllerDelegate {
    [Export ("imagePickerController:didFinishPickingImage:editingInfo:"), EventArgs
("UIImagePickerControllerImagePicked")]
    void FinishedPickingImage (UIImagePickerController picker, UIImage image, NSDictionary editingInfo);
}
```

The above declaration will generate a `UIImagePickerControllerImagePickedEventArgs` class that derives from EventArgs and packs both parameters, the `UIImage` and the `NSDictionary`. The generator produces this:

```
public partial class UIImagePickerControllerImagePickedEventArgs : EventArgs {
    public UIImagePickerControllerImagePickedEventArgs (UIImage image, NSDictionary editingInfo);
    public UIImage Image { get; set; }
    public NSDictionary EditingInfo { get; set; }
}
```

It then exposes the following in the `UIImagePickerController` class:

```
public event EventHandler<UIImagePickerControllerImagePickedEventArgs> FinishedPickingImage { add; remove; }
```

EventNameAttribute

This attribute is used to allow the generator to change the name of an event or property generated in the class. Sometimes it is useful when the name of the Model class method makes sense for the model class, but would look odd in the originating class as an event or property.

For example, the `UIWebView` uses the following bit from the `UIWebViewDelegate`:

```
[Export ("webViewDidFinishLoad:"), EventArgs ("UIWebView"), EventName ("LoadFinished")]
void LoadingFinished (UIWebView webView);
```

The above exposes `LoadingFinished` as the method in the `UIWebViewDelegate`, but `LoadFinished` as the event to hook up to in a `UIWebView`:

```
var webView = new UIWebView (...);
webView.LoadFinished += delegate { Console.WriteLine ("done!"); }
```

ModelAttribute

When you apply the `[Model]` attribute to a type definition in your contract API, the runtime will generate special code that will only surface invocations to methods in the class if the user has overwritten a method in the class. This attribute is typically applied to all APIs that wrap an Objective-C delegate class.

NoDefaultValueAttribute

Specifies that the method on the model does not provide a default return value.

This works with the Objective-C runtime by responding `false` to the Objective-C runtime request to determine if the specified selector is implemented in this class.

```
[BaseType (typeof (NSObject))]
[Model][Protocol]
interface CameraDelegate {
    [Export ("shouldDisplayPopup"), DefaultValue]
    bool ShouldUploadToServer ();
}
```

See also: [\[DefaultValue\]](#), [\[DefaultValueFromArgument\]](#)

Protocols

The Objective-C protocol concept does not really exist in C#. Protocols are similar to C# interfaces but they differ in that not all of the methods and properties declared in a protocol must be implemented by the class that adopts it. Instead some of the methods and properties are optional.

Some protocols are generally used as Model classes, those should be bound using the [\[Model\]](#) attribute.

```
[BaseType (typeof (NSObject))]
[Model, Protocol]
interface MyProtocol {
    // Use [Abstract] when the method is defined in the @required section
    // of the protocol definition in Objective-C
    [Abstract]
    [Export ("say")]
    void Say (string msg);

    [Export ("listen")]
    void Listen ();
}
```

Starting with Xamarin.iOS 7.0 a new and improved protocol binding functionality has been incorporated. Any definition that contains the [\[Protocol\]](#) attribute will actually generate three supporting classes that vastly improve the way that you consume protocols:

```
// Full method implementation, contains all methods
class MyProtocol : IMyProtocol {
    public void Say (string msg);
    public void Listen (string msg);
}

// Interface that contains only the required methods
interface IMyProtocol: INativeObject, IDisposable {
    [Export ("say")]
    void Say (string msg);
}

// Extension methods
static class IMyProtocol_Extensions {
    public static void Optional (this IMyProtocol this, string msg);
}
}
```

The **class implementation** provides a complete abstract class that you can override individual methods of and get full type safety. But due to C# not supporting multiple inheritance, there are scenarios where you might require a different base class, but still want to implement an interface.

This is where the generated **interface definition** comes in. It is an interface that has all the required methods from the protocol. This allows developers that want to implement your protocol to merely implement the

interface. The runtime will automatically register the type as adopting the protocol.

Notice that the interface only lists the required methods and does expose the optional methods. This means that classes that adopt the protocol will get full type checking for the required methods, but will have to resort to weak typing (manually using Export attributes and matching the signature) for the optional protocol methods.

To make it convenient to consume an API that uses protocols, the binding tool also will produce an extensions method class that exposes all of the optional methods. This means that as long as you are consuming an API, you will be able to treat protocols as having all the methods.

If you want to use the protocol definitions in your API, you will need to write skeleton empty interfaces in your API definition. If you want to use the MyProtocol in an API, you would need to do this:

```
[BaseType (typeof (NSObject))]
[Model, Protocol]
interface MyProtocol {
    // Use [Abstract] when the method is defined in the @required section
    // of the protocol definition in Objective-C
    [Abstract]
    [Export ("say")]
    void Say (string msg);

    [Export ("listen")]
    void Listen ();
}

interface IMyProtocol {}
```

```
[BaseType (typeof(NSObject))]
interface MyTool {
    [Export ("getProtocol")]
    IMyProtocol GetProtocol ();
}
```

The above is needed because at binding time the `IMyProtocol` would not exist, that is why you need to provide an empty interface.

Adopting protocol-generated interfaces

Whenever you implement one of the interfaces generated for the protocols, like this:

```
class MyDelegate : NSObject, IUITableViewDelegate {
    nint IUITableViewDelegate.GetRowHeight (nint row) {
        return 1;
    }
}
```

The implementation for the required interface methods gets exported with the proper name, so it is equivalent to this:

```
class MyDelegate : NSObject, IUITableViewDelegate {
    [Export ("getRowHeight")]
    nint IUITableViewDelegate.GetRowHeight (nint row) {
        return 1;
    }
}
```

This will work for all required protocol members, but there is a special case with optional selectors to be aware of.

Optional protocol members are treated identically when using the base class:

```
public class UrlSessionDelegate : NSUrlSessionDownloadDelegate {  
    public override void DidWriteData (NSUrlSession session, NSUrlSessionDownloadTask downloadTask, long bytesWritten, long totalBytesWritten, long totalBytesExpectedToWrite)
```

but when using the protocol interface it is required to add the [Export]. The IDE will add it via autocomplete when you add it starting with override.

```
public class UrlSessionDelegate : NSObject, INSUrlSessionDownloadDelegate {  
    [Export ("URLSession:downloadTask:didWriteData:totalBytesWritten:totalBytesExpectedToWrite:")]  
    public void DidWriteData (NSUrlSession session, NSUrlSessionDownloadTask downloadTask, long bytesWritten,  
    long totalBytesWritten, long totalBytesExpectedToWrite)
```

There is a slight behavior difference between the two at runtime:

- Users of the base class (NSUrlSessionDownloadDelegate in example) provides all required and optional selectors, returning reasonable default values.
- Users of the interface (INSUrlSessionDownloadDelegate in example) only respond to the exact selectors provided.

Some rare classes can behave differently here. In almost all cases however it is safe to use either.

Protocol inlining

While you bind existing Objective-C types that have been declared as adopting a protocol, you will want to inline the protocol directly. To do this, merely declare your protocol as an interface without any [\[BaseType\]](#) attribute and list the protocol in the list of base interfaces for your interface.

Example:

```
interface SpeakProtocol {  
    [Export ("say:")]  
    void Say (string msg);  
}  
  
[BaseType (typeof (NSObject))]  
interface Robot : SpeakProtocol {  
    [Export ("awake")]  
    bool Awake { get; set; }  
}
```

Member definitions

The attributes in this section are applied to individual members of a type: properties and method declarations.

AlignAttribute

Used to specify the alignment value for property return types. Certain properties take pointers to addresses that must be aligned at certain boundaries (in Xamarin.iOS this happens for example with some [GLKBaseEffect](#) properties that must be 16-byte aligned). You can use this property to decorate the getter, and use the alignment value. This is typically used with the [OpenTK.Vector4](#) and [OpenTK.Matrix4](#) types when integrated with Objective-C APIs.

Example:

```

public interface GLKBaseEffect {
    [Export ("constantColor")]
    Vector4 ConstantColor { [Align (16)] get; set; }
}

```

AppearanceAttribute

The `[Appearance]` attribute is limited to iOS 5, where the Appearance manager was introduced.

The `[Appearance]` attribute can be applied to any method or property that participate in the `UIAppearance` framework. When this attribute is applied to a method or property in a class, it will direct the binding generator to create a strongly-typed appearance class that is used to style all the instances of this class, or the instances that match certain criteria.

Example:

```

public interface UIToolbar {
    [Since (5,0)]
    [Export ("setBackgroundImage:forToolbarPosition:barMetrics:")]
    [Appearance]
    void SetBackgroundImage (UIImage backgroundImage, UIToolbarPosition position, UIBarMetrics barMetrics);

    [Since (5,0)]
    [Export ("backgroundImageForToolbarPosition:barMetrics:")]
    [Appearance]
    UIImage GetBackgroundImage (UIToolbarPosition position, UIBarMetrics barMetrics);
}

```

The above would generate the following code in `UIToolbar`:

```

public partial class UIToolbar {
    public partial class UIToolbarAppearance : UIView.UIViewAppearance {
        public virtual void SetBackgroundImage (UIImage backgroundImage, UIToolbarPosition position,
        UIBarMetrics barMetrics);
        public virtual UIImage GetBackgroundImage (UIToolbarPosition position, UIBarMetrics barMetrics)
    }
    public static new UIToolbarAppearance Appearance { get; }
    public static new UIToolbarAppearance AppearanceWhenContainedIn (params Type [] containers);
}

```

AutoReleaseAttribute (Xamarin.iOS 5.4)

Use the `[AutoReleaseAttribute]` on methods and properties to wrap the method invocation to the method in an `NSAutoReleasePool`.

In Objective-C there are some methods that return values that are added to the default `NSAutoReleasePool`. By default, these would go to your thread `NSAutoReleasePool`, but since Xamarin.iOS also keeps a reference to your objects as long as the managed object lives, you might not want to keep an extra reference in the `NSAutoReleasePool` which will only get drained until your thread returns control to the next thread, or you go back to the main loop.

This attribute is applied for example on heavy properties (for example `UIImage.FromFile()`) that returns objects that have been added to the default `NSAutoReleasePool`. Without this attribute, the images would be retained as long as your thread did not return control to the main loop. If your thread was some sort of background downloader that is always alive and waiting for work, the images would never be released.

ForcedTypeAttribute

The `[ForcedTypeAttribute]` is used to enforce the creation of a managed type even if the returned unmanaged

object does not match the type described in the binding definition.

This is useful when the type described in a header does not match the returned type of the native method, for example take the following Objective-C definition from `NSURLSession`:

```
- (NSURLSessionDownloadTask *)downloadTaskWithRequest:(NSURLRequest *)request
```

It clearly states that it will return an `NSURLSessionDownloadTask` instance, but yet it **returns** a `NSURLSessionTask`, which is a superclass and thus not convertible to `NSURLSessionDownloadTask`. Since we are in a type-safe context an `InvalidCastException` will happen.

To comply with the header description and avoid the `InvalidCastException`, the `[ForcedTypeAttribute]` is used.

```
[BaseType (typeof (NSObject), Name="NSURLSession")]
interface NSUrlSession {

    [Export ("downloadTaskWithRequest:")]
    [return: ForcedType]
    NSURLSessionDownloadTask CreateDownloadTask (NSURLRequest request);
}
```

The `[ForcedTypeAttribute]` also accepts a boolean value named `owns` that is `false` by default `[ForcedType (owns: true)]`. The `owns` parameter is used to follow the [Ownership Policy for Core Foundation objects](#).

The `[ForcedTypeAttribute]` is only valid on parameters, properties, and return value.

BindAsAttribute

The `[BindAsAttribute]` allows binding `NSNumber`, `NSValue` and `NSString` (enums) into more accurate C# types. The attribute can be used to create better, more accurate, .NET API over the native API.

You can decorate methods (on return value), parameters and properties with `BindAs`. The only restriction is that your member **must not** be inside a `[Protocol]` or `[Model]` interface.

For example:

```
[return: BindAs (typeof (bool?))]
[Export ("shouldDrawAt:")]
NSNumber ShouldDraw ([BindAs (typeof (CGRect))] NSValue rect);
```

Would output:

```
[Export ("shouldDrawAt:")]
bool? ShouldDraw (CGRect rect) { ... }
```

Internally we will do the `bool? <-> NSNumber` and `CGRect <-> NSValue` conversions.

The current supported encapsulation types are:

- `NSValue`
- `NSNumber`
- `NSString`

NSValue

The following C# data types are supported to be encapsulated from/into `NSValue`:

- `CGAffineTransform`

- NSRange
- CGVector
- SCNMatrix4
- CLLocationCoordinate2D
- SCNVector3
- SCNVector4
- CGPoint / PointF
- CGRect / RectangleF
- CGSize / SizeF
- UIEdgeInsets
- UIOffset
- MKCoordinateSpan
- CMTIMERange
- CMTIME
- CMTIMEMapping
- CATransform3D

NSNumber

The following C# data types are supported to be encapsulated from/into `NSNumber`:

- bool
- byte
- double
- float
- short
- int
- long
- sbyte
- ushort
- uint
- ulong
- nfloat
- nint
- nuint
- Enums

NSString

`[BindAs]` works in conjunction with [enums backed by a NSString constant](#) so you can create better .NET API, for example:

```
[BindAs (typeof (CAScroll))]
[Export ("supportedScrollMode")]
NSString SupportedScrollMode { get; set; }
```

Would output:

```
[Export ("supportedScrollMode")]
CAScroll SupportedScrollMode { get; set; }
```

We will handle the `enum <-> NSString` conversion only if the provided enum type to `[BindAs]` is [backed by a](#)

[NSString](#) constant.

Arrays

[\[BindAs\]](#) also supports arrays of any of the supported types, you can have the following API definition as an example:

```
[return: BindAs (typeof (CAScroll []))]
[Export ("getScrollModesAt:")]
NSString [] GetScrollModes ([BindAs (typeof (CGRect []))] NSValue [] rects);
```

Would output:

```
[Export ("getScrollModesAt")]
CAScroll? [] GetScrollModes (CGRect [] rects) { ... }
```

The `rects` parameter will be encapsulated into a `NSArray` that contains an `NSValue` for each `CGRect` and in return you will get an array of `CAScroll?` which has been created using the values of the returned `NSArray` containing `NSstrings`.

BindAttribute

The `[Bind]` attribute has two uses one when applied to a method or property declaration, and another one when applied to the individual getter or setter in a property.

When used for a method or property, the effect of the `[Bind]` attribute is to generate a method that invokes the specified selector. But the resulting generated method is not decorated with the `[Export]` attribute, which means that it can not participate in method overriding. This is typically used in combination with the `[Target]` attribute for implementing Objective-C extension methods.

For example:

```
public interface UIView {
    [Bind ("drawAtPoint:withFont:")]
   .SizeF DrawString ([Target] string str, CGPoint point, UIFont font);
}
```

When used in a getter or setter, the `[Bind]` attribute is used to alter the defaults inferred by the code generator when generating the getter and setter Objective-C selector names for a property. By default when you flag a property with the name `fooBar`, the generator would generate a `fooBar` export for the getter and `setFooBar:` for the setter. In a few cases, Objective-C does not follow this convention, usually they change the getter name to be `isFooBar`. You would use this attribute to inform the generator of this.

For example:

```
// Default behavior
[Export ("active")]
bool Active { get; set; }

// Custom naming with the Bind attribute
[Export ("visible")]
bool Visible { [Bind ("isVisible")] get; set; }
```

AsyncAttribute

Only available on Xamarin.iOS 6.3 and newer.

This attribute can be applied to methods that take a completion handler as their last argument.

You can use the `[Async]` attribute on methods whose last argument is a callback. When you apply this to a method, the binding generator will generate a version of that method with the suffix `Async`. If the callback takes no parameters, the return value will be a `Task`, if the callback takes a parameter, the result will be a `Task<T>`.

```
[Export ("upload:complete:")]
[Async]
void LoadFile (string file, NSAction complete)
```

The following will generate this async method:

```
Task LoadFileAsync (string file);
```

If the callback takes multiple parameters, you should set the `ResultType` or `ResultTypeName` to specify the desired name of the generated type which will hold all the properties.

```
delegate void OnComplete (string [] files, nint byteCount);

[Export ("upload:complete:")]
[Async (ResultTypeName="FileLoading")]
void LoadFiles (string file, OnComplete complete)
```

The following will generate this async method, where `FileLoading` contains properties to access both `files` and `byteCount`:

```
Task<FileLoading> LoadFile (string file);
```

If the last parameter of the callback is an `NSError`, then the generated `Async` method will check if the value is not null, and if that is the case, the generated async method will set the task exception.

```
[Export ("upload:onComplete:")]
[Async]
void Upload (string file, Action<string,NSError> onComplete);
```

The above generates the following async method:

```
Task<string> UploadAsync (string file);
```

And on error, the resulting Task will have the exception set to an `NSErrorException` that wraps the resulting `NSError`.

AsyncAttribute.ResultType

Use this property to specify the value for the returning `Task` object. This parameter takes an existing type, thus it needs to be defined in one of your core api definitions.

AsyncAttribute.ResultTypeName

Use this property to specify the value for the returning `Task` object. This parameter takes the name of your desired type name, the generator will produce a series of properties, one for each parameter that the callback takes.

AsyncAttribute.MethodName

Use this property to customize the name of the generated async methods. The default is to use the name of the

method and append the text "Async", you can use this to change this default.

DesignatedInitializerAttribute

When this attribute is applied to a constructor it will generate the same `[DesignatedInitializer]` in the final platform assembly. This is to help the IDE indicate which constructor should be used in subclasses.

This should map to Objective-C/clang use of `__attribute__((objc_designated_initializer))`.

DisableZeroCopyAttribute

This attribute is applied to string parameters or string properties and instructs the code generator to not use the zero-copy string marshaling for this parameter, and instead create a new `NSString` instance from the C# string. This attribute is only required on strings if you instruct the generator to use zero-copy string marshaling using either the `--zero-copy` command line option or setting the assembly-level attribute `ZeroCopyStringsAttribute`.

This is necessary in cases where the property is declared in Objective-C to be a `retain` or `assign` property instead of a `copy` property. These typically happen in third-party libraries that have been wrongly "optimized" by developers. In general, `retain` or `assign` `NSString` properties are incorrect since `NSMutableString` or user-derived classes of `NSString` might alter the contents of the strings without the knowledge of the library code, subtly breaking the application. Typically this happens due to premature optimization.

The following shows two such properties in Objective-C:

```
@property(nonatomic,retain) NSString *name;
@property(nonatomic,assign) NSString *name2;
```

DisposeAttribute

When you apply the `[DisposeAttribute]` to a class, you provide a code snippet that will be added to the `Dispose()` method implementation of the class.

Since the `Dispose` method is automatically generated by the `bmac-native` and `btouch-native` tools, you need to use the `[Dispose]` attribute to inject some code in the generated `Dispose` method implementation.

For example:

```
[BaseType (typeof (NSObject))]
[Dispose ("if (OpenConnections > 0) CloseAllConnections ();")]
interface DatabaseConnection {
}
```

ExportAttribute

The `[Export]` attribute is used to flag a method or property to be exposed to the Objective-C runtime. This attribute is shared between the binding tool and the actual Xamarin.iOS and Xamarin.Mac runtimes. For methods, the parameter is passed verbatim to the generated code, for properties, a getter and setter Exports are generated based on the base declaration (see the section on the [\[BindAttribute\]](#) for information on how to alter the behavior of the binding tool).

Syntax:

```

public enum ArgumentSemantic {
    None, Assign, Copy, Retain.
}

[AttributeUsage (AttributeTargets.Method | AttributeTargets.Constructor | AttributeTargets.Property)]
public class ExportAttribute : Attribute {
    public ExportAttribute();
    public ExportAttribute (string selector);
    public ExportAttribute (string selector, ArgumentSemantic semantic);
    public string Selector { get; set; }
    public ArgumentSemantic ArgumentSemantic { get; set; }
}

```

The `selector` represents the name of the underlying Objective-C method or property that is being bound.

ExportAttribute.ArgumentSemantic

FieldAttribute

This attribute is used to expose a C global variable as a field that is loaded on demand and exposed to C# code. Usually this is required to get the values of constants that are defined in C or Objective-C and that could be either tokens used in some APIs, or whose values are opaque and must be used as-is by user code.

Syntax:

```

public class FieldAttribute : Attribute {
    public FieldAttribute (string symbolName);
    public FieldAttribute (string symbolName, string libraryName);
    public string SymbolName { get; set; }
    public string LibraryName { get; set; }
}

```

The `symbolName` is the C symbol to link with. By default this will be loaded from a library whose name is inferred from the namespace where the type is defined. If this is not the library where the symbol is looked up, you should pass the `libraryName` parameter. If you're linking a static library, use `__Internal` as the `libraryName` parameter.

The generated properties are always static.

Properties flagged with the Field attribute can be of the following types:

- `NSString`
- `NSArray`
- `nint` / `int` / `long`
- `nuint` / `uint` / `ulong`
- `nfloat` / `float`
- `double`
- `CGSize`
- `System.IntPtr`
- Enums

Setters are not supported for `enums backed by NSString constants`, but they can be manually bound if needed.

Example:

```
[Static]
interface CameraEffects {
    [Field ("kCameraEffectsZoomFactorKey", "CameraLibrary")]
    NSString ZoomFactorKey { get; }
}
```

InternalAttribute

The `[Internal]` attribute can be applied to methods or properties and it has the effect of flagging the generated code with the `internal` C# keyword making the code only accessible to code in the generated assembly. This is typically used to hide APIs that are too low-level or provide a suboptimal public API that you want to improve upon or for APIs that are not supported by the generator and require some hand-coding.

When you design the binding, you would typically hide the method or property using this attribute and provide a different name for the method or property, and then on your C# complementary support file, you would add a strongly-typed wrapper that exposes the underlying functionality.

For example:

```
[Internal]
[Export ("setValue:forKey:")]
void _SetValueForKey (NSObject value, NSObject key);

[Internal]
[Export ("getValueForKey:")]
NSObject _GetValueForKey (NSObject key);
```

Then, in your supporting file, you could have some code like this:

```
public NSObject this [NSObject idx] {
    get {
        return _GetValueForKey (idx);
    }
    set {
        _SetValueForKey (value, idx);
    }
}
```

IsThreadStaticAttribute

This attribute flags the backing field for a property to be annotated with the .NET `[ThreadStatic]` attribute. This is useful if the field is a thread static variable.

MarshalNativeExceptions (Xamarin.iOS 6.0.6)

This attribute will make a method support native (Objective-C) exceptions. Instead of calling `objc_msgSend` directly, the invocation will go through a custom trampoline which catches ObjectiveC exceptions and marshals them into managed exceptions.

Currently only a few `objc_msgSend` signatures are supported (you will find out if a signature isn't supported when native linking of an app that uses the binding fails with a missing `monotouch_objc_msgSend` symbol), but more can be added at request.

NewAttribute

This attribute is applied to methods and properties to have the generator generate the `new` keyword in front of the declaration.

It is used to avoid compiler warnings when the same method or property name is introduced in a subclass that

already existed in a base class.

NotificationAttribute

You can apply this attribute to fields to have the generator produce a strongly-typed helper Notifications class.

This attribute can be used without arguments for notifications that carry no payload, or you can specify a `System.Type` that references another interface in the API definition, typically with the name ending with "EventArgs". The generator will turn the interface into a class that subclasses `EventArgs` and will include all of the properties listed there. The `[Export]` attribute should be used in the `EventArgs` class to list the name of the key used to look up the Objective-C dictionary to fetch the value.

For example:

```
interface MyClass {
    [Notification]
    [Field ("MyClassDidStartNotification")]
    NSString DidStartNotification { get; }
}
```

The above code will generate a nested class `MyClass.Notifications` with the following methods:

```
public class MyClass {
    [...]
    public Notifications {
        public static NSObject ObserveDidStart (EventHandler<NSNotificationEventArgs> handler)
        public static NSObject ObserveDidStart (NSObject objectToObserve,
        EventHandler<NSNotificationEventArgs> handler)
    }
}
```

Users of your code can then easily subscribe to notifications posted to the `NSNotificationCenter` by using code like this:

```
var token = MyClass.Notifications.ObserveDidStart ((notification) => {
    Console.WriteLine ("Observed the 'DidStart' event!");
});
```

Or to set a specific object to observe. If you pass `null` to `objectToObserve` this method will behave just like its other peer.

```
var token = MyClass.Notifications.ObserveDidStart (objectToObserve, (notification) => {
    Console.WriteLine ("Observed the 'DidStart' event on objectToObserve!");
});
```

The returned value from `ObserveDidStart` can be used to easily stop receiving notifications, like this:

```
token.Dispose ();
```

Or you can call `NSNotification.DefaultCenter.RemoveObserver` and pass the token. If your notification contains parameters, you should specify a helper `EventArgs` interface, like this:

```

interface MyClass {
    [Notification (typeof (MyScreenChangedEventArgs))]
    [Field ("MyClassScreenChangedNotification")]
    NSString ScreenChangedNotification { get; }
}

// The helper EventArgs declaration
interface MyScreenChangedEventArgs {
    [Export ("ScreenXKey")]
    nint ScreenX { get; set; }

    [Export ("ScreenYKey")]
    nint ScreenY { get; set; }

    [Export ("DidGoOffKey")]
    [ProbePresence]
    bool DidGoOff { get; }
}

```

The above will generate a `MyScreenChangedEventArgs` class with the `ScreenX` and `ScreenY` properties that will fetch the data from the `NSNotification.UserInfo` dictionary using the keys `ScreenXKey` and `ScreenYKey` respectively and apply the proper conversions. The `[ProbePresence]` attribute is used for the generator to probe if the key is set in the `UserInfo`, instead of trying to extract the value. This is used for cases where the presence of the key is the value (typically for boolean values).

This allows you to write code like this:

```

var token = MyClass.NotificationsObserveScreenChanged ((notification) => {
    Console.WriteLine ("The new screen dimensions are {0},{1}", notification.ScreenX, notification.ScreenY);
});

```

In some cases, there is no constant associated with the value passed on the dictionary. Apple sometimes uses public symbol constants and sometimes uses string constants. By default the `[Export]` attribute in your provided `EventArgs` class will use the specified name as a public symbol to be looked up at runtime. If this is not the case, and instead it is supposed to be looked up as a string constant then pass the `ArgumentSemantic.Assign` value to the Export attribute.

New in Xamarin.iOS 8.4

Sometimes, notifications will begin life without any arguments, so the use of `[Notification]` without arguments is acceptable. But sometimes, parameters to the notification will be introduced. To support this scenario, the attribute can be applied more than once.

If you are developing a binding, and you want to avoid breaking existing user code, you would turn an existing notification from:

```

interface MyClass {
    [Notification]
    [Field ("MyClassScreenChangedNotification")]
    NSString ScreenChangedNotification { get; }
}

```

Into a version that lists the notification attribute twice, like this:

```

interface MyClass {
    [Notification]
    [Notification (typeof (MyScreenChangedEventArgs))]
    [Field ("MyClassScreenChangedNotification")]
    NSString ScreenChangedNotification { get; }
}

```

NullAllowedAttribute

When this is applied to a property it flags the property as allowing the value `null` to be assigned to it. This is only valid for reference types.

When this is applied to a parameter in a method signature it indicates that the specified parameter can be null and that no check should be performed for passing `null` values.

If the reference type does not have this attribute, the binding tool will generate a check for the value being assigned before passing it to Objective-C and will generate a check that will throw an `ArgumentNullException` if the value assigned is `null`.

For example:

```

// In properties

[NullAllowed]
UIImage IconFile { get; set; }

// In methods
void SetImage ([NullAllowed] UIImage image, State forState);

```

OverrideAttribute

Use this attribute to instruct the binding generator that the binding for this particular method should be flagged with an `override` keyword.

PreSnippetAttribute

You can use this attribute to inject some code to be inserted after the input parameters have been validated, but before the code calls into Objective-C.

Example:

```

[Export ("demo")]
[PreSnippet ("var old = ViewController;")]
void Demo ();

```

PrologueSnippetAttribute

You can use this attribute to inject some code to be inserted before any of the parameters are validated in the generated method.

Example:

```

[Export ("demo")]
[Prologue ("Trace.Entry ();")]
void Demo ();

```

PostGetAttribute

Instructs the binding generator to invoke the specified property from this class to fetch a value from it.

This property is typically used to refresh the cache that points to reference objects that keep the object graph referenced. Usually it shows up in code that has operations like Add/Remove. This method is used so that after elements are added or removed that the internal cache be updated to ensure that we are keeping managed references to objects that are actually in use. This is possible because the binding tool generates a backing field for all reference objects in a given binding.

Example:

```
[BaseType (typeof (NSObject))]
[Since (4,0)]
public interface NSOperation {
    [Export ("addDependency:")]
    [PostGet ("Dependencies")]
    void AddDependency (NSOperation op);

    [Export ("removeDependency:")]
    [PostGet ("Dependencies")]
    void RemoveDependency (NSOperation op);

    [Export ("dependencies")]
    NSOperation [] Dependencies { get; }
}
```

In this case, the `Dependencies` property will be invoked after adding or removing dependencies from the `NSOperation` object, ensuring that we have a graph that represents the actual loaded objects, preventing both memory leaks as well as memory corruption.

PostSnippetAttribute

You can use this attribute to inject some C# source code to be inserted after the code has invoked the underlying Objective-C method

Example:

```
[Export ("demo")]
[PostSnippet ("if (old != null) old.DemoComplete ();")]
void Demo ();
```

ProxyAttribute

This attribute is applied to return values to flag them as being proxy objects. Some Objective-C APIs return proxy objects that can not be differentiated from user bindings. The effect of this attribute is to flag the object as being a `DirectBinding` object. For a scenario in Xamarin.Mac, you can see the [discussion on this bug](#).

RetainListAttribute

Instructs the generator to keep a managed reference to the parameter or remove an internal reference to the parameter. This is used to keep objects referenced.

Syntax:

```
public class RetainListAttribute: Attribute {
    public RetainListAttribute (bool doAdd, string listName);
}
```

If the value of `doAdd` is true, then the parameter is added to the `__mt_{0}_var List<NSObject>;`. Where `{0}` is replaced with the given `listName`. You must declare this backing field in your complementary partial class to the API.

For an example see [foundation.cs](#) and [NSNotificationCenter.cs](#)

ReleaseAttribute (Xamarin.iOS 6.0)

This can be applied to return types to indicate that the generator should call `[Release]` on the object before returning it. This is only needed when a method gives you a retained object (as opposed to an autoreleased object, which is the most common scenario)

Example:

```
[Export ("getAndRetainObject")]
[return: Release ()]
NSObject GetAndRetainObject ();
```

Additionally this attribute is propagated to the generated code, so that the Xamarin.iOS runtime knows it must retain the object upon returning to Objective-C from such a function.

SealedAttribute

Instructs the generator to flag the generated method as sealed. If this attribute is not specified, the default is to generate a virtual method (either a virtual method, an abstract method or an override depending on how other attributes are used).

StaticAttribute

When the `[Static]` attribute is applied to a method or property, this generates a static method or property. If this attribute is not specified, then the generator produces an instance method or property.

TransientAttribute

Use this attribute to flag properties whose values are transient, that is, objects that are created temporarily by iOS but are not long-lived. When this attribute is applied to a property, the generator does not create a backing field for this property, which means that the managed class does not keep a reference to the object.

WrapAttribute

In the design of the Xamarin.iOS/Xamarin.Mac bindings, the `[Wrap]` attribute is used to wrap a weakly-typed object with a strongly-typed object. This comes into play mostly with Objective-C delegate objects which are typically declared as being of type `id` or `NSObject`. The convention used by Xamarin.iOS and Xamarin.Mac is to expose those delegates or data sources as being of type `NSObject` and are named using the convention "Weak" + the name being exposed. An `id delegate` property from Objective-C would be exposed as an `NSObject WeakDelegate { get; set; }` property in the API contract file.

But typically the value that is assigned to this delegate is of a strong type, so we surface the strong type and apply the `[Wrap]` attribute, this means that users can choose to use weak types if they need some fine-control or if they need to resort to low-level tricks, or they can use the strongly-typed property for most of their work.

Example:

```
[BaseType (typeof (NSObject))]
interface Demo {
    [Export ("delegate"), NullAllowed]
    NSObject WeakDelegate { get; set; }

    [Wrap ("WeakDelegate")]
    DemoDelegate Delegate { get; set; }
}

[BaseType (typeof (NSObject))]
[Model][Protocol]
interface DemoDelegate {
    [Export ("doDemo")]
    void DoDemo ();
}
```

This is how the user would use the weakly-typed version of the Delegate:

```
// The weak case, user has to roll his own
class SomeObject : NSObject {
    [Export ("doDemo")]
    void CallbackForDoDemo () {}

}

var demo = new Demo ();
demo.WeakDelegate = new SomeObject ();
```

And this is how the user would use the strongly-typed version, notice that the user takes advantage of C#'s type system and is using the override keyword to declare his intent and that he does not have to manually decorate the method with `[Export]`, since we did that work in the binding for the user:

```
// This is the strong case,
class MyDelegate : DemoDelegate {
    override void Demo DoDemo () {}
}

var strongDemo = new Demo ();
demo.Delegate = new MyDelegate ();
```

Another use of the `[Wrap]` attribute is to support strongly-typed version of methods. For example:

```
[BaseType (typeof (NSObject))]
interface XyzPanel {
    [Export ("playback:withOptions:")]
    void Playback (string fileName, [NullAllowed] NSDictionary options);

    [Wrap ("Playback (fileName, options == null ? null : options.Dictionary)")]
    void Playback (string fileName, XyzOptions options);
}
```

When the `[Wrap]` attribute is applied on a method inside a type decorated with a `[Category]` attribute, you need to include `This` as the first argument since an extension method is being generated. For example:

```
[Wrap ("Write (This, image, options?.Dictionary, out error)")]
bool Write (CIIImage image, CIIImageRepresentationOptions options, out NSError error);
```

The members generated by `[Wrap]` are not `virtual` by default, if you need a `virtual` member you can set to

`true` the optional `isVirtual` parameter.

```
[BaseType (typeof (NSObject))]
interface FooExplorer {
    [Export ("fooWithContentsOfURL:")]
    void FromUrl (NSURL url);

    [Wrap ("FromUrl (NSURL.FromString (url))", isVirtual: true)]
    void FromUrl (string url);
}
```

`[Wrap]` can also be used directly in property getters and setters. This allows to have full control on them and adjust the code as needed. For example, consider the following API definition that uses smart enums:

```
// Smart enum.
enum PersonRelationship {
    [Field (null)]
    None,
    [Field ("FMFather", "__Internal")]
    Father,
    [Field ("FMMother", "__Internal")]
    Mother
}
```

Interface definition:

```
// Property definition.

[Export ("presenceType")]
NSString _PresenceType { get; set; }

PersonRelationship PresenceType {
    [Wrap ("PersonRelationshipExtensions.GetValue (_PresenceType)")]
    get;
    [Wrap ("_PresenceType = value.GetConstant ()")]
    set;
}
```

Parameter attributes

This section describes the attributes that you can apply to the parameters in a method definition as well as the `[NullAttribute]` that applies to a property as a whole.

BlockCallback

This attribute is applied to parameter types in C# delegate declarations to notify the binder that the parameter in question conforms to the Objective-C block calling convention and should marshal it in this way.

This is typically used for callbacks that are defined like this in Objective-C:

```
typedef returnType (^SomeTypeDefinition) (int parameter1, NSString *parameter2);
```

See also: [CCallback](#).

CCallback

This attribute is applied to parameter types in C# delegate declarations to notify the binder that the parameter in question conforms to the C ABI function pointer calling convention and should marshal it in this way.

This is typically used for callbacks that are defined like this in Objective-C:

```
typedef returnType (*SomeTypeDefinition) (int parameter1, NSString *parameter2);
```

See also: [BlockCallback](#).

Params

You can use the `[Params]` attribute on the last array parameter of a method definition to have the generator inject a "params" in the definition. This allows the binding to easily allow for optional parameters.

For example, the following definition:

```
[Export ("loadFiles")]
void LoadFiles ([Params]NSURL [] files);
```

Allows the following code to be written:

```
foo.LoadFiles (new NSURL (url));
foo.LoadFiles (new NSURL (url1), new NSURL (url2), new NSURL (url3));
```

This has the added advantage that it does not require users to create an array purely for passing elements.

PlainString

You can use the `[PlainString]` attribute in front of string parameters to instruct the binding generator to pass the string as a C string, instead of passing the parameter as an `NSString`.

Most Objective-C APIs consume `NSString` parameters, but a handful of APIs expose a `char *` API for passing strings, instead of the `NSString` variation. Use `[PlainString]` in those cases.

For example, the following Objective-C declarations:

```
- (void) setText: (NSString *) theText;
- (void) logMessage: (char *) message;
```

Should be bound like this:

```
[Export ("setText")]
void SetText (string theText);

[Export ("logMessage")]
void LogMessage ([PlainString] string theText);
```

RetainAttribute

Instructs the generator to keep a reference to the specified parameter. The generator will provide the backing store for this field or you can specify a name (the `WrapName`) to store the value at. This is useful to hold a reference to a managed object that is passed as a parameter to Objective-C and when you know that Objective-C will only keep this copy of the object. For instance, an API like `SetDisplay (SomeObject)` would use this attribute as it is likely that the SetDisplay could only display one object at a time. If you need to keep track of more than one object (for example, for a Stack-like API) you would use the `[RetainList]` attribute.

Syntax:

```
public class RetainAttribute {  
    public RetainAttribute ();  
    public RetainAttribute (string wrapName);  
    public string WrapName { get; }  
}
```

RetainListAttribute

Instructs the generator to keep a managed reference to the parameter or remove an internal reference to the parameter. This is used to keep objects referenced.

Syntax:

```
public class RetainListAttribute: Attribute {  
    public RetainListAttribute (bool doAdd, string listName);  
}
```

If the value of `doAdd` is true, then the parameter is added to the `__mt_{0}_var List<NSObject>`. Where `{0}` is replaced with the given `listName`. You must declare this backing field in your complementary partial class to the API.

For an example see [foundation.cs](#) and [NSNotificationCenter.cs](#)

TransientAttribute

This attribute is applied to parameters and is only used when transitioning from Objective-C to C#. During those transitions the various Objective-C `NSObject` parameters are wrapped into a managed representation of the object.

The runtime will take a reference to the native object and keep the reference until the last managed reference to the object is gone, and the GC has a chance to run.

In a few cases, it is important for the C# runtime to not keep a reference to the native object. This sometimes happens when the underlying native code has attached a special behavior to the lifecycle of the parameter. For example: the destructor for the parameter will perform some cleanup action, or dispose some precious resource.

This attribute informs the runtime that you desire the object to be disposed if possible when returning back to Objective-C from your overwritten method.

The rule is simple: if the runtime had to create a new managed representation from the native object, then at the end of the function, the retain count for the native object will be dropped, and the Handle property of the managed object will be cleared. This means that if you kept a reference to the managed object, that reference will become useless (invoking methods on it will throw an exception).

If the object passed was not created, or if there was already an outstanding managed representation of the object, the forced dispose does not take place.

Property attributes

NotImplementedAttribute

This attribute is used to support an Objective-C idiom where a property with a getter is introduced in a base class, and a mutable subclass introduces a setter.

Since C# does not support this model, the base class needs to have both the setter and the getter, and a subclass can use the [OverrideAttribute](#).

This attribute is only used in property setters, and is used to support the mutable idiom in Objective-C.

Example:

```
[BaseType (typeof (NSObject))]
interface MyString {
    [Export ("initWithValue:")]
    IntPtr Constructor (string value);

    [Export ("value")]
    string Value {
        get;
        [NotImplemented ("Not available on MyString, use MyMutableString to set")]
        set;
    }
}

[BaseType (typeof (MyString))]
interface MyMutableString {
    [Export ("value")]
    [Override]
    string Value { get; set; }
}
```

Enum attributes

Mapping `NSString` constants to enum values is a easy way to create better .NET API. It:

- allows code completion to be more useful, by showing **only** the correct values for the API;
- adds type safety, you cannot use another `NSString` constant in a incorrect context; and
- allows to hide some constants, making code completion show shorter API list without losing functionality.

Example:

```
enum NSRunLoopMode {

    [DefaultValue]
    [Field ("NSDefaultRunLoopMode")]
    Default,

    [Field ("NSRunLoopCommonModes")]
    Common,

    [Field (null)]
    Other = 1000
}
```

From the above binding definition the generator will create the `enum` itself and will also create a `*Extensions` static type that includes two-ways conversion methods between the enum values and the `NSString` constants. This means the constants remains available to developers even if they are not part of the API.

Examples:

```
// using the NSString constant in a different API / framework / 3rd party code
CallApiRequiringAnNSString (NSRunLoopMode.Default.GetConstant ());
```

```
// converting the constants from a different API / framework / 3rd party code
var constant = CallApiReturningAnNSString ();
// back into an enum value
CallApiWithEnum (NSRunLoopModeExtensions.GetValue (constant));
```

DefaultEnumValueAttribute

You can decorate **one** enum value with this attribute. This will become the constant being returned if the enum value is not known.

From the example above:

```
var x = (NSRunLoopMode) 99;
Call (x.GetConstant ()); // NSDefaultRunLoopMode will be used
```

If no enum value is decorated then a `NotSupportedException` will be thrown.

ErrorDomainAttribute

Error codes are bound as an enum values. There's generally an error domain for them and it's not always easy to find which one applies (or if one even exists).

You can use this attribute to associate the error domain with the enum itself.

Example:

```
[Native]
[ErrorDomain ("AVKitErrorDomain")]
public enum AVKitError : nint {
    None = 0,
    Unknown = -1000,
    PictureInPictureStartFailed = -1001
}
```

You can then call the extension method `GetDomain` to get the domain constant of any error.

FieldAttribute

This is the same `[Field]` attribute used for constants inside type. It can also be used inside enums to map a value with a specific constant.

A `null` value can be used to specify which enum value should be returned if a `null` `NSString` constant is specified.

From the example above:

```
var constant = NSRunLoopMode.NewInWatchOS3; // will be null in watchOS 2.x
Call (NSRunLoopModeExtensions.GetValue (constant)); // will return 1000
```

If no `null` value is present then an `ArgumentNullException` will be thrown.

Global attributes

Global attributes are either applied using the `[assembly:]` attribute modifier like the `[LinkWithAttribute]` or can be used anywhere, like the `[Lion]` and `[Since]` attributes.

LinkWithAttribute

This is an assembly-level attribute which allows developers to specify the linking flags required to reuse a bound

library without forcing the consumer of the library to manually configure the `gcc_flags` and extra `mtouch` arguments passed to a library.

Syntax:

```
// In properties
[Flags]
public enum LinkTarget {
    Simulator      = 1,
    ArmV6          = 2,
    ArmV7          = 4,
    Thumb          = 8,
}

[AttributeUsage(AttributeTargets.Assembly, AllowMultiple=true)]
public class LinkWithAttribute : Attribute {
    public LinkWithAttribute ();
    public LinkWithAttribute (string libraryName);
    public LinkWithAttribute (string libraryName, LinkTarget target);
    public LinkWithAttribute (string libraryName, LinkTarget target, string linkerFlags);
    public bool ForceLoad { get; set; }
    public string Frameworks { get; set; }
    public bool IsCxx { get; set; }
    public string LibraryName { get; set; }
    public string LinkerFlags { get; set; }
    public LinkTarget LinkTarget { get; set; }
    public bool NeedsGccExceptionHandling { get; set; }
    public bool SmartLink { get; set; }
    public string WeakFrameworks { get; set; }
}
```

This attribute is applied at the assembly level, for example, this is what the [CorePlot bindings](#) use:

```
[assembly: LinkWith ("libCorePlot-CocoaTouch.a", LinkTarget.ArmV7 | LinkTarget.ArmV7s |
LinkTarget.Simulator, Frameworks = "CoreGraphics QuartzCore", ForceLoad = true)]
```

When you use the `[LinkWith]` attribute, the specified `libraryName` is embedded into the resulting assembly, allowing users to ship a single DLL that contains both the unmanaged dependencies as well as the command line flags necessary to properly consume the library from Xamarin.iOS.

It's also possible to not provide a `libraryName`, in which case the `LinkWith` attribute can be used to only specify additional linker flags:

```
[assembly: LinkWith (LinkerFlags = "-lsqlite3")]
```

LinkWithAttribute constructors

These constructors allow you to specify the library to link with and embed into your resulting assembly, the supported targets that the library supports and any optional library flags that are necessary to link with the library.

Note that the `LinkTarget` argument is inferred by Xamarin.iOS and does not need to be set.

Examples:

```

// Specify additional linker:
[assembly: LinkWith (LinkerFlags = "-sqlite3")]

// Specify library name for the constructor:
[assembly: LinkWith ("libDemo.a");

// Specify library name, and link target for the constructor:
[assembly: LinkWith ("libDemo.a", LinkTarget.Thumb | LinkTarget.Simulator);

// Specify only the library name, link target and linker flags for the constructor:
[assembly: LinkWith ("libDemo.a", LinkTarget.Thumb | LinkTarget.Simulator, SmartLink = true, ForceLoad =
true, IsCxx = true)];

```

LinkWithAttribute.ForceLoad

The `ForceLoad` property is used to decide whether or not the `-force_load` link flag is used for linking the native library. For now, this should always be true.

LinkWithAttribute.Frameworks

If the library being bound has a hard requirement on any frameworks (other than `Foundation` and `UIKit`), you should set the `Frameworks` property to a string containing a space-delimited list of the required platform frameworks. For example, if you are binding a library that requires `CoreGraphics` and `CoreText`, you would set the `Frameworks` property to `"CoreGraphics CoreText"`.

LinkWithAttribute.IsCxx

Set this property to true if the resulting executable needs to be compiled using a C++ compiler instead of the default, which is a C compiler. Use this if the library that you are binding was written in C++.

LinkWithAttribute.LibraryName

The name of the unmanaged library to bundle. This is a file with the extension ".a" and it can contain object code for multiple platforms (for example, ARM and x86 for the simulator).

Earlier versions of Xamarin.iOS checked the `LinkTarget` property to determine the platform your library supported, but this is now auto-detected, and the `LinkTarget` property is ignored.

LinkWithAttribute.LinkerFlags

The `LinkerFlags` string provides a way for binding authors to specify any additional linker flags needed when linking the native library into the application.

For example, if the native library requires libxml2 and zlib, you would set the `LinkerFlags` string to `"-lxml2 -lz"`.

LinkWithAttribute.LinkTarget

Earlier versions of Xamarin.iOS checked the `LinkTarget` property to determine the platform your library supported, but this is now auto-detected, and the `LinkTarget` property is ignored.

LinkWithAttribute.NeedsGccExceptionHandling

Set this property to true if the library that you are linking requires the GCC Exception Handling library (`gcc_eh`)

LinkWithAttribute.SmartLink

The `SmartLink` property should be set to true to let Xamarin.iOS determine whether `ForceLoad` is required or not.

LinkWithAttribute.WeakFrameworks

The `WeakFrameworks` property works the same way as the `Frameworks` property, except that at link-time, the `-weak_framework` specifier is passed to gcc for each of the listed frameworks.

`WeakFrameworks` makes it possible for libraries and applications to weakly link against platform frameworks so that they can optionally use them if they are available but do not take a hard dependency on them which is useful if your library is meant to add extra features on newer versions of iOS. For more information on weak

linking, see Apple's documentation on [Weak Linking](#).

Good candidates for weak linking would be `Frameworks` like Accounts, CoreBluetooth, CoreImage, GLKit, NewsstandKit and Twitter since they are only available in iOS 5.

SinceAttribute (iOS) and LionAttribute (macOS)

You use the `[Since]` attribute to flag APIs as having been introduced at a certain point in time. The attribute should only be used to flag types and methods that could cause a runtime problem if the underlying class, method or property is not available.

Syntax:

```
public SinceAttribute : Attribute {  
    public SinceAttribute (byte major, byte minor);  
    public byte Major, Minor;  
}
```

It should in general not be applied to enumerations, constraints or new structures as those would not cause a runtime error if they are executed on a device with an older version of the operating system.

Example when applied to a type:

```
// Type introduced with iOS 4.2  
[Since (4,2)]  
[BaseType (typeof (UIPrintFormatter))]  
interface UIVIEWPrintFormatter {  
    [Export ("view")]  
    UIVIEW View { get; }  
}
```

Example when applied to a new member:

```
[BaseType (typeof (UIViewController))]  
public interface UITableViewcontroller {  
    [Export ("tableview", ArgumentSemantic.Retain)]  
    UITableView Tableview { get; set; }  
  
    [Since (3,2)]  
    [Export ("clearsSelectionOnViewWillAppear")]  
    bool ClearsSelectionOnViewWillAppear { get; set; }  
}
```

The `[Lion]` attribute is applied in the same way but for types introduced with Lion. The reason to use `[Lion]` versus the more specific version number that is used in iOS is that iOS is revised very often, while major OS X releases happen rarely and it is easier to remember the operating system by their codename than by their version number

AdviceAttribute

Use this attribute to give developers a hint about other APIs that might be more convenient for them to use. For example, if you provide a strongly-typed version of an API, you could use this attribute on the weakly-typed attribute to direct the developer to the better API.

The information from this attribute is shown in the documentation and tools can be developed to give user suggestions on how to improve

RequiresSuperAttribute

This is a specialized subclass of the `[Advice]` attribute that can be used to hint to the developer that overriding

a method **requires** a call to the base (overridden) method.

This corresponds to `clang __attribute__((objc_requires_super))`

ZeroCopyStringsAttribute

Only available in Xamarin.iOS 5.4 and newer.

This attribute instructs the generator that the binding for this specific library (if applied with `[assembly:]`) or type should use the fast zero-copy string marshaling. This attribute is equivalent to passing the command line option `--zero-copy` to the generator.

When using zero-copy for strings, the generator effectively uses the same C# string as the string that Objective-C consumes without incurring the creation of a new `NSString` object and avoiding copying the data from the C# strings to the Objective-C string. The only drawback of using Zero Copy strings is that you must ensure that any string property that you wrap that happens to be flagged as `retain` or `copy` has the `[DisableZeroCopy]` attribute set. This is required because the handle for zero-copy strings is allocated on the stack and is invalid upon the function return.

Example:

```
[ZeroCopyStrings]
[BaseType (typeof (NSObject))]
interface MyBinding {
    [Export ("name")]
    string Name { get; set; }

    [Export ("domain"), NullAllowed]
    string Domain { get; set; }

    [DisableZeroCopy]
    [Export ("someRetainedNSString")]
    string RetainedProperty { get; set; }
}
```

You can also apply the attribute at the assembly level, and it will apply to all the types of the assembly:

```
[assembly:ZeroCopyStrings]
```

Strongly-typed dictionaries

With Xamarin.iOS 8.0 we introduced support for easily creating strongly-typed classes that wrap `NSDictionary`.

While it has always been possible to use the `DictionaryContainer` data type together with a manual API, it is now a lot simpler to do this. For more information, see [Surfacing Strong Types](#).

StrongDictionary

When this attribute is applied to an interface, the generator will produce a class with the same name as the interface that derives from `DictionaryContainer` and turns each property defined in the interface into a strongly-typed getter and setter for the dictionary.

This automatically generates a class that can be instantiated from an existing `NSDictionary` or that has been created new.

This attribute takes one parameter, the name of the class containing the keys that are used to access the elements on the dictionary. By default each property in the interface with the attribute will lookup a member in

the specified type for a name with the suffix "Key".

For example:

```
[StrongDictionary ("MyOptionKeys")]
interface MyOption {
    string Name { get; set; }
    nint Age { get; set; }
}

[Static]
interface MyOptionKeys {
    // In Objective-C this is "NSString *MYOptionNameKey;"
    [Field ("MYOptionNameKey")]
    NSString NameKey { get; }

    // In Objective-C this is "NSString *MYOptionAgeKey;"
    [Field ("MYOptionAgeKey")]
    NSString AgeKey { get; }
}
```

In the above case, the `Myoption` class will produce a string property for `Name` that will use the `MyOptionKeys.NameKey` as the key into the dictionary to retrieve a string. And will use the `MyOptionKeys.AgeKey` as the key into the dictionary to retrieve an `NSNumber` which contains an int.

If you want to use a different key, you can use the export attribute on the property, for example:

```
[StrongDictionary ("MyColoringKeys")]
interface MyColoringOptions {
    [Export ("TheName")] // Override the default which would be NameKey
    string Name { get; set; }

    [Export ("TheAge")] // Override the default which would be AgeKey
    nint Age { get; set; }
}

[Static]
interface MyColoringKeys {
    // In Objective-C this is "NSString *MYColoringNameKey"
    [Field ("MYColoringNameKey")]
    NSString TheName { get; }

    // In Objective-C this is "NSString *MYColoringAgeKey"
    [Field ("MYColoringAgeKey")]
    NSString TheAge { get; }
}
```

Strong dictionary types

The following data types are supported in the `StrongDictionary` definition:

C# INTERFACE TYPE	NSDICTIONARY STORAGE TYPE
<code>bool</code>	<code>Boolean</code> stored in an <code>NSNumber</code>
Enumeration values	integer stored in an <code>NSNumber</code>
<code>int</code>	32-bit integer stored in an <code>NSNumber</code>
<code>uint</code>	32-bit unsigned integer stored in an <code>NSNumber</code>

C# INTERFACE TYPE	NSDICTIONARY STORAGE TYPE
nint	NSInteger stored in an NSNumber
nuint	NSUInteger stored in an NSNumber
long	64-bit integer stored in an NSNumber
float	32-bit integer stored as an NSNumber
double	64-bit integer stored as an NSNumber
NSObject and subclasses	NSObject
NSDictionary	NSDictionary
string	NSString
NSString	NSString
C# Array of NSObject	NSArray
C# Array of enumerations	NSArray containing NSNumber values

Creating Bindings with Objective Sharpie

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section provides an introduction to Objective Sharpie, Xamarin's command line tool used to automate the process of creating a binding to an Objective-C Library

- [Overview & History](#)
- [Getting Started](#)
- [Tools & Commands](#)
- [Features](#)
- [Examples](#)
- [Complete Walkthrough](#)
- [Release History](#)

Overview

Objective Sharpie is a command line tool to help bootstrap the first pass of a binding. It works by parsing the header files of a native library to map the public API into the [binding definition](#) (a process that previously was manually done).

Objective Sharpie uses Clang to parse header files, so the binding is as exact and thorough as possible. This can greatly reduce the time and effort it takes to produce a quality binding.

IMPORTANT

Objective Sharpie is a tool for experienced Xamarin developers with advanced knowledge of Objective-C (and by extension, C). Before attempting to bind an Objective-C library you should have solid knowledge of how to build the native library on the command line (and a good understanding of how the native library works).

History

We have been evolving and using the Objective Sharpie internally at Xamarin for the last three years. As a testament to the power of Objective Sharpie, APIs introduced in Xamarin.iOS and Xamarin.Mac since iOS 8, Mac OS X 10.10, and watchOS 2.0 were bootstrapped entirely with Objective Sharpie. Xamarin relies heavily on Objective Sharpie internally for building its own products.

However, Objective Sharpie is a very advanced tool that requires advanced knowledge of Objective-C and C, how to use the clang compiler on the command line, and generally how native libraries are put together. Because of this high bar, we felt that having a GUI wizard sets the wrong expectations, and as such, Objective Sharpie is currently only available as a command line tool.

Related Links

- [Objective Sharpie download](#)
- [Walkthrough: Binding an Objective-C Library](#)
- [Binding Objective-C Libraries](#)
- [Binding Details](#)
- [Binding Types Reference Guide](#)
- [Xamarin for Objective-C Developers](#)

Getting Started With Objective Sharpie

11/2/2020 • 2 minutes to read • [Edit Online](#)

IMPORTANT

Objective Sharpie is a tool for experienced Xamarin developers with advanced knowledge of Objective-C (and by extension, C). Before attempting to bind an Objective-C library you should have solid knowledge of how to build the native library on the command line (and a good understanding of how the native library works).

Installing Objective Sharpie

Objective Sharpie is currently a standalone command line tool for Mac OS X 10.10 and newer, and is *not a fully supported Xamarin product*. It should only be used by advanced developers to assist in creating a binding project to a 3rd party Objective-C library.

Objective Sharpie can be downloaded as a standard OS X package installer. Run the installer and follow all of the on-screen prompts from the installation wizard:

- **Current Version: 3.4**
 - [Download Latest Release](#)
 - [Forum Announcement](#)

TIP

Use the `sharpie update` command to update to the latest version.

Basic Walkthrough

Objective Sharpie is a command line tool provided by Xamarin that assists in creating the definitions required to bind a 3rd party Objective-C library to C#. Even when using Objective Sharpie, the developer *will* need to modify the generated files after Objective Sharpie finishes to address any issues that could not be automatically handled by the tool.

Where possible, Objective Sharpie will annotate APIs with which it has some doubt on how to properly bind (many constructs in the native code are ambiguous). These annotations will appear as [\[Verify\]](#) [attributes](#).

The output of Objective Sharpie is a pair of files - [ApiDefinition.cs](#) and [StructsAndEnums.cs](#) - that can be used to create a binding project which compiles into a library you can use in Xamarin apps.

IMPORTANT

Objective Sharpie comes with one **major** rule for proper usage: you must absolutely pass it the correct clang compiler command line arguments in order to ensure proper parsing. This is because the Objective Sharpie parsing phase is simply a tool [implemented against the clang libtooling API](#).

This means that Objective Sharpie has the full power of Clang (the C/Objective-C/C++ compiler that actually compiles the native library you would bind) and all of its internal knowledge of the header files for binding. Instead of translating the parsed [AST](#) to object code, Objective Sharpie translates the AST to a C# binding

"scaffold" suitable for input to the `bmac` and `btouch` Xamarin binding tools.

If Objective Sharpie errors out during parsing, it means that clang errored out during its parsing phase trying to construct the AST, and you need to figure out why.

NEW! version 3.0 attempts address some of this complexity by supporting Xcode projects directly. If a native library has a valid Xcode project, Objective Sharpie can evaluate the project for a specified target and configuration to deduce the necessary input header files and compiler flags.

If no Xcode project is available, you will need to be more familiar with the project by deducing the correct input header files, header file search paths, and other necessary compiler flags. It is important to realize that the compiler flags used to build the native library are the same that must be passed to Objective Sharpie. This is a more manual process, and one that does require a bit of familiarity with compiling native code on the command line with the Clang toolchain.

NEW! version 3.0 also introduces a tool for easily binding [CocoaPods](#) via the `sharpie pod` command. If the library you're interested in is available as a CocoaPod, we recommend you start by attempting to bind the CocoaPod with Objective Sharpie (versus attempting to bind against the source directly).

Objective Sharpie Tools & Commands

10/28/2019 • 2 minutes to read • [Edit Online](#)

Overview of the tools included with Objective Sharpie, and the command line arguments to use them.

Once Objective Sharpie is successfully [installed](#), open a terminal and familiarize yourself with the *commands* Objective Sharpie has to offer:

```
$ sharpie -help
usage: sharpie [OPTIONS] TOOL [TOOL_OPTIONS]

Options:
  -h, --help           Show detailed help
  -v, --version        Show version information

Telemetry Options:
  -tlm-about          Show a detailed overview of what usage and binding
                      information will be submitted to Xamarin by
                      default when using Objective Sharpie.
  -tlm-do-not-submit  Do not submit any usage or binding information to
                      Xamarin. Run 'sharpie -tlm-about' for more
                      information.
  -tlm-do-not-identify  Do not submit Xamarin account information when
                      submitting usage or binding information to Xamarin
                      for analysis. Binding attempts and usage data will
                      be submitted anonymously if this option is
                      specified.

Available Tools:
  xcode               Get information about Xcode installations and available SDKs.
  pod                 Create a Xamarin C# binding to Objective-C CocoaPods
  bind                Create a Xamarin C# binding to Objective-C APIs
  update              Update to the latest release of Objective Sharpie
  verify-docs         Show cross reference documentation for [Verify] attributes
  docs                Open the Objective Sharpie online documentation
```

Objective Sharpie provides the following tools:

TOOL	DESCRIPTION
xcode	Provides information about the current Xcode installation and the versions of iOS and Mac SDKs that are available. We will be using this information later when we generate our bindings.
pod	Searches for, configures, installs (in a local directory), and binds Objective-C CocoaPod libraries available from the master Spec repository. This tool evaluates the installed CocoaPod to automatically deduce the correct input to pass to the <code>bind</code> tool below. New in 3.0!
bind	Parses the header files (<code>*.h</code>) in the Objective-C library into the initial ApiDefinition.cs and StructsAndEnums.cs files.
update	Checks for newer versions of Objective Sharpie and downloads and launches the installer if one is available.

TOOL	DESCRIPTION
verify-docs	Shows detailed information about [Verify] attributes.
docs	Navigates to this document in your default web browser.

To get help on a specific Objective Sharpie tool, enter the name of the tool and the `-help` option. For example, `sharpie xcode -help` returns the following output:

```
$ sharpie xcode -help
usage: sharpie xcode [OPTIONS]

Options:
  -h, -help      Show detailed help
  -v, -verbose   Be verbose with output

Xcode Options:
  -sdks          List all available Xcode SDKs. Pass -verbose for more details.
```

Before we can start the binding process, we need to get information about our current installed SDKs by entering the following command into the Terminal `sharpie xcode -sdks`. Your output may differ depending on which version(s) of Xcode you have installed. Objective Sharpie looks for SDKs installed in any `Xcode*.app` under the `/Applications` directory:

```
$ sharpie xcode -sdks
sdk: appletvos9.0  arch: arm64
sdk: iphoneos9.1   arch: arm64  armv7
sdk: iphoneos9.0   arch: arm64  armv7
sdk: iphoneos8.4   arch: arm64  armv7
sdk: macosx10.11   arch: x86_64 i386
sdk: macosx10.10   arch: x86_64 i386
sdk: watchos2.0    arch: armv7
```

From the above, we can see that we have the `iphoneos9.1` SDK installed on our machine and it has `arm64` architecture support. We will be using this value for all the samples in this section. With this information in place, we are ready to parse an Objective-C library header files into the initial `ApiDefinition.cs` and `StructsAndEnums.cs` for the Binding project.

Objective Sharpie Features

10/28/2019 • 2 minutes to read • [Edit Online](#)

Read through these pages to better understand Objective Sharpie's features:

[ApiDefinitions.cs & StructsAndEnums.cs](#)

These two files are emitted by Objective Sharpie, to be included in your binding project. Learn more about them [here](#).

[Native Frameworks](#)

Some libraries are distributed as frameworks rather than as source. Objective Sharpie lets you use these libraries with the `-framework` option.

[Verify](#)

Objective Sharpie adds `Verify` attributes to signal that you need to manually inspect and update the generated binding.

ApiDefinitions & StructsAndEnums Files

11/2/2020 • 2 minutes to read • [Edit Online](#)

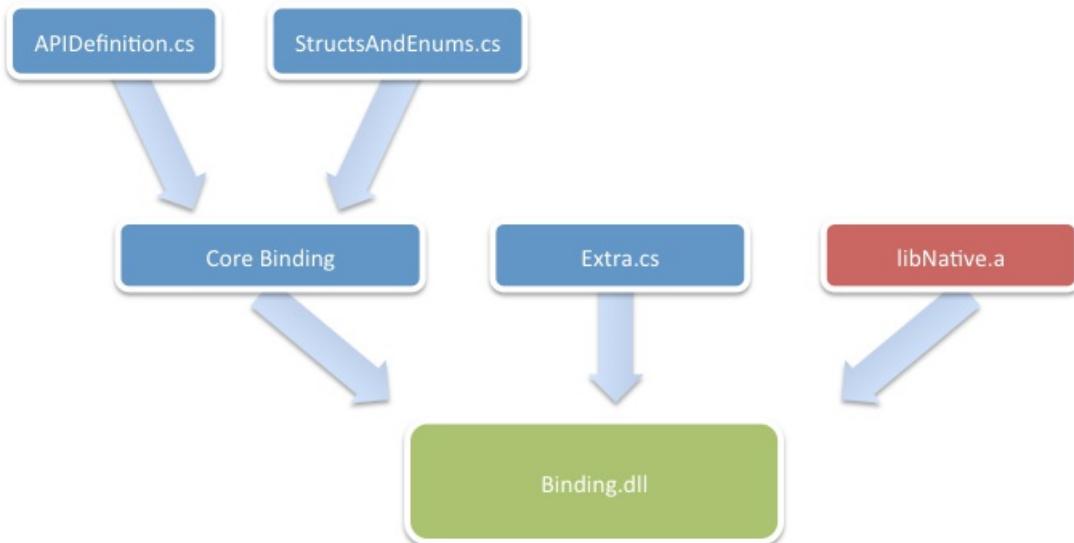
When Objective Sharpie has run successfully, it generates `Binding/ApiDefinitions.cs` and `Binding/StructsAndEnums.cs` files. These two files are added to a binding project in Visual Studio for Mac or passed directly to the `btouch` or `bmac` tools to produce the final binding.

In *some* cases these generated files might be all you need, however more often the developer will need to manually modify these generated files to fix any issues that could not be automatically handled by the tool (such as those flagged with a `Verify` attribute).

Some of the next steps include:

- **Adjusting Names:** Sometimes you will want to adjust the names of methods and classes to match the .NET Framework Design Guidelines.
- **Methods or Properties:** The heuristics used by Objective Sharpie sometimes will pick a method to be turned into a property. At this point, you could decide whether this is the intended behavior or not.
- **Hook up Events:** You could link your classes with your delegate classes and automatically generate events for those.
- **Hook up Notifications:** It is not possible to extract the API contract of notifications from the pure header files, this will require a trip to the API documentation. If you want strongly typed notifications, you will need to update the result.
- **API Curation:** At this point, you might choose to provide extra constructors, add methods (to allow for C# initialize-on-construction syntax), operator overloading and implement your own interfaces on the extra definitions file.

See the [binding an API](#) description to see how these files fit into the binding process, as shown in the diagram below:



Refer to the [binding Types reference](#) for more information on the contents of these files.

Binding Native Frameworks

2/13/2020 • 2 minutes to read • [Edit Online](#)

Sometimes a native library is distributed as a **framework**. Objective Sharpie provides a convenience feature for binding properly defined frameworks through the `-framework` option.

For example, binding the [Adobe Creative SDK Framework](#) for iOS is straightforward:

```
$ sharpie bind \
  -framework ./AdobeCreativeSDKFoundation.framework \
  -sdk iphoneos8.1
```

In some cases, a framework will specify an **Info.plist** which indicates against which SDK the framework should be compiled. If this information exists and no explicit `-sdk` option is passed, Objective Sharpie will infer it from the framework's **Info.plist** (either the `DTSDKName` key or a combination of the `DTPPlatformName` and `DTPPlatformVersion` keys).

The `-framework` option does not allow explicit header files to be passed. The umbrella header file is chosen by convention based on the framework name. If an umbrella header cannot be found, Objective Sharpie will not attempt to bind the framework and you must manually perform the binding by providing the correct umbrella header file(s) to parse, along with any framework arguments for clang (such as the `-F` framework search path option).

Under the hood, specifying `-framework` is just a shortcut. The following bind arguments are identical to the `-framework` shorthand above. Of special importance is the `-F .` framework search path provided to clang (note the space and period, which are required as part of the command).

```
$ sharpie bind \
  -sdk iphoneos8.1 \
  ./AdobeCreativeSDKFoundation.framework/Headers/AdobeCreativeSDKFoundation.h \
  -scope AdobeCreativeSDKFoundation.framework/Headers \
  -c -F .
```

Objective Sharpie Verify Attributes

1/31/2020 • 2 minutes to read • [Edit Online](#)

You will often find that bindings produced by Objective Sharpie will be annotated with the `[Verify]` attribute. These attributes indicate that you should *verify* that Objective Sharpie did the correct thing by comparing the binding with the original C/Objective-C declaration (which will be provided in a comment above the bound declaration).

Verification is recommended for *all* bound declarations, but is most likely *required* for declarations annotated with the `[Verify]` attribute. This is because in many situations, there is not enough metadata in the original native source code to infer how to best produce a binding. You may need to reference documentation or code comments inside the header files to make the best binding decision.

Once you have verified that the binding is correct or have fixed it to be correct, *remove* the `[Verify]` attribute from the binding.

IMPORTANT

`[Verify]` attributes intentionally cause C# compilation errors so that you are forced to verify the binding. You should remove the `[Verify]` attribute when you have reviewed (and possibly corrected) the code.

Verify Hints Reference

The hint argument supplied to the attribute can be cross referenced with documentation below. Documentation for any produced `[Verify]` attributes will be provided on the console as well after the binding has completed.

<code>[VERIFY]</code> HINT	DESCRIPTION
InferredFromPreceedingTypedef	The name of this declaration was inferred by common convention from the immediately preceding <code>typedef</code> in the original native source code. Verify that the inferred name is correct as this convention is ambiguous.
ConstantsInterfaceAssociation	There's no fool-proof way to determine with which Objective-C interface an extern variable declaration may be associated. Instances of these are bound as <code>[Field]</code> properties in a partial interface into a near-by concrete interface to produce a more intuitive API, possibly eliminating the 'Constants' interface altogether.
MethodToProperty	An Objective-C method was bound as a C# property due to convention such as taking no parameters and returning a value (non-void return). Often methods like these should be bound as properties to surface a nicer API, but sometimes false-positives can occur and the binding should actually be a method.

[VERIFY] HINT	DESCRIPTION
StronglyTypedNSArray	<p>A native <code>NSArray*</code> was bound as <code>NSObject[]</code>. It might be possible to more strongly type the array in the binding based on expectations set through API documentation (e.g. comments in the header file) or by examining the array contents through testing. For example, an <code>NSArray*</code> containing only <code>NSNumber*</code> instances can be bound as <code>NSNumber[]</code> instead of <code>NSObject[]</code>.</p>

You can also quickly receive documentation for a hint using the `sharpie verify-docs` tool, for example:

```
sharpie verify-docs InferredFromPreceedingTypedef
```

Objective Sharpie Examples

10/28/2019 • 2 minutes to read • [Edit Online](#)

Follow these examples to better understand how Objective Sharpie works:

- [Xcode](#)
- [Cocoapod](#)
- [Advanced \(manual\)](#)

The [complete walkthrough for iOS](#) provides a more detailed step-by-step set of instructions to follow.

Completed iOS Bindings

A number of completed bindings can be viewed or downloaded from the [monotouch-bindings Github repository](#).

Real-World Example using an Xcode Project

10/28/2019 • 2 minutes to read • [Edit Online](#)

This example uses the [POP library from Facebook](#).

New in version 3.0, Objective Sharpie supports Xcode projects as input. These projects specify the correct header files and compiler flags necessary to compile the native library, and thus necessary to bind it too. Objective Sharpie will select the first *target* and its default configuration of a project if not instructed to do otherwise.

Before Objective Sharpie attempts to parse the project and header files, it must build it. Projects often have build phases that will correctly structure header files for external consumption and integration, so it is best to always build the full project before attempting to bind it.

```
$ git clone https://github.com/facebook/pop.git
Cloning into 'pop'...
(more git clone output)

$ cd pop
$ sharpie bind pop.xcodeproj -sdk iphoneos9.0
```

Real-world example using CocoaPods

10/28/2019 • 2 minutes to read • [Edit Online](#)

NOTE

This example uses the [AFNetworking CocoaPod](#).

New in version 3.0, Objective Sharpie supports binding CocoaPods, and even includes a command (`sharpie pod`) to make downloading, configuring, and building CocoaPods very easy. You should [familiarize yourself with CocoaPods](#) in general before using this feature.

Creating a binding for a CocoaPod

The `sharpie pod` command has one global option and two subcommands:

```
$ sharpie pod -help
usage: sharpie pod [OPTIONS] COMMAND [COMMAND_OPTIONS]

Pod Options:
  -d, --dir DIR      Use DIR as the CocoaPods binding directory,
                      defaulting to the current directory

Available Commands:
  init      Initialize a new Xamarin C# CocoaPods binding project
  bind      Bind an existing Xamarin C# CocoaPods project
```

The `init` subcommand also has some useful help:

```
$ sharpie pod init -help
usage: sharpie pod init [INIT_OPTIONS] TARGET_SDK POD_SPEC_NAMES

Init Options:
  -f, --force      Initialize a new Podfile and run actions against
                   it even if one already exists
```

Multiple CocoaPod names and subspec names can be provided to `init`.

```
$ sharpie pod init ios AFNetworking
** Setting up CocoaPods master repo ...
  (this may take a while the first time)
** Searching for requested CocoaPods ...
** Working directory:
  ** - Writing Podfile ...
  ** - Installing CocoaPods ...
  **   (running `pod install --no-integrate --no-repo-update`)
Analyzing dependencies
Downloading dependencies
Installing AFNetworking (2.6.0)
Generating Pods project
Sending stats
** Success! You can now use other `sharpie podn` commands.
```

Once your CocoaPod has been set up, you can now create the binding:

```
$ sharpie pod bind
```

This will result in the CocoaPod Xcode project being built and then evaluated and parsed by Objective Sharpie. A lot of console output will be generated, but should result in the binding definition at the end:

```
(... lots of build output ...)

Parsing 19 header files...

Binding...
[write] ApiDefinitions.cs
[write] StructsAndEnums.cs

Done.
```

Next steps

After generating the `ApiDefinitions.cs` and `StructsAndEnums.cs` files, take a look at the following documentation to generate an assembly to use in your apps:

- [Binding Objective-C overview](#)
- [Binding Objective-C libraries](#)
- [Walkthrough: Binding an iOS Objective-C library](#)

Advanced (manual) Real-World Example

3/10/2020 • 5 minutes to read • [Edit Online](#)

This example uses the [POP library from Facebook](#).

This section covers a more advanced approach to binding, where we will use Apple's `xcodebuild` tool to first build the POP project, and then manually deduce input for Objective Sharpie. This essentially covers what Objective Sharpie is doing under the hood in the previous section.

```
$ git clone https://github.com/facebook/pop.git
Cloning into 'pop'...
_(more git clone output)_

$ cd pop
```

Because the POP library has an Xcode project (`pop.xcodeproj`), we can just use `xcodebuild` to build POP. This process may in turn generate header files that Objective Sharpie may need to parse. This is why building before binding is important. When building via `xcodebuild` ensure you pass the same SDK identifier and architecture that you intend to pass to Objective Sharpie (and remember, Objective Sharpie 3.0 can usually do this for you!):

```
$ xcodebuild -sdk iphoneos9.0 -arch arm64

Build settings from command line:
    ARCHS = arm64
    SDKROOT = iphoneos8.1

==== BUILD TARGET pop OF PROJECT pop WITH THE DEFAULT CONFIGURATION (Release) ===

...
CpHeader pop/POPAnimationTracer.h build/Headers/POP/POPAnimationTracer.h
    cd /Users/aaron/src/sharpie/pop
    export
PATH="/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin:/Applications
/Xcode.app/Contents/Developer/usr/bin:/Users/aaron/bin:/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/opt/X1
1/bin:/usr/local/git/bin:/Users/aaron/.rvm/bin"
    builtin-copy -exclude .DS_Store -exclude CVS -exclude .svn -exclude .git -exclude .hg -strip-debug-
symbols -strip-tool
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/strip -resolve-src-
symlinks /Users/aaron/src/sharpie/pop/pop/POPAnimationTracer.h
/Users/aaron/src/sharpie/pop/build/Headers/POP

...
** BUILD SUCCEEDED **
```

There will be a lot of build information output in the console as part of `xcodebuild`. As displayed above, we can see that a "CpHeader" target was run wherein header files were copied to a build output directory. This is often the case, and makes binding easier: as part of the native library's build, header files are often copied into a "publicly" consumable location which can make parsing easier for binding. In this case, we know that POP's header files are in the `build/Headers` directory.

We are now ready to bind POP. We know that we want to build for SDK `iphoneos8.1` with the `arm64` architecture, and that the header files we care about are in `build/Headers` under the POP git checkout. If we look in the `build/Headers` directory, we'll see a number of header files:

```
$ ls build/Headers/POP/
POP.h          POPAnimationTracer.h    POPDefines.h
POPAAnimatableProperty.h POPAnimator.h    POPGeometry.h
POPAnimation.h   POPAnimatorPrivate.h   POPLayerExtras.h
POPAnimationEvent.h POPBasicAnimation.h  POPPropertyAnimation.h
POPAnimationExtras.h POPCustomAnimation.h POPSpringAnimation.h
POPAnimationPrivate.h POPDecayAnimation.h
```

If we look at `POP.h`, we can see it is the library's main top-level header file that `#import`s other files. Because of this, we only need to pass `POP.h` to Objective Sharpie, and clang will do the rest behind the scenes:

```
$ sharpie bind -output Binding -sdk iphoneos8.1 \
-scope build/Headers build/Headers/POP/POP.h \
-c -Ibuild/Headers -arch arm64

Parsing Native Code...

Binding...
[write] ApiDefinitions.cs
[write] StructsAndEnums.cs

Binding Analysis:
Automated binding is complete, but there are a few APIs which have been flagged with [Verify] attributes. While the entire binding should be audited for best API design practices, look more closely at APIs with the following Verify attribute hints:

ConstantsInterfaceAssociation (1 instance):
There's no fool-proof way to determine with which Objective-C interface an extern variable declaration may be associated. Instances of these are bound as [Field] properties in a partial interface into a near-by concrete interface to produce a more intuitive API, possibly eliminating the 'Constants' interface altogether.

StronglyTypedNSArray (4 instances):
A native NSArray* was bound as NSObject[]. It might be possible to more strongly type the array in the binding based on expectations set through API documentation (e.g. comments in the header file) or by examining the array contents through testing. For example, an NSArray* containing only NSNumber* instances can be bound as NSNumber[] instead of NSObject[].

MethodToProperty (2 instances):
An Objective-C method was bound as a C# property due to convention such as taking no parameters and returning a value (non-void return). Often methods like these should be bound as properties to surface a nicer API, but sometimes false-positives can occur and the binding should actually be a method.

Once you have verified a Verify attribute, you should remove it from the binding source code. The presence of Verify attributes intentionally cause build failures.

For more information about the Verify attribute hints above, consult the Objective Sharpie documentation by running 'sharpie docs' or visiting the following URL:

http://xmnn.io/sharpie-docs

Submitting usage data to Xamarin...
Submitted - thank you for helping to improve Objective Sharpie!

Done.
```

You will notice that we passed a `-scope build/Headers` argument to Objective Sharpie. Because C and Objective-C libraries must `#import` or `#include` other header files that are implementation details of the library and not API you wish to bind, the `-scope` argument tells Objective Sharpie to ignore any API that is not defined in a file somewhere within the `-scope` directory.

You will find the `-scope` argument is often optional for cleanly implemented libraries, however there is no harm in explicitly providing it.

TIP

If the library's headers import any iOS SDK headers, e.g. `#import <Foundation.h>`, then you will need to set the scope otherwise Objective Sharpie will generate binding definitions for the iOS SDK header that was imported, resulting in a huge binding that will likely generate errors when compiling the binding project.

Additionally, we specified `-c -Ibuild/headers`. Firstly, the `-c` argument tells Objective Sharpie to stop interpreting command line arguments and pass any subsequent arguments *directly to the clang compiler*. Therefore, `-Ibuild/Headers` is a clang compiler argument that instructs clang to search for includes under `build/Headers`, which is where the POP headers live. Without this argument, clang would not know where to locate the files that `POP.h` is `#import`ing. *Almost all "issues" with using Objective Sharpie boil down to figuring out what to pass to clang*.

Completing the Binding

Objective Sharpie has now generated `Binding/ApiDefinitions.cs` and `Binding/StructsAndEnums.cs` files.

These are Objective Sharpie's basic first pass at the binding, and in a few cases it might be all you need. As stated above however, the developer will usually need to manually modify the generated files after Objective Sharpie finishes to [fix any issues](#) that could not be automatically handled by the tool.

Once the updates are complete, these two files can now be added to a binding project in Visual Studio for Mac or be passed directly to the `btouch` or `bmac` tools to produce the final binding.

For a thorough description of the binding process, please see our [Complete Walkthrough instructions](#).

Objective Sharpie Release History

1/23/2020 • 5 minutes to read • [Edit Online](#)

3.4 (October 11, 2017)

[Download v3.4.0](#)

- Support for Xcode 9: iOS 11, macOS 10.13, tvOS 11, and watchOS 4
- Issues with SIMD and tgmath should now be fixed
- Telemetry has been removed completely

3.3 (August 3, 2016)

[Download v3.3.0](#)

- Support for Xcode 8 Beta 4, iOS 10, macOS 10.12, tvOS 10 and watchOS 3.
- Updated to latest Clang master build (2016-08-02)
- [Persist telemetry submission options](#) from `sharpie pod bind` to `sharpie bind`.

3.2 (June 14, 2016)

[Download v3.2.3](#)

- Support for Xcode 8 Beta 1, iOS 10, macOS 10.12, tvOS 10 and watchOS 3.

3.1 (May 31, 2016)

[Download v3.1.1](#)

- Support for CocoaPods 1.0
- Improved CocoaPods binding reliability by first building a native `.framework` and then binding that
- Copy CocoaPods `.framework` and binding definition into a `Binding` directory to make integration with Xamarin.iOS and Xamarin.Mac binding projects easier

3.0 (October 5, 2015)

[Download v3.0.8](#)

- Support for new Objective-C language features including lightweight generics and nullability, as introduced in Xcode 7
- Support for the latest iOS and Mac SDKs.
- Xcode project building and parsing support! You can now pass a full Xcode project to Objective Sharpie and it will do its best to figure out the right thing to do (e.g. `sharpie bind Project.xcodeproj -sdk ios`).
- [CocoaPods](#) support! You can now configure, build, and bind CocoaPods directly from Objective Sharpie (e.g. `sharpie pod init ios AFNetworking && sharpie pod bind`).
- When binding frameworks, Clang modules will be enabled if the framework supports them, resulting in the most correct parsing of a framework, since the structure of the framework is defined by its `module.modulemap`.
- For Xcode projects that build a framework product, parse that product instead of intermediate product targets as non-framework targets in an Xcode project may still have ambiguities which cannot be

automatically resolved.

2.1.6 (March 17, 2015)

- Fixed binary operator expression binding: the left-hand side of the expression was incorrectly swapped with the right-hand (e.g. `1 << 0` was incorrectly bound as `0 << 1`). Thanks to Adam Kemp for noticing this!
- Fixed an issue with `NSInteger` and `NSUInteger` being bound as `int` and `uint` instead of `nint` and `nuint` on i386; `-DNS_BUILD_32_LIKE_64` is now passed to Clang to make parsing `objc/NSObjCRuntime.h` work as expected on i386.
- The default architecture for Mac OS X SDKs (e.g. `-sdk macosx10.10`) is now `x86_64` instead of `i386`, so `-arch` can be omitted unless overriding the default is desired.

2.1.0 (March 15, 2015)

[Download v2.1.0](#)

- [bxc#27849](#): Ensure `using ObjCRuntime;` is produced when `ArgumentSemantic` is used.
- [bxc#27850](#): Ensure `using System.Runtime.InteropServices;` is produced when `DllImport` is used.
- [bxc#27852](#): Default `DllImport` to loading symbols from `__Internal`.
- [bxc#27848](#): Skip forward-declared Objective-C container declarations.
- [bxc#27846](#): Bind protocol types with a single qualification as concrete interfaces (`id<Foo>` as `Foo` instead of `Foundation.NSObject<Foo>`).
- [bxc#28037](#): Bind `UInt32`, `UInt64`, and `Int64` literals as `Int32` to drop the `u` and/or `uL` suffixes when the values can safely fit into `Int32`.
- [bxc#28038](#): Fix enum name mapping when original native name starts with a `k` prefix.
- `sizeof` C expressions whose argument type does not map to a C# primitive type will be evaluated in Clang and bound as an integer literal to avoid generating invalid C#.
- Fix Objective-C syntax for properties whose type is a block (Objective-C code appears in comments above bound declarations).
- Bind decayed types as their original type (`int[]` decays to `int*` during semantic analysis in Clang, but bind it as the original as-written `int[]` instead).

Thanks very much to Dave Dunkin for reporting many of the bugs fixed in this point release!

2.0.0: March 9, 2015

[Download v2.0.0](#)

Objective Sharpie 2.0 is a major release that features an improved Clang-based driver and parser and a new NRefactory-based binding engine. These improved components provide for *much* better bindings, particularly around type binding. Many other improvements have been made that are internal to Objective Sharpie which will yield many user-visible features in future releases.

Objective Sharpie 2.0 is based on Clang 3.6.1.

Type binding improvements

- Objective-C blocks are now supported. This includes anonymous/inline blocks and blocks named via `typedef`. Anonymous blocks will be bound as `System.Action` or `System.Func` delegates, while named blocks will be bound as strongly named `delegate` types.
- There is an improved naming heuristic for anonymous enums that are immediately preceded by a `typedef` resolving to a builtin integral type such as `long` or `int`.

- C pointers are now bound as C# `unsafe` pointers instead of `System.IntPtr`. This results in more clarity in the binding for when you may wish to turn pointer parameters into `out` or `ref` parameters. It is not possible to always infer whether a parameter should be `out` or `ref`, so the pointer is retained in the binding to allow for easier auditing.
- An exception to the above pointer binding is when a 2-rank pointer to an Objective-C object is encountered as a parameter. In these cases, convention is predominant and the parameter will be bound as `out` (e.g. `NSError **error` → `out NSError error`).

Verify attribute

You will often find that bindings produced by Objective Sharpie will now be annotated with the `[Verify]` attribute. These attributes indicate that you should *verify* that Objective Sharpie did the correct thing by comparing the binding with the original C/Objective-C declaration (which will be provided in a comment above the bound declaration).

Verification is *recommended* for all bound declarations, but is most likely *required* for declarations annotated with the `[Verify]` attribute. This is because in many situations, there is not enough metadata in the original native source code to infer how to best produce a binding. You may need to reference documentation or code comments inside the header files to make the best binding decision.

See the [Verify Attributes](#) documentation for more details.

Other notable improvements

- `using` statements are now generated based on types bound. For instance, if an `NSURL` was bound, a `using Foundation;` statement will be generated as well.
- `struct` and `union` declarations will now be bound, using the `[FieldOffset]` trick for unions.
- Enum values with constant expression initializers will now be properly bound; the full expression is translated to C#.
- Variadic methods and blocks are now bound.
- Frameworks are now supported via the `-framework` option. See the documentation on [Binding Native Frameworks](#) for more details.
- Objective-C source code will be auto-detected now, which should eliminate the need to pass `-ObjC` or `-xobjective-c` to Clang manually.
- Clang module usage (`@import`) is now auto-detected, which should eliminate the need to pass `-fmodules` to Clang manually for libraries which use the new module support in Clang.
- The Xamarin Unified API is now the default binding target; use the `-classic` option to target the 32-bit only Classic API.

Notable bug fixes

- Fix `instancetype` binding when used in an Objective-C category
- Fully name Objective-C categories
- Prefix Objective-C protocols with `I` (e.g. `INSCopying` instead of `NSCopying`)

1.1.35: December 21, 2014

[Download v1.1.35](#)

Minor bug fixes.

1.1.1: December 15, 2014

[Download v1.1.1](#)

1.1.1 was the first major release after 1.5 years of internal use and development at Xamarin following the initial preview of Objective Sharpie in April 2013. This release is the first to be generally considered stable and usable for a wide variety of native libraries, featuring a new Clang backend.

Binding troubleshooting

11/2/2020 • 3 minutes to read • [Edit Online](#)

Some tips for troubleshooting bindings to macOS (formerly known as OS X) APIs in Xamarin.Mac.

Missing bindings

While Xamarin.Mac covers much of the Apple APIs, sometimes you may need to call some Apple API that doesn't have a binding yet. In other cases, you need to call third party C/Objective-C that is outside the scope of the Xamarin.Mac bindings.

If you are dealing with an Apple API, the first step is to let Xamarin know that you are hitting a section of the API that we don't have coverage for yet. [File a bug](#) noting the missing API. We use reports from customers to prioritize which APIs we work on next. In addition, if you have a Business or Enterprise license and this lack of a binding is blocking your progress, also follow the instructions at [Support](#) to file a ticket. We can't promise a binding, but in some cases we can get you a work around.

Once you notify Xamarin (if applicable) of your missing binding, the next step is to consider binding it yourself. We have a full guide [here](#) and some unofficial documentation [here](#) for wrapping Objective-C bindings by hand. If you are calling a C API, you can use C#'s P/Invoke mechanism, documentation is [here](#).

If you decide to work on the binding yourself, be aware that mistakes in the binding can produce all sorts of interesting crashes in the native runtime. In particular, be very careful that your signature in C# matches the native signature in number of arguments and the size of each argument. Failure to do so may corrupt memory and/or the stack and you could crash immediately or at some arbitrary point in the future or corrupt data.

Argument exceptions when passing null to a binding

While Xamarin works to provide high quality and well tested bindings for the Apple APIs, sometimes mistakes and bugs slip in. By far the most common issue that you might run into is an API throwing `ArgumentNullException` when you pass in null when the underlying API accepts `nil`. The native header files defining the API often do not provide enough information on which APIs accept nil and which will crash if you pass it in.

If you run into a case where passing in `null` throws an `ArgumentNullException` but you think it should work, follow these steps:

1. Check the Apple documentation and/or examples to see if you can find proof that it accepts `nil`. If you are comfortable with Objective-C, you can write a small test program to verify it.
2. [File a bug](#).
3. Can you work around the bug? If you can avoid calling the API with `null`, a simple null check around the calls can be an easy work around.
4. However, some APIs require passing in null to turn off or disable some features. In these cases, you can work around the issue by bringing up the assembly browser (see [Finding the C# member for a given selector](#)), copying the binding, and removing the null check. Please make sure to file a bug (step 2) if you do this, as your copied binding won't receive updates and fixes that we make in Xamarin.Mac, and this should be considered a short term work around.

Reporting bugs

Your feedback is important to us. If you find any problems with Xamarin.Mac:

- Check the [Xamarin.Mac Forums](#)
- Search the [issue repository](#)
- Before switching to GitHub issues, Xamarin issues were tracked on [Bugzilla](#). Please search there for matching issues.
- If you cannot find a matching issue, please file a new issue in the [GitHub issue repository](#).

GitHub issues are all public. It's not possible to hide comments or attachments.

Please include as much of the following as possible:

- A simple example reproducing the issue. This is **invaluable** where possible.
- The full stack trace of the crash.
- The C# code surrounding the crash.

Native References in iOS, Mac, and Bindings Projects

11/2/2020 • 3 minutes to read • [Edit Online](#)

Native references gives you the ability to embed a native framework into a Xamarin.iOS or Xamarin.Mac project or binding project.

Since iOS 8.0 it's been possible to create an embedded framework to share code between app extensions and the main app in Xcode. Using the Native Reference feature it will be possible to consume these embedded frameworks (created with Xcode) in Xamarin.iOS.

IMPORTANT

It will not be possible to create embedded frameworks from any type of Xamarin.iOS or Xamarin.Mac projects, Native References only allow for the consumption of existing native (Objective-C) frameworks.

Terminology

In iOS 8 (and later), **Embedded Frameworks** can be both embedded statically linked and dynamically linked Frameworks. To properly distribute them, you must make them into "fat" Frameworks that included all of their *Slices* for each device architecture that you want to support with your app.

Static vs. Dynamic Frameworks

Static Frameworks are linked at compile time where **Dynamic Frameworks** are linked at runtime and therefore can be modified without re-linking. If you have used any 3rd-party Framework prior to iOS 8, you were using a **Static Framework** that was compiled into your app. See Apple's [Dynamic Library Programming](#) documentation for more details.

Embedded vs. System Frameworks

Embedded Frameworks are included in your apps bundle and are only accessible to your specific app via its sandbox. **System Frameworks** are stored at the Operating System Level and are available to all apps on the device. Currently, only Apple has the ability to create Operating System Level Frameworks.

Thin vs. Fat Frameworks

Thin Frameworks contain only the compiled code for a specific system architecture where **Fat Frameworks** contain code for multiple architectures. Each architecture-specific codebase compiled into a Framework is referred to as a *Slice*. So, for example, if we had a Framework that was compiled for the two iOS Simulator architectures (i386 and X86_64), it would contain two Slices.

If you tried to distribute this sample Framework with your app, it would run correctly on the Simulator, but fail on the device since the Framework does not contain any code-specific Slices for an iOS device. To ensure that a Framework will work in all instances, it would also need to include device-specific Slices such as arm64, armv7 and armv7s.

Working with Embedded Frameworks

There are two steps that must be completed to work with Embedded Frameworks in a Xamarin.iOS or Xamarin.Mac app: Creating a Fat Framework and Embedding the Framework.

Creating a Fat Framework

As stated above, to be able to consume an Embedded Framework in your app, it must be a Fat Framework that includes all of the system architectures Slices for the devices that your app will run on.

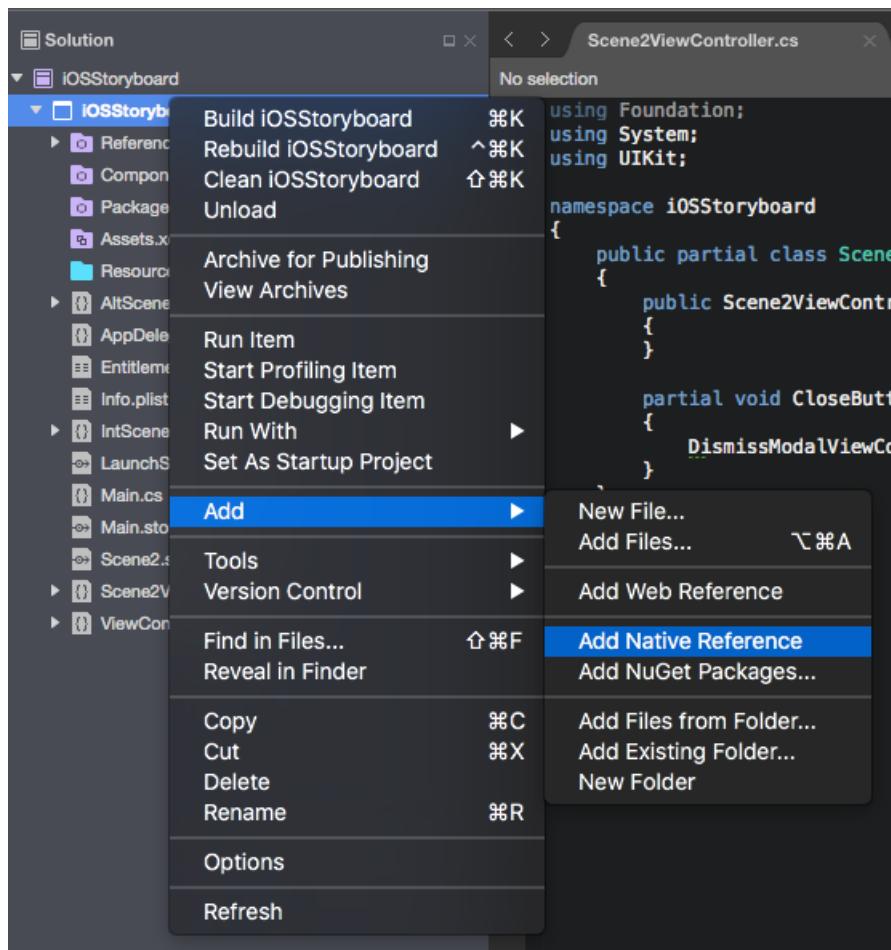
When the Framework and the consuming app are in the same Xcode project, this is not a problem as Xcode will build both the Framework and the App using the same build settings. Since Xamarin apps cannot create Embedded Frameworks, this technique cannot be used.

To solve this issue, the `lipo` command line tool can be used to merge two or more Frameworks into one Fat Framework containing all of the necessary Slices. For more information on working with the `lipo` command, please see our [Linking Native Libraries](#) documentation.

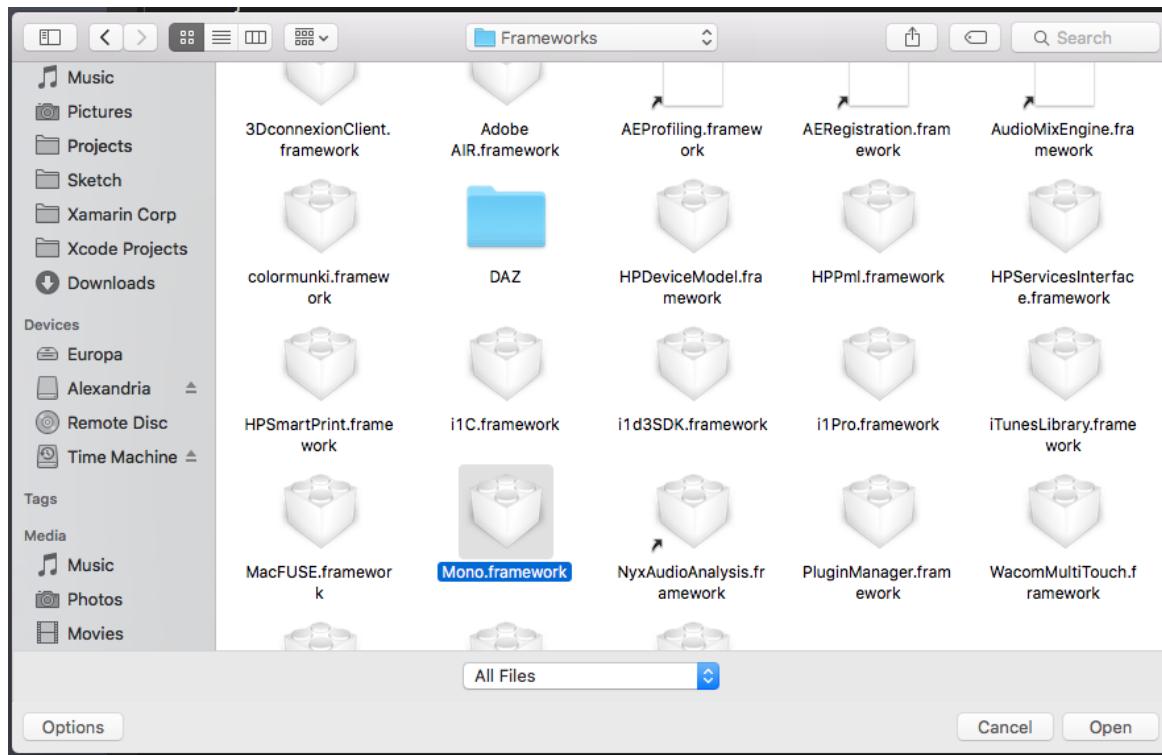
Embedding a Framework

The follow step are required to embed a framework in a Xamarin.iOS or Xamarin.Mac project using Native References:

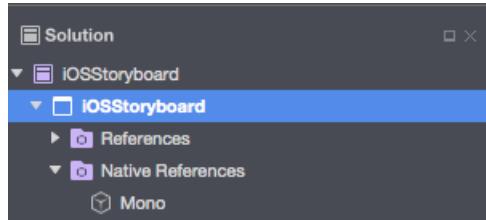
1. Create a new or open an existing Xamarin.iOS, Xamarin.Mac or Binding project.
2. In the **Solution Explorer**, right-click on the project name and select **Add > Add Native Reference**:



3. From the **Open** dialog box, select the name of the Native Framework that you want to embed and click the **Open** button:



4. The framework will be added to the project's tree:



When the project is compiled, the Native Framework will be embedded in the App's bundle.

App Extensions and Embedded Frameworks

Internally Xamarin.iOS may take advantage of this feature to link with the Mono runtime as a framework (when the deployment target is \geq iOS 8.0), thus reducing app size significantly for apps with extensions (since the Mono runtime will be included only once for the entire app bundle, instead of once for the container app and once for each extension).

Extensions will link with the Mono runtime as a framework, because all extensions require iOS 8.0.

Apps that don't have extensions and apps that target iOS

Summary

This article has taken a detailed look at embedding a native Framework into a Xamarin.iOS or Xamarin.Mac application.

Native Types for iOS and macOS

10/28/2019 • 2 minutes to read • [Edit Online](#)

Mac and iOS APIs use architecture-specific data types that are always 32 bits on 32-bit platforms and 64 bits on 64-bit platforms.

For example, Objective-C maps the `NSInteger` data type to `int32_t` on 32-bit systems and to `int64_t` on 64-bit systems.

To match this behavior, on our unified API, we are replacing the previous uses of `int` (which in .NET is defined as always being `System.Int32`) to a new data type: `System.nint`. You can think of the "n" as meaning "native", so the native integer type of the platform.

With these new data types, the same source code is compiled for 32-bit and 64-bit architectures, depending on your compilation flags.

New Data Types

The following table shows the changes in our data types to match this new 32/64-bit world:

NATIVE TYPE	32-BIT BACKING TYPE	64-BIT BACKING TYPE
<code>System.nint</code>	<code>System.Int32</code> (<code>int</code>)	<code>System.Int64</code> (<code>long</code>)
<code>System.nuint</code>	<code>System.UInt32</code> (<code>uint</code>)	<code>System.UInt64</code> (<code>ulong</code>)
<code>System.nfloat</code>	<code>System.Single</code> (<code>float</code>)	<code>System.Double</code> (<code>double</code>)

We chose those names to allow your C# code to look more or less the same way that it would look today.

Implicit and Explicit Conversions

The design of the new data types is intended to allow a single C# source file to naturally use 32 or 64 bit storage depending on the host platform and the compilation settings.

This required us to design a set of implicit and explicit conversions to and from the platform-specific data types to the .NET integral and floating point data types.

Implicit conversions operators are provided when there is no possibility for data loss (32 bit values being stored on a 64 bit space).

Explicit conversions operators are provided when there is a potential for data loss (64 bit value is being stored on a 32 or potentially 32 storage location).

`int`, `uint` and `float` are all implicitly convertible to `nint`, `nuint` and `nfloat` as 32 bits will always fit in 32 or 64 bits.

`nint`, `nuint` and `nfloat` are all implicitly convertible to `long`, `ulong` and `double` as 32 or 64 bit values will always fit in 64 bit storage.

You must use explicit conversions from `nint`, `nuint` and `nfloat` into `int`, `uint` and `float` since the native types might hold 64 bits of storage.

You must use explicit conversions from `long`, `ulong` and `double` into `nint`, `nuint` and `nfloat` since the

native types might only be able to hold 32 bits of storage.

CoreGraphics Types

The point, size and rectangle data types that are used with CoreGraphics use 32 or 64 bits depending on the device they are running on. When we originally bound the iOS and Mac APIs we used existing data structures that happened to match the sizes of the host platform (The data types in `System.Drawing`).

When moving to Unified, you will need to replace instances of `System.Drawing` with their `CoreGraphics` counterparts as shown in the following table:

OLD TYPE IN SYSTEM.DRAWING	NEW DATA TYPE COREGRAPHICS	DESCRIPTION
<code>RectangleF</code>	<code>CGRect</code>	Holds floating point rectangle information.
<code>SizeF</code>	<code>CGSize</code>	Holds floating point size information (width, height)
<code>PointF</code>	<code>CGPoint</code>	Holds a floating point, point information (X, Y)

The old data types used floats to store the elements of the data structures, while the new one uses `System.nfloat`.

Related Links

- [Working with Native Types in Cross-Platform Apps](#)
- [Classic vs Unified API differences](#)

32/64-bit platform considerations

11/2/2020 • 4 minutes to read • [Edit Online](#)

While iOS and macOS have historically supported both 32 and 64-bit apps, Apple has gradually deprecated 32-bit support.

As of iOS 11, 32-bit apps will no longer launch, and [all submissions to the App Store must support 64-bit](#).

Starting in January 2018, [new apps submitted to the Mac App Store must support 64-bit](#), and existing apps must be updated by June 2018.

Xamarin's Classic API (`xamMac.dll` and `monotouch.dll`) supported only 32-bit applications. However, new Xamarin.iOS and Xamarin.Mac applications use the [Unified API](#) (`Xamarin.iOS` and `Xamarin.Mac`) by default, and can therefore target both 32 and 64-bit, as necessary.

iOS

Enabling 64-bit builds of Xamarin.iOS apps

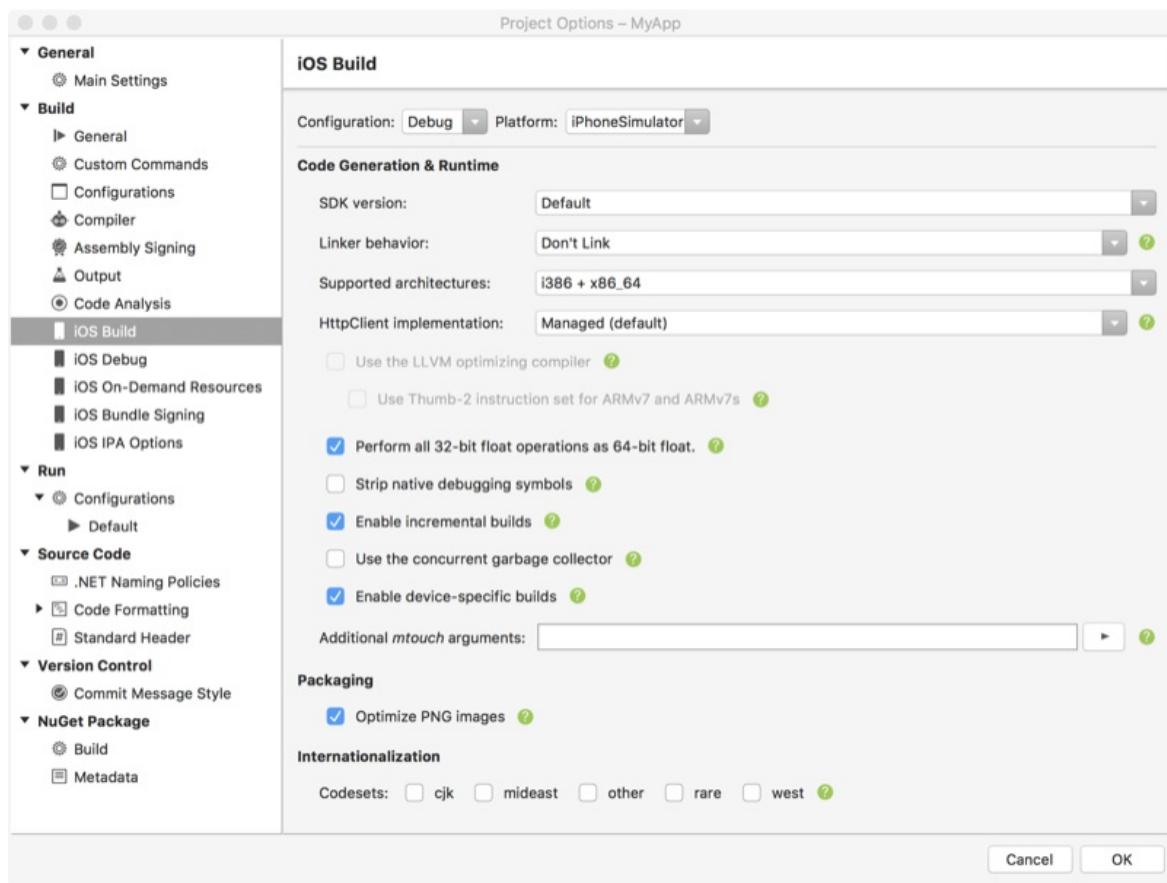
WARNING

This section is included for historic reasons, and to help move older Xamarin.iOS projects to the Unified API and support 64-bit. All new Xamarin.iOS projects will use the Unified API and target 64-bit by default.

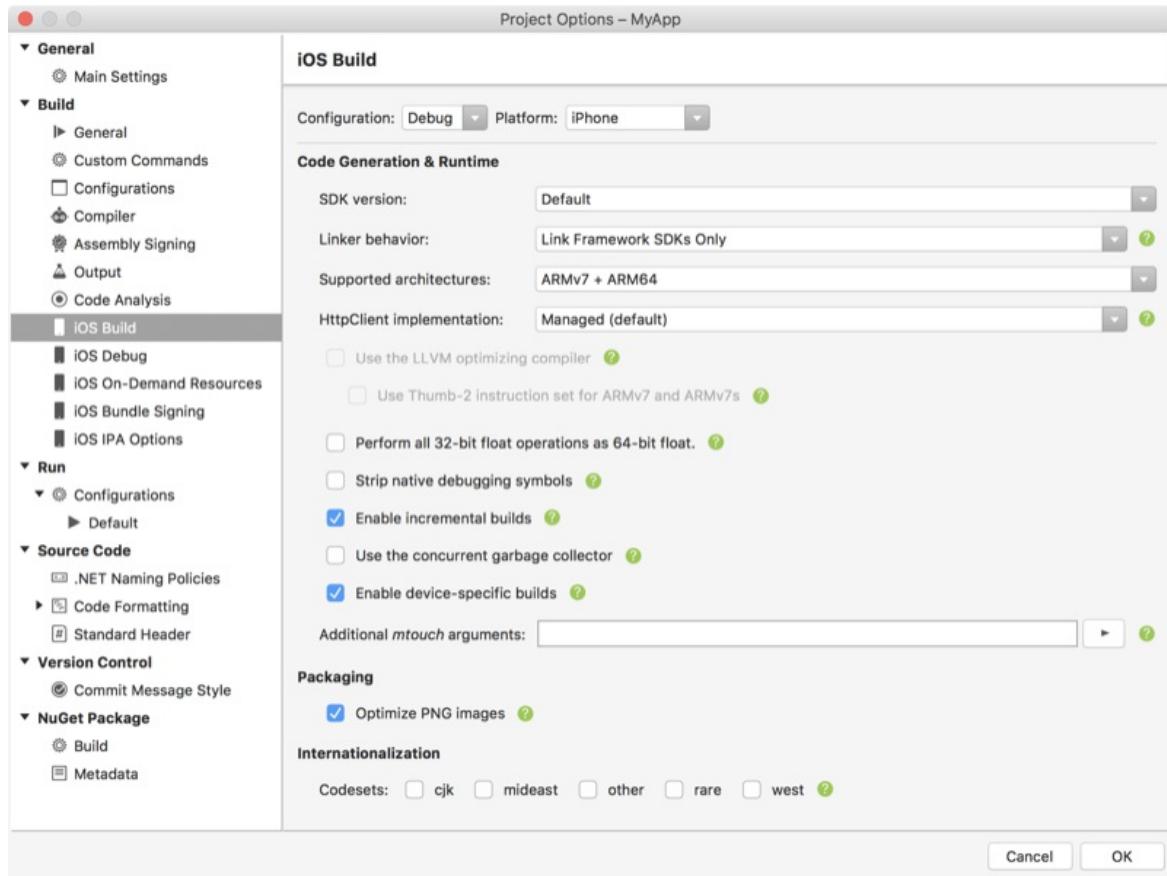
For Xamarin.iOS mobile applications that have been converted to the Unified API, developers must manually update the build settings to target 64-bit:

- [Visual Studio for Mac](#)
- [Visual Studio](#)

1. In the **Solution Pad**, double-click the app's project to open the **Project Options** window.
2. Select **iOS Build**.
3. For the iPhone Simulator, in the **Supported architectures** dropdown, select either **x86_64** or **i386 + x86_64**:



4. For physical devices, select one of the available **ARM64** combinations:



5. Click **OK**.

6. Perform a clean build.

ARMv7s is supported only by the A6 processor included in the iPhone 5 (or greater). ARMv7 code is faster and smaller than the ARMv6, only works with the iPhone 3GS and later, and is required by Apple when targeting the

iPad or a minimum iOS version of 5.0. ARMv6 works on all devices but is no longer supported by the compiler shipped with Xcode 4.5 and later.

ARM64 is required to support iOS 8 on iPhone 6 or other 64-bit devices and will be required by Apple when submitting new or updating existing applications in the iTunes App Store.

For a comprehensive look at the capabilities of various iOS devices, check out Apple's [Device Compatibility](#) document.

64-bit and binary size increases

During Apple's transition from 32-bit to 64-bit, iOS apps will need to run on both 32-bit and 64-bit hardware. Because of this, Xamarin's Unified API allows developers to target both.

Targeting both 32-bit and 64-bit architectures will significantly increase the size of an application. However, doing so will allow newer devices to run optimized code while still supporting older devices.

IMPORTANT

If you receive the following message when submitting an iOS application to the iTunes App Store, "*WARNING ITMS-9000: Missing 64-bit support. Starting February 1, 2015, new iOS apps uploaded to the App Store must include 64-bit support and be built with the iOS 8 SDK, included in Xcode 6 or later. To enable 64-bit in your project, we recommend using the default Xcode build setting of "Standard architectures" to build a single binary with both 32-bit and 64-bit code.*" You need to switch the supported architectures to one of the available **ARM64** combination (as shown above), recompile and resubmit.

Mac

IMPORTANT

Starting in January 2018, all new Mac apps submitted to the Mac App Store must support 64-bit. Existing Mac App Store apps and their updates must support 64-bit starting in June 2018. See [Apple's announcement](#) and [a guide that describes how to update your Xamarin.Mac apps to 64-bit](#).

Most modern Mac computers support both 32-bit and 64-bit applications. MacOS 10.6 (Snow Leopard) was the last operating system to run on 32-bit systems. Most Macs released since 2010 support both systems.

Unlike iOS, many of the new frameworks introduced in recent versions of macOS are only supported in 64-bit mode (CloudKit, EventKit, GameController, LocalAuthentication, MediaLibrary, MultipeerConnectivity, NotificationCenter, GLKit, SpriteKit, Social, and MapKit, among others).

The Unified API allow developers to choose what kind of applications they want to produce: 32-bit or 64-bit.

32-bit applications will run on both 32-bit and 64-bit Mac computers, have an address space limited to 32 bits, and require that all libraries are 32 bits.

You will typically use this mode if you have 32-bit dependencies that do not run in 64-bit mode, if you want to have a smaller download, or if there are no performance benefits in moving to 64-bit.

This mode is limiting as you wont be able to use many frameworks available in macOS Mavericks and macOS Yosemite.

64-bit applications will only run on 64-bit Mac devices.

For Mac, this is the preferred mode of operation as most Macs in use today support 64-bit mode, and you have access to the complete set of frameworks provided by Apple.

Enabling 64-bit builds of Xamarin.Mac apps

For information about building a 64-bit app using Xamarin.Mac, please see the [Updating Xamarin.Mac Unified applications to 64-bit](#) guide.

Related links

- [Classic vs Unified API differences](#)

Updating Xamarin.Mac Unified applications to 64-bit

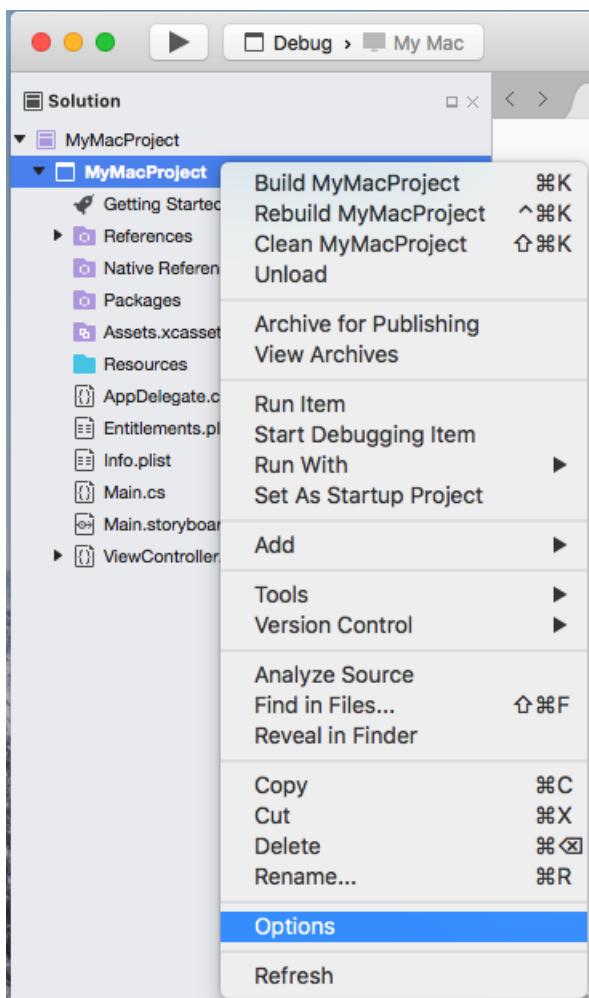
10/28/2019 • 2 minutes to read • [Edit Online](#)

As of January 2018, Apple requires that new [Mac App Store submissions target 64-bit](#). Apps already available on the Mac App Store must be updated to target 64-bit by June 2018.

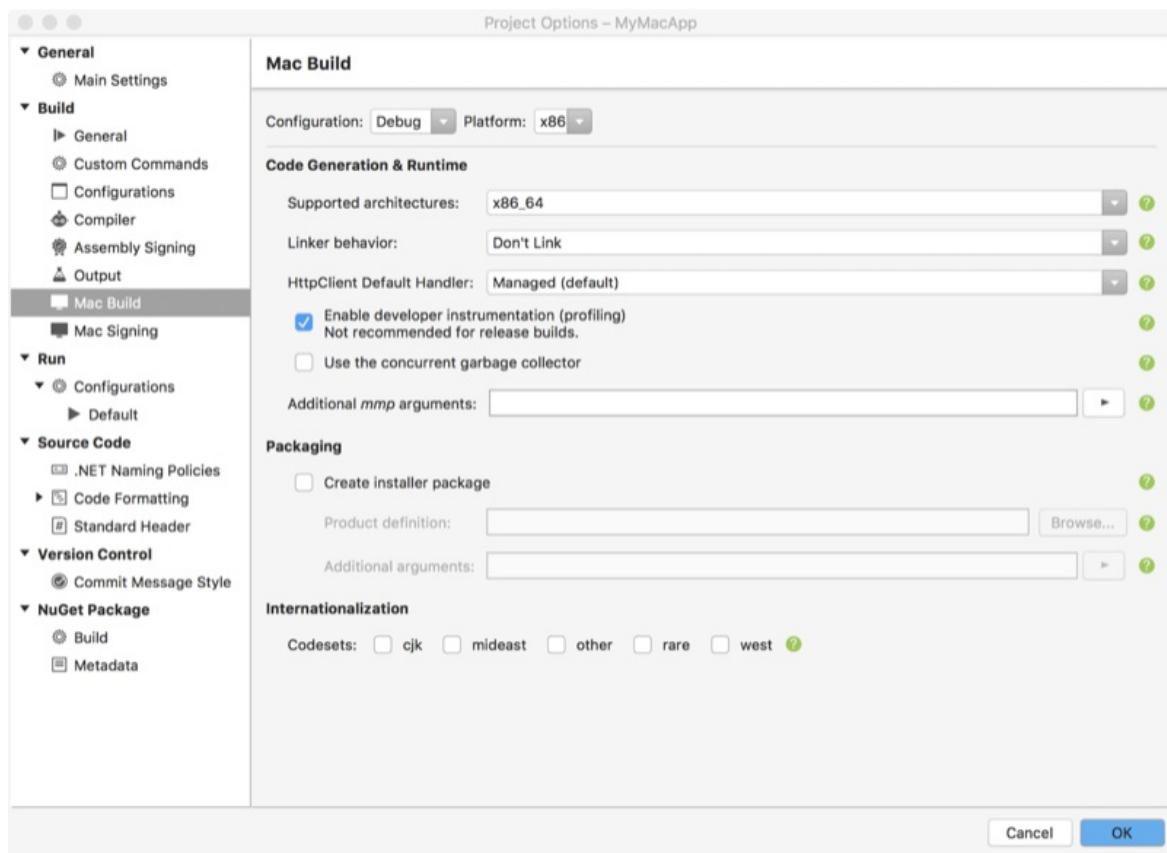
The **File > New** Xamarin.Mac project template creates 64-bit applications by default, so any recently created apps are already 64-bit compatible and will not require any changes.

Targeting 64-bit

1. Open the **Project Options** window for your Xamarin.Mac app:



2. Select **Mac Build** and set **Supported architectures** to **x86_64**:



3. If your app has any external dependencies such as native references or binding projects, update them to target 64-bit.

Errors

The first time you build or run your application with 64-bit support, you may encounter link errors from clang or runtime issues. These errors can occur if third-party dependencies — for example, native references in your Xamarin.Mac or bindings projects, or manually-loaded system-wide frameworks — have not been updated to 64-bit.

TIP

Converting your project to 64-bit is a major change and may indirectly uncover various programming errors. In particular it may change the size and alignment of data structures, which would affect p/invoke signatures and native code linked in your project. Consider reviewing any build warnings given and test your application thoroughly afterwards to catch potential issues.

Example error resulting from a dynamically-linked third-party dependency that does not target 64-bit:

```
ld : warning : ignoring file PATH/ThirdPartyLibrary.framework/ThirdPartyLibrary,
file was built for i386 which is not the architecture being linked (x86_64):
PATH/ThirdPartyLibrary.framework/ThirdPartyLibrary
```

This error could be followed at runtime by `dlopen` returning `IntPtr.Zero` instead of an expected handle.

Example error resulting from a statically-linked third-party dependency that does not target 64-bit:

```
Undefined symbols for architecture x86_64:
 "_LibraryFunction", referenced from:
 -u command line option
ld: symbol(s) not found for architecture x86_64
```

To build and run successfully, update these dependencies to 64-bit and recompile your app.

Working with Native Types in Cross-Platform Apps

11/2/2020 • 7 minutes to read • [Edit Online](#)

This article covers using the new iOS Unified API Native types (`nint`, `nuint`, `nfloat`) in a cross-platform application where code is shared with non-iOS devices such as Android or Windows Phone OSes.

The 64-types native types work with the iOS and Mac APIs. If you are writing shared code that runs on Android or Windows as well, you'll need to manage the conversion of Unified types into regular .NET types that you can share.

This document discusses different ways to interoperate with the Unified API from your shared/common code.

When to Use the Native Types

Xamarin.iOS and Xamarin.Mac Unified APIs still include the `int`, `uint` and `float` data types, as well as the `RectangleF`, `SizeF` and `PointF` types. These existing data types should continue to be used in any shared, cross-platform code. The new Native data types should only be used when making a call to a Mac or iOS API where support for architecture-aware types are required.

Depending on the nature of the code being shared, there might be times where cross-platform code might need to deal with the `nint`, `nuint` and `nfloat` data types. For example: a library that handles transformations on rectangular data that was previously using `System.Drawing.RectangleF` to share functionality between Xamarin.iOS and Xamarin.Android versions of an app, would need to be updated to handle Native Types on iOS.

How these changes are handled depends on the size and complexity of the application and the form of code sharing that has been used, as we will see in the following sections.

Code Sharing Considerations

As stated in the [Sharing Code Options](#) document, there are two main ways to sharing code between cross-platform projects: Shared Projects and Portable Class Libraries. Which of the two types has been used, will limit the options we have when handling the Native data types in cross-platform code.

Portable Class Library Projects

A Portable Class Library (PCL) allows you to target the platforms you wish to support, and use interfaces to provide platform-specific functionality.

Since the PCL Project type is compiled down to a `.DLL` and it has no sense of the Unified API, you'll be forced to keep using the existing data types (`int`, `uint`, `float`) in the PCL source code and type cast the calls to the PCL's classes and methods in the front-end applications. For example:

```
using NativePCL;
...
CGRect rect = new CGRect (0, 0, 200, 200);
Console.WriteLine ("Rectangle Area: {0}", Transformations.CalculateArea ((RectangleF)rect));
```

Shared Projects

The Shared Asset Project type allows you to organize your source code in a separate project that then gets included and compiled into the individual platform-specific front end apps, and use `#if` compiler directives as required to manage platform-specific requirements.

The size and complexity of the front end mobile applications that are consuming shared code, along with the size and the complexity of the code being shared, needs to be taken into account when choosing the method of support for Native data types in a cross-platform Shared Asset Project.

Based on these factors, the following types of solutions might be implemented using the `#if __UNIFIED__ ... #endif` compiler directives to handle the Unified API specific changes to the code.

Using Duplicate Methods

Take the example of a library that is doing transformations on rectangular data given above. If the library only contains one or two very simple methods, you might choose to create duplicate versions of those methods for Xamarin.iOS and Xamarin.Android. For example:

```
using System;
using System.Drawing;

#if __UNIFIED__
using CoreGraphics;
#endif

namespace NativeShared
{
    public class Transformations
    {
        #region Constructors
        public Transformations ()
        {
        }
        #endregion

        #region Public Methods
        #if __UNIFIED__
            public static nfloat CalculateArea(CGRect rect) {

                // Calculate area...
                return (rect.Width * rect.Height);

            }
        #else
            public static float CalculateArea(RectangleF rect) {

                // Calculate area...
                return (rect.Width * rect.Height);

            }
        #endif
        #endregion
    }
}
```

In the above code, since the `CalculateArea` routine is very simple, we have used conditional compilation and created a separate, Unified API version of the method. On the other hand, if the library contained many routines or several complex routines, this solution would not be feasible, as it would present an issue keeping all of the methods in sync for modifications or bug fixes.

Using Method Overloads

In that case, the solution might be to create an overload version of the methods using 32-bit data types so that they now take `CGRect` as a parameter and/or a return value, convert that value to a `RectangleF` (knowing that converting from `nfloat` to `float` is a lossy conversion), and call the original version of the routine to do the actual work. For example:

```

using System;
using System.Drawing;

#if __UNIFIED__
using CoreGraphics;
#endif

namespace NativeShared
{
    public class Transformations
    {
        #region Constructors
        public Transformations ()
        {
        }
        #endregion

        #region Public Methods
        #if __UNIFIED__
            public static nfloat CalculateArea(CGRect rect) {

                // Call original routine to calculate area
                return (nfloat)CalculateArea((RectangleF)rect);

            }
        #endif

        public static float CalculateArea(RectangleF rect) {

            // Calculate area...
            return (rect.Width * rect.Height);

        }
        #endregion
    }
}

```

Again, this is a good solution as long as the loss of precision doesn't affect the results for your application's specific needs.

Using Alias Directives

For areas where the loss of precision is an issue, another possible solution is to use `using` directives to create an alias for Native and CoreGraphics data types by including the following code to the top of the shared source code files and converting any needed `int`, `uint` or `float` values to `nint`, `nuint` or `nfloat`:

```

#if __UNIFIED__
    // Mappings Unified CoreGraphic classes to MonoTouch classes
    using RectangleF = global::CoreGraphics.CGRect;
    using SizeF = global::CoreGraphics.CGSize;
    using PointF = global::CoreGraphics.CGPoint;
#else
    // Mappings Unified types to MonoTouch types
    using nfloat = global::System.Single;
    using nint = global::System.Int32;
    using nuint = global::System.UInt32;
#endif

```

So that our example code then becomes:

```

using System;
using System.Drawing;

#if __UNIFIED__
    // Map Unified CoreGraphic classes to MonoTouch classes
    using RectangleF = global::CoreGraphics.CGRect;
    using SizeF = global::CoreGraphics.CGSize;
    using PointF = global::CoreGraphics.CGPoint;
#else
    // Map Unified types to MonoTouch types
    using nfloat = global::System.Single;
    using nint = global::System.Int32;
    using uint = global::System.UInt32;
#endif

namespace NativeShared
{
    public class Transformations
    {
        #region Constructors
        public Transformations ()
        {
        }
        #endregion

        #region Public Methods
        public static nfloat CalculateArea(RectangleF rect) {

            // Calculate area...
            return (rect.Width * rect.Height);

        }
        #endregion
    }
}

```

Note that here we have changed the `CalculateArea` method to return an `nfloat` instead of the standard `float`. This was done so that we would not get a compile error trying to *implicitly* convert the `nfloat` result of our calculation (since both values being multiplied are of type `nfloat`) into a `float` return value.

If the code is compiled and run on a non Unified API device, the `using nfloat = global::System.Single;` maps the `nfloat` to a `Single` which will implicitly convert to a `float` allowing the consuming front-end application to call the `CalculateArea` method without modification.

Using Type Conversions in the Front End App

In the event that your front end applications only make a handful of calls to your shared code library, another solution could be to leave the library unchanged and do type casting in the Xamarin.iOS or Xamarin.Mac application when calling the existing routine. For example:

```

using NativeShared;
...

CGRect rect = new CGRect (0, 0, 200, 200);
Console.WriteLine ("Rectangle Area: {0}", Transformations.CalculateArea ((RectangleF)rect));

```

If the consuming application makes hundreds of calls to the shared code library, this again, might not be a good solution.

Based on our application's architecture, we might end up using one or more of the above solutions to support Native Data Types (where required) in our cross-platform code.

Xamarin.Forms Applications

The following is required to use Xamarin.Forms for cross-platform UIs that will also be shared with a Unified API application:

- The entire solution must be using version 1.3.1 (or greater) of the Xamarin.Forms NuGet Package.
- For any Xamarin.iOS custom renders, use the same types of solutions presented above based on how the UI code has been shared (Shared Project or PCL).

As in a standard cross-platform application, the existing 32-bit data types should be used in any shared, cross-platform code for most situations. The new Native Data Types should only be used when making a call to a Mac or iOS API where support for architecture-aware types is required.

For more details, see our [Updating Existing Xamarin.Forms Apps](#) documentation.

Summary

In this article we have seen when to use the Native Data Types in a Unified API application and their implications cross-platform. We have presented several solutions that can be used in situations where the new Native Data Types must be used in cross-platform libraries. Also, we've seen a quick guide to supporting Unified APIs in Xamarin.Forms cross-platform applications.

Related Links

- [Unified API](#)
- [Native Types](#)
- [Sharing Code Options](#)
- [Code Sharing Sample](#)

HttpClient and SSL/TLS implementation selector for iOS/macOS

10/28/2019 • 4 minutes to read • [Edit Online](#)

The **HttpClient Implementation Selector** for Xamarin.iOS, Xamarin.tvOS, and Xamarin.Mac controls which `HttpClient` implementation to use. You can switch to an implementation that uses iOS, tvOS, or macOS native transports (`NSURLSession` or `CFNetwork`, depending on the OS). The upside is TLS 1.2-support, smaller binaries, and faster downloads; the downside is that it requires the event loop to be running for async operations to be executed.

Projects must reference the `System.Net.Http` assembly.

WARNING

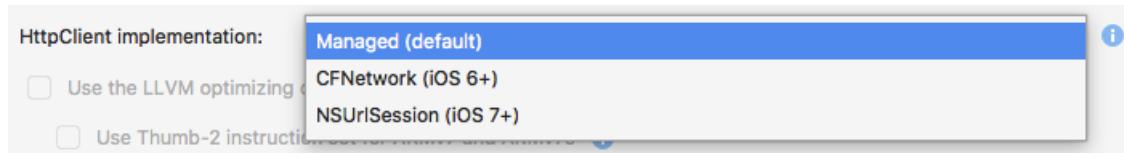
April, 2018 – Due to increased security requirements, including PCI compliance, major cloud providers and web servers are expected to stop supporting TLS versions older than 1.2. Xamarin projects created in previous versions of Visual Studio default to use older versions of TLS.

In order to ensure your apps continue to work with these servers and services, **you should update your Xamarin projects with the `NSURLSession` setting shown below, then re-build and re-deploy your apps to your users.**

Selecting an HttpClient stack

To adjust the `HttpClient` being used by your app:

1. Double-click the **Project Name** in the **Solution Explorer** to open the Project Options.
2. Switch to the **Build** settings for your project (for example, **iOS Build** for a Xamarin.iOS app).
3. From the **HttpClient Implementation** dropdown, select the `HttpClient` type as one of the following:
NSURLSession (recommended), **CFNetwork**, or **Managed**.



TIP

For TLS 1.2 support the `NSURLSession` option is recommended.

NSURLSession

The `NSURLSession`-based handler is based on the native `NSURLSession` framework available in iOS 7 and newer. **This is the recommended setting.**

Pros

- It uses native APIs for better performance and smaller executable size.
- Support for the latest standards such as TLS 1.2.

Cons

- Requires iOS 7 or later.

- Some `HttpClient` features/options are not available.

CFNetwork

The `CFNetwork`-based handler is based on the native `CFNetwork` framework available in iOS 6 and newer.

Pros

- It uses native APIs for better performance and smaller executable size.
- Support for newer standards such as TLS 1.2.

Cons

- Requires iOS 6 or later.
- Not available on watchOS.
- Some `HttpClient` features/options are not available.

Managed

The Managed handler is the fully managed `HttpClient` handler that has been shipped with previous version of Xamarin.

Pros

- It has the most compatible feature set with Microsoft .NET and older Xamarin versions.

Cons

- It is not fully integrated with the Apple OSes and is limited to TLS 1.0. It may not be able to connect to secure web servers or cloud services in the future.
- It typically much slower at things like encryption than the native APIs.
- It requires more managed code, thus creating a larger app distributable.

Programmatically setting the `HttpMessageHandler`

In addition to the project-wide configuration shown above, you can also instantiate an `HttpClient` and inject the desired `HttpMessageHandler` through the constructor, as demonstrated in these code snippets:

```
// This will use the default message handler for the application; as
// set in the Project Options for the project.
HttpClient client = new HttpClient();

// This will create an HttpClient that explicitly uses the CFNetworkHandler
HttpClient client = new HttpClient(new CFNetworkHandler());

// This will create an HttpClient that explicitly uses NSUrlSessionHandler
HttpClient client = new HttpClient(new NSUrlSessionHandler());
```

This makes it possible to use a different `HttpMessageHandler` from what is declared in the **Project Options** dialog.

SSL/TLS implementation

SSL (Secure Socket Layer) and its successor, TLS (Transport Layer Security), provide support for HTTP and other network connections via `System.Net.Security.SslStream`. Xamarin.iOS, Xamarin.tvOS or Xamarin.Mac's

`System.Net.Security.SslStream` implementation will call Apple's native SSL/TLS implementation instead of using the managed implementation provided by Mono. Apple's native implementation supports TLS 1.2.

WARNING

The upcoming Xamarin.Mac 4.8 release will only support macOS 10.9 or higher. Previous versions of Xamarin.Mac supported macOS 10.7 or higher, but these older macOS versions lack sufficient TLS infrastructure to support TLS 1.2. To target macOS 10.7 or macOS 10.8, use Xamarin.Mac 4.6 or earlier.

App Transport Security

Apple's *App Transport Security* (ATS) enforces secure connections between internet resources (such as the app's back-end server) and your app. ATS ensures that all internet communications conform to secure connection best practices, thereby preventing accidental disclosure of sensitive information either directly through your app or a library that it is consuming.

Since ATS is enabled by default in apps built for iOS 9, tvOS 9 and OS X 10.11 (El Capitan) and newer, all connections using `NSURLConnection`, `CFURL` or `NSURLSession` will be subject to ATS security requirements. If your connections do not meet these requirements, they will fail with an exception.

Based on your HttpClient Stack and SSL/TLS Implementation selections, you may need to make modifications to your app to work correctly with ATS.

To find out more about ATS, please see our [App Transport Security guide](#).

Known issues

This section will cover known issues with TLS support in Xamarin.iOS.

Project failed to load with error "Requested value AppleTLS wasn't found"

Xamarin.iOS 9.8 introduced some new settings contained the `.csproj` file for a Xamarin.iOS application. These changes may cause problems when the project is opened with older versions of Xamarin.iOS. The following screenshot is a example of the error message that may be displayed in this scenario:



This error is caused by the introduction of the `MtouchTlsProvider` setting to the project file in Xamarin.iOS 9.8. If it is not possible to update to Xamarin.iOS 9.8 (or higher), the work around is to manually edit the `.csproj` file application, remove the `MtouchTlsprovider` element, and then save the changed project file.

The following snippet is an example of what the `MtouchTlsProvider` setting may look like inside a `.csproj` file:

```
<MtouchTlsProvider>Default</MtouchTlsProvider>
```

Related links

- [Transport Layer Security \(TLS\)](#)
- [App Transport Security](#)

Build optimizations

11/2/2020 • 11 minutes to read • [Edit Online](#)

This document explains the various optimizations that are applied at build time for Xamarin.iOS and Xamarin.Mac apps.

Remove UIApplication.EnsureUIThread / NSApplication.EnsureUIThread

Removes calls to `UIApplication.EnsureUIThread` (for Xamarin.iOS) or `NSApplication.EnsureUIThread` (for Xamarin.Mac).

This optimization will change the following type of code:

```
public virtual void AddChildViewController (UIViewController childController)
{
    global::UIKit.UIApplication.EnsureUIThread ();
    // ...
}
```

into the following:

```
public virtual void AddChildViewController (UIViewController childController)
{
    // ...
}
```

This optimization requires the linker to be enabled, and is only applied to methods with the `[BindingImpl (BindingImplOptions.Optimizable)]` attribute.

By default it's enabled for release builds.

The default behavior can be overridden by passing `--optimize=[+|-]remove-uithread-checks` to mtouch/mmp.

Inline IntPtr.Size

Inlines the constant value of `IntPtr.Size` according to the target platform.

This optimization will change the following type of code:

```
if (IntPtr.Size == 8) {
    Console.WriteLine ("64-bit platform");
} else {
    Console.WriteLine ("32-bit platform");
}
```

into the following (when building for a 64-bit platform):

```

if (8 == 8) {
    Console.WriteLine ("64-bit platform");
} else {
    Console.WriteLine ("32-bit platform");
}

```

This optimization requires the linker to be enabled, and is only applied to methods with the `[BindingImpl (BindingImplOptions.Optimizable)]` attribute.

By default it's enabled if targeting a single architecture, or for the platform assembly (`Xamarin.iOS.dll`, `Xamarin.TVOS.dll`, `Xamarin.WatchOS.dll` or `Xamarin.Mac.dll`).

If targeting multiple architectures, this optimization will create different assemblies for the 32-bit version and the 64-bit version of the app, and both versions will have to be included in the app, effectively increasing the final app size instead of decreasing it.

The default behavior can be overridden by passing `--optimize=[+|-]inline-intptr-size` to mtouch/mmp.

Inline NSObject.IsDirectBinding

`NSObject.IsDirectBinding` is an instance property that determines whether a particular instance is of a wrapper type or not (a wrapper type is a managed type that maps to a native type; for instance the managed `UIKit.UIView` type maps to the native `UIView` type - the opposite is a user type, in this case `class MyUIView : UIKit.UIView` would be a user type).

It's necessary to know the value of `IsDirectBinding` when calling into Objective-C, because the value determines which version of `objc_msgSend` to use.

Given only the following code:

```

class UIView : NSObject {
    public virtual string SomeProperty {
        get {
            if (IsDirectBinding) {
                return "true";
            } else {
                return "false"
            }
        }
    }
}

class NSUrl : NSObject {
    public virtual string SomeOtherProperty {
        get {
            if (IsDirectBinding) {
                return "true";
            } else {
                return "false"
            }
        }
    }
}

class MyUIView : UIView {
}

```

We can determine that in `UIView.SomeProperty` the value of `IsDirectBinding` is not a constant and cannot be inlined:

```

void uiView = new UIView ();
Console.WriteLine (uiView.SomeProperty); /* prints 'true' */
void myView = new MyUIView ();
Console.WriteLine (myView.SomeProperty); // prints 'false'

```

However, it's possible to look at all the types in the app and determine that there are no types that inherit from `NSURL`, and it's thus safe to inline the `IsDirectBinding` value to a constant `true`:

```

void myURL = new NSURL ();
Console.WriteLine (myURL.SomeOtherProperty); // prints 'true'
// There's no way to make SomeOtherProperty print anything but 'true', since there are no NSURL subclasses.

```

In particular, this optimization will change the following type of code (this is the binding code for `NSURL.AbsoluteUrl`):

```

if (IsDirectBinding) {
    return Runtime.GetNSObject<NSURL> (global::ObjCRuntime.Messaging.IntPtr_objc_msgSend (this.Handle,
Selector.GetHandle ("absoluteURL")));
} else {
    return Runtime.GetNSObject<NSURL> (global::ObjCRuntime.Messaging.IntPtr_objc_msgSendSuper
(this.SuperHandle, Selector.GetHandle ("absoluteURL")));
}

```

into the following (when it can be determined that there are no subclasses of `NSURL` in the app):

```

if (true) {
    return Runtime.GetNSObject<NSURL> (global::ObjCRuntime.Messaging.IntPtr_objc_msgSend (this.Handle,
Selector.GetHandle ("absoluteURL")));
} else {
    return Runtime.GetNSObject<NSURL> (global::ObjCRuntime.Messaging.IntPtr_objc_msgSendSuper
(this.SuperHandle, Selector.GetHandle ("absoluteURL")));
}

```

This optimization requires the linker to be enabled, and is only applied to methods with the `[BindingImpl (BindingImplOptions.Optimizable)]` attribute.

It is always enabled by default for Xamarin.iOS, and always disabled by default for Xamarin.Mac (because it's possible to dynamically load assemblies in Xamarin.Mac, it's not possible to determine that a particular class is never subclassed).

The default behavior can be overridden by passing `--optimize=[+|-]inline-isdirectbinding` to mtouch/mmp.

Inline Runtime.Arch

This optimization will change the following type of code:

```

if (Runtime.Arch == Arch.DEVICE) {
    Console.WriteLine ("Running on device");
} else {
    Console.WriteLine ("Running in the simulator");
}

```

into the following (when building for device):

```
if (Arch.DEVICE == Arch.DEVICE) {
    Console.WriteLine ("Running on device");
} else {
    Console.WriteLine ("Running in the simulator");
}
```

This optimization requires the linker to be enabled, and is only applied to methods with the `[BindingImpl (BindingImplOptions.Optimizable)]` attribute.

It is always enabled by default for Xamarin.iOS (it's not available for Xamarin.Mac).

The default behavior can be overridden by passing `--optimize=[+|-]inline-runtime-arch` to mtouch.

Dead code elimination

This optimization will change the following type of code:

```
if (true) {
    Console.WriteLine ("Doing this");
} else {
    Console.WriteLine ("Not doing this");
}
```

into:

```
Console.WriteLine ("Doing this");
```

It will also evaluate constant comparisons, like this:

```
if (8 == 8) {
    Console.WriteLine ("Doing this");
} else {
    Console.WriteLine ("Not doing this");
}
```

and determine that the expression `8 == 8` is always true, and reduce it to:

```
Console.WriteLine ("Doing this");
```

This is a powerful optimization when used together with the inlining optimizations, because it can transform the following type of code (this is the binding code for `NFCIso15693ReadMultipleBlocksConfiguration.Range`):

```

NSRange ret;
if (IsDirectBinding) {
    if (Runtime.Arch == Arch.DEVICE) {
        if (IntPtr.Size == 8) {
            ret = global::ObjCRuntime.Messaging.NSRange_objc_msgSend (this.Handle, Selector.GetHandle
("range"));
        } else {
            global::ObjCRuntime.Messaging.NSRange_objc_msgSend_stret (out ret, this.Handle,
Selector.GetHandle ("range"));
        }
    } else if (IntPtr.Size == 8) {
        ret = global::ObjCRuntime.Messaging.NSRange_objc_msgSend (this.Handle, Selector.GetHandle
("range"));
    } else {
        ret = global::ObjCRuntime.Messaging.NSRange_objc_msgSend (this.Handle, Selector.GetHandle
("range"));
    }
} else {
    if (Runtime.Arch == Arch.DEVICE) {
        if (IntPtr.Size == 8) {
            ret = global::ObjCRuntime.Messaging.NSRange_objc_msgSendSuper (this.SuperHandle,
Selector.GetHandle ("range"));
        } else {
            global::ObjCRuntime.Messaging.NSRange_objc_msgSendSuper_stret (out ret, this.SuperHandle,
Selector.GetHandle ("range"));
        }
    } else if (IntPtr.Size == 8) {
        ret = global::ObjCRuntime.Messaging.NSRange_objc_msgSendSuper (this.SuperHandle, Selector.GetHandle
("range"));
    } else {
        ret = global::ObjCRuntime.Messaging.NSRange_objc_msgSendSuper (this.SuperHandle, Selector.GetHandle
("range"));
    }
}
return ret;

```

into this (when building for a 64-bit device, and when also able to ensure there are no

`NFCIso15693ReadMultipleBlocksConfiguration` subclasses in the app):

```

NSRange ret;
ret = global::ObjCRuntime.Messaging.NSRange_objc_msgSend (this.Handle, Selector.GetHandle ("range"));
return ret;

```

The AOT compiler is already able to do eliminate dead code like this, but this optimization is done inside the linker, which means that the linker able to see that there are multiple methods that are not used anymore, and may thus be removed (unless used elsewhere):

- `global::ObjCRuntime.Messaging.NSRange_objc_msgSend_stret`
- `global::ObjCRuntime.Messaging.NSRange_objc_msgSendSuper`
- `global::ObjCRuntime.Messaging.NSRange_objc_msgSendSuper_stret`

This optimization requires the linker to be enabled, and is only applied to methods with the `[BindingImpl (BindingImplOptions.Optimizable)]` attribute.

It is always enabled by default (when the linker is enabled).

The default behavior can be overridden by passing `--optimize=[+|-]dead-code-elimination` to mtouch/mmp.

Optimize calls to BlockLiteral.SetupBlock

The Xamarin.iOS/Mac runtime needs to know the block signature when creating an Objective-C block for a

managed delegate. This might be a fairly expensive operation. This optimization will calculate the block signature at build time, and modify the IL to call a `SetupBlock` method that takes the signature as an argument instead. Doing this avoids the need for calculating the signature at runtime.

Benchmarks show that this speeds up calling a block by a factor of 10 to 15.

It will transform the following [code](#):

```
public static void RequestGuidedAccessSession (bool enable, Action<bool> completionHandler)
{
    // ...
    block_handler.SetupBlock (callback, completionHandler);
    // ...
}
```

into:

```
public static void RequestGuidedAccessSession (bool enable, Action<bool> completionHandler)
{
    // ...
    block_handler.SetupBlockImpl (callback, completionHandler, true, "v@?B");
    // ...
}
```

This optimization requires the linker to be enabled, and is only applied to methods with the `[BindingImpl (BindingImplOptions.Optimizable)]` attribute.

It is enabled by default when using the static registrar (in Xamarin.iOS the static registrar is enabled by default for device builds, and in Xamarin.Mac the static registrar is enabled by default for release builds).

The default behavior can be overridden by passing `--optimize=[+|-]blockliteral-setupblock` to mtouch/mmp.

Optimize support for protocols

The Xamarin.iOS/Mac runtime needs information about how managed types implements Objective-C protocols. This information is stored in interfaces (and attributes on these interfaces), which is not a very efficient format, nor is it linker-friendly.

One example is that these interfaces store information about all protocol members in a `[ProtocolMember]` attribute, which among other things contain references to the parameter types of those members. This means that simply implementing such an interface will make the linker preserve all types used in that interface, even for optional members the app never calls or implements.

This optimization will make the static registrar store any required information in an efficient format that uses little memory that's easy and quick to find at runtime.

It will also teach the linker that it does not necessarily need to preserve these interfaces, nor any of the related attributes.

This optimization requires both the linker and the static registrar to be enabled.

On Xamarin.iOS this optimization is enabled by default when both the linker and the static registrar are enabled.

On Xamarin.Mac this optimization is never enabled by default, because Xamarin.Mac supports loading assemblies dynamically, and those assemblies might not have been known at build time (and thus not optimized).

The default behavior can be overridden by passing `--optimize=-register-protocols` to mtouch/mmp.

Remove the dynamic registrar

Both the Xamarin.iOS and the Xamarin.Mac runtime include support for [registering managed types](#) with the Objective-C runtime. It can either be done at build time or at runtime (or partially at build time and the rest at runtime), but if it's completely done at build time, it means the supporting code for doing it at runtime can be removed. This results in a significant decrease in app size, in particular for smaller apps such as extensions or watchOS apps.

This optimization requires both the static registrar and the linker to be enabled.

The linker will attempt to determine if it's safe to remove the dynamic registrar, and if so will try to remove it.

Since Xamarin.Mac supports dynamically loading assemblies at runtime (which were not known at build time), it's impossible to determine at build time whether this is a safe optimization. This means that this optimization is never enabled by default for Xamarin.Mac apps.

The default behavior can be overridden by passing `--optimize=[+|-]remove-dynamic-registrar` to mtouch/mmp.

If the default is overridden to remove the dynamic registrar, the linker will emit warnings if it detects that it's not safe (but the dynamic registrar will still be removed).

Inline Runtime.DynamicRegistrationSupported

Inlines the value of `Runtime.DynamicRegistrationSupported` as determined at build time.

If the dynamic registrar is removed (see the [Remove the dynamic registrar](#) optimization), this is a constant `false` value, otherwise it's a constant `true` value.

This optimization will change the following type of code:

```
if (Runtime.DynamicRegistrationSupported) {
    Console.WriteLine ("do something");
} else {
    throw new Exception ("dynamic registration is not supported");
}
```

into the following when the dynamic registrar is removed:

```
throw new Exception ("dynamic registration is not supported");
```

into the following when the dynamic registrar is not removed:

```
Console.WriteLine ("do something");
```

This optimization requires the linker to be enabled, and is only applied to methods with the `[BindingImpl (BindingImplOptions.Optimizable)]` attribute.

It is always enabled by default (when the linker is enabled).

The default behavior can be overridden by passing `--optimize=[+|-]inline-dynamic-registration-supported` to mtouch/mmp.

Precompute methods to create managed delegates for Objective-C blocks

When Objective-C calls a selector that takes a block as a parameter, and then managed code has overridden that

method, the Xamarin.iOS / Xamarin.Mac runtime needs to create a delegate for that block.

The binding code generated by the binding generator will include a `[BlockProxy]` attribute, which specifies the type with a `Create` method that can do this.

Given the following Objective-C code:

```
@interface ObjCBlockTester : NSObject {
}
-(void) classCallback: (void (^)(()completionHandler;
-(void) callClassCallback;
@end

@implementation ObjCBlockTester
-(void) classCallback: (void (^)(()completionHandler
{
}

-(void) callClassCallback
{
    [self classCallback: ^{
        NSLog (@"called!");
    }];
}
@end
```

and the following binding code:

```
[BaseType (typeof (NSObject))]
interface ObjCBlockTester
{
    [Export ("classCallback")]
    void ClassCallback (Action completionHandler);
}
```

the generator will produce:

```
[Register("ObjCBlockTester", true)]
public unsafe partial class ObjCBlockTester : NSObject {
    // unrelated code...

    [Export ("callClassCallback")]
    [BindingImpl (BindingImplOptions.GeneratedCode | BindingImplOptions.Optimizable)]
    public virtual void CallClassCallback ()
    {
        if (IsDirectBinding) {
            ApiDefinition.Messaging.void_objc_msgSend (this.Handle, Selector.GetHandle
("callClassCallback"));
        } else {
            ApiDefinition.Messaging.void_objc_msgSendSuper (this.SuperHandle, Selector.GetHandle
("callClassCallback"));
        }
    }

    [Export ("classCallback")]
    [BindingImpl (BindingImplOptions.GeneratedCode | BindingImplOptions.Optimizable)]
    public unsafe virtual void ClassCallback ([BlockProxy (typeof (Trampolines.NIDActionArity1V0))]
System.Action completionHandler)
    {
        // ...
    }
}
```

```

static class Trampolines
{
    [UnmanagedFunctionPointerAttribute (CallingConvention.Cdecl)]
    [UserDelegateType (typeof (System.Action))]
    internal delegate void DActionArity1V0 (IntPtr block);

    static internal class SDActionArity1V0 {
        static internal readonly DActionArity1V0 Handler = Invoke;

        [MonoPInvokeCallback (typeof (DActionArity1V0))]
        static unsafe void Invoke (IntPtr block) {
            var descriptor = (BlockLiteral *) block;
            var del = (System.Action) (descriptor->Target);
            if (del != null)
                del (obj);
        }
    }

    internal class NIDActionArity1V0 {
        IntPtr blockPtr;
        DActionArity1V0 invoker;

        [Preserve (Conditional=true)]
        [BindingImpl (BindingImplOptions.GeneratedCode | BindingImplOptions.Optimizable)]
        public unsafe NIDActionArity1V0 (BlockLiteral *block)
        {
            blockPtr = _Block_copy ((IntPtr) block);
            invoker = block->GetDelegateForBlock<DActionArity1V0> ();
        }

        [Preserve (Conditional=true)]
        [BindingImpl (BindingImplOptions.GeneratedCode | BindingImplOptions.Optimizable)]
        ~NIDActionArity1V0 ()
        {
            _Block_release (blockPtr);
        }

        [Preserve (Conditional=true)]
        [BindingImpl (BindingImplOptions.GeneratedCode | BindingImplOptions.Optimizable)]
        public unsafe static System.Action Create (IntPtr block)
        {
            if (block == IntPtr.Zero)
                return null;
            if (BlockLiteral.IsManagedBlock (block)) {
                var existing_delegate = ((BlockLiteral *) block)->Target as System.Action;
                if (existing_delegate != null)
                    return existing_delegate;
            }
            return new NIDActionArity1V0 ((BlockLiteral *) block).Invoke;
        }

        [Preserve (Conditional=true)]
        [BindingImpl (BindingImplOptions.GeneratedCode | BindingImplOptions.Optimizable)]
        unsafe void Invoke ()
        {
            invoker (blockPtr);
        }
    }
}

```

When Objective-C calls `[objcBlockTester callClassCallback]`, the Xamarin.iOS / Xamarin.Mac runtime will look at the `[BlockProxy (typeof (Trampolines.NIDActionArity1V0))]` attribute on the parameter. It will then look up the `Create` method on that type, and call that method to create the delegate.

This optimization will find the `Create` method at build time, and the static registrar will generate code that looks

up the method at runtime using the metadata tokens instead using the attribute and reflection (this is much faster, and also allows the linker to remove the corresponding runtime code, making the app smaller).

If mmp/mtouch is unable to find the `Create` method, then a MT4174/MM4174 warning will be shown, and the lookup will be performed at runtime instead. The most probable cause is manually written binding code without the required `[BlockProxy]` attributes.

This optimization requires the static registrar to be enabled.

It is always enabled by default (as long as the static registrar is enabled).

The default behavior can be overridden by passing `--optimize=[+|-]static-delegate-to-block-lookup` to mtouch/mmp.

Under the hood in Xamarin.Mac

10/28/2019 • 2 minutes to read • [Edit Online](#)

Ahead of time compilation (AOT)

Ahead of time (AOT) compilation is a powerful optimization technique for improving startup performance. However, it also affects your build time, application size, and program execution in profound ways, so it's worthwhile understanding how it works.

Mac architecture

Xamarin.Mac's relationship to Objective-C, including concepts such as compilation, selectors, registrars, app launch, and the generator.

Xamarin.Mac registrar

Xamarin.Mac bridges the gap between the managed world and Cocoa's runtime, allowing managed classes to call unmanaged Objective-C classes and be called back when events occur. The work required to perform this "magic" is handled by the registrar, but understanding what's going on "under the hood" can sometimes be helpful.

Xamarin.Mac ahead of time compilation

10/28/2019 • 3 minutes to read • [Edit Online](#)

Overview

Ahead of time (AOT) compilation is a powerful optimization technique for improving startup performance. However, it also affects your build time, application size, and program execution in profound ways. To understand the tradeoffs it imposes, we're going to dive a bit into the compilation and execution of an application.

Code written in managed languages, such as C# and F#, is compiled to an intermediate representation called IL. This IL, stored in your library and program assemblies, is relatively compact and portable between processor architectures. IL, however, is only an intermediate set of instructions and at some point that IL will need to be translated into machine code specific to the processor.

There are two points in which this processing can be done:

- **Just in time (JIT)** – During startup and execution of your application the IL is compiled in memory to machine code.
- **Ahead of time (AOT)** – During build the IL is compiled and written out to native libraries and stored within your application bundle.

Each option has a number of benefits and tradeoffs:

- **JIT**
 - **Startup Time** – JIT compilation must be done on startup. For most applications this is on the order of 100ms, but for large applications this time can be significantly more.
 - **Execution** – As the JIT code can be optimized for the specific processor being used, slightly better code can be generated. In most applications this is a few percentage points faster at most.
- **AOT**
 - **Startup Time** – Loading pre-compiled dylibs is significantly faster than JIT assemblies.
 - **Disk Space** – Those dylibs can take a significant amount of disk space however. Depending on which assemblies are AOTed, it can double or more the size of the code portion of your application.
 - **Build Time** – AOT compilation is significantly slower than JIT and will slow builds using it. This slowdown can range from seconds up to a minute or more, depending on the size and number of assemblies compiled.
 - **Obfuscation** – As the IL, which is significantly easier to reverse engineer than machine code, is not necessarily required it can be stripped to help obfuscate sensitive code. This requires the "Hybrid" option described below.

Enabling AOT

AOT options will be added to the Mac Build pane in a future update. Until then, enabling AOT requires passing a command line argument via the "Additional mmp arguments" field in Mac Build. The options are as follows:

```
--aot[=VALUE]      Specify assemblies that should be AOT compiled
                  - none - No AOT (default)
                  - all - Every assembly in MonoBundle
                  - core - Xamarin.Mac, System, mscorelib
                  - sdk - Xamarin.Mac.dll and BCL assemblies
                  - |hybrid after option enables hybrid AOT which
                    allows IL stripping but is slower (only valid
                    for 'all')
                  - Individual files can be included for AOT via +
                    FileName.dll and excluded via -FileName.dll

Examples:
--aot:all,-MyAssembly.dll
--aot:core,+MyOtherAssembly.dll,-mscorlib.dll
```

Hybrid AOT

During execution of a macOS application the runtime defaults to using machine code loaded from the native libraries produced by AOT compilation. There are, however, some areas of code such as trampolines, where JIT compilation can produce significantly more optimized results. This requires the managed assemblies IL to be available. On iOS, applications are restricted from any use of JIT compilation; those section of code are AOT compiled as well.

The hybrid option instructs the compiler to both compile these section (like iOS) but also to assume that the IL will not be available at runtime. This IL can then be stripped post build. As noted above, the runtime will be forced to use less optimized routines in some places.

Further considerations

The negative consequences of AOT scale with the sizes and number of assemblies processed. The Full [target framework](#) for example contains a significantly larger Base Class Library (BCL) than Modern, and thus AOT will take significantly longer and produce larger bundles. This is compounded by the Full target framework's incompatibility with Linking, which strips out unused code. Consider moving your application to Modern and enabling Linking for the best results.

One additional benefit of AOT comes with improved interactions with native debugging and profiling toolchains. Since a vast majority of the codebase will be compiled ahead of time, it will have function names and symbols that are easier to read inside native crash reports, profiling, and debugging. JIT generated functions do not have these names and often show up as unnamed hex offsets that are very difficult to resolve.

Available Assemblies

11/2/2020 • 2 minutes to read • [Edit Online](#)

Xamarin.iOS, Xamarin.Android, and Xamarin.Mac all ship with over a dozen assemblies. Just as Silverlight is an extended subset of the desktop .NET assemblies, Xamarin platforms is also an extended subset of several Silverlight and desktop .NET assemblies.

Xamarin platforms are not ABI compatible with existing assemblies compiled for a different profile. You must recompile your source code to generate assemblies targeting the correct profile (just as you need to recompile source code to target Silverlight and .NET 3.5 separately).

Xamarin.Mac applications can be compiled in three modes: one that uses Xamarin's curated Mobile Profile, the Xamarin.Mac .NET 4.5 Framework which allows you target existing full desktop assemblies, and an unsupported one that uses the .NET API found in a system Mono installation. For more information, please see our [Target Frameworks](#) documentation.

.NET Standard Libraries

In addition to the iOS, Android, and Mac bindings, Xamarin projects can consume [.NET Standard libraries](#).

Portable Class Libraries

Xamarin projects can also consume [.NET Portable Class Libraries](#), although this technology is being deprecated in favor of .NET Standard.

Supported Assemblies

These are the assemblies available in the **Reference Manager > Assemblies > Framework** (Visual Studio 2017) and **Edit References > Packages** (Visual Studio for Mac), and their compatibility with Xamarin platforms.

ASSEMBLY	API COMPATIBILITY	XAMARIN IOS	XAMARIN ANDROID	XAMARIN MAC
FSharp.Core.dll		✓	✓	✓
I18N.dll	Includes CJK, MidEast, Other, Rare, West	✓	✓	✓
Microsoft.CSharp.dll		✓	✓	✓
Mono.CSharp.dll		✓	✓	✓
Mono.Data.Sqlite.dll	ADO.NET provider for SQLite; see limitations.	✓	✓	✓

ASSEMBLY	API COMPATIBILITY	XAMARIN IOS	XAMARIN ANDROID	XAMARIN MAC
Mono.Data.Tds.dll	TDS Protocol support; used for System.Data.SqlClient support within System.Data .	✓	✓	✓
Mono.Dynamic.Interpreter.dll		✓		
Mono.Security.dll	Cryptographic APIs.	✓	✓	✓
monotouch.dll	This assembly contains the C# binding to the CocoaTouch API. This is only available within Classic iOS Projects.	✓		
MonoTouch.Dialog-1.dll		✓		
MonoTouch.NUnitLite.dll		✓		
mscorlib.dll	Silverlight	✓	✓	✓
OpenTK-1.0.dll	The OpenGL/OpenAL object oriented APIs, extended to provide iPhone device support.	✓	✓	✓

ASSEMBLY	API COMPATIBILITY	XAMARIN IOS	XAMARIN ANDROID	XAMARIN MAC
System.dll	Silverlight, plus types from the following namespaces: System.Collections.Specialized System.ComponentModel System.ComponentModel.Design System.Diagnostics System.IO System.IO.Compression System.IO.Compression.FileSystem System.Net System.Net.Cache System.Net.Mail System.Net.Mime System.Net.NetworkInformation System.Net.Security System.Net.Sockets System.Runtime.InteropServices System.Runtime.Versioning System.Security.AccessControl System.Security.Authentication System.Security.Cryptography System.Security.Permissions System.Threading System.Timers	✓	✓	✓
System.ComponentModel.Composition.dll		✓	✓	✓
System.ComponentModel.DataAnnotations.dll		✓	✓	✓
System.Core.dll	Silverlight	✓	✓	✓
System.Data.dll	.NET 3.5 , with some functionality removed.	✓	✓	✓
System.Data.Services.Client.dll	Full oData client.	✓	✓	✓
System.IO.Compression		✓	✓	✓

ASSEMBLY	API COMPATIBILITY	XAMARIN IOS	XAMARIN ANDROID	XAMARIN MAC
System.IO.Compression.FileSystem		✓	✓	✓
System.Json.dll	Silverlight	✓	✓	✓
System.Net.Http.dll		✓	✓	✓
System.Numerics.dll		✓	✓	✓
System.Runtime.Serialization.dll	Silverlight	✓	✓	✓
System.ServiceModel.dll	WCF stack as present in Silverlight	✓	✓	✓
System.ServiceModel.Internals.dll		✓	✓	✓
System.ServiceModel.Web.dll	Silverlight, plus types from the following namespaces: System System.ServiceModel.Channels System.ServiceModel.Description System.ServiceModel.Web	✓	✓	✓
System.Transactions.dll	.NET 3.5; part of System.Data support.	✓	✓	✓
System.Web.Services.dll	Basic Web services from the .NET 3.5 profile, with the server features removed.	✓	✓	✓
System.Windows.dll		✓	✓	✓
System.Xml.dll	.NET 3.5	✓	✓	✓
System.Xml.Linq.dll	.NET 3.5	✓	✓	✓
System.Xml.Serialization.dll		✓	✓	✓
Xamarin.iOS.dll	This assembly contains the C# binding to the CocoaTouch API. This is only used in Unified iOS Projects.	✓		

ASSEMBLY	API COMPATIBILITY	XAMARIN IOS	XAMARIN ANDROID	XAMARIN MAC
Java.Interop.dll			✓	
Mono.Android.dll			✓	
Mono.Android.Export.dll			✓	
Mono.Posix.dll			✓	
System.EnterpriseServices.dll			✓	
Xamarin.Android.NUnitLite.dll			✓	
Mono.CompilerServices.SymbolWriter.dll	For compiler writers.			✓
Xamarin.Mac.dll				✓
System.Drawing.dll	System.Drawing is not supported in the Unified API for the Xamarin.Mac, .NET 4.5, or Mobile frameworks. System.Drawing support can be added to iOS and macOS using the sysdrawing-coregraphics library	✓		✓

Xamarin.Mac architecture

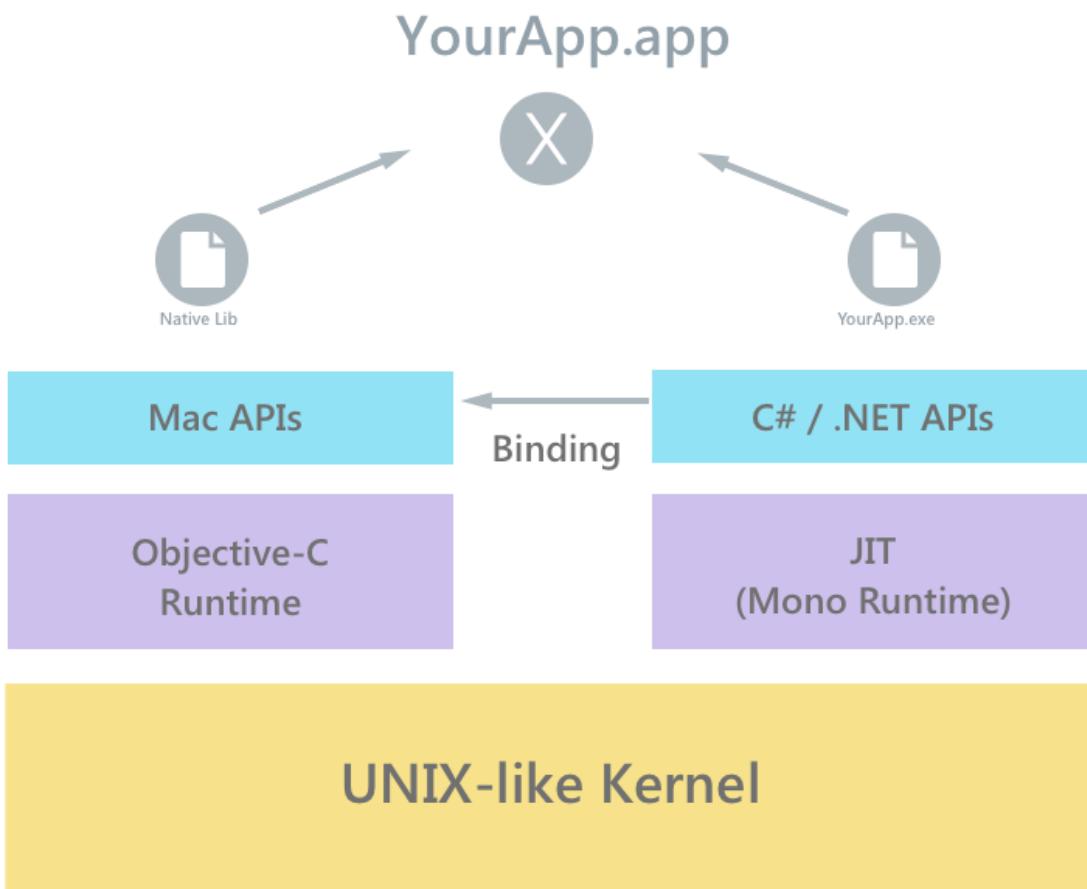
10/28/2019 • 6 minutes to read • [Edit Online](#)

This guide explores Xamarin.Mac and its relationship to Objective-C at a low level. It explains concepts such as compilation, selectors, registrars, app launch, and the generator.

Overview

Xamarin.Mac applications run within the Mono execution environment, and use Xamarin's compiler to compile down to Intermediate Language (IL), which is then Just-in-Time (JIT) compiled to native code at run-time. This runs side-by-side with the Objective-C Runtime. Both runtime environments run on top of a UNIX-like kernel, specifically XNU, and expose various APIs to the user code allowing developers to access the underlying native or managed system.

The diagram below shows a basic overview of this architecture:



Native and managed code

When developing for Xamarin, the terms *native* and *managed* code are often used. Managed code is code that has its execution managed by the .NET Framework Common Language Runtime, or in Xamarin's case: the Mono Runtime.

Native code is code that will run natively on the specific platform (for example, Objective-C or even AOT compiled code, on an ARM chip). This guide explores how your managed code is compiled to native code, and explains how a Xamarin.Mac application works, making full use of Apple's Mac APIs through the use of bindings, while also having access to .NET's BCL and a sophisticated language such as C#.

Requirements

The following is required to develop a macOS application with Xamarin.Mac:

- A Mac running macOS Sierra (10.12) or greater.
- The latest version of Xcode (installed from the [App Store](#))
- The latest version of Xamarin.Mac and Visual Studio for Mac

Running Mac applications created with Xamarin.Mac have the following system requirements:

- A Mac running Mac OS X 10.7 or greater.

Compilation

When you compile any Xamarin platform application, the Mono C# (or F#) compiler will run and will compile your C# and F# code into Microsoft Intermediate Language (MSIL or IL). Xamarin.Mac then uses a *Just in Time (JIT)* compiler at runtime to compile native code, allowing execution on the correct architecture as needed.

This is in contrast to Xamarin.iOS which uses AOT compilation. When using the AOT compiler, all assemblies and all methods within them are compiled at build time. With JIT, compilation happens on demand only for the methods that are executed.

With Xamarin.Mac applications, Mono is usually embedded into the app bundle (and referred to as **Embedded Mono**). When using the Classic Xamarin.Mac API, the application could instead use **System Mono**, however, this is unsupported in the Unified API. System Mono refers to Mono that has been installed in the operating system. On application launch, the Xamarin.Mac app will use this.

Selectors

With Xamarin, we have two separate ecosystems, .NET and Apple, that we need to bring together to seem as streamlined as possible, to ensure that the end goal is a smooth user experience. We have seen in the section above how the two runtimes communicate, and you may very well have heard of the term ‘bindings’ which allows the native Mac APIs to be used in Xamarin. Bindings are explained in depth in the [Objective-C binding documentation](#), so for now, let’s explore how Xamarin.Mac works under the hood.

First, there has to be a way to expose Objective-C to C#, which is done via Selectors. A selector is a message which is sent to an object or class. With Objective-C this is done via the `objc_msgSend` functions. For more information on using Selectors, refer to the iOS [Objective-C Selectors](#) guide. There also has to be a way to expose managed code to Objective-C, which is more complicated due to the fact that Objective-C doesn’t know anything about the managed code. To get around this, we use a **registrar**. This explained in more detail in the next section.

Registrar

As mentioned above, the registrar is code that exposes managed code to Objective-C. It does this by creating a list of every managed class that derives from `NSObject`:

- For all classes that are not wrapping an existing Objective-C class, it creates a new Objective-C class with Objective-C members mirroring all the managed members that have an `[Export]` attribute.
- In the implementations for each Objective-C member, code is added automatically to call the mirrored managed member.

The pseudo-code below shows an example of how this is done:

C# (managed code):

```
class MyViewController : UIViewController{
    [Export ("myFunc")]
    public void MyFunc ()
    {
    }
}
```

Objective-C (native code):

```
@interface MyViewController : UIViewController
- (void)myFunc;
@end

@implementation MyViewController
- (void)myFunc {
    // Code to call the managed C# MyFunc method in MyViewController
}
@end
```

The managed code can contain the attributes, `[Register]` and `[Export]`, that the registrar uses to know that the object needs to be exposed to Objective-C. The `[Register]` attribute is used to specify the name of the generated Objective-C class in case the default generated name is not suitable. All classes derived from `NSObject` are automatically registered with Objective-C. The required `[Export]` attribute contains a string, which is the selector used in the generated Objective-C class.

There are two types of registrars used in Xamarin.Mac – dynamic and static:

- Dynamic registrars – This is the default registrar for all Xamarin.Mac builds. The dynamic registrar does the registration of all types in your assembly at runtime. It does this by using functions provided by Objective-C's runtime API. The dynamic registrar therefore has a slower startup, but a faster build time. Native functions (usually in C), called trampolines, are used as method implementations when using the dynamic registrars. They vary between different architectures.
- Static registrars – The static registrar generates Objective-C code during the build, which is then compiled into a static library and linked into the executable. This allows for a quicker startup, but takes longer during build time.

Application launch

Xamarin.Mac startup logic will differ depending on whether embedded or system Mono is used. To view the code and steps for Xamarin.Mac application launch, refer to the [launch header](#) file in the `xamarin-macos` public repo.

Generator

Xamarin.Mac contains definitions for every Mac API. You can browse through any of these on the [MacOS github repo](#). These definitions contain interfaces with attributes, as well as any necessary methods and properties. For example, the following code is used to define an `NSBox` in the [AppKit namespace](#). Notice that it is an interface with a number of methods and properties:

```
[ BaseType (typeof (NSView)) ]
public interface NSBox {

    ...

    [Export ("borderRect")]
    CGRect BorderRect { get; }

    [Export ("titleRect")]
    CGRect TitleRect { get; }

    [Export ("titleCell")]
    NSObject TitleCell { get; }

    [Export ("sizeToFit")]
    void SizeToFit ();

    [Export ("contentViewMargins")]
    CGSize ContentViewMargins { get; set; }

    [Export ("setFrameFromContentFrame:")]
    void SetFrameFromContentFrame (CGRect contentFrame);

    ...

}
```

The Generator, called `bmac` in Xamarin.Mac, takes these definition files and uses .NET tools to compile them into a temporary assembly. However, this temporary assembly is not useable to call Objective-C code. The generator then reads the temporary assembly and generates C# code that can be used at runtime. This is why, for example, if you add a random attribute to your definition .cs file, it won't show up in the outputted code. The generator doesn't know about it, and therefore `bmac` doesn't know to look for it in the temporary assembly to output it.

Once the Xamarin.Mac.dll has been created, the packager, `mmp`, will bundle all the components together.

At a high level, it achieves this by executing the following tasks:

- Create an app bundle structure.
- Copy in your managed assemblies.
- If linking is enabled, run the managed linker to optimize your assemblies by removing unused parts.
- Create a launcher application, linking in the launcher code talked about along with the registrar code if in static mode.

This is then run as part of the user build process that compiles user code into an assembly that references Xamarin.Mac.dll and runs `mmp` to make it a package

For more detailed information on the linker and how it is used, refer to the iOS [Linker](#) guide.

Summary

This guide looked at compilation of Xamarin.Mac apps, and explored Xamarin.Mac and its relationship to Objective-C.

How Xamarin.Mac works

11/2/2020 • 7 minutes to read • [Edit Online](#)

Most of the time the developer will never have to worry about the internal "magic" of Xamarin.Mac, however, having a rough understanding of how things works under the hood will help in both interpreting existing documentation with a C# lens and debugging issues when they arise.

In Xamarin.Mac, an application bridges two worlds: There is the Objective-C based runtime containing instances of native classes (`NSString`, `NSApplication`, etc) and there is the C# runtime containing instances of managed classes (`System.String`, `HttpClient`, etc). In between these two worlds, Xamarin.Mac creates a two way bridge so an app can call methods (selectors) in Objective-C (such as `NSApplication.Init`) and Objective-C can call the app's C# methods back (like methods on an app delegate). In general, calls into Objective-C are handled transparently via P/Invokes and some runtime code Xamarin provides.

Exposing C# classes / methods to Objective-C

However, for Objective-C to call back into an app's C# objects, they need to be exposed in a way that Objective-C can understand. This is done via the `Register` and `Export` attributes. Take the following example:

```
[Register ("MyClass")]
public class MyClass : NSObject
{
    [Export ("init")]
    public MyClass ()
    {

    }

    [Export ("run")]
    public void Run ()
    {
    }
}
```

In this example, the Objective-C runtime will now know about a class called `MyClass` with selectors called `init` and `run`.

In most cases, this is an implementation detail that the developer can ignore, as most callbacks an app receives will be either via overridden methods on `base` classes (such as `AppDelegate`, `Delegates`, `DataSources`) or on `Actions` passed into APIs. In all of those cases, `Export` attributes are not necessary in the C# code.

Constructor runthrough

In many cases, the developer will need to expose the app's C# classes construction API to the Objective-C runtime so it can be instantiated from places such as when called in Storyboard or XIB files. Here are the five most common constructors used in Xamarin.Mac apps:

```

// Called when created from unmanaged code
public CustomView (IntPtr handle) : base (handle)
{
    Initialize ();
}

// Called when created directly from a XIB file
[Export ("initWithCoder:")]
public CustomView (NSCoder coder) : base (coder)
{
    Initialize ();
}

// Called from C# to instance NSView with a Frame (initWithFrame)
public CustomView (CGRect frame) : base (frame)
{
}

// Called from C# to instance NSView without setting the frame (init)
public CustomView () : base ()
{
}

// This is a special case constructor that you call on a derived class when the derived called has an
[Export] constructor.
// For example, if you call init on NSString then you don't want to call init on NSObject.
public CustomView () : base (NSObjectFlag.Empty)
{
}

```

In general, the developer should leave the `IntPtr` and `NSCoder` constructors that are generated when creating some types such as custom `NSViews` alone. If Xamarin.Mac needs to call one of these constructors in response to an Objective-C runtime request and you've removed it, the app will crash inside native code and it may be difficult to figure out exactly the issue.

Memory management and cycles

Memory management in Xamarin.Mac is in many ways very similar to Xamarin.iOS. It also is a complex topic, one beyond the scope of this document. Please read the [Memory and Performance Best Practices](#).

Ahead of time compilation

Typically, .NET applications do not compile down to machine code when they are built, instead they compile to an intermediate layer called IL code that gets *Just-In-Time* (JIT) compiled to machine code when the app is launched.

The time that it takes the mono runtime to JIT compile this machine code can slow the launch of a Xamarin.Mac app by up to 20%, as it takes time for the necessary machine code to be generated.

Because of limitations imposed by Apple on iOS, JIT compilation of the IL code is not available to Xamarin.iOS. As a result, all Xamarin.iOS app are full *Ahead-Of-Time* (AOT) compiled to machine code during the build cycle.

New to Xamarin.Mac is the ability to AOT the IL code during the app build cycle, just like Xamarin.iOS can. Xamarin.Mac uses a *Hybrid* AOT approach that compiles a majority of the needed machine code, but allows the runtime to compile needed trampolines and the flexibility to continue to support `Reflection.Emit` (and other use cases that currently work on Xamarin.Mac).

There are two major areas where AOT can help a Xamarin.Mac app:

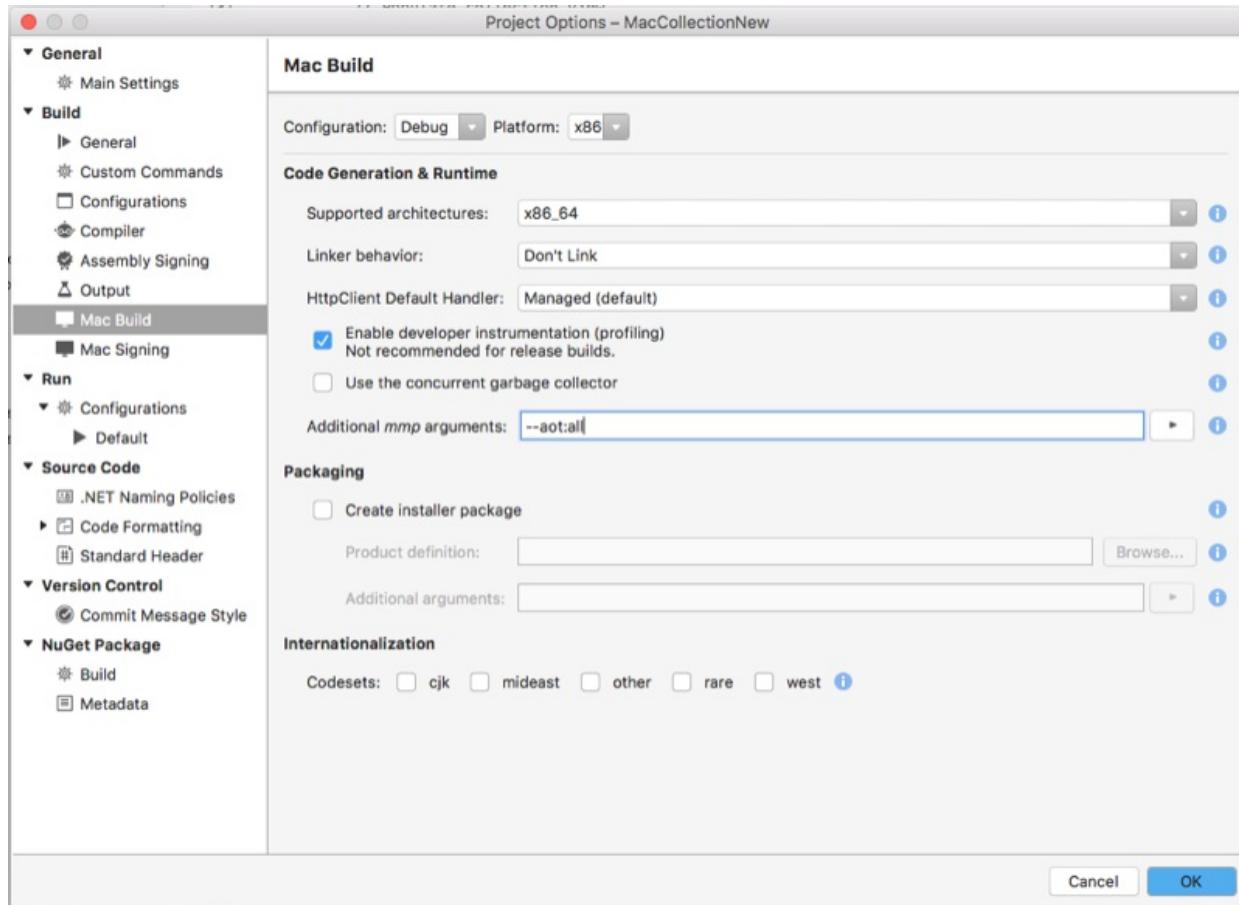
- **Better "native" crash logs** - If a Xamarin.Mac application crashes in native code, which is common

occurrence when making invalid calls into Cocoa APIs (such as sending a `null` into a method that doesn't accept it), native crash logs with JIT frames are difficult to analyze. Since the JIT frames do not have debug information, there will be multiple lines with hex offsets and no clue what was going on. AOT generates "real" named frames and the traces are much easier to read. This also means Xamarin.Mac app will interact better with native tools such as **lldb** and **Instruments**.

- **Better launch time performance** - For large Xamarin.Mac applications, with a multiple second startup time, JIT compiling all of the code can take a significant amount of time. AOT does this work up front.

Enabling AOT compilation

AOT is enabled in Xamarin.Mac by double-clicking the **Project Name** in the **Solution Explorer**, navigating to **Mac Build** and adding `--aot:[options]` to the **Additional mmp arguments:** field (where `[options]` is one or more options to control the AOT type, see below). For example:



IMPORTANT

Enabling AOT compilation dramatically increases build time, sometimes up to several minutes, but it can improve app launch times by an average of 20%. As a result, AOT compilation should only be enabled on **Release** builds of a Xamarin.Mac app.

Aot compilation options

There are several different options that can be adjusted when enabling AOT compilation on a Xamarin.Mac app:

- `none` - No AOT compilation. This is the default setting.
- `a11` - AOT compiles every assembly in the MonoBundle.
- `core` - AOT compiles the `Xamarin.Mac`, `System` and `mscorlib` assemblies.
- `sdk` - AOT compiles the `Xamarin.Mac` and Base Class Libraries (BCL) assemblies.
- `|hybrid` - Adding this to one of the above options enables hybrid AOT which allows for IL stripping, but will result in longer compile times.

- `+` - Includes a single file for AOT compilation.
- `-` - Removes a single file from AOT compilation.

For example, `--aot:all,-MyAssembly.dll` would enable AOT compilation on all of the assemblies in the MonoBundle *except* `MyAssembly.dll` and `--aot:core|hybrid,+MyOtherAssembly.dll,-mscorlib.dll` would enable hybrid, code AOT include the `MyOtherAssembly.dll` and excluding the `mscorlib.dll`.

Partial static registrar

When developing a Xamarin.Mac app, minimizing the time between completing a change and testing it can become important to meeting development deadlines. Strategies such as modularization of codebases and unit tests can help to decrease compile times, as they reduce the number of times that an app will require an expensive full rebuild.

Additionally, and new to Xamarin.Mac, *Partial Static Registrar* (as pioneered by Xamarin.iOS) can dramatically reduce the launch times of a Xamarin.Mac app in the **Debug** configuration. Understanding how using the Partial Static Registrar can squeezed out an almost a 5x improvement in debug launch will take a bit of background on what the registrar is, what the difference is between static and dynamic, and what this "partial static" version does.

About the registrar

Under the hood of any Xamarin.Mac application lies the Cocoa framework from Apple and the Objective-C runtime. Building a bridge between this "native world" and the "managed world" of C# is the primary responsibility of Xamarin.Mac. Part of this task is handled by the registrar, which is executed inside `NSApplication.Init ()` method. This is one reason that any use of Cocoa APIs in Xamarin.Mac requires `NSApplication.Init` to be called first.

The registrar's job is to inform the Objective-C runtime of the existence of the app's C# classes that derive from classes such as `NSApplicationDelegate`, `NSView`, `NSWindow`, and `NSObject`. This requires a scan of all types in the app to determine what needs registering and what elements on each type to report.

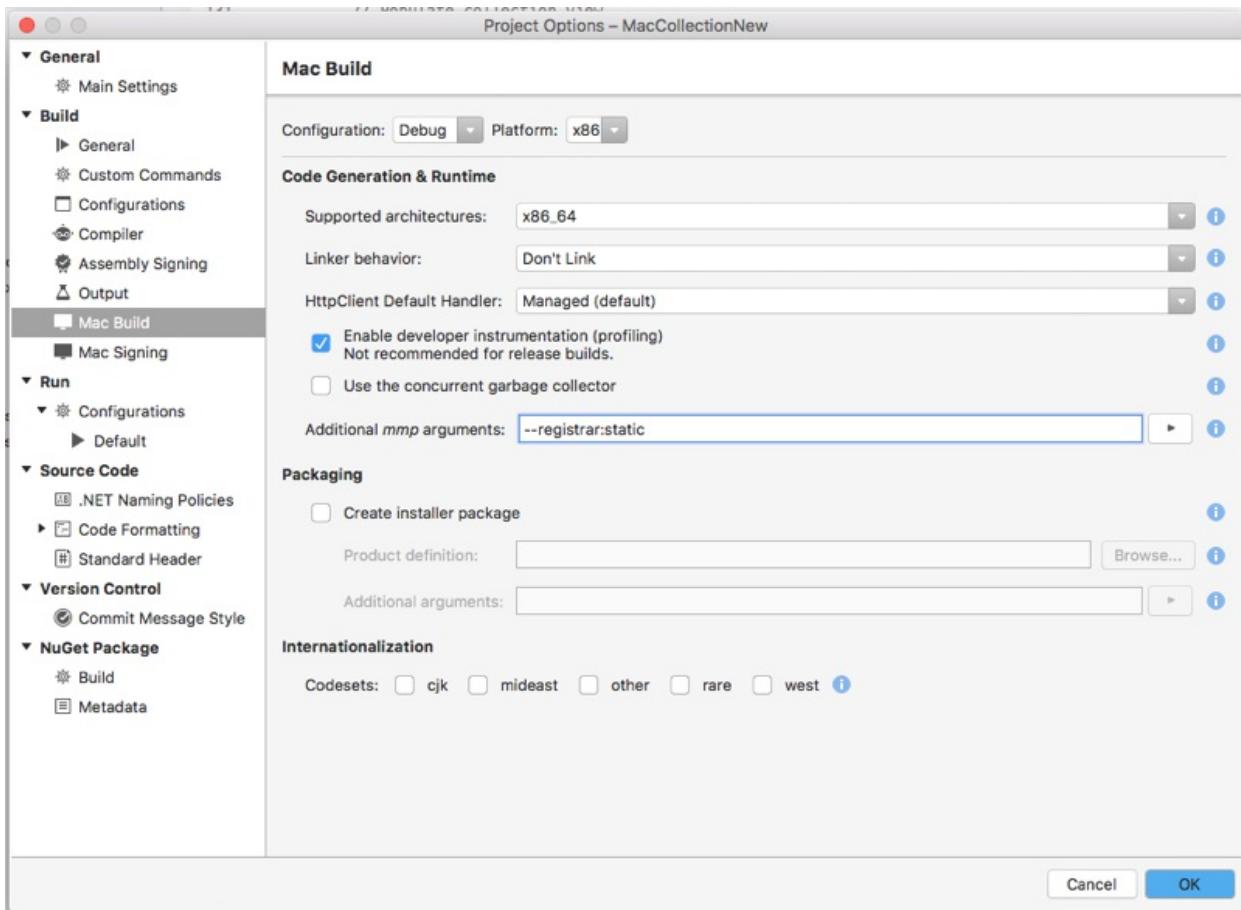
This scan can be done either **dynamically**, at startup of the application with reflection, or **statically**, as a build time step. When picking a registration type, the developer should be aware of the following:

- Static registration can drastically reduce launch times, but can slow down builds times significantly (typically more than double debug build time). This will be the default for **Release** configuration builds.
- Dynamic registration delays this work until application launch and skips code generation, but this additional work can create a noticeable pause (at least two seconds) in application launch . This is especially noticeable in debug configuration builds, which defaults to dynamic registration and whose reflection is slower.

Partial Static Registration, first introduced in Xamarin.iOS 8.13, gives the developer the best of both options. By pre-computing the registration information of every element in `Xamarin.Mac.dll` and shipping this information with Xamarin.Mac in a static library (that only needs to be linked to at build time), Microsoft has removed most of the reflection time of the dynamic registrar while not impacting build time.

Enabling the partial static registrar

The Partial Static Registrar is enabled in Xamarin.Mac by double-clicking the **Project Name** in the **Solution Explorer**, navigating to **Mac Build** and adding `--registrar:static` to the **Additional mmp arguments:** field. For example:



Additional resources

Here are some more detailed explanations of how things work internally:

- [Objective-C Selectors](#)
- [Registrar](#)
- [Xamarin Unified API for iOS and OS X](#)
- [Threading Fundamentals](#)
- [Delegates, Protocols, and Events](#)
- [About `newrefcount`](#)

Xamarin.Mac Frameworks

10/28/2019 • 2 minutes to read • [Edit Online](#)

The following macOS frameworks are supported by Xamarin.Mac:

- AVFoundation
- Accelerate
- Accounts
- AVKit
- AddressBook
- AppKit
- AudioToolbox
- AudioUnit
- AudioUnitWrapper
- CoreAnimation
- CoreBluetooth
- CoreData
- CoreFoundation
- CoreGraphics
- CoreImage
- CoreLocation
- CoreMedia
- CoreMidi
- CoreServices
- CoreText
- CoreVideo
- CoreWlan
- CloudKit
- CFNetwork
- Darwin
- Foundation
- GameKit
- ImageIO
- ImageKit
- LocalAuthentication
- MapKit
- MediaAccessibility
- ObjCRuntime
- OpenAL
- OpenGL
- PdfKit
- QTKit
- QuartzComposer
- QuickLook

- SceneKit
- ScriptingBridge
- Security
- StoreKit
- VideoToolbox
- WebKit

Xamarin.Mac registrar

10/28/2019 • 2 minutes to read • [Edit Online](#)

This document describes the purpose of the Xamarin.Mac registrar and its different usage configurations.

Overview

Xamarin.Mac bridges the gap between the managed (.NET) world and Cocoa's runtime, allowing managed classes to call unmanaged Objective-C classes and be called back when events occur. The work required to perform this "magic" is handled by the registrar and is, in general, hidden from view.

There are performance implications of this registration, specifically on application start up time, and understanding a bit of what's going on "under the hood" can sometimes be helpful.

Configurations

Fundamentally the registrar's job at startup can be separated into two categories:

- Scan every managed class for those deriving from `NSObject` and collect a list of items to be exposed to the Objective-C runtime.
- Register this information with the Objective-C runtime.

Over time, three different registrar configurations have been created to cover different use cases. Each has different build and run time consequences:

- **Dynamic registrar** – During startup, use .NET reflection to scan every loaded type, determine the list of relevant items, and inform the native runtime. This option adds zero time to the build but is very expensive to compute during launch (up to multiple seconds).
- **Static registrar** – During build, compute the set of items to be registered and generate Objective-C code to handle registration. This code is invoked during startup to quickly register all items. Adds a significant pause to build but can cut a significant amount of time from application start.
- **"Partial" static** – A newer "hybrid" approach which brings most of the advantages of both. Since the exports from `Xamarin.Mac.dll` are constant, save a precomputed library to handle their registration and link that in. Use reflection to handle user libraries, but as user libraries export much fewer types than the platform bindings this is often rather quick. A neglectable build time impact and reduces a vast majority of the "cost" of dynamic.

Today partial static is the default for Debug configuration and Static is the default for Release configurations.

There are some scenarios:

- Plugins loaded after launch with classes deriving from `NSObject`
- Dynamically created class instances deriving from `NSObject`

where the registrar is unable to know that it needs to register some type at start. The `ObjCRuntime.Runtime.RegisterAssembly` method is provided to inform the registrar that it has additional types to consider.

Xamarin.Mac troubleshooting

10/28/2019 • 2 minutes to read • [Edit Online](#)

Documents in this section cover features specific to troubleshooting with Xamarin.Mac.

Troubleshooting tips

Troubleshooting tips and tricks.

Errors messages (mmp)

An errors reference guide, showing the most common errors you may experience when building Xamarin.Mac applications.

Xamarin.Mac troubleshooting tips

3/5/2021 • 5 minutes to read • [Edit Online](#)

Overview

Sometimes we all get stuck while working on a project, either on the inability to get an API to work the way we want or in trying to work around a bug. Our goal at Xamarin is for you to be successful in writing your mobile and desktop applications, and we've provided some resources to help.

With any of these resources, there are some steps of preparation you can take to help them solve your issue quickly:

- Determine the root cause of the issue as best as possible to report crashes:
 - "My application crashes" is difficult to diagnose. "My application crashes when I return an empty array to this call" is much easier to work on fixing.
 - "I can't get NSTable to work" is less helpful than "None of the methods on my NSTableDelegate seem to be called in this case."
- If possible provide a small example program showing the issue. Digging through pages of source code looking for the issue takes orders of magnitude more time and effort.
- Knowing what changes you've made to your application to cause an issue to appear can quickly narrow down the source of the problem. Noting if you've recently upgraded versions of Xamarin.Mac, trimming out sections of your application to find the part causing the issue, or testing previous builds to find what change introduced the issue can be very helpful.

What to do when your app crashes with no output

In most cases, the debugger in Visual Studio for Mac will catch exceptions and crashes in your application and help you track down the root cause. However there are some cases where your application will bounce on the dock and then exit with little or no output. These can include:

- Code signing issues.
- Certain mono runtime crashes.
- Some Objective-c exceptions and crashes.
- Some crashes very early the process lifetime.
- Some stack overflows.
- The macOS version listed in your **Info.plist** is newer than your currently installed macOS version or it is invalid.

Debugging these programs can be frustrating, as finding the information necessary can be difficult. Here are a few approaches that may help:

- Ensure that the macOS version listed in the **Info.plist** is the same one as the version of macOS currently installed on the computer.
- Check the Visual Studio for Mac Application Output (**View -> Pads -> Application Output**) for stack traces or output in red from Cocoa that may describe the output.
- Run your application from the command line and look at the output (in the **Terminal** app) by using:

`MyApp.app/Contents/MacOS/MyApp` (where `MyApp` is the name of your application)

- You can increase the output by adding "MONO_LOG_LEVEL" to your command on the command line, for example:

```
MONO_LOG_LEVEL=debug MyApp.app/Contents/MacOS/MyApp
```

- You could attach a native debugger (`lldb`) to your process to see if that provides any more information (this requires a paid license). For example, do the following:

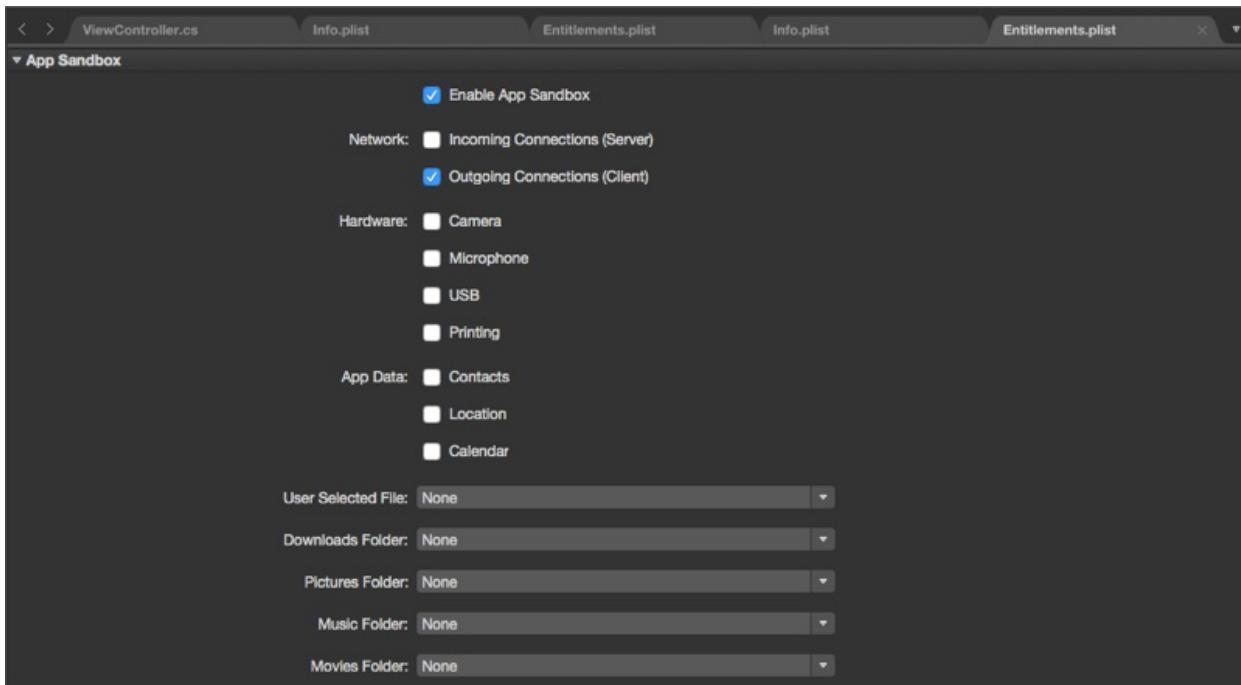
1. Enter `lldb MyApp.app/Contents/MacOS/MyApp` in the Terminal.
 2. Enter `run` in the Terminal.
 3. Enter `c` in the Terminal.
 4. Exit when finished debugging.
- As a last resort, before calling `NSApplication.Init` in your `Main` method (or in other places as required), you could write text to a file in a known location to track down at what step of launch you are running into trouble.

Known issues

The following sections cover known issues and their solutions.

Unable to connect to the debugger in sandboxed apps

The debugger connects to Xamarin.Mac apps through TCP, which means that by default when you enable sandboxing, it is unable to connect to the app, so if you try to run the app without the proper permissions enabled, you get an error "*Unable to connect to the debugger*".

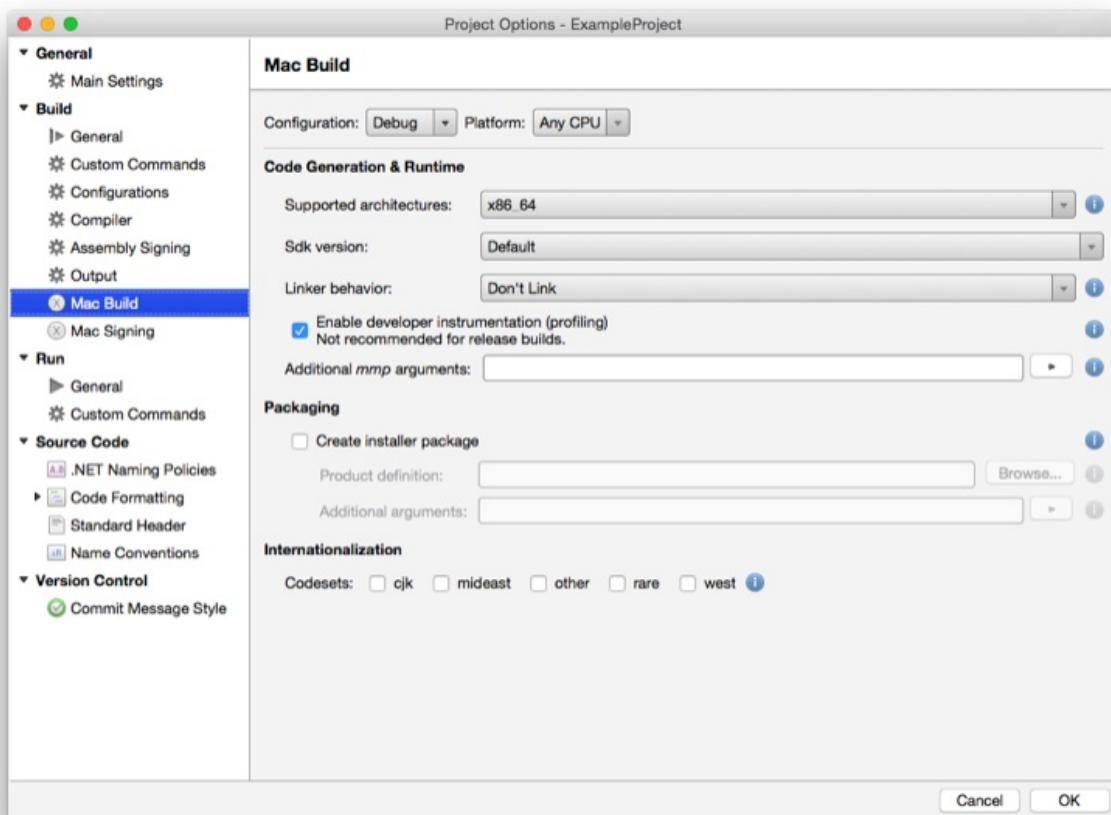


The Allow Outgoing Network Connections (Client) permission is the one required for the debugger, enabling this one will allow debugging normally. Since you can't debug without it, we have updated the `CompileEntitlements` target for `msbuild` to automatically add that permission to the entitlements for any app that is sandboxed for debug builds only. Release builds should use the entitlements specified in the entitlements file, unmodified.

System.NotSupportedException: no data is available for encoding 437

When including 3rd party libraries in your Xamarin.Mac app, you might get an error in the form "System.NotSupportedException: No data is available for encoding 437" when trying to compile and run the app. For example, libraries, such as `Ionic.Zip.ZipFile`, may throw this exception during operation.

This can be solved by opening the options for the Xamarin.Mac project, going to **Mac Build > Internationalization** and checking the **West** internationalization:



Failed to compile (mm5103)

This error is usually caused when a new version of Xcode is released and you have installed the new version but have not yet run it. Before trying to compile with a new version of Xcode, you need to first run that version at least once.

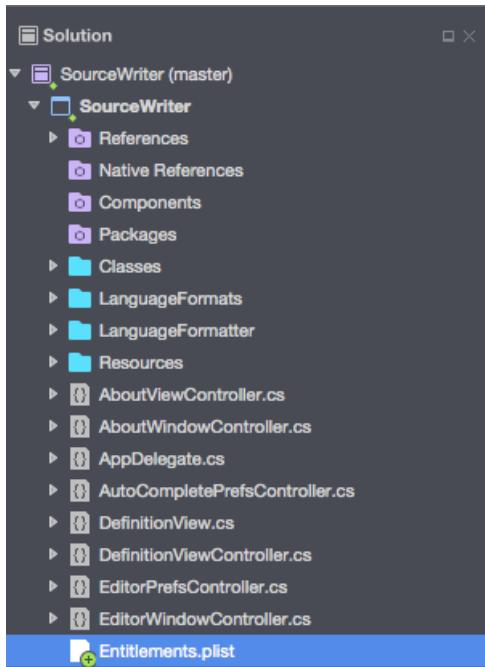
The first time you run a new version of Xcode, it installs several command line tools that are required by Xamarin.Mac. Additionally, you should do a clean build after updating Xcode or your Xamarin.Mac version.

If you cannot resolve this issue, please [file a bug](#).

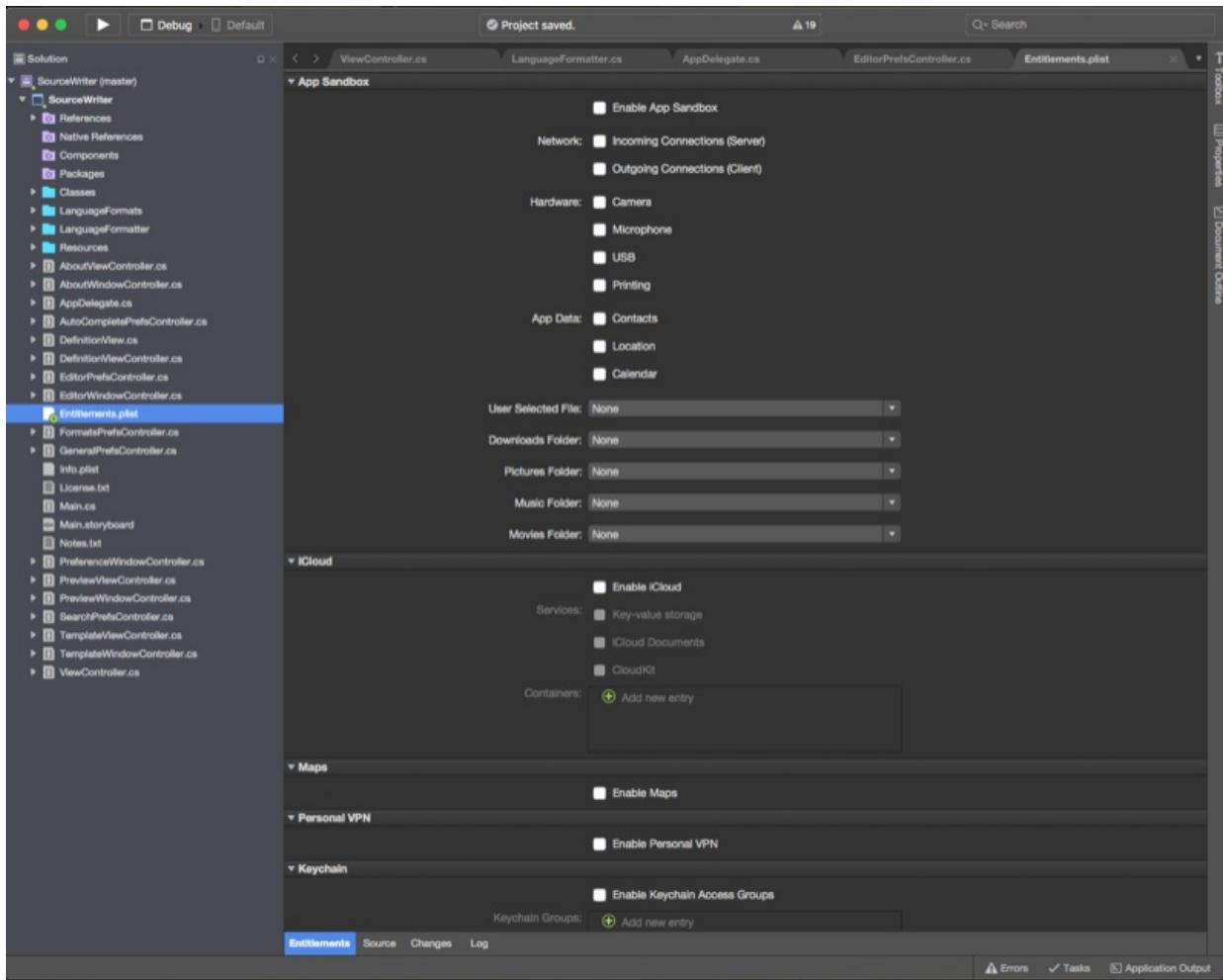
Missing entitlements.plist

The latest version of Visual Studio for Mac has removed the Entitlements section from the **Info.plist** editor and placed it in separate **Entitlements.plist** editor (for better cross-platform support with Xamarin.iOS).

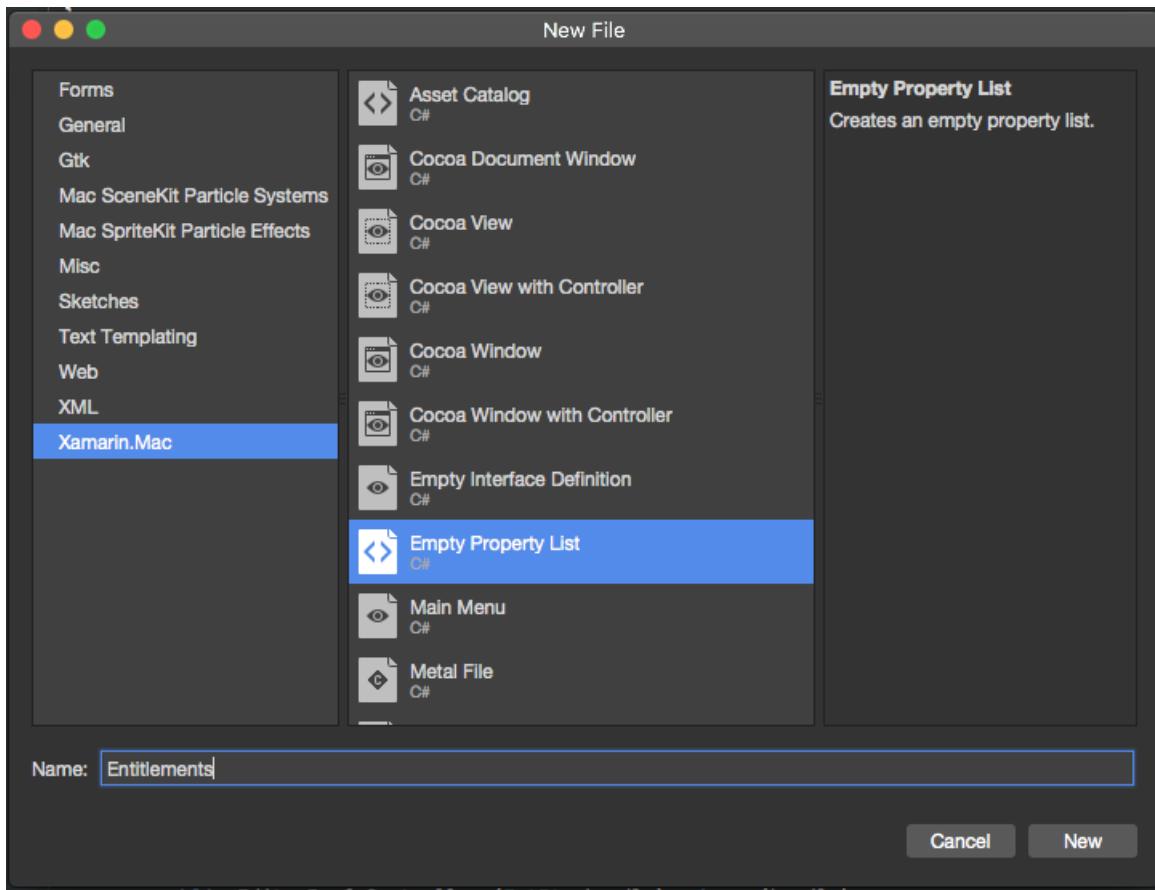
With the new Visual Studio for Mac installed, when you create a new Xamarin.Mac app project, an **Entitlements.plist** file will automatically be added to the project tree:



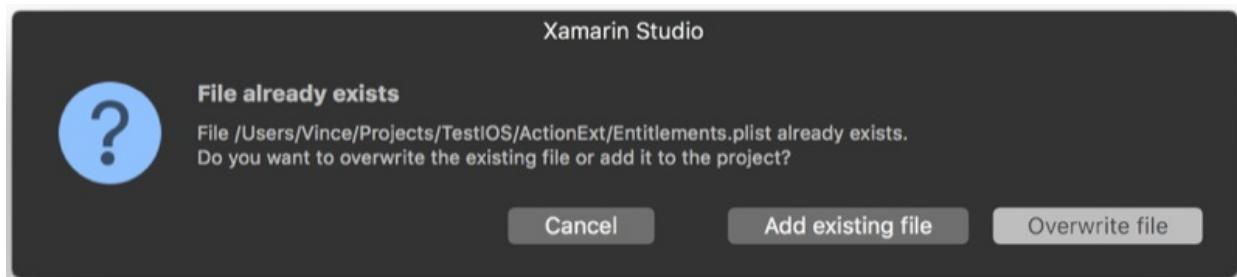
If you double-click the `Entitlements.plist` file, the Entitlements Editor will be displayed:



For existing Xamarin.Mac projects, you will need to manually create the `Entitlements.plist` file by right-clicking on the project in the **Solution Pad** and selecting **Add > New File...**. Next, select **Xamarin.Mac > Empty Property List**:



Enter `Entitlements` for the name and click the **New** button. If your project previously included an Entitlements file, you will be prompted to add it to the project instead of creating a new file:



Community support on the forums

The community of developers using Xamarin products is amazing and many visit our [Xamarin.Mac forums](#) to share experiences and their expertise. In addition, Xamarin engineers periodically visit the forum to help.

Filing a bug

Your feedback is important to us. If you find any problems with Xamarin.Mac:

- Search the [issue repository](#)
- Before switching to GitHub issues, Xamarin issues were tracked on [Bugzilla](#). Please search there for matching issues.
- If you cannot find a matching issue, please file a new issue in the [GitHub issue repository](#).

GitHub issues are all public. It's not possible to hide comments or attachments.

Please include as much of the following as possible:

- A simple example reproducing the issue. This is **invaluable** where possible.

- The full stack trace of the crash.
- The C# code surrounding the crash.

Xamarin.Mac error messages (mmp)

11/2/2020 • 13 minutes to read • [Edit Online](#)

MM0xxx: mmp error messages

E.g. parameters, environment, missing tools.

MM0000: Unexpected error - Please file a bug report at <https://github.com/xamarin/xamarin-macios/issues/new>

An unexpected error condition occurred. Please [file a bug report](#) with as much information as possible, including:

- Full build logs, with maximum verbosity (e.g. `-v -v -v -v` in the Additional mmp arguments);
- A minimal test case that reproduce the error; and
- All version informations

The easiest way to get exact version information is to use the **Xamarin Studio** menu, **About Xamarin Studio** item, **Show Details** button and copy/paste the version informations (you can use the **Copy Information** button).

MM0001: This version of Xamarin.Mac requires Mono {0} (the current Mono version is {1}). Please update the Mono.framework from <http://mono-project.com/Downloads>

MM0003: Application name '{0}.exe' conflicts with an SDK or product assembly (.dll) name.

MM0007: The root assembly '{0}' does not exist

MM0008: You should provide one root assembly only, found {0} assemblies: '{1}'

MM0009: Error while loading assemblies: *.

An error occurred while loading the assemblies from the root assembly references. More information may be provided in the build output.

MM0010: Could not parse the command line arguments: {0}

MM0016: The option '{0}' has been deprecated.

MM0017: You should provide a root assembly

MM0018: Unknown command line argument: '{0}'

MM0020: The valid options for '{0}' are '{1}'.

MM0023: Application name '{0}.exe' conflicts with another user assembly.

MM0026: Could not parse the command line argument '{0}': {1}

MM0043: The Boehm garbage collector is not supported. The SGen garbage collector has been selected instead.

MM0050: You cannot provide a root assembly if --no-root-assembly is passed.

MM0051: An output directory (--output) is required if --no-root-assembly is passed.

MM0053: Cannot disable new refcount with the Unified API.

MM0056: Cannot find Xcode in any of our default locations. Please install Xcode, or pass a custom path using --sdkroot=<path>

MM0059: Could not find the currently selected Xcode on the system: {0};

MM0060: Could not find the currently selected Xcode on the system. 'xcode-select --print-path' returned '{0}', but that directory

does not exist.

MM0068: Invalid value for target framework: {0}.

MM0071: Unknown platform: *. This usually indicates a bug in Xamarin.Mac; please file a bug report at <https://bugzilla.xamarin.com> with a test case.

This usually indicates a bug in Xamarin.Mac; please file a bug report at <https://bugzilla.xamarin.com> with a test case.

MM0073: Xamarin.Mac * does not support a deployment target of * (the minimum is *). Please select a newer deployment target in your project's Info.plist.

The minimum deployment target is the one specified in the error message; please select a newer deployment target in the project's Info.plist.

If updating the deployment target is not possible, then please use an older version of Xamarin.Mac.

MM0074: Xamarin.Mac * does not support a deployment target of * (the maximum is *). Please select an older deployment target in your project's Info.plist or upgrade to a newer version of Xamarin.Mac.

Xamarin.Mac does not support setting the minimum deployment target to a higher version than the version this particular version of Xamarin.Mac was built for.

Please select an older minimum deployment target in the project's Info.plist, or upgrade to a newer version of Xamarin.Mac.

MM0079: Internal Error - No executable was copied into the app bundle. Please contact 'support@xamarin.com'

MM0080: Disabling NewRefCount, --new-refcount:false, is deprecated.

MM0091: This version of Xamarin.Mac requires the * SDK (shipped with Xcode *). Either upgrade Xcode to get the required header files or use the dynamic registrar or set the managed linker behaviour to Link Platform or Link Framework SDKs Only (to try to avoid the new APIs).

Xamarin.Mac requires the header files, from the SDK version specified in the error message, to build your application with the static registrar. The recommended way to fix this error is to upgrade Xcode to get the required SDK, this will include all the required header files. If you have multiple versions of Xcode installed, or want to use an Xcode in a non-default location, make sure to set the correct Xcode location in your IDE's preferences.

One potential, alternative solution, is to enable the managed linker. This will remove unused API including, in most cases, the new API where the header files are missing (or incomplete). However this will not work if your project uses API that was introduced in a newer SDK than the one your Xcode provides.

A second potential, alternative solution, is use the dynamic registrar instead. This will impose a startup cost by dynamically registering types but remove the header file requirement.

A last-straw solution would be to use an older version of Xamarin.Mac, one that supports the SDK your project requires.

MM0097: machine.config file '{0}' can not be found.

MM0098: AOT compilation is only available on Unified

MM0099: Internal error {0}. Please file a bug report with a test case (<https://bugzilla.xamarin.com>).

MM0114: Hybrid AOT compilation requires all assemblies to be AOT compiled.

MM0129: Debugging symbol file for '*' does not match the assembly and is ignored.

The debugging symbols – either a .pdb (portable pdb only) or a .mdb file – for the specified assembly could not be loaded.

This generally means the assembly is newer or older than the symbols. Since they do not match they cannot be used and the symbols are ignored.

This warning won't affect the application being built, however you might not be able to debug it entirely (in particular the code from specified assembly). Also exceptions, stack traces and crash reports might be missing some information.

Please report this issue to the publisher of the assembly package (e.g. NuGet author) so this can be fixed in their future releases.

MM0130: No root assemblies found. You should provide at least one root assembly.

When running `--runregistrar`, at least one root assembly should be provided.

MM0131: Product assembly '{0}' not found in assembly list: '{1}'

When running `--runregistrar`, the assembly list should include the product assembly, Xamarin.Mac, XamMac.

MM0132: Unknown optimization: *. Valid values are: *

The specified optimization was not recognized.

The accepted format is `[+|-]optimization-name`, where `optimization-name` is one of the values listed in the error message.

See [Build optimizations](#) for a complete description of each optimization.

MM0133: Found more than 1 assembly matching '{0}' choosing first: '{1}'

MM0134: 32-bit applications should be migrated to 64-bit.

Apple has announced that it will not allow macOS App Store submissions of 32-bit apps (starting January 2018).

In addition 32-bit applications will not run on the version of macOS after High Sierra "without compromises".

For more details: <https://developer.apple.com/news/?id=06282017a>

Consider updating your application and any dependencies to 64-bit.

MM0135: Did not link system framework '{0}' (referenced by assembly '{1}') because it was introduced in {2} {3}, and we're using the {2} {4} SDK.

To build your application, Xamarin.Mac must link against system libraries, some of which depend upon the SDK version specified in the error message. Since you are using an older version of the SDK, invocations to those APIs may fail at runtime.

The recommended way to fix this error is to upgrade Xcode to get the needed SDK. If you have multiple versions of Xcode installed or want to use an Xcode in a non-default location, make sure to set the correct Xcode location in your IDE's preferences.

Alternatively, enable the managed [linker](#) to remove unused APIs, including (in most cases) the new ones which require the specified library. However, this will not work if your project requires APIs introduced in a newer SDK than the one your Xcode provides.

As a last-straw solution, use an older version of Xamarin.Mac that does not require these new SDKs to be present during the build process.

MM1xxx: file copy / symlinks (project related)

MM1034: Could not create symlink '{file}' -> '{target}': error {number}

MM14xx: Product assemblies

MM1401: The required '{0}' assembly is missing from the references

MM1402: The assembly '{0}' is not compatible with this tool

MM1403: {0} '{1}' could not be found. Target framework '{0}' is unusable to package the application.

MM1404: Target framework '{0}' is invalid.

MM1405: useFullXamMacFramework must always target framework .NET 4.5, not '{0}' which is invalid

MM1406: Target framework '{0}' is invalid when targeting Xamarin.Mac 4.5 .NET framework.

MM1407: Mismatch between Xamarin.Mac reference '{0}' and target framework selected '{1}'.

MM15xx: Assembly gathering (not requiring linker) errors

MM1501: Can not resolve reference: {0}

MachO.cs

MM1600: Not a Mach-O dynamic library (unknown header '0x{0}'): {1}.

MM1601: Not a static library (unknown header '{0}'): {1}.

MM1602: Not a Mach-O dynamic library (unknown header '0x{0}'): {1}.

MM1603: Unknown format for fat entry at position {0} in {1}.

MM1604: File of type {0} is not a MachO file ({1}).

MM2xxx: Linker

MM20xx: Linker (general) errors

MM2001: Could not link assemblies

MM2002: Can not resolve reference: {0}

MM2003: Option '{0}' will be ignored since linking is disabled

MM2004: Extra linker definitions file '{0}' could not be located.

MM2005: Definitions from '{0}' could not be parsed.

MM2006: Native library '{0}' was referenced but could not be found.

MM2007: Xamarin.Mac Unified API against a full .NET profile does not support linking. Pass the -nolink flag.

MM2009: Referenced by {0}.{1} ** This message is related to MM2006 **

MM2010: Unknown HttpResponseMessageHandler {0}. Valid values are HttpClientHandler (default), CFNetworkHandler or NSUrlSessionHandler

MM2011: Unknown TLSProvider '{0}'. Valid values are default or applets

MM2012: Only first {0} of {1} "Referenced by" warnings shown. ** This message related to 2009 **

MM2013: Failed to resolve the reference to "{0}", referenced in "{1}". The app will not include the referenced assembly, and may fail at runtime.

MM2014: Xamarin.Mac Extensions do not support linking. Request for linking will be ignored. ** This message is obsolete in XM 3.6+ **

MM2016: Invalid TlsProvider {0} option. The only valid value {1} will be used.

MM2017: Could not process XML description: {0}

MM202x: Binding Optimizer failed processing {0}.

MM2100: Xamarin.Mac Classic API does not support Platform Linking.

MM2103: Error processing assembly '*': *

An unexpected error occurred when processing an assembly.

The assembly causing the issue is named in the error message. In order to fix this issue the assembly will need to be provided in a [bug report](#) along with a complete build log with verbosity enabled (i.e. `-v -v -v -v` in the Additional mtouch arguments).

MM2104: Unable to link assembly '{0}' as it is mixed-mode.

Mixed-mode assemblies can not be processed by the linker.

See <https://docs.microsoft.com/cpp/dotnet/mixed-native-and-managed-assemblies> for more information on mixed-mode assemblies.

MM2106: Could not optimize the call to BlockLiteral.SetupBlock[Unsafe] in * at offset * because *.

The linker reports this warning when it can't optimize a call to `BlockLiteral.SetupBlock` or `Block.SetupBlockUnsafe`.

The message will point to the method that calls `BlockLiteral.SetupBlock[Unsafe]`, and it may also give clues as to why the call couldn't be optimized.

Please file an [issue](#) along with a complete build log so that we can investigate what went wrong and possibly enable more scenarios in the future.

MM2107: It's not safe to remove the dynamic registrar because {reasons}

The linker reports this warning when the developer requests removal of the dynamic registrar (by passing `--optimize:remove-dynamic-registrar` to mmp), but the linker determines that it's not safe to do so.

To remove the warning either remove the optimization argument to mmp, or pass `--nowarn:2107` to ignore it.

By default this option will be automatically enabled whenever it's possible and safe to do so.

MM2108: '{0}' was stripped of architectures except '{1}' to comply with App Store restrictions. This could break existing codesigning signatures. Consider stripping the library with lipo or disabling with `--optimize=-trim-architectures`;

The App Store now rejects applications which contain libraries and frameworks which contain 32-bit variants.

The library was stripped of unused architectures when copied into the final application bundle.

This is in general safe and will reduce application bundle size as an added benefit. However, any bundled framework that is code signed will have its signature invalidated (and resigned later if the application is signed).

Consider using `lipo` to remove the unnecessary architectures permanently from the source library. If the application is not being published to the App Store, this removal can be disabled by passing `--optimize=-trim-architectures` as Additional MMP Arguments.

MM2109: Xamarin.Mac Classic API does not support Platform Linking.

MM3xxx: AOT

MM30xx: AOT (general) errors

MM3001: Could not AOT the assembly '{0}'

MM3009: AOT of '{0}' was requested but was not found

MM3010: Exclusion of AOT of '{0}' was requested but was not found

MM4xxx: code generation

MM40xx: driver.m

MM4001: The main template could not be expanded to {0}.

MM41xx: registrar

MM4134: Your application is using the '{0}' framework, which isn't included in the MacOS SDK you're using to build your app (this framework was introduced in OSX {2}, while you're building with the MacOS {1} SDK.) This configuration is not supported with the static registrar (pass --registrar:dynamic as an additional mmp argument in your project's Mac Build option to select). Alternatively select a newer SDK in your app's Mac Build options.

MM4173: The registrar can't compute the block signature for the delegate of type {delegate-type} in the method {method} because *.

This is a warning indicating that the registrar couldn't inject the block signature of the specified method into the generated registrar code, because the registrar couldn't compute it.

This means that the block signature has to be computed at runtime, which is somewhat slower.

There are currently two possible reasons for this warning:

1. The type of the managed delegate is either a `System.Delegate` or `System.MulticastDelegate`. These types don't represent a specific signature, which means the registrar can't compute the corresponding native signature either. In this case the fix is to use a specific delegate type for the block (alternatively the warning can be ignored by adding `--nowarn:4173` as an additional mmp argument in the project's Mac Build options).
2. The registrar can't find the `Invoke` method of the delegate. This shouldn't happen, so please file an [issue](#) with a test project so that we can fix it.

MT4174: Unable to locate the block to delegate conversion method for the method {method}'s parameter #{parameter}.

This is a warning indicating that the static registrar couldn't find the method to create a delegate for an Objective-C block. An attempt will be made at runtime to find the method, but it will likely fail as well (with an MT8009 exception).

One possible reason for this warning is manually writing bindings for an API that uses blocks. It's recommended to use a binding project to bind Objective-C code – in particular when it involves blocks – since it's quite complicated to get it right when doing it manually.

If this is not the case, please file an [issue](#) with a test case.

MM5xxx: GCC and toolchain

MM51xx: compilation

MM5101: Missing '{0}' compiler. Please install Xcode 'Command-Line Tools' component.

MM5103: Failed to compile. Error code - {0}. Please file a bug report at <http://bugzilla.xamarin.com>

MM52xx: linking

MM5202: Mono.framework MDK is missing. Please install the MDK for your Mono.framework version from <http://mono-project.com/Downloads>

MM5203: Can't find libxammac.a, likely because of a corrupted Xamarin.Mac installation. Please reinstall Xamarin.Mac.

MM5204: Invalid architecture. x86_64 is only supported on non-Classic profiles.

MM5205: Invalid architecture '{0}'. Valid architectures are i386 and x86_64 (when --profile=mobile).

MM5218: Can't ignore the dynamic symbol {symbol} (--ignore-dynamic-symbol={symbol}) because it was not detected as a dynamic symbol.

See the [equivalent mtouch warning](#).

MM53xx: other tools

MM5301: pkg-config could not be found. Please install the Mono.framework from <http://mono-project.com/Downloads>

MM5305: Missing 'otool' tool. Please install Xcode 'Command-Line Tools' component

MM5306: Missing dependencies. Please install Xcode 'Command-Line Tools' component

MM5308: Xcode license agreement may not have been accepted. Please launch Xcode.

MM5309: Native linking failed with error code 1. Check build log for details.

MM5310: install_name_tool failed with an error code '{0}'. Check build log for details.

MM5311: lipo failed with an error code '{0}'. Check build log for details.

MM8xxx: runtime

MM800x: misc

MM8017: The Boehm garbage collector is not supported. Please use SGen instead.

MM8025: Failed to compute the token reference for the type '{type.AssemblyQualifiedName}' because {reasons}

This indicates a bug in Xamarin.Mac. Please file a bug at <https://bugzilla.xamarin.com>.

A potential workaround would be to disable the `register-protocols` optimization, by passing
`--optimize:-register-protocols` as an additional mmp argument in the project's Mac Build options.

MM8026: * is not supported when the dynamic registrar has been linked away.

This usually indicates a bug in Xamarin.Mac, because the dynamic registrar should not be linked away if it's needed. Please file a bug at <https://bugzilla.xamarin.com>.

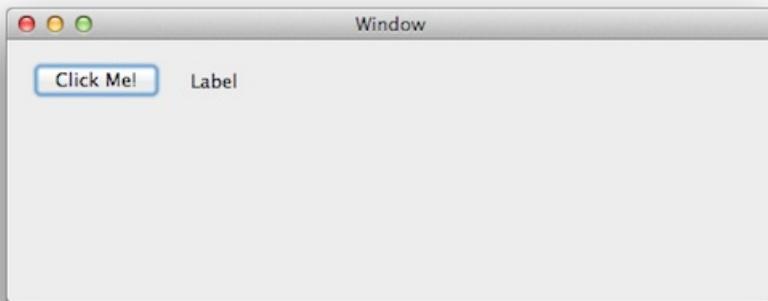
It's possible to force the linker to keep the dynamic registrar by adding `--optimize=-remove-dynamic-registrar` to the additional mmp arguments in the project's Mac Build options.

Xamarin.Mac samples

11/2/2020 • 2 minutes to read • [Edit Online](#)

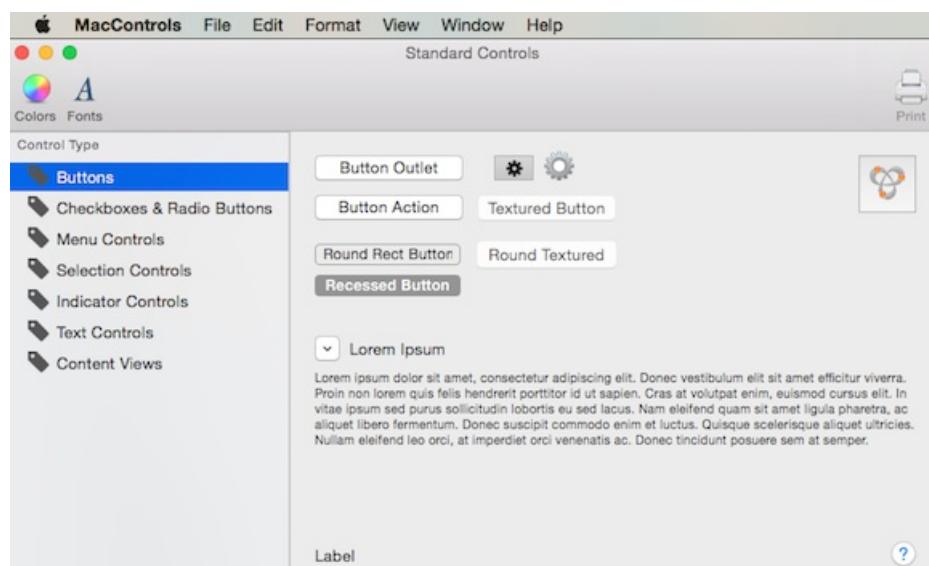
Xamarin.Mac sample apps and code demos to help you get started building mobile apps with C# and Xamarin.

[All Xamarin.Mac samples](#)



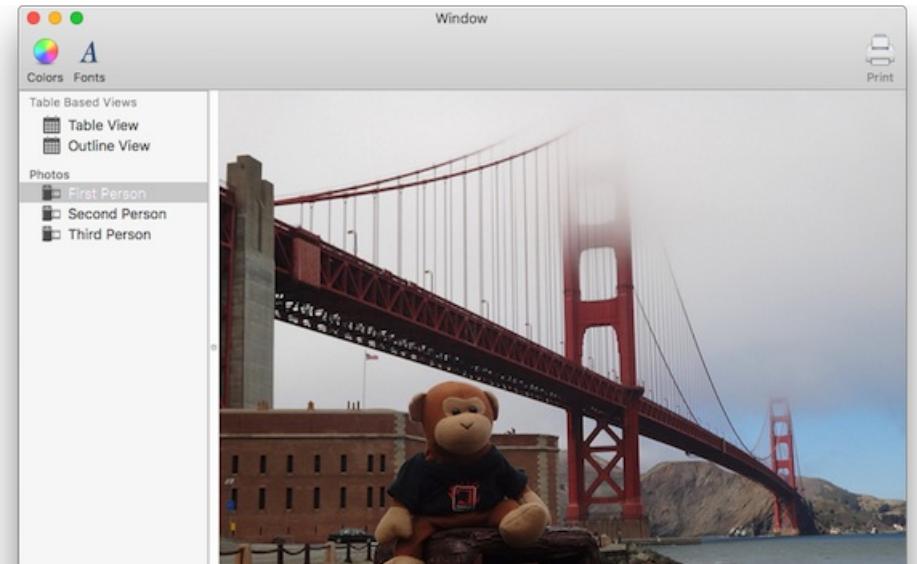
Hello, Mac

Hello World app to get started with Mac development.



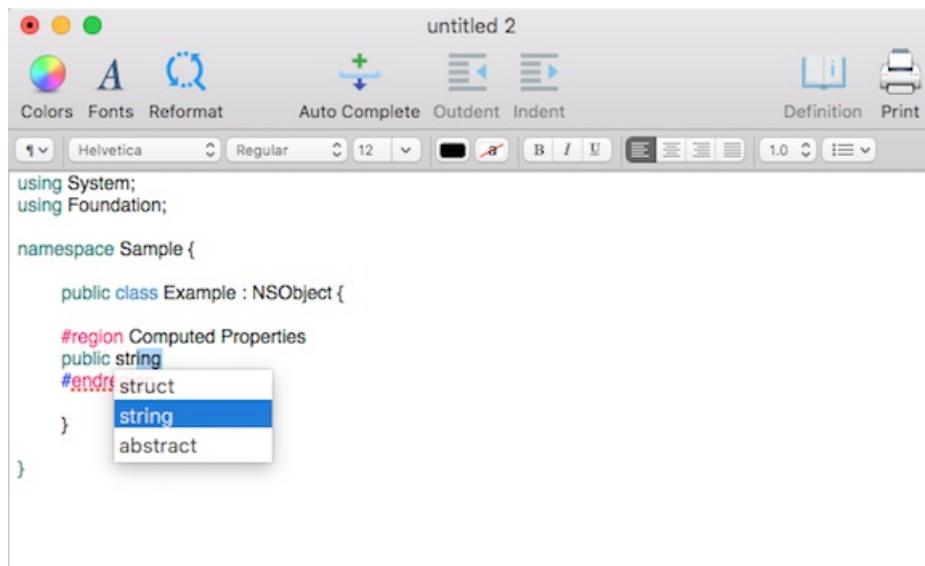
MacControls

UI control examples for Mac apps.



MacImages

UI control examples for Mac apps.



SourceWriter

Simple editor that provides support for code completion and syntax highlighting.

All samples

For the complete set of Xamarin.Mac sample apps and code demos, see [All Xamarin.Mac samples](#).